Abstract

The "METRO-W" (Managing Efficient Transportation Rail Operations Wirelessly) project demonstrates an innovative approach to model railway control through wireless communication technology and microcontroller automation. Using a Raspberry Pi 4 as the central controller and Pi Picos for locomotives, the system enables real-time monitoring and control through an ad hoc network. Features include rotary encoder speed monitoring, RFID-based track switching, and RSSI distance monitoring with emergency stop functionality. The system incorporates dual GUI interfaces for train control and custom 3D-printed tracks for servo-controlled switching. This implementation provides a cost-effective alternative to traditional DCC systems for battery-powered locomotives.

Acknowledgements

We would like to extend sincere gratitude to Texas Tech University's Electrical and Computer Engineering Department for providing the resources and facilities necessary for research and development, also extending this gratitude to our instructor, Samuel Storrs, for the guidance and encouragement throughout the project development process.

Additionally, we are grateful to the communities and forums, found and appropriated in References, for providing platforms to seek advice and share knowledge. The insights and solutions offered by these communities led to addressing technical issues and enhancing our understanding within the project.

Finally, we would like to thank our families and friends for their unwavering support throughout this project. Their encouragement helped us overcome various challenges and achieve our project goals.

We affirm that the work presented in this report was completed specifically for ECE 3332 Project Laboratory and has/will not be submitted for credit within another course.

Contents

List of Figures

List of Tables

1. Introduction

The evolution of model railway control systems has traditionally relied on Digital Command Control (DCC) for advanced functionality, creating a significant cost barrier for hobbyists and limiting innovations in battery-powered locomotives. METRO-W introduces a wireless microcontroller-based solution that delivers control features without requiring track-based power delivery or expensive DCC hardware. The system's TCP/IP communication architecture enables reliable real-time control while keeping low latency.

The implementation combines Raspberry Pi 4's processing capabilities with Raspberry Pi Picos' precise PWM control to achieve locomotive management. An H-bridge motor control system enables variable speed and bidirectional operation, while rotary encoders provide accurate velocity feedback. The RSSI-based distance monitoring system maintains train separation with 0.1-meter precision, automatically engaging emergency stops when the safety thresholds breach. Custom 3D-printed track sections integrate with standard track infrastructure, accommodating servo-controlled switching mechanisms.

The system architecture accentuates reliability through redundant detection methods, combining RFID positioning with IR sensors for precise depot control. The OpenCV-based dual GUI implementation enables independent control of multiple locomotives while continuously monitoring system safety parameters. This technical approach delivers DCC-comparable functionality for battery-powered locomotives at significantly reduced cost, while adding advanced safety features not typically available in traditional systems.

2. Hardware

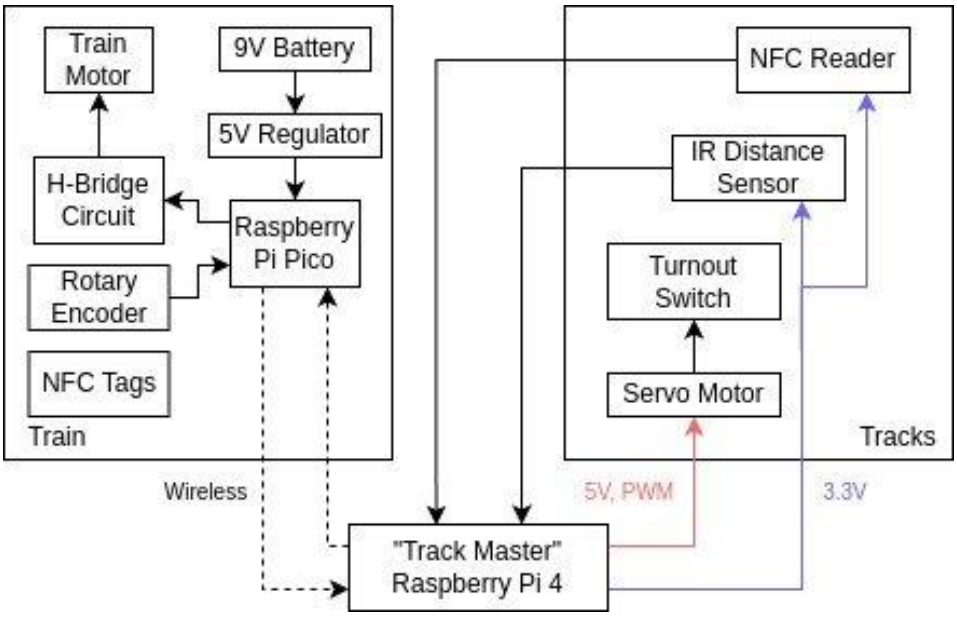


Figure 1: Hardware Block Diagram

2.1 Hardware Overview

The METRO-W system architecture enacts a hierarchical control structure utilizing a Raspberry Pi 4B as the central controller interfacing with multiple Raspberry Pi Pico W units for locomotive control. *Figure 1* illustrates the complete hardware system block diagram, showing the primary interconnections between components.

2.2 Central Control Station

The central control station employs a Raspberry Pi 4B operating at 1.5 GHz as the primary controller. The system establishes a wireless ad hoc network (WANET) through the onboard wireless interface, configured using hostapd and dnsmasq services.

The network configuration utilizes the WLAN0 interface configured as an access point with a static IP address of 192.168.4.1. The system operates on a 2.4 GHz frequency band under the network SSID "track-master", with client communications managed through TCP port 5000.

Table I: Base Station Peripherals

MFRC522 RFID:	**MG995 Servo:**	**HW-201 IP Sensor:**
SPI connection	PWM (GPIO12)	DigInput (GPIO11)
3.3 V supply	Operates with 5 V	Operates with 3.3 V
13.56 MHz Freq.	1350-1700 µs pulse	2-30 cm spot range
2-3 cm range	180 degree range	

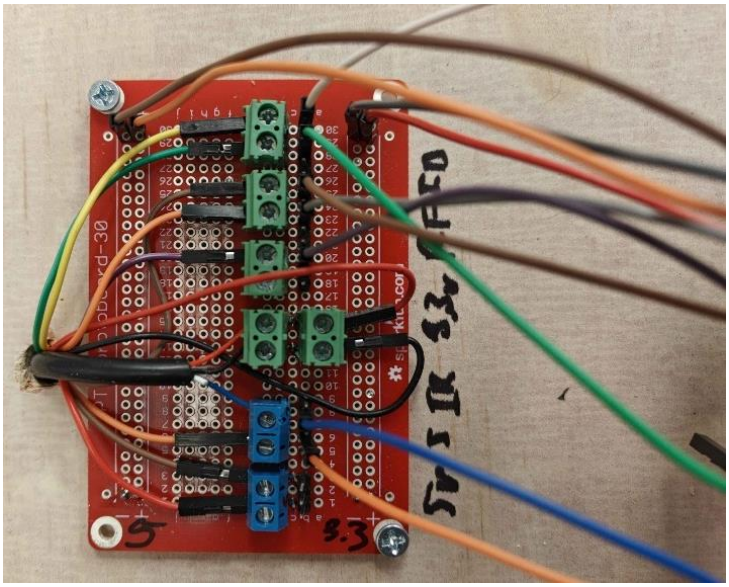


Figure 2: Hardware Central Control Station

2.3 Locomotive Units

Each locomotive unit integrates a Raspberry Pi Pico W microcontroller with motor control and power management systems. *Figure 3* and *Figure 4* presents the locomotive unit's hardware configuration, illustrating interconnections and power distribution.

