The Power Threading Library
Brought to you by
Jeffrey Richter and Wintellect
(Last update: October 23, 2010)

The Power Threading Library consists of a number of classes to assist you with building responsive, reliable, and scalable applications and components. There are also a bunch of general-purpose utility classes in this library that have nothing to do with threading but I threw them in here anyway. This document provides an overview of the namespaces and classes defined within the library.

Note that there are three versions of this DLL library (and accompanying documentation/debug files):

Usage	Files
For Desktop/Server CLR applications	Wintellect.Threading.dll Wintellect.Threading.xml Wintellect.Threading.pdb
For SIlverlight applications	Wintellect.Threading.Silverlight.dll Wintellect.Threading.Silverlight.xml Wintellect.Threading.Silverlight.pdb
For Windows Phone applications	Wintellect.Threading.WindowsPhone.dll Wintellect.Threading.WindowsPhone.xml Wintellect.Threading.indowsPhone.pdb

Due to Silverlight's security model, its subset of the .NET Framework's class library, and the nature of SIlverlight applications, the Winellect.Threading.Silverlight.dll contains a subset of the functionality offered by the Wintellect.Threading.dll. This means that some of the types described below are not available in the Silverlight version of the DLL. Of course, this also makes the Silverlight version of the DLL significantly smaller in size which is ideally suited for Silverlight's downloading nature.

For similar reasons, the Windows Phone version of the library, Wintellect.Threading.WindowsPhone.dll, also contains a subset of the functionality offered by the Wintellect.Threading.dll. Furthermore, since I personally do very little Windows Phone development, this version of the library has not been rigorously tested. The Windows Phone version is offered AS IS. Certainly, I appreciate any feedback on the library and there is a good chance that I will fix any bugs or make good suggested improvements. All three versions are produced from the same source code base.

The Wintellect.Threading.AsyncProgModel Namespace

I suspect that most developers will find the types defined in the Wintellect.Threading.AsyncProgModel namespace to be the most useful. This namespace defines a number of classes that are directly related to threading and the CLR's Asynchronous Programming Model (APM). Chapter 27 of my CLR via C#, 3rd Edition book (Microsoft Press, 2010) goes into great detail about the CLR's APM. Specifically, the types in this namespace aid developers who are defining their own types that support the APM. Types in this namespace can also be used by developers who need to invoke several asynchronous requests and coordinate their response.

The AsyncEnumerator class is particularly useful for developers who are trying to build responsive, reliable, and scalable applications and components. This class allows developers to leverage all the features of the APM using the simpler, more-familiar synchronous programming model. In fact, the AsyncEnuemrator class allows programmers to take their existing synchronous code and apply just a few changes to it in order to convert it into asynchronous code thereby reducing the number of threads and context switches required in their application and components.

In addition, my AsyncEnumerator class offers many productivity features:

- The ability to coordinate many concurrently executing asynchronous operations.
- Coordinate the cancellation and timeout of asynchronous operations
- Optionally use SynchronizationContext for callbacks so that marshalling is not necessary for GUI applications (such as Windows Forms and Windows Presentation Foundation) and also ASP.NET web applications.

The AsyncEnumerator class works with any class that supports the IAsyncResult APM this includes all Stream-derived class (such as FileStream and NetworkStream), Pipes, all WebRequest-derived classes, DNS, Socket, SqlCommand, all delegates, and more. It also works in all application models including Console apps, Windows Forms, Windows Presentation Foundation, ASP.NET Web Services and Web Forms, Window Communication Foundation, NT Services, and more.

The AsyncEnumerator offers many usage patterns. Some common usage patterns are listed below:

- Issue one asynchronous operation and process it when it completes
- Issue many asynchronous operations and process all of them when all of them complete
- Issue many asynchronous operations and process each of them as each completes
- Issue many asynchronous operations, process some and discard the rest.
- Issue many asynchronous operations, process some until cancellation or timeout occurs; discard the rest.
- Call asynchronous subroutines allowing for composition of asynchronous routines.
- When used with the SyncGate class, several AsyncEnumerators that are running concurrently can access shared data in a thread-safe reader/writer fashion without ever blocking a thread.

