NGWS runtime

Debug Reference

This is preliminary documentation and subject to change

Last updated: 14 June 2000

Table of Contents

1 Overview of Debug Interfaces....5
2 Security Considerations....6
3 Debug Interface Scenarios....7
3.1 Debugging a Runtime Process....7
3.2 Controlling the Program....7
3.2.1 Setting a Breakpoint in Managed Code....8
3.2.2 Stepping through Managed and Unmanaged Code....8
3.2.3 Handling Exceptions....9
3.3 Examining the Program....10
3.3.1 Accessing Call Stacks....10
3.3.2 Evaluating Expressions....10
3.4 Injecting Code Dynamically....11
3.4.1 Composing the Function....11
3.4.2 Injecting the Code....12
3.4.3 Executing Injected Code....12
3.4.4 Miscellaneous Issues....13
3.4.5 Getting the Signature From the Metadata....13
4 In-Process Debugging APIs....15
4.1 GetInprocInspectionInterface....15
4.2 GetInprocInspectionIThisThread....15
5 Debug Interfaces....16
5.1 ICorDebug : IUnknown....19
5.2 ICorDebugAppDomain : ICorDebugController....22
5.3 ICorDebugAppDomainEnum : ICorDebugEnum....25
5.4 ICorDebugArrayValue : ICorDebugHeapValue....25
5.5 ICorDebugAssembly : IUnknown....27
5.6 ICorDebugAssemblyEnum : ICorDebugEnum....29
5.7 ICorDebugBoxValue : ICorDebugHeapValue....29
5.8 ICorDebugBreakpoint : IUnknown....30
5.9 ICorDebugBreakpointEnum : ICorDebugEnum....30
5.10 ICorDebugChain : IUnknown....31
5.11 ICorDebugChainEnum : ICorDebugEnum....34
5.12 ICorDebugClass : IUnknown....35
5.13 ICorDebugCode : IUnknown....35

5.14 ICorDebugContext : ICorDebugObjectValue....................................38
5.15 ICorDebugController : IUnknown....................................38
5.16 ICorDebugEditAndContinueSnapshot : IUnknown....................................42
5.17 ICorDebugEnum : IUnknown....................................45
5.18 ICorDebugErrorInfoEnum : ICorDebugEnum....................................46
5.19 ICorDebugEval : IUnknown....................................47
5.20 ICorDebugFrame : IUnknown....................................50
5.21 ICorDebugFrameEnum : ICorDebugEnum....................................53
5.22 ICorDebugFunction : IUnknown....................................53
5.23 ICorDebugFunctionBreakpoint : ICorDebugBreakpoint....................................55
5.24 ICorDebugGenericValue : ICorDebugValue....................................56
5.25 ICorDebugHeapValue : ICorDebugValue....................................56
5.26 ICorDebugILFrame : ICorDebugFrame....................................57
5.27 ICorDebugManagedCallback : IUnknown....................................60
5.28 ICorDebugModule : IUnknown....................................69
5.29 ICorDebugModuleBreakpoint : ICorDebugBreakpoint....................................73
5.30 ICorDebugModuleEnum : ICorDebugEnum....................................74
5.31 ICorDebugNativeFrame : ICorDebugFrame....................................74
5.32 ICorDebugObjectValue : ICorDebugValue....................................78
5.33 ICorDebugObjectEnum : ICorDebugEnum....................................80
5.34 ICorDebugProcess : ICorDebugController....................................80
5.35 ICorDebugProcessEnum : ICorDebugEnum....................................85
5.36 ICorDebugRegisterSet : IUnknown....................................86
5.37 ICorDebugReferenceValue : ICorDebugValue....................................88
5.38 ICorDebugStepper : IUnknown....................................89
5.39 ICorDebugStepperEnum : ICorDebugEnum....................................92
5.40 ICorDebugStringValue : ICorDebugHeapValue....................................93
5.41 ICorDebugThread : IUnknown....................................93
5.42 ICorDebugThreadEnum : ICorDebugEnum....................................98
5.43 ICorDebugUnmanagedCallback : IUnknown....................................98
5.44 ICorDebugValue : IUnknown....................................99
5.45 ICorDebugValueBreakpoint : ICorDebugBreakpoint....................................101
5.46 ICorDebugValueEnum : ICorDebugEnum....................................101
6 Debug Type Definitions....................................102
6.1 COR_DEBUG_STEP_RANGE....................................102
6.2 COR_IL_MAP....................................102

6.3 CorDebugChainReason....102
6.4 CorDebugCreateProcessFlags....103
6.5 CorDebugIlToNativeMappingTypes....103
6.6 COR_DEBUG_IL_TO_NATIVE_MAP....103
6.7 CorDebugIntercept....103
6.8 CorDebugMappingResult....104
6.9 CorDebugRegister....104
6.10 CorDebugStepReason....105
6.11 CorDebugThreadState....106
6.12 CorDebugUnmappedStop....106
6.13 CorDebugUserState....107
6.14 LoggingLevelEnum....107
6.15 LogSwitchCallReason....107
7 HRESULT DEFINITIONS....109

1 Overview of Debug Interfaces

The NGWS runtime Debugging Services API enables tools vendors to write debuggers used to debug applications that run in the NGWS runtime environment. The code to be debugged can be any type of code that is supported by the runtime.

The debug interfaces consist of a collection of Component Object Model (COM) objects and interfaces implemented by the runtime and a collection of COM callback interfaces that must be implemented by the debugger. The debugger application is usually written in a language that can access these interfaces and objects for communicating with the runtime and controlling the runtime execution environment. Because these debug interfaces are all based on COM, debugger programs can be easily extended to allow remote debugging using Distributed COM (DCOM).

The debug interfaces can be organized into the functional categories shown in the following table.

Category	**Purpose**
Registration	Interfaces called by the debugger to register with the runtime and request to be notified when specific events occur
Notification	Callback interfaces that must be implemented by the debugger through which the runtime notifies the debugger of various events and returns requested information.
Breakpoint	Interfaces called by the debugger to retrieve information about breakpoints.
Execution	Interfaces called by the debugger to control execution of debuggee and access call stacks.
Information	Interfaces called by the debugger to obtain information about debuggee.
Enumeration	Interfaces called by the debugger to enumerate objects.
Modification	Interfaces called by the debugger to modify the code that is being debugged.

This specification identifies several scenarios in which the interfaces of the Debug API will be used, and explains which interfaces will be used for each scenario. In addition, it briefly describes each debugging interface by category and lists the supported methods.

This specification is intended to be used in conjunction with the Debug Architecture specification, which provides general information about why Debugging Services are needed and what features are supported.

The interfaces in this document are organized alphabetically.

2 Security Considerations

This section describes a few debugging scenarios in the context of runtime security.

Attaching to a Process. Under Windows NT/2000, the debuggee process must be created with a security descriptor that grants the debugger full access. The debugging process must have the SE_DEBUG_NAME privilege granted and enabled to debug any process. By default, a Windows NT/2000 administrator is granted this privilege. Windows 95/98 and Windows CE are less secure operating systems and do not impose special requirements for attaching to a process.

Debugger Injects a Delta PE During Edit and Continue. An assembly is verified when it is loaded. Subsequently, the debugger modifies some code and sends a delta PE into the debuggee process. The delta PE is not verified. The updated methods are verified after they are compiled by the JIT compiler.

Metadata of Signed Assembly Modified. A signed assembly is loaded. A debugger modifies the running code by changing the metadata associated with the debuggee using Edit and Continue. The operation changes the hash used to compute the assembly’s signature. The operation does not result in additional security checks. The integrity of the assembly is checked and the correct permissions granted only when the assembly is loaded. Permissions that were assigned by the runtime will continue to be in force.

3 Debug Interface Scenarios

Tools vendors implementing debuggers for tools that support the runtime will use the interfaces of the Debug API to handle many debugging scenarios. The following basic debugging scenarios are described in detail in the sections that follow:

- Debugging a process in the runtime environment
- Controlling the program during debugging.
- Examining the program during debugging.

3.1 Debugging a Runtime Process

The following is a step-by-step description of how a runtime process is debugged:

- **The debugger creates an instance of *ICorDebug*.** The debugger invokes *CoCreateInstance* using the CLSID CLSID_CorDebug to obtain an instance of *ICorDebug*.
- **The debugger initializes the debugging API by calling Initialize().**
- **The debugger registers a managed event handler.** The debugger invokes *SetManagedHandler* on *ICorDebug* to register an instance of *ICorDebugManagedCallback* as the callback for receiving notification and information about events in managed code.
- **The debugger optionally registers an unmanaged event handler.** If the debugger wants to debug unmanaged code, it invokes *SetUnmanagedHandler* on *ICorDebug* to register an instance of *ICorDebugUnmanagedCallback* as the callback for receiving notification and information about events in unmanaged code.
- **The debugger creates the debuggee process.** The debugger calls *CreateProcess* on *ICorDebug* to create a process.
- **The Debugger Interface notifies the debugger about the new debuggee process.** The Debugger Interface calls callbacks on *ICorDebugManagedCallback* starting with the callbacks *CreateProcess*. This may be followed by calls to the callbacks *LoadModule*, *LoadClass*, *CreateThread*, etc.
- **The debugger stops debugging.** At some point the debugger will get an ExitProcess event, meaning that the debuggee is no longer executing. Sometime thereafter, the debugger releases all references to any interfaces it has, and then calls ICorDebug::Terminate.

3.2 Controlling the Program

During debugging, controlling the program consists of setting breakpoints in managed code, stepping through managed and unmanaged code, and handling first and last chance exceptions. In the sections that follow, brief descriptions are presented to explain how to use the debug interfaces to accomplish these tasks.

3.2.1 Setting a Breakpoint in Managed Code

The following is a step-by-step description of how a breakpoint is set in managed code.

- **The debugger obtains a module object for the given function.** The debugger calls *GetModuleFromMetaDataInterface* on *ICorDebugAppDomain* with the metadata interface to obtain an *ICorDebugModule* object for the function's module.
- **The debugger obtains a function object for the given function.** The debugger calls *GetFunctionFromToken* on *ICorDebugModule* to obtain the function object.
- **The debugger obtains the code object for the given function.** The debugger calls *GetILCode* on *ICorDebugFunction* to obtain the code object for the function.
- **The debugger creates a breakpoint in the managed code.** The debugger calls *CreateBreakpoint* on *ICorDebugCode* with a specific offset to set a breakpoint in the function. *CreateBreakpoint* returns an instance of *ICorDebugBreakpoint*. The breakpoint will be created in the active state.
- **The debugger continues execution of the process.** The debugger calls *Continue* on the *ICorDebugProcess* object for the current debuggee process.
- **The Debugger Interface notifies the debugger when the breakpoint is hit.** The Debugger Interface calls the *Breakpoint* callback on *ICorDebugManagedCallback* when a thread reaches the breakpoint.

3.2.2 Stepping through Managed and Unmanaged Code

The following is a step-by-step description of how a debugger single-steps through managed code. In this scenario, unmanaged code is called during the stepping.

- **The debugger creates a stepper given the thread in which the single-step is to occur.** The debugger calls *CreateStepper* on the *ICorDebugThread* object for the thread being stepped. Alternatively, the debugger calls *CreateStepper* on the *ICorDebugFrame* object for the frame relative to which the stepping is to occur. It is assumed that the process is stopped when the stepper is created.
- **The debugger steps the thread.** The debugger calls *Step* on the *ICorDebugStepper* object created in the previous step.
- **The debugger continues execution of the process.** The debugger calls *Continue* on the *ICorDebugProcess* object for the current debuggee process.
- **The Debugger Interface informs the debugger that the step has completed.** The Debugger Interface calls *StepComplete* on the *ICorDebugManagedCallback* object that the debugger had registered with the runtime.
- **The debugger steps the thread.** The debugger calls *Step* on the *ICorDebugStepper* object again to step the thread.
- **The debugger continues execution of the process.** The debugger calls *Continue* on the *ICorDebugProcess* object for the current debuggee process.

- **Step alternatives:**
    - **The debugger optionally skips stepping in native code.** The debugger calls *StepOut* on the *ICorDebugStepper* object to skip stepping until the previous frame is reactivated. The Debugger Interface component will call *StepComplete* on *ICorDebugManagedCallback* when the managed code is reentered.
    - **The debugger optionally steps into next managed code.** The debugger calls *Step* on the *ICorDebugStepper* object so that control is returned when previous managed code frame is reentered or when new managed code is called by the unmanaged code.
    - **The debugger continues execution of the process.** The debugger calls *Continue* on the *ICorDebugProcess* object for the current debuggee process.
    - **The Debugger Interface informs the debugger that unmanaged code is being stepped into.** The Debugger Interface component calls *DebugEvent* on the *ICorDebugUnmanagedCallback* interface.

3.2.3 Handling Exceptions

The following is a step-by-step description of how a first chance exception is handled.

- **The runtime informs the debugger that a first chance exception has occurred.** The Debugger Interface calls *Exception* on the *ICorDebugManagedCallback* interface that the debugger registered with the runtime.
- **The debugger obtains information about the exception.** The debugger calls *GetCurrentException* on the *ICorDebugThread* object it was passed in the callback to obtain an exception object **(***ICorDebugValue***).**
- **The debugger obtains the *ICorDebugObjectValue* object for the exception.** The debugger calls *QueryInterface* on the *ICorDebugValue* object to obtain the *ICorDebugObjectValue* object for the exception.
- **The debugger obtains the class of the exception object that was thrown.** The debugger calls *GetClass* on the *ICorDebugObjectValue* exception object.
- **The debugger decides to ignore the exception by simply continuing.**
- **The runtime informs the debugger that a last chance exception has occurred.** The Debugger Interface component calls *Exception* on the *ICorDebugManagedCallback* object and specifies that the exception is a “last chance” exception.
- **The user decides the exception is inconsequential.** The debugger calls *ClearCurrentException* on the *ICorDebugThread* object for the current debuggee thread. This clears the exception and prevents the exception from being thrown.
- **The debugger continues execution of the process.** The debugger calls *Continue* on the *ICorDebugProcess* object for the current debuggee process.

3.3 Examining the Program

To examine a program during debugging, the debugger needs to be able to access managed stack frames and evaluate expressions. Step-by-step descriptions are given below for each of these tasks.

3.3.1 Accessing Call Stacks

The following is a step-by-step description of how a debugger accesses managed stack frames. The debuggee process must be stopped at this point.

- **The debugger obtains an enumerator for the stack chains.** The debugger calls *EnumerateChains* on the *ICorDebugThread* object for the thread for which stack chains are to be accessed to obtain an *ICorDebugChainEnum* object to enumerate the stack chains.
- **The debugger iterates through the stack chains.** The debugger calls *Next* on the *ICorDebugChainEnum* object to iterate through the stack chains.
- **The debugger obtains an enumerator for the stack frames in the chain.** The debugger calls *EnumerateFrames* on the *ICorDebugChain* object.
- **The debugger iterates through the stack frames.** The debugger calls *Next* on *ICorDebugFrameEnum* to iterate through the stack frames in the chain.
- **The debugger optionally obtains the IP address.** The debugger calls *GetIP* on *ICorDebugILFrame* to obtain the IP relative to the start of the function for the stack frame.
- **The debugger optionally obtains other information about the stack frame.** The debugger calls *GetFunctionToken* to obtain the metadata token for the function for the code that the stack frame is running. The debugger calls *GetCode* to obtain an object representing the code that the stack frame is running.

3.3.2 Evaluating Expressions

Expressions in unmanaged native code are evaluated using the same mechanisms debuggers use today. In managed code, the debugger can evaluate an expression as follows:

- Parse the expression
- Call the Debugging APIs to
  - Access values of variables in the expression.
  - Invoke functions in the expression.

Alternatively, the debugger can do the following:

- Wrap the expression in a global function and compile the function.
- Call the Debugging API (Edit & Continue) to add the global function.
- Call the Debugging API to evaluate the function.

The following is a step-by-step description of how a debugger evaluates an expression. In this scenario, the expression is **A + Foo()** where **A** is assumed to be in a register and the code being debugged is native managed.

- **The debugger obtains the value of A.** The debugger calls *GetLocalRegisterValue* on the *ICorDebugNativeFrame* object for the stack frame in which the expression is to be evaluated.
- **The debugger creates an evaluation object.** The debugger calls *CreateEval* on the *ICorDebugThread* object for the thread in which the expression is to be evaluated.
- **The debugger computes the value of Foo().** The debugger calls *CallFunction* on the *ICorDebugEval* object.
- **The debugger evaluates the expression.** The debugger applies constant folding to the expression using the values obtained in the previous two steps.

3.4 Injecting Code Dynamically

This section describes Dynamic Code Injection using the Debugging Services. Dynamic Code Injection executes a function that wasn't present in the original PE, for example, an expression in the Immediate Window in Microsoft Visual Studio for Visual Basic. The runtime hijacks an active thread to execute the code. The debugger may request the runtime to run or freeze the remaining threads. Because Dynamic Code Injection is built on Function Evaluation, a dynamically injected function can be debugged as if it were a regular code, i.e., all normal Debugging Services such as setting breakpoints, stepping, etc. can be invoked on the dynamically injected code.

A debugger must do the following to dynamically inject code:

- Compose a function that will execute the dynamic code.
- Inject the code into the debuggee using Edit and Continue.
- Execute the code, repeatedly if the user desires, by invoking the composed function.

3.4.1 Composing the Function

1. The first step is to compute the signature for the function that will execute the dynamic code.
   To do this, append the signature for the local variables to the signature for the method being executed in the leaf frame. A signature constructed this way does not require the minimal subset of used variables to be computed. The runtime will ignore the unused variables. See the section Getting the Signature From the Metadata for details.

   All arguments to the function must be declared to be *ByRef*. This will allow Function Evaluation to propagate any changes to the variables within the body of the injected function back to the leaf frame within the debuggee.

   It is possible that some variables won't be in scope when the dynamic code is executed. In such situations, a Null reference should be passed. If such a variable is referenced, a *NullReferenceException* will be thrown. When this happens, the function evaluation will complete by calling the *ICorDebugManagedCallback::EvalException* callback.
2. Choose a unique name for the function. The name must be special. The special name allows a debugger to prevent a user from browsing the function. The Debugging Services expects the function name to be prefixed with the string

"_Hidden:". When the method is added, a flag will be set indicating that the function name is special.

3.4.2 Injecting the Code

1. The debugger should ask the compiler to build a function whose body is the code to be injected dynamically.
2. The debugger computes a delta PE. To do this, the debugger calls *ICorDebugModule::GetEditAndContinueSnapshot* to obtain an *ICorDebugEditAndContinueSnapshot* object. The debugger calls methods on *ICorDebugEditAndContinueSnapshot* to create the delta PE image. The debugger calls *ICorDebugAppDomain::CommitChanges* to install the delta PE in the running image.
3. The dynamically injected function should be placed at the same level of visibility as the leaf frame in which it will execute. If the leaf frame is an instance method, then the dynamically injected function should also be an instance method within the same class. If the leaf frame is a static method (global methods are static methods belong to a particular class), then the dynamically injected function should also be a static method.
4. It is important to note that the function will exist within the debuggee even after the dynamic code injection process completes. This allows previously injected code to be reevaluated repeatedly without having to compose the function and inject the code again, i.e., the steps described in this section and the previous section can be skipped.

