Switchgear System Control Interface

Michael Dean, Huy Tran, and Christopher Miller

Dept. of Electrical and Computer Engineering
University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — The idea of this project is to create a device that will allow technicians to safely de-energize and re-energize the equipment in a switchgear. If a fault occurs when a technician is near the switchgear, then he/she can be seriously injured. Our device will allow the technician to be at a safe distance from the switchgear so even if a fault occurs they will be in no danger. We have designed two devices that ensure this will happen, a small hand-held device and a software-based web application that will do the same job as the handheld device but on a larger scale.**

***Index Terms* — controller, fault, microcontroller, power, safety, switchgear**

I. Introduction

Switchgears are used to control and protect electrical equipment. They are also important for technicians that need to work on the equipment inside of the switchgear because they can use the built-in circuit breakers to de-energize the equipment. It is very possible that during the process of re-energizing the equipment in the switchgear, something can go wrong, and a fault occurs. Depending on the type of equipment in the switchgear, the fault can have catastrophic consequences and can badly hurt anybody near it.

We were tasked with the job of creating a device that will make it safer to re-energize the equipment. Inside the switchgears that our sponsor, ABB, makes there are circuit breakers that sit on a racking system. This racking system moves the breakers back and forth from the disconnected state to an intermediary test state, and then finally to the connected state where the equipment can then be powered back up. The intermediary test state is used to open and close the breakers so that the technician can make sure that the breakers are fully operational. The actions that move the breaker between these states are 'Rack In', 'Rack Out', 'Open', and 'Close'. These states and the actions that move the breaker between the states are the focus of our project.

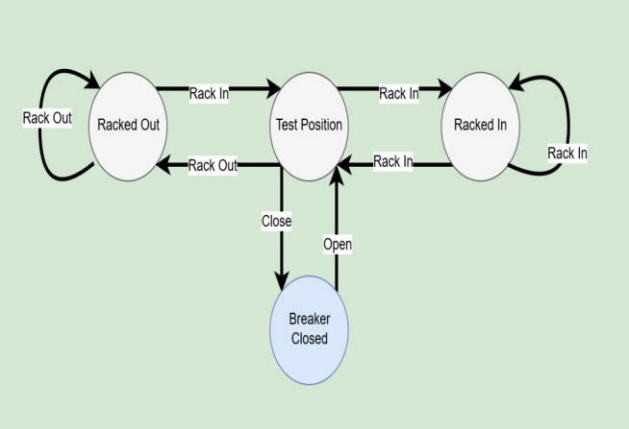

Figure 1: Simple state graph to illustrate the racking system

The racking system that the breaker sits on top of already has the circuitry that allows the breaker to switch between these states, so our controller just needs to send the system the necessary signals. The racking system drives the circuit breakers forward and backward using a DC motor that has an operating voltage of 120V and an operating current of .33A.

There is a special case during the operation of the racking system where the breakers can become stuck on some object that is its path. If it becomes stuck, the racking system is not smart enough to automatically stop and so it will continue to try and move forward. This causes the current to the motor to spike up to 1.8A, and, since the breaker is still trying to move forward while stuck, it can become damaged.

Our system consists of five parts: a small hand-held controller, a software-based web application, a Raspberry Pi computer, a control board, and a racking system simulator.

Our specifications for the hand-held controller were: it must be able to be used from at least fifty feet away, it will be connected to the racking system using a standard RJ45 cable, and it contains all the buttons needed to send the desired commands to the racking system.

The software-based web application requirements were that one operator must be able to see the state of many racking systems at the same time and be able to issue commands to them from a single computer. We decided to do this by including the third part of our design, a Raspberry Pi, so that it could run a REST API that would transform HTTP requests from a web or desktop application into an electrical signal that can be sent to the racking system.

The fourth piece of our design is the control board. This is a separate PCB that has a MSP430 microcontroller

on it that receives signals from both the Raspberry Pi and the hand-held controller, interprets them, and sends the racking system the signals it needs to move the breaker. The control board also contains a circuit that can be used to determine when the racking system is stuck and, in that case, will allow the technician to either cut the power to the motor or send a signal that will reverse the circuit breaker away from the blockage where it got stuck.

