Profiling

This is preliminary documentation and subject to change

Last updated: 8 June 2000

Table of Contents

1 Profiling – Introduction

Profiling, in this document, means monitoring the performance and memory usage of a program, which is executing on the NGWS runtime. This document details the interfaces, provided by the runtime, to access such information. Typically, a very limited audience will use these APIs – developers of profiling tools.

Just to give the flavor, a typical use for profiling is to measure how much time (elapsed, or wall-clock, and/or CPU time) is spent within each routine, or within all code that is executed *from* a given *root* routine. To do this, a profiler asks the runtime to inform it whenever execution enters or leaves each routine; the profiler notes the wall-clock and CPU time for each such event, and accumulates the results at the end of the program.

Note: we use the term *routine* in this document to mean a section of code that has an entry point and an exit point. Different languages use different names for this same concept -- function, procedure, method, co-routine, subroutine, etc.

Profiling an NGWS program requires more support than profiling conventionally-compiled machine code. This is because NGWS routines are JIT-compiled (converting Intermediate Language into native machine code) at runtime – a profiler cannot discover in advance what the generated code will be, nor where within the address space of the process it will be loaded. The profiling APIs safely provide this missing information

In addition, the runtime may choose to discard the machine code it has JIT-compiled for a class, in order to free up memory for more urgent uses – we refer to this process as *code pitching*. The memory released may be used to hold the results of JIT-compiling a totally different routine. Clearly, a profiling tool needs to be informed this has happened, as it could confuse the profiling statistics of the old routine with that of the new.

Note that JIT-compiling routines at runtime provides good opportunities, as the APIs allow a profiler to change the in-memory IL code stream for a routine, and then request that it be JIT-compiled anew. In this way, the profiler can dynamically add instrumentation code to particular routines that need deeper investigation. Although this approach is possible in conventional scenarios, it's much easier to do this for the NGWS runtime

2 Goals for the Profiling APIs

- Expose information that existing profilers will require for a user to determine and analyze performance of a program run on the NGWS runtime. Specifically:
  - Execution engine startup and shutdown events
  - Application domain creation and shutdown events
  - Assembly loading and unloading events
  - Module load/unload events
  - Classic Com callable wrappers creation and destruction events
  - JIT-compiles, and code pitching events
  - Class load/unload events
  - Thread birth/death/synchronization
  - Routine entry/exit events

  - Exceptions
  - Transitions between managed and unmanaged execution
  - Transitions between different runtime *contexts*
  - Security checks
  - Information about runtime suspensions
  - Information about the runtime memory heap and garbage collection activity
- Callable from any COM-compatible language
- Efficient, in terms of CPU and memory consumption – the act of profiling should not cause such a big change upon the program being profiled that the results are misleading
- Useful to both *sampling* and *non-sampling* profilers. [A *sampling* profiler inspects the profilee at regular clock ticks – maybe 5 milliseconds apart, say. A *non-sampling* profiler is informed of events, synchronously with the thread that causes them]

---

3 Non-goals for the Profiling APIs

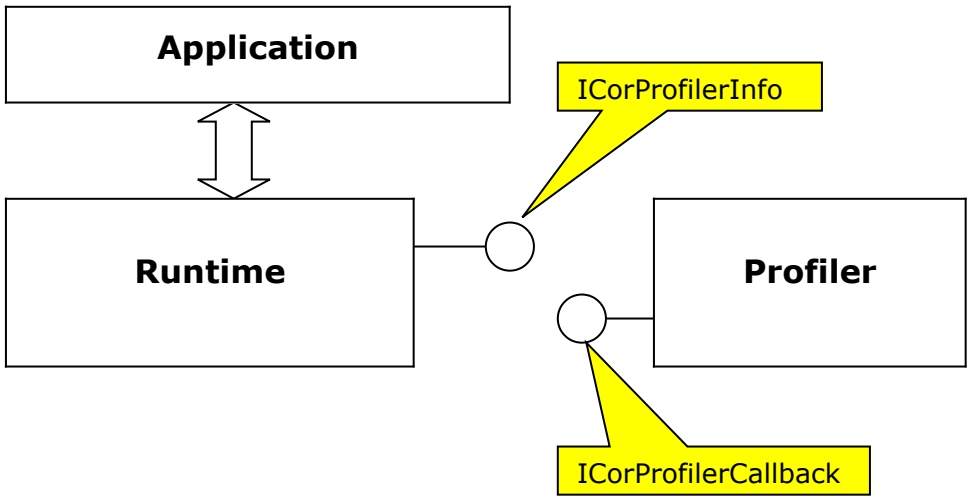
- Support for profiling unmanaged code. Existing mechanisms must instead be used to profile unmanaged code. The NGWS profiling APIs work only for managed code. However, we provide the profiler with managed/unmanaged transition events to determine the boundaries between managed and unmanaged code.
- Information needed to check bounds. The runtime provides intrinsic support for bounds checking of all managed code.

---

4 Profiling APIs – Overview

The profiling APIs within NGWS allow you to monitor the execution and memory usage of a running application. Typically, these APIs will be used to write a code profiler package. In the sections that follow, we will talk about a profiler as a package built to monitor execution of *any* managed application.

The profiling APIs are implemented as two COM interfaces, shown in the diagram below. One is implemented by the runtime (*ICorProfilerInfo*), the other is implemented by the profiler (*ICorProfilerCallback*).



The *ICorProfilerCallback* interface consists of methods with names like *ClassLoadStarted*, *ClassLoadFinished*, *FunctionEnter*, *FunctionLeave*. So, each time the runtime loads/unloads a class, or enters/leaves a function, it calls the corresponding method in the profiler's *ICorProfilerCallback* interface. (And similarly for all of the other notifications; see later for details)

So, for example, a simple profiler could measure code performance via the two notifications *FunctionEnter* and *FunctionLeave*. It simply timestamps each notification, accumulates results, then outputs a list indicating which functions consumed most cpu time, or most wall-clock time, during execution of the application.

So, if it helps, you can think of the *ICorProfilerCallback* interface as the "notifications API".

The other interface involved for profiling is *ICorProfilerInfo*. The profiler calls this, as required, to obtain more information to help its analysis. For example, whenever the runtime calls *FunctionEnter* it supplies a value for the *FunctionId*. The profiler can discover more information about that *FunctionId* by calling the *ICorProfilerInfo::GetFunctionInfo* to discover the function's parent class, its name, etc, etc.

The picture so far describes what happens once the application and profiler are running. But how are the two connected together when an application is started? Well, the runtime makes the connection during its initialization in each process. It decides whether to connect to a profiler, and which profiler that should be, depending upon the value for two environment variables, checked one after the other:

- Cor_Enable_Profiling – only connect with a profiler if this environment variable exists and is set to a non-zero value.
- Cor_Profiler – connect with the profiler with this CLSID or ProgID (which must have been stored previously in the Registry). The Cor_Profiler environment variable is defined as a string: eg
  set Cor_Profiler={32E2F4DA-1BEA-47ea-88F9-C5DAF691C94A}
  or
  set Cor_Proflier="MyProfiler"
  The profiler class is the one that implements *ICorProfilerCallback*

When both checks above pass, the runtime creates an instance of the profiler in a similar fashion to *CoCreateInstance*. The profiler is not loaded through a direct call to *CoCreateInstance* so that a call to *CoInitialiaze* may be avoided, which requires setting the threading model. It then calls the *ICorProfilerCallback::Initialize* method in the profiler. The signature of this method is:

```
HRESULT Initialize(IUnknown *pICorProfilerInfoUnk, DWORD
*pdwRequestedEvents)
```

The profiler must QueryInterface pICorProfilerInfoUnk for an ICorProfilerInfo interface pointer and save it so that it can call for more info during later profiling. It then sets the *pdwRequestedEvents* bitmask to say which categories of notifications it is interested in. For example:

```
*pdwRequestedEvents = COR_PRF_MONITOR_CALLS |
COR_PRF_MONITOR_GC
```

if interested only in function enter/leave notifications and Garbage Collection notifications. The profiler then simply returns, and we're off and running!

By setting the notifications mask in this way, the profiler can limit which notifications it receives. This obviously helps you to build a simpler, or special-purpose profiler; it also reduces wasted cpu time in sending notifications that the profiler would simply 'drop on the floor' (see later for details)

You can see from the above explanation that only one profiler can be profiling any process at one time.

Note: we provide a simple GUI tool, called "Profiler Picker" which helps set and inspect the profiler Registry settings (see later)

The profiler must be implemented as an inproc32 COM server – a DLL which is mapped into the same address space as the process being profiled. We do not support any other type of COM server; if a profiler, for example, wants to monitor applications from a remote computer, it must implement 'collector agents' on each machine, which batch results and communicate them to the central data collection machine.

---

5 Profiling APIs – Recurring Concepts

This brief section explains a few concepts that apply throughout the profiling APIs, rather than repeat them with the description of each method.

5.1 IDs

Runtime notifications supply an ID for reported classes, threads, AppDomains, etc. These IDs can be used to query the runtime for more info. These IDs are simply the address of a block in memory that describes the item; however, they should be treated as opaque handles by any profiler.

Because IDs are simply memory addresses, ObjectIDs point into the garbage-collected heap and may change their value with each garbage collection. Thus, an ObjectID value is only valid between the time it is received and when the next garbage collection begins. The runtime also supplies notifications that allow a profiler to update its internal maps that track objects, so that a profiler may maintain a valid ObjectID across garbage collections.

5.2 Return Values

As a profiler, you can return a status, as an HRESULT, for each notification the runtime gives you. That status may have the value S_OK or E_FAIL. However, in first release, the runtime ignores this status value in every callback except ObjectReference (see method description below).

5.3 Notification Thread

In most cases, the notifications are executed by the same thread as generated the event. Such notifications (for example, *FunctionEnter* and *FunctionLeave)* don't need to supply the explicit ThreadID. Also, the profiler might choose to use thread-local storage to store and update its analysis blocks, as compared with indexing into global storage, based off the ThreadID of the affected thread.

Each notification in the lists below document which thread does the call – either the thread which generated the event, or some utility thread (eg garbage collector) within the runtime. For any callback that might be invoked by a different thread, you can call the ICoreProfilerInfo::GetCurrentThreadID to discover the thread that generated the event.

