Common Language Runtime

Profiling

This is preliminary documentation and subject to change

Last updated: 2 July 2001

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 Common Language Runtime (CLR). 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 that the term *routine* is being in this document to indicate 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 a CLR program requires more support than profiling conventionally compiled machine code. This is because the CLR has introduced new concepts such as application domains, garbage collection, managed exception handling, JIT compilation of code (converting Microsoft Intermediate Language into native machine code) etc that the existing conventional profiling mechanisms are unable to identify and provide useful information. The profiling APIs provide this missing information in an efficient way that causes minimal impact on the performance of the CLR.

Note that JIT-compiling routines at runtime provide good opportunities, as the APIs allow a profiler to change the in-memory MSIL 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 CLR.

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 CLR. Specifically:
  - Common Language Runtime startup and shutdown events
  - Application domain creation and shutdown events
  - Assembly loading and unloading events
  - Module load/unload events
  - Com VTable 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*
  - 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 CLR 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 CLR provides intrinsic support for bounds checking of all managed code.

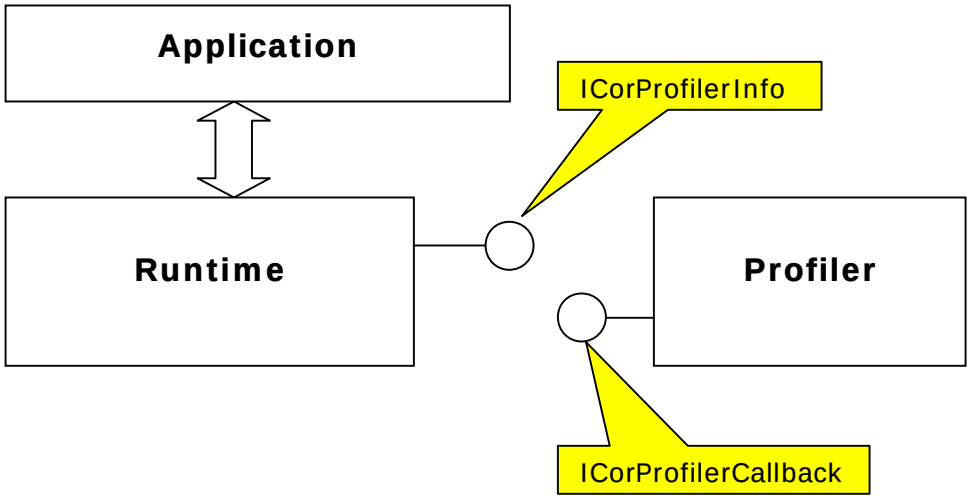
The CLR code profiler interfaces do not support remote profiling due to the following reasons:

- 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.

4 Profiling APIs – Overview

The profiling APIs within CLR allow to the user 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 CLR 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 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.

The *ICorProfilerCallback* interface can be considered to be 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 CLR 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 CLR 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 CLR 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 *CoInitialize* 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_ENTERLEAVE |
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 the user 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)

Note that only one profiler can be profiling a process at one time in a given environment. In different environments it is possible to have two different profilers registered in each environment, each profiling separate processes..

Certain profiler events are IMMUTABLE which means that once they are set in the *IcorProfilerCallback::Initialize* callback they cannot be turned off using ICorProfilerInfo::SetEventMask(). Trying to change an immutable event will result in SetEventMask returning a failed HRESULT.

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. If an invalid ID is used in any API then the results are undefined. Most likely, the result will be an access violation. The user has to ensure that the ID's used are perfectly valid. The profiler does not perform any type of validation since that would create overhead and it would slow down the execution considerably.

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 CLR 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

A profiler returns a status, as an HRESULT, for each notification triggered by the CLR. That status may have the value S_OK or E_FAIL. Currently 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 documents, which thread does the call – either the thread, which generated the event, or some utility thread (e.g. garbage collector) within the Runtime. For any callback that might be invoked by a different thread, a user can call the *ICorProfilerInfo::GetCurrentThreadID* to discover the thread that generated the event.

Note that these callbacks are not serialized. The user must protect his code appropriately by creating thread safe data structures and by locking the profiler code where necessary to prevent parallel access from multiple threads. Therefore, in certain cases it is possible to receive an unusual sequence of callbacks. For example assume a managed application is spawning two threads, which are executing identical code. In this case it is possible to receive a JITCompilationStarted event for some function from one thread and before receiving the respective JITCompilationFinished callback, the other thread has already sent a FunctionEnter callback. Therefore the user will receive a FunctionEnter callback for a function that it seems not fully JIT compiled yet!

5.4 Nesting of Notifications

Notifications to the 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 notifications; and so on. The nesting looks like this:

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

5.5 GC-Safe Callouts

When the CLR calls certain functions in the *ICorProfilerCallback*, the Runtime cannot perform a garbage collection until the Profiler returns control from that call. This is because profiling services cannot always construct the stack into a state that is safe for a garbage collection; instead we disable garbage collection around that callback. For these cases, the Profiler should take care to return control as soon as possible. The callbacks where this applies are:

- FunctionEnter, FunctionLeave, FunctionTailCall
- ExceptionOSHandlerEnter, ExceptionOSHandlerLeave
- ExceptionUnwindFunctionEnter, ExceptionUnwindFunctionLeave
- ExceptionUnwindFinallyEnter, ExceptionUnwindFinallyLeave
- ExceptionCatcherEnter, ExceptionCatcherLeave
- ExceptionCLRCatcherFound, ExceptionCLRCatcherExecute
- COMClassicVTableCreated, COMClassicVTableDestroyed

In addition, the following callbacks may or may not allow the Profiler to block. This is indicated, call-by-call, via the fIsSafeToBlock argument. This set includes:

- JITCompilationStarted, JITCompilationFinished

Note that if the Profiler *does* block, it will delay garbage collection. This is harmless, as long as the Profiler code itself does not attempt to allocate space in the managed heap, which could induce deadlock.

5.6 How to profile a NT Service

Profiling is enabled through environment variables, and since NT Services are started when the Operating System boots, those environment variables must be present and set to the required value at that time. Thus, to profile an NT Service, the appropriate environment variables must be set in advance, system-wide, via:

> MyComputer -> Properties -> Advanced -> EnvironmentVariables -> System Variables

Both **Cor_Enable_Profiling** and **COR_PROFILER have to be set**, and the user must ensure that the Profiler DLL is registered. Then, the target machine should be re-booted so that the NT Services pick up those changes. Note that this will enable profiling on a system-wide basis. So, to prevent every managed application that is run subsequently from being profiled, the user should delete those system environment variables after the re-boot.

