Debug Overview

This is preliminary documentation and subject to change

Last revised: 15 August 2001

TABLE OF CONTENTS

1. INTRODUCTION 3

1.1 DESIGN GOALS 3
1.2 FURTHER INFORMATION 3

2. CLR DEBUGGING REQUIREMENTS 3

2.1 SCENARIOS 3
2.2 ENVIRONMENTS 4
2.3 API 5
2.4 FUNCTIONALITY OVERVIEW 5
2.4.1 Launching or attaching to a program 5
2.4.2 Execution control 5
2.4.3 Examining Program state 6
2.4.4 Modifying Program State 6
2.4.5 Edit and Continue 7
2.4.6 Function evaluation 8
2.4.7 Dynamic Code Injection 9

3. DEBUGGING ARCHITECTURE 9

3.1 PROCESS ARCHITECTURE 9
3.2 MANAGED CODE DEBUGGER 10
3.3 THREAD SYNCHRONIZATION 10
3.4 THE IN-PROCESS HELPER THREAD 11
3.5 INTERACTIONS WITH THE JUST IN TIME COMPILERS 11
3.6 DEBUGGING MODES 12

1. Introduction

This document describes the debugging services in the Common Language Runtime (CLR). The first part outlines the goals and requirements for the services. The second part provides an overview of the CLR debugging API. Lastly, the third part gives a picture of the architecture of the debugging services.

1.1 Design Goals

The design goals for the debugging services in the CLR are as follows:

- The debugging model should be on par with what the CLR provides. That is, the model of a program as viewed through the debugging services should be similar to that of a program as presented to the runtime for execution.
- The CLR debugging API is a low-level API. It should provide the minimal functionality necessary to 'get the job done'. The API is called by debugging packages, rather than by end-users directly. There is no need therefore to perform extensive argument checking, for example – instead, it is the caller's responsibility to perform such checks
- It should be usable by existing tools. A large investment has already been made in sophisticated tools and debugging environments and the CLR debugging services should fit in with these as easily as possible.
- Changes in the execution logic when debugging versus when not debugging should be minimized. The goal is to avoid the scenario where a user cannot reproduce some behavior when debugging is enabled, and is therefore forced to debug by more primitive means.

1.2 Further Information

Debug Reference – defines the details of the debugging API

Enabling Profiling and Debugging – describes how to control JIT-attach debugging

2. CLR Debugging Requirements

2.1 Scenarios

The CLR debugging services must operate in the following scenarios. Note that CLR does not necessarily have to enable each scenario, but it must at least inter-operate with current methods of supporting them.

Out-of-process debugging

Out-of process debugging is when the debugger is in a process other than the process being debugged. This reduces interactions between the debugger and the process being debugged, thus allowing a more accurate picture of the process.

The CLR debugging services will directly support out-of-process debugging. The services handle all communication for managed code debugging between the debugger and managed portions of the debuggee process.

Even though the CLR debugging API is used out-of-process, some of the debugging logic (e.g., thread synchronization) occurs in-process with the debuggee. This is an implementation detail that should be transparent to the debugger.

In-process debugging

The CLR debugging services support limited in-process debugging. The inspection parts of the debugging API are available to be used in-process. The execution control portions of the API are not available. See the Debug Reference specification for more information on which methods are available in-process.

Remote process debugging

Remote process debugging is when the debugger user interface is on a separate machine from the process being debugged. This can be useful if the debugger interferes with the debuggee when they are running on the same machine, due to limited resources, location dependencies, or bugs which interfere with OS operation.

The CLR debugging services do not directly support remote process debugging. A debugger based on the CLR debugging API still must exist out-of-process from the debuggee, so this solution requires a proxy process on the same machine with the debuggee.

Unmanaged code debugging

Since managed code can coexist in the same process with unmanaged (native) code, there will be many situations where the user will want to debug both kinds of code simultaneously.

The CLR debugging services do not directly support debugging of unmanaged code. However they are designed to coexist with an unmanaged code debugger by sharing the Win32 debugging facilities. Also, facilities are provided to support stepping through boundaries between managed and unmanaged code.

Furthermore, the CLR debugging services provide two options for debugging a process: managed only "soft attach", and a managed/unmanaged mode "hard attach". When soft attached, only the managed portions of a process are debugged. In contrast, when hard attached, both the managed and unmanaged portions of a process are debugged, and all Win32 debug events are exposed through the debugging services.

Mixed language environments

In component software, different components can be built with different languages. A debugger needs to understand when the code is in different languages so it displays data with the proper format, does expression evaluation with the correct syntax, etc.

