How To Build a ROS Robot

SDSMT Robotics Team

Ian Carlson

August 10, 2015

Contents

List of Figures

1

Author Info

You can contact me at carlson_ian@hotmail.com if you have any questions, comments, or concerns about this document. This includes errors, necessary updates, or things you just plain didn't like.

2

Introduction

2.1 Purpose

Building a sophisticated robot can be a complex and potentially daunting task. While a single afternoon can be sufficient to strap a few servos and an arduino to a chassis, doing anything interesting required an investment of both time and resources. As the cost of a robot increases, it becomes more important to have a solid plan to follow. This document attempts to outline the major phases of robotics development, as well as providing the basic information required to build a basic robot featuring ROS. It should be noted that this document is entirely a product of my own experiences - mostly the failures - during my time on the Robotics team, in the hope that others will not repeat them. That being said, this document is neither expected to be entirely complete nor perfectly accurate. It is a certain truth that failure is a better teacher than success, but experience is the best teacher of all - preferably someone else's.

The major sections are as follows:

- Mechanical Design - Basic mobile robot kinematics and design principles.
- Electrical Design - How not start your robot on fire. Probably.
- Setting Up The Odroid - How to go from a fresh Odroid to one running Fedora 22 and ROS
- ROS Architecture - The basic design for a remote controlled robot in ROS

This guide should be delivered alongside its companion document - Robot Paper Template - and potentially the example I created using that template - Mecanum and SMP Design Document. All of the code referenced in this document is available on the SDSMT Robotics Team GitHub pages - SMD-ROS-DEVEL and SMD-ROBOTICS.

3

Mechanical Design

3.1 Design Considerations

The most important aspect of your robot is ensuring that it is mechanically capable of fulfilling requirements - and making sure it doesn't collapse under its own weight. No amount of sophisticated code, nor the most powerful computer in the world, can force your robot to function if the chassis is compromised. It is tempting - we fell in to this trap several times during my time on the Robotics Team - to want to design your perfect robot from the ground up. Don't. If this is your first robot, or even second or third, take my word for it and just buy a chassis.

Seriously.

Mechanical design is time-consuming and, unless you have several skilled mechanical designers around, likely going to be difficult. On the Mines campus, it has become increasingly difficult to do any real fabrication, thanks in part to the increasingly strict attention to safety, and the CAT lab's understandable desire to cut down on machining costs.

Going for it anyway? Good for you, however, keep a few things in mind:

First, keep it simple. Complex designs are more expensive, have tighter tolerances, take a great deal longer to design and fabricate, and - most importantly - have more ways to fail. The best design in the world is useless if it never makes it off paper.

Second, keep moving parts to a minimum. This is similar to point one, but it's important enough to say twice. Every moving part means you need to deal with routing power i.e. keeping cables from getting tangled. Every moving piece also generally adds weight, cost, and complexity.

Third, if the words "we'll attach this here somehow" are ever uttered, take a closer look. We like to think in the CSC department that Mechanical Engineering is just hitting things with hammers and turning wrenches, but building real things is actually very difficult. Every piece of your robot has to be attached somehow - avoid glue - and that's not something that just sorts itself out once you have all the pieces.

Fourth - No, hot glue is not a good idea. Trust me. Been there. It's not. Epoxy can work, but it's no substitute for bolts.

Fifth - 3M velcro tape is better, but still not good, and is only appropriate for an indoor non-competition research vehicle where frequent reconfiguration is expected. See the SMP and Mecanum chassis for examples. It doesn't hold up under vibration as well as you would think. It holds great against normal stress, but not terribly well against sheer stress, and not at all against twisting stress.

Sixth, iterate. Don't expect your first build to be perfect. In all honesty, it probably won't even work. You forgot something or overlooked something or assumed something that isn't right. I promise. Give me a call before you start designing, and I will bet you $100. Expect your design to fail, and get something built as early as possible. A failed design isn't the end of the world unless it's the day of competition. More than one of our designs failed because we were half-way into the spring semester before we got anything built, and there was no time to try again.

Seventh, failure is good. You learn a whole lot from mistakes. Don't be afraid to try something just because it might not work. It *definitely* won't work the first time anyway.

3.2 Requirements

What does your robot need to do? Agile development is great for software, and there are some applications that can be adapted to hardware, but you really need to know what you're getting into before you start buying or building things. I can't possibly predict every possible requirement, but here are a few important ones I have come across before.

How does the robot need to move? Whether mobile or fixed, your robot likely needs to actuate within its workspace, and this will be one of the main factors in the mechanical design.This will be addressed further in section 3.3.

How fast does the robot need to be? If your robot doesn't need to be very fast, you can reduce power requirements, use smaller and/or cheaper motors, and you don't have to worry as much about vibrations. If the robot needs to be fast, the inverse is true.

How strong does the robot need to be? If the robot only needs to drive from A to B on a card table carrying a few paperclips, an Arduino taped to a few continuous rotation servos and some cardboard will probably do the trick. If you need to carry a cinderblock around an outdoor obstacle course, you need a robust frame, bigger motors, bigger batteries, and space to put it all.

How durable does the robot need to be? This one is pretty self-explanatory. However, keep in mind that as soon as you have a working RC robot, people will want to play with it. They will crash it. Hard. Once the robot is driving autonomously, it will crash itself. Often *and* hard. Make sure all potential contact surfaces on the robot can survive impact at maximum speed with something sharp and/or hard.

Does it need to be waterproof or water-resistant? Some competitions - like the Sparkfun AVC - are outdoors. Given Murphy's law, this means it will probably rain. Plan early about ways to keep water out and the magic smoke in. Never put holes in the top, and consider rubber grommets or seals around connectors and cabling. Also consider having angled non-porous surfaces on the top of the robot to deflect water such that it will drip harmlessly away from the electronics.

Does it need to be heat-tolerant? This, unfortunately, is in direct conflict with being waterproof, because you need to put holes in the robot for air circulation. We never came up with a perfect solution that handles both. Assume your CPU, GPU (if you have one), motor controller, and motors will attempt to melt anything in contact with them as soon as you close the case. Make sure you have proper heat-sinking and, if possible, forced air-flow over those heat-sinks. Also make sure the air has a way to escape. If operating outdoors, make sure you have filtered air intake.

Finally, what does it need to do? Claws, grippers, baskets, sensors, and payloads all need secure mounting. Make sure you include some sort of mounting bracket for whatever tools your robot will need to complete its job.

3.3 Kinematic Models

Now that you know what your robot needs to do, it's time to actually start designing something. For a mobile robot, kinematics are everything. There are a few common kinematic models that I want to cover in some detail, including the math. It is very important to understand - at least conceptually - the mathematics behind robot motion if you want to have any hope for autonomy.

Each kinematic model has different advantages and disadvantages, so it is important to consider how your robot needs to be able to move before selecting one.

3.3.1 Differential Drive

The simplest and most common drive system for a mobile robot is differential drive. This involves a fixed axle with two independently driven motors at the ends. A picture is worth a thousand words, so check out Figure 3.3.1, borrowed from the nice people at www.robotplatform.com.

Differential Drive is very simple because it only requires two powered wheels. In the image, the front wheel is a caster wheel, or a skid-pad, and is only present to keep the robot upright.

One major advantage of the differential drive paradigm is the ability to turn in place by spinning one wheel forward, and one back. However, because of the skid-pad or caster wheel, differential drive is typically

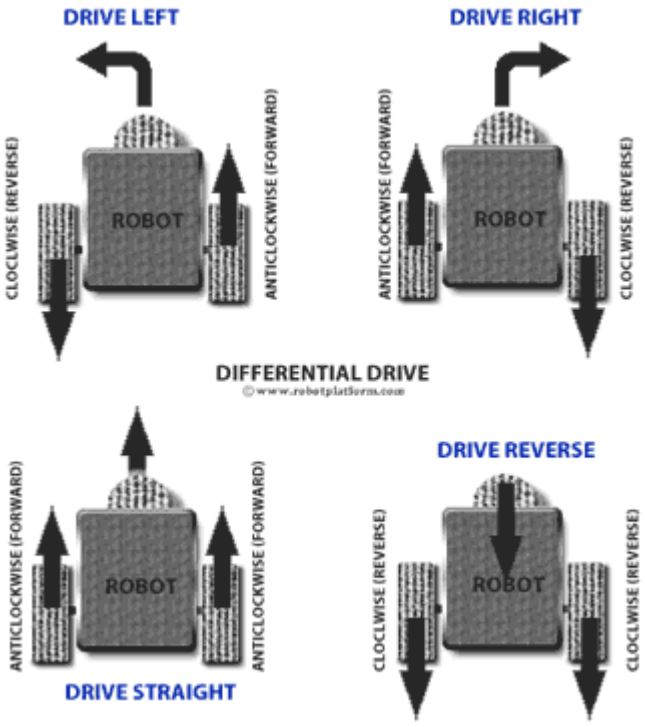

Figure 3.1: Differential Drive

only used in low-speed applications. The extra drag and wobbling of an uncontrolled wheel could cause vibrations or instability at high speeds.

I won't go into the derivation of the kinematic model, but I will present it here so you can use it for reference. There are two kinematic models of interest, the forward kinematic model and the reverse.

The reverse kinematic model tells you how the robot will move given a set of wheel velocities, and is shown in Equation **??**.

TODO: Add kinematics?

3.3.2 Ackermann Drive

Ackermann drive is a common choice for high speed robots because it provides better stability and less drag than the differential drive. It is also the same steering system that most cars use. However, an Ackermann steered vehicle cannot turn in place.

An Ackermann drive vehicle must have two front wheels which can be turned, and two fixed rear wheels. It is common to have powered rear wheels since the pivoting front wheels adds additional complexity when attempting to mount motors, but powered front wheels are not unheard of. See Figure 3.3.2.

There is one additional complication to consider when attempting to build your own Ackermann Drive system. When an Ackermann vehicle turns, each wheel is essentially drawing out the circumference of a circle. To take that thought a step further, both steering wheels will be drawing out circles, but of slightly different diameters - the difference being the center to center distance from wheel to wheel. Therefore, to avoid wheel slip while drawing two circles of different sizes, each wheel must be at a slightly different angle. The difference changes depending on how sharp of a turn is desired. Obviously designing a mechanical system to automatically handle this would be complex, and writing code to handle turning each wheel individually would be non-trivial at best. I will once again recommend buying a professionally made drive system over building one unless you have a really good reason not to.

TODO: Add kinematics?

Figure 3.2: Ackermann Drive

3.3.3 Skid Steer

Skid Steer A.K.A. Tank Drive, is another common form of locomotion. It's similar two having two differential drive systems mounted with a rigid bar between them. See Figure 3.3.3 for an example of how Skid Steer drives. Each side, regardless of how many wheels it has, is unified, meaning that all wheels on the left drive at the same time and speed, as do all the wheels on the right. Tank drive is arguably better for an outdoor robot because it is much harder to end up high-centered when all of your wheels are powered. This also gives you more traction. However, to do anything other than drive straight forward or backward, some of the wheels have to slip. The largest disadvantage to Skid Steer is that any sort of wheel speed or encoder based odometry becomes very unreliable.