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 someone realizes that they fall into about 12 categories, and that often within a category, they come "four at a time". The categories are:

> Runtime, AppDomain, Assembly, Module, Class, Function, Thread, Remoting, Transitions, Runtime Suspension, Garbage Collection, and Exceptions

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 a callback is received 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 CLR calls *Initialize* to setup the code profiler whenever a new CLR application is started. The call provides an *IUnknown* interface pointer that should be QI'd for an *ICorProfilerInfo* interface pointer. Note that in that callback the code profiler has to specify which events is interested in monitoring by calling *ICorProfilerInfo::SetEventMask()*.

```
HRESULT Initialize( IUnknown *pICorProfilerInfoUnk )
```

Parameter	Description
`[in] pICorProfilerInfoUnk`	A pointer to an `IUnknown` object within the CLR, which can be QueryInterface'd for an ICorProfilerInfo interface pointer. The profiler can call methods in this object to obtain more info about notifications

6.1.2 Shutdown

The CLR calls Shutdown to notify the code profiler that the application 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 behavior.

```
HRESULT Shutdown()
```

After Shutdown is fired from the EE, all the profiler notifications that were specified for the current profiler are internally turned off, except certain IMMUTABLE events (see section 9.1).

Note that Shutdown will only fire where the managed application that is being profiled was started running managed code (that's to say, the bottom frame on the process' stack is managed). If the application being profiled started life as unmanaged code, which later 'jumped into' managed code (thereby creating an instance of the CLR), then Shutdown will **not** fire. In these cases, the profiler should include a DllMain routine in their library that uses Win32's DLL_PROCESS_DETACH call to free any resources and perform tidy-up processing of its data (flush traces to disk, etc) [the issue is that by the time CLR is executing, and in a position to call Shutdown, the profiler DLL has already been unloaded by Win32].

Even more extreme is the case where the process being profiled is 'killed' by a call to Win32's *TerminateProcess*. In this case, neither Shutdown nor DllMain fires, so the profiler must cope as best it can with required tidy-up. These scenarios may arise under Win9x when an unhandled exception is being thrown.

Note also that during shutdown, the runtime is possible to violently kill certain managed threads (this happens only if they are "background" managed threads) that remain alive for an extended period of time during shutdown. As a result, all the notifications related to those threads will stop abruptly and the code profiler will have to simulate those events if it counts on receiving them.

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

Called when an AppDomain shutdown is starting. The AppDomain specified by the *appDomainId* is still valid to use.

```
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. The app*DomainId* cannot be used anymore for any queries to 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

6.3.1 AssemblyLoadedStarted

It would be expected that the CLR would notify an assembly load, followed by one or more module loads for that assembly. However, what actually happens is that Runtime notifies the profiler of a module load, then the load of its containing assembly; after that the profiler is possible top receive 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, the user 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 the profiler by firing a *ModuleAttachedToAssembly* callback.

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. The *assemblyId* cannot be used 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] assemblyId`	ID for the assembly just unloaded
`[in] hrStatus`	Status for whether the assembly unload succeeded

Note that currently the profiling API does not provide any notification about shared assemblies.

6.4 Module

6.4.1 ModuleLoadStarted

The CLR 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 statistics 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. The *moduleId* cannot be used 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 ModuleAttachedToAssembly

The CLR calls *ModuleAttachedToAssembly* 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 *ModuleLoadFinished* 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.

```
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. The *classId* cannot be used 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 CLR 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. This event can be used 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. The *classId* cannot be used 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 JITCompliationStarted

The CLR 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] functioned	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 CLR 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 CLR calls *FunctionUnloadStarted* to notify the code profiler that a function is being unloaded. After returning from this call, the *functionId* is no longer valid. This method is not implemented in the current version of the Runtime.

```
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 pre-compiled function is starting. The value of *pbUseCachedFunction* will inform the 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 that the functionId is not valid for calls to any *ICorProfilerInfo* APIs until the profiler has received the corresponding *JITCompilationFinished* or *JITCompilationSearchFinished* with a result that indicates a successful search.

```
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 install-time code generated 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 install-time generated code. The *result* indicates 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 CLR calls *JITFunctionPitched* to notify the profiler that a JIT compiled 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-JIT compiled. The *functionId* is not valid until it is re-JIT compiled. When it is re-JIT compiled, 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 JIT 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 the user is trying to get an accurate call-graph, they should set FALSE. Be aware that setting FALSE will affect performance, since inlining typically increases speed and reduces separate JIT 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

It is possible to disable globally JIT-lining in the Initialize callback by setting the bit COR_PRF_DISABLE_INLINING.

6.7 Thread

Unlike other categories, the runtime does **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 CLR 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] threaded	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] threaded	ID for the thread just destroyed

6.7.3 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: Eeach of the following pairs of callbacks will occur on the same thread

RemotingClientInvocationStarted and *RemotingClientSendingMessage*

RemotingClientReceivingReply and *RemotingClientInvocationFinished*

RemotingServerInvocationReturned and *RemotingServerSendingReply*

RemotingServerInvocationStarted and *RemotingServerReceivingMessage*

Note that you cannot make calls to any method from the in-process debugging API from any of the Remoting callbacks described in this section.

There are a few issues with the remoting callbacks that should be outlined. First, remoting function execution is not reflected by the profiler API, so the notifications for functions that are called from the client and executed to the server are not properly received. The actual invocation happens via a proxy object. That creates the illusion to the profiler that certain functions get jit-compiled but they never get used. Second, the profiler does not receive accurate notifications for asynchronous remoting events. Both issues will be addressed in the future version.

6.8.1 RemotingClientInvocationStarted

The CLR 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 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 CLR 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 any arbitrary thread may service the request.

```
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 CLR 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 CLR 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 CLR 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 CLR calls *UnmanagedToManagedTransition* to notify the code profiler that a transition from unmanaged code to managed code has occurred. The function *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.

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

On the other hand, 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.

When 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 CLR calls *ManagedToUnmanagedTransition* to notify the code profiler that a transition from managed code to unmanaged code has occurred. The *functionId* corresponds always to 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 a managed function called by an unmanaged one.

If the reason is COR_PRF_TRANSITION_CALL, then the *functionId* is that of the unmanaged function, and will never have been JIT compiled. Unmanaged functions

still have some basic information associated with them, such as a name, and some metadata.

When 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 COMClassicVTableCreated

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

```
HRESULT COMClassicVTableCreated( ClassID wrappedClassId,
                                 REFGUID implementedIID,
                                 void *pVTable,
                                 ULONG cSlots )
```

Parameter	Description
[in] wrappedClassId	ID of the managed class the VTable gives access to
[in] implementedIID	IID of the interface this VTable provides access to
[in] pVTable	pointer to the VTable
[in] cSlots	number of slots in the VTable

6.9.4 COMClassicVTableDestroyed

Notify that the Runtime has destroyed a CCW (see *COMClassicVTableCreated*, above). This callback is likely never to occur. The Reason is that the CLR is turning off all the notifications except the immutable ones when the shutdown callback occurs and the VTables are released just before shutting down the CLR itself. For that reason the shutdown callback should be considered as a “virtual” *COMClassicVTableDestroyed* callback.

