Mechatronics Project Guide

MEEN 3230 – Mechatronics – Spring 2024

Table of Contents

Introduction

This document is designed to be a reference for many aspects of mechatronic designs, especially applications related to the MEEN 3230 course here at The U. Throughout the course you will build a mechatronic system, and the information provided here may be of use to you in creating a well-functioning system. These concepts are also generally applicable, as is all the knowledge you learn in this course, to any mechatronic system you may encounter throughout the rest of your life.

We recommend using this document in a piecewise manner, investigating the sections that are applicable to you at a given time rather than trying to read the entire document all at once. It is a good idea to give a brief scan over the full document early on so you are aware what type of information is provided, and then go back later and dive further into the details of a particular component once you are dealing with that component.

Mechanical Design Considerations

Robustness

In any process there are going to be variations, tolerances, and unexpected conditions. Building a system that is robust to such concepts will ensure smooth operation throughout the semester. A system that is robust and capable of completing a task even if the environment has changed, or the object interactions are not perfectly positioned, will result in better performance than a system that can do everything perfectly, but only as long as the conditions are just right. Think about where possible locations of environment and task variation might occur: maybe an object is placed in a slightly angled or shifted position from where you expected it, maybe the tolerance on the position of a drop off location is somewhat large. If you think about where these variations can occur, you can design your system and your strategy to be robust to them, and a small variation in the tasks won't cause a major problem for your system.

Adjustability

Designing your system to be adjustable can save you time and energy throughout the semester. There are many ways to do this, including choices like installing a slot for a bolt to go through instead of a hole, or mounting a component to a flat surface rather than a custom-fit inlaid location. As an example, if you decide to change your distance sensor to a sensor that has a different sensing range, the new sensor will likely be a different size. If your mounting bracket is designed to fit only your original sensor then you'll need to redesign it and remanufacture it. But if you designed it to have slots instead of bolt holes, and a large flat mounting surface, you could mount the new sensor to it with no extra effort. When mounting something in a slot be sure to securely fasten the bolts so the component doesn't shift during operation.

However, this doesn't mean you should make all holes into slots and design every component of your system to be adjustable. The position of some components is critical, and a mounting slot would likely be inappropriate for those. Mounting your Arduino to your base plate could likely use mounting slots so it can be shifted to make space for new components in the future, but something like the motor for your manipulator would likely use mounting holes to be secure and not shift its position.

Think carefully about your design choices and include adjustability when appropriate. You can save yourself a lot of effort in the future if done right, but make sure you don't overdo it and cause yourself headaches from components shifting when you don't want them to.

3D Printing

Be aware of printing times and misprints: get started early. The printers in the library, as well as the senior design space, will get backed up and have misprints that cause printing schedules to take longer than desired or expected. Give yourself plenty of time.

Tapping thread into 3d printed material can yield poor results. The material strength of the filament is very low compared to the bolt that is screwed into it and can cause the bolt to strip the hole and come out. A different method is to design a through hole in the part, pass a bolt all the way through, and screw a nut onto the bolt on the other side. There are some methods that can be used when designing a 3D printed part that make using a nut and bolt more effective or space saving. These methods include designing placeholders in the part for the bolt head or nut to go, often called captive nuts (See the figure). For some examples you can refer to online guides like: https://www.instructables.com/CAPTIVE-NUTS-AND-MORE-IN-3D-PRINTING/

Acrylic

Acrylic sheets are similarly a common construction material, and sheets that are 12"x12" are available for

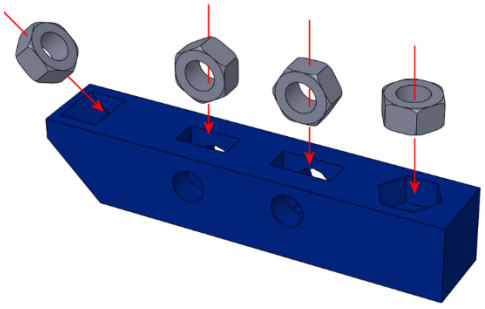

Figure 1: Captive nuts example in a 3D printed part.

purchase from the lab in various thicknesses. This material can be drilled and cut manually or can be cut on a laser cutter. A laser cutter is available in the senior design space, as well as Lassonde. When drilling or cutting through acrylic be careful and go slow and steady, acrylic is very brittle and can crack if too much pressure is applied to it. If you design a model of what you want and are using a laser cutter, there are some considerations for that process. Sharp corners can cause excess stress, if you can round the corner it can reduce the likelihood the part will crack. If you need a sharp corner, say for a finger/box joint, you can add a tiny hole on the corner to provide stress relief. Information on these two tips is expanded upon in the Laser Cutting file on Canvas.

Chassis Cover

Don't forget to design a chassis cover. There will be a base that all your components will mount to, a cover provides protection to all those components. You don't want anything falling on your robot and dislodge

one of your components, and you definitely don't want anything metal falling into your wiring and shorting/damaging your electronics.

When you design your chassis cover, design placeholders for your switches and buttons to be installed in it. Mount switches and buttons in a way that you can operate the switch from the outside, and the wiring is on the inside. This allows you easy access without needing to remove the cover. It might be useful to design placeholders in the cover for other components like the fuse or the USB port on the Arduino. This way you can access components that you interact with often without having to remove your chassis cover. You will need to remove/disassemble the chassis cover though at various times, so try to design the cover in a way that is easily, and hopefully quickly, installable and removeable so you can make easy repairs when needed.

Size

Your system is allowed to be up to 12"Wx12"L, this does not mean your system *has to* be that size. While a larger system will allow you more room to install your components, you may run into problems navigating around the field. Opting for being slightly smaller than maximum size can be beneficial in many ways.

Other

Think about how your system will assemble and disassemble. If you design it in a way that is difficult to assemble you can cause yourself grief down the line when you need to make changes to internal components. Along the same lines, think about where the components you interact with the most will be placed. Placing an Arduino in the bowels of your system is likely going to impede your ability to quickly access wiring. You don't want to design your system in a way that the entire thing needs to be disassembled just for small fixes like a loose wire. That being said, you also probably don't want to do things like place your Arduino on top of your system where objects are going to bump into it constantly during operation.

Mechanical Components

Wheels

There are three scooter wheel sizes to choose from: 70, 84, and 100 mm. These scooter wheels will fit on the 37 mm Pololu motors with **appropriate hubs**. There is only one size of wheels that will fit on the Micro Metal Gearmotors. These wheels are 32mm in diameter.

If your system has two drive wheels and you plan to provide additional wheeled support with ball castors, it is recommended to use a single castor instead of two. If you use two castors your system will have four points of contact with the ground. With four points of contact you can only guarantee that three points will touch the ground at a time. This is why a four-legged stool can be wobbly. To get good traction for all four points of contact some additional suspension mechanism would be necessary. If one of your points of contact lifts off the ground, and it ends up being one of your drive wheels, then your robot won't drive well. Using only one castor provides three points of contact and won't result in your system being inherently wobbly. However, if you place your drive wheels in the center of your system you can place a castor on both sides, thus having two castors on your system. Adjust the height of the castors so that they are slightly higher than the wheels, and now you can guarantee your drive wheels will always be touching the ground, and one of the castors will be too.

