FEASIBILITY STUDY OF OFF-THE-SHELF COMPONENTS
ON A SPLIT-CYCLE MOTOR AND ESC TESTBED

by

Hayden Cole Lotspeich

THESIS

Submitted in partial fulfillment of the requirements

for the degree of Master of Science in Computer Science at

The University of Texas at Arlington

December, 2023

Arlington, Texas

Supervising Committee:

Christopher McMurrough, Supervising Professor

Chris Conly

Shawn Gieser

ABSTRACT

FEASIBILITY STUDY OF OFF-THE-SHELF COMPONENTS ON A SPLIT-CYCLE MOTOR AND ESC TESTBED

Hayden Cole Lotspeich, Masters of Science in Computer Science

The University of Texas at Arlington, 2023

Supervising Professor: Christopher McMurrough

This project aims to create a testbed for split-cycle flapping wing systems that allows for testing of different motors and ESC protocols to find a suitable set for a flapping wing system. In order for a flapping-wing drone to be able to maneuver, it has to be able to flap its wings at different speeds when flapping forward and flapping backwards. The arching back-and-forth motion is what the output wing would be connected to, so this system is used to calculate the maximum split-cycle time ratio that can be achieved when set up with different motors and ESC protocols.

TABLE OF CONTENTS

ABSTRACT……………………………………………………………………………………………..ii

CHAPTER 1: INTRODUCTION…………………………………………………………………....1

CHAPTER 2: INSPIRATION…………………………………………………………………...…..4

CHAPTER 3: BACKGROUND……………………………………………………………………..6

Microcontrollers………………………………………………………………………………..6

PWM……………………………………………………………………………………………...8

ESCs and Motors……………………………………………………………………………...9

PID Controllers……………………………………………………………………………....11

Quadcopter Flight Design/Requirements……………………………………………….....13

Flapping Wing Design/Requirements…………………………………………………….…..15

CHAPTER 4: SYSTEM OVERVIEW……………………………………………………….......18

CHAPTER 5: WIRING LAYOUT…………………………………………………………….....23

CHAPTER 6: CODE……………………………………………………………………………......26

CHAPTER 7: SUBSYSTEMS/COMPONENTS……………………………………………...32

CHAPTER 8: EXPERIMENT PLAN…………………………………………………………...37

CHAPTER 9: EXPERIMENT RESULTS……………………………………………………...39

CHAPTER 10: CONCLUSION…………………………………………………………………..49

CHAPTER 11: FUTURE WORKS……………………………………………………………….51

REFERENCES…………………………………………………………………………………..53

CHAPTER 1

INTRODUCTION

In the world of aviation, flight systems must be checked numerous times in order to be determined as fitting for a certain type of aircraft. Each system must go through multiple tests to determine if it is a good fit for the system being created. One type of aircraft that has seen a boost in popularity lately is the UAV (unmanned aerial vehicle, usually referred to as drones).

Much like commercial and private airplanes, drone systems must go through many tests in order to figure out the best flight system to use for whatever task is attempting to be accomplished. Drones are made up of many different subsystems, each pertaining to a certain task the drone is being created to be able to perform. One of the most common systems almost every drone has is a flight subsystem.

The flight subsystem is what the drone uses in order to be able to fly. These typically consist of a flight controller, ESCs (electronic speed controllers), motors, and some form of wing (usually a propeller). The flight controller is what receives directional commands from the pilot. It then transforms these commands into ESC speed control commands and sends them out to the ESCs on the drone. The ESCs take in the speed control commands and turn those into motor speed commands and send those to the motors. The motors then spin at the specified speed, which turns the propellers (or wings), which then spin and generate lift for the drone to allow it to fly.

There are many different types of flight subsystems used in drones. There are multi-rotor drones, fixed-wing drones, and single-rotor drones, just to name a few. The most common type of drone is the multi-rotor drone. There are many different variations of multi-rotor drones, but the most popular is the quadcopter. The quadcopter has four propellers in an ‘X’ formation around its body. An example of a fixed-wing drone would be an unmanned plane. It has a single

propeller in the front (nose) portion of the body, with two fixed wings, one on either side of the body, and a tail wing.

Another type of drone would be the flapping-wing drone. The flapping-wing drone operates much like a bird or an insect. It usually has two wings, one on either side of the body, that flap up and down, side to side, or a combination of both in order to be able to fly. The flapping-wing drone isn't as popular as other drones, such as the quadcopter, mostly because the fight system is much more complex. In a quadcopter, the propellers spin in a single direction creating a circular motion. The only adjustment made to each propeller is the speed at which it is turning. In the flapping-wing drone design, the wings must change direction, as they do not spin in a complete circle.

One of the most common solutions to the odd flight pattern of the flapping-wing drone is to use a linkage system to transform circular motion into an arcing motion. One such system is the Crank-Rocker Four-Bar Linkage system. The system would work by having the flight controller command the ESCs to spin the motor. The motor would be connected to the four-bar linkage system, which would spin one of the crank pieces, which is connected to another crank piece that would turn the output crank. The output crank would be moving in an arc. In this system, usually one full rotation of the initial motor results in an entire back and forth motion from the output crank on the arc.

Using this system, given a motor speed, the crank system would flap the output crank at a certain speed on the arc. In flapping-wing drone design, it is necessary to have a different speed when on the forward flap of the wing versus the backward flap of the wing. This is called a split-cycle motor rotation, and is necessary so that the drone can have maneuverability. Having the different speeds for different directions allows the flapping-wing drone to change the direction it is facing, allows it to move forward, backwards, or side to side, and even allows it to adjust its altitude. These same movements are achieved by spinning each motor at different speeds on a quadcopter, allowing it to travel in any direction.

When testing a flight system, there are many things that are taken into account. For instance, one important factor is the ESC being used. There are many different types of ESCs, each with their own communication protocols and speeds. Some are faster than others, and some are more versatile. Another major component is the motor itself. There are many different types of motors, but the majority of the difference in motors lies in the top speed they can spin at. The ESC is mainly used to decide how fast the motor can be updated, and the motor itself is used to decide how fast the top speed would be.

Another part that should be taken into account is the length of the linkage system components themselves. The output speed and arc are dependent on the sizes chosen for the linkage components. These components are also mostly chosen due to the available size of the flight system being designed. For instance, the drone being built might be a micro-UAV, which means that all the components on it must be very small in order to fit inside the body of the drone. Because of this, the design team may have to use a smaller linkage system in order to make the linkage fit in the drone frame.

The goal of this project is to design a test bed for designing, implementing, and testing different split-cycle flight system components to see which ones match well within a given system design. The test bed will allow the users to swap out different ESCs and motors to see which combination gives them the desired output speed necessary for the flight system they are designing.

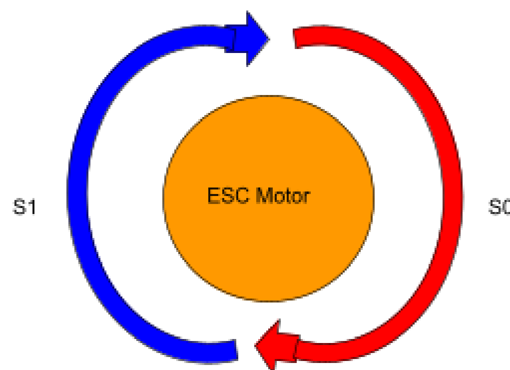
CHAPTER 2

INSPIRATION

The inspiration for this project stems mostly from two different published articles. The first is a paper titled *Wingbeat Shape Modulation for Flapping-Wing Micro-Air-Vehicle Control During Hover* by David B. Doman, Michael W. Oppenheimer, and David O. Sigthorsson. The second paper is titled *Flight Performance of a Swashplateless Micro Air Vehicle* by James Paulos and Mark Yim.

The first paper talks about how using a split-cycle rotation on a motor can help lead to maneuverability of a drone body using a flapping-wing flight system. This is what inspired the idea of creating a testbed that would allow for different motor and ESC combinations to be tested to see how well they work with this idea of a split-cycle rotation. It also mentioned the cyclic-averaged speed of a rotation, which was what I used as my similarity goal (between different split-cycle time ratios).

Although this testbed operates on a much more primitive version of the ideas presented in the paper, it does allow for basic testing of the principles on all different kinds of motor and ESC combinations. This testbed is more made to test whether the chosen components would work well in a given system.



Above is a picture of what the split-cycle is doing. It is taking a full rotation of the motor and splitting it into two cycles, S0 and S1. Each cycle will most likely be spun at different speeds, causing the instantaneous speed to be different throughout the rotation. This is also why we are using the cyclic-averaged speed, as it represents the average speed per rotation of the motor.

In the second paper, there are ideas of being able to control a coaxial copter without a swashplate. This is accomplished by using an oscillating motor speed with a hinge-based flapping wing mechanism. This combination allows for control over roll, pitch and yaw, making it very effective considering it only uses two motors.

In future works, I think it would be interesting to attempt to combine the ideas in these two papers. Using the testbed proposed in this paper, it would be interesting to see how the split-cycle controller would affect the hinge mechanism on an output wing flap. The split-cycle controller doesn’t necessarily oscillate the motor, but if it was updated fast enough, it might be able to resemble an oscillating pattern.

The hope would be that it would allow for the same types of control over roll, pitch, and yaw, while operating on a split-cycle rotation from the motor through the crank-rocker 4-bar linkage system. This would mean that the same types of operations could be performed on a flapping-wing drone as well as a coaxial copter.

CHAPTER 3

BACKGROUND

Microcontrollers:

Microcontrollers are programmable embedded pieces of hardware. They are typically used for things that require a small amount of functionality, as they can only handle doing so much. They are typically small pieces of hardware that have multiple electrical pins sticking off of them. They usually consist of a processor, some internal memory for holding the program, and input/output pins.

Microcontrollers are used mostly in embedded systems. They work very well for contained systems and can handle multiple peripheral “modules” or devices at the same time. They communicate with these external devices usually via a wired connection on one or more of their electrical pins. These pins allow values to be sent to and from the microcontroller to the devices it is connected to.

Since this is very low level hardware, the communication on these pins (usually) only consists of high and low voltage values, either ones or zeros. It is still possible to communicate more complex values, such as numbers, but this requires extra work. There are some hardware options for sending numbers over pins, such as SPI (Serial Peripheral Interface) or I2C (Inter-Integrated Circuit), as well as software based solutions, such as Bit-Banging.

These solutions work by sending packets (groups of ones and zeros to be kept together to form a single value) of data instead of single values. This requires that the device taking in the input knows how to interpret the data, usually by knowing when a packet starts and when it ends. This is usually done by either having a set length of each packet, or by getting a signal stating the packet is about to be transmitted (a start signal), and a signal for when the packet has been sent (an end signal).