The CLR debugging services do not provide any direct support for mixed language environments, since CLR has no concept of source language. A debugger's existing source mapping facilities should allow it to map a given function to the language in which it was implemented.

Multiple processes and distributed programs

A component program can consist of cooperating components running in different processes or even on different machines throughout a network. A debugger should be able to trace execution logic between processes and machines to provide a logical view of what is going on.

The CLR debugging services do not provide any direct support for multiple process debugging. Again, a debugger using the services is free to provide such support, and existing methods to do so should continue to work.

2.2 Environments

CLR debugging facilities are available on all processors and operating systems supported by CLR

2.3 API

The CLR debugging services are implemented primarily with unmanaged code. Therefore, the API is presented as a set of COM interfaces.

The debugging API relies on the metadata API to handle inspection of static program information such as classes and method type information. The debugging API relies on the symbol store API to support source level debugging for managed code debuggers.

2.4 Functionality Overview

2.4.1 Launching or attaching to a program

CLR allows you to attach to a running program or launch a process. The CLR debugging services support just-in-time debugging by allowing attachment to a program which throws an unhandled exception. However, a program which was not run as "debuggable" may have less debugging information available. There are various ways for a program to always run itself in a debuggable mode to alleviate this problem. For details see the specification Enabling Profiling and Debugging

2.4.2 Execution control

The CLR debugging services provide a number of ways to control execution of a program. These include breakpoints, single stepping, exception catching, function evaluation, and other events related to startup and shutdown of a program.

Execution control is only provided for managed code. Debuggers that wish to perform execution control in unmanaged code must implement it separately.

2.4.2.1 Breakpoints

Breakpoints can be created by specifying the code and MSIL or native offset of the location where the break should occur. The debugger will then be notified when the breakpoint is hit. Conditional breakpoints are not directly supported; a debugger can implement these by evaluating an expression in response to a breakpoint and deciding whether to inform the user of the stop or not.

2.4.2.2 Stepping

The CLR debugging services provide a wide variety of stepping functionality. A program can step by single instruction or by a range of instructions. It can either skip over function calls or step into them. It can also step out of a function. The debugger also will be notified if an exception occurs which interrupts the stepping operation.

Although the debugging services do not directly support stepping through unmanaged code, they will provide callbacks when a stepping operation reaches unmanaged code, to hand off control to the debugger. They also provide functionality that allows the debugger to determine when managed code is about to be entered from unmanaged code.

Stepping operations may cross thread boundaries if the CLR performs the marshalling. Stepping operations do not currently recognize other forms of cross-thread or cross-process stepping (such as COM or DCOM marshalling.)

Source level stepping is not directly provided by CLR. A debugger can provide this functionality by using range stepping in conjunction with its own source mapping information. The symbol store interfaces can be used to obtain source level information. See Debug Reference for a specification of the symbol store interfaces.

2.4.2.3 Exceptions

The CLR debugging services allow a debugger to catch both first chance and last (second) chance exceptions in managed code. The thrown object is available for inspection at each point.

Native exceptions in unmanaged code are not handled by the CLR (unless they propagate up to managed code.) However, the Win32 debugging services shared with the CLR debugging services can still be used to deal with unmanaged exceptions.

2.4.2.4 Program events

The CLR debugging services notify a debugger when many program events occur. These include process creation and exit, thread creation and exit, application domain creation and exit, assembly loading and unloading, module loading and unloading, and class loading and unloading. To insure good performance, class loading and unloading events can be disabled per module if desired. Events for class loading and unloading are disabled by default.

2.4.2.5 Thread control

The CLR debugging services provide APIs for suspending and resuming individual (managed) threads.

2.4.3 Examining Program state

The CLR debugging services provide a detailed means to inspect the parts of a process which are running managed code when the process is in a stopped state. A process can be inspected to get a list of physical threads.

A thread can be examined to find out its call stack. A thread's call stack is decomposed at two levels. First it is decomposed into *chains*. A chain is a contiguous logical call stack segment containing entirely managed or unmanaged stack frames. In addition, all managed call frames in a single chain share the same CLR context. If a cross-thread call has occurred within the thread, the chains provide information to allow a debugger to easily track logical call stacks across threads.

A chain can be either managed or unmanaged. Each managed chain can be further decomposed into single stack frames. Each stack frame represents one method invocation. Stack frames can be queried to obtain the code they are executing. A stack frame can also be queried to obtain arguments, the local variables, as well as the native registers.

