CPE 316 Final Report

Author: Ryan Cramer

In Collaboration With: Avery White

Table of Contents:

I. Project Description

In this project, we implement a SVPWM (space vector pulse width modulation) brushless DC motor electronic speed controller.

To accomplish this, we use the inverse Park-Clarke transform in software to map the desired 3 phase voltages into oscillating PWM duty cycles which control the motor windings. This is to have positional control of the motor. We use the STM32L476RGT6 with TIM1 configured as our PWM, with 6 output channels (one for each high and low side MOSFET). We had to create power circuitry to control the driving of the motor, which required off-board gate driver ICs, N-Channel MOSFETs, and general passive components.

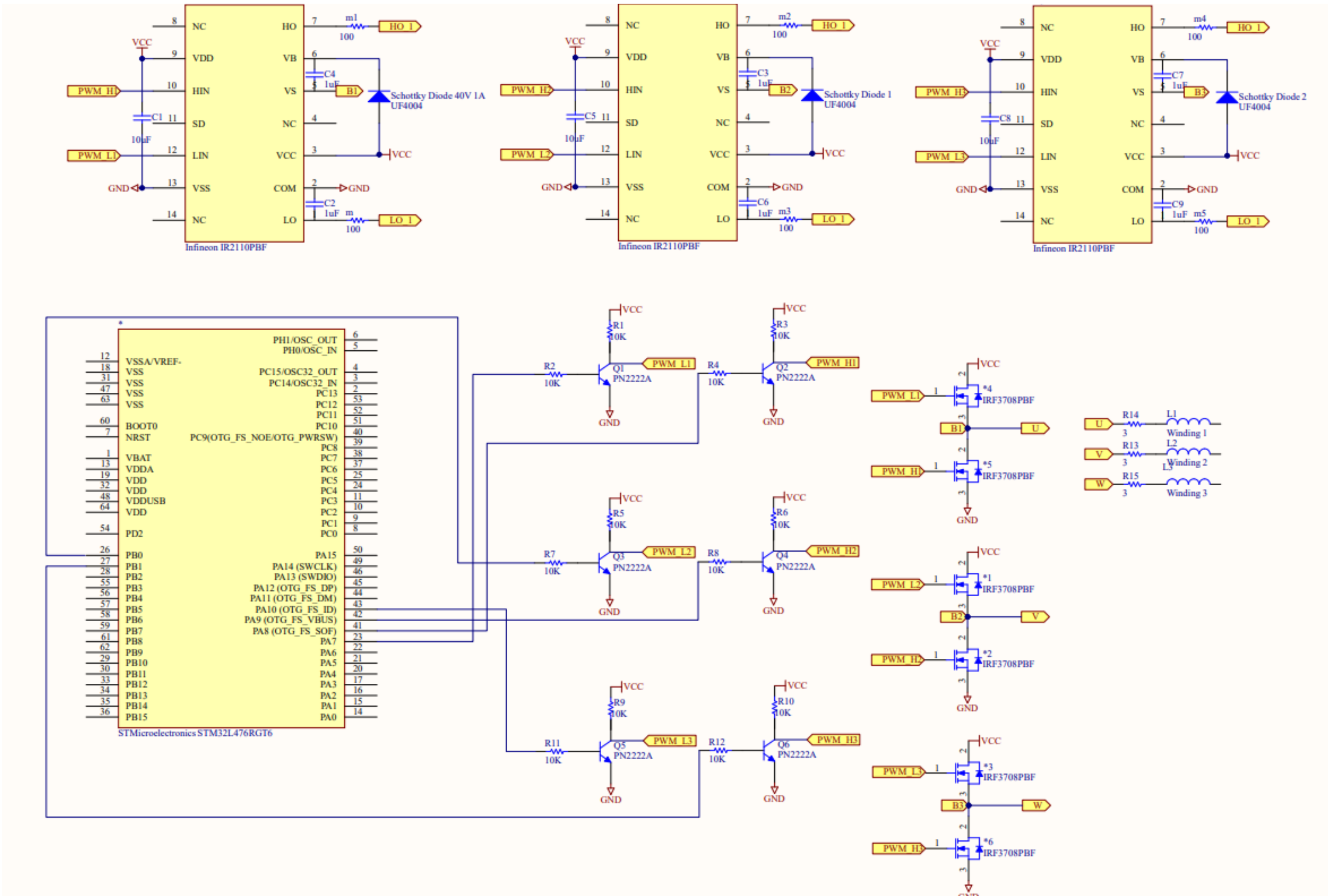
II. Project Implementation

a. Hardware Design

B.O.M. –

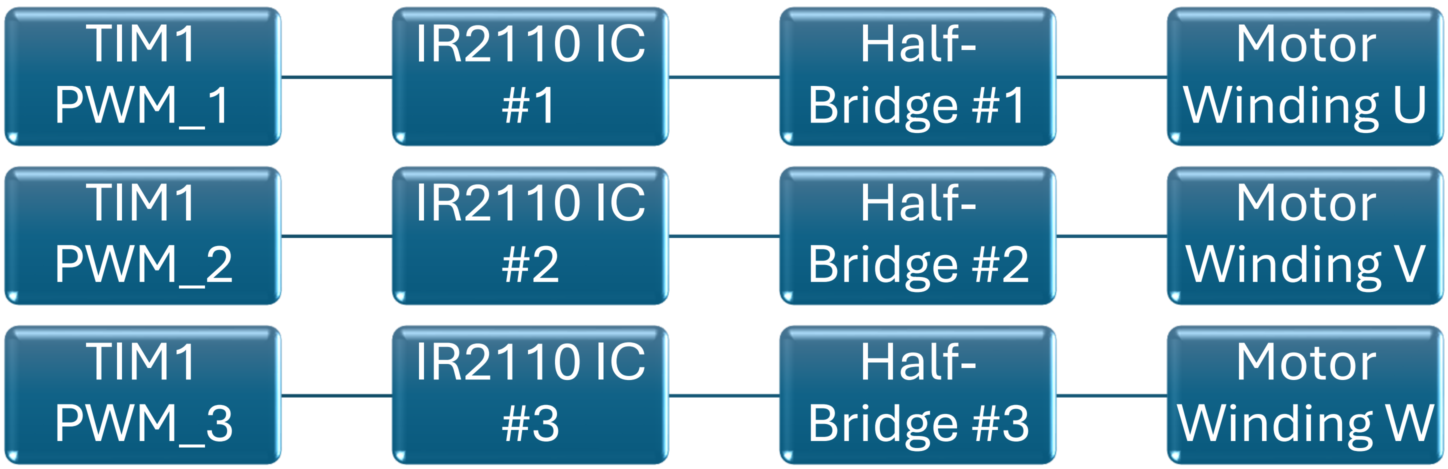
1. 3x Infineon IR2110 Gate Driver IC
2. 1x Adafruit Protoboard
3. 1x Turnigy 2200kV BLDC Drone Motor
4. 6x Infineon IRF-3708 40V 3A MOSFET
5. 6x PN2222A NPN Transistor
6. 6x 100uF ceramic capacitor
7. 3x 10uF ceramic capacitor
8. 1x STM32 L476RGT6 Nucleo Board
9. 1x 30V/3A DC Power Supply
10. 3x 40V 1A Schottky Diode

Schematic:



In the schematic diagram, we see the STM32 board sending 6 PWM signals to the gate of 6 NPNs, which have a V_{col} of +12V, to shift the +3.3V of the STM32 to the desired IR2110 V_{cc} of +12V. We then pass the NPN collector output to the IR2110, which drives a clean 12V logic HI and LO to the 3 half-bridges (comprised of 2 N-Channel MOSFETs each). We then take the output of the half-bridges (the connection between the high-side drain and the low-side source) to the 3-Phase BLDC motor windings.

Black Box Diagram:



As you can see from the black box diagram, we have inverted PWM pairs (with small 5% deadtime inserted) which go to their corresponding IR2110 gate driver, which drives the half-bridge, which drives the corresponding motor winding. This all relies on the correct PWM being driven at the correct frequency.

