WPI

NSF Workshop
Move It!

Last modification: January 3, 2022

Workshop Note:
The document here includes several tasks that a student in a course would do – more than we have time for in the actual Workshop. For those who want to work through the preliminary activities, we provide some skeleton code, the instructions below, and nominal solutions.
In the Workshop, we will start in Section 6 and do the experiments needed to calibrate the robot's motion. If you choose not to do the preliminary activities, you can start with the solution code that we provide.

1 Introduction

In this activity, you will implement basic driving functionality for your robot. Specifically, after completing the activity you will be able to command your robot to drive or turn a specified distance by sending signals from an IR remote. You will not only program the robot to respond to commands, but calibrate the robot's *kinematic parameters* so that it can accomplish accurate motions.

1.1 Objectives

By the end of the exercise, the student will have:

- Implemented a state machine to control basic motion of a robot,
- Calibrated robot parameters for the purpose of dead reckoning, and
- Performed tests to verify performance.

1.2 Problem statement

Your goal is to program your Romi such that when you press one of the arrow keys on your IR remote, the Romi will drive or turn a specified amount, after which it will come to a stop. The Romi must resume driving in response to subsequent button presses. In addition, it must respond to a designated "emergency stop" button by stopping and entering an idle state.

2 Preparation

2.1 Resources

- You should understand the basic concepts of bits and bytes, as well as variable types. This Arduino tutorial has a good overview.
- How an IR remote works. You do not need to be an expert in IR technology, but we will use IR signals as a way to demonstrate how hardware is designed for selectivity.
  - Very high level overview.
  - More detailed explanation of how signals are encoded. The section titled, "The NEC Code" is the most relevant section.
- As part of the Pre-lab activities, you will do some research on kinematics of a differential drive robot.

2.2 Materials

- Your Romi,
- The IR receiver,
- The IR remote,
- A ruler for measuring distances, and
- A protractor or some graph paper for measuring angles.

3 Commanding your Romi

The first function we'll tackle is enabling the Romi to respond to user commands. In the introductory activity ("Hello, World!"), you used the pushbutton on the Romi to detect input from the user. The pushbutton wouldn't be terribly practical for a mobile robot, however, so we'll add a wireless solution, specifically an IR remote paired with an IR receiver.

3.1 IR remote control

We don't go into too much detail about how an IR remote works here, but we'll give enough of an overview to get you started.

When you push a key on an IR remote, it sends a unique code for that key. Depending on the remote, the code typically contains between 8 and 32 *bits* of information, where each bit is a 1 or a 0. The bits are not determined by a simple turning the IR emitter on or off, however. The IR remote uses a *modulated* signal to send information. In this case, the IR light is toggled on and off at 38 kHz. The information – 1's and 0's – are actually encoded by the *length of time between bursts* of the 38 kHz modulated signal.

The links in Section 2.2 explains the process in more detail.

The Vishay IR receiver included with your Romi detects the IR bursts and converts the signal to a series of high and low voltages – voltages that are easily detected by microcontroller. The receiver is resilient to ambient IR radiation – you want to be able to control your TV under any lighting conditions – because it uses a number of techniques to *filter* the incoming signal. In addition to a physical IR filter,[1] the receiver is tuned only to look for modulated signals near 38 kHz. Doing so makes the receiver robust against ambient conditions.

We have simplified the use of the IR remote by encapsulating the functionality into a library that runs in the background of your program. All you need to do is initialize the IR receiver routines and periodically check for new IR codes. The example code demonstrates how to do this.

Procedure

1. When you constructed your Romi, you should have added the IR receiver to a breadboard circuit. If you haven't completed the build, do so now.
2. Open the `Move It!` skeleton program, which can be found in the sample codes.

   > **Warning!** With `platformio`, you must open folders that contains a `platformio.ini` file. If you open a folder one directory above that, you won't see the compile and upload buttons. If that happens, select `File->Close Folder` and open the correct folder.
   > In this case, you need to open the folder titled, `act01-skeleton`.

3. Upload the program (as is) and open the Serial Monitor.
4. Point your IR remote at the receiver and press some buttons. What do you see in the Serial Monitor? What happens when you hold a button down?
5. In the line where the program checks for a new key press, edit the code to pass `true` as an argument to the `irDecoder.getCode()` function call. What happens now when you hold down a key?

   You should be able to hold down `CTRL` (`command` on a mac) on your keyboard and click on the function, at which point `platformio` will take you to the function definition. You don't have to worry about the details, but you can see that that passing `true` to the function will allow it to accept repeat codes. By default, the parameter is set to `false`, which means it will ignore repeats – to send the same code twice, you have to release the button and push it again.