Omni/Mechanum wheels allow for movement in all directions when using three or more of these wheels. Each wheel requires its own drive motor. When planning for this type of wheel in your design keep in mind the additional costs of buying more drive motors, and motor drivers. We have limited supply of omni-wheels for purchase, meaning you might end up purchasing them yourselves.

We do not recommend tank treads, and we don't sell them. In general, the use of treads is designed in a way that they slip along the ground when turning. This means that attempts to do accurate positional tracking of the robot would not work due to inaccurate wheel odometry.

Shafts

If you use a shaft for anything in your design, make sure it is supported properly. Free-standing shafts (as in shafts that are not built into a motor/gearbox) most often should be supported in two locations (preferably on either side of where the load is applied) to prevent it from acting like a cantilever beam and bending. You should be careful to not apply too much radial force to a motor shaft otherwise you can bend it (like putting a chain and pulley on it and then over-tightening the chain). However, direct applications like putting a wheel on the gearbox output shaft of the 37mm D motors are common in this course.

Belt Drives

If you are using a belt and pulleys, be careful about tension. You need enough tension to keep the belt from slipping. This means in some cases you might need some mechanism designed to tension the belt. This could be something like a roller on a bolt that is positioned in a slot, and it pushes against the belt to tension it. If you install the pulley directly on a motor shaft, remember to not bend the motor shaft by applying too much radial force (tightening the belt too much). One way to avoid this is to place the pulley on a free-standing shaft, then support that shaft with two bearing mounts, and connect a motor to it with a coupling.

The belts we sell come in specific circumferences, so plan for that. Or you can buy a straight length of belt, but you'll need to connect the ends to form a loop somehow. Be aware, that could pose a problem if you're doing continuous rotation because the connection point will have to go over the pulleys.

Bearings

If using a shaft in your system, we recommend using bearings to support that shaft. Most bearings support radial loads, some support axial loads. Be aware of your needs and choose the bearing that fits your application. If you are using a purely rotational design, a radial load bearing will do the job for you. If there is any chance of an axial force being applied to the rotating shaft, it is likely that you'll want a bearing that can handle axial forces. If you have a shaft sliding axially back and forth, there are linear bearings that can support that motion.

Linear Motion Interfaces

There are many designs that result in linear motion of components: rack and pinion, screw drives, etc. When considering these mechanisms, it is important to think about not only the actuation process, but also the interface between the moving and non-moving parts. Will your system have two pieces that slide across one another? What material will those pieces be made of, and how much friction will there be between them? Does anything hold your components in position like a track, groove, or rail? Something like a groove can be designed into a 3D printed part, but make sure you're aware of the tolerances necessary to keep your part stable without causing too much friction or jamming. A 3D printed track or groove is a simple way to design a system, but it can also be prone to jamming if not designed and tested appropriately. Off-the-shelf components can be purchased through places like McMaster-Carr for linear rail/bearing systems. These types of components provide low friction motion with good stability, but at a higher cost.

Other

Often times small prototypes can be beneficial in the early stages of design and manufacturing. If you plan on making a component sliding in a groove, maybe manufacture a small version of that piece and the groove, and then see if it jams when sliding. These small efforts in the beginning of your process can save effort later down the line.

Actuators

DC Gearmotors

DC gearmotors are generally used for continuous rotation applications (rotating more than 360 degrees) like the drive wheels of your system, or a belt drive that has a large angular range. These motors are driven by a motor driver, see the motor driver section for more info on those components. The motors themselves have two wires that supply power to them from the motor driver. For the 37mm D motors that come with an encoder pre-attached, the two motor wires are the red and black wires. The other 4 out of 6 wires coming out of those motors are for the encoder. The 20mm D motors and micro-motors don't come with an encoder pre-attached, but we sell the encoders and you can solder them on yourself. To provide some wire relief (so that something that tugs on the wires doesn't directly tug on the encoder) you can ziptie or tape the motor wires to the body of the motor. This way a tug on the wires will only tug on the ziptie/tape.

The DC motors we provide have mounting brackets for purchase so you can easily mount them to your system. The only mounting holes on the motors are generally on the front face of the motor where the shaft is sticking out. The mounting brackets will provide a flat surface with bolt holes to mount the motor to a baseplate or the like.

While the motors are capable of operating at 12V, the robots are limited by the batteries which supply 9.6V. It is safe to assume that the maximum voltage you can supply will be about 9V since there is a voltage drop across the motor drivers used to control the motors. Motor specs can be found online from sources like https://www.pololu.com/product/4753.

The 37mm Diameter motors are the most powerful and largest option we sell. They are often good for drive wheels and heavy lifting operations. The 20mm D motors are a good step down from the 37mm motors, and provide good torque and speed abilities in a smaller form factor. The micro-motors are very small and are good for small actuation purposes without a large torque requirement.

Servos

There are a variety of servo motors available for purchase in this course. They have different torque ratings and can be used for operations where only a fraction of rotation is needed (up to about 180°). While such devices can be useful for mechatronic systems, you should always determine whether the servo you select is capable of moving your load. Before finalizing the servo you intend to use, it is **STRONGLY** recommended that you perform a calculation to estimate the expected torque of your load. It is common for teams to overestimate the capabilities of the servo motors and are forced to redesign. Taking the time to perform these calculations can save your team time later on.

For connecting a servo GND should be a common ground with the Arduino and every other electrical component on the robots. The voltage supply should typically be 4.5-6 V. Check your servo datasheet to verify. The PWM signal will come from your Arduino. You can use the Servo.h or PWMServo.h libraries to control servos (PWMServo tends to be better in our experience, as it is immune to other libraries that use interrupts such as Encoder.h, but will restrict you to pins 11, 12, and 13).

Servo output shafts are small, often plastic, and have tiny gear teeth on them. They are not a standard shaft like a motor. Therefore, servos come with a servo horn which is a small plastic piece that is designed to fit onto the servo shaft. These servo horns will look sort of like propellers that have a flat face with mounting holes that you can connect components to. Be sure to use these servo horns and design your pieces to attach to them.

Motor Driver Shields

Motor Driver Board options (this connects the microcontroller to the motors):

- L298N Motor Driver Board
    a. Generally used for micro-motors and 20mm D motors for your mechanism actuator. Not generally used for drive motors.
    b. Capable of 2A continuous per motor, 2.5A peak for 10 ms
- Pololu Dual TB9051FTG Motor Driver Shield
    a. Generally used for 37mm D drive motors.
    b. Can operate two motors at once.
    c. Capable of 2.6A continuous per motor, 5A peak
- Pololu Single TB9051FTG Motor Driver
    a. Same as the dual motor driver shield, but only operates a single motor, and is not a shield so it doesn't mount directly to an Arduino.

