Last modification: October 22, 2024

RBE 2002 (B-24)
Lab 1: Introduction to the Romi
Lab Activities

There are individual activities that must be completed before the start of lab. Specifically, you must be ready to go with your robot and programming environment *before* the start of lab.

Overview

In this lab, you will be introduced to Pololu's Romi platform and the programming framework that we will use for this term's activities. You will build some basic controllers that you will rely on throughout the term, culminating with programming your robot to follow a line and then performing experiments to assess your robot's performance. Presumably, you have programmed a robot to follow a line at least once in your career, so here you can concentrate on the code structure, timing of the control loops, and so forth.

1 Objectives

To complete this set of activities, the student will:

- Explore the Romi hardware and the library framework,
- Implement PID controllers,
- Implement basic kinematic routines, and
- Verify the performance of the system.

Preparation

Given the short fuse for the first lab, there is no prep-quiz. Instead, you will need to build your robot and set up your programming environment. There will be a "Post-Prep-Quiz," due Friday, which will be used to "backfill" some useful information about the robot.

2 Reference materials

The Romi Control Board. The Romi Control Board User's Guide lists the functionality of all of the pins and other useful information. You will want to bookmark or create easy access to the graphical pinout. Trust us.

Analog-to-digital Conversion. I assume you're familiar with ADCs; a video has been placed on Canvas if you'd like a refresher.

Tuning the speed controller for the Romi motors. A video has been placed on Canvas that explains why proportional control is insufficient by itself for motor speed control and how you can tune the controller gains.

Control background. A handout has been placed on Canvas that covers the strategy for controlling the motion of your Romi.

3 Library framework

The development framework that you will use for this course consists of a unified GitHub repository for the Romi, plus several utility libraries that your program will link to. The framework is developed using the `platformio` extension in `VSCode`. Within that framework, you will find the main code for your robot in the `src` directory. To help organize the code, libraries that you are expected to edit are found in the `lib` directory. Utility libraries, which you are *not* expected to edit will be included using the `lib_deps` feature of `platformio`.

Warning: In the libraries, a number of objects are defined with the keyword `extern`. By declaring them `extern`, certain methods are easier to access from *interrupt service routines* (ISRs). The downside is that you must name certain objects with the same name as declared `extern`. For example, your `Chassis` object is declared

```
    extern Chassis chassis;
```

and so any instantiation of `Chassis` must be called `chassis`. Same with

```
    extern IRDecoder decoder;
```

Many of the objects are already declared in the code, but you might find that an object is not yet instantiated. If you name it incorrectly, you will get a linker error – course staff can help you work through them, but a lot of headaches can be avoided if you **spend some time exploring the library**.

3.1 The control architecture

The control of the robot is accomplished primarily through two classes: The `Chassis` class manages the low level control of the robot, including the motor speed controllers. The `Robot` class integrates several components and sub-systems using a number of classes (`Chassis`, the `LSM6` IMU, etc.). It also manages the high-level behavior of the robot, such as commanding the robot to drive, turn, follow a line, and so forth. Most of your coding will involve editing or adding functions to the `Robot` class to implement the correct behaviors. The framework provides many placeholders and `TODOs` to help you keep your code organized. **The framework is written using a checker/handler paradigm**. You are expected to adhere to the structure that we have provided; failure to do so is the number one cause of lengthy trips into the rabbit hole.

Pre-lab Activities

The activities in this section are to be completed **before the start of lab. For this lab, each student must build and test their own Romi.**

4 Getting started

4.1 Assemble your robot

You will be provided with a Romi that is mostly pre-built. Refer to the Pololu Romi User's Guide for any additional assembly that may be needed. Your base Romi must match the build at the end of Section 4 of that document. Lab staff can assist with any soldering that may be needed.

4.2 Line follower array

If your robot does not have a 6-sensor reflectance sensor array on it, attach one now. A typical configuration will use two sensor elements separated roughly be the width of the tape, but you can connect it in lab. For now, just mount it with standoffs (3/4" preferred).

4.3 IR receiver

If your robot does not have an IR receiver on it, attach one now. While you can use a breadboard if you want, because the sensor is so simple, it's often easiest just to insert it directly into the sockets on your Romi. The `2x7` header in the middle is a good place to mount it, as you have easy access to `5V`, `GND`, and an interrupt pin. **You must connect the output pin of the sensor to an interrupt-enabled pin**. Don't forget to declare the pin where you construct the IRDecoder object in `robot-remote.cpp`.

5 Set up the programming environment

Follow the instructions from the Platform IO webpage to get set up. If you do not have `VSCode` installed on your computer, instructions for installing it can be found on that website, as well.

5.1 Clone the repository

You will be provided with an individual GitHub repository that will be unique to you. You will switch pairs each week, so it is imperative that you update your code with each week's activity, so that you are not at a disadvantage in the following week. When grading code, we will select randomly from the pair.

To clone your repository, you can use the command line, GitHub Desktop, or connect `VSCode` to your GitHub account. If you're new to `git` and GitHub, easiest is to navigate to your GitHub repo online and select "Open with GitHub Desktop" and follow instructions there.

Since GitHub serves as your cloud storage, you should put the code in a directory that is *not* managed by OneDrive (including your Documents folder).

5.2 Verify your configuration

Using `VSCode`, upload the skeleton code to your Romi. Open the Serial Monitor and point your IR remote at the Romi and press a number key. You should see a number (corresponding to a key code) printed on the screen.

Lab Activities

You will work in pairs for the week, but each student will update the parameters on their own Romi. You may want to explore the Live Share option in `VSCode`, but note that you will need to connect `VSCode` to `git` for that to work.

6 Motor control

Here, you will implement PI control and tune the control gains for the motor speed controller. By default, the proportional gain is set to $K_p = 1.0$ and the integral and derivative gains are zero in the `Romi32U4MotorBase` class. Further, neither integral nor derivative control is implemented in `Romi32U4MotorBase::ControlMotorSpeed()`. If you command the motors to spin, you will find that the performance is poor.

6.1 A note on batteries

To drive the motors, you will need to put batteries in your Romi. Note that the *only* component on the Romi that requires battery power are the drive motors, with the one asterisk that later in the term, the servo motor will work, but be extremely weak on USB power. *Everything else will run on USB power*.

Warning: Because you can work on the robot with the battery power off, **be careful when turning the power on**. If your robot is trying to command the motors with the battery power off, the integral term in your PI controller will build up and up, so that when you engage power, your robot will take off. Best is to get in a habit of holding the Romi up when engaging power or hitting reset so that you know the robot is in an `IDLE` state.

There are rechargeable batteries provided in the lab. **You are not to keep batteries in your robot when you are not using it**. At the end of the lab or when you finish with your robot any time outside of lab, **you must return your batteries to the battery bins**.

6.2 Encoders

The libraries that come with the Romi keep track of the encoder counts for you: whenever there is an encoder transition, the count is either incremented or decremented, depending on the direction the motor is spinning. You will not have to write code to interpret the encoder ticks, but you will have to show how the code interprets the encoder pulses. The User's Guide describes the encoders in more detail.

Unlike most quadrature encoders, where each channel is connected to its own encoder pin, Pololu wired the encoders in a clever way to conserve resources; namely, they've wired the two encoder channels together with an `XOR` gate. You do not have to write the code to interpret the signals, but here you will walk through how they work to "verify" that the code works correctly. The relevant functions from the library are given below. `^` is the *exclusive OR*, or `XOR`, operator.

The first thing to note is that the pins are read using the `FastGPIO` library. This library uses `C++` *templates* to read the pin hardware directly, as opposed to the (relatively) slow Arduino function, `digitalRead()`. Though the syntax may be confusing, the functionality is the same as calling `digitalRead()` on the appropriate pin.

