Introduction

My name is Cole Fuerth, and for my Electronics Engineering Technology capstone project, I built a control system for an electric motorcycle. This project, unlike most of its peers, does not have a set cycle or repeated function. Instead, my system monitors ongoing telemetry, and uses input from the user and peripheral devices, to safely control an electric vehicle, with some safeties, and a basic throttle control. This system monitors total battery voltage, as well as the voltage of individual lithium cells, and the current draw of the total system from a 60VDC source, and the current draw of the control system at 12V.

The biggest challenge of creating this system is creating a safe and reliable interface between a 5V TTL ecosystem, a 12V control system, and a 60V drive system. Instrumentation amplifiers and operational amplifiers are used for measuring and isolating analog input and output signals with the Arduino Mega controller, and opto-isolators isolate the 12V inputs, while opto-isolated relay packs handle 12V output control, and some control of the EV Controller functions, in place of switches.

Much effort was put into hardware safeties. One of these important features is the ability to monitor individual cell voltages. Lithium cells can be dangerous when misused, so if a heater safety is tripped on a cell in parallel with others, or one cell has a much higher internal resistance than others, the load is not evenly distributed, and there is a discrepancy between capacities of the cells in series. This can become a big problem in the short and long term, as it is important all the cells wear evenly in a system with 200 lithium cells networked together.

Overall, I am very proud of this achievement in creating this project, and although the conclusion of the Capstone approaches, this system will continue to grow and improve as its use continues long past the conclusion of EET 617.

System Design:

Processor Logic Layout

Since the processor needs to handle and map a large volume of information within and outside its architecture, arrays were used for data handling, and referenced locations within these arrays by defining pointers into the arrays to access information when it was needed. This proved to be a very effective system, as the code was very messy and hard to understand before this system was implemented. This allowed handling of all the IO at once when mapping inputs and outputs, by defining arrays in global memory for storing the status of bits and integers being used, along with their corresponding hardware pins when applicable.

Within the main program loop, all logic is done within subroutines, and all subroutines are called within the loop. This allows the separation of logic into routines with a specific task, so one task can be skipped according to user settings, for example, the LCD display can be disabled in settings.

The program starts with importing libraries for the DS3231 (Nextion libraries were scrapped, and replaced with methods that were developed (because the stock Nextion methods were too slow), then definitions, followed by all global memory declarations, including arrays, settings, and pointers into arrays, and misc global memory used to support methods, and then the main program flow. The program starts by running the setup() method, which sets up auxiliary devices and digital IO based on user settings and feedback from auxiliary power, and sets any status bits that need an initial value before the main program cycle begins. After setup completes, the processor begins looping the loop() method, which calls all of my methods I created to handle the vehicle functions.

Methods

Many processes within the logic are repetitive, so methods were created to promote efficiency. A method is a section of code that can be called with or without parameters, that returns either void or a variable defined with the method. This resulted in much cleaner memory management, and an easier to understand program. A good example of this format is my OneShot logic. There are 32 globally available oneshot bits, as well as 32 dedicated to each fault, for displaying fault messages once when a fault goes true. OneShot are managed automatically, with a tracker that is reset at the end of every cycle, and every time a oneshot is called, that bit is incremented, so the call will adapt to the number of oneshot calls used.

Other calls were created for things such as writing Nextion values and settings, whose process is detailed below in the Nextion HMI Display section.

Other examples of methods created are voltage to analog translations, which would be able to convert back and forth between floating point voltages, and integer values for either analogRead or analogWrite values, an on timer, or 'TON', a limit method that returned a Boolean, similar to that used in PLCs, and methods for addressing button presses on the HMI.

There also exist methods for returning string values; one is called every scan and cycles through active faults, to be displayed on general information pages at the top in a fault window, and another generates a list of the top 10 active faults sorted by priority, which can be returned individually by requesting spot 0-9, to be displayed in a list of faults on a window in the HMI.