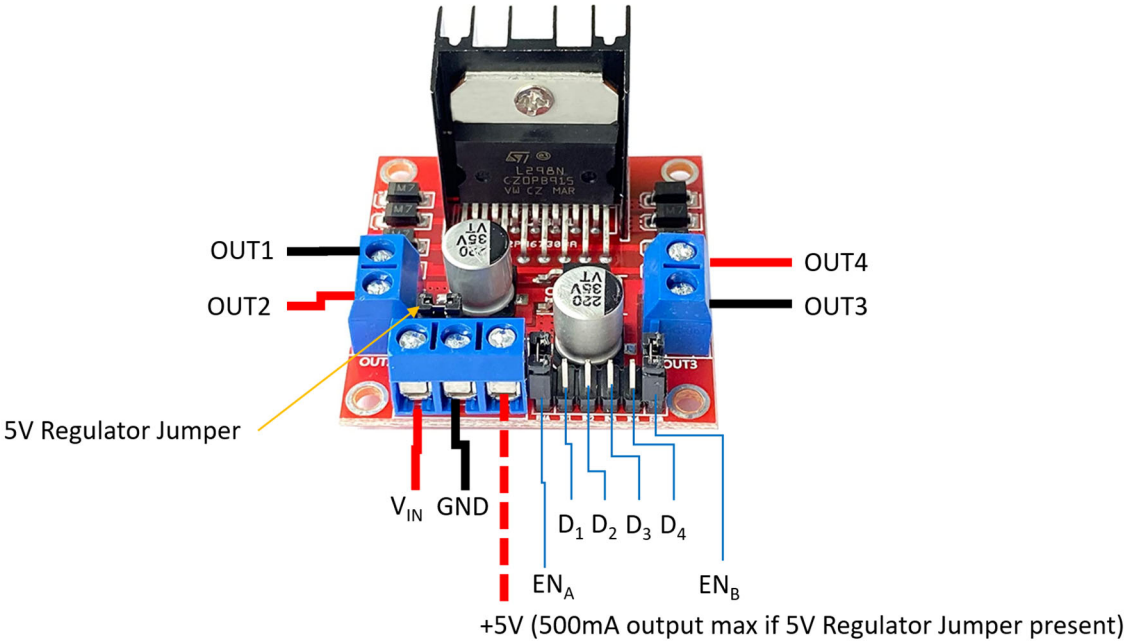

Figure 2. L298N Motor driver board and related input/output connections.
Based upon modified Amazon product listing image.

L298N Driver board

Figure 2 shows a L298N Motor Driver board. This writeup uses information from the data sheet https://www.st.com/resource/en/datasheet/l298.pdf . Table 1 highlights details about the pin connections and jumpers on the board. Some pin connections and jumpers are more obvious than others. V_{IN} is the power supply pin for the board and chip. Per the data sheets, V_{IN} can be connected to a power source

ranging from 2.5 V to 45 V. Ground is connected to the power source ground. IT IS RECOMMEND TO NOT USE THE 5V OUTPUT.

Connection Pin	Functionality
V_{IN}	Power input. 2.5 V - 45V DC.
GND	Power ground.
+5V	5V Logic power. Can output up to 500 mA safely.
OUT1, OUT 2, OUT 3, OUT 4	Outputs to motors.
D1, D2, D3, D4	Output logic pins. Linked to outputs 1 to 4, respectively.
EN1	Logic pin that enables outputs OUT1 and OUT 2.
EN2	Logic pin that enables outputs OUT3 and OUT 4.

Table 1. L298N Drive Board Connections.

The OUT pins are intended to connect to motors. For example, OUT1+OUT2 could be combined to drive Motor 1 and OUT3+OUT4 would be combined to drive Motor 2. The logic pins, D1 to D4 and EN1 and EN2, control operation of the outputs. EN_1 must be "high", which means 5v, for the motor to be enabled, otherwise, the motor is allowed to coast freely. Operation of the board can be achieved via two schemes:

Scheme 1 (Always Enabled): This configuration requires two PWM pins per motor from the Arduino. In this scheme, the black plastic jumper caps on the EN1/EN2 pins are installed (which connects the EN1/EN2 pins to 5V). Supply PWM signals to both D1/D2 or D3/D4 to then control the direction and speed of the corresponding motor. For example, forward operation would be achieved by sending a PWM signal to D_1 and 0V to D_2. Reverse is just the opposite. If D_1 PWM is equal to D_2 PWM, then the chip operates as brake to stop the motor.

Scheme 2 (Enable based PWM): This configuration requires one PWM and two digital logic pins per motor. In this case, the plastic jumper caps are removed from EN1/EN2 and a PWM signal is sent to the enable pin to control speed. D1/D2 or D3/D4 are connected to digital I/O pins (which can be non-PWM pins or PWM pins) in order to change direction or braking of the motor. For example, sending a 2.5V PWM signal to EN1, HIGH to D1, and LOW to D2 will cause the motor to turn forward at half speed. Reversing the HIGH and LOW signals will reverse the direction of the motor. Changing the PWM value applied to EN1 will change the motor speed. Note that the motor will coast (rotate freely) if the PWM for the EN pin is set to zero. Braking is only possible if the EN pin PWM is greater than zero AND $D_1 = D_2$.

An example of connecting the motor driver board to the Arduino and motors is shown in Figure 3. This is based upon Scheme 1, where the EN pins have the jumper caps installed and only the digital logic pins D_1 to D_4 are connected to the Arduino. Four PWM pins are used in this configuration, which can be selected by the user. In this case, Pins 6, 9, 10, and 11 were used. Note that the motor driver, battery, and Arduino board all use a common ground (i.e. all of the ground wires are connected together), which is **critical to assure proper operation.** An example of connecting the motor driver using Scheme 2 is shown in Figure 4.

If you are using an L298 motor driver, we have made a library that is optimized for use with the Mega: *L298NMotorDriverMega.* The .zip folder for this library can be found on canvas. Our L298 Library uses the Timers on the Mega to generate 20kHz PWM signals, which is outside the audible range of normal humans. See the demo file for which pins to use on your Mega.