   Being able to click on functions to go to their definitions is a very useful feature of `platformio`.
6. From experience, you will probably not want to respond to repeat codes, so change the function call back to how it was when you started.

3.2 Object-oriented programming

In `C++`, code can be organized into *objects*, hence the term "object-oriented programming." Though not necessary, objects in code often correspond to objects in the real world, for example a motor

[1] This one is centered on 950 nm.

or a particular sensor. Each object has information about itself and a set of functions, usually called *methods*, that are used to interact with it.

In this case, the IR receiver is treated as an object, but the code that manages the receiver does much more than detect the voltage created by the sensor – it keeps track of all the bit patterns, decodes them, and provides the latest result when asked.

The standard syntax for calling a member function, often called a *method*, is,

```
<object>.<method>(<arguments>);
```

where each method may take zero to several *arguments*.

Some questions:

1. What line of code is used to define the `IRDecoder` object? What does it take as an argument?

   Solution: The `IRDecoder` is declared on line 15 and takes the pin that the receiver is connected to as an argument.

2. What line is used to *initialize* the decoder?

   Solution: It is initialized in the `setup()` function by calling `decoder.init();`

3. What code is printed when you press the up arrow button on your remote?

   Solution: The up arrow has a code value of 5.

4. What code is printed when you press the `1` button? What about the `2`?

   Solution: Button 1 has a code of 16. Button 2 is 17.

We have provided another file, `ir_codes.h`, that maps the codes from the IR remote to a meaningful name to use in your code.

Now that you have the remote working, let's use it to command the Romi.

Procedure

1. Edit the `handleKeyPress()` function to implement the basic state machine in Figure 1.

   - If the `ENTER_SAVE` button is pressed, the robot goes to the `ROBOT_IDLE` state.

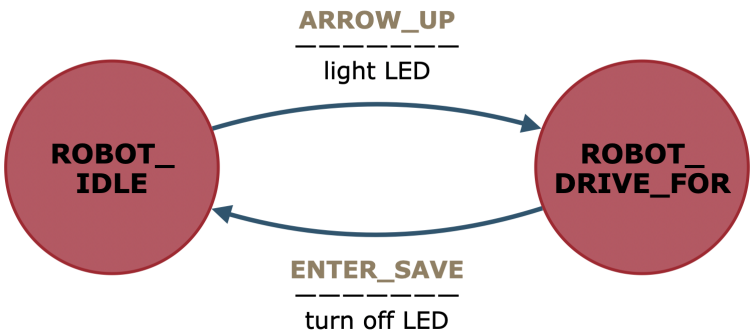

Figure 1: Basic state machine.

Solution:
Add the following to `handleKeyPress()`, before the `switch` statement.

```
//ENTER_SAVE idles, regardless of state -- E-stop
if(keyPress == ENTER_SAVE) idle();
```

- If the robot is in the `ROBOT_IDLE` state and the `ARROW_UP` button is pressed, call `drive()`. For now, `drive()` just lights the LED and changes the state to `ROBOT_DRIVE_FOR` (we'll edit the definition of `drive()` and actually make the Romi drive below).

Solution:
You should have something like,

```
switch(robotState)
{
  case ROBOT_IDLE:
    if(keyPress == UP_ARROW) drive();
    break;
```

2. Run the code and open the Serial Monitor. Are you able to switch states? How can you tell?

Solution: The LED should turn on and off (a basic debugging feature).

4 Let's go!

The next function we'll tackle is driving, a core function of your Romi – there isn't a whole lot you can do without your Romi being able to move around. Fundamentally, the two drive motors provide the power to drive around, and they are supplemented by wheels, gears, brackets, even the Romi body

to *provide mobility*. Here, we'll focus on the motors and the *encoders*, sensors that are connected to the motors and used to monitor the motor's motion.

4.1 Motors

The motors that come with your robot are *brushed DC motors*. A brushed DC motor – often referred to simply as a "DC motor," even though there are many other types of DC motors – has a set of fixed magnets and a rotor that acts as an electromagnet to induce rotational motion. The term "brushed" refers to the electrical contacts that allow electrical current to be switched back and forth over the course of each rotation.

Attached to the motors are special sensors called *encoders*, which are attached to a motor shaft and allow a microcontroller to track the motor's rotation. The small disk that is attached to the motor shaft on each motor contains a number a magnets, and there are sensors that detect the magnets as they pass by. By keeping track of the number of passes, the microcontroller can use the encoders to determine the position of the motors, and from that determine the speed.

