4-15-2024

Development of MSFC ET-30s Next Generation of the Structural Load Test Measurement Acquisition System using LabVIEW

Mary Claire Marguerite Corell
University of Alabama in Huntsville

Development of MSFC ET-30's Next Generation of the Structural Load Test Measurement Acquisition System using LabVIEW

by

Mary Claire Marguerite Corell

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

Date

Honors Capstone Project Director: Mr. Phillip Hood

Mary Claire Corell 4/15/2024

Student (signature) Date

Phillip G Hood 4/15/2024

Project Director (signature) Date

04/20/2024

Department Chair (signature) Date

Sean Lane Digitally signed by Sean M Lane
Date: 2024.04.25 17:40:51
-05'00'

Honors College Dean (signature) Date

Honors College

Frank Franz Hall

+1 (256) 824-6450 (voice)

+1 (256) 824-7339 (fax)

honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted with the final manuscript.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_Mary Claire Marguerite Corell_____

Student Name (printed)

Student Signature

___ 4/15/2024_____

Date

Table of Contents

Dedication

To my mentor, who saw my potential and gave me the incredible opportunity to work on this project and hone my technical skills.

To my husband and parents, who have encouraged me to reach for the stars and never stopped supporting me along the way.

Abstract

The Structural Load Test Measurement and Acquisition System (SLTMAS) is software created by NASA Marshall Space Flight Center's (MSFC) Structural Strength Test Branch to collect and process voltage data correlating to strain, temperature, and load during static structural tests. The data collected by the SLTMAS system is used to visualize data that will be relevant for analysis for both development tests and qualification tests. Qualification tests will take test articles to limits that qualify an article for flight by applying loads that include expected flight loads and their safety factors. Previously, the SLTMAS program was run on SCXI chassis with a Windows operating system. The SCXI hardware has reached end of life, and this project encompasses a portion of the development effort to transition to new PXIe hardware utilizing LabVIEW on the NI Linux Real-Time Platform. Additional concepts include multithreading and networking. This paper will explore the process of creating a prototype that will meet the requirements given to me by my mentor.

Introduction

Since the SCXI hardware has entered end of life, this project will provide detail on exploring the transition to using PXIe hardware with NI Linux Real-Time. The operating system currently on the SCXI chassis is Windows. However, since the original program on the SCXI was written, NI has released NI Linux Real-Time (Introduction to NI Linux Real-Time 2023). NI Linux Real-Time is not compatible with SCXI chassis, so we would like to explore development on the PXIe chassis. The sampling rate for the PXIe program will reach as high 25,600 Hz compared to the highest sampling rate used for the SCXI program being 1000 Hz. Thus, it would be useful to inspect if the real-time performance of NI Linux Real-Time with the increased sampling rate of the PXIe chassis would be worthwhile to switch to. There will be multiple requirements set by my mentor that I will have to follow to prove that it could replace the current system that will be discussed in the requirements section. In this paper, I will highlight the steps I took and the knowledge I used to create a proof of concept for the PXIe program that runs on NI Linux Real-Time and what lessons I have learned in the process.

Requirements

The purpose of this project is to create the LabVIEW code that allows the PXIe chassis to operate as a part of the SLTMAS system. The SLTMAS operator will load a database with information about how the sensors being used for that test are set up, start the SCXI chassis with the local program written for Windows, and then monitor the system while the load is applied to the test article. This functionality will need to be replicated for the new system. The requirements for this project are listed below:

1. The program must run on the National Instruments Linux Real-Time (NILVRT) Operating System.
2. The program must take data from at least two different sensors.
3. The program must be able to take sensor data at a maximum of 25,600 Hz.
4. The program must be able to send properly formatted ZMQ messages that will be received by a ZMQ broker.
5. The program must be able to receive and parse ZMQ messages from a ZMQ broker.
6. The program must have multiple states: setup, standby, data acquire, shutdown.
7. The program must be able to acquire, store, and send data at variable rates given via ZMQ messages.
8. During the data acquire state, the program must be able to receive a trigger signal that will empty
9. Once the program has connected to the ZMQ broker, the program must send heartbeat messages to confirm it is connected.

Process

Before I could begin work, I needed to figure out what tools I needed and ensure I had them ready. First of all, I needed a PXIe chassis with cards to support taking strain, CLTS, and thermocouple measurements. I also needed to ensure that NI Linux Real-Time was the system's operating system and that I could connect it to the internet and use the package manager to download necessary libraries. After I received a PXIe chassis, I installed an 8-channel strain module, a 20-channel cryogenic linear

temperature sensor (CLTS) module, and a 32-channel thermocouple module into the PXIe chassis. Normally, there would be more modules hooked up to the PXIe during a real test, but for my purposes, I started small.