ABB informed us that our task is not to make sure that our design physically works with their current racking system. They just requested that we design a system that will send the correct signals to the correct pins and it is their job to take that design and integrate it into their racking systems. To test our design and to show that it works, we created a racking system simulator that reacts to the input signals just like a real ABB racking system would.

II. Overall System Design

The system design required that it could be powered from an internal voltage source inside of the switchgear. The control board sits inside of the low voltage compartment of the switchgear and is powered from that internal voltage source. We did this because we did not want a technician to have to switch out a battery from inside the switchgear every few months or years. The hand-held controller and Raspberry Pi are also powered from the internal switchgear voltage source; therefore, the entire design does not require any batteries. In other words, the hand-held controller does not need to have any power if it is not connected to anything.

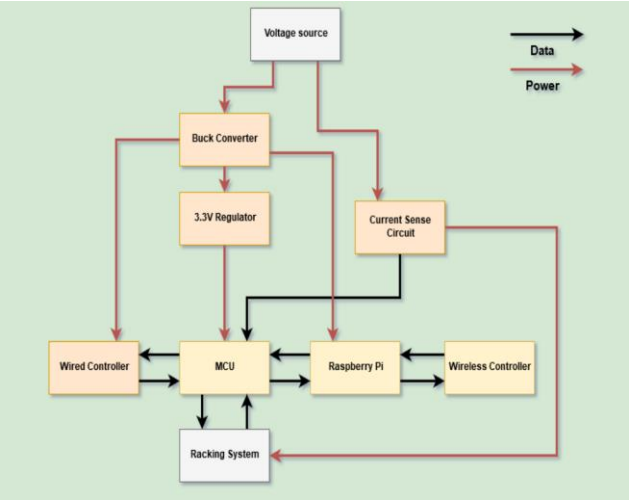

Figure 2: Overall block diagram for the entire system

There are two voltage regulators that are very important to our design. The first one is the buck converter that ensures there is a steady 5V and up to 5A that can be supplied to the other components on the board. The MSP430 microcontroller needs a 3.3V source, which is why there is another 3.3V voltage regulator. The current sensing circuit is used to provide a signal to the MSP430 that communicates the message that the racking system is stuck and that an action needs to occur to prevent damage.

There is only one component that directly interfaces with the racking system and that is the MSP430 microcontroller. It has output pins that connect directly to the pins on the racking system to send it instructions. It receives binary-encoded signals from both the Raspberry Pi and the hand-held controller that let it know which actions to take and if those signals are even valid based on how the racking system can move.

As stated earlier, the Raspberry Pi is included in our design because it allows the use of HTTP requests to send signals to the MSP430. A Django REST API runs on the Raspberry Pi, allowing computers on the same network as it to control the racking system, to see the state of the racking system, and to determine which switchgear it belongs to through a web application on a computer. This is necessary since the hand-held controller can only control one racking system at a time. However, with a web application sending HTTP requests it can control and see the status of all the racking systems on the network. We chose to use a REST API over web sockets because we wanted the application to be more flexible; it is easier to change the front end of a web application based off the response of a HTTP request than it is over web sockets.

III. Design Components

A. *Racking System Simulator*

Since ABB did not provide us with their racking system we had to simulate one to properly test our design. There are six pins on the racking system which are used to send commands to it and receive output from it. These pins are as follows:

- Rack In
- Rack Out
- Close Breaker
- Open Breaker
- Cut power to the DC motor
- In motion

To simulate the use of these pins, a simple Python script was written on the Raspberry Pi. We chose to write the simulator in Python because of the very easy-to-use GPIOZero library that allows for control of the GPIO pins on the Raspberry Pi. We set six pins on the Raspberry Pi to have the same functionality as a racking system. The first five pins listed are all inputs that change the state of the racking system/the breaker. The last pin, “In Motion”,

is the only pin that provides feedback. When the racking system is moving, the last pin is in the logical high state.

Using this simulator, we were able to properly test our design, and our system could act on and respond to the racking system just like it would if it was plugged into a real racking system in a switchgear.

