Visual Studio Sample Debug Engine Walkthrough 1

*Jackson Davis – Software Engineer, Visual Studio Debugger*

The Visual Studio Sample Debug Engine is intended to demonstrate what is required to implement a custom debug engine that integrates with the Visual Studio Debugger. It is a basic 32-bit Win32 Debugger which supports applications built in debug mode with Visual C++. The sample makes use of the Win32 Debugger API and the DIA symbol API to enable debugging 32-bit x86 win32 applications. However, its usage of both is limited and is intended only as an implementation detail of the sample.

The sample is split into three separate projects:

1. A front end implemented in C# which implements the AD7 interfaces and interacts with the SDM. The root object of this project is AD7Engine which implements IDebugEngine2. This project is called Microsoft.VisualStudio.Debugger.SampleEngine. This project is the primary focus of the sample.
2. A back end implemented in mixed mode C++ which interacts with the win32 debugging API and DIA, the public symbol interfaces. This project is called Microsoft.VisualStudio.Debugger.SampleEngineWorker.
3. A Visual Studio Package which is used to launch Visual C++ projects using the new engine. This project is named ProjectLauncher. It adds a new command button “ProjectLauncher” to the Visual Studio Tools menu which will launch a form that displays the current projects in Visual Studio that can be launched with the new engine.

A few definitions are in order:

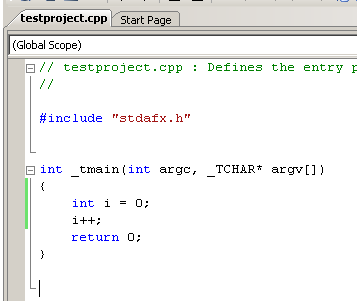
1. AD7 (Active Debugging 7) – the name of the interfaces between the SDM and debug engines. AD7 is defined in msdbg.idl and is imported into managed code via Microsoft.VisualStudio.Debugger.InteropA.dll.   
   The interfaces in AD7 are documented on MSDN at <http://msdn2.microsoft.com/en-us/library/bb147088.aspx>.
2. The Debugger UI is the windows and commands in the Visual Studio debugger that the user actually interacts with. A good example of this is the watch window or the little red circle that appears in the text editor as a breakpoint.
3. The SDM stands for session debug manager. It is a debug engine multiplexer. Admittedly, that’s a confusing explanation. Essentially, the SDM’s job to combine all of the events and commands for the various debug engines into one unified stream for the UI. The debugger UI only displays one ”view” of what is being debugged at a time. Even if the user is debugging multiple processes or threads, they are only looking at one of them.
4. Win32 Debugging Api – The functions provided by windows for debugging native applications. Documented at [http://msdn2.microsoft.com/en-us/library/ms679300(VS.85).aspx](http://msdn2.microsoft.com/en-us/library/ms679300(VS.85).aspx%20)
5. DIA – The Debug Interface Access SDK. Used to read and interpret PDB files. DIA is documented at <http://msdn2.microsoft.com/en-us/library/x93ctkx8(VS.80).aspx>

# Getting Started

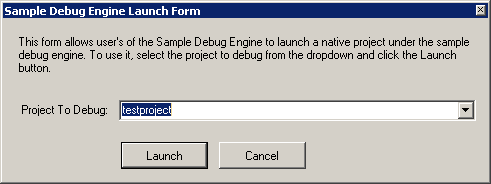
The sample Visual Studio Debug Engine requires Microsoft Visual Studio 2013, though there is nothing in it which cannot be made to work in many previous versions of Visual Studio. The sample solution can be downloaded from <http://code.msdn.microsoft.com/debugenginesample>.