5.4 Nesting of Notifications

Notifications to a profilers follow the obvious nesting sequence. For example, after an *AssemblyUnloadStarted*, the profiler should expect to see a flurry of *ModuleUnloadStarted* notifications; then a flurry of *ClassUnloadStarted notification*s; and so on. The nesting looks like this:

```
AssemblyUnloadStarted (AssemA)
        ModuleUnloadStarted (ModuleA)
                ClassUnloadStarted (ClassA)
                        FunctionUnloadStarted (FuncA)
                        FunctionUnloadFinished (FuncA)
                        . . .
                ClassUnloadFinished (ClassA)
                . . .
        ModuleUnloadFinished (ModuleA)
        . . .
AssemblyUnloadFinished (AssemA)
```

---

6 ICorProfilerCallback – Details

As explained earlier, the ICorProfilerCallback interface is the "notifications API" that a profiler implements. Though the interface contains many methods, understanding them is easier once you realize they fall into about 12 categories, and that often within a category, they come "four at a time". The categories are:

> Assemblies, AppDomains, Modules, Classes, Objects, Functions, Threads, Interop, Exceptions, Garbage Collections, Remoting, and runtime Suspensions

If we take Modules as an example, they have four notifications, recording the birth (start and finish) and death (start and finish) of a given Module. Their names are:

- ModuleLoadStarted, ModuleLoadFinished
- ModuleUnloadStarted, ModuleUnloadFinished

And we follow a similar naming scheme throughout the API.

Almost all of the notifications provide an ID to the item being of interest – for example, ModuleID, ClassID, FunctionID. These are opaque 32-bit handles. A profiler uses them to keep track of notifications (for example, the number of times each function in an application is called). The profiler can also use that ID to ask for more information about the item, via the ICorProfilerInfo methods provided by the runtime. The IDs are valid to use until you receive a callback indicating the specific ID has been unloaded, deleted or otherwise invalidated.

The next sections list all the methods on ICorProfilerCallback, gathered together into categories

6.1 Runtime

6.1.1 Initialize

The NGWS runtime calls *Initialize* to setup the code profiler whenever a new NGWS application is started. The call provides an *IUnknown* interface pointer that should be QI'd for an *ICorProfilerInfo* interface pointer, and a pointer to a DWORD that should be filled out with all of the values from the COR_PRF_MONITOR enum that the profiler wishes to receive events for.

```
HRESULT Initialize(IUnknown *pICorProfilerInfoUnk, DWORD *pdwRequestedEvents);
```

Parameter	Description
[in] pICorProfilerInfoUnk	A pointer to an IUnknown object within the runtime which can be QueryInterface'd for an IcorProfilerInfo interface pointer. The profiler can call methods in this object to obtain more info about notifications
[out] pdwRequestedEvents	The profiler sets this bitmask to tell runtime which notifications it wants to receive. The bit values are defined in the COR_PRF_MONITOR enum in CorProf.h. This is the only opportunity to enable callbacks that are a part of COR_PRF_MONITOR_IMMUTABLE, since they can no longer be changed after returning from this function.

6.1.2 Shutdown

The NGWS runtime calls *Shutdown* to notify the code profiler that the applicationlication is exiting. This is the profiler's last opportunity to safely call functions on the *ICorProfilerInfo* interface. After returning from this function the runtime will proceed to unravel its internal data structures and any calls to *ICorProfilerInfo* are undefined in their behaviour.

```
HRESULT Shutdown()
```

6.2 AppDomain

6.2.1 AppDomainCreationStarted

Called when an AppDomain creation has begun. The id is not valid for any information request until after the AppDomain has been fully created. One may only cache the id provided in AppDomainCreationStarted for later use.

```
HRESULT AppDomainCreationStarted(AppDomainID appDomainId)
```

Parameter	Description
[in] appDomainId	ID for the AppDomain being created

6.2.2 AppDomainCreationFinished

Called when an AppDomain creation has finished. The hrStatus provides the success or failure of the operation.

```
HRESULT AppDomainCreationFinished(AppDomainID appDomainId, HRESULT hrStatus)
```

Parameter	Description
[in] appDomainId	ID for the AppDomain just created
[in] hrStatus	Status for whether the AppDomain creation succeeded

6.2.3 AppDomainShutdownStarted

Notify that runtime is starting to shut down an AppDomainAppDomainShutdownStarted is the last point at which the AppDomainID is valid for calls to the *ICorProfilerInfo* interface

Syntax

```
HRESULT AppDomainShutdownStarted(AppDomainID appDomainId)
```

Parameter	Description
[in] appDomainId	ID for the AppDomain being shut down

6.2.4 AppDomainShutdownFinished

Notify that runtime has finished shutting down an AppDomain. You cannot use app*DomainId* to query the runtime for info during or after this notification – it is supplied only so the profiler knows which AppDomain has just been shut down. The hrStatus provides the success or failure of the operation.

```
HRESULT AppDomainShutdownFinished(AppDomainID appDomainId, HRESULT hrStatus)
```

Parameter	Description
[in] appDomainId	ID for the AppDomain just shut down
[in] hrStatus	Status for whether the AppDomain shutdown succeeded

6.3 Assembly

You might expect that runtime would notify an assembly load, followed by one or more module loads for that assembly. However, what actually happens is that

runtime notifies you of a module load, then the load of its containing assembly; after that you may obtain zero or more notifications of module loads for that assembly. Thus, the *"first child begets the parent".*

There is another, unusual path through module loading to be aware of. That is when a module is loaded via a legacy mechanism, such as a call to the Win32 *LoadLibrary* routine, or implicitly due to entries in the Import Address Table of the current image. In such cases, you will see a module load notification. Some time later (when the runtime actually needs to execute code from that 'legacy' module) it will discover which assembly it is a part of. At that point, runtime will notify you by calling your *ModuleAttachedToAssembly* method.

6.3.1 AssemblyLoadStarted

Called when an Assembly load has begun. The id is not valid for any information request until after the assembly has been fully loaded. One may only cache the id provided in AssemblyLoadStarted for later use.

```
HRESULT AssemblyLoadStarted(AssemblyID assemblyId)
```

Parameter	Description
[in] assemblyID	ID for the Assembly being loaded

6.3.2 AssemblyLoadFinished

Called when an Assembly load has begun. The id is now valid for any information request through the ICorProfilerInfo interface. The hrStatus provides the success or failure of the operation.

```
HRESULT AssemblyLoadFinished(AssemblyID assemblyId, HRESULT hrStatus)
```

Parameter	Description
[in] assemblyId	ID for the Assembly just loaded
[in] hrStatus	Status for whether the Assembly load succeeded

6.3.3 AssemblyUnloadStarted

Called before and after an assembly is unloaded. AssemblyUnloadStarted is the last point at which the AssemblyID is valid for calls to the ICorProfilerInfo interface.

```
HRESULT AssemblyUnloadStarted(AssemblyID assemblyId)
```

Parameter	Description
[in] assemblyId	ID for the Assembly being unloaded

6.3.4 AssemblyUnloadFinished

Notify that runtime has finished unloading an Assembly. You cannot use *assemblyId* to query the runtime for info after this notification – it is supplied only so the profiler knows which Assembly has just been unloaded. The hrStatus provides the success or failure of the operation.

```
HRESULT AssemblyUnloadFinished(AssemblyID assemblyId, HRESULT hrStatus)
```

Parameter	Description
[in] appDomainId	ID for the Assembly just unloaded
[in] hrStatus	Status for whether the Assembly unload succeeded

6.4 Module

6.4.1 ModuleLoadStarted

The runtime calls ModuleLoadStarted to notify the code profiler that a module is about to be loaded. The ModuleID is not valid in calls to ICorProfilerInfo until the profiler receives a ModuleLoadFinished callback for the same ModuleID

```
HRESULT ModuleLoadStarted(ModuleID moduleId)
```

Parameter	Description
[in] moduleId	ID for the Module being loaded

6.4.2 ModuleLoadFinished

The runtime calls ModuleLoadFinished to notify the code profiler that a module has been loaded. The hrStatus provides the success or failure of the operation.

```
HRESULT ModuleLoadFinished(ModuleID moduleId, HRESULT hrStatus)
```

Parameter	Description
[in] moduleId	ID for the Module just loaded
[in] hrStatus	Status for whether the Module load succeeded

6.4.3 ModuleUnloadStarted

Called before a module is being unloaded. Use this event to collect final statics that require the ModuleID to be valid. After returning from ModuleUnloadStarted, the ModuleID is no longer valid.

```
HRESULT ModuleUnloadStarted(ModuleID moduleId)
```

Parameter	Description
[in] moduleId	ID for the Module being unloaded

6.4.4 ModuleUnloadFinished

Notify that runtime has finished unloading a Module. You cannot use *moduleId* to query the runtime for info after this notification – it is supplied only so the profiler knows which Module just been unloaded. The hrStatus provides the success or failure of the operation.

```
HRESULT ModuleUnloadFinished(ModuleID moduleId, HRESULT hrStatus)
```

Parameter	Description
[in] moduleId	ID for the Module just shut unloaded
[in] hrStatus	Status for whether the Module unload succeeded

6.4.5 NotifyModuleAttachedToAssembly

The runtime calls *NotifyModuleAttachedToAssembly* to notify the code profiler that a module has been attached to an assembly. A module can get loaded through legacy means, (i.e., Import Address Table or LoadLibrary) or through a metadata reference. The runtime loader therefore has many code paths for determining what assembly a module lives in. It is therefore possible that after a *NotifyModuleLoadFinished* event, the module does not know what assembly it is in and getting the parent AssemblyID is not possible. This event is fired when the module is officially attached to its parent assembly. Calling GetModuleInfo after this point will return the proper parent assembly.

Syntax

```
HRESULT NotifyModuleAttachedToAssembly(ModuleID moduleId, AssemblyID
assemblyId);
```

Parameter	Description
[in] moduleId	The ModuleID of the module loaded.
[in] assemblyId	The AssemblyID of the parent assembly.

6.5 Class

6.5.1 ClassLoadStarted

Notify that runtime is starting to load a class. You cannot use *classId* to query the runtime for info until after the class load is finished. The ClassID is not valid for calls to the ICorProfilerInfo interface until the profiler receives a ClassLoadFinished event for the same ClassID.

```
HRESULT ClassLoadStarted(ClassID classId)
```

Parameter	Description
[in] classId	ID for the class being loaded

6.5.2 ClassLoadFinished

The runtime calls ClassLoadFinished to notify the code profiler that a class has been loaded. The ClassID is now valid for calls to the ICorProfilerInfo interface. The hrStatus provides the success or failure of the operation.

```
HRESULT ClassLoadFinished(ClassID classId, HRESULT hrStatus)
```

Parameter	Description
[in] classId	ID for the Class just created
[in] hrStatus	Status for whether the class load succeeded

6.5.3 ClassUnloadStarted

The given class is about to be unloaded. Use this event to gather final status and clean up anything that requires the ClassID to be valid. After returning from this callback the ClassID is no longer valid in calls to the ICorProfilerInfo interface.

```
HRESULT ClassUnloadStarted(ClassID classId)
```

Parameter	Description
[in] classId	ID for the class being unloaded

6.5.4 ClassUnloadFinished

Notify that runtime has finished unloading a class. You cannot use class*Id* to query the runtime for info after this notification – it is supplied only so the profiler knows which class has just been unloaded. The hrStatus provides the success or failure of the operation.

```
HRESULT ClassUnloadFinished(ClassID classId, HRESULT hrStatus)
```

Parameter	Description
[in] classId	ID for the class just unloaded
[in] hrStatus	Status for whether the class unload succeeded

6.6 Function

6.6.1 JITCompilationStarted

The runtime calls JITCompilationStarted to notify the code profiler that the JIT compiler is starting to compile a function.