```
HRESULT COMClassicVTableDestroyed( ClassID wrappedClassId,
                                   REFGUID implementedIID,
                                   void *pVTable )
```

Parameter	Description
`[in] wrappedClassId`	ID of the managed class the VTable gave access to
`[in] implementedIID`	IID of the interface this VTable provided access to
`[in] pVTable`	pointer to the VTable's interface

6.10 Runtime Suspension

6.10.1 RuntimeSuspendStarted

The CLR 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. Any code pitching callbacks will occur between the *RuntimeSuspendFinished* and *RuntimeResumeStarted* events. (Note: code pitching is not implemented with the V1 JIT compiler)
- 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_FOR_GC_PREP: the Runtime is preparing for a GC.
- COR_PRF_SUSPEND_FOR_INPROC_DEBUGGER: the runtime is suspending for in-process debugging.
- 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 CLR calls *RuuntimeSuspendFinished* 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 *RuntimeSuspendStarted*.

Note: It is guaranteed that this event will occur on the same *threadId* as *RuntimeSuspendStarted* occurred on.

HRESULT RuntimeSuspendFinished()

6.10.3 RuntimeSuspendAborted

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

Note: If a thread successfully suspended the runtime, then it is *NOT* guaranteed that the call to *RuntimeResumeStarted* will be called on the same *threadId*. In this case, it is also not guaranteed that the thread calling *RuntimeResumeStarted* will have a non-zero *threadId*. However, if a thread unsuccessfully suspends the runtime (i.e., there was a conflict with another suspension), then it ***is*** guaranteed that the calls to *RuntimeSuspspendStarted*, *RuntimeSuspendAborted* and *RuntimeResumeStarted* will be called from the same *threadId*.

HRESULT RuntimeResumeStarted()

6.10.5 RuntimeResumeFinished

The CLR calls *RuntimeResumeFinished* to notify the code profiler that the Runtime has finished resuming all its threads and is now back in normal operation.

Note: It is ***not*** guaranteed that this event will occur on the same *threadId* as the *RuntimeSuspendStarted* occurred on, but is guaranteed to occur on the same *threadId* as the *RuntimeResumeStarted* occurred on.

HRESULT RuntimeResumeFinished()

6.10.6 RuntimeThreadSuspended

The CLR 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] threaded	The ID of the thread that was suspended.

6.10.7 ThreadResumed

The CLR 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] threaded	The ID of the thread that was resumed.

Remarks

It is possible for the profiler to receive one or more *RuntimeSuspendiStarted* callbacks originating from different threads. Only one suspension request will succeed and eventually cause the suspension. All the other suspension events will be aborted. Every thread that gets suspended will fire a *ThreadSuspended* callback and when it gets resumed it will fire a *ThreadResumed* callback. Note, however, that the *ThreadSuspended*/*Resumed* events are not guaranteed to be called by the threads that are actually being suspended. If a thread during a suspension attempt is running native code, then we will not receive any events for that thread. Also if we have two subsequent suspensions, it is possible for certain threads not to be resumed simply because they did not have enough time to do so between the consecutive suspensions.

6.11 Garbage Collection

When the user specifies the COR_PRF_MONITOR_GC flag, all the GC events will be triggered in the profiler except the *ICorProfilerCallback::ObjectAllocated* events. They are explicitly controlled by another flag (see next section), for performance reasons. Note that when the COR_PRF_MONITOR_GC is enabled, the Concurrent Garbage Collection is turned off.

The code profiler identifies that a GC is taking place by monitoring the suspend/resume related callbacks when the suspension reason is COR_PRF_SUSPEND_FOR_GC. During shutdown though, the runtime also gets suspended and it is possible for one or more GC's to take place but the code profiler will not receive any notifications for them, since the runtime is already in a suspended state. Therefore, the code profiler is possible to continue receiving object

allocations between the GC events from the runtime while everything seesm to be stopped!. Detecting when a GC has completed in those circumstances is not trivial. The code profiler has to detect the very first *ObjectAllocated* callback that took place AFTER an *ObjectReferences* or *RootReferences* callbacks.

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. For these events to fire, the profiler must set the COR_PRF_MONITOR_OBJECT_ALLOCATED bit in the notifications mask.

It is possible to receive a *classId* that corresponds to a regular class but it has not been loaded yet. The profiler will receive a class load callback for that class immediately after the object creation callback.

```
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

ObjectsAllocatedByClass 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. The callback 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). This callback provides information about Generation 0 allocated classes. Notice that this callback is not reporting objects that are allocated in the large heap.

```
HRESULT ObjectsAllocatedByClass( ULONG cClassCount,
                                 ClassID classIds[],
                                 LONG 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.

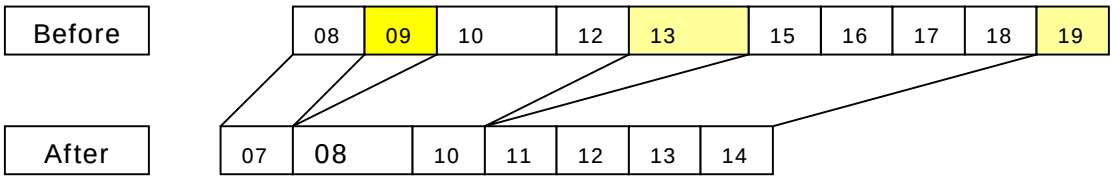
MovedReferences may be invoked multiple times during a GC if the list of moved references exceeds the size of the profiling services' internal buffer.

```
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. Note that *MovedReferencesCallback* is reporting the new layout of the object BEFORE they actually get relocated in the heap. So the old *ObjectIDs* are still valid for calls to the *ICorProfilerInfo* interface (and the new *ObjectIDs* are not).

6.11.4 ObjectReferences

The CLR 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 value]	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 CLR will ensure that each object reference is reported only once by ObjectReferences.

6.11.5 RootReferences

The CLR 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.

It is possible to get NULL *ObjectIDs* in the *RootReferences* callback. For example, all object references declared on the stack (such as "Object A = NULL;") are treated as roots by the GC, and will always be reported.

When a GC operation completes, normally we should expect for every surviving object to be either a root reference or have a parent that is a root reference. At times it is possible to have some objects that do not belong to any of the aforementioned categories. Those objects are either allocated internally by the runtime or are weak references to delegates. Unfortunately, the profiling API currently does not let the user identify those objects and this will be addressed in a future version.

