RIG INDUCTION

PROJECT REPORT

RIGGU CONTROL INTERFACE

TABLE OF CONTENTS

- OBJECTIVES
- KEY TAKEAWAYS
- CONCLUSION

OBJECTIVES

The given objective of RIG Induction round 3 is:

- Developing Python nodes in ROS2 to control the robot
- Implement a sound-based head-rotating mechanism that aligns the head to the source of the sound
- Design and Implementing Battery management circuit

KEY TAKEAWAYS

- Learned basic functionality of microcontroller-based boards: ESP8826-12E and Arduino Uno.
- Simulated circuits with online tools like TinkerCad and Wokwi.
- Built an RC car using ESP8826-12E and L293D to control dual motors, exploring multiple motion control methods.
- Developed communication skills with the microcontroller via Serial communication and WiFi, enabling remote control of the RC car.
- Controlling motors using keyboard w, a, s, d keys through Arduino serial monitor using esp8266, Also controlling motors using arrow keys through html server.
- Learned the l293d motor drive working in controlling the motors
- Learned some basic functions in Arduino programming
- Dual booted system and installed popos for working on ros2 and gazebo simulation

- Learned the working of boosters and buck converters and their purpose in riggu
- Also installed proteus and learned a few functionalities for circuit simulation, and applied it to BMS
- Learned about types of motors and their applications
- Observed all different components of the riggu robot
- Learned the functions of lipo battery and also the consequences of improper functioning of lipo battery
- Learned how the lidar sensor works and its functionality in Riggu
- Explored algorithms used in sound detection for head rotating mechanism in the direction of sound
- Started working on the battery management circuit
- Started learning about the ros2 and implementation of nodes
- Started working on the head rotating mechanism, and started working on its CAD model
- Tried tuning the buck converters, and boosters and setting them to a certain voltage

ELECTRONICS

Here’s a detailed breakdown of Riggu’s battery management circuit:

- Primary Components:
  - Utilizes an ATmega32 microcontroller.
  - Includes voltage sensors, current sensors, and a relay board.

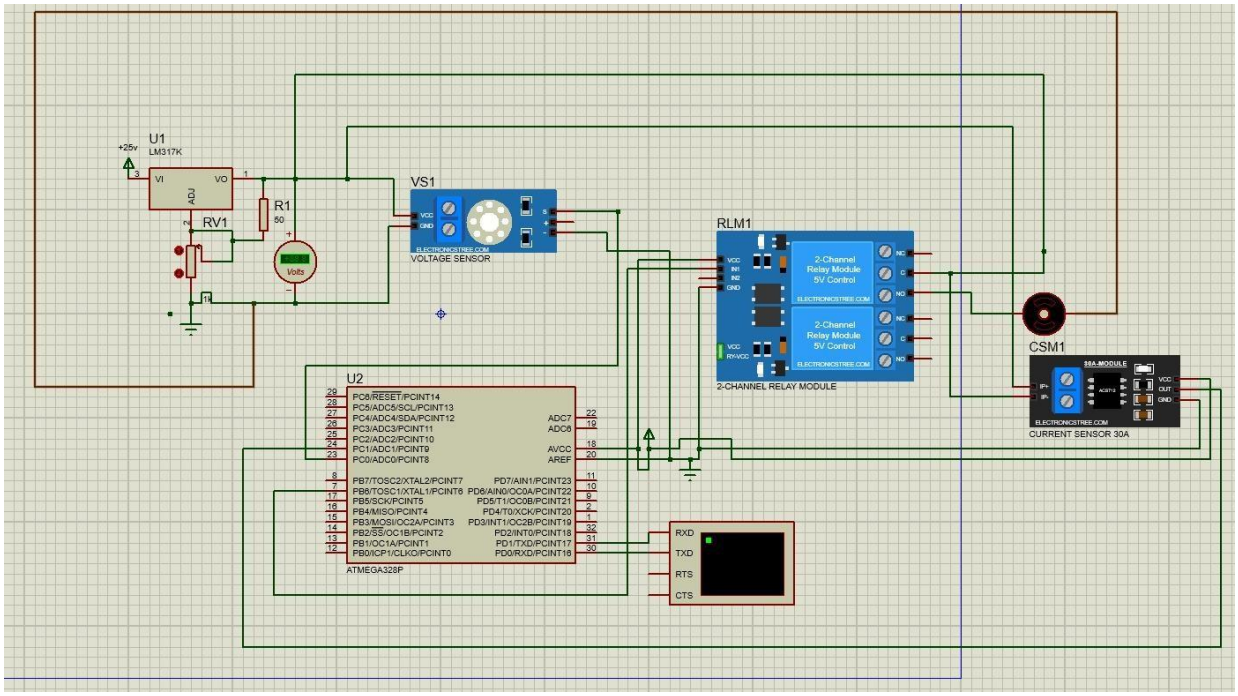
- Battery Monitoring Functionality:
  - Continuously tracks the voltage and discharging current of the battery.
  - Automatically alerts the user when:
    - Battery voltage falls below a predetermined threshold.
    - Discharging current reaches a level that could damage the battery.

