[Parallel Programming with .NET](http://blogs.msdn.com/b/pfxteam/)

**Async/Await FAQ**

From time to time, I receive questions from developers which highlight either a need for more information about the new “async” and “await” keywords in C# and Visual Basic. I’ve been cataloguing these questions, and I thought I’d take this opportunity to share my answers to them.

**Conceptual Overview**

**Where can I get a good overview of the async/await keywords?**

Generally, you can find lots of resources (links to articles, videos, blogs, etc.) on the Visual Studio Async page at <http://msdn.com/async>. To call out just a few specific resources, the October 2011 issue of MSDN Magazine included a trio of articles that provided a good introduction to the topic. If you read them all, I recommend you read them in the following order:

1. [Asynchronous Programming: Easier Asynchronous Programming with the New Visual Studio Async CTP](http://msdn.microsoft.com/en-us/magazine/hh456401.aspx)
2. [Asynchronous Programming: Pause and Play with Await](http://msdn.microsoft.com/en-us/magazine/hh456403.aspx)
3. [Asynchronous Programming: Understanding the Costs of Async and Await](http://msdn.microsoft.com/en-us/magazine/hh456402.aspx)

The .NET team blog also includes a good overview of asynchrony in .NET 4.5: [Async in 4.5: Worth the Await](http://blogs.msdn.com/b/dotnet/archive/2012/04/03/async-in-4-5-worth-the-await.aspx).

**Why do I need the compiler to help me with asynchronous programming?**

Anders Hejlsberg’s [Future directions for C# and Visual Basic](http://channel9.msdn.com/events/BUILD/BUILD2011/TOOL-816T) talk at //BUILD/ provides a great tour through why a compiler is really beneficial here. In short, the compiler takes on the responsibility of doing the kind of complicated transformations you’d otherwise be doing by hand as you manually invert your control flow using callbacks and continuation-passing style. You get to write your code using the language’s control flow constructs, just as you would if you were writing synchronous code, and the compiler under the covers applies the transformations necessary to use callbacks in order to avoid blocking threads.

**To achieve the benefits of asynchrony, can’t I just wrap my synchronous methods in calls to Task.Run?**

It depends on your goals for why you want to invoke the methods asynchronously. If your goal is simply to offload the work you’re doing to another thread, so as to, for example, maintain the responsiveness of your UI thread, then sure. If your goal is to help with scalability, then no, just wrapping a synchronous call in a Task.Run won’t help. For more information, see [Should I expose asynchronous wrappers for synchronous methods?](http://blogs.msdn.com/b/pfxteam/archive/2012/03/24/10287244.aspx) And if from your UI thread you want to offload work to a worker thread, and you use Task.Run to do so, you often typically want to do some work back on the UI thread once that background work is done, and these language features make that kind of coordination easy and seamless.

**The “Async” Keyword**

**What does the “async” keyword do when applied to a method?**

When you mark a method with the “async” keyword, you’re really telling the compiler two things:

1. You’re telling the compiler that you want to be able to use the “await” keyword inside the method (you can use the await keyword if and only if the method or lambda it’s in is marked as async). In doing so, you’re telling the compiler to compile the method using a state machine, such that the method will be able to suspend and then resume asynchronously at await points.
2. You’re telling the compiler to “lift” the result of the method or any exceptions that may occur into the return type. For a method that returns Task or Task<TResult>, this means that any returned value or exception that goes unhandled within the method is stored into the result task. For a method that returns void, this means that any exceptions are propagated to the caller’s context via whatever “SynchronizationContext” was current at the time of the method’s initial invocation.

**Does using the “async” keyword on a method force all invocations of that method to be asynchronous?**

No. When you invoke a method marked as “async”, it begins running synchronously on the curren thread. So, if you have a synchronous method that returns void and all you do to change it is mark it as “async”, invocations of that method will still run synchronously. This is true regardless of whether you leave the return type as “void” or change it to “Task”. Similarly, if you have a synchronous method that returns some TResult, and all you do is mark it as “async” and change the return type to be “Task<TResult>”, invocations of that method will still run synchronously.

Marking a method as “async” does not affect whether the method runs to completion synchronously or asynchronously. Rather, it enables the method to be split into multiple pieces, some of which may run asynchronously, such that the method may complete asynchronously. The boundaries of these pieces can occur only where you explicitly code one using the “await” keyword, so if “await” isn’t used at all in a method’s code, there will only be one piece, and since that piece will start running synchronously, it (and the whole method with it) will complete synchronously.

**Does the “async” keyword cause the invocation of a method to queue to the ThreadPool? To create a new thread? To launch a rocket ship to Mars?**

No. No. And no. See the previous questions. The “async” keyword indicates to the compiler that “await” may be used inside of the method, such that the method may suspend at an await point and have its execution resumed asynchronously when the awaited instance completes. This is why the compiler issues a warning if there are no “awaits” inside of a method marked as “async”.

**Can I mark any method as “async”?**

No. Only methods that return void, Task, or Task<TResult> can be marked as async. Further, not all such methods can be marked as “async”. For example, you can’t use “async”:

* On your application’s entry point method, e.g. Main. When you await an instance that’s not yet completed, execution returns to the caller of the method. In the case of Main, this would return out of Main, effectively ending the program.
* On a method attributed with:
  + [MethodImpl(MethodImplOptions.Synchronized)]. For a discussion of why this is disallowed, see [What’s New for Parallelism in .NET 4.5 Beta](http://blogs.msdn.com/b/pfxteam/archive/2012/02/29/10274035.aspx); attributing a method as Synchronized is akin to wrapping the entire body of the method with lock/SyncLock.
  + [SecurityCritical] and [SecuritySafeCritical]. When you compile an async method, the implementation / body of the method actually ends up in a compiler-generated MoveNext method, but the attributes for it remain on the signature you defined. That means that attributes like [SecuritySafeCritical] (which is meant to have a direct impact on what you’re able to do in the body of the method) would not work correctly, and thus they’re prohibited, at least for now.
* On a method with ref or out parameters.  The caller would expect those values to be set when the synchronous invocation of the method completes, but the implementation might not set them until its asynchronous completion much later.
* On a lambda used as an expression tree.  Async lambda expressions cannot be converted to expression trees.

**Are there any conventions I should use when writing methods marked as “async”?**

Yes. The Task-based Asynchronous Pattern (TAP) is entirely focused on how asynchronous methods that return Task or Task<TResult> should be exposed from libraries. This includes, but is not limited to, methods implemented using the “async” and “await” keywords. For an in-depth tour through the TAP, see the [Task-based Asynchronous Pattern](http://www.microsoft.com/download/en/details.aspx?id=19957) document.

**Do I need to “Start” Tasks created by methods marked as “async”?**

No.  Tasks returned from TAP methods are “hot”, meaning the tasks represent operations that are already in-progress.  Not only do you not need to call “.Start()” on such tasks, but doing so will fail if you try.  For more details, see [FAQ on Task.Start](http://blogs.msdn.com/b/pfxteam/archive/2012/01/14/10256832.aspx).

**Do I need to “Dispose” Tasks created by methods marked as “async”?**

No. In general, you don’t need to Dispose of any tasks.  See [Do I need to dispose of Tasks?](http://blogs.msdn.com/b/pfxteam/archive/2012/03/25/10287435.aspx).

**How does “async” relate to the current SynchronizationContext?**

For methods marked as “async” that return Task or Task<TResult>, there is no method-level interaction with the SynchronizationContext. However, for methods marked as “async” that return void, there is a potential interaction.

When an “async void” method is invoked, the prolog for the method’s invocation (as handled by the AsyncVoidMethodBuilder that is created by the compiler to represent the method’s lifetime) will capture the current SynchronizationContext (“capture” here means it accesses it and stores it). If there is a non-null SynchronizationContext, two things will be affected:

* The beginning of the method’s invocation will result in a call to the captured context’s OperationStarted method, and the completion of the method’s execution (whether synchronous or asynchronous) will result in a call to that captured context’s OperationCompleted method. This gives the context a chance to reference count outstanding asynchronous operations; if instead the method had returned a Task or Task<TResult>, the caller could have done the same tracking via that returned task.
* If the method completes due to an unhandled exception, the throwing of that exception will be Post’d to the captured SynchronizationContext. This gives the context a chance to deal with the failure. This is in contrast to a Task or Task<TResult>-returning async method, where the exception can be marshaled to the caller through the returned task.

If there isn’t a SynchronizationContext when the “async void” method is called, no context is captured, and then as there are no OperationStarted / OperationCompleted methods to call, none are invoked. In such a case, if an exception goes unhandled, the exception is propagated on the ThreadPool, which with default behavior will cause the process to be terminated.

**The “Await” Keyword**

**What does the “await” keyword do?**

The “await” keyword tells the compiler to insert a possible suspension/resumption point into a method marked as “async”.

Logically this means that when you write “await someObject;” the compiler will generate code that checks whether the operation represented by someObject has already completed. If it has, execution continues synchronously over the await point. If it hasn’t, the generated code will hook up a continuation delegate to the awaited object such that when the represented operation completes, that continuation delegate will be invoked. This continuation delegate will re-enter the method, picking up at this await location where the previous invocation left off. At this point, regardless of whether the awaited object had already completed by the time it was awaited, any result from the object will be extracted, or if the operation failed, any exception that occurred will be propagated.

In code, this means that when you write:

await someObject;

the compiler translates that into something like the following (this code is an approximation of what the compiler actually generates):

private class FooAsyncStateMachine : IAsyncStateMachine   
{   
    // Member fields for preserving “locals” and other necessary state   
    int $state;   
    TaskAwaiter $awaiter;   
    …   
    public void MoveNext()   
    {   
        // Jump table to get back to the right statement upon resumption   
        switch (this.$state)   
        {   
            …   
            case 2: goto Label2;   
            …   
        }   
        …   
        // Expansion of “await someObject;”   
        this.$awaiter = someObject.GetAwaiter();   
        if (!this.$awaiter.IsCompleted)   
        {   
            this.$state = 2;   
            this.$awaiter.OnCompleted(MoveNext);   
            return;   
            Label2:   
        }   
        this.$awaiter.GetResult();   
        …   
    }   
}

**What are awaitables? What are awaiters?**

While Task and Task<TResult> are two types very commonly awaited, they’re not the only ones that may be awaited.

An “awaitable” is any type that exposes a GetAwaiter method which returns a valid “awaiter”. This GetAwaiter method may be an instance method (as it is in the case of Task and Task<TResult>), or it may be an extension method.

An “awaiter” is any type returned from an awaitable’s GetAwaiter method and that conforms to a particular pattern. The awaiter must implement the System.Runtime.CompilerServices.INotifyCompletion interface, and optionally may implement the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface. In addition to providing an implementation of the OnCompleted method that comes from INotifyCompletion (and optionally the UnsafeOnCompleted method that comes from ICriticalNotifyCompletion), an awaiter must also provide an IsCompleted Boolean property, as well as a parameterless GetResult method. GetResult returns void if the awaitable represents a void-returning operation, or it returns a TResult if the awaitable represents a TResult-returning operation.

Any type that follows the awaitable pattern may be awaited. For a discussion of several approaches to implementing custom awaitables, see [await anything;](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx). You can also implement awaitables customized for very specific situations: for some examples, see [Advanced APM Consumption in Async Methods](http://blogs.msdn.com/b/pfxteam/archive/2012/01/23/10259822.aspx) and [Awaiting Socket Operations](http://blogs.msdn.com/b/pfxteam/archive/2011/12/15/10248293.aspx).