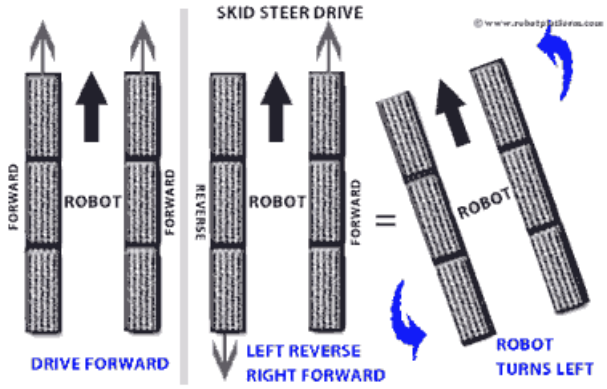

Figure 3.3: Skid Steer

However, if you have some other means of localization, like the LIDAR on the SMP robots, or the ASUS RGBD sensor, this becomes less of a problem. You end up with less drag than traditional differential drive and potentially more power because each wheel may (but doesn't have to) have its own motor. In addition, this design is much simpler to implement than Ackermann Steering or Omni-Drive, which will be discussed next. For these reasons, the SMP robots use Skid Steer. See the SMP in Figure 3.3.3.

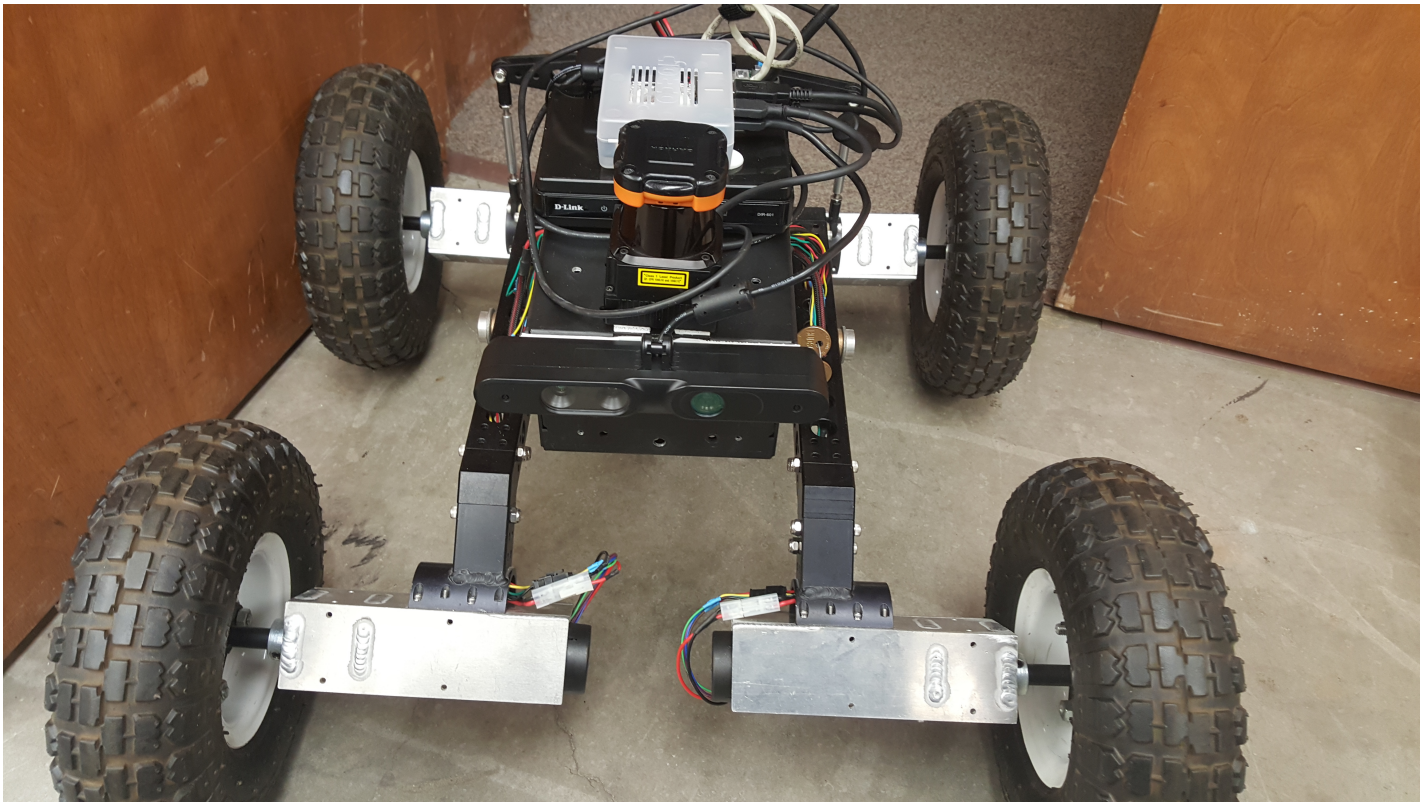

Figure 3.4: SMP Robot

3.3.4 Omni-Driv

An Omni-Drive robot is one that can move in any direction. There are several ways to construct an Omni-Drive robot, but I will focus on a four wheeled robot with fixed wheels in "traditional" positions. The Mecanum robot shown in Figure **??** - named so for the style of wheels, shown in Figure 3.3.4 - is one such robot.

Omni-Drive kinematics are complex, and out of the scope for this guide. However, I will attempt a brief summary. The rollers on a mecanum wheel are mounted at 45 degrees. The means that of any amount of force applied to rotating a wheel, only approximately half will be exerted along the line parallel with the direction of travel. The other half will be exerted along the perpendicular axis. The wheels are not mounted symmetrically, however. All wheels are mounted such that, if all wheels turn forward, the perpendicular forces will point inward and cancel, allowing the robot to move forward. By operating the wheels in opposing pairs, a force in any direction can be generated, allowing the robot to translated and rotate in any direction freely. This is very useful for a robot, because it means there aren't any constraints that have to be considered when attempting to operater the robot autonomously. If you want to move in a certain way, you can, no questions asked.

While the actual mathematics for driving an Omni-Drive robot are complex, the concept is simple. Just calculate the translation you want to preform, then the rotation, and add them. In this case, the total is just the sum of the parts. Special care should be taken, however. Attempting to both drive forward and turn at full speed works fine mathematically, but once you command a wheel to drive faster than its top speed, the robot will begin to behave erratically, as the forces will no longer properly cancel.

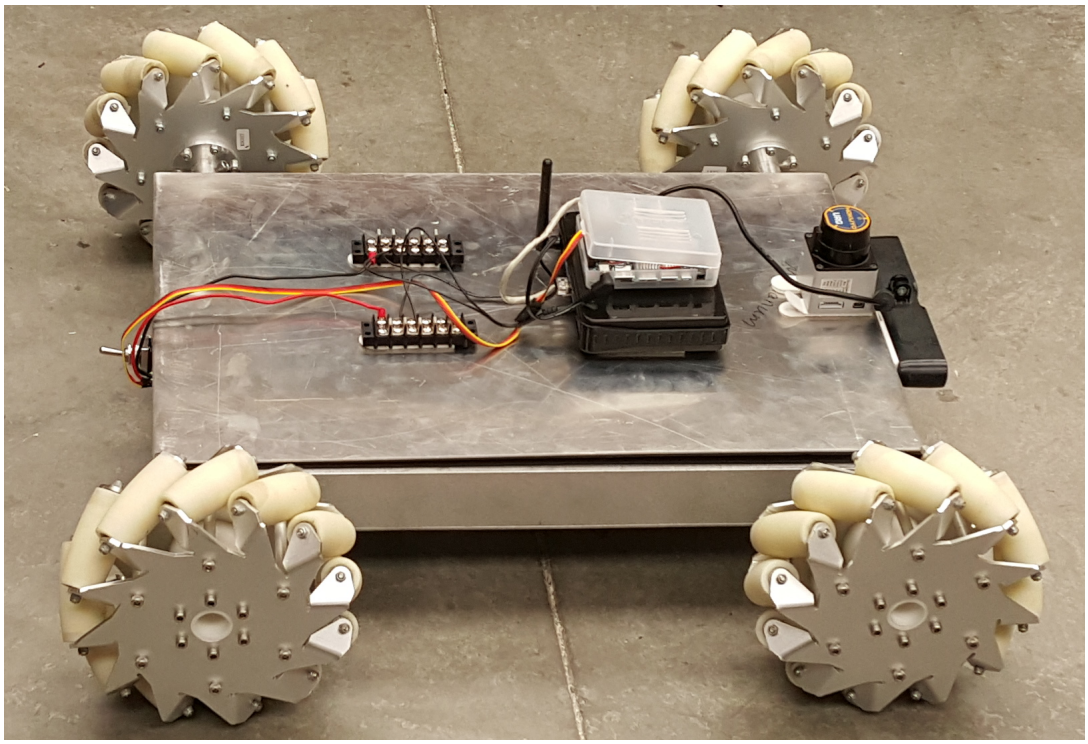

Figure 3.5: Mecanum Robot

Figure 3.6: Mecanum Wheel

4

Electrical Design

4.1 Design Considerations

The most important aspect of your robot is ensuring that it is electrically capable of fulfilling requirements - and making sure it doesn't light itself on fire or electrocute anyone. The best mechanical design in the world is useless if there's nothing to supply power to the wheels, and your computer is just a hunk of plastic and metal if it doesn't get the right amount of power. While electrical design for a robot is not very complicated (unless you want it to be), it does require some fundamental knowledge.

4.1.1 The Basics

We'll start off easy. I'm going to restrict the discussion here to (relatively) low voltage DC circuits. Within that domain, there are three very important terms. Voltage, measured in Volts (V), Current, measured in Amperes "Amps" (A), and Resistance, measured in Ohms Ω. A very smart German dude by the name of Georg Ohm came up with probably the most important (for us at least) law in all of electronics back in 1827, and published it in his book Die *galvanische Kette, mathematisch bearbeitet.* (Translated to English, that's *The Galvanic Circuit Investigated Mathematically*). The book is available online, but I wouldn't recommend it unless you're in for some heavy reading. It was written in the early 1800s after all. The important part - the law we care about here - is called Ohm's law and is shown in Equation 4.1.

$$V = IR \tag{4.1}$$

That's it. That's all there is to it. Voltage (V) equals Current (I) times Resistance(R). This law has the interesting property that, unlike everything else we do in Electrical Engineering, it almost always holds true. So, if something isn't behaving the way you think it should, refer to this law.

Ok, so now we have the fundamental law of the universe, but so what? What are all these funky terms? Let's talk more about that. Sparkfun also has a pretty excellent page explaining this, so if my explanation doesn't work out for you, just search for *Sparkfun Voltage, Current, Resistance, and Ohm's Law.*

4.1.2 Voltage

Voltage behaves very much like water pressure in a pipe. With water, pressure - or more accurately, pressure differential - is what causes water to flow. However, just because there is pressure doesn't mean the water actually flows. It's just a measure of how much the water *wants* to flow. For example, consider a completely full, sealed container of water. Now shrink the volume of the container by half without letting any water out. Some laws that I vaguely remember from chemistry say that the pressure has to increase. (Or temperature, but likely it will be some of both). Now there's more pressure, but the container is still sealed with no route for the water to escape, so there's no flow.

Voltage behaves almost exactly the same. Except instead of water, we have a high density of electrons. Because they are negatively charged, and like charges repel one another, electrons like to spread out as much as they can to reach the minimum energy configuration. Sort of like when you drop a glass of water on the

floor, and the water spreads out to get as low to the ground as possible - thereby reducing it's gravitational potential energy and reaching a minimal energy state. So, compressing a lot of electrons in to a small space creates this electrical pressure - voltage, or electromotive force for people who like words with many syllables.