To use the sample:

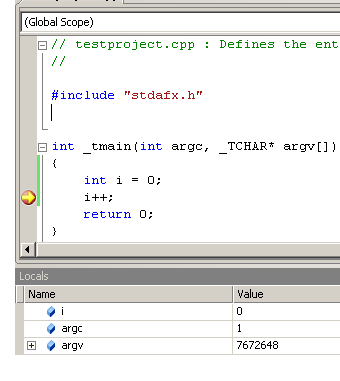
1. Download the zip file and decompress it onto your machine
2. Install the Visual Studio 2013 SDK
3. Open engine.sln in Visual Studio 2013:
4. Make sure that the ‘Project Launcher’ is the startup project.
5. If you are interested in debugging the ‘backend’ of this project. You should follow these additional steps, ignore if you are only interested in debugging the front end:
   1. Open project properties for the ‘Project Launcher’ project
   2. Check ‘enable native code debugging’
   3. It may also be useful to enable ‘Use Managed Compatibility Mode’ in ‘Tools->Options->Debugging’ as this improves the experience debugging C++/CLI code.
6. Start Debugging
   1. A second instance of Visual Studio 2013 should launch
7. Create a new Visual C++ Console project in this instance of Visual Studio
8. Add some code to the project:



1. Build the project
2. Add a breakpoint to the code
3. Navigate to Tools->Project Launcher in Visual Studio. A new form should open:



1. Make sure your project is selected in Project To Debug and choose “Launch”
2. Your application should launch using the sample engine and the breakpoint should be hit:



If you can complete these steps, you are ready to dig into the sample source code.

# An Architectural Overview

In Visual Studio, debugger control and debuggee data flow between the Debugger UI and the Debug Engines in two directions. Requests from the user (such as launch process, create breakpoint, continue running, …) flow from the Debugger UI, through the SDM, and into the appropriate engine(s). Events about the state of the debuggee process (thread create, breakpoint hit, module load…) flow from the engines, into the SDM, and into the Debugger UI. Most debug engines (including this sample) require at least two threads: one to handle requests coming down from the SDM and another to listen to events from the debuggee process.



There are two types of debuggee events in AD7: stopping and non-stopping. A stopping event is an event that causes the debugger UI to enter break mode. A good example of a stopping event is IDebugBreakpointEvent2 which is sent when the debuggee hits a breakpoint. Non-Stopping events are events that do not cause the debugger to enter break-mode. An example non-stopping event is IDebugModuleLoadEvent2 which is sent when a new module is loaded into the debuggee. Non-stopping events usually result in some visual state changing (such as a new module in the modules window or a notification in the output window), but the debugger does not enter break-mode.

Both stopping and non-stopping events can be sent synchronously or asynchronously. However, asynchronous is far more common). The engine passes different values of the EVENTATTRIBUTES enumeration to control if the event is being sent synchronously or not.

When an engine is sending an Asynchronous event, the SDM queues the event up to eventually be sent to the debugger UI and then returns control to the engine immediately. If the event was a stopping event, then the engine waits to continue execution in the debuggee until the UI asks it to continue (usually because the user hits F5). If the event was non-stopping, the engine fires the event, and immediately continues execution in the debuggee.

When an engine sends a synchronous event, the engine waits for the UI to respond with a call to IDebugEngine2:: ContinueFromSynchronousEvent. The only synchronous event in this sample is AD7ProgramDestroyEvent.

Events are sent from the engine to the SDM by passing an instance of an event interface (such as IDebugModuleLoadEvent2) to the instance of IDebugEventCallback which is given to the Debug Engine by the SDM when debugging is being started. In the sample engine, AD7Engine (the root object of the engine) holds a reference to an instance of EngineCallback which manages sending events to the SDM. The event interfaces are implemented in AD7Events.cs. There are four base classes that define the synchronous / stopping behavior. All the event objects derive from one of the four base classes: AD7AsynchronousEvent, AD7StoppingEvent, AD7SynchronousEvent, and AD7SynchronousStoppingEvent.

Since the sample engine is a 32-bit win32 debugger, it relies on the win32 debugging API to notify it about events in the debuggee. The primary function for this api is kernel32 WaitForDebugEvent. WaitForDebugEvent returns a DEBUG\_EVENT structure for each of the supported events in the debuggee. These include: process create, thread create, module load, exception, module unload, thread destroy, process destroy and rip. Win32 debuggers receive these events by calling WaitForDebugEvent in a loop. In the sample engine, the call to WaitForDebugEvent occurs in WorkerApi.cpp WaitForAndDispatchDebugEvent which is called in a loop in OpertionThread.cs ThreadFunc. This loop checks if a new debug event has been sent by calling WaitForDebugEvent with a timeout. If none had been received, it then checks to see if a new request from the user has been sent. It is done this way because several of the debugging API functions require that they are called on the same thread as WaitForDebugEvent.