**Where can’t I use “await”?**

You can’t use await:

* Inside of a method or lambda not marked with “async”. The “async” keyword is what tells the compiler that it’s ok to use “await” inside of a method. (For a discussion of why the “async” keyword is required, see [Asynchrony in C# 5 Part Six: Whither async?](http://blogs.msdn.com/b/ericlippert/archive/2010/11/11/whither-async.aspx).)
* Inside of a property getter or setter. Properties are meant to return to the caller quickly, and thus are not expected to need asynchrony, which is geared for potentially long-running operations. If you must use asynchrony in your properties, you can do so by implementing an async method which is then used from your property.
* Inside of a lock/SyncLock block. For a discussion of why this is disallowed, as well as for a look at SemaphoreSlim.WaitAsync (which can be used with await), see [What’s New for Parallelism in .NET 4.5 Beta](http://blogs.msdn.com/b/pfxteam/archive/2012/02/29/10274035.aspx). You can also read about building a variety of custom async synchronization primitives at:
  + [Building Async Coordination Primitives, Part 1: AsyncManualResetEvent](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266920.aspx)
  + [Building Async Coordination Primitives, Part 2: AsyncAutoResetEvent](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266923.aspx)
  + [Building Async Coordination Primitives, Part 3: AsyncCountdownEvent](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266930.aspx)
  + [Building Async Coordination Primitives, Part 4: AsyncBarrier](http://blogs.msdn.com/b/pfxteam/archive/2012/02/11/10266932.aspx)
  + [Building Async Coordination Primitives, Part 5: AsyncSemaphore](http://blogs.msdn.com/b/pfxteam/archive/2012/02/12/10266983.aspx)
  + [Building Async Coordination Primitives, Part 6: AsyncLock](http://blogs.msdn.com/b/pfxteam/archive/2012/02/12/10266988.aspx)
  + [Building Async Coordination Primitives, Part 7: AsyncReaderWriterLock](http://blogs.msdn.com/b/pfxteam/archive/2012/02/12/10267069.aspx)
* Inside of an unsafe region. Note that you can use the “unsafe” keyword inside of a method marked as “async”, you just can’t use “await” inside of an unsafe region. This restriction has to do with how the compilers preserve state across an “await” point, and difficulties inherent to preserving unmanaged pointers.
* Inside of a catch or finally block. You can use “await” inside of a try block, regardless of whether it has associated catch or finally blocks, but you can’t use it inside of the catch or finally blocks. Doing so would disrupt the semantics of CLR exception handling.
* With most aspects of LINQ query comprehension syntax. “await” may only be used in a query expression within the first collection expression of the initial “from” clause or within the collection expression of a “join” clause.

**Is “await task;” the same thing as “task.Wait()”?**

No.

“task.Wait()” is a synchronous, potentially blocking call: it will not return to the caller of Wait() until the task has entered a final state, meaning that it’s completed in the RanToCompletion, Faulted, or Canceled state. In contrast, “await task;” tells the compiler to insert a potential suspension/resumption point into a method marked as “async”, such that if the task has not yet completed when it’s awaited, the async method should return to its caller, and its execution should resume when and only when the awaited task completes. Using “task.Wait()” when “await task;” would have been more appropriate can lead to unresponsive applications and deadlocks; see [Await, and UI, and deadlocks! Oh my!](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115163.aspx).

There are some other potential pitfalls to be aware of when using “async” and “await”. For some examples, see:

* [Potential pitfalls to avoid when passing around async lambdas](http://blogs.msdn.com/b/pfxteam/archive/2012/02/08/10265476.aspx)
* [Keeping Async Methods Alive](http://blogs.msdn.com/b/pfxteam/archive/2011/10/02/10219048.aspx)
* [Don’t Forget To Complete Your Tasks](http://blogs.msdn.com/b/pfxteam/archive/2011/10/02/10218999.aspx)
* [Are deadlocks still possible with await?](http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293249.aspx)

**Is there a functional difference between “task.Result” and “task.GetAwaiter().GetResult()”?**

Yes, but only if the task completes non-successfully.  If the task ends in the RanToCompletion state, these are completely equivalent statements.  If, however, the task ends in the Faulted or Canceled state, the former will propagate the one or more exceptions wrapped in AggregateException, while the latter will propagate the exception directly (and if there are more than one in the task, it’ll just propagate one of them).  For background on why this difference exists, see [Task Exception Handling in .NET 4.5](http://blogs.msdn.com/b/pfxteam/archive/2011/09/28/10217876.aspx).

**How does “await” relate to the current SynchronizationContext?**

This is entirely up to the type being awaited. For a given await, the compiler generates code that ends up calling an awaiter’s OnCompleted method, passing in the continuation delegate to be executed. The compiler-generated code knows nothing about SynchronizationContext, and simply relies on the awaited object’s OnCompleted method to invoke the provided callback when the awaited operation completes. It’s the OnCompleted method, then, that’s responsible for making sure that the delegate is invoked in the “right place,” where “right place” is left entirely up to the awaiter.

The default behavior for awaiting a task (as implemented by the TaskAwaiter and TaskAwaiter<TResult> types returned from Task’s and Task<TResult>’s GetAwaiter methods, respectively) is to capture the current SynchronizationContext before suspending, and then when the awaited task completes, if there had been a current SynchronizationContext that got captured, to Post the invocation of the continuation delegate back to that SynchronizationContext. So, for example, if you use “await task;” on the UI thread of your application, OnCompleted when invoked will see a non-null current SynchronizationContext, and when the task completes, it’ll use that UI’s SynchronizationContext to marshal the invocation of the continuation delegate back to the UI thread.

If there isn’t a current SynchronizationContext when you await a Task, then the system will check to see if there’s a current TaskScheduler, and if there is, the continuation will be scheduled to that when the task completes.

If there isn’t such a context or scheduler to force the continuation back to, or if you do “await task.ConfigureAwait(false)” instead of just “await task;”, then the continuation won’t be forced back to the original context and will be allowed to run wherever the system deems appropriate. This typically means either running the continuation synchronously wherever the awaited task completes or running the continuation on the ThreadPool.

**Can I use “await” in console apps?**

Sure. You can’t use “await” inside of your Main method, however, as entry points can’t be marked as async. Instead, you can use “await” in other methods in your console app, and then if you call those methods from Main, you can synchronously wait (rather than asynchronously wait) for them to complete, e.g.

public static void Main()   
{   
    FooAsync().Wait();   
}

private static async Task FooAsync()   
{   
    await Task.Delay(1000);   
    Console.WriteLine(“Done with first delay”);   
    await Task.Delay(1000);   
}

You could also use a custom SynchronizationContext or TaskScheduler to achieve similar capabilities. For more information, see:

* [Await, SynchronizationContext, and Console Apps: Part 1](http://blogs.msdn.com/b/pfxteam/archive/2012/01/20/10259049.aspx)
* [Await, SynchronizationContext, and Console Apps: Part 2](http://blogs.msdn.com/b/pfxteam/archive/2012/01/21/10259307.aspx)
* [Await, SynchronizationContext, and Console Apps: Part 3](http://blogs.msdn.com/b/pfxteam/archive/2012/02/02/10263555.aspx)

**Can I use “await” with other asynchronous patterns, like the Asynchronous Programming Model (APM) pattern and the Event-based Async Pattern (EAP)?**

Sure. You can either implement a custom awaitable for your asynchronous operation, or you can convert the existing asynchronous operation to something that’s already awaitable, like Task or Task<TResult>. Here are some examples:

* [Tasks and the APM Pattern](http://blogs.msdn.com/b/pfxteam/archive/2009/06/09/9716439.aspx)
* [Tasks and the Event-based Asynchronous Pattern](http://blogs.msdn.com/b/pfxteam/archive/2009/06/19/9791857.aspx)
* [Advanced APM Consumption in Async Methods](http://blogs.msdn.com/b/pfxteam/archive/2012/01/23/10259822.aspx)
* [Implementing a SynchronizationContext.SendAsync method](http://blogs.msdn.com/b/pfxteam/archive/2012/01/20/10259082.aspx)
* [Awaiting Socket Operations](http://blogs.msdn.com/b/pfxteam/archive/2011/12/15/10248293.aspx)
* [await anything;](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx)
* [The Nature of TaskCompletionSource<TResult>](http://blogs.msdn.com/b/pfxteam/archive/2009/06/02/9685804.aspx)

**Does the code generated by async/await result in efficient asynchronous execution?**

For the most part, yes, as a lot of work has been done to optimize the code generated by the compiler and the .NET Framework methods on which the generated code relies. For more information, including on best practices for minimizing the overhead of using tasks and async/await, see:

* [TPL Performance Improvements in .NET 4.5](http://blogs.msdn.com/b/pfxteam/archive/2011/11/10/10235962.aspx)
* ["The Zen of Async" at the MVP Summit 2012](http://blogs.msdn.com/b/pfxteam/archive/2012/03/03/10277034.aspx)
* [Asynchronous Programming: Understanding the Costs of Async and Await](http://msdn.microsoft.com/en-us/magazine/hh456402.aspx)
* [19 Comments](http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/async-await-faq.aspx#comments)



|  |  |
| --- | --- |
|  |  |

[Parallel Extensions](http://blogs.msdn.com/b/pfxteam/archive/tags/Parallel+Extensions/), [Async](http://blogs.msdn.com/b/pfxteam/archive/tags/Async/), [.NET 4.5](http://blogs.msdn.com/b/pfxteam/archive/tags/-NET+4-5/)

Easier Asynchronous Programming with the New Visual Studio Async CTP

[Eric Lippert](http://msdn.microsoft.com/en-us/magazine/ee532098.aspx?sdmr=EricLippert&sdmi=authors)

 Imagine what the world would be like if people worked the same way as computer programs:

1. void ServeBreakfast(Customer diner)
2. {
3. var order = ObtainOrder(diner);
4. var ingredients = ObtainIngredients(order);
5. var recipe = ObtainRecipe(order);
6. var meal = recipe.Prepare(ingredients);
7. diner.Give(meal);
8. }

Each subroutine can, of course, be broken down further; preparing the meal might involve heating pans, cooking omelets and toasting bread. Were humans to perform these sorts of tasks like typical computer programs, we’d carefully write down everything as sequences of hierarchical tasks in a checklist and obsessively ensure that each job was complete before embarking on the next.

A subroutine-based approach seems reasonable—you can’t cook the eggs before you get the order—but in fact it both wastes time and makes the application appear unresponsive. It wastes time because you want the bread to be toasting while the eggs are frying, not after the eggs are done and getting cold. It appears unresponsive because if another customer arrives while the current order is still cooking, you want to be taking his order, not making him wait at the door until the current customer has been served her breakfast. A server slavishly following a checklist does not have any ability to respond in a timely manner to unexpected events.

**Solution One: Hire More Staff by Making More Threads**

Making someone’s breakfast is a whimsical example, but the reality, of course, is anything but. Every time you transfer control into a long-running subroutine on the UI thread, the UI becomes completely unresponsive until the subroutine finishes. How could it be otherwise? Applications respond to UI events by running code on the UI thread, and that thread is obsessively busy doing something else. Only when every job on its list is done will it get around to dealing with the queued-up commands of the frustrated  user. The usual solution to this problem is to use concurrencyto do two or more things “at the same time.” (If the two threads are on two independent processors, they might truly be running at the same time. In a world with more threads than processors to dedicate to them, the OS will simulate at-the-same-time concurrency by periodically scheduling a time slice for each thread to control a processor.)

One concurrent solution might be to create a thread pool and assign each new client a specific thread to handle its requests. In our analogy, you could hire a group of servers. When a new diner comes in, an idle server is assigned to the new diner. Each server then independently does the work of taking the order, finding the ingredients, cooking the food and serving it.

The difficulty with this approach is that UI events typically arrive on the same thread and expect to be fully serviced on that thread. Most UI components create requests on the UI thread, and expect to be communicated with only on that thread. Dedicating a new thread to each UI-related task is unlikely to work well.

To address this problem you could have a single foreground thread listening to UI events that does nothing but “take orders” and farm them out to one or more background worker threads. In this analogy, there’s only one server who interacts with the customers, and a kitchen full of cooks who actually do the requested work. The UI thread and the worker threads are then responsible for coordinating their communications. The cooks never talk directly to the diners, but somehow the food gets served anyway.

