Concord Additional Download Readme

Contents

[Overview 1](#_Toc320797640)

[Introduction to Concord 2](#_Toc320797641)

[What is Concord? 2](#_Toc320797642)

[Architecture 3](#_Toc320797643)

[Extensibility Scenarios 6](#_Toc320797644)

[Component discovery and configuration 6](#_Toc320797645)

[Component Levels 6](#_Toc320797646)

[Navigating the Concord API 7](#_Toc320797647)

[Concord Threading 8](#_Toc320797648)

[Data Container API 9](#_Toc320797649)

[Creating and Closing objects 9](#_Toc320797650)

[AddRef/Release Semantics 10](#_Toc320797651)

[Description of Included Files 10](#_Toc320797652)

[HelloWorld sample 13](#_Toc320797653)

[Browsing the Hello World project 13](#_Toc320797654)

[Running the Sample in Glass 14](#_Toc320797655)

[Running the Sample in Visual Studio 14](#_Toc320797656)

[Additional information on glass2.exe 15](#_Toc320797657)

[Logging from the engine 16](#_Toc320797658)

# Overview

Starting with Visual Studio 2012, Visual Studio includes the Concord debug engine, a new debug engine for debugging native, managed, script, and GPU code. A debug engine is a pluggable backend to the Visual Studio debugger, and the debug engine extension point has existed in Visual Studio for many releases. Concord is the first Microsoft Debug engine which exposes a large and extensible API.

This download is aimed at assisting developers that wish to plug into Concord. The Visual Studio SDK already includes support for building Concord components. This download provides additional content to developers targeting Concord – additional tests, samples, and documentation.

# Introduction to Concord

## What is Concord?

As stated in the overview, Concord is a new debug engine that ships with Visual Studio starting with Visual Studio 2012. The basic design of Concord was that instead of having the debug engine as a monolithic entity; instead break it up into different parts. These parts would communicate through a new API. This new API would support transparent managed/native marshaling so that the various parts could be implemented in managed or native code, remoting so that the debugger could have a simple remote debugging story that didn’t depend on DCOM, and an API which is largely public so that there could be a great extensibility story without a lot of additional cost to Microsoft.

## Architecture

To kick off an overview of the Concord architecture, here is a simplified architectural diagram for Concord.

Debugger UI & SDM

Dispatcher

AD7 AL

Stack Provider

Breakpoint Manager

PDB Stack Walker

C++ EE

PDB Symbol Provider

Remoting boundary

Stack Merger

Stepping manager

Shim C#/VB EE

Managed DM

Native DM

Win32 Base DM

Target Application

At the top of this diagram, there is the Debugger UI & SDM, which make up the front end of the VS debugger architecture. The SDM/UI interact with Concord through the debug engine interfaces which Concord implements (IDebug\* interfaces). These components interact with the AD7 AL, which is the adaptation layer between the debug engine APIs (known as AD7) and the Concord APIs.

On the left side of the diagram is the box for the Dispatcher. The Dispatcher knows nearly nothing about debugging, but instead it is a routing, loading, and remoting service used by the rest of Concord. The dispatcher exposes the Concord API. When a Concord API method is called, the dispatcher finds the best implementation of this API, loads up the implementation, and calls this component through an interface. If the caller and implementer are on different computers, the Dispatcher remotes the method call to the correct computer. If the method is implemented in managed code, the Dispatcher marshals the call from the native Concord API universe to the managed Concord API universe.

In the middle of the diagram are a bunch of blue boxes. Each blue box represents a Concord ‘Component’. Components are the basic unit of extensibility in Concord. Each component has a configuration file that describes to the dispatcher what interfaces it implements along with a filter to indicate when this interface should be used. As an example, there is an interface for setting breakpoints. This same interface is implemented by the Managed DM for setting breakpoints in managed code, the Script DM for setting breakpoints in JavaScript code, and the Win32 Base DM for setting breakpoints in native code. Each implementation has a different filter so that the dispatcher can pick the correct implementation depending on where the breakpoint is being set.

Microsoft provides more components than could fit on a diagram or that would make sense to describe, but here is a quick overview of some of the more important components from an extensibility perspective. This description tries to illustrate the granularity that Microsoft used when creating components and therefore the granularity with which the API was created.