Not shown here:

- NPN level shifting for IR2110 gate driver
- Power supply connection:
  (12V 2A to MOSFETS, 12V 10mA to IR2110 Gate Driver)
  (5V to STM32 Nucleo Board)
- PWM_X comprised of PWM_X_HI and PWM_X_LO

b. Software Design

Description:

sample potentiometer input
adjusts motor drive characteristics using SVPWM
Timer1 synchronizes PWM, main loop adjusts PWM and LCD based on the potentiometer input

Initialization (fill lookup tables once at startup)

```
for (int i = 0; i < 360; ++i) {
    float rad = (float)i * M_PI / 180.0f;
    cos_table[i] = cosf(rad); // precompute cosines
    sin_table[i] = sinf(rad); // precompute sines
}
```

Interrupt-driven timing and ADC acquisition

```
// ADC end-of-conversion interrupt
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc) {
     flags |= ADC_EOC; // signal that a new sample is ready
     HAL_ADC_Stop_IT(hadc); // halt until restarted in main
}

// TIM2 tick every 1 ms for software timers
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM2) {
         TIMER2_HANDLE(); // decrement sTimer[] entries
    }
}
```

Align rotor at startup

```
static void AlignRotor(void) {
SetDutyCycles(ALIGN_DUTY_HIGH, ALIGN_DUTY_LOW, 50); // static align
HAL_Delay(ALIGN_DURATION_MS); // wait 500 ms
}
```

ComputeSVPWMDuties() → Performs the inverse Park+Clarke transform
• Converts voltage vector (α, β) into 3-phase voltages (Va, Vb, Vc)
• Normalizes and scales to duty cycles (0–100%)

```
static void ComputeSVPWMDuties(int theta, float magnitude, uint16_t *pdutyA,
uint16_t *pdutyB, uint16_t *pdutyC) {
float v_alpha = magnitude * cos_table[theta];
float v_beta = magnitude * sin_table[theta];
```

SetDutyCycles() → Updates TIM1 compare registers

- Reads PWM auto-reload value (ARR)
- Converts duty % into compare values
- Writes compare registers for channels 1–3

```
static void SetDutyCycles(uint16_t dutyA_pct, uint16_t dutyB_pct, uint16_t
dutyC_pct) {
uint32_t arr = __HAL_TIM_GET_AUTORELOAD(&htim1);
uint16_t cmpA = (arr * dutyA_pct) / 100;
uint16_t cmpB = (arr * dutyB_pct) / 100;
uint16_t cmpC = (arr * dutyC_pct) / 100;
__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, cmpA);
__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, cmpB);
__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, cmpC);
}
```

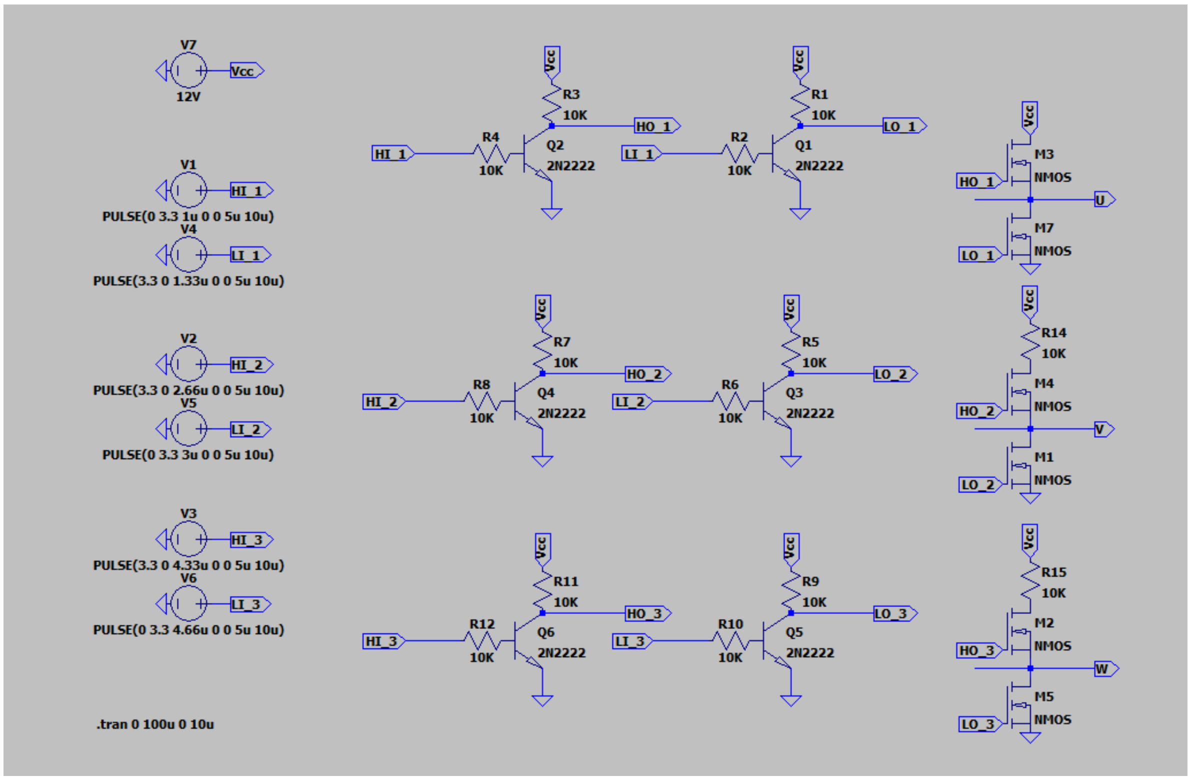
The firmware uses one-millisecond TIM2 ticks and ADC end-of-conversion interrupts to drive a nonblocking SVPWM control loop: it samples a potentiometer via the ADC, low-pass filters the result, and maps the voltage ratio to both a modulation magnitude (0.3–0.75) and an angle step (3°–15°/ms). At startup it pre-aligns the rotor with fixed 75/25% duties for 500 ms, then in each 1 ms update computes the Clarke-Park inverse transform using 360° sine/cosine lookup tables, generates centered three-phase voltages (Va, Vb, Vc), scales them to 0–100% duty, and updates TIM1 compare registers for complementary PWM outputs. The LCD is refreshed every 200 ms to display the filtered voltage, and all peripheral setup and ISR wiring is handled via _Init() calls to keep the application focused on timing, control math, and duty updates.

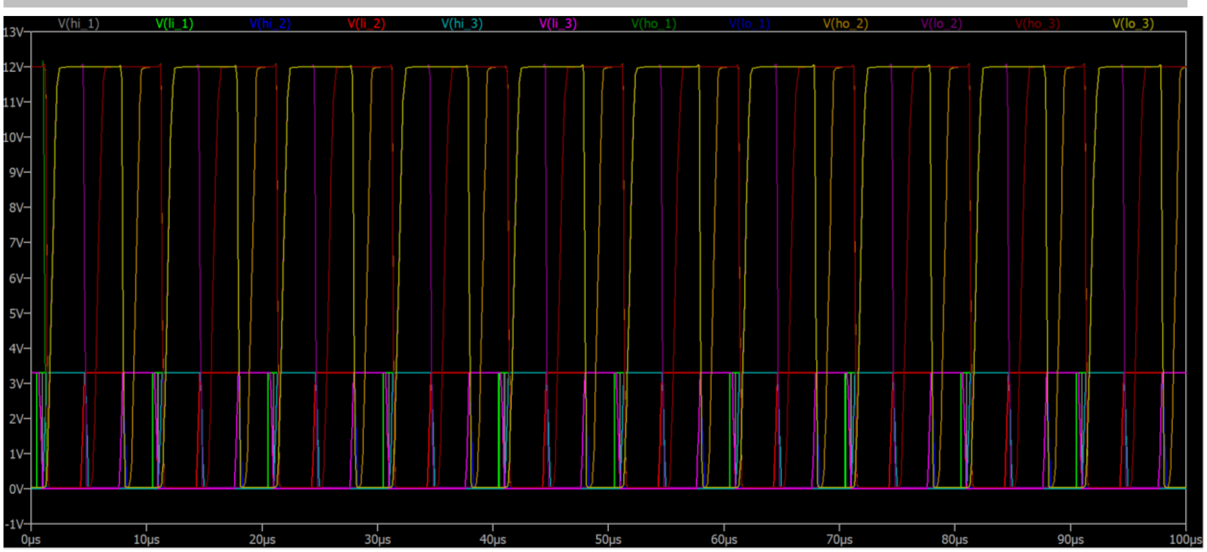
III. Testing and Demonstration:

Here is the link to the demo: [demo video]

a. Hardware Testing

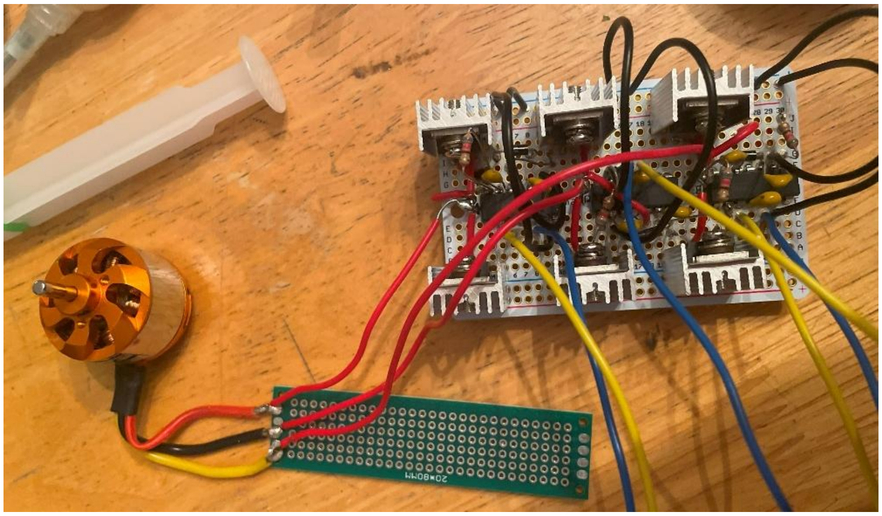
To first understand how the system worked, we implemented an LTSpice simulation of the hardware, to analyze how the 3-phase windings would be driven by the PWM logic with the deadtime.