Documentation for the AsyncEnumerator can be found here:

- http://channel9.msdn.com/posts/Charles/Jeffrey-Richter-and-his-AsyncEnumerator/
- http://channel9.msdn.com/shows/geekSpeak/geekSpeak-recording-The-PowerThreading-Library-for-Silverlight-with-Jeffrey-Richter/
- http://msdn.microsoft.com/en-us/magazine/cc163323.aspx
- http://msdn.microsoft.com/en-us/magazine/cc546608.aspx
- http://msdn.microsoft.com/en-us/magazine/cc721613.aspx

Type Name	Type Description
AsyncEnumerator	Use this class to execute asynchronous operations using a synchronous programming model via C#'s iterator feature.
SyncGate	Used to have multiple AsyncEnumerator objects access shared data in a thread-safe way. The technique employed by this class is similar to that employed by my ReaderWriterGate class (see the Wintellect.Threading.ReaderWriterGate namespace below).
EventApm	Use this class to turn an event-based asynchronous programming model into the IAsyncResult-base programming model. This class is typically used to integrate classes that use the Event-base APM with the AsyncEnumerator.
AsyncResult AsyncResult<TResult>	Use these classes to help you implement the APM on one of your own types. http://msdn.microsoft.com/en-us/magazine/cc163467.aspx
CountdownTimer	This class offers a countdown timer that adheres to the APM.
ApmWrap	This class contains a static SyncContextCallback method that wraps an AsyncCallback method with the calling thread's SynchronizationContext

	so that the method is invoked using the application model's threading model. I explain this method on page 772 of my CLR via C#, 3rd Edition book.
ApmWrap<T>	This class allows you to associate an arbitrary piece of data with any IAsyncResult object.

The Wintellect.Threading.ReaderWriterGate Namespace

This namespace defines my ReaderWriterGate synchronization primitive. For details on how to use this type and how it works, please see the November 2006 issue of MSDN Magazine's Concurrent Affairs column which discusses it.

Type Name	Type Description
ReaderWriterGate ReaderWriterGateCallback ReaderWriterGateReleaser	Construct a ReaderWriterGate and then queue into it methods matching the ReaderWriterGateCallback delegate. Your callback method will receive a RederWriterGateReleaser object.

The Wintellect.Threading.ResourceLocks Namespace

This namespace defines several classes that allow locking of resources/objects. For details on how to use all of these related types, please see the June 2006 issue of MSDN Magazine's Concurrent Affairs column which discusses them. In short, ResourceLock is an abstract base class that defines a reader/writer lock programming model. Throughout all your code, you think about your locks using reader/writer semantics and you code to this abstract base class. Then, I also provide several concretes classes derived from ResourceLock. In your program, you decide which of the concrete classes to construct and you can change from class to another easily without affecting the rest of your code.

Type Name	Type Description
ResourceLock	This abstract base class defines the object model for all the other types derived from it. This class allows you to always code using reader-writer lock semantics and then select or change the actual lock at a later time.
NullResourceLock	This lock performs no locking; it allows you to remove thread synchronization without modifying any code.
MonitorResourceLock	This lock wraps the .NET Framework's Monitor class.
OneManyResourceLock	This lock wraps my reader/writer lock presented in the June 2006 issue of MSDN Magazine's Concurrent Affairs column.
ReaderWriterSlimResourceLock	This lock wraps the .NET Framework's ReaderWriterLockSlim class. Your application must be running on .NET Framework v3.5 or later to use this class.

The Wintellect.Threading.ResourceLocks.Diagnostics Namespace

This namespace defines several classes that modify or observe the behavior of the locks defined in the Wintellect.Threading.ResourceLocks namespace.

Type Name	Type Description
ResourceLockDelegator	This class is the base of all ResourceLock modifier classes and all ResourceLock observer classes.
ResourceLockModifier	This class is the base of all ResourceLock modifier classes.
ExclusiveOwnerResourceLockModifier	This class modifies an exclusive lock (non reader/writer lock) forcing the thread that obtained the lock to be the same thread that releases the lock.
RecursionResourceLockModifier	This class modifies any lock by adding recursion support to the lock. That is, a thread that gets a lock can get it multiple times.
ResourceLockObserver	This class is the base of all ResourceLock observer classes.
TimeoutNotifierResourceLockObserver	This class observes a lock and throws an exception if the lock has waited more than the specified amount of time.
StatisticsGatheringResourceLockObserver	This class observes a lock and keeps historical information about the lock's behavior and use as well as moment in time information (useful for debugging and logging).
ThreadSafeCheckerResourceLockObserver	This class observes a lock and ensures that that lock is behaving as expected. This is used to test a lock that you have implemented yourself.