An unmanaged chain is not composed of stack frames. Rather, an unmanaged chain simply provides the range of stack addresses that are due to unmanaged code. It is up to an unmanaged debugger to decode the unmanaged portion of the stack and provide a stack trace.

Note that CLR debugging services do not understand the concept of local variables as they exist in source code. It is up to the debugger to map between local variables and where they are allocated (either at a variable index for MSIL functions or in a register/stack location for managed native code.)

Access to global, class static, and thread local variables is also provided.

2.4.4 Modifying Program State

Although it is an inherently dangerous operation, the CLR debugging services allows a debugger to change the physical location of the instruction pointer during execution. The instruction pointer may be changed successfully under the following conditions:

- The target instruction pointer is at a sequence point. Sequence points represent statement boundaries.
- The current instruction pointer is at a sequence point.
- The target instruction pointer is not located in an exception filter.

- The target instruction pointer is not located in a catch block.
- The target instruction pointer is not located in a finally block.
- From within a catch block, the target instruction pointer is not located outside the catch block.
- The target instruction pointer must be within the same frame as the current instruction pointer.

Variables at the current instruction pointer location will be mapped to the variables at the target instruction pointer location. GC references at the target instruction pointer location will be properly initialized.

After the instruction pointer is changed, the services mark any cached stack information as invalid and refresh the information the next time it is needed. Debuggers that cache pointers to stack information (frames, chains, etc.) should refresh this information after changing the instruction pointer.

The debugger is also allowed to modify the data of a program when it is stopped. Local variables and arguments in a function that is currently running can be changed in a manner similar to how they are inspected. Also, fields of arrays and objects can be updated, as well as static fields and global variables.

2.4.5 Edit and Continue

Edit and Continue is a feature which makes it possible to be in the middle of a debugging session, edit source code, recompile the modified source, and continue the debugging session without having to rerun the executable from the start. From a functional perspective, Edit and Continue implies the ability to modify the code that is running in the debugger while preserving the rest of the run time state of the executable being debugged.

Note: Edit and Continue mode is not supported in first release of the product. However, the information in this section is retained, for information.

It is intended to support the following scenarios in a future release:

Compiler Support. This scenario allows a debugger process to request a snapshot of the metadata for the running process, and have this marshaled back to the compiler to use. The compiler will be able to use the metadata APIs to open the metadata in Edit and Continue mode. This means that no tokens will move. Hence, if the code generation is deterministic, no changes should be perceived except those made by the user. Adding new metadata and having it reflected in the running process is not part of this scenario.

Replace an Active Method. The programmer makes a change to a JIT-compiled, active method and the change is reflected. The debugger will support hijacking of the active method for the EE to know when it hits the top of the stack. The EE will perform the required stub work to swap out the method. The common allocator will throw away the old method and the JIT compiler will compile the new method.

Adding Static Data. The programmer adds a new text string for rdata inclusion and then updates the program to use ldstr. The same holds for other instructions. This requires the EE to return a valid RVA base for the snapshot call, recognize the new data, and put the new data into the scratch data area of the running process.

Adding Local Variables. New variables are added to an existing method. This requires a new standalone signature in the metadata describing the new variable. The EE and JIT compiler extend the stack frame for a method which gets a new variable (layout of the original stack must remain fixed – this is by design).

Add a Static Method to a Class, Call It. Add a static method to an existing class definition. This requires the metadata to be extended with the new method definition and the method layout to be changed to include knowledge of the new method.

Add non-Virtual Method to a Class, Call It. This is same as the previous scenario, but only allow for non-virtual, non-static method to be added to the class. This change requires the method table layout to have knowledge about the new method, and allow calls to the new method.

Add Virtual Method to a Class, Call It. This is same as the previous scenario, only allow for a virtual method to be added to the class. This change requires the method table layout to have knowledge about the

new method, and allow calls to the new method. Space must be reserved in the v-table layout to allow for the new method to be added.

Add Virtual Method to Parent Class, Call It. This is same as the previous scenario, only the new method is in a parent class for which an instantiated instance of a derived class exists. Extra space must be reserved in the parent class's v-table to allow up to some maximum number of new methods to be added this way. This change requires new metadata and changes to the EE for method table layout.

Add Pinvoke Method to a Class, Call It. This is same as the previous scenario,, only for adding an Pinvoke surrogate value with full metadata. This change requires new metadata and some support in the EE to fault in the new library (if not already there) and the new entry point.