3.4.3 Executing Injected Code

1. For each variable in the signature, get its value (an *ICorDebugValue* object) using the debugging inspection routines. Use either *ICorDebugThread::GetActiveFrame* or *ICorDebugChain::GetActiveFrame* to get the leaf frame, *QueryInterface* for *ICorDebugILFrame*. Call *ICorDebugILFrame::EnumerateLocalVariables*, *ICorDebugILFrame::EnumerateArguments*, *ICorDebugILFrame::GetLocalVariable*, or *ICorDebugILFrame::GetArguments* to get the actual variables.

   Note that if the debugger is attached to a debuggee that doesn't have CORDBG_ENABLE set (i.e., to a debuggee that has not been collecting debugging information), the debugger will not be able to get an *ICorDebugILFrame*, and thus will not be able to collect values for the Function Evaluation.

   It does not matter if the objects that are arguments to the dynamically injected function are dereferenced weakly or strongly. When the function is evaluated, the runtime will hijack the thread in which the injection occurred. This will leave the original leaf frame on the stack, along with all the original, strong references. The only case where this would not be true is if all the references within the debuggee were weak, in which case running the dynamic code injection might trigger a garbage collection that may cause the object to be garbage collected.
2. Use *ICorDebugThread::CreateEval* to create an *ICorDebugEval* object. *ICorDebugEval* provides methods to evaluate a function. Call one of these methods. Methods such as *ICorDebugEval::CallFunction* only set up the function evaluation. The debugger needs to call *ICorDebugProcess::Continue* or *ICorDebugAppDomain::Continue* to run the debuggee and evaluate the function.

When the evaluation has completed, the Debugging Services will call *ICorDebugManagedCallback::EvalComplete* or *ICorDebugManagedCallback::EvalException* to notify the debugger about the function evaluation.

If the function evaluation returns an object, the object will be a strongly referenced.

If the dynamically injected code attempts to dereference a *Null* reference passed to the function that wraps the code, the Debugging Services will call *ICorDebugManagedCallback::EvalException*. In response, the debugger could notify the user that it cannot evaluate the injected code.

Note that *ICorDebugEval::CallFunction* does not do virtual dispatches; use *ICorDebugObjectValue::GetVirtualMethod* if you want virtual dispatches.

If the debuggee is multithreaded and the debugger does not want any of the other threads running, the debugger should call *ICorDebugAppDomain::SetAllThreadsDebugState* or *ICorDebugProcess::SetAllThreadsDebugState* and set the state of all threads except that of the thread used for function evaluation to THREAD_SUSPEND. There is a risk that this may result in a deadlock, depending on what the dynamically injected code does.

3.4.4 Miscellaneous Issues

- Since the runtime security is determined by context policies, unless one specifically takes actions to affect security, dynamic code injection will operate with the same security permissions and capabilities as the leaf frame.
- Dynamically injected functions can be added wherever Edit and Continue allows the functions to be added. A logical choice is to add them to the leaf frame.
- Note that there are no limits on the number of fields, instance methods, or static methods that can be added to a class using Edit and Continue operations. The maximum amount of static data allowed is predefined and limited at this time to 1 MB per module.
- Non-local GoTo statements are not allowed in dynamically injected code.

3.4.5 Getting the Signature From the Metadata

Obtaining a MetaData Dispenser

- The debugger calls ICorDebugModule::GetMetaDataInterface with REFIID IID_IMetaDataDispenser to obtain a metadata dispenser.
- The debugger calls IMetaDataDispenser::OpenScope with the REFIID IID_IMetaDataImport to obtain the IMetaDataImport interface.

Finding the method using IMetaDataImport

- The debugger calls *IMetaDataImport::GetMethodProps* to look up the method using its token. The method token can be obtained using *ICorDebugFunction::GetToken*. *GetMethodProps* will return the signature of the method.
- The debugger calls *IMetaDataImport::GetSigFromToken* to obtain the local signature, i.e., the signature of the local variables. The debugger must supply the

token for the signature of the local variables. The debugger can obtain this token by calling *ICorDebugFunction::GetLocalVarSigToken*.

Constructing the Function Signature

The format of the signature is documented in the specification *Type and Signature Encoding in Metadata*, and takes precedence over anything described in this document.

The section *Method Declaration* in the signature specification describes the format for the method signature. The format is a single byte for the calling convention, followed by a single byte for the count of arguments, followed by a list of types each of which can have a different size. If the calling convention is CALLCONV_VARARG, the count of arguments will be the total number of arguments, i.e., fixed arguments plus variable arguments. An ELEMENT_TYPE_SENTINEL byte marks where the fixed arguments end and where the variable arguments begin.

The section *Stand-Alone Signatures* in the signature specification describes the format of local signatures. Standalone signatures do not use the VARARGS calling convention.

After obtaining the method signature and the local signature, the debugger should allocate space for a new signature. The debugger should then iterate through the method signature, and for each type, put the ELEMENT_TYPE_BYREF byte in the new signature followed by the type. The process is repeated this until the end of method signature is reached or a type marked ELEMENT_TYPE_SENTINEL is reached. Next, the debugger should copy the local signature types, marking each type ELEMENT_TYPE_BYREF. If the method signature has a VARARGS calling convention, the debugger should copy those types marking each of them ELEMENT_TYPE_BYREF. Finally, the debugger should update the count of arguments.

4 In-Process Debugging APIs

The In-Process Debugging API ("Inproc Debugging API") is intended for use by various tools and clients that run in the same process as the debuggee – the profiler being the most notable example. They allow for inspection of, but not manipulation of, variables, and specifically do not allow for flow-control or edit and continue.

The Inproc Debugging API interfaces are the same as those used by the out-of-process debugger, although some methods will return CORDBG_E_INPROC_NOT_IMPL, to indicate that the method is unavailable in-process. There are a couple ways to obtain one of these interfaces within the process.

4.1 GetInprocInspectionInterface

GetInprocInspectionInterface retrieves an ICorDebug interface.

4.2 GetInprocInspectionIThisThread

GetInprocInspectionIThisThread returns an ICorDebugThread interface that inspects the managed thread that the profiler callback is currently being called from.

Note that this method may fail if there is no managed thread that is attached to the native operating system thread. For example, the Win32 thread that executes the Initialize callback will allow the profiler to take actions, even though the managed Thread object hasn't been created yet.

5 Debug Interfaces

The Debug API supplies interfaces for debugging that can be organized into the following categories of functionality:

- Registration
- Notification
- Breakpoints
- Execution
- Information
- Enumeration

Interface	**Inherits From**	**Description**
ICorDebug	IUnknown	The interface pointer to this object represents an event processing loop for a debugger process.
ICorDebugAppDomain	ICorDebugController	This interface provides methods that apply to application domains.
ICorDebugAppDomainEnum	ICorDebugEnum	This interface provides methods for enumerating application domains.
ICorDebugArrayValue	ICorDebugHeapValue	This interface provides methods for accessing array elements.
ICorDebugAssembly	IUnknown	This interface provides methods that apply to assemblies.
ICorDebugAssemblyEnum	ICorDebugEnum	This interface provides methods for enumerating assemblies.
ICorDebugBoxValue	ICorDebugHeapValue	This interface provides methods that apply to boxed value class objects.
ICorDebugBreakpoint	IUnknown	This interface provides methods for retrieving information about breakpoints.
ICorDebugBreakpointEnum	ICorDebugEnum	This interface provides methods for enumerating breakpoints.
ICorDebugChain	IUnknown	This interface provides access to call stacks in the stack chain.
ICorDebugChainEnum	ICorDebugEnum	This interface provides methods for enumerating stack chains.
ICorDebugClass	IUnknown	This interface provides methods for obtaining information about classes.
ICorDebugCode	IUnknown	This interface provides methods for

		obtaining information about code.
ICorDebugContext	ICorDebugObjectValue	This interface provides methods for obtaining information about contexts.
ICorDebugController	IUnknown	The ICorDebugContext interface represents a scope at which program execution context can be controlled. It represents either a process or an application domain.
ICorDebugEditAndContinueSnapshot	IUnknown	This interface provides methods for Edit & Continue operations.
ICorDebugEnum	IUnknown	This interface provides methods for enumerating objects. It is the root of the interface hierarchy for all the enumeration interfaces described below.
ICorDebugErrorInfoEnum	ICorDebugEnum	This interface provides methods for enumerating error information objects.
ICorDebugEval	IUnknown	This interface provides methods for running code inside the debuggee.
ICorDebugFrame	IUnknown	This interface provides access to call stacks within the threads of the debuggee. Each stack frame represents the state of execution within a method.
ICorDebugFrameEnum	ICorDebugEnum	This interface provides methods for enumerating stack frames.
ICorDebugFunction	IUnknown	This interface provides methods for obtaining information about functions.
ICorDebugFunctionBreakpoint	ICorDebugBreakpoint	This interface provides methods for retrieving information about function breakpoints.
ICorDebugGenericValue	ICorDebugValue	This interface provides methods for obtaining generic values.
ICorDebugHeapValue	ICorDebugValue	This interface provides methods that apply to garbage collected objects.
ICorDebugILFrame	ICorDebugFrame	This interface provides methods for obtaining information about IL frames.
ICorDebugManagedCallback	IUnknown	This interface provides methods that allow the runtime to communicate with the debugger concerning events in managed code

		in the debuggee process.
ICorDebugModule	IUnknown	This interface provides methods for obtaining information about modules.
ICorDebugModuleBreakpoint	ICorDebugBreakpoint	This interface provides methods for retrieving information about module breakpoints. *This interface is not yet implemented.*
ICorDebugModuleEnum	ICorDebugEnum	This interface provides methods for enumerating modules.
ICorDebugNativeFrame	ICorDebugFrame	This interface provides methods for obtaining information about native frames.
ICorDebugObjectEnum	ICorDebugEnum	This interface provides methods for enumerating managed objects.
ICorDebugObjectValue	ICorDebugValue	This interface provides methods for obtaining values of objects.
ICorDebugProcess	ICorDebugController	This interface provides methods for controlling and inspecting a debuggee process.
ICorDebugProcessEnum	ICorDebugEnum	This interface provides methods for enumerating process objects.
ICorDebugReferenceValue	ICorDebugValue	This interface provides methods that apply to values that are references (to objects).
ICorDebugRegisterSet	IUnknown	This interface provides methods for obtaining information about registers.
ICorDebugStepper	IUnknown	This interface provides methods for controlling stepping.
ICorDebugStepperEnum	ICorDebugEnum	This interface provides methods for enumerating steppers.
ICorDebugStringValue	ICorDebugHeapValue	This interface provides methods for obtaining string values.
ICorDebugThread	IUnknown	This interface provides access to threads in the runtime.
ICorDebugThreadEnum	ICorDebugEnum	This interface provides methods for enumerating thread objects.
ICorDebugUnmanagedCallback	IUnknown	This interface provides methods that allow the runtime to communicate with the debugger concerning events in unmanaged code in the debuggee process.

ICorDebugValue	IUnknown	This interface provides methods for obtaining values.
ICorDebugValueBreakpoint	ICorDebugBreakpoint	This interface provides methods for retrieving information about value breakpoints.
ICorDebugValueEnum	ICorDebugEnum	This interface provides methods for enumerating values.

5.1 ICorDebug : IUnknown

The runtime implements the ICorDebug interface, which provides methods that allow a debugger to register callback interfaces for notification about managed and unmanaged events. This interface also provides methods for managing debuggee processes.

The debugger must wait for the ExitProcess callback before releasing the ICorDebugProcess and ICorDebug interfaces.

CreateProcess

Not Implemented In-Process.

Launches a process under the control of the debugger. All parameters are the same as the Win32 CreateProcess call.

Note, that the DEBUG_PROCESS flag (passed in dwCreationFlags), if set, will enable unmanaged debugging. If only managed debugging is desired, do not set this flag. (Note that unmanaged debugging can also be enabled later by the EnableUnmanagedDebuging entry point on the process).

Note that if debuggingFlags is set to DEBUG_ENABLE_EDIT_AND_CONTINUE, then E&C will be allowed for the process. Otherwise, the argument should be zero'd out, and no E&C will be allowed. E&C is not allowed when JIT attaching to a process.

```
HRESULT CreateProcess(LPCWSTR lpApplicationName, LPWSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes, LPSECURITY_ATTRIBUTES
lpThreadAttributes, BOOL bInheritHandles, DWORD dwCreationFlags, PVOID
lpEnvironment, LPCWSTR lpCurrentDirectory, LPSTARTUPINFOW
lpStartupInfo, LPPROCESS_INFORMATION lpProcessInformation,
CorDebugCreateProcessFlags debuggingFlags, ICorDebugProcess
**ppProcess)
```

In/ Out	Parameter	Description
in	lpApplicationName	Pointer to name of executable.
in	lpCommandLine	Pointer to command line string to be passed to application.
in	lpProcessAttributes	Pointer to process security attributes.
in	lpThreadAttributes	Pointer to thread security attributes.

in	bInheritHandles	Handle inheritance flag.
in	dwCreationFlags	Creation flags.
in	lpEnvironment	Pointer to new environment block.
in	lpCurrentDirectory	Pointer to current directory name.
in	lpStartupInfo	Pointer to STARTUPINFO.
in	lpProcessInformation	Pointer to PROCESS_INFORMATION.
in	debuggingFlags	Debugging flags
out	ppProcess	Pointer to pointer to a process object.

DebugActiveProcess

Not Implemented In-Process

Used to attach to an existing process.

If win32Attach is TRUE, then the debugger becomes the Win32 debugger for the process and will begin dispatching the unmanaged callbacks.

```
HRESULT DebugActiveProcess(DWORD id, BOOL win32Attach, ICorDebugProcess
*ppProcess)
```

In/ Out	Parameter	Description
in	id	Process ID of the process to attach to.
in	win32Attach	If TRUE, mixed-mode debugging will be enabled.
out	ppProcess	Pointer to pointer to a process object.

EnumerateProcesses

Returns an enumerator (**ICorDebugProcessEnum**) for all processes being debugged.

```
HRESULT EnumerateProcesses(ICorDebugProcessEnum **ppProcess)
```

In/ Out	Parameter	Description
out	ppProcess	Pointer to pointer to an enumerator for the processes.

GetProcess

Returns the process object (**ICorDebugProcess**) for the process with the given Win32 process ID.

```
HRESULT GetProcess(DWORD dwProcessId, ICorDebugProcess **ppProcess)
```

In/ Out	Parameter	Description
in	dwProcessId	The Win32 process ID of the process.
out	ppProcess	Pointer to pointer to the process object for the process with the specified Win32 process ID.

Initialize

The debugger calls this method at creation time to initialize the debugging services, and must be called at creation time before any other method on ICorDebug is called. It is not necessary to call this method when using the In Process Debugging API.

```
HRESULT Initialize()
```

SetManagedHandler

Not Implemented In-Process

The debugger calls this method to provide a callback that should receive notification and information regarding managed events in the debuggee process. The debugger must pass an interface to its own COM object that implements ICorDebugManagedCallback.

```
HRESULT SetManagedHandler(ICorDebugManagedCallback *pCallback)
```

In/ Out	Parameter	Description
in	pCallback	A pointer to the COM interface implemented by the debugger that is to receive notifications for the debuggee process.

SetUnmanagedHandler

Not Implemented In-Process

The debugger calls this method to provide a callback that should receive notification and information regarding unmanaged events in the debuggee process. The debugger must pass an interface to its own COM object that implements ICorDebugUnmanagedCallback.

```
HRESULT SetUnmanagedHandler(ICorDebugUnmanagedCallback *pCallback)
```

In/ Out	Parameter	Description
in	pCallback	A pointer to the COM interface implemented by the debugger that is to receive notifications for the debuggee process.

Terminate

Not Implemented In-Process

The debugger calls this method when ICorDebug is no longer needed.

```
HRESULT Terminate()
```

5.2 ICorDebugAppDomain : ICorDebugController

This interface represents an application domain.

The debugger obtains an ICorDebugAppDomain object by calling ICorDebugProcess::EnumerateAppDomains and then enumerating the application domain objects, by calling ICorDebugThread::GetAppDomain, or by calling ICorDebugAssembly::GetAppDomain.

The debugger must wait for the ExitAppDomain callback before releasing the ICorDebugAppDomain interfaces.

Attach

Not Implemented In-Process

Attach attaches the debugger to this application domain. The debugger will receive all application domain related events.

```
HRESULT Attach()
```

EnumerateAssemblies

Returns an enumerator object (**ICorDebugAssemblyEnum**) for all assemblies in the application domain.

```
HRESULT EnumerateAssemblies(ICorDebugAssemblyEnum **ppAssemblies)
```

In/ Out	Parameter	Description
out	ppAssemblies	Pointer to pointer to an enumerator object for all assemblies in the application domain.

EnumerateBreakpoints

Not Implemented In-Process

EnumerateBreakpoints returns an enum (**ICorDebugBreakpointEnum**) of all active breakpoints in the app domain. This includes all types of breakpoints : function breakpoints, data breakpoints, etc.

```
HRESULT EnumerateBreakpoints(ICorDebugBreakpointEnum **ppBreakpoints)
```

In/ Out	Parameter	Description

out	ppBreakpoints	Pointer to pointer to an enumerator object for all breakpoints in the debuggee process.

EnumerateSteppers

Not Implemented In-Process

Returns an enumerator object (**ICorDebugStepperEnum**) for all active steppers in the debuggee process.

```
HRESULT EnumerateSteppers(ICorDebugStepperEnum **ppSteppers)
```

In/ Out	Parameter	Description
out	ppSteppers	Pointer to pointer to an enumerator for all active steppers.

GetID

Returns the ID of this application domain.

```
HRESULT GetID(ULONG23 *pId)
```

In/ Out	Parameter	Description
out	**pId**	Pointer to the 32 bit number that represents the ID of this application domain. This number is unique across appdomains, within a single process.

GetModuleFromMetaDataInterface

Returns a module object (**ICorDebugModule**) for the given metadata interface.

```
HRESULT GetModuleFromMetaDataInterface(IUnknown *pIMetaData,
ICorDebugModule **ppModule)
```

In/ Out	Parameter	Description
in	pIMetaData	Pointer to the metadata interface.
out	ppModule	Pointer to pointer to the object for the module corresponding to the metadata interface.

GetName

Returns the name of the application domain.

```
HRESULT GetName(ULONG32 cchName, ULONG32 *pcchName, WCHAR szName[])
```

In/ Out	Parameter	Description
in	cchName	The allocated size of string buffer.
out	pcchName	The number of characters available for return. No more than cchName are actually returned in the buffer
out	szName[]	The string buffer.

GetObject

Returns the runtime application domain object.

Note: *This method is not yet implemented.*

```
HRESULT GetObject(ICorDebugValue **ppObject)
```

In/ Out	Parameter	Description
out	ppObject	Pointer to pointer to an object that represents the runtime application domain object.

GetProcess

Returns the process containing the application domain.

```
HRESULT GetProcess(ICorDebugProcess **ppProcess)
```

In/ Out	Parameter	Description
out	ppProcess	Pointer to pointer to an object that represents the process containing the application domain.

IsAttached

Not Implemented In-Process

IsAttached returns whether or not the debugger is attached to the application domain. The methods of ICorDebugController cannot be used until the debugger attaches to the application domain.

```
HRESULT IsAttached(BOOL *pbAttached)
```

In/ Out	Parameter	Description
out	pbAttached	Pointer to boolean which is TRUE if a debugger is attached to the application domain.