The fIsSafeToBlock argument tells the profiler whether or not blocking will affect the operation of the runtime. If true, blocking may cause the runtime to wait for the calling thread to return from this callback, especially if the runtime is attempting a suspension. Although this will not harm the runtime, it will skew the profiling results.

```
HRESULT JITCompilationStarted(FunctionID functionId, BOOL
fIsSafeToBlock)
```

Parameter	Description
[in] functionId	ID for the function being JIT-compiled
[in] fIsSafeToBlock	whether it's safe to perform a time consuming operation while profiling

6.6.2 JITCompilationFinished

The runtime calls JITCompilationFinished to notify the code profiler that the JIT compiler has finished compiling a function. The FunctionID is now valid in ICorProfilerInfo APIs. The hrStatus provides the success or failure of the operation

The fIsSafeToBlock argument tells the profiler whether or not blocking will affect the operation of the runtime. If true, blocking may cause the runtime to wait for the calling thread to return from this callback. Although this will not harm the runtime, it will skew the profiling results.

```
HRESULT JITCompilationFinished(FunctionID functionId, HRESULT hrStatus, BOOL
fIsSafeToBlock)
```

Parameter	Description
[in] functionId	ID for the function just created
[in] hrStatus	Status for whether the JIT-compile succeeded
[in] fIsSafeToBlock	whether it's safe to perform a time consuming operation while profiling

6.6.3 FunctionUnloadStarted

The runtime calls FunctionUnloadStarted to notify the code profiler that a function is being unloaded. After returning from this call, the FunctionID is no longer valid.

NOTE: currently not implemented.

```
HRESULT FunctionUnloadStarted(FunctionID functionId)
```

Parameter	Description
[in] functionId	ID for the function being unloaded

6.6.4 JITCachedFunctionSearchStarted

This notifies the profiler when a search for a prejitted function is starting. You return *pbUseCachedFunction* to tell runtime whether it should use the function found or not. In the latter case, runtime will JIT-compile the function (resulting in a matched pair of JITCompilationStarted and JITCompilationFinished notification) instead of using the cached version. NOTE: the FunctionID is not valid for calls to any ICorProfilerInfo APIs until the profiler has received the corresponding JITCompilationFinished.

`HRESULT` **`JITCachedFunctionSearchStarted`**(FunctionID functionId, BOOL *pbUseCachedFunction)

Parameter	Description
[in] `functionId`	ID for the function being unloaded
`[out] pbUseCachedFunction`	• if true, the EE uses the cached function (if applicable) • if false, the EE jits the function instead of using a pre-jitted version.

6.6.5 JITCachedFunctionSearchFinished

Notify that runtime has finished searching for a previously-JIT-compiled function. This notification occurs only when a module is found to contain pre-JIT-compiled code. The *result* tells you whether it found the function or not.

`HRESULT` **`JITCachedFunctionSearchFinished`**`(FunctionID functionId, COR_PRF_JIT_CACHE result)`

Parameter	Description
[in] functionId	ID for the function being searched for
[in] result	Whether function was found in JIT cache There are two possible results: • COR_PRF_CACHED_FUNCTION_FOUND • COR_PRF_CACHED_FUNCTION_NOT_FOUND

Note that the COR_PRF_JIT_CACHE enum at the moment has only two values – in effect, found or not found. We keep it as an enum (rather than use a BOOL) as a

placeholder for future extensions – for example, to report the version of the JIT-compiled function that was found as current or old.

6.6.6 JITFunctionPitched

The runtime calls JITFunctionPitched to notify the profiler that a jitted function was removed from memory. If the pitched function is called in the future, the profiler will receive new JIT compilation events as it is re-jitted. NOTE: the FunctionID is not valid until it is re-jitted. When it is re-jitted, it will use the same FunctionID value.

`HRESULT` **`JITFunctionPitched`**(FunctionID functionId)

Parameter	Description
[in] functionId	ID for the function that is being pitched.

6.6.7 JITInlining

The runtime calls JITInlining to notify the profiler that the jitter is about to inline calleeId into callerId. Set pfShouldInline to FALSE to prevent the callee from being inlined into the caller, and set to TRUE to allow the inline to occur.

NOTE: Inlined functions do not provide Enter/Leave events, so if you desire an accurate callgraph, you should set FALSE. Be aware that setting FALSE will affect performance, since inlining typically increases speed and reduces separate jitting events for the inlined method.

`HRESULT` **`JITInlining`**(FunctionID callerId, FunctionID calleeId, BOOL *pfShouldInline)

Parameter	Description
[in] `callerId`	ID for the function that will have the callee inlined into it
[in] `calleeId`	ID for the function to be inlined
[out] `pfShouldInline`	• Set to TRUE to allow the inline to occur • Set to FALSE to prevent the inline from occurring

Note that you can disable all JIT-lining in the *Initialize* callback by setting the bit COR_PRF_DISABLE_INLINING

6.7 Thread

Note that unlike other categories, we do **not** provide separate *Started* and *Finished* notifications on thread create and destroy. This simplification was chosen simply because the number of instructions executed for these operations by the runtime is quite small; also, it seems reasonable that profilers should attribute the cycles consumed to that thread, rather than gathered as "runtime overhead"

6.7.1 ThreadCreated

The runtime calls ThreadCreated to notify the code profiler that a thread has been created. The ThreadID is valid immediately.

```
HRESULT ThreadCreated(ThreadID threadId)
```

Parameter	Description
[in] threadId	ID for the thread just created

6.7.2 ThreadDestroyed

The runtime calls ThreadDestroyed to notify the code profiler that a thread has been destroyed. The ThreadID is no longer valid.

```
HRESULT ThreadDestroyed(ThreadID ThreadId)
```

Parameter	Description
[in] threadId	ID for the thread just destroyed

6.7.3 ThreadAcquiringMonitor

The runtime calls ThreadAcquiringMonitor to notify the code profiler that a thread is attempting to acquire a monitor on an object.

NOTE: currently not implemented.

```
HRESULT ThreadAcquiringMonitor(ThreadID threadId, MonitorID monitorId,
        ObjectID objectId, ClassID classId)
```

Parameter	Description
[in] threadId	ID for the thread acquiring the monitor
[in] monitorId	ID for the monitor being acquired
[in] objectId	ID for the object whose monitor is being acquired
[in] classId	ID of the class of the object identified by *objectId*

6.7.4 ThreadBlockedMonitor

Notify that a thread's execution has blocked, waiting to acquire a monitor. This notification will occur between a *ThreadAcquiringMonitor* and a subsequent *ThreadAcquiredMonitor* notification – but only if the thread could not acquire the monitor because it was held by another thread. [It might be thought this notification is not required – after all, why not simply count the time between *Acquiring* and *Acquired*? – because that time difference could be due to a thread scheduling switch

rather than a genuine stall. The *ThreadBlockMonitor* notification allows a profiler to measure genuine contention on the monitor].

NOTE: currently not implemented.

```
HRESULT ThreadBlockedMonitor(ThreadID threadId, MonitorID monitorId,
        ObjectID objectId, ClassID classId)
```

Parameter	Description
[in] threadId	ID for the thread that just blocked
[in] monitorId	ID for the monitor that the thread is blocked on
[in] objectId	ID for the object whose monitor is being acquired
[in] classId	ID of the class of the object identified by *objectId*

6.7.5 ThreadAcquiredMonitor

Notify that a thread has acquired a monitor

NOTE: currently not implemented.

```
HRESULT ThreadAcquiredMonitor(ThreadID threadId, MonitorID monitorId,
        ObjectID objectId, ClassID classId)
```

Parameter	Description
[in] threadId	ID for the thread that just acquired the monitor
[in] monitorId	ID for the monitor being acquired
[in] objectId	ID for the object whose monitor is being acquired
[in] classId	ID of the class of the object identified by *objectId*

6.7.6 ThreadReleasedMonitor

Notify that a thread has just released a monitor

NOTE: currently not implemented.

```
HRESULT ThreadReleasedMonitor(ThreadID threadId, MonitorID monitorId,
        ObjectID objectId, ClassID classId)
```

Parameter	Description
[in] threadId	ID for the thread that just released the monitor
[in] monitorId	ID for the monitor that the thread just released
[in] objectId	ID for the object whose monitor was just released
[in] classId	ID of the class of the object identified by *objectId*

6.7.7 ThreadAssignedToOSThread

Notify that a runtime thread has just been assigned to execute by the assigned OS thread. During its execution lifetime, a given runtime thread may be switched between different threads, or not – at the whim of both the runtime and external components running within the process. This notification is called immediately after a ThreadCreated event to indicate what OS thread the newly-created runtime thread will execute on.

```
HRESULT ThreadAssignedToOSThread(ThreadID managedThreadId, DWORD
osThreadId)
```

Parameter	Description
[in] managedThreadId	ID for the managed thread
[in] osThreadId	ID for the OS thread mated with the managed thread

6.8 Remoting

NOTE: each of the following pairs of callbacks will occur on the same thread

RemotingClientInvocationStarted and RemotingClientSendingMessage

RemotingClientReceivingReply and RemotingClientInvocationFinished

RemotingServerInvocationReturned and RemotingServerSendingReply

RemotingServerInvocationStarted and RemotingServerReceivingMessage

6.8.1 RemotingClientInvocationStarted

The runtime calls RemotingClientInvocationStarted to notify the profiler that a remoting call has begun. This event is the same for synchronous and asynchronous calls.

```
HRESULT RemotingClientInvocationStarted()
```

6.8.2 RemotingClientSendingMessage

The runtime calls RemotingClientSendingMessage to notify the profiler that a remoting call is requiring the the caller to send an invocation request through a remoting channel.

```
HRESULT RemotingClientSendingMessage(GUID *pCookie, BOOL fIsAsync)
```

Parameter	Description
[in] pCookie	if remoting GUID cookies are active, this value will correspond with the the value provided in RemotingServerReceivingMessage, if the channel succeeds in transmitting the message, and if GUID cookies are active on the server-side process. This allows easy pairing of remoting calls, and the creation of a logical call stack.
[in] fIsAsync	is true if the call is asynchronous.