Override Method to Hide. The programmer may add a method override in a derived class that hides the parent method. The EE doesn't track all existing call sites that are made out of date. In theory the source dependency rules would force those targets to be recompiled and hence compiled with the JIT compiler using the proper reference.

Remapping Breakpoints. Allow breakpoints to be set in a method, then force the method body to be changed while keeping the breakpoints. This requires the breakpoints to be removed from the old method and mapped into the new method. As for any case where the method body changes, the debugger also needs to track the new JIT sequence point map. The compiler provides a map from the old MSIL sequence points to the new MSIL sequence points.

Add a New Class. The programmer may add a new class definition to a module. This change requires new metadata and EE to add the class to the delay load list.

Add a New Field to a Class. The programmer may add a new field to an existing class definition, of which there may be active instances. The change involves extending the metadata to include the new field, having the EE include this extra data, and any required work to provide the JIT compiler access to the new field. There may also be GC work to track what could be a new reference field.

The CLR does not provide functionality to delete methods and fields from a class. If a debugger supports deletion of methods and variables, it is up to the debugger to ensure that the deleted methods and fields are not used. A debugger could "tombstone" a method by using Edit and Continue to replace the deleted method with a method that throws an exception when it is invoked.

Edit and Continue operations may only be performed when the CLR is in a sane state with all managed threads stopped at safe points.

JIT Restrictions

In general, under Edit and Continue, the JIT-compiler tries to produce code (and stack frame) which is independent of previous versions of the method, but is still upgradeable from previous frames.

However, the frame layout depends on certain other factors and we don't put Edit and Continue restrictions on these. The JIT compiler has to handle these as gracefully as possible. All of these apply only to updating of a method currently on the stack.

- At the update point, the set of the older variables in scope has to be the same before and after. Newly added variables are acceptable.
- You can't be inside of a handler at the update point before or after.
- No Edit and Continue can be made if the older method used localloc.

2.4.6 Function evaluation

In order to evaluate user expressions and dynamic properties of objects, a debugger needs a way to run the code of the process being debugged. The CLR debugging services accomplish this by allowing the debugger to make a function or method call and have it run inside the debuggee's process.

Because this may be a dangerous operation (e.g., it may trigger a deadlock with existing code), the CLR allows such an operation to be aborted by the debugger. If the evaluation is aborted successfully, the thread will be treated as if the evaluation never happened, minus any side effects on locals, etc, from the partial

evaluation. If the function calls into unmanaged code, or blocks in some fashion, it may be impossible abort the evaluation.

The CLR notifies the debugger via a callback when the function evaluation completes normally or if the function throws an exception. The result of an evaluation is available for inspection via the ICorDebugValueXXX API methods in the debugging API.

The debugging services will set up a new chain on the thread to start a function evaluation and call the requested function. Once started, all aspects of the debugging API are available: execution control, inspection, function evaluation, etc. Nested evaluations are supported and breakpoints are hit as usual.

2.4.7 Dynamic Code Injection

Some debuggers allow a user to enter arbitrary statements in an "immediate window" and execute the statements. The CLR debugger services support this scenario. Within reason, there are no restrictions on what code you can inject dynamically. For example, non-local "goto" statements are not allowed.

Dynamic Code Injection is implemented using a combination of Edit and Continue operations and Function Evaluation. The code to be injected is wrapped in a function and injected using Edit and Continue. The injected function is then evaluated. The wrapper function can be supplied with arguments that are declared to be ByRef so that side effects are immediate and permanent if desired.

