Visual Studio Sample Debug Engine Walkthrough 3 – Breakpoints and Stopping Events

*Jackson Davis – Software Engineer, Visual Studio Debugger*

Walkthrough1 described the basic architecture of AD7 and this sample debug engine. As described in that walkthrough, there are two categories of events sent from a debug engine to the Visual Studio Debugger: stopping events and non-stopping. Both types of events can be sent synchronously or asynchronously, although most all events should be sent asynchronously.

A non-stopping event is sent when there is no need for the debugger to enter break-mode. A stopping event is sent when the debugger should enter break-mode.

Walkthrough1 and 2 demonstrated several non-stopping events including thread-create and module load. This walkthrough will step through the breakpoint binding process, hitting a breakpoint, and continuing executing after hitting a breakpoint.

Binding a breakpoint is the process of converting the location the user wanted the breakpoint to be into an actual breakpoint. That is, mapping from a source file / line number pair into an address in the debuggee’s instruction stream and ultimately placing a breakpoint at that location. There are several types of breakpoints supported by the Visual Studio debugger: including source/line breakpoints, function breakpoints, data breakpoints. This sample only supports source/line breakpoints.

Once the breakpoint is bound, the debug engine actually creates the breakpoint. For x86 native debuggers, this means replacing the instruction in the debuggee’s machine instruction stream into an interrupt that the debugger will interpret as a breakpoint.

When the breakpoint is hit (the instruction is executed) the debugger is notified of the breakpoint event via WaitForDebugEvent. The debugger then notifies the Visual Studio Debugger UI of the breakpoint (a stopping event) and waits for the user to continue execution (by hitting F5 in most cases).

To continue executing passed the breakpoint, the debugger must replace the breakpoint instruction with the original instruction, execute that instruction, and then re-apply the breakpoint so it can be hit again.

# Binding Breakpoints

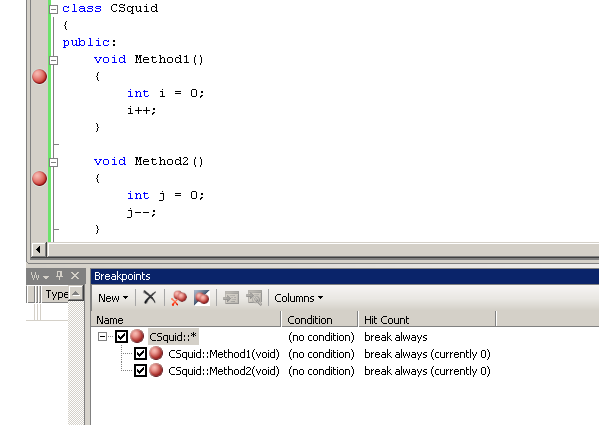
AD7 has the concept of “pending” breakpoints and “bound” breakpoints. A pending breakpoint is an abstract representation of a breakpoint created when the user adds the breakpoint (by hitting F9). A bound breakpoint is a breakpoint that has actually been bound to a location in the debuggee and will result in a Breakpoint event if the breakpoint is hit.

When the user adds a new breakpoint, the engine is notified by debugger UI calling IDebugEngine2.CreatePendingBreakpoint. CreatePendingBreakpoint takes an instance of IDebugBreakpointRequest2 and returns an instance of IDebugPendingBreakpoint2.

The breakpoint request contains the information needed to bind the desired breakpoint such as:

1. A source file / line number pair for source/line breakpoints or
2. An address for data breakpoints (break when the contents of this address changes) or
3. A function name to bind to

The engine returns an instance of IDebugPendingBreakpoint2 to the debugger during the call to CreatePendingBreakpoint. IDebugPendingBreakpoint2 is an abstract representation of the user’s notion of breakpoint. It represents the potential for breakpoint’s to be bound to one or more locations in the debuggee. The bound breakpoints become children of the pending breakpoint. This parent / child relationship can be seen in Visual Studio’s breakpoint window when a breakpoint binds to one or more locations:

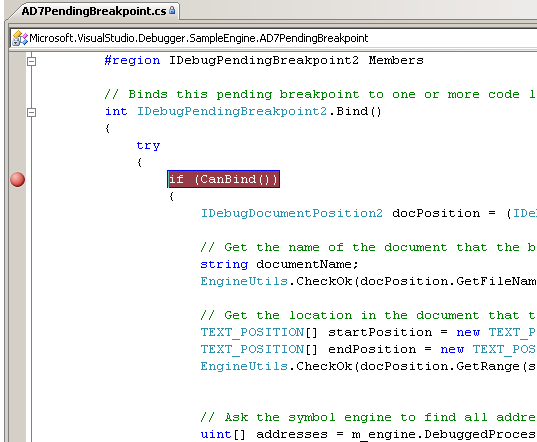


In this example, CSquid::\* is the pending breakpoint which resulted in two bound breakpoints: CSquid::Method1 and CSquid::Method2. If the pending breakpoint only results in one bound breakpoint, the Visual Studio UI only shows one entry in the breakpoint window instead of the tree.

In the sample engine, the implementation of IDebugEngine2.CreatePendingBreakpoint is in AD7Engine.cs. It defers its work to BreakpointManager:: CreatePendingBreakpoint which creates a new instance of AD7PendingBreakpoint and adds it to the engine’s breakpoint collection.

Pending breakpoints are bound in two ways in AD7: during a call to IDebugPendingBreakpoint2::Bind which is called by the Visual Studio Debugger UI and after symbols for a module are loaded. This sample does not bind breakpoints when new symbols are loaded because if only supports symbols on the primary exe of the debuggee. However, a real debug engine will need to bind in that scenario.

In the sample engine, AD7PendingBreakpoint implements IDebugPendingBreakpoint2 . To step through a breakpoint bind, open the debug engine sample, open AD7PendingBreakpoint.cs and set a breakpoint in the Bind method, and start debugging.



When the new instance of Visual Studio launches under the debugger, create a new Visual C++ console application and add a breakpoint in the main function. Build the project and then use the Project Launcher to launch it with the sample debug engine (tools->Project Launcher). The breakpoint in AD7PendingBreakpoint::Bind will now get hit. The first thing Bind does is check that it can bind by verifying that this breakpoint has not been deleted and the bpLocationType field in the breakpoint request is BPLT\_CODE\_FILE\_LINE. BPLT\_CODE\_FILE\_LINE is the type of breakpoint request created when the user adds a breakpoint to in the text editor of Visual Studio.

Next, Bind extracts the instance of IDebugDocumentPosition2 from the breakpoint requests union fields with the following line:

IDebugDocumentPosition2 docPosition = (IDebugDocumentPosition2)(Marshal.GetObjectForIUnknown(m\_bpRequestInfo.bpLocation.unionmember2));

IDebugDocumentPosition2 represents and abstract location in a source file. Bind obtains the source file name and line / column from the document position and then asks the engine back-end to return the addresses that match that document position:

// Ask the symbol engine to find all addresses in all modules with symbols that match this source and line number.

uint[] addresses = m\_engine.DebuggedProcess.GetAddressesForSourceLocation(

null,  
 documentName,   
 startPosition[0].dwLine + 1,   
 startPosition[0].dwColumn);

Finally, the implementation of Bind creates instances of AD7BreakpointResolution and AD7BoundBreakpoint to represent each address that matches the document position and asks the engine back-end to add breakpoints to those locations.

Stepping into the call to m\_engine.DebuggedProcess.SetBreakpoint will land you in WorkerAPI.cpp DebuggedProcess::SetBreakpoint, you can see the backend details of creating the breakpoint, and finally, a call to m\_callback->OnBreakpointBound which will send a non-stopping asynchronous debug event which is an instance of IDebugBreakpointBoundEvent2 to the SDM. This notifies the SDM that a new bound breakpoint exists.