The Wintellect.Threading Namespace

This namespace defines a number of utility classes that are directly related to threading.

Type Name	Type Description
ArbitraryWaitHandle	This class, derived from System.Threading.WaitHandle, can be used to turn an arbitrary IntPtr handle, or SafeHandle-derived object into a WaitHandle-derived object so allowing the object to be waited on. Good for processes, threads, or other Windows kernel objects not natively supported by the .NET Framework class library.
InterlockedEx	This static class defines a number of static methods that perform thread-safe atomic manipulation of values. Operations include: Math (Add, AddModulo, Decrement, DecrementIfGreaterThan, Increment, Max, Min), Bit (And, Or, Xor, BitTestAndCompliment, BitTestAndReset, BitTestAndSet), Masked Bit (MaskedAdd, MaskedAnd, MaskedExchange, MaskedOr, MaskedXor), Generic Morph (Morph), Convenience methods (IfThen), and support for types not natively supported by the .NET Framework including UInt32 and UInt64 (Exchange, CompareExchange).
Progress	A thread-safe progress management and reporting class.
SyncContextEventRaiser	This class raises an event by calling the callback method using the application model's desired thread as determined by the SynchronizationContext.
ThreadUtility	This static class defines a number of static methods that perform various thread operations. The NameFinalizerThreadForDebugging method sets the name of the CLR's Finalizer thread. The Spin method wastes CPU time for the number of milliseconds specified. The StallThread method temporarily stalls a thread and is typically used for testing. The IsSingleCpuMachine field returns true if machine has just 1 CPU; else false. Most of the remaining methods are just P/Invoke wrappers to Win32 methods: BeginBackgroundProcessingMode / EndBackgroundProcessingMode CancelSynchronousIo GetCurrentProcessorNumber GetCurrentWin32ThreadHandle GetCurrentWin32ThreadId GetWindowProcessId GetWindowThreadId OpenThread

	SwitchToThread

The Wintellect Namespace

This namespace defines a number of general-purpose classes that are not directly related to threading.

Type Name	Type Description
Disposer	Use this sealed class to wrap an IDisposable's Dispose method.
Exception<T>	A generic Exception class that makes it very easy to define new exception types.
OperationTimer	Use this class to time an algorithm's performance. It also shows how many Garbage Collections occurred due to objects being created in the algorithm.
SafePinnedObject	A class that pins an object to be passed to native code. If the object is unreferenced, this object ensures that the pinned object becomes unpinned. This class is used with the Wintellect.IO.DeviceIO class.
Singleton	A class that ensures that a singleton object is created only once even if multiple threads try to create it simultaneously.

The Wintellect.CommandArgumentParser Namespace

This namespace defines a number of classes to help with command-line argument parsing.

Type Name	Type Description
CmdArgAttribute	Use this custom attribute on a type's public field/property to associate it with a command-line argument.
CmdArgEnumValueDescriptionAttribute	Use this custom attribute to assign help text to a type's public field/property.
CmdArgParser	Use this class to display your application's Usage or to parse a command-line into a type's object setting the object's fields/properties.
ICmdArgs	Implement this interface to let your type have more control over the parsing of command-line arguments.

The Wintellect.IO Namespace

This namespace defines some classes that allow low-level device I/O control operations.

Type Name	Type Description
DeviceControlCode	Instance of this structure abstract a device I/O control code.
DeviceIO	Use this static class to open a device, send codes to the device, get device information, and set device information.
DeviceIOInfo	Create instance of this class to open a device, send control codes to the device, get device information, and set device information.

The Wintellect.Threading.LogicalProcessor Namespace

This namespace defines types for querying the computer's logical processor information.

Type Name	Type Description
LogicalProcessorInformation	This sealed class defines a static method that queries the CPU's logical processor information. It is basically a wrapper around the Win32 GetLogicalProcessorInformation method.