**Stack Provider**: Responsible for building up the call stack to the form where it is ready to display to users. To do this, the stack provider makes requests to other components to walk the stack, filter the stack to include appropriate descriptions and to hide frames that should be hidden, and lastly obtain the frame description string in whatever language the frame was implemented.

**Breakpoint Manager**: Responsible for managing the various user-level breakpoint requests. It will match them up to symbols as modules are loaded and send information down to one of the Debug Monitors in order to add or clear breakpoints.

**PDB Stack Walker**: Responsible for walking call stacks through x86 code which has a PDB loaded. It will use data from the PDB and the stack walk APIs from the MSDIA dll to walk the call stack.

**C++ EE**: Expression Evaluator (EE) for native C++ code output by the Microsoft C++ compiler. Expression evaluators are responsible for evaluation and formatting in the various evaluation windows in the debugger (watch, immediate, data tips, etc) as well as formatting frames in the call stack. To make this work, the expression evaluator relies heavily on debug info from PDB files. It obtains this information from the MSDIA interfaces, which it in turn obtains from the PDB symbol provider.

**PDB Symbol Provider**: Responsible for loading PDB files, implementing the Concord symbol provider API using information in these PDB files, and for providing the MSDIA API to other Concord components. The Concord symbol provider API covers a few common symbol operations that various other Concord components need such as mapping between source lines and symbols. For more advanced scenarios, such as the data inspection needs of the PDB stack walker and the C++ EE, the PDB Symbol Provider will instead provide the MSDIA interfaces.

**Stack Merger**: In many cases, such as managed code running under the .NET runtime, or native code running in an ARM or x64 process, it is possible to walk the call stack without access to symbols. The stack merger is responsible for working with components that walk the stack using this runtime data, and sending back the complete results to the stack provider.

**Stepping Manager**: Responsible for coordinating stepping operations across the various runtime environments loaded in the target process.

**Shim C#/VB EE**: This is an adaptation layer between the Concord expression evaluator, object inspection and symbol APIs and the portions of the corresponding legacy APIs used by vbdebug.dll and cscompee.dll.

**Managed DM**: This is the Debug Monitor (DM) responsible for inspecting low-level state, and controlling execution of managed code running in the target process. Debug monitors are the lowest-level debugging components in the Concord architecture. There are two categories of debug monitor:

* *Runtime debug monitors*: There is a runtime debug monitor for each type of execution environment (known as a ‘runtime instance’ in the Concord API) which is loaded and debugged in the target process. So for, example, if a target process had loaded native and .NET Framework code, and both were being debugged at the same time, then the target process would have two runtime instances (native and managed), and two runtime debug monitors would be active.
* *Base debug monitors*: The base debug monitor is the component that owns the underlying connection to the target process. Unlike runtime debug monitors, there can be only one base debug monitor. Base debug monitors provide the lowest level primitives such as reading or writing memory or registers, and basic execution control.

For Visual Studio 2012, the Managed DM only functions as the Base debug monitor for debugging managed code running under the .NET Framework. In a future version, Concord will support managed/native interop debugging, and in such scenarios, the managed DM will function as a runtime debug monitor.

**Native DM**: Runtime debug monitor for native code built using the Visual C++ compiler. It builds on top of the base DM APIs to implement source level stepping, and C++ exception decoding among other things.

**Win32 Base DM**: Base debug monitor which is implemented on top of the Win32 debugging APIs (ReadProcessMemory/WriteProcessMemory/WaitForDebugEvent, etc).

## Extensibility Scenarios

TODO

## Component discovery and configuration

As mentioned previously, the basic unit of extensibility in Concord is a component. A comment contains code (compiled dll) and a configuration file which describes when the code should be used. The configuration is in a ‘.vsdconfig’ file which is shipped with the dll. Vsdconfig files are a binary format which is created with a tool that ships in the Visual Studio SDK (vsdconfigtool.exe). Vsdconfigtool.exe takes an XML input file (.vsdconfigxml) that contains this configuration information, it validates the configuration, and then spits out the .vsdconfig file.

Concord dynamically discovers the set of components which are available when it is first initialized in the Visual Studio process. It does so my looking for .vsdconfig files. The recommended way to ship .vsdconfig files is by including it in a VSIX package. The other option is to install it into the debugger’s directory (<vsinstallroot>\Common7\Packages\Debugger). Vsdconfig files do NOT need to be installed on the remote computer – the debugger will automatically transfer them over the wire. However, implementation dlls which must be loaded on the target computer need to be installed next to msvsmon.exe (dlls aren’t automatically transferred).