The common source of voltage differentials in mobile robotics are batteries or DC power supplies.

4.1.3 Current

If voltage is pressure, than current is the amount of flow. If there is a pressure differential, and a route for water to escape, then flow happens in a pipe. This flow continues until the pressure differential is balanced out. Batteries work the same way. (By the way, think of power supplies as infinite batteries.) A fully charged battery has some voltage, and if there is a path from the negative terminal to the positive terminal, electrons will flow along that path. As the electrons flow from negative to positive and thereby "spread out" to the minimal energy state, the battery voltage will decrease. This flow is current.

4.1.4 Resistance

Resistance is not a property of the electricity itself, but a property of the material it flows through. The easy analogy is to think of the resistance as the size of the pipe. A narrower pipe allows less water to flow through. Likewise, a material with higher resistances "resists" the flow of electrons more strongly. In low power DC applications, you can generally assume that wires have a resistance of 0.

Often, we will use components called "Resistors" to purposefully introduce current in a circuit. This is particularly useful when we have components like LEDs (light-emitting diodes) which can only handle so much current. We'll have an example circuit a little later demonstrating how to choose the right resistor for an LED.

There is one more thing you need to know about resistors before we move on. What happens if you have more than one? There are two ways to hook up multiple resistors. You can put them in parallel or in series. In Figure 4.1.4 - from wikipedia, which covers this subject in great depth- there are N resistors hooked up in series. This case is very easy. The value of the resistors just adds up, so you can pretend like you have one big resistors with a value that is the sum of all the resistors in series. Because I like math, the oh-so-difficult math for this is shown in Equation 4.2.

$$R = R_1 + R_2 + \cdots + R_n \tag{4.2}$$

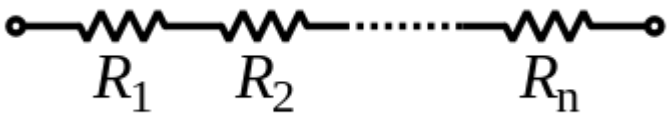

Figure 4.1: Resistors In Series

The other way to hook up resistors is in parallel. This one is a little more complicated. An example is shown in Figure 4.1.4.

Putting It All Together

Example time! First, look at the picture in Figure 4.1.6. In that example, there are multiple paths for the electricity to follow. Meditating on Ohm's law for a while will eventually lead you to the mathematical reason behind this, but for now just take my word for it: the electricity divides itself up among the possible paths inversely proportional to the resistance of each path. Ohm's law requires this, and it's actually convenient for us because it generates the least possible waste heat - but that's an advanced topic I won't cover here. Let's attach some real numbers to this example. Consider a circuit with a 10V voltage source and 2 resistors

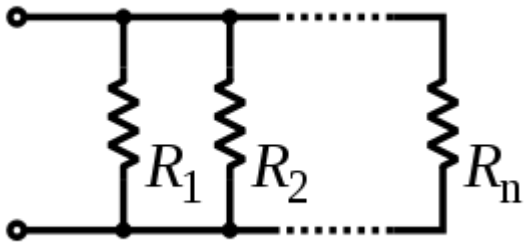

Figure 4.2: Resistors In Parallel

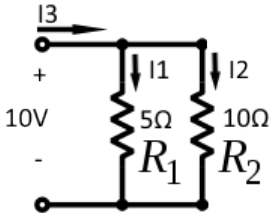

Figure 4.3: Parallel Resistors Example

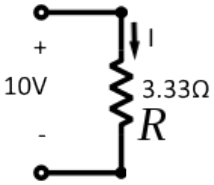

Figure 4.4: Parallel Resistors Example Reduced

in parallel, one with 5Ω and one with 10Ω. This example circuit is shown in Figure 4.1.4. Reducing the two resistors with Equations **??**, gives the circuit showin in Figure 4.1.4.

There are a few things to think about in this example. First, note that I3 in Figure 4.1.4 must be equivalent to the current I in Figure 4.1.4 since they are in truth the same circuit. In Figure4.1.4, we can easily solve for I using Ohm's law.

$$\begin{aligned} V &= IR \\ 10 &= I(3.33) \\ I &= 10/3.33 \\ I &= 3A \end{aligned} \tag{4.3}$$

We can likewise solve for I1 and I2 in Figure 4.1.4 using the same manner, but with R values of 5Ω and 10Ω respectively. This gives us I1 = 2A and I2 = 1A. Notice that I3 = I1 + I2. Another side-effect of Ohm's law is that the sum of all currents entering and leaving a node is 0, which is intuitive if you think about it. It's the same thing as splitting a hose. The sum of the volumes of water that come out of the two hoses must be the same as the volume that goes in to the un-split end.

4.1.5 Blinking Lights

One of the first things most people want to do with circuits is make lights blink. This is good. In the micro-controller world, you often don't have a terminal output until after you get through a decent amount of setup, so blinking LEDs can be the only way to debug problems during boot. The first thing most people do when they want to blink a light is blow one up. This should generally be avoided, although LEDs are cheap and it's a good learning experience.

First, let's look at the proper way to hook up an LED, shown in Figure 4.1.5. The main thing to note here is the resistor. Supplying too much current to an LED lets out the magic smoke, and remember, if you let out the magic smoke, the component no longer works. The next question, is how do you choose a value

for R? This depends on the LED, and the voltage source. For kicks, let's assume you've got a 5V power supply, from an arduino GPIO for example. By the way, drawing too much power from an arduino GPIO is a good way to kill it, although the LED usually goes first. I'll assume you've got an LED that wants 10mA of current, although this may vary. Check the datasheet for your specific part.

There is one caveat about LEDs. We haven't talked about diodes because it's out of the scope of this document. However, LEDs are a special type of component called a "semi-conductor" which means it is only conductive some of the time. In this case, the LED is only conductive when it has at least a certain threshold of voltage pushing in the "correct" direction. Pay attention to which side the Anode and Cathode of your LED are plugged in to. If it's in the wrong way, current will not flow. (Unless you supply enough voltage to cause the diode to enter the "breakdown" voltage region. In this case, that's the amount of voltage where the LED breaks, but the breakdown region is actually very important for some components. Google "Zener Diode" for more on that. Hint: They're used in motor driver boards for back-emf protection.)

So why do we care about the LED being a semi-conductor? Because semi-conductors usually have a property called a "voltage-drop". This property causes the component to eat a portion of the voltage in the circuit. Your LED datasheet should say something about the voltage drop of the LED, but a typical value is 0.7V for hobby level LEDs. We typically model this voltage drop as a small voltage source pointed in the "wrong" direction. See Figure 4.1.5.

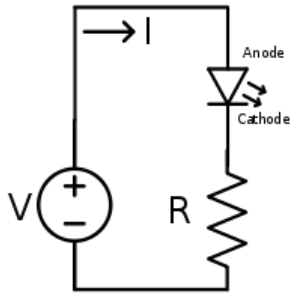

Figure 4.5: LED Proper Hookup

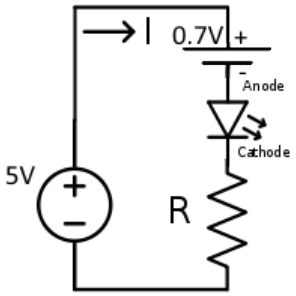

Figure 4.6: LED Proper Hookup - With Voltage Drop

In this case we can just sum the two voltage sources, and pretend we have a 4.3V power supply. So, we have a 4.3V source, and we want a 10mA current. Easy, right? Just use Ohm's law as shown in Equation 4.5.

$$\begin{aligned} V &= IR \\ 4.3 &= (.01)R \\ R &= 4.3/.01 \\ R &= 430\Omega \end{aligned} \tag{4.4}$$

It so happens that LEDs are generally pretty robust. You can probably go plus or minus about 50% on

that R value. A smaller value with make the LED brighter, while a larger value will make it dimmer. Too large a value will burn it out. An excessively large value *may* make it explode. I highly recommend trying it once under controlled circumstances. Wear eye protection! There will probably be exactly one or two pieces of shrapnel, but it would really suck to get one in your eye. For best results, connect the LED to a power supply and slowly turn up the voltage. You'll get to see all phases of operation for the LED. Below 0.7V it probably will not turn on. From 0.7 to somewhere around 4-6V it will steadily get brighter. At some point it will reach a maximum brightness and then begin to heat up, at which point it will probably start to dim. At some point after that, it will go out. Cranking up the voltage more will (probably) cause it to explode. Sometimes the conductive element inside just melts and you don't get an explosion. It really depends on the LED.

4.1.6 Voltage Divider

There's one last circuit I want to mention because it becomes very important when dealing with analog sensors. The voltage divider. An example is shown in Figure 4.1.6. The basic idea here is that you have a known resistor value for R1, and some variable value for R2. A common one is a thermistor - a resistor that changes its value based on temperature. In fact, an excellent experiment to try is to use a thermistor in exactly this circuit configuration to measure the temperature of water. It's very easy, and will teach you about the analog to digital converter. There's also the matter of sensor calibration, but I won't go in to the details on that here. Suffice it to say that all sensors are terrible, and you should never trust them until you've tested them rigorously.

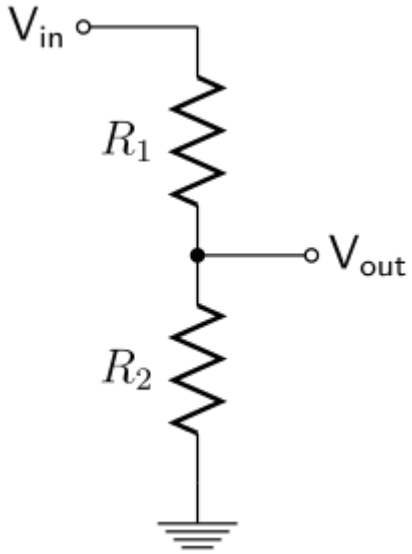

Figure 4.7: Differential Drive

The general plan is to supply a known voltage to V_{in}, have a known R_1, and measure V_{out}. With these pieces of information, you can solve for R_2 using Ohm's law. The derivation isn't difficult, but other people have already done it, so here it is. Once you have R_2 you just refer to your sensor calibration or the datasheet for the part, and you have your value for (in this case) temperature.

$$R_2 = R_1 * \frac{1}{(\frac{V_{in}}{V_{out}} - 1)} \tag{4.5}$$

4.2 Batteries

4.2.1 The Basics

We have had a lot of problems dealing with batteries over the years. Partly because people are lazy, but mostly because they don't understand how batteries work or how to properly maintain them. All batteries are basically slow, controlled, chemical reactions. They are not magical devices that store and dispense electric charge. The closest thing to that is a capacitor, which is two metal plates with an insulator between them. By applying voltage, you charged the plates, storing energy in the form of actual electric charge. However, capacitors tend to discharge all of their stored energy at once. In adddition, the total energy they can store is far less than most batteries. Batteries, on the other hand, store energy in the form of chemical potential energy. This is far more stable than storing raw electric charge, but it does lead to a few problems. The big on is that, since the energy storage relies on chemistry, temperature is important. Being stored in a place that is too hot or too cold can cause a battery to burst, or drain it. Another is that, over time, the chemistry can get disrupted, causing the battery's maximum storage potential to decline.