- Variable Voltage Source Design:
  - Built using:
    - A constant power supply.
    - LM317 voltage regulator to ensure stable output.
    - Potentiometer for adjusting the voltage.

- Voltage Level Measurement:
  - Voltage sensor measures the battery’s voltage and sends readings to the analog input of the Arduino.

- Automatic Relay Activation:
  - Arduino is programmed to activate the relay if the voltage drops below the set limit, ensuring battery protection.

This design enhances the reliability and safety of Riggu's battery system by maintaining optimal operating conditions.



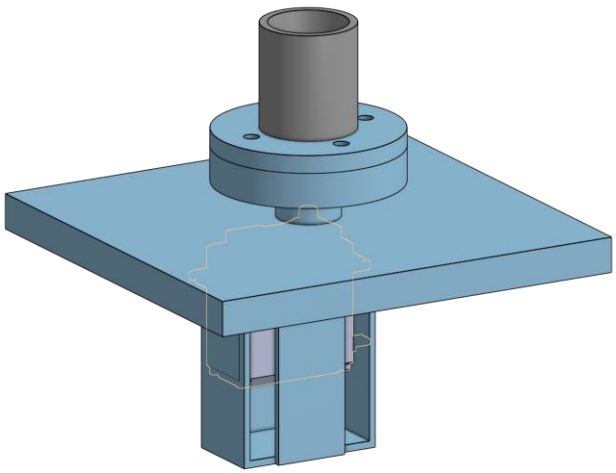
The Arduino code:

```
const int voltagePin = A0;
const int currentPin = A1;
const int relayPin = 7;


float voltageFactor = 5.0 / 1023.0 * (10000 + 2000) / 2000;
float acsOffset = 2.5;
float acsSensitivity = 0.3;


float voltageThreshold = 10.5;
float currentThreshold = 5.0;

void setup() {
  Serial.begin(9600);
  pinMode(relayPin, OUTPUT);
  digitalWrite(relayPin, LOW);
}

void loop() {

  int rawVoltage = analogRead(voltagePin);
  float batteryVoltage = rawVoltage * voltageFactor;


  int rawCurrent = analogRead(currentPin);
  float current = (rawCurrent * 5.0 / 1023.0 - acsOffset) /
acsSensitivity;
```

```
  Serial.print("Battery Voltage: ");
  Serial.print(batteryVoltage);
  Serial.println(" V");

  Serial.print("Battery Current: ");
  Serial.print(current);
  Serial.println(" A");


  if (batteryVoltage < voltageThreshold || current >
currentThreshold) {
    digitalWrite(relayPin, LOW);
    Serial.println("Relay OFF - Voltage or Current threshold
exceeded.");
  } else {
    digitalWrite(relayPin, HIGH);
    Serial.println("Relay ON");
  }

  delay(1000);
}
```

DESIGN AND CAD MODEL OF ROTATING MECHANISM

Here’s the clear and structured breakdown of the CAD



model for Riggu’s new head-rotating mechanism:

- Base Mounting Plate:
  - Constructed with a hole in the geometric center to secure the servo motor shaft.