Fault Handling

A Fault-oriented safety system handles safety processes, with a motor contactor that can break the connection between the battery and the EV controller. This connection is broken either by the processor via a relay when a Level 1 (Highest level) fault, or a throttle input exception is detected. The contactor coil is connected in series with a control relay, and a kill switch on the handlebars, so that if the processor freezes, the user still has a hard-stop to kill power to the motor. A level 1 fault is defined by a critical program fault in my functions, such as a miscount of automatically handled functions like ONS or Timers, or connection via RJ45 to the handlebars is lost, or throttle input leaves the acceptable range. A level 1 fault is a fault that requires a disconnect of power to the EV controller. The processor uses a watchdog to ensure IO updates are occurring fast enough to be reliable, and if deemed unreliable, throws a level 1 fault.

Faults are handled with an array of fault flags, differentiated by pointers that point to their memory bit within the array. Parallel arrays to the fault flag array contain information on fault level, as well as a message to go along with the fault when it is flagged.

<u>Analog inputs</u>

Operational Amplifiers were used for analog signal isolation in and out of the processor, chosen for their reliable electrical isolation between input and output, and their ability to only repeat an analog voltage within the supply rails.

A stack of custom soldered breadboard circuits handles all analog signals between the Arduino processor and the rest of the system. This includes total battery voltage, voltage on each of the four 4S Lithium cells in series that make up the battery, signal levels from the two current shunts, and throttle input and output levels.

Current levels are determined using current shunts, which are very low resistance resistors, that are made to an extreme level or precision, and to a certain spec. An example of this spec would be, that my 12V shunt is rated to 75mV at 15A across the shunt. This means, when the shunt is passing 15 amps of current, it will cause a potential drop of 75mV. This potential difference is the analog signal that is processed by my stack.

Analog values are processed by Instrumentation Amplifiers in the stack. The type of amplifier used was an INA126P. This amplifier is fed rail voltages of +16V and -9V, so that any and all analog readings on the input pins are well within the rail voltages, ensuring accuracy. The default gain on an INA126P is 5, and the output potential of this package can be represented by: $V_{out} = (in_{+} - in_{-}) * (5 + \frac{80k\Omega}{R_G})$. In the case of a very low input differential, getting an accurate and useable output means we need a gain resistor, R_G. R_G was calculated for the 12V shunt, for example, by substituting the maximum current reading signal level as the voltage differential, and the maximum output potential desired on the Arduino input as V_{out} (4.50V), and solving for R_G. The closest value to the result that could be produced with the parts on hand was $1.47k\Omega$, which would produce a potential of 4.425V at 15A through the shunt. A similar process was followed for the 50mV 150A shunt for measuring current directly at the battery negative terminal, which calculated an R_G of $1k\Omega$.

Voltage levels on individual cells were collected using INA126P amplifiers and voltage dividers. Each cell node was divided, so that the four levels, spanning from ground to cell 4, were at a potential of 3.4V. This meant that, the potential drop across each cell after the voltage divider was about 0.85V. After using these references to generate a signal for each cell voltage through INA126P Instrumentation Amplifiers, the optimal signal level for each cell was approximately 4.25V. This level is easily and accurately able to be read and mapped to a real voltage level for the Arduino to use, and display to the user.

The schematic of the four cell voltage sensors:

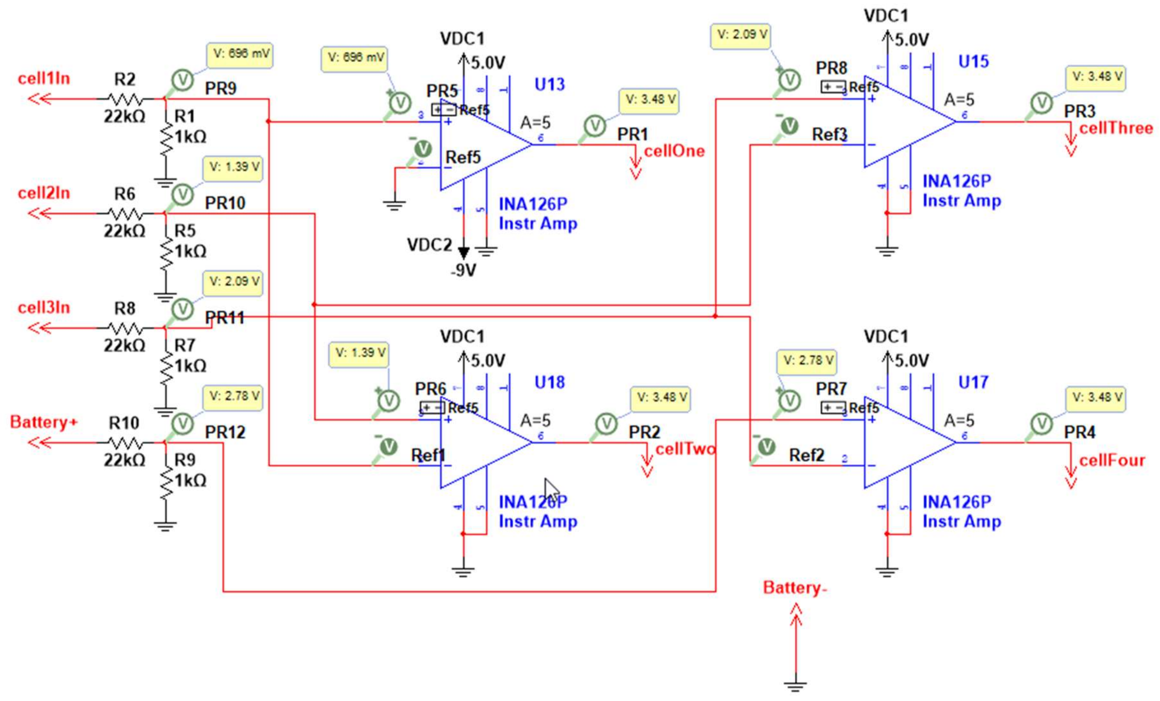

Figure 5-1

The total battery voltage was read using a voltage divider, same as above, so the battery voltage signal at 68V (charged), was 3.4V.

All analog inputs were buffered through CA3140 op amps, which provided reliable electrical isolation, and are linear op amps matching the pin configuration of LM741s, while being able to reliably operate close to the rail voltages, making them very useful as buffers.

The EV controller

The EV controller used was a 9kW CA Controller. This controller operates between 48V and 100V DC, and controls a brushless motor. The inputs of this controller include an analog throttle reading of 1-4V, as well as wires used for selecting modes of operation by grounding the connections. The throttle signal is provided by a high-frequency (980Hz) PWM using duty cycle to control the DC level, which is passed through a CA3140 linear operational amplifier and a 2N4244 NPN transistor to provide electrical isolation, and a power amplifier capable of delivering up to 500mA. The controller does not sink this much current, but a CA3140 can only source 14mA of current, so a power amplifier circuit was implemented to avoid stressing the analog buffer. Control of the mode selection was accomplished using a relay pack dedicated to ESC control, which operates the motor contactor solenoid, as well as shorting out control wires as mentioned previously.

Light Control

Lights are controlled by 6-channel relay packs, at the front and rear of the vehicle. Lights all operate at 12V DC, and are controlled by the Arduino, using switches on the handlebars of the vehicle. The bike features a headlight and taillight, brake light, front and rear turn signals, and fog/trail lights. Some features unique to this bike from others is, most two-wheeled vehicles do not feature hazard lights or fog lights, or a master kill switch for lights, but these were implemented in this system. Fog lights are used at night, to light up the operator's peripheral vision. The light kill switch is useful for power conservation, and it never hurts to be sneaky, especially on an electric vehicle, when noise is already at a minimum. Turn signals were added so that the vehicle could be safe when passing though an urban environment.

Speed Sensing

Speed sensing is done using an inductive proximity sensor picking up spokes on the front brake disc. The front wheel has a 70cm outer diameter, and there are 8 spokes on the disc. Using the outer diameter, the wheel RPM was calculated at different speeds, and then using that information, the positive pulses per second could be determined at different speeds. The points found were at 100km/h, the wheel rotates at 12.64RPM, and generates 101 pulses per second, so ~10ms between each pulse, and at 25km/h, there would be

40ms between each pulse. A map() method was used with these two points for reference, since the relation is linear, to generate a value in km/h by measuring the time between each positive edge trigger from the PX to calculate the speed of the front wheel.