4.2.2 Which Numbers Are Important?

Now you know what voltage, current, and resistance all mean, but how do you use them in practice?

Voltage - V

We'll start with voltage because that's the easiest one to handle. You want to make sure the voltage your batteries produces matches the voltage rating for the things you want to power: motors, cpu, sensors, etc. However, there's another consideration to make. You almost never want your batteries directly connected to your sensitive electronics. You always want a regulator of some sort in between. Why? As you drain a battery, the voltage will fluctuate. This can cause damage to unshielded electronics. We'll talk about regulators in Section 7.3.1. In addition to the power fluctuations, you rarely have motors that you want to drive with the same voltage as the computer. In practice, you'll want to match your main battery voltage to the voltage of the motors you want to power, and then use regulators to smooth and reduce the voltage for the other components.

Capacity - mAh

This is the best measure of how much actual energy the battery can hold. To put it simply, a 1000mAh battery could sustain a drain of 1A for 1 hour before being depleted. If you draw 2A it will only last half an hour. At 0.5A, two hours. It's a very silly unit when you think about it, but it makes the math easy.

C

Most batteries have a C rating. This is a somewhat cryptic value that tells you how quickly a battery can discharge without damaging itself. This is not a limit on how much current the battery can draw. Remember Ohm's law? You thought it wasn't important, didn't you? The thing is, Ohm's law will dictate the current that gets drawn from the battery. The C rating just tells you what is safe. The tricky bit is that the actual safe rate of discharge depends on the size of the battery. Weird right? Here's the simple way to think about it. The amount of continuous current drain a battery can handle without damaging itself is obtained by multiplying the C rating with the battery capacity. For instance, a 1500mAh battery with a 5C rating can handle a continuous drain of 30,000mA or 30A.

4.2.3 Lead Acid Batteries - Pb

Lead acid batteries are very stable. That's why we use them in cars. They also have a pretty good capacity, and are able to source a tremendous amount of current at once. This is good, because it takes a lot of force to turn over an engine. For a rugged, outdoor robot that may experience a variety of temperature conditions, lead-acid may be the way to go. However, lead-acid batteries are generally larger and heavier

than their counterparts. Charging them is easy. Just hook them up to a power supply, set the power supply to the battery's rated voltage, and limit the current. What you limit the current to depends on the battery. Car batteries can generally handle up to 6A. The real problem is heat. If you charge too fast, the metal plates in the battery heat up, and can cause the acid to boil. This creates potentially toxic vapors and, if the vapors escape the battery housing, reduce the lifespan and charge of the battery. Other than charging to quickly, there isn't much to worry about here. Please charge in a well-ventilated area, just in case. Running a lead-acid battery dead isn't really a big deal as long as you don't leave it dead for a long time, or it doesn't get to cold while dead.

4.2.4 Lithium Polymer - LiPo

In the robots I built during my time on the team, we used these most commonly. They are light-weight, small, and can store a great deal of power. A LiPo battery generally consists of some number of cells. Each cell has a rated voltage of 3.7V. Batteries with more voltage are built by putting multiple cells in series. This voltage has to do with the internal chemistry. LiPo batteries are not nearly as stable as lead acid. There are three major concerns when dealing with a LiPo. First, only charge using a LiPo charger. There are some extra pins on a LiPo battery that tell the charger important information about the cells within. Second, make sure you look up the rating for charging a LiPo. The general rule of thumb is that a LiPo can be charged at the rate of 1C. If you have a 1500 mAh battery, you can charge it at 1.5A. Charging it slower is fine. Charging faster can cause the LiPo to heat up, swell, and potentially burst. When the LiPo bursts, the chemicals inside will spontaneously burn, creating fire, pressure, and heat. Seriously bad news. Third, never ever cut or puncture a LiPo. An externally ruptured LiPo is bad news, but one that ruptures internally, or without a good pressure relief, is a thermal grenade.

That all said, LiPo batteries are pretty safe if you follow the guidelines I set out for charging. There are two more things to worry about. 3.7V is the rated voltage for a cell, but when you charge, you generally charge to about 4.2V - the charger will handle this. Never manually overcharge the LiPo. However, unlike a lead-acid battery, letting the charge get too low in a LiPo will permanently damage it. The minimum safe level for a single cell is 3V. Always monitor the voltage of LiPo batteries you are using and ensure they don't drop below this level. The difficult part is, LiPo voltage doesn't drop linearly. Figure 4.2.4 shows voltage versus charge for a LiPo. As you can see from the graph, you have to monitor the battery voltage very carefully, because it decreases rapidly once the charge gets low.

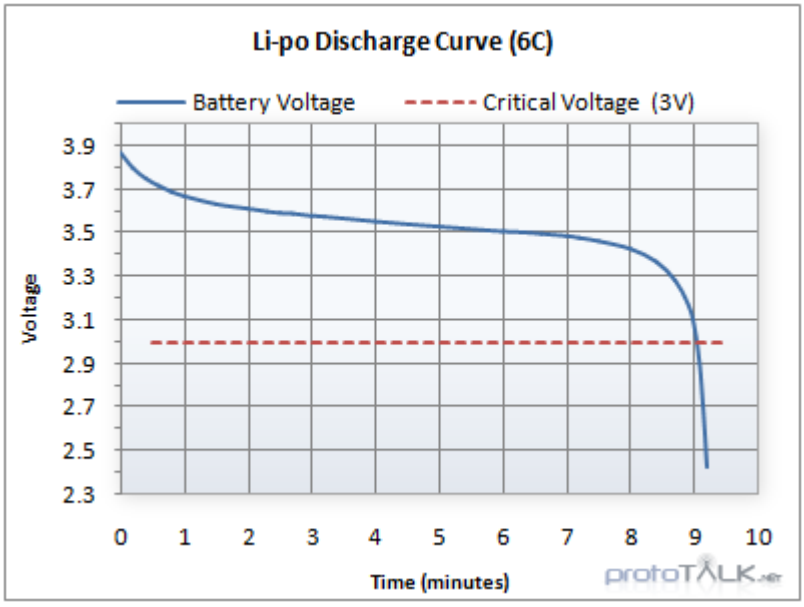

Figure 4.8: LiPo Voltage VS Charge

As a side note, this chart is for one specific battery I found on a forum post at Traxxas.com. Individual

results may vary in specifics, but the point is the same. Monitor your voltage and don't let it drop below 3.0V per cell. There is one additional caveat. LiPo batteries don't hold their charge forever, and they will, if left sitting on a shelf for long periods of time, eventually degrade. It is recommended to discharge and charge a LiPo battery every few months when it is not in active use. Discharging can be done by using the LiPo while closely monitoring the voltage, or with a dedicated charger. The chargers in the lab have this capability.

The final thing to mention is balancing. Because a LiPo battery may be made up of multiple cells, the total voltage isn't enough to tell the health of the battery. In the lab we have at least two balances. Some chargers will also balance while they charge. Balancing slowly bleeds charge from one cell and puts it into another cell. This keeps the cells from becoming unbalanced. This is a good thing. Each cell may discharge differently because chemistry is a sloppy science and never works quite the way it's supposed to. This can lead to one cell with a much higher or lower voltage than the others. If one cell gets overcharged, it may swell and/or rupture. If one cell gets too low, it may "die". Dead cells are ones that have dropped to a low enough voltage that they cannot safely be recharged. Most chargers will refused to charge the battery if there are dead cells. If you know what you're doing, sometimes dead cells can be nursed back to life, but it's a delicate and potentially dangerous procedure. It's usually better to recycle the battery and get a new one.

4.2.5 Other Batteries

There are three more common types of batteries. Lithium Iron Phosphate - LiFePO4 often pronounced "LieFo" - Nickle Metal Hydride - NiMH - and Nickle Cadmium - NiCd "Nigh-Cad". I haven't personally worked with either, so the internet is a better resource then I am. My vague understanding is that LiFePO4 are similar to LiPo batteries but more stable and more expensive. A quick Google search tells me that NiMH is the new NiCd, and is used in general consumer electronics, so it must be pretty stable, but doesn't (in general) have as much energy density as Lithium batteries.

4.3 Tips For Not Lighting Things On Fire

Forethought and attentiveness are key for keeping the magic smoke inside the components. All electrical components are made using plastic, silicon, some trace metals, and a mystical substance called magic smoke. If you give the component too much voltage, current, or heat the magic smoke will use this extra energy to break free and escape. It is a very clever substance, and even just a momentary spike is enough to free it. At this point the component will no longer work. Nobody is perfect, but here are a few tips I've picked up over the years to keep the magic smoke locked up tight.

Turn Off The Power

This should be common sense, but you'd be surprised how often people get overconfident about what they're doing and modify a circuit while it's powered. Sure, if you know what you're doing you're theoretically safe. The real problem arises when one of those tiny wires gets away from you. Powered wires appear to be supernaturally attracted to conductive terminals, particularly ones that will causes sparks. Just don't do it. The five seconds it takes to flip off the power could save you three days of waiting for new parts.

Electrical Tape

If you're working on a circuit, if you aren't immediately dealing with a wire, tape the end. Most of the instances in which I blew something up were because I forgot about a wire that wasn't connected, and it brushed up against something else, making a short. This is particularly important when dealing with batteries. Remember, you can't turn a battery off. If both terminals of a battery aren't connected to something, the free wires should be taped over.

Fun Fact: If you plug the gate pin on a MOSFET in to 24VDC, while the source is floating and the drain is grounded, it explodes.

Be Careful Around Batteries

I've alluded to this several times, but I'll say it again. Always be careful around batteries. I had six years of robotics team experience, and four years of that was while taking courses in Electrical Engineering. At the end of my last semester, I was putting different connectors on some LiPo batteries. I thought to myself, "I'll just cut both wires on the battery at the same time so they'll be the same length."

Technically speaking, it worked out. Turns out the reflex when you hear a loud SNAP and see a bright flash of light is to close your fist, which for me finished cutting the wires and yanked the tool away from the battery cables. It could have been worse though. Just think about what you're doing around batteries. Remember, they can't be turned off.

Don't Work When You're Tired

Or drunk. Or sick. Or angry. It makes you inattentive, and then things like the above happen. It doesn't matter how smart you are. Tired people make mistakes.

Ground Loops

Ground loops are tricky devils. Explaining the whole mechanism behind them is outside the scope of this document, however, if you are ever dealing with a circuit with multiple power sources, look them up. A ground loop can ruin your whole day. Essentially, ground loops occur when two things you consider "ground" have some resistance between them. If that happens, current through one loop induces a voltage difference in the other loop. Wikipedia is your friend here. Moral of the story - use isolated power supplies, and be very careful when plugging multiple power sources in to one set of electronics. When in doubt, unplug your computer from the wall before plugging anything grounded into it. USB cables, for instance.

Use the Mult-Meter

The multi-meter is your friend. When powering up a system for the first time, unplug all sensitive electronics, power the system, and check voltages at your main terminals. Make sure the value *and* the polarity are correct. Then power down, plug in just one piece of equipment, and power back up. Repeat until everything is plugged in. This is useful for two reasons. First, if you have a problem, it will be very obvious which piece is the culprit. Second, if something blows, you risk damaging the minimum possible number of expensive items.

5

Odroid Setup

5.1 Purpose Of This Chapter

This chapter is intended to be a guide for setting up an Odroid XU3 with Fedora 21, using an EMMC as the storage device. Step by step guides for using an SD card exist in plenty on the internet, and only the first two steps are any different. The steps should work with Fedora 22 as well, but Fedora 22 was still very new when I last worked with the Odroids and wasn't ready for installation on an arm architecture yet. For installation on other Odroid platforms, the steps should be very similar, but may not be exactly the same.

5.2 Before You Begin

You will need the following items to complete this guide:

1. Odroid XU3
2. USB to MicroSD Adapter
3. MicroSD to EMMC Adapter
4. EMMC
5. Computer Running Fedora 20/21
6. USB to Serial Adapter

Check your version of Fedora before beginning. You can do this using the command:

```
$ cat /etc/issue
```

You want to check if you're running Fedora 20 or Fedora 21+. This will be important later.

5.3 Obtain the Fuser

The Fuser is a utility written by Scott Logan to set up the EMMC in such a way that the Fedora image fits on the EMMC efficiently. It sets the "fuse" bit, which are sort of like software-writable settings on the EMMC. Modifying these bits manually is pretty difficult, which is why you'll want to use the Fuser to do it for you.

On Fedora 21+, run:

```
$ sudo yum install smd-odroid-release
$ sudo yum install odroid-xu3-sd-fuser
```

These two commands download repository information for the smd-odroid-release repository - which Scott maintains - and then downloads and installs the fuser itself.

Of Fedora 20, the repository headers will not automatically download. You'll have to download the fuser more directly. Instead run:

```
$ sudo yum install http://csc.mcs.sdsmt.edu/smd-odroid/fedora/linux/21/x86_64/odroid-xu3-
```

5.4 Run The Fuser

If you run in to problems, there is a wiki page with instructions specifically for the fuser at `https://github.com/sdsmt-robotics-odroids/odroid-xu/wiki/Boot-Media-Fusing`.

Before inserting the EMMC, run the command:

```
$ ls /dev/mmcblk*
```

Take note of the entries there. Whatever you see there before inserting the EMMC is what you will NOT want to target in the next steps. Next, connect the EMMC to your computer. Plug the EMMC into the EMMC to MicroSD adapter. If your machine has a MicroSD slot, you can plug that directly in to your computer. If not, (or if you encounter problems, which can happen rarely mostly due to driver issues) you'll need a MicroSD to USB adapter.

Run the ls command again. There should be new entries, probably something like /dev/mmcblkX... in a few various forms. Take note of the value of X. Now run the following command, substituting your number (usually 0) for X.

```
$ sudo odroid-xu3-emmc-fuser /dev/mmcblkX
```

You should see the following:

```
 Successfully enabled write access to /dev/mmcblk0boot0.
Fusing boot blob to /dev/mmcblk0boot0...
1262+0 records in
1262+0 records out
646144 bytes (646 kB) copied, 1.71663 s, 376 kB/s
Success!
```

If you don't see something like this, specifically if there are 0+0 records in or out, something has gone wrong. Try again or check the wiki instructions for the Fuser.

5.5 Loading The EMMC Image

First you need to get the image to write to the EMMC. The images are hosted at `http://csc.mcs.sdsmt.edu/smd-odroid/fedora/linux/21/Images/armhfp/`. You'll want to download the "latest" image for your type of Odroid. In this case, the filename will probably be something like: f21-odroid-xu3-minimal-latest.img.xz.

Once you have downloaded the image, uncompress it using the following command with <foo> replaced by the image name. This will take about a minute.

```
$ xz -d <foo>.img.xz
```

Next, you need to unmout the EMMC partitions or they won't write correctly. Do so with the following command, again substituting your value for X:

```
$ sudo umount /dev/mmcblkXp*
```

Be careful with this next command. dd is a very powerful tool, and it can totally erase your harddrive if you let it. Mostly just make sure you don't accidentally point it at the wrong device. if is the input file - the image you uncompressed. of is the target device, the mmblkX we identified earlier. Substitute your value for X. bs is the block size. Just leave it alone.

```
$ sudo dd if=<foo>.img of=/dev/mmcblkX bs=8M
```

This takes a while. On the order of 3 or 4 minutes when I did it last. Be patient. When it finishes, you should expect to see:

```
  204+1 records in
 204+1 records out
 1713373184 bytes (1.7 GB) copied, 91.7598 s, 18.7 MB/s
```

Next, to make sure the OS has flushed all the buffers to the device, run:

```
$ sync
```

If there is a noticeable pause after running sync, run it again just in case. That pause means it's working on something. If there is no noticeable pause, it didn't do anything, and you're good to go.

5.6 What Next?

At this point, you have a functional EMMC with Fedora 21 installed. Go ahead and plug it back in to the Odroid. I recommend getting a USB-Serial converter cable to debug with. I also recommend getting a CMOS battery for the Odroid, or the system clock will reset every time the board is restarted. This makes yum unhappy and you'll have a bad time. ROS can throw some very strange time-related errors as well. If you only needed Fedora installed, you're done with this chapter - except perhaps see the section on common error messages. Otherwise, move on to the next section to install ROS. It's possible the Scott has the kick-start version up by now, with ROS already installed, but at the time of writing, that hasn't happened yet.

5.7 Installing ROS

The ROS build for arm isn't in with the normal ROS or Linux repositories. It was in fact hosted from, when I left, a set of 10 Odroids over in M113. To install this repository, run:

```
$ sudo yum install http://csc.mcs.sdsmt.edu/smd-pub/fedora/linux/21/armhfp/smd-ros-releas
```

You'll also need rpm fusion, which is an extended set of packages for Fedora not bundled in to the main set of rpms. To install those, run:

```
  $ sudo yum localinstall --nogpgcheck http://download1.rpmfusion.org/free/fedora/rpmfusion
-E %fedora).noarch.rpm http://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-rele
```

Now, you should be able to use yum to install pretty much any ROS Fedora package. The following command will install the basic starting ROS packages. It doesn't not include PCL or Gazebo, but you shouldn't be running those on Odroids anyway.

```
$ sudo yum install ros-indigo-robot
```

If you want to do on-board compiling, which you almost certainly will unless you've already finished development and are just deploying functional binaries, run the following command:

```
$ sudo yum install python-rosdep python-rosinstall_generator python-wstool python-rosinst
```

That's it. Installing ROS is way easier than it used to be. If in doubt, here's a link to the ROS wiki page dealing with installation issues: `http://wiki.ros.org/indigo/Installation/Source`.

This is a fairly basic install, so there may be some unresolved dependencies. If you're having dependency issues, run the following command where <src> is a directory containing ROS packages.

```
$ rosdep install --from-path <src> --ignore-src
```

5.8 Setting up a static IP

One thing you will probably want to do at some point is to give the Odroid a static ethernet IP address. This saves you from needing to hunt down a serial cable to get in and run ip addr every time you connect it to a new network. This is, thankfully, very easy. Just find the /etc/network/interfaces file. It should look something like:

```
# The loopback network interface
auto lo
iface lo inet loopback
\begin{lstlisting}[language=bash]
# The primary network interface
auto eth0
iface eth0 inet static
 address 192.168.0.1
 netmask 255.255.255.0
```

Yours probably doesn't have an iface eth0 entry yet. Add one, and replace the address with whatever address you want the odroid to have.

5.9 Common Errors

There are a handful of pretty common errors that I found solutions for:

```
  One of the configured repositories failed (Fedora 21 - x86_64 - Updates),
and yum doesn't have enough cached data to continue. At this point the only
safe thing yum can do is fail. There are a few ways to work "fix" this:

 1. Contact the upstream for the repository and get them to fix the problem.

 2. Reconfigure the baseurl/etc. for the repository, to point to a working
    upstream. This is most often useful if you are using a newer
    distribution release than is supported by the repository (and the
    packages for the previous distribution release still work).

 3. Disable the repository, so yum won't use it by default. Yum will then
    just ignore the repository until you permanently enable it again or use
    --enablerepo for temporary usage:

        yum-config-manager --disable updates

 4. Configure the failing repository to be skipped, if it is unavailable.
    Note that yum will try to contact the repo. when it runs most commands,
    so will have to try and fail each time (and thus. yum will be be much
    slower). If it is a very temporary problem though, this is often a nice
    compromise:
```

```
        yum-config-manager --save --setopt=updates.skip_if_unavailable=true

Cannot retrieve metalink for repository: updates/21/x86_64. Please verify its path and try
```

So this is a pretty scary error, but the actual cause is pretty tame. This happens once on almost every new Odroid install. The root cause is actually that the internal clock is set way back in the past to a time where the repository didn't exist. So when yum tries to check for the repository certificates, none of them are valid yet. To verify that this was the problem use the command:

```
$ date
```

If your Odroid is kickin' it in the '70s. You've found the problem. This is an easy fix. Run the following commands. The first updates the current time using the ntp server on campus. The second updates the hardware clock, which persists between power cycles provided you have that CMOS battery installed. If not, you'll have to do this every time.

```
$ ntpdate ntp.sdsmt.edu
$   hwclock -w
```

There was, last time I spoke to Scott, a potential problem with a dtb file not being updated in the kernel. If this happens and causes problems, just get a new image and repeat this process. Scott was hoping to have this fixed soon, so it's probably already done by now, but I'll leave this warning here anyway.

6

ROS Intro and Architecture

6.1 What is ROS?

There are a dozen tutorials online that do a better job of giving beginner's ROS tutorials. I'm not going to try to repeat that here. However, there are some things I've learned about ROS that are worth passing on that seem to get forgotten or difficult to find in the tutorials.

First, ROS isn't big, scary, or even all that complex (for the user). Fundamentally, all ROS does for you is pass messages. Just think of it as a communication system for mini-programs you write, called Nodes. The channels they communicate over are called Topics. There is a main ROS control program typically called the Master, or roscore.

It is definitely true that there are very big and scary ROS constructs out there. The TF library for one. Hector SLAM for another. These are not, however, part of ROS. They just *use* ROS. There are some excellent "first program" tutorials on the ROS wiki, but it is really more of a reference than a guide.

6.2 ROS Lingo

There are a few important terms in ROS. They have very precise definitions, but I've summarized the easy version for a few of the more common terms here:

Node

A ROS Node is a single computational unit. Generally, this is a single process, but you can have a single node launch multiple threads if you wish. Usually, this is some chunk of code written in Python or C++, although other languages are supported. A good rule of thumb to follow is that each node should do only one task. Much like functions in a traditional program, nodes should be designed to be as small and modular as possible - while still maintaining some level of usefulness. Code reuse is king in ROS.

You can see which nodes are currently active, and see some information about the node using the commands:

```
\$ rosnode list
\$ rosnode info <node>
```

Master

The master is a special node, started by running the command:

```
\$ roscore
```

Do note that the process that starts roscore does not end, so that terminal will be unusable as long as roscore is running. It is inadvisable to detatch the roscore process from the terminal when running manually,

because you won't be able to see warnings or errors. The master is essential to running all other ROS commands. This node is responsible for setting up channels of communication between the other nodes in the system. However, it does not take an active role in that communication once it is established.

Publisher

A publisher is a type of node that publishes information, making it available to other nodes.

Subscriber

A subscriber is a type of node that consumes information from publishers.

Topic

A topic is a named communication channel. This is the mechanism by which publishers and subscribers produce or consume information. Any number of nodes can publish to a single topic, and any number can subscribe to it, but there is no ROS-standard way to designate a particular receipient. This is by design and allows for greater flexibility and generality. These topics are almost always one-directional. If you have two nodes that talk back and forth, you should use two different topics; one for each direction of information flow.