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. Note that the Exception notifications are not providing any *threadID's* but the user can always call *ICorProfilerInfo::GetCurrentThreadID* to discover which managed thread throws the exception.

Remarks:

In Interop scenarios, the profiler will get an *ExceptionThrown* callback every time exception unwinding crosses the unmanaged–to-managed boundary. The profiler will get a full search and unwind cycle for every managed chain and the subsequent *ExcptionThrown* callbacks will have the same *objectId*. In this way, the profiler can identify that this is not a new exception but the old one that is getting propagated to a new managed chain.

Figure 2 displays how the code profiler receives the various callbacks, when monitoring exception and CLR exception events. The arrow indicates the initial state of the program (no exception), and the states that are marked with lavender color indicate the states that the profiler may end up after receiving all the events that are related to specific exception. If the final state is NORMAL then we either found a managed catcher for the exception (transition from HANDLED state via *ExceptionCatcherEnter* callback) or the CLR handled the exception internally (transition from UNWIND PHASE state via *ExceptionCLRCatcherFound*). If the final state is UNWIND PHASE, then the exception is unhandled. Note that if an exception is thrown in a managed layer, crosses to the unamanged world and enters back into the "managed world", the profiler will receive and *ExceptionThrown* every time the exception handling mechanism enters the managed world but the exception object will be the same. To illustrate this assume that you have the scenario shown in Figure 1:

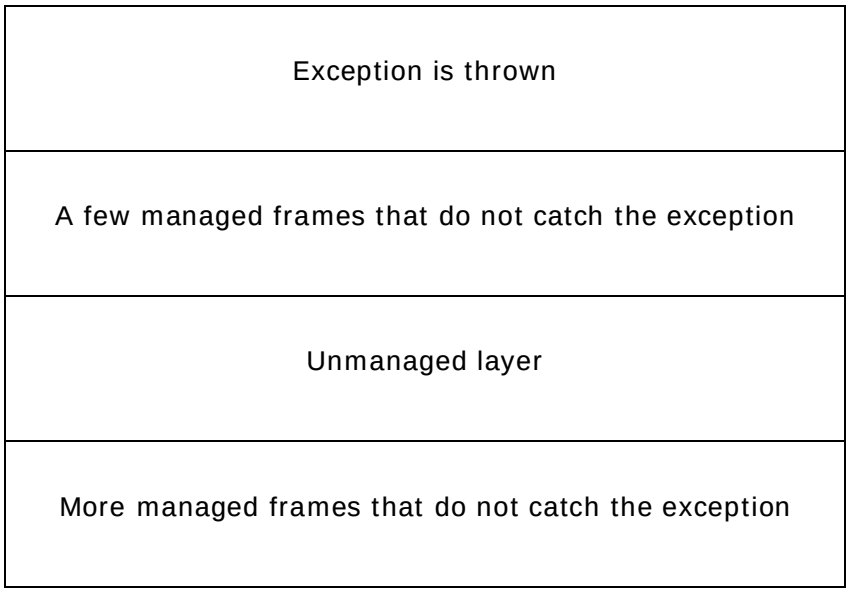


Figure 1 – Exception handling through interop layers

In that case the sequence of callbacks that will be received looks as follows:

ExceptionThrown with Object: 0x00bb60e4 **// exception is originally thrown**
ExceptionSearchFunctionEnter for Function: 0x03a019d0 **// search phase starts here**
ExceptionSearchFunctionLeave
ExceptionSearchFunctionEnter for Function: 0x03a019a8
ExceptionSearchFunctionLeave
ExceptionSearchFunctionEnter for Function: 0x03a01980
ExceptionSearchFunctionLeave
ExceptionCLRCatcherFound **// exception enters the unmanaged world and get handled by CLR**
ExceptionUnwindFunctionEnter for Function: 0x03a019d0 **// unwind phase starts**
ExceptionUnwindFunctionLeave
ExceptionUnwindFunctionEnter for Function: 0x03a019a8
ExceptionUnwindFunctionLeave
ExceptionUnwindFunctionEnter for Function: 0x03a01980
ExceptionUnwindFunctionLeave
ExceptionCLRCatcherExecute **// internal catcher is executed**
ExceptionThrown with Object: 0x00bb60e4 **// exception moves into the managed world with the same ID**
ExceptionSearchFunctionEnter for Function: 0x003750d8
ExceptionSearchCatcherFound for Function: 0x003750d8
ExceptionSearchFunctionLeave
ExceptionUnwindFunctionEnter for Function: 0x003750d8
ExceptionCatcherEnter for Exception Object 0x00bb60e4 in Function 0x003750d8
ExceptionCatcherLeave

Note that if the code profiler is not interpreting properly the *ExceptionThrown* callbacks in the above scenario it may falsely identify 2 exception but in fact it is the same exception traveling through managed and unmanaged layers. Also note that if a GC occurs an exception object may move and the code profiler has to be able to track all the exception objects that potentially moved in memory.