  - Plate is dimensioned and cut from durable acrylic boards.

- Servo Motor Mounting:
  - MG 996 servo motor fixed securely to the base plate using a bracket and screws.

- Servo Head and Shaft Connection:
  - Circular servo head with a wider circular base attached to the top of the motor shaft.

- Neck Structure:
  - An additional circular base with a hollow cylindrical piece forms the "neck" of the robot.

- Clearances and Bearings:
  - Designed clearances between parts.
  - Bearings incorporated to reduce friction and enhance smooth rotation.

This configuration ensures precision in head rotation and minimizes mechanical resistance.

Developing Python nodes for joystick-based control of the robot

This project aims to control Riggu, a differential-drive robot, using a joystick. The system utilizes ROS 2 to translate joystick inputs into motor control commands for Riggu, enabling real-time and responsive control. The setup involves:

- A **ROS 2 Node (`driver_node`)** to interpret joystick inputs.
- A **Microcontroller Node (`connector_node`)** on ESP32 to control the motors.
- **Micro-ROS agent** running in Docker for communication with ESP32.
- **Differential Drive Control Logic** to manage left and right wheels separately.

Implementation Overview

- **Setup ROS 2 Workspace and Package**
  - Created workspace (`driver_workspace`) and package (`driver`) for `driver_node`.
- **Joystick Input Configuration**
  - Utilized ROS 2 `joy_node` to publish joystick data on `/joy`.
- **Driver Node Development**
  - `driver_node` subscribes to `/joy`, converting joystick axes into PWM values for motor control.
  - **PWM Calculation**: Y-axis for linear velocity, X-axis for angular velocity, scaled to motor requirements.
  - Published PWM signals to `/left_wheel/control_effort` and `/right_wheel/control_effort`.
- **Microcontroller Node Setup**
  - `connector_node` on the microcontroller subscribes to `/control_effort` topics, relaying PWM signals to motors.
- **Hardware Connections**
  - Left Wheel: GPIO 18, 19; Right Wheel: GPIO 21, 22 for PWM and direction.

- **Testing and Validation**
  - Verified node connections with `rqt_graph` and ensured accurate motor response to joystick input.

driver_node- Processes joystick data and publishes PWM values for differential drive control.

Python Code

```
import rclpy

from rclpy.node import Node

from sensor_msgs.msg import Joy from

std_msgs.msg import Float32


class DriverNode(Node): def init

    _____(self):

        super()._____init____('driver_node')

        self.joy_subscriber = self.create_subscription(Joy, '/joy',
self.joy_callback, 10)
```

```
        self.left_wheel_pub = self.create_publisher(Float32,
'/left_wheel/control_effort', 10)

        self.right_wheel_pub = self.create_publisher(Float32,
'/right_wheel/control_effort', 10)

        self.max_pwm = 255.0


    def joy_callback(self, msg): linear =

        msg.axes[1] angular = msg.axes[0]

        left_pwm = (linear + angular) * self.max_pwm right_pwm = (linear -

        angular) * self.max_pwm

        left_pwm = max(min(left_pwm, self.max_pwm),
-self.max_pwm)

        right_pwm = max(min(right_pwm, self.max_pwm),
-self.max_pwm)

self.left_wheel_pub.publish(Float32(data=left_pwm))self.right_wheel_pub.publis

h(Float32(data=right_pwm))


        self.get_logger().info(f"Left PWM: {left_pwm}, Right PWM:
{right_pwm}")


def main(args=None):

    rclpy.init(args=args) node =

    DriverNode() try:

        rclpy.spin(node)
```

```
        except KeyboardInterrupt: pass
        node.destroy_node()
        rclpy.shutdown()
if ___name_____== '____main____':
        main()
```