This certainly solves the “responding to UI events in a timely manner” problem, but it doesn’t resolve the lack of efficiency; the code running on the worker thread is still waiting synchronously for the eggs to cook fully before the bread goes in the toaster. That problem could be solved in turn by adding even more concurrency: You could have two cooks per order, one for the eggs and one for the toast. But that might get pretty expensive. Just how many cooks are you going to need, and what happens when they have to coordinate their work?

Concurrency of this sort introduces many well-known difficulties. First, threads are notoriously heavyweight; a thread by default consumes a million bytes of virtual memory for its stack and many other system resources. Second, UI objects are often “affinitized” to the UI thread and can’t be called from worker threads; the worker thread and the UI thread must come to some complex arrangement whereby the UI thread can send necessary information from the UI elements over to the worker, and the worker can send updates back to the UI thread, rather than to the UI elements directly. Such arrangements are difficult to code and prone to race conditions, deadlocks and other threading problems. Third, many of the pleasant fictions that we all rely upon in the single-threaded world—such as reads and writes of memory happening in a predictable and consistent sequence—are no longer reliable. This leads to the worst kinds of difficult-to-reproduce bugs.

It just seems wrong to have to use the big hammer of thread-based concurrency to build simple programs that remain responsive and run efficiently. Somehow real people manage to solve complex problems while remaining responsive to events. In the real world you don’t have to allocate one waiter per table or two cooks per order to serve dozens of customer requests that are all pending at the same time. Solving the problem with threading makes for too many cooks. There’s got to be a better solution that doesn’t involve so much concurrency.

**Solution Two: Develop Attention Deficit Disorder with DoEvents**

A common non-concurrent “solution” to the problem of UI unresponsiveness during long-running operations is to liberally sprinkle the magic words Application.DoEvents around a program until the problem goes away. Though this is certainly a pragmatic solution, it’s not a very well-engineered one:

1. void ServeBreakfast(Customer diner)
2. {
3. var order = ObtainOrder(diner);
4. Application.DoEvents();
5. var ingredients = ObtainIngredients(order);
6. Application.DoEvents();
7. var recipe = ObtainRecipe(order);
8. Application.DoEvents();
9. var meal = recipe.Prepare(ingredients);
10. Application.DoEvents();
11. diner.Give(meal);
12. }

Basically, using DoEvents means “see if anything interesting happened while I was busy doing that last thing. If something happened that I need to respond to, remember what I was doing just now, deal with the new situation, and then come back to where I left off.” It makes your program behave like it has attention deficit disorder: anything new that comes along gets attention right away. That sounds like a plausible solution to improve responsiveness—and sometimes even works—but there are a number of problems with this approach.

First, DoEvents works best when the delay is caused by a loop that has to execute many times, but each individual loop execution is short. By checking for pending events every few times through the loop, you can maintain responsiveness even if the whole loop takes a long time to run. However, that pattern is usually not the cause of a responsiveness problem. More often the problem is caused by one inherently long-running operation taking a lot of time, such as attempting to synchronously access a file over a high-latency network. Perhaps in our example the long-running task is in preparing the meal, and there’s no place to put the DoEvents that helps. Or perhaps there is a place where DoEvents would help, but it’s in a method you don’t have the source code for.

Second, calling DoEvents causes the program to attempt to fully service all the more recent events before finishing work associated with earlier events. Imagine if no one could get his meal until after every customer who came in got his meal! If more and more customers keep arriving, the first customer might never get his meal, resulting in starvation. In fact, it could happen that no customers get their meals. The completion of work associated with earlier events can be pushed off arbitrarily far into the future as servicing newer events continues to interrupt the work being done for earlier events.

Third, DoEvents poses the very real danger of unexpected reentrancy. That is, while serving one customer you check to see if there have been any recent interesting UI events and accidentally start serving the same diner again, even though he’s already being served. Most developers don’t design their code to detect this kind of reentrancy; it’s possible to end up in some very strange program states indeed when an algorithm never intended to be recursive ends up calling itself unexpectedly via DoEvents.

In short, DoEvents should be used only to fix a responsiveness problem in the most trivial cases; it’s not a good solution for managing UI responsiveness in complex programs.

**Solution Three: Turn Your Checklist Inside-out with Callbacks**

The non-concurrent nature of the DoEvents technique is attractive, but clearly not quite the right solution for a complex program. A better idea is to break down the items on the checklist into a series of short tasks, each of which can be completed rapidly enough that the application can appear to be responsive to events.

That idea is nothing new; dividing a complex problem into small parts is why we have subroutines in the first place. The interesting twist is that instead of rigidly running down a checklist to determine what has already been done and what needs to be done next, and only returning control to the caller when everything is completed, each new task is given the list of work that must come after it. The work that must come after a particular task is finished is called the “continuation” of the task.

When a task has finished, it can look at the continuation and finish it off right there. Or it might schedule the continuation to run later. If the continuation requires information computed by the previous task, the previous task can pass that information along as an argument to the call that invokes the continuation.

With this approach, the total body of work is essentially broken up into little pieces that can each be executed rapidly. The system appears responsive because pending events can be detected and handled between the executions of any two of the small pieces of work. But because any activities associated with those new events can also be broken down into small parts and queued up to execute later, we don’t have the “starvation” problem whereby new tasks prevent old tasks from completing. New long-running tasks are not dealt with immediately, but they are queued up for eventual processing.

The idea is great, but it’s not at all clear how to implement such a solution. The essential difficulty is determining how to tell each small unit of work what its continuation is; that is, what work needs to come next.

In traditional asynchronous code, this is typically done by registering a “callback” function. Let’s suppose we have an asynchronous version of “Prepare” that takes a callback function that says what to do next—namely, serve the meal:

1. void ServeBreakfast(Diner diner)
2. {
3. var order = ObtainOrder(diner);
4. var ingredients = ObtainIngredients(order);
5. var recipe = ObtainRecipe(order);
6. recipe.PrepareAsync(ingredients, meal =>
7. {
8. diner.Give(meal);
9. });
10. }

Now ServeBreakfast returns immediately after PrepareAsync returns; whatever code called ServeBreakfast is then free to service other events that occur. PrepareAsync does no “real” work itself; rather, it quickly does whatever is necessary to ensure that the meal will be prepared in the future. Moreover, PrepareAsync also ensures that the callback method will be invoked with the prepared meal as its argument at some time after the meal preparation task is completed. Thus, the diner will eventually be served, though she might have to wait briefly if there’s an event that requires attention between the end of the preparation and the serving of the meal.

Note that none of this necessarily involves a second thread. Perhaps PrepareAsync causes the meal preparation work to be done on a separate thread, or perhaps it causes a series of short tasks associated with meal preparation to be queued up on the UI thread to be executed later. It really doesn’t matter; all we know is that PrepareAsync somehow guarantees two things: that the meal will be prepared in a manner that doesn’t block the UI thread with a high-latency operation, and that the callback will somehow be invoked after the work of preparing the requested meal is done.

But suppose any of the methods for obtaining the order, obtaining the ingredients, obtaining the recipe or preparing the meal might be the one that’s slowing down the UI. We could solve this larger problem if we had an asynchronous version of each of these methods. What would the resulting program look like? Remember, each method must be given a callback that tells it what to do when the unit of work is completed:

1. void ServeBreakfast(Diner diner)
2. {
3. ObtainOrderAsync(diner, order =>
4. {
5. ObtainIngredientsAsync(order, ingredients =>
6. {
7. ObtainRecipeAsync(order, recipe =>
8. {
9. recipe.PrepareAsync(ingredients, meal =>
10. {
11. diner.Give(meal);
12. })})})});
13. }

This might seem like an awful mess, but it’s nothing compared to how bad real programs get when they’re rewritten using callback-based asynchrony. Think about how you might deal with making a loop asynchronous, or how you’d deal with exceptions, try-finally blocks or other non-trivial forms of control flow. You end up essentially turning your program inside-out; the code now emphasizes how all the callbacks are wired together, and not what the logical workflow of the program should be.

**Solution Four: Make the Compiler Solve the Problem with Task-Based Asynchrony**

Callback-based asynchrony does keep the UI thread responsive and minimize time wasted by synchronously waiting for long-running work to complete. But the cure seems worse than the disease. The price you pay for responsiveness and performance is that you have to write code that emphasizes how the mechanisms of the asynchrony work while obscuring the meaning and purpose of the code.

The upcoming versions of C# and Visual Basic instead allow you to write code that emphasizes its meaning and purpose, while giving enough hints to the compilers to build the necessary mechanisms for you behind the scenes. The solution has two parts: one in the type system, and one in the language.

The CLR 4 release defined the type Task<T>—the workhorse type of the Task Parallel Library (TPL)—to represent the concept of “some work that’s going to produce a result of type T in the future.” The concept of “work that will complete in the future but returns no result” is represented by the non-generic Task type.

Precisely how the result of type T is going to be produced in the future is an implementation detail of a particular task; the work might be farmed out to another machine entirely, to another process on this machine, to another thread, or perhaps the work is simply to read a previously cached result that can be accessed cheaply from the current thread. TPL tasks are typically farmed out to worker threads from a thread pool in the current process, but that implementation detail is not fundamental to the Task<T> type; rather, a Task<T> can represent any high-latency operation that produces a T.

The language half of the solution is the new await keyword. A regular method call means “remember what you’re doing, run this method until it’s completely finished, and then pick up where you left off, now knowing the result of the method.” An await expression, in contrast, means “evaluate this expression to obtain an object representing work that will in the future produce a result. Sign up the remainder of the current method as the callback associated with the continuation of that task. Once the task is produced and the callback is signed up, immediately return control to my caller.”

Our little example rewritten in the new style reads much more nicely:

1. async void ServeBreakfast(Diner diner)
2. {
3. var order = await ObtainOrderAsync(diner);
4. var ingredients = await ObtainIngredientsAsync(order);
5. var recipe = await ObtainRecipeAsync(order);
6. var meal = await recipe.PrepareAsync(ingredients);
7. diner.Give(meal);
8. }

In this sketch, each asynchronous version returns a Task<Order>, Task<List<Ingredient>> and so on. Every time an await is encountered, the currently executing method signs up the rest of the method as the thing to do when the current task is complete, and then immediately returns. Somehow each task will complete itself—either by being scheduled to run as an event on the current thread, or because it used an I/O completion thread or worker thread—and will then cause its continuation to “pick up where it left off” in executing the rest of the method.

Note that the method is now marked with the new async keyword; this is simply an indicator to the compiler that lets it know that in the context of this method, the keyword await is to be treated as a point where the workflow returns control to its caller and picks up again when the associated task is finished. Note also that the examples I’ve shown in this article use C# code; Visual Basic will have a similar feature with similar syntax. The design of these features in C# and Visual Basic was heavily influenced by F# asynchronous workflows, a feature that F# has had for some time.

**Where to Learn More**

This brief introduction merely motivates and then scratches the surface of the new asynchrony feature in C# and Visual Basic. For a more detailed explanation of how it works behind the scenes, and how to reason about the performance characteristics of asynchronous code, see the companion articles in this issue by my colleagues Mads Torgersen and Stephen Toub.

To get your hands on a preview release of this feature, along with samples, white papers and a community forum for questions, discussions and constructive feedback, please go to [msdn.com/async](http://msdn.com/async). These language features and the libraries that support them are still in development; the design team would love to have as much of your feedback as possible.

Pause and Play with Await

[Mads Torgersen](http://msdn.microsoft.com/en-us/magazine/ee532098.aspx?sdmr=MadsTorgersen&sdmi=authors)

[**Download the Code Sample**](http://code.msdn.microsoft.com/mag201110Await)

Asynchronous methods in the upcoming versions of Visual Basic and C# are a great way to get the callbacks out of your asynchronous programming. In this article, I’ll take a closer look at what the new await keyword actually does, starting at the conceptual level and working my way down to the iron.