The library for the Romi has functions that allow you to set the target speed of the motors, read the actual speed, determine the absolute position, and several others. Here, you will explore the Romi motor interface.

4.2 Speed control

You'll start by controlling the speed of the motors, which will allow you to explore how *object-oriented programming* can be used to simplify code.

Here, you'll add code that will make the motors move when you press the button on your IR remote. Though the LED will turn on and off with only the USB cable plugged in, **the motors will not run unless you have batteries in your Romi and the power turned on**. There are both a button and a switch in the back left corner of the Romi Control Board – the switch has been known to break, so we recommend just using the button to turn the Romi on and off.

Procedure

1. *Gently* turn the wheels back and forth. They should move freely, though you will hear the gears engaging. If the motors are hard to turn or you hear clicking, the magnetic encoder disks may be pressed on too far. For any stuck motor, *gently* pry off the small brown disk on the side of the motor and then push it back on. It should be pressed on such that the outside edge is just flush with the end of the motor shaft.
2. **Set your Romi on a box so it doesn't drive off on you.** We're just showing that we can get the motors going here, and we don't want your Romi to do a gainer off the table!
3. At the top of the `main.cpp` file, add the line,

   ```
   #include <Chassis.h>
   ```

   which will tell the compiler to include the files that define all of the classes for controlling the motors.

4. Create a `Chassis` object. If you open the `Chassis.h`, you will see that the constructor for `Chassis` takes three arguments:

   - The wheel diameter,
   - The number of encoder ticks per wheel rotation, and
   - The wheel track, which is the distance between the two drive wheels.

   The nominal values for these parameters can be found in the Romi User's Guide.

   Solution: Good starting values to use are,

   ```
   Chassis chassis(7.0, 1440, 14.7);
   ```

   We will adjust these later.

5. To initialize the `chassis` object, add the following to the `setup()` routine:

   ```
   Chassis.init();
   ```

6. Uncomment the line in `idle()` that calls `chassis.idle()`.

7. Add the following line of code to the `drive()` function (which is still called in responds to the `ARROW_UP` key):

   ```
   chassis.setWheelSpeeds(60, 0);
   ```

8. Upload the code. Press the `ARROW_UP` button on your IR remote. If all goes well, the left wheel should start spinning slowly (what are the units on the argument again?). The wheel should stop when you press `ENTER_SAVE`.

   In the Romi library, motors are objects. Each motor keeps track of its own speed and position, among other things, and there are functions that can be used to command the motor to go a certain speed or move to a particular position.

   The nice part is that with the object-oriented approach, each object contains its own data and is independent of other objects. So, setting the speed for the left motor does not affect the right motor. To demonstrate, do the following:

9. Add a line of code to command `right_motor` to turn *twice as fast* as the left motor. Look at the definition of `setWheelSpeeds()` to see what each arguement does.

10. Upload and run your code. Did the wheels spin at different speeds?

11. Try commanding the right wheel to turn in the opposite direction.

Figure 2: Control loop for motor speed control.

5 Tracking motion and dead reckoning

Controlling the motion of your robot with the IR remote is useful, but eventually, you'll want the Romi to drive autonomously. We'll cover line following in the next activity, but here we'll get started with *dead reckoning*. With dead reckoning, the Romi estimates its motion (position, speed, orientation) with only *proprioceptive* sensors.[2] Imagine closing your eyes and walking across the room. How well do you think you could keep track of your location?

Estimating the Romi's complete *pose*, the position and orientation of the robot, is a subject for more advanced courses. Here, we'll concentrate on driving a specified (straight) distance or turning a specified angle. To do this, the program will monitor the wheel motion as the Romi drives (in the `ROBOT_DRIVE_FOR` state).

5.1 Setting PID control parameters

The library has a speed controller built into the `Motor` class. The controller is designed to use proportional-integral-derivative (PID) control adjust the power sent to the motors so that they turn at the designated speed. By default, only the proportional control is enabled, which will result in a poor performance. To maintain a prescribed speed, you will need to tune the control parameters.

Figure 2 shows a standard control loop for a motor. Starting with a target speed, the control loop determines the error between the target and actual speeds. A control law is used to convert the error to an effort – the voltage sent to the motor, which results in motion. Encoders are used to register the actual motor speed, which is fed back through the control loop, which is why this type of control is called feedback control.

