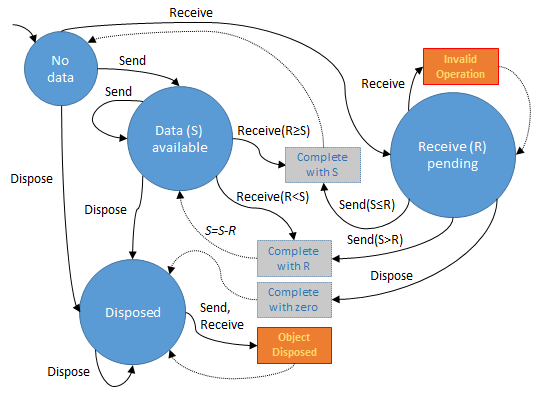
To demonstrate these principles along with typical .NET 4.5 asynchronous patterns, I came up with a useful and complete example — MemoryChannel. MemoryChannel is a simple one-way communication channel which allows a sender to pass raw byte buffers to an asynchronous receiver. With some appropriate adapters, it could replace a named pipe or TCP socket (thus helping to eliminating external dependencies, for example, in a set of unit tests).

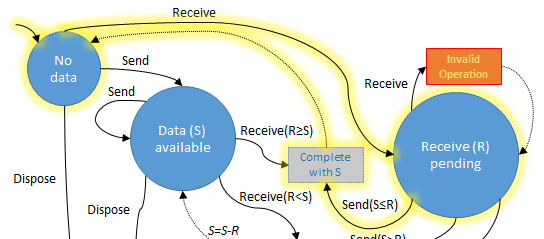
Before writing the code, I sketched a crude [FSM diagram](http://en.wikipedia.org/wiki/Finite-state_machine) on a piece of paper. Here is a much more refined version of that diagram (courtesy of [PowerPoint](http://office.microsoft.com/en-us/powerpoint/), my diagramming tool of choice):



There are only four states (no data, data available, receive pending, and disposed) of the channel and three operations (send, receive, dispose) applied to the channel. The only real complications here are the conditions relating to the sent and received data sizes. For example, if there is some data in the channel and a receiver requests more data than is available, the request should still complete but notify the receiver that only S bytes were transferred. However, if less data is requested (R), the excess (S - R) should remain available for later receive attempts.

In true TDD style, I started with the minimum possible interface and decided to focus on the receive side of the state machine.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class MemoryChannel  {      public MemoryChannel()      {      }        public Task ReceiveAsync(byte[] buffer)      {          throw new NotImplementedException();      }        public void Send(byte[] buffer)      {          throw new NotImplementedException();      }  } |

Here is the first unit test, covering just the highlighted part of the state machine:  


So initially, the channel is in “No data” state, then there is a request to receive data. The channel should go into “Receive pending” state. If anyone send to the channel later with size equal to the requested data size, it complete with S and goes to the “No data” state

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public void Pending\_receive\_completes\_after\_send\_with\_same\_data\_size()  {      MemoryChannel channel = new MemoryChannel();        byte[] receiveBuffer = new byte[3];      Task<int> receiveTask = channel.ReceiveAsync(receiveBuffer);        Assert.False(receiveTask.IsCompleted);      Assert.False(receiveTask.IsFaulted);        byte[] sendBuffer = new byte[] { 1, 2, 3 };      channel.Send(sendBuffer);        Assert.Equal(TaskStatus.RanToCompletion, receiveTask.Status);      Assert.Equal(3, receiveTask.Result);      Assert.Equal(new byte[] { 1, 2, 3 }, receiveBuffer);  } |

To fulfill the conditions of this test, we need to return a Task<int> from ReceiveAsync which *has not yet completed* and only does so when a matching Send call is made. Additionally, to satisfy the unit testing principles above we need to do this without external threads. How can we do this?

Enter [TaskCompletionSource](http://blogs.msdn.com/b/pfxteam/archive/2009/06/02/9685804.aspx" \o "The Nature of TaskCompletionSource<TResult>) — one of the most useful library classes in the async .NET toolbox. Essentially, TCS lets us hand out a Task representing a custom asynchronous operation while giving us the ability to explicitly control the lifetime of this operation.