Configuration files contain the following information –

* It contains a GUID to identify the component. Use a GUID generation tool (ex: Tools->Create GUID in Visual Studio) to create a new value.
* It contains the information needed to load the implementation class(es) in the component. If the class is implemented in native code, this is the name of the dll, and the CLSID value. If the class is implemented in managed code, this is the assembly and class name.
* It contains the ‘component level’ that the component will load at. See the next section for more information.
* It contain the set of interfaces that each class implements, along with filters to indicate the scenarios where the class loads

The ‘Hello World’ sample gives an example of a configuration file, which is a good place to start in better understanding things. The vsdconfigxml format also has an annotated schema file (vsdconfig.xsd) which provides additional documentation (as well as validation when editing the file).

### Component Levels

In Concord, each component has a numeric value associated with it called a component level. This level tells the Dispatcher two things –

1. When remote debugging, will this component load on the target computer, or the Visual Studio computer? Components above 100,000 are IDE components.
2. When sending events or other broadcast notifications, the component level controls the order various components are notified. For events,

There are two general rules to follow when selecting a component level:

1. Select a level higher (larger number) than dependent components. For example, if the target process had an interpreter, and this interpreter was implemented in managed code, then a debug monitor which is built to add a debugging experience for the interpreter must have a higher component level than the Managed DM.
2. Follow the advice in the following chart

|  |  |  |
| --- | --- | --- |
| **IDE Computer Component Levels** | | Values > 100,000 |
|  | AD7 AL | 1,000,000,000 |
|  | Disassembly Provider | 9,998,000 |
|  | Stack Query – Components that wish to query the call stack | 9,997,000 |
|  | Stack Provider | 9,996,000 |
|  | Stack Filter - level where stacks can be filtered and annotated | 9,995,000 |
|  | Breakpoint Manager | 9,994,000 |
|  | IDE Expression Evaluation | 9,992,000 |
|  | Symbol Stack Walkers – stack walkers that need access to symbols | 9,991,000 |
|  | IDE SymbolProvider - Components which provide symbol information to rest of the debugger. The symbol path should not be used below this level. | 1,999,000 |
| **Target Computer Component Levels** | | Values < 99,999 |
|  | Monitor Symbol Provider – Symbol providers when the symbolic state is built on the target computer (ex: interpreter, dynamically compiled/emitted code) | 75,000 |
|  | Breakpoint Condition Processor - This level is for processing breakpoint conditions such as condition expressions and hit counts. Below this point all physical breakpoint events will be visible regardless of whether or not they have false conditions. | 70,000 |
|  | Monitor Task Provider – This is the level for task data mining in the target process | 65,500 |
|  | Monitor Expression Evaluator | 65,000 |
|  | Monitor Coordination – Components which arbitrate between the various monitors for stepping, breakpoints by native address, stack walk, etc. | 60,000 |
|  | Monitor Stack Walkers | 55,000 |
|  | Custom Debug Monitor – Reserved for third party debug monitors which wish to make use of services provided by the standard debug monitors. | 40,500 |
|  | Runtime Debug Monitor - Provides data inspection and execution control for managed/native/script code | 40,000 |
|  | Base Debug Monitor | 10,000 |
|  | Base Debug Monitor Services – provides utility services to base debug monitors (ex: process creation) as well as pre-debugging services (ex: process enumeration) | 1,000 |

## Navigating the Concord API

The Concord API is large enough that it can be a bit overwhelming at first glance. Here are a few hints which may make navigating the API easier –

1. When deciding what interfaces to implement, we have organized all the interfaces by the category of component that implements the interface. If you look in the native API header file (vsdebugeng.h) you can see each interface listed by section, as well as a description of what that category of component does.
2. When looking for methods to call, ignore interfaces. The Concord API is exposed to callers as methods on classes rather than interface methods.
3. Findstr in vsdebugeng.h: even if you are implementing a Concord component in managed code, sometimes the fastest way to find the method that you want is to look for strings in vsdebugeng.h. Since the API is the same between native and managed code, if you find the native API, it will work in managed also.
4. Browse the references in Microsoft.VisualStudio.Debugger.Engine – this is the flip side of #3, even if you are consuming the Concord API from native code, Visual Studio ships really nice support for browsing through managed APIs.