The power system contains a 9 V Li-ion battery as power source with voltage regulation providing a stable 5 V output. Power distribution routes 5 V to Picos through its VSYS pin, while the motor driver receives 9 V directly from the battery. The system operates with current between 100-500 mA depending on motor load conditions.

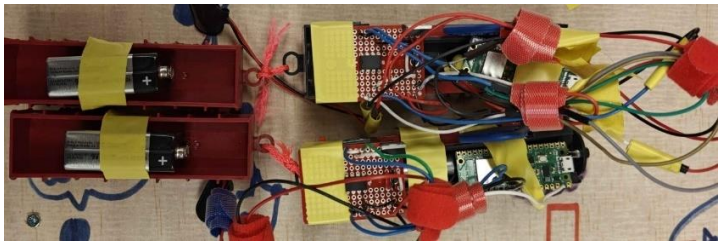


Figure 3: Locomotive Unit Configuration

The motor control system implements an L9110H H-bridge driver operating at 9 V with a PWM frequency of 100 Hz. The driver receives control signals through two GPIO connections, with pin GPIO 20 managing forward motion and GPIO 19 controlling reverse operation. Motor speed control uses 16-bit PWM resolution, providing a duty cycle range from 0-65536. The wireless interface operates at 2.4 GHz in client mode, establishing TCP protocol connections with the central station.

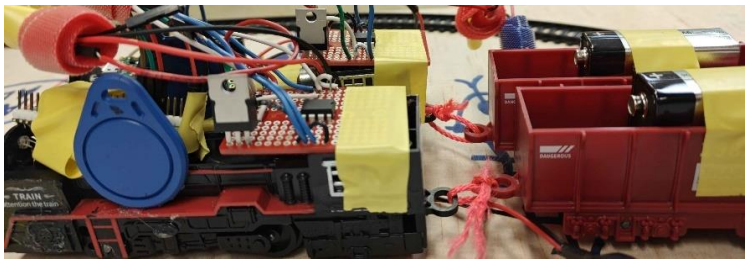


Figure 4: Locomotive Undercarriage RFID Tag

Mechanical integration of components includes RFID tag placement outside the locomotive undercarriage, with Train 1 and Train 2 availing unique identification numbers. The battery occupies the cargo compartment space, while the main circuit board mounts in the rear chassis section for optimal weight distribution and accessibility.

2.4 Track Switching Mechanism

The track switching system employs a servo-controlled turnout mechanism integrated with a custom 3D-printed track section. *Figure 5* demonstrates version evolution of the custom track, *Figure 6* demonstrates the RFID detection infrastructure, and *Figure 7* illustrates the mechanical integration of the servo system with the track components.

Figure 5: Track Printing Evolution

The servo control system uses an MG995 servo motor operating at 5 V, controlled through hardware PWM signals from the Raspberry Pi 4's GPIO 12 pin. Pulse width modulation signals range from 1350 µs to 1700 µs, corresponding to the turnout and straight positions. The servo operates at 50 Hz with torque specifications of 8.5 kg·cm.

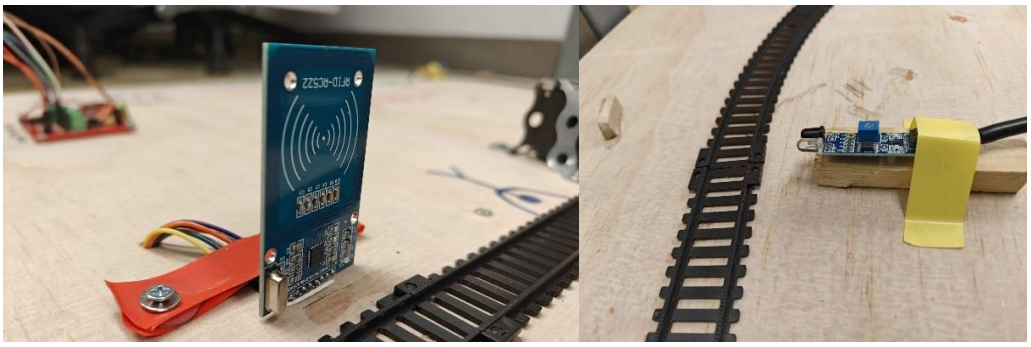


Figure 6: RFID & IR Sensor Detection Stand

Custom 3D-printed track sections maintain standard gauge while accommodating the servo mechanism's movement range. Mechanical connection points maintain compatibility with the existing track infrastructure, allowing smooth passage through the complete layout. The switching mechanism responds to RFID detection events through the MFRC522 reader, positioned 10 cm before the turnout for track realignment before/after train arrival. An IR sensor mounted at the depot terminal provides additional positioning feedback, triggering train stoppage when detected as occupied.

The testing demonstrated switch transition times of 200 ms between positions, with reliable switch mechanics through 100+ testing cycles. The servo mounting system maintains precise alignment while distributing stress across the custom support structure.

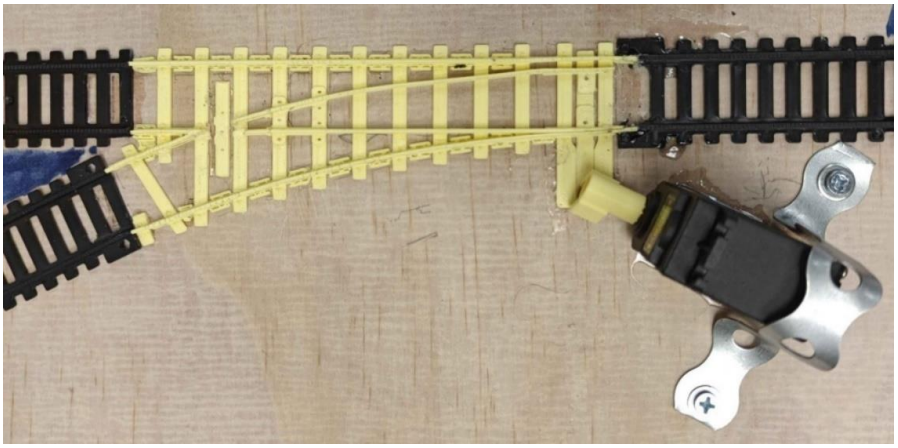


Figure 7: Custom Servo-Trigger Track Switch

2.5 Speed Monitoring System

The speed monitoring system implements a rotary encoder configuration providing velocity measurements through pulse counting. *Figure 8* demonstrates the encoder circuit while *Figure 9* displays the circuitry in LTspice.

The encoder generates 18 pulses per complete revolution, using two terminals for signal output. The GPIO 1 pin on each Pico monitors these pulses with an internal pull-up resistor, eliminating the need for external voltage reference circuitry. The system calculates speed using the following relationship:

$$\textbf{Speed}\left(\frac{\textbf{cm}}{\textbf{s}}\right) = \left(\frac{1}{36}\right) * Pulse_{Count} * 5.65487 * 2$$

The equation above includes a 1/36 conversion factor for pulse counting, while 5.65487 accounts for the wheel circumference measurement. The final multiplication by 2 provides adjustment for the measurement time window of the system.

