In Lab 6, our final lab, we were asked to create a data acquisition server that could filter and transmit captured signals to a client. This lab is the final project of the semester, and as such, is a culmination of the skills learned throughout the semester. The following class objectives were tested during this lab:

* Real-time data acquisition
* Parallel processing
* Signal filtering
* Thread synchronization
* Network communication

These concepts proved invaluable in implementing the design and final solution for the lab. I also added an additional challenge to this lab, by implementing the solution in C# using Microsoft’s .NET framework and the OpenLayer’s .NET libraries. Often in this lab I was asked why I chose to write this solution in C#, and I came to the conclusion after reading the reference material from Data Translation about their device libraries for the DT9816, that the .NET libraries would provide a better and more robust solution for implementing data acquisition. I began the solution by designing a basic flow diagram detailing the algorithm for achieving my solution. The diagram is in two parts: Server and Client. The essence of the idea is to:

1. Setup a client/server connection that allows both parties to communicate a basic set of commands to one another. Even though the command set is small, we still must ensure that each side of communication is synchronized with the other so both parties must agree on a protocol to communicate messages and data back and forth.
2. Use the Open Layer’s library to communicate with the DT9816, and encapsulate that in a small class that provides only the functionality necessary for our acquisition: Continuous Analog Input and Digital Output.
3. Use parallel libraries to implement a convolution algorithm that filters the signals acquired from the device using a user-selected set of coefficients.
4. The client then issues the start command to the server and waits to receive buffers of data until the user is ready to stop.
5. Server acquires and filters the data and then sends the data packets to the client until receiving the stop command, or the signal-processing switch is deasserted.

Sounds simple, right? Well, the idea and algorithm weren’t terribly difficult to create, but the implementation did prove tricky. My process was to basically build each portion in pieces since I was starting over from a C# implementation which did not have the benefit of an already functional client/server and data acquisition algorithm. So I built unit tests for each portion of the project. The first unit test I ran was to setup a client/server connection and then use the test coefficients and signals from our assignment to verify functionality of both the data transmission and convolution. You can view this verification in the screenshot below.

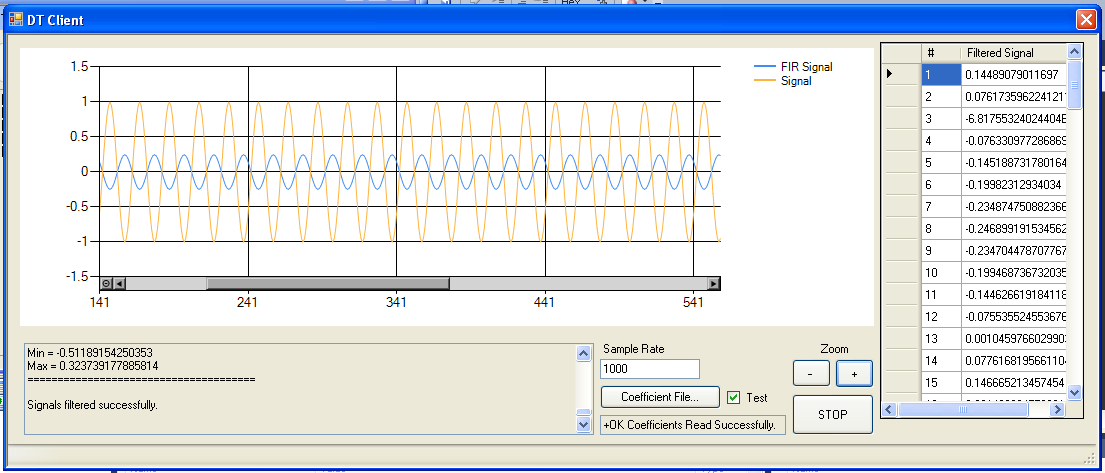
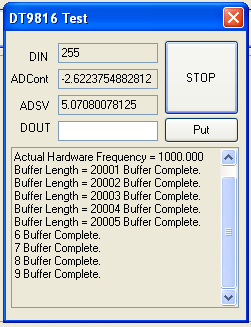


Figure Coefficient-Signal Test

Although it may not be apparent from the screenshot, it does indeed match the output given for the assignment, a sine wave that cycles between approximately -0.2 and 0.2. This is also verified functionality of the client and server, as the simulation was done as if acquiring a real-time signal. For example, the client sent the coefficients and the server then ran the coefficients against the convolution filter and transmitted it back to the client.

The next test was also rather trivial. In this test I wanted to ensure the functionality of the continuous data acquisition. Here I used a small test application that allowed would read digital and analog signals from the device and output a digital value that the user entered. A screenshot of the test application is below.