## Concord Threading

**Considerations when calling the API:** Most of the Concord API may be called on any thread. However, the Dispatcher requires knowing the current component. So in order to use the Concord API from a thread that your code creates, or from some sort of Visual Studio worker thread (not the main thread, and not one of the debugger threads), the thread must register with the Dispatching using DkmComponentManager.InitializeThread.

One set of methods that cannot be called on any thread, are the various methods that raise events. These APIs should generally only be called on the event thread. These will fail when called from another thread.

The only other restriction on what thread can call a method is that there are a few APIs that either accept or return COM objects. These APIs either require that they are called from the MTA (for interfaces which are MTA-only) or Visual Studio’s main STA (for interfaces which are implemented there).

**Consideration when implementing Concord interfaces**: Concord will nearly always call implementations on one of two threads – the request thread, which handles requests coming from the IDE, and the event thread, which handles debug events coming from the target process. This said it is possible for callers to create their own threads, so implementers shouldn’t assume that only the two threads are possible.

For components that wish to prevent reentrancy, the component can set ‘Synchronized="true"’ in its component configuration. This will instruct the dispatcher to essentially place a lock around the component which it will enter/leave before calling the component. Such components can use DkmComponentManager.AllowComponentReentrancy/ DisableComponentReentrancy if there are scenarios where they need to temporarily leave and reenter the lock.

## Data Container API

One of the basic building blocks used through Concord is the Data Container API. This API is implemented in the DkmDataContainer class and a few other related types. The basic idea of the data container API is to allow ‘virtual fields’ to be added to all the important objects. This is similar to the 'expando' concept JavaScript, though it doesn’t have the same type safety or namespace pollution problems that JavaScript expandos have.

The API works by having components add ‘data items’ to a data container object. The caller can then obtain the data item back when they need to. Thus the implementation class for a Concord component is usually has little state, and instead uses the data container objects to do the work of holding onto the state. This ensures that state is uniformly cleaned up across Concord rather than having a situation where each component is responsible for finding an appropriate place to clean up their state. For this reason, components generally do not need to remove their data items but can rather just wait for the data items to be automatically removed when the container object is closed.

More information on this subject is in the Hello World sample as well as in the API documentation.

## Creating and Closing objects

All objects that can be created through the Concord API are created through a static ‘Create’ method that the object’s class exposes.

Objects that represent entities in the target process (DkmProcess, DkmThread, DkmRuntimeInstance, etc) will fire a create event as part of their implementation of this ‘Create’ method. Various components can receive this creation event by implementing the appropriate interface. For these objects, the Create method must be called on the event thread, and should be called when the corresponding action happened in the target process.

All objects that are data containers (derive from DkmDataContainer) have a ‘Close’ method that should be called when the object is no longer valid. Having an explicit ‘Close’ is required because the logical object can exist on multiple computers, and can exist in both managed and native code, so relying on the refcount variable (native code) or the GC (managed code) is not an explicit enough gesture to indicate that the logic object should go away (though these techniques are used to indicate that the memory backing the physical object can be cleaned up).

Comments in the Create and/or Close method indicate which component should be responsible a given object.

In some cases, the close method is internal and therefore will not show up in the public API. In which cases, you don’t need to worry about closing the object – this will happen automatically by Microsoft-provided components at the appropriate time. For example, DkmProcess, DkmThread, DkmRuntimeInstance and DkmModuleInstance objects are automatically closed after the object’s unload event has finished being processed.

Dispatcher objects are chained so that most objects have a parent. For example, the object that represents a target process (DkmProcess) is the parent object for threads in that process (DkmThread). When the parent object is closed, children are implicitly closed as well.

Components that wish to be informed when an object is closed can add a data item to the object. The data item class will then be notified through IDkmDisposableDataItem (when implementing the data item in native code) or by overriding DkmDataItem.OnClose (when implementing the data item in managed code).

## AddRef/Release Semantics

NOTE: This section applies only when consuming the Concord API from native code.