B. Hand-held Controller

The hand-held controller had to have five buttons on it for the four different actions the breaker can take and for an emergency stop function that cuts all power the racking system, stopping the movement of the breaker. There is also an additional switch on the controller that determines what to do if the current going to the DC motor in the racking system goes above its normal operating current. There are six LEDs that turn on and off in certain combinations depending on feedback from the racking system, so the user of the hand-held controller can know what state the racking system is currently in.

Since the hand-held controller has six inputs to and six outputs from the control board, twelve wires – not including the two for VCC and GND – are required. Since our design required the use of a standard eight-pin RJ45 cable, we had to use encoding/multiplexing to reduce the input and output wire count to six. To reduce the needed wires for the input buttons we used an 8:3 priority encoder. We chose the UA741, which is an IC that could be powered with 5V and has very low current. This chip allowed us to encode the six buttons to a binary representation using only three wires.

The priority of the buttons is as follows: E-Stop, Rack In, Rack Out, Close Breaker, Open Breaker, Stop/Reverse switch. This order was chosen because, E-Stop button's signal should always be sent with priority over a Rack In or a Rack Out button's signal. The remaining buttons' priority orders are not important until the Stop/Reverse switch. This switch had to have the lowest priority because if the other buttons had lower priority and the switch was in the on position, then none of the buttons would register when pressed. In other words, if the switch was the highest priority it would always take priority over everything else, and the other buttons would become useless.

The output LEDs were arranged in a manner that takes advantage of tri-state logic called Charlieplexing. This also allowed us to drive the six required LEDs with just three wires, which, in the end, brought us down to only eight wires. The disadvantage with Charlieplexing is that only one LED can be turned on at a time and our design requires multiple lights to turn on simultaneously. The solution to this is use the MSP430 on the control board to loop through the LEDs at a fast frequency, making it appear as if multiple LEDs are on at the same time. There will also be times when the LEDs need to appear to be flashing, such as when errors occur. The solution to this problem was to set the frequency of the strobing lower during error conditions which allows the user to know when there is a problem occurring.

The schematic for the hand-held controller is simple: each of the buttons has a .1uF capacitor in parallel with it that debounces the switch. Without that capacitor, software debouncing would need to be utilized. However, we chose to only add six capacitors to the design since it was low-cost and the space on the board was not a concern. All the buttons are pulled down with a 100KΩ pull-down resistor. Without this resistor, the input to the 8:3 encode on the board was floating and it would rapidly change outputs. A pull-down resistor was used instead of the more common pull-up resistor because otherwise the logic to the priority encoder and the logic to the MSP430 would be backwards and would take longer to code and debug.

C. Control Board

The control board is a PCB that the hand-held controller plugs into through an RJ45 jack on the front of the board. It provides all the components that give the hand-held controller power, contains the logic to send the correct signals to the racking system in the switchgear, and has the pin headers that allow the Raspberry Pi to send commands to the racking system.

The board is powered from two male pins on its side, one for VCC and the other for ground. This is so the board can easily be powered from a power source that is internal to the switchgear. The next stage of the power delivery is a TPS54560 buck converter. This converter was ideal for our project because it can convert a maximum of 60V down to our required 5V with a maximum current output of 5A. We needed at least 3A output since we will also be powering the Raspberry Pi from this power source as well, and the Raspberry Pi requires anywhere between 800mA to 2.5A depending on what type of load the CPU is under and how many I/O ports are being used. The max input of 60V was also important to us since the switchgear can be outfitted with a variety of voltage sources so it is important that the customer is not locked into one configuration.

The output of the buck converter powers the Raspberry Pi, the hand-held controller, the current sense circuit, and the 3.3V regulator. All these components have voltage and current requirements, so it is important to make sure the source is regulated, which is what the buck converter does for us. A maximum of 3A could be required by our circuit which is why we chose a non-linear voltage regulator. A linear voltage regulator with that much current being supplied from it would have to have a heatsink on it because

it would generate so much heat that it could negatively affect the chip.

The next major component of the control board is the current sense circuit. This circuit uses a INA138 chip, which allows us to turn the current going from the power supply to the DC motor into a measurable voltage. There is a .05 Ohm shunt resistor across two of the pins on the chip that all the current going to the motor goes through. This is known as high-side current sensing. We decided on high-side current sensing rather than low-side because it is easier to implement and it is more accurate since we measure the current before it goes through the DC motor.