**Sequential Composition**

Visual Basic and C# are imperative programming languages—and proud of it! This means they excel in letting you express your programming logic as a sequence of discrete steps, to be undertaken one after the other. Most statement-level language constructs are control structures that give you a variety of ways to specify the order in which the discrete steps of a given body of code are to be executed:

* Conditional statements such as if and switch  let you choose different subsequent actions based on the current state of the world.
* Loop statements such as for, foreach and while let you repeat the execution of a certain set of steps multiple times.
* Statements such as continue, throw and goto let you transfer control non-locally to other parts of the program.

Building up your logic using control structures results in sequential composition, and this is the lifeblood of imperative programming. It is indeed why there are so many control structures to choose from: You want sequential composition to be really convenient and well-structured.

**Continuous Execution**

In most imperative languages, including current versions of Visual Basic and C#, the execution of methods (or functions, or procedures or whatever we choose to call them) is continuous. What I mean by that is that once a thread of control has begun executing a given method, it will be continuously occupied doing so until the method execution ends. Yes, sometimes the thread will be executing statements in methods called by your body of code, but that’s just part of executing the method. The thread will never switch to do anything your method didn’t ask it to.

This continuity is sometimes problematic. Occasionally there’s nothing a method can do to make progress—all it can do is wait for something to happen: a download, a file access, a computation happening on a different thread, a certain point in time to arrive. In such situations the thread is fully occupied doing nothing. The common term for that is that the thread is blocked; the method causing it to do so is said to be blocking.

Here’s an example of a method that is seriously blocking:

1. static byte[] TryFetch(string url)
2. {
3. var client = new WebClient();
4. try
5. {
6. return client.DownloadData(url);
7. }
8. catch (WebException) { }
9. return null;
10. }

A thread executing this method will stand still during most of the call to client.DownloadData, doing no actual work but just waiting.

This is bad when threads are precious—and they often are. On a typical middle tier, servicing each request in turn requires talking to a back end or other service. If each request is handled by its own thread and those threads are mostly blocked waiting for intermediate results, the sheer number of threads on the middle tier can easily become a performance bottleneck.

Probably the most precious kind of thread is a UI thread: there’s only one of them. Practically all UI frameworks are single-threaded, and they require everything UI-related—events, updates, the user’s UI manipulation logic—to happen on the same dedicated thread. If one of these activities (for example, an event handler choosing to download from a URL) starts to wait, the whole UI is unable to make progress because its thread is so busy doing absolutely nothing.

What we need is a way for multiple sequential activities to be able to share threads. To do that, they need to sometimes “take a break”—that is, leave holes in their execution where others can get something done on the same thread. In other words, they sometimes need to be discontinuous. It’s particularly convenient if those sequential activities take that break while they’re doing nothing anyway. To the rescue: asynchronous programming!

**Asynchronous Programming**

Today, because methods are always continuous, you have to split discontinuous activities (such as the before and after of a download) into multiple methods. To poke a hole in the middle of a method’s execution, you have to tear it apart into its continuous bits. APIs can help by offering asynchronous (non-blocking) versions of long-running methods that initiate the operation (start the download, for example), store a passed-in callback for execution upon completion and then immediately return to the caller. But in order for the caller to provide the callback, the “after” activities need to be factored out into a separate method.

Here’s how this works for the preceding TryFetch method:

1. static void TryFetchAsync(string url, Action<byte[], Exception> callback)
2. {
3. var client = new WebClient();
4. client.DownloadDataCompleted += (\_, args) =>
5. {
6. if (args.Error == null) callback(args.Result, null);
7. else if (args.Error is WebException) callback(null, null);
8. else callback(null, args.Error);
9. };
10. client.DownloadDataAsync(new Uri(url));
11. }

Here you see a couple of different ways of passing callbacks: The DownloadDataAsync method expects an event handler to have been signed up to the DownloadDataCompleted event, so that’s how you pass the “after” part of the method. TryFetchAsync itself also needs to deal with its callers’ callbacks. Instead of setting up that whole event business yourself, you use the simpler approach of just taking a callback as a parameter. It’s a good thing we can use a lambda expression for the event handler so it can just capture and use the “callback” parameter directly; if you tried to use a named method, you’d have to think of some way to get the callback delegate to the event handler. Just pause for a second and think how you’d write this code without lambdas.

But the main thing to notice here is how much the control flow changed. Instead of using the language’s control structures to express the flow, you emulate them:

* The return statement is emulated by calling the callback.
* Implicit propagation of exceptions is emulated by calling the callback.
* Exception handling is emulated with a type check.

Of course, this is a very simple example. As the desired control structure gets more complex, emulating it gets even more so.

To summarize, we gained discontinuity, and thereby the ability of the executing thread to do something else while “waiting” for the download. But we lost the ease of using control structures to express the flow. We gave up our heritage as a structured imperative language.

**Asynchronous Methods**

When you look at the problem this way, it becomes clear how asynchronous methods in the next versions of Visual Basic and C# help: They let you express discontinuous sequential code.

 Let’s look at the asynchronous version of TryFetch with this new syntax:

1. static async Task<byte[]> TryFetchAsync(string url)
2. {
3. var client = new WebClient();
4. try
5. {
6. return await client.DownloadDataTaskAsync(url);
7. }
8. catch (WebException) { }
9. return null;
10. }

Asynchronous methods let you take the break inline, in the middle of your code: Not only can you use your favorite control structures to express sequential composition, you can also poke holes in the execution with await expressions—holes where the executing thread is free to do other things.

A good way to think about this is to imagine that asynchronous methods have “pause” and “play” buttons. When the executing thread reaches an await expression, it hits the “pause” button and the method execution is suspended. When the task being awaited completes, it hits the “play” button, and the method execution is resumed.

**Compiler Rewriting**

When something complex looks simple, it usually means there’s something interesting going on under the hood, and that’s certainly the case with asynchronous methods. The simplicity gives you a nice abstraction that makes it so much easier to both write and read asynchronous code. Understanding what’s happening underneath is not a requirement. But if you do understand, it will surely help you become a better asynchronous programmer, and be able to more fully utilize the feature. And, if you’re reading this, chances are good you’re also just plain curious. So let’s dive in: What do async methods—and the await expressions in them—actually do?

When the Visual Basic or C# compiler gets hold of an asynchronous method, it mangles it quite a bit during compilation: the discontinuity of the method is not directly supported by the underlying runtime and must be emulated by the compiler. So instead of you having to pull the method apart into bits, the compiler does it for you. However, it does this quite differently than you’d probably do it manually.

The compiler turns your asynchronous method into a statemachine. The state machine keeps track of where you are in the execution and what your local state is. It can either be running or suspended. When it’s running, it may reach an await, which hits the “pause” button and suspends execution. When it’s suspended, something may hit the “play” button to get it back and running.

The await expression is responsible for setting things up so that the “play” button gets pushed when the awaited task completes. Before we get into that, however, let’s look at the state machine itself, and what those pause and play buttons really are.

**Task Builders**

Asynchronous methods produce Tasks. More specifically, an asynchronous method returns an instance of one of the types Task or Task<T> from System.Threading.Tasks, and that instance is automatically generated. It doesn’t have to be (and can’t be) supplied by the user code. (This is a small lie: Asynchronous methods can return void, but we’ll ignore that for the time being.)

From the compiler’s point of view, producing Tasks is the easy part. It relies on a framework-supplied notion of a Task builder, found in System.Runtime.CompilerServices (because it’s not normally meant for direct human consumption). For instance, there’s a type like this:

1. public class AsyncTaskMethodBuilder<TResult>
2. {
3. public Task<TResult> Task { get; }
4. public void SetResult(TResult result);
5. public void SetException(Exception exception);
6. }

The builder lets the compiler obtain a Task, and then lets it complete the Task with a result or an Exception. **Figure 1** is a sketch of what this machinery looks like for TryFetchAsync.

**Figure 1 Building a Task**

1. static Task<byte[]> TryFetchAsync(string url)
2. {
3. var \_\_builder = new AsyncTaskMethodBuilder<byte[]>();
4. ...
5. Action \_\_moveNext = delegate
6. {
7. try
8. {
9. ...
10. return;
11. ...
12. \_\_builder.SetResult(…);
13. ...
14. }
15. catch (Exception exception)
16. {
17. \_\_builder.SetException(exception);
18. }
19. };
20. \_\_moveNext();
21. return \_\_builder.Task;
22. }

Watch carefully:

* First a builder is created.
* Then a \_\_moveNext delegate is created. This delegate is the “play” button. We call it the resumption delegate, and it contains:
  + The original code from your async method (though we have elided it so far).
  + Return statements, which represent pushing the “pause” button.
  + Calls that complete the builder with a successful result, which correspond to the return statements of the original code.
  + A wrapping try/catch that completes the builder with any escaped exceptions.
* Now the “play” button is pushed; the resumption delegate is called. It runs until the “pause” button is hit.
* The Task is returned to the caller.

Task builders are special helper types meant only for compiler consumption. However, their behavior isn’t much different from what happens when you use the TaskCompletionSource types of the Task Parallel Library (TPL) directly.

So far I’ve created a Task to return and a “play” button—the resumption delegate—for someone to call when it’s time to resume execution. I still need to see how execution is resumed and how the await expression sets up for something to do this. Before I put it all together, though, let’s take a look at how tasks are consumed.

**Awaitables and Awaiters**

As you’ve seen, Tasks can be awaited. However, Visual Basic and C# are perfectly happy to await other things as well, as long as they’re awaitable; that is, as long as they have a certain shape that the await expression can be compiled against. In order to be awaitable, something has to have a GetAwaiter method, which in turn returns an awaiter. As an example, Task<TResult> has a GetAwaiter method that returns this type:

1. public struct TaskAwaiter<TResult>
2. {
3. public bool IsCompleted { get; }
4. public void OnCompleted(Action continuation);
5. public TResult GetResult();
6. }

The members on the awaiter let the compiler check if the awaitable is already complete, sign up a callback to it if it isn’t yet, and obtain the result (or Exception) when it is.

We can now start to see what an await should do to pause and resume around the awaitable. For instance, the await inside our TryFetchAsync example would turn into something like this:

1. \_\_awaiter1 = client.DownloadDataTaskAsync(url).GetAwaiter();
2. if (!\_\_awaiter1.IsCompleted) {
3. ... // Prepare for resumption at Resume1
4. \_\_awaiter1.OnCompleted(\_\_moveNext);
5. return; // Hit the "pause" button
6. }
7. Resume1:
8. ... \_\_awaiter1.GetResult()) ...

Again, watch what happens:

* An awaiter is obtained for the task returned from DownloadDataTaskAsync.
* If the awaiter is not complete, the “play” button—the resumption delegate—is passed to the awaiter as a callback.
* When the awaiter resumes execution (at Resume1) the result is obtained and used in the code that follows it.

Clearly the common case is that the awaitable is a Task or Task<T>. Indeed, those types—which are already present in the Microsoft .NET Framework 4—have been keenly optimized for this role. However, there are good reasons for allowing other awaitable types as well:

* Bridging to other technologies: F#, for instance, has a type Async<T> that roughly corresponds to Func<Task<T>>. Being able to await Async<T> directly from Visual Basic and C# helps bridge between asynchronous code written in the two languages. F# is similarly exposing bridging functionality to go the other way—consuming Tasks directly in asynchronous F# code.
* Implementing special semantics: The TPL itself is adding a few simple examples of this. The static Task.Yield utility method, for instance, returns an awaitable that will claim (via IsCompleted) to not be complete, but will immediately schedule the callback passed to its OnCompleted method, as if it had in fact completed. This lets you force scheduling and bypass the compiler’s optimization of skipping it if the result is already available. This can be used to poke holes in “live” code, and improve responsiveness of code that isn’t sitting idle. Tasks themselves can’t represent things that are complete but claim not to be, so a special awaitable type is used for that.

Before I take a further look at the awaitable implementation of Task, let’s finish looking at the compiler’s rewriting of the asynchronous method, and flesh out the bookkeeping that tracks the state of the method’s execution.

