Design of a Real-data Data Acquisition System with Network Communications

By: Alex Limon Naranjo
ID: 1000818599
Prepared for: Dr. Walker
Section: CSE 4342-001

Introduction:

This project is a real time data acquisition client-server system that takes in an analog sine wave from a signal generator and based on user commands, filters the data and outputs some statistics about the data to the client. The system is based in C/C++ using Windows API, and DataAcq SDK. The acquisition of the analog data is interfaced using a powerful analog to digital converter – the DT9816. The DT9816 uses serial communication using a USB and DataAcq SDK to interface with the PC.

Overview of Data Flow:

The program begins with the client prompting the user for 3 parameters: the sampling rate, the name of an output file, and the name of the coefficients file. The sampling rate is used to set how frequently the DT9816 will acquire different points, since we are converting analog values to digital we cannot use the continuous number concept. Digital values are based in binary and therefore need to be discrete numbers. Discretizing values means setting a limit on how often we will "look" at our continuous analog signal, the sampling rate is simply how often we will check the signal. The output file name is used in the future to have a reference to all our filtered values in a .txt file in the server side. The coefficient file name is a file that already exists, and coefficients are used when convoluting two sets of numbers. The purpose of convoluting two sets of numbers is to provide a low-pass filter effect to convert a 100Hz signal to a 90Hz signal.

CLIENT SIDE:

```
  printf("Sending sampling rate, filename and coefficients...\n");
  send(comm->datasock, inputS[0], sizeof(inputS[0]), 0);
  send(comm->datasock, inputS[1], sizeof(inputS[1]), 0);
  send(comm->datasock, "startcoef", sizeof("startcoef"), 0);

  while (!feof(coef)) // we are sending the coeffecients
  {
    fscanf(coef, "%lf", &tempNum);
    sprintf(inputChar, "%lf", tempNum); //convert the double to a string
    send(comm->datasock, inputChar, sizeof(inputChar), 0);
  }
```

SERVER SIDE:

```
if (i == 1)
    {
      samplingRate = atoi(recieveBuffer);
      hX = (double*)malloc(sizeof(double)*(samplingRate + 101));
    }
    if (i == 2)
    {
      strcpy(fileName, recieveBuffer);

    }
    else if (strcmp(recieveBuffer, "startcoef") == 0) // kick off sending the coeffecients
    {
      recievingCoef = true;
    }
    else if (strcmp(recieveBuffer, "endcoef") == 0)
```

```
    {
      recievingCoef = false;
      printf("Done recieving the coefficients\n");
    }
    else if (recievingCoef)
    {
      sscanf(recieveBuffer, "%lf", &hX[co]);
      co++;
    }
```

The parameters are immediately sent to the server after an enter keystroke, and the program then prompts the user to press enter again to start data acquisition. This causes the server to light up an LED. The client-server uses sockets to communicate over the local network. A start signal is sent over the socket to the server, and the server then waits for a switch connected to the DT9816 to be on high or 5V.

Traditionally, one would use polling for checking the switch value (explanation of how values are kept in a buffer are discussed later), but the design for the program uses windows events to mock an interrupt. Once the server is started it creates two threads – one for constantly receiving values from its port, and another to hold off until the event is kicked off. The event happens when the 5V is detected from the channel 1 of the DT9816 as mentioned above.

```
/kick off the checking the recieving buffer thread
  hClientThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)client_iface_thread,
(LPVOID)&profiler, 0, &recieveClientThreadID);

  // set up the event for when the client is ready to begin data acquision
  channelargs[1].channelN = 1;
  channelargs[1].startacq = CreateEvent(NULL, TRUE, FALSE, NULL);
  ch1dataThread = (HANDLE)_beginthreadex(NULL, 0, &ch1switchAq, &channelargs[1], 0, NULL);
……….
else if (strcmp(recieveBuffer, "start") == 0)
    {
      printf("we started\n"); // if the client sends a start signal, we kick off the DT9816
      SetEvent(channelargs[1].startacq);
```

Once the 5V signal is sent in channel 1 the event kicks off and the DT9816 is initialized for AD converting. The first thing that is done is that every even number from the buffer is taken out of the pBuffer. pBuffer is the buffer that is returned by the DT9816 hardware using the API from the SDK. The way pBuffer works is it stores based on the number of channels it has. For example, in this scenario the buffer would look like this : pBuffer[ch0value, ch1value, ch0value]. This is why we need the even numbers from the buffer, these are the values from the sine wave that was mentioned earlier.

```
  for (i = 0; i < samplingRate * 2; i += 2)
          {
            tempVal = pBuffer[i];
```

The pBuffer length is obviously twice the sampling rate because it needs to store two different channels. In order to check the 5 Volt switch mentioned earlier, we simply need to check the last value in the buffer, it will suffice since it is only a constant signal.

After acquiring the buffer from the DT9816 we need to use some binary arithmetic to convert the value into a double from a binary value. Once we have our array/ buffer of all the sampling values of the continuous sine wave we then go into processing mode.