5.3 ICorDebugAppDomainEnum : ICorDebugEnum

The **ICorDebugAppDomainEnum** interface provides methods for enumerating objects that represent application domain.

Next

This method is used to retrieve application domain objects. The number of application domain objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorAppDomain *appDomains[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of application domain objects requested to be retrieved.
out	appDomains[]	Array of pointers to application domain objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.4 ICorDebugArrayValue : ICorDebugHeapValue

This interface is used to obtain information about an array value object such as the number of elements in an array or the value of a specific element of an array. Value objects are returned by the following methods: ICorDebugArrayValue:: GetElement, ICorDebugClass::GetStaticFieldValue, ICorDebugObjectValue::GetFieldValue, ICorDebugILFrame::GetLocalVariable, ICorDebugILFrame::GetArgumentValue, ICorDebugILFrame::GetStackValue, and the various register value access functions defined on ICorDebugILFrame.

GetBaseIndicies

GetBaseIndicies returns the base index of each dimension in the array.

```
HRESULT GetBaseIndicies(ULONG32 cdim, ULONG32 indicies[])
```

In/ Out	Parameter	Description
in	cdim	The size of the array indicies[].
out	indicies[]	The array that will contain the base index of each dimension in the array.

GetCount

GetCount returns the number of elements in all dimensions of the array.

HRESULT GetCount(ULONG32 *pnCount)

In/ Out	Parameter	Description
out	pnCount	Pointer to the number of elements in the array.

GetDimensions

GetDimensions returns the dimensions of the array.

HRESULT GetDimensions(ULONG32 cdim, ULONG32 dims[])

In/ Out	Parameter	Description
In	cdim	The size of the array dims[].
out	dims[]	The array that will contain the returned dimensions.

GetElement

Returns a value object (ICorDebugValue) representing an element of the array. The indicies array must not be null.

HRESULT GetElement(ULONG32 cdim, ULONG32 indicies[], ICorDebugValue **ppValue)

In/ Out	Parameter	Description
in	cdim	The size of the array indicies[].
in	indicies[]	The array that contains the index of each dimension in the array.
out	ppValue	Pointer to pointer to the object that represents the value of the element of the array.

GetElementAtPosition

Returns a value object (ICorDebugValue) representing the element at the given position in the array. The position is over all elements of the array.

HRESULT GetElementAtPosition(ULONG32 nPosition, ICorDebugValue **ppValue)

In/ Out	Parameter	Description
in	nPosition	The position of the element of the array.
out	ppValue	Pointer to pointer to the object that represents the value

		of the element of the array.

GetElementType

Returns the simple type of the elements of the array.

```
HRESULT GetElementType(CorElementType *pType)
```

In/ Out	Parameter	Description
out	pType	Pointer to the type of the elements of the array.

GetRank

GetRank returns the number of dimensions in the array.

```
HRESULT GetRank(ULONG32 *pnRank)
```

In/ Out	Parameter	Description
out	pnRank	Pointer to the number of dimensions in the array.

HasBaseIndicies

HasBaseIndicies returns whether or not the array has base indices.

```
HRESULT HasBaseIndicies(BOOL *pbHasBaseIndicies)
```

In/ Out	Parameter	Description
out	pbHasBaseIndicies	Pointer to a Boolean indicating whether or not the array has base indices. If this is set to FALSE, then 0 is used as the base index.

5.5 ICorDebugAssembly : IUnknown

This interface represents an assembly.

The debugger obtains an ICorDebugAssembly object by calling ICorDebugManagedCallback::LoadAssembly, ICorDebug::GetSystemAssembly, ICorDebugModule::GetAssembly, or ICorDebugAppDomain::EnumerateAssemblies.

EnumerateModules

Returns an enumerator object (**ICorDebugModuleEnum**) for all loaded modules in the assembly.

```
HRESULT EnumerateModules(ICorDebugModuleEnum **ppModules)
```

In/ Out	Parameter	Description
out	ppModules	Pointer to pointer to an enumerator object for all the modules in the assembly.

GetAppDomain

Returns the application domain containing the assembly. Returns **null** if this is the system assembly.

```
HRESULT GetAppDomain(ICorDebugAppDomain **ppAppDomain)
```

In/ Out	Parameter	Description
out	ppAppDomain	Pointer to a pointer to the application domain object that contains the assembly.

GetCodeBase

Returns the code base used to load the assembly (for example, a URL where the code was loaded from).

Note: *This method is not yet implemented.*

```
HRESULT GetCodeBase(ULONG32 cchName, ULONG32 *pcchName, WCHAR szName[])
```

In/ Out	Parameter	Description
in	cchName	The allocated size of string buffer.
out	pcchName	The number of characters available for return. No more than cchName are actually returned in the buffer
out	szName[]	The string buffer.

GetName

Returns the name of the assembly.

```
HRESULT GetName(ULONG32 cchName, ULONG32 *pcchName, WCHAR szName[])
```

In/ Out	Parameter	Description
in	cchName	The allocated size of string buffer.
out	pcchName	The number of characters available for return. No more than cchName are actually returned in the buffer
out	szName[]	The string buffer.

GetProcess

Returns the process containing the assembly.

```
HRESULT GetProcess(ICorDebugProcess **ppProcess)
```

In/ Out	Parameter	Description
out	ppProcess	Pointer to pointer to the process object that contains the assembly.

5.6 ICorDebugAssemblyEnum : ICorDebugEnum

The **ICorDebugAssemblyEnum** interface provides methods for enumerating objects that represent assemblies.

Next

This method is used to retrieve assembly objects. The number of assembly objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorAssembly *assemblies[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of assembly objects requested to be retrieved.
out	assemblies[]	Array of pointers to assembly objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.7 ICorDebugBoxValue : ICorDebugHeapValue

This interface provides methods that return information about boxed value class objects.

GetObject

Returns the value object that is in the box.

```
HRESULT GetObject(ICorDebugObjectValue **ppObject)
```

In/ Out	Parameter	Description
out	ppObject	Pointer to pointer to the value object that is in the box.

5.8 ICorDebugBreakpoint : IUnknown

Not Implemented In-Process : Neither this, nor any of the interfaces which inherit from this.

This interface provides methods that return information about breakpoints that can be breakpoints set in a function or a watchpoint set on a value. An ICorDebugBreakpoint object is created by calling the various CreateBreakpoint methods. Note that breakpoints have no direct support for deactivation or condition expressions. The debugger must implement this functionality on top of this interface if desired.

Activate

The debugger calls this method to set the active state of the breakpoint.

`HRESULT Activate(BOOL bActive)`

In/ Out	Parameter	Description
in	bActive	If TRUE, activate the breakpoint. If FALSE, deactivate the breakpoint.

IsActive

The debugger calls this method to check if the breakpoint is active.

`HRESULT Activate(BOOL *pbActive)`

In/ Out	Parameter	Description
out	pbActive	Pointer to a Boolean that is TRUE if the breakpoint is active.

5.9 ICorDebugBreakpointEnum : ICorDebugEnum

Not Implemented In-Process

This interface provides a method for enumerating breakpoint objects.

Next

This method is used to retrieve breakpoint objects. The number of breakpoint objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugBreakpoint *breakpoints[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of breakpoint objects requested to be retrieved.
out	breakpoints[]	Array of pointers to breakpoint objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.10 ICorDebugChain : IUnknown

This interface provides methods to obtain information about stack chains. A stack chain is a segment of a physical or logical call stack. All frames in a chain occupy contiguous stack space, and they share the same thread and context. A chain may represent either managed or unmanaged code, although an unmanaged chain will have no visible frames.

The debugger obtains a ICorDebugChain object by calling ICorDebugThread::EnumerateChains, ICorDebugChain::GetCaller, or ICorDebugChain::GetCallee.

EnumerateFrames

Returns an enumerator for all the stack frames in the chain, starting at the most recently active one. This should be called only for managed chains.

> **Note:** *The Debugging Services does not provide methods for obtaining information about unmanaged chains. The debugger needs to use other means to obtain this information.*

```
HRESULT EnumerateFrames(ICorDebugFrameEnum **ppFrames)
```

In/ Out	Parameter	Description
out	ppFrames	Pointer to pointer to an enumerator for all of the stack frames in the chain.

GetActiveFrame

GetActiveFrame is a convenience routine to return the active (most recent) frame on the chain, if any.

HRESULT GetActiveFrame([out] ICorDebugFrame **ppFrame)

In/ Out	Parameter	Description
out	ppFrame	Pointer to pointer to the frame that is most active.

GetCallee

Returns a chain object (**ICorDebugChain**) for the chain which this chain is waiting on before it resumes. Note that this may be a chain on another thread in the case of cross-thread-marshalled calls. The callee will be NULL if the chain is currently actively running.

```
HRESULT GetCallee(ICorDebugChain **ppChain)
```

In/ Out	Parameter	Description
out	ppChain	Pointer to pointer to the chain object for the chain that this chain is waiting on.

GetCaller

Returns a chain object (**ICorDebugChain**) for the chain which called this chain. Note that this may be a chain on another thread in the case of cross-thread-marshalled calls. The caller will be NULL for spontaneously called chains (e.g., the ThreadProc, a debugger initiated call, etc.)

```
HRESULT GetCaller(ICorDebugChain **ppChain)
```

In/ Out	Parameter	Description
out	ppChain	Pointer to pointer to the chain object for the chain that called this chain.

GetContext

Returns the context object (**ICorDebugContext**) for all of the frames in the chain.

Note: *This method is not yet implemented.*

```
HRESULT GetContext(ICorDebugContext  **ppContext)
```

In/ Out	Parameter	Description
out	ppContext	Pointer to pointer to the context object for all of the frames in the chain.

GetNext

Returns a pointer to a pointer to a chain object (**ICorDebugChain**) for the chain which was on this thread after the current one, if there is one.

```
HRESULT GetNext(ICorDebugChain **ppChain)
```

In/ Out	Parameter	Description
out	ppChain	Pointer to pointer to a chain object.

GetPrevious

Returns a pointer to a pointer to a chain object (**ICorDebugChain**) for the chain which was on this thread before the current one was pushed, if there is one.

```
HRESULT GetPrevious(ICorDebugChain **ppChain)
```

In/ Out	Parameter	Description
out	ppChain	Pointer to pointer to a chain object.

GetReason

Returns the reason for the genesis of this calling chain.

```
HRESULT GetReason(CorDebugChainReason *pReason)
```

In/ Out	Parameter	Description
out	pReason	Pointer to a structure describing the reason.

GetRegisterSet

Returns the register set for the active part of the chain.

```
HRESULT GetRegisterSet(ICorDebugRegisterSet **ppRegisters)
```

In/ Out	Parameter	Description
out	ppRegisters	Pointer to pointer to a register set object.

GetStackRange

Returns the absolute address range of the stack segment for the call chain. Note that you cannot make any assumptions about what is actually stored on the stack – the numeric range is to compare stack frame locations only.

```
HRESULT GetStackRange(CORDB_ADDRESS *pStart, CORDB_ADDRESS *pEnd)
```

In/ Out	Parameter	Description
out	pStart	Pointer to the real Win32 minimum value that bounds the stack segment.
out	pEnd	Pointer to the real Win32 maximum value that bounds

		the stack segment.

GetThread

Returns the thread object (ICorDebugThread) for the thread to which this call chain belongs.

```
HRESULT GetThread(ICorDebugThread **ppThread)
```

In/ Out	Parameter	Description
out	ppThread	Pointer to pointer to a thread object for the thread to which the stack frame belongs.

IsManaged

Determines whether or not the chain is currently running managed code.

```
HRESULT IsManaged(BOOL *pManaged)
```

In/ Out	Parameter	Description
out	pManaged	Pointer to a Boolean that is TRUE if the chain is running managed code.

5.11 ICorDebugChainEnum : ICorDebugEnum

The ICorDebugChainEnum interface provides methods for enumerating objects that represent stack chains. The debugger obtains an ICorDebugChainEnum object by calling ICorDebugThread::EnumerateChains.

Next

This method is used to retrieve stack chain objects. The number of stack chain objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugChain *chains[], ULONG *pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of stack chain objects requested to be retrieved.
out	chains[]	Array of pointers to chain objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.12 ICorDebugClass : IUnknown

This interface provides methods for accessing information about a class, such as field values and metadata. The debugger obtains an ICorDebugClass object by calling ICorDebugFunction::GetClass, ICorDebugObjectValue::GetClass, or ICorDebugModule::GetClassFromToken.

GetModule

Returns the module object (**ICorDebugModule**) for the class.

```
HRESULT GetModule(ICorDebugModule **ppModule)
```

In/ Out	Parameter	Description
out	ppModule	Pointer to pointer to the module object for the class.

GetStaticFieldValue

GetStaticFieldValue returns a value object for the given static field variable. If the static field could possibly be relative to either a thread, context, or appdomain, then pFrame will help the debugger determine the proper value.

```
HRESULT GetStaticFieldValue(mdFieldDef fieldDef, ICorDebugFrame
*pFrame, ICorDebugValueFrame **ppValue)
```

In/ Out	Parameter	Description
in	fieldDef	Field definition.
in	pFrame	Pointer to a frame
out	ppValue	Pointer to pointer to a value object for the static field.

GetToken

Returns the metadata typedef token for the class.

```
HRESULT GetToken(mdTypeDef *pTypeDef)
```

In/ Out	Paramete r	Description
out	pTypeDef	Pointer to the metadata typedef token for the class.

5.13 ICorDebugCode : IUnknown

The ICorDebugCode interface provides methods to access information about IL code such as size and address. The interface also provides methods to set breakpoints in code.

The debugger obtains an ICorDebugCode object by calling ICorDebugFrame::GetCode or ICorDebugFunction::GetILCode.

CreateBreakpoint

Not Implemented In-Process

Sets a breakpoint in the function at the given offset.

Note that the breakpoint must be activated before it is active.

If this code is IL, and there is a JIT-compiled version of the code, the breakpoint will be applied in the JIT-compiled code as well.

```
HRESULT CreateBreakpoint(ULONG32 offset, ICorDebugFunctionBreakpoint
**ppBreakpoint)
```

In/ Out	Parameter	Description
in	offset	Offset from the beginning of the function.
out	ppBreakpoint	Pointer to pointer to the breakpoint object for the breakpoint set.

GetAddress

Returns the address of the code.

```
HRESULT GetAddress(CORDB_ADDRESS *pStart)
```

In/ Out	Parameter	Description
out	pStart	Pointer to the address of the code.

GetCode

Returns the code of the method suitable for disassembly. Note that instruction boundaries aren’t checked.

```
HRESULT GetCode(ULONG32 startOffset, ULONG32 endOffset, ULONG32
cBufferAlloc, BYTE buffer[], ULONG32 *pcBufferSize)
```

In/ Out	Parameter	Description
in	startOffset	Starting offset of code from the beginning of the method.
in	endOffset	Ending offset of code from the beginning of the method.
in	cBufferAlloc	Size of allocated buffer array
out	buffer	Buffer in which code is to be returned.

out	pcBufferSize	The number of bytes available for return. No more than cBufferAlloc are actually returned in the buffer

GetFunction

Returns an **ICorDebugFunction** object representing the function for the code.

```
HRESULT GetFunction(ICorDebugFunction **ppFunction)
```

In/ Out	Parameter	Description
out	ppFunction	Pointer to pointer to the function object for code.

GetILToNativeMapping

GetILToNativeMapping returns a map from IL offsets to native offsets for this code. An array of COR_DEBUG_IL_TO_NATIVE_MAP structs will be returned, and some of the ilOffsets in this array map may be the values specified in CorDebugIlToNativeMappingTypes. Note: this method is only valid for ICorDebugCodes representing native code that was jitted from IL code.

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

In/ Out	Parameter	Description
in	cMap	Pointer to the map array.
out	pcMap	The number of elements written into the map array.
out	map	Space allocated by the caller to receive the IL to native mapping information.

GetSize

Returns the size of the code.

```
HRESULT GetSize(ULONG32 *pcBytes)
```

In/ Out	Parameter	Description
out	pcBytes	Pointer to the size of the code.

GetVersionNumber

Returns the version number of the code.

```
HRESULT GetVersioNumber(ULONG32 *nVersion)
```

In/ Out	Parameter	Description
out	nVersion	Pointer to the version of the code.

IsIL

Returns TRUE if IL code is being executed.

```
HRESULT IsIL(BOOL *pbIL)
```

In/ Out	Parameter	Description
out	pbIL	Pointer to a Boolean indicating if the code is IL.

5.14 ICorDebugContext : ICorDebugObjectValue

The ICorDebugContext interface provides methods that return information about a context.

The debugger obtains an ICorDebugContext object by calling ICorDebugChain::GetContext.

5.15 ICorDebugController : IUnknown

The ICorDebugController interface represents a scope at which program execution context can be controlled. It represents either a process or an application domain. The interfaces ICorDebugProcess and ICorDebugAppDomain extend ICorDebugController.

If this is the controller of a process, the controller affects all threads in the process. Otherwise the controller only affects the threads of a particular application domain.

CanCommitChanges

Not Implemented In-Process

Checks if the delta PE's can be applied to the running application domain. If there are any known problems with the changes, then information about the error is returned. The runtime will ensure that circular dependencies are handled.

```
HRESULT CanCommitChanges(ULONG cSnapshots, const
ICorDebugEditAndContinueSnapshot *pSnapshots[], ICorDebugErrorInfoEnum
**pError)
```

In/ Out	Parameter	Description
in	cSnapshots	Number of elements in the pSnapshots array.

in	pSnapshots	Array of pointer to ICorDebugEditAndContinueSnapshot objects representing Edit & Continue snapshots.
out	pError	Pointer to enumerator for error information.

CommitChanges

Not Implemented In-Process

Applies the delta PE's to the running application domain. If failures occur, detailed information about the errors is returned. There are no rollback guarantees when a failure occurs. Applying delta PE's to a running application domain must be done in the order the snapshots are retrieved and may not be interleaved, i.e., there is no merging of multiple snapshots applied out of order or with the same root. The runtime will ensure that circular dependencies are handled. Partial commits are not supported.

```
HRESULT CommitChanges(ULONG cSnapshots, const
ICorDebugEditAndContinueSnapshot *pSnapshots[], ICorDebugErrorInfoEnum
**pError)
```

In/ Out	Parameter	Description
in	cSnapshots	Number of elements in the pSnapshots array.
in	pSnapshots	Array of pointer to ICorDebugEditAndContinueSnapshot objects representing Edit & Continue snapshots.
out	pError	Pointer to enumerator for error information.

Continue

Not Implemented In-Process

Continues the process or application domain after a call to **Stop**.

```
HRESULT Continue(fIsOutOfBand)
```

In/ Out	Parameter	Description
in	fIsOutOfBand	fIsOutOfBand is set to TRUE if continuing from an unmanaged event that was sent with the fOutOfBand flag in the unmanaged callback and it is set to FALSE if continuing from a managed event or a normal unmanaged event.

Detach

Not Implemented In-Process

Detaches the debugger from the application domain or process. The application domain or process continues execution normally (thus the debugger shouldn't call Continue after Detach()ing). The ICorDebugProcess object (if detaching from a process) or ICorDebugAppDomain object (if detaching from an application domain) is no longer valid after the detach.