We can see that the 3.3 Volt logic is correctly shifted up by the NPNs driving the three half-bridge windings in our LTSpice simulation.

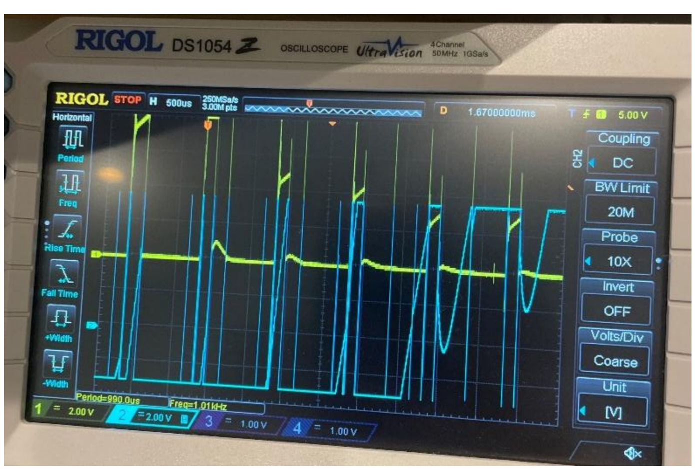
Once we confirmed the simulation of the hardware worked, we had to implement it on the breadboard/protoboard. This required continuous testing of shorts using a RIGOL DMM in continuity mode. We had to probe separate pins to make sure that they were not shorting separate rows after we soldered.



After implementing the hardware and testing connections, it was time for implementation testing.

In this clip, we demonstrate the PWM from the STM32 working correctly:
https://www.youtube.com/shorts/36C-fn4Lms8

Once we connected the STM32 to the IR2110 and the IR2110 to the windings, we had to monitor the voltages going to the windings. Here are some pictures of seeing the (rather shaky) 12V voltage applied to the motor winding.



b. Software Testing

To test the software, we gathered no visual results, but we compiled a list of testing that occurred to get the software to work.

In initial testing, the angles would jump by a large amount, which was odd, but we realized that we were computing the angle as an integer, so it had 360 discrete steps. Since we opted for finer angle control, we adjusted the angles to floats for greater precision

At first, the motor was turning 1 cycle per 5 seconds, and we realized it was because we had such a low PWM frequency (3Hz). Initially, we thought the motor would need a start phase where we start at a low frequency, then increase frequency as we increase duty cycle, but we actually figured out that we could just set hard frequencies with changing duty cycles, and that drove the motor better. The final PWM driving speed we settled on was 100kHz.

We made the firmware robust by first isolating each algorithmic block in unit tests: the 90/10 low-pass filter was exercised with step-and-sine inputs to confirm its time constant, and ComputeSVPWMDuties was run over θ=0…359° and magnitude=0.3–0.75 to verify that all duty outputs stayed within 0–100% and that wrap-around logic for the angle index worked without error. On the STM32 we added simple serial-print debug statements and LED-toggle markers in the TIM2 ISR so we could observe in a terminal and by eye that our 1 ms update cadence was rock-solid and that mv_filtered tracked the ADC input smoothly from 0 to full scale. Finally, the LCD provided live verification of potentiometer readings and filtered values at runtime, giving us confidence that the software logic performed correctly under all operating conditions.

IV. Collaboration and Teamwork

Avery and I joined forces late into the project, due to switching projects from the Landscaping to the SVPWM motor, therefore Avery had designed most of the software design when I came to him. However, he had not built or tested it. After meeting several times, we collaborated to build the final circuit, as well as discuss purchasing of components.

The first thing Avery did was configure the .ioc file for the project, and write the script. After joining the project, I tested PWM coming from the STM and the IR2110 gate drivers. After realizing that the breadboard prototype would ot suit our needs, we met and soldered the components together. At first, Avery's design did not work, so most of the testing was done on my circuit. Avery also conducted his own electrical tests.

Both Avery and I debugged the software and the hardware, both facing separate challenges. On one end, Avery was using an archived version of CubeMX, so the configuration file was different, and had to be migrated to work with my version. Avery's protoboard did not work, he had multiple shorts, but mine worked after soldering, though my wiring was incorrect at first and had to be changed.

After Avery left for ROTC, I compiled the presentation and demo videos for class, as well as this report. I presented by myself, though Avery contributed most of the final design, except for a few tweaks to the software and passive component values changed.

Avery frequently communicated with me throughout the entire process and worked with me on the github at this repository:
https://github.com/ryancramuh/SVPWM-STM32-Motor-Project

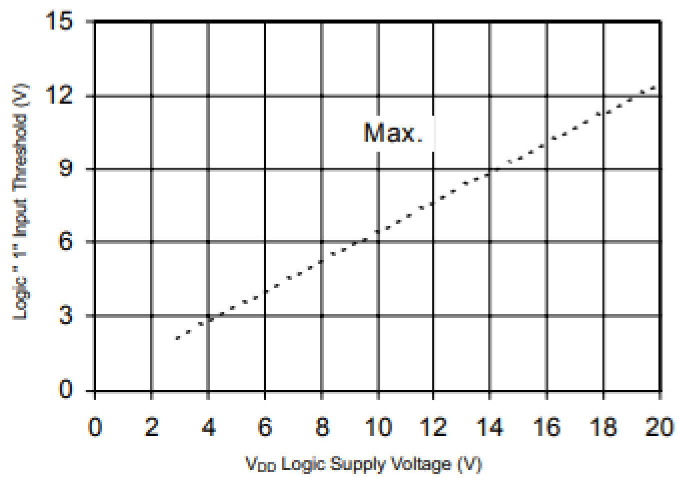
V. Troubleshooting and Debug

In this section we will present several challenges, and how we resolved them as they appeared:

1. SVPWM Duty Cycles needed dead time

   To correctly drive SVPWM, you need dead time, which means that although the high side and low side are inverted, they overlap. Therefore, we had to configure dead time, which was a challenge, until we found the built in dead time configuration settings in the .ioc.

2. IR2110 Gate Driver has changing logic level with Vcc



   Therefore, we had to add PN2222A NPN transistors to shift the logic level from 3.3 Volts to 12 Volts, due to 3.3 Volts not being valid with a Vcc of 12 Volts, it needs switching logic of 7.5 Volts at that supply voltage. After adding the NPNs, the design worked correctly.

3. 3-Phase windings can be wired wrong

   Avery did not encounter this issue, however I did. When I first hooked up the motor to test, it began bouncing around and did not spin at all. After swapping the half bridge output of two of the windings, the motor spun.

4. MOSFET Heat from switching frequency
   After changing the STM to output 100kHz PWM instead of 3Hz, it was immediately noticeable that the MOSFETs got extremely hot. Therefore, we had to add TO-220 package heat sinks, which kept the MOSFETs cool during operation, even at their maximum 3A rating.

5. Dtheta Resolution
   Originally, we calculated the angle using an integer, but for dtheta, the change in theta, the number was very small, and therefore needed to be precise. After some testing, we changed the dtheta and theta angle variables to floats, however they would not compute fast enough for our calculations, and had to precompute LUTs to save computation.

6. BLDC Motor Power Challenges
   The 2200 kV Turnigy drone motor we used barely spun at 12V 1A supply to the MOSFETs, despite being a “small” drone motor. After consulting the datasheet, we found that to get any useful spin on the motor, you need at least 7 volts, which can be accomplished with 6 amps. Because the motor is a huge power suck, and the supply we used only had 30V/3A, we attempted to get the maximum rating we could while maintaining safety with small 22 AWG jumper wires (rated for up to 2A). After supplying 2.85 amps, the wires got hot, but didn’t melt their plastic sheet, and the motor spun much better.