To make the test above pass, we can use TCS to represent the pending receive operation like so, initializing it on start of receive and completing it (via [SetResult](http://msdn.microsoft.com/en-us/library/dd449202(v=vs.110).aspx" \o "TaskCompletionSource<TResult>.SetResult Method)) when Send is invoked:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class MemoryChannel  {      private TaskCompletionSource<int> pendingReceive;      private byte[] pendingReceiveBuffer;        public MemoryChannel()      {      }        public Task<int> ReceiveAsync(byte[] buffer)      {          this.pendingReceive = new TaskCompletionSource<int>();          this.pendingReceiveBuffer = buffer;          return this.pendingReceive.Task;      }        public void Send(byte[] buffer)      {          buffer.CopyTo(this.pendingReceiveBuffer, 0);          this.pendingReceive.SetResult(buffer.Length);      }  } |

Note that SetResult will transition the task to a completed state as well **run any synchronous continuations**. This can be undesirable in some situations which I will get into in a later post.

You can track the evolution of MemoryChannel into its full implementation in my [CommSample project](https://github.com/brian-dot-net/writeasync/tree/master/projects/CommSample" \o "CommSample project). The[commit history](https://github.com/brian-dot-net/writeasync/commits/master/projects/CommSample) shows the changes following every new unit test or refactoring.

## Section II

In a [previous post](http://writeasync.net/?p=211), I described MemoryChannel and how I used TDD to implement it. After [the 25th commit](https://github.com/brian-dot-net/writeasync/commit/4f739a14e3833fdf7c2ebc1e56c6b4e5103ccfd7), 20 unit tests later, I had a fully functional *but single-threaded* implementation.

Simply by inspection, I knew that this code would fail almost immediately if a sender and receiver were ever executed on concurrent threads. For example, Send and ReceiveAsync can potentially modify the excessBuffers [LinkedList](http://msdn.microsoft.com/en-us/library/he2s3bh7(v=vs.110).aspx) concurrently which may corrupt its internal state.

I figured that a thread-safe MemoryChannel would be generally more useful, if a bit more complex. Thus I began devising an integration test application, knowing that my unit tests could not be of much help here.

I started by writing a simple skeleton test app with a Receiver class (async receive loop) and a Sender class (background thread send loop). The test app started out like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Logger logger = new Logger();  MemoryChannel channel = new MemoryChannel();    Receiver receiver = new Receiver(channel, logger, 16);  Sender sender = new Sender(channel, logger, 16, 1);    Task receiverTask = receiver.RunAsync();  Task senderTask = sender.RunAsync();    Task.WaitAll(receiverTask, senderTask);    channel.Dispose();    logger.WriteLine("Done."); |

[At this point](https://github.com/brian-dot-net/writeasync/commit/b62bf367206c774e43830d674b44b77078334f66), I had only implemented the receive loop — the Sender was just a single Send operation running on a background thread. But this was enough to *consistently* trigger an invalid state:

System.InvalidOperationException: A receive operation is already in progress.  
at CommSample.MemoryChannel.ReceiveAsync(Byte[] buffer) in CommSample\source\CommSample.Core\MemoryChannel.cs:line 39  
at CommSample.Receiver.<RunAsync>d\_\_0.MoveNext() in CommSample\source\CommSample.App\Receiver.cs:line 34  
--- End of inner exception stack trace ---  
at System.Threading.Tasks.Task.WaitAll(Task[] tasks, Int32 millisecondsTimeout, CancellationToken cancellationToken)  
at System.Threading.Tasks.Task.WaitAll(Task[] tasks, Int32 millisecondsTimeout)  
at System.Threading.Tasks.Task.WaitAll(Task[] tasks)  
at CommSample.Program.Main(String[] args) in CommSample\source\CommSample.App\Program.cs:line 24

Using the debugger and setting a breakpoint on the exception condition in MemoryChannel.ReceiveAsync, we can get a more useful stack trace that illustrates the problem:

* CommSample.Core.dll!CommSample.MemoryChannel.ReceiveAsync(byte[] buffer) Line 36
* CommSample.App.exe!CommSample.Receiver.RunAsync() Line 34
* [Resuming Async Method]
* *[ . . . ]*
* mscorlib.dll!System.Threading.Tasks.TaskCompletionSource<int>.SetResult(int result)
* CommSample.Core.dll!CommSample.MemoryChannel.ReceiveRequest.TryComplete(bool disposing) Line 161
* CommSample.Core.dll!CommSample.MemoryChannel.Send(byte[] buffer) Line 75
* CommSample.App.exe!CommSample.Sender.RunInner() Line 41
* *[ . . . ]*
* mscorlib.dll!System.Threading.ThreadHelper.ThreadStart(object obj)

Ah, so it is the **synchronous continuation execution** triggered by TaskCompletionSource.SetResult that is biting us here!