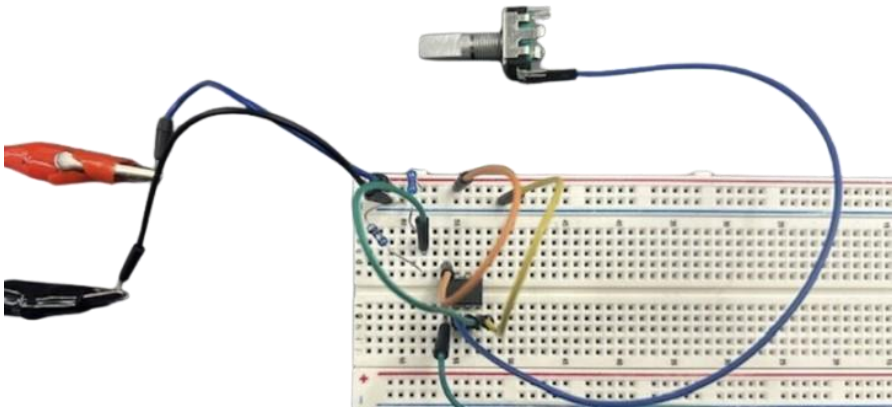


Figure 8: Rotary Encoder Circuit Demonstration

The monitoring system samples encoder pulses at 10 ms intervals, collecting counts over a 100-sample period to calculate average velocity. This sampling rate provides temporal resolution while maintaining pulse detection. The encoder mounts directly to the locomotive's wheel assembly, though mechanical limitations of the small wheel diameter affected reliable rotation during demonstration.

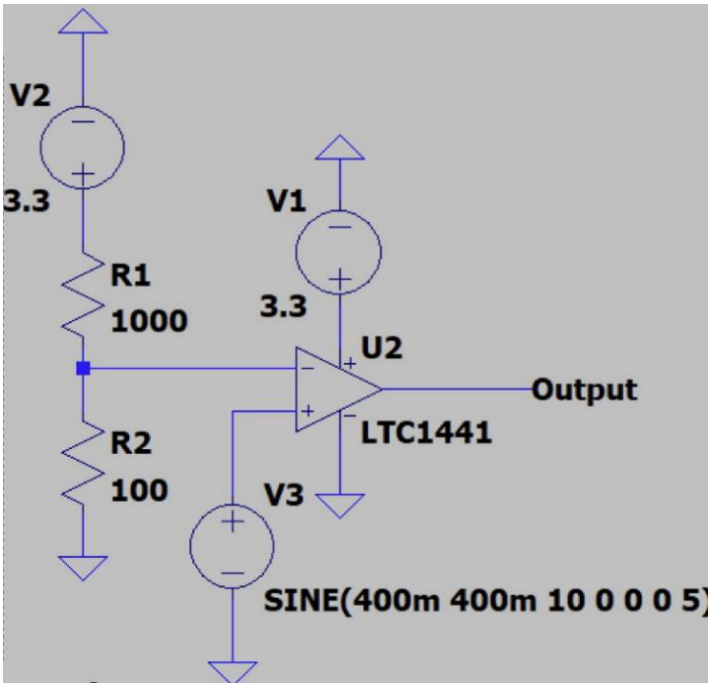


Figure 9: Rotary Encoder LTSpice Demonstration

2.6 Motor Control Circuit

The motor control system includes an L9110H H-bridge driver circuit for bidirectional locomotive control. *Figure 10* illustrates the H-bridge circuit configuration and PWM control connections, while *Figure 11* demonstrates the circuit in LTspice.

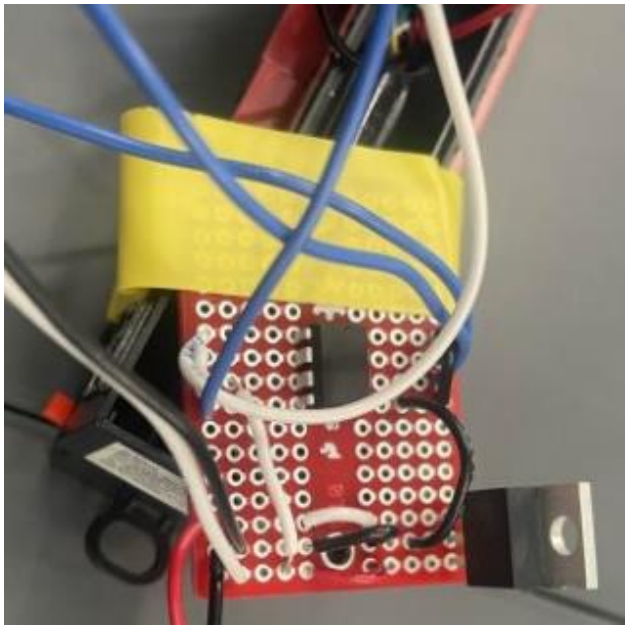


Figure 10: H-bridge Driver Configuration

The H-bridge circuit takes from the 9 V battery while accepting 3.3 V logic-level inputs from the Pico W. Two PWM channels allow independent motion control, forward (GPIO 20) and reverse (GPIO 19), both operating at 100 Hz and 16-bit resolution. The perfboard mixes the H-bridge and 5 V regulator, directly mounting to the rear chassis. Evaluating the motor control provided full speed range from 100 mA (idle) to 500 mA.

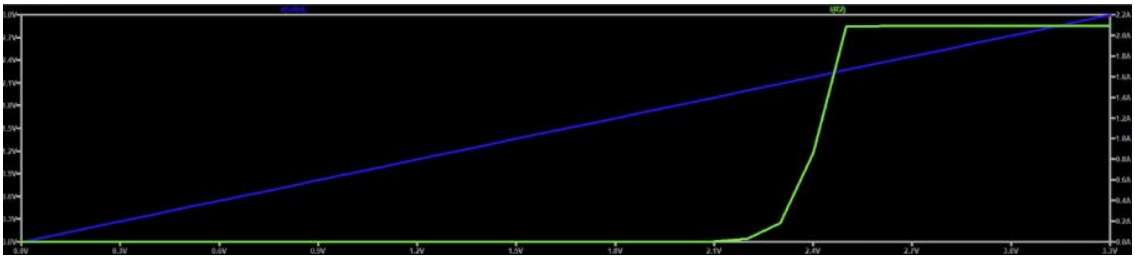


Figure 11: LTSpice H-Bridge Simulation

3. Software

Figure 12: Network Commands Flowchart

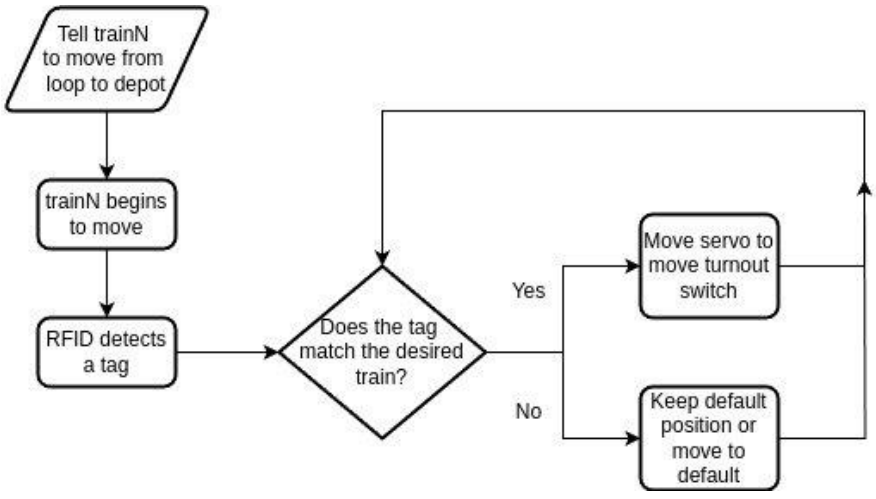


Figure 13: RFID Flowchart

3.1 Software Overview

The METRO-W system implements a multi-threaded software architecture operating across all microcontroller platforms. *Figure 13* presents the network commands block diagram and *Figure 14* illustrates the RFID components and their interactions.