3. Debugging Architecture

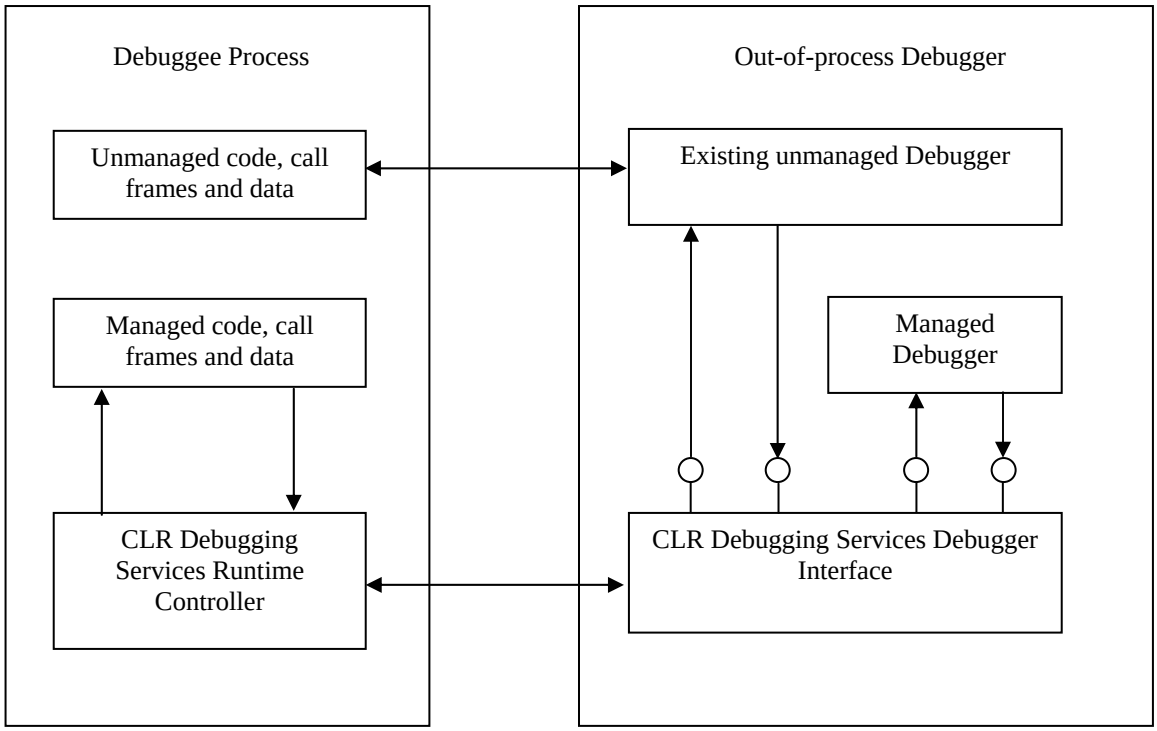
The CLR debugging services are designed to be used as if it were part of the operating system kernel. Today, when a program generates an exception the kernel gets involved to suspend execution of the process and pass the exception information to the debugger via the Win32 debugging API. The CLR debugging services serves the same purpose for managed code. When managed code generates an exception, the CLR debugging services will get involved to suspend the execution of the process and pass the exception information to the debugger. The sections below describe how and when the CLR debugging services get involved and what services they provide.

3.1 Process Architecture

One portion of the debugging services is implemented as a component that is always loaded into the same process with the program being debugged. This *Runtime Controller* component is responsible for communicating with the CLR and performing execution control and inspection of threads running managed code.

The other part of the code is implemented as a component that is loaded into a different process. This *Debugger Interface* component is responsible for communicating with the Runtime Controller on behalf of the debugger. It is also responsible for handling Win32 debugging events coming from the process being debugged, and either handling them or passing them on for use by an unmanaged code debugger. Finally, the Debugger Interface is the only part of the CLR debugging services with an exposed API.

The figure below illustrates where the different components of the CLR debugging services are located and how they interact with the CLR and the debugger.



3.2 Managed Code Debugger

It is possible to build a purely managed code debugger. The CLR debugging services allow such a debugger to attach to a process on-demand using a "soft attach" mechanism. A debugger that is soft attached to a process can subsequently detach from the process.

3.3 Thread Synchronization

The CLR debugging services have conflicting requirements when it comes to process architecture. On one hand, there are many compelling reasons for keeping the debugging logic in the same process as the program being debugged. For example, the data structures are complex and in many cases are manipulated by APIs rather than by a fixed memory layout. It is much easier to call the APIs directly rather than trying to decode the data structures from outside the process. Another example to keep the debugging logic in the same process is for better performance. There is no need for cross process communication in this case. Lastly, an important feature of CLR debugging is the ability to run user code in process with the debuggee, which obviously requires some cooperation with the debuggee process.

On the other hand, CLR debugging must coexist with unmanaged native code debugging. This can only be performed correctly from an outside process. Also, an out-of-process debugger is safer than an in-process debugger because interference between the debugger's operation and the process being debugged is minimized.

Because of these conflicting requirements, the CLR debugging services combine some of each approach. The primary debugging interface is out-of-process and coexists with the native Win32 debugging services. However, the CLR debugging services add the ability to *synchronize* with the debuggee process so that it can safely run code in the user process. To perform this synchronization the services collaborate with the OS and the CLR to suspend all threads in the process at a place where they are not in the middle of an operation which leaves the runtime in an incoherent state. The debugger is then able to run code in a special

thread which can examine the state of the runtime using the correct APIs, as well as call user code if the need arises.