VI. Lessons Learned

1. Thoroughly studying every component's datasheet before you begin a design is indispensable. Datasheets provide critical information about pin configurations, timing diagrams, and voltage and current limits; neglecting them can lead to wasted time chasing obscure issues that the manufacturer has already documented. By understanding the recommended operating conditions, any quirks or special requirements, and the context of each specification, you can integrate parts smoothly and avoid surprises during prototyping.

2. It may sound obvious, but parts are engineered to be easy to hook up—if something feels unnecessarily hard, it's usually a sign you've missed a detail. Manufacturers design breakout boards, reference schematics, and evaluation kits precisely to simplify the connection process. When you lean on these resources and follow the proven wiring examples, you minimize the chance of introducing wiring errors or spending hours debugging a misconnected pin.

3. Ordering parts at the last minute is a recipe for compromise. With global supply chain pressures, many components are in short supply or on long lead times. If you defer ordering, you risk settling for whatever is available—often obsolete, counterfeit, or of lower quality—and that can derail your entire project. By planning ahead and placing orders early, you give yourself the flexibility to find suitable alternatives or redesign around a different footprint without panic.

4. Verifying your power requirements before soldering is critical to both performance and safety. A circuit that draws more current than its traces or regulators can handle may run hot, behave erratically, or even fail

catastrophically. By calculating the expected voltage and current demands of each subsystem and selecting regulators and copper wire accordingly, you ensure capable operation from day one and avoid the headache of desoldering and board rework.

5. CircuitMaker offers a powerful, no-cost entry point to professional PCB design, mirroring many of Altium Designer's capabilities. Using a free tool lets you practice advanced layout techniques such as interactive routing and design rule checking without the high up front license cost.

6. Choosing common, well-supported parts simplifies simulation, schematic capture, and footprint creation. Popular microcontrollers, op-amps, and passive components are more likely to have ready-to-use SPICE models, verified libraries, and community-driven design examples. This reduces the time you spend writing custom models or debugging library errors, allowing you to focus on the unique aspects of your design rather than reinventing generic building blocks.

7. A solid grounding in physics is necessary. Understanding electromagnetic theory, semiconductor behavior, and thermodynamics allows you to predict how circuits will respond under actual conditions. When you know why a filter's cutoff frequency shifts with temperature or how switching losses arise in a MOSFET, you can make better informed trade-offs and make better designs in general.

VII. Future Possibility

Building on the SVPWM-STM32-Motor-Project (https://github.com/ryancramuh/SVPWM-STM32-Motor-Project), the next step is to integrate real-time back-EMF sensing. By sampling the motor's phase voltages during PWM off-times, we can implement sensorless rotor position estimation and closed-loop speed control without external Hall sensors. This will improve efficiency and reduce wiring complexity, making the system more robust for a wider range of motors.

To drive higher-power applications, the design must be scaled to use 18 AWG magnet wire capable of handling up to 10 A continuous current. This involves selecting appropriately rated power MOSFETs, beefing up gate-driver circuitry, and verifying thermal management strategies such as heatsinking or forced airflow. Strengthening the PCB layout for low impedance current paths and incorporating Kelvin-sense routing will ensure minimal voltage drop and clean commutation waveforms under high load.

Initial validation should be done with through-hole components on a breadboard or perfboard prototype. This allows rapid iteration and easy modification while confirming clean motor startup, steady-state operation, and fault recovery across temperature extremes and supply variations. Key tests include measuring phase current waveforms on an oscilloscope, verifying PWM timing accuracy, and confirming that the low-pass filter and lookup table algorithms respond correctly to changing loads.
Once the through-hole prototype demonstrates reliable performance, the final step is to transition to an SMD-based PCB. Equivalent surface-mount MOSFETs, drivers, and passive components should be chosen for their thermal characteristics and ease of assembly. A custom PCB will reduce parasitics, improve EMI performance, and shrink the overall form factor, making the design suitable for embedded and commercial applications.

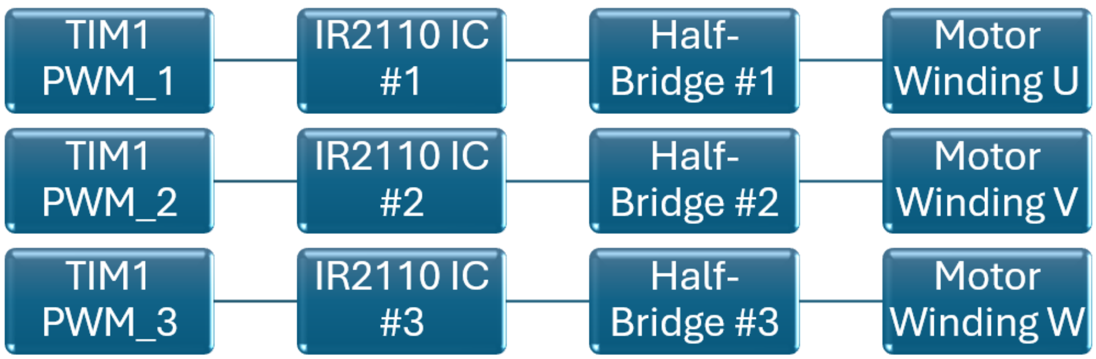
VIII. Summary and Conclusion

In this project, we successfully designed and implemented a space-vector PWM brushless DC motor controller using an STM32L476RGT6 microcontroller, off-board IR2110 gate drivers, and discrete MOSFETs on a custom protoboard. Our firmware leverages one-millisecond TIM2 ticks and ADC interrupts to sample a potentiometer, apply a 90/10 low-pass filter, map the filtered voltage into modulation magnitude and angular step, and compute three-phase duty cycles via an inverse Clarke–Park transform using 360° lookup tables. Hardware validation began in LTSpice, progressed through continuity and logic-level checks on the breadboard, and culminated in live demonstrations showing clean PWM driving of the motor and real-time LCD feedback. Collaboration between Avery and I enabled rapid iteration on both the .ioc configuration and the soldered prototype, while systematic troubleshooting—ranging from dead-time configuration to NPN level shifting and winding verification—ensured robust operation under varied conditions.

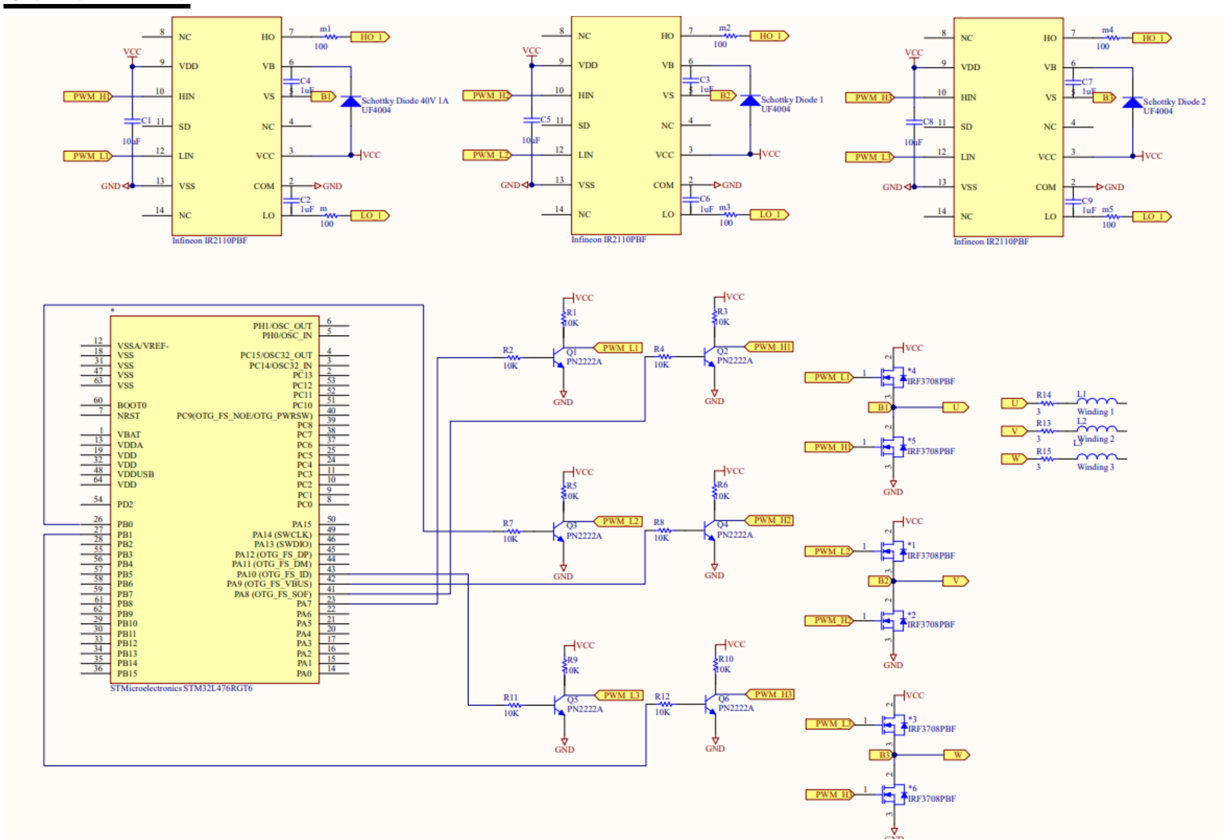
Looking ahead, the project laid a solid foundation for integrating sensorless back-EMF estimation, scaling to higher currents with 18 AWG wiring and power-rated MOSFETs, and transitioning to a compact SMD PCB for improved EMI and thermal performance. The experience reinforced the importance of datasheet mastery, early part sourcing, and physics-based design decisions. Overall, our work demonstrates a clear path from concept through prototype to a production-ready motor controller, and highlights both the practical challenges and rewarding successes of end-to-end embedded power electronics development.