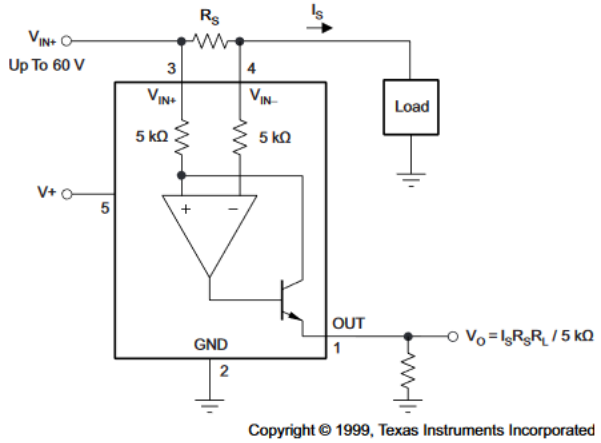

Figure 3. application of INA138 [2]

To know when the current is higher than normal, we created the circuit so that at 1.3A, the output is 3.5V. This way, we can set a pin on the MSP430 that is just as high and take the necessary actions. The following formula was used to determine the output voltage.

$$V_0 = I_s R_s R_l / 5000 \qquad (1)$$

We set V_o= 3.5V, I_s= 1.3A, R_s= .05Ω and that meant we had to set R_l as 270 KΩ in our circuit.

The biggest component of the control board is the MSP430 microcontroller. We chose the MSP430 because everybody in the group was familiar with it due to using the Launchpad Dev board many times in the past. It is also ultra-low power, drawing just 230µA when the microcontroller is active. Code Composer Studio, which is the development environment for the MSP430, is very easy to use as well, and has a great debugging feature that allows for real time changing of register values and allows us to view the state that every GPIO pin on the board is in. Based on these two factors we chose to use this microcontroller over any of the ATMEGA microcontrollers.

Since the MSP430 needs a 3.3V source to power it we had to include the 3.3V linear regulator in our design. We chose not to go with a switching regulator for this because the circuit was simpler without it, and we did not have to worry about the regulator getting hot because the MSP430 draws so little current that it is not an issue.

Before deciding on using the MSP430, we were experimenting with just using discrete logic chips since the operation of the racking system was thought to be very simple. As time went on, we discovered that this was not a possible approach because making sure that the operator of either controller does not make a critical mistake that ruins the racking system was not easy. Also, we discovered that the signals that the racking system needs to move the breaker had to last around 60ms. If discrete logic chips were used, then the onus would be on the operator to make sure that they hold the button for the required amount of time.

To get around relying on the operator to perfectly use the racking system, we decided that using discrete logic chips was not ideal and that we would instead use a microcontroller. The microcontroller is also needed to interpret the signals from the Raspberry Pi since they are a combinational input to the MCU and not one pin per instruction to the racking system.

D. MSP430 Microcontroller

The MSP430 microcontroller is responsible for figuring out what to do with the inputs sent to it and for making sure that the correct LEDs on the hand-held controller are turned on. The inputs from the hand-held controller trigger interrupts in the MSP430 on port1. Interrupt-based programming was ideal for the inputs since all inputs from both the controller and the Raspberry Pi should be handled immediately. The output LEDs are controlled by using three pins on port 2. The three pins switch between output low, output high, and input high impedance to turn on different LEDs at various positions on the circuit. Each LED has a structure that is stored in an array on the microcontroller which contains the pin configuration needed to light it and a Boolean that determines if it needs to be lit. There is a timer interrupt that is set to go off fifty times per second for each LED so that they will blink fast enough that they will appear to always be on.

Our sponsor ABB specified the LED outputs for each stage of the racking system.

