Visual Studio Sample Debug Engine Walkthrough 2 – Modules and Threads

*Jackson Davis – Software Engineer, Visual Studio Debugger*

This walkthrough describes how modules and threads are represented to the Visual Studio Debugger UI in the Visual Studio Debug Engine Sample available for download at <http://code.msdn.microsoft.com/debugenginesample>. This document is an extension to the first walkthrough of the sample available for download at the sample location. Walkthrough 1 contains a section describing the architecture of debug engines in Visual Studio and the architecture of the debug engine sample. This walkthrough assumes you are familiar with the concepts of that walkthrough and have setup the debug engine sample as described in the “Getting Started” section of Walkthrough 1.

# Module Loads

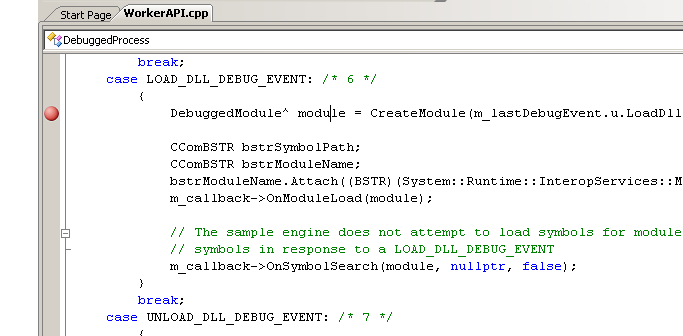
The Visual Studio Debugger becomes aware of the modules in the debuggee process when an instance of IDebugModuleLoadEvent2 is sent from the debug engine to the SDM. Visual Studio can retrieve module information after the event by using the module enumeration returned from IDebugProgram2::EnumModules.

The Win32 debugging API notifies the sample debug engine of a module load via WaitForDebugEvent. In this case, the DEBUG\_EVENT structure’s union field will be an instance of LOAD\_DLL\_DEBUG\_INFO. LOAD\_DLL\_DEBUG\_INFO contains several important bits of information about the module including the base address of the dll, the size of the dll, an open hFile to the dll’s module on disk.

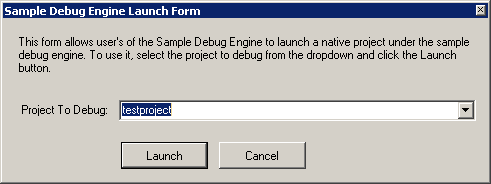
When WaitForDebugEvent returns a module load, the sample engine adds the module to its collection of modules and sends an instance of IDebugModuleLoadEvent2 to the SDM. This collection of modules is used by AD7Engine’s implementation of IDebugProgram2::EnumModules to construct a module enumeration when the UI requests it.

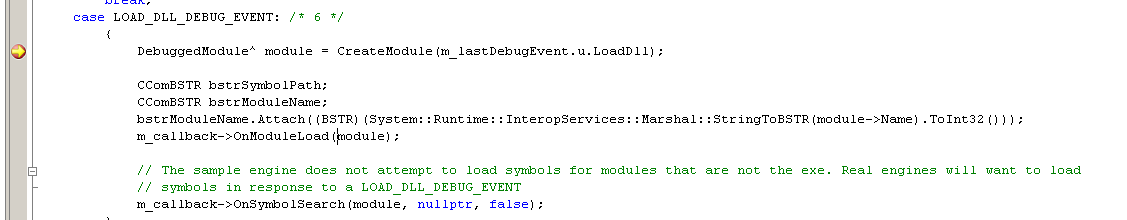
To step through a module load in the sample engine, open the debug engine solution in Visual Studio, open WorkerAPI.cpp in Microsoft.VisualStudio.Debugger.SampleEngineWorker, and navigate to DebuggedProcess::DispatchDebugEvent method. As discussed in Walkthrough1, DispatchDebugEvent is a large switch statement which switches on dwEventCode field of the DEBUG\_EVENT structure. For this portion of the walkthrough, we are interested in LOAD\_DLL\_DEBUG\_EVENT.