These microcontrollers are programmable, but the programs are structured a little differently than normal written computer code. There are usually two main components to the embedded system code, the setup and the loop. The setup is a section of code that runs once before the loop starts. This is where much of the initialization of the system starts. This is where programmers set which pins will be used as input/output, as well as declaring basic variables and potentially setting up data structures to hold the information the device is working on.

The loop is the section of repeated code, which runs after the setup portion is completed. This part contains the set of instructions that are to be repeated endlessly. This is typically where programmers put parts about reading in inputs, processing them, and potentially producing some actionable output. Depending on the system being implemented, this can also be where repeated actions are programmed into, such as constantly spinning a motor at a certain speed.

One of the most famous microcontrollers is the arduino. There are many versions of the arduino, such as the uno, mega, and nano, just to name a few. One microcontroller that has gained much popularity in recent years is the ESP32. Both have very similar functionalities, but as far as speed goes, the ESP32 is faster than the arduino. Most arduino boards have a clock speed of 16MHz, meaning that about 16 million cycles (instructions) can be run per second. The ESP32, on the other hand, has a clock speed of 80MHz, 160MHz, or 240MHz. This value can be changed by the programmer, depending on the project needs. Faster clock speeds give more processing power, meaning more calculations can be done per second, but this also uses up more power.

In this project, I've decided to use the ESP32 over the arduino due to the faster clock speed values. Since some ESCs for a flapping wing drone require a faster update speed, a faster clock speed should hopefully help with being able to update the ESC (and in turn, the motor) at a rate more suitable for a flapping wing drone.

One downside to the ESP32 is that the pins have a voltage value of 3.3 volts for the binary value one (a turned on/high pin outputs 3.3 volts). Arduino has a value of 5 volts for its binary high value. The value for high must be the same for both the microcontroller and the devices that are being communicated with, such as the ESCs.

During experimentation, I found that the ESCs used 5 volts to mean high, instead of 3.3 volts. Because of this, I had to use a 3.3 volts to 5 volts logic level shifter, which just takes in a 3.3 volts signal and boosts it to 5 volts. This allows the ESP32 to output 5 volts to its pins for a high signal, which is what the ESCs are expecting as input.

PWM:

PWM (Pulse-Width Modulation) is a digital signal that is used by microcontrollers to control analog devices. The signal that the PWM sends out consists of only high or low voltage values, but these values can be interpreted as analog values by measuring how long the signal is high (or on). The analog devices measure how long the signal is on for, and use that as an analog input value for controlling whatever device they are designed to control by viewing the time on as a percent of the entire frame.

The way the values are changed in PWM is by changing the amount of time the high or on value is set (also known as the pulse). In this experiment, the possible values for a PWM signal to the ESC were between 1000 microseconds and 2000 microseconds. A 1000 microseconds signal would indicate an off or low value to the ESC, meaning the motor would not spin. A 2000 microseconds signal would indicate an on or high value to the ESC, meaning the motor would spin as fast as it could. All values in between 1000 and 2000 microseconds would correspond to a percentage of the speed. For example, 1500 microseconds would spin the motor at half the fastest speed possible for the motor (50%).

The duty cycle is the amount of percentage the “on time” of the signal is in a given time interval. For example, 1000 microseconds would correlate to a 0% duty cycle, since the signal is never on. 2000 microseconds would correlate to a 100% duty cycle, since the signal is never

turned off. 1500 microseconds would correlate to a 50% duty cycle, since the signal spends half of the data frame time turned on and the other half turned off.

The PWM protocol is used to control many devices with microcontrollers. It is a very useful and versatile protocol that almost all devices can interface with. In most instances and projects, it is plenty fast to handle normal operations, but with projects that require an extremely fast update speed, it becomes less and less useful.

PWM is plenty fine when flying a basic drone, such as a quadcopter or hexacopter, but flapping wing drones usually require many updates per revolution to be able to perform the complex back and forth movement required to keep the drone airborne. Because of this, PWM doesn't usually offer a fast enough update rate to control a flapping wing drone well enough for flight. Even though PWM is the ESC protocol being used in this testbed, that doesn't mean that other ESCs won't work with it. The idea of this testbed is to be able to experiment with different types of ESC protocols to see which one might work best for a given flapping wing drone.

ESCs and Motors:

An ESC (Electronic Speed Controller) is used to control an electric motor with a given input speed. It does this by receiving the signal of a throttle speed, processing this input speed, and then outputting the correct commands for the motor to follow to achieve the input speed command.

ESC current ratings are calculated by figuring out the maximum current an ESC can output within ideal operating conditions. This rating is usually measured in Amperes. For example, the one used for the PWM protocol in this project is rated to be able to support up to 30 Amperes of current.

There are four main components to an ESC system: the power supply, the microcontroller, the data receiver, and the motor controls. The ESC itself does not have enough power to control the motor, so it has two wires (a positive and a negative) that are used to receive power from an external source, usually either a battery or a power supply. For this

project, I have the ESCs attached to a PDB (power distribution board), which receives power from a battery and distributes it to the attached ESCs.

The microcontroller is the piece of hardware that all the external wires are attached to. It is in charge of receiving and distributing the power properly, receiving and interpreting the data given to it in the form of throttle commands, and it is in charge of outputting the proper commands to control the motor. The microcontroller is programmed with firmware that is designed to allow it interpret the incoming throttle commands properly and to match them to the correct motor output commands.

The input throttle commands are also referred to as ESC protocols. There are many different forms of ESC protocols, the oldest being PWM. The way that each ESC protocol is different is in the way it formats the throttle data it is trying to transmit to the ESC. PWM uses a pulse (measured usually between 1000 microseconds and 2000 microseconds) to indicate a percentage of total power the motor should spin at. One of the newer and more popular ESC protocols is DShot. It sends the data in a 16-bit binary packet with a throttle value, a telemetry value, and a checksum. Whichever ESC protocol is being used is sent into the ESC via a data wire (receiver), which is connected directly to the ESCs internal microcontroller.

There are many different types of ESC protocols. PWM, DShot, OneShot, and Multishot are just to name a few. Each one has its own set of benefits and drawbacks, but usually the most important thing people look for when choosing an ESC is the update speed it can support. Each one has its own speed, and some are more popular than others just because they can update faster.

The last component of the ESC is the motor control wires. In a typical brushless motor design (like the one used in this experiment), there are three wires that connect the motor to the ESC. The motor itself has a magnet inside of it, surrounded by a circle wrapped in coils of wire. When power is supplied to the motor, electricity travels through these wires and induces a

magnetic field, which in turn spins the magnet, which turns the motor. The faster the induced magnetic field changes, the faster the motor spins.

The magnetic field is created from the three wires that are attached to the ESC. Each wire creates a phase current, combining together to make an alternate three-phase current. The magnetic field shifts locations in order to keep the motor moving. The ESC controls the wires with a switching frequency, which changes how fast the magnetic field changes locations. The faster the switching frequency, the faster the magnetic field changes, which in turn moves the motor faster.

PID Controllers:

PID is a control algorithm used to match an input value to an expected value by changing a certain output variable. This variable is usually used as input to the sensor that is being measured for control. In this project, for example, the input value would be the measured speed of the arcing output speed arm, the expected value would be the desired arcing output speed arm, and the adjusted output value would be the ESC command speed.

PID controllers are implemented in closed loop systems. This means that the system requires feedback from sensors in order to regulate the system to get a desired output. This typically comes in the form of a feedback loop system, meaning that the sensor is used as input, then calculations are done on the input, which produces an output to be sent back into the system as input.

In the PID algorithm, there are three main components. The first component is the proportional (P) component, the second is the integral (I) component, and the third is the derivative (D) component. They all have their own calculations, but are combined back together to form the output of the PID control algorithm. Each component calculates a new output value based on the error between the input value and the expected value. These calculations combine to form an adjustment to the output value that should move the input value closer to the expected value, by decreasing the error.

The proportional component depends purely on the error between the input value and the expected value. When creating a PID controller, there is a proportional gain value that is used to calculate an error response ratio. Usually, increasing the proportional gain value increases the speed at which the output value changes. If the proportional gain value is set too high, this will cause the system to oscillate around the expected value, due to the output value overshooting and undershooting the expected value due to too large of a proportional gain value.

The integral component works by summing the error values over time. This component constantly gets larger, unless the error value is zero. The smaller the error, the smaller the integral component increases. This component receives an integral gain value, just like the proportional component. The integral components goal is to decrease the steady state error, which is the final error value between the input value and the expected value.

The derivative component focuses on slowing down the increase of the output variable when it is increasing too rapidly. It receives a derivative gain value, which determines how strongly the derivative component will react to a faster rate of change in the control system output response. In most systems, this value is either left at zero, or kept to be a very small value, as it is very sensitive to noise in the output value.

When initializing a PID controller, there are three gain values that are adjusted to fit the system appropriately. These gain values are the proportional gain, integral gain, and the derivative gain. Each one corresponds to the error calculation portion of the PID controller with the same name. These values are set by the programmer upon setting up the PID loop, and are usually adjusted to work well with the system they are being implemented in. The process of changing the gain values to work with the system is called parameter tuning.

In order to tune the parameters properly, the operator would most likely use the guess and check method. They would set the proportional gain to one, integral gain to zero, and derivative gain to zero. They would increase the proportional gain until it started to overshoot

the expected value. Once that happened, they would start increasing the derivative gain to smooth out the output slope, until they got a desirable response time and smooth speed change curve. After the proportional and derivative values start giving reasonable PID outputs, they would then adjust the integral component to decrease the steady state error. This process of tuning the parameters would need to be done for any ESC and motor combination being used.

Although there is no current setup for a PID loop in the testbed, there could easily be one added once the crank-rocker 4-bar linkage was set up. If another encoder was set up to measure the output crank movements, then these could be used to adjust the initial input motors speed to reach a desired output speed. This could also be used to auto-adjust the ESC commanded speed values in order to try and get the desired average wingbeat speed, while attempting to reach the highest stable split-cycle time ratio.

Quadcopter Flight Design/Requirements:

Quadcopters have exploded in popularity in the hobby industry. They have even become full time jobs for some people, as they offer unmanned ways to observe, analyze, and even deliver things that normally require a person to do. Due to their rise in popularity, there have been many new products/applications that they have been used for, with many more ideas still to come.