Note that Channel B can be read directly from the pin state, but Channel A has to be decoded from the `XOR`'ed signal. The encoder count is then updated with an odd-looking – but fast – line that uses more `XOR` operators. Finally, the current state of each encoder is copied to a variable to be used next time the function is called.

The Worksheet asks you to walk through the calculations to show that `CW` and `CCW` rotations lead to incrementing and decrementing the `encoderCount`, respectively.

6.3 Tuning the speed controller

To implement motor control, find the `ControlMotorSpeed()` function in the `Romi32U4MotorBase` class and implement a PI-controller. Optionally, you can add a derivative term and also an open-loop[1] term. Read the rest of this section and then tune the gains until you get good performance.

If you uncomment the `__MOTOR_DEBUG__` flag in the `platformio.ini` file, the program will print useful information about the motor controller to the Serial Monitor (dig into the motor controller to see what exactly is being printed). It's hard to interpret the numbers as they are scrolling by, but if you install the Arduino programming environment, you can use the Serial *Plotter* in Arduino to visualize the data. You'll find it much easier to determine a good set of gains with a graphical interface. In the end, you'll want a quick response time, but too much overshoot and substantial oscillations are hard on the cheap, plastic gears in the motors.

Warning: You cannot simultaneously connect to the Romi with Arduino and `VSCode`, so it is a pain to change gains through repeated uploads. To make your life a *little* easier, we've added a `SETUP` mode to the Romi, which allows you to use your IR remote to change gains. See Appendix 15 for instructions on how to do that, and we've put a video on Canvas to demonstrate.

While you can use different gains for each motor, historically, one common set of gains is easiest and performs well. When you are finished tuning, you should update the default gains in the motor class definition. You might also want to add an *integral cap* to prevent the integral (which is a sum in discrete time) from building up too much. To find a good cap, use your value for K_I to figure out what value for the integral (sim) will correspond to the maximum motor effort. Add 20-30% extra to the cap so that you don't cap it too low. Be sure to limit the integral (sum) on the negative side, as well!

6.4 Speed control of the Romi

In the motor library, the motor speeds are measured and controlled in "encoder ticks per control interval." For many future activities, you will need to control both the forward speed and angular velocity of the Romi. Therefore, you will need to create a function that converts linear speed of the Romi axles to target motor speeds.

Edit the function, `Chassis::SetWheelSpeeds()`, to convert from target speeds in $\mathrm{cm/s}$ to motor

[1]You could also call it a "feedforward" term, but technically, there is a difference between the two.

speeds in $\text{encoder counts/interval}$. You will need to either do experiments or research to determine the kinematic parameters to convert motor speed to the equivalent linear motion.

Because we have not covered kinematics, we provide a function, `Robot::SetTwist()`, which converts robot motion (forward speed and angular velocity) to linear wheel speeds. The `ROBOT_RADIUS` defaults to 7.35cm, which is the datasheet value for the Romi radius and should work well. Later in the term, you may have a chance to adjust the parameter for better performance.

6.5 Unit Testing

Following the structure in `Robot::HandleKeyCode()`, add code so that pressing the `REWIND` button in `AUTO` mode will engage the line following behaviour at the commanded speed (specifically, call the `EnterLineFollowing()` function with a speed parameter by converting the `keyString` to and integer).

Command your robot to drive at 10cm/sec using the line following machinery (because you haven't yet implemented line following, the robot will simply drive more or less straight). Drive the robot on the measured meter and time how long it takes to verify that it is driving the prescribed speed. Qualitatively, note if the robot drifts. You may want to adjust the kinematic parameters for each wheel independently to get the Romi to drive straight (see the `Chassis` class for the available parameters).

7 Tuning the line follower

Your final task is to implement a "two-stage" line following controller: first, determine the "turning effort" needed to align the robot with the line, and then command the motors to drive at the proper speed.