1. Set a breakpoint in the switch statement after the LOAD\_DLL\_DEBUG\_EVENT case statement



1. Hit F5 to start debugging . A new instance of Visual Studio should launch under the debugger.
2. Create a new Visual C++ project (call it testproject) and add some code to it.
3. Click on the Tools->ProjectLauncher menu option in the new instance of Visual Studio
4. The ProjectLauncher form should now launch:



1. Select your project in the dropdown and click on the Launch button
2. Visual Studio should now launch the Visual C++ project and enter debug mode (as described in Walkthrough1).
3. The breakpoint you set in DispatchDebugEvent should now get hit  
   

The sample debug engine will now create an instance of DebuggedModule which represents the module in the debuggee to the debug engine sample. After creating the instance of DebuggedModule, the sample debug engine sends the module load event and a symbol search event to the SDM.

If you step into the call to CreateModule from DispatchDebugEvent, you can see the sample engine goes to a lot of trouble to convert from the module base address and hFile to the full-path of the module on disk. Once it has obtained to module’s path, it reads in the IMAGE\_NT\_HEADERS from the PE header to obtain the file size in bytes. It then adds the module to two collections: the address map, which maps the address space taken up by the binaries code (base address + size), and the linked list of the modules in the debuggee.

Once the DebuggedModule object is loaded, two events are sent to the managed front-end: The mod-load event and the symbol search event. The handler for the mod-load (EngineCallback.cs OnModuleLoad) constructs an instance of AD7Module and places it in the Client field of the DebuggedModule. AD7Module implements the IDebugModule2, and IDebugModule3 AD7 interfaces.

Next, an instance of IDebugModuleLoadEvent2 is then sent to the SDM as an asynchronous debug event. This event is used to place a message about the module load in the Output window of the Visual Studio UI.

In the sample engine, IDebugModuleLoadEvent2 is implemented by AD7ModuleLoadEvent. The interesting method on IDebugModuleLoadEvent2 is GetModule which is called returns information about the loaded module and is used to display the message in the output window.

After sending the module load event, DispatchDebugEvent calls OnSymbolSearch notifying the front-end of the engine that symbols were not loaded for this binary. If you step into this call, you can see the sample debug engine sends an instance of IDebugSymbolSearchEvent2 to the SDM. This event results in another message in the Output window specifying if symbols were loaded for this binary or not. Since the sample debug engine only loads symbols for the primary exe, the message from this event usually only says “Symbols not loaded”.

As mentioned in Walkthrough1, the debugger does not receive a LOAD\_DLL\_DEBUG\_INFO for the primary exe in the application. Instead, the module load event is faked up during the process launch scenario. Walkthrough1 steps through the process launch scenario and demonstrates where symbols for this module are loaded.

# Thread Creates

Thread Creates are very similar to module loads. The debugger receives a DEBUG\_EVENT with the union field set to a CREATE\_THREAD\_DEBUG\_EVENT. CREATE\_THREAD\_DEBUG\_EVENT contains an optional hThread, a pointer to the thread’s thread local storage, and a pointer to the start address for the thread. The new thread’s thread-id is passed in the DEBUG\_EVENT structure. The handler for a thread create event coming out of WaitForDebugEvent is also in WorkerApi.cpp DebuggedProcess::DispatchDebugEvent. A new instance of DebuggedThread is created and the front-end of the debug engine is notified. DebuggedThread holds the thread identifier, a handle to the thread, and the thread’s starting address. The implementation of OnThreadStart in (EngineCallback.cs) creates a new instance of AD7Thread which implements IDebugThread2. It then sends IDebugThreadCreateEvent2 to the SDM as an asynchronous event. IDebugThreadCreateEvent2 results in a new entry in the output window letting the user know that the new thread is created. Just like module load, the thread create for the main thread in the application is sent during the process launch sequence (WorkerApi.cpp, DebuggedProcess::ResumeFromLaunch. see Walkthrough1 for more information). If you’d like to actually step through this thread create, your debuggee will need to create another thread.