This diagram shows some of the major objects in the sample and how they relate to each other.



**Some Important objects in the engine front-end:**

|  |
| --- |
| AD7Engine is the root object of the sample engine. It represents the engine as well as the process being debugged. AD7Engine maintains collections of the threads and modules in the process being debugged.  It implements the following AD7 interfaces:  IDebugEngine2: This interface represents a debug engine (DE). It is used to manage various aspects of a debugging session, from creating breakpoints to setting and clearing exceptions.  IDebugEngineLaunch2: Used by a debug engine (DE) to launch and terminate programs.  IDebugProgram3: This interface represents a program that is running in a process. Since this engine only debugs one process at a time and each process only contains one program, it is implemented on the engine.  IDebugEngineProgram2: This interface provides simultaneous debugging of multiple threads in a debuggee. |
| AD7Thread represents a thread in the debuggee process. It implements the IDebugThread2 AD7 Interface which allows the debugger to obtain information about the thread (such as its thread id and name) as well as an enumeration of AD7StackFrame which represent the current callstack for the thread. |
| AD7Module represents a module loaded in the debuggee process. It implements IDebugModule2 and IDebugModule3 |

# Walkthrough 1: Tracing Through Debuggee Launch

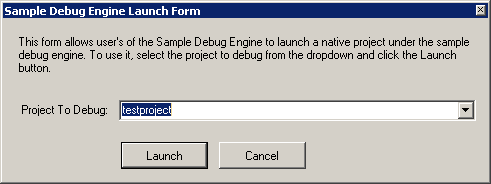
The sample debug engine makes use of the engine launch functionality of the Visual Studio Debugger. This path allows debug engines to launch debuggee processes on the user’s local machine. When the user has launched the debuggee process, Visual Studio will create a new instance of the Debug Engine and call IDebugEngineLaunch2.LaunchSuspended. Normally, Visual Studio launches a program using the IDebugPortEx2::LaunchSuspended method and then attaches the debugger to the suspended program. However, there are circumstances in which the debug engine may need to launch a program. For example, if the debug engine is part of an interpreter and the program being debugged is an interpreted language.

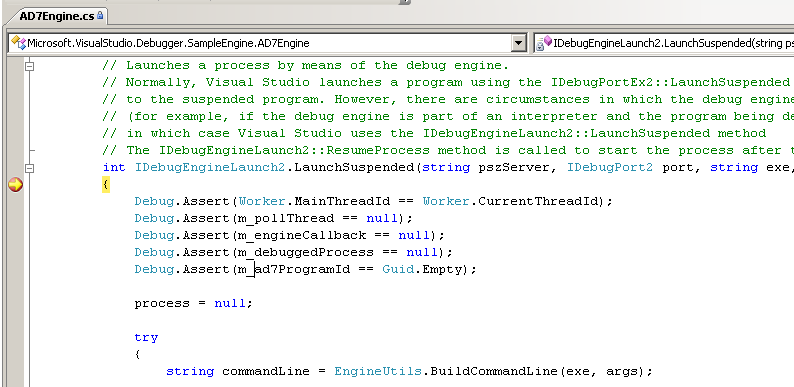
To trace the launch steps in the debuggee process:

1. Launch Visual Studio 2013 as an administrator and open the engine solution
2. Build the solution
3. Open AD7Engine.cs and set a breakpoint in the method “LaunchSuspended”.
   1. AD7Engine.cs can be found Solution Explorer in the AD7Impl folder of the Microsoft.VisualStudio.Debugger.SampleEngine project
4. Start Debugging
5. A new instance of Visual Studio should launch.
6. Create a new native Visual C++ console project in the new instance of visual studio. Name it Test Project
7. Add the following code to main in TestProject

|  |
| --- |
| int \_tmain(int argc, \_TCHAR\* argv[])  {  int i = 0;  i++;  return 0;  } |

1. Set a breakpoint on the line i++
2. Go to the Tools menu and choose “ProjectLauncher. The ProjectLauncher form should open:



1. Click on Launch
2. The instance of Visual Studio that contains TestProject should now enter debug mode
3. The breakpoint you added in LaunchSuspended should be hit:  
   
4. You are now ready be begin stepping through the launch process.
5. As you can see, the code in LaunchSuspend does the following:
   1. Constructs the debuggee command line from the exe and args parameters
      1. string commandLine = EngineUtils.BuildCommandLine(exe, args)
   2. Creates an instance of ProcessLaunchInfo which is used to pass information about what to launch to the back-end.
   3. Creates an instance of the worker thread object which creates a the debug event thread and sets up some ManualResetEvents used to notify the event thread when the request thread needs it to perform an operation
      1. m\_pollThread = new WorkerThread();
   4. Creates a new instance of EngineCallback passing in the instance of IDebugEventCallback2 that was passed by the sdm.
   5. Requests that the event thread actually launch the process:  
      m\_pollThread.RunOperation(new Operation(delegate

{

m\_debuggedProcess = Worker.LaunchProcess(  
 m\_engineCallback, processLaunchInfo);

}));

* 1. The launch must occur on the event thread per win32 debugging api requirements. The RunOperation method of m\_polllThread blocks the current thread and uses the manual reset events to get to the other thread. RunOpertation takes an instance of Operation as its parameter which takes a delegate. In this case, it is an anonymous delegate that is calling the LaunchProcess method. The delegate is executed on the other thread.

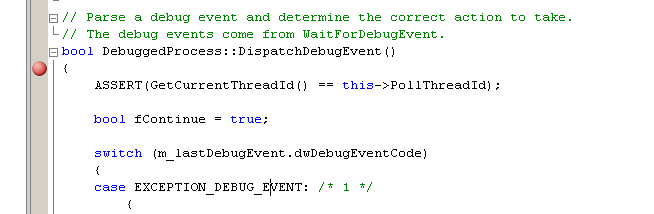
1. To see the actual process launch, open up WorkerApi.cpp in Microsoft.VisualStudio.Debugging.SampleEngineWorker and set a breakpoint in Worker::LaunchProcess and Hit F5 from where you currently are in AD7Engine::LaunchSuspended
2. Once the breakpoint in LaunchProcess is hit, you can see in the callstack, which you are now executing on the event thread of the sample engine.
3. If you switch back to the Main Thread via the thread’s window, you can see the main thread of Visual Studio is blocked waiting for the call to launch to complete.

Worker::LaunchProcess uses the process launch info to call kernel32!CreateProcess with the usual arguments. However, the dwCreationFlags parameter is set to DEBUG\_ONLY\_THIS\_PROCESS. This instructs windows to launch the process with debugging support and begin sending debug events to WaitForDebugEvent.

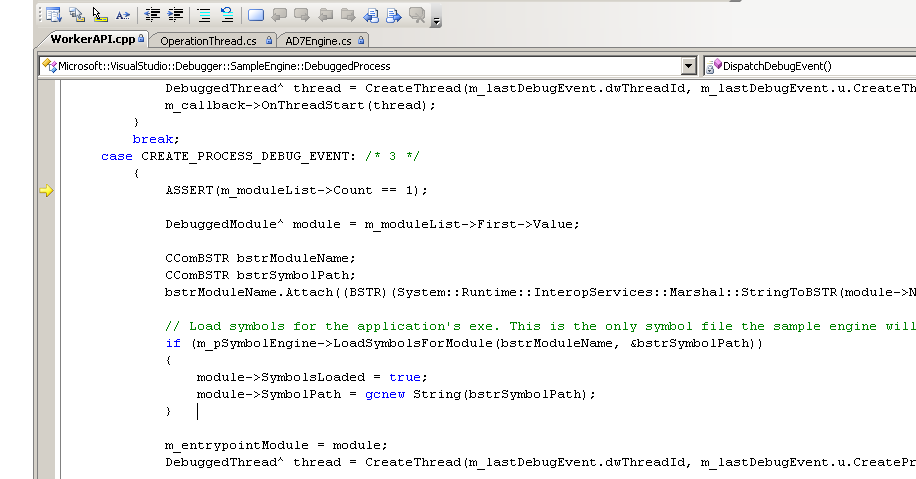
If you look up the callstack from where you are in Worker::LaunchProcess to the ThreadFunc frame, you can see the debug event loop. ThreadFunc is the debug loop for the sample engine and does the following:

1. Check for and dispatch debug events (wait timeout for new debug events set to 50ms)
2. Check for and execute requests from the UI
3. Repeat.