The quadcopter is one of the more basic designs for a drone. It has a basic set of components that allow it to be able to fly. Due to its rotary wing nature, the quadcopter is able to vertically take off and land, much like a helicopter. The main components of a quadcopter drone are the propellers, the ESCs and motors, the flight controller, the battery, and most of the time they will also have a radio controller.

The propellers are what are used by the drone to push air beneath it to give it lift. They work much like helicopter blades do, by spinning either clockwise or counter-clockwise (depending on which arm of the quadcopter they are attached to) to generate the required lift for the drone. The faster the propellers spin, the more lift force is created. These propellers are

usually designed to have two wing flaps, one on either side of the motor, to help produce extra lift with a single rotation of the motor.

The propellers are attached to the motor, which is controlled by the ESC. The ESC receives a commanded speed input from the microcontroller/flight controller and turns that commanded speed into values that the motor can understand. The ESC needs an external power source, which usually comes in the form of a battery attached to a PDB (power distribution board) that powers all the ESCs on the drone. There are usually four ESCs and motors attached to a quadcopter, but there is also the possibility of a 4-in-1 ESC that all the motors are attached to.

The flight controller is the brains of the drone. Since controlling four separate motors simultaneously is extremely difficult for humans, we have offloaded the need to control each motor independently onto the flight controller. All the ESCs are attached to the flight controller in order to receive their input speed values from it. The flight controller is either programmed with a flight path, or given real time instructions, usually via a radio controller for directions. The flight controller then takes these directions, and outputs the proper corresponding ESC speed values to each of the ESCs. This allows for the human pilots to enter a command such as “fly up” or “fly to the right”, and the flight controller will command the ESCs to spin at certain speeds to create the desired direction/path.

Most hobby drones receive directional commands from a drone pilot given via a radio frequency. The pilot has a remote controller, which encodes directional commands (such as up, down, left, right, forward, backward, etc.) into radio frequencies that are then transmitted to the drone. The drone has a receiver for these radio frequencies, and decodes them into ESC speed instructions to produce the desired result. These radio frequencies can also be passed to the drone via a telemetry module, which could be plugged into a computer with a usb port in order to send commands not from a handheld controller.

There are different commands that the drone can receive for directions, and the ESCs will be spun accordingly. For example, in order to fly up, the flight controller will spin all the motors faster to generate more lift. To fly down, it will spin all the motors slower to generate less lift. To fly to the left, it will spin the left two motors slower and the right two motors faster. The opposite is true for flying to the right. There are many more complicated flight commands, but that is how the quadcopter can perform the basic maneuvers.

Even with the basic capabilities of moving up, down, left, right, forward, and backward, this makes the quadcopter practically omnidirectional (it can travel in any direction). This is due to its control over four distinct motors that it has control over. Each of the ESCs and motors only have to spin in a single direction, making adjustments to speed the only thing they need to do. This allows the motors to preserve momentum, which also helps them to not need such fast update rates. The PWM ESC protocol works very well for quadcopter drones because of this. A faster update rate is helpful in that it makes the drone more responsive to flight control inputs from the pilot, but they aren't as necessary to maintain flyability.

Flapping Wing Design/Requirements:

Flapping wing drones are not as popular as quadcopter drones due to their more complicated design requirements. The complexity comes mostly in the form of the way the wings move. The wings on a flapping wing drone don't spin like they do on a quadcopter, but rather are moved back and forth to generate lift and thrust.

There are different kinds of flight patterns for flapping wing drones. There is the traditional bird flapping wing design, which involves moving the wings up and down (sometimes folding them) to generate a forward movement, much like most basic birds such as crows and pigeons use. Another type of flight pattern is more similar to that of the hummingbird. The wings flap back and forth, as well as up and down. This allows them to hover in place and move omnidirectionally, much like the movements of a helicopter.

The hummingbird moves its wings in an infinity figure, or a sideways eight figure movement. This allows it to move back and forth, while also moving up and down, all in one fluid movement of the wing. The wing itself uses both sides of the wing (the bottom and top panels of the wing) when flapping. This means that the top part of the wing on the forward movement becomes the bottom part of the wing on the backward movement. This requires that both the top and the bottom part of the wing are similar so that they can produce the same kind of lift in either direction.

The wings themselves are much different from traditional drone propellers. Drone propellers typically have two fixed wings for every one propeller, whereas the flapping wing drones usually only have a single wing on either side. Since the flapping wing drone only has the two wings, this puts a lot more exertion on the wings, requiring them to be able to make more complex movements than traditional drone propellers. There is also the tail wing, but it doesn't flap rapidly, it is more used like a rudder on a plane or a boat. It is used to control directionality, and it could easily be controlled by a servo type of device. Because of this extra complexity in movement, the flapping wing drone wings usually need to be able to update their speeds at a faster rate than traditional drone propellers.

Other than the wing path/flapping pattern, wing design, and necessary update speed, flapping wing drones and quadcopter drones operate on a similar level. They both have a flight controller that is controlling the ESCs which control the motors to move the wings, though the instructions for the ESCs are different, since the way the wings move is different on each vehicle.

The tip of the wing is typically used as the indicator for the drone on when to move the wing back and forth. Once the wing tip has reached what is essentially its peak or the apex of its path, it then starts to flap the wing in the other direction. This is why an encoder is necessary. The encoder would be used to measure how far the wing has traveled, which will tell the flight controller/microcontroller when to switch the direction/speed the wing is moving.

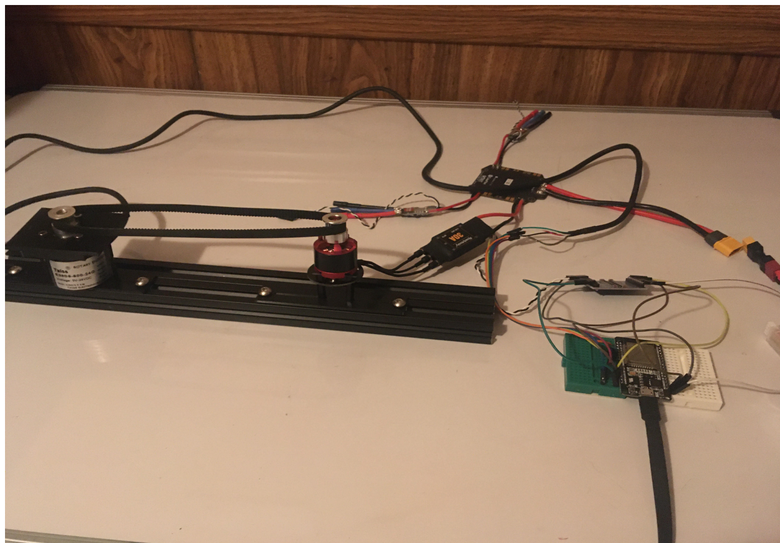
There are a couple different ways to change the wing direction. One of them is to have a linkage system that can transform the circular motion of the motor into an arcing figure on the output wing path. This would allow the motor to spin in a single direction, while also getting the wing to move back and forth. Another option would be to have a bidirectional motor and ESC. This would allow the ESC to move the motor either forwards or backwards, meaning the entire control of the wings movement can be controlled from the ESC.

In order for a flapping wing drone to have maneuverability, it needs to be able to change the speed the wings move on the forward and backward strokes, all while maintaining a cyclic-averaged speed. The goal of this testbed is to develop a system that can be used to evaluate different split-cycle time ratios (different speeds for forward and backward motion) while still maintaining a desired cyclic-averaged speed.

CHAPTER 4

SYSTEM OVERVIEW

This testbed is meant to be used to test out different ESCs and motors to see if they can produce a desired cyclic-averaged speed, and to test how large of a split-cycle time ratio they can achieve with said ESC and motor combo while maintaining the desired cyclic-averaged speed. The testbed does this by spinning a motor at one speed, then switching to the other speed when half a rotation has been completed (or half of the simulated gear ratio). Once another half rotation has been completed, it displays the average speed over the entire rotation (cyclic-averaged speed), as well as the amount of time spent on the first speed and the second speed. It then switches the speed back to the first speed and repeats the process.



When looking at this system, there are three main subsystems to consider. The first is the microcontroller, which is essentially the brains of the entire testbed. The next subsystem would be the ESC and motor combination. These are what are being commanded to spin by the microcontroller. The last main subsystem would be the optical encoder. The optical encoder is being used to determine how far the motor has spun so far. This is done by keeping track of a

counter, which is incremented every time the motor moves the optical encoder shaft a “step” or “click”.

The microcontroller is the commander of the entire testbed. It is in charge of running the split-cycle controller code. It collects all the information from the other subsystems and uses it to determine what actions need to be taken next. The microcontroller is in charge of running the firmware (code) that has been uploaded to it to run continuously. It starts by running the setup portion of the code, which is used to initialize many of the modules connected to the microcontroller. It then runs the loop portion of the code on repeat, which is in charge of continuously updating the values the modules are depending on in order to operate correctly.

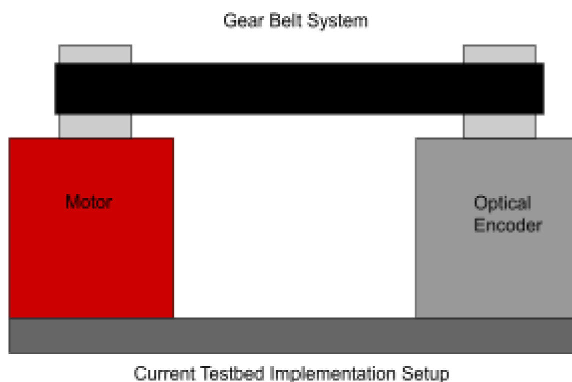
In the case of this testbed, the microcontroller being used is an ESP32. It is operating at a frequency of 240 MHz. The other subsystems are connected to the ESP32 in order to provide it with the necessary information and control it needs in order to perform the testbed experiment. The optical encoder is set up to inform the ESP32 when its shaft has been rotated a step. This is done by setting up an interrupt on the ESP32 that triggers when it receives a signal from the optical encoder telling it to update the counter value. The ESP32 is connected to the PWM ESC in order to tell it what command speed to run the motor at. The ESP32 determines which commanded speed to send to the PWM ESC by checking which half rotation it is currently on. Once that half rotation is complete, the ESP32 then informs the PWM ESC to spin at the alternate speed.

The ESP32 is also in charge of displaying the desired information to a computer for the human operator to be able to interpret. It is in charge of keeping track of the speed and time variables that the operator is wishing to see. It then outputs this information in a format the operator can understand by sending the information through a serial connected (USB) on the computer the ESP32 is plugged into, and the computer then displays this information for the operator to see. The output information is the cyclic-averaged speed, the time spent on S0 rotation, and the time spent on S1 rotation (the split-cycle times).