Installing NI Linux Real-Time

Once I installed everything, it was time to change the operating system from Windows to NI Linux Real-Time. One of my coworkers gave me a USB with NI Linux Real-Time to boot from. However, when I went to boot from USB, there was no option for what I wanted to do in the BIOS setup. I looked up what PXIe chassis are compatible with NI Linux Real-Time, and found that the one I had was not compatible. I learned my first lesson of this project: always check to make sure what you have is compatible with what you are trying to do. So, I grabbed a chassis that I checked was compatible with NI Linux Real-Time and installed the modules onto that one instead. With the new chassis, I was able to boot from the USB with the right operating system and was able to get started writing LabVIEW. I used OPKG to download the ZMQ library, and then I was off to coding.

Starting Small

To familiarize myself with LabVIEW, I started with the simple task of taking strain from one channel before moving on to more complex concepts. Instrumentation like strain gauges were new to me, but luckily with background knowledge from my circuits class, I could understand them in no time. In LabVIEW, DAQmx is one of the ways we can control our devices. There were settings for setting up DAQmx for strain gauges that I did not know how to set. Were we using an external or internal voltage source?

What type of bridge configuration do the strain gauges have? What are the values for the lead wire resistance, and voltage excitation? I asked my coworkers and found that we used an internal voltage source and strain gauges with a quarter bridge configuration. However, the rest of the values were variable between strain gauges. We use a database that keeps up with the different values for the different gauges we use. So, for my initial testing, I hardcoded the values that were variable. I set the DAQmx Create Virtual Channel node to take strain gauge from one channel. Once I accomplished this, it was time to move on to taking strain from multiple channels.

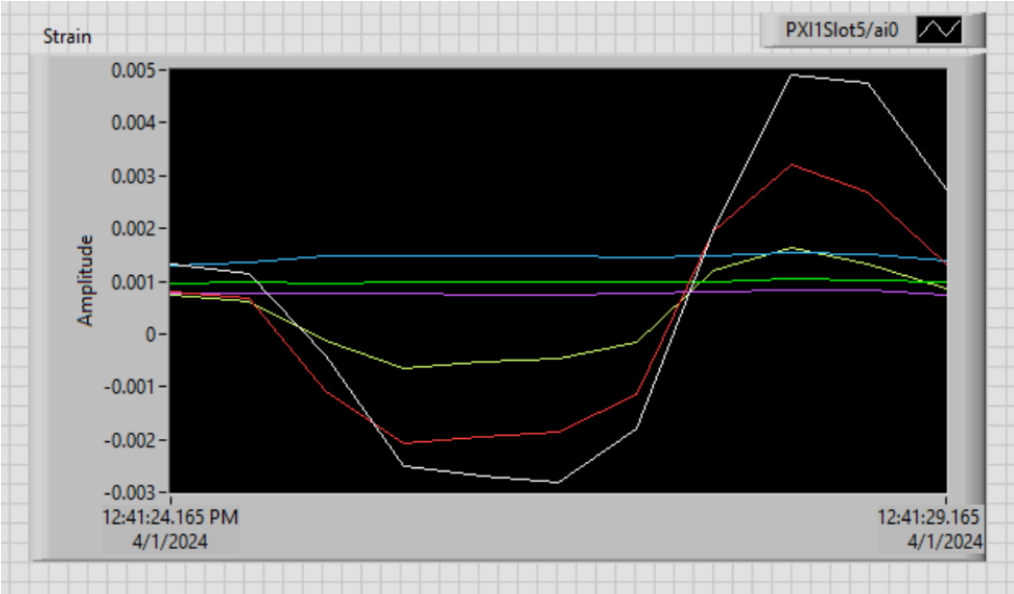
Multiple Channels

For multiple channels, I had to create a loop that would add the new virtual channel to the previous list of virtual channels. Once I was able to do this, I felt it was time to take the variability of values into account. One of the issues I ran into was VI's that were supported on Windows but not supported on Linux Real-Time. One of these toolkits is the Database Connectivity toolkit. So, in the meantime, I created a CSV file to parse through and pull values from. Once I got the values out, I needed to figure out how to have the DAQmx Create Virtual Channel VI read them in. The data types of the variables wired in were important, and not readily apparent to me then. To start, I created a cluster with all of the setting values and wired it into the node. I found that I could not simply wire a string input into the node, but that the value the node wanted was an integer. I eventually found the corresponding codes by reading NI's documentation. The code for "Internal" was 10200 and the code for "Quarter Bridge II" was 10272. With the settings wired correctly, I could take data from the strain gauges. Once I had an understanding of taking data from strain, it was fairly easy to add in