Continuing on from LaunchProcess:

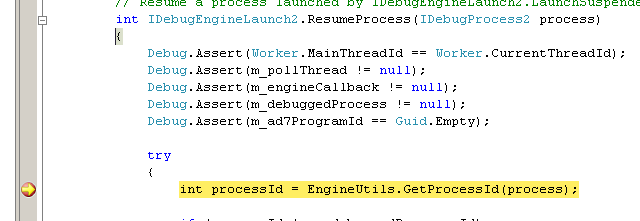
1. Set a breakpoint in WorkerApi.cpp at the beginning of DispatchDebugEvent in WorkerApi.cpp.
2. 
3. Hit F5 to run
4. The breakpoint in DispatchDebugEvent will now get hit

DispatchDebugEvent is a large switch statement which handles the various win32 debug events. The first debug event to be received from windows is the CREATE\_PROCESS\_DEBUG\_EVENT. Step from the current location in DispatchDebugEvent into the switch statement.

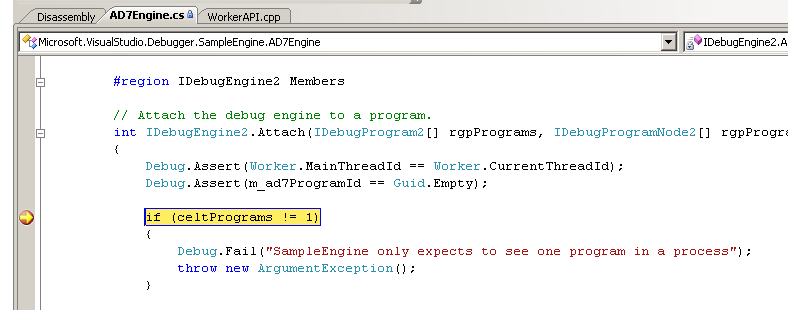


1. When handling the create process event, the sample debug engine will:
   1. Attempt to load symbols for the primary exe in the process (which was saved when the instance of DebuggedProcess was created).
   2. Create an instance of DebuggedThread to represent the main thread in the application
   3. Save the main module and main thread objects for a later call to DebuggedProcess::ResumeFromLaunch.
   4. No debug events are sent to the UI while processing the CREATE\_PROCESS\_DEBUG\_EVENT for a launch. The call to DispatchDebugEvent unwinds and leaves the debuggee broken in anticipation of the UI calling IDebugEngineLaunch2.ResumeProcess

Now, navigate back into the C# front end, open AD7Engine.cs and set a breakpoint in IDebugEngineLaunch2.ResumeProcess. This is the next method the debugger will call after LaunchSuspended and is used to resume execution of the debuggee after launch. Remember, that the handler for CREATE\_PROCESS\_DEBUG\_EVENT did not continue the event so the debuggee process is still suspended. Hit F5 and wait for the breakpoint in ResumeProcess to get hit:

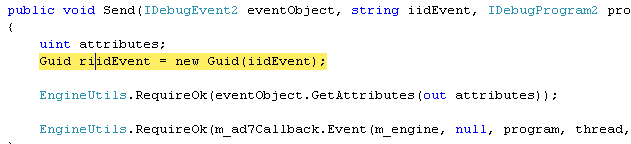


The first thing ResumeProcess does is call portNotify.AddProgramNode which calls back into the SDM. The SDM will synchronous call back to the engine via IDebugEngine2.Attach. Set a breakpoint in IDebugEngine2.Attach and hit f5:

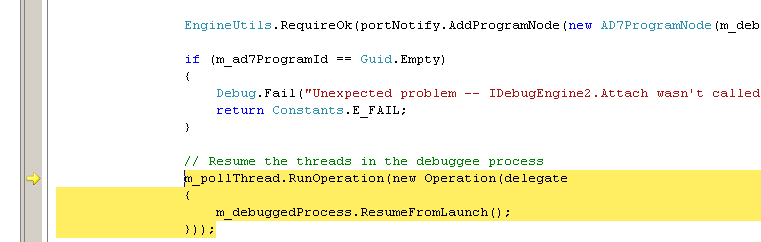


IDebugEngine2.Attach is called in when the user is launching a new process and when the user is attaching to an existing process. In this case, we are launching a new process.

