ECE-4220 Logic Analyzer Report

Dallas Fletchall

5/14/2014

**Objectives and Lab Description**

The overall objective of a Logic Analyzer is to determine whether a signal is logic high, low, or somewhere between. This implementation allows for four channels to be sampled at the same time. Another objective was to demonstrate graphical display of the results. This allows the user to quickly determine signal values for a longer range of time then would be able with data in tabular format. Ease of use was further facilitated by allowing the user to enter names for the individual channels being monitored. Finally, the voltage levels considered logic level low and logic level high, needed to be adjustable. This would allow the Logic Analyzer to be used in more situations than if the levels were constant.

**Implementation**

**Hardware Implementation**

The hardware required in this implementation is divided into three separate categories. These categories are signal conditioning, data collection server, and display client. Each category is explained individually in this section of this report.

***Signal Conditioning***

******

Figure 1: Comparator Configuration of Operational Amplifiers

To implement the Logic Analyzer, the input signals were sent to the data collection server. The requirements of these signals were direct reflection of the raspberry pi, which was chosen for the data collection server. Input signals were provided to the raspberry pi via the general purpose input output (GPIO) pins. The GPIO pins have a max input voltage rating of 3.3Vdc, while the implementation allows for any input voltage 0-5Vdc. This is accomplished by using two LM324n single supply voltage, operational amplifiers in a comparator configuration as seen in Figure 1. The user was allowed to input the maximum voltage level that could be considered logic level low, and the minimum voltage level that could be considered logic level high. This voltage was compared to the input from the channels being tested. If the inputs were less than the compared voltage, the output of the comparator was 0Vdc. Otherwise the output was 5Vdc. This voltage was then sent through a voltage divider to lower to the 3.3Vdc required by the raspberry pi. This comparator configuration was also useful in providing a high impedance connection to the signal being tested. This reduced the loading effect of the signal, and helped provide accurate results from the input signals.

To produce the comparator voltage, two MCP4901 digital-to-analog converters (DAC) were used. The outputs of the DACs were provided to the inputs of the comparator circuits for each signal being tested. Each signal was compared to the output of both DACs, and sent to the raspberry pi for collection. This required two GPIO pins for each input signal. This allowed the Logic Analyzer to determine whether the signal was logic high, low, or between. Table 1 shows the relationship between the inputs of the GPIO pins and the logic level determination. The need to change the output of the DACs was handled by serial peripheral interface (SPI). Connecting the DACs as slave nodes and the raspberry pi as the master allowed for communications according to the interface. The DACs were then able to be programmed to the desired output. The traditional SPI is full duplex and requires four wires. A clock source, master-out-slave-in (MOSI), master-in-slave-out (MISO), and a chip select. The MCP4901 was designed to only use the MOSI connection, therefore the MISO connection was not used, and increased the number of GPIO pins available for other uses.

|  |  |  |
| --- | --- | --- |
| **Ch Low Output** | **Ch High Output** | **Determined Signal Value** |
| 0 | 0 | Logic Level Low/0 |
| 0 | 1 | Error |
| 1 | 0 | Undetermined/Floating |
| 1 | 1 | Logic Level High/1 |

Table 1: Channel Determination Table

***Server Hardware***

The data collection server chosen was the raspberry pi. The purpose of this server was to collect the data from the input signals, program the DACs over SPI, and pass data to the client over a socket connection. The raspberry pi worked well for this implementation because it had seventeen GPIO pins available on the P1 header. Four of these pins were needed to implement the SPI interface with the two DACs, and each input signal was compared to the output of each DAC. This required eight GPIO pins for data collection. This left five GPIO pins unused and available for further additions to the system. Network connection was handled over an Ethernet connection with a maximum speed of 100 Mbps

***Client Hardware***

The client was the least specific of all the hardware, since it only needed network connection and a display output. To facilitate these requirements, a laptop running VMware player with virtual Ubuntu Desktop was used. The keyboard allowed for user input to the system, while the screen supported the display requirements. The only other requirement of the client was that it needed to support GnuPlot, an open source software program that was used to display a graph of the collected data.