There are a few other interesting methods int IDebugPendingBreakpoint2. EnumBoundBreakpoints returns an enumerator for all the bound breakpoints. EnumErrorBreakpoints returns an enumerator for breakpoints that failed to bind for various reasons. There are several control methods for enabling the breakpoint, disabling the breakpoint, and obtaining breakpoint state.

# Breakpoint Implemenation in the Sample

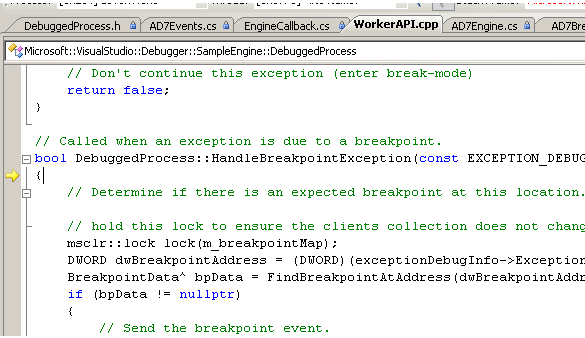
This engine sample makes use of the x86 breakpoint instruction to implement code breakpoints. In x86, the breakpoint instruction is int3 (0xcc in machine language). When this instruction is executed, the processor will notify the operating system (via the interrupt table) which will notify the debugger via WaitForDebugEvent.

To see this in action, set a breakpoint in DebuggedProcess::SetBreakpoint in the engine sampl e. Run the sample under the debugger. When the new instance of Visual Studio opens, create a new native Visual C++ application and add a breakpoint. Launch this Visual C++ application using the project launcher. The breakpoint in DebuggedProcess::SetBreakpoint will get during a call to AD7PendingBreakpoint::Bind. DebuggedProcess::SetBreakpoint first checks to see if a breakpoint already exists at this address. If it does, then a new addition is made to that breakpoint’s linked list of breakpoints. However, if the breakpoint doesn’t exist, the sample engine does the following:

1. Read one byte out of the debuggee’s instruction stream at the address the breakpoint is being bound to. This is the original byte in the instruction stream.
2. Do some book-keeping to keep track of the fact that the breakpoint at this address was added by the debugger
3. Make sure the original instruction was not a breakpoint (possible if the user embeds a breakpoint in assembly language)
4. Write the breakpoint opcode into the debuggee’s instruction stream and flush the instruction cache to ensure the stream is updated in the cache.

To see the breakpoint actually get hit, set a breakpoint in WorkerAPI.cpp DebuggedProcess::DispatchDebugEvent in the EXCEPTION\_DEBUG\_EVENT case statement. This is the handler for every exception that fires in the debuggee process while the debugger is attached. Since breakpoint are implemented using an interrupt in x86, the debugger is notified of them using an exception notification. Hit F5 to allow the debuggee to run.

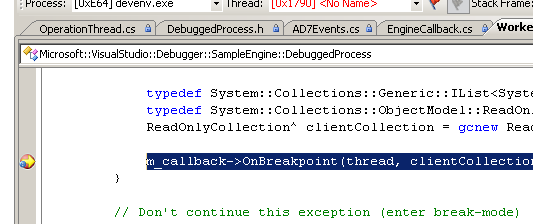
The DispatchDebugEvent will get with an EXCEPTION\_DEBUG\_EVENT once for the entrypoint breakpoint. Ignore this for now and continue execution. The second time DispatchDebugEvent is called with EXCEPTION\_DEBUG\_EVENT, is the breakpoint actually getting hit in the debuggee. DispatchDebugEvent checks to see if the exception is a breakpoint exception by examining the exception code in the EXCEPTION\_DEBUG\_EVENT. This is done is done by calling IsBreakpointException which compares the exception code to 0x80000003, the exception code for int3 in x86. Once it determines the exception is due to a breakpoint, and the debugger is not expecting an asynchronous break (the user hitting the pause button), the debugger DebuggedProcess::HandleBreakpointException:



HandleBreakpointException verifies this is a breakpoint the debugger added by checking its collection of breakpoints for a breakpoint at the address the exception occurred at. If no breakpoint is found, then this breakpoint was an embedded breakpoint added by the user in assembly. That scenario is not supported by the sample engine.