A topic is intended for long-term, continuous communication. For one-shot, or very infrequent communication, use a service.

You can view which topics are currently active and get information about them using:

```
\$ rostopic list
\$ rostopic info <topic>
```

Workspace

A workspace is a directory structure in which you will do your work when you want to compile and run your own ROS nodes. There is an excellent tutorial about setting up your first workspace at `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`. I recommend putting workspaces in the home directory as a general rule.

Service

A service is sort of like a special, one-shot communication between nodes. This is generally used for something like buttons that the robot must respond to when pressed. The robots in the lab often have a "safe-start" service, which enables the motor controllers. Services should not be used for frequent communication.

Launch File

A launch file is an .xml document. It is essentially a script for starting up a master node and several other nodes. It also makes it very easy to run nodes with arguments. In short, if you end up running more than one node together, or if you have a lot of parameters, you'll want to make a launch file. Refer to Section 6.3 for an example.

Namespace

A namespace is a way to group related nodes and topics. Generally, we create a namespace for each robot. That way, if multiple robots were running on the same network, we avoid naming collisions with the nodes and topics. For example, every node and topic created in the Mecanum launch file will show up as /mecanum/<node>. Namespaces can be nested, but we haven't needed to do that.

Parameters

Most nodes have parameters, much like command line arguments in C/C++. In ROS, these parameters are handled as key-value pairs. Each parameter has a name and a value. One example is the joy topic name, addressed in Section **??**.

Driver

A driver is a very special node. Drivers are the only nodes in ROS that are not supposed to be platform independent. That's because they're the only nodes that get to talk to hardware. A prime example of a driver is joy_node from the package joy that reads input from a PlayStation 3 controller. There are also several drivers we wrote in house including the roboteq_nxtgen_controller, roboclaw_driver, and yei_tss_usb. These nodes have exactly one responsibility: communicating with their piece of hardware.

Message

A message is a single packaged unit of information to be sent over a topic. ROS has a large set of message types built in to the core system. In general, you won't want to define your own messages. Not only is it somewhat difficult to do, your code won't work with nodes written by other people. In ROS, code re-usability and modularity is king. Again, try to avoid creating your own message types.

Common message types include: joy, Point3D, twist, quaternion, and many others

The standard types like int, bool, string, and arrays are also supported, but you end up using those rarely in robotics.

Distribution

There are several versions of ROS. Most of my work was done in ROS Indigo. As of early August 2015, the wiki still said ROS Jade was on track to be released in May of 2015, so we can probably expect to see that soon, but not yet.

As a side note, the ROS mascot is a turtle, and the distributions are following alphabetical progression. The versions thus far have been:

1. ROS Box Turtle - March 2, 2010
2. ROS C Turtle - August 2, 2010
3. ROS Diamondback - March2, 2011
4. ROS Electric Emys (usually just called Electric) - August 30, 2011
5. ROS Fuerte Turtle - April 23, 1012
6. ROS Groovy Galapagos "Groovy" - December 31, 2012
7. ROS Hydro Medusa "Hydro" - September 4, 2013
8. ROS Indigo Igloo (Stable) - July 22, 2014
9. ROS Jade Turtle (Unstable) - May 23, 2015?

Also, if you think ROS is complicated and hard to use now, take Fuerte or Electric for a spin.

6.3 Launch Files

Launch files are essential when putting together a bunch of nodes to run a robot. Listing **??** shows the launch file used by both SMP robots.

```
\label{lst:launchfile}
\caption{SMP Launch File}
<?xml version="1.0"?>
<launch>
  <group ns="$(env ROBOT)">
    <node pkg="nodelet" name="nodelet_manager" type="nodelet"
          args="manager" output="screen" />

    <node name="joy_to_twist" pkg="joy_to_twist" type="joy_to_twist_node"
          output="screen">
    </node>

    <node pkg="roboteq_nxtgen_controller" name="nxtgen_right_driver_node"
          type="nxtgen_driver_node" output="screen">
      <param name="hardware_id" value="Right RoboteQ Controller" />
      <param name="port" value="/dev/robot/right_motor_controller" />
      <param name="ch1_joint_name" value="right_front_wheel_joint"/>
      <param name="ch2_joint_name" value="right_rear_wheel_joint"/>
      <param name="use_encoders" value="true" />
      <param name="operating_mode" value="closed-loop speed" />
      <param name="encoder_ppr" value="420" />
      <param name="reset_encoder_count" value="true" />
      <param name="ch1_max_motor_rpm" value="72" />
      <param name="ch2_max_motor_rpm" value="72" />
      <param name="invert" value="true" />
      <remap from="joint_states" to="right_joint_states" />
    </node>

    <node pkg="roboteq_nxtgen_controller" name="nxtgen_left_driver_node"
          type="nxtgen_driver_node" output="screen">
      <param name="hardware_id" value="Left RoboteQ Controller" />
      <param name="port" value="/dev/robot/left_motor_controller" />
      <param name="ch1_joint_name" value="left_front_wheel_joint"/>
      <param name="ch2_joint_name" value="left_rear_wheel_joint"/>
      <param name="use_encoders" value="true" />
      <param name="operating_mode" value="closed-loop speed" />
      <param name="encoder_ppr" value="420" />
      <param name="reset_encoder_count" value="true" />
      <param name="ch1_max_motor_rpm" value="72" />
      <param name="ch2_max_motor_rpm" value="72" />
      <param name="invert" value="false" />
      <remap from="joint_states" to="left_joint_states" />
    </node>

    <node pkg="skid_drive_controller" name="skid_drive_controller" type="base_controller_no
      <param name="wheel_base1" value="0.5969" type="double"/>
      <param name="wheel_radius" value="0.1016" type="double"/>
    </node>
  </group>
</launch>
```

There are a few important things to note. First, this is just a normal .xml document. Everything works just the same any any other .xml document. There's a lot in here, but it's really not that scary. Take each

block one at a time and it becomes easier. There are a few things I want to highlight about this document.

<launch>

The launch tag is just the root of the document and is required to be present in order for the file to be parsed correctly.

<group ns="$(env ROBOT)">

This is one of the more obscure tags. What this is doing is assigning everything in the block to a certain namespace. That namespace is defined by the environment variable ROBOT. On the SMP and Mecanum robots, ROBOT is defined in the bringup service. We'll talk about that in Section **??**.

<node...>

The node tag defines that this block pertains to launching a single node. If you want to launch multiple nodes of the same type, each node needs to make a unique appearance in the launch file. They do not need to be given unique names. ROS automatically appends a large pseudo-random number to each node name to avoid name collisions. Topics, however, will collide if not named uniquely. This allows multiple nodes to speak or listen to the same topic.

pkg, name, type

These are named a bit non-intuitively.

"name" is the name you want the node to appear with in utilities like rosnode list.

"pkg" is the package the node is found in.

"type" is the actual name of the node you want to launch.

Getting type and name confused is very common. I did it all the time. In fact, I never actually knew for certain which was which until I bothered to look it up while writing this guide.

output="screen"

Adding this to a node causes it to pipe all output to the terminal. By default, the output is logged in ROS, but not displayed. I recommend turning this on basically always.

<param>

This allows you to supply arguments for the key-value parameters for a node. While the ROS community has a set of guidelines for naming parameters, and exactly what parameters should be available, the actual parameter set depends on the node.

<remap>

This allows you to change the name of a topic that would be subscribed to or published by the node. This is useful when you want to write a single generic piece of code for a node, but each instance of the node needs a separate channel over which to communicate.

6.4 Running Nodes Manually

The alternative to using launch files is to launch nodes from the command line. The general format for this is:

```
$ rosrun <package> <node>
```

However, sometimes you also want to specify parameters or in some way modify the behavior of the node. All parameters can be set, using the syntax:

```
$ rosrun <package> <node> <paramter>:=<value> <parameter>:=value ...
```

In addition to the parameters set up in the node itself, there are five special keys that are built in to ROS and don't require the parameters to be specified in the node itself. Use of these parameters is generally not encouraged outside a launch file, but they are there if you need them for testing.

__name

You can override the default name for a node using the __name paramter.

__ns

You can supply a namespace using the __ns parameter.

__log

You can supply a path for the node's log file.

__ip and __hostname

You can supply ROS_IP and ROS_MASTER_URI values using __ip and __hostname respectively.

__master

You can supply a value for ROS_MASTER_URI in this way as well.

6.5 Running ROS

When running ROS, there are a few important steps you must make in *each* terminal you open. Another option is to add these commands to your /.bashrc file so they are automatically run whenever a new terminal is opened.

Sourcing

When you start up a terminal, it generally doesn't know what ROS binaries are available, or where they exist. Commands like rostopic list will not be understood by the terminal if you have forgotten to source. To source your general ROS installation, use:

```
$ source /opt/ros/<ros-version>/setup.bash
```

To use ROS binaries you compiled, you need to source the setup.bash in your workspace. The command should look something like:

```
$ source ~/my_workspace/devel/setup.bash
```

Exports

There are two very important system variables that must be exported for ROS to function correctly. The ROS_IP and ROS_MASTER_URI. The ROS_IP tells the master at what IP address the nodes launched on the local machine can be found. The ROS_MASTER_URI tells the nodes on the local machine where to find the master. If you are having messages along the lines of "Failed to contact master" these are a good place to start. To set your ROS_IP run:

```
$ ip addr
```

Or

```
$ ifconfig
```

To find your ip address on the network of interest. If are connected via Ethernet, there should be an "eth" interface. "lo" is loop-back, which is 127.0.0.1, and you almost never want that one unless you are only running all nodes locally. I believe this is used by default if ROS_IP is not set. "wlp10s0" or similar are wireless networks. In all cases, you are looking for an IPv4 address.

Once you find your address, set the ROS_IP with the command:

```
$ export ROS_IP=<ip>
```

The ROS_MASTER_URI is a bit trickier. First, you have to find the IP for the master node the same way you found the IP for ROS_IP. Of course, it needs to be on the machine the master will be running on. If you're running everything locally, the IPs will be the same. However, for reasons that are probably good but unknown to me, the ROS_MASTER_URI is treated as a URL instead of an IP address. So the command looks like this instead:

```
$ export ROS_MASTER_URI=http://<ip>:11311
```

11311 is the port number that the ROS master listens on for new communications. If you have a firewall active, this port must be opened correctly.

Firewall

ROS can be made to play nicely with firewalls, but it's generally more trouble than it is worth. If you want to talk between machines, or if you have problems communicating on a single machine, disable the firewall using this command:

```
$ sudo systemctl stop firewalld
```

This command only stops the current firewalld process. To disable firewalld from launching on boot, as well as stopping the running process, use:

```
$ sudo systemctl disable firewalld
```

However, this does leave your machine vulnerable. If your machine is ever expected to connect to the internet again, make sure to renable and restart the firewalld process using:

```
$ sudo systemctl enable firewalld
$ sudo systemctl start firewalld
```

6.6 Making a ROS Robot

Now you have enough knowledge to get by with. The next step is actually starting to put something together. This section assumes you already have Linux and ROS installed on whatever computer you intend to use as the brain of the robot.

7

Robot Parts

7.1 What Is A Robot?

We've talked a lot so far about kinematics, not lighting yourself on fire, and getting ROS up and running, but what about all the bits that make your robot a robot? This is where things get tricky. There isn't one solution for all robots. I'm going to restrict this guide to RC wheeled robots and talk about some of the basic components you're going to need. This is just a high-level conceptual overview. For an example of how to set up power and data wiring for these components, refer to the Mecanum/SMP design document.

7.2 Locomotion

7.2.1 Brushed DC Motors

The typical drive motor. DC motors spin when power is applied, and spin the other way when power is applied with reverse polarity. DC motors aren't fancy, but they give you a good mix of speed and power. Almost every robot I have worked with used DC motors for motion. So, how do you choose a motor? That's actually pretty tricky, but there are a few numbers that are important.

Torque

Put very, very simply, torque is a measure of the strength of the motor. The units are weird, because torque is actually a measure of the force a motor can extert at a given distance from its center of rotation. More specifically, it is the cross product of force times distance, so you end up with things like newton-meters or ounce-inches.

But how much torque do you actually need? That involves some pretty hard math, actually. Your best bet is to look at the motors used on the SMP and Mecanum robots and judge based on those numbers. You can find those details in the design document that should be provided with this document.

Voltage

Remember when we talked about voltage? It's back! Brushed DC motors are odd in that they can actually handle being given higher or lower voltages than they are rated for. In fact, most good motors come with voltage vs torque curves for multiple supply voltages. Typically the datasheet will give absolute maximums and minimums for safe operation. The rated voltage is the one at which you will get the optimal tradeoff between torque and lifespan. Increasing the voltage will generally increase the torque and decreased the lifespan. Decreasing the voltage does the opposite. Of course, increasing the voltage too much will overheat the motor and destroy it. Remember Ohm's law? Increasing the voltage increases the current through the motor coil, and more current creates more heat.

Current

This one is a bit tricky conceptually, but easy to deal with in practice. Motors have an internal resistance, often called coil resistance. Ohm's law states that, given some resistance and some voltage, you will get some fixed current. However, in practice, the current draw actually depends on how hard the motor is being worked. The maximum current draw from a motor occurs when it is being driven constantly without being able to turn. This is called the stall current, and is generally listed in the motor documentation. For the sake of safety, assume the motors will draw a little more than their stall current.

RPM

Revolutions Per Minute. This is basically the speed at which the output shaft of (generally speaking) the gearbox will turn when the motor is spinning with no resistance. When you load it down with a motor, you can expect this number to drop considerably unless the motor has a phenomenal amount of torque.

Gear Ratio

Most motors come with a gearbox. In fact, often you will find a whole bunch of motors for sale that differ only in the gearbox. This makes sense, because making a different gearbox is way easier than making a different motor. The gearbox will have some gear-ratio, which probably means a bunch of really interesting things to people more mechanically inclined than I am. The easy version is that the gear-ratio is the number of times the motor shaft turns before one full revolution of the output shaft. A higher gear-ratio gives more torque, but reduces the actual speed of the output shaft.

7.2.2 Brushless DC Motors

Brushless DC motors are actually quite different in operation compared to Brushed DC motors, and require different control hardware. They are typically used in quad-copters and other applications that require extremely high RPM. They are also used in some performance RC cars. Typically, brushless motors do not have gearboxes because the wear at those speeds would make them explode, but there are exceptions.

I don't actually know what happens when you give a brushless motor more or less voltage than it expects. Never tried it. I would do some research though. Brushless motors can be temperamental.

7.2.3 Stepper Motors

Stepper motors work fundementally differently from DC motors. I won't get in to the details - the internet will do better by far - but to boil it down, stepper motors step instead of spinning continuously. You will most certainly want a dedicated driver board for a stepper motor, as the power has to be supplied to the magnetic coils in a very precise manner to get the most out of the stepper motor. Steppers may or may not have gearboxes, but adding a gearbox in this case both increases torque and the resolution of the steps while decreasing the maximum turning speed.

To drive a stepper motor, you generally hold a direction pin - DIR - high or low for clockwise or counter-clockwise (whether clockwise is high or low may vary based on the driver hardware) and then pulse a STEP pin to advance the motor by one step. This is very, very useful in situations that require a high degree of precision. You don't need additional hardware to tell exactly how far the motor has turned - in theory. In my experience, stepper motors have the potential to be forced to spin when commanded to stay still. There is no way to know when this has happened without feedback. However, if you can prevent that from happening by properly sizing the motor to the job, they are very accurate. If you are going to do something that requires a high degree of precision, like a 3D printer, stepper motors are an excellent solution.

7.2.4 Servos

Servos deserve a mention here because they are a very simple budget option for small robots. Continuous rotation servos can deliver a modest amount of power and speed, and take very little work to power or control. Servos are generally slower and weaker than DC or stepper motors, and are not terribly accurate. Feedback of some sort is essential, whether its in the form of encoders, line-following, or a vision system.

7.3 Power

Powering a robot is tricky business. We talked about batteries in Section 4.2, which should get you at least as that far, but there are more considerations to be made.

7.3.1 Regulators

Regulators are your friend. They keep your parts from exploding. A regulator takes a potentially noisy input (i.e. one that has spikes and voltage fluctuations) and smooths it. If there's a spike it can't handle, the regulator dies valiantly protecting the rest of your robot. This is a good thing. Regulators are much cheaper than, for example, a LiDAR.

7.3.2 DC-DC Converters

DC-DC converters take one voltage and convert it to another voltage. This is pretty cool. It means your 24V batteries can power your 3.3V cpu. It is worth noting that, because of the way transformers work, you can actually get more current out the low-voltage side than you put in. Of course, the total power usage is the same. Why exactly this happens is a topic for Circuits 2, but Equation 7.1 shows a handy formula to demonstrate how the converter deals with power in and out, where E is the efficiency of the converter. This is a value you can find in the datasheet. The voltages are, in general, fixed, as is the efficiency. So you can predict the high-voltage current draw based on what you expect the low-voltage current draw to be.

Do note that most DC-DC converters have some maximum output current. Trying to draw more current than the converter is rated for will overheat and eventually release the magic smoke from the converter.

$$V_{in} * I_{in} * E = V_{out} * I_{out} \tag{7.1}$$

7.3.3 Motor Controller

On any robot more sophisticated than an arduino strapped to a bread-board, you're going to want dedicated motor control hardware. There are several good reasons for this.

First, most of your hardware is not going to be able to handle sourcing the ten, twenty, thirty, sixty amps that your motors are going to demand. Remember Ohm's law? In this case, Ohm's law says that, when you apply a given voltage across a given resistance, a certain current will happen. That current will continue to happen until something gives. In this case, it will almost certainly be your control boards, not the motors or the wires.

Second, motors generate a lot of noise. They're basically DC generators in reverse. When you supply current, the coil windings create a magnetic field to turn the motors. However, if the motor is turned by an outside force, or its own momentum when at speed, the spinning of the motor induces a current in the motor windings that can cause large voltage spikes. Failure to protect against this voltage, commonly referred to as "back-emf", releases the magic smoke.

Another good reason is dealing with feedback. To manually handle feedback from encoders, you would have to dedicate almost all of your CPU time to just watching the pins for the encoders, and that's just for one. At the very least, you need an intermediate piece of hardware to handle encoder ticks. The HCTL-2032 is one such chip. I strongly recommend against this route. I tried that once and it was a huge pain both in software and wiring. Not remotely worth the money saved on getting an actual motor controller.

There are a few important things to look for in a motor controller.

Channels

A motor controller generally has 1, 2 or 4 channels. This is the number of motors that the controller can control independantly. That doesn't necessarily mean more channels is better. I generally recommend using two 2-channel motor controllers over a single 4-channel motor controller. 4-channel motor controllers are generally either a) really expensive, or b) actually just two 2-channel controllers on a single board. If option

b) is implemented poorly, you actually end up with a lower maximum power output. I once bought a 4-channel motor controller that claimed it could handle 2A per channel, but if you pulled more than 4A total, a power trace that carried power from the controller A side to the controller B side would blow up.

Max Current

This one is pretty self-explanatory. Make sure this number, generally indicated per channel, is higher than the maximum current draw you expect to see from your motors. The number you are actually interested in is the continuous current draw maximum, as opposed to the burst or instantaneous maximum. For some applications, where current spikes are expected but the typical current draw is much lower, the burst current is important. This is generally not the case for the robots we build.

Voltage

There are actually two imprtant values here as well. The supply voltage, and the logic voltage. The supply voltage is the voltage the motor controller will use to drive the motors, and most motor controllers can actually handle a pretty wide range of input values. Interanlly, there's just a few MOSFETs and zener diodes that actually have to interface with that voltage level. Just make sure that the battery voltage you will be supplying to the motor controller falls within the acceptable range. The logic voltage is the voltage the controller requires to power the actually controller board. This varies depending on the controller, although typical values are 3.3V, 5V, or 12V. Again, just make sure the voltage you are supplying falls in the allowable range on the motor controller datasheet. In some instances, motor controllers have built in DC converters that allow them to draw power from the main supply voltage.

Quadrature Decoding

If you're going to bother with a motor controller, get one that has quadrature decoding. This allows the motor controller to handle feedback from quadrature encoders, which I would recommend.

PID Speed Control

While it may, in some cases, be useful to do your own velocity control, in general you won't want to bother. It's computationally expensive, and a robust control algorithm is non-trivial. Also, someone has already spent a lot of money to do that job better than you will. Trust me, just get a motor controller than can do PID control for you. The Roboclaw motor controllers in the Mecanum, and the RoboteQ motor controllers in the SMPs all do PID speed control.

7.4 Encoders

While encoders specifically aren't necessary - there are other forms of feedback - I recommend including encoders on any ROS robot. Without encoders, you can't utilize the PID speed control from the motor controllers. I recommend using quadrature encoders. Look specifically for the word quadrature. Quadrature implies that the encoder has two inputs which are 90 degrees out of phase. This allows the quadrature decoder to infer both speed *and* direction.

It is certainly possible to implement your own encoders, or to buy separate encoders and attach them to your motor drive shaft. However, in my experience, this is stupid. Don't do it. Just spend extra money and get motors with built-in quadrature encoders. Seriously. Trust me. If you attach your own encoders you have to worry about extra mounting hardware and wires that are right up against parts that spin. All that, and it will never be quite as accurate as you had hoped.

7.5 On/Off Switch

This is one that often gets forgotten, but becomes very important with robots that draw any significant amount of power. Your typicaly solution may vary, but I recommend getting a regular switch and either

a relay or a contactor. Use the switch - which almost certainly can't handle the current you want to draw from the batteries - to activate the relay or contactor, and use the contactor to actually deliver power to the system.

7.6 Parts List

To reiterate, here is a basic outline of the parts you want on your typical mobile RC robot.

- Motors (4)
- Motor Controller (1x4 channel, 2x2 channel, or 4x1 channel)
- Voltage Regulator (If powering anything using raw battery power)
- DC DC Converter (1 per voltage level required other than battery voltage)
- Brain (The Odroid XU3 in this guide)
- Encoders
- Power Switch
- Wireless Router

The wireless router isn't really necessary for the robot, but it's my recommended way to communicate with robots. Set up an onboard router and put the odroid at a static ip address - addressed in Chapter 5. That way all you have to do is connect to the network and ssh to a known ip address to get access to the robot. No more cable hunting and no need to pull the odroid off the robot.

Otherwise, that's really all you need for a very basic robot. It'll be a glorified, expensive, and very complicated RC car, but it will drive. The cool stuff starts happening when you add sensors or autonomy. Many sensors can be just plugged in via USB, but some require power lines as well. This is where additional DC DC converters and regulators come in.

8

Robot ROS Setup

So you have a robot with all the parts in, Linux booting, and ROS installed. What next? The Section will give you a guide to getting your robot under remote control over a network using a PS3 controller, which is our standard procedure for demos. If you already have a robot set up and just want to drive it around, skip to Chapter **??**.

8.1 Packages

The first step is making sure you have all the right packages installed in a place that ROS can find them. There's a lot to do right away, but don't worry, you'll get the hang of it. First, I'll go through a step-by-step guide and explain each step. At the end of the section, I'll reiterate with just a list of commands to get from 0 to RC in... well, probably not 3 seconds, but you get the idea.

Creating the Workspace

First, we need a workspace to put everything in. I'm going to assume you're making a workspace called demo_ws and putting it in the home directory for the odroid user. The following commands will create the workspace. (This is copied directly from the online tutorial)

```
$ mkdir -p ~/demo\_ws/src
$ cd ~/demo_ws/src
$ catkin_init_workspace
# cd ~/demo_ws
$ catkin_make
```

ROS needs to do some setup in the directories before you can really use them. This includes creating a few directories and creating a CMakeList file. Don't touch the CMakeList file. It's just a symlink to the CMakeList in /opt/ros/<distro>, which you really don't want to break. Each package will have its own CMakeList that you need to modify, but we'll get to that later.

Getting Packages

For this guide, I'll be using the SMP robot as an example. To get the demo-bot up and running, you'll need the following packages.

- joy
- joy_to_twist
- roboteq_nxtgen_controller
- skid_drive_controller

- smp_bringup

What do all of these packages do? Excellent question.

The joy_node in the joy package is a driver that communicates with a PlayStation 3 controller. At the time of writing, there was a slight issue with the joy_node. On Fedora 22, they have moved away from the js system for joystick inputs and instead moved to the event system. I believe an update to the joy_node was in development, but I never got a chance to use it. For now, it may be best to just use a machine running Fedora 21 to take commands from the PS3 controller, because the joy_node works very easily that way. In any case, the joy_node talks to a PS3 controller and publishes a joy message, which is just a bunch of floats in the range -1 to 1 to indicate the state of each button and joystick axis.

The joy_to_twist isn't in our GitHub repositories like the rest of the packages to follow. It currently exists in two places on GitHub: `https://github.com/cottsay/joy_to_twist` and `https://github.com/CBJamo/joy_to_twist`. CBJamo's version is (currently) a branch from cottsay's version and more recent. Eventually, the pull request by CBJamo will likely be folded into the master branch in cottsay's repo at which point the CBJamo repo will likely cease to exist. Basically, look at both and pick the one that has the most recent commits.

Anyway, the joy_to_twist node subscribes to the /joy topic, and spits out what we call a twist message on the /cmd_vel topic. cmd_vel stands for command velocity, and is the desired speed of the robot given the controller input. This includes both translation and rotation. The twist contains a whole bunch of position, velocity, and acceleration data, but the joy_to_twist node only populates the velocity data.

The purpose of the skid_drive_controller is to take a commanded velocity from /cmd_vel and produce a set of motor commands. The skid_drive_controller is our first node that isn't entirely system agnostic. Ideally, the only system dependent nodes should be drivers, but in practice this is rarely feasible. In this case, the Mecanum robot and the SMP robots have different kinematic models, so they must deal with velocity commands differently. The skid_drive_controller node is aware of the fact that the robot cannot translate freely, and produces motor command accordingly. The skid_drive_controller also uses multiple topics to output commands, and those topics are expected to be remapped. This is because the skid_drive_controller node doesn't know ahead of time how many motors there will be, what motor controller will be in use, or what topics it needs to publish to. See the launch file in smp_bringup for details.

The roboteq_nxtgen_controller subscribes to one topic for each motor being controlled. For the SMP, two instances of this node will be launched by smp_bringup, each subscribing to two topics. The roboteq_nxtgen_controller also publishes a bunch of really useful diagnostic info and allows the transmission of commands to the motor controllers. For this guide, we're going to ignore all of that. The roboteq_nxtgen_controller is a driver node, which means that its job is strictly to be the interface between the rest of the ROS system and the hardware.

The smp_bringup package doesn't actually contain any nodes, but it does contain the launch file for starting up all of the nodes mentioned thus far with the proper parameters, namespaces, and topic names.

So, how do you get the packages? There are actually two ways. For packages that are in the main ROS repositories, you can use the package manager yum. For instance, to install the joy package in ROS Indigo:

```
$ sudo yum install ros-indigo-joy
```

For the packages which are not part of the core ROS repositories, you'll have to download the repositories from git-hub and compile them. Fortunately, we were really nice when we made the repositories and it's real easy to do.

```
$ cd ~/demo_ws/src
$ git clone <package-repo>.git
```

There, that wasn't so bad. We put each package in its own repository so they could be selected individually for inclusion on specific robots.

To compile everything just run the following commands:

```
$ cd ~/demo_ws
$ catkin_make
```

If all goes well, you won't have any compiler errors... but... well, you know how that goes. Things in ROS are changing all the time, and sometimes we break the repositories. Sometimes you'll get this far before you find out your ROS install is broken. Fortunately, there are some pretty smart people at SDSMT who can help you out. You can also send me (the author) an email and I can try to help you out. My contact info should be somewhere near the beginning of the document.

Udev Rules

Hardware always throws a wrench in things. Say you've got two motor controllers plugged in via serial or USB (serial is better, btw). How do you know which device is which at boot? The answer? You don't. They'll mount in whatever order they want. Fortunately, there is a simple solution. Refer to listing **??**.

```
  \label{lst:udevrules}
  \caption{99-smp2.rules}
  ACTION=="add",SUBSYSTEMS=="usb",ATTRS{idVendor}=="20d2",ATTRS{idProduct}=="5740",ATTRS{se
ACTION=="add",SUBSYSTEMS=="usb",ATTRS{idVendor}=="20d2",ATTRS{idProduct}=="5740",ATTRS{seri
ACTION=="add",SUBSYSTEMS=="usb",ATTRS{idVendor}=="15d1",ATTRS{idProduct}=="0000",SYMLINK+="
ACTION=="add",SUBSYSTEMS=="usb",SUBSYSTEM=="tty",ATTRS{idVendor}=="2476",ATTRS{idProduct}==
```

This looks a little confusing at first, partially because of word wrap. Do note that each ACTION should get its own line. Line breaks are significant here, so don't put any extra in. Basically, each line sets up a filter. Whenever the system detects that a device has been plugged in, it will check the device against each filter in the udev rule files. (There can be several udev rule files, but they must all begin with a two digit number and be placed in the /usr/lib/udev/rules.d folder or they will be ignored. The number is the priority, with 1 being the lowest and 99 being the highest. It is generally recommended to use very high numbers, because using a very low number can actually interfere with some of the things the OS wants to do. In practice you only generally have one udev rule file and you call it 99-<whatever-you-want>.rules.

If you want to create a new udev rule, the process generally follows the following steps:

- Unplug the device
- Run the `ls /dev`lstinline command
- Plug the device back in
- Run the `ls /dev`lstinline command
- Find the new device
- Run the `udevadm info -a -p $(udevadm info -q path -n /dev/video2)`lstinline command
- Find the idVendor and idProduct for the device
- Copy an old udev rule and replace the idVendor and idProduct
- Change the SYMLINK field to whatever you want it to show up as
- Google udev rules because it didn't quite work

That last step is optional. Also, not all fields are required. If you want any, for instance, roboteq motor controller, to mount to the robot/left_motor_controller mount point, you can leave off the ATTRSserial field. The serial number is unique to each device. The product id is unique to a specific part number - vendors are real good about playing nice with Linux this way. The vendor id is unique to each manufacturer, but generally you want to use both vendor and product ids. There's nothing stopping two manufacturers from using the same in-house part number after all.

Autorun at Boot

Usually, with robots, you don't want to ssh in to the robot and run the launch commands every time you want to do something with the robot. What you really want is for the robot to just start up whenever it receives power and await commands. Fedora 21 makes this pretty easy with the systemd boot services. Here's the file you'll need, I'll talk about it here in a second.

```
\label{lst:smpbringup.service}
\caption{SMP Bringup Service}
[Unit]
Description=ROS Launch for smp
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
User=odroid
Environment=ROS_MASTER_URI=http://192.168.0.2:11311
Environment=ROS_IP=192.168.0.2
Environment=ROBOT=smp2
ExecStart=/opt/ros/indigo/env.sh /home/odroid/demo_ws/install_isolated/env.sh roslaunch smp
KillMode=process

[Install]
WantedBy=multi-user.target
```

So this is actually pretty tricky. We had to experiment to get this service to start up right. There are still a few wacky things in ROS that they're trying to iron out in Jade and future releases. One such issues is ROS's tendancy to freak the **** out if it tries to start up before the networking drivers are loaded. That's why the After= and Wants= targets have to be in there. That basically forces this service to wait until those two things are ready before running. You'll notice the Environment= values are just the environment variables talked about in Section 6. The ExecStart= command just tells it to go find and run the launchfile as a bash script.

You'll notice that we have the insatll_isolated directory in our workspace instead of just the install directory. Don't worry about it too much. Once you have a working robot, you can compile an "isolated" version of your packages. This strips out all the build files and makes everything a bit smaller and faster. This can be important on mobile robots with limited space and power, but don't worry about it until you're finished developing.

The rest is basically boiler plate - meaning I don't actually remember what it does, but everything breaks if it's not there.

This file, which we called smp_bringup.service, belongs in the /etc/systemd/system/ directory. The only special naming convention there is that each file must end with ".service".

One big plus of doing it this way, is you can startup, shutdown, or restart your ROS programs using systemctl.

```
$ sudo systemctl start|stop|restart smp_bringup
```

9

Demo Day

So, you've got five minutes. Here comes Dr. Riley with a tour group and you weren't ready. Never fear! These instructions will have you up and running with an RC robot in half that time.

You'll need the following:

- USB-to-Serial adapter (optional)
- USB-USBmini cable
- PS3 Controller
- Computer with Wifi running Fedora 21 with ROS and joy installed

Start the timer. If you set up a static ip address for the odroid as shown in Chapter 5, skip step 2.

1. Power on robot
2. Plug the PS3 controller in to the computer
3. Plug/Unplug the PS3 controller and push the button until the player 1 light stays on
4. Connect wirelessly to the robot
5. Serial in to the Odroid to find the ip address - optional
6. Source your ROS install

```
$ source /opt/ros/indigo/setup.bash
```

7. Find your own ip address - it'll be a 192.168 address
8. Export IPs

```
$ export ROS_IP=<your ip>
$ export ROS_MASTER_URI=http://<robot ip>:11311
```

9. Run the joy node locally

```
$ rosrun joy joy_node joy:=/<robot>/joy
```

In the last step <robot> is the name of the robot used in the the launchfile as a namespace. See Chapter 6 for details.

Done. Probably. You should be able to move the joysticks on the PS3 controller and get the robot to move at this point. If that doesn't work, you probably missed a step during setup. Or I missed a step with this guide. If that's the case, let me know so I can update it.

Bibliography

[1] Paul Oliver, *Guppies - Evolving neural networks (w.i.p.)*, https://youtu.be/tCPzYM7B338, 2012.