The next subsystem is the ESC and motor. The ESC receives two inputs. One is a power source, and the other is an input speed control value. The ESC requires more power than the ESP32 is capable of outputting, so the power supply wires for the ESC are connected to a PDB (Power Distribution Board), which is hooked up to a battery. The input speed values are provided by the ESP32, which are sent through a logic level shifter, and then sent over to the ESC. The ESC that is currently being used in this testbed uses a PWM ESC protocol, which is one of the oldest ESC protocols.

The ESC is in charge of taking the commanded input speed, and making the motor spin at that speed. Since the current ESC uses the PWM protocol, the input speed comes in the form of a percentage. The input PWM values are in the range of 1000 to 2000 microseconds. If the input value is 1500 microseconds, then the ESC will spin the motor at 50% of its max speed. The ESC is connected to the motor with three wires/bullet connectors. It uses a 3 phase current in order to spin the motor with an induced magnetic field.

The motor is topped with a small gear belt system which connects the motor to the optical encoder subsystem. This is set up so that every rotation of the motor corresponds to a rotation on the optical encoder. Currently, the gears on the top of the encoder and the motor are the same size so that there is a 1:1 ratio of movements, meaning every 1 rotation of the motor results in 1 rotation of the optical encoder shaft.



Current Testbed Implementation Setup

The optical encoder is in charge of keeping track of how far the motor has spun. It does this by issuing interrupts to the ESP32 when it notices a change in its shaft position. The ESP32

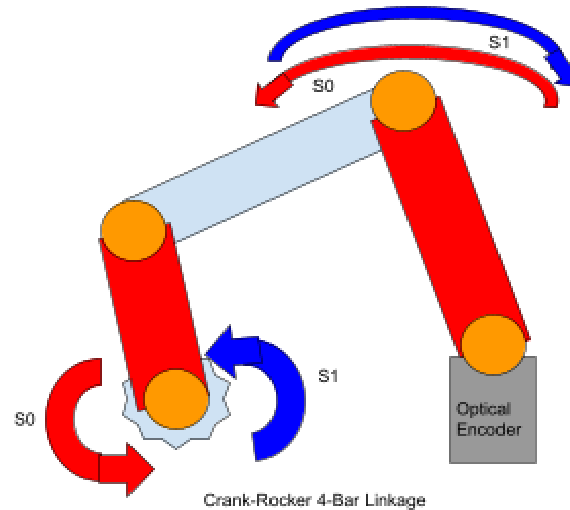
is keeping track of the current position using a counter variable. When an interrupt is issued by the optical encoder, the ESP32 either increments or decrements the counter variable, depending on which direction the optical encoder shaft was moved.

The optical encoder shaft is connected to the motor via a gear belt system. Every rotation of the motor correlates to the same rotation on the optical encoder shaft. The ESP32 uses this counter variable to tell where the optical encoder is in its current rotation. It checks to see if the counter variable has reached the halfway point, and if so, it updates the speed value with the alternate speed value for the motor to spin at.

Currently, the gear ratio system is simulated. This is done by changing the values for the full and half (change) rotation variables in the code to represent larger or smaller gears. In the experiment run in this paper, a 10:1 gear ratio was used. This means that instead of using 1200 as the full rotation count, I used 12000. This meant that the code counted it as a single rotation when the encoder counter got to the value 12000 instead of 1200. This meant that the encoder had to complete 10 rotations in order for the code to count it as a single rotation. This was done to make sure that each commanded motor speed could be seen by the human eye.

In the future, it would be nice to have a physical gear system, where different gear ratios could be swapped out to either speed up or slow down the output speed. The current simulated version works for just using the testbed, but a physical gear system would be required when testing the actual output speed on the 4-bar linkage system.

Though the gear system is simulated, the 4-bar linkage would hopefully be another physical future implementation. This crank-rocker 4-bar linkage system takes in the speed from the motor (potentially altered by a gear ratio) and transforms it from a circular motion into an arcing motion. This arcing motion would represent the flap of a flapping wing drone. In a physical implementation of this system, the operator would be able to swap out different bar lengths to see which ones give a desired output speed/arc.

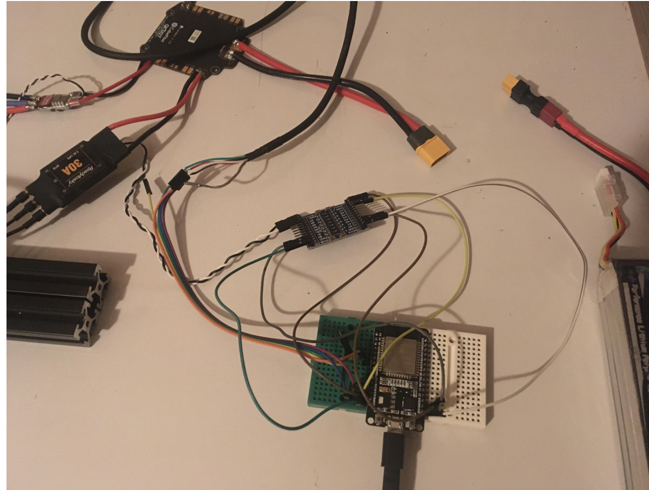


Crank-Rocker 4-Bar Linkage

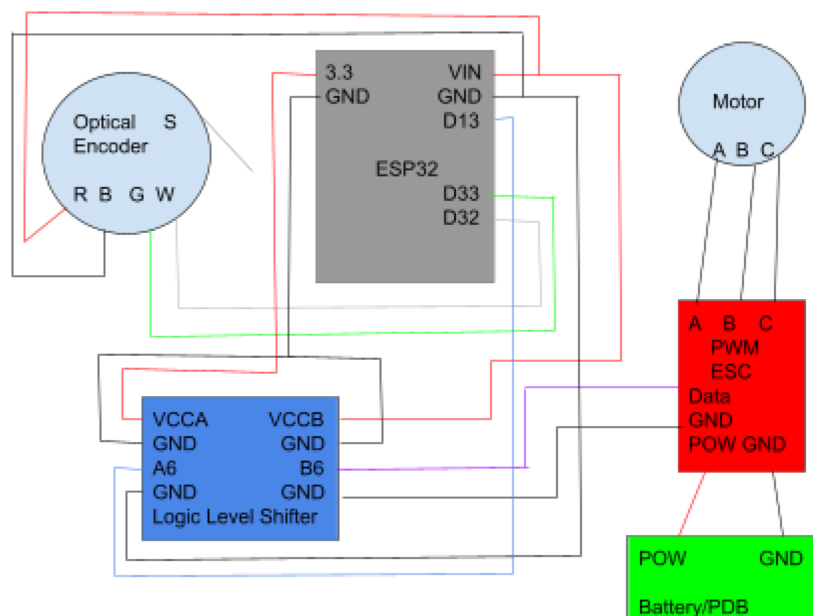
Due to the design of the crank-rocker 4-bar linkage, every full rotation of the motor (or gear ratio output) results in a full back-and-forth motion on the arc. This means that half of the motors rotation correlates to the speed of the forward arc, and the other half of the motors rotation correlates to the speed of the backward arc. This is why the testbed is set up to spin the motor at two different speeds, running a split-cycle rotation. It allows the operator to try flapping the forward and backward arcs at different speeds to gain maneuverability in the flapping wing drone system.

CHAPTER 5

WIRING LAYOUT



The image above shows the wiring of the physical testbed system. The wires are somewhat hard to trace and appear cluttered. Because of this, I have created a diagram that is hopefully more helpful in understanding the actual wiring layout. This diagram is presented below.



To start out with the wiring, it is important to note that the ESP32 that was used in this testbed did not have enough pins for things like power and ground. Because of this, breadboards were used to expand the potential wire spots for these pins so they could be used for multiple modules. Since the ESP32 is the main “brains” of the system, most of the modules are connected via wires to it.

To start out with the wiring layout, set up the ESP32 so that it has the ability to support multiple power and ground wires (usually using a breadboard). The next module to hook up would be the optical encoder. There are five wires that come out of the optical encoder. There is one red, one black, one white, one green, and one silver wire. The black wire is connected to a ground (GND) pin on the ESP32. The red wire is connected to the VIN (5V) pin on the ESP32. The white wire is connected to the D32 pin, and the green wire is connected to the D33 pin. The silver wire is another ground wire, but it doesn’t need to be connected to anything, so it is left alone.

The next part that needs to be wired up is the logic level shifter. This module works by taking in a 3.3 volt signal and boosting it up to a 5 volt signal. This is done because the ESP32 can only set its output pins to 3.3 volts, but the PWM ESC operates on 5 volts. When it is wired up properly, the input signal in 3.3 volts should then be output on the other side as a 5 volt signal. There are two sides on the logic level shifter, an A side (VCCA) and a B side (VCCB). On each side, there is a power (VCCA/VCCB) and a ground (GND) pin. The VCCA pin should be connected to the 3v3 (3 volts) pin on the ESP32 board. The VCCB pin should be connected to the VIN (5 volts) pin on the ESP32 board. Both of the grounds on either side of the logic level shifter should be connected to a ground pin on the ESP32 board.

Now the logic level shifter is set up to be used, so the next step would be to set up the connection between the ESP32 and the ESC data communication wires. The ESC data wires (consisting of a white and black wire) are connected to the VCCB side of the logic level shifter, where the black wire lines up with the ground and the white wire lines up with one of the B pin

values (with the board being used in this experiment, it ends up being B6). Then, on the ESP32 side, connect a ground wire to the ground wire on the A side of the logic level shifter, and connect a data wire from the D13 pin to the corresponding A side of the wire that matches up with the data wire that the ESC is connected to on the B side (which would be A6, using the board used in this setup).

The only other wires in this setup would be the ESC power wires, the motor wires, and the ESP32 serial wire. The ESC needs power from an external source, as the ESP32 doesn't provide enough power for the ESC and the motor. In the current system, this is accomplished by soldering the ESC to a PDB (Power Distribution Board) that is connected to a battery. The motor wires (3 of them) are connected to the ESC using bullet connectors. The final wire is the USB data wire that is connected from a computer to the ESP32. This data wire writes the firmware onto the ESP32, powers the board, and should be capable of serial communication with the board.

CHAPTER 6

CODE

The code for this testbed is currently set up to support the PWM ESC protocol with an encoder that has 1200 steps every revolution, all running on an ESP32 microcontroller. This is important to note as different hardware may require some changes to the code in order for it to still be usable and accurate. For example, the change value is half the full value, which is currently set to 12000 because 1200 is how many steps are in a single rotation of the encoder being used, and the gear ratio being simulated is 10:1, which means every 10 rotations on the motor is equal to 1 rotation on the output system.