Change Log

October 23, 2010

- The Windows Phone version of the Power Threading library is introduced in this version of the library.
- The AsyncEnumerator class has two new properties: SuspendCallback and ResumeCallback. These two properties default to null but they can refer to a callback method (a delegate). Whenever the iterator is suspending itself due to a "yield return" statement, the SuspendCallback delegate will be invoked. This allows you to specify code that will execute whenever an iterator suspends itself. Similarly, the ResumeCallback delegate will be invoked just before an iterator resumes with the code following a "yield return" statement. For example, you could have the ResumeCallback method call Debugger.Break() to easily single-step back into an iterator in a debugger when an iterator resumes execution.
- Added the non-generic Wintellect.Threading.AsyncProgModel.ApmWrap class with its static SyncContextCallback method.
- Fixed a bug in the SyncGate and ReaderWriterGate classes that could cause a thread to block indefinitely.
- On the ResourceLock base class, the WaitToWrite & WaitToRead methods have been removed and replaced with a single Enter method that takes a Boolean indicating if you want write or read access. This change can simplify code and allows the choice to be data-derived instead of code driven. In addition, the Enter method returns void; it does NOT return an IDisposable (as the WaitToRead and WaitToWrite methods did) and therefore the Enter method cannot be used with C#'s using statement. The C# using statement releases a lock if an unexpected exception occurs and this allows other threads to access corrupt data causing your program to experience unpredictable behavior. In addition, the exception handling code that the using statement produces hurts the performance of the lock. Since the new Enter method cannot be used with the using statement, better programmer practices are encouraged as you must explicitly release the lock in your code making your application faster and less likely to continue running after data corruption. Furthermore, the DoneWriting & DoneReading methods have been removed and replaced with a single Leave method. The Leave method knows whether the lock was entered for write or read access and leaves the lock the correct way. This improvement reduces the potential for programmer error.
- Removed a bunch of classes under the Wintellect.Threading.ResourceLocks namespaces. The removed classes were not efficient and were best not used. In addition, I removed the deadlock detection code as it contained a bug that would occasionally cause it to report false positives. Removing these classes has the positive side-effect of improving the performance of the remaining ResourceLock-derived classes.

February 1, 2009

- Added the Wintellect.Threading.AsyncProgModel.EventApm class to the library

- Obsoleted the Execute method of the AsyncEnumerator and AsyncEnumerator<TResult> class as this encouraged poor programming practices and causes deadlocks in Windows Forms and WPF apps if not used correctly with AsyncEnumerator's SyncContext property. You can mimic Execute's behavior using code similar to:

  ```
  ae.EndExecute(ae.BeginExecute(…));
  ```

- Added greatly improved debugging support to the AsyncEnumerator.
  - You can pass a debugTag string to AsyncEnumerator's ctor to uniquely identify each AsyncEnumerator object.

  - You can now call `SetOperationTag` before calling an End/EndVoid method to associate a string debug tag with an operation. This is useful in debugging scenarios as it allows you to easily identify operations via a string name. Note that the `SetOperationTag` method is marked with a `[Conditional("AsyncEnumeratorDebug")]` attribute so that the compiler only emits calls to this method if the AsyncEnumeratorDebug symbol is defined at compile time. I recommend you define this symbol for debug builds and do not define it for release builds. This way, your release builds do not incur any performance hit associated with calling the `SetOperationTag` method.

  - When an End/EndVoid method is called, these methods now capture the calling thread's stack information effectively recording the location in your code of the operations you initiate.
    Since call stack capturing incurs a performance cost, the AsyncEnumerator will only capture stacks if AsyncEnumerator's static `EnableDebugSupport` method is called first. Typically you would invoke this method in a debug build of your application and not invoke it in a release build.
    Note that call stacks are not captured in the Silverlight or Compact Framework version of the library due to these platforms not offering this feature.
    Using the Visual Studio debugger, you can now look at each AsyncEnumerator object to determine which asynchronous operations it is still waiting to complete.

  - In addition to the above, when your iterator executes a "yield return" statement, the AsyncEnumerator object now records the computer's timestamp allowing you to know how long the AsyncEnumerator has been waiting for operations to complete.
    TIP: In the debugger, just hover over your AsyncEnumerator variable to see what it is waiting for and how long it has been waiting.

  - The AsyncEnumerator has a static `GetInProgressList()` method that returns a list of all the AsyncEnumerator objects that are currently processing an iterator. The list returned is sorted in last-yield-return time order with the AsyncEnumerator object that has been waiting the longest at element 0. You can examine each element's debugTag (passed to each AsyncEnumerator's constructor) and then examine each object's

operations that have not completed yet.
TIP: You may want to add AsyncEnumerator.GetInProgressList() to the debugger's watch window to easily see all the operations that your application is working on.