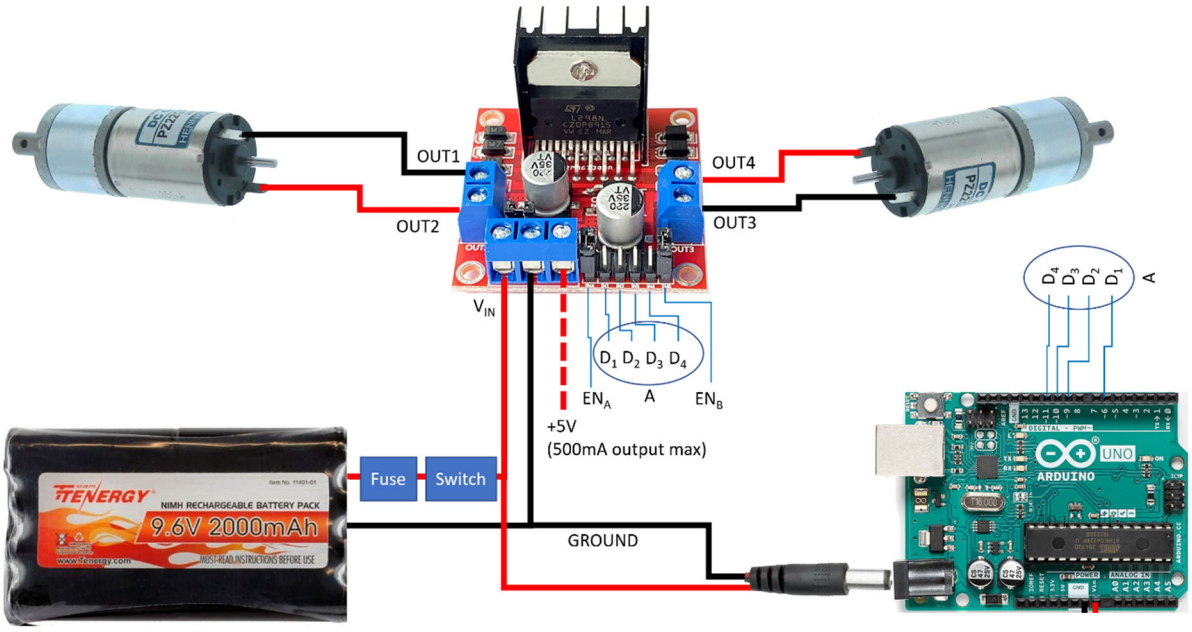

Figure 3. Example of L298N Motor Control Board connection with motors, battery, fuse, switch, and Arduino. Note that this using Scheme 1 since the EN pins are jumped and not connected to the Arduino.

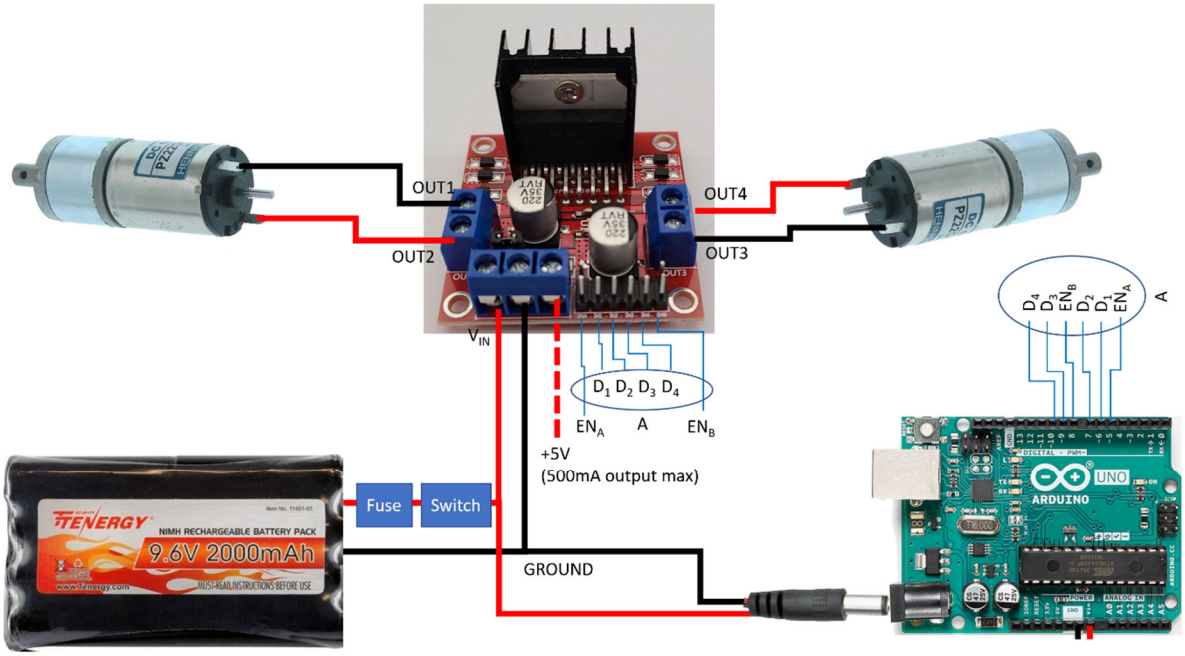

Figure 4. Example of L298N Motor Control Board connection with motors, battery, fuse, switch, and Arduino. Note that this using Scheme 2 since the EN pins are connected to the Arduino and the EN jumpers must be removed.

Pololu Dual TB9051FTG Motor Driver Shield

You will need to solder together the Motor Driver Shield to interface your robot's motors with the Arduino Mega. Please use proper soldering techniques for all electrical connections, and whenever possible, use heat shrink to provide insulation and strain relief.

1. Solder the blue terminal housings to the M1A/M1B, GND/VIN, and M2A/M2B positions. Be sure to make sure these connections are good, otherwise you may encounter power issues for the Arduino and DC motors.
2. Solder all four stackable headers through their appropriate connections on the board**.**
   NOTE: Do not trim the header leads as these are needed to connect the driver shield to the Arduino.
3. You can now place the shield on the Mega. Keep track of which pins the shield actually uses so you know which ones are available for other circuits. The default pins used by the Motor Shield are A0 and A1 for detecting current draw of each motor, D2, D4, D6, D7, D8, D9, D10, and D12 for motor control. Each motor has three pins that will be the most important for our project. Motor 1 uses D2, D7, and D9 to enable, choose direction, and dictate speed, respectively. Motor 2 uses D4, D8, and D10 for the same purposes. The remaining two pins are used for diagnostic purposes on each motor. Note that the speed pin MUST be a PWM pin so that you can provide a range of values. You can use the *DualTB9051FTGMotorShieldUnoMega* library when using a single motor driver shield. This library can be found on canvas.

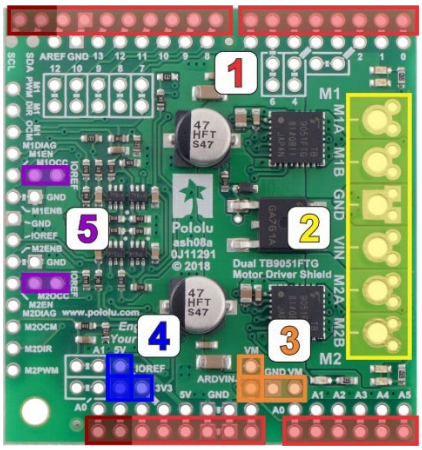

Figure 5: Red sections (section 1) are for soldering the stackable headers. The yellow section (Section 2) is for soldering the blue terminal housings.