Note: If a debugger attempts to detach from a soft-attached process, the detach will always succeed. If a debugger attempts to detach from a hard-attached process, i.e., the attach will succeed subject to operating system limitations.

```
HRESULT Detach()
```

EnumerateThreads

Returns an enumeration of all runtime threads active in the debuggee process or application domain

```
HRESULT EnumerateThreads(ICorDebugThreadEnum **ppThreads)
```

In/ Out	Parameter	Description
out	ppThreads	Pointer to pointer to an enumerator for all active runtime threads in the debuggee process or application domain.

HasQueuedCallbacks

Not Implemented In-Process

Returns TRUE if there are currently managed callbacks that are queued up for the given thread. These callbacks will be dispatched one at a time, each time **Continue** is called.

If NULL is given for the pThread parameter, HasQueuedCallbacks will return TRUE if there are currently managed callbacks queued for any thread.

```
HRESULT HasQueuedCallbacks (ICorDebugThread *pThread, BOOL *pbQueued)
```

In/ Out	Parameter	Description
in	pThread	Pointer to the specified thread. If NULL, HasQueuedCallbacks will return TRUE if there are currently managed callbacks queued for any thread.
out	pbQueued	Pointer to a Boolean that is TRUE if there are currently mapped callbacks that are queued up.

IsRunning

Not Implemented In-Process

Returns TRUE if the threads in the process or application domain are running freely.

```
HRESULT IsRunning(BOOL *pbRunning)
```

In/ Out	Parameter	Description

out	pbRunning	Pointer to a Boolean that represents the running state of the process or application. TRUE is returned if the process or application domain is currently running.

SetAllThreadsDebugState

Not Implemented In-Process

Sets the current debug state of each thread. See ICorDebugThread::SetDebugState for details.

```
HRESULT SetAllThreadsDebugState(CorDebugThreadState state,
ICorDebugThread *pExceptThisThread)
```

In/ Out	Parameter	Description
in	state	The debug state to which the thread should be set.
in	pExceptThisThread	Pointer to the thread that is exempt from the debug state change. Use NULL if you want to affect all threads.

Stop

Not Implemented In-Process

Performs a cooperative stop on all threads running managed code in the process or application domain. Threads running managed code are suspended (unless this is in-process). If a cooperative stop fails due to a deadlock, all threads are suspended and E_TIMEOUT is returned.

```
HRESULT Stop(DWORD dwTimeout)
```

In/ Out	Parameter	Description
in	dwTimeout	The time period in milliseconds after which this function should timeout.

Terminate

Not Implemented In-Process

Terminates the process or application domain with extreme prejudice.

Note. If the process or application domain is stopped when **Terminate** is called, the process or application domain should be continued using **ICorDebugController::Continue** so that the **ExitProcess** or **ExitAppDomain** callback is received.

> **Note:** *This method is not yet implemented by an AppDomain, but is implemented at the process level.*

```
HRESULT Terminate(UINT *exitCode)
```

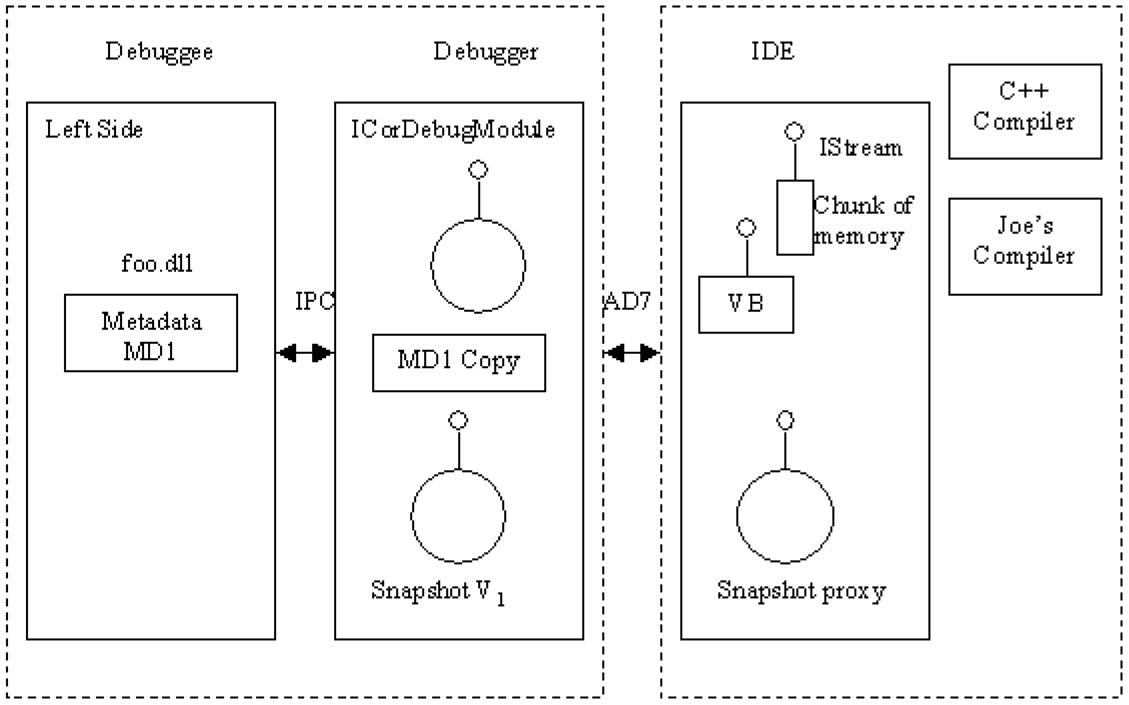
In/ Out	Parameter	Description
in	exitCode	Exit code for the process or application domain.

5.16 ICorDebugEditAndContinueSnapshot : IUnknown

Not Implemented In-Process

This interface provides methods that allow a debugger to modify the code being debugged using Edit & Continue operations.

The debugger obtains an **ICorDebugEditAndContinueSnapshot** object by calling **ICorDebugModule::GetEditAndContinueSnapshot**.



The snapshot objects are managed objects. The compiler allocates a chunk of memory and calls **CopyMetadata** to request the metadata to be written to the memory. Alternatively, the compiler can request **CopyMetadata** to write the metadata to a file. **CopyMetadata** takes a pointer to a GUID and a pointer to an object that implements the **IStream** interface. The compiler must provide an implementation of the **IStream** interface. The GUID identifies the metadata version of the snapshot.

Once the metadata (version 1) has been copied to memory or to a file, the compiler does an "open scope" operation to load the metadata. The compiler now emits the changes required by the Edit and Continue operation to the metadata scope. The IDE then copies the changes to the metadata, i.e., the delta PE, over to the debuggee resulting in modified metadata (version 2).

Normally, the compiler applies sequences of Edit and Continue operations generating successive versions of snapshots. The compiler can avoid repeatedly copying metadata from the debuggee by maintaining a cache of snapshots. In this scenario, the compiler requests a snapshot when it first applies Edit and Continue operations to obtain the first version of the snapshot. For subsequent Edit and Continue operations, the compiler invokes **GetMvid** to obtain a pointer to the GUID that represents the current version of the snapshot in the debuggee. The compiler compares this GUID with the GUID that it obtained when it called **CopyMetadata**. If the two GUIDs match, the compiler does not need to call **CopyMetadata** again. Instead, the compiler can apply the Edit and Continue operations to its copy of the metadata corresponding to version 2 of the snapshot and generate the next version of the snapshot and return the new delta PE.

CopyMetadata

Saves a copy of the executing metadata from the debuggee for this snapshot to the output stream. The stream implementation must be supplied by the caller and will typically either save the copy to memory or to disk. Only the IStream::Write method will be called by this method. The MVID value returned is the unique metadata ID for this copy of the metadata. It may be used on subsequent edit and continue operations to determine if the client has the most recent version already (performance win to cache).

```
HRESULT CopyMetadata(IStream *pIStream, GUID *pMvid)
```

In/ Out	Parameter	Description
in	pIStream	Pointer to an IStream object to which the metadata is to be copied.
out	pMvid	Pointer to a GUID that represents the version of the metadata in the debuggee.

GetMvid

Returns the currently active metadata ID for the executing process. This value can be used in conjunction with CopyMetaData to cache the most recent copy of the metadata and avoid expensive copies. So for example, if you call CopyMetaData once and save that copy, then on the next E&C operation you can ask for the current MVID and see if it is already in your cache. If it is, use your version instead of calling CopyMetaData again.

```
HRESULT GetMvid(GUID *pMvid)
```

In/ Out	Parameter	Description
out	pMvid	Pointer to a GUID that represents the version of the metadata in the debuggee.

GetRoDataRVA

Returns the base RVA that should be used when adding new static read-only data to an existing image. The EE will guarantee that any RVA values embedded in the code are valid when the delta PE is applied with new data. The new data will be added to a page that is marked read-only. The new data must be contained in a .rdata section.

```
HRESULT GetRoDataRVA(ULONG32 *pRoDataRVA)
```

In/ Out	Parameter	Description
out	pRoDataRVA	Pointer to base address of space for read-only data that a compiler wishes to emit

GetRwDataRVA

Returns the base RVA that should be used when adding new static read/write data to an existing image. The EE will guarantee that any RVA values embedded in the code are valid when the delta PE is applied with new data. The new data will be added to a page that is marked to allow read and write access. The new data must be contained in a .data section.

```
HRESULT GetRwDataRVA(ULONG32 *pRwDataRVA)
```

In/ Out	Parameter	Description
out	pRwDataRVA	Pointer to base address of space for read/write data that a compiler wishes to emit

SetILMap

This method is called once for every method being replaced that has active instances on a call stack on a thread in the target process. It is up to the caller of the API to determine that this is the case. The target process should be halted before making this check and before calling this method.

The method tells the runtime how the old code maps to the new code. The runtime will map breakpoints from the locations in the old code to the corresponding locations in the new code. If there are no breakpoints that need to be mapped, the mapping table is only required for live instruction pointers. If breakpoints need to be mapped, the full IL-map must be provided.

Right now, only sequence points should be mapped.

```
HRESULT SetILMap(mdToken mdFunction, ULONG cMapSize, const COR_IL_MAP
ilmap[])
```

In/ Out	Parameter	Description
in	mdFunction	Function metadata token.
in	cMapSize	Number of elements in the IL-map array.
in	ilMap	Mapping table (see below).

SetPEBytes

This method gives the snapshot object a pointer to the delta PE which was based on the snapshot. This pointer value will be AddRef'd and cached until *ICorDebugProcess::CanCommitChanges* and/or *ICorDebugProcess::CommitChanges* are called, at which point the engine will read the delta PE and remote it into the debuggee process where the changes will be checked/applied.

```
HRESULT SetPEBytes(IStream *pIStream)
```

In/ Out	Parameter	Description
in	pIStream	IStream from which the delta PE is to be read.

SetPESymbolBytes

This method gives the snapshot object a pointer to the updated symbols for the delta PE which was based on the snapshot. This pointer value will be AddRef'd and cached until *ICorDebugProcess::CanCommitChanges* and/or *ICorDebugProcess::CommitChanges* are called, at which point the engine will read the delta PE and remote it into the debuggee process where the changes will be checked/applied.

```
HRESULT SetPESymbolBytes(IStream *pIStream)
```

In/ Out	Parameter	Description
in	pIStream	IStream from which the updated symbols for the delta PE is to be read.

5.17 ICorDebugEnum : IUnknown

The ICorDebugEnum interface provides methods for enumerating objects. This is the root of the interface hierarchy for all enumeration interfaces.

ICorDebugEnum objects are returned by various methods defined on the runtime Debugging interfaces.

Clone

Copies a pointer to the current position in the list to another enumerator object.

```
HRESULT Clone(ICorDebugEnum **ppEnum )
```

In/ Out	Parameter	Description
out	ppEnum	Pointer to pointer to the target enumerator object.

GetCount

Gets the number of elements pointed to by the enumerator object.

```
HRESULT GetCount(ULONG *pcelt)
```

In/ Out	Parameter	Description
out	pcelt	Pointer to the number of elements pointed to by the enumerator object.

Reset

Sets or resets the position of the enumerator to the beginning of the list.

```
HRESULT Reset()
```

Skip

Moves the position of the enumeration forward. The number of objects to be skipped is based on a parameter passed to the method.

```
HRESULT Skip(ULONG celt)
```

In/ Out	Parameter	Description
in	celt	The number of elements to be skipped.

5.18 ICorDebugErrorInfoEnum : ICorDebugEnum

The ICorDebugErrorInfoEnum interface provides methods for enumerating objects that represent error information objects.

The debugger obtains an ICorDebugErrorInfoEnum object by calling ICorDebugProcess:CanCommitChanges or ICorDebugProcess::CommitChanges.

Next

Used to retrieve error information objects. The number of such objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, IErrorInfo *errorInfo[], ULONG *pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of IErrorInfo objects requested to be retrieved.

out	errorInfo[]	Array of pointer to error information objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched

5.19 ICorDebugEval : IUnknown

Not Implemented In-Process

ICorDebugEval collects functionality that runs code inside the debuggee. Note that the operations do not complete until ICorDebugProcess::Continue is called, and the EvalComplete callback is called.

An ICorDebugEval object is created in the context of a specific thread that will be used to perform the evaluation. If you need to use this functionality without changing the state of the program, set the DebugState of the program’s threads to STOP before calling Continue.

Note that since user code is running when the evaluation is in progress, any debug events can occur, including class loads, breakpoints, etc. Callbacks will be called normally in such a case. The state of the Eval will be seen as part of the normal program state inspection; the full debugger API continues to operate as normal in such a circumstance. Evals can even be nested. Also, the user code may never complete due to deadlock or infinite looping. In this case you will need to Abort the Eval before resuming the program.

The following apply to function evaluation:

- If the function evaluation terminates with an exception, an attached debugger can inspect the exception that was thrown.
- Execution control is in effect during evaluation. Breakpoints in the function being evaluated can be hit.
- Function evaluations can be nested.
- When a debugger receives event callbacks and it inspects the stack chain, the chain reason for the function evaluation will show up as CHAIN_FUNC_EVAL.
- A debugger will be able to evaluate a function only if the debuggee thread on which the function evaluation is done is at a GC safe point.
- Strong and weak references are supported.
- Currently, a debugger cannot evaluate a function if it is stopped in some internal runtime code.

In future, the following fill be supported:

- Ability to evaluate a function when the debuggee process is at any point.

Abort

Aborts the current computation. Note that in the case of nested Evals, this may fail unless it is the most recent Eval.

```
HRESULT Abort()
```

CallFunction

Sets up a function call. Note that the given function is called directly; there is no virtual dispatch. Use ICorDebugObjectValue::GetVirtualMethod to do a virtual dispatch.

```
HRESULT CallFunction(ICorDebugFunction *pFunction, ULONG32 nArgs,
ICorDebugValue *pArgs[])
```

In/ Out	Parameter	Description
in	pFunction	Pointer to the function to be called.
in	nArgs	Number of arguments.
in	pArgs	Array of pointers to arguments of the function.

CreateValue

Creates an IcorDebugValue of the given type for the sole purpose of using it in a function evaluation. These can be use to pass user constants as parameters. The value has a zero or NULL initial value. Use ICorDebugValue::SetValue to set the value.

pElementClass is only required for value classes. Pass NULL otherwise.

If elementType == ELEMENT_TYPE_CLASS, then you get an ICorDebugReferenceValue representing the NULL object reference. You can use this to pass NULL to evals that have object reference parameters. You cannot set the ICorDebugReferenceValue to anything. It always remains NULL.

```
HRESULT CreateValue(CorElementType elementType, ICorDebugClass
*pElementClass, IcorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	elementType	The element type of the value.
in	pElementClass	Pointer to a value class.
out	ppValue	Pointer to pointer to a value object.

GetResult

Returns the result of the evaluation. This is only valid after the evaluation is completed. If the evaluation completes normally, the result will be the return value. If it terminates with an exception, the result is the exception thrown.

```
HRESULT GetResult(ICorDebugValue **ppResult)
```

In/ Out	Parameter	Description
out	ppResult	Pointer to a pointer to the object representing the result of the evaluation.

GetThread

Returns the thread that this eval was created from.

```
HRESULT GetThread(ICorDebugThread **ppThread)
```

In/ Out	Parameter	Description
in	ppThread	Pointer to a pointer to the object representing the thread that this eval was created from.

IsActive

Returns whether or not the Eval has an active computation.

```
HRESULT IsActive(BOOL *pbActive)
```

In/ Out	Parameter	Description
in	pbActive	Pointer to a Boolean representing the active state of the Eval.

NewArray

Allocates a new array with the given element type and dimensions.

```
HRESULT NewArray(CorElemetnType elementType, ICorDebugClass
*pElementClass, ULONG32 rank, ULONG32 dims[], ULONG32 lowBounds[])
```

In/ Out	Parameter	Description
in	elementType	The element type of the array.
in	pElementClass	Pointer to class of the objects that represent the elements of the array.
in	rank	The rank of the array.
in	dims[]	The dimensions of the array.
in	lowBounds[]	The lower bounds of the array.

NewObject

Allocates and calls the constructor for an object.

```
HRESULT NewObject(ICorDebugFunction *pConstructor, ULONG32 nArgs,
ICorDebugValue *pArgs[])
```

In/ Out	Parameter	Description
in	pConstructor	Pointer to the function that represents the constructor.
in	nArgs	Number of arguments.
in	pArgs	Array of pointers to arguments of the constructor.

NewObjectNoConstructor

Allocates a new object without attempting to call any constructor on the object.

```
HRESULT NewObjectNoConstructor(ICorDebugClass *pClass)
```

In/ Out	Parameter	Description
in	pClass	Pointer to the class of the object.

NewString

Allocates a string object with the given contents.

```
HRESULT NewString (LPWSTR string)
```

In/ Out	Parameter	Description
in	string	Pointer to string representing the contents of the string object.

5.20 ICorDebugFrame : IUnknown

This interface provides methods that allow a debugger to view and change code that is about to be executed. ICorDebugFrame also provides methods that return information about the function that activated the stack frame.

The debugger obtains an ICorDebugFrame object by enumerating frame objects using the enumerator returned by ICorDebugChain::EnumerateFrames.

CreateStepper

Not Implemented In-Process

Creates a stepper object which operates relative to the frame. The Stepper API must then be used to perform actual stepping.

Note that if this frame is not active, the frame will typically have to be returned to before the step is completed.

```
HRESULT CreateStepper(ICorDebugStepper **ppStepper)
```

In/ Out	Parameter	Description
out	ppStepper	Pointer to pointer to the created stepper object.

GetCallee

Returns a pointer to the frame in the current chain which this frame called, or NULL if this is the innermost frame in the chain.

```
HRESULT GetCallee(ICorDebugFrame **ppFrame)
```

In/ Out	Parameter	Description
out	ppFrame	Pointer to pointer to the frame object in the current chain which this frame called.

GetCaller

Returns a pointer to the frame in the current chain that called this frame, or NULL if this is the outermost frame in the chain.

```
HRESULT GetCaller(ICorDebugFrame **ppFrame)
```

In/ Out	Parameter	Description
out	ppFrame	Pointer to pointer to the frame object in the current chain that called this frame.

GetChain

Returns an **ICorDebugChain** object representing the chain of which this stack frame is a part.

```
HRESULT GetChain(ICorDebugChain **ppChain)
```

In/ Out	Parameter	Description
out	ppChain	Pointer to pointer to a chain object.

GetCode

Returns an **ICorDebugCode** object representing the code that the stack frame is running.

```
HRESULT GetCode(ICorDebugCode **ppCode)
```

In/	Parameter	Description

Out		
out	ppCode	Pointer to pointer to a code object for code that the stack frame is running.

GetFunction

Returns an **ICorDebugFunction** object representing the function for the code that the stack frame is running.

```
HRESULT GetFunction(ICorDebugFunction **ppFunction)
```

In/ Out	Parameter	Description
out	ppFunction	Pointer to pointer to a function object representing the function for the code that the stack frame is running.

GetFunctionToken

Returns the token for the function for the code which this stack frame is running.

```
HRESULT GetFunctionToken(mdMethodDef *pToken)
```

In/ Out	Parameter	Description
out	pToken	Pointer to the metadata function token for the code which this stack frame is running.

GetStackRange

Returns the absolute address range of the stack frame. This is useful for piecing together interleaved stack traces gathered from multiple EE engines.) Note that you cannot make any assumptions about what is actually stored on the stack – the numeric range is to compare stack frame locations only.

Note: *This method is not yet implemented.*

```
HRESULT GetStackRange(CORDB_ADDRESS *pStart, CORDB_ADDRESS *pEnd)
```

In/ Out	Parameter	Description
out	pStart	Pointer to the real Win32 minimum value that bounds the stack frame.
out	pEnd	Pointer to the real Win32 maximum value that bounds the stack frame.

5.21 ICorDebugFrameEnum : ICorDebugEnum

The ICorDebugFrameEnum interface provides methods for enumerating objects that represent stack frames.

Next

This method is used to retrieve stack frame objects. The number of frame objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugFrame *frames[], ULONG *pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of stack frame objects requested to be retrieved.
out	frames[]	Array of pointers to frame objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.22 ICorDebugFunction : IUnknown

This interface provides methods that return information about a function, such as its metadata, the class in which the function is defined, the module to which the class belongs, etc.

The debugger obtains an ICorDebugFunction object by calling ICorDebugCode::GetFunction or ICorDebugModule::GetFunctionFromToken.

CreateBreakpoint

Not Implemented In-Process

CreateBreakpoint creates a breakpoint at the start of the function.

```
HRESULT CreateBreakpoint(ICorDebugFunctionBreakpoint **ppBreakpoint)
```

In/ Out	Parameter	Description
out	ppBreakpoint	Pointer to pointer to the function breakpoint object.

GetClass

Returns the class object (ICorDebugClass) for the function.

```
HRESULT GetClass(ICorDebugClass **ppClass)
```

In/ Out	Parameter	Description
out	ppClass	Pointer to pointer to the class object for the function. Null is returned if the function is not a member.

GetCurrentVersionNumber

Obtains the current version of the function, which is the same version as that obtained by ICorDebugCode::GetVersionNumber with the pointer that GetILCode or GetNativeCode returns

HRESULT GetCurrentVersionNumber(ULONG32 *pnCurrentVersion)

In/ Out	Parameter	Description
out	pnCurrentVersion	The current version.

GetILCode

Returns the IL code for the function. Returns NULL if there is no IL code.

```
HRESULT GetILCode (ICorDebugCode **ppCode)
```

In/ Out	Parameter	Description
out	ppCode	The code object.

GetLocalVarSigToken

Returns the token for the local variable signature for this function.

```
HRESULT GetLocalVarSigToken(mdSignature *pmdSig)
```

In/ Out	Parameter	Description
out	pmdSig	Pointer to a metadata signature token. NULL is returned if the function has no local variables.

GetModule

Returns the module object (**ICorDebugModule**) for the function.

```
HRESULT GetModule(ICorDebugModule **pModule)
```

In/ Out	Parameter	Description
out	pModule	Pointer to pointer to the module object for the function.

GetNativeCode

Returns the native code for the function. Returns NULL if there is no native code.

```
HRESULT GetNativeCode (ICorDebugCode **ppCode)
```

In/ Out	Parameter	Description
out	ppCode	Pointer to pointer to the code object that represents native code for the function.

GetToken

Returns the metadata methodDef token for the function.

```
HRESULT GetToken(mdMethodDef *pMethodDef)
```

In/ Out	Parameter	Description
out	pMethodDef	Pointer to the metadata methodDef token for the function.

5.23 ICorDebugFunctionBreakpoint : ICorDebugBreakpoint

Not Implemented In-Process

This interface provides methods that return information about function breakpoints. An ICorDebugFunctionBreakpoint object is created by calling ICorDebugFunction::CreateBreakpoint or ICorDebugCode::CreateBreakpoint.

GetFunction

The debugger calls this method to get the function at which the breakpoint occurred.

```
HRESULT GetFunction(ICorDebugFunction **ppFunction)
```

In/ Out	Parameter	Description
out	ppFunction	Pointer to pointer to the object that represents the function at which the breakpoint occurred.

GetOffset

The debugger calls this method to get the offset within the function at which the breakpoint occurred.

```
HRESULT GetOffset(ULONG32 *pnOffset)
```

In/ Out	Parameter	Description
out	pnOffset	Pointer to the offset within the function at which the breakpoint occurred.

5.24 ICorDebugGenericValue : ICorDebugValue

ICorDebugGenericValue applies to all values.

GetValue

Copies the value into the specified buffer. The buffer should be the appropriate size for the simple type.

```
HRESULT GetValue(void *pTo)
```

In/ Out	Parameter	Description
out	pTo	Pointer to the value.

SetValue

Not Implemented In-Process

Copies a new value from the specified buffer. The buffer should be the appropriate size for the simple type.

```
HRESULT SetValue(void *pFrom)
```

In/ Out	Parameter	Description
in	pFrom	Pointer to the value.

5.25 ICorDebugHeapValue : ICorDebugValue

ICorDebugHeapValue represents a garbage collected object. Since heap values are represented with strong or weak handles, they can be held onto indefinitely (across Continues).

CreateRelocBreakpoint

Not Implemented In-Process

Creates a breakpoint that will be triggered when the address in the reference changes due to a garbage collection.

Note: *This method is not yet implemented.*

```
HRESULT CreateRelocBreakpoint(ICorDebugValueBreakpoint **ppBreakpoint)
```

In/ Out	Parameter	Description
out	ppBreakpoint	Pointer to a pointer to a value breakpoint object for the breakpoint that will be triggered.

IsValid

Tests whether the object is valid. The object becomes invalid if the garbage collector reclaims the object.

```
HRESULT IsValid(BOOL *pbValid)
```

In/ Out	Parameter	Description
out	pbValid	Pointer to a Boolean that is TRUE if the object is valid.

5.26 ICorDebugILFrame : ICorDebugFrame

This interface provides methods that return information about IL frames. The IL frames can be running interpreted or JIT-compiled code.

The debugger obtains an ICorDebugILFrame object by calling ICorDebugChain::EnumerateChains, which returns an enumerator for stack frames, and then calling the methods defined on ICorDebugFrameEnum to enumerate the stack frames.

CanSetIP

Not Implemented In-Process

CanSetIP attempts to determine if it is safe to set the instruction pointer to the IL at the given offset. If this returns S_OK, then executing SetIP will result in a safe, correct, continued execution. If CanSetIP returns anything else, SetIP can still be invoked, but continued, correct execution of the debuggee cannot be guaranteed. See the "Debugging Services" specification for details of the conditions under which the instruction pointer may be set.

```
HRESULT CanSetIP(ULONG32 nOffset)
```

In/ Out	Parameter	Description
in	nOffset	IL offset to which instruction pointer should be set.

EnumerateArguments

Returns a list of the arguments available in the frame. Note that this will include varargs arguments as well as arguments declared by the function signature.

```
HRESULT EnumerateArguments(ICorDebugValueEnum **ppValueEnum)
```

In/ Out	Parameter	Description
out	ppValueEnum	Pointer to pointer to the objects representing arguments available in the frame.

EnumerateLocalVariables

Returns a list of the local variables available in the frame. Note that this may not include all the locals in the running function, as some of them may not be active.

```
HRESULT EnumerateLocalVariables(ICorDebugValueEnum **ppValueEnum)
```

In/ Out	Parameter	Description
out	ppValueEnum	Pointer to pointer to an enumerator object for local variables in the frame.

GetArgument

Returns the value object (ICorDebugValue) for the value of an argument in an IL frame. This can be used either in an interpreted or JIT-compiled frame.

The slot number is as follows:

- If a method has 'this', then 'this' is slot 0. All parameters (including return values) follow.
  So, if a method is not static and has a return value, then 'this' is slot 0, the return value is slot 1, the first user argument is 2, etc.
- If a method is not static and doesn't have a return value, then 'this' is slot 0, the first user argument is 1, etc.

Basically, 'this' is always in slot 0. All arguments are in slot 1..n, slot 0..n if the method is static. An optional return value is passed to a method as the first argument, it is just hidden from the user. So you just have to be sure to count your return values in the arguments.

```
HRESULT GetArgument(DWORD dwIndex, ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	dwIndex	Location on the stack frame.
out	ppValue	Pointer to pointer to the object that represents the value of the argument.

GetIP

Returns the byte offset into the IL code from the start of the function for this stack frame. If this stack frame is active, this address is the next instruction to execute. If this stack frame is not active, this is the next instruction to execute when the stack frame is reactivated.

Note that if this is a JITted frame, the IP will be determined by mapping backwards from the actual native IP, so the value may not be exact.

```
HRESULT GetIP(ULONG32 *pnOffset, CorDebugMappingResult *pMappingResult)
```

In/ Out	Parameter	Description
out	pnOffset	Pointer to the offset of IL from the start of the function.
out	pMappingResult	Pointer to an object describing the details of how the IP was obtained (see below).

GetLocalVariable

Returns the value object (ICorDebugValue) for a local variable in an IL frame. This can be used either in an interpreted or JIT-compiled frame.

```
HRESULT GetLocalVariable(DWORD dwIndex, ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	dwIndex	Location on the stack frame.
out	ppValue	Pointer to pointer to the object that represents the value of the variable.

GetStackDepth

Returns the operand stack depth of the IL stack frame.

Note: *This method is not yet implemented.*

```
HRESULT GetStackDepth(ULONG32 *pDepth)
```

In/ Out	Parameter	Description
out	pDepth	Pointer to the return value of the stack depth.

GetStackValue

Returns a value from the operand stack in an IL frame.

Note: *This method is not yet implemented*.

```
HRESULT GetStackValue(DWORD dwIndex, ICorDebugValue **ppValue)
```

In/	Parameter	Description

Out		
in	dwIndex	Index of the operand.
out	ppValue	Pointer to pointer to the object that represents the value of the operand.

SetIP

Not Implemented In-Process

Sets the instruction pointer to the IL at the given offset. Note that this is an inherently dangerous thing to do. The debugger (or debugger user) is responsible for adjusting the local state of the function so that it remains consistent. In particular, for interpreted methods the stack may need adjustment. See the "Debugging Services" specification for details of the conditions under which the instruction pointer may be set.

```
HRESULT SetIP(ULONG32 nOffset)
```

In/ Out	Parameter	Description
in	nOffset	The offset of IL from the start of the function.

HRESULT Codes Returned

- CORDBG_E_CANT_SET_IP_INTO_CATCH
- CORDBG_E_CANT_SET_IP_INTO_FINALLY
- CORDBG_E_CODE_NOT_AVAILABLE
- CORDBG_S_BAD_END_SEQUENCE_POINT
- CORDBG_S_BAD_START_SEQUENCE_POINT
- CORDBG_S_INSUFFICIENT_INFO_FOR_SETIP
- E_FAIL
- S_OK

5.27 ICorDebugManagedCallback : IUnknown

Not Implemented In-Process

The debugger implements this callback interface and registers it with the runtime by calling **ICorDebug::SetManagedHandler**. The runtime calls methods defined on this interface to notify the debugger about managed events in the debuggee process.

All callbacks are called with the process in the synchronized state. All callbacks are serialized, and are called in the same thread. Each callback implementor must call **Continue** in a callback to resume execution. If Continue is not called before returning, the process will remain stopped. **Continue** must be called before any more event callbacks will happen.

Break

Notifies the debugger when a break opcode in the code stream is executed.

```
HRESULT Break(ICorDebugAppDomain *pAppDomain,  ICorDebugThread
*pThread)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.

Breakpoint

Notifies the debugger when a breakpoint is hit by the debuggee.

```
HRESULT Breakpoint(ICorDebugAppDomain *pAppDomain, ICorDebugThread
*pThread, ICorDebugBreakpoint *pBreakpoint)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	pBreakpoint	Pointer to the object that represents the code or data breakpoint.

ControlCTrap

ControlCTrap is called if a CTRL-C is trapped in the process being debugger. All appdomains within the process are stopped for this callback. Return values:
S_OK   : Debugger will handle the ControlC Trap S_FALSE : Debugger won't handle the ControlC Trap

HRESULT ControlCTrap(ICorDebugProcess *pProcess)

In/ Out	Parameter	Description
in	pProcess	Pointer to the process object that represents the process that generated the event.

CreateAppDomain

Notifies the debugger when an application domain is created.

```
HRESULT CreateAppDomain(ICorDebugProcess *pProcess, ICorDebugAppDomain
*pAppDomain)
```

In/ Out	Parameter	Description
in	pProcess	Pointer to the process object that represents the process that generated the event.
in	pAppDomain	Pointer to the application domain object that was created.

CreateProcess

Notifies the debugger when a process is first attached to or started. This entry point won't be called until the Execution Engine is initialized. Process object is returned from create/attach in case EE never starts up.

```
HRESULT CreateProcess(ICorDebugProcess *pProcess)
```

In/ Out	Parameter	Description
in	pProcess	Pointer to the process object that represents the process that generated the event.

CreateThread

Notifies the debugger when a thread first begins executing managed code. The thread will be positioned immediately at the first managed code to be executed.

```
HRESULT CreateThread(ICorDebugAppDomain *pAppDomain,  ICorDebugThread
*pThread)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.

DebuggerError

Notifies the debugger when an error occurs while attempting to handle an event from the runtime. The process is placed in pass through mode, possibly permanently, depending on the nature of the error.

```
HRESULT DebuggerError(ICorDebugProcess *pProcess, HRESULT errorHR,
DWORD errorCode)
```

In/ Out	Parameter	Description

in	pProcess	Pointer to the process object that represents the process that generated the event.
in	errorHR	The HRESULT for the error that occurred.
in	errorCode	Additional information about the error that occurred.

EvalComplete

Notifies the debugger when an evaluation has completed.

```
HRESULT EvalComplete(ICorDebugAppDomain *pAppDomain, ICorDebugThread
*pThread, ICorDebugEval *pEval)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	pEval	Pointer to the evaluation object used to perform the evaluation.

EvalException

Notifies the debugger when an evaluation terminates with an unhandled exception.

```
HRESULT EvalException(ICorDebugAppDomain *pAppDomain, ICorDebugThread
*pThread, ICorDebugEval *pEval)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	pEval	Pointer to the evaluation object used to perform the evaluation.

Exception

Notifies the debugger when an exception occurs in managed code. The specific exception can be retrieved from the thread object.

```
HRESULT Exception(ICorDebugAppDomain *pAppDomain,  ICorDebugThread
*pThread, BOOL unhandled)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	unhandled	If FALSE, this is a “first chance” exception that hasn’t had a chance to be processed by the application. If TRUE, this is an unhandled exception which will terminate the process.

ExitAppDomain

Notifies the debugger when an application domain exits.

```
HRESULT ExitAppDomain(ICorDebugProcess *pProcess, ICorDebugAppDomain
*pAppDomain)
```

In/ Out	Parameter	Description
in	pProcess	Pointer to the process object that represents the process that generated the event.
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.

ExitProcess

Notifies the debugger about an exit-process debugging event.

```
HRESULT ExitProcess(ICorDebugProcess *pProcess)
```

In/ Out	Parameter	Description
in	pProcess	Pointer to the process object that represents the process that generated the event.

ExitThread

Notifies the debugger about an exit-thread debugging event.

```
HRESULT ExitThread(ICorDebugAppDomain *pAppDomain,  ICorDebugThread
*pThread)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents

		the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.

LoadAssembly

Notifies the debugger when an assembly is successfully loaded.

```
HRESULT LoadAssembly(ICorDebugAppDomain *pAppDomain, ICorDebugAssembly
*pAssembly)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pAssembly	Pointer to the assembly object that represents the assembly that generated the event.

LoadClass

Notifies the debugger when the runtime has finished loading a class. This callback only occurs if ClassLoading has been enabled for the class's module.

```
HRESULT LoadClass(ICorDebugAppDomain *pAppDomain,  ICorDebugClass
*pClass)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pClass	Pointer to the class that was loaded.

LoadModule

Notifies the debugger when the runtime has loaded a module. This is an appropriate time to examine metadata for the module or enable or disable JIT debugging.

```
HRESULT LoadModule(ICorDebugAppDomain *pAppDomain,  ICorDebugModule
*pModule)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pModule	Pointer to the module that was loaded.

LogMessage

LogMessage is called when a managed thread calls the Log class in the System.Diagnostics package to log an event.

```
HRESULT LogMessage(ICorDebugAppDomain *pAppDomain, ICorDebugThread
*pThread, LONG lLevel, WCHAR pLogSwitchName[], WCHAR pMessage[])
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	lLevel	The level for the Log message.
in	pLogSwitchName	Array of characters representing the log switch name.
in	pMessage	Array of characters representing the log message.

LogSwitch

LogSwitch is called when a managed thread calls the **LogSwitch** class in the System.Diagnostics package to log an event.

```
HRESULT LogSwitch(ICorDebugAppDomain *pAppDomain, ICorDebugThread
*pThread, LONG lLevel, ULONG ulReason, WCHAR pLogSwitchName[], WCHAR
pParentName[])
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	lLevel	The level for the Log message.
in	ulReason	The log switch reason.
in	pLogSwitchName	Array of characters representing the log switch name.
in	pParentName	Array of characters representing the name of the parent of the log switch.

NameChange

NameChange is called if either an AppDomain's or Thread's name changes.

```
HRESULT NameChange(ICorDebugAppDomain *pAppDomain,
                   ICorDebugThread *pThread)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.

StepComplete

Notifies the debugger when an execution step completes in the debuggee. The stepper may be used to continue stepping if desired (except for TERMINATE reasons.)

```
HRESULT StepComplete(ICorDebugAppDomain *pAppDomain,  ICorDebugThread
*pThread, ICorDebugStepper *pStepper, CorDebugStepReason reason)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pThread	Pointer to the thread object that represents the thread that generated the event.
in	pStepper	Pointer to the stepper object that represents the stepping process.
in	reason	Type of step complete.

UnloadAssembly

Notifies the debugger when an assembly is unloaded. The assembly should not be used after this point.