When managed code executes a breakpoint instruction or generates an exception, the Runtime Controller is notified. This component will determine which threads are executing managed code and which threads are executing unmanaged code. Usually, threads that are running managed code will be allowed to continue executing until they reach a state where they may be safely suspended. This may include completing a GC in progress. Once the managed code threads have reached safe states, all threads will be suspended. The Debugger Interface then informs the debugger that a breakpoint or exception has been received.

When unmanaged code executes a breakpoint instruction or generates an exception, it is the Debugger Interface component which receives notification (through the Win32 debugging API). This notification is passed to an unmanaged debugger. If the debugger decides that it wants to perform synchronization (for instance, so managed code stack frames can be inspected), the Debugger Interface must first restart the stopped debuggee process, and then inform the Runtime Controller to perform the synchronization. The Debugger Interface is then notified when synchronization is complete. This synchronization is transparent to the unmanaged debugger.

The thread which generated the breakpoint instruction or exception must not be allowed to execute while all of this synchronization is going on. In order to facilitate this, the Debugger Interface "hijacks" the thread by placing a special exception filter in the thread's filter chain. When the thread is restarted it will enter this filter which will place the thread under the Runtime Controller's control. When it is later time to continue exception processing (or cancel the exception) the filter will return control to the thread's normal exception filter chain (or return the proper result to resume execution.).

In rare cases, the thread which generates the native exception may be holding crucial locks which need to be released before the runtime's synchronization can complete. (Typically these will be low-level library locks, for instance a lock on the malloc heap.) In such cases, the synchronization operation must timeout and the synchronization will fail. This will cause certain operations that require the synchronization to fail.

3.4 The In-Process Helper Thread

In order for the CLR debugging services to operate correctly, a single *debugger helper thread* is used within every CLR process. This helper thread is responsible for handling many of the inspection services provided by the API in addition to assisting with thread synchronization under certain circumstances.

3.5 Interactions with the Just in Time compilers

In order to allow a debugger to debug JIT-compiled code, the CLR debugging services must be able to map information from the MSIL version of the function to the native version of the function. This information includes sequence points in the code and local variable location information. Typically, this information takes extra resources to produce and keep around, so it is not produced unless the runtime is in debugging mode.

Also, JIT-compiled code can be highly optimized. Optimizations such as common sub-expression elimination, inline expansion of functions, loop unwinding, code hoisting, etc. can lead to loss of correlation between the MSIL code of a function and the native code which will be called to execute it. Thus, the JIT compiler's ability to correctly provide mapping information is severely impacted by these aggressive code optimization techniques. Therefore, when the runtime is run in debugging mode, the JIT compiler will not perform certain optimizations. This will allow debuggers to accurately determine the source line mapping and location of all local variables and arguments.

3.6 Debugging Modes

While we have tried to avoid introducing special modes for debugging, there are two cases where such modes are required.

The first case is when Edit and Continue functionality is desired. In this case the runtime actually operates differently to allow code to be later changed. This is because the layout of certain runtime data structures needs to be different to support Edit and Continue. Since this has a negative impact in performance, this mode should not be used unless Edit and Continue is desired. This mode is termed “Edit and Continue” mode.

The second case is needed so the JIT compiler can generate mapping information (see above discussion about JIT). Again, since this has a negative impact on performance (both from decreased code quality and increased effort and space to generate and remember mapping information), this mode should not be used unnecessarily. This mode is called simply “JIT Debugging Mode”.

If a program is debugged while not in Edit and Continue mode, Edit and Continue functionality is not supported. If a program is debugged while not in debugging mode, most of the debugging features will still be supported; however only raw native code will be visible for JIT-compiled method frames – source and local variable mapping will be unavailable.

Both of these modes can be enabled programmatically through the CLR debugging services by a debugger which gains control of a process before the runtime has initialized itself. This is adequate for many purposes. However, a debugger attaching to a process which has already been running for a while (for instance, during Just-In-Time debugging) will be unable to activate these modes.

To help deal with these problems, a program can be run in Just-In-Time mode or Debugging mode independently of a debugger. Mechanisms for enabling debugging are described in the specification Enabling Profiling and Debugging.

JIT optimizations can make an application less debuggable. The CLR debugging services allow inspection of stack frames and inspection of local variables with JIT-compiled code that has been optimized. Stepping is supported but the stepping may be imprecise. A program can be run that instructs the JIT compiler to turn off all JIT optimizations causing the JIT compiler to produce “debuggable code.” For details see the specification Enabling Profiling and Debugging.