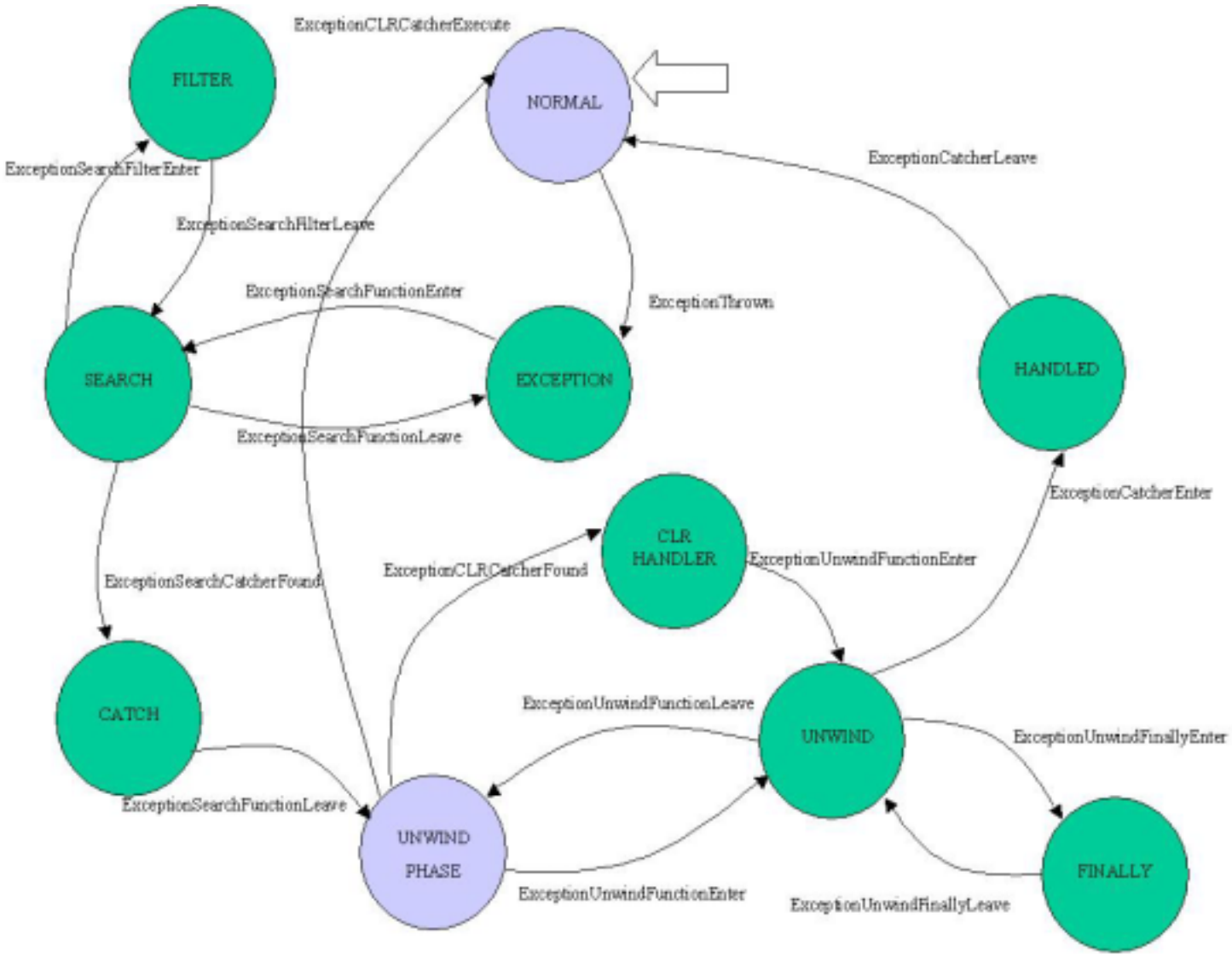


Figure 2 – Exception callback sequence

6.12.1 ExceptionThrown

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

```
HRESULT ExceptionSearchFunctionLeave()
```

6.12.4 ExceptionSearchFilterEnter

The CLR will call *ExceptionSearchFilterEnter* just before executing 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 CLR will call *ExceptionSearchFilterLeave* immediately after executing a user filter.

```
HRESULT ExceptionSearchFilterLeave()
```

6.12.6 ExceptionSearchCatcherFound

The CLR 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

Note: This callback is currently inactive.

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

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

6.12.8 ExceptionOSHandlerLeave

Note: This callback is currently inactive.

```
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 CLR 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 CLR calls *ExceptionUnwindFunctionLeave* to notify the profiler that the unwind phase of exception handling has left a function. The function instance and its stack data have 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 CLR 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 CLR 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 CLR 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 compiled 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 CLR 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()

6.12.15 ExceptionCLRCatcherFound

When the CLR itself is catching an exception, the following two callbacks are invoked to notify the profiler. Note that between the *ExceptionCLRCatcherFound/Execute* the CLR may also unwind managed frames. In that case *ExceptionUnwind* callbacks will be received in-between. For backward compatibility, the profiler must set the COR_PRF_MONITOR_CLR_EXCEPTIONS event flag to receive those events.

The CLR calls ExceptionCLRCatcherFound 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 ExceptionCLRCatcherFound()

6.12.16 ExceptionCLRCatcherExecute

The CLR calls ExceptionCatcherExecute 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 ExceptionCLRCatcherExecute()

7 ICorProfilerInfo

The CLR 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 MSIL for a function and request that be re-JIT compiled. *ICorProfilerInfo* interface as the "help desk" for profilers.

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

The CLR 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 *CoInitialize'*d) at any time.

The methods in *ICorProfilerInfo* return S_OK on success, or one of the error codes defined in CorError.h (CORPROF_E_*) on failure.

7.1 BeginInprocDebugging

The profiler MUST call this function before using the in-process debugging APIs. The parameter *fThisThreadOnly* indicates whether in-proc debugging will be used to trace the stack of the current managed thread only, or whether it might be used to trace the stack of any managed thread. The *pdwProfilerContext* argument is information that the profiler must save so that it may use it during *EndInprocDebugging*. The profiler must pass the value returned by this API to *EndInprocDebugging* in an unaltered state.

```
HRESULT BeginInprocDebugging( BOOL fThisThreadOnly,
                              DWORD *pdwProfilerContext )
```

Parameter	Description
[in] fThisThreadOnly	flag that indicates whether we wish to enable in-proc debugging for the current thread or for all the threads.
[out] pdwProfilerContext	Parameter where the context is saved; this context needs to be used later in EndInprocDebugging call

In-process debugging is not supported from all the callbacks and a call to BeginInprocDebugging will fail with CORPROF_E_INPROC_NOT_AVAILABLE if invoked from the wrong callback. For more details please review the in-process debugging section of the Debugging API specification. If we call multiple times *BeginInprocDebugging* the method will return CORPROF_E_INPROC_ALREADY_BEGUN.

7.2 EndInprocDebugging

The profiler MUST call this function when it is done using the in-process debugging APIs. Failing to do so will result in undefined behavior of the runtime. The *dwProfilerContext* is the value returned by *BeginInprocDebugging*.

HRESULT EndInprocDebugging(DWORD *pdwProfilerContext)

Parameter	Description
[in] pdwProfilerContext	Parameter where the context was saved during the BeginInprocDebugging function call

7.3 ForceGC

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

HRESULT ForceGC()

This method needs to be called from a thread that does not have any profiler callbacks on its stack. The most efficient way to implement it is to create a listener thread dedicated for GC operations and when a managed thread wishes to force a GC it simply creates an event and continues its execution. Never use *ICorProfilerInfo::ForceGC* before *ICorProfilerCallback::Initialize()* callback and/or after the ICorProfilerCallback::Shutdown callback are received. This can cause unspecified behavior of the runtime.

7.4 GetAppDomainInfo

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

```
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.
[out] pProcessId	The Win32 processId where the appdomain belongs to

7.5 GetAssemblyInfo

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

```
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.

7.6 GetClassFromObject

The code profiler calls *GetClassFromObject* to obtain the *classId* of an object given its *objectId*.

```
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.

7.7 GetClassFromToken

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

```
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.

7.8 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.

```
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.

Remarks

It is possible to receive different *classId* values for the same class. This can occur only for __COMObject classes, the reason being that Interop requires a different EE class for each Application Domain so the user can potentially create a different v-table for each one of them. The profiler currently does not provide a method to get the current Application Domain.