```
HRESULT UnLoadAssembly(ICorDebugAppDomain *pAppDomain,
ICorDebugAssembly *pAssembly)
```

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pAssembly	Pointer to the assembly object that represents the assembly that generated the event.

UnloadClass

Notifies the debugger when the runtime is about to unload a class. The class should not be referenced after this point. This callback only occurs if ClassLoading has been enabled for the class's module.

HRESULT UnloadClass(ICorDebugAppDomain *pAppDomain, ICorDebugClass *pClass)

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pClass	Pointer to the class that was unloaded.

UnloadModule

Notifies the debugger that a module has been unloaded. The module should not be used after this point.

HRESULT UnloadModule(ICorDebugAppDomain *pAppDomain, ICorDebugModule *pModule)

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pModule	Pointer to the module that was unloaded.

UpdateModuleSymbols

UpdateModuleSymbols is called when an NGWS module's symbols have changed. This is a debugger's chance to update its view of a module's symbols, typically by calling ISymUnmanagedReader::UpdateSymbolStore or ISymUnmanagedReader::ReplaceSymbolStore.

HRESULT UpdateModuleSymbols(ICorDebugAppDomain *pAppDomain, ICorDebugModule *pModule, IStream *pSymbolStream)

In/ Out	Parameter	Description
in	pAppDomain	Pointer to the application domain object that represents the application domain that generated the event.
in	pModule	Pointer to the module
in	pSymbolStream	Pointer to the new symbol stream.

5.28 ICorDebugModule : IUnknown

The ICorDebugModule interface provides methods that return information about a module, such as the process to which the module belongs or the metadata for the module. ICorDebugModule also provides methods for enabling debugging in JIT-compiled code, doing Edit and Continue operations, etc.

The debugger obtains an ICorDebugModule object by calling ICorDebugClass::GetModule, ICorDebugFunction::GetModule, or ICorDebugProcess::GetModuleFromToken.

CreateBreakpoint

Not Implemented In-Process

Creates a breakpoint that will be triggered when any code in the module is executed.

Note: *This method is not yet implemented.*

```
HRESULT CreateBreakpoint (ICorDebugModuleBreakpoint **ppBreakpoint)
```

In/ Out	Parameter	Description
out	ppBreakpoint	Pointer to pointer to an object that represents the module breakpoint.

EnableClassLoadCallbacks

Not Implemented In-Process

Controls whether or not ClassLoad callbacks are called for the particular module. ClassLoad callbacks are off by default.

```
HRESULT EnableClassLoadCallbacks (BOOL bClassLoadCallbacks)
```

In/ Out	Parameter	Description
in	bClassLoadCallbac ks	TRUE to enable ClassLoad callbacks.

EnableJITDebugging

Not Implemented In-Process

EnableJITDebugging controls whether the JITer preserves mapping information between the IL version of a function and the JIT-compiled version for functions in the module. Turning this on will also disable certain optimizations in the code that the JITer generates. If bAllowJitOpts is true, then the JITer will generate code with certain (JIT-specific) optimizations

JIT debugging is enabled by default for all modules loaded when the debugger is active. Programmatically enabling/disabling these settings will override global settings.

```
HRESULT EnableJITDebugging(BOOL bJITDebugging, BOOL bAllowJitOpts)
```

In/ Out	Parameter	Description
in	bJITDebugging	TRUE to enable JIT debugging.
in	bAllowJitOpts	TRUE to enable JIT optimizations.

GetAssembly

Returns the assembly that contains this module.

```
HRESULT GetAssembly(ICorDebugAssembly **ppAssembly)
```

In/ Out	Parameter	Description
out	ppAssembly	Pointer to a pointer to the assembly object that contains the module.

GetBaseAddress

Returns the base address of the module.

```
HRESULT GetBaseAddress(CORDB_ADDRESS *pAddress)
```

In/ Out	Parameter	Description
out	pAddress	Pointer to the base address of the module.

GetClassFromToken

Returns a class object (**ICorDebugClass**) for a given class metadata token.

```
HRESULT GetClassFromToken(mdTypeDef typedef, ICorDebugClass **ppClass)
```

In/ Out	Parameter	Description
in	typeDef	Metadata typedef token.
out	ppClass	Pointer to pointer to an object that represents the class corresponding to the typedef token.

GetEditAndContinueSnapshot

Not Implemented In-Process

This method produces a snapshot of the running process. This snapshot can then be fed into the compiler to guarantee the same token values are returned by the metadata during compilation, to find the address where new static data should go, etc. These changes are committed using ICorDebugProcess.

```
HRESULT GetEditAndContinueSnapshot(ICorDebugEditAndContinueSnapshot
**ppEditAndContinueSnapshot)
```

In/ Out	Parameter	Description
out	ppEditAndContinueSnapshot	Pointer to pointer to an Edit & Continue object for edits.

GetFunctionFromRVA

Returns a function object (**ICorDebugFunction**) from the relative address of the function in the module.

Note: *This method is not yet implemented.*

```
HRESULT GetFunctionFromRVA(CORDB_ADDRESS rva, ICorDebugFunction
**ppFunction)
```

In/ Out	Parameter	Description
in	rva	Relative address of the function in the module.
out	ppFunction	Pointer to pointer to an object that represents the function.

GetFunctionFromToken

Returns a function object (**ICorDebugFunction**) for a given function metadata token. Returns CORDBG_E_FUNCTION_NOT_IL if called with a methodDef that does not refer to an IL method.

```
HRESULT GetFunctionFromToken(mdMethodDef methodDef, ICorDebugFunction
**ppFunction)
```

In/ Out	Parameter	Description
in	methodDef	Metadata reference for the function member.
out	ppFunction	Pointer to pointer to an object that represents the function corresponding to the member reference.

GetGlobalVariableValue

Returns a value object for the given global variable.

HRESULT GetGlobalVariableValue(mdFieldDef fieldDef,

ICorDebugValue **ppValue)

In/ Out	Parameter	Description
in	fieldDef	Metadata reference for the global variable.
out	ppValue	Pointer to pointer to an object that represents the value of the global variable.

GetMetaDataInterface

This method returns a metadata interface pointer that can be used to examine the metadata for this module.

```
HRESULT GetMetaDataInterface(REFIID riid, IUnknown **ppObj)
```

In/ Out	Parameter	Description
in	riid	The REFIID of the metadata interface.
out	ppObj	Pointer to pointer to the metadata interface object.

GetName

Returns the name of the module.

```
HRESULT GetName(ULONG32 cchName, ULONG32 *pcchName, WCHAR szName[])
```

In/ Out	Parameter	Description
in	cchName	The allocated size of string buffer.
out	pcchName	The number of characters available for return. No more than cchName are actually returned in the buffer.
out	szName[]	The string buffer.

GetProcess

Returns a process object (**ICorDebugProcess**) for the process to which this module belongs.

```
HRESULT GetProcess(ICorDebugProcess **ppProcess)
```

In/ Out	Parameter	Description
out	ppProcess	Pointer to pointer to an object that represents a

		debuggee process.

GetSize

Returns the size, in bytes, of the module.

HRESULT GetSize(ULONG32 *pcBytes)

In/ Out	Parameter	Description
out	pcBytes	The size, in bytes, of the module.

GetToken

Returns the token for the module table entry for this object. The token may then be passed to the metadata import APIs.

`HRESULT GetToken(mdModule *pToken)`

In/ Out	Parameter	Description
out	pToken	Pointer to the token for the module table entry for the module.

IsDynamic

If this is a dynamic module, IsDynamic sets *pDynamic to true, otherwise sets *pDynamic to false.

HRESULT IsDynamic(BOOL *pDynamic)

In/ Out	Parameter	Description
out	pDynamic	If this is a dynamic module, IsDynamic sets *pDynamic to true, otherwise sets *pDynamic to false.

5.29 ICorDebugModuleBreakpoint : ICorDebugBreakpoint

Not Implemented In-Process

This interface provides methods that return information about module breakpoints. An ICorDebugModuleBreakpoint object is created by calling ICorDebugModule::CreateBreakpoint.

GetModule

The debugger calls this method to get the module at which the breakpoint occurred.

```
HRESULT GetModule(ICorDebugModule **ppModule)
```

In/ Out	Parameter	Description
out	ppModule	Pointer to pointer to the object that represents the module at which the breakpoint occurred.

5.30 ICorDebugModuleEnum : ICorDebugEnum

The ICorDebugModuleEnum interface provides methods for enumerating module objects.

The debugger obtains an ICorDebugModuleEnum object by calling ICorDebugProcess::EnumerateModules on an object that represents a debuggee process.

Next

This method is used to retrieve module objects. The number of module objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugModule *modules[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of module objects requested to be retrieved.
out	modules[]	Array of pointers to module objects that is retrieved.
out	pceltFetched	The pointer to the number of actual values fetched.

5.31 ICorDebugNativeFrame : ICorDebugFrame

This interface provides methods to access information about frames running native managed or JIT-compiled managed code. Information returned includes the value of the instruction pointer associated with the stack frame and the value of hardware registers. This interface also provides a method to set the value of the instruction pointer.

The debugger obtains an ICorDebugNativeFrame object by calling ICorDebugChain::EnumerateChains, which returns an enumerator for stack frames, and then calling the methods defined on ICorDebugFrameEnum to enumerate the stack frames.

CanSetIP

Not Implemented In-Process

CanSetIP attempts to determine if it is safe to set the instruction pointer to the given native offset. If this returns S_OK, then executing SetIP will result in a safe, correct, continued execution. If CanSetIP returns anything else, SetIP can still be invoked, but continued, correct execution of the debuggee cannot be guaranteed. See the "Debugging Services" specification for details of the conditions under which the instruction pointer may be set.

```
HRESULT CanSetIP(ULONG32 nOffset)
```

In/ Out	Parameter	Description
in	nOffset	Native offset to which instruction pointer should be set.

GetIP

Returns the byte offset into the native code from the start of the function for this stack frame. If this stack frame is active, this address is the next instruction to execute. If this stack frame is not active, this is the next instruction to execute when the stack frame is reactivated.

```
HRESULT GetIP(ULONG32 *pnOffset)
```

In/ Out	Parameter	Description
out	pnOffset	Pointer to the offset of native code from the start of the function.

GetLocalDoubleRegisterValue

Return the value of a local variable or argument stored in a register pair of the native frame.

```
HRESULT GetLocalDoubleRegisterValue (CorDebugRegister highWordReg,
CorDebugRegister lowWordReg, ULONG cbSigBlob, PCCOR_SIGNATURE
pvSigBlob, ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	highWordReg	The register identifier for the high word register.
in	lowWordReg	The register identifier for the low word register.
in	cbSigBlob	Count in bytes of the signature blob.
in	pvSigBlob	Pointer to the signature blob.
out	ppValue	Return value of the variable.

GetLocalMemoryRegisterValue

Return the value of a local variable which is stored half in a register and half in memory.

`HRESULT GetLocalMemoryRegisterValue(CORDB_ADDRESS highWordAddress, CorDebugRegister lowWordRegister, ULONG cbSigBlob, PCCOR_SIGNATURE pvSigBlob, ICorDebugValue **ppValue)`

In/ Out	Parameter	Description
in	highWordAddress	Address for high word.
in	lowWordRegister	Register specifier for low word.
in	cbSigBlob	Count in bytes of the signature blob.
in	pvSigBlob	Pointer to the signature blob.
out	ppValue	Return value of the variable.

GetLocalMemoryValue

Return the value of a local variable stored at the given address.

`HRESULT GetLocalMemoryValue(CORDB_ADDRESS address, ULONG cbSigBlob, PCCOR_SIGNATURE pvSigBlob, ICorDebugValue **ppValue)`

In/ Out	Parameter	Description
in	address	Local variable stored at the given address.
in	cbSigBlob	Count in bytes of the signature blob.
in	pvSigBlob	Pointer to the signature blob.
out	ppValue	Return value of the variable.

GetLocalRegisterMemoryValue

Return the value of a local variable which is stored half in a register and half in memory.

`HRESULT GetLocalRegisterMemoryValue(CorDebugRegister highWordReg, CORDB_ADDRESS lowWordAddress, ULONG cbSigBlob, PCCOR_SIGNATURE pvSigBlob, ICorDebugValue **ppValue)`

In/ Out	Parameter	Description
in	highWordReg	Register specifier for high word.
in	lowWordAddress	Address for low word.
in	cbSigBlob	Count in bytes of the signature blob.

in	pvSigBlob	Pointer to the signature blob.
out	ppValue	Return value of the variable.

GetLocalRegisterValue

Returns the value of a local variable or argument stored in a register of a native frame.

```
HRESULT GetLocalRegisterValue(CorDebugRegister reg, ULONG cbSigBlob,
PCCOR_SIGNATURE pvSigBlob, ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	reg	The register identifier.
in	cbSigBlob	Count in bytes of the signature blob.
in	pvSigBlob	Pointer to the signature blob.
out	ppValue	Pointer to pointer to an object that represents the value of the register.

GetRegisterSet

Returns the register set for this frame.

```
HRESULT GetRegisterSet(ICorDebugRegisterSet **ppRegisters)
```

In/ Out	Parameter	Description
out	ppRegisters	Pointer to a pointer to a register set object for this frame.

SetIP

Not Implemented In-Process

Sets the instruction pointer to the native code at the given offset. Note that this is an inherently dangerous thing to do. The debugger (or debugger user) is responsible for adjusting the local state of the function so that it remains consistent. See the "Debugging Services" specification for details of the conditions under which the instruction pointer may be set.

```
HRESULT SetIP(ULONG32 nOffset)
```

In/ Out	Parameter	Description
in	nOffset	The offset of native code from the start of the function.

5.32 ICorDebugObjectValue : ICorDebugValue

The ICorDebugObjectValue interface provides methods that return information about an object that represents an object value. Information returned includes the class of the object and value of fields of the object. Objects are not necessarily heap values because they may be value objects. You must do a separate QI to access heap functionality.

GetClass

Returns the class of the object in the value. The object must not be null.

```
HRESULT GetClass(ICorDebugClass **ppClass)
```

In/ Out	Parameter	Description
out	ppClass	Pointer to pointer to an object that represents the class of the object in the value.

GetContext

Returns the runtime context for the object.

Note: *This method is not yet implemented.*

```
HRESULT GetContext(ICorDebugContext **ppContext)
```

In/ Out	Parameter	Description
out	ppContext	Pointer to pointer to an object that represents the runtime context for this object.

GetFieldValue

Returns a value for the given field in the given class. The class must be on the class hierarchy of the object's class, and the field must be a field of that class.

```
HRESULT GetFieldValue(ICorDebugClass *pClass, mdFieldDef fieldDef,
ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	pClass	Pointer to an object that represents the class of the given field.
in	fieldDef	The metadata token for the field definition.
out	ppValue	Pointer to pointer to an object that represents the value of the field.

GetManagedCopy

GetManagedCopy will return an IUnknown that is a managed copy of a value class object. This can be used with COM Interop to get information about the object, like calling System.Object::ToString on it.

Returns CORDB_E_OBJECT_IS_NOT_COPYABLE_VALUE_CLASS if the class of this object is not a value class.

```
HRESULT GetManagedCopy(IUnknown **ppObject)
```

In/ Out	Parameter	Description
out	ppObject	Pointer to an IUnknown that is a managed copy of a value class object.

GetVirtualMethod

Returns the most derived function for the given reference on this object.

Note: *This method is not yet implemented.*

```
HRESULT GetVirtualMethod(mdMemberRef memberRef, ICorDebugFunction
**ppFunction)
```

In/ Out	Parameter	Description
in	memberRef	The metadata token member reference.
out	ppFunction	Pointer to pointer to an object that represents the most derived function for the given reference.

IsValueClass

Returns true if the class of this object is a value class.

```
HRESULT IsValueClass(BOOL *pbIsValueClass)
```

In/ Out	Parameter	Description
in	pbIsValueClass	Set to TRUE if this is actually a value class.

SetFromManagedCopy

Not Implemented In-Process

SetFromManagedCopy will update a object's contents given a managed copy of the object. This can be used after using GetManagedCopy to update an object with a changed version.

SetFromManagedCopy returns CORDB_E_OBJECT_IS_NOT_COPYABLE_VALUE_CLASS if the class of this object is not a value class.

```
HRESULT SetFromManagedCopy(IUnknown *pObject)
```

In/ Out	Parameter	Description
in	pObject	Managed copy of the value class to update the object's contents with.

5.33 ICorDebugObjectEnum : ICorDebugEnum

The ICorDebugObjectEnum interface provides methods for enumerating managed objects.

The debugger obtains an ICorDebugObjectEnum object by calling ICorDebugProcess::EnumerateObjects.

Next

This method is used to retrieve managed objects. The number of objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, CORDB_ADDRESS objects[], ULONG *pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of objects requested to be retrieved.
out	objects	Array of pointers to objects that are retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.34 ICorDebugProcess : ICorDebugController

The ICorDebugProcess interface provides methods that return information about a debuggee process.

The debugger obtains an ICorDebugProcess by calling ICorDebug::EnumerateProcesses and then enumerating the debuggee process objects, or by calling ICorDebug::GetProcess with the Win32 process ID of a debuggee process.

The debugger must wait for the ExitProcess callback before releasing the ICorDebugProcess and ICorDebug interfaces.

ClearCurrentException

Not Implemented In-Process

Clears the current unmanaged exception on the given thread. Call this before calling Continue when a thread has reported an unmanaged exception that should be ignored by the debuggee.

```
HRESULT ClearCurrentException(DWORD threadId)
```

In/ Out	Parameter	Description
in	threadId	Thread Id of the thread for which the unmanaged exception needs to be cleared.

EnableLogMessages

Not Implemented In-Process

Enables sending of log messages to the debugger.

```
HRESULT EnableLogMessages(BOOL fOnOff)
```

In/ Out	Parameter	Description
in	fOnOff	If TRUE, enables sending of log messages from the debuggee to the debugger.

EnumerateAppDomains

Returns an enumerator object (**ICorDebugAppDomainEnum**) for all application domains in the debuggee process.

```
HRESULT EnumerateAppDomains(ICorDebugAppDomainEnum **ppAppDomains)
```

In/ Out	Parameter	Description
out	ppAppDomains	Pointer to pointer to an enumerator object for all application domains in the debuggee process.

EnumerateObjects

Not Implemented In-Process

Returns an enumerator object (**ICorDebugObjectEnum**) for all managed objects in the debuggee process. This should only be called when the process is in a stopped or synchronized state.

> **Note:** *This method is not yet implemented, and is tentative.*

```
HRESULT EnumerateObjects(ICorDebugObjectEnum **ppObjects)
```

In/ Out	Parameter	Description
out	ppObjects	Pointer to pointer to an enumerator for all managed objects in the debuggee process.

GetHandle

Returns the handle for the debuggee process.

```
HRESULT GetHandle(HANDLE *phProcessHandle)
```

In/ Out	Parameter	Description
out	phProcessHandle	Pointer to the handle for the process

GetID

Returns the Win32 process ID for the debuggee process.

```
HRESULT GetID(DWORD *pdwProcessId)
```

In/ Out	Parameter	Description
out	pdwProcessId	Pointer to the Win32 process ID for the process

GetObject

Returns the runtime process object.

Note: *This method is not yet implemented.*

```
HRESULT GetObject(ICorDebugObjectValue **ppObject)
```

In/ Out	Parameter	Description
out	ppObject	Pointer to pointer to an object that represents the runtime process object.

GetThread

Returns the **ICorDebugThread** given the Win32 thread ID.