**Software Implementation**

The software needed to support this system was split between the server and the client applications. These applications will be discussed now.

***Server Software***

The purpose of the server is to facilitate data collection, communicate over SPI with the DACs, and communicate with the client for data transfer. To facilitate data collection, the server must first receive the instruction to start recording, accompanied by the duration to record. The final implementation estimated the number of samples that could be taken per second, and used this to determine the total number of samples that would be recorded. The time was taken before collection, and directly after collection. This value was then sent across the network, and sample times were estimated by equally spacing these times over the total collection time. This allowed for the value of the GPIO level register (32 bits), to be the only information sent to the client for display. This was the best implementation with respect to samples per second, and minimized the effects of data bursts.

The communication with the DACs was handled using SPI. This interface is supported by hardware on the raspberry pi, but without the need for a MISO connection, and since the update of the DACs was not crucially time sensitive, this implementation mimicked the interface using GPIO lines. One line was designated the clock, one for data, and each DAC needed its own chip select. The DACs use a shift register for input, and simply shifted the data line into the register at the rising edge of each clock signal. The transfer was required to be sixteen bits long, and the output voltage would not be set until sixteen rising edges of the clock had been recorded. The first four bits were configuration bits for the DAC. Since the configuration never changed in the Logic Analyzer, these were hard coded into the software. The next eight data bits were used by the DAC to set the voltage output level. They were determined by the message sent from the client. Finally, there were 4 don't care bits that were sent with the same value as the last data bit for ease.

***Client Software***

The first software requirement of the client was the network communication. The client was responsible for making requests to the server to set the DACs, and also to start data collection. These messages were sent to the server, then the client waited to receive a reply from the server to see whether the message was valid or not. This message would be forwarded to the user to inform a problem had occurred. The other main responsibility of the client was the display of recorded signals received from the server. This was handled through third party open source software called GnuPlot. GnuPlot is a separate application that receives commands through a named pipe, which allowed the client program to change the configuration of the GnuPlot display. This program also required that the data be recorded to a file in a specific format. The client saved the register values directly to a binary file while receiving data from the server; it then needed to transform the data into a file the GnuPlot could interpret. This could be implemented after all samples had been recorded. After the data was transformed, a message sent over the named pipe was used to inform the GnuPlot program to update the display according to the new file. The client also handled changing the channel name by sending a message over the pipe to the GnuPlot program.

**Experiments and Results**

The first experiment performed was the client communication with the GnuPlot program. This was needed to display the results, and with this display the results of the tests would be easier to determine. The different configuration options used for the GnuPlot application were to close when the controlling application exited, remove the y-axis numbers, setting the x & y-axis titles, setting the window title, and moving the key outside the graph. With these options setup, the next experiment was to determine the format needed by the GnuPlot program to present the data in the correct way. The original display resulted in all signals being layered on top of each other. The solution was to level shift each one of the signals by the channel number minus one. This displayed the data with each channel separated from the previous, but introduced two errors. First, the numbers on the y-axis were confusing. The first channel was between zero and one, but each subsequent channel was shifted up for viewing. This left channels with values of one to two, two to three, and and three to four. To fix this problem, the removal of y-axis numbering allowed the user to view the individual channel without seeing the actual value on the plot. The next problem resulted from the signals being the same value. For example, if the channel one signal was high, and the channel 2 signal was low, their values would both equal one. This made the graph extremely difficult to read. To solve this problem, the valid options for a channel was 0.95 for high, 0.05 for low, and 0.50 for floating/undetermined. With the y-axis values removed, this gave enough of a gap to delineate each individual signal. These channels were then level shifted, and the results are shown in Figure 2. The other requirement that used the GnuPlot program was the changing of channel names. The user could select the option to change the names in the client program, and then the newly chosen names were sent over the named pipe to the GnuPlot application. For the application to update the channel names, the replot command was sent, which not only relabeled the names, but also replotted the data. This resulted in a very slow update for the channel names. A solution to this observation was not determined, and is still a part of the finished project. A screenshot of the Logic Analyzer data displayed by the GnuPlot application is shown in Figure 2.