Now for the grand finale: connect the device to a generated sine wave and acquire, filter and transmit the data in real time. Since the testing for each phase had already been completed this step was also completed, but took a bit of grunt work to iron out a few unexpected errors. The biggest issue we had at this phase was that the filtering and transmission of the data could not keep up with the acquisition. The DT9816 only allows up 9 buffers for data capture and if you can remove the buffers before it reaches capacity it ends up throwing a runtime error, which causes the application to crash. This issue was solved by increasing the sample size. It turns out that trying to filter 100 samples at a time is probably not an efficient use of a processor that can execute a billion instructions per second. When the sample size was increased from 100 to 1000 our issues with buffer overflow were resolved.

The only other issue to overcome was testing true real-time data acquisition over a network on two different computers. So we already had the issue with the data not being filtered and transmitted fast enough because the sample size was too small, well we had another similar issue once we ran the client and server on two different computers. It also turns out that sending small packet sizes is also not an efficient use of computer (or rather, network) resources, and again, we kept getting runtime errors because the buffers were too small. We resolved this by increasing our packet size from 100 signals to 1000 signals – an entire sample – which ended up working perfectly. Now our device had no issue emptying the buffer and processing the signals.

Test screenshots are below:

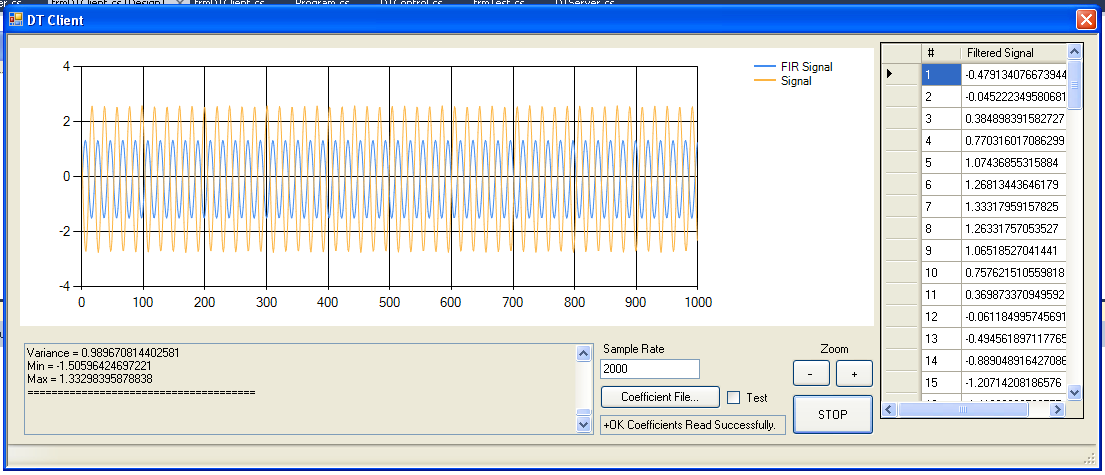


Figure 70HZ coefficients, 70HZ input signal

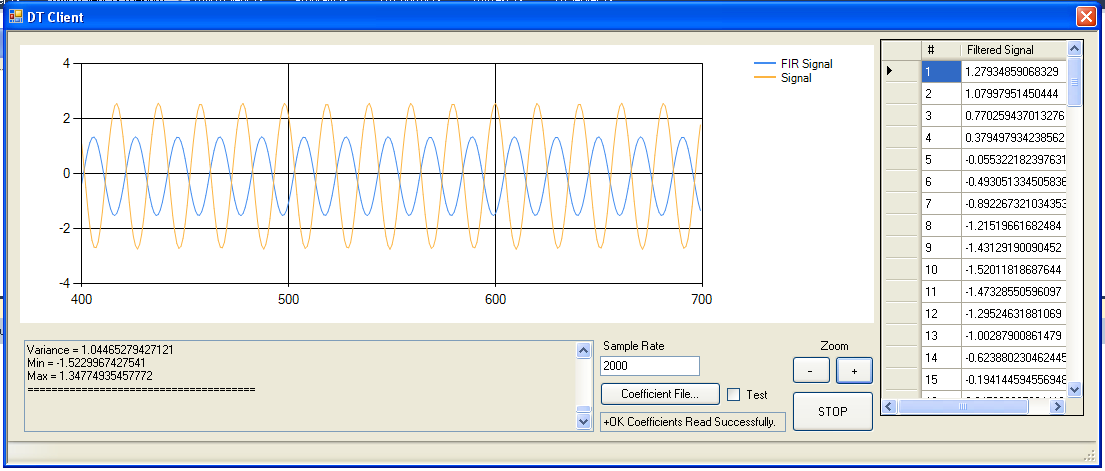


Figure 100HZ coefficients, 100HZ signals (w/ ZOOM)

Additional test screenshots are included in the attachments.

