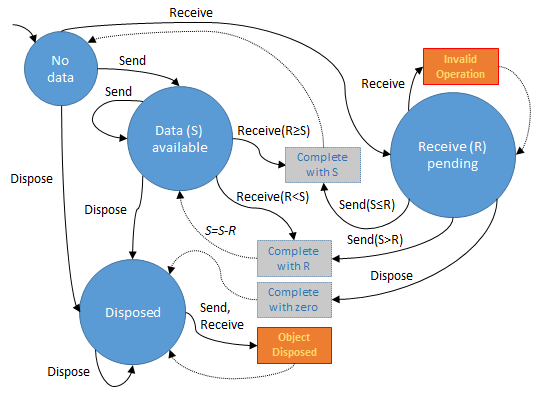
To demonstrate these principles along with typical .NET 4.5 asynchronous patterns, I came up with a useful and complete example — MemoryChannel. MemoryChannel is a simple one-way communication channel which allows a sender to pass raw byte buffers to an asynchronous receiver. With some appropriate adapters,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

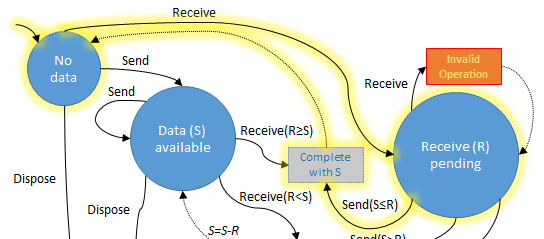
Before writing the code, I sketched a crude [FSM diagram](http://en.wikipedia.org/wiki/Finite-state_machine) on a piece of paper. Here is a much more refined version of that diagram (courtesy of [PowerPoint](http://office.microsoft.com/en-us/powerpoint/), my diagramming tool of choice):



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.