The next portion of the application that was tested was the communication between the server and client. The skeleton of a previous lab was used to setup the socket connection, but some of the properties needed to change. First, the skeleton created connections using datagram or UDP connection. This was not ideal since UDP does not guarantee that data is received by the recipient, and because data is not guaranteed to be in order. A socket stream, or TCP connection, was chosen to improve reliability of the socket connection between server and client. To test this connection a simple echo server application was constructed on the raspberry pi to facilitate the passing of messages. After communication had been established and messages could be sent and received by the client, the transmission of signals was tested with the use of dummy binary files. These files contained random valid register values like the ones that were to be sent over the network connection. The server application was updated to send the thirty-two bit values, mimicking the collection of actual data and transmission over the network. The client application was then updated to store the information in a binary file. To test that the files were identical, a tool called Vimdiff was used to compare the values of the files in binary format. This proved that the information could be sent from the server to the client correctly, maintaining the order of bytes as well.

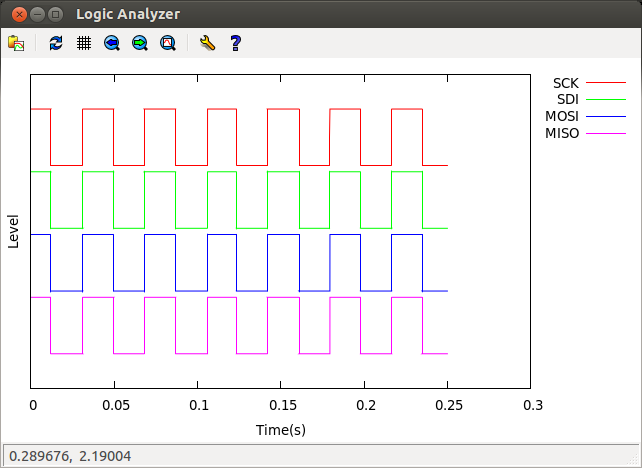


Figure 2: GnuPlot Display of of Arduino Pin Input

With the transmission of data complete, and the display operational, the next portion of the program that needed to be tested was the programming of the DACs. Without the DAC voltages input to the comparator circuits, valid data could not be stored. To test the software, a message format for the low DAC and a message format for the high DAC were created. This message included which DAC to adjust, and the value that the user wanted as output. All error checking was left up to the server. The reason for this is that the DACs and the operational amplifiers were based off of the input voltage. In this implementation that was 5Vdc, but for another implementation, 3.3Vdc or even 12Vdc might be necessary. With the server in charge of determining valid output voltages, the Logic Analyzer could be changed to either of these voltages by recompiling the server application with these values in mind. To test that the messages were being received correctly the echo server was again used. After the message was received in the correct format, the software to update the DAC was implemented. This required sending the sixteen bit sequence across the SPI bus, and verifying with a voltmeter that the output voltage was correct.

With all other pieces verified working correctly, the final piece of the software was to collect data from the comparators and actually send this data instead of the preconfigured binary file. The Discussion portion of this report covers the reason for data collection method, but ultimately the GPIO level register was sent over the socket in entirety to improve the number of samples per second. The value was simply sent across the socket, and received by the client. To test this portion of the application, each signal was placed in a logic state by itself, and the results were inspected using the GnuPlot display. Once all signals were tested individually as logic high, then low, a voltage divider was used to verify that intermediate voltages were also able to be distinguished.

**Discussion**

The discussion portion of this project contains all of the problems that were encountered using this implementation, and the efforts to reduce the effects of these problems. The most important problem encountered was the number of samples per second that could be recorded and sent over the socket. Then the problem encountered while using the wiringPi library is considered. Next were problems with the GPIO selection that ultimately required a software implementation of the SPI. Next, the accuracy of the DAC output is considered. Finally, the downside of using the raspberry pi as a server is discussed.