Once it determines the breakpoint is owned by the debugger, it prepares to enter break-mode by keeping track of the last stopping event: m\_lastStoppingEvent = Breakpoint and sending the OnBreakpoint event to the SDM. Notice the sample engine does not call ContinueDebugEvent for breakpoint exceptions. This is because breakpoints are stopping events so the debuggee process should be left broken at the breakpoint.

Step into the call to OnBreakpoint:



This will land you in EngineCallback.cs OnBreakpoint which will enumerate all of the bound breakpoints that were just hit and send an instance of AD7BreakpointEvent to the SDM. AD7BreakpointEvent implements IDebugBreakpointEvent2 and derives from AD7StoppingEvent which makes it a stopping event to the SDM.

The Visual Studio Debugger will now enter break-mode and show the debuggee as broken at the breakpoint.

# Continuing After a Entering Break-Mode for a Breakpoint

At the end of the last section, Visual Studio entered break-mode at the breakpoint because the breakpoint stopping event was sent to the SDM. This section is going to walkthrough what happens when the user hits F5 to continue execution.

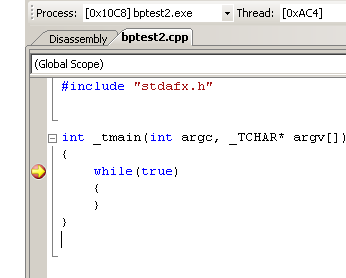
There are two ways Visual Studio asks debug engines to continue after asynchronous stopping events. IDebugProgram3::ExecuteOnThread and IDebugProgram2::Continue. The difference between the two methods has to do with stepping. When the engine receives ExecuteOnThread, it means stop all active steppers and cause the debuggee to begin running. Continue on the other hands, means leave steppers active. When a stepper finishes stepping, the debuggee should once again enter break-mode.

“Steppers” in this case is an abstract notion for making a debuggee step. An example of continue vs. execute might make this more clear:

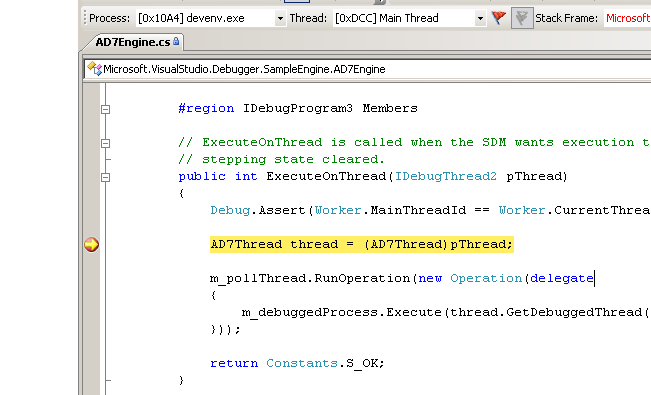
1. Let’s assume the user has stepped over a function call statement in Visual Studio. This makes a stepper active.
2. Let’s now assume the function being called has a breakpoint in it.
   1. If the breakpoint is a normal run-of-the-mill breakpoint that does not have its “when hit” property set, the debugger will enter break-mode because the breakpoint was hit.
      1. When the user hits F5, ExecuteOnThread is called, so the controlling engine clears the steppers and begins executing.
   2. Now, let’s instead assume the “When Hit” property of the breakpoint is set to print the current call-stack to the output window and then continue
      1. When the breakpoint is hit, the debugger enters break-mode, and prints the callstack (there is a delay which makes this transition invisible to the user).
      2. After printing the callstack, the UI will call Continue instead of ExecuteOnThread leaving steppers active. The debugger will now enter break mode at the line after the function call. Exactly as if the breakpoint was never hit.