7.1 Calculating the error

1. Edit `LineSensor::CalcError()` to calculate the *error* for the line sensor array. Ultimately, you'd like the error to be the deviation between the center of the robot and the center of the line. Given that the sensors are highly non-linear, however, you will have to use an approximation. There are a number of ways to approximate the error, with different levels of complexity and accuracy:

   Sensor element difference. The simplest is to merely take the difference between two sensor elements. If the sensors are arranged such that they are each near one of the edges, then at least for small deviations, the changes in "grey" that the sensors see are good enough.

   Approximate position with two sensors. As a sensor element moves across the edge of the tape, the response can be approximated by a non-linear function, for example a sigmoid curve, arctangent, or a hyperbolic tangent. You could take the sensor reading(s) and "invert" the response to calculate the position of the sensor with respect to the edge of the tape and then use that to calculate the error. Note that trigonometry is expensive on the `ATmega32U4`, but it's still an interesting approach.

   Multi-element. You could also connect multiple array elements to ADCs on your Romi. Then process all of the sensors to calculate a relative position of the array w.r.t. the tape. One way to do that is to take the *weighted average* of each sensor to find the deviation from the center of the tape:

$$\text{deviation} \approx \frac{\sum \sigma_i r_i}{\sum r_i}$$

where σ_i is a weight that is proportional to the distance between element i and the center of the array and r_i is the percent reflectance of that element.[2]

7.2 Following the line

Once you have a method for calculating the error, then you need to calculate the effort and add code to adjust the motion. To do that, add your control code to `Robot::LineFollow()`. At a minimum, include proportional control, but you might want to experiment with derivative control, particularly if you plan on fine tuning performance.

8 Verification

Place each Romi on the taped course provided in lab. Command each in turn to line follow at the speed of your choice (maybe start slow?). Time how long it takes to drive a lap and, using the distance written for the course on the white board, calculate the average speed. Repeat the experiment at five different speeds and record your results in the table in the Worksheet.

9 The Calvin Page "I'm not dead, yet!" ~~Memorial~~ Pursuit Challenge

If you'd like an extra challenge, enter the Page Pursuit Challenge![3] Tune your robot(s) to go as fast as they (reasonably) can and sign up for the challenge bracket, which will include other students and possibly lab staff.

10 Summary

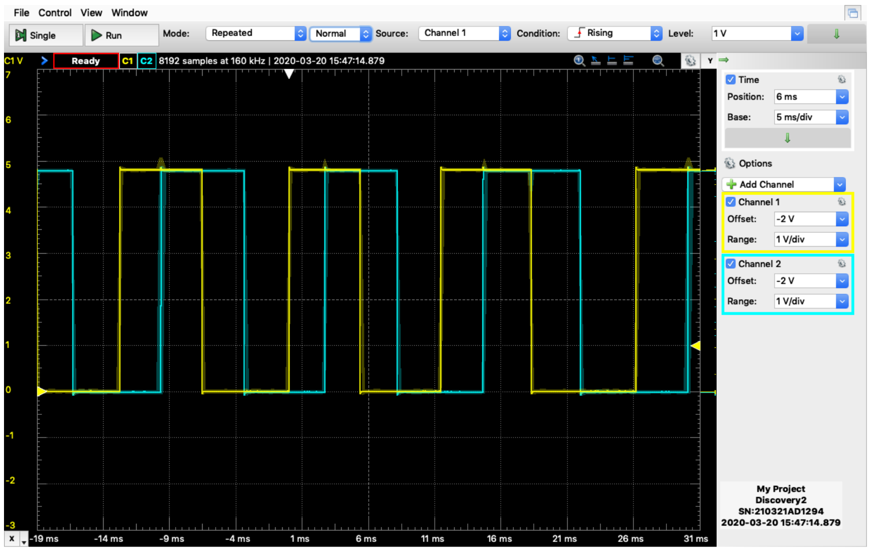
The basic set of tasks for the week consist of implementing PI control for motor speed, P control with two line sensors for line following, and the verification activities. Above and beyond work can include adding D control and demonstrating improved performance (moderate to high); implementing an advanced line error calculation (moderate to high); participating in the Pursuit Challenge (moderate); winning the Pursuit Challenge (high).