thermocouples and CLTSs, though I only had one CLTS to work with. I added a column to the CSV file that would indicate what type of sensor it was and then added a case structure that would choose what type of channel to create based on the input. The program would then loop, and each time add the sensor's virtual channel to the pre-existing list of virtual channels. For the CLTS, I set the channel to read resistance, since CLTS is not pre-set in LabVIEW. From this resistance, we would be able to calculate the temperature ourselves. Despite having only one CLTS, I ensured my program would be scalable to add more.

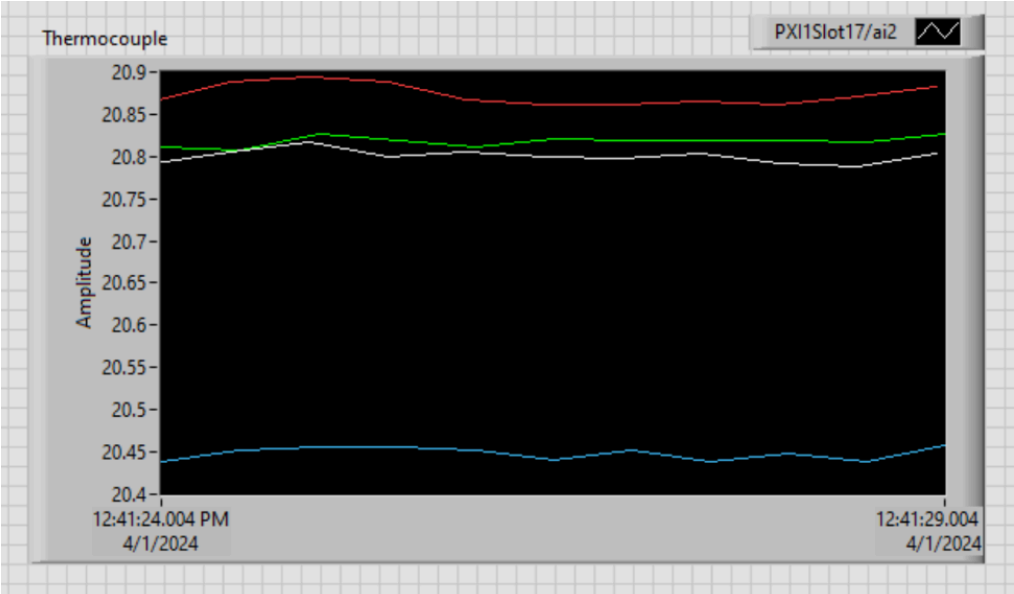
A strain sensor comprises a metal grid pattern on conductive foil. Two leads are attached to the gauge, and a current is sent through. When a gauge experiences compressing or stretching, the gauge changes in resistance. The measured change in resistance can be used to calculate the strain. See Figure 1 for an example of the output of six channels of strain. For the strain example, I bent a pliable metal plate with two three-wire strain gauges up and down. The data, which is in microstrain, dips below zero, then dips above zero as I bent the metal plate in the opposite direction.