# Enumerating Modules and Threads

When interacting with a debuggee’s threads and modules populating the Thread and Module windows, the Visual Studio Debugger UI will obtain enumerations for both collections. In the sample debug engine, these collections are created in AD7Engine.cs EnumThreads and EnumModules respectively. For threads, an implementation of IEnumDebugThreads2 is returned. For modules, an instance of IEnumDebugModules2 is returned. The objects contained in these enumerations are implemented in AD7Thread.cs and AD7Module.cs.

AD7Thread implements IDebugThread2. This interface allows the debugger to obtain information about the thread (name, thread id, suspend count, …) as well as obtain an enumerator of stack frames (IDebugThread2.EnumFrameInfo which will be described in a later walkthrough). IDebugThread2.GetThreadProperties is the method that returns general thread information. It takes a bit field that describes what information the UI is requesting and returns an instance of THREADPROPERTIES to the UI. The function signature actually returns an array of THREADPROPERTIES but this is to work-around an limitation in com-interop. For each field in the bit-field, the sample debug engine sets the appropriate field in the THREADPROPERTIES and then sets the equivalent field in the THREADPROPERTIES’ bit field to state that field is valid. A good example of this is when the bit for TPF\_NAME is set. The engine sets the bstrName field and then sets the TPF\_NAME field as follows:

if ((dwFields & (uint)enum\_THREADPROPERTY\_FIELDS.TPF\_NAME) != 0)

{

props.bstrName = ThreadNameString;

props.dwFields |= (uint)enum\_THREADPROPERTY\_FIELDS.TPF\_NAME;

}

Most of the time, the debugger will request multiple fields simultaneously.

Modules are very similar. AD7Module implements IDebugModule2 and IDebugModule3. IDebugModule2.GetInfo is to modules as IDebugThread2.GetThreadProperties is to threads. The implementation is also very similar. It takes in a bit-field and returns an instance of MODULE\_INFO. A good example is when it is asked for the location symbols were loaded from:

if ((dwFields & (uint)enum\_MODULE\_INFO\_FIELDS.MIF\_URLSYMBOLLOCATION) != 0)

{

if (this.DebuggedModule.SymbolsLoaded)

{

info.m\_bstrUrlSymbolLocation = this.DebuggedModule.SymbolPath;

info.dwValidFields |=  
 (uint)enum\_MODULE\_INFO\_FIELDS.MIF\_URLSYMBOLLOCATION;

}

}

If symbols are loaded, the m\_bstrUrlSymbolLocation field is set to the symbol path and the MIF\_URLSYMBOLLOCATION flag is set.

One interesting thing to point out about IDebugModule2::GetInfo is the MIF\_FLAGS field. This is used to return Boolean flags about the module back. The debug engine sample returns if symbols are loaded or not using MIF\_FLAGS:

if ((dwFields & (uint)enum\_MODULE\_INFO\_FIELDS.MIF\_FLAGS) != 0)

{

info.m\_dwModuleFlags = 0;

if (this.DebuggedModule.SymbolsLoaded)

{

info.m\_dwModuleFlags |= (uint)(enum\_MODULE\_FLAGS.MODULE\_FLAG\_SYMBOLS);

}

info.dwValidFields |= (uint)enum\_MODULE\_INFO\_FIELDS.MIF\_FLAGS;

}

The actual enumerators for threads and modules are implemented in AD7Enums.cs. There is a base class that implements the actual enumerators (AD7Enum<T,I>). Since each enumerator type is a separate interface in AD7 (IDebugEnumThreads2 and IDebugEnumModules2) there is a separate class for each type.

That completes the walkthrough of Threads and Modules in the sample debug engine. Hopefully, this will be useful in getting you started investigating this portion of the engine.