At the top of the code, there are some global variables. Many of these variables are used to keep track of the position, time, and velocity of the motor movements. There are position and time values for a single rotation of the encoder and motor, stored in variables named newpos, oldpos, newtime, oldtime, and velocity. The counter variable is the variable keeping track of how many steps (encoder movements) have been taken at any point in time. There are also two variables, called full and change, which are equal to the amount of encoder steps in a single rotation and the halfway point on that encoder count, respectively. Currently, the full value is set to 12000 instead of 1200, in order to simulate a 10:1 gear ratio. This means that the motor will have to spin the encoder 10 full rotations in order to be counted as a single rotation.

There are also the global variables for the s0 and s1 speed portions. Each full rotation on the motor consists of half of the rotation at s0 speed and the other half at s1 speed, making it a split-cycle rotation. These two speeds will most likely be different. They will both travel the same distance (half a rotation), but they will each take a different amount of time to finish their half of the rotation. Because of this, I have added variables to keep track of the start and end

time for both s0 and s1 (s0start, s0end, s1start, and s1end). There are also the variables of s0 and s1. Currently, the test bed is set up to run a PWM ESC, so these values are storing the PWM (microseconds) speed values that the motor will spin at for each split (half) of the cycle.

There is a function called encoder_isr(). This function is set up to trigger on interrupts, which occur when the optical encoder senses a movement. This function either increments or decrements the counter value, depending on which direction the encoder moved. For this experiment, the encoder should only be moving in the positive direction, so the counter value should only be able to increase when the interrupt is triggered.

In the setup portion of the code, a serial communication is set up on 115200, which is the default serial communication rate for the ESP32. The pin modes and interrupt are set up for the optical encoder, and the initial speed is set to the s0 speed. Since this test bed is running a PWM ESC, the ESC must be calibrated first. This is done by writing 1000 microseconds to the ESC (equivalent to a speed of zero) for three seconds. After the ESC is calibrated, the s0 start time is started, and the motor will start to spin at s0 speed.

The loop is the repeated part of the code and where most of the speed calculations take place. There are two main spots in the main loop. The first is when the counter reaches the halfway point of the cycle (rotation). When this happens, the s0 end time is recorded and the speed value is switched from s0 to s1 and the s1 timer starts. The other main spot in the main loop is when a full rotation has been completed.

When a full rotation has been completed, results of the full rotation are shown. The full rotation time and the s1 time are stopped, and the new position value is found. Using the oldtime, newtime, oldpos, and newpos, the cyclic-averaged speed for that cycle is calculated and displayed. The cyclic-averaged speed is multiplied by 10,000, due to the calculated number being too small to display. The s0 total time and s1 total time are calculated and displayed as well. Once that is done, the old position and old time values are updated. The speed value is set back to s0, and the s0 timer starts again.

For this instance of the experiment, using the PWM ESC, 1200 step optical encoder, and ESP32 microcontroller, the only values that are changed for the experiment are the s0 and s1 values, as well as the full value to simulate a 10:1 gear ratio. In order to check the maximum supported split-cycle time ratio, one of the two speed values is decreased, while the other is increased. Once the new s0 and s1 values are found that generate the same average cyclic speed, the time for the s0 and s1 speeds are added together to get the total time in the rotation. Once the total time for the full rotation is found, then it is just a matter of dividing s0 by the total and s1 by the total to get the split cycle time ratio.

For example, say the s0 time is 700 microseconds and the s1 time is 300 microseconds. When they are added together, there is a total time of 1000 microseconds. The 700 microseconds for s0 is divided by 1000 microseconds, giving a value of 0.7. The same is done with 300 microseconds for s1, resulting in a value of 0.3. This means that the ratio that can be supported by this grouping of PWM ESC and motor can be 70:30. This means that the first split of the cycle can spin at s0 for 70% of the time and s1 at 30% of the time and still result in the same cyclic-averaged wingbeat speed.

The code used in this testbed is shown below.

```
#include <ESP32Servo.h>

Servo ESC;

// Encoder values
volatile int counter = 0; //This variable will increase or decrease
depending on the rotation of encoder
long newpos, oldpos = 0; // Position values
unsigned long newtime, oldtime = 0; // Time values
double velocity;

// s0 and s1 time values
unsigned long s0start = 0, s0end = 0;
unsigned long s1start = 0, s1end = 0;
// Time flag
```

```
int tf = 0;

// Speed values
int s0 = 1145;
int s1 = 1285;
int speed = 0;

// Step count change, should be half of the step count for a single
rotation.
// This is a gear ratio of 10:1, so every 10 rotations counts as a single
rotation output.
// 1200 steps per revolution
int full = 12000;
int change = full * 0.5;

#define ENCODER_A 33 // Pin for Encoder A
#define ENCODER_B 32 // Pin for Encoder B

// Instructions for wiring
// Plug Power (Red) into VIN pin
// Plug Ground (Black) into GND
// Plug Green into D33
// Plug White into D32
// ESC data pin: D13

// Findings
// PWM  : Speed (steps/microsecond) * 10,000
// 1000 : 0
// 1050 : 0
// 1100 : 185
// 1150 : 285
// 1200 : 419
// 1250 : 544.6
// 1300 : 631

void encoder_isr() {
  // Reading the current state of encoder A and B
  int A = digitalRead(ENCODER_A);
  int B = digitalRead(ENCODER_B);
  // If the state of A changed, it means the encoder has been rotated
```

```
  if ((A == HIGH) != (B == LOW)) {
    counter--;
  } else {
    counter++;
  }
}

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200); // Initialize serial communication 115200
  // Set up encoder
  pinMode(ENCODER_A, INPUT_PULLUP);
  pinMode(ENCODER_B, INPUT_PULLUP);
  // Attaching the ISR to encoder A
  attachInterrupt(digitalPinToInterrupt(ENCODER_A), encoder_isr, CHANGE);
  // Set up ESC and calibrate
  speed = s0;
  ESC.attach(13);
  ESC.writeMicroseconds(1000);
  delay(3000);
  s0start = micros();
  oldtime = s0start;
  ESC.writeMicroseconds(speed);
}

void loop() {
  // put your main code here, to run repeatedly:
  // Check if step count is ready to switch speeds
  // Halfway through the rotation.
  if (counter >= oldpos + change){
    // Only update values once
    if(tf == 0){
      s0end = micros();
      // Change the speed value to s1
      speed = s1;
      s1start = micros();
      ESC.writeMicroseconds(speed);
      tf = 1;
    }
  }
```

```
  // Check if a full rotation has been completed
  if (counter >= oldpos + full)
  {
    // Update new time and new position values
    newtime = micros();
    s1end = newtime;
    newpos = oldpos + full;
    // Calculate value of velocity
    velocity = (newpos - oldpos) * 10000 / (newtime - oldtime);
    Serial.print(velocity); // Prints velocity value
    Serial.println(" steps/microsecond (avg)");
    Serial.print(s0end-s0start);
    Serial.print(" s0 time, ");
    Serial.print(s1end-s1start);
    Serial.println(" s1 time");
    // Update old time and old position values
    oldpos = newpos;
    oldtime = newtime;
    // Reset speed value back to s0
    speed = s0;
    s0start = micros();
    tf = 0;
    ESC.writeMicroseconds(speed);
  }


}
```

CHAPTER 7

SUBSYSTEMS/COMPONENTS

ESP32:

The ESP32 that was used in this testbed is an ESP-WROOM-32. This ESP32 was chosen over an arduino due to the frequency at which it operated. The arduino operates at 16 MHz, whereas the ESP32 can operate at 80 MHz, 160 MHz, or 240 MHz. These faster frequencies can allow the ESP32 to handle much faster ESC protocols, such as the DShot ESC protocol.

This microcontroller runs the code, which means it is in charge of producing the split-cycle controller actions necessary for the testbed. Since it operates at such a high frequency, the time stamps used in this testbed are set to microseconds. There is the option to measure the time in milliseconds, but the ESP32 is very fast, meaning that it usually completes multiple iterations of the loop portion of the code per millisecond.

PWM ESC and Motor:

The PWM ESC used in this setup is a Readytosky 30A ESC and the motor is a Readytosky drone motor. They are standard off-the-shelf pieces of hardware that are used most commonly in hobby drones. They are relatively cheap components, making them easy to acquire, but not of the highest quality. They work well for standard drone operating procedures and tasks.

Battery and PDB:

The battery used in this setup is a 5200 mAh 11.1V High Performance Lithium Polymer Battery. The PDB was a part from a drone frame, where the power distribution board was built

into the bottom part of the frame. The PWM ESC was then soldered onto this power distribution board, along with a power and ground cable that the battery could plug into.

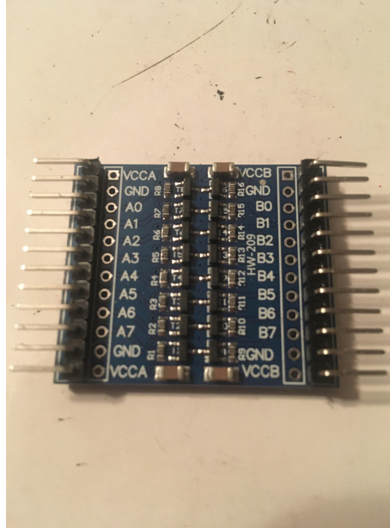
Optical Encoder:

The optical encoder used in this setup is a Taiss Rotary Encoder. It has 1200 steps per rotation. It operates by sending an interrupt signal to the ESP32 if it detects a movement on the optical encoder shaft. The ESP32 responds by either incrementing or decrementing a counter variable, depending on the direction the optical encoder shaft turns.

Gear Belt System:

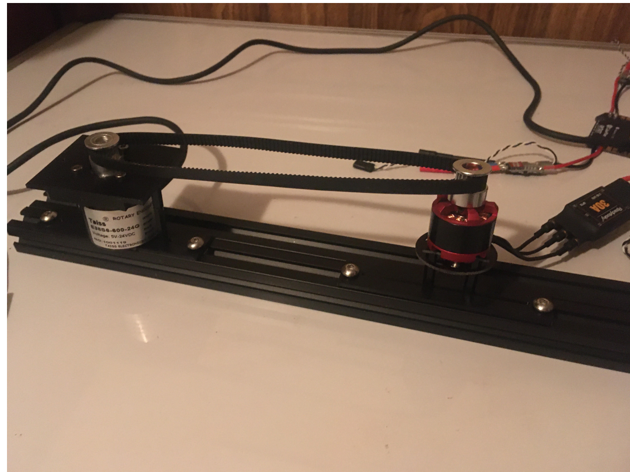
The gear belt system consists of two gear belt holders and a gear drive belt. The gear belt holders used in this setup were the same size. This made it to where every rotation on the motor resulted in the same rotation on the optical encoder side. The motor and optical encoder were placed far enough apart to make the belt taut so that there wouldn't be any slipping of the belt on the gear belt holders.