Breaker error table						
BREAKER STATUS		RELAY OUTPUTS				
CLOSED	OPEN	DISCONNECT	TEST	CONNECT	RACKING	Error conditions
X	X	ON	OFF	OFF	OFF	locked into DISCONNECT position
X	X	OFF	ON	OFF	ON	locked into TEST position
X	X	OFF	OFF	ON	OFF	locked into CONNECT position
OFF	ON	ON	ON	OFF	ON	racking between DISCONNECT and TEST
OFF	ON	OFF	ON	ON	ON	racking between CONNECT and TEST
OFF	ON	FLASHING	FLASHING	OFF	FLASHING	error between TEST & DISCONNECT
OFF	ON	OFF	FLASHING	FLASHING	FLASHING	error, between TEST & CONNECT
FLASHING	OFF	X	X	X	FLASHING	Racking operation attempted breaker in closed position

Figure 4. LED output configurations for each state

There is an 8-bit unsigned variable that keeps track of what state the racking system is in. each bit relates to a feature of the racking system.

7	6	5	4	3	2	1	0
Overcurrent	In motion	Racked In	Test Position	Racked Out	Open/Close	E-Stop	S/R

Figure 5. Rack State variable

- Bit 0: Controls whether the racking system stops or reverses when too much current does to the DC motor
- Bit 1: 1 if the racking system is in the emergency stop state, else 0
- Bit2: 1 if breaker is open and 0 if breaker is closed
- Bit 3-5: 1 if racking system is locked into that position
- Bit 6: 1 if the racking system is moving else 0
- Bit 7: 1 if the overcurrent pin on the MSP430 is high else 0

When an instruction is issued from either the hand-held controller or the software-based web application, this variable displayed in figure 5 is checked. Depending on which bits are set in the variable, the command that was issued might not be viable. For example, if the breaker is already racked in and the user issues a rack in command, then that is not a valid command.

Since each command needs to send a certain signal to the racking system, there is a dedicated pin on the microcontroller per each pin on the racking system that will go from logical low to logical high to tell the racking system to do that action.

Action	Pin
Rack-In	P1.7
Rack-Out	P1.6
Close Breaker	P2.4
Open Breaker	P2.5
E-Stop	P2.6

Figure 6. MSP430 pins mapped to the action the enact

E. Raspberry Pi

The Raspberry Pi is used in junction with the control board to allow the capability to control the racking system through HTTP requests. It also allowed us the ability to put the Windows IoT operating system on it which fits in perfectly with the environment that ABB has set up. For the moment, we are just demonstrating that the HTTP calls work, so we have Raspbian installed as an operating system. The GPIO headers are connected, through wires, to the male pins headers on the control board which allows for communication between the two. The drawback of this approach is that there would have to be a Raspberry Pi for every racking system in each switchgear. It is possible to have a Raspberry Pi per switchgear and it could control up to six or eight racking systems at a time with Bluetooth modules on each control board, but at this time we did not explore that since we were only concerned with controlling one racking system.

There are 6 pins on the control board that connect to the Raspberry Pi's GPIO. Three are outputs for the combinational signals that can control the racking system and the other three are inputs that are connected to the outputs of the hand-held controller so that the software-based web application can update properly. Without those three headers connected to the outputs of the hand-held controller, a situation could arise where the racking system looks like it is racked out on the web application even though a technician has already used the hand-held controller and has put it to the racked in position. In that situation, the operator of the web application would be confused as to why they are telling the racking system to rack in and it is not working. That is why it is important to always have a process running that is updating the database. This way there is never a part of the system that is out of sync with the rest of it.

IV. Software Design

The software design for this project had three separate parts: the web application, the Django API, and the functions that controlled the GPIO. All three of these had to work together to complete the system. The purpose of the web application is to allow the user to see the states of multiple racking systems and control them at the same time. Meanwhile, the Django API and GPIO functions were responsible for taking the user input and turning it into signals that are sent to the microcontroller.

A. *GPIOZero*

GPIOZero is a python library that we used to control the GPIO pins on the Raspberry Pi. Initially, we tried to use the lower level RPI.GPIO library, but we ran into issues. When setting pin state changes to certain functions, the function would always be executed twice even though the event only occurred once. This was not ideal since we would have to program in a roundabout way to make sure the functions only executed once. GPIOZero is built on RPI.GPIO and it cleans up the problems that we were experiencing. It has buttons and LED objects that are used as higher level I/O objects. All the inputs to the Raspberry Pi were defined as buttons because, even though they were just pins, they act as buttons. The buttons were able to be defined with a hold time which was useful because the real racking system needs a logical high signal for at least 60ms to perform its function.