This sample debug engine does not implement stepping, and therefore ExecuteOnThread and Continue have the same behavior. However, in an engine where stepping is supported, this subtle different will need to be accounted for.

To walk through the process of continuing from a breakpoint, open the debug engine sample solution and set a breakpoint in AD7Engine.cs ExecuteOnThread. Hit F5 to start another instance of Visual Studio under the debugger. Create a new Visual C++ console project in the new instance of Visual Studio, and add a breakpoint. Launch the process using the project launcher. The breakpoint in the Visual C++ application will now be hit:

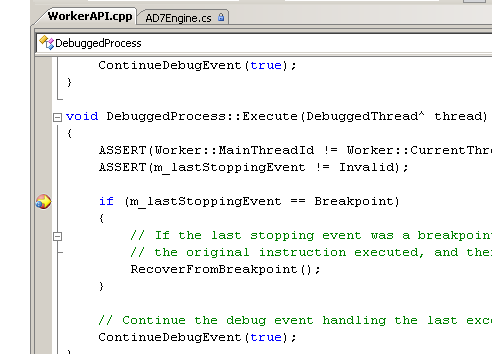


Hit F5 to continue from the breakpoint. The Visual Studio debugger will now ExecuteOnThread. The breakpoint in that function will now get hit:



The first thing the sample engine does is cast the IDebugThread2 pointer back to to AD7Thread (this is actually a QueryInterface) . Next, it transfers control to the event thread and calling DebuggedProcess::Execute.

Open WorkerAPI.cpp and set a breakpoint in DebuggedProcess::Execute. Then, hit F5



DebuggedProcess::Execute sees the last stopping event was a breakpoint, so special processing is required to restore the original opcode in the debuggee’s instruction stream. The call to RecoverFromBreakpoint overwrites the breakpoint’s int3 instruction with the original instruction. The x86 trap flag is set on the processor and then the breakpoint DEBUG\_EVENT is then continued.

Because the trap-flag is set, the debuggee process single-steps over the original instruction. Next, the debugger receives a single-step exception when it calls WaitForDebugEvent. The debugger then restores the breakpoint opcode, and continues the single-step exception so the debuggee process begins running normally.

If you’d like to see the single-step exception in action, set another breakpoint in WorkerAPI.cpp, DispatchDebugEvent in the EXCEPTION\_DEBUG\_EVENT case statement. This time, the exception code in will be 0x80000004, the code for a single-step exception. The function IsSingleStepException detects that. Next DispatchDebugEvent calls DebuggedProcess::HandleBreakpointSingleStepException to put the breakpoint opcode back in the debuggee process. Finally, the single-step exception is continued allowing the debuggee to run.

# Other Stopping Events

This debug engine sample supports two other stopping events: load complete and async-break.

LoadComplete is sent after all of the debuggee’s modules have loaded but before any code is executed. Normally, the Visual Studio Debugger UI immediately calls ExecuteOnThread for this event. However, if the user started the project via a step-in (instead of a launch), the debugger will stop at the LoadComplete event.

Async-Break is what happens when the user clicks on the pause button in Visual Studio while the debuggee is running. The debug engine receives a call to IDebugEngine2.CauseBreak. The engine should bring the debuggee to a stop and then send the async-break event (IDebugBreakEvent2). This sample does support Async-break. The implementation of CauseBreak is in AD7Engine.cs. Sending the event is in EngineCallback.cs OnAsyncBreakComplete. Since the engine uses kernel32! DebugBreakProcess to force the debuggee to break, the top-of-the callstack will be in ntdll.dll. Since the engine doesn’t load symbols for anything but the main exe, the callstack will be incomplete.

Both LoadComplete and Async-Break are continued via ExecuteOnThread when the user hits F5.