3.2 Network Communication System

The system provides a TCP/IP-based client-server architecture operating on a wireless ad hoc network (WANET). The Raspberry Pi 4 hosts the network server with an IP address and port configuration of 192.168.4.1 and 5000.

The TCP protocol utilizes socket programming (SOCK_STREAM), with SO_REUSEADDR enabled, allowing a non-blocking operation with 1-second timeout intervals while maintaining a client/locomotive tracking dictionary. Each Pico operates as a network client with connection-retry automation (25 attempts), MAC registration, asynchronous handling, and connection-status monitoring. The network system achieves command transmission with latency under 100ms and packet delivery success exceeding 98% during testing.

3.3 Train Control Interface

The train control system implements a dual-window GUI using OpenCV 4.8.0, providing independent control interfaces for each locomotive. *Appendix D* presents the following implementations for GUI layout and control signal flow.

Each control window utilizes two trackbars for train operation control (*HostControl.py* – Lines 274-278). The system processes these trackbar positions at 250 ms intervals to ensure smooth control response (*HostControl.py* – Lines 297-307). Within the locomotive units, received commands are converted to PWM signals for motor control (*PicoConnect.py* – Lines 98-107). For safety purposes, the system monitors RSSI-derived

distances and implements emergency stops when safety thresholds are breached (*HostControl.py* – Lines 72-82).

The tests demonstrated control latency under 100 ms from user input to motor response, with emergency stop activation within 200 ms of safety threshold violation.

3.4 RFID Control System

The RFID system implements tag detection and turnout control logic for automated depot routing. *Appendix D* presents the implementations for RFID processing and servo control.

The system continuously monitors RFID tag detection using the MFRC522 reader (*HostControl.py* – Lines 241-256). Upon detection, the system verifies the tag ID against known train identifier values and checks the depot status flags to determine appropriate turnout actions. When a valid tag is detected and the corresponding train is designated for depot entry, the servo turnout activates through PWM signal adjustment.

Testing validated RFID detection reliability exceeding 95% at 3 cm range with servo response times under 200 ms.

3.5 Speed Monitoring System

The speed monitoring software implements an edge detection algorithm for rotary encoder pulse counting. *Appendix D* presents the implementations for signal processing and speed calculation methodology.

The system monitors encoder state changes through GPIO input with internal pull-up configurations (*PicoConnect.py* – Lines 32 & 186-190). Velocity computation executes at 100 ms intervals using the speed formula declared in Hardware section 2.5 (Speed Monitoring System) (*PicoConnect.py* – Line 196). To maintain accuracy, measurement

windows occur through counter reset operations (*PicoConnect.py* – Lines 192-199), with 200 MHz frequency for accurate pulse detection on inspection (*PicoConnect.py* – Line 36).

Testing validated speed measurements up to 30 cm/s with accuracy within ±5% under controlled conditions.

3.6 RSSI & Distance Estimation

The system provides live distance monitoring between locomotives using RSSI measurements, with an emergency stop at distances under 0.1 m giving response times at 200 ms. *Appendix D* presents the following implementations for RSSI processing and distance estimation.

Each Pico W reports signal strength at 1-second intervals to maintain consistent monitoring (*PicoConnect.py* – Lines 43-47). The system converts these RSSI values to distance estimates using the path loss model equated below for accurate positioning (*HostControl.py* – Lines 45-51). For safety purposes, the system implements three distance-based safety zones to ensure proper train separation (*HostControl.py* – Lines 40-41 & 75-82). The distance status prompts occur at 1-second intervals to the console output (*HostControl.py* – Lines 97-103).

$$distance = 10 ** \frac{|-50| - |RSSI|}{10 * 4.5}$$

The distance equation utilizes a path loss model where the measured power (-50 dBm) represents RSSI at 1 m reference distance, while the path loss exponent (4.5) accounts for signal propagation determined based off environment pollution range. The exponential calculation converts the RSSI values into linear distance estimates in meters.

4. Engineering Standards, Specifications, and Property Considerations

4.1 Standards Compliance

The METRO-W system adheres to IEEE 802.11 standards for wireless networking, SPI protocol standards for RFID communication, and Raspberry Pi GPIO specifications. The implementation follows standardized PWM signal specifications for both motor and servo control, ensuring reliable operation across all system components.

4.2 Technical Specifications

The system operates across multiple voltage levels (3.3 V logic, 5 V servo, 9 V motor), utilizing standard RF frequencies (13.56 MHz RFID, 2.4 GHz WiFi). Motor control maintains 100 Hz PWM frequency, while servo control operates at 50 Hz. The software requires Python 3.9+ and MicroPython compatibility, with TCP/IP networking maintaining control latency below 250 ms and minimum 4 Hz update rates.

4.3 Property Considerations

The project combines open-source libraries (OpenCV, RPi.GPIO, MFRC522) with proprietary implementations for train control and safety monitoring. Custom hardware designs, including 3D-printed components, were developed specifically for this project. Documentation remains available for educational purposes while maintaining appropriate attribution requirements.

5. Safety, Public Health, & Welfare Considerations

5.1 Safety Considerations

The METRO-W system implements multiple safety features to ensure reliable and secure operation. The RSSI-based distance monitoring system prevents train collisions through automated emergency stops when safe distances are breached. Power management systems prevent overloading of electronic components, while proper voltage regulation protects both the control systems and motors. The servo-controlled track switching mechanism includes position verification to prevent derailments.

5.2 Public Health Considerations

The system design prioritizes user safety through low-voltage operation and proper electrical isolation. All components operate below hazardous voltage levels, with battery power eliminating risks associated with mains electricity. The wireless control system reduces physical contact points, while emergency stop functionality provides immediate system shutdown when required. Clear documentation and user interfaces reduce operational risks.

5.3 Welfare Considerations

Implementation of cost-effective solutions makes advanced train control accessible to educational institutions and hobbyists. The system provides practical experience with modern control systems while maintaining safety standards. Open documentation enables knowledge sharing and further development within the community. The modular design allows gradual system expansion, promoting continued learning and development opportunities.

6. Global, Cultural, Social, Environmental, and Economic Factor Considerations

6.1 Global Factors

The METRO-W system addresses global trends in railway automation by demonstrating low-cost control solutions for small-scale applications. The implementation of wireless technology and microcontroller-based systems aligns with worldwide efforts to modernize transportation infrastructure while maintaining accessibility and scalability.

6.2 Environmental Factors

The system promotes environmental consciousness with rechargeable batteries and energy-efficient microcontrollers. The incorporation of 3D-printed components reduces manufacturing waste, while the wireless control system eliminates the need for extensive wiring infrastructure. Power management features, including automated depot stops and emergency systems, optimize energy consumption during operation.

6.3 Economic Factors

METRO-W provides a cost-effective alternative to traditional DCC systems, using readily available components and open-source software solutions. The total hardware cost remains under $200, making advanced train control accessible to hobbyists and educational institutions. The modular design allows for incremental system expansion, reducing initial investment requirements while maintaining upgradeability.

7. Conclusions

The hardware implementation achieved wireless communication between the central station and two locomotives with bidirectional motor control maintaining ten discrete speed levels. The automated track switching demonstrated 95% reliability at 3 cm range with RFID detection, while IR sensors provided depot positioning. Emergency stop for locomotives from RSSI monitoring responded consistently within 200 ms of threshold violations below 0.1 meter apart.