Procedure

1. In the `setup()` function, add a line of code right after `chassis.init()` to call,

   ```
   chassis.setMotorPIDcoeffs(1, 0);
   ```

2. In the `drive()` function, command both wheels to spin at 180 degrees per second.

3. Adjust proportional by increasing the first value, which is the proportional control gain, by increments of 1, each time uploading the new code to check performance. Gently grab a wheel to see how it reacts – once the motors start vibrating a little, back the value off a bit.

   If you time how fast the wheels are going, you will see that they are not spinning at 180 degrees per second. To get them to track speed, you need to add integral control.

4. Add integral control by adjusting the second argument in increments of 0.1. Again, if you get oscillations, back the value off.

[2] Proprioceptive refers to sensors that detect internal properties.

Solution: Though there is no guarantee the values will work for your motors, we have found that a good solution is,

```
chassis.setPIDCoeffients(5, 0.5);
```

6 Calibrating the robot motion

Here, you will use library functions to command your robot to drive straight a certain distance and turn a specified angle. You'll do some tests to *calibrate* the kinematic parameters so that the specified motion matches the actual motion.

6.1 Calling library functions

The library provides a method for driving a set distance and another for turning a specified number of degrees. We have also included a function, `Chassis::checkMotionComplete()` that determines if the commanded motion is finished.

Procedure

1. Update the `drive()` function definition to take two arguments: a distance and a speed. What data types will you use?

   Solution: A good solution looks like:
   ```
   void driveFor(float distance, float speed)
   ```

2. Remove the line that previously commanded the motors to spin and replace it with a line,

   ```
   chassis.driveFor(50, 10);
   ```

   How far and how fast will the wheel turn? (Hint: Click into the function definition.)

   Solution: `chassis.moveFor()` takes a distance in cm and a speed in cm/sec as arguments.[a]

   [a] `chassis.moveFor()` has code to ensure that the distance and speed are in the same direction.

3. Add code to check if the motion is complete and handle the event.

   - In the state machine in the `loop()` function, uncomment the line that calls `chassis.checkMotionComplete()`.

> **Workshop Note:** We'll discuss event handling in greater detail in the next activity when we'll program the robots to detect intersections. Here, the checker is part of the library, so we've written that for you.

- Write a new function, `handleMotionComplete()`, that simply calls `idle()`. Note that we don't call `idle()` directly from the state machine because we'll add more complex behavior in later activities and we don't want to clutter the state machine.

4. Run the code and open the Serial Monitor. Press the `ARROW_UP` button. Does the wheel turn for a bit and then stop? What happens if you press the `ENTER_SAVE` button while the wheel is turning? Does it stop?

5. Add code so that the `ARROW_DOWN` button commands the Romi to drive backwards the same amount.

6. Add code so that the left and right arrow buttons on the remote command the Romi to turn left and right by 90 degrees. We have provided you with a skeleton `turn()` function, which you can fill in using `chassis.turnFor()`.

> **Solution:** A good solution looks like:
>
> ```
> // A helper function to turn a set angle
> void turn(float ang, float speed)
> {
>   Serial.println("turn()");
>   setLED(HIGH);
>   chassis.turnFor(ang, speed);
>   robotState = ROBOT_DRIVE_FOR;
> }
> ```
>
> You will also need to augment the state machine in `handleKeyPress()`.

6.2 Adjusting the kinematic parameters

In Section 4.2, you entered nominal dimensions for you robot. Unfortunately, those value won't match the actual dimensions, which will cause your robot to drive (or turn) the wrong amount. Later on, you will need to execute accurate motions so you can pick up and drop off containers.

Here, you will adjust the wheel diameter and wheel track so that when you command your Romi to drive 50 cm, it drives the correct amount.

Procedure

1. Using a tape measure, mark out 50 cm. Set your Romi at one end and command it to drive. Did it go exactly 50 cm?

2. Adjust the wheel diameter so that the Romi will drive 50 cm.
3. Now command the Romi to turn 90 degrees. Did it over- or undershoot?
4. Adjust the wheel track until the turn is 90 degrees.

Workshop Note: We usually give the students less guidance here. Given the equations that they found in the Pre-lab, they should be able to figure out how each parameter affects how the motion is calculated.
Note that you can change the wheel diameter or encoder counts / revolution – the two are always multiplied together.
Note that wheel diameter/encoder counts affects both distance and turn angle, but wheel track only affect turning. So they should determine the former based on driving straight and then 'lock' it in place.
Note that if you increase the wheel diameter, the robot will drive a *shorter* distance. This is because the larger diameter will make the robot "think" it is driving farther, so it will stop sooner – it's trying to go 50 cm.