Figure 1. Six Channels of Strain Data



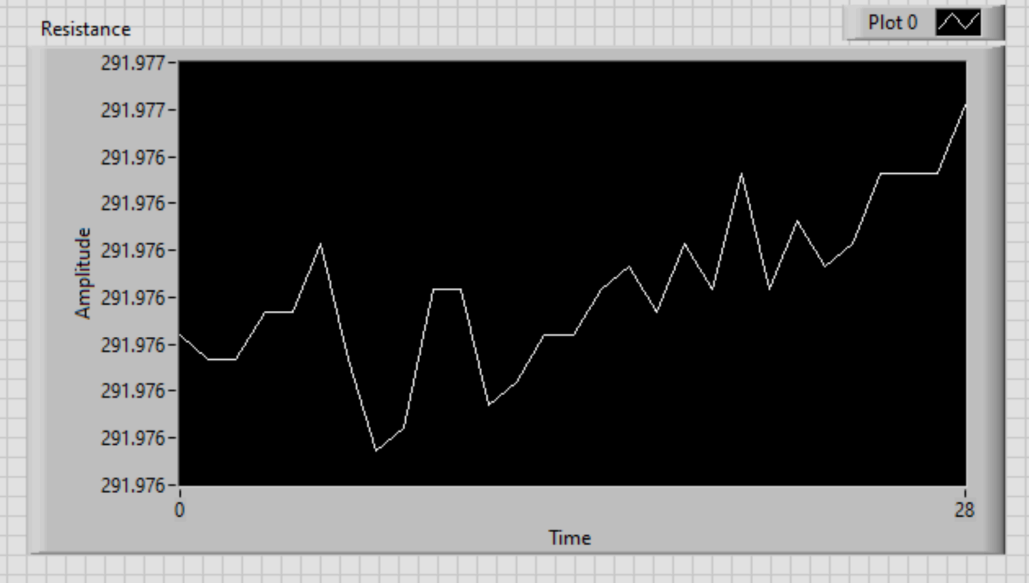
A thermocouple is a sensor composed of two wires of dissimilar metals. The two wires are bound together to form a junction. When this junction is heated or cooled, a voltage is induced which can be used to calculate the temperature. See Figure 2 for an example output of a 4-channel Thermocouple reading. The thermocouple read the room to be about 20° Celsius, which is because the thermocouples were placed closer to the chassis area where fans cool the equipment off.

Figure 2. Four Channels of Thermocouple Data



A CLTS sensor is primarily used to measure temperatures below -150°. It is similar to strain in that it measures resistance. The CLTS's resistance was about 192 Ohms, which when I used our equation to convert to Celsius yielded 77.52° Fahrenheit or 25.29° Celsius. The difference of 5° degrees Celsius can be attributed to the CLTS not being placed close to the fans like the thermocouples were.

Figure 3. One Channel of CLTS Data



Writing to Binary Files

After collecting the data, it must be recorded to a binary file. However, when I wired an array of data to the Write to Binary File VI, my file was larger than I expected. I went through and converted the values back to what I expected them to be, and the data matched what was being displayed by the program, but there were still extra values. Eventually, I realized that the extra values indicated the size of the array of sensor data. To deal with this, instead of building an array, I converted the values to strings and concatenated them together for each loop iteration. Once all of the values were concatenated, when I translated the binary file into a readable format, I did not get any extra bytes, and everything read like it was supposed to.

Using ZMQ

Now that I had completed data acquisition and data logging, it was time for the real challenge: integrating ZMQ into LabVIEW. Using shared object libraries on NI Linux Real-Time presented more challenges than on Windows, especially since I had previously been more accustomed to working with Windows machines.

The documentation I found online for accessing shared object libraries would work for Windows systems, not NI Linux Real-Time. On Windows, LabVIEW will automatically create functions based on the header file that accompanies the .dll file. However, it does not do this for Linux .so files. So, I had to manually create all of the functions which means I also had to interpret data types from C into LabVIEW data types. For most of the values, it was fairly straightforward. I had trouble figuring out what datatype a void pointer should be. Looking at NI's documentation, I eventually settled on a 32-byte integer (Supported Data Types for the Import Shared Library Wizard 2023). So, I initially set up the program to send data. I used zmq_context to create a context and zmq_socket to create a socket. Then, I fed the socket into zmq_connect which would connect to the IP address of the broker. The zmq_connect function was set to publish. Then, I connected the socket to zmq_send to begin sending data.

One of the biggest mistakes I made during this time was using the wrong function with Call Library Function Node. I mixed up zmq_msg_send and zmq_send. I had been using zmq_send, but using the function parameters for zmq_msg_send instead. With zmq_msg_send, it sends a message of zmq_msg_t type, and it just needs the socket and the flag. For zmq_send, it is given a pointer to a buffer variable and the length of the message the buffer is pointing to. It also requires the socket and flag argument

(ZMQ API reference). The program was not sending any messages to the existing ZMQ broker. I was curious if that flag I had set was incorrect. So, I adjusted what I thought was the flag argument in zmq_msg_send and set it to five. Suddenly, it was sending five bytes of data. I thought that was strange, so I set it to ten. Once again, it was sending ten bytes of data. At first, I thought that the function itself was messed up and that the flag argument was misnamed. However, I realized that I had been looking at the wrong function for parameters. Once I figured out that I was using the wrong function parameters, I made sure to use the correct function prototypes, and everything started to work as intended. The broker began to receive messages with the data.