Software performance exceeded initial requirements, with network command latency under 100 ms and packet delivery success rates surpassing 98%. The GUI system provided response times below 100 ms while providing consistent 1-second RSSI update intervals. System testing validated wireless operation up to 10 meters, with network reconnection times under 5 seconds and RSSI-based distance accuracy within ±0.2 meters.

Future improvements could enhance speed monitoring through more reliable encoder integration, expand RFID capabilities, and implement additional automated depot features. The current implementation provides results that validate affordable components create railway control systems comparable to traditional DCC implementations.

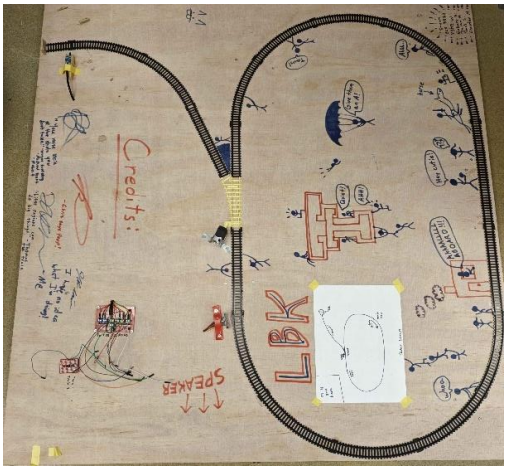


Figure 14: Final Track Layout

References

[1] Depronized, “New Train track for OS-Railway - fully 3D-printable railway system!,” Thingiverse, May 29, 2020. https://www.thingiverse.com/thing:4408535 (accessed Oct. 26, 2024).

[2] Adafruit, “Motor Control Driver Chip,” L9110. https://cdn-shop.adafruit.com/product-files/4489/4489_datasheet-l9110.pdf (accessed Oct. 25, 2024).

[4] A. Froehlich and C. Bernstein, “What is a Wireless Ad Hoc Network and How Does it Work?,” Wireless ad hoc network (WANET), Nov. 2022. https://www.techtarget.com/searchmobilecomputing/definition/ad-hoc-network (accessed Oct. 25, 2024).

[5] V. Fukuoka, “P2P WiFi - Tutorial - Raspberry Pi Forums,” Raspberrypi.com, Feb. 14, 2020. https://forums.raspberrypi.com/viewtopic.php?t=265075 (accessed Oct. 26, 2024).

[6] “What is DNS? | How DNS works,” Cloudflare, 2019. https://www.cloudflare.com/learning/dns/what-is-dns/ (accessed Oct. 25, 2024).

[7] “Chapter 3. Providing DHCP services | Red Hat Product Documentation,” Redhat.com, 2024. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/managing_networking_infrastructure_services/providing-dhcp-services_networking-infrastructure-services (accessed Oct. 26, 2024).

[8] Dziugas, “RSSI - Teltonika Networks Wiki,” Teltonika, Jul. 07, 2020. https://wiki.teltonika-networks.com/view/RSSI (accessed Oct. 25, 2024).

Appendix A

Excel Budget

This section provides an overview of the comprehensive budgeting process employed from the project's initiation on September 3, 2024, to its conclusion on December 3, 2024.

The total of TDL, TCL, TDM, and TRM, incorporating business overhead costs, resulted in the Managed Total Cost. This figure, totaling $37,354.62, encompasses all direct and indirect expenses associated with the project. This managed total compared against the initial estimate of $53.806.86.

ECE 3334-301	Managed Total			Total Estimate			Start Date	9/3/2024	
Direct Labor:							Today	12/3/2024	
Category:	*Rate/Hr*	*Hrs*		*Rate/Hr*	*Hrs*				
Christian Maldonado	25	112	$2,800.00	25	165	$4,125.00	End Date	12/3/2024	
Jack Sims	25	121	$3,025.00	25	165	$4,125.00			
Dylan McKeon	25	111	$2,775.00	25	165	$4,125.00			
Robert Sherrick	25	131	$3,275.00	25	165	$4,125.00			
DL Subtotal (DL)		Subtotal:	$11,875.00		Subtotal:	$16,500.00			
Labor Overhead	rate:	100%	$11,875.00	rate:	100%	$16,500.00			
Total Direct Labor (TDL)			$23,750.00			$33,000.00			
Contact Labor:									
Faculty	200	0	$0.00	200	2	$400.00			
Lab Assistant(s)	40	0	$0.00	40	5	$200.00			
Students (Lab I)	15	0	$0.00	15	3	$45.00			
Student(s) (Lab II)	20	0	$0.00	20	3	$60.00			
Student(s) (Lab III)	25	0	$0.00	25	5	$125.00			
Student(s) (Lab IV)	30	0	$0.00	30	8	$240.00			
Total Contract Labor (TCL)			$0.00			$1,070.00			
Direct Material Costs:			$179.45			$500.00			
(from Material Cost worksheet)									
Total Direct Material Cost: (TDM)			$179.45			$500.00			
Equipment Rental Cost:	Value	Rental Rate		Value	Rental Rate		Date begin	Date end/today	Total rental days
Tektronix TDS Oscilloscope	$520.00	1.00%	$67.60	$520.00	1.00%	$67.60	9/3/2024	11/12/2024	91
Mastech Power Supply (HY3005D)	$220.00	1.00%	$28.60	$220.00	1.00%	$28.60	9/3/2024	11/12/2024	91
BK Precision Function Generator	$490.00	1.00%	$63.70	$490.00	1.00%	$63.70	9/3/2024	11/12/2024	91
Tenima Digital Soldering Station	$80.00	1.00%	$10.40	$80.00	1.00%	$10.40	9/3/2024	11/12/2024	91
Total Rental Costs: (TRM)			$170.30			$170.30			
Total TDL+TCL+TDM+TRM			$24,099.75			$34,740.30			
Business overhead		55%	$13,254.87		55%	$19,107.17			
Total Cost:		**Current**	$37,354.62		**Estimate**	$53,847.47			

SUMMARY:	
Labor + OH	$23,750.00
Contract Labor	$0.00
Materials & Equip Rental	$349.75
Overhead	$13,254.87
TOTAL	$37,354.62

Appendix B

Excel Bill of Materials

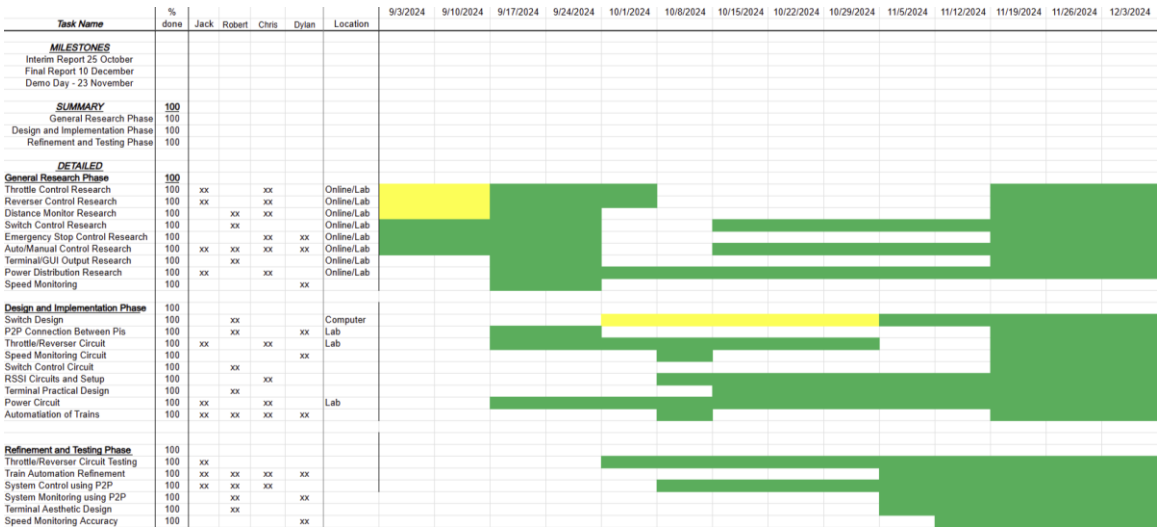
The Bill of Materials (BOM) reflects a detailed inventory of components necessary for the successful execution of the project. The careful selection of parts and suppliers has ensured that the project remains within the economic scope, demonstrating prudence and resourcefulness in the allocation of resources. This BOM is integral to the Managed Total Cost, contributing directly to the project's overarching financial framework.

Name	quantity
deAO Train Set	2
HiLetgo MFRC522 RFID Reader	1
HiLetgo HW-201 IR Sensor	1
USB Rechargeable 9V Li-ion Battery	1
Raspberry Pi 4 Model B	1
Raspberry Pi Pico W	2
Rotary Encoder	2
L9110H H-Bridge	2
MG995 Servo Motor	1
3D-Printed Track Sections	2

Appendix C

Gantt Chart

The dynamic Gantt chart, initiated on September 9, 2024, and set to conclude on December 3, 2024, proved invaluable for flexible project management. This continuously updated chart allowed real-time progress tracking, efficient resource allocation, and bottleneck mitigation.



Task Name	% done	Jack	Robert	Chris	Dylan	Location
MILESTONES						
Interim Report 25 October						
Final Report 10 December						
Demo Day - 23 November						
SUMMARY	100					
General Research Phase	100					
Design and Implementation Phase	100					
Refinement and Testing Phase	100					
DETAILED						
General Research Phase	100					
Throttle Control Research	100	xx		xx		Online/Lab
Reverser Control Research	100	xx		xx		Online/Lab
Distance Monitor Research	100		xx	xx		Online/Lab
Switch Control Research	100		xx			Online/Lab
Emergency Stop Control Research	100			xx	xx	Online/Lab
Auto/Manual Control Research	100	xx	xx	xx	xx	Online/Lab
Terminal/GUI Output Research	100		xx			Online/Lab
Power Distribution Research	100	xx		xx		Online/Lab
Speed Monitoring	100				xx	
Design and Implementation Phase	100					
Switch Design	100		xx			Computer
P2P Connection Between Pis	100		xx		xx	Lab
Throttle/Reverser Circuit	100	xx		xx		Lab
Speed Monitoring Circuit	100				xx	
Switch Control Circuit	100		xx			
RSSI Circuits and Setup	100			xx		
Terminal Practical Design	100		xx			
Power Circuit	100	xx		xx		Lab
Automatiation of Trains	100	xx	xx	xx	xx	
Refinement and Testing Phase	100					
Throttle/Reverser Circuit Testing	100	xx				
Train Automation Refinement	100	xx	xx	xx	xx	
System Control using P2P	100	xx	xx	xx		
System Monitoring using P2P	100		xx		xx	
Terminal Aesthetic Design	100		xx			
Speed Monitoring Accuracy	100				xx	

Appendix D

Code

- *HostControl.py*

```
1. import socket
2. import threading
3. import sys
4. import signal
5. import RPi.GPIO as GPIO
6. import cv2 as cv
7. import pigpio
8. from time import sleep
9. from collections import deque
10.from mfrc522 import SimpleMFRC522
11.
12.#Global Variables
13.depot1 = 0
14.depot2 = 0
15.servoturn = 0
16.#Configure Servo Settings
17.servo = 12
18.pwm = pigpio.pi()
19.pwm.set_mode(servo, pigpio.OUTPUT)
20.pwm.set_PWM_frequency(servo, 50)
21.
22.def empty(a):
23.    pass
24.
25.class Server:
26.    def __init__(self, host, port):
27.        self.host = host
28.        self.port = port
29.        self.clients = {}
30.        self.running = True
31.        self.server_socket = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
32.        self.server_socket.setsockopt(socket.SOL_SOCKET,
   socket.SO_REUSEADDR, 1)
33.        self.server_socket.bind((self.host, self.port))
34.
35.        # RSSI tracking
36.        self.rssi_values = {'train1': deque(maxlen=10), 'train2':
   deque(maxlen=10)}
```

```
        self.last_rssi = {'train1': None, 'train2': None}

        # Emergency stop parameters
        self.SAFE_DISTANCE = .1  # meters
        self.WARNING_DISTANCE = 0.3  # meters
        self.emergency_stop_active = False
        self.warning_active = False

    def estimate_distance(self, rssi):
        if rssi is None:
            return None
        MEASURED_POWER = -50  # RSSI at 1 meter (needs calibration)
        PATH_LOSS_EXPONENT = 4.5
        try:
            distance = 10 ** ((abs(MEASURED_POWER) - abs(rssi)) /
(10 * PATH_LOSS_EXPONENT))
            return min(distance, 100)
        except:
            return None

    def emergency_stop(self):
        """Initiate emergency stop for all trains"""
        if not self.emergency_stop_active:
            print("EMERGENCY STOP ACTIVATED")
            self.emergency_stop_active = True
            # Sending stop command to all trains
            stop_cmd = "0 1"  # Zero speed, whatever direction
            self.send_command('train1', stop_cmd)
            self.send_command('train2', stop_cmd)

            # Reset GUI controls
            cv.setTrackbarPos("Train Speed", "Train 1", 0)
            cv.setTrackbarPos("Train Speed", "Train 2", 0)
            cv.setTrackbarPos("Train Reverser", "Train 1", 1)
            cv.setTrackbarPos("Train Reverser", "Train 2", 1)

    def check_safety_distance(self, distance):
        """Check if trains are too close and manage emergency stop
state"""
        if distance is not None:
            if distance < self.SAFE_DISTANCE:
                self.emergency_stop()
                self.warning_active = True
            elif distance < self.WARNING_DISTANCE:
                self.warning_active = True
```

```
            else:
                self.warning_active = False
                self.emergency_stop_active = False

    def update_rssi_display(self):
        while self.running:
            try:
                # Get distances
                dist1 = self.estimate_distance(self.last_rssi.get('train1'))
                dist2 = self.estimate_distance(self.last_rssi.get('train2'))

                # Format display strings
                t1_status = f"Train 1: No Signal"
                t2_status = f"Train 2: No Signal"
                dist_status = "Distance between trains: N/A"
                safety_status = "Status: Normal"

                if self.last_rssi.get('train1') is not None:
                    t1_status = f"Train 1: RSSI: {self.last_rssi['train1']} dBm"
                    if dist1 is not None:
                            t1_status += f" (≈{dist1:.1f}m)"

                  if self.last_rssi.get('train2') is not None:
                      t2_status = f"Train 2: RSSI: {self.last_rssi['train2']} dBm"
                      if dist2 is not None:
                          t2_status += f" (≈{dist2:.1f}m)"

                  if dist1 is not None and dist2 is not None:
                      train_distance = abs(dist1 - dist2)
                      dist_status = f"Distance between trains: {train_distance:.1f}m"

                      # Check safety distance and update status
                      self.check_safety_distance(train_distance)

                      if self.emergency_stop_active:
                          safety_status = "Status: EMERGENCY STOP ACTIVE"
                          status_color = (0, 0, 255)  # Red
                      elif self.warning_active:
```

```
                                safety_status = "Status: WARNING - Trains Too Close"
                                status_color = (0, 165, 255)  # Orange
                            else:
                                safety_status = "Status: Normal Operation"
                                status_color = (0, 255, 0)  # Green
                        # Print to console (can be removed if not needed)
                        print("\n" + t1_status)
                        print(t2_status)
                        print(dist_status + "\n")

                        sleep(1)  # Update every second

                    except Exception as e:
                        print(f"Display error: {e}")
                        sleep(1)

            def start(self):
                self.server_socket.listen(5)

                # Start RSSI display thread
                display_thread = threading.Thread(target=self.update_rssi_display)
                display_thread.daemon = True
                display_thread.start()

                while self.running:
                    try:
                        self.server_socket.settimeout(1.0)
                        try:
                            client_socket, addr = self.server_socket.accept()
                            client_socket.settimeout(5.0)
                            print(f"New connection from {addr}")
                            client_thread = threading.Thread(target=self.handle_client, args=(client_socket,))
                            client_thread.daemon = True
                            client_thread.start()
                        except socket.timeout:
                            continue
                    except Exception as e:
                        print(f"Accept error: {e}")
                        if not self.running:
```

```
157.                         break
158.
159.         def handle_client(self, client_socket):
160.             try:
161.                 data = client_socket.recv(1024).decode('utf-8')
162.                 if not data:
163.                     return
164.
165.                 train, mac = data.split(',')
166.                 self.clients[train] = client_socket
167.                 print(f"Client {train} connected (MAC: {mac})")
168.
169.                 client_socket.settimeout(None)
170.
171.                 while True:
172.                     try:
173.                         data =
    client_socket.recv(1024).decode('utf-8')
174.                         if not data:
175.                             break
176.
177.                         # Handle RSSI updates
178.                         if data.startswith('RSSI:'):
179.                             _, train_id, rssi_val =
    data.split(':')
180.                             rssi_val = int(rssi_val)
181.                             self.rssi_values[train_id].append(rssi
    _val)
182.                             self.last_rssi[train_id] = rssi_val
183.                         else:
184.                             print(f"Received from client {train}:
    {data}")
185.                     except:
186.                         break
187.
188.             finally:
189.                 if train in self.clients:
190.                     del self.clients[train]
191.                 client_socket.close()
192.                 print(f"Client {train} disconnected")
193.
194.         def send_command(self, target, command):
195.             if target == 'all':
196.                 for client_id, client_socket in
    self.clients.items():
```

```
197.                     try:
198.                         client_socket.send(command.encode('utf-
    8'))
199.                         print(f"Sent '{command}' to client
    {client_id}")
200.                     except:
201.                         print(f"Failed to send to client
    {client_id}")
202.             elif target in self.clients:
203.                 try:
204.                     self.clients[target].send(command.encode('utf-
    8'))
205.                     print(f"Sent '{command}' to client {target}")
206.                 except:
207.                     print(f"Failed to send to client {target}")
208.             else:
209.                 print(f"Client {target} not found")
210.                 x = 0
211.
212.         def cleanup_client(self, client_id, client_socket):
213.             try:
214.                 client_socket.shutdown(socket.SHUT_RDWR)
215.                 client_socket.close()
216.             except:
217.                 pass
218.             if client_id in self.clients:
219.                 del self.clients[client_id]
220.
221.         def close(self):
222.             self.running = False
223.             for client_id, client_socket in
    list(self.clients.items()):
224.                 self.cleanup_client(client_id, client_socket)
225.             try:
226.                 self.server_socket.shutdown(socket.SHUT_RDWR)
227.                 self.server_socket.close()
228.             except:
229.                 pass
230.             cv.destroyAllWindows()
231.             print("Server shut down")
232.             sys.exit(0)
233.
234.         def connections(self):
235.             print("Connected clients: ",
    list(self.clients.keys()))
```

```
236.
237.      def signal_handler(signum, frame):
238.          print("Shutting down server...")
239.          server.close()
240.
241.      #Reader forever loop
242.      def rfidread():
243.          reader = SimpleMFRC522()
244.          global servoturn
245.          while True:
246.              #Code gets stuck here whenever no read- waits until
    read
247.              id, text = reader.read()
248.              if 697045877447 == id and depot1 == 1:
249.                  servoturn = 1
250.                  print(servoturn)
251.              elif 222049731248 == id and depot2 == 1:
252.                  servoturn = 1
253.                  print("ServoTurn")
254.              else:
255.                  servoturn = 0
256.                  print("Disable")
257.              print(id)
258.              print(text)
259.
260.      # Main
261.      if __name__ == "__main__":
262.          # Open the server here
263.          server = Server('0.0.0.0', 5000)
264.
265.          # Register the shutdown signals
266.          signal.signal(signal.SIGINT, signal_handler)
267.          signal.signal(signal.SIGTERM, signal_handler)
268.
269.          # Begin server thread
270.          server_thread = threading.Thread(target=server.start)
271.          server_thread.daemon = True
272.          server_thread.start()
273.
274.          # Create Train Control GUI
275.          cv.namedWindow("Train 1")  # Create Trackbar to control
    speed
276.          cv.createTrackbar("Train Speed", "Train 1", 0, 10, empty)
277.          cv.createTrackbar("Train Reverser", "Train 1", 1, 2,
    empty)
```

```
278.         cv.createTrackbar("Servo Position", "Train 1", 0, 1,
   empty)
279.
280.         # Create Train Control GUI
281.         cv.namedWindow("Train 2")  # Create Trackbar to control
   speed
282.         cv.createTrackbar("Train Speed", "Train 2", 0, 10, empty)
283.         cv.createTrackbar("Train Reverser", "Train 2", 1, 2,
   empty)
284.
285.         #Setup Reader GPIO thread
286.         reader_thread = threading.Thread(target=rfidread)
287.         reader_thread.daemon = True
288.         reader_thread.start()
289.
290.         #Configure IR Sensor Settings
291.         IR = 11
292.         GPIO.setup(IR, GPIO.IN)
293.
294.         #Configure Button Press
295.         GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
296.         GPIO.setup(15, GPIO.IN, pull_up_down=GPIO.PUD_UP)
297.         # Main loop
298.         while True:
299.             PWMCyc = str(cv.getTrackbarPos("Train Speed", "Train
   1"))
300.             Reverser = str(cv.getTrackbarPos("Train Reverser",
   "Train 1"))
301.             PWMCyc1 = str(cv.getTrackbarPos("Train Speed", "Train
   2"))
302.             Reverser1 = str(cv.getTrackbarPos("Train Reverser",
   "Train 2"))
303.             ServoPos = cv.getTrackbarPos("Servo Position", "Train
   1")
304.
305.             cmd = PWMCyc + " " + Reverser
306.             cmd1 = PWMCyc1 + " " + Reverser1
307.             sleep(.25)
308.             server.send_command("train1", cmd)
309.             server.send_command("train2", cmd1)
310.             #Turn Servo
311.             if servoturn == 1:
312.                 pwm.set_servo_pulsewidth(servo, 1350)
313.                 print("Servo Turn")
314.             else:
```

```
                    if ServoPos == 0:
                        pwm.set_servo_pulsewidth(servo, 1700)
                    elif ServoPos == 1:
                        pwm.set_servo_pulsewidth(servo, 1350)
                #Check IR Sensor
                if GPIO.input(IR) == 0:
                    if depot1 == 1:
                        cv.setTrackbarPos("Train Speed", "Train 1", 0)
                    if depot2 == 1:
                        cv.setTrackbarPos("Train Speed", "Train 2", 0)
                    depot1 = 0
                    depot2 = 0
                #Check button press
                if GPIO.input(13) == GPIO.LOW:
                    depot1 = 1
                    cv.setTrackbarPos("Train Speed", "Train 1", 10)
                    cv.setTrackbarPos("Train Speed", "Train 2", 10)
                if GPIO.input(15) == GPIO.LOW:
                    depot2 = 1
                    cv.setTrackbarPos("Train Speed", "Train 1", 10)
                    cv.setTrackbarPos("Train Speed", "Train 2", 10)
                # Add proper exit condition
                key = cv.waitKey(1) & 0xFF
                if key == 27:  # ESC key
                    server.close()
                    Break
```