IX. Appendix

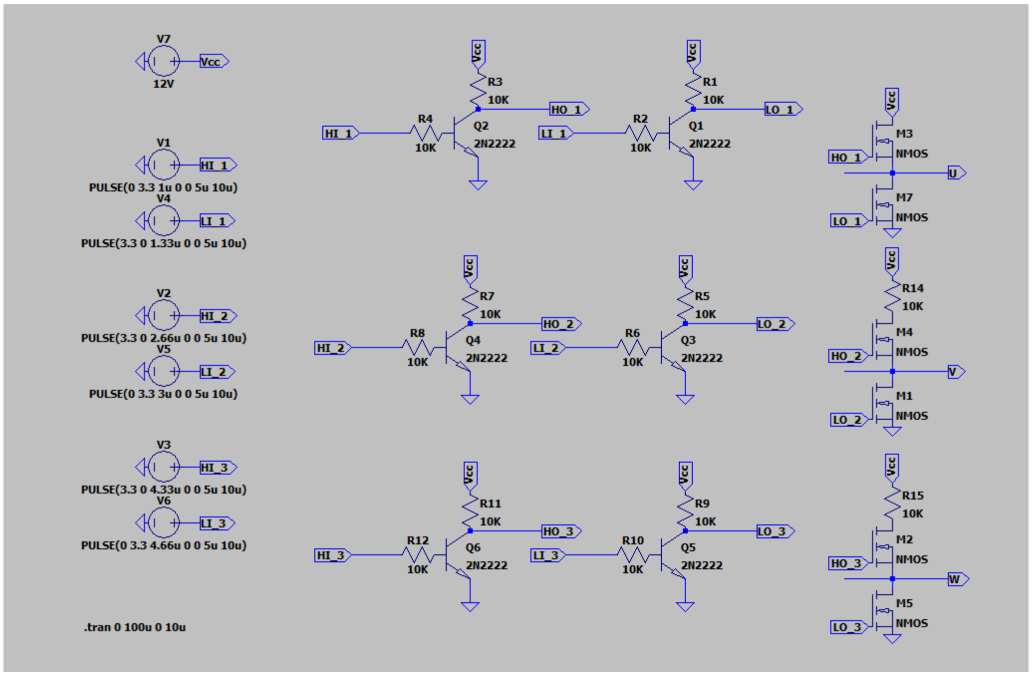
Block Diagram



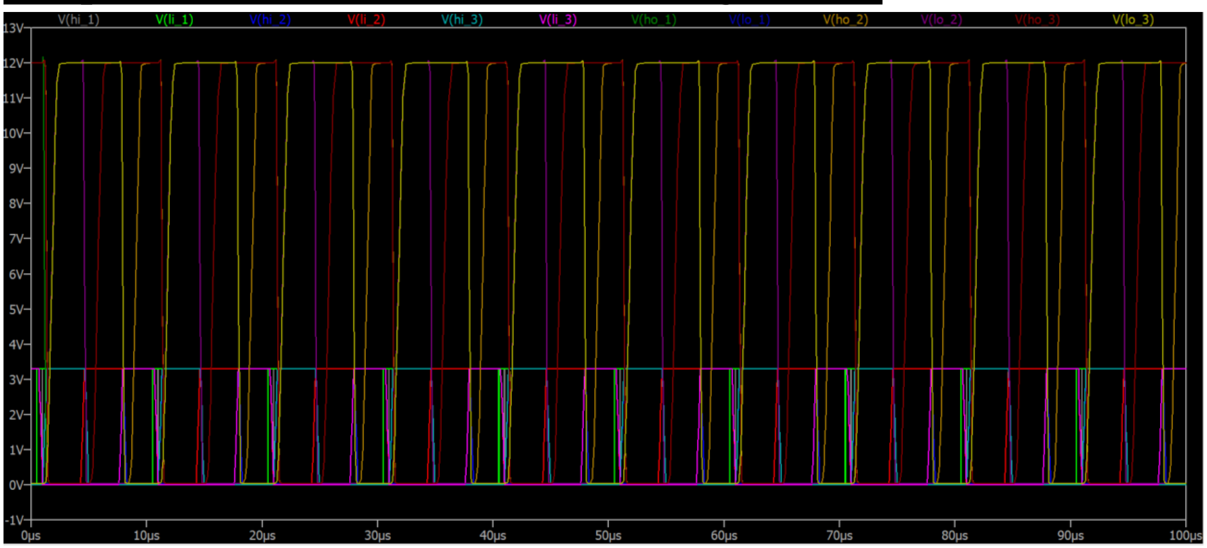
Schematic



LTSpice Basic Schematic (3.3V Level Shifting to 12V)

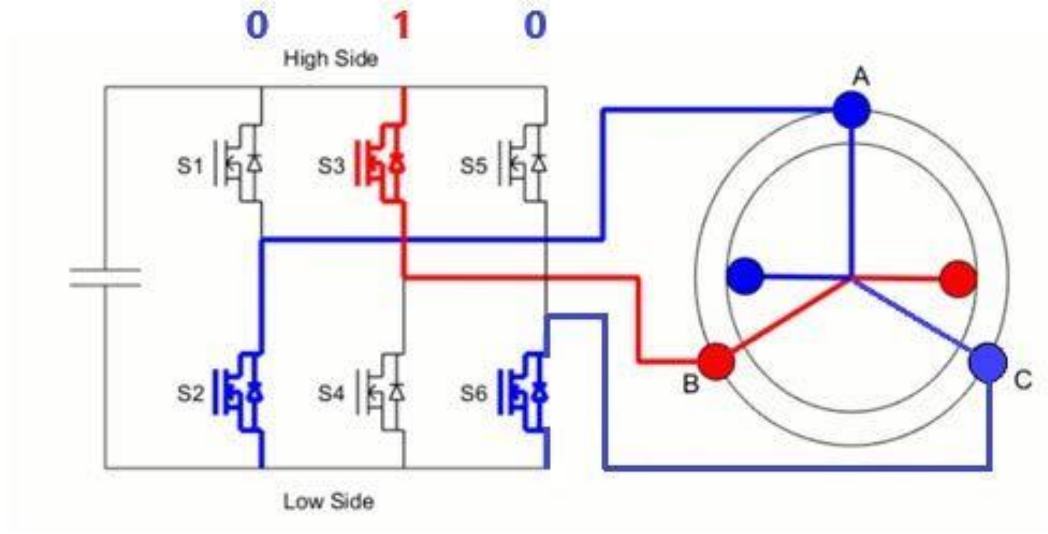


LTSpice Simulation (3.3V Level Shifting to 12V)