6.8.3 RemotingClientReceivingReply

The runtime calls RemotingClientReceivingReply to notify the profiler that the server-side portion of a remoting call has completed and that the client is now receiving and about to process the reply.

```
HRESULT RemotingClientReceivingReply(GUID *pCookie, BOOL fIsAsync)
```

Parameter	Description
[in] pCookie	if remoting GUID cookies are active, this value will correspond with the the value provided in RemotingServerSendingReply, if the channel succeeds in transmitting the message, and if GUID cookies are active on the server-side process. This allows easy pairing of remoting calls.
[in] fIsAsync	is true if the call is asynchronous

6.8.4 RemotingClientInvocationFinished

The runtime calls RemotingClientInvocationFinished to notify the profiler that a remoting invocation has run to completion on the client side. If the call was synchronous, this means that it has also run to completion on the server side. If the call was asynchronous, a reply may still be expected when the call is handled. If the call is asynchronous, and a reply is expected, then the reply will occur in the form of a call to RemotingClientReceivingReply and an additional call to RemotingClientInvocationFinished to indicate the required secondary processing of an asynchronous call.

```
HRESULT RemotingClientInvocationFinished();
```

6.8.5 RemotingServerReceivingMessage

The runtime calls RemotingServerReceivingMessage to notify the profiler that the process has received a remote method invocation (or activation) request. If the message request is asynchronous, then the request may be serviced by any arbitrary thread.

```
HRESULT RemotingServerReceivingMessage(GUID *pCookie, BOOL fIsAsync)
```

Parameter	Description
[in] pCookie	if remoting GUID cookies are active, this value will correspond with the the value provided in RemotingClientSendingMessage, if the channel succeeds in transmitting the message, and if GUID cookies are active on the client-side process. This allows easy pairing of remoting calls.
[in] fIsAsync	is true if the call is asynchronous.

6.8.6 RemotingServerInvocationStarted

The runtime calls RemotingServerInvocationStarted to notify the profiler that the process is invoking a method due to a remote method invocation request.

```
HRESULT RemotingServerInvocationStarted()
```

6.8.7 RemotingServerInvocationReturned

The runtime calls RemotingServerInvocationReturned to notify the profiler that the process has finished invoking a method due to a remote method invocation request.

```
HRESULT RemotingServerInvocationReturned()
```

6.8.8 RemotingServerSendingReply

The runtime calls RemotingServerSendingReply to notify the profiler that the process has finished processing a remote method invocation request and is about to transmit the reply through a channel.

```
HRESULT RemotingServerSendingReply(GUID *pCookie, BOOL fIsAsync)
```

Parameter	Description
`[in] pCookie`	if remoting GUID cookies are active, this value will correspond with the value provided in RemotingClientReceivingReply, if the channel succeeds in transmitting the message, and if GUID cookies are active on the client-side process. This allows easy pairing of remoting calls.
`[in] fIsAsync`	is true if the call is asynchronous.

6.9 Transitions

6.9.1 UnmanagedToManagedTransition

The runtime calls UnmanagedToManagedTransition to notify the code profiler that a transition from unmanaged code to managed code has occurred. functionId is always the ID of the callee, and reason indicates whether the transition was due to a call into managed code from unmanaged, or a return from an unmanaged function called by a managed one.

Note that if the reason is COR_PRF_TRANSITION_RETURN, then the functioned is that of the unmanaged function, and will never have been jitted. Unmanaged functions still have some basic information associated with them, such as a name, and some metadata.

Note that if the reason is COR_PRF_TRANSITION_RETURN and the callee was a PInvoke call indirect, then the runtime does not know the destination of the call and functionId will be NULL.

Note that if the reason is COR_PRF_TRANSITION_CALL then it may be possible that the callee has not yet been JIT-compiled.

```
HRESULT UnmanagedToManagedTransition(FunctionID functionId,
COR_PRF_TRANSISTION_REASON reason)
```

Parameter	Description
`[in] functionId`	ID of the callee
`[in] reason`	May be either COR_PRF_TRANSITION_CALL or COR_PRF_TRANSITION_RETURN.

6.9.2 ManagedToUnmanagedTransition

The runtime calls ManagedToUnmanagedTransition to notify the code profiler that a transition from managed code to unmanaged code has occurred. functionId is always the ID of the callee, and reason indicates whether the transition was due to a call into unmanaged code from managed, or a return from an managed function called by an unmanaged one.

Note that if the reason is COR_PRF_TRANSITION_CALL, then the functionId is that of the unmanaged function, and will never have been jitted. Unmanaged functions still have some basic information associated with them, such as a name, and some metadata.

Note that if the reason is COR_PRF_TRANSITION_CALL and the callee is a PInvoke call indirect, then the runtime does not know the destination of the call and functionId will be NULL.

```
HRESULT UnmanagedToManagedTransition(FunctionID functionId,
COR_PRF_TRANSISTION_REASON reason)
```

Parameter	Description
[in] functionId	ID of the callee
[in] reason	May be either COR_PRF_TRANSITION_CALL or COR_PRF_TRANSITION_RETURN.

6.9.3 COMClassicWrapperCreated

Notify that the runtime has created a COM-Callable-Wrapper, or CCW; this is a proxy object that allows unmanaged Apps to call managed COM objects

NOTE: currently unimplemented.

```
HRESULT COMClassicWrapperCreated(ClassID wrappedClassId, REFGUID
implementedIID, void* pUnk, ULONG cSlots)
```

Parameter	Description
[in] wrapperClassId	ID of the managed class the CCW gives access to
[in] implementedIID	IID of the interface this CCW provides access to
[in] punk	pointer to the CCW's IUnknown interface
[in] cSlots	number of slots in the CCW vtable

6.9.4 COMClassicWrapperDestroyed

Notify that the runtime has destroyed a CCW (see *COMClassicWrapperCreated*, above)

```
HRESULT COMClassicWrapperDestroyed(ClassID wrappedClassId, REFGUID
implementedIID, void* pUnk)
```

Parameter	Description
[in] wrapperClassId	ID of the managed class the CCW gave access to
[in] implementedIID	IID of the interface this CCW provided access to
[in] punk	pointer to the CCW's IUnknown interface

6.10 Runtime Suspension

6.10.1 RuntimeSuspendStarted

The runtime calls RuntimeSuspendStarted to notify the code profiler that the runtime is about to suspend all of the runtime threads. All runtime threads that are in unmanaged code are permitted to continue running until they try to re-enter the runtime, at which point they will also suspend until the runtime resumes. This also applies to new threads that enter the runtime. All threads within the runtime are either suspended immediately if they are in interruptible code, or asked to suspend when they do reach interruptible code.

suspendReason make be any of the following values:

- COR_PRF_SUSPEND_FOR_GC: the runtime is suspending to service a GC request. The GC-related callbacks will occur between the RuntimeSuspendFinished and RuntimeResumeStarted events.
- COR_PRF_SUSPEND_FOR_CODE_PITCHING: the runtime is suspending so that code pitching may occur. This only occurs when the EJit is active with code pitching enabled. Code pitching callbacks will occur between the RuntimeSuspendFinished and RuntimeResumeStarted events.
- COR_PRF_SUSPEND_FOR_APPDOMAIN_SHUTDOWN: the runtime is suspending so that an AppDomain can be shut down. While the runtime is suspended, the runtime will determine which threads are in the AppDomain that is being shut down, set them to abort when they resume, and then resumes the runtime. There are no AppDomain-specific callbacks during this suspension.
- COR_PRF_SUSPEND_FOR_SHUTDOWN: the runtime is shutting down, and it must suspend all threads to complete the operation.
- COR_PRF_SUSPEND_OTHER: the runtime is suspending for a reason other than those listed above.

```
HRESULT RuntimeSuspendStarted(COR_PRF_SUSPEND_REASON suspendReason)
```

Parameter	Description
`[in] suspendReason`	The reason that the runtime is suspending

6.10.2 RuntimeSuspendFinished

The runtime calls SyncForSuspendFinished to notify the code profiler that the runtime has suspended all threads needed for a runtime suspension. Note that not all runtime threads are required to be suspended, as described in the comment for SyncForSuspendStarted.

NOTE: It is guaranteed that this event will occur on the same ThreadID as RuntimeSuspendStarted occurred on.

```
HRESULT RuntimeSuspendFinished()
```

6.10.3 RuntimeSuspendAborted

The runtime calls RuntimeSuspendAborted to notify the code profiler that the runtime is aborting the runtime suspension that was occurring. This may occur if two threads simultaneously attempt to suspend the runtime.

NOTE: It is guaranteed that this event will occur on the same ThreadID as the RuntimeSuspendStarted occurred on, and that only one of RuntimeSuspendFinished and RuntimeSuspendAborted may occur on a single thread following a RuntimeSuspendStarted event.

```
HRESULT RuntimeSuspendAborted()
```

6.10.4 RuntimeResumeStarted

The runtime calls RuntimeResumeStarted to notify the code profiler that the runtime is about to resume all of the runtime threads.

NOTE: It is guaranteed that this event will occur on the same ThreadID as the RuntimeSuspendStarted occurred on.

```
HRESULT RuntimeResumeStarted()
```

6.10.5 RuntimeResumeFinished

The runtime calls RuntimeResumeFinished to notify the code profiler that the runtime has finished resuming all of it's threads and is now back in normal operation.

NOTE: It is guaranteed that this event will occur on the same ThreadID as the RuntimeSuspendStarted occurred on.

```
HRESULT RuntimeResumeFinished()
```

6.10.6 RuntimeThreadSuspended

The runtime calls ThreadSuspended to notify the code profiler that a particular thread has been suspended. All threads within managed code must be suspended. If a thread is in unmanaged code, it will be allowed to continue, but will suspend

upon re-entering the runtime and will fire this event. Thus, this notification could occur after a suspension has completed, but before the runtime resumes.

```
HRESULT RuntimeThreadSuspended(ThreadID threadId)
```

Parameter	Description
[in] threadId	The ID of the thread that was suspended.

6.10.7 RuntimeThreadResumed

The runtime calls ThreadResumed to notify the code profiler that a particular thread has been resumed after being suspended due to a runtime suspension.

```
HRESULT RuntimeThreadResumed(ThreadID threadId)
```

Parameter	Description
[in] threadId	The ID of the thread that was resumed.

6.11 Garbage Collection

6.11.1 ObjectAllocated

Notify that memory in the GC heap has just been allocated for an object. (This notification does not fire for allocations from the stack, nor from unmanaged memory)

Allocating objects in the heap is likely to be a very frequent operation in an Application. Therefore, this particular notification would fire very often, stealing CPU cycles from the running Application. You must set the COR_PRF_MONITOR_OBJECT_ALLOCATED bit in the notifications mask for these events to fire.