The most important factor of the Logic Analyzer is correctness. If the results are not correct then there are no uses for this system. The second most important factor is the sampling speed of the system. This will determine the maximum signal speed that the system will be able to analyze. The first restriction on this implementation is the speed of the network connection. This implementation was theoretically limited by a 100Mbps of the raspberry pi. This would limit the number of four byte samples to 3.125M samples/sec (In reality, this is not reasonable because the Transfer Control Protocol contains bits as well). In reality, this implementation was not even close to that fast. Averaging the time taken to send a million samples showed that the transfer rate was closer to 333k samples/sec. This is the fastest transfer that my application could achieve.

The first implementation that was considered utilized the wiringPi library to determine the value of the pins. This was ultimately too slow. The API call used multiple conditional statements to determine board revision and pin numbering scheme before returning the value of a single pin. After calling this function eight times, and then shifting the values for each call, the fastest sampling achieved was approximately 200k samples/second. To improve upon this sampling rate, I decided to read the GPIO level register in its entirety, saving the actual computation until all samples were recorded. This increased the sample rate to almost 300k samples/sec. Finally, I removed the time calculation from each sample. Estimating the number of samples taken per second allowed a rough estimate of the number of samples to take. Then sending the total time for all the samples allowed the client to determine the average time between samples. This could be used to format the data to the specifications of the GnuPlot program. This decreased the number of bytes sent across the socket from twelve to four. This led to the 333k sample rate that the final implementation used. Note that this is not completely accurate since the time of each sample is not recorded. My implementation simply takes 333k samples/sec and distributes them equally over the amount of time recording was requested.

Another implementation considered was to store the data in a file and then transmit the entire file using the sendfile() Linux function. The manual page for this function said that this was the fastest way to send files since it was implemented in a kernel module. Although the transfer may have been faster, this implementation was flawed. The assumption that writing to a file would take less time was incorrect. As seen in Figure 4, the time taken for 300k samples was closer to 4 seconds.

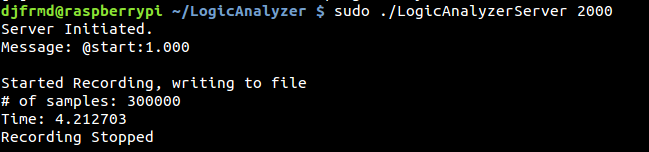


Figure 3: Screenshot, writing register value to a file

Ultimately the decision to send each individual register value over the socket was determined the fastest, most reliable implementation available with the hardware chosen for the Logic Analyzer.

As mentioned above, the wiringPi library was inefficient in sampling that GPIO pins. The library abstracts from the user the real implementation of the raspberry pi. Looking at the documentation of the wiringPi library shows that the pin numbering scheme adopted by the library is not the same as the raspberry pi hardware. The raspberry pi hardware extends some of the BCM-2835 GPIO pins to the P1 header. These are the available pins for user applications. Table 2 shows the wiringPi pin labeling scheme compared to the actual pin number of the BCM-2835. Pins used for this implementation are bold and italicized. The wiringPi library labels these pins starting at zero up to sixteen. This led to problems with reading the GPIO level register, because the pin labels did not match the bit numbers in the register. The ultimate goal was to mask off the eight bits needed to determine the signal levels, and then record this value. The actual pins available from the BCM-2835 were not consecutive. There are two sets of consecutive pins (8-11 & 22-25). This made determining the pin values much more time consuming. All the pins could be masked simultaneously, but the shifting had to be done separately, and then the numbers had to be combined. In an attempt to increase the sampling speed, this was implemented at the time of the sample, but ultimately was not best solution.