B. *Django API*

Since we did not decide to install Windows IoT on the Raspberry Pi, we were able to pick whichever language/framework we wanted to use to implement the API. In the end, we chose Django since it is written in Python and it has an extension called Django REST Framework that allows for easy implementation of a REST API. Using this framework, it was easy to set the API's endpoints and tie them to a function that controlled the GPIO pins on the Raspberry Pi using the GPIOZero library.

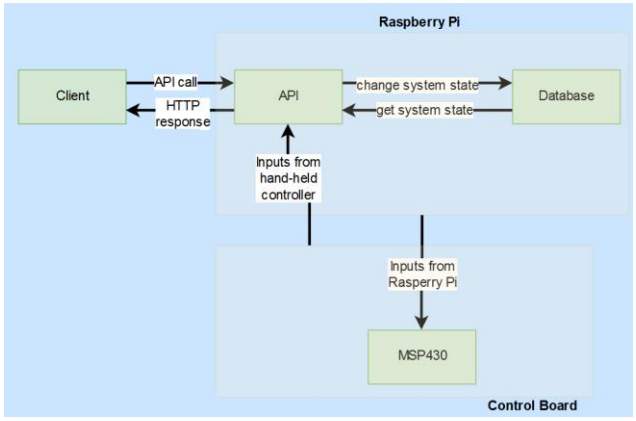

Figure 7. model of our design with the API

Figure 7 displays how the API fits in with our design. It is the piece of the system that translates button clicks on the web application to the setting of pins to high on the Raspberry Pi GPIO headers, which act as inputs to the MSP430.

When it comes to security, only hosts that are in the application's list of known hosts can access the API. This application also works inside of ABB's current network. Therefore, whatever security measures they deploy will also keep the API secure.

There are several endpoints that the API recognizes. These endpoints allow the user to move the breaker with rack in or rack out, to open and close the breaker, and to cut power to the racking system. They also allow some actions that get information from the racking system, such as what state it is in and who has used this racking system and when. If a command attempts to hit an endpoint that is not defined inside of the Django framework, then the user will receive a 404 error.

Action	Call	Endpoint
Rack In	GET	/rackin/
Rack Out	GET	/rackout/
Close breaker	GET	/breakerClose/
Open Breaker	GET	/breakerOpen/
E-Stop	GET	/estop/
state of racking system	GET	/racking system/
add a racking system	POST	/rackingsystem/
get list of operators	GET	/operators/
add operator	POST	/operators/
history of racking system	GET	/history/

Figure 8. List of possible HTTP requests

Figure 8 shows all the requests that the web application can send to the API. When the rack in, rack out, close breaker, open breaker, or e-stop requests are sent and valid, the database entry holding the state of the racking system is updated accordingly. Updating the database is important because it is the web application's source to determine what to display on screen and what commands are valid.

C. *Web Application*

The software-based web application is intended to be the next level product up from the hand-held controller. Encompassing all the features and usability of the hand-held controller, the software-based web application allows the operator to control the switchgear systems from a computer interface instead of interacting physically with the switchgear system itself. We were tasked with developing a prototype-level program that can represent this behavior.

The web application was created using C# and Microsoft Visual Studio 2017 for maximum compatibility with the existing Microsoft Windows network infrastructure located at the ABB switchgear installation sites. Using C# as the programming language for the web application allowed for full integration with the Microsoft Windows operating system through the .NET Framework

4.6.1. This increases the ease of implementation, maintenance, and addition of features to the controller.

The web application is divided into six major classes/feature sets:

- Login:

  This is the first window that the user of the web application will see. In the prototype version of the application, the login is not connected to a network or the server and instead makes use of a local credentials file stored in the csv format for ease of testing. For the production level version, the login will instead integrate with the Windows login server that is located at the switchgear system site. This allows for a centralized set of credentials to be used and will be far more secure than a local file.