Since we are dealing with real-time we need to grab the previous buffer and take the last 100 values from it and add it to the current buffer. The reason this is done is to remove the transient region and give it a real-time feel. Once the real time buffer is put together, we then convolute the two signals

SERVER SIDE:

```
for (i = 100; i < samplingRate + 100; i++)
  {
    fullBuffer[i] = presentSignal[i - 100];
  }


  if (firstTimech0)
  {
    for (i = 0; i < 100; i++) fullBuffer[i] = 0;
  }
  else
  {

    j = 100;
    for (i = 0; i < 100; i++)
    {

      fullBuffer[i] = previousSignal[samplingRate - j];
      j--;

    }
  }

  conv(fullBuffer);
```

Once the convolution is done we are sure we now have the 90Hz signal we need. These are the values the client needs to compute the min, max and to use the trapezoidal rule to integrate the sine wave.

It is necessary to signal to the client that we are about to send the filtered values, so it will know to store them in a buffer. Since we are doing real time, we also need to send the values right away after getting the convolution. The values are in the double format so we also need to convert them into a string to send them nicely. The client will then take in the values as a string and convert them back to a double to do the min, max and integration.

SERVER SIDE:

```
 send(comm->datasock, "filterrecv", sizeof("filterrecv"), 0);
  for (j = 0; j < samplingRate + 201; j++)
  {
    tempNum = tempFilteredData[j];
    sprintf(inputChar, "%lf", tempNum);
    send(comm->datasock, inputChar, sizeof(inputChar), 0);
  }
```

CLIENT SIDE:

```
// get each of the values and convert them to double while we are it
      sscanf(recieveDouble, "%lf", &tempFilteredValues[f]);

      f++;
...
// we are done with the data so we recieved at -1 so lets save the numbers and computer the
necessary things
      if (strcmp(recieveDouble, "-1") == 0)
      {
        maxvalues.open("maxvalues.txt", std::ios::app);
        minvalues.open("minvalues.txt", std::ios::app);
        areavalues.open("areavalues.txt", std::ios::app);

        printf("\n");

        max = getMax();
        min = getMin();
        area = integrate();

        printf("Max: %lf, Min: %lf Area: %lf \n", max, min, area);

        maxvalues << max;
        maxvalues << "\n";

        minvalues << min;
        minvalues << "\n";

        areavalues << area;
        areavalues << "\n";

        recievingFilter = false;

        maxvalues.close();
        minvalues.close();
        areavalues.close();

        f = 0;
```

After the values are received into the tempFilteredvalues buffer, the 3 functions are used to compute the max, min and integral. Max and min are pretty self explanatory to calculate, but the trapezoidal rule requires a bit more computational power. The integrate function is shown below:

```
double integrate()
{
  int i;
  int aa = (samplingRate) / 4;
  double area = tempFilteredValues[0] / 2;

  for (i = 1; i < aa - 1; i++) area += tempFilteredValues[i];

  area += tempFilteredValues[aa];
  area = area / ((samplingRate) * 2 * 3.14159265359);
  return area;
}
```

Once we are done computing the 3 functions, we then print the values every second to the user. We also want to save the values into a file. The program will stop data acquisition and computation once the switch is no longer 5V on the server side.

Discussion/Conclusion:

Overall the system worked pretty smoothly, except for the hardware. The sampling values from the DT9816 were getting an intermediate buffer with garbage values, and it would occur every fourth of second. The values were printed out raw into the console and it was obvious there was something weird with the hardware. To solve this problem, it would have been ideal to change up the DT9816 when performing testing and demoing. Points were lost from this error, but it provided a better lesson to not rely on one piece of hardware, and attempt testing on different circuits. Because of the random garbage values, the plotted values were thrown off and even though the filtering worked it created a weird graph.

The client-server communication worked flawlessly under the same host, although multiple hosts were not tested with varying IP addresses. The LED's also worked fine so the digital output code and hardware had no error.

The buffer was controlled by using a lot of memory allocation, and that was the main bulk of the problems when debugging. After some time, and better placement of the memory allocation, the program ran flawlessly in respect to buffer management.

Another bulk of the problems encountered in this project were from so many configurations in visual studio due to the various libraries needed, and switching back and forth from v100 and v140 of visual studio didn't make the problem any better. Although it was advised to use preexisting configurations from a source file provided by the professor, DTconsole.cpp, I decided to start my program from scratch. It was a very painful experience at first, but I think it was a great way to learn to do the correct configurations in future embedded projects.

Some improvement considerations for making this program more efficient include using threads for processing the convolution, min, max and integration. To take advantage of all the cores and computing power of the PC, this would have been the more performance efficient solution. Due to time constraints, threads were only used for checking channel1. There are a lot of consequences that can occur from adding a lot of threads in the program, such as race conditions, and there was no time for ensuring these things would not happen.

Flow Chart:

Please see flowchart file in directory. Too big to place in here.

Block Diagram: