Application Report

Designing a Motorized Resistive Load

Abdallah Obidat

ABSTRACT

This paper describes a method for developing a high-power motorized resistive load, which can be used for a number of different testing applications. Load simulators are helpful in sweeping the load applied to a system or device under test (DUT) and using a dynamic resistive load enables this testing without having to repeatedly change out the power resistors. The relatively small size also makes the load easier to shield for applications sensitive to radiated emissions. Another application for this load is a passive alternative to an active electronic load. Electronic loads are active current sinks that are not ideal for some low-voltage applications or where the voltage under test is changing. By utilizing a stepper motor that employs feedback, different resistor combinations may be dynamically set between two terminals in a small and easy-to-shield footprint. The described motorized resistive load allows for easy evaluation of a device or system with a passive load, and eliminates the need to customize high-power loads for different voltage rails. This resistive load provides a robust solution to the practical problem of loading a system or device with a range of passive loads.

Download the project collateral discussed in this application report from the following URL: www.ti.com/lit/zip/SNVAA20.

Table of Contents

List of Figures

Trademarks

1 Introduction

A motorized resistive load is a programmable load that can be used to help test or characterize a system or DUT. This type of load is useful in evaluating DUT behavior under steady-state operation or during start-up, shutdown, or other dynamic conditions. To use this load, the user connects the DUT to the load terminals and sets a target resistance. After confirming the selection, the motorized resistive load forms the target resistance between its input terminals. At this point, the DUT may be enabled, tested, and its behavior under load can be observed. The motorized resistive load design described herein offers advantages in utility, size, and ease-of-use compared to alternative solutions.

Electronic loads are typically utilized when static or steady-state performance of the system is of interest, and these loads can exhibit undesirable behavior when used during transient or dynamic operation. Electronic loads utilize active circuitry to implement feedback that could potentially interact with the DUT and as a result exhibit load profiles that are different than the profile generated by a resistive load. For example, during the initial start-up of a buck converter, its output voltage will increase linearly and if the applied load is resistive, then the load will also increase linearly until the output voltage reaches its nominal value. Applying a nonresistive load via an electronic load could result in an abnormal load profile or oscillatory behavior under the same conditions, even if the load is set for constant-resistance mode. Given these unpredictable load profiles and behaviors, system or DUT behavior can be difficult to characterize, making unexpected DUT behavior difficult to debug unless a resistive load is employed.

Electronic loads typically have minimum voltage requirements to ensure expected operation, whereas the described motorized resistive load does not have this limitation. Depending on the electronic load used, operation is not guaranteed under a set voltage threshold. This disqualifies electronic loads from being utilized for low-voltage applications or when the input voltage to the system is in dropout (when the input voltage is near the set output voltage). Conversely, a resistive load can be used for extremely low voltages and reliably produce accurate results. Figure 1-1 shows the load profile when using the motorized resistive load compared to a standard electronic load when loading a buck converter during its start-up.

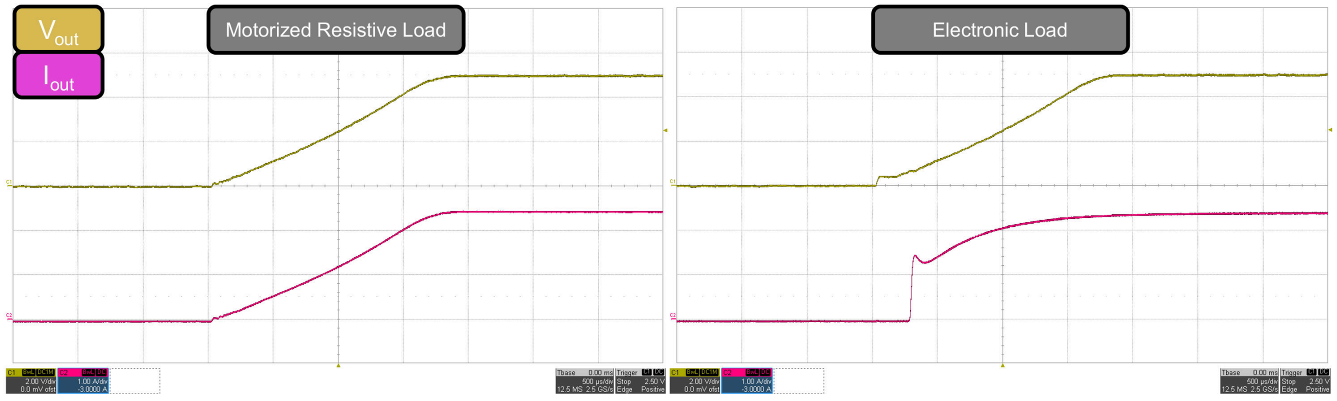


Figure 1-1. Load Profile Comparison

A motorized resistive load is a more compact and easy-to-use alternative to an electronic load. Commercial electronic loads can be relatively large, while a motorized resistive load can be manufactured with roughly the same load range at a fraction of the size. The design described in this paper is circular and has a diameter of approximately 10 in, and is approximately 3 in tall, whereas electronic loads are typically rectangular and can be nearly twice as large. The only setting required for the motorized resistive load is the target resistance, making it an intuitive solution. Additionally, the programmability of the motorized resistive load can help reduce test time and is more convenient than manually changing individual load resistors.

2 Motorized Resistive Load Architecture

This motorized resistive load design is comprised of two PCBs and a Raspberry Pi™. The Raspberry Pi serves as the system controller, which facilitates communication between the components. The Raspberry Pi connects directly to the first PCB, the controller board, which contains the majority of the system blocks including: the protection, power management, motor drive, and analog-to-digital components. The Raspberry Pi and the controller board together form the controller module, which allows users to interface with the apparatus. The second PCB contains the actual load in the form of a resistor track. This circular PCB includes a stepper motor that is installed at its center along with a simple mechanical arm assembly that forms the variable resistance value between the input terminals of the apparatus. The user powers up the controller module and turns a knob (potentiometer) which changes the target resistance which is displayed on an LCD screen located on the controller board. The user then presses a button which triggers motor movement until the measured resistance value, which is also displayed on the LCD screen, matches the target resistance value within a predefined threshold. The number of steps and direction taken by the stepper motor is determined via feedback that is described in Section 3.4.

This programmable resistor is comprised of a number of interdependent blocks that contribute to the functionality of the overall design. Protection against reverse polarity is applied to the input adapter which is then split into multiple power rails that interact with system control elements and peripheral components. These blocks all perform critical functions that enable selecting a target resistance, displaying the selection, and rotating the motor appropriately to form the target resistance between the apparatus terminals. Figure 2-1 shows the system block diagram.

- LM74610-Q1: Smart-Diode Controller for Reverse Polarity Protection
- LMR33630: Synchronous Buck Converter with Ultra-Low EMI
- DRV8833: Dual H-Bridge Motor Driver
- LP2980-ADJ: Micropower Ultra Low-Dropout Adjustable Voltage Regulator
- ADC121C021: I2C-Compatible, 12-Bit Analog-to-Digital Converter with Alert Pin

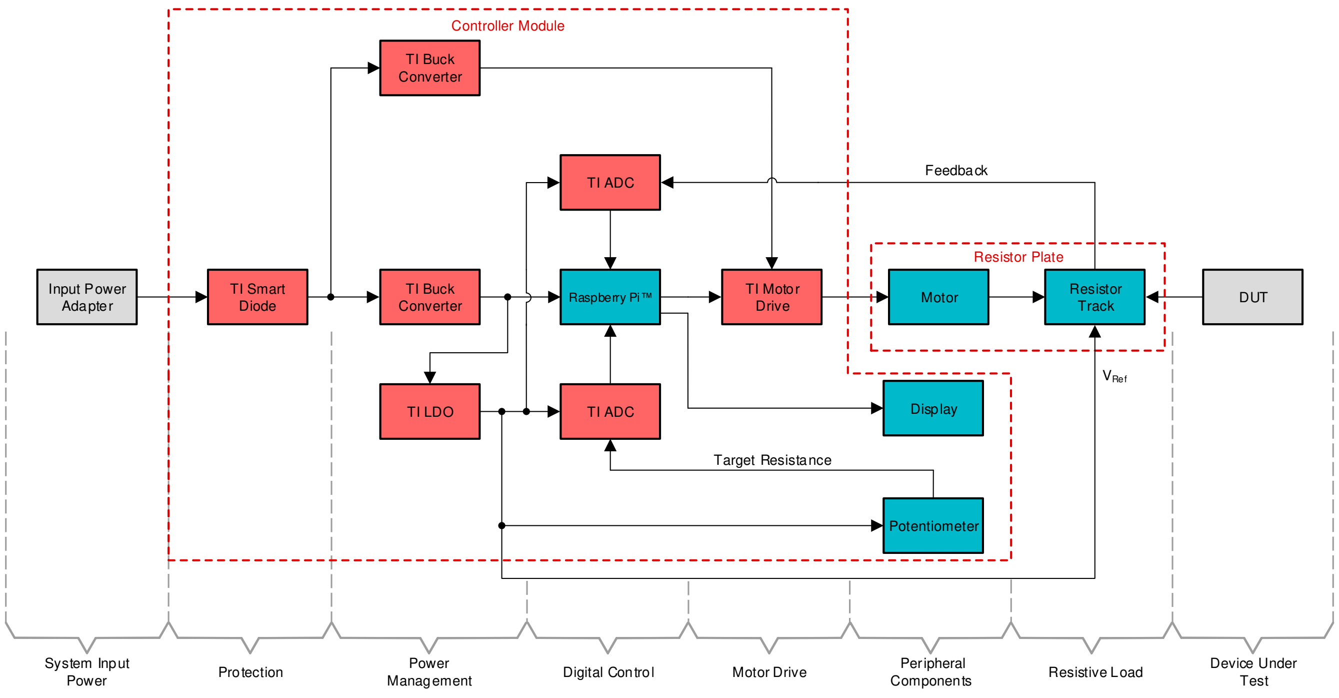


Figure 2-1. System Block Diagram

2.1 Controller Board

The controller PCB is a 4-layer board that houses the user interface, handles the power management and sends the motor drive signals to the stepper motor located on the resistive load PCB. The Raspberry Pi 4 model B was used for this design and it mates with the controller board via the general-purpose input output (GPIO) pin header via a connector that is used as a spacer. The 4079 from Adafruit Industries is a suitable female-to-male

2 × 20 position connector with a 0.100 in pin pitch. Each pin on this connector is rated for 3 A which meets the requirements of this design. The Raspberry Pi is typically powered from a designated USB Type-C® port. The controller PCB eliminates the need to use the USB Type-C port by back-powering the Raspberry Pi via the power pins (pins 2 and 4) on the GPIO header of the Raspberry PI. Power comes in through a barrel connector and passes through the reverse-polarity protection provided by LM74610-Q1. Power then continues to feed into two buck converters, a pair of LMR33630 devices. The first buck converter provides power to the motor driver DRV8833. The second buck converter powers the Raspberry Pi with 5 V and a low-dropout regulator (LDO), the LP2980-ADJ which then provides a quiet 3.3-V power rail for the two ADC121C021 analog-to-digital converters (ADCs). The LDO also provides a reference voltage for the potentiometer that is used to determine the target resistance selected by the user, in addition to providing power to the LCD screen. The backlight of the LCD screen, however, is powered by a 3.3-V rail provided by the Raspberry Pi.

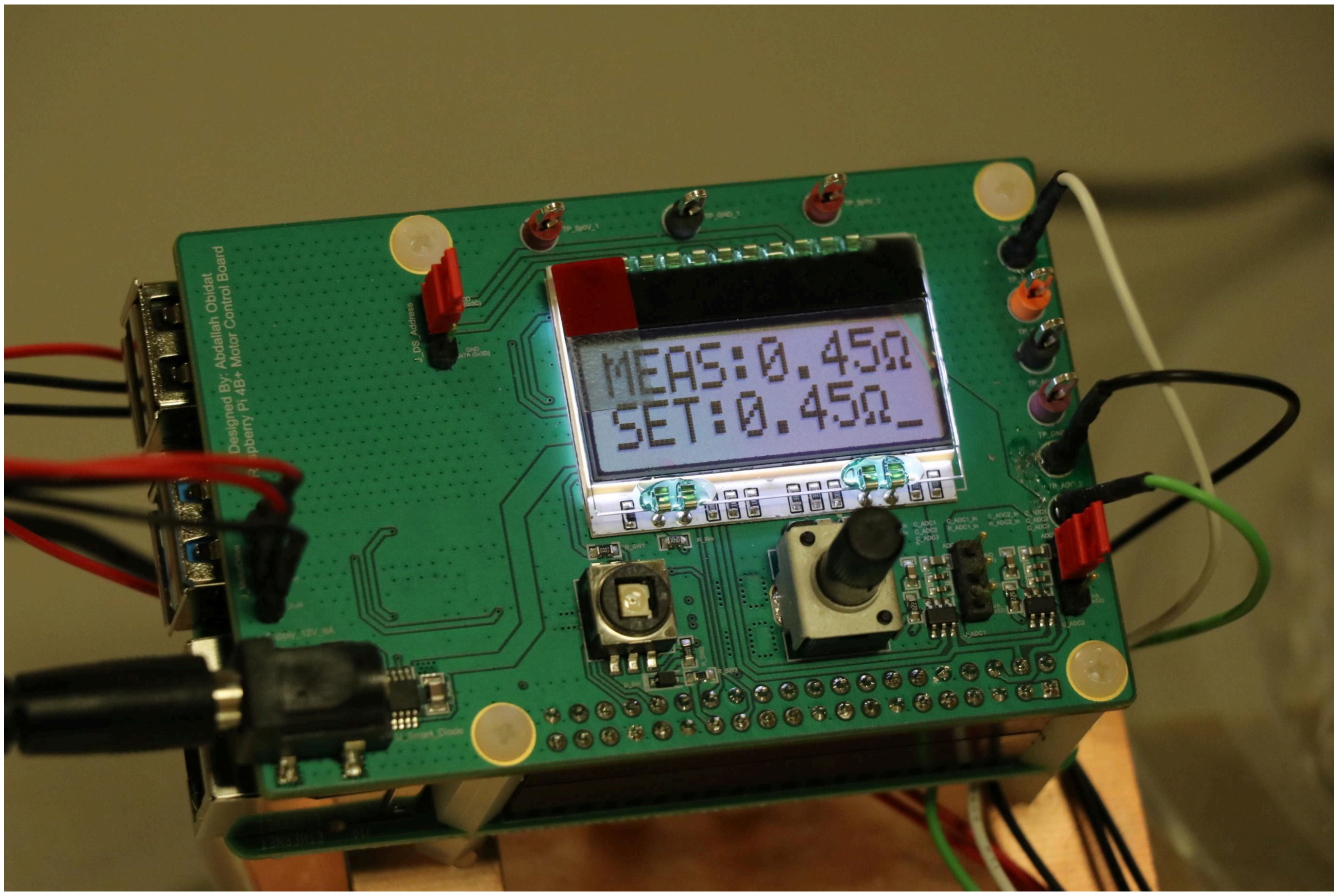

MEAS:0.45Ω
SET:0.45Ω_
Designed By: Abdallah Obidat
Raspberry Pi 4B+ Motor Control Board

Figure 2-2. Controller Board

2.2 Resistor Plate

The resistor plate is a circular 2-layer PCB with a stepper motor at its center, resistors around its circumference at equal angular offsets, and a copper mechanical arm assembly. The shape of the board enables the stepper motor to select different series resistor combinations. The resistors form an open circular track around the PCB and the positive input terminal is connected just before the smallest value resistor on the resistor track. The smallest and largest value resistors are not directly connected to each other, breaking the resistor track. The negative input terminal connects to GND, which is connected to the mechanical arm assembly. Therefore, the current path flows between the positive terminal, through the resistors between the positive terminal and the mechanical arm, and finally through the mechanical arm into the GND layer of the board and back into the negative terminal. The resistor plate also contains an input port for a reference voltage, a low current fuse for this reference, and a small resistor. The reference voltage, small resistor, and the portion of resistor track which are not used in generating the load, are used in implementing feedback control over the stepper motor.

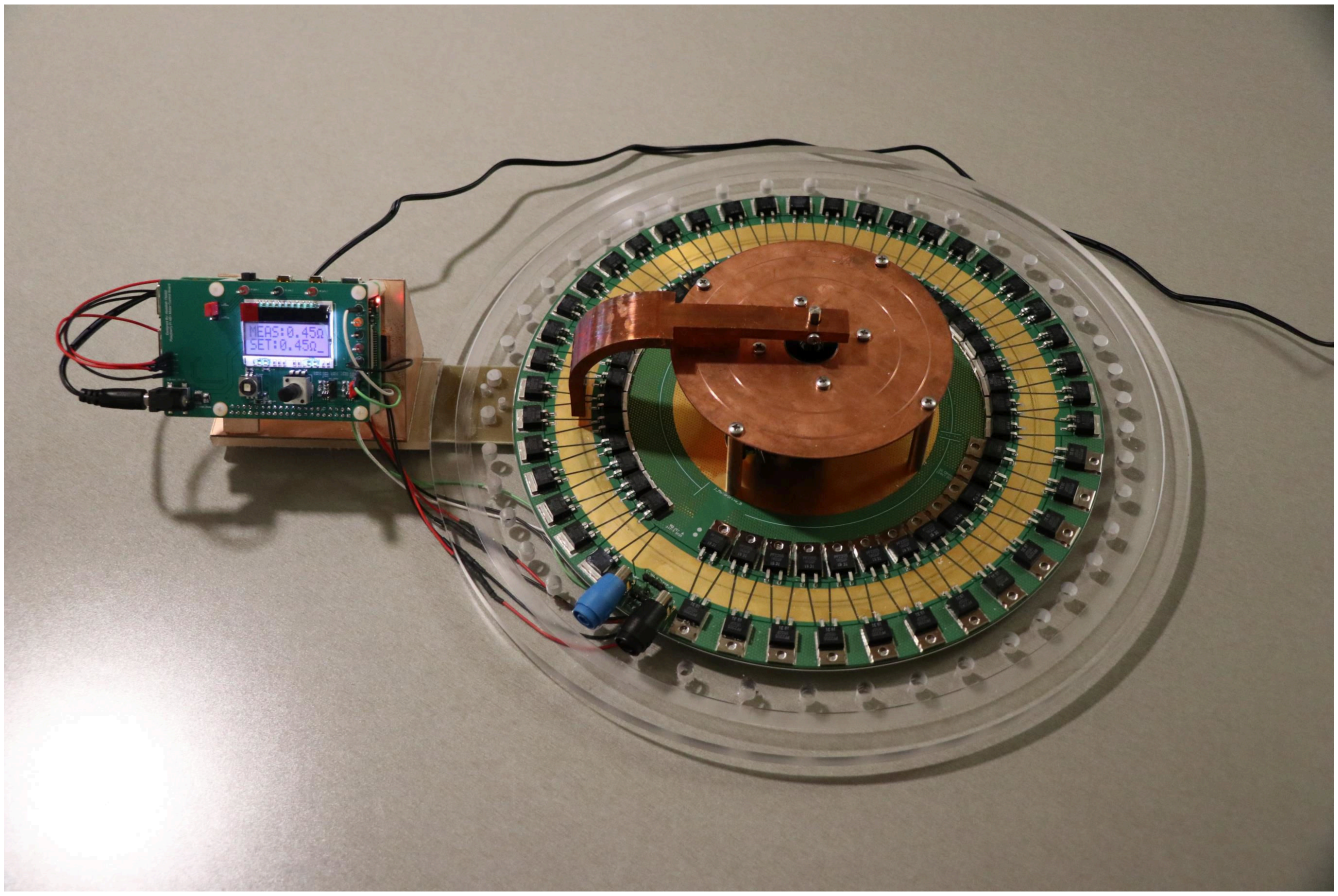


Figure 2-3. Motorized Resistive Load (Protective Cover Removed)

3 Motorized Resistive Load Design

Manufacturing the described motorized resistive load requires designing the controller board and resistor plate, both of which can be broken down into smaller subsystems to simplify the design process. The controller board needs to manage and deliver power to all of the subsystems and generate the motor drive signals. The controller board also requires a user interface that includes a button, knob, and screen at a minimum to allow basic user control, in addition to an ADC that connects to the resistor plate to implement feedback to exact control on the resistive load. The resistor plate contains different ranges of load resistors, although a single range may be used to simplify the design. The resistor plate also contains the motor, mechanical arm assembly, and the feedback divider that will connect to a controller board ADC.

3.1 Controller Board Design

3.1.1 Power Management

Proper power management entails that the total power must be calculated, and divided appropriately between the different critical components. The two elements that require the most power are the Raspberry Pi and the motor; therefore, their power consumption is calculated first. The Raspberry Pi 4 model B is typically powered by a USB Type-C adapter, and so requires 5 V and a maximum current of 3 A. It should be noted that minimal processing power is required and so the Raspberry Pi consumes less than 1 A for this design. The motor requires a motor power supply voltage (VM) of 5 V and 1 A per phase. The total power consumption of the main system components is:

$$\begin{aligned} Power &= P = Voltage \times Current \\ P_{System} &= P_{Controller} + P_{Motor} \\ P_{System} &= \left(V_{USB\_C} \times I_{Processor}\right) + \left(V_M \times \left(I_{Phase1} + I_{Phase2}\right)\right) \\ P_{System} &= \left(5 \times 1\right) + \left(5 \times \left(1 + 1\right)\right) = 15\ W \end{aligned} \tag{1}$$

To ensure that this power is provided to the critical components, a standard 12-V, 2-A adapter is used to power the system. Such an adapter ensures that sufficient power will be available to fully power the system:

$$P_{Adapter} = 12 \times 2 = 24\,W \quad (2)$$

The remaining components require minimal power, but their power consumption can be calculated for completeness. The LCD requires an input of 3.3 V but only consumes 440 µA which equates to:

$$P_{LCD} = 3.3 \times 440 \times 10^{-6} \approx 1.5\,mW \quad (3)$$

The LCD backlight requires more power as it takes in an input voltage of 3.3 V and consumes 30 mA which also passes through a 10-Ω resistor resulting in:

$$P_{Backlight} = \left(3.3 \times 0.030\right) + \left(0.030^{2}\right) \times \left(10\right) \approx 110\,mW \quad (4)$$

The data sheet for the ADCs indicates that each requires 0.26 mW with a 3-V rail, totaling 0.52 mW. To conserve power, the motor driver may be disabled when not in use. This also prevents the motor from locking into place when not in use. If the motor driver is disabled, the weight of the mechanical arm assembly is enough to keep the arm in place when the motor is not in motion, but the arm may move if acted upon by an external force.

3.1.2 Power Converter Selection

After determining the power requirements of this apparatus, and the required voltage and current levels for this design, suitable power converters must be selected. A standard 12-V power adapter is the starting point and this voltage is split into two 5-V rails. The first rail powers the motor via the motor driver and the second rail powers the Raspberry Pi and the rest of the components. To split the main power rail into two rails, two LMR33630 devices are utilized. These buck converters offer an easy-to-use pinout with a convenient leaded package in a small solution size. The second rail also feeds into an LDO, the LP2980-ADJ, which provides a quiet rail for the ADCs. Selecting a 5-V buck converter to power the LDO ensures that there is some headroom for the LDO, which has a 3.3-V output. The Raspberry Pi also provides a 3.3-V rail that can be used for the LED backlight, bringing the total number of voltage rails on the controller board to two 5-V rails, a 3.3-V rail, and a quiet, low-power 3.3-V rail as shown in Figure 3-1. It should be noted that an external Schottky diode is placed between the output (anode) and input (cathode) LDO to limit the reverse voltage across the LDO, in the event of the input voltage falling below the output voltage, which could cause an internal diode to latch on. This external leakage current of the Schottky diode must be minimal to ensure that the control loop can maintain control over the LDO output voltage.

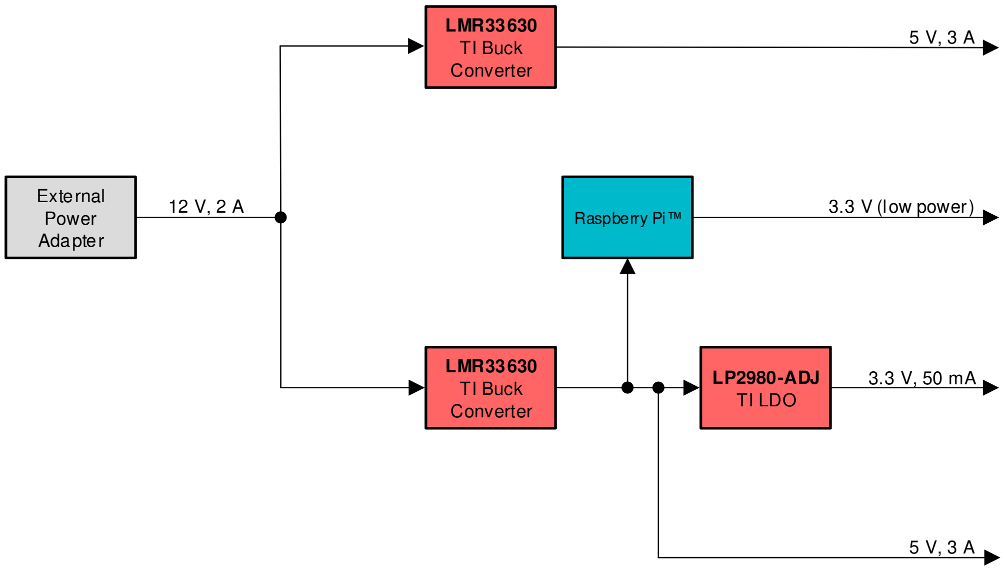


Figure 3-1. Controller Module Power Rails

3.1.3 Interface and ADC Selection

To interface with the motorized resistive load, the user must turn a knob that controls the target resistance, and press a tactile button to confirm the target resistance and initiate the command. The knob is a potentiometer with the center tap connected to an ADC, and the remaining two taps are connected to the quiet 3.3-V rail, and GND respectively. As the knob is turned, the center tap voltage changes, and its value can be used as a reference to set the target resistance. To signal the Raspberry Pi that the button has been pressed, a GPIO pin is configured to be an active-low input pin and an internal pullup resistor is connected to the pin. This causes that pin to be pulled up to 3 V by default and when this pin is pulled down to 0 V, the Raspberry Pi is signaled to take action. The circuit for the interface is shown in Figure 3-2.

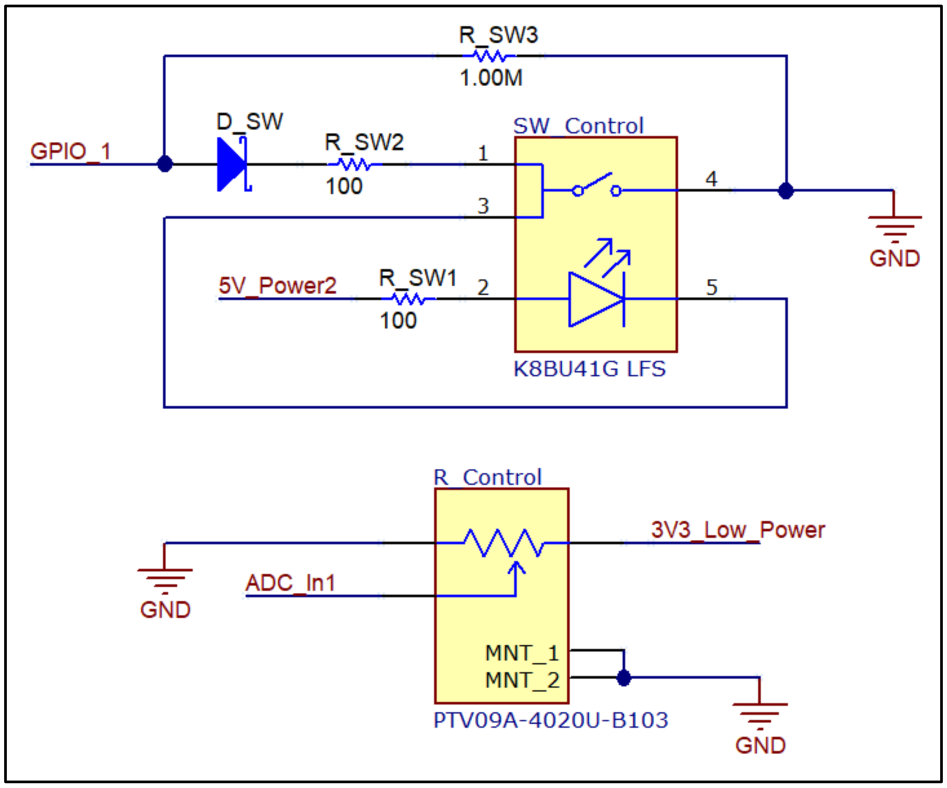


Figure 3-2. User Input Interface Circuit

The selected button for this design includes an embedded blue LED. Blue LEDs have a relatively high forward voltage drop and so the 5-V rail that powers the Raspberry Pi also biases the anode of this LED. The cathode connects to one side of the switch, that also connects to a GPIO pin. To protect the 3-V pin from the 5-V source, a diode and current-limiting resistor are placed in series with the GPIO pin and the switch node connected to the LED. The other end of the switch is connected to GND. When the button is pressed, the GPIO pin is pulled to 0 V through the switch. During this time current also flows from the 5-V rail through the LED, lighting the LED up and indicating to the user that the pin has been pulled low. When the button is not pressed, the protection diode is reverse biased with 5 V on the cathode and 3 V on the anode, protecting the GPIO pin. In this state, the LED will not be lit. The 3-V GPIO pin is connected to GND via a high impedance resistor to prevent it from floating during power up.

Raspberry Pi models are typically equipped with GPIO pins only. These pins are digital and current Raspberry Pi models do not have analog input/output pins. This makes the selection of an external ADC necessary to convert user inputs into valid signals. To make a functional knob, a potentiometer can be used with 3.3 V applied on one terminal, an external ADC input on the center tap (variable terminal), and GND on the last terminal. This allows the ADC input signal to vary from 3.3 V to 0 V. The ADC reading can be mapped to a resistance and used as the target value. Limiting the resistor range by using the same resistors on the resistor track allows for a simpler mapping of the potentiometer voltage to resistance. To match the target resistance to the actual resistance, the potentiometer voltage is first mapped to a target resistance. Then the motor is commanded to rotate in single step increments, with the ADC taking a single measurement between each step, until the measured voltage on the second ADC matches the expected voltage, within a suitable threshold. The load range and resistor values will affect the threshold value.

The ADC121C021 is an I2C-Compatible, 12-bit analog-to-digital converter that can communicate with the Raspberry Pi via I2C and has enough resolution to ensure accurate voltage measurements:

$$ADC_{Voltage\_Resolution} = \frac{V_{ADC\_Ref}}{2^{n\_bits}} \tag{5}$$
$$ADC_{Voltage\_Resolution} = \frac{3.3}{2^{12}} \approx 0.8\,mV$$

The other ADC121C021 is used to read in the current load resistance value and its input terminal can be connected to an external port to facilitate its connection to the resistor plate PCB.

3.2 Resistor Plate Design

3.2.1 Motor and Motor Driver Selection

The selected motor needs to be able to rotate a copper mechanical arm 360 degrees around the circumference of the resistor plate with a high degree of accuracy. Stepper motors can be controlled with high accuracy, have full 360 degrees range of motion, and can apply relatively high torque at lower speeds, which makes it a suitable motor for this application. The drawback of using a stepper motor, is that there is no indicator of the orientation of the motor at any given point, without using external means of determining the orientation of the shaft or rotor. Feedback will be applied later to overcome this limitation.

A bipolar NEMA-17 stepper motor with a step-angle of 1.8 degrees allows for 200 discrete steps per rotation. A motor such as Stepperonline's 17HS15-1504S-X1 or DFRobot's FIT0278, is a suitable selection for this design. The NEMA-17 class is large enough to produce enough torque at slower speeds to move a small arm assembly. The motor of this design has two windings with a rated current of 1.5 A per phase, and a winding resistance of 2.3 Ω. The DRV8833 is a dual H-Bridge motor driver that can be used to drive this motor as it can provide enough current for the motor in a small package. The motor driver will be configured to limit the current to 1 A per phase at VM = 5 V, by setting the sense resistor to 0.15 Ω, as illustrated in the *DRV8833 Dual H-Bridge Motor Driver* data sheet. The digital signals used to drive the motor are shown in Figure 3-3.

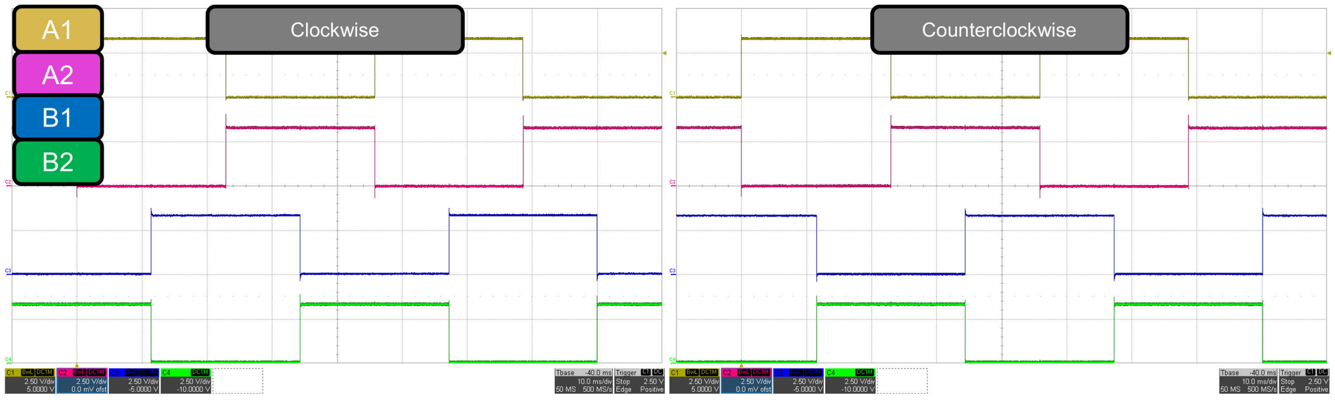


Figure 3-3. Motor Drive Signals

3.2.2 Resistor Track

A series array of power resistors arranged in a circular pattern form the resistor track. These resistors have surface mount packages, which are cheap, easy to assemble and have a low profile. These packages usually include a pad that can be connected to a heatsink, but these pads can be soldered down directly onto the resistive plate PCB instead, to prioritize minimizing PCB size over thermal capability. To ensure the largest possible resistance range, the resistors are placed in ascending order according to their resistance values. Different resistance values can be selected to increase the total resistance range, but this adds complexity to the feedback implementation. For this design, three different resistor values were selected: 0.05 Ω, 0.10 Ω, and 1.5 Ω and are distributed around the circumference of the resistor plate. The resistors for this design are placed clockwise in ascending order around the circumference of the board. The initial third of the circumference contains 0.05-Ω resistors, and so the resistance increases in steps of 0.05 Ω for a total of 1.5 Ω. The second section of the resistor track contains 0.1-Ω resistors and the series resistance increases in steps of 0.1 Ω for a total of 3 Ω. The remaining section contains 1.5-Ω resistors to achieve a total resistance of 36 Ω. With all of the resistors in series, the total available resistance can vary from 0.05 Ω to 40.5 Ω. Mounting holes can be included on the resistor plate to allow for a heatsink, enabling higher continuous power dissipation through the apparatus. The resistance from the positive terminal to the negative terminal is the sum of the series resistors between the positive terminal and the mechanical copper arm (GND). The remaining resistors on the resistor track form the RDiv which is used to apply feedback control to the apparatus. It is critical to expose the PCB copper and remove the silkscreen along the path of the mechanical arm assembly to ensure that electrical contact can be made between the track and the arm.

One important consideration when designing the resistor track is the size-to-accuracy trade off. By using larger resistors, their radial arrangement will likely require a plate with a larger radius which is generally undesirable.

The maximum number of resistors are limited by the step size of the chosen motor. For instance, if the motor is capable of rotating 1.8 degrees per step for a total of 200 steps per rotation, then at most, the load can contain 200 discrete resistance values, if they are arranged such that each resistor is 1.8 degrees away from the next resistor. However, this is a very small step size and would push out the resistor track farther away from the center. This particular design favors a small solution size and so the number of resistors used is less than 200 and they are placed at angles larger than 1.8 degrees from each other. Figure 3-4 shows a section of the track.

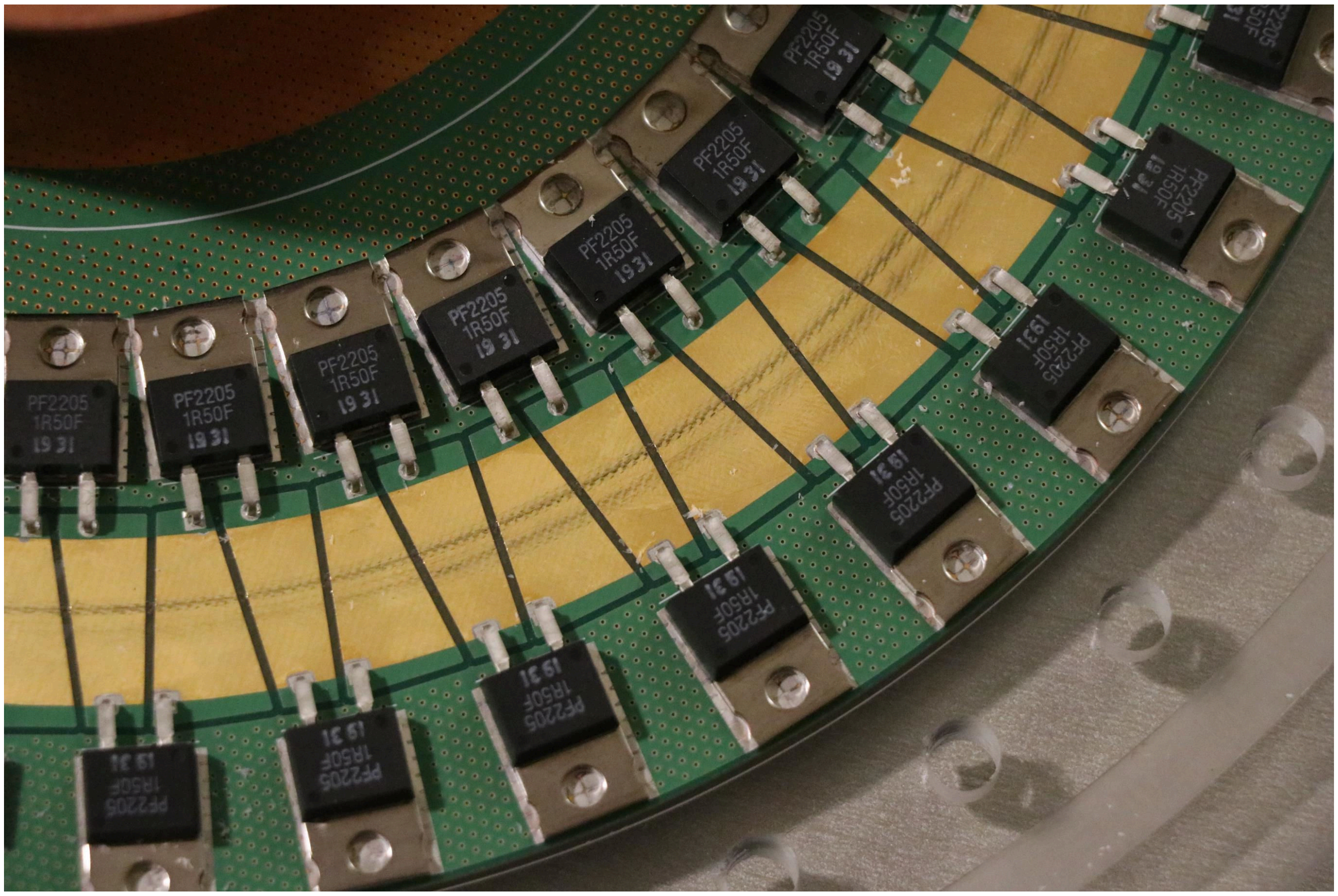


Figure 3-4. Resistor Track

3.2.3 Mechanical Arm Assembly

The load applied to the DUT is equal to the total equivalent series resistance between the positive terminal of the apparatus and GND in addition to the resistance of the leads connecting the DUT to the apparatus. The resistors on the resistive track all connect to each other in series, forming an open loop where there is no direct connection between the first and last resistors. The stepper motor moves a mechanical arm assembly that connects one node of this resistive track to GND and acts similarly to a wiper on a rheostat. This mechanical arm contacts the resistive track via high-current pogo pins. The mechanical arm assembly is connected to the rotor of the stepper motor, and is fixed in place using a set-screw. Between the connection point from the resistive track and the rotor, the mechanical arm makes another connection to an elevated copper base plate via additional pogo pins as shown in Figure 3-5. By using the mechanical arm to set the load, one end of the resistive track can be used as a load, while the other end can be used to apply positional feedback. The mechanical arm assembly allows the input terminals of the resistive load to remain stationary and eliminates the need for a slip ring.

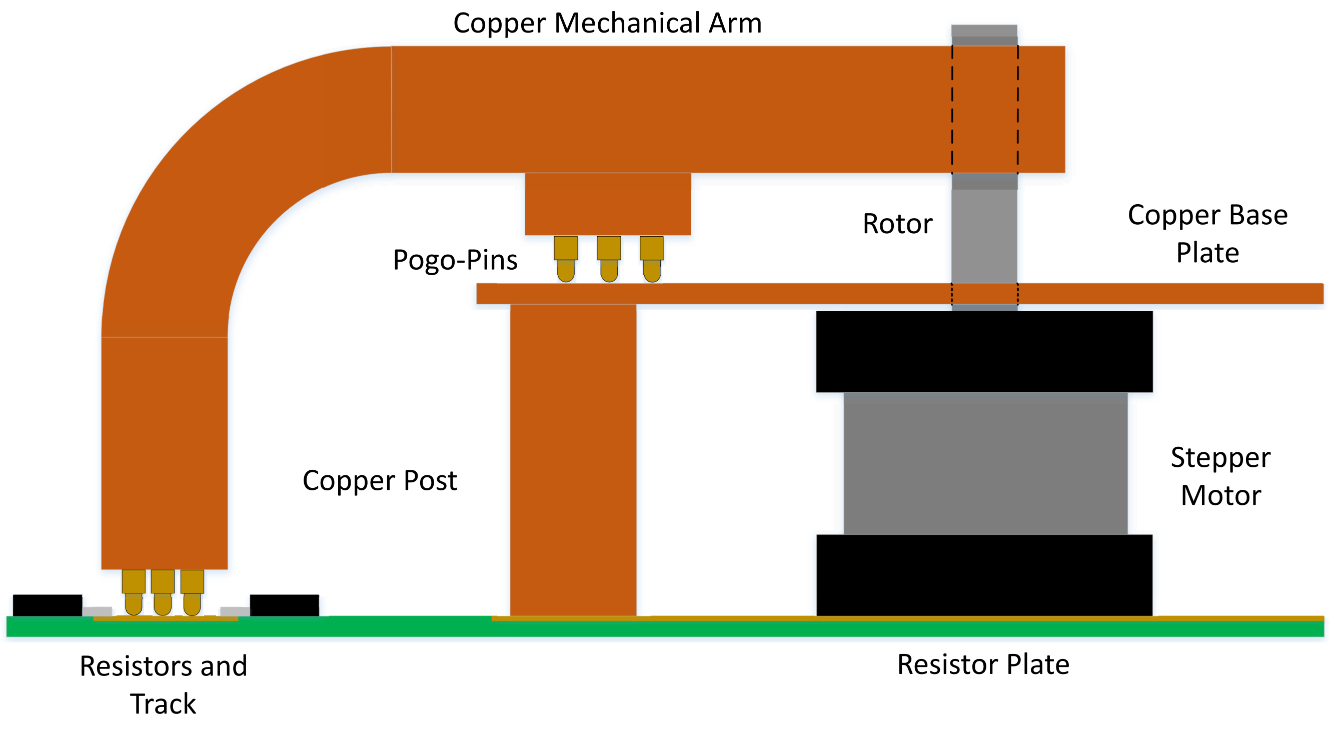


Figure 3-5. Mechanical Arm Assembly

3.2.4 Feedback Control

Stepper motors can be controlled with high accuracy, but do not inherently have feedback to indicate their position. An external means of feedback is therefore necessary to control the stepper motor effectively. The resistive track provides a means to do so by using an external reference voltage which connects to the resistive track via a fuse and series resistor, forming a variable resistive divider. The middle node voltage of this variable resistive divider changes depending on the position of the metal arm assembly and is tracked by the second ADC located on the controller module. The middle node voltage can vary from 0 V up to the divided voltage, VDiv, which is defined by the fixed series resistance chosen as per equation:

$$V_{Div} = V_{Ref} \times \frac{R_{Div}}{\left(R_{Div} + R_{Ref}\right)} \tag{6}$$

Where RDiv is calculated according to:

$$R_{Div} = R_{Total} - R_{Load} \tag{7}$$

This design uses the quiet 3.3-V rail from the controller board as a reference voltage and a 50-Ω series resistor, which limits VDiv to values between 0 V (load of 40.5 Ω) and approximately 1.5 V (load of 0.0 Ω). The feedback scheme is outlined in Figure 3-6.

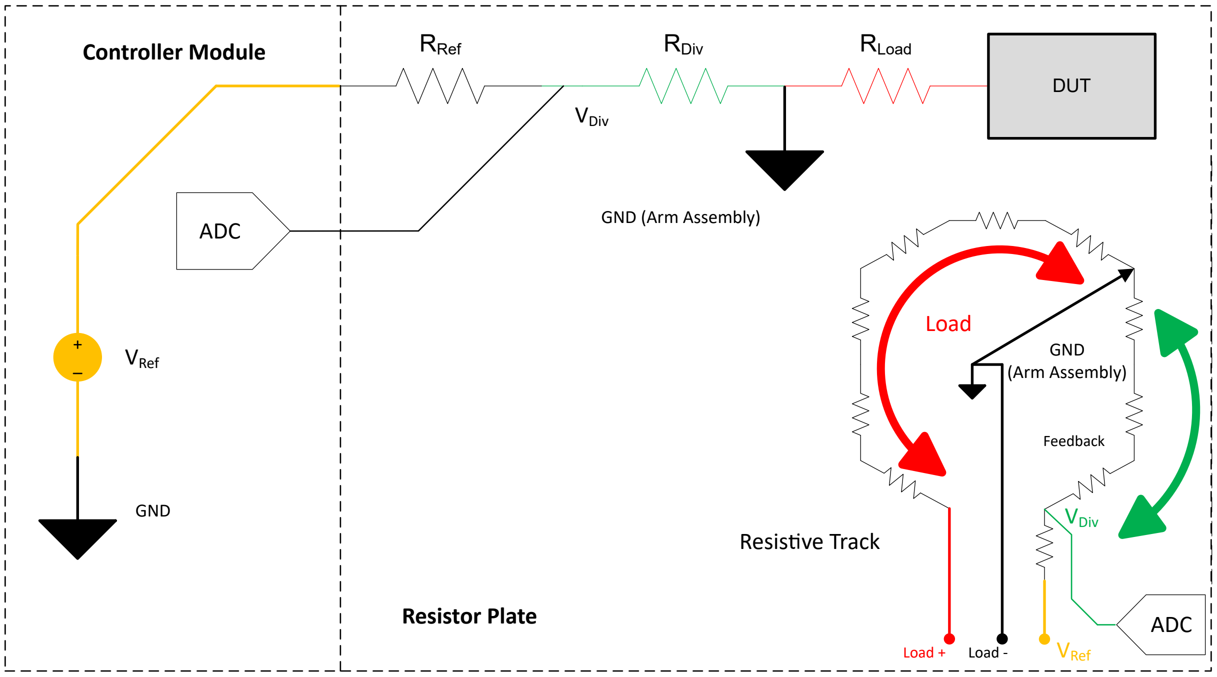


Figure 3-6. Feedback Control

Care must be taken to ensure that the LDO is able to provide the maximum required current. The selected peak current output current of the LDO is typically 150 mA and is no less than 100 mA, making it a suitable selection. The maximum LDO current for this design is defined as:

$$Max\quad LDO\quad Current = I_{LDO\_Max} = V_{Ref}/R_{Ref} \qquad (8)$$
$$I_{LDO\_Max} = 3.3/50 \approx 70\ mA$$

If a single value of resistor was used for the entire resistive track, the voltage and resistance relationship would be linear, but since different sized resistors were used, the relationship is piece-wise. The relationship can be empirically determined between the resistance and position. Alternatively, the voltage VDiv, can be calculated for each resistor.

4 Thermal Considerations

There are a number of considerations to take into account when designing such a resistive load with regards to its power and thermal ratings. The resistors on the resistor plate will be the hottest components when applying a load to a voltage source. The thermal limit of the components will set the primary limit of how much current the motorized resistive load can handle. The resistors selected for this design are rated for a maximum temperature of 155°C and a power rating of 50 W. The resistors will generally hit their temperature limit at a power level under their maximum power rating. The power rating generally applies when the resistor is sufficiently cooled via heatsinking, or active cooling. Thermally-optimized layout can generally help cool the resistors, but not as effectively as a heatsink or active cooling. If the resistor is not heatsinked properly or actively cooled, then the power rating is derated.

Given the series connection of the load resistors, a different number of resistors will form the load for a given voltage. For example, to load a 1-V voltage rail with 10 A, the motorized resistive load needs to ensure that the resistance between the load terminals is 0.1 Ω. If the leads and arm are assumed to have a resistance of 0.05 Ω then the voltage will be applied over a single 0.05-Ω resistor. The temperature and power rating can then be compared against the ratings of this resistors. In this example, the temperature rises to 100°C and stabilizes around that point without active cooling as shown in Figure 4-1. The power dissipated in the resistor is the product of the applied voltage over the resistor by the current through it as shown:

$$P_{Resistor} = V \times I \qquad (9)$$
$$P_{Resistor} = I^2 \times R$$
$$P_{Resistor} = 10^2 \times 0.05 = 5\,W$$

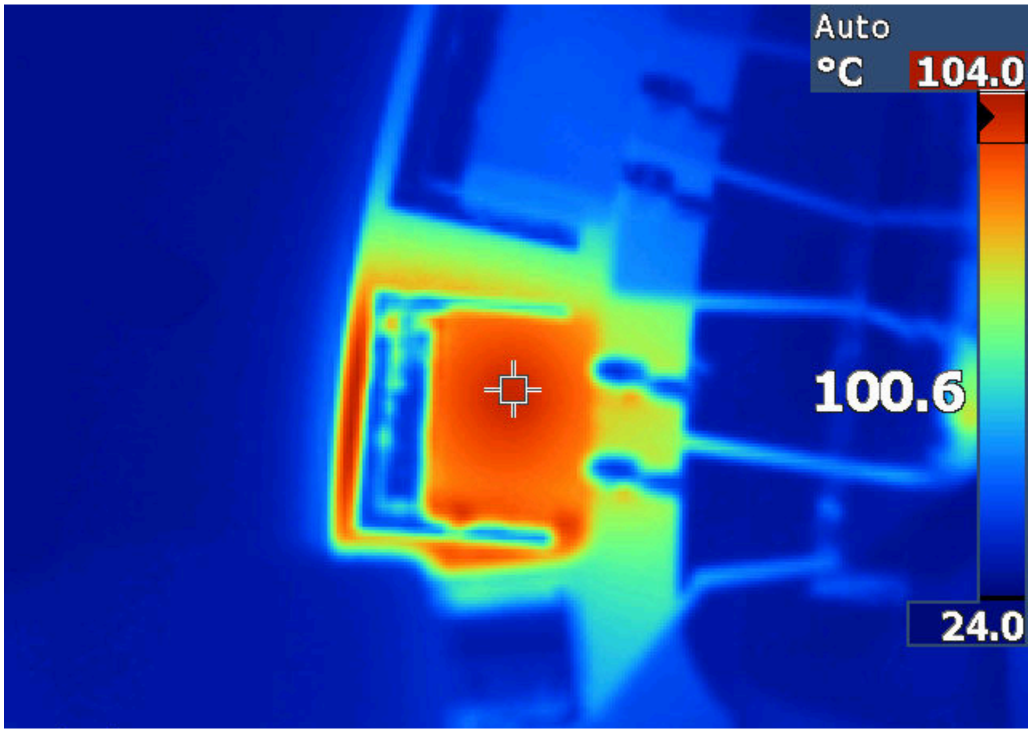


Figure 4-1. Thermal Performance at 1 V, 10 A, and 0.05 Ω

In this case the temperature and power rating of the resistor are not violated, but if the voltage were increased to 3.3 V, then the necessary resistance to apply 10 A would be 0.33 Ω. On this design that translates to 5 resistors, if we continue assuming that the arm and leads are 0.05 Ω. Now the power dissipated per resistor is less, and so higher currents may be applied, but if the load is held continuously then the temperature will rise due to the increased heating of adjacent resistors. The thermal image shown in Figure 4-2 demonstrates this effect, as the resistors continue to heat up after applying the load for less than 5 seconds and do not stabilize at an acceptable temperature. This example illustrates two main points: active cooling is required for higher current capability, and assuming only one resistor value is used for all resistors, then the power dissipated by each resistor is:

$$P_{Resistor} = \left(V_{Applied} \times I_{Load}\right) / \left(n_{Resistors\_in\_circuit}\right) \tag{10}$$

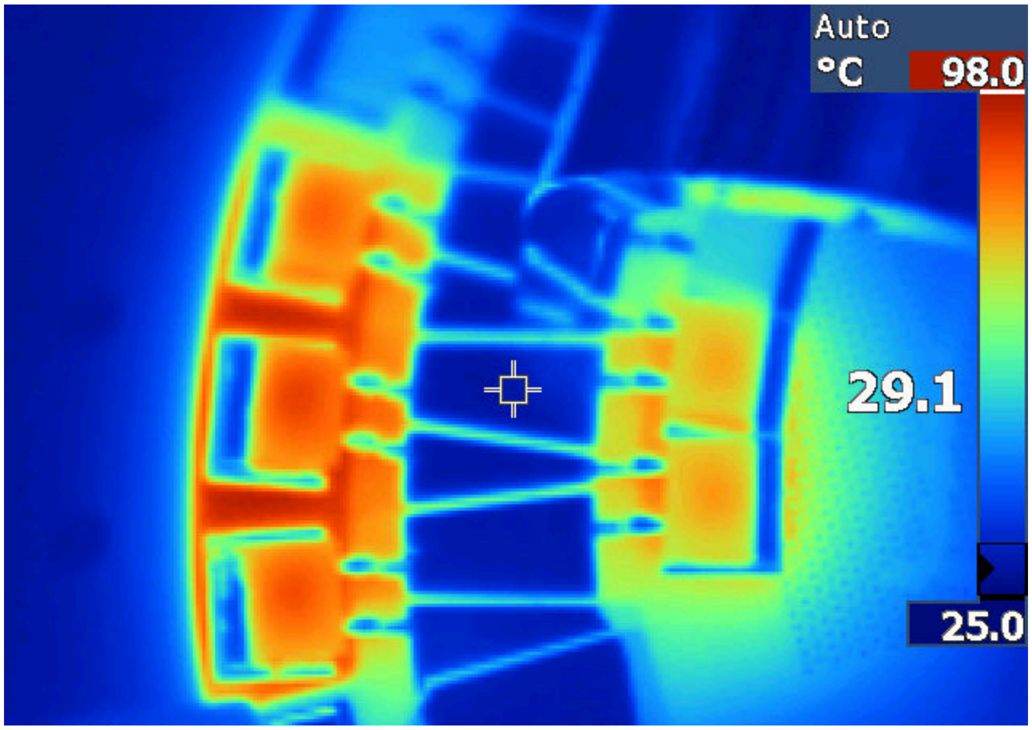


Figure 4-2. Thermal Performance at 3.3 V, 10 A, and 0.33 Ω

5 Performance and Results

The overall design is a powerful tool that can be used to apply resistive loads. This design utilizes several different resistor values along the load resistor loop to increase the dynamic range of available load resistances. The resistance ranges of this design include a low, middle, and high range which are [0 to 1.5 Ω], [1.6 to 4.5 Ω] and [6 to 40.5 Ω]. These ranges are ideal for common voltage rails such as 1.8 V, 3.3 V and 5 V. The overall profile is shown in Figure 5-1 and a demonstration of current sinking capability is shown in Figure 5-2.

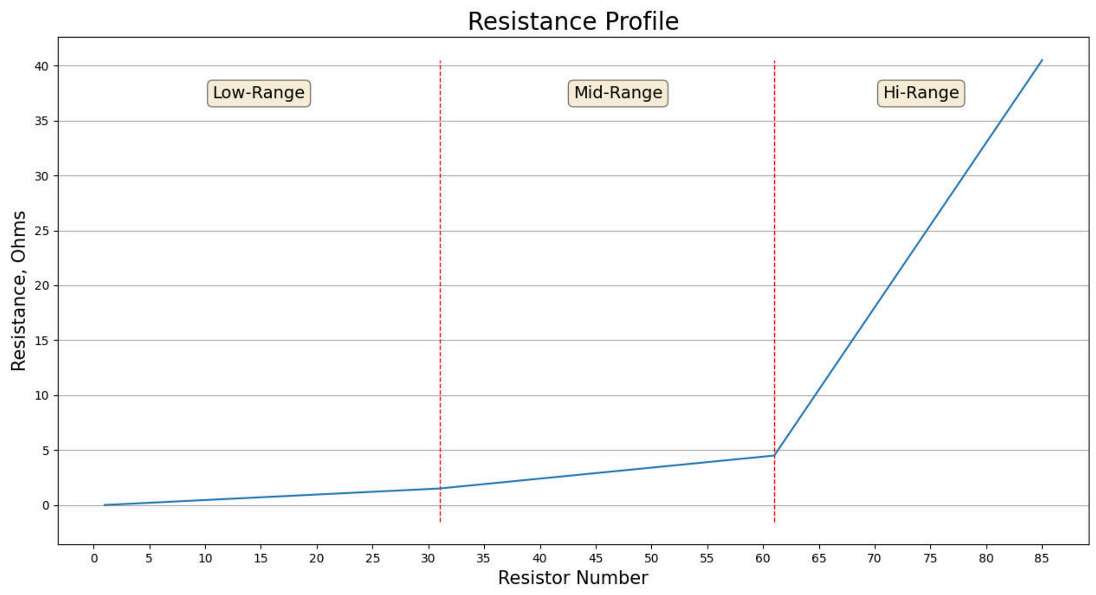


Figure 5-1. Resistance Profile

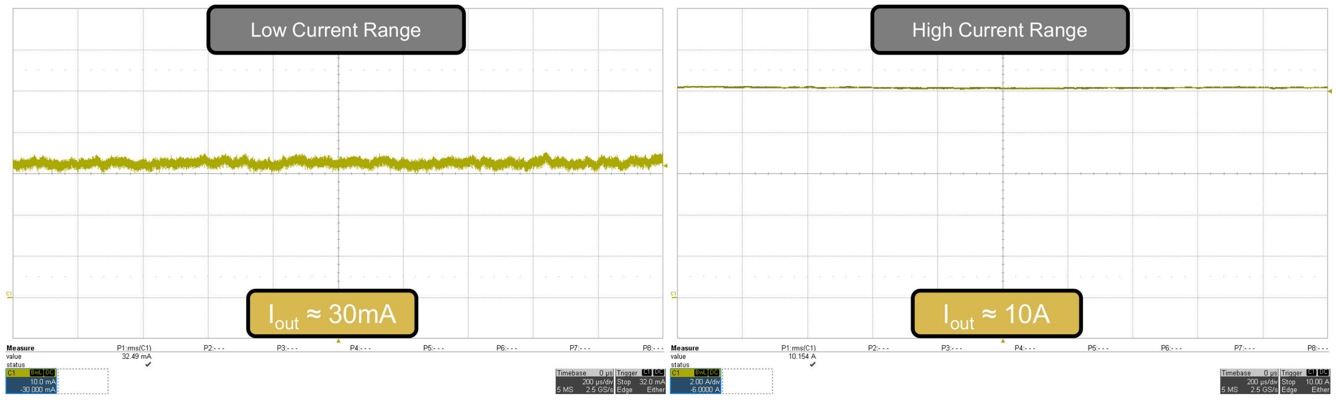


Figure 5-2. Demonstrated Load Range

6 Summary

A motorized resistive load is an effective and convenient means of loading different systems and voltage rails. The resistive load track enables feedback control while maintaining a load that can operate at voltages as low as zero volts, making it suitable for testing voltage rails, even as they ramp up or down. Resistive loads are also preferred for stability testing, such as generating a bode plot for a control loop of the power converter. This resistive load offers the functionality of an automated rheostat in a convenient and relatively compact form.

Additional modifications can also be introduced to meet specialized applications. For example, active cooling can be introduced to potentially increase the current rating of the apparatus, or the code and controller module can be modified to suit a controller other than the Raspberry Pi or allow for automation and automated testing. Additionally, the feedback scheme can be modified to sense the voltage over the first sense resistor, which will be proportional to the current drawn by the load, rather than use an external voltage reference with a variable resistive divider. Calibration routines can also be programmed to maintain accuracy and account for changes in resistance over time as the apparatus wears with use.

7 Appendix

7.1 Controller Board Main Schematic

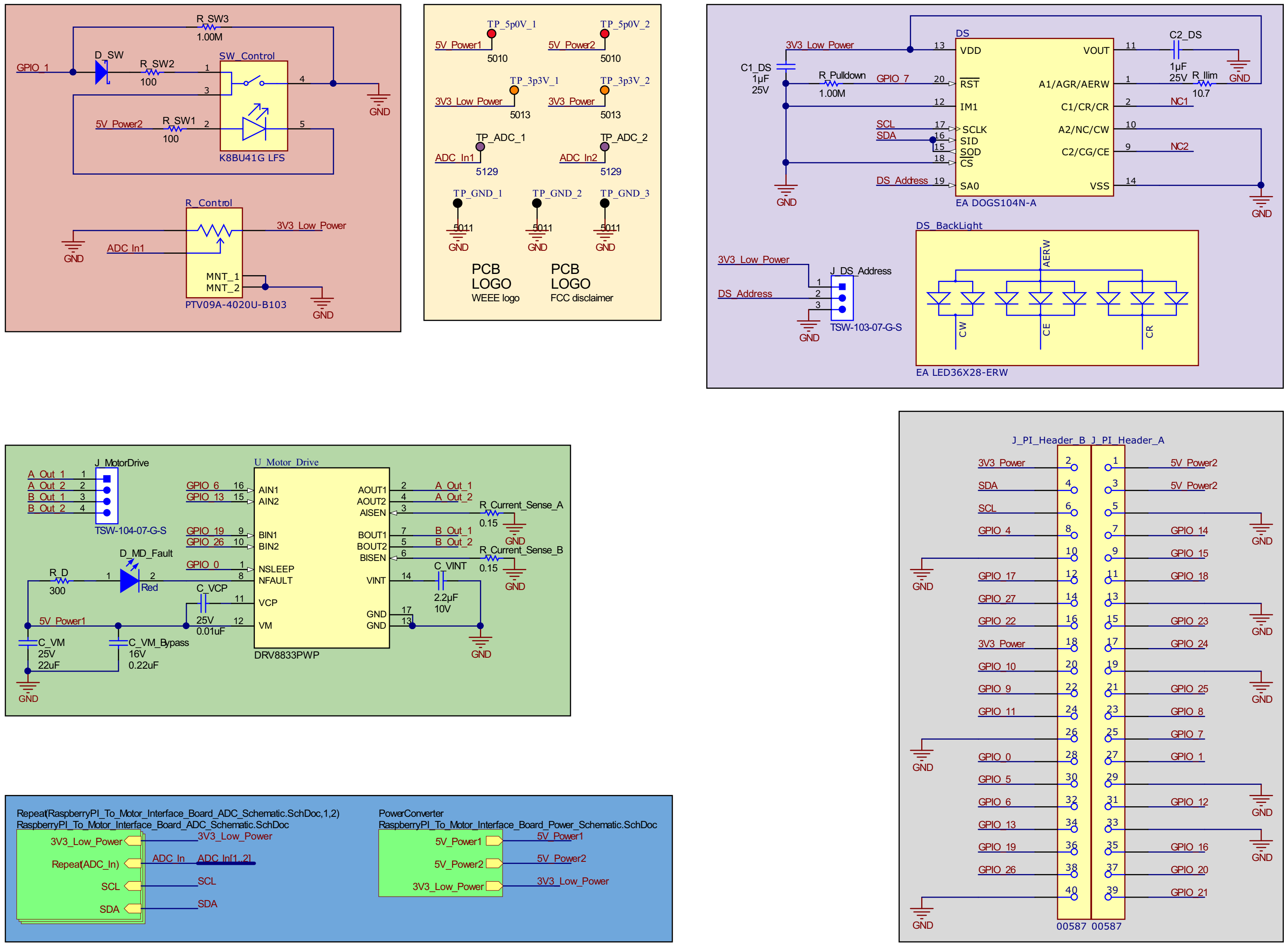


Figure 7-1. Controller Board: Main Schematic

7.2 Controller Board Sub-Schematics

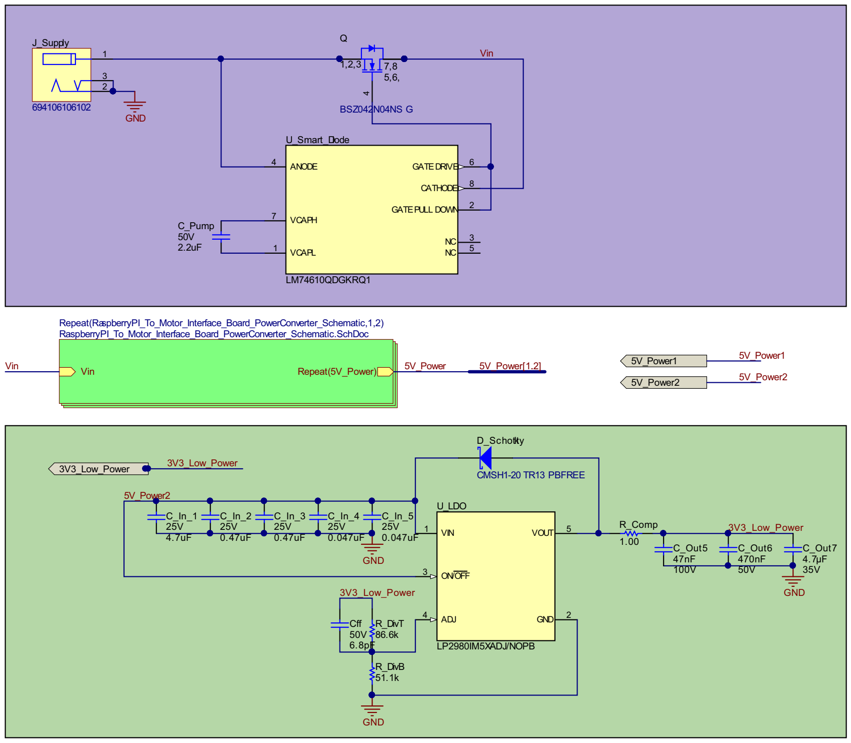


Figure 7-2. Controller Board: Power Sub-Schematic

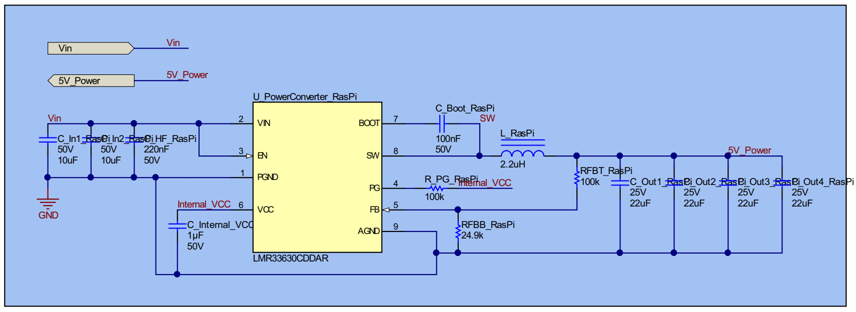


Figure 7-3. Controller Board: Power Converter Sub-Schematic

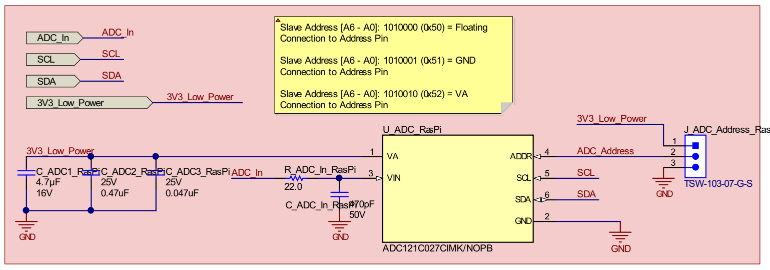
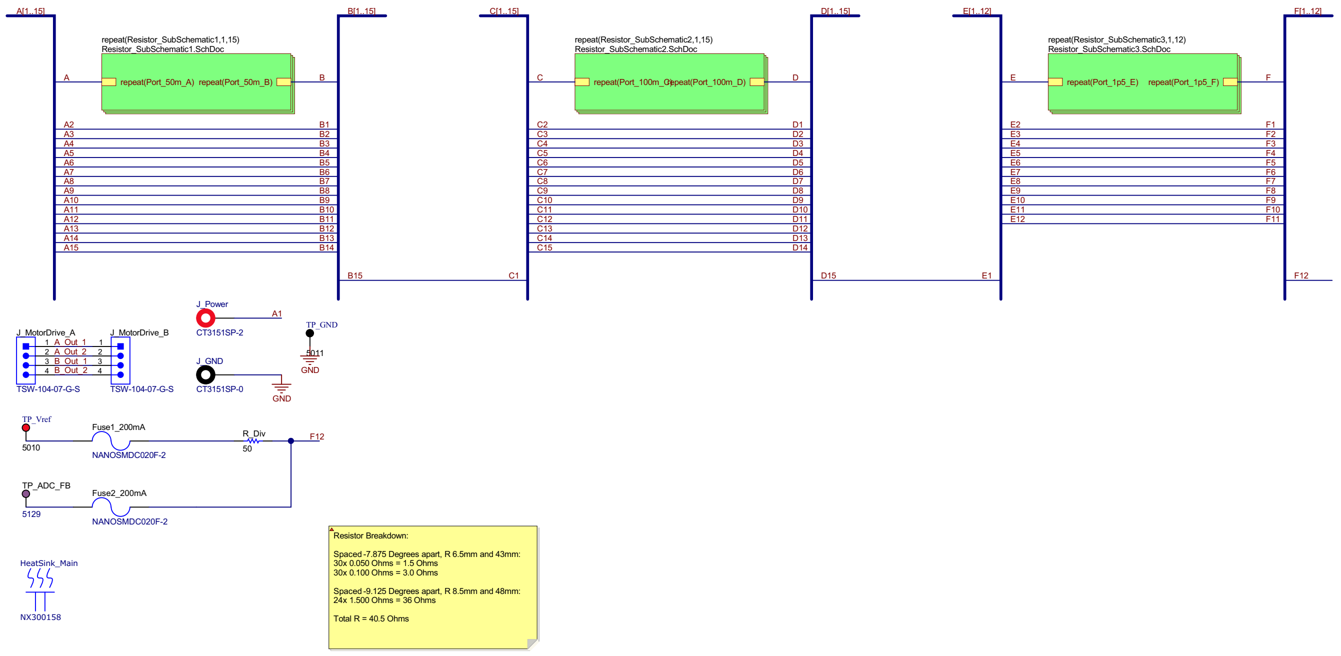


Figure 7-4. Controller Board: ADC Sub-Schematic

7.3 Resistor Plate Schematics



Resistor Breakdown:

Spaced -7.875 Degrees apart, R 6.5mm and 43mm:
30x 0.050 Ohms = 1.5 Ohms
30x 0.100 Ohms = 3.0 Ohms

Spaced -9.125 Degrees apart, R 8.5mm and 48mm:
24x 1.500 Ohms = 36 Ohms

Total R = 40.5 Ohms

Figure 7-5. Resistor Plate: Main Schematic

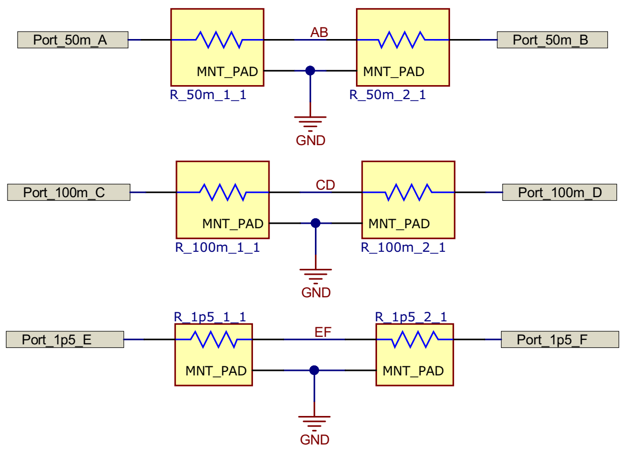


Figure 7-6. Resistor Plate: Resistor Sub-Schematics

7.4 Python Code

```
# Motorized Resistive Load - Python Code by Abdallah Obidat

# Note that the Raspberry Pi was configured to boot into this program. This can be achieved in a
number of ways.

import smbus as i2cBUS  # import smbus to allow control of the I2C bus
import RPi.GPIO as GPIO  # import RPi.GPIO to allow control of the GPIO pins
import time as tm  # import time to allow the insertion of time delays in the code

bus = i2cBUS.SMBus()  # create a SMBus object
bus.open(1)  # open communication line of the i2c bus object - port 1 is being used here

display_register_address = 0x3C
tm.sleep(5)  # wait for two seconds after initializing the i2c bus

# the last value turns the display on or off (0x08 = OFF, 0x0F = ON)
initialization_list = [0x3A, 0x09, 0x06, 0x1E, 0x39, 0x1B, 0x6E, 0x56, 0x7A, 0x38, 0x0F]

# add 1 to the display reg address to set the write bit high
bus.write_i2c_block_data(display_register_address+1, 0x00, initialization_list)

# Create a dictionary to easily output words on the display
Alpha = {"A": 0x41, "B": 0x42, "C": 0x43, "D": 0x44, "E": 0x45, "F": 0x46, "G": 0x47, "H":0x48,
"I": 0x49, "J": 0x4A,
         "K": 0x4B, "L": 0x4C, "M": 0x4D, "N": 0x4E, "O": 0x4F, "P": 0x50, "Q": 0x51, "R": 0x52,
"S":0x53, "T": 0x54,
         "U": 0x55, "V": 0x56, "W": 0x57, "X": 0x58, "Y": 0x59, "Z": 0x5A, "Ohm": 0xB5, ":": 0x3A,
"-": 0x2D}

Num = {"0": 0x30, "1": 0x31, "2": 0x32, "3": 0x33, "4": 0x34, "5": 0x35, "6": 0x36, "7": 0x37, "8":
0x38, "9": 0x39,
       ".": 0x2E}

# select ROM_A for the display
# reg values: RE high, ROM selection, ROM choice (0x00 = ROM_A, 0x04 = ROM_B, 0x08 = ROM_C)
# note that 0x40 or 0x5F need to be written in order to actually select the ROM, bits are 10XX XXXX
bus.write_i2c_block_data(display_register_address+1, 0x00, [0x3A])
bus.write_i2c_block_data(display_register_address+1, 0x00, [0x72])
bus.write_i2c_block_data(display_register_address+1, 0x40, [0x00])
bus.write_i2c_block_data(display_register_address+1, 0x00, [0x38])

# set display to double height mode
bus.write_i2c_block_data(display_register_address+1,0x00,[0x3A, 0x1B, 0x3C])

# clear display
bus.write_i2c_block_data(display_register_address+1, 0x00, [0x01])

direction = "clockwise"  # set direction global variable

# The motor used has 200 steps per full revolution, which is 1.8 degrees per step
measured_step = 0  # variable used to show position on LCD display

# set ADC variables
VDD = 3.3  # VDD and full scale voltage Of ADC
ADC_Bits = 12  # the ADC121C021 is a 12-bit ADC
ADC_full_scale = (2**ADC_Bits)-1  # 2^12 States for the ADC output, but indexing starts at 0, so
the variable is
# decremented by 1

# ADC1 is used for the set value and reads in the voltage controlled by the potentiometer knob
ADC_1_device_address = 0x50 # the device address of the ADC121C021, which is assigned as a
hexadecimal number
ADC_1_result_register_address = 0x0
final_read_result_1 = ""  # initialize a blank string that will form the final read result

# ADC2 is used for positional feedback and reads the voltage on the dynamic resistive divider to
provide current
# location information
ADC_2_device_address = 0x51
ADC_2_result_register_address = 0x0
final_read_result_2 = ""

# create a list of 0 ohm values that corresponds to the length of the track where a short exists
R_set0 = [0 for i in range(1, 7)]
```

```
# there are 30x 0.050 Ohm resistors spaced ~two steps from each other
# this list comprehension creates a list of sequenced resistances that matches the actual design of
the motorized load
# and is interleaved with itself to double its size
R_set1 = [round(i * 0.05, 2) for i in range(0, 31)]
R_set1 = [value for pair in zip(R_set1, R_set1) for value in pair]

# there are 30x 0.10 Ohm resistors spaced ~two steps from each other
# this list comprehension creates a list of sequenced resistances that matches the actual design of
the motorized load
# and is interleaved with itself to double its size
R_set2 = [round((i + 1) * 0.10 + R_set1[-1], 2) for i in range(0, 30)]
R_set2 = [value for pair in zip(R_set2, R_set2) for value in pair]

# there are 24x 1.5 Ohm resistors spaced ~three steps from each other
# this list comprehension creates a list of sequenced resistances that matches the
# motorized load and is interleaved with itself to double its size
R_set3 = [round((i + 1) * 1.5 + R_set2[-1], 2) for i in range(0, 24)]
R_set3 = [value for pair in zip(R_set3, R_set3, R_set3) for value in pair]

# merge the interleaved lists together to create a "set" array of resistors,
# the values of which correspond to the steps that the motor can target
R_set = R_set0 + R_set1 + R_set2 + R_set3

# create a list of feedback voltages that correspond to different loads
V_fb = list()

# initialize a list to store the current state of the motor's logic inputs
state_list = [GPIO.HIGH, GPIO.HIGH, GPIO.LOW, GPIO.LOW]


def RotateOneStepClockwise(state_list_param):
    # define an interrupt/callback that accepts the global variable "state_list" to single rotate
the motor,
    # by a single clockwise step, based on the current state of the motor driver logic inputs

    global Delay  # allow the callback to access the global "Delay" variable
    global VDD
    global ADC_Bits
    global ADC_full_scale
    global ADC_1_device_address
    global ADC_1_result_register_address
    global final_read_result_1
    global ADC_2_device_address
    global ADC_2_result_register_address
    global final_read_result_2

    # initialize the new state list variable "new_state_list"
    new_state_list = [None, None, None, None]

    # set the new state, based on the current state, such that the motor rotates by a single
clockwise step
    new_state_list[0] = state_list[3]
    new_state_list[1] = state_list[0]
    new_state_list[2] = state_list[1]
    new_state_list[3] = state_list[2]

    # set the Raspberry Pi digital outputs (motor drive logic inputs) according to the newly
determined state,
    # which is based on the current state
    GPIO.output(6, state_list[0])
    GPIO.output(19, state_list[1])
    GPIO.output(13, state_list[2])
    GPIO.output(26, state_list[3])

    tm.sleep(Delay)  # delay between steps according to the global "Delay" variable

    # read in two bytes (16 bits) of data
    ADC_2_read_result = bus.read_i2c_block_data(ADC_2_device_address,
ADC_2_result_register_address, 2)

    # concatenate results to form final read result
    # the first byte returned is shifted by 8 digits before concatenating results
    final_read_result_2 = ((ADC_2_read_result[0]) << 8) | (ADC_2_read_result[1])

    # for debug
    # final_read_voltage/VDD = final_read_result/ADC_full_scale -> solve for final_read_voltage
    # current_read_voltage = VDD*(final_read_result)/ADC_full_scale -> convert the reading into a
voltage
```

```
    return new_state_list  # return the new state of the motor drive inputs

def RotateOneStepCounterClockwise(state_list_param):
    # define an interrupt/callback that accepts the global variable "state_list" to single rotate 
the motor,
    # by a single counterclockwise step, based on the current state of the motor driver logic inputs

    global Delay  # allow the callback to access the global "Delay" variable
    global VDD
    global ADC_Bits
    global ADC_full_scale
    global ADC_1_device_address
    global ADC_1_result_register_address
    global final_read_result_1
    global ADC_2_device_address
    global ADC_2_result_register_address
    global final_read_result_2

    # initialize the new state list variable "new_state_list"
    new_state_list = [None, None, None, None]

    # set the new state, based on the current state, such that the motor rotates by a single step
    new_state_list[0] = state_list[1]
    new_state_list[1] = state_list[2]
    new_state_list[2] = state_list[3]
    new_state_list[3] = state_list[0]

    # set the Raspberry Pi digital outputs (motor drive logic inputs) according to the newly 
determined state,
    # which is based on the current state
    GPIO.output(6, state_list[0])
    GPIO.output(19, state_list[1])
    GPIO.output(13, state_list[2])
    GPIO.output(26, state_list[3])

    tm.sleep(Delay)  # delay between steps according to the global "Delay" variable

    # read in two bytes (16 bits) of data
    ADC_2_read_result = 
bus.read_i2c_block_data(ADC_2_device_address,ADC_2_result_register_address,2)

    # concatenate results to form final read result
    # the first byte returned is shifted by 8 digits before concatenating results
    final_read_result_2 = ((ADC_2_read_result[0]) << 8) | (ADC_2_read_result[1])

    # for debug
    # final_read_voltage/VDD = final_read_result/ADC_full_scale -> solve for final_read_voltage
    # current_read_voltage = VDD*(final_read_result)/ADC_full_scale -> convert the reading into a 
voltage

    return new_state_list  # return the new state of the motor drive inputs

def MotorStepInterrupt(self):
    # define an interrupt/callback that accesses a number of global variables and controls the 
motor movement to move it
    # from its current position to the desired position, such that the target resistance is formed 
between its terminals
    #
    # The callback/interrupt wakes up the motor drive chip. It then determines the direction of 
rotation and enters a
    # while loop. After entering the loop, it will call either the "RotateOneStepClockwise" 
function or the
    # "RotateOneStepCounterClockwise" function depending on the "direction" variable value. The 
loop exits once the
    # final voltage target obtained via the first ADC matches the voltage, within an acceptable 
tolerance, on the second
    # ADC. The current state of the motor drive logic inputs are tracked and updated with each 
iteration in the while
    # loop.

    # declare global variables that the callback/interrupt will need access to
    global direction
    global measured_step
    global state_list
    global VDD
    global ADC_Bits
    global ADC_full_scale
    global ADC_1_device_address
    global ADC_1_result_register_address
    global final_read_result_1
    global ADC_2_device_address
    global ADC_2_result_register_address
    global final_read_result_2
    global R_set

    # read in two bytes (16 bits) of data
    ADC_1_read_result = bus.read_i2c_block_data(ADC_1_device_address,
ADC_1_result_register_address, 2)

    # concatenate results to form final read result
    # the first byte returned is shifted by 8 digits before concatenating results
    final_read_result_1 = ((ADC_1_read_result[0]) << 8) | (ADC_1_read_result[1])

    # convert the ADC reading into a target voltage
    final_read_voltage = VDD*final_read_result_1/ADC_full_scale
    print('ADC Reading = '+str(final_read_voltage) + ' V')

    # read in two bytes (16 bits) of data
    ADC_2_read_result = bus.read_i2c_block_data(ADC_2_device_address,
ADC_2_result_register_address, 2)

    # concatenate results to form final read result
    # the first byte returned is shifted by 8 digits before concatenating results
    final_read_result_2 = ((ADC_2_read_result[0]) << 8) | (ADC_2_read_result[1])

    # convert the ADC reading into a voltage that represents current position
    current_read_voltage = VDD*final_read_result_2/ADC_full_scale

    # select the target resistance based on the knob (potentiometer) voltage reading
    R_target_index = round((final_read_result_1/ADC_full_scale)*len(R_set))

    # do not allow the motor to turn more than a full rotation to get to its target
    if R_target_index > 199:
        R_target_index = 199

    # select the target resistance which is equal to the desired load resistance
    R_target = R_set[R_target_index]

    # choose a threshold that defines the range for an acceptable reading - this changes based on
resistor values used
    # these values were verified empirically for this specific design
    if R_target < 1.5:
        tolerance = 0.001
    elif R_target > 4.5:
        tolerance = 0.030
    else:
        tolerance = 0.001

    # calculate Vdiv, the expected voltage at the position of interest
    final_voltage_target = ((3.3/(49.9+40.5-R_target))*(40.5-R_target))

    # determine direction of rotation based on the current position and target resistance
    if final_voltage_target > current_read_voltage:
        direction = "counter_clockwise"
    else:
        direction = "clockwise"

    i = 0  # reset index
    while 1:
        Delay = 10/1000  # sets an additional delay between rotations
        current_read_voltage = VDD*final_read_result_2/ADC_full_scale
        if direction == "clockwise":
            state_list = RotateOneStepClockwise(state_list)  # clockwise rotation
        else:
            state_list = RotateOneStepCounterClockwise(state_list)  # counterclockwise rotation
        i = i+1  # increment the counter after either function is called

        if (abs(final_voltage_target - current_read_voltage) < tolerance) or (i > 199):  # while
loop exit condition
            measured_step = R_target_index
            print("R_target: " + str(R_target))
            print("final_voltage_target: " + str(final_voltage_target))
            print("current_read_voltage: " + str(current_read_voltage))
            break

    # additional delay
    tm.sleep(Delay)
```

```
# main code

GPIO.setwarnings(False)  # disable warnings that occur when configuring the GPIO signals

# refer to the GPIO signals based on the broadcom chip number and not the board's breakout header
containing the pins
GPIO.setmode(GPIO.BCM)

GPIO.setup(1, GPIO.IN, GPIO.PUD_UP) # set GPIO 1 as a digital input pin and have it pulled up by
default

# configure a trigger event for the callback/interrupt "MotorStepInterrupt". The trigger is set to
occur when a digital
# falling edge occurs on GPIO 5 and a debounce time of 3000ms is set following an event
GPIO.add_event_detect(1, GPIO.FALLING, callback=MotorStepInterrupt, bouncetime=3000)

# enable/wakeup the motor drive IC
GPIO.setup(0, GPIO.OUT)
GPIO.output(0, GPIO.HIGH)

# set all of the GPIO signals that will be designated as logic inputs to the motor drive to be
digital outputs on the
# Raspberry Pi
GPIO.setup(6, GPIO.OUT)
GPIO.setup(19, GPIO.OUT)
GPIO.setup(13, GPIO.OUT)
GPIO.setup(26, GPIO.OUT)

# set all of the digital inputs to the motor drive to logic low initially
GPIO.output(6, GPIO.LOW)
GPIO.output(19, GPIO.LOW)
GPIO.output(13, GPIO.LOW)
GPIO.output(26, GPIO.LOW)

Delay = 10  # Delay in s
Delay = Delay/1000  # Delay in ms

while 1:

    # setting all of the  the Raspberry Pi outputs (motor drive logic inputs) has almost the same
effect
    # as disabling the motor drive IC. This reduces the consumed power while the motor isn't in
motion,
    # but doesn't protect the motor from unwanted rotation due to any potential external torque
sources
    GPIO.output(6, GPIO.LOW)
    GPIO.output(19, GPIO.LOW)
    GPIO.output(13, GPIO.LOW)
    GPIO.output(26, GPIO.LOW)

    # read in two bytes (16 bits) of data
    ADC_1_read_result = bus.read_i2c_block_data(ADC_1_device_address,
ADC_1_result_register_address, 2)

    # concatenate results to form final read result
    # the first byte returned is shifted by 8 digits before concatenating results
    final_read_result_1 = ((ADC_1_read_result[0]) << 8) | (ADC_1_read_result[1])
    final_read_voltage = VDD*final_read_result_1/ADC_full_scale

    # user's current resistor selection to be displayed on LCD display
    current_step = round(final_read_result_1*199/ADC_full_scale)

    # prepare strings to be displayed on LCD display
    set_string = "MEAS:"+str(R_set[round(measured_step)])
    meas_string = "SET:"+str(R_set[round(current_step)])
    set_string_list = list()
    meas_string_list = list()

    for character in set_string:
        try:
            set_string_list.append(Alpha[character])
        except:
            set_string_list.append(Num[character])

    for character in meas_string:
        try:
            meas_string_list.append(Alpha[character])
        except:
            meas_string_list.append(Num[character])

    set_string_list.append(Alpha["Ohm"])
    meas_string_list.append(Alpha["Ohm"])

    # clear display
    bus.write_i2c_block_data(display_register_address+1, 0x00, [0x01])

    # display resistor value to be set (SET) on LCD display
    bus.write_i2c_block_data(display_register_address+1, 0x40, set_string_list)

    # move cursor position to 1st position on 1st row
    bus.write_i2c_block_data(display_register_address+1, 0x00, [0xA0])

    # display current resistance (MEAS) on LCD display
    bus.write_i2c_block_data(display_register_address+1, 0x40, meas_string_list)

    # add a delay of at least half a second between setting the GPIO low and then setting it high.
    tm.sleep(500*1/1000)

# close I2C communication bug - code only reachable while debugging
# bus.close() # close communication line of the bus object
```