7.9 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.

```
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.

7.10 GetCurrentThreadID

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

```
HRESULT GetCurrentThreadID( ThreadID *pThreadId )
```

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

GetCurrentThreadID may return CORPROF_E_NOT_MANAGED_THREAD if the current thread is an internal EE thread, and the returned value of *pThreadId* will be NULL. This scenario can arise if for example someone is using *ICorProfilerInfo::GetCurrentThreadID* from any of the Suspend/Resume related callbacks in conjunction with ConcurrentGC.

7.11 GetEventMask

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

```
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 enumerator.

7.12 GetFunctionFromIP

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

```
HRESULT GetFunctionFromIP( LPCBYTE 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 CLR may choose to unload a function to recover memory. In such instances, the instruction pointer mapping becomes invalid. The CLR 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 MSIL offsets from the beginning of the function.

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

7.13 GetFunctionFromToken

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

```
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.

7.14 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*.

```
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.

7.15 GetHandleFromThread

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

```
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.

7.16 GetILFunctionBodyAllocator

MSIL 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. This method has to be called on a per module basis otherwise the code profiler can potentially create fragmentation of the available memory in the heap.

```
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.

7.17 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 MSIL 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.

```
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.

7.18 GetILToNativeMapping

The profiler calls *GetILToNativeMapping* when it wants to get a map from MSIL offsets to native offsets for this code. An array of COR_PROF_IL_TO_NATIVE_MAP structures will be returned, and some of the *ilOffsets* in this array may be the values specified in *CorProfILToNativeMappingTypes*. The rest of the values will be the actual MSIL offsets for that method. Note that if the JIT is not tracking debug information, the MSIL offsets will not be exact.

```
HRESULT GetILToNativeMapping( FunctionID functionId,
                              ULONG32 cMap,
                              ULONG32 *pcMap,
                              COR_DEBUG_IL_TO_NATIVE_MAP map[] )
```

Parameter	Description
[in] functionId	FoduleID of the given function.
[in] cMap	Indicates the max number of offsets we wish to read
[out] *pcMap	Indicates how many items were actually written to the map[]
[out] map	An array where the ILToNative structs will be placed

7.19 GetInprocInspectionInterface

The code profiler calls *GetInprocInspectionInterface* to get an interface to the in-process portion of the debug interface, which is useful for things like doing a stack trace. It is expected that the returned interface will be queried for *ICorDebug*. In order to use this callback, the profiler needs to call *BeginInprocDebugging* first requesting in-process support for all threads.

```
HRESULT GetInprocInspectionInterface( IUnknown **ppicd )
```

Parameter	Description
[out] ppicd	*ppicd will be filled in with a pointer to the interface, or NULL if the interface is unavailable.

7.20 GetInprocInspectionThisThread

The code profiler calls *GetInprocInspectionIThisThread* to get an interface to the in-process portion of the debug interface that is specific to the current thread. It's expected that the returned interface will be queried for *ICorDebugThread*, which can then be used to immediately do a stack trace. In order to use this callback, the profiler needs to call *BeginInprocDebugging* first requesting in-process support for all threads or for the current thread only.

```
HRESULT GetInprocInspectionIThisThread( IUnknown **ppicd )
```

Parameter	Description
[out] ppicd	*ppicd will be filled in with a pointer to the interface, or NULL if the interface is unavailable.

7.21 GetModuleInfo

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

```
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.

7.22 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 reada and 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.

```
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.

7.23 GetObjectSize

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

```
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.

7.24 GetThreadContext

The code profiler calls *GetThreadContext* to get the *contextId* currently associated with the calling Runtime thread. This method set *pContextId* to NULL if the calling thread is not a Runtime thread.

```
HRESULT GetThreadContext( ThreadID threadId,
                          ContextID *pContextId )
```

Parameter	Description
[in] threadId	The ThreadID of the thread the code profiler is interested in.
[out] pContextId	Pointer to the context structure we are interested in retrieving

7.25 GetThreadInfo

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

```
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.

7.26 GetTokenAndMetadataFromFunction

The code profiler calls this method for a given function, in order to retrieve the token value and an instance of the metadata interface, which can be used against this token.

```
HRESULT GetTokenAndMetaDataFromFunction( FunctionID functionId,
                                         REFIID riid,
                                         IUnknown **ppImport,
                                         mdToken *pToken )
```

Parameter	Description
[in] functionId	The functionID of the method we are interested in.
[in] riid	The REFIID of the metadata interface.
[out] ppImport	Pointer to the pointer to the returned metadata interface that maps to the given module.
[out] pToken	Pointer to the metadata token for the specific function

7.27 IsArrayClass

This method allows the profiler to get information about classes that are arrays and get reported to the profiler either from *ObjectAllocated* or from *ObjectAllocatedByClass* callbacks. In the above callbacks, the profiler is possible to receive *classId*'s, which were never encountered before. Those classes correspond to arrays for which we have allocated space on the heap and they can get garbage collected like any other object. If the *classId* does not correspond to an array the method returns S_FALSE.

```
HRESULT IsArrayClass( ClassID classId,
                      CorElementType *pBaseElemType,
                      ClassID *pBaseClassId,
                      ULONG *pcRank )
```

Parameter	Description
[in] classId	The ClassID of the class we are interested in.
[out] pBaseElemType	The type of the element of the array.
[out] pBaseClassID	In which class the array belongs to.
[out] pcRank	The rank of the array

7.28 SetEnterLeaveFunctionHooks

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

```
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.

7.29 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.

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.

7.30 SetFunctionIDMapper

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

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.

7.31 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 MSIL prior to the JIT. By this means, a tool can effectively replace a function at Runtime. Note that all the active instances of the function are not affected by the replacement and that this methods fails when used with methods that are pre-compiled.

HRESULT SetFunctionReJIT(FunctionID functionId)

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

7.32 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.

```
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 MSIL method header.
[in] cbNewMethod	Pointer to the size of the new MSIL method header.

Using *ICorProfilerInfo::GetILFunctionBody* and *ICorProfilerInfo::SetILFunctionBody* is not trivial. Assume that we have the trivial scenario where we simply want to copy over the body of the original function to a new one. The first thing to ensure is that we have allocated the memory for the new function body using *IMethodMalloc::Alloc*. Next we need to look for variable sized data at the end of the MSIL function body and make sure that we are copying them properly. An example of a function that has additional data after the main function body would be the case where we have a try-catch block. The profiler has to respect the calling convention if a call is going to be attempted in the injected MSIL code and also make sure that the stack will grow properly in case there are any operations included.