If we do

this.pendingReceiveTaskSource.SetResult(lengh);

this.pendingReceiveBuffer = null;

SetResult will trigger the continuation task while the next null assignment has not executed yet.

Another issue is the state sync between pendingReceiveBuffer and pendingReceiveTaskSource. If one receive followed by two consecutive calls of send(),

Refectory needed:

To help explain further, let’s make a small modification to the Receiver loop:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | do  {      this.logger.WriteLine("Before await...");      try      {          bytesRead = await this.channel.ReceiveAsync(buffer);      }      catch (Exception e)      {          this.logger.WriteLine("Receive threw: {0}", e);          throw;      }        this.logger.WriteLine("After await...");      totalBytes += bytesRead;  }  while (bytesRead > 0); |

Running the app now results in the following output:

[0000.004/T01] Before await...  
[0000.008/T01] Sender B=16/F=0x1 starting...  
[0000.010/T03] After await...  
[0000.010/T03] Before await...  
[0000.012/T03] Receive threw: System.InvalidOperationException: A receive operation is already in progress.  
at CommSample.MemoryChannel.ReceiveAsync(Byte[] buffer) in CommSample.Core\MemoryChannel.cs:line 39  
at CommSample.Receiver.d\_\_0.MoveNext() in CommSample\source\CommSample  
.App\Receiver.cs:line 37  
[0000.012/T03] Sender B=16/F=0x1 completed. Sent 16 bytes.

As expected, the first await we hit returns control back to the caller with one receive now in progress. Note that the code *after* the first await and *before* the next await is scheduled as a continuation at this point. The sender then runs which fulfills the pending receive. This triggers the continuation, which in turn raises an exception since the code to set pendingReceive to null hasn’t had a chance to run yet.

Technically this issue could be induced by a single-threaded unit test with some clever use of [ContinueWith](http://msdn.microsoft.com/en-us/library/dd270696(v=vs.110).aspx). However, the “fix” one would likely make in response would not solve the broad issue of race conditions inherent in the code at this point.

Instead, I forged ahead and reorganized the code to eliminate thread-safety issues. I started by [adding locking to the receive logic](https://github.com/brian-dot-net/writeasync/commit/ea52a9bfac7ccb8fb42818c983ed7edb8c78930e). I noted that excessBuffers, being a reference type and a read-only private instance field, would work fine as a lock object. I moved all of the logic of this method under this lock, save for the final conditional logic to complete the result (to prevent deadlocks, you should generally avoid running arbitrary user code such as event handlers or in this case, continuations, under a lock).

This was a fine start but it was only the beginning. With locking, *everyone* must participate in order to ensure thread-safety. To complete the implementation, I added locking to [send](https://github.com/brian-dot-net/writeasync/commit/be3abffb31b0a5f681575b23258e79e62bd62690) (with one [later fix](https://github.com/brian-dot-net/writeasync/commit/b6e3aeb623670eeed99c72ede8847e1ca34d4afe)) and [dispose](https://github.com/brian-dot-net/writeasync/commit/29f712bef7ff32a8c24ba902b5dbe4d8779cc659).

At this point, I was ready to complete the real integration test application. The story continues in the next post…

MemoryChannel integration test

In a [previous post](http://writeasync.net/?p=311), I discussed the concurrency issues with the initial MemoryChannel implementation and how unit tests were insufficient to uncover them.

I came up with these basic requirements/invariants to guide my integration test design:

* Data from separately sent buffers **must not** be mixed or interleaved. That is, if sender 1 writes "FFFF" and sender 2 writes "CCCC", the receiver may only see "CCCCFFFF" or "FFFFCCCC" (depending on how the writes were serialized).
* Assuming the channel is fully drained, all sent buffers **must** eventually be delivered; data **must not** be lost.
* The requirements above **must** hold given a single receiver and one or more senders operating concurrently.

The basic test flow would thus be as follows:

1. Create MemoryChannel.
2. Start senders on background threads; run until canceled.
3. Start async receiver; validate data after each receive operation.
4. Run for some predetermined duration.
5. Cancel senders; wait for tasks to complete.
6. Dispose MemoryChannel; this will unblock the last receive operation.
7. Validate sent and received data size.