When I finally got the program to properly send data, it seemed that all I had to do was create a different socket for subscribing, and setting the zmq_connect flag to subscribe. This was not the case, as one of the parameters in the zmq_recv function is a void pointer, which contains the message. I ran into some issues with this and encountered memory issues within LabVIEW. The program started to crash frequently and it was not receiving any messages. Eventually, I found that LabVIEW has built-in manager functions that assist with allocating and deallocating pointers. I used LabVIEW's DSNewPtr function to create a pointer to feed into the zmq_recv function and then copied the pointer's value into a string. From there, I could parse the message to find out the type of message and the data in the message.

Implementing State Machines

Once I got the program to send/receive messages and acquire data, it was time to create the various states. After reviewing NI's documentation for state machines, I used case structures (Application Design Patterns: State Machines 2023). The first

state created the socket and context to connect the subscriber and publisher. It was also here that I initialized the queues that would share data between states that were running at the same time. The next state receives the CSV files necessary to set up the virtual channels. Once the channels are set up, there will be three states going forward: the heartbeat state, the command receive state, and the data acquiring state. In the heartbeat state, the program will use zmq_send to send status messages to the broker. The command receive state will receive commands, parse them, and then add what command needs to be executed to a queue that will tell the data-acquiring state what to do. The data acquire loop uses a case structure to determine what to do based on what comes through the queue. If no command has been received, the data-acquiring state will stay idle. Once a start message has been received through the queue, the program will use the virtual channels that have been set up to take data. The data is saved into two different queues: the storage queue and the sending queue. If an exit command has been received, all states will stop execution and move to a shutdown state, where pointers will be de-allocated, zmq_close will destroy the socket, and zmq_ctx_destroy will destroy the context.

Once these states were working, I added variable rates for acquiring and sending. The program receives these rates in Hz from the broker, converts them to milliseconds, and uses the Wait (ms) function to ensure the loops follow the data rate given. If 100 Hz is given, the program will convert it to 10 ms, and then the loop will wait 10 ms, execute the loop, wait 10 ms, execute the loop, and so on. The data will be acquired faster than is sent or stored. For example, the program can take data and store data in a buffer at 25,000 Hz but only send data every 100 Hz. This means that if

an event occurs, the user can send a trigger command, which will dump the buffer into a file to be analyzed later.

Finalizing My Code

It was at this point, where everything was working together, that I decided it was time to make passes through my code to make it better. I was able to meet with a LabVIEW expert who guided me through LabVIEW standards of coding and how to refactor some of my code to increase its readability. For instance, everything in your VI should generally fit on one screen without scrolling. Before this, I had to scroll in both the X and Y directions in order to look at all of my code. To get your code to fit in one screen and be tidy, it is important to create sub VI's of tasks that can be repeated by multiple different portions of the code. I had sub VI's for acquiring data since the program would be acquiring multiple different types of sensors at different times so the acquiring code could easily be reused. I also split the code into VI's based on state as well.

So, I started going through my code and splitting items into subVI's. I color-coded the subVI's based on their function. Helper functions like converting timestamps into the format we were using were yellow. SubVI's that were for ZMQ were green. SubVI's for general states were blue. I cleaned my wires so the eyes could easily follow the program's execution. This also greatly helped with debugging, since everything was partitioned, it was a lot easier to find the error. For example, I could check which subVI's finished their execution and which didn't to find what was causing a program to crash. For each subVI I created, I would also add a description of each input and output so that future users would have a basic understanding of how to use them. I also ensured that my code had

plenty of comments to indicate why I was doing what I was doing. Finally, I had code that not only fit the requirements but was much easier to read and understand.

Conclusion

Before this project, I had little experience in combined hardware and software systems. However, once I got acquainted with LabVIEW's DAQmx functions and the types of sensors we used, implementing simple code to take data became a breeze. The hard part, I soon found out, was using the Call Function Library Node in LabVIEW to call on the ZMQ shared object library file on the PXIe machine. Once I figured out how to call ZMQ functions, it was finally time to combine all of the pieces. I was able to create a prototype that fit all of the requirements given to me. As we transition to NI Linux Real-Time, future development will be able to continue from where this project has ended.

References

Application design patterns: state machines. (2023, November 6). NI. https://www.ni.com/en/support/documentation/supplemental/16/simple-state-machine-template-documentation.html

Introduction to NI Linux Real-Time. (2023, June 1). NI. https://www.ni.com/en/shop/linux/introduction-to-ni-linux-real-time.html#section-2079855842

Supported Data Types for the Import Shared Library Wizard - NI. (2023, July, 17). https://www.ni.com/docs/en-US/bundle/labview/page/supported-data-types-for-the-import-shared-library-wizard.html

ZMQ API reference. (n.d.). https://libzmq.readthedocs.io/en/latest/