**The State Machine**

In order to stitch it all together, I need to build up a state machine around the production and consumption of the Tasks. Essentially, all the user logic from the original method is put into the resumption delegate, but the declarations of locals are lifted out so they can survive multiple invocations. Furthermore, a state variable is introduced to track how far things have gotten, and the user logic in the resumption delegate is wrapped in a big switch that looks at the state and jumps to a corresponding label. So whenever resumption is called, it will jump right back to where it left off the last time. **Figure 2** puts the whole thing together.

**Figure 2 Creating a State Machine**

1. static Task<byte[]> TryFetchAsync(string url)
2. {
3. var \_\_builder = new AsyncTaskMethodBuilder<byte[]>();
4. int \_\_state = 0;
5. Action \_\_moveNext = null;
6. TaskAwaiter<byte[]> \_\_awaiter1;
8. WebClient client = null;
10. \_\_moveNext = delegate
11. {
12. try
13. {
14. if (\_\_state == 1) goto Resume1;
15. client = new WebClient();
16. try
17. {
18. \_\_awaiter1 = client.DownloadDataTaskAsync(url).GetAwaiter();
19. if (!\_\_awaiter1.IsCompleted) {
20. \_\_state = 1;
21. \_\_awaiter1.OnCompleted(\_\_moveNext);
22. return;
23. }
24. Resume1:
25. \_\_builder.SetResult(\_\_awaiter1.GetResult());
26. }
27. catch (WebException) { }
28. \_\_builder.SetResult(null);
29. }
30. catch (Exception exception)
31. {
32. \_\_builder.SetException(exception);
33. }
34. };
36. \_\_moveNext();
37. return \_\_builder.Task;
38. }

Quite the mouthful! I’m sure you’re asking yourself why this code is so much more verbose than the manually “asynchronized” version shown earlier. There are a couple of good reasons, including efficiency (fewer allocations in the general case) and generality (it applies to user-defined awaitables, not just Tasks). However, the main reason is this: You don’t have to pull the user logic apart after all; you just augment it with some jumps and returns and such.

While the example is too simple to really justify it, rewriting a method’s logic into a semantically equivalent set of discrete methods for each of its continuous bits of logic between the awaits is very tricky business. The more control structures the awaits are nested in, the worse it gets. When not just loops with continue and break statements but try-finally blocks and even goto statements surround the awaits, it’s exceedingly difficult, if indeed possible, to produce a rewrite with high fidelity.

Instead of attempting that, it seems a neat trick is to just overlay the user’s original code with another layer of control structure, airlifting you in (with conditional jumps) and out (with returns) as the situation requires. Play and pause. At Microsoft, we’ve been systematically testing the equivalence of asynchronous methods to their synchronous counterparts, and we’ve confirmed that this is a very robust approach. There’s no better way to preserve synchronous semantics into the asynchronous realm than by retaining the code that describes those semantics in the first place.

**The Fine Print**

The description I’ve provided is slightly idealized—there are a few more tricks to the rewrite, as you may have suspected. Here are a few of the other gotchas the compiler has to deal with:

**Goto Statements** The rewrite in **Figure 2** doesn’t actually compile, because goto statements (in C# at least) can’t jump to labels buried in nested structures. That’s no problem in itself, as the compiler generates to intermediate language (IL), not source code, and isn’t bothered by nesting. But even IL doesn’t allow jumping into the middle of a try block, as is done in my example. Instead, what really happens is that you jump to the beginning of a try block, enter it normally and then switch and jump again.

**Finally Blocks** When returning out of the resumption delegate because of an await, you don’t want the finally bodies to be executed yet. They should be saved for when the original return statements from the user code are executed. You control that by generating a Boolean flag signaling whether the finally bodies should be executed, and augmenting them to check it.

**Evaluation Order** An await expression is not necessarily the first argument to a method or operator; it can occur in the middle. To preserve the order of evaluation, all the preceding arguments must be evaluated before the await, and the act of storing them and retrieving them again after the await is surprisingly involved.

On top of all this, there are a few limitations you can’t get around. For instance, awaits aren’t allowed inside of a catch or finally block, because we don’t know of a good way to reestablish the right exception context after the await.

**The Task Awaiter**

The awaiter used by the compiler-generated code to implement the await expression has considerable freedom as to how it schedules the resumption delegate—that is, the rest of the asynchronous method. However, the scenario would have to be really advanced before you’d need to implement your own awaiter. Tasks themselves have quite a lot of flexibility in how they schedule because they respect a notion of scheduling context that itself is pluggable.

The scheduling context is one of those notions that would probably look a little nicer if we had designed for it from the start. As it is, it’s an amalgam of a few existing concepts that we’ve decided not to mess up further by trying to introduce a unifying concept on top. Let’s look at the idea at the conceptual level, and then I’ll dive into the realization.

The philosophy underpinning the scheduling of asynchronous callbacks for awaited tasks is that you want to continue executing “where you were before,” for some value of “where.” It’s this “where” that I call the scheduling context. Scheduling context is a thread-affine concept; every thread has (at most) one. When you’re running on a thread, you can ask for the scheduling context it’s running in, and when you have a scheduling context, you can schedule things to run in it.

So this is what an asynchronous method should do when it awaits a task:

* On suspension: Ask the thread it’s running on for its scheduling context.
* On resumption: Schedule the resumption delegate back on that scheduling context.

Why is this important? Consider the UI thread. It has its own scheduling context, which schedules new work by sending it through the message queue back on the UI thread. This means that if you’re running on the UI thread and await a task, when the result of the task is ready, the rest of the asynchronous method will run back on the UI thread. Thus, all the things you can do only on the UI thread (manipulating the UI) you can still do after the await; you won’t experience a weird “thread hop” in the middle of your code.

Other scheduling contexts are multithreaded; specifically, the standard thread pool is represented by a single scheduling context. When new work is scheduled to it, it may go on any of the pool’s threads. Thus, an asynchronous method that starts out running on the thread pool will continue to do so, though it may “hop around” among different specific threads.

In practice, there’s no single concept corresponding to the scheduling context. Roughly speaking, a thread’s SynchronizationContext acts as its scheduling context. So if a thread has one of those (an existing concept that can be user-implemented), it will be used. If it doesn’t, then the thread’s TaskScheduler (a similar concept introduced by the TPL) is used. If it doesn’t have one of those either, the default TaskScheduler is used; that one schedules resumptions to the standard thread pool.

Of course, all this scheduling business has a performance cost. Usually, in user scenarios, it’s negligible and well worth it: Having your UI code chopped up into manageable bits of actual live work and pumped in through the message pump as waited-for results become available is normally just what the doctor ordered.

Sometimes, though—especially in library code—things can get too fine-grained. Consider:

1. async Task<int> GetAreaAsync()
2. {
3. return await GetXAsync() \* await GetYAsync();
4. }

This schedules back to the scheduling context twice—after each await—just to perform a multiplication on the “right” thread. But who cares what thread you’re multiplying on? That’s probably wasteful (if used often), and there are tricks to avoid it: You can essentially wrap the awaited Task in a non-Task awaitable that knows how to turn off the schedule-back behavior and just run the resumption on whichever thread completes the task, avoiding the context switch and the scheduling delay:

1. async Task<int> GetAreaAsync()
2. {
3. return await GetXAsync().ConfigureAwait(continueOnCapturedContext: false)
4. \* await GetYAsync().ConfigureAwait(continueOnCapturedContext: false);
5. }

Less pretty, to be sure, but a neat trick to use in library code that ends up being a bottleneck for scheduling.

**Go Forth and Async’ify**

Now you should have a working understanding of the underpinnings of asynchronous methods. Probably the most useful points to take away are:

* The compiler preserves the meaning of your control structures by actually preserving your control structures.
* Asynchronous methods don’t schedule new threads—they let you multiplex on existing ones.
* When tasks get awaited, they put you back “where you were” for a reasonable definition of what that means.

If you’re like me, you’ve already been alternating between reading this article and typing in some code. You’ve multiplexed multiple flows of control—reading and coding—on the same thread: you. That’s just what asynchronous methods let you do.

Async Performance: Understanding the Costs of Async and Await

[Stephen Toub](http://msdn.microsoft.com/en-us/magazine/ee532098.aspx?sdmr=StephenToub&sdmi=authors)

 Asynchronous programming has long been the realm of only the most skilled and masochistic of developers—those with the time, inclination and mental capacity to reason about callback after callback of non-linear control flow. With the Microsoft .NET Framework 4.5, C# and Visual Basic deliver asynchronicity for the rest of us, such that mere mortals can write asynchronous methods almost as easily as writing synchronous methods. No more callbacks. No more explicit marshaling of code from one synchronization context to another. No more worrying about the flowing of results or exceptions. No more tricks that contort existing language features to ease async development. In short, no more hassle.

Of course, while it’s now easy to get started writing asynchronous methods (see the articles by [Eric Lippert](http://msdn.microsoft.com/en-us/magazine/hh456401.aspx) and [Mads Torgersen](http://msdn.microsoft.com/en-us/magazine/hh456403.aspx) in this issue of MSDN Magazine), doing it really well still requires an understanding of what’s happening under the covers. Any time a language or framework raises the level of abstraction at which a developer can program, it invariably also encapsulates hidden performance costs. In many cases, such costs are negligible and can and should be ignored by the vast number of developers implementing the vast number of scenarios. However, it still behooves more advanced developers to really understand what costs exist so they can take any necessary steps to avoid those costs if they do eventually become visible. Such is the case with the asynchronous methods feature in C# and Visual Basic.

In this article, I’ll explore the ins and outs of asynchronous methods, providing you with a solid understanding of how asynchronous methods are implemented under the covers and discussing some of the more nuanced costs involved. Note that this information isn’t meant to encourage you to contort readable code into something that can’t be maintained, all in the name of micro-optimization and performance. It’s simply to give you information that may help you diagnose any problems you may run across, as well as supply a set of tools to help you overcome such potential issues. Note also that this article is based on a preview release of the .NET Framework 4.5, and it’s likely that specific implementation details will change prior to the final release.

**Getting the Right Mental Model**

For decades, developers have used high-level languages like C#, Visual Basic, F# and C++ to develop efficient applications. This experience has informed those developers about the relevant costs of various operations, and that knowledge has informed best development practices. For example, for most use cases, calling a synchronous method is relatively cheap, even more so when the compiler is able to inline the callee into the call site. Thus, developers learn to refactor code into small, maintainable methods, in general without needing to think about any negative ramifications from the increased method invocation count. These developers have a mental model for what it means to call a method.

With the introduction of asynchronous methods, a new mental model is needed. While the C# and Visual Basic languages and compilers are able to provide the illusion of an asynchronous method being just like its synchronous counterpart, under the covers it’s no such thing. The compiler ends up generating a lot of code on behalf of the developer, code akin to the quantities of boilerplate code that developers implementing asynchronicity in days of yore would’ve had to have written and maintained by hand. Further still, the compiler-generated code calls into library code in the .NET Framework, again increasing the work done on behalf of the developer. To get the right mental model, and then to use that mental model to make appropriate development decisions, it’s important to understand what the compiler is generating on your behalf.

**Think Chunky, Not Chatty**

When working with synchronous code, methods with empty bodies are practically free. This is not the case for asynchronous methods. Consider the following asynchronous method, which has a single statement in its body (and which due to lack of awaits will end up running synchronously):

1. public static async Task SimpleBodyAsync() {
2. Console.WriteLine("Hello, Async World!");
3. }

An intermediate language (IL) decompiler will reveal the true nature of this function once compiled, with output similar to what’s shown in **Figure 1**. What was a simple one-liner has been expanded into two methods, one of which exists on a helper state machine class. First, there’s a stub method that has the same basic signature as that written by the developer (the method is named the same, it has the same visibility, it accepts the same parameters and it retains its return type), but that stub doesn’t contain any of the code written by the developer. Rather, it contains setup boilerplate. The setup code initializes the state machine used to represent the asynchronous method and then kicks it off using a call to the secondary MoveNext method on the state machine. This state machine type holds state for the asynchronous method, allowing that state to be persisted across asynchronous await points, if necessary. It also contains the body of the method as written by the user, but contorted in a way that allows for results and exceptions to be lifted into the returned Task; for the current position in the method to be maintained so that execution may resume at that location after an await; and so on.