Nextion HMI Display

Design of the HMI display took a long time, as the system started with attempts to use a TFT display, driven by an Arduino Mega. This approach proved inefficient, so a different platform was adopted. Nextion is a very polished system, in relation to its competitors in the Arduino community, and proved to be easy to develop, thanks to its GUI interface for development. The stock Nextion libraries were unreliable and slow, so custom libraries were developed by studying how the default libraries would handle incoming and outgoing data, and replicating that serial format to create custom libraries designed around the vehicle's ecosystem. This method proved much faster and more reliable, and stopped the processor from 'hanging' during serial transfers, like it did when using the stock libraries.

The Nextion display has three screens: a main screen, an alarms screen, and a statistics screen. The 'MAIN' screen only displays essential information in a larger font, for ease of use on the go. The 'ALARMS' screen displays up to ten active alarms, in order of importance. The 'STAT' screen displays more information than the main screen, in a smaller font, for active monitoring of telemetry within the system.

The system is optimized such that, only elements on the current page are updated, and only one element is updated at a time. Elements are also only updated when the value changes from the previously written value, using a buffer. Further optimization was achieved by changing the BAUD rate on the Nextion from 9600 to 115200, using a custom method.

Drive Safeties

The vehicle was designed such that, in order to arm the vehicle, a certain set of conditions must first be met. Firstly, the kill switch must be on. Secondly, the kill switch must have been in the 'open' position at least once since boot, to prevent accidental arming on boot. Thirdly, the throttle must be at idle when the start PB is pressed, in order to arm the motor. Any level 1, or 'critical' faults, will automatically open the motor contactor.

Charging Circuit

A charging circuit was implemented, to avoid disassembly of the battery pack for charging. Each 16S pack is made up of four 4S Li-Ion packs in series, so when in operation the pack operates as a 16S 5200mAh battery pack, but when in charging mode, an 8-channel relay board is provided power, and all four of the 4S packs switch into a parallel configuration, on a different connector than the one used for normal operation. This allows for charging of all the packs at once, without disassembly of the entire battery assembly. When no power is applied to the relay packs, the pack defaults to the NC contacts, and the pack is once again in a 16S series configuration.

A dedicated 5V rail and charging rail are entirely isolated from the normal, operating circuits, so in charging mode, the cells are completely disconnected from the system.

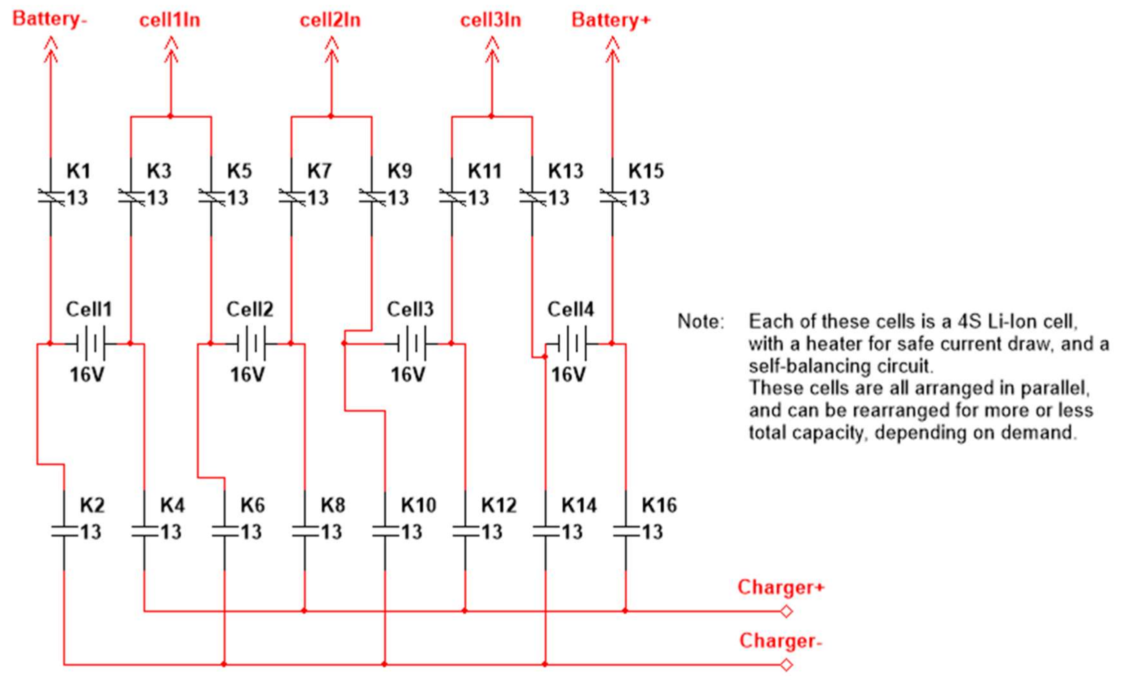
Battery-
cell1In
cell2In
cell3In
Battery+
K1 13
K3 13
K5 13
K7 13
K9 13
K11 13
K13 13
K15 13
Cell1 16V
Cell2 16V
Cell3 16V
Cell4 16V
K2 13
K4 13
K6 13
K8 13
K10 13
K12 13
K14 13
K16 13
Charger+
Charger-
Note: Each of these cells is a 4S Li-Ion cell, with a heater for safe current draw, and a self-balancing circuit.
These cells are all arranged in parallel, and can be rearranged for more or less total capacity, depending on demand.