2. Arduino Code for connector_node on ESP32

This Arduino code subscribes to `/left_wheel/control_effort` and `/right_wheel/control_effort`, receives the PWM values from `driver_node`, and sends them to the Cytron motor driver to control Riggu's left and right wheels

Arduino Code

```
#include <micro_ros_arduino.h> #include <rcl/rcl.h>
#include <rcl/error_handling.h> #include <rclc/rclc.h>
#include <rclc/executor.h>
#include <std_msgs/msg/float32.h>

rcl_subscription_t left_wheel_sub;
rcl_subscription_t right_wheel_sub; std_msgs_msg
_____Float32 left_msg; std_msgs__msg
_____Float32 right_msg; rclc_executor_t
executor; rclc_support_t support;  rcl_allocator_t
allocator;
rcl_node_t node;
#define LEFT_PWM_PIN 18
#define LEFT_DIR_PIN 19
#define RIGHT_PWM_PIN 21
#define RIGHT_DIR_PIN 22

void error_loop(){ while(1){
  delay(100);
  }
}

void left_wheel_callback(const void * msgin) { const std_msgs
_____msg___Float32 * msg = (const std_msgs_msg__
```

```
_____Float32 *)msgin;
  int pwm_value = constrain(abs(msg->data), 0, 255); bool direction =
  (msg->data >= 0); digitalWrite(LEFT_DIR_PIN, direction);
  analogWrite(LEFT_PWM_PIN, pwm_value);
}

void right_wheel_callback(const void * msgin) { const std_msgs
_____msg___Float32 * msg = (const std_msgs_msg__
_____Float32 *)msgin;
  int pwm_value = constrain(abs(msg->data), 0, 255); bool direction =
  (msg->data >= 0);
  digitalWrite(RIGHT_DIR_PIN, direction); analogWrite(RIGHT_PWM_PIN,
  pwm_value);
}

void setup() { set_microros_transports();

  pinMode(LEFT_PWM_PIN, OUTPUT); pinMode(LEFT_DIR_PIN, OUTPUT);
  pinMode(RIGHT_PWM_PIN, OUTPUT); pinMode(RIGHT_DIR_PIN,
  OUTPUT);

  allocator = rcl_get_default_allocator();

  RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));
RCCHECK(rclc_node_init_default(&node, "connector_node", "",
&support));


RCCHECK(rclc_subscription_init_default(&left_wheel_ sub, &node,
ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),
"left_wheel/control_effort"));
```

```
RCCHECK(rclc_subscription_init_default(&right_wheel
_sub, &node, ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg,
Float32), "right_wheel/control_effort"));

  RCCHECK(rclc_executor_init(&executor, &support.context, 2, &allocator));

RCCHECK(rclc_executor_add_subscription(&executor, &left_wheel_sub,
&left_msg, &left_wheel_callback, ON_NEW_DATA));

RCCHECK(rclc_executor_add_subscription(&executor,  &right_wheel_sub,
&right_msg, &right_wheel_callback, ON_NEW_DATA));
}

void loop() { RCCHECK(rclc_executor_spin_some(&executor,
RCL_MS_TO_NS(100)));
}
```

CONCLUSION

Throughout the development of Riggu, I learned the importance of balancing multiple needs to create a reliable and efficient robot. For example, **power usage** needed to be carefully managed to keep the robot operating for extended periods without frequent recharging or risking damage to internal electronics. **Smooth mechanical motion** was another crucial factor. The head rotation mechanism required precise control to move fluidly without strain or interruption. This allowed Riggu’s head to turn smoothly, making it appear more lifelike and interactive. A well-designed mechanism not only adds functionality but also contributes to the robot’s overall reliability and lifespan by reducing wear on moving parts.