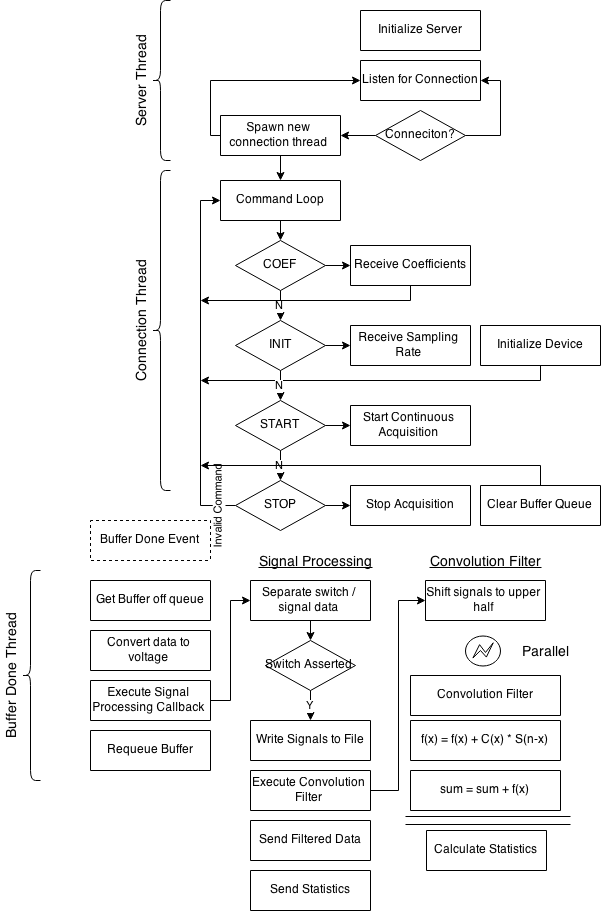


Figure Server Flow Diagram

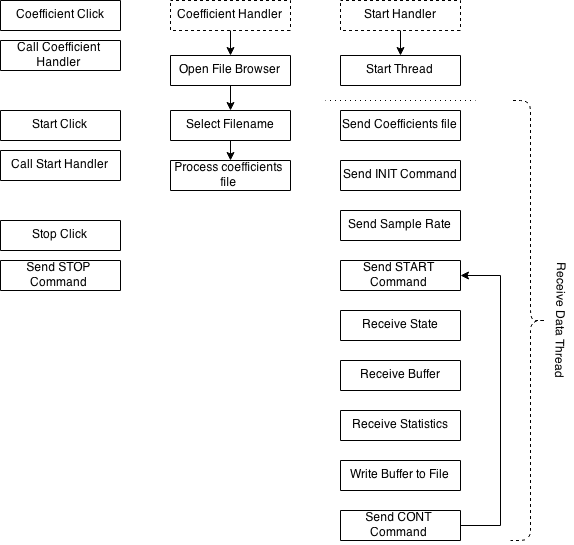


Figure Client Flow Diagram

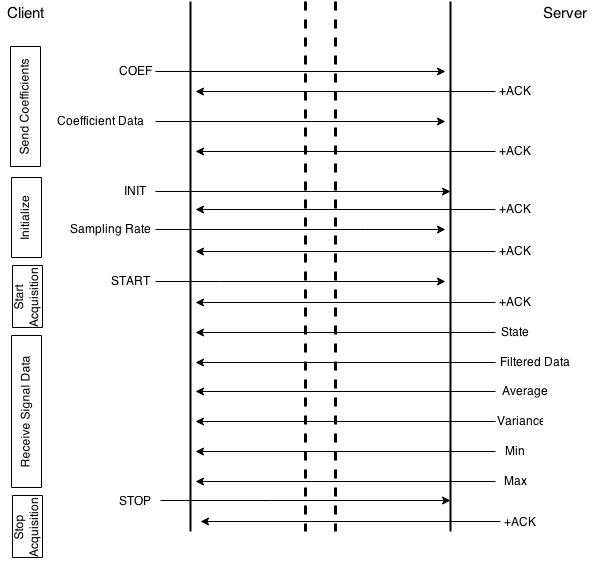


Figure Client/Server Protocol



Figure Block Diagram