The native Concord API adheres to standard COM conventions for AddRef/Release, meaning that if an object is returned as the return value, this is done without increasing the ref count on the object. If instead the object comes back as an out param, then Release should be called. The dispatcher will guarantee that any input object to a Concord interface method will stay valid until after the implementer returns from the method, and it will also guarantee that if an object provides an accessor method which returns another object, then this return value is immutable and doesn’t need to be AddRef’ed/Release’ed (though doing so will certainly not hurt anything).

Examples to illustrate these rules:

{

DkmThread\* pDkmThread = pDkmProcess->Thread();

pDkmThread->Release(); **// BUG: pDkmThread should not be Released**

}

{

DkmThread\* pDkmThread;

pDkmProcess->FindSystemThread(12, &pDkmThread);

} **// BUG: pDkmThread is leaked; pDkmThread->Release() must be called**

{

// Correct, but the extra AddRef/Release that CComPtr will add isn't required

// as long as the code isn't somehow releasing its reference to DkmProcess

CComPtr<DkmThread> pDkmThread = pDkmProcess->Thread();

}

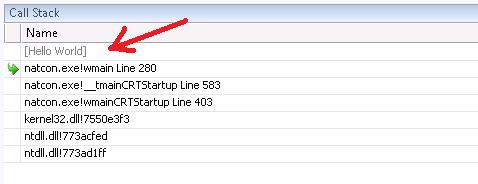
# Description of Included Files

The following files are either included in this download, or Concord related files which ship in the Visual Studio SDK.

|  |  |
| --- | --- |
| **File** | **Description** |
| inc\vsdebugeng.h | Native C++ header file declaring the native version of the Concord API. This file also includes the complete documentation. |
| Inc\vsdebugeng.templates.h | Native C++ header file containing templates to aid in using the Concord API. |
| Lib\Win32\VSDebugEng.lib | X86 Import library for vsdebugeng.dll. Native C++ clients of the Concord API link to this library. |
| Reference Assemblies\ Microsoft.VisualStudio.Debugger.Engine.dll | Managed reference assembly for the Concord API. Managed clients/extensions compile against this assembly. |
| Reference Assemblies\ Microsoft.VisualStudio.Debugger.Engine.xml | Documentation file for the managed Concord API. |
| Samples\HelloWorld\\* | HelloWorld Concord sample – provides an introduction to writing Concord components. |
| Tools\SetEngineLogMode.cmd | Batch script to enable/disable/dump logging options. See ‘Logging from the Engine’ section for more information. |
| Tools\vsdconfigtool.exe | Tool for processing Concord component configuration files (.vsdconfigxml files). This tool is used by the build process of a Concord extension. |
| Tools\Microsoft.VSDebugger.targets | Defines the build rules to use vsdconfigtool.exe during the build of a managed Concord extension. |
| Tools\Microsoft.VSDebugger.Native.targets | Defines the build rules to use vsdconfigtool.exe during the build of a native Concord extension. |
| Tools\glass\\* | Test environment for Visual Studio debugger extensions. This tool is used to run all of the tests that come with this SDK. It is also hooked up to the Concord samples as the ‘program to launch’ during Debugger launch. This provides extension developers a fast edit-compile-debug cycle. |
| Tools\glass\RegisterGlass.cmd | Batch script to register the glass tool (test environment for Visual Studio debugger extensions). |
| Tools\ProjectLauncher\\* | Visual Studio Addin to launching projects through custom debug engines.  Concord supports being associated with custom engine GUIDs where a new GUID is added to the Visual Studio registry hive ($(RegRoot)\ad7metrics\engines\{guid\_value}), and then you can configure your components to be used with this new engine GUID. The project launcher can be used while boot strapping this effort to quickly launch your custom debug engine without needing to first develop a new project system to direct the debugger launch command. |
| Schema\vsdconfig.xsd | Schema Definition for the custom XML language used to define the configuration of Concord components (.vsdconfigxml file schema). |
| Tests\RunTests.cmd | Batch script for running the tests that are included in this SDK. |
| Tests\\*\TestScript.xml | Various tests of Concord. Extension-authors can run these tests to ensure that they haven’t broken core Concord scenarios and as a template for additional tests specific to their own component. |
| Tests\Debuggees\bin\\* | Debuggees (target processes) used by one or more of the tests. |
| Tests\Debuggees\src\\* | Source code and project used to create the debuggee. |

# HelloWorld sample

To demonstrate the basic parts of a Concord component, this SDK starts with the same sample that every new platform starts with – hello world. In the case of Concord, this means adding an annotated ‘Hello World’ frame to the call stack:



## Browsing the Hello World project

This download includes two different HelloWorld samples – one written in native code (native C++), and one written in managed code (C#). They both are found in the ‘<DownloadRoot>\Samples\HelloWorld’ folder. The Concord API is nearly identical between managed and native code, and any extension may be written in either managed or native code. So open up whichever sample you are most likely to develop your extension in.