NOTE: not yet implemented.

```
HRESULT ObjectAllocated(ObjectID objectID, ClassID classId)
```

Parameter	Description
[in] objected	ID of the newly-allocated object
[in] classId	ID for the class of which this object is an instance

6.11.2 ObjectsAllocatedByClass

Notify counts of all the objects allocated for each class since the previous garbage collection. Called whilst all threads in the target process are still halted.

This notification provides summary information suitable for building a chart of object creation rates, by class. If that's all you want, it provides a much cheaper way of

obtaining that info than counting each allocation (with the *ObjectAllocated* notification). The arrays omit any classes which have created no objects since the last gc (rather than supply a value of zero in the cObjects[] array)

```
HRESULT ObjectsAllocatedByClass(ULONG cClassCount, ClassID classIds[],
        ULONG cObjects[])
```

Parameter	Description
[in] cClassCount	number of entries in the parallel arrays *classIds[]* and *cObjects[]*
[in] classIds[]	array of IDs for the classes of object allocated
[in] cObjects[]	count of object allocated for each class in *classIds[]*

Example: suppose that since the previous garbage collection, runtime has allocated at total of 35 objects, spread across 4 different classes. Then the notification would have cClassCount = 4, and the parallel arrays classIds[0..3] and cObjects[0..3] might contain the values shown in the table below:

	classIds[]	cObjects[]
0	0x5231 8840	4
1	0x4800 2150	23
2	0x4799 3147	1
3	0x6123 4196	7

6.11.3 MovedReferences

Garbage collection reclaims the memory occupied by 'dead' objects and compacts that freed space. As a result, live objects are moved within the heap. The effect is that ObjectIDs handed out by previous notifications change their value (the internal state of the object itself does not change (other than it's references to other objects), just its location in memory, and therefore its ObjectID). The *MovedReferences* notification lets a profiler update its internal tables that are tracking info by ObjectID.

The number of Objects in the heap can number thousands or millions. With such large numbers, it's impractical to notify their movement by providing a before-and-after ID for each object. However, the garbage collector tends to move contiguous runs of live objects as a 'bunch' – so they end up at new locations in the heap, but they still contiguous. This notification reports the before and after ObjectID of these contiguous runs of objects. (see example below)

In other words, if an ObjectID value lies within the range

oldObjectIDRangeStart[i] <= ObjectID < oldObjectIDRangeStart[i] + cObjectIDRangeLength[i]

for 0 <= i < cMovedObjectIDRanges, then the ObjectID value has changed to

ObjectID - oldObjectIDRangeStart[i] + newObjectIDRangeStart[i]

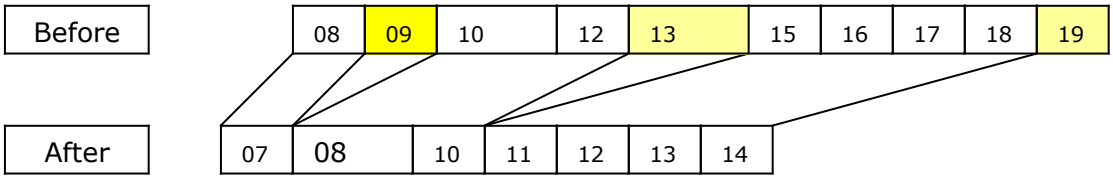
All of these callbacks are made while the runtime is suspended, so none of the ObjectID values can change until the runtime resumes and another GC occurs.

MovedReferences may be invoked multiple times during a GC if the list of moved references exceeds the size of the profiling services' internal buffer.

```
HRESULT MovedReferences(
        ULONG cMovedObjectRefs,
        ObjectID oldObjectRefs[],
        ObjectID newObjectRefs[],
        ULONG cObjectRefSize)
```

Parameter	Description
[in] cMovedObjectIDRanges	a count of the number of ObjectID ranges that were moved.
[in] oldObjectIDRangeStart	an array of elements, each of which is the start value of a range of ObjectID values before being moved.
[in] newObjectIDRangeStart	an array of elements, each of which is the start value of a range of ObjectID values after being moved.
[in] cObjectIDRangeLength	is an array of elements, each of which states the size of the moved ObjectID value range.

Example: The diagram below shows 10 objects, before garbage collection. They lie at start addresses (equivalent to ObjectIDs) of 08, 09, 10, 12, 13, 15, 16, 17, 18 and 19. ObjectIDs 09, 13 and 19 are dead (shown shaded); their space will be reclaimed during garbage collection.



The "After" picture shows how the space occupied by dead objects has been reclaimed to hold live objects. The live objects have been moved in the heap to the new locations shown. As a result, their ObjectIDs all change. The simplistic way to describe these changes is with a table of before-and-after ObjectIDs, like this:

	oldObjectIDRangeStart []	newObjectIDRangeStart []
0	08	07
1	09	
2	10	08
3	12	10
3	13	
4	15	11
5	16	12
6	17	13
7	18	14
8	19	

This works, but clearly, we can compact the information by specifying starts and sizes of contiguous runs, like this:

	oldObjectIDRangeStart []	newObjectIDRangeStart []	cObjectIDRangeLength []
0	08	07	1
1	10	08	2
2	15	11	4

This corresponds to exactly how MovedReferences reports the information

6.11.4 ObjectReferences

The runtime calls ObjectReferences to provide information about objects in memory referenced by a given object. This function is called for each object remaining in the GC heap after a collection has completed. If the profiler returns an error HRESULT from this callback, the profiling services will discontinue invoking this callback until the next GC. This callback can be used in conjunction with the RootReferences callback to create a complete object reference graph for the runtime.

```
HRESULT ObjectReferences(ObjectID objectId, ClassID classId, ULONG
cObjectRefs, ObjectID objectRefIds[]);
```

Parameter	Description
[in] objectId	ID of the object being reported
[in] classId	ID of the class of which the object is an instance
[in] cObjectRefs	number of entries in *objectIds[]*
[in] objectRefIds	array of ObjectIDs contained within *objectId*
[return]	If the code profiler returns E_FAIL, the runtime will halt enumerating the heap. However, the garbage collector continues to traverse the heap. If the code profiler returns S_OK, the heap dump will proceed normally.

Remarks

The runtime will ensure that each object reference is reported only once by *ObjectReferences*.

6.11.5 RootReferences

The runtime calls RootReferences with information about root references after a garbage collection has occurred. Static object references and references to objects on a stack are co-mingled in the arrays. This callback may occur multiple times for a particular GC if the profiling services' internal buffer fills up and there are remaining root references.

```
HRESULT RootReferences(ULONG cRoots, ObjectID objectIds[]);
```

Parameter	Description
[in] cRoots	number of roots listed
[in] objectIds	array of ObjectIDs

Remarks

The application is halted following a COR_PRF_EVENT_GC_FINISHED event until the runtime is done passing information about the heap to the code profiler.

The method *ICorProfilerInfo::GetClassFromObject* can be used to obtain the ClassID of the class of which the object is an instance. The method *ICorProfilerInfo::GetTokenFromClass* can be used to obtain metadata information about the class.

6.12 Exceptions

Notifications of exceptions are the most difficult of all notifications to describe and to understand. This is because of the inherent complexity in exception processing. The set of exception notifications described below was designed to provide all the information required for a sophisticated profiler – so that, at every instant, it can keep track of which pass (first or second), which frame, which filter and which finally block is being executed, for every thread in the profilee process.

A simpler profiler, that tracked cpu-time-by-function might choose to ignore exception notifications entirely – actual cpu time might be attributed to the 'wrong' function by doing so, but results might well still be useful.

Notes:

if a user-code filter throws an exception, all bets are off (runtime may decide to simply swallow that exception -- still TBD)

managed code does not allow a filter to fixup the exception and request continuation from the original throw location. But SEH does -- so later parts of exception processing can be totally missed because of such a "continue execution" user filter in an unmanaged chain. The only way for the profiler to resolve where it is in the shadow stack is to monitor unmanaged exception processing too.

we provide no ThreadIDs, but you can call *GetCurrentthreadID* to discover the identity of the reporting thread

A rethrow in a handler will kick off a fresh exception handling process -- in which case, the *ExceptionCatcherLeave* notification is not sent.

6.12.1 ExceptionThrown

The runtime calls ExceptionThrown to notify the code profiler that an exception has been thrown. This function is only called if the runtime exception handler is called to process an exception.

```
HRESULT ExceptionThrown(ObjectID thrownObjectId)
```

Parameter	Description
`[in] thrownObjectId`	The ID of the Exception object thrown.

6.12.2 ExceptionSearchFunctionEnter

The runtime calls ExceptionSearchFunctionEnter to notify the profiler that the search phase of exception handling has entered a function.

`HRESULT` **`ExceptionSearchFunctionEnter`**(FunctionID functionId)

Parameter	Description
`[in] functionId`	The ID of the function that we're searching for a handler in.

6.12.3 ExceptionSearchFunctionLeave

The runtime calls ExceptionSearchFunctionLeave to notify the profiler that the search phase of exception handling has left a function.

`HRESULT` **`ExceptionSearchFunctionLeave`**`()`

6.12.4 ExceptionSearchFilterEnter

The runtime will call ExceptionSearchFilterEnter just before excecuting a user filter. The functionID is that of the function containing the filter.

`HRESULT` **`ExceptionSearchFilterEnter`**(FunctionID functionId)

Parameter	Description
`[in] functionId`	The ID of the function containing the filter that we are entering.

6.12.5 ExceptionSearchFilterLeave

The runtime will call ExceptionSearchFilterLeave immediately after executing a user filter.

`HRESULT ExceptionSearchFilterLeave()`

6.12.6 ExceptionSearchCatcherFound

The runtime will call ExceptionSearchCatcherFound when the search phase of exception handling has located a handler for the exception that was thrown.

`HRESULT` **`ExceptionSearchCatcherFound`**(FunctionID functionId)

Parameter	Description
`[in] functionId`	The ID of the function that will handle the exception.

6.12.7 ExceptionOSHandlerEnter

The runtime calls ExceptionOSHandlerEnter when the runtime's [Win32 SEH] exception handler is entered AND there is at least one JIT'ed function guarded by that instance of the handler. This function will not be called if there is no managed code guarded by the instance of the handler, nor if there is only internal runtime code guarded by the instance of the handler. This notification is provided to allow profilers to detect unmanaged-to-managed transitions in stack searches and unwinds. The functionID is that of the first function encountered on the search or unwind.

The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

NOTE: This callback is currently innacurate - this will be addressed at a later date.

`HRESULT` **`ExceptionOSHandlerEnter`**(FunctionID functionId)

Parameter	Description
`[in] functionId`	The ID of the first function encountered on the search or unwind.

