LETTER OF TRANSMITTAL

Cole Fuerth
SCC Electronics Engineering Technology
2000 Talbot Rd
Windsor, ON. N0R 1V0

March 26, 2020

John Schlichther, Program Coordinator
SCC Electronics Engineering Technology
2000 Talbot Rd
Windsor, ON. N0R 1V0

Dear Mr. Schlichther and Mr. Watson,

I am writing with the project report titled “Control System for an Electric Motorcycle” enclosed. This project gained approval on October 3, 2019, from John Schlichther, and was completed on March 25, 2020, as it completed final preparation for submission. This report focuses on the milestones of progress throughout the project, as well as mostly focusing on program and electrical design. This project is still going to undergo further improvement and development, as it undergoes active use, but as of submission, its current condition is considered ‘final’ in this report.

Sincerely,
Cole Fuerth
SCC Electronics Engineering Technology, EET 617

Control System for an Electric Motorcycle

EET 617 Progress Report

March 26, 2020

Cole Fuerth

DECLARATION

I hereby declare that the project work entitled "Control System for an Electric Motorcycle" submitted to St. Clair College of Applied Arts and Technology, is a record of an original work done by me under the guidance of Mr. John Schlichther, Professor of Electronics Engineering Technology, St. Clair College of Applied Arts and Technology, and this project work is submitted in the partial fulfillment of the requirements for the award of the diploma of Electronics Engineering Technology – Industrial Automation. The results embodied in this thesis have not been submitted to any other College or Institute for the award of any degree or diploma.

Cole Fuerth

Cole Fuerth

March 26, 2020

Table of Contents:

List of Figures:

List of Tables

ACKNOWLEDGEMENTS

There are a few key people that were important in the design and fabrication of this project.

First, my friend Scott MacKay, who was around for the entire mechanical build of the project; he did most of the cutting and shaping of aluminum and steel, and all of the steel welding. The panels would not have turned out as well as they did without his help, and he was around for many, many hours during build.

My father, Brian Fuerth, welded the rear fender, and cut the front panels where the headlights and charging circuits are. He took an interest in the mechanical construction, and he will be helping with construction of the drive system of the bike, including motor and controller mounts, and a chassis for batteries.

Al Repmann is another individual who helped, he did most of the aluminum welding with his spool welder and some Tig welding.

Marko Jovonovic and John Schlichther were some individuals I consulted on opamp design, which were issues I would not have been able to efficiently tackle myself.

ABSTRACT

The EET Capstone project, “Control System for an Electric Motorcycle”, is a project designed to control a fully electric motorcycle, with lithium-ion cells, a charging circuit, an electric motor/controller, and safety and monitoring systems using an Atmega 2560 processor. This system uses amplifiers, optocouplers, relay boards, voltage converters, an EV controller, lithium balancing circuits, and several more systems to create a ‘smart’ system for monitoring the status of the system. Both the user via the physical killswitch, or the processor, with a relay killswitch, will be able to disconnect the battery from the EV controller. Individual monitoring systems on-board include:

- Current monitoring of a 12V supply
- Current monitoring of total draw from battery, up to 150A
- Monitoring of individual cell voltages, each cell is 4S lithium, and 4 of those in series
- Monitoring of total current draw from the battery in mAh
- Monitoring of the front wheel speed
- Monitoring of the total battery voltage
- Monitoring and recreation of the throttle value

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.

Analog inputs

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:

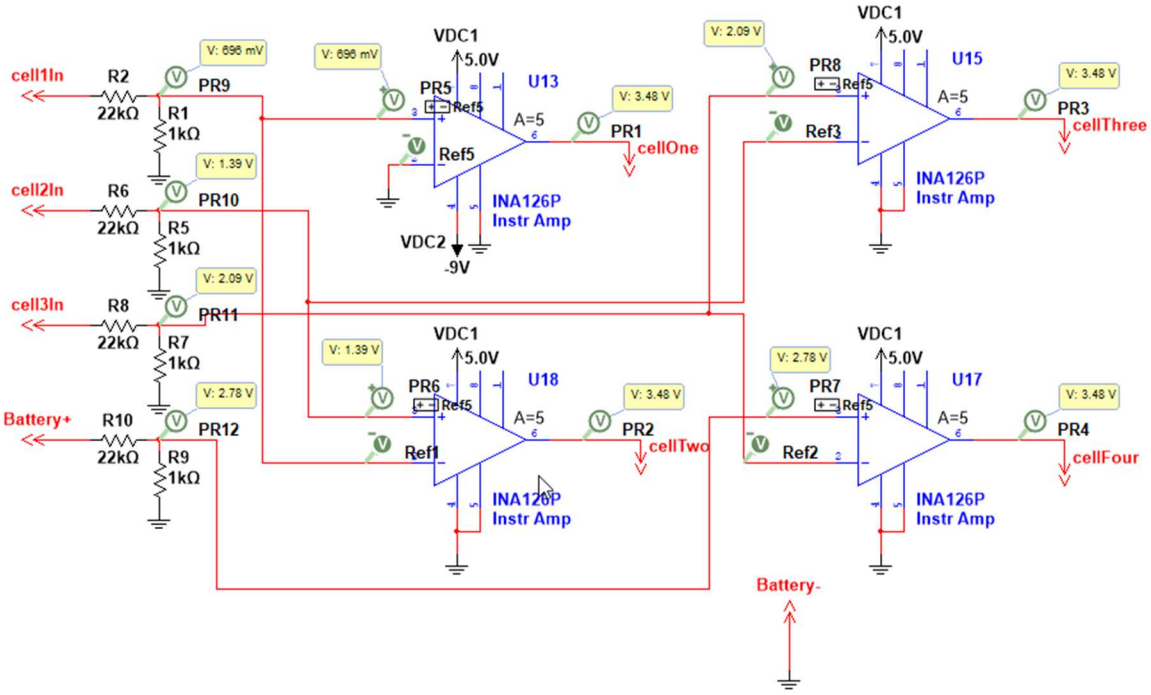


Figure 5-1

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.

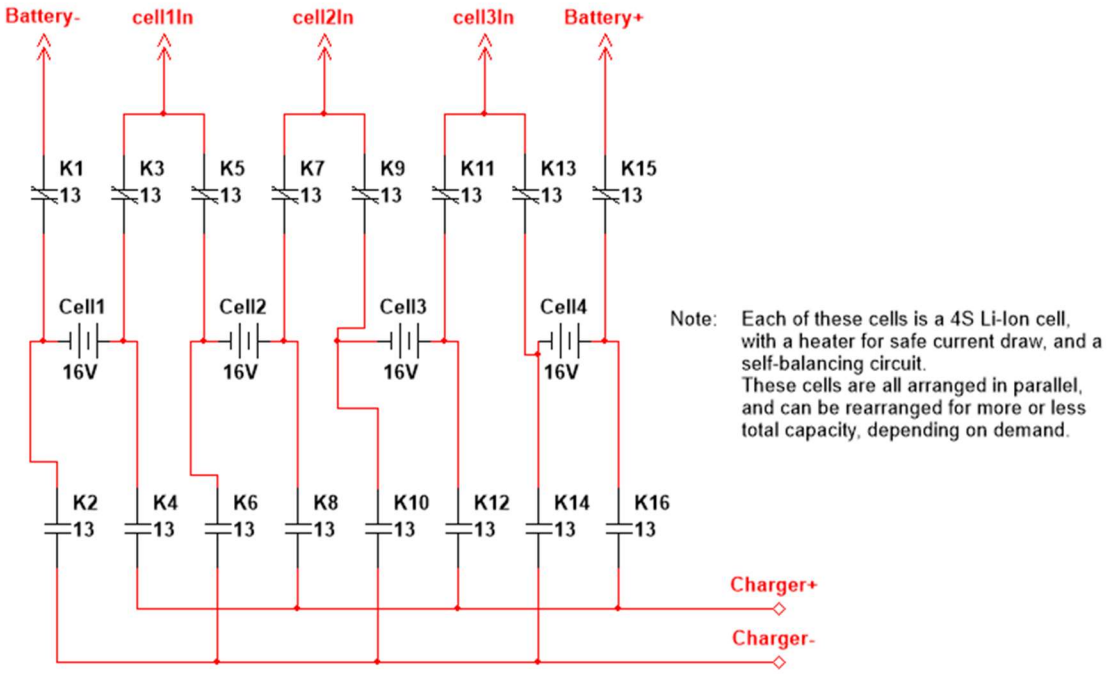


Figure 8-1

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.

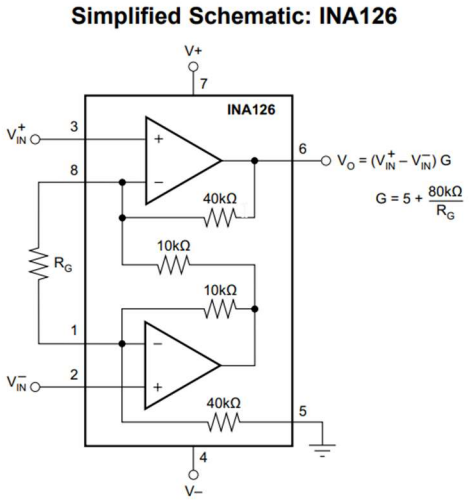


Figure 9-1

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:

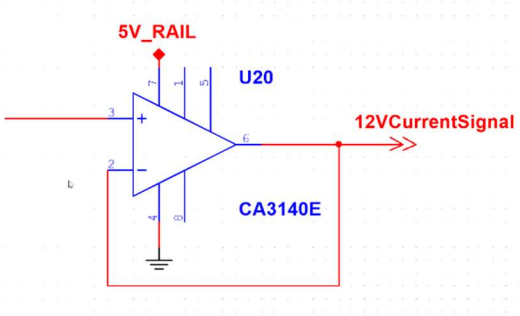


Figure 10-1

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:

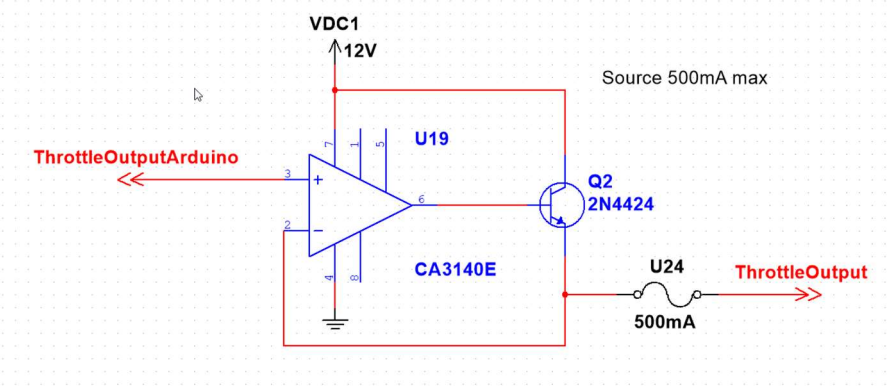


Figure 10-2

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.

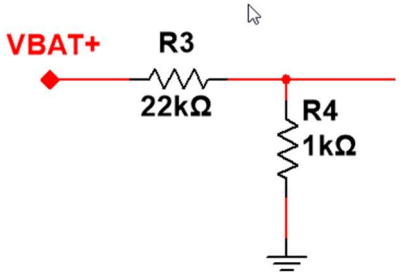


Figure 11-1

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.

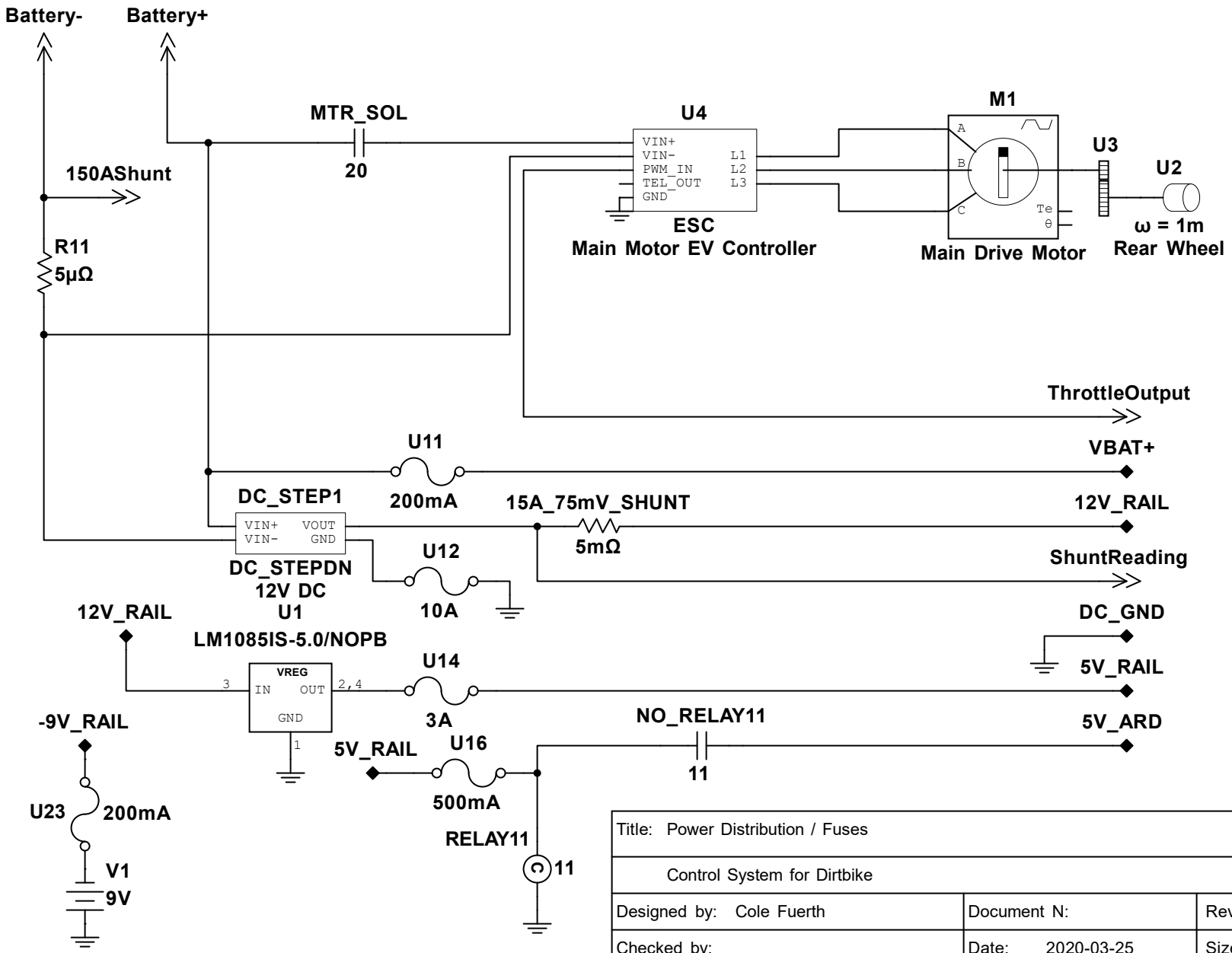
The program is the largest section of the project; many standards were developed for it, and it has undergone 7 months of constant growth, refinement, and optimization. Although there are still some bugs and issues, they will continue to be addressed and this system will grow beyond the conclusion of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

References:

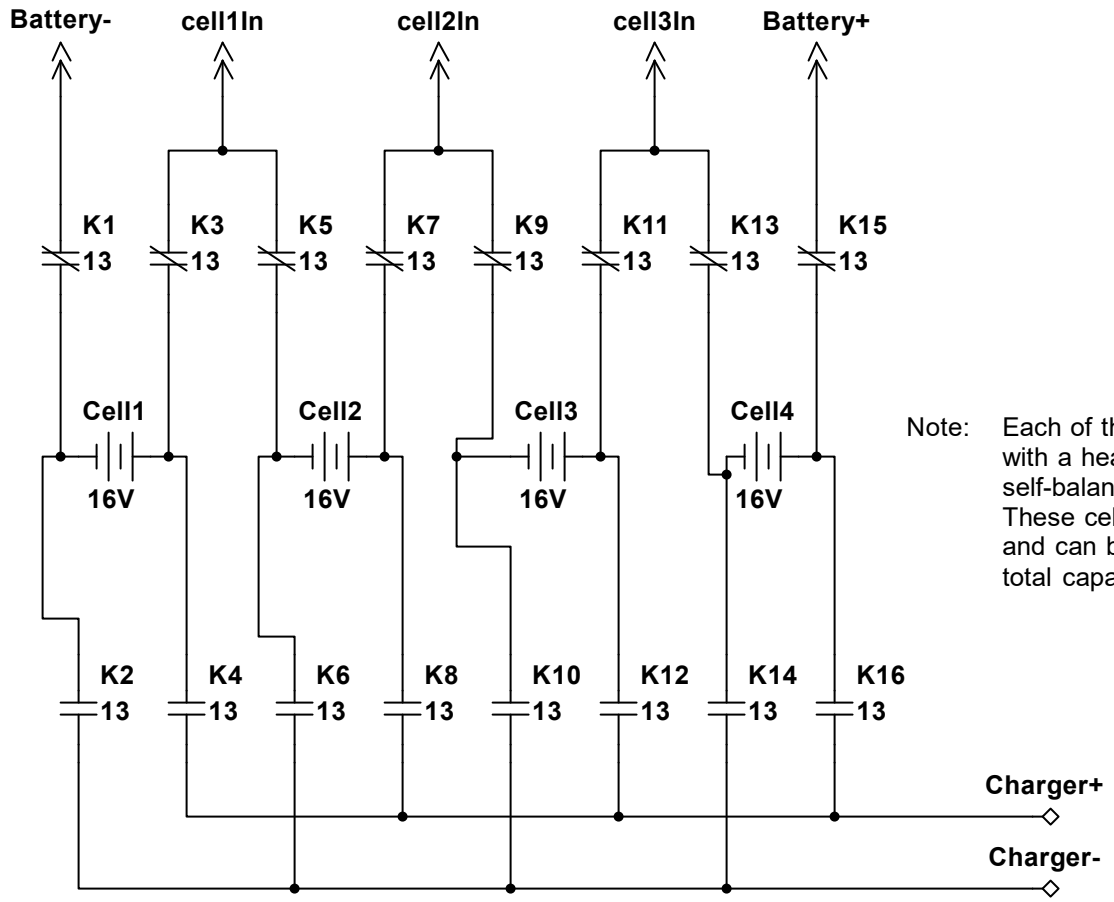
- Nextion send and received logic was copied from the Nextion master library, provided by Nextion on GitHub, and modified for efficiency and speed within the vehicle's ecosystem. Core concepts of the code, although interpreted and structured using custom methods, and not exactly the original, were created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

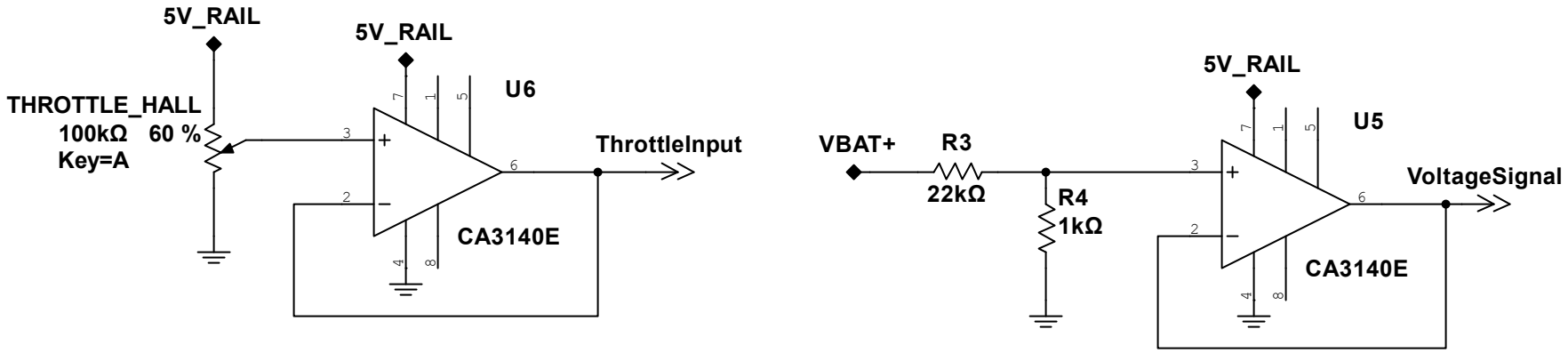


Title: Power Distribution / Fuses		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 1 of 12	