The samples include a VSIX project for deploying the sample into an experimental Visual Studio instance. This requires the Visual Studio 2012 SDK, which can be found at <http://www.microsoft.com/en-us/download/details.aspx?id=30668>. If you would like to open the project without installing the SDK, open the project file from the ‘.dll’ folder instead.

Let’s look at some of the more interesting files in the sample:

|  |  |
| --- | --- |
| **File** | **Description** |
| \_HelloWorldService.cpp/.h/.cs | The one and only public (exported) class in the sample. In Concord, classes implement interfaces, and Concord calls into these classes through the implemented interfaces. The sample implements the IDkmCallStackFilter interface on this class.  In native code, Concord uses COM activation (DllGetClassObject) to obtain this object. So this class includes the ATL glue for hooking up to DllGetClassObject. Note: while Concord uses the standard DllGetClassObject approach, it is not calling CoCreateInstance, so unlike standard COM, in Concord your CLSID does not need to be registered.  In managed code, this class is created through reflection. So the class is simply marked as public, and has a default public constructor.  This class defines one interface method – FilterNextFrame. FilterNextFrame is responsible for creating the new ‘[Hello World]’ frame and for echoing back all other frames to its caller. |
| HelloWorldDataItem.cpp/.h/.cs | FilterNextFrame is called once for each frame in the call stack. However it needs to know which frame is the first frame so that it can be sure to add the ‘[Hello World]’ frame exactly once for each call stack. To do so, the sample needs to associate a bit of data with each call stack.  This data item class is the mechanism that Concord uses to associate information with a Concord object. Most of the important Concord objects (including DkmStackContext in this sample) derive from DkmDataContainer. DkmDataContainer allows any Concord component to store its own private information in a Concord object.  In the sample, this data item class uses a ‘State’ enum to keep track of if the ‘[Hello World]’ frame has been added yet. |
| HelloWorld.vsdconfigxml | HelloWorld.vsdconfigxml is an XML file which describes the hello world component – what classes does it have, which interfaces does each class implement, when should these interfaces be called. The sample’s build process runs a custom tool over this file (vsdconfigtool.exe) which validates this file, and turns it into a binary file (HelloWorld.vsdconfig) which is deployed along side HelloWorld.dll. The Concord component loading system then loads all these .vsdconfig files on startup so that it can route requests to each registered component. |
| TestScript.xml | Finally, the sample contains a test script which can be used to test the sample. This test script is run by glass2.exe, a test environment for Visual Studio debugger extensions. The sample project has been configured to automatically run this test script as the Debugger launch action. So if you start debugging this solution, Visual Studio will launch glass2.exe, directing glass to execute this script. |

## Running the Sample in Glass

This SDK includes glass2.exe, a test environment for Visual Studio debugger extensions.

Before glass can be run, it needs to be registered. Open a command prompt with administrative privileges then run ‘<path\_to\_download>\Tools\Glass\RegisterGlass.cmd’ to register glass.

After registering glass, simply build the sample project and hit F5 to launch it under the debugger. Set breakpoints and step through the code to better understand how the sample works.

## Running the Sample in Visual Studio

After you have finished developing the sample in glass, it is time to try it out in Visual Studio.

To do so:

1. Right click on the ‘vsix’ project and use ‘Set As StartUp Project’.
2. Hit F5 to launch an experimental instance of Visual Studio under the debugger.
3. In the experimental instance of Visual Studio:
4. Open or create a C++ project
5. F10
6. When the project finishes launching, open the Call stack window and notice that there is now a ‘Hello World’ frame on top.

# Additional information on glass2.exe

As mentioned previously, this SDK includes glass2.exe, a test environment for Visual Studio debugger extensions. This section includes additional information on using glass.