If you are using the TB9051FTG motor driver shields, you may have already downloaded the DualTB9051FTGMotorShield Arduino Library from GitHub. This one will work okay, but it is actually optimized for use with an Arduino Uno[1]. We have made our own version of the library, which will work equally well on an Uno or Mega: *DualTB9051FTGMotorShieldUnoMega.* Furthermore, if you plan to use a second TB9051 motor shield, we have a modified version of the library that will allow you to control both shields (all four motors): *DualTB9051FTGMotorShieldMod3230.* Both of these libraries can be found in .zip format on Canvas. There is more information about using two driver shields at the same time in a following section of this document.

For any of these libraries, you can look in the examples folder to find demos of how to use them, and you can look at the keywords file and .cpp file to see what functions exist within the library and how they work.

Single TB9051FTG Motor Driver

There is also a Single TB9051FTG Motor Driver Carrier. This is a small circuit board that can be used to drive a single motor, rather than being capable of driver two motors. This board is not capable of being an Arduino shield, and therefore will sit somewhere else on your system rather than stacked on top of the Arduino board. You will assemble it in a similar way as the dual driver shield, and you will need to run similar motor commands to it.

[1] The original TB9051 library from GitHub assumes you are using an Arduino Uno, which uses Timer 1 to generate high frequency (20kHz) PWM signals on Pins 9 and 10. If you use an Arduino Mega, the timers are on different pins and the library will default to using the AnalogWrite command, which results in a slow PWM frequency of 490 Hz. It will work fine, but 490 Hz is in the audible range of normal humans and will make your motors emit an annoying whining noise. Our modified libraries use Timer 2 for pins 9 and 10, and Timer 5 for pins 45 and 46, so all four motors can be driven with high frequency PWM signals that are outside the audible range of normal humans.

Power Components

Battery

The battery used in this course is a 9.6V rechargeable battery. You will need a battery charger and adapter cable to go with it. You can also purchase (and we recommend doing so) a cable that has the white, plastic, Molex connector clip on one end and open wires on the other end. This will provide you with a way to solder a battery connection to your system, but still have an easy way to disconnect the battery.

When mounting the battery be sure to secure it properly so it doesn't slide around or dislodge during operation. You'll want to make sure the cable is in an accessible area so you can disconnect it when you need to power down the system or charge the battery. Along those same lines you might want to make the battery easily removable in case you want to take it out while it charges. If you get two batteries this is definitely important as you'll swap them in and out of your system while one is charging and the other is being used.

Power Switch and Fuse

A power switch will be used as the master switch to turn on/off power supply from the 9.6V rechargeable batteries to all components of the robot. This is a 'hard' switch that physically cuts power to the robot. Consider where on the robot you will want to place the master power switch. This switch will be used to turn on/off all power to the mobile robot and should be easily accessible, likely mounted on, and sticking through, the robot chassis cover.

You will need to create a fused connection to provide power from the 9.6V rechargeable battery to your system. Motors and sensors can easily be damaged or destroyed if too much current is passed through them. To prevent this, fuses are often used and we **REQUIRE** all robots to be fuse protected. Fuses are typically made of an inexpensive low resistance material that can be sacrificed to provide overcurrent protection. A variety of blade type fuses are available in the laboratory (1A, 2A, 3A, 4A, & 5A) along with fuse holsters. You will protect the Arduino and Motor Driver Shield with a fuse of up to 10A max.

Buck Converter

A buck converter is an adjustable DC-DC voltage regulator, Figure , which is based upon a LM2596S IC. You supply a power source on the input side and you can vary the voltage output. The buck converter can vary voltage from 1.5 V up to the battery input voltage (35 volts max) and can provide up to 3A. The buck converter is larger than a LM7805, but it is more efficient and is adjustable via the potentiometer. Connections are made via the "in" and "out" pins indicated on the board. Again, a common ground needs to be used across the system to assure proper operation. An example of the buck converter combined with a servo and the L298N is provided in Figure 7.

When you first set up your buck converter you need to adjust the screw on the potentiometer and measure the output voltage until it reads the 5V or 6V output that you desire.

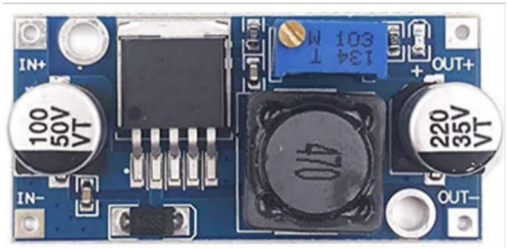

Figure 6. Ziktex Adjustable DC-DC buck converter. Amazon item B07VVXF7YX.