Note that eventually there will not be a one-to-one correspondence between Win32 threads and the runtime threads; so, this entry point will go away.

```
HRESULT GetThread(DWORD dwThreadId, ICorDebugThread **ppThread)
```

In/ Out	Parameter	Description
in	dwThreadId	Win32 thread ID.
out	ppThread	Pointer to pointer to an object that represents a debuggee thread that was created by the runtime.

GetThreadContext

Returns the context for the given thread. The debugger should call this function rather than the Win32 GetThreadContext, because the thread may actually be in "hijacked" state where its context has been temporarily changed. Note that the threadID argument must be for an *unmanaged* thread.

```
HRESULT GetThreadContext(DWORD threadID, ULONG32 contextSize, BYTE
context[])
```

In/ Out	Parameter	Description
in	threadID	Win32 thread ID of thread.
in	contextSize	Size of the context[] buffer.
out	context[]	Context structure for the current platform.

IsOSSuspended

Returns whether or not the thread has been suspended as part of the debugger logic of stopping the process, i.e., it has had its Win32 suspend count incremented by one. The debugger may want to take this into account if shows the user the OS suspend count of the thread.

> This method only makes sense in the context of unmanaged debugging—during managed debugging threads are not OS suspended (they are cooperatively suspended).

```
HRESULT IsOSSuspended (DWORD threadID, BOOL *pbSuspended)
```

In/ Out	Parameter	Description
in	threadID	Win32 thread ID of thread.
in	pbSuspended	Pointer to boolean which is TRUE if thread has been suspended.

IsTransitionStub

Not Implemented In-Process

Tests whether an address is inside of a transition stub which will cause a transition to managed code. This can be used by unmanaged stepping code to decide when to return stepping control to a managed stepper.

Note that, tentatively, these stubs may also be able to be identified ahead of time by looking at information in the PE file.

```
HRESULT IsTransitionStub(CORDB_ADDRESS address,  BOOL
*pbTransitionStub)
```

In/ Out	Parameter	Description

in	address	Address of the function.
out	pbTransitionStub	Pointer to a Boolean that is TRUE if the address is inside of a transition stub.

ModifyLogSwitch

Not Implemented In-Process

Modifies the specified switch's severity level.

```
HRESULT ModifyLogSwitch(WCHAR *pstrLogSwitchName, LONG lLevel)
```

In/ Out	**Parameter**	**Description**
in	pstrLogSwitchName	Pointer to a string that is the name of the log switch.
in	lLevel	The new log switch levelI.

ReadMemory

Reads memory from the process. Any debugger patches will be automatically removed.

```
HRESULT ReadMemory(CORDB_ADDRESS address, DWORD size, BYTE buffer[],
DWORD *read)
```

In/ Out	**Parameter**	**Description**
in	address	Address at which memory is to be read.
in	size	Size of buffer[].
out	buffer	Buffer for returning data read from memory.
out	read	Number of bytes actually read.

SetThreadContext

Not Implemented In-Process

Sets the context for the given thread. The debugger should call this function rather than the Win32 SetThreadContext because the thread may actually be in a "hijacked" state where its context has been temporarily changed. The data returned is a context structure for the current platform.

This is a dangerous call which can corrupt the runtime if used improperly.

```
HRESULT SetThreadContext(DWORD threadID, ULONG32 contextSize, BYTE
context[])
```

In/ Out	Parameter	Description
in	threadID	Win32 thread ID of thread.
in	contextSize	Size of the context[] buffer.
out	context[]	Context structure for the current platform.

ThreadForFiberCookie

Given a fiber cookie from the Runtime Hosting API, returns the matching ICorDebugThread. If the thread is found, returns S_OK. Returns S_FALSE otherwise.

```
HRESULT ThreadForFiberCookie(DWORD fiberCookie, IcorDebugThread
**ppThread)
```

In/ Out	Parameter	Description
in	fiberCookie	The fiber cookie.
out	ppThread	Pointer to pointer to a thread object.

WriteMemory

Not Implemented In-Process

Writes memory in the process. Any debugger patches will be automatically written behind.

```
This is a dangerous call that can corrupt the runtime if used
improperly.
HRESULT WriteMemory(CORDB_ADDRESS address, DWORD size, BYTE buffer[],
DWORD *written)
```

In/ Out	Parameter	Description
in	address	Address at which memory is to be written.
in	size	Size of buffer[].
out	buffer	Buffer containing data to be written to memory.
out	written	Number of bytes actually written.

5.35 ICorDebugProcessEnum : ICorDebugEnum

The ICorDebugProcessEnum interface provides methods for enumerating objects that represent debuggee processes.

The debugger obtains an ICorDebugProcessEnum object by calling ICorDebug::EnumerateProcesses.

Next

Used to retrieve process objects. The number of process objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugProcess *processes[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of process objects requested to be retrieved.
out	processes[]	Array of pointer to process objects that are retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.36 ICorDebugRegisterSet : IUnknown

The **ICorDebugRegisterSet** interface provides methods for getting and setting registers and contexts.

The debugger obtains a **ICorDebugRegisterSet** object by calling ICorDebugThread::GetRegisterSet.

GetRegisters

Returns an array of register values corresponding to the given mask. The registers which have their bit set in the mask will be packed into the resulting array. No room is assigned in the array for registers whose mask bit is not set. Thus, the size of the array should be equal to the number of 1's in the mask.

If an unavailable register is indicated by the mask, an indeterminate value will be returned for the corresponding register.

```
HRESULT GetRegisters(ULONG64 mask, ULONG32 regCount, CORDB_REGISTER
regBuffer[])
```

In/ Out	Parameter	Description
in	mask	Mask indicating selected registers.
in	regCount	Number of elements in the buffer regBuffer to receive the register values. If the value of regCount is too small for the number of registers indicated by the mask, the higher numbered registers will be truncated from the set. Or, if the value of regCount is too large, the unused regBuffer elements will be unmodified.
out	regBuffer	Array that will contain the returned register values.

GetRegistersAvailable

Returns a mask indicating which registers are available in the given register set. Registers may be unavailable if their value is undeterminable for the given situation.

```
HRESULT GetRegistersAvailable(ULONG64 *pAvailable)
```

In/ Out	Parameter	Description
out	pAvailable	Pointer to a word that contains a bit for each register (1 << register index), which will be 1 if the register is available or 0 if it is not.

GetThreadContext

Returns the context for the thread.

```
HRESULT GetThreadContext(ULONG32 contextSize, BYTE context[])
```

In/ Out	Parameter	Description
in	contextSize	Size of the buffer context[].
out	context[]	Context structure for the current platform.

SetRegisters

Not Implemented In-Process

Sets the value of the registers specified by the given mask. For each bit set in the mask, the corresponding register will be set from the corresponding element in regBuffer[]. Note that the correlation is by sequence, not by the position of the bit, i.e., regBuffer is “packed”; there are no elements corresponding to registers whose bit is not set. If an unavailable register is indicated by the mask, the register will not be set (although a value for that register is recognized from the regBuffer[].)

Note that setting registers this way is inherently dangerous. CorDebug makes no attempt to ensure that the runtime remains in a valid state when register values are changed. E.g., if the instruction pointer were set to point to non-managed code, the results would be unpredictable.

```
HRESULT SetRegisters(ULONG64 mask, ULONG32 regCount, CORDB_REGISTER
regBuffer[])
```

In/ Out	Parameter	Description
in	mask	Mask indicating selected registers.
in	regCount	Number of elements in the buffer regBuffer. If the value of regCount is too small for the number of registers

		indicated by the mask, the higher numbered registers will not be set. Or, if the value of regCount is too large, the extra values will be ignored.
in	regBuffer	Array containing the register values.

SetThreadContext

Not Implemented In-Process

Sets the context for the thread.

```
HRESULT SetThreadContext(ULONG32 contextSize, BYTE context[])
```

In/ Out	Parameter	Description
in	contextSize	Size of the buffer context[].
in	context[]	Context structure for the current platform.

5.37 ICorDebugReferenceValue : ICorDebugValue

ICorDebugReferenceValue applies to values which are references.

Dereference

Returns a value representing the object referenced. The resulting value is a "weak reference" which will become invalid if the object is garbage collected.

```
HRESULT Dereference(ICorDebugHeapValue **ppValue)
```

In/ Out	Parameter	Description
in	ppValue	Pointer to a pointer to the returned value representing the object referenced.

DereferenceStrong

Returns a value representing the object referenced. The resulting value is a "strong reference" which will cause the object referenced to not be collected as long as it exists.

```
HRESULT DereferenceStrong(ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
in	ppValue	Pointer to a pointer to the returned value representing the object referenced.

GetValue

Copies the value into the specified buffer. The buffer should be the appropriate size for the simple type.

```
HRESULT GetValue(CORDB_ADDRESS *pValue)
```

In/ Out	Parameter	Description
out	pValue	Pointer to the value of the reference.

IsNull

Tests whether the reference is null.

```
HRESULT IsNull (BOOL *pbNull)
```

In/ Out	Parameter	Description
out	pbNull	TRUE if the reference is null.

SetValue

Not Implemented In-Process

Copies a new value from the specified buffer. The buffer should be the appropriate size for the simple type.

```
HRESULT SetValue(CORDB_ADDRESS value)
```

In/ Out	Parameter	Description
in	value	Value of the reference.

5.38 ICorDebugStepper : IUnknown

Not Implemented In-Process

The ICorDebugStepper interface provides methods that allow a debugger to manage stepping in the debuggee.

A stepper object represents a stepping operation being performed by the debugger. Note that there can be more than one stepper per thread; for instance a breakpoint may be hit in the midst of stepping over a function, and the user may wish to start a new stepping operation inside that function. (Note that it is up to the debugger how to handle this; it may want to cancel the original stepping operation, or nest them. This API allows either behavior.)

Also, a stepper may migrate between threads if a cross-thread marshaled call is made by the EE.

This object serves several purposes. It serves as an identifier between a step command issued and the completion of that command. It also provides a central interface to encapsulate all of the stepping that can be performed. Finally, it provides a way to prematurely cancel a stepping operation.

The debugger obtains a ICorDebugStepper object by calling ICorDebugFrame::CreateStepper or ICorDebugProcess:EnumerateSteppers to obtain an enumerator for steppers, and then calling ICorDebugStepperEnum to enumerate steppers.

Deactivate

Causes a stepper to cancel the last stepping command it received. A new stepping command may then be issued.

```
HRESULT Deactivate()
```

IsActive

Determines whether the stepper is active (that is, whether it is currently stepping).

Any step action remains active until one of the following callbacks are called:

- StepComplete – the step has completed normally
- StepInToUnmanaged – an unmanaged function was stepped into
- StepOutToUnmanaged – the current function returned to unmanaged code
- StepException – an exception caused the Stepper's stack frame to be unwound

Note that a stepper is always deactivated when passed to a Step callback. A stepper may also be deactivated prematurely by calling Deactivate before a callback condition is reached.

```
HRESULT IsActive(BOOL *pbActive)
```

In/ Out	Parameter	Description
out	pbActive	Pointer to a Boolean that is TRUE if the stepper is active.

SetInterceptMask

Controls which intercept code will be stepped into by the stepper. If the bit for an interceptor is set, the stepper will complete with reason STEPPER_INTERCEPT when the given type of intercept occurs. If the bit is cleared, the intercepting code will be skipped.

Note that **SetInterceptMask** may have unforeseen interactions with **SetUnmappedStopMask** (from the user's point of view). For example, if the only visible (i.e., non internal) portion of class initialization code lacks mapping information (STOP_NO_MAPPING_INFO) and STOP_NO_MAPPING_INFO isn't set, then we'll step over the class initialization.

By default, only INTERCEPT_NONE will be used.

```
HRESULT SetInterceptMask(CorDebugIntercept mask)
```

In/ Out	Parameter	Description
in	mask	Mask that controls which intercept code will be stepped into.

SetUnmappedStopMask

Controls whether the stepper will stop in JIT-compiled code that is not mapped to IL.

If the given flag is set, then that type of unmanaged code will be stopped in. Otherwise stepping transparently continues.

It should be noted that if one doesn't use a stepper to enter a method (for example, the main() method of C++), then one won't necessarily step over prologs, etc.

By default, STOP_OTHER_UNMAPPED will be used.

```
HRESULT SetUnmappedStopMask(CorDebugUnmappedStop mask)
```

In/ Out	Parameter	Description
in	mask	Mask that controls type of unmapped code.

Step

Step is called when a thread is to be single stepped. The step will complete at the next managed instruction executed by the EE in the stepper's frame.

If bStepIn is TRUE, any function calls made during the step will be stepped into. Otherwise, they will be skipped.

If **Step** is called on a stepper which is not in managed code, the step will complete when the next managed code is executed by the thread. (If bStepIn is FALSE, it will only complete when managed code is returned to, not when it is stepped into.)

```
HRESULT Step(BOOL bStepIn)
```

In/ Out	Parameter	Description
in	bStepIn	If TRUE, any function calls made during the step will be stepped into. Otherwise, they will be skipped.

StepOut

This operation will complete after the current frame is returned normally and the previous frame is reactivated. If this is called when in unmanaged code, the step will complete when the calling managed code is returned to.

```
HRESULT StepOut()
```

StepRange

StepRange works just like Step, except it will not complete until code outside the given range is reached. This can be more efficient than stepping one instruction at a time.

Ranges are specified as a list of offset pairs [start, end) (note that the end is exclusive) from the start of the stepper's frames' code.

Ranges are relative to the IL code of a method. Call SetRangeIL(FALSE) to specify ranges relative to the native code of a method.

```
HRESULT Step(BOOL bStepIn, COR_DEBUG_STEP_RANGE ranges[], ULONG32
cRangeCount)
```

In/ Out	Parameter	Description
in	bStepIn	If TRUE, any function calls made during the step will be stepped into. Otherwise they will be skipped.
in	ranges	An array of code ranges.
in	cRangeCount	The number of elements in the ranges array.

StepRangeIL

StepRangeIL is used to set whether the ranges passed to StepRange are relative to the IL code or the native code for the method being stepped in.

By default, the range is in IL.

```
HRESULT StepRangeIL(BOOL bIL)
```

In/ Out	Parameter	Description
in	bIL	TRUE, if ranges are relative to IL.

5.39 ICorDebugStepperEnum : ICorDebugEnum

Not Implemented In-Process

The ICorDebugStepperEnum interface provides methods to enumerate stepper objects.

The debugger obtains an ICorDebugStepperEnum object by calling ICorDebugProcess:EnumerateSteppers.

Next

Used to retrieve stepper objects. The number of stepper objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugStepper *steppers[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of stepper objects requested to be retrieved.
out	steppers[]	An array of pointer to stepper objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.40 ICorDebugStringValue : ICorDebugHeapValue

The ICorDebugStringValue interface provides methods that return information about an object that represents a string value.

The debugger obtains ICorDebugStringValue objects by calling methods on various interfaces such as ICorDebugArrayValue, ICorDebugClass, ICorDebugILFrame, etc.

GetLength

Returns the length of a string.

```
HRESULT GetLength (ULONG32 *pcchString)
```

In/ Out	Parameter	Description
out	pcchString	Pointer to the length of string

GetString

Returns the contents of a string.

```
HRESULT GetString(ULONG32 cchString, ULONG32 *pcchString, WCHAR
szString[])
```

In/ Out	Parameter	Description
in	cchString	The allocated size of string buffer.
out	pcchString	The number of characters available for return. No more than cchString are actually returned in the buffer.
out	szString[]	The string buffer.

5.41 ICorDebugThread : IUnknown

The ICorDebugThread interface provides methods to obtain information about a debuggee thread, such as enumerating the stack frames for a thread, controlling execution of a debuggee thread, getting the ID of the thread, etc.

The debugger obtains an ICorDebugThread object by calling ICorDebugChain::GetThread, CreateThread, or ICorDebugProcess::EnumerateThreads. The last method returns an enumerator for thread objects that can be used to enumerate all thread objects for debuggee threads in a debuggee process.

ClearCurrentException

Not Implemented In-Process

Clears the current exception object, preventing it from being thrown. This should be called before the exception callback returns.

```
HRESULT ClearCurrentException()
```

CreateEval

Not Implemented In-Process

CreateEval creates an evaluation object which operates on the given thread. The Eval will push a new chain on the thread before doing its computation.

Note that this interrupts the computation currently being performed on the thread until the evaluation completes.

```
HRESULT CreateEval(ICorDebugEval **ppEval)
```

In/ Out	Parameter	Description
out	ppEval	Pointer to pointer to an evaluation object.

CreateStepper

Not Implemented In-Process

Creates a stepper object which operates relative to the active frame in the given thread. (Note that the frame may be running unmanaged code.) The methods defined on ICorDebugStepper must then be used to perform the actual stepping.

```
HRESULT CreateStepper(ICorDebugStepper **ppStepper)
```

In/ Out	Parameter	Description
out	ppStepper	Pointer to pointer to the created stepper object.

EnumerateChains

Returns an enumerator which will return all the stack chains in the thread, starting at the top (innermost) one. These chains represent the physical call stack for the thread.

Chain boundaries occur for several reasons:

- Managed to unmanaged and unmanaged to managed transitions.
- Context switches.
- Debuggger hijacking of user threads.

Note that in the simple case for a thread running purely managed code in a single context, there will be a one to one correspondence between threads and chains.

A debugger may want to rearrange the physical call stacks of all threads into logical call stacks. This would involve sorting all the threads' chains by their caller/callee relationships and regrouping them.

If called by the In Process Debugging API, this will cause a new stack trace to be done.

```
HRESULT EnumerateChains(ICorDebugChainEnum **ppChains)
```

In/ Out	Parameter	Description
out	ppChains	Pointer to pointer to an enumerator for all the stack chains in the thread.

GetActiveChain

Returns the active (most recent) chain on the thread, if any.

```
HRESULT GetActiveChain(ICorDebugChain **ppChain)
```

In/ Out	Parameter	Description
out	ppChain	Pointer to pointer to the active chain on the thread.

GetActiveFrame

Returns the active (most recent) frame on the thread, if any.

```
HRESULT GetActiveFrame(ICorDebugFrame **ppFrame)
```

In/ Out	Parameter	Description
out	ppFrame	Pointer to pointer to the active frame on the thread.

GetAppDomain

Returns the application domain that owns this thread.

```
HRESULT GetAppDomain(ICorDebugAppDomain **ppAppDomain)
```

In/ Out	Parameter	Description

out	ppAppDomain	Pointer to pointer to the appliction domain object for the application domain that owns the thread.

GetCurrentException

Not Implemented In-Process

Returns the exception object which is currently being thrown by the thread. This will only exist for the duration of an exception callback.

```
HRESULT GetCurrentException(ICorDebugValue **ppExceptionObject)
```

In/ Out	Parameter	Description
out	ppExceptionObject	Pointer to the pointer to the object that represents the exception object being thrown.

GetDebugState

Returns the current debug state of the thread. If the process is currently stopped, the "current debug state" represents the debug state if the process were to be continued, not the actual state.

```
HRESULT GetDebugState(CorDebugThreadState *pState)
```

In/ Out	Parameter	Description
out	pState	Pointer to the current debug state of the thread.

GetHandle

Returns a handle to the thread.

```
HRESULT GetHandle(DWORD *phThreadHandle)
```

In/ Out	Parameter	Description
out	phThreadHandle	Pointer to the handle to the thread.

GetID

Returns the Win32 thread id.