Figure 8-1

Background Theory:

Instrumentation Amplifiers

An instrumentation amplifier is a type of operational amplifier that produces an output signal proportional to the difference between two input pins. The instrumentation amplifier used was an INA126P, which is an 8-pin op-amp package, which has a pin configuration compatible with standard solderable boards, or breadboards. Both inputs must be within a ±0.7V range outside of the supply rails, so a negative and positive supply greater than the range of inputs is recommended, but the INA126P is a package capable of single-ended operation, meaning that the negative supply rail can be grounded, and the package will still behave reliably. The package is capable of up to ±18V supplied, or a 36V difference in potential between the supply rails.

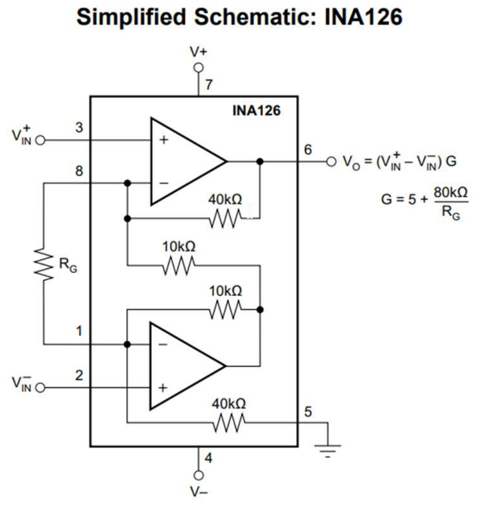
Simplified Schematic: INA126
V+
7
INA126
V_{IN}^{+}
3
8
6
$V_O = (V_{IN}^{+} - V_{IN}^{-})\,G$
$G = 5 + \frac{80k\Omega}{R_G}$
40kΩ
10kΩ
R_G
10kΩ
1
V_{IN}^{-}
2
40kΩ
5
4
V−

Figure 9-1

Output potential on pin 6 of this package can be calculated with the formula:

$$V_{out} = (in_{+} - in_{-}) * (5 + \frac{80k\Omega}{R_G})$$

R_G is a gain resistor, connected across pins 1 and 8. With no gain resistor, this package defaults to a gain of 5, but when a higher gain is desired, output can be amplified. This is especially useful when conditioning a signal from a current shunt.

Linear Operational Amplifiers and Power Amplifiers

The CA3140 Operational Amplifier is a package matching the pin configuration of an LM741, but able to reliably operate on a single-ended supply, and still reproduce an analog signal near the voltage supply rails. This characteristic is imperative in an analog buffer or power amplifier.

The CA3140 operates on a supply of up to 36V across the supply rails but is rated to operate reliably in a 5V TTL system.

The CA3140 can be configured as an analog buffer using the following schematic:

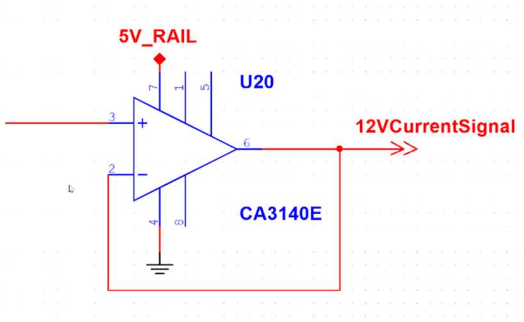

Figure 10-1

The CA3140 can only source up to 10mA maximum in a 5V TTL environment, so in a situation where this package is used as a voltage out buffer into a larger system, a power amplifier circuit is useful. Connecting the emitter of an NPN transistor package to the feedback (-) pin of the CA3140, the output of the package to the base of the transistor, and the collector of the transistor to the supply rail of the package produces a current supply of whatever the transistor is capable of, with the precision and isolation of the CA3140. This functionality can be accomplished with the following schematic:

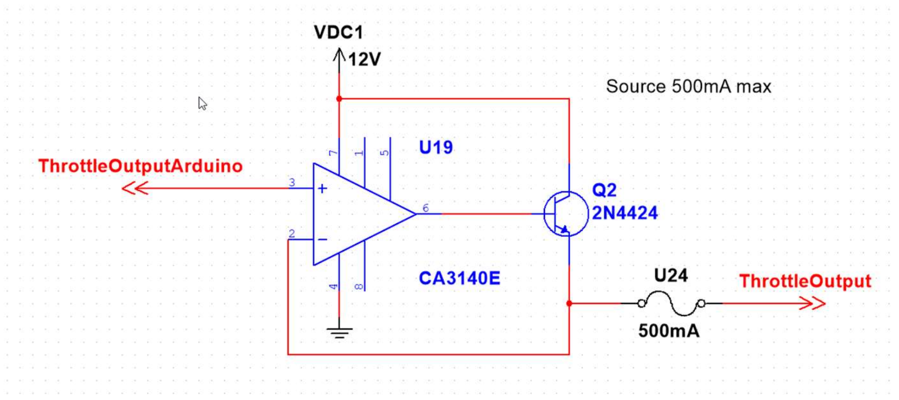

Figure 10-2

The 2N4424 Transistor package is an NPN transistor capable of sourcing up to 500mA, so in this configuration, a TTL processor can provide a PWM DC signal via pin 3 on the CA3140, and have electrical isolation from the output, along with a 500mA current source available, should it be needed.

Voltage Dividers

A voltage divider is a small, simple circuit that typically 'divides' a large signal into a smaller, useable signal. In this application, voltage dividers are used to sample battery voltage levels, and step them down to a level able to be processed by TTL circuits.

A voltage divider can be represented by the equation: $V_{out} = V_{in}(\frac{R_1}{R_1+R_2})$. For example, the total battery voltage is 68V, so the voltage divider used was $R_1 = 1k\Omega$, $R_2 = 22k\Omega$. This meant, the 68V input potential is reduced to 2.96V, which is well within the range of analog voltages handled in a TTL environment.

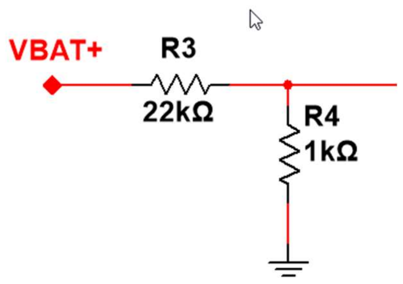

Figure 11-1

Opto-Isolators and Relays

Opto-Isolators and Relays are useful IO interfaces between a processor and external components in a control system. Opto-isolators utilize LED lights and photodiodes to provide complete electrical isolation between a processor and a control system, while operating at extremely fast speeds. Opto-isolator connections are typically active-low, meaning that when no potential difference is present across the inputs, the output to the processor is a TTL 1, but when a signal is detected at the input, a TTL 0 is sent to the processor.

Relays can be used together with optocouplers to provide two levels of isolation: The first level is the opto-isolated logic input from the processor, where a LOW signal is considered a logic TRUE. Then, an auxiliary power supply (5V, in this instance), will provide power to the relay coil when a TRUE signal is received, energizing the coil. The contacts on the relay can then operate up to a 250V / 10A control, providing complete isolation between the processor, 5V supply, and 12V control circuit.

Nextion HMI Libraries

The Nextion 3.2” Standard model was used as an HMI display. This model uses a GUI to design the interface, where different types of elements can be placed. These elements are each given an elementRef and a number to identify which element is which.

Serial communication to the Nextion is conditioned manually via a custom method, as the stock Nextion libraries are buggy.

A typical message to Nextion to set a ‘text’ of elementRef ‘t0’ is conditioned as follows:

‘t0.txt=”test”’ is sent using a Serial.print() command, followed by three Serial.write(0xFF) commands to signify the end of a message. All elements are set this way, by sending strings over serial to the Nextion followed up by ‘0xFF’ three times, which is then interpreted by the Nextion as a command.

Some other examples of Nextion commands used are:

Command	Descripton
baud=115200	Sets the serial baud rate to 115200
t0.txt=”test”	Sets text element “t0” to “test”
b0.val=20	b0 is a progress bar, which accepts any integer between 0-100
t0.pco=0x07E0	t0 text element font color is set to green

Table 12-1

Nextion also sends serial messages to the Arduino on event of a button press, if the checkbox “send component ID” is checked for either a press or release event. This message is structured such that:

- Message begins with 0x65
- Three bytes follow the initializer, which represent, respectively, the page number, component ID, and event type (always 0x01 for button presses)
- 0xFF received three times in a row signifies the end of a message

Monitoring the serial buffer for messages in this configuration allows the processor to interpret serial messages from the Nextion HMI.

Conclusions:

Overall, this project, “Control System for an Electric Motorcycle”, although not perfect yet, has evolved over time. The initial features planned are mostly nonexistent here, replaced with equally advanced, if not more advanced, features that were deemed more important to operation.

Initial pulse check for light operation was removed and replaced with monitoring of the individual 4S cell voltages making up the battery pack. Odometer tracking was removed, as it is unimportant in an off-road vehicle, and was replaced with more advanced safety systems and faults.

One feature that was not finished because of hardware issues, was SD Card logging. This feature, although not present at the time of submission, will be completed later, as it is essential to debugging once the vehicle is in active use.

The panels and electrical build did turn out better than expected. The panels are sleek and formed to the shape of the frame and provide ample space for any and all electrical components mounted inside the frame.

Due to issues with suppliers, the drive motor and controller have not yet arrived, and will not be present at the time of demonstration and will be replaced by a small drone motor and controller, for demonstration purposes.

The system does have full lighting controls, as well as safe, isolated IO on the processor. Other features that were successfully implemented include analog sensing on electrical circuits throughout the vehicle, including 12V and Motor current readings, as well as total battery power and monitoring of individual 4S cell potentials within the 16S pack.

A lithium charging circuit was developed, built-in to the custom packs, to remove the need for disassembly when charging. This auto-switching charging circuit is a massive quality-of-life improvement.

The program is the largest section of the project; many standards were developed for it, and it has undergone 7 months of constant growth, refinement, and optimization. Although there are still some bugs and issues, they will continue to be addressed and this system will grow beyond the conclusion of this project.

References:

- Nextion send and received logic was copied from the Nextion master library, provided by Nextion on GitHub, and modified for efficiency and speed within the vehicle's ecosystem. Core concepts of the code, although interpreted and structured using custom methods, and not exactly the original, were created by Nextion.

Bibliography:

- Nextion logic and examples were used to develop custom Nextion serial functions
- DS3231 Libraries were used for generating a time value from an RTC