First, this SDK contains a batch file for running the tests contained in this SDK – RunTests.cmd. Run this script with no arguments to run all the tests.

Each test in the SDK works by running glass2.exe against an XML input script. This input script tells glass what commands to run, and what should happen as a result of running the command. Glass can then produce two different output files – an error log which contains differences between the expected values and the observed values, and a session log which may be helpful in understanding what happened. It’s possible to determine if tests pass or fail by either diffing the error file against an error file which lists no errors, or by checking the exit code (if the ‘-err’ command line option was used).

To debug a failure in glass, it is recommended to run glass under the debugger. The sample includes the recommended set of command line arguments for this: ‘-f TestScript.xml -e ErrorLog.xml -q -err -diag'. If you bring up the output window, you will find various ‘GLASS2’ lines to help trace through the target process and the script to understand what went wrong. If you double click on a line, the output window will open the file and bring it to the right spot. Example output:

c:\ConcordSDK\Tests\nat1\TestScript.xml(3,4) : GLASS2 : Executing 'launch native ..\Debuggees\bin\nat1.exe'

c:\ConcordSDK\Tests\Debuggees\src\nat1\nat1.cpp(17,1) : GLASS2 : Stopping event 'IDebugEntryPointEvent2'

To understand glass2 scripts, a good place to start is the script for the HelloWorld sample. This script has been commented. One important concept is an expected event – expected events are used to prevent the script from moving on to the next command before the previous command has completed. Expected events are declared with the 'expected="True"' attribute.

To develop new glass2-based tests, it’s helpful to run glass in interactive mode where glass acts as a simple console-UI debugger. To do so, run glass2 without an input script (no ‘-f <script\_name>’ command line argument). From this mode you can use the ‘help’ command to list out all the available commands or to obtain more information about a given command. If you pass the ‘-s SessionLog.xml’ command line argument, you can also use the session log as the basis for the glass script.

# Logging from the engine

vsdebugeng.dll contains builtin logging support which is activated through the registry. The SDK contains a batch file to make this easy – tools\SetEngineLogMode.cmd. Three modes of operation are available:

Default mode – significant errors are set to the debug output when a debugger is attached to Visual Studio (or whatever process has loaded vsdebugeng). Default mode is what one gets after a clean install of Visual Studio.

Error mode – significant errors are set to %tmp%\vsdebugeng.dll.log if no debugger is attached to visual studio. If a debugger is attached to Visual Studio, default mode and error mode are the same.

Method mode – In addition to the tracing in ‘error’ mode, method entry/exit tracing is provided. Four different types of output are sent -- ‘CALL’ lines are used when an implementation was successfully loaded and is about to be called. ‘RETURN’ lines are used when a called component returns successfully. If the operation fails, a ‘RETURN ERROR’ line will appear instead. If no implementation could be found or if the implementation fails to load a ‘CALL ERROR’ line will appear. In addition to this, after a CALL/CALL ERROR line, there are ‘Skipped’ lines explaining why components are not called.

Example method logging output is below.

CALL: IDkmRuntimeMonitorBreakpointHandler.EnableRuntimeBreakpoint (ThreadId=6080 Class=Win32BDM.CBaseDebugMonitor IP=0x577D3B00 Object=0x10BD4F90 TickCount=217496250 ComponentId={F50FC269-4428-4680-8A45-462B8C5D37CD})

Skipped: ManagedDM::CCommonEntryPoint {38A59583-E6B1-4EE4-A53C-133BE0F45E55} for object visibility

Skipped: MinidumpBDM.CBaseDebugMonitor {016426E1-A85F-4CAB-941A-7C5EB5C82DC4} for 'BaseDebugMonitorId' filter

RETURN: IDkmRuntimeMonitorBreakpointHandler.EnableRuntimeBreakpoint (hr = 0x0, ThreadId=6080, TickCount=217496250)

Definition for terms in the log:

|  |  |
| --- | --- |
| IP | Instruction pointer of the method being called |
| Class | Name of the class as defined in the .vsdconfigxml file |
| ComponentId | GUID value for the component as defined in the .vsdconfigxml file |
| Object | Pointer to the DkmObject that was called |
| TickCount | Value returned by GetTickCount() at the time of the log |
| hr | HRESULT code for the error. Non-system error codes are generally defined in vsdebugeng.h. |