```
HRESULT GetID(DWORD *pdwThreadId)
```

In/ Out	Parameter	Description

out	pdwThreadId	Pointer to the Win32 thread ID.

GetObject

Returns the runtime thread object.

```
HRESULT GetObject(ICorDebugValue **ppObject)
```

In/ Out	Parameter	Description
out	ppObject	Pointer to pointer to an object that represents the runtime thread object.

GetProcess

Returns the process object (**ICorDebugProcess**) for the process that contains the thread.

```
HRESULT GetProcess(ICorDebugProcess **ppProcess)
```

In/ Out	Parameter	Description
out	ppProcess	Pointer to pointer to the process object for the process that contains the thread.

GetRegisterSet

Returns the register set for the active part of the thread.

```
HRESULT GetRegisterSet(ICorDebugRegisterSet **ppRegisters)
```

In/ Out	Parameter	Description
out	ppRegisters	Pointer to pointer to the register set for the active part of the thread.

GetUserState

Returns the user state of the thread, i.e, the state which it has when the program being debugged examines it.

```
HRESULT GetUserState(CorDebugUserState *pState)
```

In/ Out	Parameter	Description
out	pState	Pointer to the current user state of the thread.

SetDebugState

Not Implemented In-Process

Sets the current debug state of the thread. The "current debug state" represents the debug state if the process were to be continued, not the actual state. The normal value for this is THREAD_RUNNING. Only the debugger can affect the debug state of a thread. Debug states do not last across continues. So, if you want to keep a thread THREAD_SUSPENDed over multiple continues, you can set it once and thereafter not have to worry about it.

```
HRESULT SetDebugState(CorDebugThreadState state)
```

In/ Out	Parameter	Description
in	state	The debug state to which the thread should be set.

5.42 ICorDebugThreadEnum : ICorDebugEnum

The ICorDebugThreadEnum interface provides methods for enumerating objects that represent debuggee threads.

The debugger obtains an ICorDebugThreadEnum object by calling ICorDebugProcess::EnumerateThreads.

Next

Used to retrieve thread objects. The number of thread objects to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

```
HRESULT Next(ULONG celt, ICorDebugThread *threads[], ULONG
*pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of thread objects requested to be retrieved.
out	threads[]	An array of pointers to thread objects that is retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

5.43 ICorDebugUnmanagedCallback : IUnknown

Not Implemented In-Process

The debugger implements this callback interface and registers it with the runtime by calling **ICorDebug::SetUnManagedHandler**. The runtime calls methods defined on this interface to notify the debugger about unmanaged events in the debuggee process.

DebugEvent

DebugEvent is called when a DEBUG_EVENT is received which is not directly related to the NGWS runtime.

This callback is an exception to the rules about callbacks. When this callback is called, the process will be in the "raw" OS debug stopped state. The process will not be synchronized. The process will automatically enter the synchronized state when necessary to satifsy certain requests for information about managed code. (Note that this may result in other nested DebugEvent callbacks.)

Call ClearCurrentException on the process to ignore an exception event before continuing the process. (Causes DBG_CONTINUE to be sent on continue rather than DBG_EXCEPTION_NOT_HANDLED)

```
HRESULT DebugEvent(LPDEBUG_EVENT pDebugEvent,  BOOL fOutofBand)
```

In/ Out	Parameter	Description
in	**pDebugEvent**	Pointer to a descriptor describing the type of event about which the debugger is being notified.
in	**fOutofBand**	fOutofBand will be FALSE if the Debugging Services support interaction with the process’s managed state while the process is stopped due to this event. fOutofBand will be TRUE if interaction with the process’s managed state is impossible until the unmanaged event is continued from.

5.44 ICorDebugValue : IUnknown

The ICorDebugValue interface provides methods that provide information about values of various items in a debuggee process.

The ICorDebugValue object represents a value in the debuggee process. ICorDebugValue allows the value of an item to be retrieved or set.

In general, ownership of a value object is passed to the debugger from the API. The recipient is responsible for removing a reference from the object when finished with it.

Depending on where the value was retrieved from, the value may not remain valid after the process is resumed. So, in general, values shouldn’t be held across "continues."

The debugger obtains ICorDebugValue objects by calling methods on various interfaces such as ICorDebugArrayValue, ICorDebugClass, ICorDebugILFrame, etc.

CreateBreakpoint

Not Implemented In-Process

Creates a breakpoint that will be triggered when the value is modified.

Note: *This method is not yet implemented.*

```
HRESULT CreateBreakpoint(ICorDebugValueBreakpoint **ppBreakpoint)
```

In/ Out	Parameter	Description
out	ppBreakpoint	Pointer to a pointer to the value breakpoint object.

GetAddress

Returns the address of the value in the debuggee process. This might be useful information for the debugger to show.

If the value is at least partly in registers, 0 is returned.

```
HRESULT GetAddress(CORDB_ADDRESS *pAddress)
```

In/ Out	Parameter	Description
out	pAddress	Pointer to the address of the value in the debuggee process.

GetType

Returns the simple type of the value. If the object has a more complex runtime type, that type may be examined through the appropriate subclasses (e.g., **ICorDebugObjectValue** can get the class of an object).

```
HRESULT GetType (CorElementType *pType)
```

In/ Out	Parameter	Description
out	pType	The type of the value.

GetSize

Returns the size in bytes of the value. Note that for reference types this will be the size of the pointer rather than the size of the object.

```
HRESULT GetSize(ULONG32 *pSize)
```

In/ Out	Parameter	Description
out	pSize	The size in bytes of the value.

5.45 ICorDebugValueBreakpoint : ICorDebugBreakpoint

Not Implemented In-Process

This interface provides methods that return information about value breakpoints. An ICorDebugValueBreakpoint object is created by calling ICorDebugValue::CreateBreakpoint.

GetValue

The debugger calls this method to get the value at which the breakpoint occurred.

```
HRESULT GetValue(ICorDebugValue **ppValue)
```

In/ Out	Parameter	Description
out	ppValue	Pointer to pointer to the object that represents the value at which the breakpoint occurred.

5.46 ICorDebugValueEnum : ICorDebugEnum

The **ICorDebugValueEnum** interface provides methods for enumerating values.

The debugger obtains an **ICorDebugValueEnum** object by calling ICorDebugILFrame::EnumerateLocalVariables or ICorDebugILFrame::EnumArguments.

Next

This method is used to retrieve values. The number of values to be retrieved is passed as one of the parameters. The enumeration pointer is incremented by that amount.

In the event of a failure, the "current" pointer will be moved one element beyond that indicated by *pceltFetched. Thus, if one were to repeatedly ask for one element to iterate through the array, you would iterate exactly ICorDebugEnum::GetCount times, regardless of individual failures.

```
HRESULT Next(ULONG celt, ICorDebugValue *values[], ULONG *pceltFetched)
```

In/ Out	Parameter	Description
in	celt	The number of values requested to be retrieved.
out	values	Array of pointers to values that are retrieved.
out	pceltFetched	Pointer to the number of actual values fetched.

6 Debug Type Definitions

6.1 COR_DEBUG_STEP_RANGE

This type definition is used by **ICorDebugStepper::StepRange**.

```
typedef struct
{
   ULONG32 startOffset, endOffset;
} COR_DEBUG_STEP_RANGE;
```

6.2 COR_IL_MAP

This typedef is used by **ICorDebugEditAndContinueSnapshot**.

```
typedef struct _COR_IL_MAP
{
   ULONG32 oldOffset; // Old IL offset relative to beginning of
function
   ULONG32 newOffset; // New IL offset relative to beginning of
function
} COR_IL_MAP;
```

6.3 CorDebugChainReason

This type definition is used by **ICorDebugThread::GetReason**.

```
typedef enum CorDebugChainReason
{
   CHAIN_NONE              = 0x000,
   CHAIN_CLASS_INIT        = 0x001,
   CHAIN_EXCEPTION_FILTER  = 0x002,
   CHAIN_SECURITY          = 0x004,
   CHAIN_CONTEXT_POLICY    = 0x008,
   CHAIN_INTERCEPTION      = 0x010,
   CHAIN_PROCESS_START     = 0x020,
   CHAIN_THREAD_START      = 0x040,
   CHAIN_ENTER_MANAGED     = 0x080,
   CHAIN_ENTER_UNMANAGED   = 0x100,
   CHAIN_DEBUGGER_EVAL     = 0x200,
   CHAIN_CONTEXT_SWITCH    = 0x400,
   CHAIN_FUNC_EVAL         = 0x800,
} CorDebugChainReason;
```

6.4 CorDebugCreateProcessFlags

This type definition is used by **ICorDebug::CreateProcess**

```
typedef enum CorDebugCreateProcessFlags
{
    DEBUG_NO_SPECIAL_OPTIONS        = 0x0000,
   DEBUG_ENABLE_EDIT_AND_CONTINUE  = 0x0001,
} CorDebugCreateProcessFlags;
```

6.5 **CorDebugIlToNativeMappingTypes**

ICorDebugCode:: GetILToNativeMapping returns an array of COR_DEBUG_IL_TO_NATIVE_MAP structures. In order to convey that certain ranges of native instructions correspond to special regions of code (for example, the prolog), an entry in the array may have it's ilOffset field set to one of these values.

typedef enum CorDebugIlToNativeMappingTypes

```
   {
        NO_MAPPING = -1,
        PROLOG     = -2,
        EPILOG     = -3
   } CorDebugIlToNativeMappingTypes;
```

6.6 COR_DEBUG_IL_TO_NATIVE_MAP

This type definition is used by **ICorDebugCode::GetILToNativeMapping**.

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

6.7 **CorDebugIntercept**

This type definition is used by **ICorDebugStepper::SetInterceptMask**.

```
typedef enum CorDebugIntercept
{
   INTERCEPT_NONE             = 0x0,
   INTERCEPT_CLASS_INIT       = 0x01,
   INTERCEPT_EXCEPTION_FILTER = 0x02,
   INTERCEPT_SECURITY         = 0x04,
   INTERCEPT_CONTEXT_POLICY   = 0x08,
   INTERCEPT_INTERCEPTION     = 0x10,
   INTERCEPT_ALL              = 0xffff
} CorDebugIntercept;
```

6.8 CorDebugMappingResult

This type definition is used by **ICorDebugILFrame::GetIP**.

```
typedef enum CorDebugMappingResult
{
   MAPPING_PROLOG             = 0x1,
   MAPPING_EPILOG             = 0x2,
   MAPPING_NO_INFO            = 0x4,
   MAPPING_UNMAPPED_ADDRESS   = 0x8,
   MAPPING_EXACT              = 0x10,
   MAPPING_APPROXIMATE        = 0x20
};
```

Mapping Result	**Description**
MAPPING_EXACT	The IP is correct. Either the frame is interpreted or there is an exact IL map for the function.
MAPPING_APPROXIMATE	The IP was successfully mapped but may only be approximate.
MAPPING_UNMAPPED_ADDRESS	Although there is mapping information for the function, the current address is not mappable to IL. An IP of 0 is returned.
MAPPING_PROLOG	The native code is in the prolog, so an IP of 0 is returned.
MAPPING_EPILOG	The native code is in the epilog, so the last IP of the method is returned.
MAPPING_NO_INFO	No mapping information is available for the method, so an IP of 0 is returned.

6.9 CorDebugRegister

This type definition is used by ICorDebugRegisterSet.

```
typedef enum CorDebugRegister
{
   // Registers potentially available on all architectures. Note that
   these overlap with the architecture- // specific registers.
   REGISTER_INSTRUCTION_POINTER = 0,
   REGISTER_STACK_POINTER,
   REGISTER_FRAME_POINTER,
   // X86 registers
   REGISTER_X86_EIP = 0,
   REGISTER_X86_ESP,
   REGISTER_X86_EBP,

   REGISTER_X86_EAX,
   REGISTER_X86_ECX,
   REGISTER_X86_EDX,
   REGISTER_X86_EBX,

   REGISTER_X86_ESI,
   REGISTER_X86_EDI,

   REGISTER_X86_FPSTACK_0,
   REGISTER_X86_FPSTACK_1,
   REGISTER_X86_FPSTACK_2,
   REGISTER_X86_FPSTACK_3,
   REGISTER_X86_FPSTACK_4,
   REGISTER_X86_FPSTACK_5,
   REGISTER_X86_FPSTACK_6,
   REGISTER_X86_FPSTACK_7,

   // other architectures here
} CorDebugRegister;
```

6.10 CorDebugStepReason

This type definition is used by **ICorDebugManagedCallback::StepComplete**.

```
typedef enum CorDebugStepReason
{
   STEP_NORMAL,
   STEP_RETURN,
   STEP_CALL,
   STEP_EXCEPTION_FILTER,
   STEP_EXCEPTION_HANDLE,
   STEP_INTERCEPT,
   STEP_EXIT
} CorDebugStepReason;
```

Reason	Description
STEP_NORMAL	Stepping completed normally in the same function.
STEP_RETURN	Stepping continued normally after the function returned.

STEP_CALL	Stepping continued normally at the start of a newly called function.
STEP_EXCEPTION_FILTER	Control passed to an exception filter after an exception was thrown.
STEP_EXCEPTION_HANDLED	Control passed to an exception handler after an exception was thrown.
STEP_INTERCEPT	Control passed to an interceptor.
STEP_EXIT	Thread exited before the step completed. No more stepping can be performed with the stepper.

6.11 CorDebugThreadState

A thread's DebugState determines whether the debugger lets a thread run or not. Possible states are determined by the following table:

State	**Description**
RUN	Thread runs freely unless a debug event occurs.
SUSPEND	Thread cannot run.

Note: We allow for message pumping via a callback provided to the hosting API. Thus, we don't need an "interrupted" state here.

```
typedef enum CorDebugThreadState
{
   THREAD_RUN,
   THREAD_SUSPEND
} CorDebugThreadState;
```

6.12 CorDebugUnmappedStop

This type definition is used by **ICorDebugStepper:SetUnmappedStopMask**.

```
typedef enum CorDebugUnmappedStop
{
   STOP_NONE            = 0x0,
   STOP_PROLOG          = 0x01,
   STOP_EPILOG          = 0x02,
   STOP_NO_MAPPING_INFO = 0x04,
   STOP_OTHER_UNMAPPED  = 0x08,
   STOP_UNMAPPED        = 0x10,
   STOP_ALL             = 0xffff
} CorDebugUnmappedStop;
```

6.13 CorDebugUserState

This type definition is used by **ICorDebugThread::GetUserState**.

```
typedef enum CorDebugUserState
{
   USER_STOP_REQUESTED     = 0x01,
   USER_SUSPEND_REQUESTED  = 0x02,
   USER_BACKGROUND         = 0x04,
   USER_UNSTARTED          = 0x08,
   USER_STOPPED            = 0x10,
   USER_WAIT_SLEEP_JOIN    = 0x20,
   USER_SUSPENDED          = 0x40
} CorDebugUserState;
```

6.14 LoggingLevelEnum

```
typedef enum LoggingLevelEnum
{
   LTraceLevel0 = 0,
   LTraceLevel1,
   LTraceLevel2,
   LTraceLevel3,
   LTraceLevel4,
   LStatusLevel0 = 20,
   LStatusLevel1,
   LStatusLevel2,
   LStatusLevel3,
   LStatusLevel4,
   LWarningLevel = 40,
   LErrorLevel = 50,
   LPanicLevel = 100
} LoggingLevelEnum;
```

6.15 LogSwitchCallReason

```
typedef enum LogSwitchCallReason
{
   SWITCH_CREATE,
   SWITCH_MODIFY,
   SWITCH_DELETE
} LogSwitchCallReason;
```

7 HRESULT DEFINITIONS

Please note that HRESULTs are divded into errors (CORDBG_E_*) and status results (CORDBG_S_*).

Code	Description
CORDBG_E_BAD_THREAD_STATE	The state of the thread is invalid. For example, if you try and do a stack trace on a dead thread.
CORDBG_E_CANT_CHANGE_JIT_SETTING_FOR_ZAP_MODULE	You can't change JIT settings for ZAP modules.
CORDBG_E_CANT_SET_IP_INTO_FINALLY	SetIP is not possible because it would move EIP from outside of an exception handling finally clause to a point inside of one.
CORDBG_E_CANT_SETIP_INTO_OR_OUT_OF_FILTER	SetIP can't leave or enter a filter.
CORDBG_E_CANT_SET_IP_OUT_OF_FINALLY	While unwinding for an exception (because the EE calls the finally during the unwind). If you stepped into the code b/c you walked off the end of the try, then you can leave the finally.
CORDBG_E_CLASS_NOT_LOADED	A class was not loaded.
CORDBG_E_CODE_NOT_AVAILABLE	For whatever reason, the code is unavailable.
CORDBG_E_FIELD_NOT_AVAILABLE	A field in a class is not available probably because the runtime optimized it away.
CORDBG_E_FUNC_EVAL_BAD_START_POINT	Func eval can't work if we're, for example, not stopped at a GC safe point.
CORDBG_E_FUNC_EVAL_NOT_COMPLETE	If you call CordbEval::GetResult before the func eval has finished, you'll get this result.
CORDBG_E_FUNCTION_NOT_IL	Attempt to get a ICorDebugFunction for a function that is not IL.
CORDBG_E_IL_VAR_NOT_AVAILABLE	An IL variable is not available at the current native IP.
CORDBG_E_INPROC_NOT_IMPL	The inproc version of the debugging API doesn't implement this function.
CORDBG_E_INVALID_OBJECT	
CORDBG_E_NON_NATIVE_FRAME	“Native frame only” operation on a non-native frame.
CORDBG_E_NONCONTINUABLE_EXCEPTION	Tried to continue on an exception that doesn’t allow continuation. Only IL “break” instruction allows continuation.

CORDBG_E_OBJECT_IS_NOT_COPYABLE_VALUE_CLASS	This object value is no longer valid.
CORDBG_E_PROCESS_NOT_SYNCHRONIZED	Process not synchronized.
CORDBG_E_PROCESS_TERMINATED	Process was terminated.
CORDBG_E_SET_IP_NOT_ALLOWED_ON_NONLEAF_FRAME	SetIP cannot be done on any frame except the leaf frame.
CORDBG_E_SET_IP_IMPOSSIBLE	SetIP isn't allowed. For example, there is insufficient memory to perform SetIP.
CORDBG_E_STATIC_VAR_NOT_AVAILABLE	A static variable isn't available because it hasn't been initialized yet.
CORDBG_E_UNRECOVERABLE_ERROR	Unrecoverable API error.
CORDBG_S_BAD_END_SEQUENCE_POINT	Attempt to SetIP when not going to a sequence point. If both this and CORDBG_E_BAD_START_SEQUENCE_POINT are true, only this error will be reported.
CORDBG_S_INSUFFICIENT_INFO_FOR_SETIP	SetIP is possible, but the Debugging Services doesn't have enough information to fix variable locations, GC references, or anything else. Use at your own risk.
CORDBG_S_FUNC_EVAL_ABORTED	The func eval completed, but was aborted.
CORDBG_S_FUNC_EVAL_HAS_NO_RESULT	Some Func evals will lack a return value, such as those whose return type is void.
CORDBG_S_VALUE_POINTS_TO_VOID	The Debugging API doesn't support dereferencing pointers of type void.