6.12.8 ExceptionOSHandlerLeave

This function is similar to ExceptionOSHandlerEnter, except that it is called just before the runtime's exception handler returns. The functionID is that of the last function encountered on the search or unwind.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

NOTE: This callback is currently inaccurate - this will be addressed at a later date.

`HRESULT` **`ExceptionOSHandlerLeave`**(FunctionID functionId)

Parameter	Description
`[in] functionId`	The ID of the last function encountered on the search or unwind.

6.12.9 ExceptionUnwindFunctionEnter

The runtime calls ExceptionUnwindFunctionEnter to notify the profiler that the unwind phase of exception handling has entered a function.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

```
HRESULT ExceptionUnwindFunctionEnter(FunctionID functionId)
```

Parameter	Description
`[in] functionId`	The ID of the function that is being unwound from the stack.

6.12.10 ExceptionUnwindFunctionLeave

The runtime calls ExceptionUnwindFunctionLeave to notify the profiler that the unwind phase of exception handling has left a function. The function instance and it's stack data has now been removed from the stack.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

```
HRESULT ExceptionUnwindFunctionLeave()
```

6.12.11 ExceptionUnwindFinallyEnter

The runtime calls ExceptionUnwindFinallyEnter to notify the profiler that the unwind phase of exception is entering a finally clause contained in the specified function.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

```
HRESULT ExceptionUnwindFinallyEnter(FunctionID functionId)
```

Parameter	Description
`[in] functionId`	The ID of the function whose finally clause is being executed.

6.12.12 ExceptionUnwindFinallyLeave

The runtime calls ExceptionUnwindFinallyLeave to notify the profiler that the unwind phase of exception is leaving a finally clause.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

```
HRESULT ExceptionUnwindFinallyLeave()
```

6.12.13 ExceptionCatcherEnter

The runtime calls this function just before passing control to the appropriate catch block. Note that this is called only if the catch point is in JIT'ed code. An exception that is caught in unmanaged code, or in the internal code of the runtime will not generate this notification. The ObjectID is passed again since a GC could have moved the object since the ExceptionThrown notification.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

```
HRESULT ExceptionCatcherEnter(FunctionID functionId, ObjectID objectId)
```

Parameter	Description
`[in] functionId`	The ID of the function containing the catch clause.
`[in] objectId`	The ID of the thrown Exception object.

6.12.14 ExceptionCatcherLeave

The runtime calls ExceptionCatcherLeave when the runtime leaves the catcher's code.

NOTE: The profiler cannot block here, since the stack may not be in a GC-friendly state and so preemptive GC cannot be enabled. If the profiler blocks here and a GC is attempted, the runtime will block until this callback returns. Also, the profiler may NOT call into managed code or in any way cause a managed memory allocation.

```
HRESULT ExceptionCatcherLeave()
```

The runtime code profiler interfaces do not support remote profiling due to the following reasons:

- It is necessary to minimize execution time using these interfaces so that profiling results will not be unduly affected. This is especially true where execution performance is being monitored. However, it is not a limitation when the interfaces are used to monitor memory usage or to obtain runtime information on stack frames, objects, etc.
- The code profiler needs to register one or more callback interfaces with the runtime on the local machine on which the application being profiled runs. This limits the ability to create a remote code profiler.

7 ICorProfilerInfo

The *runtime provides the ICorProfilerInfo interface*. It allows a profiler to ask for info about classes, function, code, stack frames, etc within the running process. It includes a small number of methods that allow the profiler to change this info; for example, to provide new IL for a function and request that be re-JIT compiled. You can think of the *ICorProfilerInfo* interface as the "help desk" for profilers.

The runtime provides the *ICorProfiler* interface to each profiler on its very first call – *ICorProfilerCallback::Initialize*

The runtime uses the free threaded model to implement the *ICorProfilerInfo* interface. Events are dispatched from within the runtime or on a thread that is making the code profiler method call. Interface methods implemented by the runtime can be called from any thread (that has been *CoInitialized*) at any time.

The methods in *ICorProfilerInfo* return S_OK on success, or E_FAIL on failure.

7.1 ForceGC

The code profiler calls *ForceGC* to force a garbage collection to occur.

NOTE: currently unimplemented.

Syntax

```
HRESULT ForceGC();
```

7.2 GetAppDomainInfo

The code profiler calls *GetAppDomainInfo* to obtain information about a given application domain.

Syntax

```
HRESULT GetAppDomainInfo(AppDomainID appDomainId, SIZE_T cchName,
SIZE_T *pcchName, WCHAR szName[], ProcessID *pProcessId);
```

Parameter	Description
[in] appDomainId	AppDomainID of the given application domain.
[in] cchName	The allocated size of string buffer for the application domain name.
[out] pcchName	The length of the string returned in the string buffer
[out] szName	The string buffer for the application domain name.

7.3 GetAssemblyInfo

The code profiler calls *GetAssemblyInfo* to obtain information about a given assembly.

Syntax

```
HRESULT GetAssemblyInfo(AssemblyID assemblyId, SIZE_T cchName, SIZE_T
*pcchName, WCHAR szName[], AppDomainID *pAppDomainId, ModuleID
*pModuleId);
```

Parameter	Description
[in] assemblyId	AssemblyID of the given assembly.
[in] cchName	The allocated size of string buffer for the assembly name.
[out] pcchName	The length of the string returned in the string buffer
[out] szName	The string buffer for the assembly name.
[out] pAppDomainId	Pointer to the AppDomainID of the application domain that contains the assembly.
[out] pModuleId	Pointer to the ModuleID of the module that contains the assembly's manifest.

7.4 GetClassFromObject

The code profiler calls *GetClassFromObject* to obtain the ClassID of an object given its ObjectID.

Syntax

```
HRESULT GetClassFromObject(ObjectID objectId, ClassID *pClassId);
```

Parameter	Description
[in] objectId	The ObjectID of the object the code profiler is interested in.
[out] pClassId	Pointer to the ClassID of the class of the object.

7.5 GetClassFromToken

The code profiler calls *GetClassFromToken* to obtain the ClassID of a class given its metadata.

Syntax

```
HRESULT GetClassFromToken(ModuleID moduleId, mdTypeDef typeDef, ClassID
*pClassId);
```

Parameter	Description
[in] moduleId	The ModuleID of the module the class is defined in.

`[in] typeDef`	The metadata typedef token for the class.
`[out] pClassId`	Pointer to the ClassID of the class the code profiler is interested in.

7.6 GetClassIDInfo

Returns the parent module that a class is defined in, along with the metadata token for the class. One can call GetModuleInfo to get the metadata interface for the ModuleID returned. The token can then be used to access the metadata for this class.

Syntax

```
HRESULT GetClassIDInfo(ClassID classId, ModuleID *pModuleId, mdTypeDef *pTypeDefToken)
```

Parameter	Description
`[in] classId`	The ClassID of the class the code profiler is interested in.
`[out] pModuleId`	Pointer to the ModuleID of the module in which the class is defined.
`[out] pTypeDefToken`	Pointer to the metadata typedef token for the class.

7.7 GetCodeInfo

The code profiler calls *GetCodeInfo* to obtain information about a JIT-compiled function. An error will be returned if GetCodeInfo is called with a FunctionID for a function that has not been JIT-compiled.

Syntax

```
HRESULT GetCodeInfo(FunctionID functionId, LPCBYTE *pStart, ULONG *pcSize)
```

Parameter	Description
[in] functionId	The FunctionID of the function the code profiler is interested in.
[out] pStart	The starting address of the JIT-compiled code.
[out] pcSize	The size of the JIT-compiled code in bytes.

Remarks

This method must be called after the code profiler has received notification that the function has been JIT-compiled.

7.8 GetEventMask

The code profiler calls *GetEventMask* to obtain the current event categories for which it is to receive event notification from the runtime.

Syntax

```
HRESULT GetEventMask (DWORD *pdwEvents);
```

Parameter	Description
[out] pdwEvents	Pointer to the bit mask of flags from COR_PRF_MONITOR indicating the events for which the code profiler is to receive notification.

Remarks

Code profilers can receive notification for any combination of the event categories defined in the COR_PRF_MONITOR enumeration.

7.9 GetFunctionFromIP

The code profiler calls *GetFunctionFromIP* to map an instruction pointer in managed code to a FunctionID.

Syntax

```
HRESULT GetFunctionFromIP (ULONG64 ip, FunctionID *pFunctionId);
```

Parameter	Description
[in] ip	The instruction pointer that the code profiler is interested in
[out] pFunctionId	Pointer to the FunctionID of the function that corresponds to the instruction pointer.

Remarks

The code profiler can call *GetCodeInfo* to obtain information about the size and starting address of the function. *GetFunctionFromIP* returns E_FAIL if it is unable to map the instruction pointer. The runtime may choose to unload a function to recover memory. In such instances, the instruction pointer mapping becomes invalid. The runtime generates a COR_PRF_EVENT_FUNCTION_UNLOAD_STARTED event. In response to this event, the code profiler should call *GetILOffsetFromIP* to map saved instruction pointers that fall within the function to IL offsets from the beginning of the function.

The method returns E_FAIL if the function is not managed code.

7.10 GetFunctionFromToken

The code profiler calls *GetFunctionFromToken* to obtain the FunctionID of a function given its metadata.

Syntax

```
HRESULT GetFunctionFromToken(ModuleID moduleId, mdToken token,
FunctionID *pFunctionId);
```

Parameter	Description
[in] moduleId	The ModuleID of the module the function is defined in.
[in] token	The metadata token for the function.
[out] pFunctionId	Pointer to the FunctionID of the function the code profiler is interested in.

7.11 GetFunctionInfo

The code profiler calls *GetFunctionInfo* to obtain metadata information about a method in a class or a function at a module level given the function's FunctionID.

Syntax

```
HRESULT GetFunctionInfo(FunctionID functionId, ClassID *pClassId,
ModuleID *pModuleId, mdToken *pToken)
```

Parameter	Description
[in] functionId	The FunctionID of the function the code profiler is interested in.
[out] pClassId	Pointer to the ClassID of the class in which the function is defined.
[out] pModuleId	Pointer to the ModuleID of the module in which the function is defined.
[out] pToken	Pointer to the metadata token for the function.

7.12 GetHandleFromThread

The code profiler calls *GetHandleFromThread* to map a ThreadID to a Win32 thread handle.

Syntax

```
HRESULT GetHandleFromThread(ThreadID threadId, HANDLE *phThread);
```

Parameter	Description
[in] threadId	The ThreadID of the thread the code profiler is interested in.

`[out] phThread`	Pointer to the Win32 thread handle.