Step the debugger to the line: AD7EngineCreateEvent.Send(this). This is the first debug event being sent to the debugger from the engine. It is notifying the debugger that the engine has been created. Although this event isn’t all that interesting, let’s step into to trace how the events are defined in the sample:

1. Step into the AD7EngineCreateEvent.Send(this)
2. This method creates a new instance of AD7EngineCreateEvent which implements the IDebugEngineCreateEvent2 AD7 interface. Notice AD7EngineCreateEvent derives from AD7AsynchronousEvent. The AD7AsynchronousEvent base class implements AD7’s IDebugEvent2 and returns EVENT\_ASYNCHRONOUS from IDebugEvent2::GetAttributes. This makes the debugger treat the event as asynchronous non-stopping.
3. The next then AD7EngineCreateEvent.Send does is call engine.Callback.Send. This is actually a thin wrapper around a call to IDebugEventCallback2 .Event. 
4. Since the event is sent non-stopping asynchronous, the debugger receives the event, and immediately returns control back to the debugger. No calls to Continue or Execute are made.
5. Once the call to Send returns, the debugger sends another event to notify the debugger about the program create.

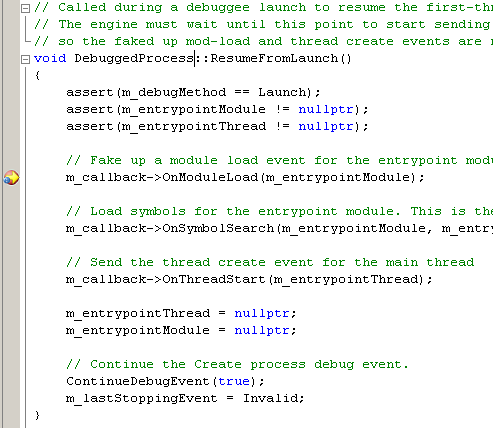
Remember the call to IDebugEngineLaunch2.ResumeProcess added a program node which resulted in the call to IDebugEngine2::Attach. Examining the callstack, you can see that ResumeProcess really did cause the debugger to call back into the Attach. If you look back at the implementation for IDebugEngineLaunch2.ResumeProcess, you can see it also makes a call back to the event thread. This case, it calls ResumeFromLaunch which allows the debuggee process to run:



The call to m\_pollThread.RunOperation takes an instance of Operation as a parameter. Operation takes delegate as a parameter to its constructor. This delegate is executed on the event thread on behalf of the request thread . See the discussion about OpertionThread.cs ThreadFunc on page five of this walkthrough.

In this case, the anonymous delegate being passed to Operation is executing the ResumeFromLaunch method of DebuggedProcess. To step through the call to ResumeFromLaunch:

1. Set a breakpoint in WorkerApi.cpp, ResumeFromLaunch and hit F5



1. The breakpoint in ResumeFromLaunch is now hit on the event thread (examine the request thread in the debugger shows that it is waiting for this call to complete.
2. ResumeFromLaunch sends a module load event to the SDM letting the debugger know that the main exe for the debuggee has loaded. This event is also sent asynchronous non-stopping.
3. ResumeFromLaunch notifies the SDM that a symbol search for the main exe has occurred and passes the results of that search. Remember: the sample engine only loads symbols for the processes exe and only searches the paths specified in the exe by the compiler.
4. Next, ResumeFromLaunch sends a thread start event to the SDM notifying it of the main thread in the applications
5. Finally, the create process debug event is continued allowing the debuggee to continue executing and the last stopping event is set to Invalid.
6. The stack now unwinds all the out of IDebugEngineLaunch2.ResumeProcess. The debuggee is now running with the debugger fully attached.
7. The debugger will now start receiving other debug events from WaitForDebugEvent and responding to user actions (such as binding breakpoints).