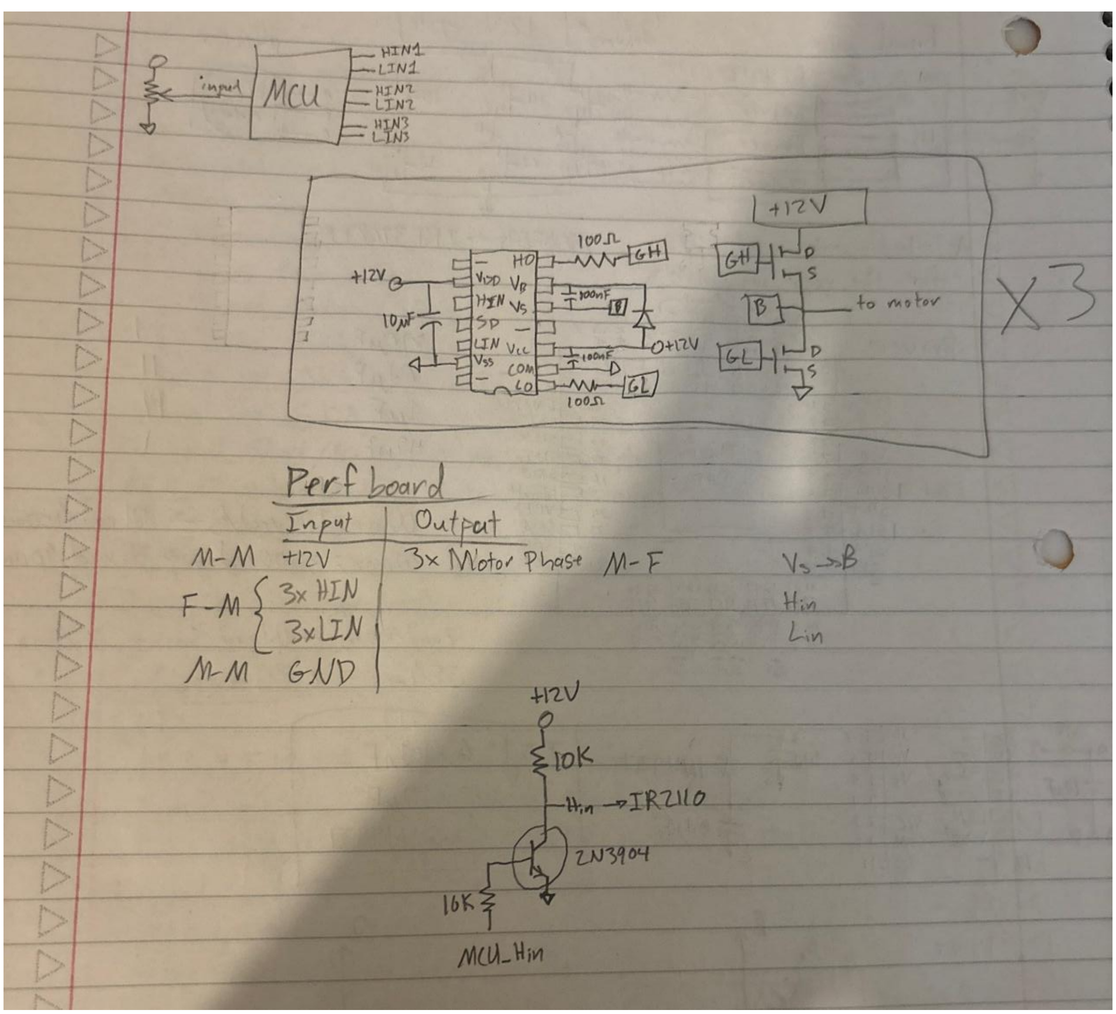


Motor Winding Capture:

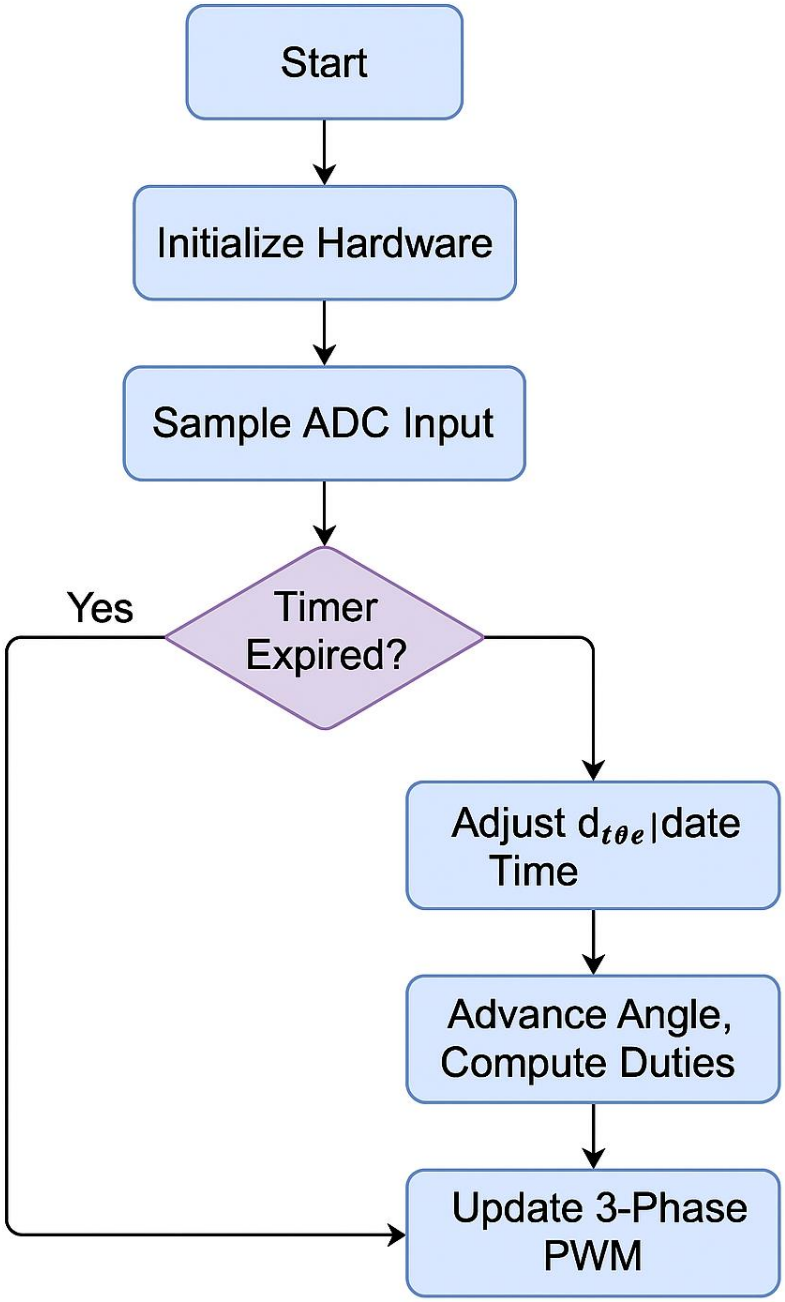
3-Phase SVPWM Connection



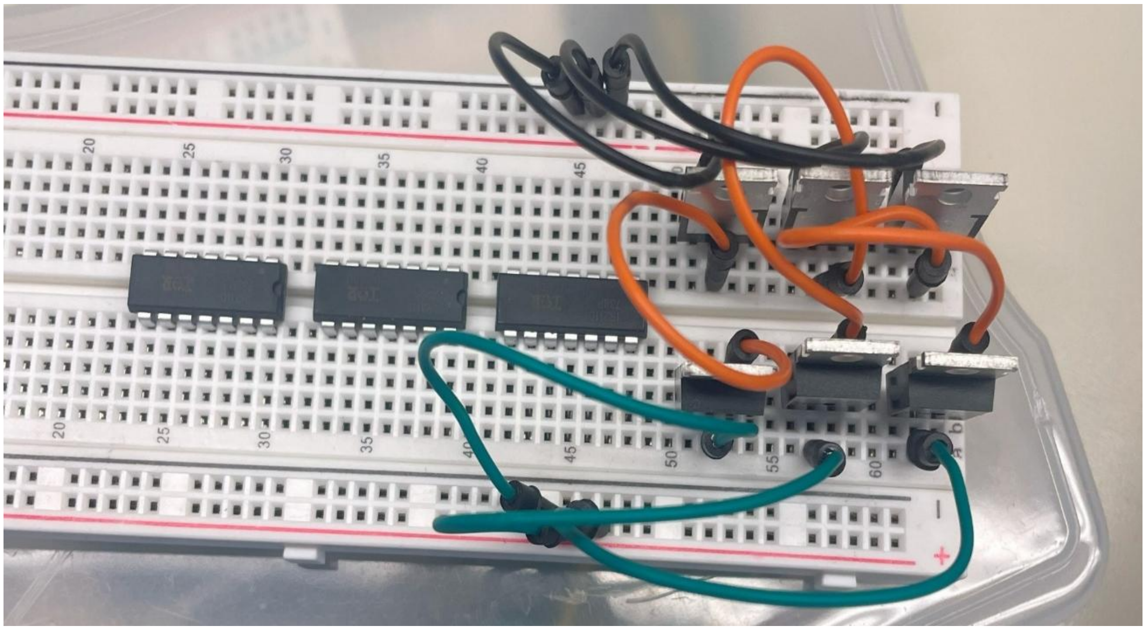
Avery's Original Schematic



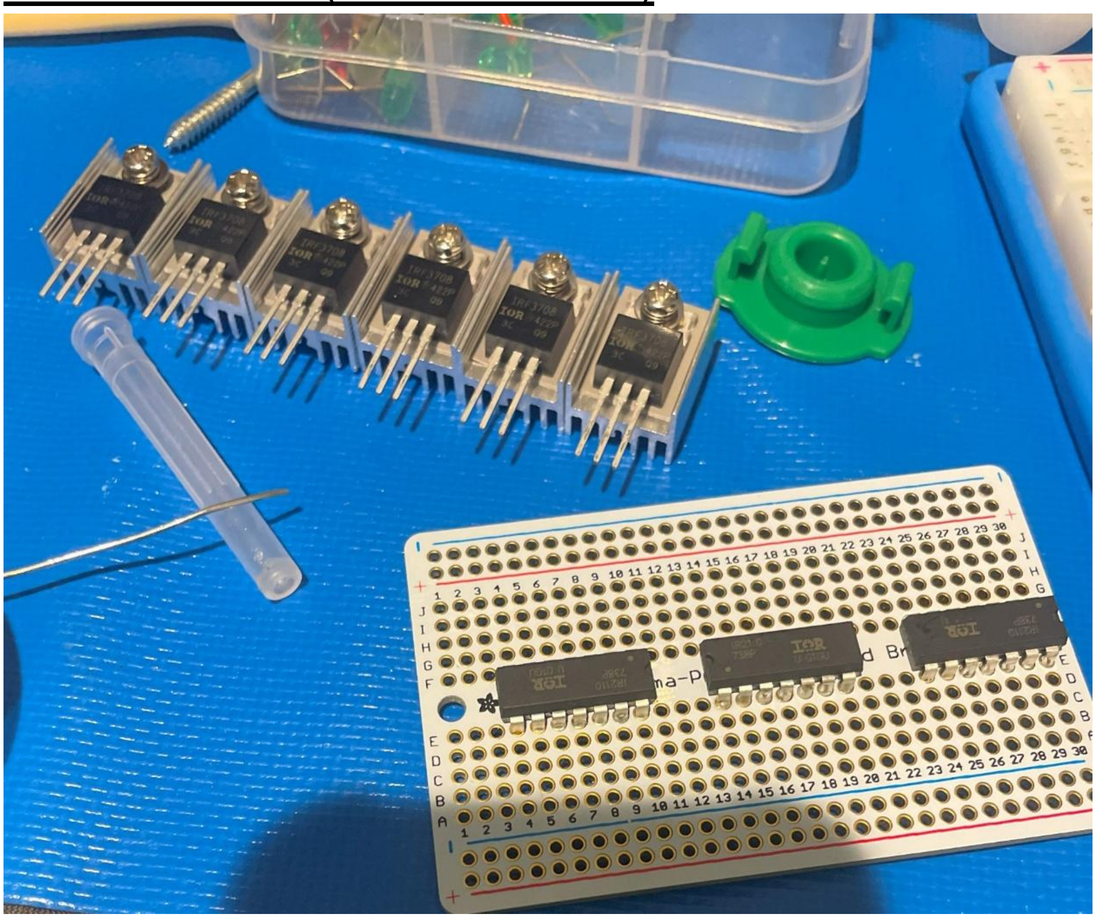
Software Flow Diagram



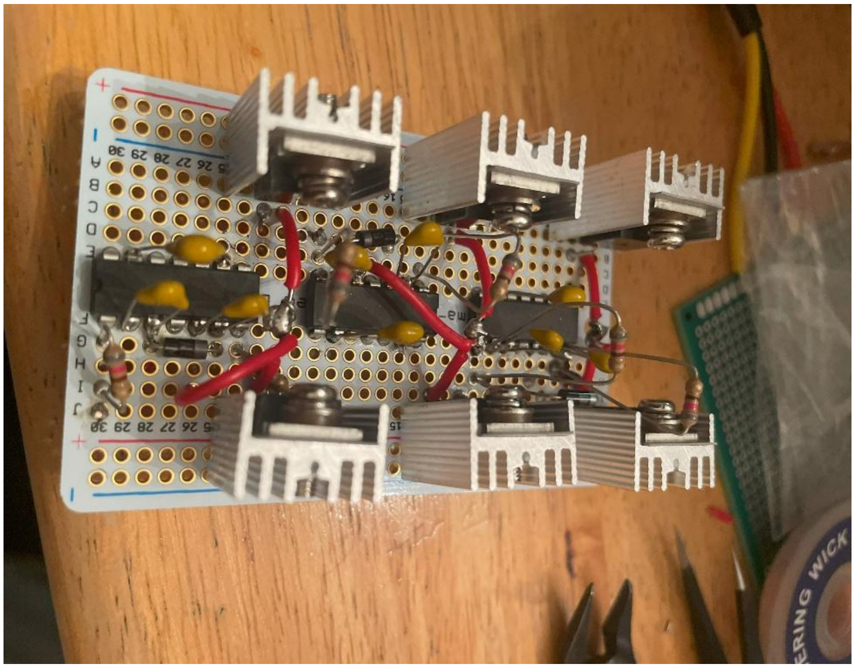
Avery Prototype



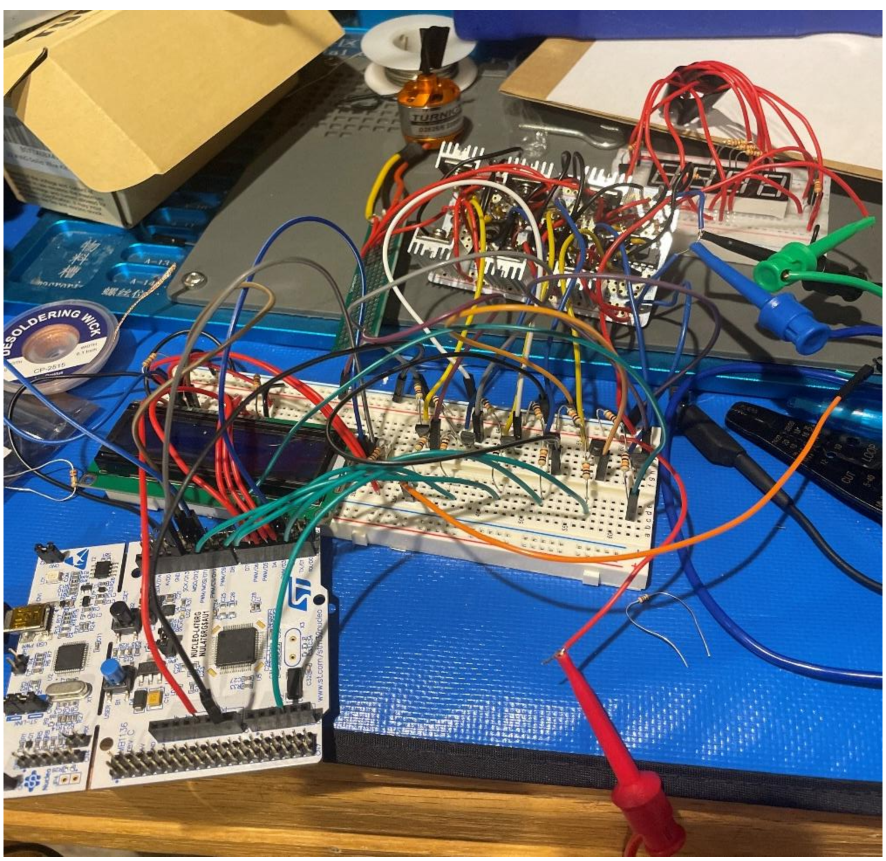
Perfboard Initial (added Heatsinks)