7.13 GetILFunctionBodyAllocator

IL method bodies must be located as RVA's to the loaded module, which means that they come after the module within 4 GB. In order to make it easier for a tool to swap out the body of a method, this allocator will ensure memory allocated after that point.

Syntax

```
HRESULT GetILFunctionBodyAllocator(ModuleID moduleId, IMethodAlloc
**ppMalloc);
```

Parameter	Description
`[in] moduleId`	ModuleID of the given module.
`[in] ppMalloc`	Pointer to pointer to memory allocator for method.

7.14 GetILFunctionBody

The code profiler calls *GetILFunctionBody* to obtain a pointer to the body of a method starting at its header. A method is scoped by the module it lives in. Because this function is designed to give a tool access to IL before it has been loaded by the runtime, it uses the metadata token of the method to find the instance desired. Note that this function has no effect on already compiled code.

Syntax

```
HRESULT GetILFunctionBody(ModuleID moduleId, mdMethodDef method,
LPCBYTE **ppMethodHeader, ULONG64 *pcbMethodSize);
```

Parameter	Description
`[in] moduleId`	ModuleID of the given module.
`[in] method`	Metadata token for method.
`[out] ppMethodHeader`	Pointer to the method header (IMAGE_COR_ILMETHOD)
`[out] pcbMethodSize`	Pointer to the size of the method.

7.15 GetModuleInfo

The code profiler calls *GetModuleInfo* to obtain information about a given module.

Syntax

```
HRESULT GetModuleInfo(ModuleID moduleId, LPCBYTE **ppBaseLoadAddress,
SIZE_T cchName, SIZE_T *pcchName, WCHAR szName[], mdModule
*pModuleToken, AssemblyID *pAssemblyId);
```

Parameter	Description
[in] moduleId	ModuleID of the given module.
[out] ppBaseLoadAddress	Pointer to the base address of the module.
[in] cchName	The allocated size of string buffer for module name.
[out] pcchName	The length of the string returned in the string buffer
[out] szName	The string buffer for the module name.
[out] pModuleToken	Pointer to metadata token for the module.
[out] pAssemblyId	Pointer to the assembly ID of the assembly that contains the module. If GetModuleInfo is called before the module is attached to the its parent assembly, the returned value for pAssemblyId will be the constant PROFILER_PARENT_UNKNOWN.

7.16 GetModuleMetaData

The code profiler calls *GetModuleMetaData* to obtain a metadata interface instance which maps to the given module. One may ask for the metadata to be opened in read+write mode, but this will result in slower metadata execution of the program, because changes made to the metadata cannot be optimized as they were from the compiler.

Syntax

```
HRESULT GetModuleMetaData(ModuleID moduleId, DWORD dwOpenFlags, REFIID
riid, IUnknown **ppOut);
```

Parameter	Description
[in] moduleId	ModuleID of the given module.
[in] dwOpenFlags	Mode flags for opening metadata.
[in] riid	The REFIID of the metadata interface.
[out] ppOut	Pointer to the pointer to the returned metadata interface that maps to the given module.

7.17 GetObjectSize

The code profiler calls *GetObjectSize* to obtain the instance size of an object.

Syntax

```
HRESULT GetObjectSize(ObjectID objectId, ULONG32 *pcSize);
```

Parameter	Description
[in] objectId	The ObjectID of the object the code profiler is interested in.
[out] pcSize	Pointer to the size of the object in memory in bytes.

7.18 GetStaticClassSize

The code profiler calls *GetClassInfo* to obtain the size of static data in a class.

Note: *This method is not yet implemented.*

Syntax

```
HRESULT GetStaticClassSize(ClassID classId, ULONG64 *pcStaticSize);
```

Parameter	Description
[in] classId	The ClassID of the class the code profiler is interested in.
[out] pcStaticSize	Pointer to the size of the static data in the class.

7.19 GetThreadInfo

The code profiler calls *GetThreadInfo* to obtain the Win32 thread ID for the specified thread.

Syntax

```
HRESULT GetThreadInfo(ThreadID threadId, DWORD *pdwWin32ThreadId);
```

Parameter	Description
[in] threaded	The ThreadID of the thread the code profiler is interested in.
[out] pdwWin32ThreadId	Pointer to the Win32 thread ID.

7.20 GetCurrentThreadID

The code profiler calls GetCurrentThreadID to get the managed thread ID for the current thread.

Syntax

```
HRESULT GetThreadInfo(ThreadID *pThreadId);
```

Parameter	Description
[out] pThreadId	Pointer to the ThreadID to set.

7.21 SetEnterLeaveFunctionHooks

The code profiler calls SetFunctionHooks to specify its own callback replacements for ICorProfilerCallback::FunctionEntry, ICorProfilerCallback::FunctionExit and ICorProfilerCallback::FunctionTailcall

Syntax

```
HRESULT SetEnterLeaveFunctionHooks(
            FunctionEnter    *pFuncEnter,
            FunctionLeave    *pFuncLeave,
            FunctionTailcall *pFuncTailcall)
```

Parameter	Description
[in] pFuncEnter	Pointer to code profiler supplied function to be used as callback on entry to functions.
[in] pFuncLeave	Pointer to code profiler supplied function to be used as callback on exit from functions.
[in] pFuncTailcall	Pointer to code profiler supplied function to be used as callback on tailcall exit from functions.

7.22 SetEventMask

The code profiler calls *SetEventMask* to sets the event categories (see COR_PRF_MONITOR) for which it is set to receive notification from the runtime.

All events but those contained in COR_PRF_MONITOR_IMMUTABLE may be set at any time.

Syntax

```
HRESULT SetEventMask(DWORD dwEvents);
```

Parameter	Description
[in] dwEvents	A bit mask of flags from COR_PRF_MONITOR indicating which events the code profiler wants to receive notification for.

7.23 SetFunctionIDMapper

The code profiler calls *SetFunctionIDMapper* to specify the function to be called to map FunctionIDs to alternative value to be passed to the function entry and function exit hooks. See the description of *ICorProfilerInfo::SetEnterLeaveFunctionHooks*.

Syntax

```
HRESULT SetFunctionIDMapper(FunctionIDMapper *pFunction)
```

Parameter	Description
`[in] pFunction`	Pointer to the function to be called to map a FunctionID to an alternative value to be passed to the function entry and function exit hooks.

7.24 SetFunctionReJIT

The code profiler calls *SetFunctionReJIT* to mark a function as requiring JIT recompilation. The function will be JIT recompiled at its next invocation. The normal profiler events will give the profiler an opportunity to replace the IL prior to the JIT. By this means, a tool can effectively replace a function at runtime. Not that active instances of the function are not affected by the replacement.

Syntax

```
HRESULT SetFunctionReJIT(FunctionID functionId)
```

Parameter	Description
`[in] functionId`	FunctionID of the function to be JIT recompiled.

7.25 SetILFunctionBody

The code profiler calls *SetILFunctionBody* to set the method body of a function in a module. This will replace the RVA of the method in the metadata to point to this new method body, and adjust any internal data structures as desired. This function can only be called on those methods that have never been compiled by a JIT-compiler. Use the *GetILFunctionBodyAllocator* method to allocate space for the new method to ensure the buffer is compatible.

Syntax

```
HRESULT SetILFunctionBody(ModuleID moduleId, mdMethodDef method,
LPCBYTE pbNewILMethodHeader, ULONG cbNewMethod);
```

Parameter	Description
`[in] moduleId`	ModuleID of the given module.
`[in] method`	Metadata token for method.
`[in] pbNewILMethodHeader`	Pointer to the new IL method header.
`[in] cbNewMethod`	Pointer to the size of the new IL method header.

7.26 SetILInstrumentedCodeMap

The code profiler calls *SetILInstrumentedCodeMap* to tell the runtime that the IL map for a function has changed.

In COR_IL_MAP, each *oldOffset* refers to the IL offset within the original unmodified IL code. *newOffset* refers to the corresponding IL offset within the new, instrumented code.

If the offset of the original IL is exactly equal to an *oldOffset*, then it's new offset within the new function body is given by *newOffset*. If the original offset is not equal to an *oldOffset*, then the new offset is equal to the value of the expression

rgILMapEntries[i].newOffset – rgILMapEntries[i].oldOffset + originalOffset

where 0 <= I < cILMapEntries and

rgILMapEntries[I].oldOffset < originalOffset,

and there does not exist a *j* such that

rgILMapEntries[I].oldOffset < rgILMapEntries[j].oldOffset < originalOffset

Syntax

HRESULT **SetILInstrumentedCodeMap**(FunctionID functionId, BOOL fStartJit, SIZE_T cILMapEntries, COR_IL_MAP rgILMapEntries[]);

Parameter	Description
[in] functionId	FunctionID of the function for which the code map is being set.
[in] fStartJit	A Boolean that should be true to indicate that the invocation is in advance of JIT compilation of the function. It should be false if this method is being called to only change the function's IL map.
[in] cILMapEntries	Number of entries in the rgILMapEntries array.
[in] rgILMapEntries	An array of entries that specify how the old IL offsets map to the new IL offsets.

7.27 SetILMapFlag

The code profiler calls *SetILMapFlag* to request the runtime to maintain information about IL mapping. This information is used to map an instruction pointer to an internal point within a function.

Note: *This method is not yet implemented.*

Syntax

```
HRESULT SetILMapFlag();
```

Remarks

A sampling code profiler should enable this flag. Non-sampling code profilers don't need to enable this flag since they often only need to map the instruction pointers corresponding to function entry points and function exit points. The runtime can do this mapping without an IL mapping table.

7.28 GetInprocInspectionInterface

7.29 GetInprocInspectionThisThread

7.30 GetThreadcontext

7.31 GetTokenAndMetadataFromFunction

8 Memory Allocation Interface (IMethodMalloc : IUnknown)

This is the interface to a very simple allocator that only allows you to allocate memory. You may not free it. This interface should be used in conjunction with *SetILMethodBody*.

8.1 Alloc

A profiler calls *Alloc* to allocate memory in conjunction with *SetILMethodBody*.

Syntax

```
HRESULT Alloc(ULONG cb);
```

Parameter	Description
[in] cb	Size of the memory to be allocated.

9 Profiling Enumerations

9.1 COR_PRF_MONITOR

The notification methods mentioned in the following table are all defined on the *ICorProfilerCallback* interface.