**Figure 1 Asynchronous Method Boilerplate**

1. [DebuggerStepThrough]
2. public static Task SimpleBodyAsync() {
3. <SimpleBodyAsync>d\_\_0 d\_\_ = new <SimpleBodyAsync>d\_\_0();
4. d\_\_.<>t\_\_builder = AsyncTaskMethodBuilder.Create();
5. d\_\_.MoveNext();
6. return d\_\_.<>t\_\_builder.Task;
7. }
9. [CompilerGenerated]
10. [StructLayout(LayoutKind.Sequential)]
11. private struct <SimpleBodyAsync>d\_\_0 : <>t\_\_IStateMachine {
12. private int <>1\_\_state;
13. public AsyncTaskMethodBuilder <>t\_\_builder;
14. public Action <>t\_\_MoveNextDelegate;
16. public void MoveNext() {
17. try {
18. if (this.<>1\_\_state == -1) return;
19. Console.WriteLine("Hello, Async World!");
20. }
21. catch (Exception e) {
22. this.<>1\_\_state = -1;
23. this.<>t\_\_builder.SetException(e);
24. return;
25. }
27. this.<>1\_\_state = -1;
28. this.<>t\_\_builder.SetResult();
29. }
31. ...
32. }

When thinking through what asynchronous methods cost to invoke, keep this boilerplate in mind. The try/catch block in the MoveNext method will likely prevent it from getting inlined by the just-in-time (JIT) compiler, so at the very least we’ll now have the cost of a method invocation where in the synchronous case we likely would not (with such a small method body). We have multiple calls into Framework routines (like SetResult). And we have multiple writes to fields on the state machine type. Of course, we need to weigh all of this against the cost of the Console.WriteLine, which will likely dominate all of the other costs involved (it takes locks, it does I/O and so forth). Further, notice that there are optimizations the infrastructure does for you. For example, the state machine type is a struct. That struct will only be boxed to the heap if this method ever needs to suspend its execution because it’s awaiting an instance that’s not yet completed, and in this simple method, it never will complete. As such, the boilerplate of this asynchronous method won’t incur any allocations. The compiler and runtime work hard together to minimize the number of allocations involved in the infrastructure.

**Know When Not to Use Async**

The .NET Framework attempts to generate efficient asynchronous implementations for asynchronous methods, applying multiple optimizations. However, developers often have domain knowledge than can yield optimizations that would be risky and unwise for the compiler and runtime to apply automatically, given the generality they target. With this in mind, it can actually benefit a developer to avoid using async methods in a certain, small set of use cases, particularly for library methods that will be accessed in a more fine-grained manner. Typically, this is the case when it’s known that the method may actually be able to complete synchronously because the data it’s relying on is already available.

When designing asynchronous methods, the Framework developers spent a lot of time optimizing away object allocations. This is because allocations represent one of the largest performance costs possible in the asynchronous method infrastructure. The act of allocating an object is typically quite cheap. Allocating objects is akin to filling your shopping cart with merchandise, in that it doesn’t cost you much effort to put items into your cart; it’s when you actually check out that you need to pull out your wallet and invest significant resources. While allocations are usually cheap, the resulting garbage collection can be a showstopper when it comes to the application’s performance. The act of garbage collection involves scanning through some portion of objects currently allocated and finding those that are no longer referenced. The more objects allocated, the longer it takes to perform this marking. Further, the larger the allocated objects and the more of them that are allocated, the more frequently garbage collection needs to occur. In this manner, then, allocations have a global effect on the system: the more garbage generated by asynchronous methods, the slower the overall program will run, even if micro benchmarks of the asynchronous methods themselves don’t reveal significant costs.

For asynchronous methods that actually yield execution (due to awaiting an object that’s not yet completed), the asynchronous method infrastructure needs to allocate a Task object to return from the method, as that Task serves as a unique reference for this particular invocation. However, many asynchronous method invocations can complete without ever yielding. In such cases, the asynchronous method infrastructure may return a cached, already completed Task, one that it can use over and over to avoid allocating unnecessary Tasks. It’s only able to do this in limited circumstances, however, such as when the asynchronous method is a non-generic Task, a Task<Boolean>, or when it’s a Task<TResult> where TResult is a reference type and the result of the asynchronous method is null. While this set may expand in the future, you can often do better if you have domain knowledge of the operation being implemented.

Consider implementing a type like MemoryStream. MemoryStream derives from Stream, and thus can override Stream’s new .NET 4.5 ReadAsync, WriteAsync and FlushAsync methods to provide optimized implementations for the nature of MemoryStream. Because the operation of reading is simply going against an in-memory buffer and is therefore just a memory copy, better performance results if ReadAsync runs synchronously. Implementing this with an asynchronous method would look something like the following:

1. public override async Task<int> ReadAsync(
2. byte [] buffer, int offset, int count,
3. CancellationToken cancellationToken)
4. {
5. cancellationToken.ThrowIfCancellationRequested();
6. return this.Read(buffer, offset, count);
7. }

Easy enough. And because Read is a synchronous call, and because there are no awaits in this method that will yield control, all invocations of ReadAsync will actually complete synchronously. Now, let’s consider a standard usage pattern of streams, such as a copy operation:

1. byte [] buffer = new byte[0x1000];
2. int numRead;
3. while((numRead = await source.ReadAsync(buffer, 0, buffer.Length)) > 0) {
4. await source.WriteAsync(buffer, 0, numRead);
5. }

Notice here that ReadAsync on the source stream for this particular series of calls is always invoked with the same count parameter (the buffer’s length), and thus it’s very likely that the return value (the number of bytes read) will also be repeating. Except in some rare circumstances, it’s very unlikely that the asynchronous method implementation of ReadAsync will be able to use a cached Task for its return value, but you can.

Consider rewriting this method as shown in **Figure 2**. By taking advantage of the specific aspects of this method and its common usage scenarios, we’ve now been able to optimize allocations away on the common path in a way we couldn’t expect the underlying infrastructure to do. With this, every time a call to ReadAsync retrieves the same number of bytes as the previous call to ReadAsync, we’re able to completely avoid any allocation overhead from the ReadAsync method by returning the same Task we returned on the previous invocation. And for a low-level operation like this that we expect to be very fast and to be invoked repeatedly, such an optimization can make a noticeable difference, especially in the number of garbage collections that occur.

**Figure 2 Optimizing Task Allocations**

1. private Task<int> m\_lastTask;
3. public override Task<int> ReadAsync(
4. byte [] buffer, int offset, int count,
5. CancellationToken cancellationToken)
6. {
7. if (cancellationToken.IsCancellationRequested) {
8. var tcs = new TaskCompletionSource<int>();
9. tcs.SetCanceled();
10. return tcs.Task;
11. }
13. try {
14. int numRead = this.Read(buffer, offset, count);
15. return m\_lastTask != null && numRead == m\_lastTask.Result ?
16. m\_lastTask : (m\_lastTask = Task.FromResult(numRead));
17. }
18. catch(Exception e) {
19. var tcs = new TaskCompletionSource<int>();
20. tcs.SetException(e);
21. return tcs.Task;
22. }
23. }

A related optimization to avoid the task allocation may be done when the scenario dictates caching. Consider a method whose purpose it is to download the contents of a particular Web page and then cache its successfully downloaded contents for future accesses. Such functionality might be written using an asynchronous method as follows (using the new System.Net.Http.dll library in .NET 4.5):

1. private static ConcurrentDictionary<string,string> s\_urlToContents;
3. public static async Task<string> GetContentsAsync(string url)
4. {
5. string contents;
6. if (!s\_urlToContents.TryGetValue(url, out contents))
7. {
8. var response = await new HttpClient().GetAsync(url);
9. contents = response.EnsureSuccessStatusCode().Content.ReadAsString();
10. s\_urlToContents.TryAdd(url, contents);
11. }
12. return contents;
13. }

This is a straightforward implementation. And for calls to GetContentsAsync that can’t be satisfied from the cache, the overhead of constructing a new Task<string> to represent this download will be negligible when compared to the network-related costs. However, for cases where the contents may be satisfied from the cache, it could represent a non-negligible cost, an object allocation simply to wrap and hand back already available data.

To avoid that cost (if doing so is required to meet your performance goals), you could rewrite this method as shown in **Figure 3**. We now have two methods: a synchronous public method, and an asynchronous private method to which the public method delegates. The dictionary is now caching the generated tasks rather than their contents, so future attempts to download a page that’s already been successfully downloaded can be satisfied with a simple dictionary access to return an already existing task. Internally, we also take advantage of the ContinueWith methods on Task that allow us to store the task into the dictionary once the Task has completed—but only if the download succeeded. Of course, this code is more complicated and requires more thought to write and maintain, so as with any performance optimizations, avoid spending time making them until performance testing proves that the complications make an impactful and necessary difference. Whether such optimizations make a difference really depends on usage scenarios. You’ll want to come up with a suite of tests that represent common usage patterns, and use analysis of those tests to determine whether these complications improve your code’s performance in a meaningful way.

**Figure 3 Manually Caching Tasks**

1. private static ConcurrentDictionary<string,Task<string>> s\_urlToContents;
3. public static Task<string> GetContentsAsync(string url) {
4. Task<string> contents;
5. if (!s\_urlToContents.TryGetValue(url, out contents)) {
6. contents = GetContentsInternalAsync(url);
7. contents.ContinueWith(delegate {
8. s\_urlToContents.TryAdd(url, contents);
9. }, CancellationToken.None,
10. TaskContinuationOptions.OnlyOnRanToCompletion |
11. TaskContinuatOptions.ExecuteSynchronously,
12. TaskScheduler.Default);
13. }
14. return contents;
15. }
17. private static async Task<string> GetContentsInternalAsync(string url) {
18. var response = await new HttpClient().GetAsync(url);
19. return response.EnsureSuccessStatusCode().Content.ReadAsString();
20. }

Another task-related optimization to consider is whether you even need the returned Task from an asynchronous method. C# and Visual Basic both support the creation of asynchronous methods that return void, in which case no Task is allocated for the method, ever. Asynchronous methods exposed publicly from libraries should always be written to return a Task or Task<TResult>, because you as a library developer don’t know whether the consumer desires to wait on the completion of that method. However, for certain internal usage scenarios, void-returning asynchronous methods can have their place. The primary reason void-returning asynchronous methods exist is to support existing event-driven environments, like ASP.NET and Windows Presentation Foundation (WPF). They make it easy to implement button handlers, page-load events and the like through the use of async and await. If you do consider using an async void method, be very careful around exception handling: exceptions that escape an async void method bubble out into whatever SynchronizationContext was current at the time the async void method was invoked.

**Care About Context**

There are many kinds of “context” in the .NET Framework: LogicalCallContext, SynchronizationContext, HostExecutionContext, SecurityContext, ExecutionContext and more (from the sheer number you might expect that the developers of the Framework are monetarily incentivized to introduce new contexts, but I assure you we’re not). Some of these contexts are very relevant to asynchronous methods, not only in functionality, but also in their impact on asynchronous method performance.

**SynchronizationContext** SynchronizationContext plays a big role in asynchronous methods. A “synchronization context” is simply an abstraction over the ability to marshal delegate invocation in a manner specific to a given library or framework. For example, WPF provides a DispatcherSynchronizationContext to represent the UI thread for a Dispatcher: posting a delegate to this synchronization context causes that delegate to be queued for execution by the Dispatcher on its thread. ASP.NET provides an AspNetSynchronizationContext, which is used to ensure that asynchronous operations that occur as part of the processing of an ASP.NET request are executed serially and are associated with the right HttpContext state. And so on. All told, there are around 10 concrete implementations of SynchronizationContext within the .NET Framework, some public, some internal.

When awaiting Tasks and other awaitable types provided by the .NET Framework, the “awaiters” for those types (like TaskAwaiter) capture the current SynchronizationContext at the time the await is issued. Upon completion of the awaitable, if there was a current SynchronizationContext that got captured, the continuation representing the remainder of the asynchronous method is posted to that SynchronizationContext. With that, developers writing an asynchronous method called from a UI thread don’t need to manually marshal invocations back to the UI thread in order to modify UI controls: such marshaling is handled automatically by the Framework infrastructure.

Unfortunately, such marshaling also involves cost. For application developers using await to implement their control flow, this automatic marshaling is almost always the right solution. Libraries, however, are often a different story. Application developers typically need such marshaling because their code cares about the context under which it’s running, such as being able to access UI controls, or being able to access the HttpContext for the right ASP.NET request. Most libraries, however, do not suffer this constraint. As a result, this automatic marshaling is frequently an entirely unnecessary cost. Consider again the code shown earlier to copy data from one stream to another:

1. byte [] buffer = new byte[0x1000];
2. int numRead;
3. while((numRead = await source.ReadAsync(buffer, 0, buffer.Length)) > 0) {
4. await source.WriteAsync(buffer, 0, numRead);
5. }

If this copy operation is invoked from a UI thread, every awaited read and write operation will force the completion back to the UI thread. For a megabyte of source data and Streams that complete reads and writes asynchronously (which is most of them), that means upward of 500 hops from background threads to the UI thread. To address this, the Task and Task<TResult> types provide a ConfigureAwait method. ConfigureAwait accepts a Boolean continueOnCapturedContext parameter that controls this marshaling behavior. If the default of true is used, the await will automatically complete back on the captured SynchronizationContext. If false is used, however, the SynchronizationContext will be ignored and the Framework will attempt to continue the execution wherever the previous asynchronous operation completed. Incorporating this into the stream-copying code results in the following more efficient version:

1. byte [] buffer = new byte[0x1000];
2. int numRead;
3. while((numRead = await
4. source.ReadAsync(buffer, 0, buffer.Length).ConfigureAwait(false)) > 0) {
5. await source.WriteAsync(buffer, 0, numRead).ConfigureAwait(false);
6. }

For library developers, this performance impact alone is sufficient to warrant always using ConfigureAwait, unless it’s the rare circumstance where the library has domain knowledge of its environment and does need to execute the body of the method with access to the correct context.

There’s another reason, beyond performance, to use ConfigureAwait in library code. Suppose the preceding code, without ConfigureAwait, was in a method called CopyStreamToStreamAsync, which was invoked from a WPF UI thread, like so:

1. private void button1\_Click(object sender, EventArgs args) {
2. Stream src = …, dst = …;
3. Task t = CopyStreamToStreamAsync(src, dst);
4. t.Wait(); // deadlock!
5. }

Here, the developer should have written button1\_Click as an async method and then await-ed the Task instead of using its synchronous Wait method. The Wait method has its important uses, but it’s almost always wrong to use it for waiting in a UI thread like this. The Wait method won’t return until the Task has completed. In the case of CopyStreamToStreamAsync, the contained awaits try to Post back to the captured SynchronizationContext, and the method can’t complete until those Posts complete (because the Posts are used to process the remainder of the method). But those Posts won’t complete, because the UI thread that would process them is blocked in the call to Wait. This is a circular dependency, resulting in a deadlock. If CopyStreamToStreamAsync had instead been written using ConfigureAwait(false), there would be no circular dependency and no deadlock.

**ExecutionContext** ExecutionContext is an integral part of the .NET Framework, yet most developers are blissfully unaware of its existence. ExecutionContext is the granddaddy of contexts, encapsulating multiple other contexts like SecurityContext and LogicalCallContext, and representing everything that should be automatically flowed across asynchronous points in code. Any time you’ve used ThreadPool.QueueUserWorkItem, Task.Run, Delegate.BeginInvoke, Stream.BeginRead, WebClient.DownloadStringAsync or any other asynchronous operation in the Framework, under the covers ExecutionContext was captured if possible (via ExecutionContext.Capture), and that captured context was then used to process the provided delegate (via ExecutionContext.Run). For example, if the code invoking ThreadPool.QueueUserWorkItem was impersonating a Windows identity at the time, that same Windows identity would be impersonated in order to run the supplied WaitCallback delegate. And if the code invoking Task.Run had first stored data into the LogicalCallContext, that same data would be accessible through the LogicalCallContext within the supplied Action delegate. ExecutionContext is also flowed across awaits on tasks.

There are multiple optimizations in place in the Framework to avoid capturing and running under a captured ExecutionContext when doing so is unnecessary, as doing so can be quite expensive. However, actions like impersonating a Windows identity or storing data into LogicalCallContext will thwart these optimizations. Avoiding operations that manipulate ExecutionContext, such as WindowsIdentity.Impersonate and CallContext.LogicalSetData, results in better performance when using asynchronous methods, and when using asynchrony in general.

**Lift Your Way out of Garbage Collection**

Asynchronous methods provide a nice illusion when it comes to local variables. In a synchronous method, local variables in C# and Visual Basic are stack-based, such that no heap allocations are necessary to store those locals. However, in asynchronous methods, the stack for the method goes away when the asynchronous method is suspending at an await point. For data to be available to the method after an await resumes, that data must be stored somewhere. Thus, the C# and Visual Basic compilers “lift” locals into a state machine struct, which is then boxed to the heap at the first await that suspends so that locals may survive across await points.

Earlier in this article, I discussed how the cost and frequency of garbage collection is influenced by the number of objects allocated, while the frequency of garbage collection is also influenced by the size of objects allocated. The bigger the objects being allocated, the more often garbage collection will need to run. Thus, in an asynchronous method, the more locals that need to be lifted to the heap, the more often garbage collections will occur.

As of the time of this writing, the C# and Visual Basic compilers sometimes lift more than is truly necessary. For example, consider the following code snippet:

1. public static async Task FooAsync() {
2. var dto = DateTimeOffset.Now;
3. var dt  = dto.DateTime;
4. await Task.Yield();
5. Console.WriteLine(dt);
6. }

The dto variable isn’t read at all after the await point, and thus the value written to it before the await doesn’t need to survive across the await. However, the state machine type generated by the compiler to store locals still contains the dto reference, as shown in **Figure 4**.

**Figure 4 Local Lifting**

1. [StructLayout(LayoutKind.Sequential), CompilerGenerated]
2. private struct <FooAsync>d\_\_0 : <>t\_\_IStateMachine {
3. private int <>1\_\_state;
4. public AsyncTaskMethodBuilder <>t\_\_builder;
5. public Action <>t\_\_MoveNextDelegate;
7. public DateTimeOffset <dto>5\_\_1;
8. public DateTime <dt>5\_\_2;
9. private object <>t\_\_stack;
10. private object <>t\_\_awaiter;
12. public void MoveNext();
13. [DebuggerHidden]
14. public void <>t\_\_SetMoveNextDelegate(Action param0);
15. }

This slightly bloats the size of that heap object beyond what’s truly necessary. If you find that garbage collections are occurring more frequently than you expect, take a look at whether you really need all of the temporary variables you’ve coded into your asynchronous method. This example could be rewritten as follows to avoid the extra field on the state machine class:

1. public static async Task FooAsync() {
2. var dt = DateTimeOffset.Now.DateTime;
3. await Task.Yield();
4. Console.WriteLine(dt);
5. }

Moreover, the .NET garbage collector (GC) is a generational collector, meaning that it partitions the set of objects into groups, known as generations: at a high-level, new objects are allocated in generation 0, and then all objects that survive a collection are promoted up a generation (the .NET GC currently uses generations 0, 1 and 2). This enables faster collections by allowing the GC to frequently collect only from a subset of the known object space. It’s based on the philosophy that objects newly allocated will also go away quickly, while objects that have been around for a long time will continue to be around for a long time. What this means is that if an object survives generation 0, it will likely end up being around for a while, continuing to put pressure on the system for that additional time. And that means we really want to ensure that objects are made available to garbage collection as soon as they’re no longer needed.

With the aforementioned lifting, locals get promoted to fields of a class that stays rooted for the duration of the asynchronous method’s execution (as long as the awaited object properly maintains a reference to the delegate to invoke upon completion of the awaited operation). In synchronous methods, the JIT compiler is able to keep track of when locals will never again be accessed, and at such points can help the GC to ignore those variables as roots, thus making the referenced objects available for collection if they’re not referenced anywhere else. However, in asynchronous methods, these locals remain referenced, which means the objects they reference may survive much longer than if these had been real locals. If you find that objects are remaining alive well past their use, consider nulling out the locals referencing those objects when you’re done with them. Again, this should be done only if you find that it’s actually the cause of a performance problem, as it otherwise complicates the code unnecessarily. Furthermore, the C# and Visual Basic compilers could be updated by final release or otherwise in the future to handle more of these scenarios on the developer’s behalf, so any such code written today is likely to become obsolete in the future.

**Avoid Complexity**

The C# and Visual Basic compilers are fairly impressive in terms of where you’re allowed to use awaits: almost anywhere. Await expressions may be used as part of larger expressions, allowing you to await Task<TResult> instances in places you might have any other value-returning expression. For example, consider the following code, which returns the sum of three tasks’ results:

1. public static async Task<int> SumAsync(
2. Task<int> a, Task<int> b, Task<int> c)
3. {
4. return Sum(await a, await b, await c);
5. }
7. private static int Sum(int a, int b, int c)
8. {
9. return a + b + c;
10. }

The C# compiler allows you to use the expression “await b” as an argument to the Sum function. However, there are multiple awaits here whose results are passed as parameters to Sum, and due to order of evaluation rules and how async is implemented in the compiler, this particular example requires the compiler to “spill” the temporary results of the first two awaits. As you saw previously, locals are preserved across await points by having them lifted into fields on the state machine class. However, for cases like this one, where the values are on the CLR evaluation stack, those values aren’t lifted into the state machine but are instead spilled to a single temporary object and then referenced by the state machine. When you complete the await on the first task and go to await the second one, the compiler generates code that boxes the first result and stores the boxed object into a single <>t\_\_stack field on the state machine. When you complete the await on the second task and go to await the third one, the compiler generates code that creates a Tuple<int,int> from the first two values, storing that tuple into the same <>\_\_stack field. This all means that, depending on how you write your code, you could end up with very different allocation patterns. Consider instead writing SumAsync as follows:

1. public static async Task<int> SumAsync(
2. Task<int> a, Task<int> b, Task<int> c)
3. {
4. int ra = await a;
5. int rb = await b;
6. int rc = await c;
7. return Sum(ra, rb, rc);
8. }

With this change, the compiler will now emit three more fields onto the state machine class to store ra, rb and rc, and no spilling will occur. Thus, you have a trade-off: a larger state machine class with fewer allocations, or a smaller state machine class with more allocations. The total amount of memory allocated will be larger in the spilling case, as each object allocated has its own memory overhead, but in the end performance testing could reveal that’s still better. In general, as mentioned previously, you shouldn’t think through these kinds of micro-optimizations unless you find that the allocations are actually the cause of grief, but regardless, it’s helpful to know where these allocations are coming from.

Of course, there’s arguably a much larger cost in the preceding examples that you should be aware of and proactively consider. The code isn’t able to invoke Sum until all three awaits have completed, and no work is done in between the awaits. Each of these awaits that yields requires a fair amount of work, so the fewer awaits you need to process, the better. It would behoove you, then, to combine all three of these awaits into just one by waiting on all of the tasks at once with Task.WhenAll:

1. public static async Task<int> SumAsync(
2. Task<int> a, Task<int> b, Task<int> c)
3. {
4. int [] results = await Task.WhenAll(a, b, c);
5. return Sum(results[0], results[1], results[2]);
6. }

The Task.WhenAll method here returns a Task<TResult[]> that won’t complete until all of the supplied tasks have completed, and it does so much more efficiently than just waiting on each individual task. It also gathers up the result from each task and stores it into an array. If you want to avoid that array, you can do that by forcing binding to the non-generic WhenAll method that works with Task instead of Task<TResult>. For ultimate performance, you could also take a hybrid approach, where you first check to see if all of the tasks have completed successfully, and if they have, get their resultsindividually—but if they haven’t, then await a WhenAll of those that haven’t. That will avoid any allocations involved in the call to WhenAll when it’s unnecessary, such as allocating the params array to be passed into the method. And, as previously mentioned, we’d want this library function to also suppress context marshaling. Such a solution is shown in **Figure 5**.

**Figure 5 Applying Multiple Optimizations**

1. public static Task<int> SumAsync(
2. Task<int> a, Task<int> b, Task<int> c)
3. {
4. return (a.Status == TaskStatus.RanToCompletion &&
5. b.Status == TaskStatus.RanToCompletion &&
6. c.Status == TaskStatus.RanToCompletion) ?
7. Task.FromResult(Sum(a.Result, b.Result, c.Result)) :
8. SumAsyncInternal(a, b, c);
9. }
11. private static async Task<int> SumAsyncInternal(
12. Task<int> a, Task<int> b, Task<int> c)
13. {
14. await Task.WhenAll((Task)a, b, c).ConfigureAwait(false);
15. return Sum(a.Result, b.Result, c.Result);
16. }

**Asynchronicity and Performance**

Asynchronous methods are a powerful productivity tool, enabling you to more easily write scalable and responsive libraries and applications. It’s important to keep in mind, though, that asynchronicity is not a performance optimization for an individual operation. Taking a synchronous operation and making it asynchronous will invariably degrade the performance of that one operation, as it still needs to accomplish everything that the synchronous operation did, but now with additional constraints and considerations. A reason you care about asynchronicity, then, is performance in the aggregate: how your overall system performs when you write everything asynchronously, such that you can overlap I/O and achieve better system utilization by consuming valuable resources only when they’re actually needed for execution. The asynchronous method implementation provided by the .NET Framework is well-optimized, and often ends up providing as good or better performance than well-written asynchronous implementations using existing patterns and volumes more code. Any time you’re planning to develop asynchronous code in the .NET Framework from now on, asynchronous methods should be your tool of choice. Still, it’s good for you as a developer to be aware of everything the Framework is doing on your behalf in these asynchronous methods, so you can ensure the end result is as good as it can possibly be.

**Should I expose asynchronous wrappers for synchronous methods?**

Lately I’ve received several questions along the lines of the following, which I typically summarize as “async over sync”:

In my library, I have a method “public T Foo();”.  I’m considering exposing an asynchronous method that would simply wrap the synchronous one, e.g. “public Task<T> FooAsync() { return Task.Run(() => Foo()); }”.  Is this something you’d recommend I do in my library?

My short answer to such a question is “no.”  But that doesn’t make for a very good blog post.  So here’s my longer, more reasoned answer…

**Why Asynchrony?**

There are two primary benefits I see to asynchrony: scalability and offloading (e.g. responsiveness, parallelism).  Which of these benefits matters to you is typically dictated by the kind of application you’re writing.  Most client apps care about asynchrony for offloading reasons, such as maintaining responsiveness of the UI thread, though there are certainly cases where scalability matters to a client as well (often in more technical computing / agent-based simulation workloads).  Most server apps care about asynchrony for scalability reasons, though there are cases where offloading matters, such as in achieving parallelism in back-end compute servers.

**Scalability**

The ability to invoke a synchronous method asynchronously does nothing for scalability, because you’re typically still consuming the same amount of resources you would have if you’d invoked it synchronously (in fact, you’re using a bit more, since there’s overhead incurred to scheduling something ), you’re just using different resources to do it, e.g. a thread from a thread pool instead of the specific thread you were executing on.  The scalability benefits touted for asynchronous implementations are achieved by decreasing the amount of resources you use, and that needs to be baked into the implementation of an asynchronous method… it’s not something achieved by wrapping around it.

As an example, consider a synchronous method Sleep that doesn’t return for N milliseconds:

public void Sleep(int millisecondsTimeout)   
{   
    Thread.Sleep(millisecondsTimeout);   
}

Now, consider the need to create an asynchronous version of this, such that the returned Task doesn’t complete for N milliseconds.  Here’s one possible implementation, simply wrapping Sleep with Task.Run to create a SleepAsync:

public Task SleepAsync(int millisecondsTimeout)   
{   
    return Task.Run(() => Sleep(millisecondsTimeout));   
}

and here’s another that doesn’t use Sleep, instead rewriting the implementation to consume fewer resources:

public Task SleepAsync(int millisecondsTimeout)   
{   
    TaskCompletionSource<bool> tcs = null;   
    var t = new Timer(delegate { tcs.TrySetResult(true); }, null, –1, -1);   
    tcs = new TaskCompletionSource<bool>(t);   
    t.Change(millisecondsTimeout, -1);   
    return tcs.Task;   
}

Both of these implementations provide the same basic behavior, both completing the returned task after the timeout has expired.  However, from a scalability perspective, the latter is much more scalable.  The former implementation consumes a thread from the thread pool for the duration of the wait time, whereas the latter simply relies on an efficient timer to signal the Task when the duration has expired.

**Offloading**

The ability to invoke a synchronous method asynchronously can be very useful for responsiveness, as it allows you to offload long-running operations to a different thread.  This isn’t about how many resources you consume, but rather is about which resources you consume.  For example, in a UI app, the specific thread handling pumping UI messages is “more valuable” for the user experience than are other threads, such as those in the ThreadPool.  So, asynchronously offloading the invocation of a method from the UI thread to a ThreadPool thread allows us to use the less valuable resources.  This kind of offloading does not require modification to the implementation of the operation being offloaded, such that the responsiveness benefits can be achieved via wrapping.

The ability to invoke a synchronous method asynchronously can also be very useful not just for changing threads, but more generally for escaping the current context.  For example, sometimes we need to invoke some user-provided code but we’re not in a good place to do it (or we’re not sure if we are).  Maybe a lock is held higher up the stack and we don’t want to invoke the user code while holding the lock.  Maybe we suspect we’re being invoked by some user code that doesn’t expect us to take a very long time. Rather than invoking the operation synchronously and as part of whatever is higher-up on the call stack, we can invoke the functionality asynchronously.

The ability to invoke a synchronous method asynchronously is also important for parallelism.  Parallel programming is all about taking a single problem and splitting it up into sub-problems that can each be processed concurrently.  If you were to split a problem into sub-problems but then process each sub-problem serially, you wouldn’t get any parallelism, as the entire problem would be processed on a single thread.  If, instead, you offload a sub-problem to another thread via asynchronous invocation, you can then process the sub-problems concurrently.  As with responsiveness, this kind of offloading does not require modification to the implementation of the operation being offloaded, such that parallelism benefits can be achieved via wrapping.

**What does this have to do with my question?**

Let’s get back to the core question: should we expose an asynchronous entry point for a method that’s actually synchronous?  The stance we’ve taken in .NET 4.5 with the [Task-based Async Pattern](http://www.microsoft.com/download/en/details.aspx?id=19957) is a staunch “no.”

Note that in my previous discussion of scalability and ofloading, I called out that the way to achieve scalability benefits is by modifying the actual implementation, whereas offloading can be achieved by wrapping and doesn’t require modifying the actual implementation.  That’s the key.  Wrapping a synchronous method with a simple asynchronous façade does not yield any scalability benefits.  And in such cases, by exposing only the synchronous method, you get some nice benefits, e.g.

* Surface area of your library is reduced.  This means less cost to you (development, testing, maintenance, documentation, etc.).  It also means that your user’s choices are simplified.  While some choice is typically a good thing, too much choice often leads to lost productivity.  If I as a user am constantly faced with both a synchronous and an asynchronous method for the same operation, I constantly need to evaluate which of the pairs is the right one for me to use in each situation.
* Your users will know whether there are actually scalability benefits to using exposed asynchronous APIs, since by definition then only APIs that benefit scalability are exposed asynchronously.
* The choice of whether to invoke the synchronous method asynchronously is left up to the developer. Async wrappers around sync methods have overhead (e.g. allocating the object to represent the operation, context switches, synchronization around queues, etc.).  If, for example, your customer is writing a high-throughput server app, they don’t want to spend cycles on overhead that’s not actually benefiting them in any way, so they can just invoke the synchronous method.  If both the synchronous method and an asynchronous wrapper around it are exposed, the developer is then faced with thinking they should invoke the asynchronous version for scalability reasons, but in reality will actually be hurting their throughput by paying for the additional offloading overhead without the scalability benefits.

If a developer needs to achieve better scalability, they can use any async APIs exposed, and they don’t have to pay additional overhead for invoking a faux async API.  If a developer needs to achieve responsiveness or parallelism with synchronous APIs, they can simply wrap the invocation with a method like Task.Run.

The idea of exposing “async over sync” wrappers is also a very slippery slope, which taken to the extreme could result in every single method being exposed in both synchronous and asynchronous forms.  Many of the folks that ask me about this practice are considering exposing async wrappers for long-running CPU-bound operations.  The intention is a good one: help with responsiveness.  But as called out, responsiveness can easily be achieved by the consumer of the API, and the consumer can actually do so at the right level of chunkiness, rather than for each chatty individual operation.  Further, defining what operations could be long-running is surprisingly difficult.  The time complexity of many methods often varies significantly.

Consider, for example, a simple method like Dictionary<TKey,TValue>.Add(TKey,TValue).  This is a really fast method, right?  Typically, yes, but remember how dictionary works: it needs to hash the key in order to find the right bucket to put it into, and it needs to check for equality of the key with other entries already in the bucket.  Those hashing and equality checks can result in calls to user code, and who knows what those operations do or how long they take.  Should every method on dictionary have an asynchronous wrapper exposed? That’s obviously an extreme example, but there are simpler ones, like Regex.  The complexity of the regular expression pattern provided to Regex as well as the nature and size of the input string can have significant impact on the running time of matching with Regex, so much so that Regex now supports optional timeouts… should every method on Regex have an asynchronous equivalent?  I really hope not.

**Guideline**

This has all been a very long-winded way of saying that I believe the only asynchronous methods that should be exposed are those that have scalability benefits over their synchronous counterparts.  Asynchronous methods should not be exposed purely for the purpose of offloading: such benefits can easily be achieved by the consumer of synchronous methods using functionality specifically geared towards working with synchronous methods asynchronously, e.g. Task.Run.

Of course, there are exceptions to this, and you can witness a few such exceptions in .NET 4.5.

For example, the abstract base Stream type provides ReadAsync and WriteAsync methods.  In most cases, derived Stream implementations work with data sources that aren’t in-memory, and thus involve disk I/O or network I/O of some kind.  As such, it’s very likely that derived implementations will be able to provide implementations of ReadAsync and WriteAsync that utilize asynchronous I/O rather than synchronous I/O that blocks threads, and thus there are scalability benefits to having ReadAsync and WriteAsync methods.  Further, we want to be able to work with these methods polymorphically, without regard for the concrete stream type, so we want to have these as virtual methods on the base class.  However, the base class doesn’t know how to implement these base implementations with asynchronous I/O, so the best it can do is provide asynchronous wrappers for the synchronous Read and Write methods (in actuality, ReadAsync and WriteAsync actually wrap BeginRead/EndRead and BeginWrite/EndWrite, respectively, which if not overridden will in turn wrap the synchronous Read and Write methods with an equivalent of Task.Run).

Another example in the same vein is TextReader, providing methods like ReadToEndAsync, which on the base class simply uses a Task to wrap an invocation of TextReader.ReadToEnd.  The expectation, however, is that the derived types developers actually use will override ReadToEndAsync to provide implementations that benefit scalability, such as StreamReader’s ReadToEndAsync method which utilizes Stream.ReadAsync.