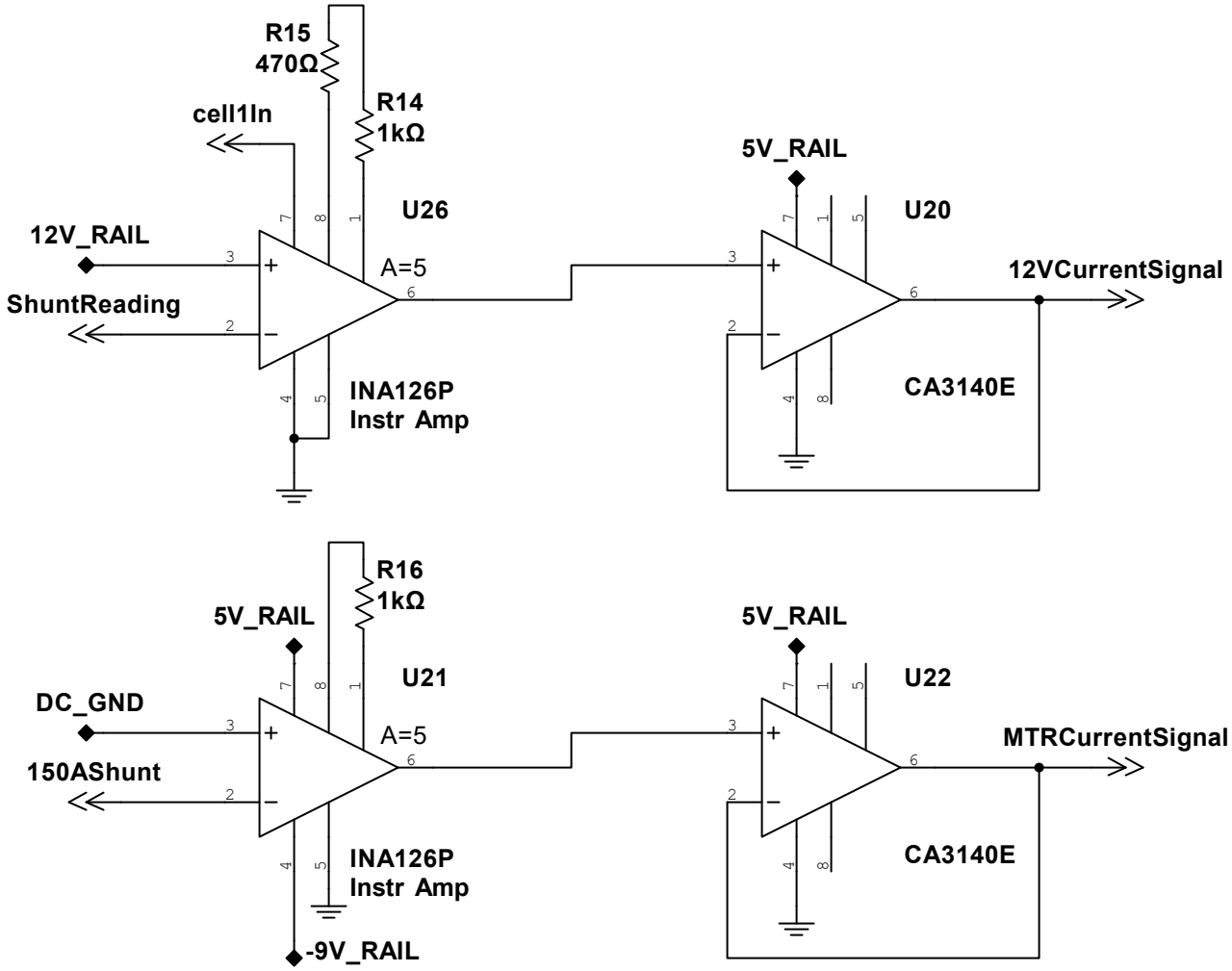


Note: Each of these cells is a 4S Li-Ion cell, with a heater for safe current draw, and a self-balancing circuit.
These cells are all arranged in parallel, and can be rearranged for more or less total capacity, depending on demand.

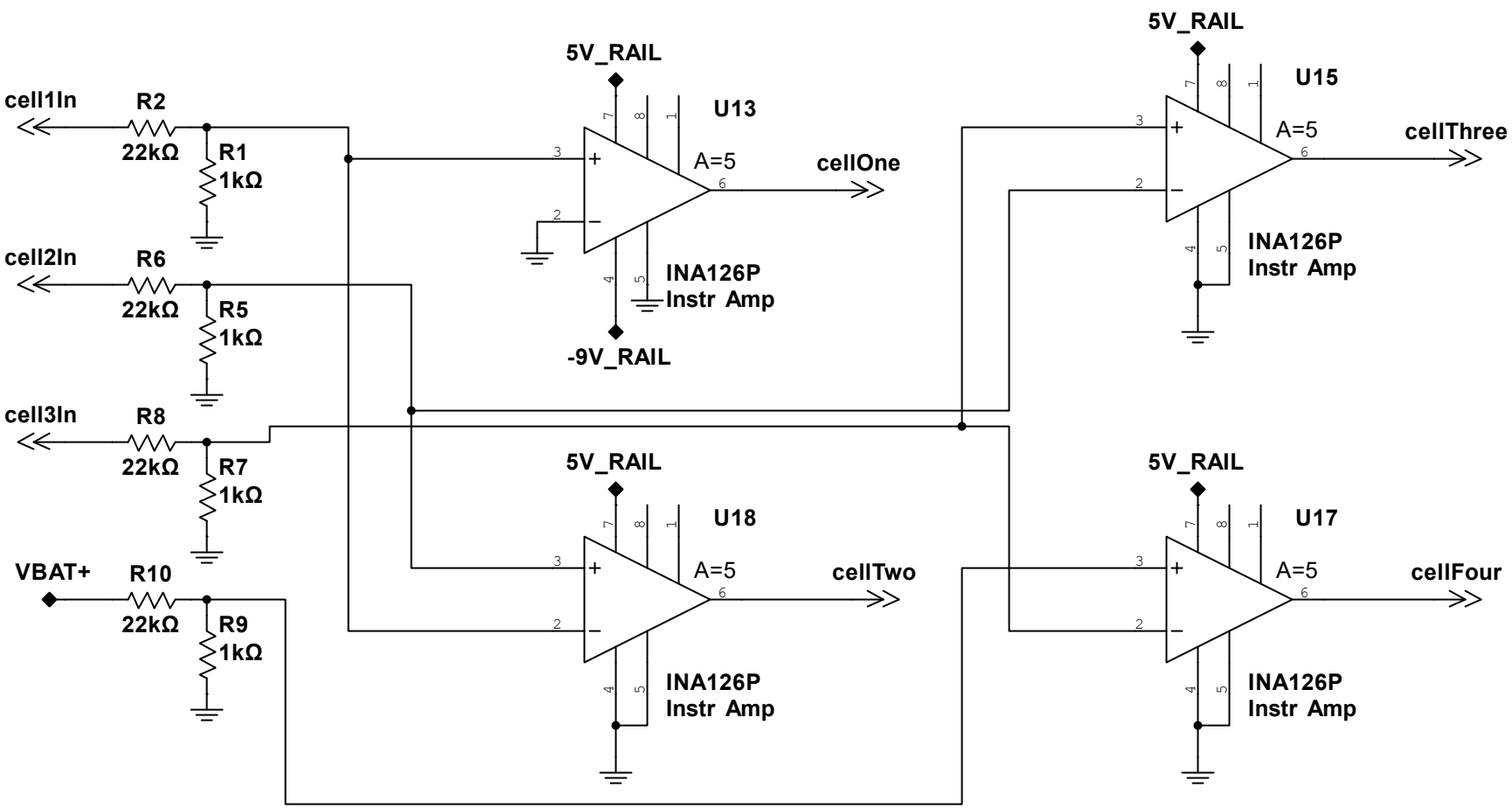
Title: Battery Charging Circuit		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 2 of 12	



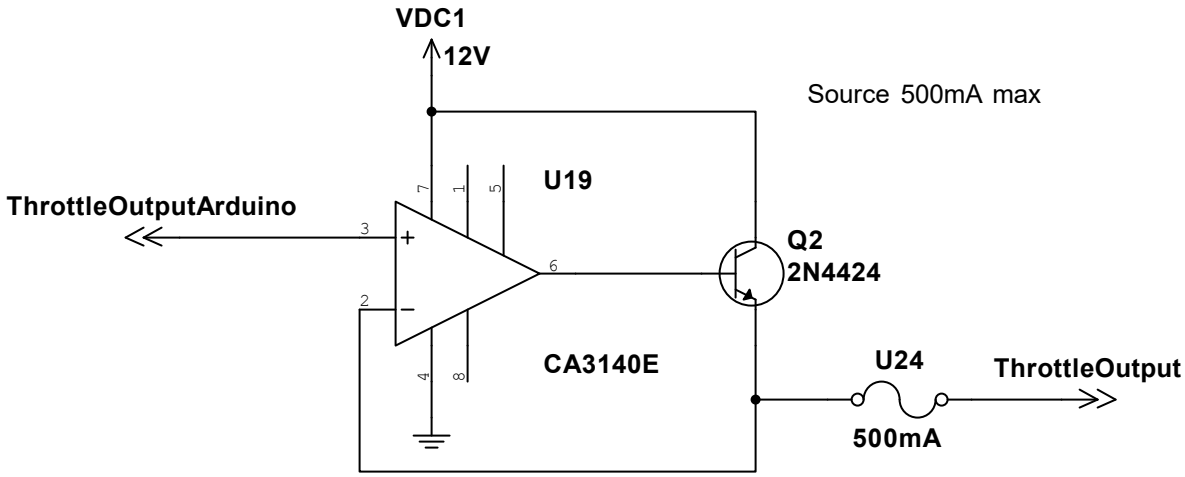
Title: Analog Sensing Circuits 1		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 3 of 12	



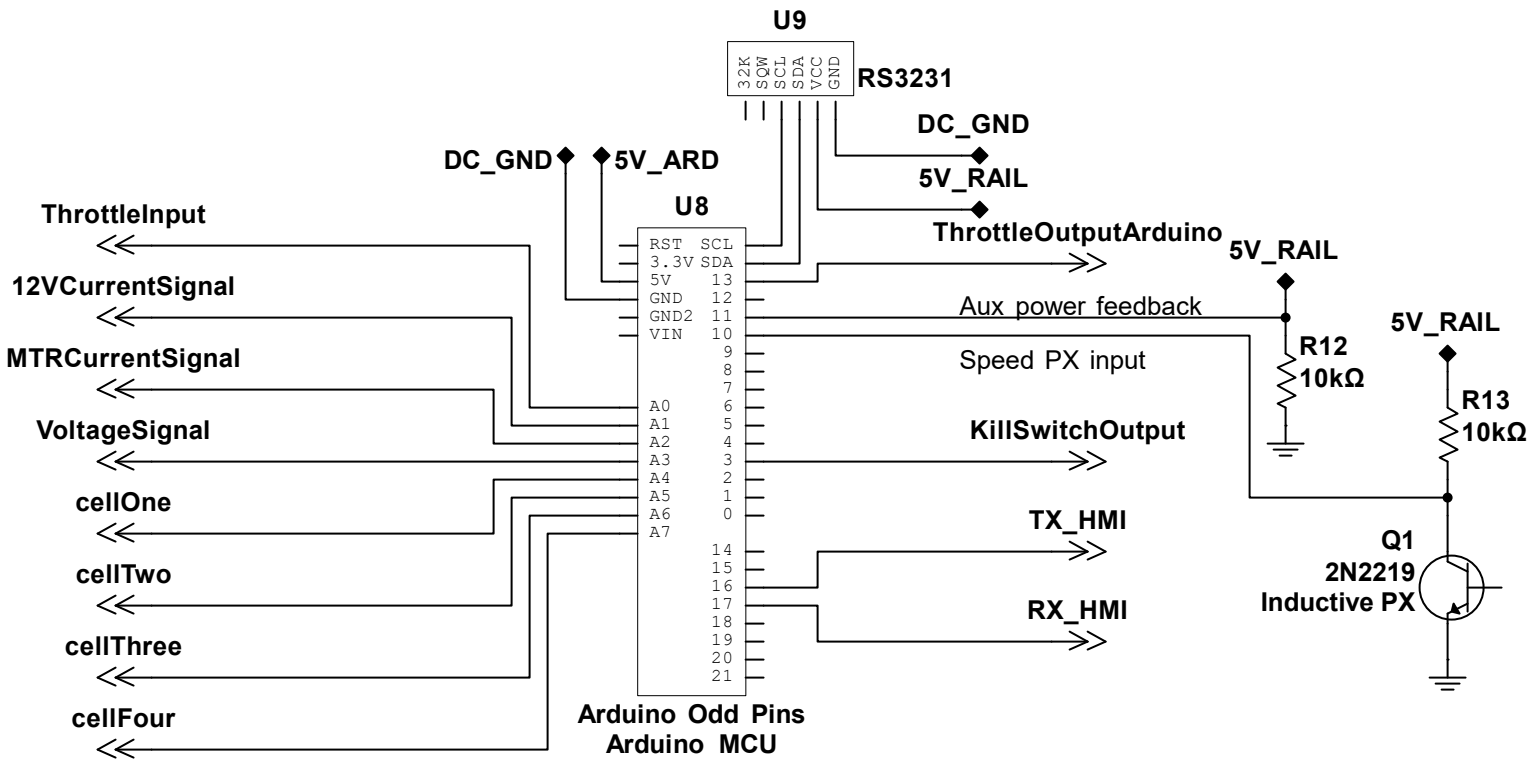
Title: Current Sensing Circuits		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 4 of 12	



Title: Battery Sensing Circuits		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 5 of 12	

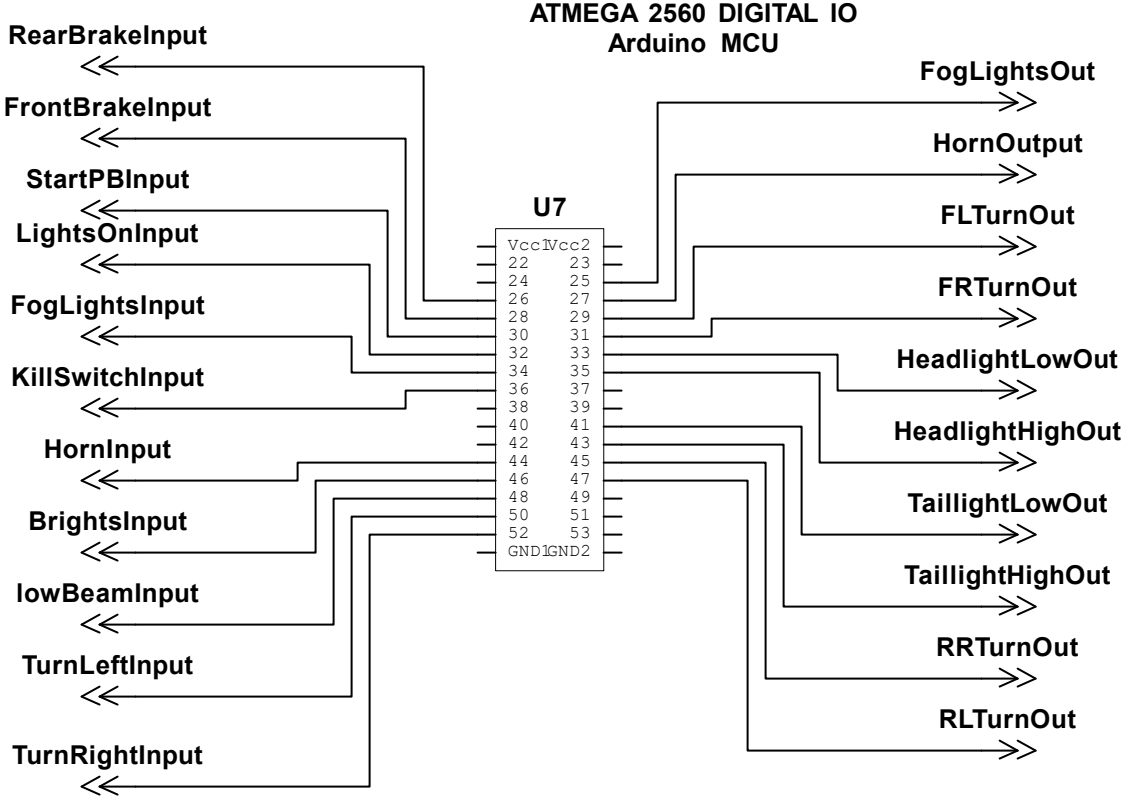


Title: Throttle Power Amplifier		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 6 of 12	

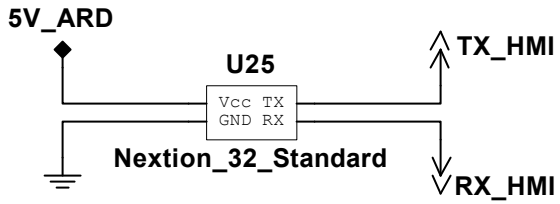


Title: Arduino Misc IO		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 7 of 12	

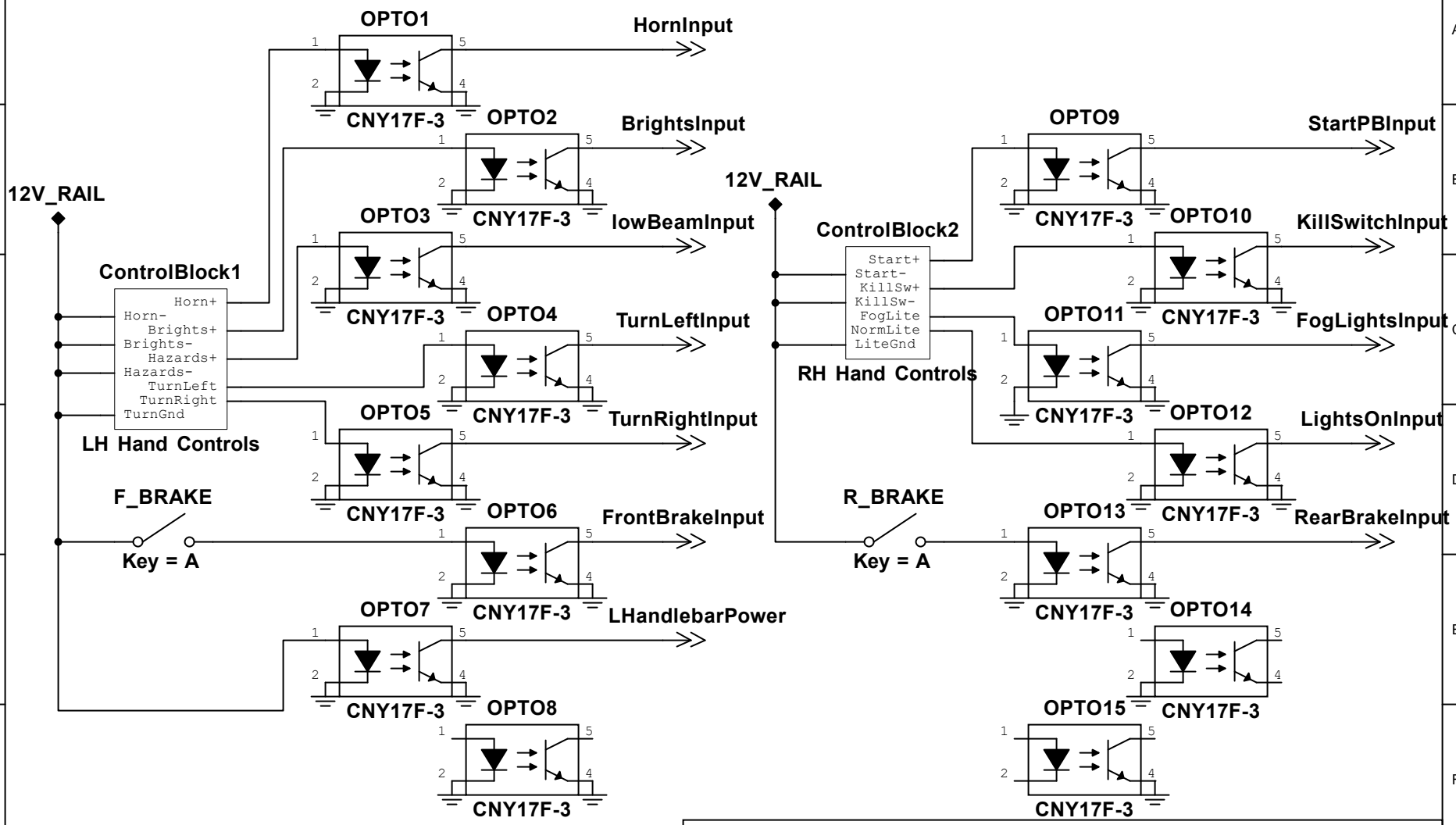
ATMEGA 2560 DIGITAL IO
Arduino MCU



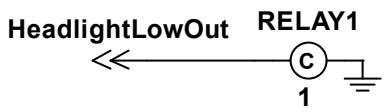
Title: Arduino Digital IO to Optocoupler and Relay Boards		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 8 of 12	

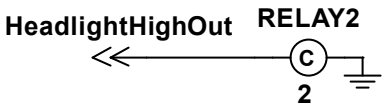


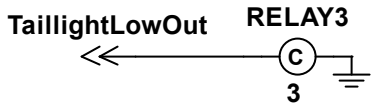
Title: Nextion HMI			
Control System for Dirtbike			
Designed by: Cole Fuerth	Document N:	Revision: A	
Checked by:	Date: 2020-03-25	Size: A	
Approved by:	Sheet 9 of 12		

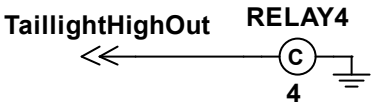


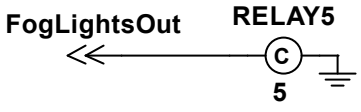
Title: Digital Inputs and Opto Isolators		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 10 of 12	

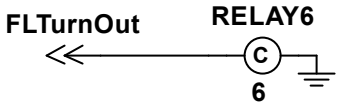


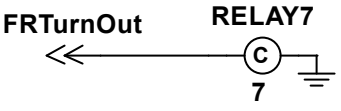


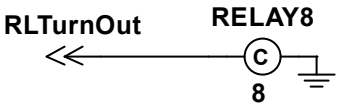


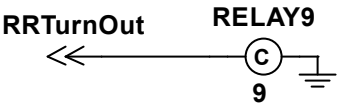


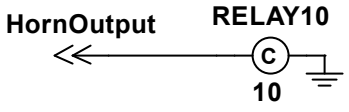


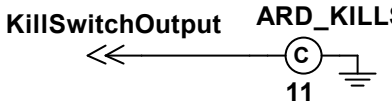


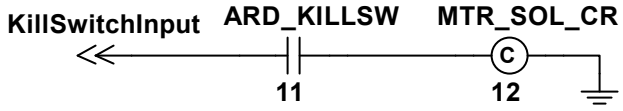


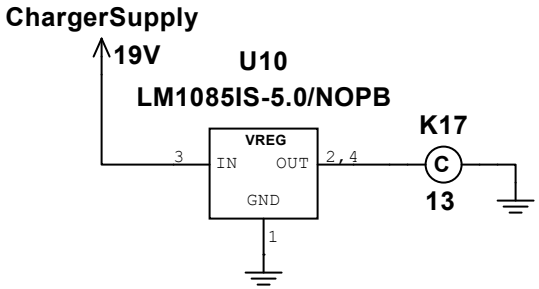




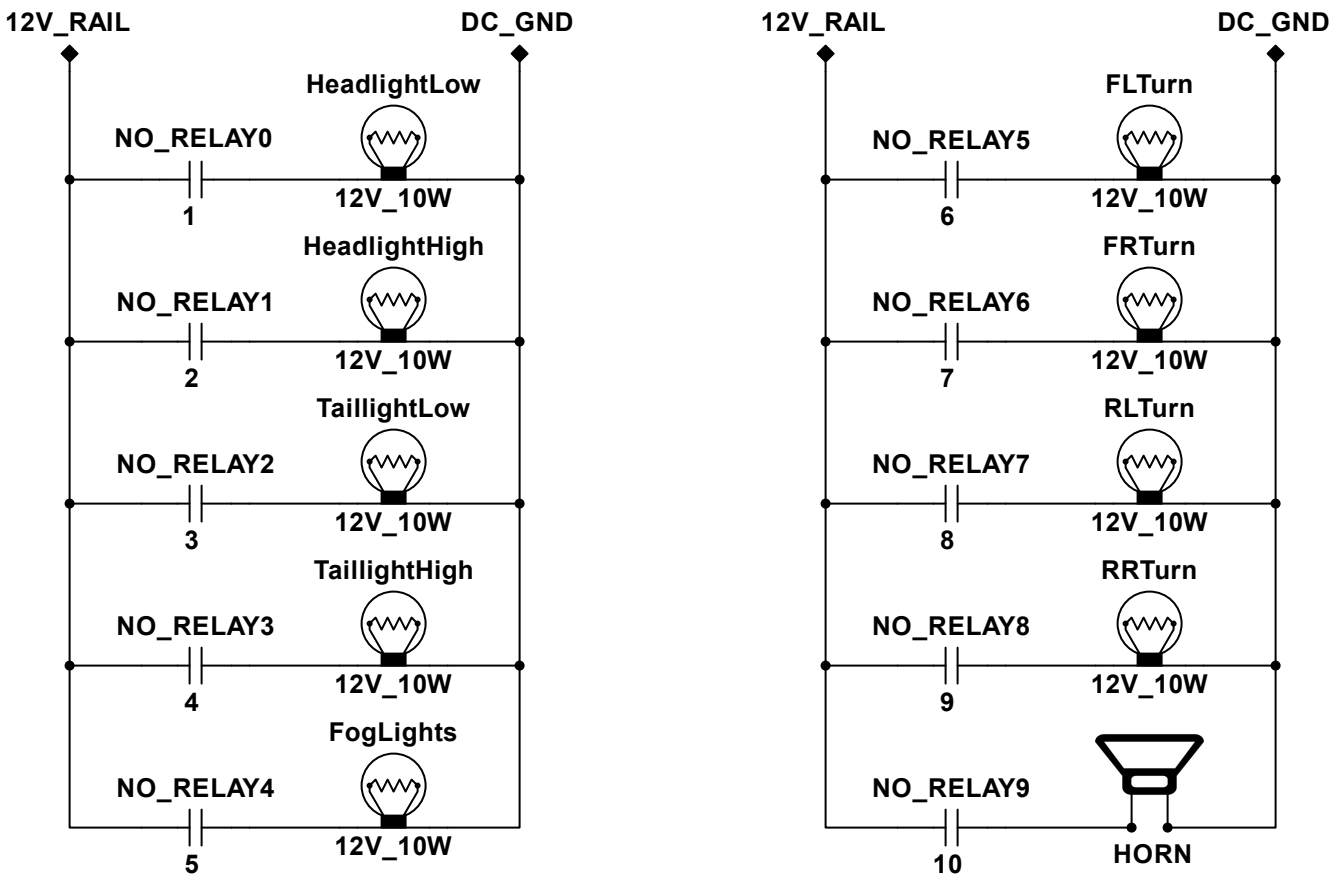




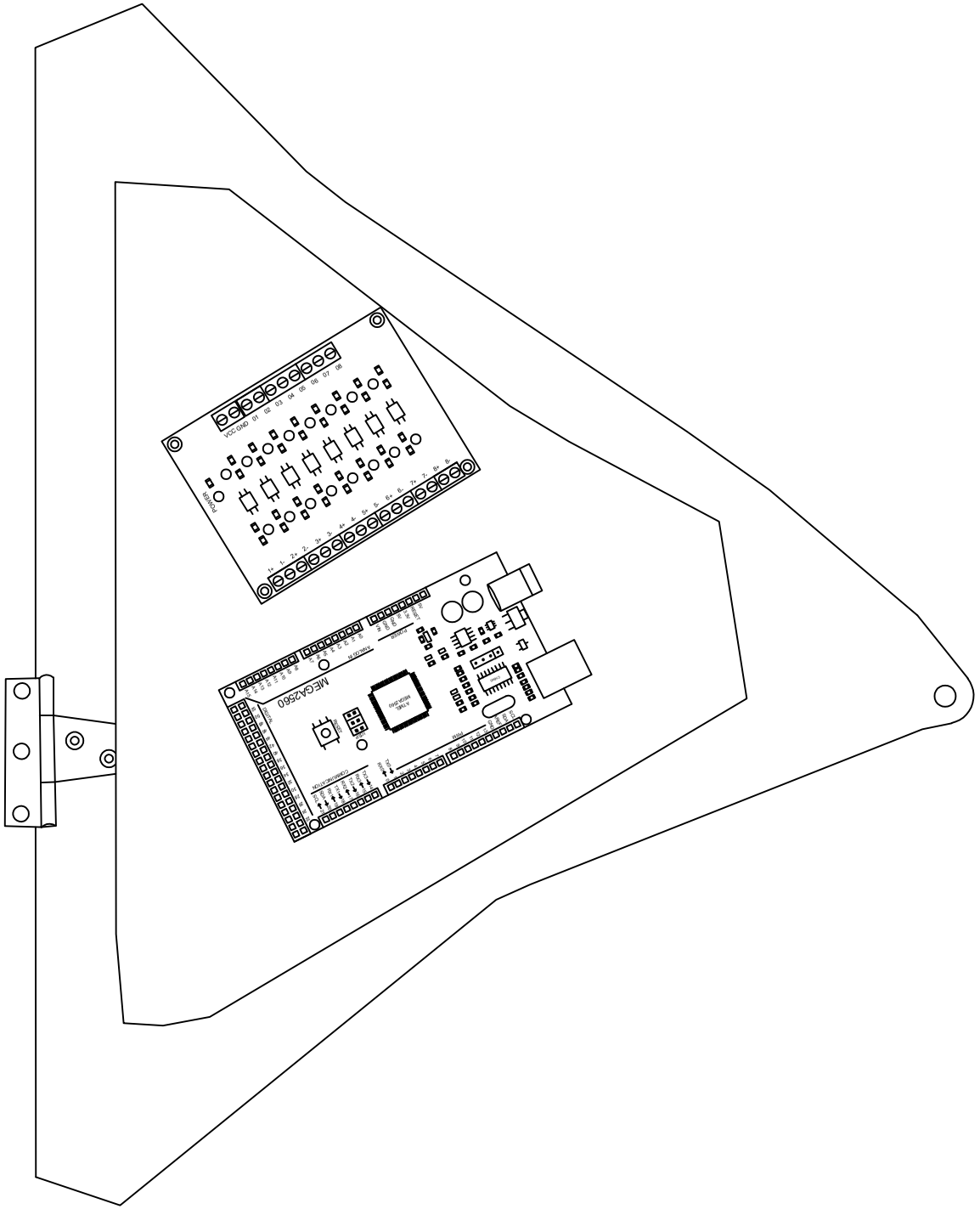




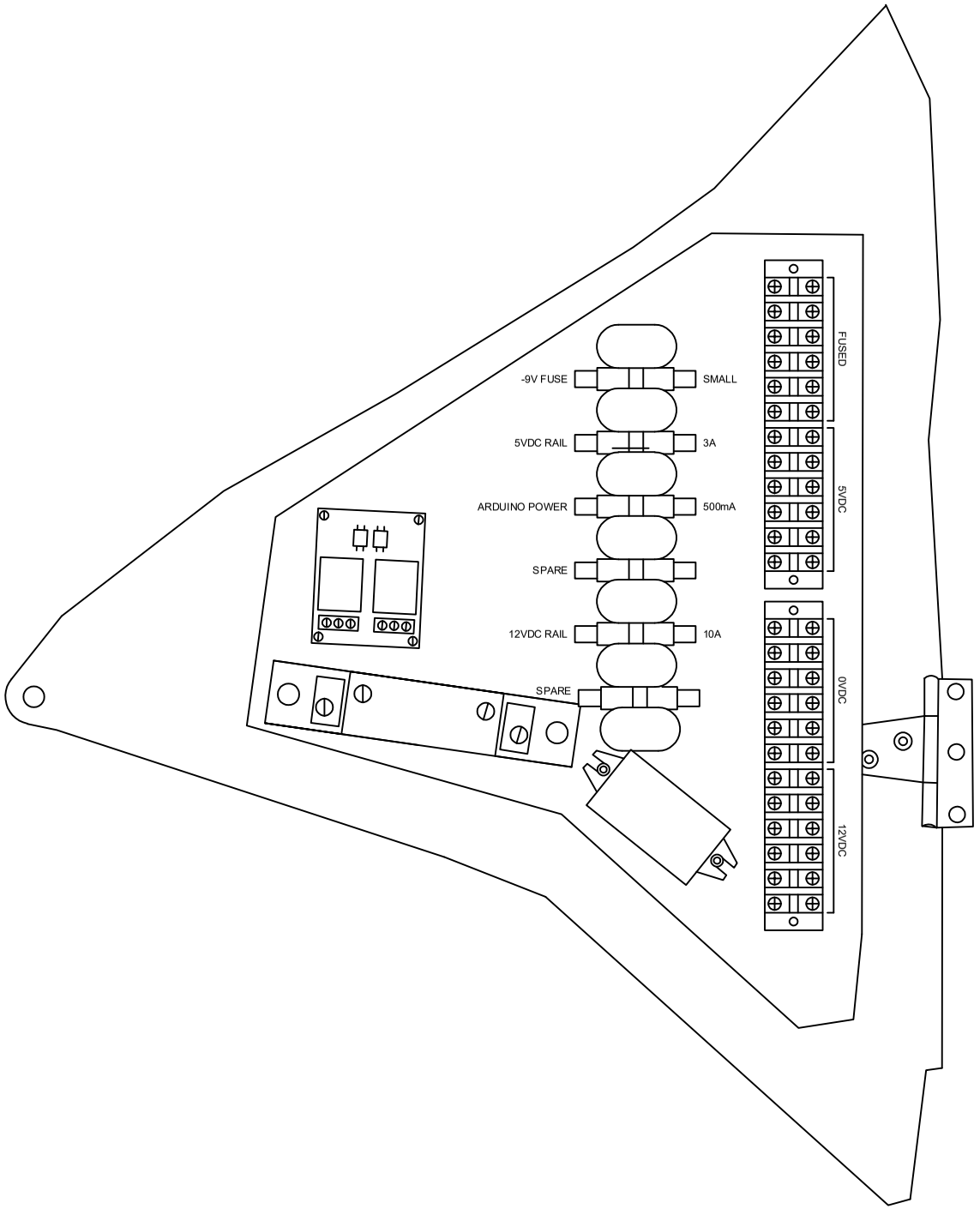
Title: Relay Coils		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 11 of 12	



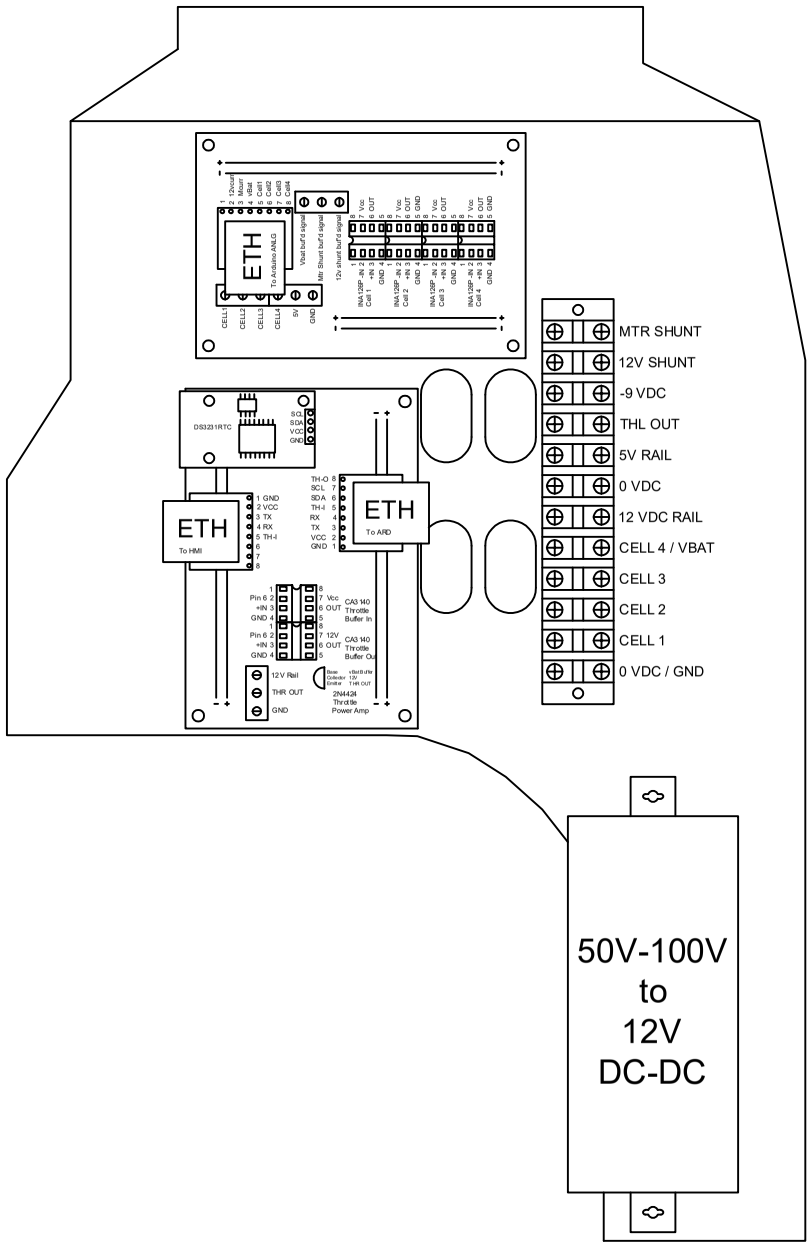
Title: Relay Contacts for Digital Outputs		
Control System for Dirtbike		
Designed by: Cole Fuerth	Document N:	Revision: A
Checked by:	Date: 2020-03-25	Size: A
Approved by:	Sheet 12 of 12	



ST. CLAIR COLLEGE	TITLE LH PANEL LAYOUT	DATE	MAR'20
DRN BY C.C.F / CLASS 001		SCALE	FULL

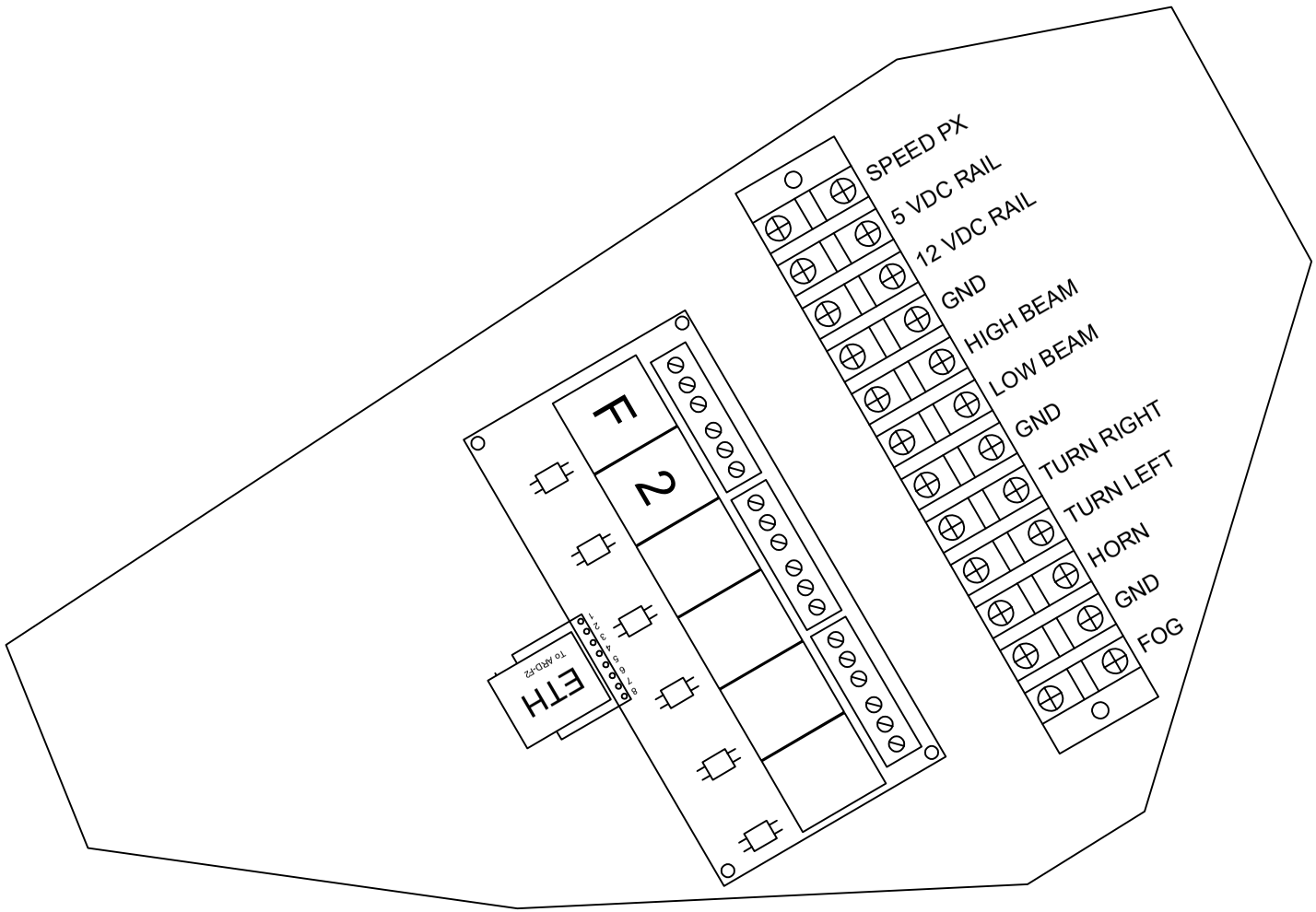


ST. CLAIR COLLEGE	TITLE RH PANEL LAYOUT	DATE	MAR'20
DRN BY C.C.F / CLASS 001		SCALE	FULL

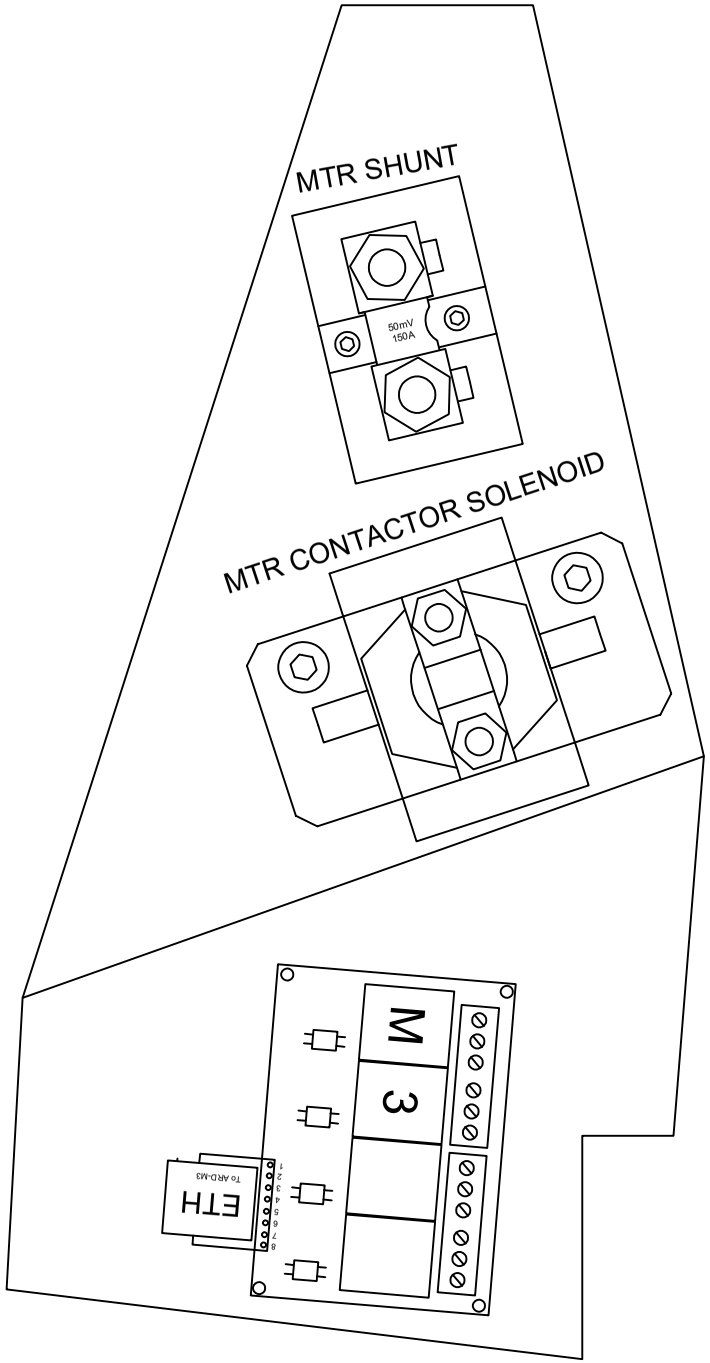


ST. CLAIR COLLEGE		TITLE CENTER PANEL LAYOUT	DATE MAR'20
DRN BY C.C.F	CLASS 001		SCALE FULL

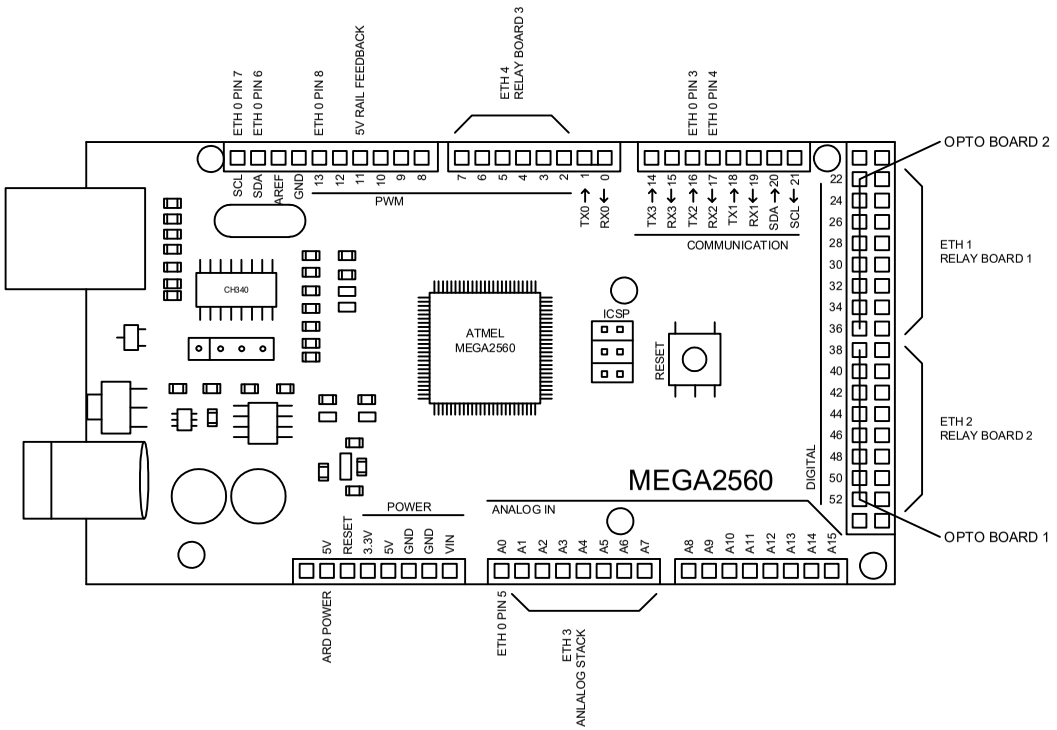
PRODUCED BY AN AUTODESK STUDENT VERSION



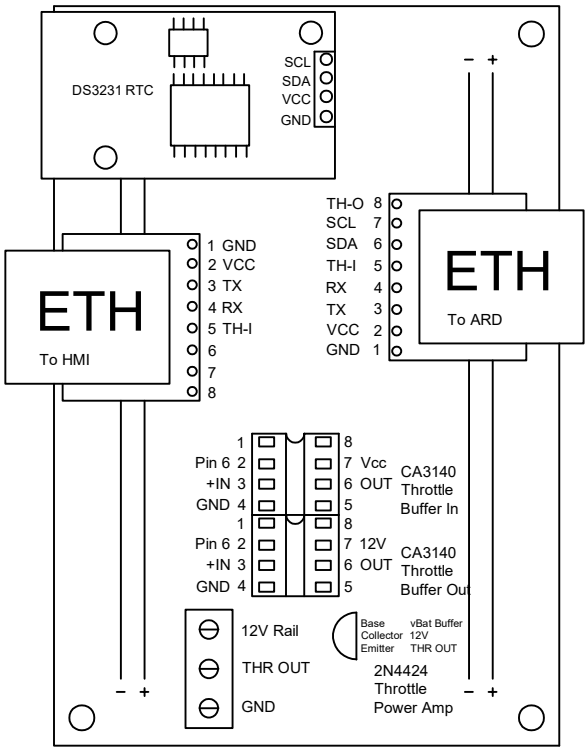
ST. CLAIR COLLEGE	TITLE FRONT PANEL LAYOUT	DATE	MAR'20
DRN BY C.C.F / CLASS 001		SCALE	FULL

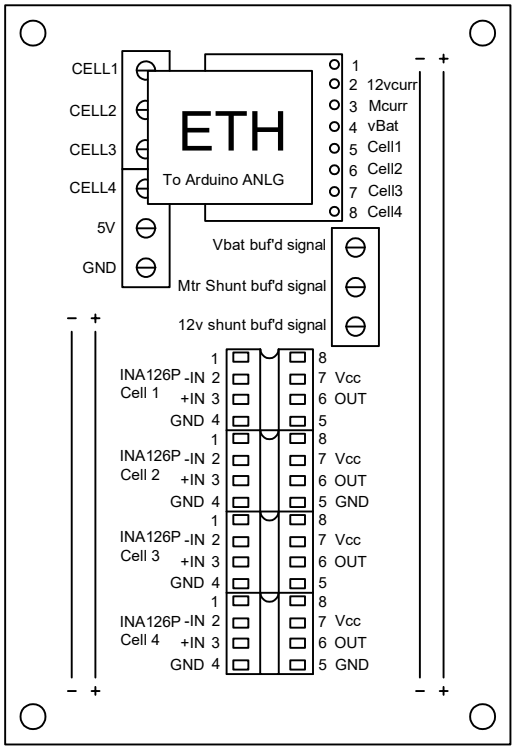


ST. CLAIR COLLEGE		TITLE MOTOR-CONTROLLER INTERFACE PANEL LAYOUT	DATE MAR'20
DRN BY C.C.F	CLASS 001		SCALE FULL



ST. CLAIR COLLEGE	TITLE	DATE	JAN'20
DRN BY C.C.F	ATMEGA 2560 PIN LAYOUT	SCALE	FULL
CLASS 001			

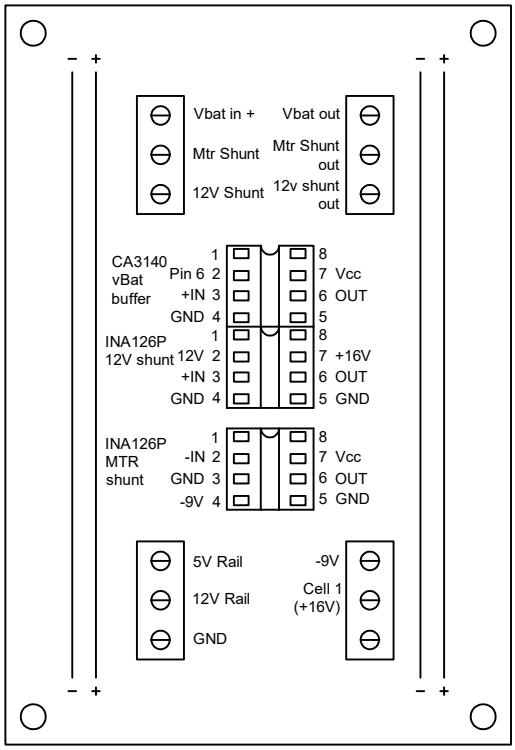


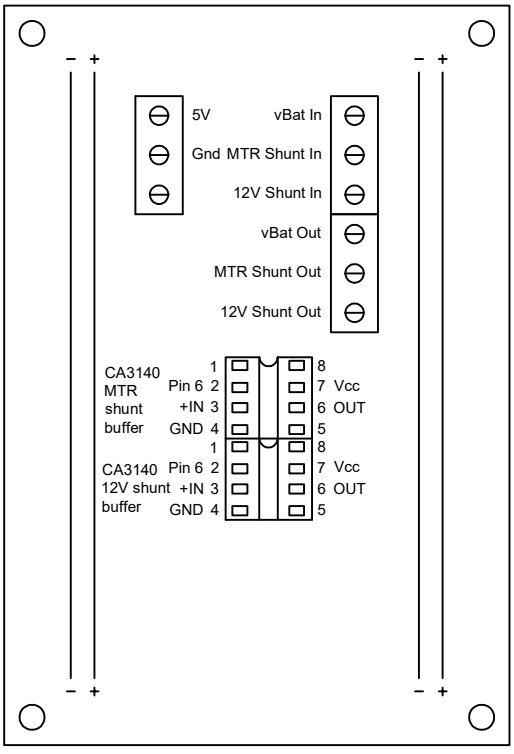


1 4

STACK ORDER,
BOTTOM TO TOP

2 3





PRODUCED BY AN AUTODESK STUDENT VERSION

PRODUCED BY AN AUTODESK STUDENT VERSION

DATE FEB'20
SCALE FULL

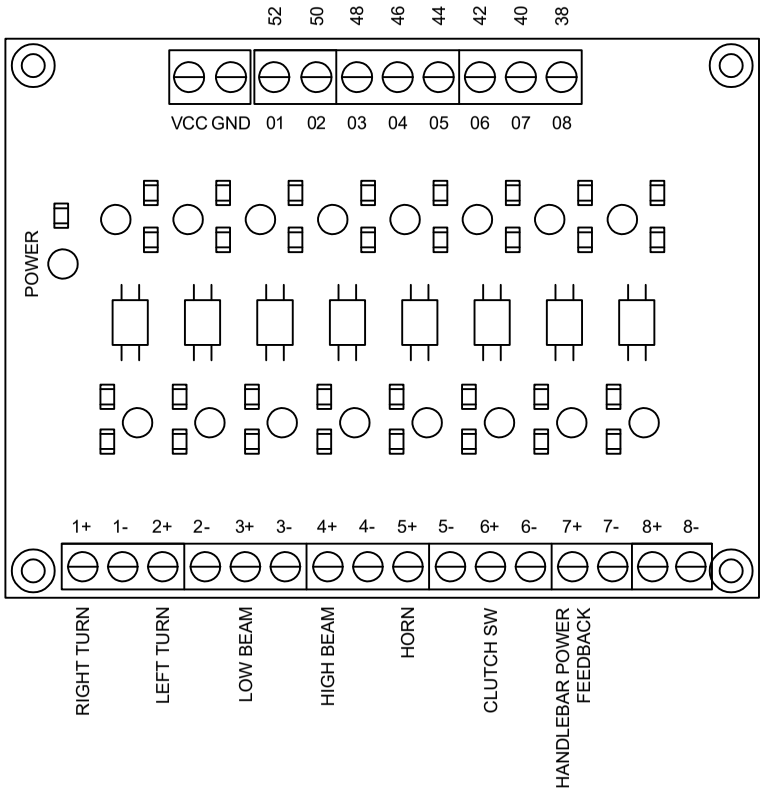
TITLE DESIGNED BOARDS STACK

ST. CLAIR COLLEGE CLASS 001

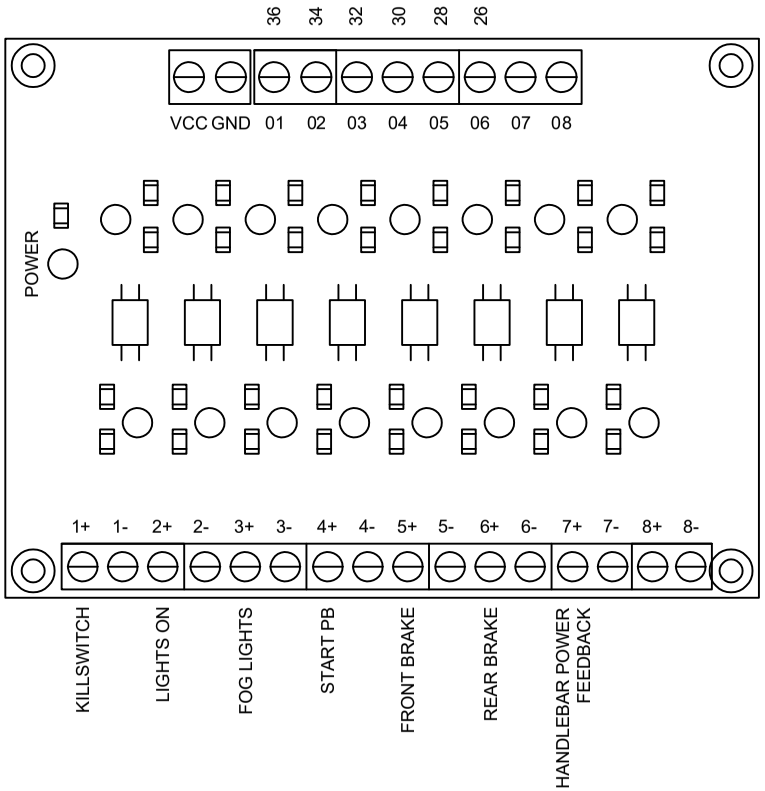
DRN BY C.C.F

PRODUCED BY AN AUTODESK STUDENT VERSION

LEFT HANDLEBAR OPTO BOARD 1

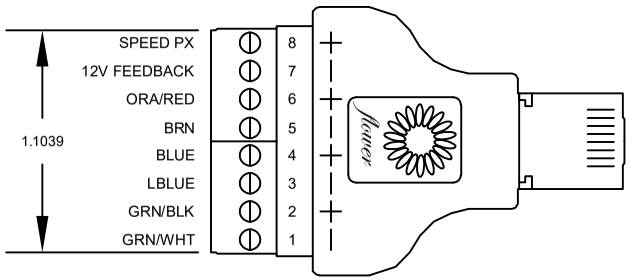


RIGHT HANDLEBAR OPTO BOARD 2

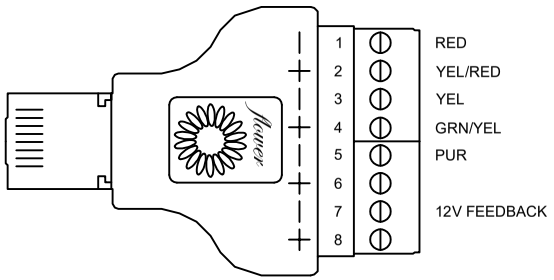


ST. CLAIR COLLEGE		TITLE OPTO BOARD LAYOUT	DATE FEB'20
DRN BY C.C.F	CLASS 001		SCALE FULL

INPUT FROM LEFT HANDLEBAR



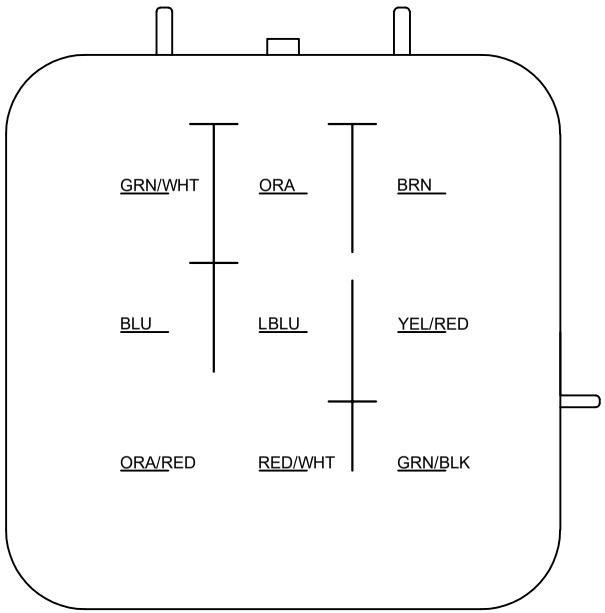
INPUT FROM RIGHT HANDLEBAR



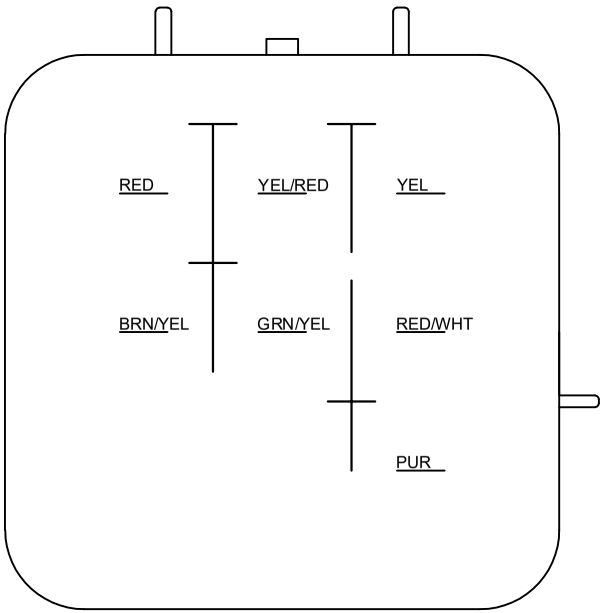
ST. CLAIR COLLEGE		TITLE		DATE	FEB'20
DRN BY	C.C.F	CLASS	001	SCALE	FULL

HANDLEBAR TO ETHERNET PINOUT

LH HAND CONTROLS



RH HAND CONTROLS



COLOR	DESCRIPTION
ORANGE	TURN SIGNAL POWER
GRN/WHT	RIGHT TURN OUTPUT
GRN/BLK	LEFT TURN OUTPUT
LBLUE	OUTPUT TO LOWBEAM
BLUE	OUTPUT TO HIGH BEAM
RED/WHT	POWER FOR HI BEAM TRIGGER / DUAL LIGHT
BRN	HORN OUTPUT
ORANGE/RED	OUTPUT FROM CLUTCH SENSOR
YELLOW/RED	POWER FOR HI/LOW HEADLIGHT

COLOR	DESCRIPTION
RED	KILLSWITCH OUPUT
YELLOW/RED	LIGHTS ON OUTPUT
YELLOW	FOG LIGHTS OUTPUT
BROWN/YELLOW	POWER FOR START PB & KILLSWITCH
GREEN/YELLOW	START PB OUTPUT
RED/WHITE	POWER FOR BRAKE AND FOGLIGHT
PURPLE	FRONT BRAKE OUTPUT

ST. CLAIR COLLEGE		TITLE HANDLEBAR PINOUT	DATE JAN'20
DRN BY C.C.F	CLASS 001		SCALE FULL

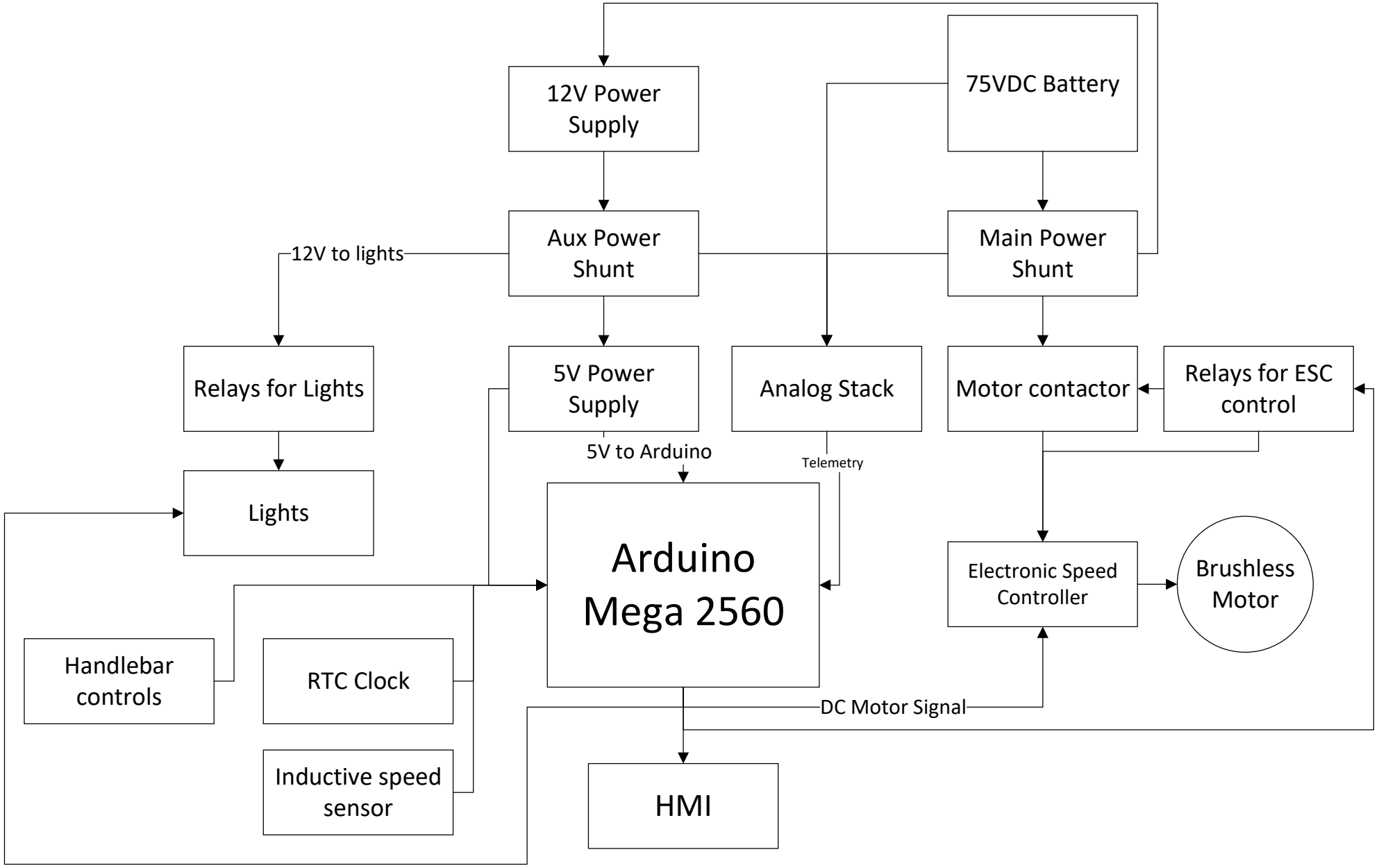
Bill of Materials

Control System for Electric Motorcycle

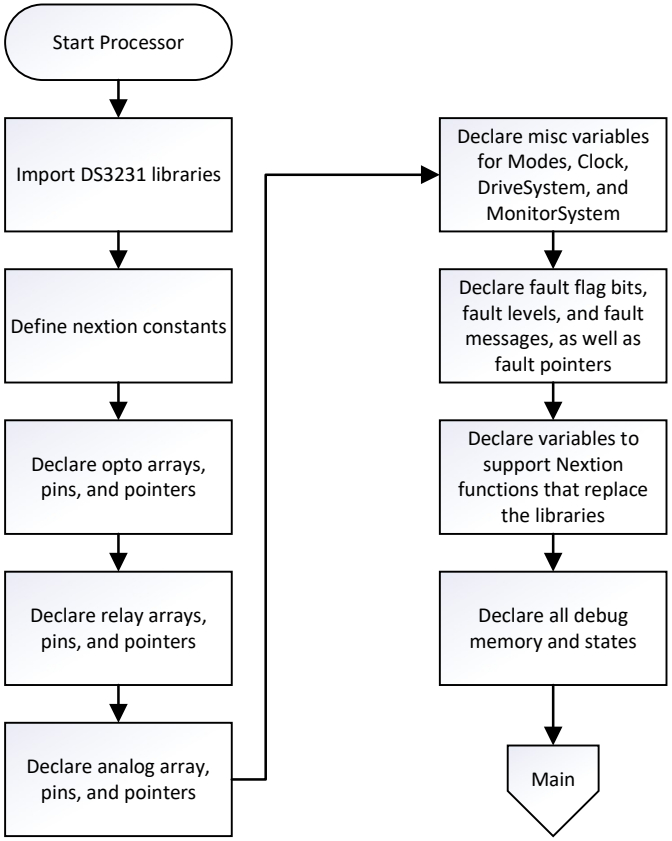
Cole Fuerth

Description	Qty	Price per unit	Subtotal	Supplier
YZ125 Dirtbike (Used)	1	$ 360.00	$ 360.00	Facebook Marketplace
324 18650 batteries, 4s packs	1	$ 200.00	$ 200.00	Facebook Marketplace
15kW 100kV Brushless Motor	1	$ 560.00	$ 560.00	Alien Power Systems
9kW 60V EV Controller	1	$ 550.00	$ 550.00	Alien Power Systems
Hall Effect Throttle	1	$ 10.00	$ 10.00	Alien Power Systems
150A shunt	1	$ 30.00	$ 30.00	Digikey
Handlebar Controls (L&R)	1	$ 23.54	$ 23.54	AliExpress
Fog Lights (2)	2	$ 22.66	$ 45.32	AliExpress
Turn Signals (4)	2	$ 7.98	$ 15.96	AliExpress
Arduino Mega	2	$ 8.32	$ 16.64	AliExpress
Nextion 3.2"	1	$ 46.99	$ 46.99	AliExpress
RTC	2	$ 0.93	$ 1.86	AliExpress
12V-5V 3A regulator	2	$ 1.44	$ 2.88	AliExpress
15A Shunt	1	$ 3.02	$ 3.02	AliExpress
18AWG Assorted Wire	1	$ 15.20	$ 15.20	AliExpress
sdcard reader	1	$ 0.74	$ 0.74	AliExpress
Arduino Mega Screw Terminal Shield	1	$ 14.00	$ 14.00	AliExpress
M3 Hardware Kit	1		$ -	AliExpress
10x xt60 connectors	1	$ 2.54	$ 2.54	AliExpress
8CH 12V to 5V OPTO Board	2	$ 14.59	$ 29.18	Banggood
30pcs PCB Screw Terminals	1	$ 6.43	$ 6.43	Banggood
inductive proximity sensor	3	$ 3.00	$ 9.00	Banggood
INA126P Instrumentation Amp	8	$ 4.80	$ 38.40	Digikey
CA3140 Opamp	6	$ 3.82	$ 22.92	Digikey
Fast blow Fuse Kit	1	$ 13.98	$ 13.98	Amazon
6pcs Solderable 1/2 Breadboard	1	$ 16.99	$ 16.99	Amazon
RJ45 Jack/Breakout Board	13	$ 2.86	$ 37.18	Digikey
2N4424 NPN Transistor	4	$ 0.56	$ 2.24	Digikey
5*12 Pos terminal blocks	1	$ 10.50	$ 10.50	Amazon
50 ft 18awg blue wire	1	$ 21.31	$ 21.31	Amazon
RJ45 Screw Terminal	1	$ 12.99	$ 12.99	Amazon
22AWG Assorted Wire	1	$ 21.00	$ 21.00	Amazon
HC-05 Bluetooth serial device	1	$ 10.99	$ 10.99	Amazon
20ft 10AWG red/blk wire	1	$ 21.97	$ 21.97	Amazon
10 pair XT90	1	$ 26.12	$ 26.12	Amazon
800pcs Assorted Heatshrink	1	$ 16.99	$ 16.99	Amazon
12V horn	1	$ 10.10	$ 10.10	Amazon
10 pair XT60	1	$ 23.10	$ 23.10	Amazon
10ft 12 awg wire red/blk	1	$ 11.97	$ 11.97	Amazon

12V 500A Solenoid	1	$ 36.99	$ 36.99	Amazon
10' door stop foam tape	2	$ 10.82	$ 21.64	Amazon
18AWG Assorted Wire	1	$ 23.88	$ 23.88	Amazon
12/24VDC to 5V 10A converter	1	$ 22.99	$ 22.99	Amazon
8 Channel relay	8	$ 12.86	$ 102.88	Amazon
10x xt30 connectors	1	$ 11.19	$ 11.19	Amazon
Arduino Mega 16u2 interface	2	$ 10.60	$ 21.20	Banggood
Arduino Nano	3	$ 2.24	$ 6.72	Banggood
2x replacement turn signals	1	$ 13.80	$ 13.80	Banggood
Taillight	1	$ 11.92	$ 11.92	Banggood
Headlight	1	$ 40.21	$ 40.21	Banggood
65ft Cat5	2	$ 6.89	$ 13.78	Banggood
50x RJ45 rubber boots	1	$ 7.60	$ 7.60	Banggood
50x RJ45 Plug	1	$ 4.52	$ 4.52	Banggood
4-pin CB connector pair	8	$ 2.31	$ 18.48	Banggood
RJ45 Screw Terminal Male	2	$ 5.89	$ 11.78	Banggood
Cat5 Crimp tool	1	$ 16.08	$ 16.08	Banggood
Subtotal of Bike Components			$ 1,680.00	
Subtotal of Capstone Components			$ 967.71	
Grand Total			$ 2,647.71	

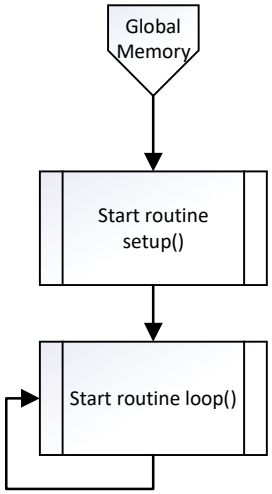


TITLE	DRAWN BY	DATE	PAGE
Block Diagram	COLE FUERTH	11/26/2019	1 OF 1



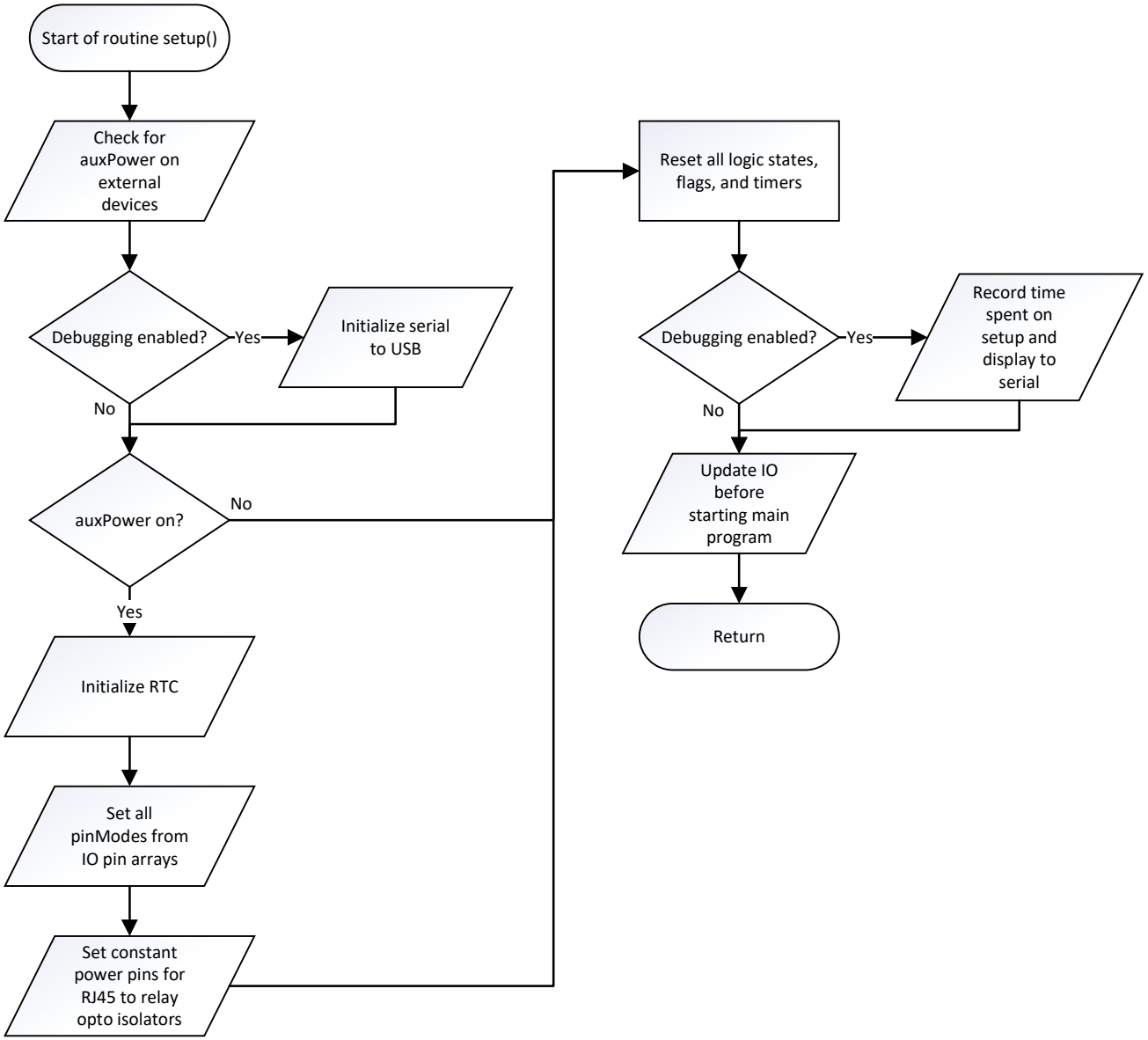
Initialize all variables and memory

TITLE	DRAWN BY	DATE	PAGE
Global Memory	COLE FUERTH	03/20/2020	1 OF 12



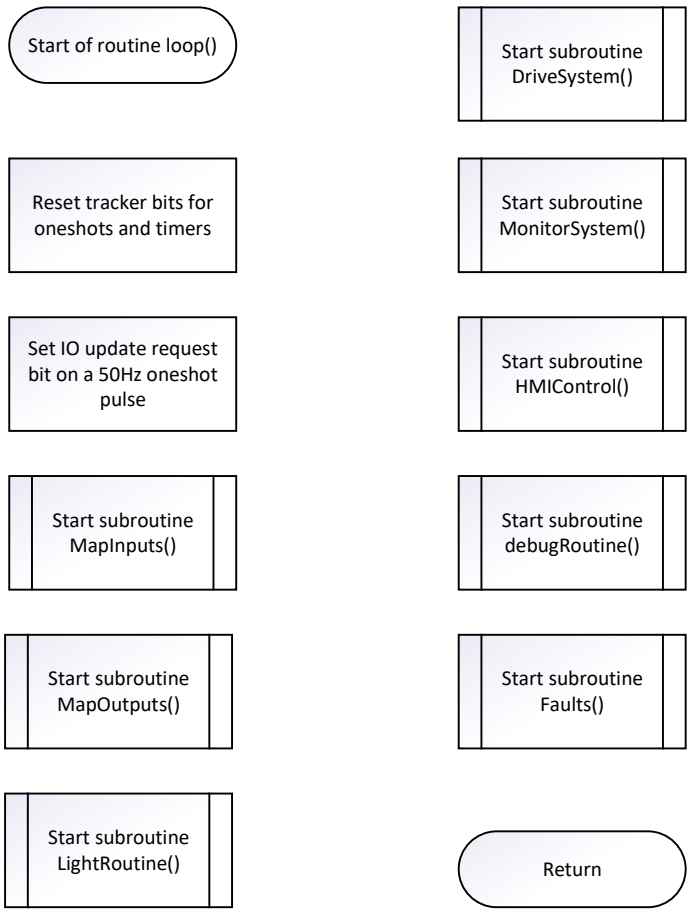
Core Arduino flow

TITLE	DRAWN BY	DATE	PAGE
Main Program Flow	COLE FUERTH	03/20/2020	2 OF 12



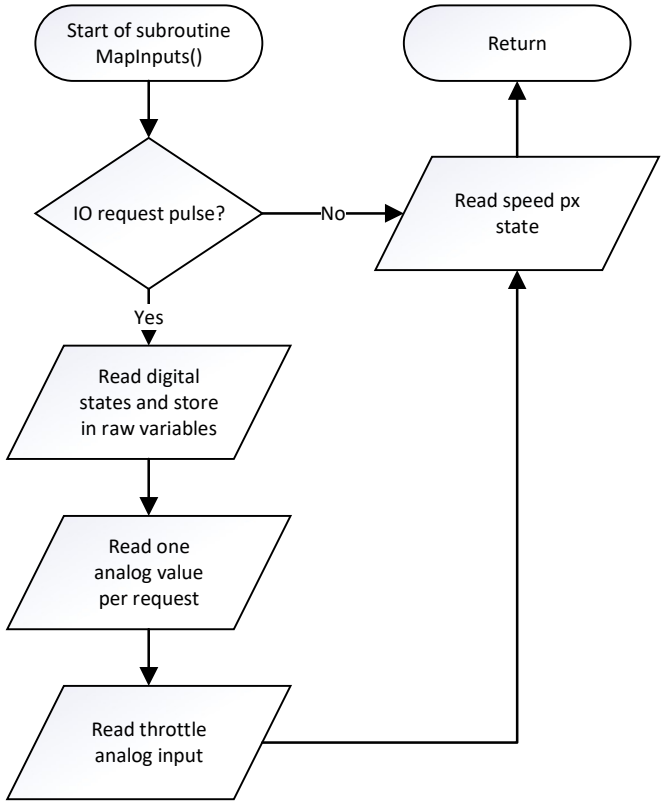
Complete setup of all initial logic states, IO, and serial connections

TITLE	DRAWN BY	DATE	PAGE
Main Program Flow	COLE FUERTH	03/20/2020	3 OF 12



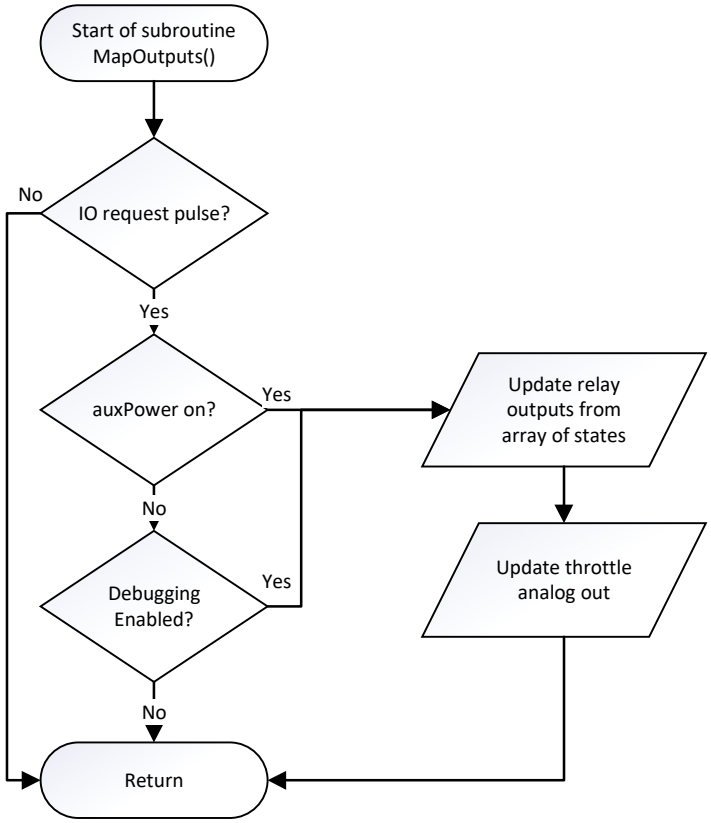
Loop through main program

TITLE	DRAWN BY	DATE	PAGE
Main Program Flow	COLE FUERTH	03/20/2020	4 OF 12



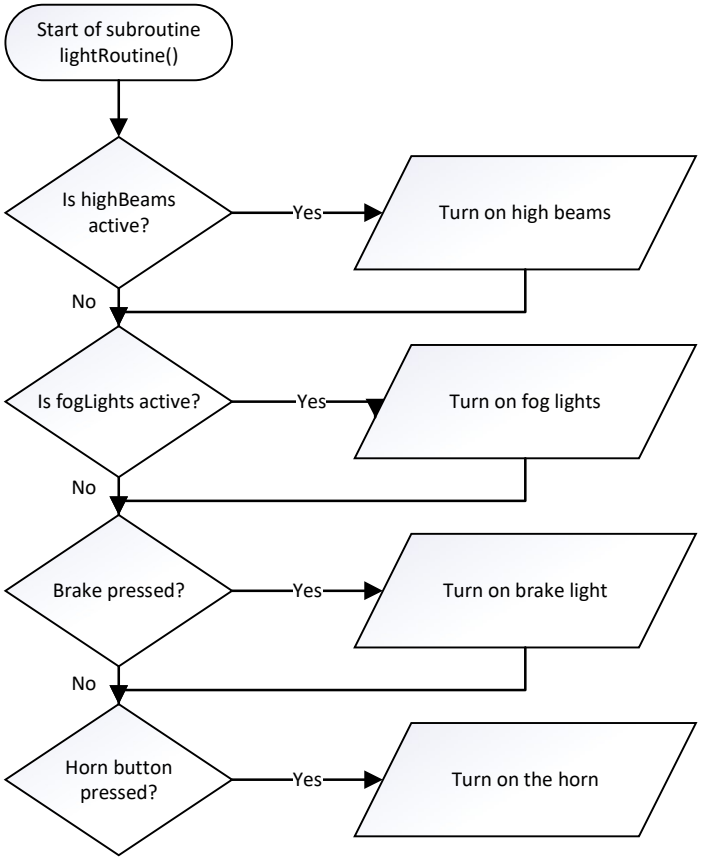
Map & Input signals and states from pins and store inputs in arrays

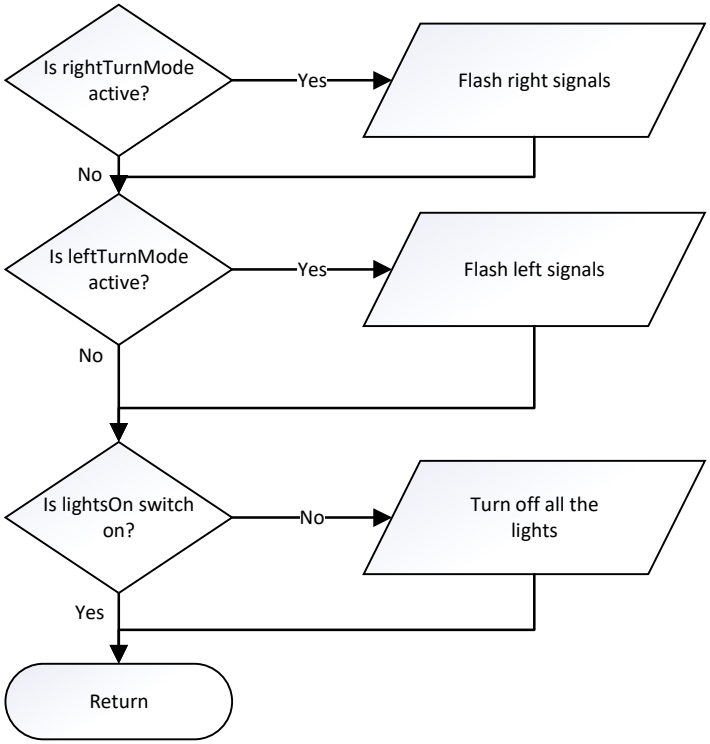
TITLE	DRAWN BY	DATE	PAGE
Subroutine MapInputs()	COLE FUERTH	03/20/2020	5 OF 12



Use arrays for bits and pins to write throttle value and relay states

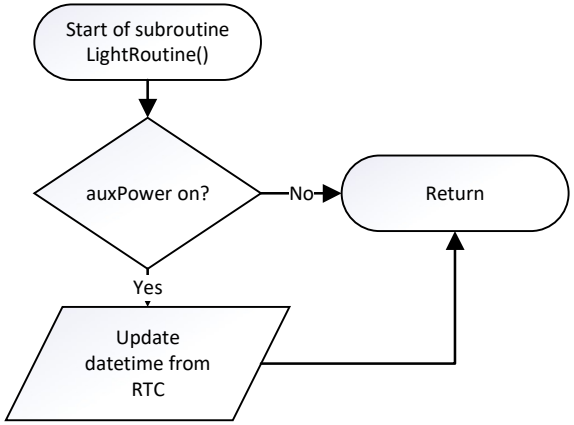
TITLE	DRAWN BY	DATE	PAGE
Subroutine MapOutputs()	COLE FUERTH	03/20/2020	6 OF 12





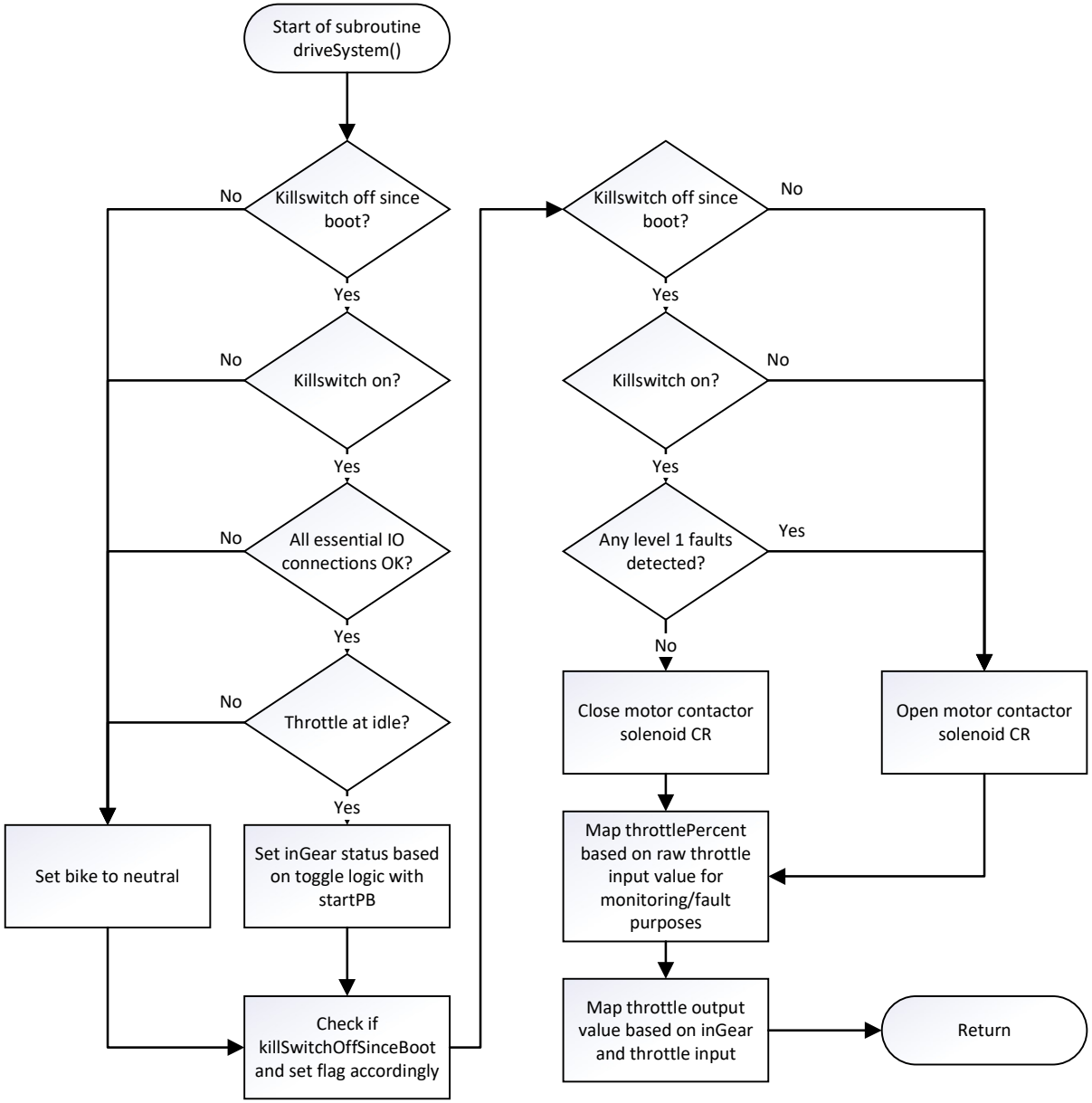
All bits used to control individual light relays are conditioned here

TITLE	DRAWN BY	DATE	PAGE
Subroutine LightRoutine()	COLE FUERTH	03/20/2020	7 OF 12



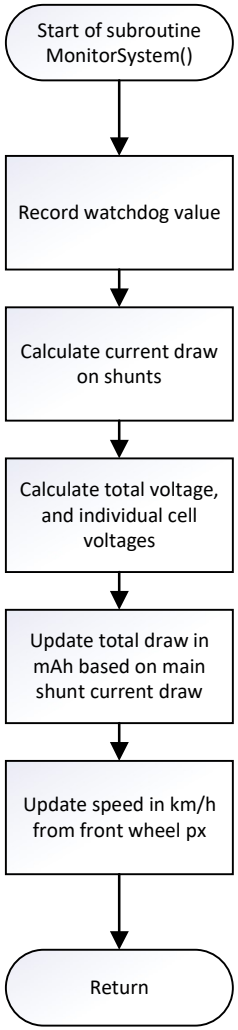
Update clock value based on RTC

TITLE	DRAWN BY	DATE	PAGE
Subroutine LightRoutine()	COLE FUERTH	03/20/2020	8 OF 12



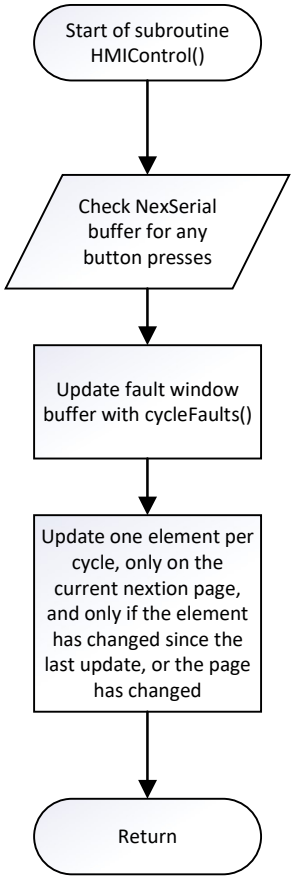
User throttle input and states from Modes() are used to generate a PWM signal for the ESC

TITLE	DRAWN BY	DATE	PAGE
Subroutine DriveSystem()	COLE FUERTH	03/20/2020	9 OF 12



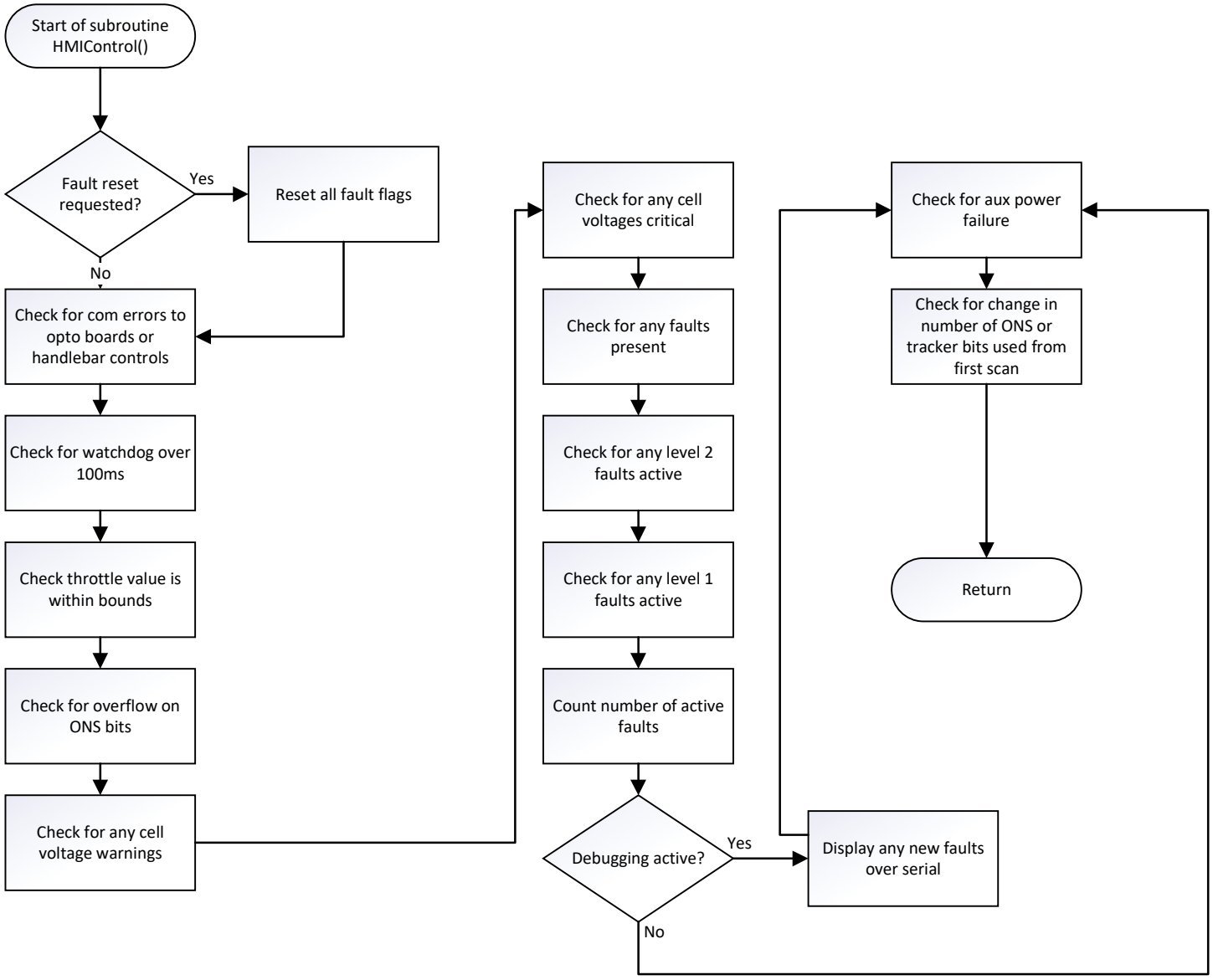
Use inputs from analog sensors to read and record telemetry from the system, including Vbat, current draw, and efficiency.

TITLE	DRAWN BY	DATE	PAGE
Subroutine MonitorSystem()	COLE FUERTH	03/20/2020	10 OF 12



Update Nextion elements based on telemetry gathered

TITLE	DRAWN BY	DATE	PAGE
Subroutine HMIControl()	COLE FUERTH	03/20/2020	11 OF 12



Set fault flags based on current machine status

TITLE	DRAWN BY	DATE	PAGE
Subroutine Faults()	COLE FUERTH	03/20/2020	12 OF 12

```
1  /*
2   * ElectricMotorcycleMainProcessor.ino
3   *
4   * Created: 1/30/2020 1:39:49 PM
5   * Author: cole_fuerth
6   * Revision: Beta 0.6.0
7   * Project: Control System for an Electric Motorcycle
8   *
9   * THIS SKETCH IS DESIGNED TO BE RUN ON AN ATMEGA 2560
10  *
11  */
12
13
14 #include <Wire.h>       // SCL SDA library for clock
15 #include <DS3231.h>     // DS3231 RTC library
16 //#include <Nextion.h>  // I have my own logic for this now
17
18 //define some color values for Nextion
19 #define BLACK   0x0000
20 #define BLUE    0x001F
21 #define RED     0xF800
22 #define GREEN   0x07E0
23 #define CYAN    0x07FF
24 #define LBLUE   0xAEBF
25 #define MAGENTA 0xF81F
26 #define YELLOW  0xFFE0
27 #define WHITE   0xFFFF
28 #define NEX_RET_EVENT_TOUCH_HEAD 0x65
29 #define nexSerial Serial2
30
31 // function call memory bits available
32     boolean oneShotBits[64];            // oneshot bits available for use for
       oneshot or toggle calls
33                                         // BITS 32-63 ARE FOR FAULTS ONLY!!
34     boolean toggledMem[32];             // memory bits for previous condition of
          toggled bit
35                                         // used with oneShotBits
36     uint8_t ONSTracker;
37
38     boolean timerInSession[32];     // for speed, so we only update timer timers
          when needed
39     boolean timerMemory[sizeof(timerInSession)];            // make function
       calls smaller by remembering previous output state
40     unsigned long timerTimers[sizeof(timerInSession)];  // debounce timers
       available for use
41     uint8_t timerTracker;
42
43 // declare opto input arrays and input pins
44     boolean opto1[8], opto2[8];
45     const uint8_t opto1pins[8] = {52,50,48,46,44,42,40,38};
46     const uint8_t opto2pins[8] = {36,34,32,30,28,26,24,22};
47
```

```
48      boolean *killswitch             = opto2 ,
49              *fogLightsIn            = opto2 + 1,
50              *lightsOn               = opto2 + 2,
51              *startPb                = opto2 + 3,
52              *frontBrake             = opto2 + 4,
53              *rearBrake              = opto2 + 5,
54              *handlebarPowerRight    = opto2 + 6;
55
56      boolean *rightTurnInput         = opto1 ,
57              *leftTurnInput          = opto1 + 1,
58              *lowBeamInput           = opto1 + 2,
59              *highBeamInput          = opto1 + 3,
60              *hornInput              = opto1 + 4,
61              *handlebarPowerLeft     = opto1 + 6,
62              *optoPower              = opto1 + 7;
63
64
65      boolean speedoPXInput;
66      uint8_t speedoPXInputPin        = 10;
67
68
69  // declare analog inputs and pins
70      const uint8_t auxPowerPin = 11;
71      uint8_t analogInputTracker = 1;     // refresh one at a time, at 25Hz,
          except A0 is read at 25Hz on its own
72      const uint8_t analogInputPins[8] =
          {PIN_A0,PIN_A1,PIN_A2,PIN_A3,PIN_A4,PIN_A5,PIN_A6,PIN_A7};
73      int analogInputs[8];
74      int *throttleInput          = analogInputs,
75          *currentDrawInput       = analogInputs + 1,
76          *motorDrawInput         = analogInputs + 2,
77          *vBatInput              = analogInputs + 3,
78          *cellVoltageRawIn[4]    = { analogInputs+4, analogInputs+5,
79                                      analogInputs+6,analogInputs+7 };
80
81  // declare relay arrays and pointers
82      boolean relayR1[6] = {0,0,0,0,0,0};
83      boolean relayF2[6] = {0,0,0,0,0,0};
84      boolean relayM3[4] = {0,0,0,0};
85      /*  note that the pins before and after each array are used for
86          power and ground for optocouplers on the relay inputs   */
87      const uint8_t relayR1Pins[6] = {41,43,45,47,49,51};
88      const uint8_t relayF2Pins[6] = {35,33,31,29,27,25};
89      const uint8_t relayM3Pins[4] = {3,4,5,6};
90
91      boolean *brakeLowOutput     = relayR1 + 0,
92              *brakeHighOutput    = relayR1 + 1,
93              *RRTurn             = relayR1 + 2,
94              *RLTurn             = relayR1 + 3;
95
96      boolean *highBeamsOut       = relayF2 + 0,
97              *lowBeamsOut        = relayF2 + 1,
```

```
 98              *FRTurn             = relayF2 + 2,
 99              *FLTurn             = relayF2 + 3,
100              *hornOutput         = relayF2 + 4,
101              *fogLightsOut       = relayF2 + 5;
102
103     boolean *ESCSolenoid         = relayM3 + 0,
104              *speedHighOut       = relayM3 + 1,
105              *speedLowOut        = relayM3 + 2;
106
107
108 // Modes
109     boolean inGear;
110
111 // clockRoutine Variables
112     RTClib RTC;
113     DateTime now;
114
115 // DriveSystem variables
116     int throttlePercent;
117     boolean killswOffSinceBoot = 0;
118
119 // monitor system variables
120     long last_cycleStart, totalDraw_mAh, last_analogCycleComplete,
          last_ioUpdate, last_cycleStartus, lastPXTime, thisPXTime;
121     int watchdog, longestCycle = 0, fastestCycle = 1000, watchdogus,
          currentDrawMotorAmps, currentDrawShuntmA, speed;
122     float cellVolts[4], vBat;
123
124
125 // misc IO Variables, not done in the analog block
126     int throttleOutput;
127     const uint8_t throttleOutputPin = 13;
128
129
130 // FAULTS
131     boolean faultFlags[32];
132     boolean faultReset = 0;
133     const uint8_t faultLevel[32] = {
134         3,  // 0
135         1,  // 1
136         1,  // 2
137         1,  // 3
138         1,  // 4
139         1,  // 5
140         2,  // 6
141         1,  // 7
142         2,  // 8
143         2,  // 9
144         2,  // 10
145         2,  // 11
146         2,  // 12
147         1,  // 13
```

```
148         1,  // 14
149         3,  // 15
150         2,  // 16
151         1,  // 17
152         2,  // 18
153         1,  // 19
154         1,  // 20
155         0,  // 21
156         0,  // 22
157         1,  // 23
158         1,  // 24
159         2,  // 25
160         2,  // 26
161         0,  // 27
162         0,  // 28
163         0,  // 29
164         0,  // 30
165         0,  // 31
166     };  // 1=critical   2=warning   3=status    0=unused
167     const String faultMessages[32] = {
168         "Fault Detected! See below for error message:", // 0
169         "Lost connection to an essential component!",   // 1
170         "Processor cycle overtime!",    // 2
171         "Opto board power loss detected",   // 3
172         "Left handlebar lost connection to controller!",    // 4
173         "Right handlebar lost connection to controller!",   // 5
174         "Throttle reading out of bounds detected!", // 6
175         "Throttle reading out of bounds detected!", // 7
176         "Cell 1 Low!",  // 8
177         "Cell 2 Low!",  // 9
178         "Cell 3 Low!",  // 10
179         "Cell 4 Low!",  // 11
180         "At least one warning detected!",   // 12
181         "Problem detected with the motor contactor!",   // 13
182         "Function tracking bits overloaded!",   // 14
183         "Throttle must be at idle to enter gear",   // 15
184         "High current detected at 12V shunt (>12A)",    // 16
185         "Critical fault detected!", // 17
186         "EEPROM values unavailable; cannot record odometer",    // 18
187         "Battery cell at critical level; please stop riding",   // 19       50   ↵
              in length, for reference (faults cannot be longer than 50 in length
188         "Throttle out feedback error",  // 20
189         "", // 21
190         "", // 22
191         "Unexpected timerTracker value detected!!", // 23
192         "Unexpected ONSTracker value detected!!",   // 24
193         "Aux power is off; IO will not work correctly. ",   // 25
194         "HMI overtime, disabling Nextion",  // 26
195         "", // 27
196         "", // 28
197         "", // 29
198         "", // 30
```

```
199         "", // 31
200     };
201     int overTimeLength;
202     int expectedONS, expectedTimer;
203     int numberOfFaults;
204 
205     boolean *anyFaultsDetected          = faultFlags,
206             *allConnectionsOK           = faultFlags + 1,
207             *watchdogFlag               = faultFlags + 2,
208             *optoPowerLost              = faultFlags + 3,
209             *handlebarPowerLeftFault    = faultFlags + 4,
210             *handlebarPowerRightFault   = faultFlags + 5,
211             //*throttleOutOfBoundsFlag  = faultFlags + 6,
212             *throttleOutOfBounds        = faultFlags + 7,
213             *cellOneLow                 = faultFlags + 8,
214             *cellTwoLow                 = faultFlags + 9,
215             *cellThreeLow               = faultFlags + 10,
216             *cellFourLow                = faultFlags + 11,
217             *anyLevel2FaultDetected     = faultFlags + 12,
218             *motorContactorFault        = faultFlags + 13,
219             *trackerBitsOverload        = faultFlags + 14,
220             *thlNotIdleWhenReq          = faultFlags + 15,
221             *controlOvercurrentFlag     = faultFlags + 16,
222             *anyLevel1FaultDetected     = faultFlags + 17,
223             *unableToLoadEEPROM         = faultFlags + 18,
224             *anyCellCritical            = faultFlags + 19,
225             *throttleFeedbackFault      = faultFlags + 20,
226             *unexpectedTimer            = faultFlags + 23,
227             *unexpectedONS              = faultFlags + 24,
228             *auxPowerFault              = faultFlags + 25,
229             *hmiOvertimeFault           = faultFlags + 26;
230 
231 
232     // Nextion Variables for my functions
233     int nextionPage = 0;        // current page of nextion to refresh;          ↵
          functionality being added later
234     int activeFaultToDisplay = 0;
235     int hmiElemToUpd = 0;
236     String nexBuffer[30];
237     boolean nextionDelay = 0;
238     uint8_t nexBytesRead[3];
239 
240     // DEBUG MEMORY
241     boolean nextionEnabled = 1;
242     boolean faultsActive = 1;
243     boolean debuggingActive = 1;
244     boolean debuggingHMIActive = 0;
245     boolean auxPower;
246     boolean firstScan = 1;
247     boolean contactorSafetiesActive = 0;
248     boolean driveSystemDebuggingActive = 0;
249     boolean analogDebuggingActive = 0;
```

```
250     long debugTimer;
251     int hmiOvertimeLength;
252
253
254 // MAIN PROGRAM
255 void setup()
256 {
257     int setupTime = millis();
258
259     pinMode(auxPowerPin, INPUT);
260     auxPower = digitalRead(auxPowerPin);
261
262     // SETUP SERIAL FUNCTIONS
263     if (debuggingActive)
264     {
265         Serial.begin(115200);
266         while (!TON(1, 200, 31) || !Serial) delay(10);  // wait 200ms for serial
              debugging to initialize
267         //USBActive = Serial;
268     }
269
270     // setup Nextion
271     if (nextionEnabled)
272     {
273         Serial.print("Initializing Nextion...");
274         nexCustomSerial(115200);    // Nextion is initially 9600 baud,
275                                     // nexCustomSerial adjusts the baud of the
                    Nextion, as well as the Nextion serial port
276         Serial.println("Done.");
277     }
278     else Serial.println("Nextion is disabled in settings.");
279
280
281     if (auxPower)
282     {
283         // setup RTC
284         Serial.print("initial read of DS3231...");
285         now = RTC.now();                // update RTC value before starting
              Serial.println
286         Serial.println("Done.");
287         Serial.println( "Current time is " + String(now.year()) + String
              (now.month()) + String(now.day()) + " " + String(now.hour()) + ":" +
              String(now.minute()) + ":" + String(now.second()) );
288     }
289     else Serial.println("No 5V aux power detected; skipping DS3231");
290
291
292     // declare all pinModes
293
294     if (debuggingActive) Serial.print("setting up opto inputs...");
295     // opto inputs
296     for (uint8_t i=0; i<sizeof(opto1pins); i++) pinMode(opto1pins[i], INPUT);
```

```
297     for (uint8_t i=0; i<sizeof(opto2pins); i++) pinMode(opto2pins[i], INPUT);
298     if (debuggingActive) Serial.println("Done.");
299
300
301     if (debuggingActive) Serial.print("setting up relay outputs...");
302     // relay outputs (R1 and R2):
303     for (uint8_t i=0; i<sizeof(relayR1Pins); i++) pinMode(relayR1Pins[i],
          OUTPUT);
304     for (uint8_t i=0; i<sizeof(relayF2Pins); i++) pinMode(relayF2Pins[i],
          OUTPUT);
305     for (uint8_t i=0; i<sizeof(relayM3Pins); i++) pinMode(relayM3Pins[i],
          OUTPUT);
306
307     // ensure the pulse to all relays so longer happens on startup
308     for (uint8_t i=0; i<sizeof(relayR1); i++) digitalWrite(relayR1Pins[i], 1);
309     for (uint8_t i=0; i<sizeof(relayF2); i++) digitalWrite(relayF2Pins[i], 1);
310     for (uint8_t i=0; i<sizeof(relayM3); i++) digitalWrite(relayM3Pins[i], 1);
311
312     // 5v/gnd constant pins for relay on-board optocouplers
313     if (auxPower){
314         // relay 1 power/gnd (front)
315         pinMode(23, OUTPUT);
316         digitalWrite(23, HIGH);
317         pinMode(37, OUTPUT);
318         digitalWrite(37, LOW);
319
320         // relay 2 power/gnd (rear)
321         pinMode(39, OUTPUT);
322         digitalWrite(39, LOW);
323         pinMode(53, OUTPUT);
324         digitalWrite(53, HIGH);
325     }
326
327     // relay 3 is powered by jumpers on the arduino screw terminal board
328     // 2 is GND, 7 is Vcc
329     if (debuggingActive) Serial.println("Done.");
330
331     // reset all oneshot/toggle/debounce bits before use
332     if (debuggingActive) Serial.print("Resetting logic states...");
333     for (uint8_t i=0; i<sizeof(toggledMem); i++)
334     {
335         oneShotBits[i] = 0;             // general use oneshot bits
336         oneShotBits[i+32] = 0;          // fault oneshot bits
337         toggledMem[i] = 0;              // memory bits for previous condition of
              toggled bit
338         faultFlags[i] = 0;              // fault bits
339     }
340     for(uint8_t i=0; i<sizeof(oneShotBits); i++) oneShotBits[i] = 0;
341     if (debuggingActive) Serial.println("Done.");
342
343     // reset all fault flags
344     for(uint8_t i=0; i<sizeof(faultFlags); faultFlags[i++] = 0);
```

```
345
346     setupTime = millis() - setupTime;
347     if (debuggingActive) Serial.println("\nSETUP COMPLETE in " + String
          (setupTime) + "ms\n");
348
349     // update IO before starting main cycle
350     MapInputs(1);
351     MapOutputs(1);
352
353     last_cycleStart = millis();
354     last_cycleStartus = micros();
355     if (debuggingActive) Serial.println("Beginning first scan");
356 }
357
358 void loop()
359 {
360     ONSTracker = 0;
361     // debounceTracker = 0;
362     timerTracker = 0;
363
364     // IO updates at 50hz
365     boolean updateIOPulse = oneShot(FlasherBit(50), ONSTracker++);
366     MapInputs(updateIOPulse);               // rountine io updates done when io
          pulse true
367     MapOutputs(updateIOPulse);              // rountine io updates done when io
          pulse true
368     // both are PET, because the serial is updated on a NET of a 50Hz pulse, so
          IO and serial updates are staggered
369     // speedometer PX needs to be updated FAST
370
371     LightRoutine();                         // sets the status of lights on
          front and rear relay boards using pointers
372
373     // only update clock if there is 5V aux available
374     if(auxPower && oneShot(FlasherBit(2), ONSTracker++)) ClockRoutine();
375
376     DriveSystem();                          // determines if it is safe to move
          the vehicle, and if so, generates a throttle value accordingly
377
378     MonitorSystem();                        // monitors analog inputs for
          voltage and current readings, and front wheel speed
379
380     if (nextionEnabled) HMIControl();       // displays data collected in
          DriveSystem and MonitorSystem and converts it into strings and sends data
          to nextion
381
382     if (debuggingActive) debugRoutine();    // temporary fixes and misc message
          generation for debugging purposes
383
384     Faults();                               // sets fault flags based on system
          status
385
```

```
386     if (debuggingActive && firstScan) Serial.println("First scan complete.");
387     if (firstScan) firstScan = 0;
388
389     delayMicroseconds(500); // for stability
390     // CREATE EEPROM UPDATE LOGIC?
391 }
392
393 // SUBROUTINES
394 void MapInputs(boolean IOUpdate)
395 {
396
397     if (IOUpdate)
398     {
399         for (uint8_t i=0; i<sizeof(opto1); i++)  opto1[i] = !digitalRead
              (opto1pins[i]);
400         for (uint8_t i=0; i<sizeof(opto2); i++)  opto2[i] = !digitalRead
              (opto2pins[i]);
401         *optoPower = !*optoPower;   // *optoPower is high when on, so need to
              re-invert input
402
403         // MapInputs is read at 50Hz, so anything that is not throttleInput is
              done one at a time for speed
404         if (analogInputTracker >= sizeof(analogInputs) / 2)
405         {
406             analogInputTracker = 1;
407             last_analogCycleComplete = millis();
408         }
409         analogInputs[0] = analogRead(analogInputPins[0]);
410         analogInputs[analogInputTracker] = analogRead(analogInputPins
              [analogInputTracker]);
411         analogInputTracker++;
412     }
413
414     speedoPXInput = digitalRead(speedoPXInputPin);  // this pin needs to update
          as fast as it can
415
416 }
417
418 void LightRoutine()
419 {
420     if (!*lightsOn)
421     {
422         *FLTurn = *RLTurn = *FRTurn = *RRTurn = 0;
423     }
424     else if (*leftTurnInput && *rightTurnInput)
425     {
426         *FLTurn = *RLTurn = *FRTurn = *RRTurn = FlasherBit(1.5);
427     }
428     else if (*leftTurnInput)
429     {
430         *FLTurn = *RLTurn = FlasherBit(1.5);
431         *FRTurn = 1;
```

```
432         *RRTurn = 0;
433     }
434     else if (*rightTurnInput)
435     {
436         *FRTurn = *RRTurn = FlasherBit(1.5);
437         *FLTurn = 1;
438         *RLTurn = 0;
439     }
440     else // front turn signals are on at idle on motorcycles
441     {
442         *FLTurn = *FRTurn = 1;
443         *RLTurn = *RRTurn = 0;
444     }
445
446     // headlight
447     *lowBeamsOut = *lowBeamInput && *lightsOn;
448     *highBeamsOut = *highBeamInput && *lightsOn;
449
450     // brake light
451     *brakeHighOutput = ((*rearBrake || *frontBrake) && *lightsOn);
452     *brakeLowOutput =  *lightsOn;
453
454     // fog lights
455     *fogLightsOut = *fogLightsIn;
456
457     // just repeat the horn out to the BOI
458     *hornOutput = *hornInput;
459 }
460
461 void ClockRoutine(){
462     // update time at 4Hz
463     if (auxPower) now = RTC.now();
464 }
465
466 void DriveSystem()
467 {
468     /*
469         - take the inGear mode and input from the throttle and generate a 1-4V
            output on PWM pin 13
470     */
471     if (!killswOffSinceBoot || !*killswitch) inGear = 0;
472     else if (!inGear && throttlePercent > 2);   // do NOT go into gear unless
          throttle is idle
473     else inGear = toggleState(*startPb, ONSTracker++);
474
475     // check if killswitch has been off since boot
476     if (!killswOffSinceBoot && !*killswitch) killswOffSinceBoot = 1;
477
478     // control logic for the ESC solenoid
479     if (!killswOffSinceBoot) *ESCSolenoid = 0;
480     else if (contactorSafetiesActive && anyLevel1FaultDetected) *ESCSolenoid =
          0;
```

```
481     else *ESCSolenoid = *killswitch;
482
483     throttlePercent = map(*throttleInput, voltsToAnalogIn(1), voltsToAnalogIn
          (4), 0, 100);       // calculate the pulse width in us for motor out
484
485     if (inGear && throttlePercent < 2) throttleOutput = map(*throttleInput,
          voltsToAnalogIn(1), voltsToAnalogIn(4), voltsToAnalogOut(1),
          voltsToAnalogOut(4));
486     else throttleOutput = voltsToAnalogOut(1);
487
488     return;
489 }
490
491 void MonitorSystem()
492 {
493
494     // reset watchdog timer
495     watchdog = millis() - last_cycleStart;
496     last_cycleStart = millis();
497
498     // check for fastest/longest cycle
499     if (watchdog > longestCycle) longestCycle = watchdog;
500     if (watchdog < fastestCycle) fastestCycle = watchdog;
501
502     // value calculated based on what ina126p output should be
503     currentDrawShuntmA = map(*currentDrawInput, 0, voltsToAnalogIn(4.425), 0,
          15000); // calculates a precise float in ma
504     currentDrawMotorAmps = map(*motorDrawInput, 0, voltsToAnalogIn(4.25), 0,
          150); // calculates a precise float in amps
505     for (uint8_t i=0; i < sizeof(cellVolts) / 4; i++) cellVolts[i] = (float)map
          (*cellVoltageRawIn[i], 0, voltsToAnalogIn(3.546), 0, 22200) / 1000.0;   //
           calculates a precise cell level in volts
506     vBat = (float)(map(*vBatInput, 0, voltsToAnalogIn(3.861), 0, 90500)) /
          1000.0;
507
508
509     // draw in mAh to be added is draw * 0.2778 = (mAh in 0.5s at draw rate in
          mA)
510     if (oneShot(FlasherBit(2), ONSTracker++)) totalDraw_mAh += (int)
          (currentDrawMotorAmps * 0.1389);        // calculation done in Amps
511     /*
512         10 A for an hour is 10000mAh
513         for a second is 2.778mAh
514         for half a second is 1.389mAh
515         Therefore, for each amp drawn, 0.1389mAh is expended in 0.5s
516     */
517
518     // SPEED LOGIC
519     if (oneShot(speedoPXInput, ONSTracker++))
520     {
521         lastPXTime = thisPXTime;
522         thisPXTime = millis();
```

```
523         speed = map(thisPXTime - lastPXTime, 10, 40, 100, 25);
524                                     // 10ms, 40ms, 100kph, 25kph
525     }
526     else if (millis() - thisPXTime > 300) speed = 0;
527     /*  speed is an integer representing speed in km/h
528         front wheel has 8 spokes on the brake
529         front wheel has a 70 cm OD
530         c = pi * 70
531         each PET = c/8 cm travelled
532         I will use the time between PETs to calculate the speed
533         At 100kph, the wheel rotates at 12.64rps
534         That is 101 pulses per second, meaning we need at least 200-300 samples
              per second
535
536         101 pulses per second at 100kph
537         10ms between each PET
538         Likewise, 25kph would be 40ms between each pulse
539         We will use a map() function with these two points to calculate the
              speed, using the micros() function, for accuracy
540     */
541
542
543 }
544
545 void HMIControl()
546 {
547     long hmiStartTime = millis();
548
549     if (nexRead()) nexCheckButtonPress(nexBytesRead);
550
551     // cycleActiveFaults() MUST be called every cycle because it uses oneshot
          bits, so we need a buffer, 'j'
552     String cycleActiveFaultsTemp = cycleActiveFaults();
553
554     int thisSpot = 0;
555     boolean nextUpdate = oneShot(!FlasherBit(50), ONSTracker++);
556     if (nextUpdate) hmiElemToUpd++;
557     //if (debuggingHMIActive && nextUpdate) Serial.print("Beginning update of
          element " + String(hmiElemToUpd) + "...");
558
559     if (oneShot(nextionDelay, ONSTracker++)) Serial.println("Nextion overloaded,
           delaying serial.");
560     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexTextFromString("alarmView0", cycleActiveFaultsTemp, 60); // 0
561     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("alarmView2", cycleActiveFaultsTemp, 60); // 1
562     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm0", listAllFaults(0), 60);          // 2
563     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm1", listAllFaults(1), 60);          // 3
564     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm2", listAllFaults(2), 60);          // 4
```

```
565     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm3", listAllFaults(3), 60);          // 5
566     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm4", listAllFaults(4), 60);          // 6
567     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm5", listAllFaults(5), 60);          // 7
568     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm6", listAllFaults(6), 60);          // 8
569     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm7", listAllFaults(7), 60);          // 9
570     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm8", listAllFaults(8), 60);          // 10
571     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 1)
          nexTextFromString("alarm9", listAllFaults(9), 60);          // 11
572
573     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexTextFromString("vBat0", String(vBat) + " V", 10);            // 12
574     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("vBat2", String(vBat) + " V", 10);            // 13
575     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("cell1v", String(cellVolts[0]) + " V", 10);// 14
576     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("cell2v", String(cellVolts[1]) + " V", 10);// 15
577     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("cell3v", String(cellVolts[2]) + " V", 10);// 16
578     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("cell4v", String(cellVolts[3]) + " V", 10);// 17
579     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexBar("voltBar", (int)map(vBat, 0, 96, 0, 100));           // 18
580     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexTextFromString("currentDis0", String(currentDrawMotorAmps) + " A", 10);
              // 19
581     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("currentDis2", String(currentDrawMotorAmps) + " A", 10);
              // 20
582     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexBar("currBar", (int)map(currentDrawMotorAmps, 0, 120, 0, 100));
                  // 21
583
584     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexTextFromString("powerDis0", String(round(currentDrawMotorAmps * vBat))
          + " W", 10); // 22
585     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("powerDis2", String(round(currentDrawMotorAmps * vBat))
          + " W", 10); // 23
586     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexBar("powerBar", (int)map(currentDrawMotorAmps * vBat, 0, 9000, 0,
          100));      // 24
587
588     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexTextFromString("mAh0", String(totalDraw_mAh) + " mAh", 10);          //
           25
```

```
589     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("mAh2", String(totalDraw_mAh) + " mAh", 10);          // 26
590
591     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
          nexTextFromString("speedDis0", String(speed) + " km/h", 10);            // 27
592     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
          nexTextFromString("speedDis2", String(speed) + " km/h", 10);        // 28
593
594     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
                                // 29
595     {
596         if (inGear && nexTextFromString("inGearInd0", "D", 2))
597         {
598             nexSetFontColor("inGearInd0", LBLUE);
599         }
600         if (!inGear && nexTextFromString("inGearInd0", "N", 2))
601         {
602             nexSetFontColor("inGearInd0", GREEN);
603         }
604     }
605     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 2)
                                // 30
606     {
607         if (inGear && nexTextFromString("inGearInd2", "D", 2))
608         {
609             nexSetFontColor("inGearInd2", LBLUE);
610         }
611         if (!inGear && nexTextFromString("inGearInd2", "N", 2))
612         {
613             nexSetFontColor("inGearInd2", GREEN);
614         }
615     }
616
617     // ADD CLOCK UPDATE LOGIC FROM 'HMI clock test' STANDARD!!!!!!
618     else if ((hmiElemToUpd == thisSpot++) && nextUpdate && nextionPage == 0)
                                // 31
619     {
620         char clockBuffer[10];
621         sprintf(clockBuffer,"%02u:%02u:%02u",now.hour(),now.minute(),now.second
              ());
622         nexTextFromString("clockDis", String(clockBuffer), 10);
623     }
624     else if (hmiElemToUpd == thisSpot++)
              // reset
625     {
626         if (debuggingHMIActive) Serial.println("\nResetting HMI elements counter
              from " + String(hmiElemToUpd));
627         hmiElemToUpd = -1;
628     }
629
```

```
630     else;
631
632     //if (debuggingHMIActive && nextUpdate && (hmiElemToUpd != -1))
          Serial.println("Done.");
633
634     hmiOvertimeLength = millis() - hmiStartTime;
635 }
636
637 void MapOutputs(boolean IOUpdate)
638 {
639     if (IOUpdate)
640     {
641         // outputs are active low
642         if (auxPower || debuggingActive)
643         {
644             for (uint8_t i=0; i<sizeof(relayR1); i++) digitalWrite(relayR1Pins
                  [i], !relayR1[i]);
645             for (uint8_t i=0; i<sizeof(relayF2); i++) digitalWrite(relayF2Pins
                  [i], !relayF2[i]);
646             for (uint8_t i=0; i<sizeof(relayM3); i++) digitalWrite(relayM3Pins
                  [i], !relayM3[i]);
647         }
648         else
649         {
650             for (uint8_t i=0; i<sizeof(relayR1); i++) digitalWrite(relayR1Pins
                  [i], 1);
651             for (uint8_t i=0; i<sizeof(relayF2); i++) digitalWrite(relayF2Pins
                  [i], 1);
652             for (uint8_t i=0; i<sizeof(relayM3); i++) digitalWrite(relayM3Pins
                  [i], 1);
653         }
654         analogWrite(throttleOutputPin, throttleOutput);
655     }
656 }
657
658 void Faults()
659 {
660
661     // fault reset
662     if (faultReset)
663     {
664         for (uint8_t i=0; i<sizeof(faultFlags); faultFlags[i++] = 0);
665         faultReset = 0;
666         if (debuggingActive) Serial.println("Faults reset.");
667     }
668
669
670     // COMMS errors
671     //if(!*optoPower || !*handlebarPowerLeft || !*handlebarPowerRight)
          *allConnectionsOK = 1;      // active bit for comms lost
672     if (!*optoPower) *optoPowerLost = 1;
673     if (!*handlebarPowerLeft) *handlebarPowerLeftFault = 1;
```

```
674     if (!*handlebarPowerRight) *handlebarPowerRightFault = 1;
675
676     // overtime errors
677     if (watchdog > 100)
678     {
679         *watchdogFlag = 1;
680         overTimeLength = watchdog;
681     }
682
683     // control system errors
684     *throttleOutOfBounds = limit(voltsFromAnalogIn(*throttleInput), 0.5, 4.5);
685     //if (*throttleOutOfBounds) *throttleOutOfBoundsFlag = 1;
686     if (ONSTracker >= sizeof(oneShotBits) || timerTracker >= 32)
          trackerBitsOverload = 1;
687     if (!inGear && throttlePercent > 2 && *startPb) *thlNotIdleWhenReq = 1;
688     if (currentDrawShuntmA > 12000) *controlOvercurrentFlag = 1;
689
690     // battery faults
691     // 14.5 is the 'low' cell value
692     float k = 14.5;
693     *cellOneLow     = (cellVolts[0] < k);
694     *cellTwoLow     = (cellVolts[0] < k);
695     *cellThreeLow   = (cellVolts[0] < k);
696     *cellFourLow    = (cellVolts[0] < k);
697     for(uint8_t i=0; i<4; i++)  // check for critical cell voltages
698     {   // only cover 2.5-3.3V; dont want to estop for a disconnected cell
699         if (limit(cellVolts[i] / 4.0, 2.5, 3.3)) { anyCellCritical = 1; break; }
                // each cell is 4S
700     }
701
702     // check for existing faults
703     for (uint8_t i=0; i<sizeof(faultFlags); i++)
704     {
705         if (faultFlags[i])
706         {
707             *anyFaultsDetected = 1;
708             break;
709         }
710     }
711     for (uint8_t i=0; i<32; i++)
712     {
713         if (faultFlags[i] && faultLevel[i] == 2)
714         {
715             *anyLevel2FaultDetected = 1;
716             break;
717         }
718     }
719     for (uint8_t i=0; i<sizeof(faultFlags); i++)
720     {
721         if (faultFlags[i] && faultLevel[i] == 1)
722         *anyLevel1FaultDetected = 1;
723         break;
```

```
724     }
725
726     // DISPLAY FAULTS ON SERIAL
727     numberOfFaults = 0;
728     if (TON(1, 250, timerTracker++) && faultsActive) // dont display faults for
          the first 1/4s of operation
729     {
730         for (uint8_t i=0; i<sizeof(faultFlags); i++)
731         {
732             if (debuggingActive && oneShot(faultFlags[i], i + 32))
                  Serial.println(faultMessages[i]);
733             if (faultFlags[i]) numberOfFaults++;
734         }
735     }
736
737     //AUX POWER FAULT
738     *auxPowerFault = !auxPower;
739
740     //  HMI FAULTS
741     if (hmiOvertimeLength > 150) // disable nextion if overtime detected
742     {
743         *hmiOvertimeFault = 1;
744         nextionEnabled = 0;
745     }
746
747     // TRACKER FAULTS MUST BE AT THE END OF EVERY CYCLE
748     if (oneShot(timerTracker != expectedTimer, ONSTracker++) && !firstScan) {
749         *unexpectedTimer = 1;
750         Serial.println("timerTracker expected was " + String(expectedTimer) + ",
              finished cycle with " + String(timerTracker));
751     }
752     if (oneShot((ONSTracker + 1) != expectedONS, ONSTracker++) && !firstScan) {
753         *unexpectedONS = 1;
754         Serial.println("ONSTracker expected was " + String(expectedONS) + ",
              finished cycle with " + String(ONSTracker));
755     }
756     if (firstScan)
757     {
758         expectedONS = ONSTracker;
759         expectedTimer = timerTracker;
760         if (debuggingActive) Serial.println("ONS used on first scan was: " +
              String(ONSTracker));
761     }
762
763 }
764
765 void debugRoutine()
766 {
767
768     // monitor cycle time
769     /*
770     watchdogus = micros() - last_cycleStartus;
```

```
771     last_cycleStartus = micros();
772     if (oneShot(FlasherBit(1), ONSTracker++)) Serial.println("Last cycle time
          was " + String(watchdogus) + "us");
773       // cycle time in us
774       */
775
776     if (oneShot(FlasherBit(1), ONSTracker++) && *watchdogFlag) Serial.println
          ("Watchdog was " + String(overTimeLength) + "ms");
777
778     //if (oneShot(FlasherBit(4), ONSTracker++)) Serial.println("px in " + String
          (*speedoPXInput));
779
780     if (driveSystemDebuggingActive && oneShot(FlasherBit(1), ONSTracker++))
          Serial.println("*killswitch is " + String(*killswitch));
781     if (driveSystemDebuggingActive && oneShot(FlasherBit(1), ONSTracker++))
          Serial.println("*escsolenoid is " + String(*ESCSolenoid));
782
783     // RESET WATCHDOG FAULT AT BEGINNING OF SCAN
784     if (oneShot(TON(1, 200, timerTracker++), ONSTracker++)) faultReset = 1;
785 }
786
787
788 // Button press function calls from Nextion
789
790 void alarmView0Callback()
791 {
792     nextionPage = 1;
793     if (debuggingActive) Serial.println("Page is now " + String(nextionPage));
794 }
795 void STAT0Callback()
796 {
797     nextionPage = 2;
798     clearNextionBuffer();
799     if (debuggingActive) Serial.println("Page is now " + String(nextionPage));
800 }
801 void STAT1Callback()
802 {
803     nextionPage = 2;
804     clearNextionBuffer();
805     if (debuggingActive) Serial.println("Page is now " + String(nextionPage));
806 }
807 void MAIN1Callback()
808 {
809     nextionPage = 0;
810     clearNextionBuffer();
811     if (debuggingActive) Serial.println("Page is now " + String(nextionPage));
812 }
813 void alarmView2Callback()
814 {
815     nextionPage = 1;
816     clearNextionBuffer();
817     if (debuggingActive) Serial.println("Page is now " + String(nextionPage));
```

```
818 }
819 void MAIN2Callback()
820 {
821     nextionPage = 0;
822     clearNextionBuffer();
823     if (debuggingActive) Serial.println("Page is now " + String(nextionPage));
824 }
825 void FLTRST0Callback()
826 {
827     faultReset  = 1;
828     if (debuggingActive) Serial.println("Fault reset issued");
829 }
830 void FLTRST1Callback()
831 {
832     faultReset  = 1;
833     if (debuggingActive) Serial.println("Fault reset issued");
834 }
835 void FLTRST2Callback()
836 {
837     faultReset  = 1;
838     if (debuggingActive) Serial.println("Fault reset issued");
839 }
840
841
842 // FUNCTION CALLS
843 boolean FlasherBit(float hz)
844 {
845     int T = round(1000.0 / hz);
846     if ( millis() % T >= T/2 ) return 1;
847     else return 0;
848 }
849
850 boolean oneShot(boolean precond, uint8_t OSR)
851 {
852     // use global memory to keep track of oneshot bits
853     if (precond == 1 && oneShotBits[OSR] == 0)
854     {
855         oneShotBits[OSR] = 1;
856         return 1;
857     }
858     else if (precond == 0 && oneShotBits[OSR] == 1)
859     {
860         oneShotBits[OSR] = 0;
861         return 0;
862     }
863     else return 0;
864 }
865
866 boolean toggleState(boolean precond, uint8_t OSR)
867 {
868     if (oneShot(precond, OSR)) toggledMem[OSR] = !toggledMem[OSR];
869     return toggledMem[OSR];
```

```
870 }
871
872 float voltsFromAnalogIn (int input)
873 {
874     int output = (float)(input * 5) / 1024.0;
875     return output;
876 }
877
878 int voltsToAnalogIn (float input)
879 {
880     int output = round(input * 1024.0) / 5;
881     return output;
882 }
883
884 int voltsToAnalogOut(float input)
885 {
886     int output = (input * 255.0 / 5.0);
887     return output;
888 }
889
890 boolean TON(boolean input, int preset, int timerNumber)
891 {
892     if (input && !timerInSession[timerNumber]) timerTimers[timerNumber] = millis ↵
        ();
893     else if (input && timerMemory[timerNumber]) return 1;
894     else if (input && millis() - timerTimers[timerNumber] >= preset)
895     {
896         timerMemory[timerNumber] = 1;
897         return 1;
898     }
899     else;
900     timerMemory[timerNumber] = 0;
901     timerInSession[timerNumber] = input;
902     return 0;
903 }
904
905 boolean limit(float input, float lower, float upper)
906 {
907     if (input < lower) return 0;
908     else if (input > upper) return 0;
909     else return 1;
910 }
911
912 String listAllFaults(int spot)
913 {
914     int j = 0;
915     String buffer[10] = {" "," "," "," "," "," "," "," "," "," "," ",};
916     for (int i=0; i<sizeof(faultFlags) && j<10; i++) {
917         if (faultFlags[i] && faultLevel[i] == 1) {
918             buffer[j] = "1: " + faultMessages[i] + "\n";
919             j++;
920         }
```

```
921     }
922     for (int i=0; i<sizeof(faultFlags) && j<10; i++) {
923         if (faultFlags[i] && faultLevel[i] == 2) {
924             buffer[j] = "2: " + faultMessages[i] + "\n";
925             j++;
926         }
927     }
928     for (int i=0; i<sizeof(faultFlags) && j<10; i++) {
929         if (faultFlags[i] && faultLevel[i] == 3) {
930             buffer[j] = "3: " + faultMessages[i] + "\n";
931             j++;
932         }
933     }
934     return buffer[spot];
935 }
936
937 String cycleActiveFaults()
938 {
939     if (oneShot(FlasherBit(0.2), ONSTracker++)) activeFaultToDisplay++;
940     if (activeFaultToDisplay >= numberOfFaults) activeFaultToDisplay = 0;
941     int j = 0;
942     for (int i=0; i<sizeof(faultFlags); i++)
943     {
944         if (faultFlags[i])
945         {
946             if (j == activeFaultToDisplay) break;
947             j++;
948         }
949     }
950     return faultMessages[j];
951 }
952
953 boolean nexTextFromString(String objName, String input, int len)
954 {
955     // function only sets the element text if a change is detected,
956     // and returns 'true' if a change is detected
957     if (input != nexBuffer[hmiElemToUpd])
958     {
959         if (debuggingHMIActive) Serial.println("buffer update on nextion element ↵
            " + objName);
960         nexBuffer[hmiElemToUpd] = input;
961         String cmd;
962         cmd += objName;
963         cmd += ".txt=\"";
964         cmd += input;
965         cmd += "\"";
966         nexSerial.print(cmd);
967         nexSerial.write(0xFF);
968         nexSerial.write(0xFF);
969         nexSerial.write(0xFF);
970         return 1;
971     }
```

```
    else return 0;
}

boolean nexBar(String objName, int val)
{
    String cmd;
    cmd += objName;
    cmd += ".val=";
    cmd += String(val);
    nexSerial.print(cmd);
    nexSerial.write(0xFF);
    nexSerial.write(0xFF);
    nexSerial.write(0xFF);
    return 1;
}

void nexCustomSerial(long speed)
{
    Serial2.begin(9600);
    delay(20);
    Serial2.print("baud=" + String(speed));
    Serial2.write(0xff);
    Serial2.write(0xff);
    Serial2.write(0xff);
    delay(20);
    Serial2.begin(speed);
    delay(20);
}

boolean nexSetFontColor(String objName, uint32_t number)
{
    char buf[10] = {0};
    String cmd;

    utoa(number, buf, 10);
    cmd += objName;
    cmd += ".pco=";
    cmd += buf;
    nexSerial.print(cmd);
    nexSerial.write(0xFF);
    nexSerial.write(0xFF);
    nexSerial.write(0xFF);

    cmd = "";
    cmd += "ref ";
    cmd += objName;
    nexSerial.print(cmd);
    nexSerial.write(0xFF);
    nexSerial.write(0xFF);
    nexSerial.write(0xFF);

    return 1;
```

```
1024 }
1025
1026 boolean nexRead()
1027 {
1028     uint8_t __buffer[10];
1029
1030     uint16_t i;
1031     uint8_t c;
1032
1033     while (nexSerial.available() > 0)
1034     {
1035         delay(1);
1036         c = nexSerial.read();
1037
1038         if (NEX_RET_EVENT_TOUCH_HEAD == c)
1039         {
1040             if (nexSerial.available() >= 6)
1041             {
1042                 __buffer[0] = c;
1043                 for (i = 1; i < 7; i++)
1044                 {
1045                     __buffer[i] = nexSerial.read();
1046                 }
1047                 __buffer[i] = 0x00;
1048
1049                 if (0xFF == __buffer[4] && 0xFF == __buffer[5] && 0xFF ==
                      __buffer[6])
1050                 {
1051                     nexBytesRead[0] = __buffer[1];
1052                     nexBytesRead[1] = __buffer[2];
1053                     nexBytesRead[2] = __buffer[3];
1054                     return 1;
1055                 }
1056
1057             }
1058         }
1059     }
1060     return 0;
1061 }
1062
1063 boolean nexCheckButtonPress(uint8_t recv[3])
1064 {
1065     // check if message received is for any buttons, and return 1 if a valid
        button press is detected
1066     if (recv[0] == 0x00 && recv[1] == 0x04 && recv[2] == 0x01)
1067     {
1068         STAT0Callback();
1069         return 1;
1070     }
1071     if (recv[0] == 0x00 && recv[1] == 0x11 && recv[2] == 0x01)
1072     {
1073         alarmView0Callback();
```

```
1074            return 1;
1075        }
1076        if (recv[0] == 0x00 && recv[1] == 0x02 && recv[2] == 0x01)
1077        {
1078            FLTRST0Callback();
1079            return 1;
1080        }
1081        if (recv[0] == 0x01 && recv[1] == 0x04 && recv[2] == 0x01)
1082        {
1083            FLTRST1Callback();
1084            return 1;
1085        }
1086        if (recv[0] == 0x01 && recv[1] == 0x03 && recv[2] == 0x01)
1087        {
1088            STAT1Callback();
1089            return 1;
1090        }
1091        if (recv[0] == 0x01 && recv[1] == 0x02 && recv[2] == 0x01)
1092        {
1093            MAIN1Callback();
1094            return 1;
1095        }
1096        if (recv[0] == 0x02 && recv[1] == 0x19 && recv[2] == 0x01)
1097        {
1098            alarmView2Callback();
1099            return 1;
1100        }
1101        if (recv[0] == 0x02 && recv[1] == 0x02 && recv[2] == 0x01)
1102        {
1103            FLTRST2Callback();
1104            return 1;
1105        }
1106        if (recv[0] == 0x02 && recv[1] == 0x01 && recv[2] == 0x01)
1107        {
1108            MAIN2Callback();
1109            return 1;
1110        }
1111
1112        return 0;
1113    }
1114
1115    void clearNextionBuffer()
1116    {
1117        for (int i=0; i<30; i++)
1118        {
1119            nexBuffer[i] = "";
1120        }
1121    }
1122
```