Event Category	Value	Description
COR_PRF_MONITOR_NONE	0x0	Do not send notifications for any event categories.
COR_PRF_MONITOR_FUNCTION_UNLOADS	0x1	Send event notification when a function is unloaded.
COR_PRF_MONITOR_CLASS_LOADS	0x2	Send event notification when a class is loaded or when the class is unloaded.
COR_PRF_MONITOR_MODULE_LOADS	0x4	Send event notification when a runtime module is loaded or when the module is unloaded.
COR_PROF_MONITOR_ASSEMBLY_LOADS	0x8	Send event notification when a runtime module is loaded or when the module is unloaded.
COR_PRF_MONITOR_APPDOMAIN_LOADS	0x10	Send event notification when an application domain is created or when an application domain is shutdown.
COR_PRF_MONITOR_ CALLS	0x20	Send event notification when a function is about to be called and upon completion of the function call.
COR_PRF_MONITOR_JIT_COMPILATION	0x40	Send event notification when a function is about to be JIT-compiled and after JIT compilation is completed.
COR_PRF_MONITOR_EXCEPTIONS	0x80	Send event notification when an exception occurs.
COR_PRF_MONITOR_GC	0x100	Send event notification when a garbage collection is about to occur.
COR_PRF_MONITOR_OBJECT_ALLOCATED	0x200	Send event notification when an object is allocated on the heap.
COR_PRF_MONITOR_THREADS	0x400	Send event notification when a thread is about to be created or destroyed.
COR_PRF_MONITOR_CONTEXT_CROSSINGS	0x800	Send event notification when a context crossing occurs.
COR_PRF_MONITOR_SECURITY_CHECKS	0x1000	Send event notification when a security check occurs.
COR_PRF_MONITOR_CODE_TRANSITIONS	0x2000	Send event notification when a code transition occurs from managed to unmanaged code or vice versa.

COR_PRF_MONITOR_SYNCHRONIZATIONS	0x4000	Send event notification when a synchronization event occurs.
COR_PRF_MONITOR_ALLOW_REJIT	0x8000	Allow JIT recompilation of methods.
COR_PRF_MONITOR_ENTERLEAVE	0x10000	Call the function entry hook when a function is entered. Call the function exit hook when a function is exited.
COR_PRF_MONITOR_ALL	0xFFFF	All of the above.

9.2 COR_PRF_ID

Enumeration	Value	Description
COR_PRF_CHAIN_ID	1	Stack chain ID
COR_PRF_CLASS_ID	2	Class ID
COR_PRF_CONTEXT_ID	3	Context ID
COR_PRF_FIELD_ID	4	Class field ID
COR_PRF_FRAME_ID	5	Stack frame ID
COR_PRF_FUNCTION_ID	6	Function ID
COR_PRF_MODULE_ID	7	Module ID
COR_PRF_MONITOR_ID	8	Monitor ID
COR_PRF_OBJECT_ID	9	Object ID of class instance
COR_PRF_THREAD_ID	10	Thread ID

10Profiling Type Definitions

10.1 COR_IL_MAP

The COR_IL_MAP type is used to describe how an IL offset in an old function body maps to the IL offset in the new function body that replaces the old function body. See the ICorProfilerInfo::SetILInstrumentedCodeMap for a description of how this type is used.

IDL Declaration

```
typedef struct _COR_IL_MAP
{
    SIZE_T oldOffset;
    SIZE_T newOffset;
} COR_IL_MAP;
```

Members

Member	Description
oldOffset	IL offset in the old function body.
newOffset	IL offset in the new function body.

10.2 COR_PRF_JIT_MAP

The COR_PRF_JIT_MAP notifies a profiler about the result of a search for a cached function.

IDL Declaration

```
typedef enum
{
    COR_PRF_CACHCED_FUNCTION_FOUND,
    COR_PRF_CACHCED_FUNCTION_NOT_FOUND
} COR_PRF_JIT_CACHE;
```

Members

Member	Description
COR_PRF_CACHED_FUNCTION_FOUND	The search for the cached function was successful.
COR_PRF_CACHED_FUNCTION_NOT_FOUND	The search for the cached function was unsuccessful.

10.3 FunctionIDMapper

The *FunctionIDMapper* type definition is used by the *ICorProfilerInfo::SetFunctionIDMapper* method to specify a function that will be called to map FunctionIDs to alternative values that will be passed to the function entry and function exit callbacks supplied to the *ICorProfilerInfo::SetEnterLeaveFunctionHooks* method.

IDL Declaration

typedef void __stdcall FunctionIDMapper(FunctionID functionId, BOOL *pbHookFunction);

Parameter	Description
functionId	FunctionID of the function for which the mapping is requested.
pbHookFunction	Pointer to a function that is called to provide the alternative value to be passed to the function entry and function exit callbacks.

10.4 FunctionEnter

The FunctionEnter type definition describes the signature of the function entry callback supplied to the *ICorProfilerInfo::SetEnterLeaveFunctionHooks* method.

IDL Declaration

typedef void __stdcall FunctionEnter(FunctionID functionId);

Parameter	Description
functionId	FunctionID of the function that was entered.

10.5 FunctionExit

The FunctionExit type definition describes the signature of the function exit callback supplied to the *ICorProfilerInfo::SetEnterLeaveFunctionHooks* method.

IDL Declaration

typedef void __stdcall FunctionExit(FunctionID functionId);

Parameter	Description
functionId	FunctionID of the function that was exited.

10.6 FunctionTailcall

The FunctionTailcall type definition describes the signature of the function tail call callback supplied to the *ICorProfilerInfo::SetEnterLeaveFunctionHooks* method.

IDL Declaration

typedef void __stdcall FunctionTailcall(FunctionID functionId);

Parameter	Description
functionId	FunctionID of the function that was exited with a tail call.

NOTE: We need to explain here why we are using __declspec(naked) and how to use it.

11Profiler Picker

The Profiler Picker tool (see figure below) in the SDK can be used to select a specific profiler for profiling an application.

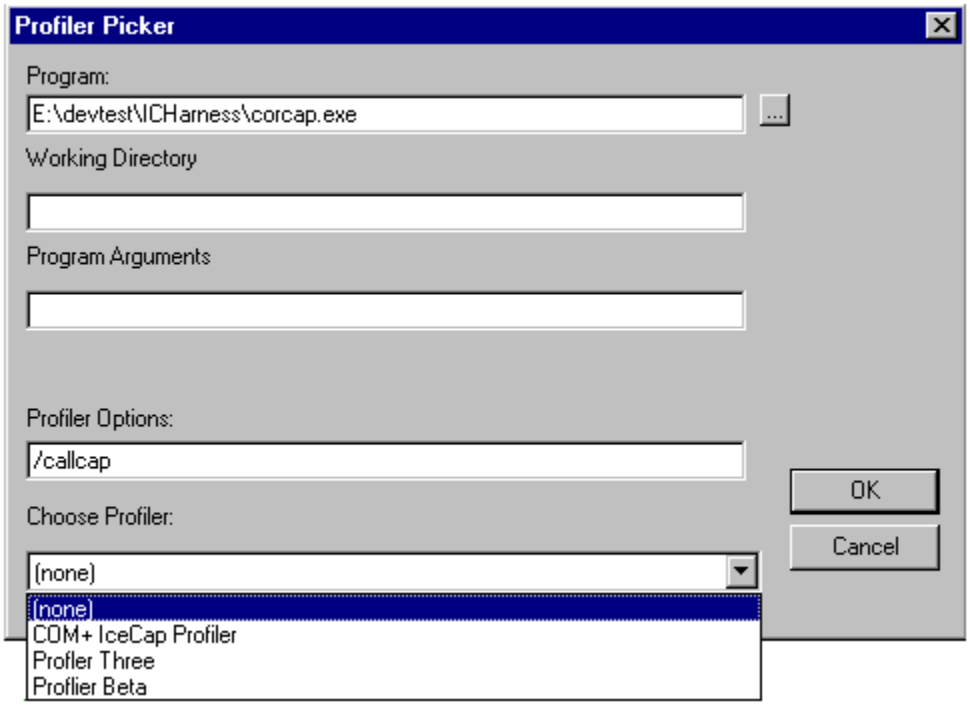


Figure. Profile Picker

Profiler Picker allows a user to specify command-line options for the selected profiler. Profiler Picker also allows a user to specify the application to be profiled, specify the command-line arguments to the application, and specify the working directory for launching the application. Profile Picker sets the environment variable COR_ENABLE_PROFILING to enable profiling and sets the environment variable COR_PROFILER to the selected profiler. The runtime maintains a list of registered profilers under the registry key `HKLM\software\microsoft\COMplus\Profilers`. Each profiler adds a sub-key with its pretty name. Each sub-key has a “HelpString” value that describes the profiler and a "ProfilerID" value which is the CLSID needed to instantiate the profiler.

12Security Issues in Profiling

The runtime Profiling Services are available in-process to a code profiler. The Profiling Services allow a code profiler to instrument code dynamically. In dynamic code instrumentation, the code profiler calls Profiling Services methods to replace methods in the profiled process with methods supplied by the code profiler. These modifications violate the enforcement of security imposed on the code in various ways that cannot be controlled easily by the runtime. The runtime does not provide a complex set of security checks. Instead, the profiler must have sufficient operating system privileges to profile a process.

This section describes a few scenarios in the context of runtime security and dynamic code instrumentation.

Dynamically Swapping Methods. In this scenario, a user runs an application under the control of a code profiler. The code being profiled is verified when it is loaded. After the application has been executing for sometime, the user decides to investigate why the code is running slow. The user uses the code profiler to dynamically instrument selected parts of the application. The code profiler selects some methods and requests the runtime to forget that the methods were compiled by the JIT compiler. The code profiler then supplies replacements for the methods. The replacement methods are not verified when they are compiled by the JIT compiler. The new code has the potential to violate runtime security.

Verification of On-disk Instrumented Code. In this scenario, a code profiler chooses to instrument code by modifying a PE on disk before it is loaded. The new checksum for the containing assembly will not match the checksum stored in the assembly's manifest. The code profiler needs to inform the runtime that the PE has been modified and request the loader to omit certain checks. The method to do this is TBD.

13Design Considerations

The runtime code profiler interfaces are designed for speed to reduce the overhead of making calls on the methods supported by the interfaces. To ensure speed, the code profiler interfaces return arrays instead of enumerators.

The runtime code profiler interfaces provide low-level access to runtime information. Interfaces that return static information about the application are provided only where necessary. The code profiler must rely on other interfaces such as metadata APIs to obtain such information.

Note that the profiling APIs are used, not by end-user applications, but by software tools. We have therefore traded speed against robust checking – argument validation is quite minimal, so misuse of the APIs can easily result in crashing the profilee's process

14Unmanaged Code

There is minimal support in the runtime profiling interfaces for profiling unmanaged code. The following functionality is provided:

- Enumeration of stack chains. This allows a code profiler to determine the boundary between managed code and unmanaged code.
- Determine if a stack chain corresponds to managed code.

These methods are available thru the "In-Proc" subset of the NGWS debugging APIs. These are defined in the file "CorDebug.IDL"