# Async Interop with IAsyncResult

8 years ago (Jul 26, 2012) • [2 Comments](https://blog.stephencleary.com/2012/07/async-interop-with-iasyncresult.html" \l "comments)

Before async/await, the standard way to represent an asynchronous operation was to use the (poorly named) “asynchronous programming model” (APM). APM dates all the way back to .NET 1.0, and is baked into the language and runtime (asynchronous delegate invocation uses APM).

The Task-based Asynchronous Pattern (TAP) is superior than APM, but there is a lot of code out there that is either producing or consuming APM. The designers of TAP took this into consideration, and ensured that TAP plays well with APM. It’s possible to write TAP wrappers for APM methods, and APM wrappers for TAP methods, though the details can get just a touch hairy.

Most of what I’ll cover here is also covered in that excellent document, [The Task-based Asynchronous Pattern](https://www.microsoft.com/en-us/download/details.aspx?id=19957).

## TAP Wrappers for APM Implementations

By far, the easiest way to create a Task from a Begin/End method pair is to use one of the TaskFactory.FromAsync overloads, introduced in .NET 4.0:

public static Task<int> DivideAsync(this Calculator calc, int numerator, int denominator){

return Task.Factory.FromAsync(calc.BeginDivide, calc.EndDivide, numerator, denominator, null);}

It’s common to provide wrappers like this as extension methods. In fact, the Async CTP did exactly this for a lot of BCL classes.

If you wanted to make your own wrapper, it’s easy enough to do using TaskCompletionSource. Just call the Begin method, passing an AsyncCallback that completes the Task based on the results of the End method (canceling the Task if End throws OperationCanceledException).

public static Task<int> DivideAsync(this Calculator calc, int numerator, int denominator){

var tcs = new TaskCompletionSource<int>();

calc.BeginDivide(numerator, denominator, iar =>

{

try { tcs.TrySetResult(calc.EndDivide(iar)); }

catch(OperationCanceledException) { tcs.TrySetCanceled(); }

catch(Exception exc) { tcs.TrySetException(exc); }

}, null);

return tcs.Task;}

The only real design decision is what to do if Begin throws an exception. TaskFactory.FromAsync will complete the Task with a default result, and then (synchronously) rethrow the exception. The code above just allows the exception to propagate.

One drawback to TaskFactory.FromAsync is that it is somewhat limited in its number of parameters. If you need more, you can pass the IAsyncResult object instead of the Begin method. However, the implementation is less efficient: instead of just passing an AsyncCallback to Begin, they have to pass the AsyncWaitHandle to ThreadPool.RegisterWaitForSingleObject.

Another alternative is AsyncFactory.FromApm in [my AsyncEx library](https://nitoasyncex.codeplex.com/). FromApm supports many more parameters. However, if Begin throws, then FromApm will place the exception on the Task rather than throwing it directly (changing the synchronous exception to an asynchronous exception).

Actually, the entire AsyncEx library got its start with FromApm. TaskFactory.FromAsync did not have enough parameters to support a lot of the Azure SDK APM methods.

So, TAP wrappers for existing APM methods are pretty easy.

Update (2014-12-01): For more details, see Recipe 7.2 in my **[Concurrency Cookbook](https://stephencleary.com/book/)**.

## APM Wrappers for TAP Implementations

You’d think this would be easy, too, especially considering that Task implements IAsyncResult. Unfortunately, there are a couple of pitfalls.

Let’s start out with the Begin method implementation:

public IAsyncResult BeginDivide(int numerator, int denominator, AsyncCallback callback, object state){

var tcs = new TaskCompletionSource<int>(state);

var task = DivideAsync(numerator, denominator);

task.ContinueWith(t =>

{

// Copy the task result into the returned task.

if (t.IsFaulted)

tcs.TrySetException(t.Exception.InnerExceptions);

else if (t.IsCanceled)

tcs.TrySetCanceled();

else

tcs.TrySetResult(t.Result);

// Invoke the user callback if necessary.

if (callback != null)

callback(tcs.Task);

});

return tcs.Task;

}

The Task returned by Begin needs to be a different instance than the Task returned by the Async implementation because [IAsyncResult.AsyncState needs to return the state passed into the Begin method](https://blogs.msdn.com/b/junfeng/archive/2006/03/28/563627.aspx).

The example code above will always complete the Task and invoke the user callback from a thread pool thread. There are situations where it would be faster to do this synchronously (TaskContinuationOptions.ExecuteSynchronously), but [this can cause complex problems](https://social.msdn.microsoft.com/Forums/en-US/async/thread/9535a4a6-6218-45fe-aa45-79332b9e5b88). ExecuteSynchronously should ****not**** be used in APM wrappers for TAP methods!

Now, consider the End implementation:

public int EndDivide(IAsyncResult asyncResult){

try

{

return ((Task<int>)asyncResult).Result;

}

catch (AggregateException ex)

{

ExceptionDispatchInfo.Capture(ex.InnerException).Throw();

throw;

}}

The End method must (synchronously) block if the operation has not completed. If there was an exception, the Task wraps it into an AggregateException, so our End method will unwrap the exception and rethrow it (preserving the stack trace). The extra “throw;” line is never executed; it [just keeps the compiler happy](https://connect.microsoft.com/VisualStudio/feedback/details/689516/exceptiondispatchinfo-api-modifications).

Note that - as of now - the example code in the Task-based Asynchronous Pattern document does not unwrap Task exceptions, so any exceptions will be wrapped in an AggregateException, which is thrown from End.

The BCL does not provide generic Begin/End implementations, but my [AsyncEx](https://nitoasyncex.codeplex.com/) library does, as AsyncFactory.ToBegin and AsyncFactory.ToEnd:

public IAsyncResult BeginDivide(int numerator, int denominator, AsyncCallback callback, object state){

var task = DivideAsync(numerator, denominator);

return AsyncFactory<int>.ToBegin(task, callback, state);}

public int EndDivide(IAsyncResult asyncResult){

return AsyncFactory<int>.ToEnd(asyncResult);}

<https://blog.stephencleary.com/2012/07/async-interop-with-iasyncresult.html>