7.33 SetILInstrumentedCodeMap

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

In each COR_IL_MAP entry in the map each oldOffset refers to the MSIL offset within the original unmodified MSIL code. newOffset refers to the corresponding MSIL offset within the new, instrumented code.

A COR_IL_MAP entry need only be created for each offset in the original code where instrumentation code has been inserted. Additional intermediate mappings can be supplied but are unnecessary. Memory for the rgILMapEntries array should be allocated using the CoTaskMemAlloc() COM API call. The code profiler should not attempt to free this memory.

Note that currently *SetILInstrumentedCodeMap* fails to set the map properly for methods that are pre-compiled. That issue will be addressed in future versions.

```
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 MSIL map.
`[in] cILMapEntries`	Number of entries in the rgILMapEntries array.
`[in] rgILMapEntries`	An array of entries that specify how the old MSIL offsets map to the new MSIL offsets.

8 Memory Allocation Interface (ImethodMalloc)

This is the interface to a very simple allocator that only allows allocating memory. The user cannot free, the profiling API is freeing the memory for the user transparently. This interface should be used in conjunction with SetILMethodBody.

8.1 Alloc

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

void Alloc(ULONG cb)

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

9 Profiling Enumerations

9.1 COR_PRF_MONITOR

The following table lists the values that can be set in the pdwRequestedEvents argument *for ICorProfilerCallBack::Initialize*. In that way the user can specify which categories of notifications wishes to receive during execution of the target App. Each value corresponds to a bit in the DWORD argument. It is possible to OR bits together to customize the received notifications.

These constants are also used to set the dwEvents argument for *ICorProfilerInfo::SetEventMask*; and to interpret the result returned in the pdwEvents argument from *ICorProfilerInfo::GetEventMask.*

Note that some of these bits are "immutable" in a sense that the user is allowed to set them once at Initialize time. It is not allowed to modify them with a call to SetEventMask (an attempt to do so returns a failure HRESULT) during the program's execution.

For brevity, we omit the COR_PRF_MONITOR_ prefix. Some flags are use to enable or disable a certain functionality in the CLR. For those assume the prefix COR_PRF_. So, NONE is shorthand for COR_PRF_MONITOR_NONE while ENABLE_JIT_MAPS is a short for COR_PRF_ENABLE_JIT_MAPS:

Event Category	Description
NONE	Send no notifications
FUNCTION_UNLOADS	Notify each function unload
CLASS_LOADS	Notify each class load or unload
MODULE_LOADS	Notify each module loaded or unload
ASSEMBLY_LOADS	Notify each assembly load or unload
APPDOMAIN_LOADS	Notify each AppDomain load or unload
JIT_COMPILATION	Notify each function just before being JIT-compiled and just after JIT-compilation finishing
EXCEPTIONS	Notify occurrence of each exception
GC	Notify when a garbage collection is about to occur
OBJECT_ALLOCATED	Notify each object being allocated on the GC heap
THREADS	Notify each thread creation or destruction
REMOTING	Notify each context crossing
CODE_TRANSITIONS	Notify each transition from managed to unmanaged code or vice versa
ENTERLEAVE	Call function entry hook and exit hook
CCW	Notify CCW events (COM-callable-wrapper)
REMOTING_COOKIE	Generate cookies so the profiler can pair remoting callbacks

	callbacks
REMOTING_ASYNC	Notify async remoting events
SUSPENDS	Notify when CLR gets suspended
CACHE_SEARCHES	Send function search notifications for install-time code generated functions
CLR_EXCEPTIONS	Notifies the profiler for CLR exceptions handling
ENABLE_REJIT	Send function search notifications for install-time code generated functions
ENABLE_IN_PROC_DEBUGGING	Enables in-process debugging, it is necessary in order to be able to use the in-process debugging API
ENABLE_JIT_MAPS	Enables JIT-map tracking information
DISABLE_INLINING	Disable method inlining (process-wide). If left enabled, then inlining events are notified via the JITInlining callback
DISABLE_OPTIMIZATIONS	Forces the JIT to disable optimizations
ENABLE_OBJECT_ALLOCATED	Allows to track object allocations
ALL	All of the above

The following bits are defined as immutable:

```
IMMUTABLE =  COR_PRF_MONITOR_CODE_TRANSITIONS
            |COR_PRF_MONITOR_REMOTING
            |COR_PRF_MONITOR_REMOTING_COOKIE
            |COR_PRF_MONITOR_REMOTING_ASYNC
            |COR_PRF_MONITOR_GC
            |COR_PRF_ENABLE_REJIT
            |COR_PRF_ENABLE_INPROC_DEBUGGING
            |COR_PRF_ENABLE_JIT_MAPS
            |COR_PRF_DISABLE_OPTIMIZATIONS
            |COR_PRF_DISABLE_INLINING
            |COR_PRF_ENABLE_OBJECT_ALLOCATED
```

Notice that for Object allocations we have two different flags. The first is enabling object allocations and the second is tracking object allocations. The profiler has to indicate to the EE that is it going to use object allocations at some point. This enables the tracking of object allocations and therefore at any point the profiler can start or stop monitoring objects by calling *SetEventMask()*.

9.2 COR_PRF_MISC

The *ICorProfilerInfo::GetFunctionInfo* returns the *classId* of the specified function. However, it's not always possible to provide this info. The constants in the following table are reserved values of *ClassId* returned in those cases. For brevity, we omit the PROFILER_ prefix. So, PARENT_UNKNOWN is shorthand for PROFILER_PARENT_UNKNOWN:

Enumeration	Description
PARENT_UNKNOWN	Owner class unknown
GLOBAL_CLASS	Function is global (within its defining module)
GLOBAL_MODULE	This is used to cover the case where we have a global method that does not belong to any class or module

9.3 COR_PRF_JIT_CACHE

The following table lists values supplied in the *functionId* argument of the *ICorProfilerCallback::JITCachedFunctionSearchFinished* callback. In the following table, for brevity, we omit the COR_PRF_JIT_CACHE_ prefix. So, FUNCTION_FOUND is shorthand for COR_PRF_JIT_CACHE_FUNCTION_FOUND:

Enumeration	Description
FUNCTION_FOUND	Found a instance of the function (whose FunctionID was notified in the immediately-preceding JITCachedFunctionSearchStarted callback) in the JIT cache
FUNCTION_NOT_FOUND	Said function not found in JIT cache

9.4 COR_PRF_SUSPEND_REASON

The following table lists the values supplied in the *ICorProfilerCallback::RuntimeSuspendStarted* notification. For brevity, we omit the COR_PRF_SUSPEND_ prefix. So, OTHER is shorthand for COR_PRF_SUSPEND_OTHER:

Enumeration	Description
OTHER	Reason other than those below
FOR_GC	For a garbage collection. Any GC-related callbacks will occur before the following RuntimeResumeStarted event
FOR_APPDOMAIN_SHUTDOWN	To shut down an AppDomain
FOR_CODE_PITCHING	For code pitching (not implemented in V1)
FOR_SHUTDOWN	For shutdown of the CLR itself
FOR_INPROC_DEBUGGER	For in-process debugging operations
FOR_GC_PREP	For GC preparation

9.5 COR_PRF_TRANSITION_REASON

The following table lists the values supplied in the **reason** parameter for the two notifications: *UnmanagedToManagedTransition* and *ManagedToUnmanagedTransition*

(in the *ICorProfilerCallback* API). For brevity, we omit the COR_PRF_TRANSITION_REASON_ prefix. So, CALL is shorthand for COR_PRF_TRANSITION_CALL:

Constant	Description
CALL	Transition was a call (into managed if UnmanagedToManagedTransition; into unmanaged if ManagedToUnmanagedTransition)
RETURN	Transition was a return of control from the previous, matching transition

9.6 CorDebugIlToNativeMappingTypes

This enumerator contains values that could be returned from *GetILtoNativeMap* method for certain areas of the code that the native instructions correspond to special regions of the code.

Constant	Description
NO_MAPPING	Indicates that we cannot map this area properly, this value is returned when we have optimizations enabled or when the profiler has not requested from the JIT to track MSIL maps
PROLOG	Indicates that we are in the prolog
EPILOG	Indicates that we are in the epilog

9.7 COR_PRF_JIT_MAP

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

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

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 Profiling Type Definitions

10.1 COR_IL_MAP

The COR_IL_MjAP type is used to describe how an MSIL offset in an old function body maps to the MSIL 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.

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

Member	Description
oldOffset	MSIL offset in the old function body.
newOffset	MSIL offset in the new function body.
fAccurate	Shows if the mapping is accurate

10.2 COR_DEBUG_IL_TO_NATIVE_MAP

This structure contains information for every MSIL offset, what is the start and end offset of the native code that it maps to.