- Main Navigation Window:

  If the user has successfully logged in with valid credentials, they will be sent to the main navigation window. Depending on the user's access level, detailed in a later section, the user will have access to certain buttons that lead to other windows.

- Manage Site Configuration Window:

  Only reachable by users with access level 2 or higher, this window allows the user to change the site configuration shown in the web application. The user can add, remove, and modify the various components from the site name all the way down to an individual circuit breaker level. It is through this window that additional web applications that are connected to other circuit breaker compartments on the various switchgear systems running at the site are added.

- Manage Account Window:

  This window is accessible by users of any access level and allows the user to change portions of their account information such as their password. In the prototype version of the web application, these settings will modify their account information in the local user credentials file only; however, in the production version, it will interact and modify the credentials located in the server at the site. Users with access level 3 will have the additional abilities of modifying, removing, and adding other user accounts.

- Manage Switchgear Systems Window:

  For users with access level 1 or higher, this is the main window that they will be operating from. It allows the user to view the status of every software controller installed in any of the circuit breaker compartments throughout the entire site. In addition, the user can issue commands to one or more switchgear system circuit breakers that are equipped with the necessary hardware for the software controller. The user interface has a couple of safeguards built outside of the protections provided in the control board. With safeguards against improper use and the remote command capabilities, it completely isolates and protects the user from what would otherwise be a substantially more dangerous job.

- Logging Window:

  All users, regardless of access level, can view the contents of the logs. This is so that if anything goes wrong, a specific individual will not be needed to access the logs and backtrack to what happened. Any commands sent or attempted to be sent are logged onto a storage card in the hardware that interfaces the web application and the circuit breaker unit in the switchgear system. All errors and actions taken by the hand-held controller are also logged onto this storage card.

VII. Conclusion

This project taught our team a lot about how difficult it can be to properly integrate software with hardware. We also gained experience with creating PCBs, which none of us had ever done before. We ran into many problems while completing this project and I am sure if we had to do it again we could do it in a quarter of the time. This was the most difficult thing any of us have done in our time here at UCF, but it was worth the struggle. We believe that we have produced a good model for ABB to work from and hope that it will be used to connect to a real racking system in a switchgear.

Acknowledgement

We would like to thank ABB for sponsoring this project and allowing us to see professional documents that detailed the existing racking system circuitry. We would also like to thank UCF for having a class like this that allowed us to learn so much.

REFERENCES

[1] Jimb0. “Using EAGLE.” *Using EAGLE: Schematic*, Sparkfun, learn.sparkfun.com/tutorials/using-eagle-schematic.

[2] Texas Instruments. “INA1x8 High-Side Measurement Current Shunt Monitor.” Texas Instruments, Dec. 1999.

[3] Texas Instruments. “CMOS 8-Bit Priority Encoder” Texas Instruments, Oct. 2003.

[4] Texas Instruments. “LM3940 1-A Low Dropout Regulator 5-V to 3.3V Conversion” Texas Instruments, May. 1999.

[5] Texas Instruments. “High Voltage 12V – 400V DC Current Sense Reference Design” Texas Instruments, Mar. 2015.

[6] Texas Instruments. “MSP430G2x53 Mixed Signal Controller” Texas Instruments, Apr. 2011.

[7] Texas Instruments. “TPS54560 4.5V to 60V Input, 5A , Step Down DC-DC Converter with Eco-mode (Rev. C)” Texas Instruments, Mar. 2013.

AUTHORS

Michael Dean is graduating UCF as a computer engineer. His interests are computer architecture and computer science. He currently works as a contractor at Kennedy Space Center doing IT security but hopes to work for a company like Amazon, Microsoft, or Intel in the future.

Christopher Miller started his engineering journey after transferring to the University of Central Florida in the Spring of 2015. His interests include modeling and simulation, military technology development, and AI/Machine learning. After graduating with his bachelor's degree in Computer Engineering, Christopher will be continuing his full-time position as a Computer Programmer II for the Naval Air Warfare Center here in Orlando.

Huy Tran is a 26-year-old graduating as an electrical engineer from the University of Central Florida. His interest includes circuit design and control systems. After graduation, he wants to work as a circuit designer for Texas Instruments.