```
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
```