```
typedef struct COR_DEBUG_IL_TO_NATIVE_MAP
{
   ULONG32 ilOffset;
   ULONG32 nativeStartOffset;
   ULONG32 nativeEndOffset;
} COR_DEBUG_IL_TO_NATIVE_MAP;
```

Member	Description
ilOffset	MSIL offset that is going to be mapped
nativeStartOffset	Native offset where the mapping of the ilOffset starts.
nativeEndOffset	Native offset where the mapping of the ilOffset ends.

Notice that *nativeStartOffset* and *nativeEndOffset* can have values defined in *CorDebugIlToNativeMappingTypes* if the MSIL instruction corresponds to the prolog or the epilog or the JIT's mapping information is not enabled.

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. The mapper can be set only once and it is recommended to do so in the *Initialize* callback.

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.

typedef void __stdcall FunctionEnter(FunctionID functioned)

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.

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.

typedef void __stdcall FunctionTailcall(FunctionID functioned)

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

For more details on how to use and implement the above callbacks, please refer to the Profiler Sample that is included in the “Tool Developers Guide\Samples” directory.

11 Security Issues in Profiling

The CLR Profiling Services security is based on the security that is applied to the user that instantiates profiling by the operating system. The CLR security for the execution of unmanaged code is minimal and therefore administrators should be careful when granting execution rights to users.

Consider for example a couple of scenarios. 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. The new methods could be managed and/or unmanaged. These modifications violate the enforcement of security imposed on the code in various ways that cannot be controlled by the Runtime. This is because the CLR does not provide a complex set of security checks for native code. The security that is applied to the profiler is basically the same as the security that is applied to the user that instantiates profiling by the operation system. Therefore, if users have enough execution privileges they can instantiate a profiler and get full trust by the CLR.

12 Combining Managed and Unmanaged Code in a Code Profiler

A close review of the available CLR Profiling API's creates the impression that someone could write a profiler that can have managed and unmanaged components that call to each other through COM Interop or ndirect calls.

Although this is possible from a design perspective, the CLR Profiling API does not support it. A CLR profiler is supposed to be purely unmanaged. Attempts to combine managed and unmanaged code from a CLR profiler can cause crashes, hangs and deadlocks. The danger is clear since the managed parts of the profiler will "fire" events back to its unmanaged component, which subsequently would call into the managed part of the profiler etc. The danger at this point is clear.

The only location that a CLR profiler could invoke managed code safely would be through replacement of the MSIL body of a method. The profiler before the JIT-compilation of a function is completed inserts managed calls in the MSIL body of a method and then lets the JIT compile it. This technique can successfully be used for selective instrumentation of managed code, or it can be used to gather statistics and times about the JIT.

Alternatively a code profiler could insert native "hooks" in the MSIL body of every managed function that call into unmanaged code. That technique could be used for instrumentation and coverage. For example a code profiler could be inserting instrumentation hooks after every MSIL block to ensure that the block has been executed. The modification of the MSIL body of a method is very delicate operation and there are many factors that should be taken into consideration.

13 Profiling an application with precompiled components

In order to improve the performance of the CLR, it is possible for an application to consist of precompiled components. If a module is likely to be used very frequently by one or more applications it is recommended to pre-compile that module for improved speed. For example notice that mscorlib.dll that contains the .net framework class library for the .net framework is being precompiled during the installation of the .NET SDK. The existence of precompiled modules creates a new picture for the CLR profiling services.

A profiler has to be able to monitor which functions get compiled in the traditional way and which functions are used in their precompiled form. This can be achieved by monitoring:

JITCompilationStarted, *JITCompilationFinished*, *JITCachedFunctionSearchStarted*, *JITCachedFunctionSearchFinished*

Additionally, the enter-leave and interop events should behave in a consistent way in light of precompiled modules for both precompiled and normally compiled methods. To achieve that the precompiled image MUST contain profiling information, otherwise the CLR loader will not use the existing pre-compiled images and it will default to the regular JIT.

The tool that is used to produce precompiled images is called *ngen.exe* and it is available in the .NET Framework SDK. To successfully profile a precompiled .NET component, you will need to use *ngen.exe* with the appropriate settings as shown in the table below:

CLR Profiler	Parameters to use while pre-compiling a module
Monitors enter-leaves and/or transitions	ngen /prof <modules you wish to precompile>
Does not monitor enter-leaves nor transitions	ngen <modules you wish to precompile>

Note that using the in-process Debugging API with pre-compiled images that contain profiling information is not supported currently.

14 Profiling Unmanaged 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 or native code.

These methods are available through the in-process subset of the CLR debugging APIs. These are defined in the CorDebug.IDL and explained in DebugRef.doc, please refer to both for more details.