[2]The method works best if 0% is assigned to the darkest reflectivity and 100% for the brightest.
[3]Calvin Page was an SA last year and spent a lot of time encouraging students to take on extra challenges.

Worksheet

Turn in one Worksheet per pair, but note that in some places, both students are expected to record data.

11 Fundamentals



Mode: Repeated Normal Source: Channel 1 Condition: Rising Level: 1 V

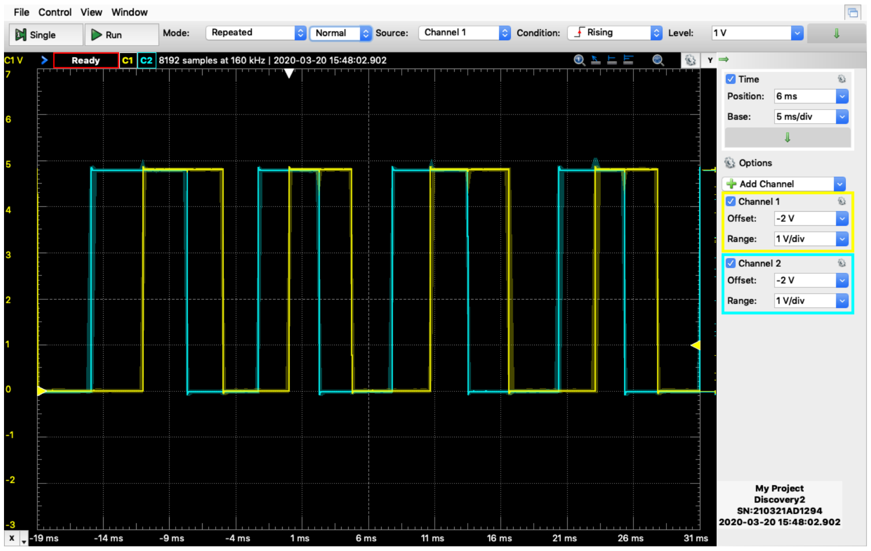
8192 samples at 160 kHz | 2020-03-20 15:47:14.879

Time
Position: 6 ms
Base: 5 ms/div

Channel 1
Offset: -2 V
Range: 1 V/div

Channel 2
Offset: -2 V
Range: 1 V/div

Figure 1: Oscilloscope capture of a quadrature encoder turning in the forward direction. Orange is Channel A. Blue is Channel B.



Mode: Repeated Normal Source: Channel 1 Condition: Rising Level: 1 V

8192 samples at 160 kHz | 2020-03-20 15:48:02.902

Time
Position: 6 ms
Base: 5 ms/div

Channel 1
Offset: -2 V
Range: 1 V/div

Channel 2
Offset: -2 V
Range: 1 V/div

Figure 2: Oscilloscope capture of a quadrature encoder turning in the reverse direction. Orange is Channel A. Blue is Channel B.

Question

Consider the encoder capture in Figure 1. Let's assume that the motor is moving "forward" in this capture, which will result in an increment of the encoder count with each transition. Referring to the ISR function (found in `Romi23U4Encoders.cpp`) for each of the transitions, show that the `ProcessEncoderTick()` will correctly increment the count.

The first two columns correspond to the transitions/states from the figure. The third column is the output of the `XOR` gate on the Control Board (which you can figure out from the previous columns). The remaining column names refer to the variables in `ProcessEncoderTick()`.

Repeat the process for the reverse direction in Figure 2.

A	B	A^B	lastA	newA	lastB	newB	newA^lastB	lastA^newB	ΔencCount
↑	H								
↓	H								
↑	L								
↓	L								
H	↑								
H	↓								
L	↑								
L	↓								

Question

With all the extra processing needed, why does Pololu use an `XOR` gate in their encoder circuit?

Question

Describe the technology behind the encoders on your Romi. What physics are they based on (e.g., light or magnetism or something else)?

Question

Do you think the encoders use some kind of *hysteresis*?

Question

Describe your motor control algorithm. What values did you use for gains? Did you cap the integral? Include a snapshot of the motor response to a step change in target speed from $20 \to 60$. If you included derivative control or an open-loop term, can you demonstrate improved performance? Include another snapshot that shows the improved performance.

Question

Describe the method you used to calculate the line following *error*. Add mathematical expressions to define precisely how the error is calculated.

Question

For your line following control, did you add a derivative term to your effort calculations? Include a mathematical expression used to calculate the *effort*.

Question

Add *clickable* links to your GitHub repos.

Question

Add a *clickable* link to a video of both robots driving the course at $20\mathrm{cm/s}$. Start them on opposite sides of the track ("pursuit style").

Question

Complete the table below, one for each robot.

Target speed	Measured speed (Robot A)	Measured speed (Robot B)

Comment on the performance of your Romis.

12 Extra Activities

12.1 Control

Question

Extra effort for control can include adding derivative control for speed control and/or line following; adding an open-loop term for speed control (moderate); using an advanced technique for calculating the line deviation (anything beyond simple subtraction of the reflectance sensor elements; or anything else that you think shows an extra effort. Though you presented your methods above, make a note here so that we have an easy-to-find record. Do not include evidence here – that should be presented in the answers above. We're only looking for a short bullet list in this response.

12.2 Pursuit Challenge

Question

If you entered the Pursuit Challenge, describe what you did to improve the performance of your robot. Adjustments can include changes to the code or physical system. How did it do in the pursuit? Marks here are *not* dependent on how well your robot did, but whether or not you can describe an earnest attempt to improve your robot. With the one exception that the overall winner will be given an extra bonus.

13 Wrapping up

Question

Team Member Contributions

1. Independently, describe how you contributed to the design tasks for this week's activities.

   Student A ______________________________

   Student B ______________________________

2. Assign a weight (as a percentage) to each team member's contribution.

Student	Percentage

14 Switching modes with the IR remote

To help you tune and test your Romi, we have built three separate modes into the Romi framework:

TELEOP, which allows the user to command basic drive motions to prove the motors.

AUTO, which is the main way to accomplish autonomous tasks (e.g., line following in this week's activities), and

SETUP, which is arguably a little cumbersome, but it allows you to change gains without re-uploading code over and over.

Figure 3 shows how to switch between modes.

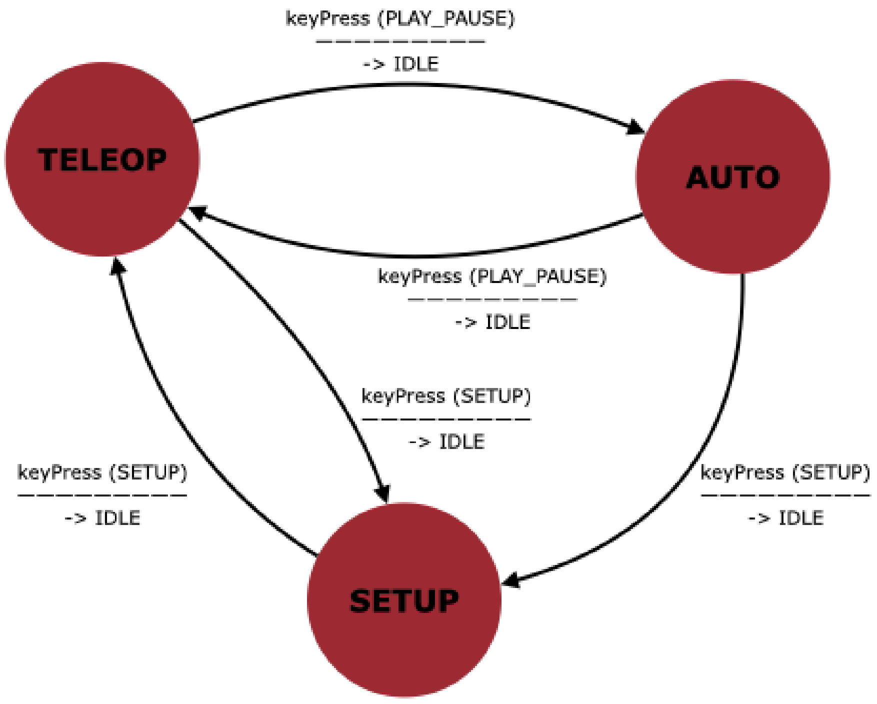


Figure 3: State transition diagram for switching modes with the IR remote.

By default, the code starts up in TELEOP. The `PLAY_PAUSE` button is used to switch between TELEOP and AUTO, while the `SETUP` button is used to enter and exit SETUP mode. Note that other buttons (e.g., the volume buttons) will have different effects in different modes! You should consult `Robot::HandleKeyCode()` to determine all of the functionality.

15 Using the Arduino Serial Plotter to tune gains

By default, the tuning methods use only the left motor, which has two benefits: it's easier to observe just the one motor while tuning, and you can also set the Romi on its right side while tuning. You could tune the right wheel separately, but historically, commons gains for the two motors work just fine.

To tune the gains, upload the default code to the Romi. Make sure you're not connected to the Romi with the Serial Monitor in `VSCode`. Open Arduino and select the port that the Romi is connected to and "Leonardo" for a board type (why?). Then open the Serial Plotter, which will draw lines for the target speed, actual speed, and effort (divided by 10 for better visuals).

Figure 4 shows the commands in `SETUP` mode. When numbers are pressed, digits will "accumulate" in a string and then pressing one of the gains keys will update the gain accordingly. **Note that the code divides the entered string by 100 so you can have fractional gains.** So "100" becomes 1.00.

Pressing the up and down arrows will cause a step change in the target speed. Pressing `ENTER_SAVE` will clear any digits and stop the motor.

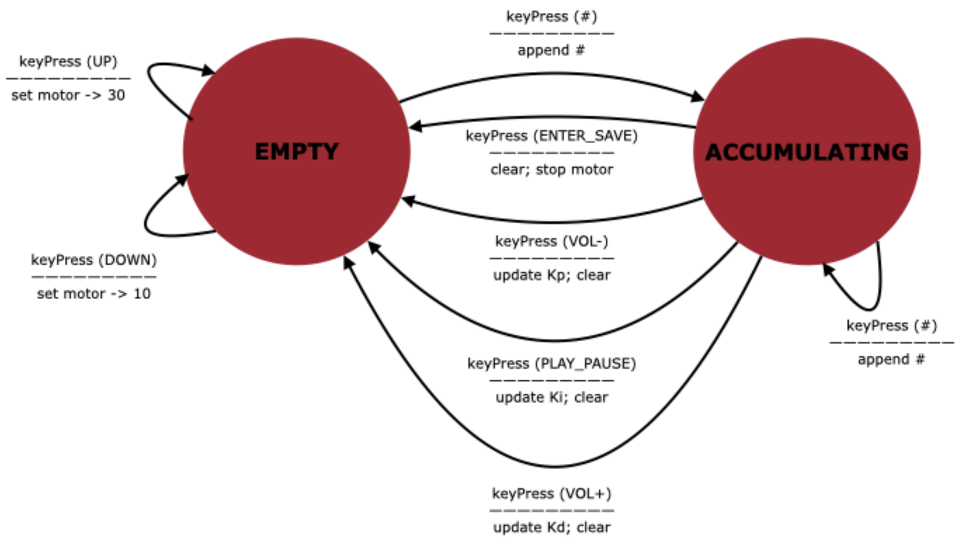


Figure 4: State transition diagram for tuning gains in `SETUP` mode.

To save program space, you can comment out the tuning portions of `Robot::HandleKeyCode()` when you're done. You can probably remove the entire `SETUP` mode.