In the [final refactoring](https://github.com/brian-dot-net/writeasync/commit/961f95afabf5f02d65d3eb434c1fa9fcf1588350), the main body of the test code ended up as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | MemoryChannel channel = new MemoryChannel();  int[] sentDataSizes = new int[] { 11, 19, 29, 41, 53, 71, 89, 101 };    using (CancellationTokenSource cts = new CancellationTokenSource())  {      DataOracle oracle = new DataOracle();      Sender[] senders = this.CreateSenders(channel, sentDataSizes, oracle);        ValidatingReceiver receiver = new ValidatingReceiver(channel, this.logger, this.receiveBufferSize, oracle);        Task[] senderTasks = new Task[senders.Length];      Task receiverTask = this.StartSendersAndReceiver(cts.Token, senders, receiver, senderTasks);      Thread.Sleep(this.duration);        cts.Cancel();      Task.WaitAll(senderTasks);        channel.Dispose();      receiverTask.Wait();        ValidateTransferredByteCount(senderTasks, receiverTask);  } |

My data validation strategy uses sent buffers of prime number sizes filled with a specific byte value per sender and a larger power of two for receive buffer size; for example, sender 1 uses a buffer of size 11 filled with 0x1, sender 2 uses a buffer of size 19 filled with 0x2 and so on, with a receiver asking for 256 bytes at a time. My thinking was that it would be easier to detect state corruption if I used numbers like these (I admittedly have not done any rigorous mathematical proofs of this…). I implemented a [ValidatingReceiver](https://github.com/brian-dot-net/writeasync/commit/fa1423bb49d5df8d597674b748c6b8c67116b8da) which scans all received buffers and looks for runs of the same byte value. On each byte value change, the results are fed to a [DataOracle](https://github.com/brian-dot-net/writeasync/commit/218c004f976532e1dc69bec8c6466ad9b19fa0f3) which knows the mapping of expected byte values to buffer length multiples:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public void VerifyLastSeen(byte lastSeen, int lastCount)  {      int expectedCountMultiple;      if (!this.patterns.TryGetValue(lastSeen, out expectedCountMultiple))      {          string message = string.Format(              CultureInfo.InvariantCulture,              "State corruption detected; byte 0x{0:X} was unexpected.",              lastSeen);          throw new InvalidOperationException(message);      }        if (lastCount % expectedCountMultiple != 0)      {          string message = string.Format(              CultureInfo.InvariantCulture,              "State corruption detected; count of {0} for byte 0x{1:X} is not a multiple of {2}.",              lastCount,              lastSeen,              expectedCountMultiple);          throw new InvalidOperationException(message);      }  } |

By the [final commit](https://github.com/brian-dot-net/writeasync/commit/c86f1c3010e2d99ce094374863c92aff48efb554), I had a fully automated integration test app which gave me reasonable confidence that my logic was correct:  
  
[ . . . ]  
[0035.048/T01] Receive loop with 8 senders, 5.0 sec, send before receive=True, receive buffer=256...  
[0035.048/T01] Sender B=11/F=0x1 starting...  
[0035.048/T01] Sender B=19/F=0x2 starting...  
[0035.048/T01] Sender B=29/F=0x3 starting...  
[0035.049/T01] Sender B=41/F=0x4 starting...  
[0035.049/T01] Sender B=53/F=0x5 starting...  
[0035.049/T01] Sender B=71/F=0x6 starting...  
[0035.049/T01] Sender B=89/F=0x7 starting...  
[0035.050/T01] Sender B=101/F=0x8 starting...  
[0035.050/T01] Receiver starting...  
[0040.050/T35] Sender B=41/F=0x4 completed. Sent 409631 bytes.  
[0040.050/T39] Sender B=101/F=0x8 completed. Sent 1009091 bytes.  
[0040.051/T36] Sender B=53/F=0x5 completed. Sent 529470 bytes.  
[0040.051/T38] Sender B=89/F=0x7 completed. Sent 889288 bytes.  
[0040.051/T37] Sender B=71/F=0x6 completed. Sent 709290 bytes.  
[0040.051/T33] Sender B=19/F=0x2 completed. Sent 189886 bytes.  
[0040.051/T32] Sender B=11/F=0x1 completed. Sent 109956 bytes.  
[0040.051/T34] Sender B=29/F=0x3 completed. Sent 289826 bytes.  
[0040.053/T01] Receiver completed. Received 4136438 bytes.  
[0040.053/T01] Done.