- To examine an AsyncEnumerator's variables, do the following: From a Visual Studio Immediate window, enter a line similar to the following to force a specific AsyncEnumerator object to "wake up" (replace 0 with whichever AsyncEnumerator object you want to debug) :

  AsyncEnumerator.GetInProgressList()[0].Cancel(null)

March 7, 2009

- Added the public static `FromAsyncResult` method. This method attempts to extract a reference to an AsyncEnumerator or AsyncEnumerator<TResult> object from an IAsyncResult object. For some applications, this can be convenient as it allows the application to not have to keep a reference to the AsyncEnumerator/ AsyncEnumerator<TResult> object itself as it can acquire a reference to it later.

May 26, 2009 (Version 4.6.0.0)

- Added a new CountdownTimer class to the Wintellect.Threading.AsyncProgModel namespace. This class offers a timer that adheres to the asynchronous programming model so that it can be used easily with the AsyncEnumerator class as well as in other scenarios.

- Added a new ApmWrap<T> struct to the Wintellect.Threading.AsyncProgModel. This light-weight struct has the ability to associate an arbitrary piece of data (of type T) with any IAsyncResult object. When the asynchronous operation completes, the associated piece of data can be retrieved to complete processing. This struct is typically used when you are implementing code that wraps an asynchronous operation and you wish to add some context or state of your own to complete the wrapping.

  Here is an example of how to use this struct:

```
public sealed class MyFileStream {
   private static ApmWrap<Int64> s_apmWrap = new ApmWrap<Int64>();
   private FileStream m_fs;   // Initialization not shown
   private Byte[] m_buffer = new Byte[1000];

   public IAsyncResult BeginXxx(AsyncCallback cb, Object state) {
      Int64 startTime = Environment.TickCount;
      return s_apmWrap.Return(startTime,
         m_fs.BeginRead(null, 0, m_buffer.Length,
            s_apmWrap.Callback(startTime, cb), state));
   }

   private Int32 EndXxx(IAsyncResult result) {
      Int64 startTime = s_apmWrap.Unwrap(ref result);
      Console.WriteLine("Operation took {0} ticks to complete.",
         Environment.TickCount - startTime);
```

```
            return m_fs.EndRead(result);
        }
    }
```

- Added the StreamExtensions class to the Wintellect.IO namespace. The methods perform an asynchronous copy of data from one stream to another. Currently, the BeginCopyStream/EndCopyStream methods are not C# 3.0 extension methods because adding this small feature would make my library depend on .NET 3.5 and I'm not ready to abandon support for .NET 2.0 yet.

- Fixed a small bug where an interator that did not execute any "yield return" statements returned a null IAsyncResult from BeginExecute.

- Improved the error handling when detecting that an asynchronous operation completed after an iterator has completed execution (and therefore, there is no way to clean up the completed operation).

    - In the new code, if an exception occurs in the iterator, then the AE will not attempt to discard any outstanding operations; and you will receive the exception that occurred in your iterator code. In this case, note that resources could be leaked until AppDomain/process shutdown if more operations complete in the future. So, an unhandled exception in an iterator should be considered fatal.

    - Only if no unhandled exceptions are thrown from inside the iterator will the AE discard any pending operations. If these operations cannot be cleaned up then an Exception<NoEndMethodCalled> will be thrown. Again, this should be considered fatal and you should fix your source code to ensure that you have specified a cleanup method for the operation. Consider using AsyncEnumerator's ThrowOnMissingDiscardGroup method to help you produce correct source code.

- Fixed a small bug where a NullReferenceException could get thrown from inside my AsyncResult object due to a race condition. This bug was never possible when using the callback method technique; it could only happen in scenarios when threads were waiting on the internal manual-reset event.

November 10, 2009

- Added two public delegate properties (SuspendCallback and ResumeCallback) to the AsyncEnumerator class. If SuspendCallback is not null, that it is invoked just after the iterator executes a "yield return" statement. If ResumeCallback is not null, it is invoked just before the iterator resumes from a "yield return" statement. These callbacks allow to have one place in your code for handling iterator suspends & resumes. You could have the ResumeCallback delegate refer to a method that calls Debugger.Break() to assist you with debugging, for example. Some customers capture the ExecutionContext in the SuspendCallback and restore it in the ResumeCallback.