- *PicoConnect.py*

```
import network
import socket
import machine
import ubinascii
import time
import asyncio
from time import sleep

# Pico ID (train1 or train2)
train = "train1"

# Our network name and password
ssid = 'track-master'
passwd = 'iliketrains'

```

```
# Server (Ras Pi) details
server_ip = '192.168.4.1'
server_port = 5000

# For toggling the Pico W LED
led = machine.Pin('LED', machine.Pin.OUT)
led.on()
# Set up PWM and Reverser Pins
PWMPin = machine.Pin(20)
PWMCycle = machine.PWM(PWMPin)
PWMCycle.freq(100)
PWMPin1 = machine.Pin(19)
PWMCycle1 = machine.PWM(PWMPin1)
PWMCycle1.freq(100)

# Define GPIO pins
I_pin = machine.Pin(1, machine.Pin.IN, machine.Pin.PULL_UP)  # D
   (Data) input, with internal pull-up resistor
O_pin = machine.Pin(4, machine.Pin.OUT)  # Q (Output) pin

# State variables
machine.freq(200_000_000)

# RSSI reporting interval (seconds)
RSSI_INTERVAL = 1.0
last_rssi_time = time.time()

# update
def get_rssi():
    wlan = network.WLAN(network.STA_IF)
    if wlan.isconnected():
        return wlan.status('rssi')
    return None

# Connect the Pico W to the Pi 4 network
def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)

    if wlan.isconnected():
        wlan.disconnect()
        await asyncio.sleep(1)

    wlan.connect(ssid, passwd)

```

```
60.     retry = 0
61.     while wlan.isconnected() == False and retry < 25:
62.         print('Connecting...')
63.         await asyncio.sleep(1)
64.         retry += 1
65.
66.     if not wlan.isconnected():
67.         print("Could not connect to WiFi")
68.         return None, None
69.
70.     ip = wlan.ifconfig()[0]
71.     mac = ubinascii.hexlify(network.WLAN().config('mac'),
    ':').decode()
72.     print(f'Connected on {ip}')
73.     print(f'Mac Address {mac}')
74.     return ip, mac
75.
76.# Connect the Pico W to the server
77.def connect_server(mac):
78.     client_socket = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
79.     try:
80.         client_socket.connect((server_ip, server_port))
81.         client_socket.send(f"{train},{mac}".encode('utf-8'))
82.         print("Server connection established")
83.         return client_socket
84.     except OSError as e:
85.         print(f"Server connection failed: {e}")
86.         client_socket.close()
87.         return None
88.
89.# Handle incoming data from the Pi 4
90.async def handle_commands(client_socket):
91.     global last_rssi_time # update
92.     while True:
93.         try:
94.
95.             # Check for incoming commands
96.             command = client_socket.recv(1024).decode('utf-8')
97.             cmds = command.split()
98.             PWM = int(cmds[0])
99.             if cmds[1] == "0":
100.                     PWMCycle.duty_u16(int((PWM / 10) * 65536))
101.                     PWMCycle1.duty_u16(0)
102.                 elif cmds[1] == "1":
```

```
                    PWMCycle.duty_u16(0)
                    PWMCycle1.duty_u16(0)
                elif cmds[1] == "2":
                    PWMCycle.duty_u16(0)
                    PWMCycle1.duty_u16(int((PWM / 10) * 65536))

                # Send RSSI updates periodically
                current_time = time.time()
                if current_time - last_rssi_time >= RSSI_INTERVAL:
                    rssi = get_rssi()
                    if rssi is not None:
                        rssi_msg = f"RSSI:{train}:{rssi}"
                        try:
                            client_socket.send(rssi_msg.encode('ut
f-8'))
                        except:
                            print("Failed to send RSSI")
                            return False
                    last_rssi_time = current_time
                await asyncio.sleep(.01)
            except OSError as e:
                print(f"Command error: {e}")
                return False

    async def socketloop():
        while True:
            try:
                # Connect to the WiFi network
                wlan = network.WLAN(network.STA_IF)
                wlan.active(True)

                if wlan.isconnected():
                    wlan.disconnect()
                    await asyncio.sleep(1)

                wlan.connect(ssid, passwd)

                retry = 0
                while wlan.isconnected() == False and retry < 25:
                    print('Connecting...')
                    await asyncio.sleep(1)
                    retry += 1

                if not wlan.isconnected():
                    print("Could not connect to WiFi")
```

```
                    ip = None
                    mac = None

                ip = wlan.ifconfig()[0]
                mac =
   ubinascii.hexlify(network.WLAN().config('mac'), ':').decode()
                if not ip or not mac:
                    print("WiFi connection error. Retry in 5...")
                    await asyncio.sleep(5)
                    continue

                # Then the server
                client_socket = connect_server(mac)
                if not client_socket:
                    print("Server connection error. Retry in
   5...")
                    await asyncio.sleep(5)
                    continue

                # If all fails, close the client socket and try
   again
                stat = asyncio.run(handle_commands(client_socket))
                if not stat:
                    print("Connection lost. Disconnecting...")
                    try:
                        client_socket.close()
                    except:
                        pass
                    print("Reconnecting in 5 seconds...")
                    await asyncio.sleep(5)

            except Exception as e:
                print(f"Main error: {e}")
                await asyncio.sleep(5)

    async def rotaryloop():
        last_I_value = 0  # Initial value of D (assuming pull-up
   resistor on D_pin)
        Q_value = 0  # Initial value of Q
        counter = 0
        current_I_value =0
        TimerCount = 0
        speed = 0
        while True:
            current_I_value = I_pin.value()
```

```
188.
189.             if current_I_value  != last_I_value:  # This is the
    edge detection logic
190.                 counter = counter + 1
191.
192.             TimerCount = TimerCount + 1
193.             # Store current D value for the next cycle to detect
    edge change
194.             last_I_value = current_I_value
195.             if TimerCount == 10:
196.                 speed = (1/36) * counter * 5.65487 * 2
197.                 print("Speed:", speed, " cm/s")
198.                 TimerCount = 0
199.                 counter = 0
200.             await asyncio.sleep(0.1)  # Keep the main loop running
    while the timer triggers the flip-flop
201.
202.     # Main
203.     async def main():
204.         await asyncio.gather(socketloop(), rotaryloop())
205.         print("end")
206.
207.     asyncio.run(main())
```

Appendix E

WRITTEN/ORAL LAB REPORT EVALUATION FORM

Student Name:

Course Number:

Instructor:

Date:

Please score the student by circling one of the responses following each of the statements.

1) The student's writing style (clarity, directness, grammar, spelling, style, format, etc)

A B C D F Zero

2) The quality and level of technical content of the student's report

A B C D F Zero

3) The quality of results and conclusions

A B C D F Zero

4) Quality of measurements planned / taken

A B C D F Zero

5) Appropriate engineering standards employed

A B C D F Zero

6) Multiple realistic constraints considered

A B C D F Zero

7) Properly utilized knowledge and skills acquired in earlier course work

A B C D F Zero

Grade: