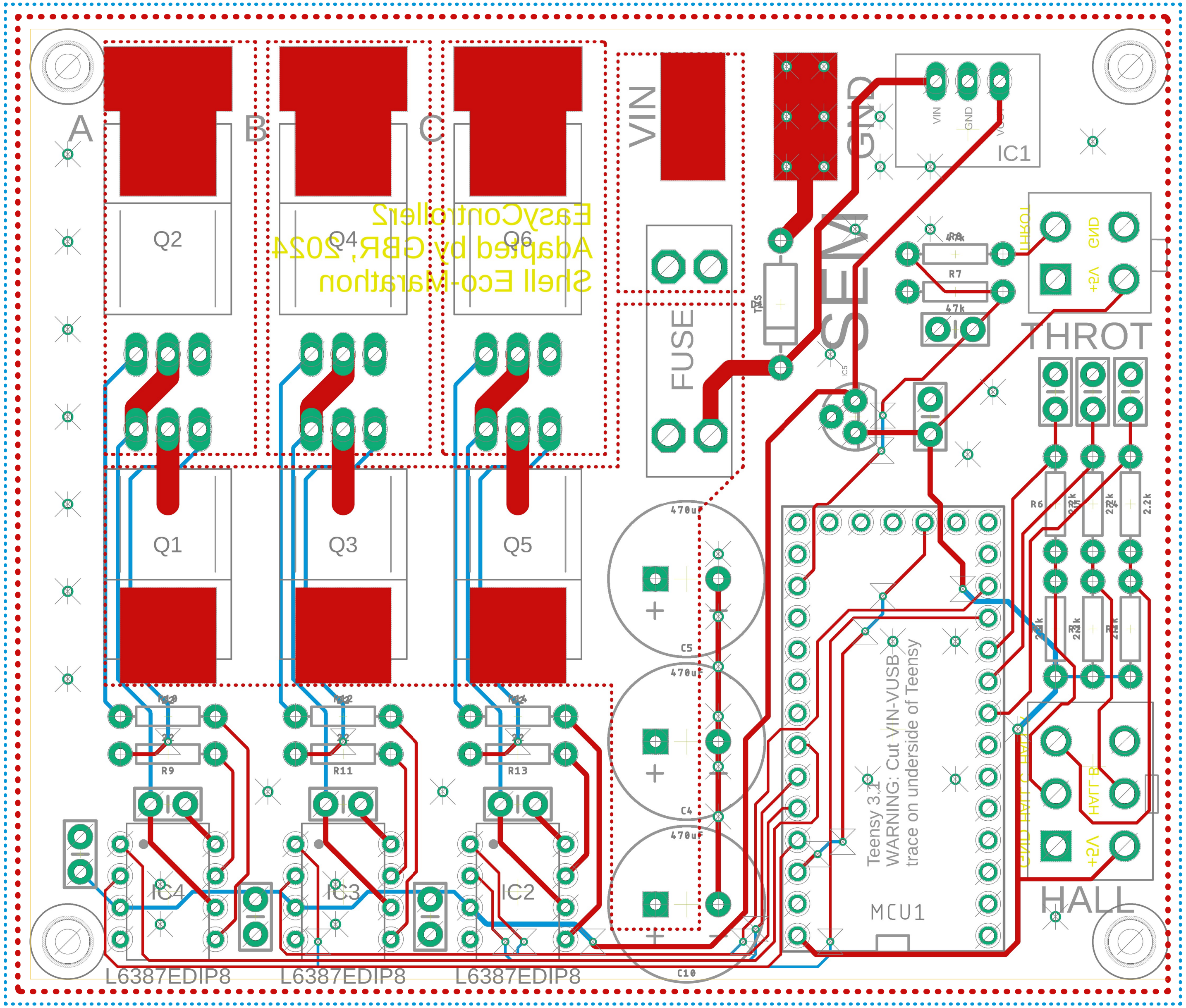
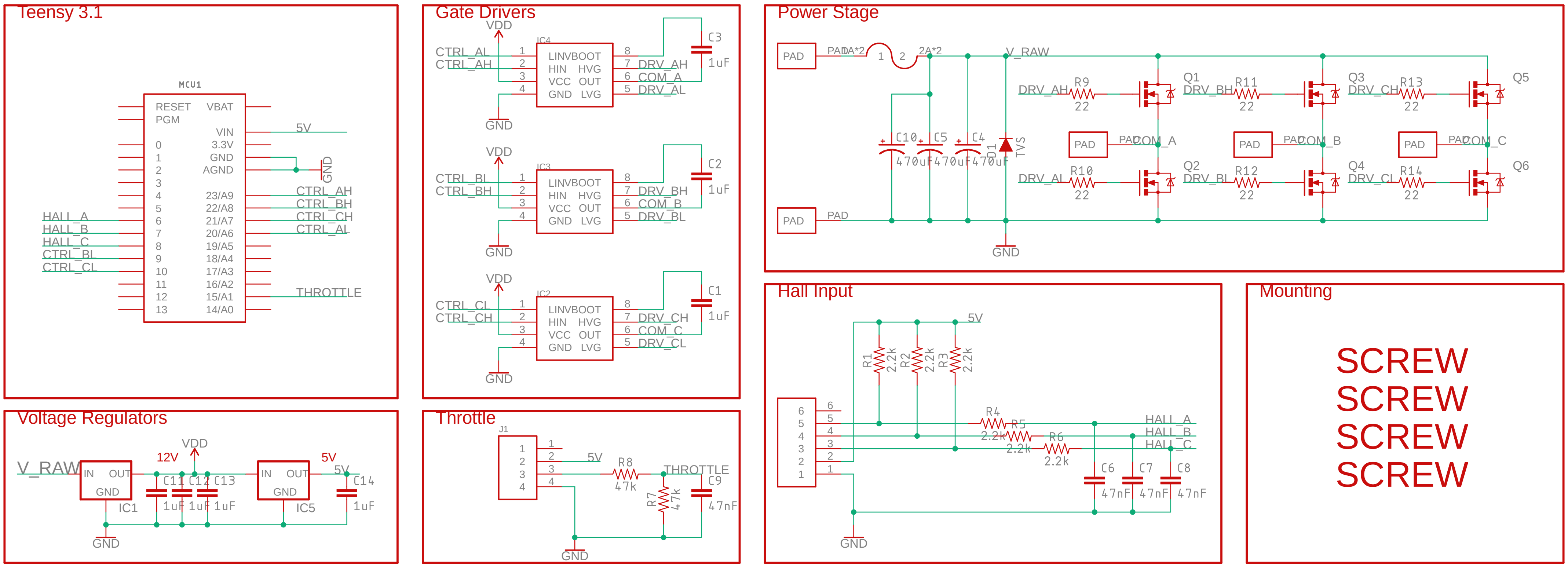
GREEN BATH RACING

Motor Controller Documentation

MOTOR CONTROLLER BOARD



MOTOR CONTROLLER SCHEMATIC



MOTOR CONTROLLER COMPONENT SELECTION

Component	Supplier	Part No.	Quantity
1 uF Cap	RS Components	242-7571	50
TVS DIODE	RS Components	687-3918	20
47nF Cap	RS Components	537-3937	25
Voltage Regulator	RS Components	174-4725	2
5V Linear Reg	RS Components	714-7768	25
14 pin male header	RS Components	828-1614	5
14 pin female header	RS Components	233-9407	5
Gate Driver	Onecall Farnell	1603665	6
NFET	Onecall Farnell	2456726	12
470uF Cap	Onecall Farnell	2900564	6
2.2k Resistor	Onecall Farnell	1265074	12
Fuse holder	Onecall Farnell	2292904	2
47k resistor	Onecall Farnell	9339558	10
22 ohm resistor	Onecall Farnell	9341560	12
4 pin female connector	Onecall Farnell	8115940	10
6 pin female connector	Onecall Farnell	8115958	10
4 pin male connector	Onecall Farnell	8116270	100
6 pin male connector	Onecall Farnell	2311108	10
Crimp pin	Onecall Farnell	1339252	20
Teensy 3.1	University of Bath	.	1

MOTOR CONTROLLER CODE

```
#define THROTTLE_PIN 15        // Throttle pin
#define THROTTLE_LOW 150       // These LOW and HIGH values are used to scale
the ADC reading. More on this below
#define THROTTLE_HIGH 710

#define HALL_1_PIN 6
#define HALL_2_PIN 7
#define HALL_3_PIN 8

#define AH_PIN 23              // Pins from the Teensy to the gate drivers. AH
= A high, etc
#define AL_PIN 20
#define BH_PIN 22
#define BL_PIN 9
#define CH_PIN 21
#define CL_PIN 10

#define LED_PIN 13             // The teensy has a built-in LED on pin 13

#define HALL_OVERSAMPLE 4      // Hall oversampling count. More on this in the
getHalls() function

uint8_t hallToMotor[8] = {255, 255, 255, 255, 255, 255, 255, 255};

void setup() {                 // The setup function is called ONCE on boot-up
  Serial.begin(115200);

  pinMode(LED_PIN, OUTPUT);
  digitalWriteFast(LED_PIN, HIGH);

  pinMode(AH_PIN, OUTPUT);     // Set all PWM pins as output
  pinMode(AL_PIN, OUTPUT);
  pinMode(BH_PIN, OUTPUT);
  pinMode(BL_PIN, OUTPUT);
  pinMode(CH_PIN, OUTPUT);
  pinMode(CL_PIN, OUTPUT);

  analogWriteFrequency(AH_PIN, 8000); // Set the PWM frequency. Since all pins
are on the same timer, this sets PWM freq for all

  pinMode(HALL_1_PIN, INPUT);         // Set the hall pins as input
  pinMode(HALL_2_PIN, INPUT);
  pinMode(HALL_3_PIN, INPUT);

  pinMode(THROTTLE_PIN, INPUT);

  identifyHalls();                   // Uncomment this if you want the
controller to auto-identify the hall states at startup!
```

```
}

void loop() {                             // The loop function is called
repeatedly, once setup() is done

  uint8_t throttle = readThrottle();  // readThrottle() is slow. So do the
more important things 200 times more often
  for(uint8_t i = 0; i < 200; i++)
  {
    uint8_t hall = getHalls();              // Read from the hall sensors
    uint8_t motorState = hallToMotor[hall]; // Convert from hall values (from
1 to 6) to motor state values (from 0 to 5) in the correct order. This line is
magic
    writePWM(motorState, throttle);         // Actually command the
transistors to switch into specified sequence and PWM value
  }
}

/* Magic function to do hall auto-identification. Moves the motor to all 6
states, then reads the hall values from each one
 *
 * Note, that in order to get a clean hall reading, we actually need to
commutate to half-states. So instead of going to state 3, for example
 * we commutate to state 3.5, by rapidly switching between states 3 and 4.
After waiting for a while (half a second), we read the hall value.
 * Finally, print it
 */

void identifyHalls()
{
  for(uint8_t i = 0; i < 6; i++)
  {
    uint8_t nextState = (i + 1) % 6;        // Calculate what the next state
should be. This is for switching into half-states
    Serial.print("Going to ");
    Serial.println(i);
    for(uint16_t j = 0; j < 200; j++)       // For a while, repeatedly switch
between states
    {
      delay(1);
      writePWM(i, 20);
      delay(1);
      writePWM(nextState, 20);
    }
    hallToMotor[getHalls()] = (i + 2) % 6;  // Store the hall state - motor
state correlation. Notice that +2 indicates 90 degrees ahead, as we're at half
states
  }
```

```
  writePWM(0, 0);                          // Turn phases off

  for(uint8_t i = 0; i < 8; i++)           // Print out the array
  {
    Serial.print(hallToMotor[i]);
    Serial.print(", ");
  }
  Serial.println();
}

/* This function takes a motorState (from 0 to 5) as an input, and decides