Perfboard Added Passives and Gate Drivers



Perfboard Hooked up to Motor and STM32 and Power Supply



Final Software Version

```c
#include "main.h"
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include "LCD.h"
#include "Timer.h"

#define TWO_PI (2.0f * M_PI)
#define SQRT3_OVER_2 0.86602540378f
#define ADC_EOC 0x01

#define MIN_DTHETA 3
#define MAX_DTHETA 15
#define MIN_MAG 0.3f
#define MAX_MAG 0.75f

#define ALIGN_DUTY_HIGH 75
#define ALIGN_DUTY_LOW 25
#define ALIGN_DURATION_MS 500

float mv = 0.0f, mv_filtered = 0.0f;
unsigned short flags = 0;
float cos_table[360], sin_table[360];

ADC_HandleTypeDef hadc1;
TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);

static void ComputeSVPWMDuties(int theta, float magnitude, uint16_t *dA, uint16_t *dB,
uint16_t *dC);
static void SetDutyCycles(uint16_t dA, uint16_t dB, uint16_t dC);
static void AlignRotor(void);

// ADC complete ISR
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc) {
    flags |= ADC_EOC;
    HAL_ADC_Stop_IT(hadc);
}

// TIM2 ISR for sTimers
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    TIMER2_HANDLE();
}

int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_TIM1_Init();
    MX_TIM2_Init();
    HAL_TIM_MspPostInit(&htim1);
```

```c
    // Init peripherals
    HAL_ADC_Start_IT(&hadc1);
    HAL_TIM_Base_Start_IT(&htim2);
    htim1.Instance->BDTR |= TIM_BDTR_MOE;

    for (int i = 0; i < 360; ++i) {
        float rad = (float)i * M_PI / 180.0f;
        cos_table[i] = cosf(rad);
        sin_table[i] = sinf(rad);
    }

    const uint32_t channels[] = {TIM_CHANNEL_1, TIM_CHANNEL_2, TIM_CHANNEL_3};
    for (int i = 0; i < 3; ++i) {
        HAL_TIM_PWM_Start(&htim1, channels[i]);
        HAL_TIMEx_PWMN_Start(&htim1, channels[i]);
    }

    // LCD startup
    LcdInit(); LcdClear();
    LcdPutS("BLDC Control");
    LcdGoto(1, 7); LcdPutS("mV");
    LcdWriteCmd(0x0C);

    AlignRotor();  // Pre-alignment before rotation

    int theta = 0;
    float magnitude = MIN_MAG;
    int dtheta = MIN_DTHETA;
    uint16_t dutyA, dutyB, dutyC;

    while (1) {
        // Handle ADC update
        if (flags & ADC_EOC) {
            uint32_t raw = HAL_ADC_GetValue(&hadc1);
            mv = (raw * 3300.0f) / 4095.0f;
            mv_filtered = 0.9f * mv_filtered + 0.1f * mv; // low-pass filter
            flags &= ~ADC_EOC;
            HAL_ADC_Start_IT(&hadc1);
        }

        // Update control loop
        if (sTimer[UPDATE_PWM_TIMER] == 0) {
            // Map ADC to dynamic params
            float ratio = mv_filtered / 3300.0f;
            magnitude = MIN_MAG + ratio * (MAX_MAG - MIN_MAG);
            if (magnitude > MAX_MAG) magnitude = MAX_MAG;

            dtheta = MIN_DTHETA + (int)(ratio * (MAX_DTHETA - MIN_DTHETA));
            if (dtheta > MAX_DTHETA) dtheta = MAX_DTHETA;

            theta = (theta + dtheta) % 360;
            ComputeSVPWMDuties(theta, magnitude, &dutyA, &dutyB, &dutyC);
            SetDutyCycles(dutyA, dutyB, dutyC);

            sTimer[UPDATE_PWM_TIMER] = 1; // 1 ms update cycle
        }

        // update LCD
        if (sTimer[UPDATE_LCD_TIMER] == 0) {
            char buff[10];
            sprintf(buff, "%7.2f", mv_filtered);
            LcdGoto(1, 0);
```

```c
            LcdPutS(buff);
            sTimer[UPDATE_LCD_TIMER] = 200;
        }
    }
}

static void AlignRotor(void) {
    SetDutyCycles(ALIGN_DUTY_HIGH, ALIGN_DUTY_LOW, 50);
    HAL_Delay(ALIGN_DURATION_MS);
}

static void ComputeSVPWMDuties(int theta, float magnitude, uint16_t *pdutyA, uint16_t
*pdutyB, uint16_t *pdutyC) {
    float v_alpha = magnitude * cos_table[theta];
    float v_beta = magnitude * sin_table[theta];

    float va = v_alpha;
    float vb = -0.5f * v_alpha + SQRT3_OVER_2 * v_beta;
    float vc = -0.5f * v_alpha - SQRT3_OVER_2 * v_beta;

    float vmax = fmaxf(fmaxf(va, vb), vc);
    float vmin = fminf(fminf(va, vb), vc);
    float v0 = 0.5f * (vmax + vmin);
    va -= v0;
    vb -= v0;
    vc -= v0;

    *pdutyA = (uint16_t)((va + 1.0f) * 50.0f);
    *pdutyB = (uint16_t)((vb + 1.0f) * 50.0f);
    *pdutyC = (uint16_t)((vc + 1.0f) * 50.0f);
}

static void SetDutyCycles(uint16_t dutyA_pct, uint16_t dutyB_pct, uint16_t dutyC_pct) {
    uint32_t arr = __HAL_TIM_GET_AUTORELOAD(&htim1);

    uint16_t cmpA = (arr * dutyA_pct) / 100;
    uint16_t cmpB = (arr * dutyB_pct) / 100;
    uint16_t cmpC = (arr * dutyC_pct) / 100;

    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, cmpA);
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, cmpB);
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, cmpC);
}


/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
```

```
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 1;
  RCC_OscInitStruct.PLL.PLLN = 10;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_MultiModeTypeDef multimode = {0};
  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Common config
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc1.Init.LowPowerAutoWait = DISABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
```

```c
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
  hadc1.Init.OversamplingMode = DISABLE;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure the ADC multi-mode
  */
  multimode.Mode = ADC_MODE_INDEPENDENT;
  if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure Regular Channel
  */
  sConfig.Channel = ADC_CHANNEL_10;
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
  sConfig.SingleDiff = ADC_SINGLE_ENDED;
  sConfig.OffsetNumber = ADC_OFFSET_NONE;
  sConfig.Offset = 0;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief TIM1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM1_Init(void)
{

  /* USER CODE BEGIN TIM1_Init 0 */

  /* USER CODE END TIM1_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};

  /* USER CODE BEGIN TIM1_Init 1 */

  /* USER CODE END TIM1_Init 1 */
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 399;
  htim1.Init.CounterMode = TIM_COUNTERMODE_DOWN;
  htim1.Init.Period = 99;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
```

```
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
  sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
  {
    Error_Handler();
  }
  sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
  sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
  sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
  sBreakDeadTimeConfig.DeadTime = 20;
  sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
  sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
  sBreakDeadTimeConfig.BreakFilter = 0;
  sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
  sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
  sBreakDeadTimeConfig.Break2Filter = 0;
  sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
  if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM1_Init 2 */

  /* USER CODE END TIM1_Init 2 */
  HAL_TIM_MspPostInit(&htim1);

}

/**
  * @brief TIM2 Initialization Function
  * @param None
```

```c
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 3999;
  htim2.Init.CounterMode = TIM_COUNTERMODE_DOWN;
  htim2.Init.Period = 19;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */

  /* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10|GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_11|GPIO_PIN_12, GPIO_PIN_RESET);
```

```c
  /*Configure GPIO pins : PB10 PB3 PB4 PB5 */
  GPIO_InitStruct.Pin = GPIO_PIN_10|GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*Configure GPIO pins : PA11 PA12 */
  GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_12;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */

  /* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

X. Reference and Bibliography

[1] R. Cramer and A. White, "SVPWM-STM32-Motor-Project," GitHub repository, Jan. 2025. Available: https://github.com/ryancramuh/SVPWM-STM32-Motor-Project

[2] Infineon Technologies AG, "IR2110: High- and Low-Side Driver," Datasheet v1.00, Jan. 2014. Available: https://www.infineon.com/dgdl/Infineon-IR2110-DataSheet-v01_00-EN.pdf?fileId=5546d462533600a4015355c80333167e

[3] Infineon Technologies AG, "IRF3708: N-Channel MOSFET," Datasheet v1.01, Jun. 2013. Available: https://www.infineon.com/dgdl/Infineon-IRF3708-DataSheet-v01_01-EN.pdf?fileId=5546d462533600a4015355df7cf5193c

[4] MathWorks, "Space Vector Modulation," Discovery page. Available: https://www.mathworks.com/discovery/space-vector-modulation.html

[5] HobbyKing, "Turnigy 2200kV BLDC Drone Motor Datasheet." Available: https://cdn-global-hk.hobbyking.com/media/file/1013956805X669553X55.pdf