Logic Level Shifter:



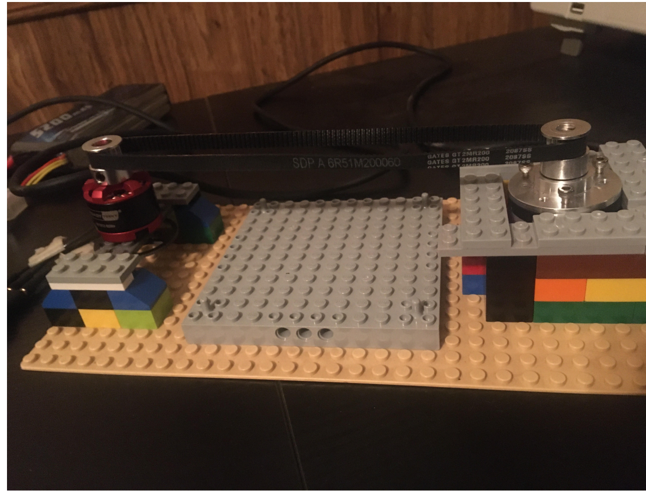
The logic level shifter used is a HiLetGo 5V/3.3V 8 Channel Converter. It was necessary because the ESP32 can only set the output on high at 3.3 volts for its pins. The PWM ESC that is being used operates on a 5 volt level, so the 3.3 volts needed to be shifted up to an output of 5 volts.

Work Bench:

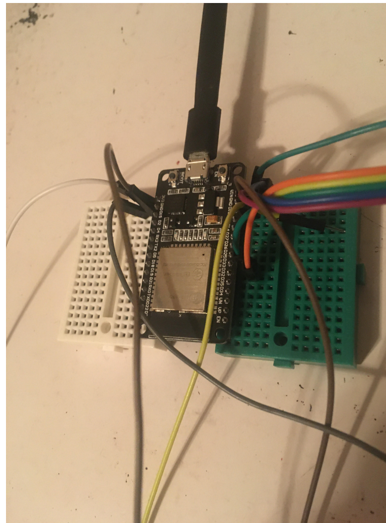


The workbench being used in the experiment was originally made out of legos. This was done to allow for rapid prototyping to make sure that the placements of the optical encoder and the motor mount count be changed if needed. It is currently made of 3D printed parts mounted

on an aluminum rail to allow for sliding back and forth of the components to make sure that the gear belt system can be taut enough to get good readings. Here is a picture of the prototype lego version of the workbench.



Breadboards:



There were two small breadboards used in this testbed. One was placed on either side of the ESP32. This is due to the fact that the ESP32 doesn't line up well on a regular breadboard. These were necessary in order to expand the number of wires that could be connected to both the power and the ground pins on the ESP32 board, since there wasn't enough for the entire project using the single pins on their own.

CHAPTER 8

EXPERIMENT PLAN

The goal of this testbed is to allow users to be able to try out different ESC protocols and motors to figure out a combination that gets them the resulting cyclic-averaged wingbeat speed they need. Along with the cyclic-averaged speed, this testbed should also allow users to be able to figure out the maximum split-cycle time ratio that can be achieved with their chosen ESC and motor combination. The components I am using in testing the testbed are an off the shelf PWM ESC and motor combination that is typically used in basic quadcopter design.

The first step of the experiment is to get the total number of optical encoder steps that are gone through in a single rotation. This value is important because the code relies heavily on the encoder counter to keep track of when to switch from s0 to s1 and s1 to s0. The code has a full variable that must be set to the total number of encoder steps in a single rotation (multiplied by the gear ratio, which I used 10:1 in this experiment). The change variable is set to half of the full value. This is because that change variable is used to signal when the ESC should start spinning at the other speed value.

The next step in the experiment would be to find some reference ESC speed values and figure out their corresponding actual speed outputs from the motor. This step isn’t necessary, but it is helpful in giving the user a good idea of what possible speed values they should be expecting as output when they change the s0 and s1 values later. These values will be different depending on which ESC protocol and motor is being used. The ESC protocol will decide how fast the motor speed is updated, and the motor will decide how fast the motor will spin given the ESC commanded speed.

Once that has been done, the next thing to do is to know the desired cyclic-averaged wingbeat speed we want to achieve. This is usually found from the flight requirements of the

entire system being developed. In the case of this experiment, I decided to use the wingbeat speed of the motor at a PWM value of 1200 microseconds as the desired cyclic-averaged wingbeat speed.

In order to find the maximum split-cycle time ratio that results in the same cyclic-averaged wingbeat speed, we have to start where s0 and s1 are the same speed and they result in the desired average wingbeat speed we are aiming to achieve. For my instance of the experiment, I am setting my desired cyclic-averaged wingbeat speed to be the speed of the motor when it is given a PWM value of 1200 microseconds. Because of this, I would start my s0 and s1 values both at 1200 microseconds, which should result in the same average wingbeat speed, as the motor is not actually going to change speeds.

After the starting s0 and s1 values have been found, it is then a matter of increasing one and decreasing the other to see how far apart they can be while still outputting the same desired cyclic-averaged wingbeat speed. The goal here is to get the speed values as far away from each other as possible, while still maintaining a stable cyclic-averaged speed. Once the average wingbeat speed starts to become unstable, meaning the average cycle speed can no longer be achieved smoothly, that will be where the motor and ESC combination cannot go for flight. The last s0 and s1 values where there was a stable average cycle speed is what will be used to determine the maximum split-cycle time ratio.

The code will be displaying the cyclic-averaged wingbeat speed, along with the amount of time spent on the s0 cycle (half rotation) and the s1 cycle (half rotation). In order to figure out the maximum split-cycle time ratio, we would take the times of s0 and s1 from the last stable average cycle speed, and add them together. This will give us the total amount of time in a single cycle. We would then divide the s0 time by the total time to get the percentage of time the s0 speed was spinning. The same is done with the s1 time. These times will give us the maximum split-cycle time ratio that can be achieved with the used set of ESC and motor.

CHAPTER 9

EXPERIMENT RESULTS

When I started working on this experiment, many things went wrong. I knew that the PWM ESC worked by writing out a signal in microseconds, ranging from 1000 to 2000. I decided to start my analysis right in the middle, at 1500 microseconds. When I would spin the motor at 1500 microseconds, I would be able to get encoder step counts from the optical encoder. The problems started to occur when I would run the motor at higher PWM values. The encoder would display values for the first couple steps, but then would suddenly stop. I later figured out that this was because the motor was spinning the encoder too fast, and it wasn't registering every step due to the increased speed.

After figuring this out, I knew I was getting values at 1500, so I set my range to be from 1000 to 1500 microseconds, but even then I was getting some weird values. I was measuring how long a single rotation was taking at a constant speed. The amount of time taken to complete a single rotation went down, up until the PWM values hit higher than 1300 microseconds. The amount of time taken to complete a rotation started to increase after that, instead of decrease, which meant that the encoder was being spun too fast to register every step. This told me that I had to change the range to be only values between 1000 and 1300 microseconds. This allowed me to stay within a range of speeds where every step would be registered by the optical encoder.

Another problem I ran into was momentum. At first, the gear ratio I was using was simulating a 1:1 gear ratio between the motor and the output rotation count. One thing I started to realize was that the faster of the two speeds would carry over some momentum into the slower speeds cycle, making it spend less time in the slower rotation than it was supposed to. In order to fix this, I used a simulated gear ratio of 10:1, meaning that the motor would have to spin

the optical encoder 10 times in order to be registered as a single rotation. This gave each part of the split-cycle more time to adjust to its correct speed, which made the s0 and s1 times more realistic.

Once these pitfalls were overcome, the rest of the experiment fell into place (luckily). To start, I wanted to get an idea of the relationship between the PWM microsecond values and the output speed they were spinning at. In order to do this, I set both s0 and s1 to be the same PWM microseconds value, so that the motor was spinning at a constant speed. Using this, I was able to find the cyclic-averaged output speed of the commanded PWM speeds. Below are the set PWM values (constant speed) and five of the outputs given for a full rotation by the code, along with the cyclic-averaged speed, s0 time, and s1 times.

PWM: 1100, both half rotations

```
185.00 steps/microsecond (avg)
322967 s0 time, 322074 s1 time
185.00 steps/microsecond (avg)
323151 s0 time, 322445 s1 time
185.00 steps/microsecond (avg)
324196 s0 time, 322448 s1 time
185.00 steps/microsecond (avg)
323565 s0 time, 322451 s1 time
185.00 steps/microsecond (avg)
322928 s0 time, 322495 s1 time
```

Average Cycle Speed (x10,000): 185 steps/microsecond

Average s0 Time: 323361.4 microseconds

Average s1 Time: 322382.6 microseconds

PWM: 1150, both half rotations

```
285.00 steps/microsecond (avg)
209856 s0 time, 210212 s1 time
285.00 steps/microsecond (avg)
209846 s0 time, 210110 s1 time
285.00 steps/microsecond (avg)
209704 s0 time, 209934 s1 time
285.00 steps/microsecond (avg)
209847 s0 time, 210157 s1 time
285.00 steps/microsecond (avg)
209811 s0 time, 210107 s1 time
```

Average Cycle Speed (x10,000): 285 steps/microsecond

Average s0 Time: 209812.8 microseconds

Average s1 Time: 210104 microseconds

PWM 1200, both half rotations

```
419.00 steps/microsecond (avg)
142403 s0 time, 143163 s1 time
419.00 steps/microsecond (avg)
142554 s0 time, 143041 s1 time
419.00 steps/microsecond (avg)
142704 s0 time, 142958 s1 time
419.00 steps/microsecond (avg)
142690 s0 time, 142841 s1 time
419.00 steps/microsecond (avg)
142769 s0 time, 142985 s1 time
```

Average Cycle Speed (x10,000): 419 steps/microsecond

Average s0 Time: 142624 microseconds

Average s1 Time: 142997.6 microseconds

PWM 1250, both half rotations

```
544.00 steps/microsecond (avg)
109939 s0 time, 110266 s1 time
544.00 steps/microsecond (avg)
109776 s0 time, 110226 s1 time
545.00 steps/microsecond (avg)
109711 s0 time, 110095 s1 time
545.00 steps/microsecond (avg)
109649 s0 time, 110162 s1 time
545.00 steps/microsecond (avg)
109801 s0 time, 109996 s1 time
```

Average Cycle Speed (x10,000): 544.6 steps/microsecond

Average s0 Time: 109775.2 microseconds

Average s1 Time: 110149 microseconds

PWM 1300, both half rotations

```
631.00 steps/microsecond (avg)
94768 s0 time, 94960 s1 time
632.00 steps/microsecond (avg)
94616 s0 time, 94882 s1 time
630.00 steps/microsecond (avg)
94989 s0 time, 95097 s1 time
631.00 steps/microsecond (avg)
94841 s0 time, 95018 s1 time
631.00 steps/microsecond (avg)
94849 s0 time, 94900 s1 time
```

Average Cycle Speed (x10,000): 631 steps/microsecond

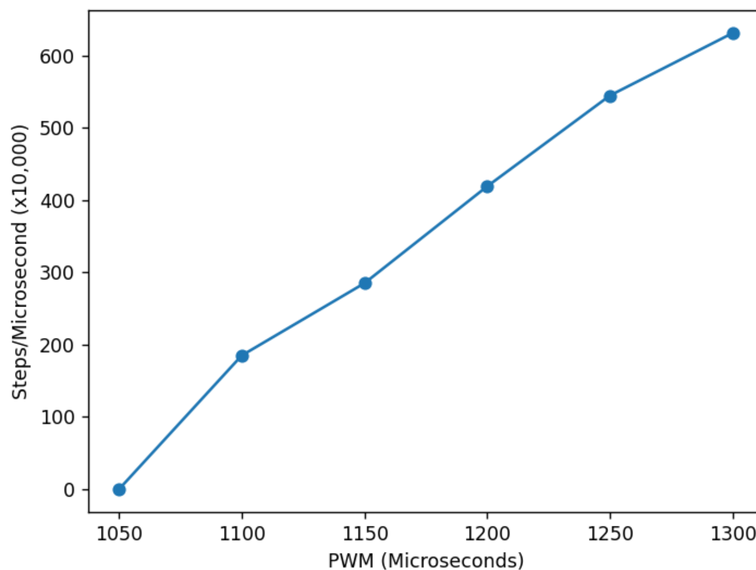
Average s0 Time: 94812.6 microseconds

Average s1 Time: 94971.4 microseconds

One thing to note is that the cyclic-averaged speed is multiplied by 10,000 in each case. This was done because the actual calculated average cycle speed values were far too small to be displayed. In reality, the actual steps/microsecond values would need to be divided by 10,000

in order to be the exact measured values, but they were multiplied by a constant of 10,000 in order to actually show up as numbers greater than zero for viewability.

I then used these values, with their commanded PWM speeds, to create a graph to get an idea of what kind of curve the average cycle speeds versus PWM values followed. Here is the resulting graph:



The graph follows a mostly linear line, though there are some fluctuations. This is mostly due to the fact that the PWM microsecond values correlate to a percentage of the motor speed. So when the PWM value is 1200, this correlates to 20% of the motors total speed. Other ESC protocols might not have a linear correlation between the input ESC value and the output motor speed.

With an idea of the cyclic-averaged speeds versus PWM values, I then attempted to find a split-cycle time ratio. In order to do this, I needed an achievable speed. Originally I was going to use the PWM value of 1500 microseconds because it was right in the middle of the possible motor speeds, but once I figured out that was too high of a speed for the optical encoder to be able to read, I had to adjust my values. I knew my range was between 1000 and 1300 microseconds, so I set the goal to be the average cycle speed at 1200 microseconds. This

meant that the average wing speed I was attempting to achieve would be 419 steps/microsecond (x10,000).

One thing I noticed as I was doing the experiment was that the cyclic-averaged speed values were not constant every cycle. For instance, some cycles would be faster than others, which was not something I had seen in the constant speed measurements. This is why I decided to take the average of five of the values and use that to compare against the 419 steps/microsecond (x10,000). I also realized that hitting an exact average cycle speed of 419 steps/microsecond (x10,000) was extremely difficult with the PWM ESC, so I gave myself a window of error of +-10 steps/microsecond (x10,000). This means that I was satisfied with average cycle speed values that were within the range of 409-429 steps/microsecond (x10,000). One thing to keep in mind is that this error of +- 10 was being applied to values that were multiplied by a constant of 10,000, so it isn't actually an error of +- 10, but rather +- 0.001 steps/microsecond when not being multiplied by 10,000.

I tried moving the s0 value down (to a slower speed) and the s1 value up (to a higher speed) to see if they could achieve an average cycle speed that was within the error window I had set. I did this with multiple sets of s0 and s1 values, adjusting them to see how close I could get to the desired 419 steps/microsecond (x10,000). The results of the closest cyclic-averaged speed values I managed to achieve are listed below, along with each one's resulting split-cycle time ratio (s0:s1).

s0 = 1150, s1 = 1275, goal = 419 steps/microsecond (x10,000)

```
418.00 steps/microsecond (avg)
166491 s0 time, 120228 s1 time
404.00 steps/microsecond (avg)
173555 s0 time, 122815 s1 time
403.00 steps/microsecond (avg)
170202 s0 time, 126628 s1 time
418.00 steps/microsecond (avg)
167164 s0 time, 119515 s1 time
403.00 steps/microsecond (avg)
174335 s0 time, 122443 s1 time
```

Average Cycle Speed (x10,000): 409.2 steps/microsecond

Average s0 Time: 170349.4 microseconds

Average s1 Time: 122325.8 microseconds

Total Time: 292675.2 microseconds

s0:s1 Split-Cycle Time Ratio: 58:42

s0 = 1150, s1 = 1300, goal = 419 steps/microsecond (x10,000)

```
448.00 steps/microsecond (avg)
153567 s0 time, 114015 s1 time
431.00 steps/microsecond (avg)
161730 s0 time, 116063 s1 time
431.00 steps/microsecond (avg)
159649 s0 time, 118382 s1 time
431.00 steps/microsecond (avg)
157777 s0 time, 120152 s1 time
431.00 steps/microsecond (avg)
155953 s0 time, 122035 s1 time
```

Average Cycle Speed (x10,000): 434.4 steps/microsecond

Average s0 Time: 157735.2 microseconds

Average s1 Time: 118129.4 microseconds

Total Time: 275864.6 microseconds

s0:s1 Split-Cycle Time Ratio: 57:43

s0 = 1125, s1 = 1300, goal = 419 steps/microsecond (x10,000)

```
397.00 steps/microsecond (avg)
186999 s0 time, 114755 s1 time
381.00 steps/microsecond (avg)
190050 s0 time, 124144 s1 time
397.00 steps/microsecond (avg)
180954 s0 time, 120843 s1 time
396.00 steps/microsecond (avg)
184543 s0 time, 117675 s1 time
381.00 steps/microsecond (avg)
188824 s0 time, 125205 s1 time
```

Average Cycle Speed (x10,000): 390.4 steps/microsecond

Average s0 Time: 186274 microseconds

Average s1 Time: 120524.4 microseconds

Total Time: 306798.4 microseconds

s0:s1 Split-Cycle Time Ratio: 61:39

s0 = 1140, s1 = 1300, goal = 419 steps/microsecond (x10,000)

```
429.00 steps/microsecond (avg)
161787 s0 time, 117020 s1 time
431.00 steps/microsecond (avg)
161099 s0 time, 116894 s1 time
431.00 steps/microsecond (avg)
159006 s0 time, 118827 s1 time
430.00 steps/microsecond (avg)
158025 s0 time, 120179 s1 time
429.00 steps/microsecond (avg)
156211 s0 time, 122863 s1 time
```

Average Cycle Speed (x10,000): 430 steps/microsecond

Average s0 Time: 159225.6 microseconds

Average s1 Time: 119156.6 microseconds

Total Time: 278382.2 microseconds

s0:s1 Split-Cycle Time Ratio: 57:43

s0 = 1145, s1 = 1280, goal = 419 steps/microsecond (x10,000)

```
404.00 steps/microsecond (avg)
172787 s0 time, 123458 s1 time
405.00 steps/microsecond (avg)
168667 s0 time, 127045 s1 time
420.00 steps/microsecond (avg)
164321 s0 time, 121027 s1 time
404.00 steps/microsecond (avg)
170657 s0 time, 125597 s1 time
420.00 steps/microsecond (avg)
166482 s0 time, 118965 s1 time
```

Average Cycle Speed (x10,000): 410.6 steps/microsecond

Average s0 Time: 168582.8 microseconds

Average s1 Time: 123218.4 microseconds

Total Time: 291801.2 microseconds

s0:s1 Split-Cycle Time Ratio: 58:42

s0 = 1145, s1 = 1285, goal = 419 steps/microsecond (x10,000)

```
419.00 steps/microsecond (avg)
165250 s0 time, 120757 s1 time
404.00 steps/microsecond (avg)
172464 s0 time, 124028 s1 time
404.00 steps/microsecond (avg)
168917 s0 time, 127793 s1 time
419.00 steps/microsecond (avg)
165408 s0 time, 120546 s1 time
404.00 steps/microsecond (avg)
172496 s0 time, 124075 s1 time
```

Average Cycle Speed (x10,000): 410 steps/microsecond

Average s0 Time: 168907 microseconds

Average s1 Time: 123439.8 microseconds

Total Time: 292346.8 microseconds

s0:s1 Split-Cycle Time Ratio: 58:42

When looking at these results, only three of them fall within the error range that I had set. The PWM values with their average cycle speeds (formatted s0, s1, Average speed) were (1150, 1275, 409.2), (1145, 1280, 410.6), and (1145, 1285, 410). Each of these sets ended with a split-cycle time ratio of 58:42, meaning that the wing would have spent 58% of the time spinning at s0 speed, and the other 42% of the time spinning at s1 speed. Since they all have the same split-cycle time ratio, the best choice would be the one that had the closest cyclic-averaged wingbeat speed, which would be the second set, with an s0 PWM value of 1145, s1 PWM value of 1280, and an average cycle speed of 410.6 steps/microsecond (x10,000).

CHAPTER 10

CONCLUSION

This testbed currently works well enough to test motors that don't spin at very high rates. This can be improved upon by using an optical encoder that will be able to read changes in values at higher speeds. The microcontroller (ESP32) is set up to run at 240 MHz, which should be able to support very fast ESC protocols, so long as there is a library or specific code written to be able to control the ESC from a microcontroller. The code allows the user to program two distinct speed values into the code and have the motor spin at those two different values, switching between them every half cycle (or simulated half rotation depending on the gear ratio).