6.3 Testing

Here you will perform some experiments to test your system performance.

Workshop Note: In a class, testing can be a lot more in depth, if desired. We often have the students program the IR remote such that different buttons correspond to different distances (e.g., 1 -> 10 cm; 2 -> 20 cm; etc.). In the interest of time, for this workshop, we'll just do one set of experiments.

Procedure

1. For each of the distances in Table 1, each student will command their robot to drive that distance for 10 trials. Using a ruler, measure the actual distance travelled for each trial and calculate the average and the standard deviation of the actual motion for each set of trials.

Solution: Good results have less than 1 cm of error, which is a good target since the tolerance of the lifting hook is about that much.

7 Worksheet

1. Describe how you could use `setWheelSpeeds()` and a timer to drive a prescribed distance.

   Solution: Knowing how far you want to drive, calculate how long you need to drive at a given speed. Start a timer for the correct amount of time.
   (We generally don't use this kind of method because it's not as precise as `driveFor()`, which tracks the actual motion of the wheels.

2. Describe a reasonable requirement for how accurately your Romi needs to be able to drive a specified distance. How did you come up with your values?

   Solution: One reasonable argument is that the hook can tolerate about 1 cm error, so that is a good threshold.

3. Describe how you calibrated your Romi to drive a given distance. Was your formula from the pre-lab accurate?

   Solution: Your formula was probably close but not perfect. One easy way to calibrate is to command a particular distance and then measure the actual and scale by the ratio.

4. Record the results of your tests in Table 1. Record each set of trials as an average $\pm$ the standard deviation, which is a representation of the reliability. For example, 19.5 ± 0.8. Note: if you don't have access to a ruler, you can use multiples of a standard 8.5" sheet of paper, which is 21.6 cm wide and do your best to estimate the true distance.

Distance	Student 1	Student 2	Student 3	Student 4
20 cm				
40 cm				
60 cm				
80 cm				
100 cm				

Table 1: **Sample worksheet**. Record your results as average $\pm$ standard deviation. Use as many columns as needed for your team.

   Solution: Data will vary. Less than 1 cm standard deviation is expected.

5. Did your robot performance meet the requirements? Explain.

   Solution: There may be cases where it didn't perform; likely the longer distances. It will be important to keep this in mind when your robot is driving autonomously.

6. In this activity, you used dead reckoning with the encoders to track how far the robot drove. List five other ways you could keep track of the motion of a wheeled vehicle.

Solution: Solutions will vary, but may include:

- Ultrasonic rangefinder
- GPS
- Radar
- LiDAR
- Camera
- Optical flow
- An IMU can be used, but calibration is difficult
- ...

8 Pre-lab worksheet

This is an individual assignment.

1. Do some research (or use your understanding of geometry) to develop formulas for how far a robot will drive or turn, given a wheel motion.

   For the first formula, assume the wheels are turning the same amount (i.e., the robot is driving straight). For the second, assume *for now* that the wheels turn the same amount, but in opposite directions.

   - The distance a robot will travel, given a wheel diameter and a wheel rotation (in degrees), and

     Solution: From the definition of a radian, the distance (length), L, that the robot will travel for a given wheel rotation is just the radius of the wheel times the angle, θ – **in radians** – that it turned.

     $$L = \theta(rad) \cdot \frac{d}{2} = \boxed{\theta(deg) \cdot \frac{\pi}{180} \cdot \frac{d}{2}} \tag{1}$$

   - The angle a robot will turn, given a wheel diameter, wheel track, and a wheel rotation (in degrees).

     Solution: Having established how far the wheel moves for a given rotation, we now consider the case where they're moving in opposite directions. In this case, each wheel will roll L in Equation 1. But L is just the length of an arc on a circle with a radius that is half the wheel track, which we'll call $D/2$. Then the angle, ϕ, subtended is just,

     $$\phi(rad) = \frac{L}{D/2}$$

     Converting to degrees gives,

     $$\phi(deg) = \frac{L}{D/2} \cdot \frac{180}{\pi} = \frac{360L}{\pi D}$$

     Finally, substituting for L, gives,

     $$\phi(deg) = \frac{360L}{\pi D} = \frac{360}{\pi D} \cdot \theta(deg) \cdot \frac{\pi}{180} \cdot \frac{d}{2} = \boxed{\frac{d}{D}\theta(deg)}$$