which transistors to turn on
 * dutyCycle is from 0-255, and sets the PWM value.
 *
 * Note if dutyCycle is zero, or if there's an invalid motorState, then it
turns all transistors off
 */

void writePWM(uint8_t motorState, uint8_t dutyCycle)
{
  if(dutyCycle == 0)                         // If zero throttle, turn all
off
    motorState = 255;

  if(motorState == 0)                        // LOW A, HIGH B
      writePhases(0, dutyCycle, 0, 1, 0, 0);
  else if(motorState == 1)                   // LOW A, HIGH C
      writePhases(0, 0, dutyCycle, 1, 0, 0);
  else if(motorState == 2)                   // LOW B, HIGH C
      writePhases(0, 0, dutyCycle, 0, 1, 0);
  else if(motorState == 3)                   // LOW B, HIGH A
      writePhases(dutyCycle, 0, 0, 0, 1, 0);
  else if(motorState == 4)                   // LOW C, HIGH A
      writePhases(dutyCycle, 0, 0, 0, 0, 1);
  else if(motorState == 5)                   // LOW C, HIGH B
      writePhases(0, dutyCycle, 0, 0, 0, 1);
  else                                       // All off
      writePhases(0, 0, 0, 0, 0, 0);
}

/* Helper function to actually write values to transistors. For the low sides,
takes a 0 or 1 for on/off
 * For high sides, takes 0-255 for PWM value
 */

void writePhases(uint8_t ah, uint8_t bh, uint8_t ch, uint8_t al, uint8_t bl,
uint8_t cl)
```

```
{
  analogWrite(AH_PIN, ah);
  analogWrite(BH_PIN, bh);
  analogWrite(CH_PIN, ch);
  digitalWriteFast(AL_PIN, al);
  digitalWriteFast(BL_PIN, bl);
  digitalWriteFast(CL_PIN, cl);
}

/* Read hall sensors WITH oversamping. This is required, as the hall sensor
readings are often noisy.
 * This function reads the sensors multiple times (defined by HALL_OVERSAMPLE)
and only sets the output
 * to a 1 if a majority of the readings are 1. This really helps reject noise.
If the motor starts "cogging" or "skipping"
 * at low speed and high torque, try increasing the HALL_OVERSAMPLE value
 *
 * Outputs a number, with the last 3 binary digits corresponding to hall
readings. Thus 0 to 7, or 1 to 6 in normal operation
 */

uint8_t getHalls()
{
  uint8_t hallCounts[] = {0, 0, 0};
  for(uint8_t i = 0; i < HALL_OVERSAMPLE; i++) // Read all the hall pins
repeatedly, tally results
  {
    hallCounts[0] += digitalReadFast(HALL_1_PIN);
    hallCounts[1] += digitalReadFast(HALL_2_PIN);
    hallCounts[2] += digitalReadFast(HALL_3_PIN);
  }

  uint8_t hall = 0;

  if (hallCounts[0] >= HALL_OVERSAMPLE / 2)     // If votes >= threshold, call
that a 1
    hall |= (1<<0);                             // Store a 1 in the 0th bit
  if (hallCounts[1] >= HALL_OVERSAMPLE / 2)
    hall |= (1<<1);                             // Store a 1 in the 1st bit
  if (hallCounts[2] >= HALL_OVERSAMPLE / 2)
    hall |= (1<<2);                             // Store a 1 in the 2nd bit

  return hall & 0x7;                            // Just to make sure we didn't
do anything stupid, set the maximum output value to 7
}

/* Read the throttle value from the ADC. Because our ADC can read from 0v-
3.3v, but the throttle doesn't output over this whole range,
```

```
 * scale the throttle reading to take up the full range of 0-255
 */

uint8_t readThrottle()
{
  int32_t adc = analogRead(THROTTLE_PIN); // Note, analogRead can be slow!
  adc = (adc - THROTTLE_LOW) << 8;
  adc = adc / (THROTTLE_HIGH - THROTTLE_LOW);

  if (adc > 255) // Bound the output between 0 and 255
    return 255;

  if (adc < 0)
    return 0;

  return adc;
}
```

MOTOR CONTOLLER FLOW DIAGRAM