Even though PWM is one of the most basic ESC protocols, there are plenty of others that should be able to be used on this testbed. With a faster ESC protocol, the motor should be able to be updated faster, making the testbed even more effective at changing the speed of the motor at faster intervals. This would result in a faster rate of change from the s0 to the s1 speed, as well as the other way around.

Unfortunately, I was limited in my potential ESC values due to the limitations of the optical encoder. This means that I was only able to use PWM speed values that were between 1000 and 1300 microseconds. This means I have 300 potential values to choose from, instead of the normal 1000. This limited my range of speed values, which is why the split-cycle time ratio wasn't able to exceed 58:42. If I had access to the entire range, I might have potentially been able to find a much faster speed for s1 and a slower speed for s0, while still having the desired cyclic-averaged wingbeat speed. This would've resulted in a larger difference in the time spent on each cycle, which would in turn create a larger difference in the split-cycle time ratio. With a larger split-cycle time ratio, whatever drone the flight system was used in would have greater

maneuverability due to the larger difference in time it could support with the same average cycle speed.

The PWM ESC protocol is considered to be relatively slow when compared to other ESC protocols. It takes a long time for a PWM packet to be sent from the microcontroller to the ESC, where other ESC protocols take a fraction of that amount of time. When compared to the DShot ESC protocol, a digital ESC protocol that has gained in popularity in recent years, the PWM protocol can take up to 20 times as long as the slowest DShot ESC protocol (DShot150). That means that the motor can be updated 20 times with DShot150 within the time it takes to make a single motor update on the PWM protocol.

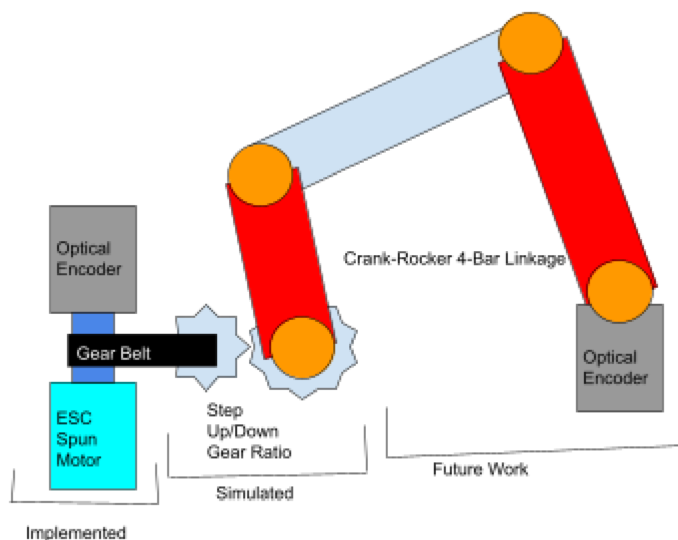
The DShot protocol is much faster than the PWM protocol, but it also has a better resolution. That means that the DShot has 2000 distinct possible ESC values, whereas PWM only has 1000. Due to the higher resolution, the DShot ESC protocol has a higher chance at getting much closer to the desired cyclic-averaged wingbeat speed than the PWM ESC protocol does.

This testbed could certainly use some specialized hardware, but it still manages to work with the PWM ESC protocol and an optical encoder with 1200 distinct step values per rotation. Even with a limited range of potential speed values, the testbed managed to achieve a 58:42 split-cycle time ratio within an error range of +-10 steps/microsecond (x10,000) away from the desired speed of 419 steps/microsecond (x10,000). With improved hardware, this testbed should be able to handle even faster ESC protocols and motor speeds, resulting in a better range of analysis capabilities.

CHAPTER 11

FUTURE WORKS

Currently, this system only works with a given ESC and a motor. It tests to figure out if a given ESC and motor can accomplish a given cyclic-averaged wingbeat speed, and tests how far the split-cycle time ratio can go while maintaining a stable average cycle speed. There are many other parts that can be implemented and added onto this system to get more information on the flight system out of it.



One thing that could be added to this testbed in the future would be a gear system. Currently, it is operating on the idea of a 1:1 gear ratio, meaning that a single revolution on the motor itself results in an output of a single rotation to the 4 bar linkage. This doesn't always have to be the case. Sometimes the motor might spin too fast for a system, which would require a step-down gear ratio. A common step-down gear ratio would be 2:1, meaning that for every 2 rotations of the motor, only 1 rotation is accomplished on the 4 bar linkage. If a gear system was set up to be something other than 1:1, there would need to be changes made in the code to reflect the new optical encoder values for what a half rotation would look like. In the experiment,

I needed a higher gear ratio than 1:1, so I multiplied the full variable by 10, giving the system a 10:1 gear ratio. If there was a physical system implemented, it would be nice to be able to physically change the gear ratio with actual gears, rather than changing the code values for it.

Another aspect that could be added into the system would be the actual physical implementation of the crank-rocker 4-bar linkage. Currently, the system only gives an output cyclic-averaged speed, but with a physical implementation of the crank-rocker 4-bar linkage, the system would be able to show an actual output speed that the wing would be traversing. Not only would it provide the user with a measurable output speed for the wing, but it would also allow for the user to swap out linkage lengths to see what effects they may have on the speed and arc of the output of the system for the wing. This would also allow the user to determine what size the entire flapping wing system might need to be to get the desired lift, while also being able to fit inside of the body of the system.

Since this system requires a manual changing of speed values to achieve the average cycle speed, it would be nice to be able to allow the system to change those values without the user having to change them manually. One of the best ways I've come across for doing this would be to implement a PID controller into the system. Once the 4-bar linkage is set up, another optical encoder could be used to measure the speed of the output crank. This could be used in a feedback loop containing the entire system, which would allow the code to change the ESC values to match an expected output speed value. Implementing this would then make the entire system a closed system. This could even be set up in a way that the user would only have to input the desired cyclic-averaged speed, and the PID loop could be used to achieve that speed and even attempt to find the maximum split-cycle time ratio that can work for that system.

During the experiment, I found that the optical encoder I was using wouldn't read properly when the PWM motor speed values were higher than 1300 microseconds. Because of this, it would be nice to have an optical encoder that would be able to read and register the correct position values of higher speed motors.

REFERENCES

David B. Doman, Michael W. Oppenheimer, and David O. Sigthorsson. "Wingbeat Shape Modulation for Flapping-Wing Micro-Air-Vehicle Control During Hover". In: *Journal of Guidance, Control, and Dynamics* 33.3 (2010), pp. 724–739. doi: 10.2514/1.47146. eprint: https://doi.org/10.2514/1.47146. Url: https://doi.org/10.2514/1.47146.

James Paulos and Mark Yim. "Flight Performance of a Swashplateless Micro Air Vehicle". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. May 2015, pp. 5284–5289. doi: 10.1109/ICRA. 2015.7139936.

Chris Landa. "DSHOT - the Missing Handbook." Brushless Whoop, 26 Feb. 2021, brushlesswhoop.com/dshot-and-bidirectional-dshot/.

Hiren Tailor. "ESP32 vs Arduino Speed Comparison.", Makerguides.Com, 19 Aug. 2023, www.makerguides.com/esp32-vs-arduino-speed-comparison/.

Earley, Eric, et al. "What Is DShot ESC Protocol." Oscar Liang, 30 Mar. 2023, oscarliang.com/dshot/.

alikhan968. "Measuring Speed Using Rotary Encoder." Arduino Forum, 9 Sept. 2018, forum.arduino.cc/t/measuring-speed-using-rotary-encoder/545398.

Sandeep. "How to Connect Optical Encoder with ESP32." Electric DIY Lab, 22 Mar. 2023, electricdiylab.com/how-to-connect-optical-encoder-with-esp32/.

"PWM, Oneshot125, OneShot42, Multishot and DSHOT Comparison." QuadMeUp, 31 Aug. 2016, quadmeup.com/pwm-oneshot125-oneshot42-and-multishot-comparison/.

"PWM, Oneshot and Oneshot125 Escs." PWM, OneShot and OneShot125 ESCs - Copter Documentation, ardupilot.org/copter/docs/common-brushless-escs.html. Accessed 11 Dec. 2023.

DroneBot Workshop. "Getting Started with the ESP32 - Using the Arduino Ide." DroneBot

Workshop, Publisher Name DroneBot Workshop Publisher Logo, 11 Apr. 2023, dronebotworkshop.com/esp32-intro/.

Staff, LME Editorial. “ESP32 Pinout Reference.” Last Minute Engineers, Last Minute Engineers, 23 Nov. 2023, lastminuteengineers.com/esp32-pinout-reference/.

“PID.” PID - Arduino Reference, www.arduino.cc/reference/en/libraries/pid/. Accessed 11 Dec. 2023.

ESP32-WROOM-32 (ESP-WROOM-32) Datasheet - Mouser Electronics, www.mouser.com/datasheet/2/891/esp-wroom-32_datasheet_en-1223836.pdf. Accessed 11 Dec. 2023.

Team, The Arduino. “Basics of PWM (Pulse Width Modulation).” Arduino Documentation, docs.arduino.cc/learn/microcontrollers/analog-output. Accessed 11 Dec. 2023.

“Pulse Width Modulation.” Pulse Width Modulation - SparkFun Learn, learn.sparkfun.com/tutorials/pulse-width-modulation/all. Accessed 11 Dec. 2023.

Staff, General. “What Is an Electronic Speed Controller & How Does an ESC Work.” Tyto Robotics, Tyto Robotics, 15 Aug. 2023, www.tytorobotics.com/blogs/articles/what-is-an-esc-how-does-an-esc-work.

Sales, APD. “What Is an ESC?” Advanced Power Drives, Advanced Power Drives, 3 Aug. 2022, powerdrives.net/blog/what-is-an-esc.

Paul, Andrew. “Reverse-Engineered Hummingbird Wings Could Inspire New Drone Designs.” Popular Science, 30 Dec. 2022, www.popsci.com/technology/reverse-engineered-hummingbird-wings-drone/.

“Bi-Directional Logic Level Converter Hookup Guide.” Bi-Directional Logic Level Converter Hookup Guide - SparkFun Learn, learn.sparkfun.com/tutorials/bi-directional-logic-level-converter-hookup-guide/all. Accessed 11 Dec. 2023.

“A Four Bar Linkage#.” A Four Bar Linkage - SymPy 1.12 Documentation, 9 May 2023,

docs.sympy.org/latest/modules/physics/mechanics/examples/four_bar_linkage_example
html.