|  |  |  |  |
| --- | --- | --- | --- |
| P1 Header Pin | Alternative Function | wiringPi Pin | BCM-2835 Pin (Corresponds to the bit number in GPIO level Register) |
| 3 | SDA | 8 | ***2*** |
| 5 | SCL | 9 | ***3*** |
| 7 | GPIO | 7 | ***4*** |
| 26 | CE1 | 11 | 7 |
| 24 | CE0 | 10 | ***8*** |
| 21 | MISO | 13 | ***9*** |
| 19 | MOSI | 12 | ***10*** |
| 23 | SCLK | 14 | ***11*** |
| 8 | UART TX | 15 | 14 |
| 10 | UART RX | 16 | 15 |
| 11 | GPIO | 0 | ***17*** |
| 12 | GPIO | 1 | 18 |
| 15 | GPIO | 3 | ***22*** |
| 16 | GPIO | 4 | ***23*** |
| 18 | GPIO | 5 | ***24*** |
| 22 | GPIO | 6 | ***25*** |
| 13 | GPIO | 2 | 27 |

Table 2: P1 Header Wiring Schemes

The consecutive GPIO pins were utilized to decrease the number of instructions necessary to determine the signal level for each sample. Since this operation happened for all 333k samples/sec, it was important to maximize the speed of this operation. This led to a problem with the SPI interface. Each of the GPIO pins on the raspberry pi is available for special functions such as SPI, I2C, UART, or PWM. These special functions are also viewable in Table 2. The pins that handle the SPI special function were already being used by the data collection portion of the Logic Analyzer. As mentioned before, the speed of the DAC update is relatively less important due to the number of times this function will be performed in comparison with the data collection. A software implementation that mimicked the SPI hardware was used to program the DACs.

Another note about the Logic Analyzer comes from the DACs 8-bit accuracy. This makes each step in voltage approximately 19.5mVdc. This introduces a small inaccuracy into the output voltage. This solution uses truncation for evaluation of the value sent to the DAC. A more accurate approach would be rounding. This would allow a max deviation of 9.25mVdc. Truncation is the standard C approach to division by integer. The overall benefit of rounding vs. truncation was viewed as unnecessary, and the final implementation utilized the truncation approach.

The raspberry pi was chosen as the server hardware for this application. The inexpensive hardware comes with loadable operating systems, to ease in development. Linux system calls are supported, and there is lots of documentation. The raspberry pi also has access to seventeen GPIO pins. This made the raspberry pi a good choice for this implementation. The best results, however, were obtained when the raspberry pi did the least computation. In the final implementation the client software handled all of the data transformation, and the displaying of the data using the GnuPlot software. Although the raspberry pi can handle all of the operations, it is much slower in comparison when using the GnuPlot software. It is also slower in transforming the data into a GnuPlot compatible file. Originally the design used the raspberry pi to format all data, and then the data was sent over the socket to the client and displayed. Average wait times were in minutes, for each second of data collected. After moving all of this implementation over to the client, wait times still happen, but are much smaller. The final implementation places more of the computational burden on the client since most personal computers are faster than a raspberry pi.

**Conclusion**

This project was successful in analyzing 4 signals, but also has many downsides. First, signals cannot be analyzed if they are changing faster than 150kHz. Notice that Nyquist's sampling theorem is valid here since this implementation tries to recreate the time domain signal. For example, assume a frequency equal to the max sampling rate, 300kHz, is present on a signal in the Analyzer. Also assume the signal is high for the first sample. The second sample will take place after the signal has gone low, and returned high. The output of the Analyzer will show a continuously high output for the duration of the recording. The only way to guarantee that the sample is not altered is to take a sample for the high and low portion of the signal. Note, however, If only one sample per logic level is captured, the result is a saw tooth wave in the GnuPlot software.

Another downside is the need for a separate power supply. To improve this project a regulated 5Vdc power supply on the board would be useful. This would allow for a portable solution. Finally, a real-time operating system would need to be used if the time of the signal change was to be accounted for. In this implementation, the time a sample was taken was speculative. The average of 333k samples/sec was based off the total time taken to record and send one million samples. The samples were then evenly spaced to display up to the time specified. A real time implementation would be more accurate.