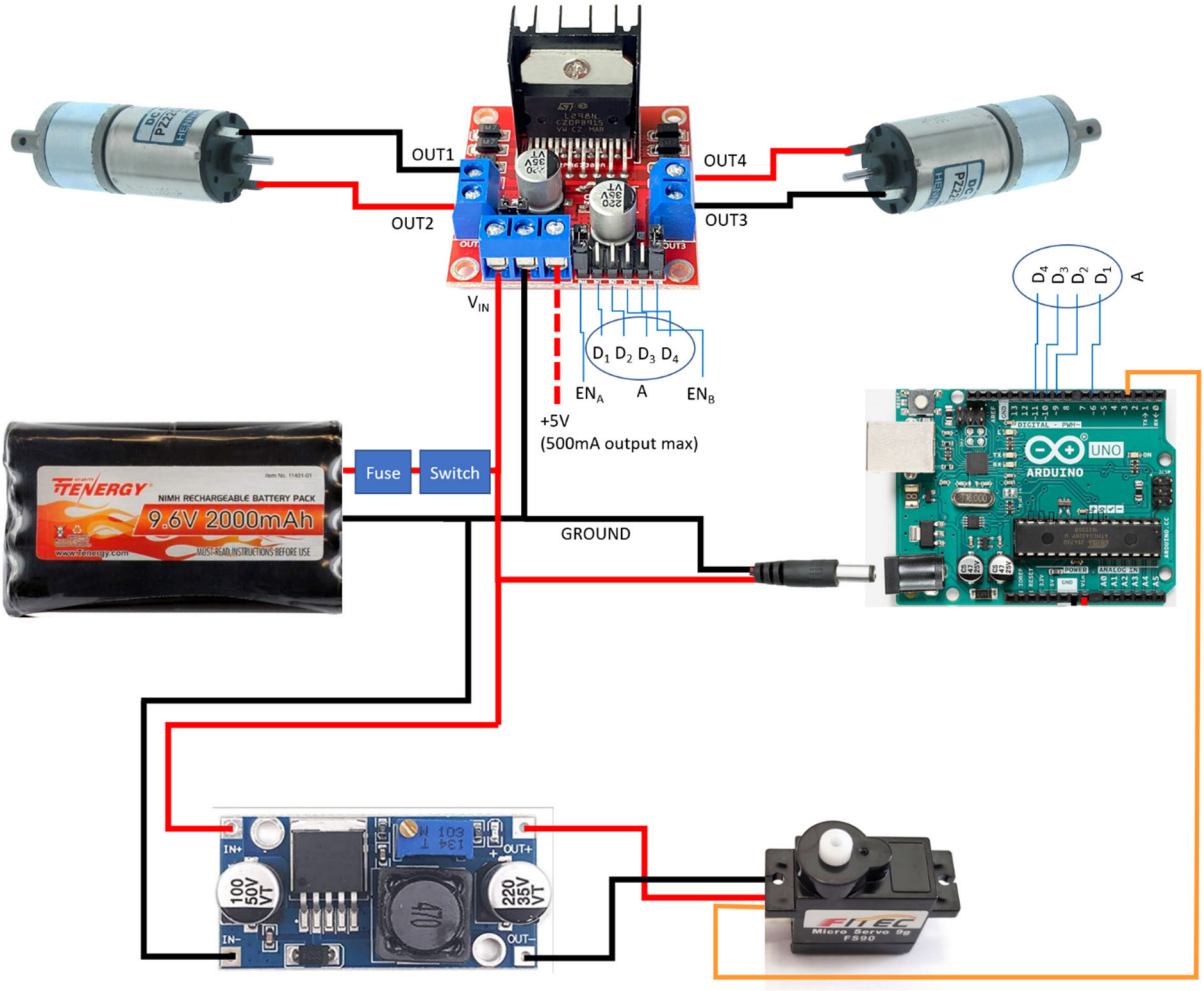

Figure 7. Example of buck converter and servo combined with L298N Scheme 1 connection.

LM 7805 Voltage Regulator

LM7805 5V regulators are another DC-DC voltage regulator option that specifically output 5V always. You can look up the details about this regulator but there are a few important things to know:

- Requires a voltage supply greater than 7V (your battery)
- Has a current limit (1-1.5 A)
- Requires additional circuitry to keep the signal clean.

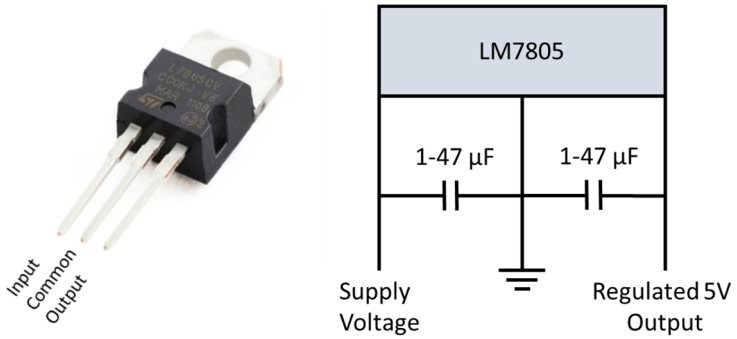

Figure 8. LM7805 Voltage regulator and simple circuit to wire the regulator.

The capacitors are used to keep the output signal clean. They remove some noise. You can choose a set of capacitors that work for you. Capacitors in the range of 1 µF to 47 µF are suggested. These will primarily be electrolytic capacitors, so be careful which direction you connect them. The higher the capacitor the cleaner the signal, but you sacrifice overall voltage. You have to choose which works best for your application. The LM7805 is rated for up to 1.5 A.

These chips produce a decent amount of heat while operating. It is suggested that you provide an outlet for the heat. This can be done in a variety of ways including but not limited to: screw on heat sink, mount it to a metal portion of the body, fan powered convection, or a combo of these. The chips have overheating protection. This means they will shut themselves down if they are getting too hot, but that means they won't regulate the voltage so try to keep them cooled down. See the datasheet for proper operating temperatures.

Connecting Power to your Arduino

USB Connection

There are multiple ways to connect power into your Arduino board. The most used way for simple hobby projects is likely the USB port that connects the board to the computer. Since the systems built in this course will not be connected to a computer, you will not use this method to power your main system (you might use this method to power a secondary Uno that is a transmission device to your main system).

Barrel Connector

The second most obvious method is the barrel jack connection on the Mega, located right next to the USB port. This is a great way to power your Mega because it is a robust, consistent connection that is easy to connect/disconnect. This power jack also has protection hardware built into the board to prevent damage to the Mega if something goes wrong with the power while using the barrel jack connection. A male barrel

connector can be purchased from the lab that can be connected to your battery power system, and then plugged into the Arduino for power.

Vin Pin DO NOT USE

The third way to supply power to an Arduino is to connect wires right into the Vin and GND header pins on the board. **THIS IS WAY IS HIGHLY DISCOURAGED IN THIS COURSE.** This method has no internal hardware protection, so any problem with the power will destroy your Arduino. Because there is no onboard protection hardware on the Vin pin it is also possible to damage your computer if there is a power surge on the board while it is connected to your computer via a USB cable. Lastly, since this method requires you to plug wires into the board, there is a higher likelihood that the wires could be plugged in backwards (applying 9V to the GND) which immediately burns up the Arduino and possibly any sensors or computers connected to it.

Motor Driver Shield

The fourth and final way to power your Arduino board is through the motor driver board. The Dual TB9051FTG motor driver shields are capable of powering Arduinos when they are assembled and installed as a shield. The key to this method is two male header pins on the shield that can be jumped together with a small blue header jumper (shorting block) that comes with the board.

In the shield's default state, when the blue jumper is disconnected, the motor driver shield and Arduino are powered separately. Power from your battery is connected to the blue screw terminals labeled Vin and GND on the shield to power the shield, and either the USB or barrel jack is connected to power the Arduino. Trying to power the shield with the Vout from the Arduino can permanently damage the Arduino and motor driver.

However, you can place the blue shorting block across the pins labeled VM and ARDVIN to provide the shield's reverse-protected power, VM, to the Arduino's VIN pin. **THE ARDUINO'S POWER JACK MUST REMAIN DISCONNECTED AT ALL TIMES IN THIS CONFIGURATION**. This method

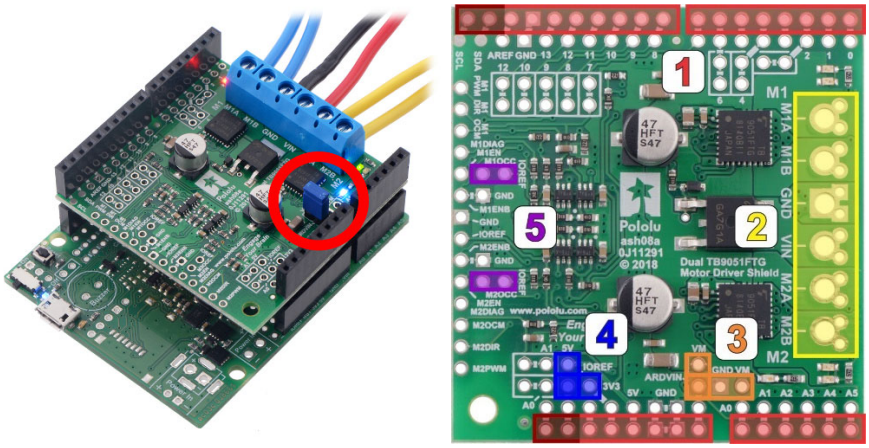

Figure 9. Left: Dual TB9051FTG Shield with blue shorting block installed (circled in red). Right: The blue shorting block goes across the VM and ARDVIN pins highlighted in orange in location 3.

will send power to the Arduino on the Vin and GND pins, which was previously mentioned as being bad. However, the shield solves the two problems about providing power on the Vin pin.

The first problem is lack of hardware protection. But the driver shield has that protection hardware built-in and solves that problem for us. The second problem is the possibility of user error when plugging in the wires. The shield also solves this problem because the user is not plugging any wires into the Arduino Vin pin, the shield makes that connection for us. Therefore, this method is a safe and reliable method to power an Arduino if you are using the Dual TB9051FTG Motor Driver Shield. More information should be read in the user's guide for the driver on Pololu's website: https://www.pololu.com/docs/0J78/3.c

Power Distribution Board

Power distribution boards are often used in mechatronic systems to provide easy access points for the different voltage busses in the system. For example, the typical mechatronics mobile robot uses ground, a 9.6 V battery, 6 V for servos, and 5 V for sensors and possibly servos. A power distribution box basically creates a rail (or bus) with a number of terminals to connect additional components. The recommended procedure for implementing a power distribution board includes the following:

1. The main input to the power distribution board will be from the 9.6V battery. Get a battery connector with wire leads to connect your battery to the board. Do not cut the wires that are coming out of your battery.
2. The ground cable coming from the battery (black) should be connected to the power distribution board. Install a row of female header pins that are all connected to each other and connect the GND from the battery to it. This way you have a centralized location, with a bunch of header pins, that all your components' GND connections can be plugged into.
3. In line with the positive wire lead coming from the battery (red) you will connect your power switch and fuse.
    a. Use crimp connections for this. Soldering directly to the switch or the fuse can damage the components, and prevents you from easily disconnecting the components in the need to replace one (like if you blow a fuse).
4. Install another row of female headers to your board and connect this main 9.6V supply line to it. You now have 9.6V and GND rails available to plug wires into to power your Arduino and Motor Driver shields.
    a. You can power the Arduino from the 9.6V rail by using a barrel connector (available for purchase from the lab), and also run 9.6V to the driver shield.
    b. You can also run 9.6V only to the shield, and power the Arduino through the shield.
5. From the location on your power board where you have 9.6V/GND, attach a buck converter or voltage regulator (or both). Take the output(s) from these regulators and run them back to the board in new locations. Install female header pins just as you did before. This will now provide a location where you can connect 5/6V power to your components with a simple header pin connection. As long as you install enough header pins, you will have plenty of space to connect many sensors/servos.

The end goal is to have a single circuit board that has rows of header pins providing you with ample locations to connect your GND wires, 5/6V supply wires, power wires to your motor drivers, and a barrel connector to power your Arduino.

Often it is good to power servos with 6V supply. If you have servos in your system, you may want to create a 6V rail on your power distribution board. If you don't have servos in your system, you don't need to do this (and therefore only need a single buck converter).

To create a 5V line you can either use a buck converter or a 5V regulator. We recommend the buck converter because it is a little more robust and will provide a cleaner supply voltage. If you have both a 5V and a 6V rail, you could use two buck converters, or a buck converter and a 5V regulator.

Two examples of how you could potentially create a power distribution board are shown in Figure 10. You could install two sets of female headers next to each other to be power and GND for each voltage (shown on the left) or you could create one large GND rail and then independent voltage rails (shown on the right). Either way is fine, and there are other ways as well, it is up to you how you would like to design your board.

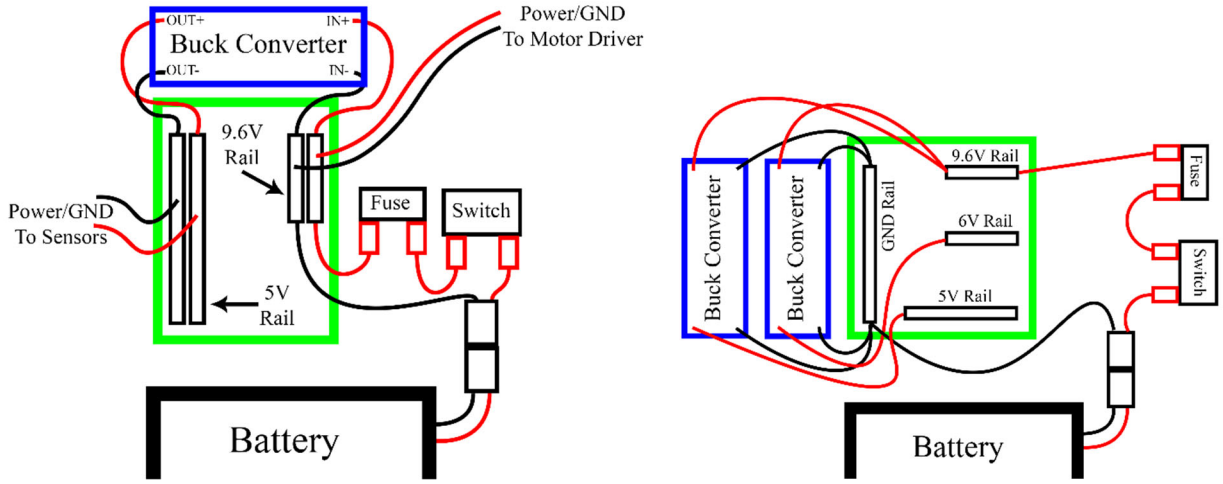

Figure 10: Two examples of how to structure a power distribution board.

Sensors

The objective of this section is to provide you and your team with information regarding the sensors that you will likely be using for your robot. Each of these sensors will be explored more thoroughly via lab assignments during this semester. However, it will be useful to familiarize yourself with these devices. In particular, understanding how the sensors operate and the Arduino pins necessary will help you in designing your robot.

IR Rangefinder (Distance) Sensor

These sensors connect to an analog input pin on the Arduino, as well as 5V and GND.

We recommend and sell IR Rangefinders for distance sensing. These are recommended instead of sonar distance sensors. There is a lab that goes over the intimate details of these rangefinder sensors. You can also look into the datasheets available at https://www.pololu.com/file/0J713/GP2Y0A41SK0F.pdf . There are two available options, one that measures distance in the range of 2-15cm and one that measures a range of 4-30cm. They can't accurately measure anything that is too close so be sure to mount these sensors in a location where they won't be trying to measure anything that is closer than the defined range. A picture of the 4-30 rangefinder is shown in Figure 11.

Limitations of Rangefinders

While the Sharp IR rangefinders are simple to use, accurate, and much less noisy than a sonar sensor, they have their limitations. The most important limitation to know is their nonlinear response. Unlike a sonar sensor where distance and time of flight are linearly related, distance and angle of incidence have a nonlinear relationship and the sensor does not correct for this. They also have a minimum detection range, where below this distance objects will appear farther away than they really are. There is also the potential for interference if two or more sensors are pointed at each other. Another problem is with semi-transparent objects, which unpredictably reflect light. In this case, the range reading will be inaccurate, but the sensor will usually detect the presence of an object. Two other issues to be aware of are: (1) excess ambient IR

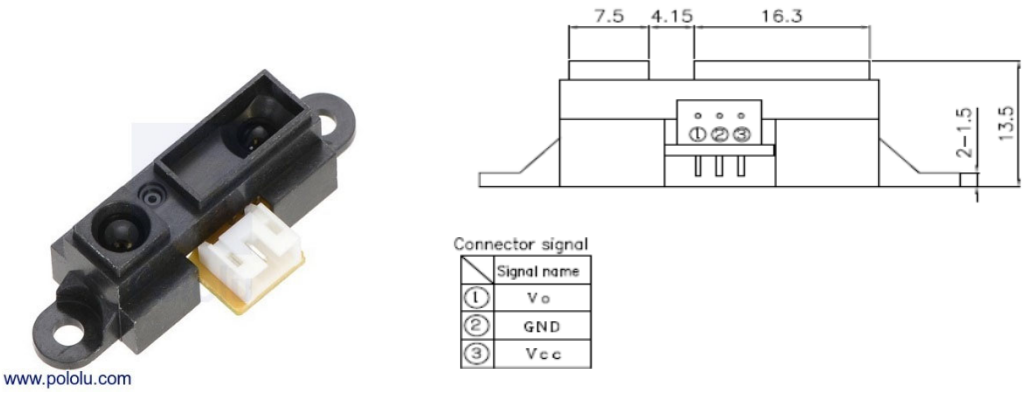

Figure 11. Sharp GP2Y0A41SK0F Infrared Rangefinder from Pololu, with dimensions and sensor connections.

light can saturate the PSD making it unable to detect the reflected pulses, and (2) the response curve has small temperature dependence. Details on these issues can be found in the sensors' datasheets.

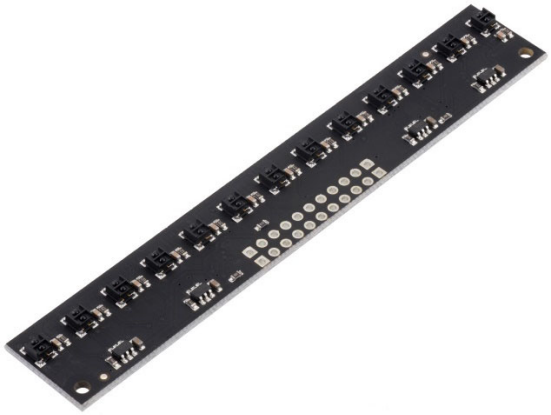

Figure 12. QTRX-MD-13RC Reflectance Sensor Array from Pololu.

IR Reflectance Sensor Array (Linefollowing)

These sensors connect to some amount of digital input pins between seven and thirteen (depends on the model), a digital output pin, as well as 5V and GND.

Once again, there will be a lab that goes over the details of these sensors, but in this course they are generally used for line following. These sensor arrays take a bunch of individual reflectance sensors and put them in a line next to each other. There are a few options to purchase in this course: a low density 7-sensor array, a low density 8-sensor array, a high density 11-sensor array, and a high density 13-sensor array. The difference is how many individual sensors are placed next to each other, and how closely they are spaced. With a high-density array, you can detect small side to side changes in a line if you need precise line locations. The number of sensors next to each other, and their density, will determine how wide the sensor array is. The wider it is means you have a wider line detection area. Considering they have a different number of sensors, the different models will be different physical sizes so be sure to think about adjustability in your mounting options for this sensor array.

When using these as line following sensors, they should be mounted to the mobile robot on the underside, pointed at the ground, with the face of the sensor being 3mm off the ground. This is a very small distance, but for them to work consistently that is the recommended sensing distance. They need to be mounted in front of the driving wheels (in the direction of motion) otherwise turning will make the sensor move the wrong way. If the distance between your drive wheels is *d*, then the optimal location for a line following array is a distance of *d/4* in front of your wheels.

To learn more about how to use and operate this sensor visit:
https://www.pololu.com/product/961.

Hall Effect Sensor

These sensors connect to an analog input pin on the Arduino, as well as 5V and GND.

Hall Effect sensors are quite small and have no mechanical mounting options. Therefore, you will need to design and fabricate a mechanical structure that holds the sensor in position by incorporating the electrical connections either with mounted pins that the sensor plugs into, or by supporting soldered connections. This sensor is sensitive to distance, so you will want to design its mounting location to be as close as possible to the object that you're sensing.

NOTE: To use the Hall Effect sensor a bypass capacitor (100nF) must be connected across the Vin and GND leads of the sensor. Without this capacitor, the sensor will be damaged.

Practical sensor circuit

Since the Hall Effect sensor has a small sensitivity, it is practical to amplify the signal. To keep the output in the range of 0-5V the reference of the amplifier has to be altered to output 2.5 V (quiescent voltage) when not in the presence of magnetic fields. When you amplify a signal, you will also amplify the noise therefore it may also be beneficial to filter the final output. This amplification and filtering will be discussed during the hall effect sensor lab. You will construct any necessary circuitry for your system, and it will need to live on your robot somewhere, so make sure to design some space for it.

Figure 13. Allegro Hall Effect sensor.

Another consideration when implementing a Hall Effect sensor, especially on a mobile mechatronic system, is whether other components of your system can interfere with the sensor. Specifically, motors can potentially create an electromagnetic field when cycling current through the actuator. If the Hall Effect sensor is close in proximity to such an actuator, this field may be detected by the sensor, providing incorrect measurements. Therefore, you should take care in planning the placement of your Hall Effect sensor so that they have minimal interference from sources of disinterest.

Color Sensor

This sensor will require 3 digital output pins, 1 analog input pin, 5V, and GND.

The color sensor is something you will build yourself from a few electrical components, and mount in your own custom housing. There are 2 main components in the color sensor: the RGB LED, and the phototransistor. These are shown in the figures below. You will need to manufacture a housing or some

structure that holds both components and points them toward an object. The idea is to illuminate the object with the LED, the light will bounce off the object, and then the phototransistor detects how much light is reflected. You should include something as a divider between the LED and the phototransistor so the light from the LED doesn't go directly into the phototransistor. This is important to guarantee that light from the LED only reaches the phototransistor if it bounces off the object. Another consideration is to surround the LED and phototransistor with some form of a shroud/shield to prevent ambient light from getting into your sensor. An example of a 3D printed housing design with a dividing wall and ambient light shroud is shown in the figure below.

This sensor should be mounted so that it is close to the object you are sensing, about an inch or less away. Preferably the sensor housing would be right up against the object so that the least amount of ambient light is getting into the sensor, and only the RGB light that's bouncing off the object is being detected.

A circuit diagram of the color sensor components is shown in Figure 14. A 5V power source is connected to a single 330 Ω resistor, which then connects to the common anode lead of the RGB LED. The resistor limits the current passing through the LED to prevent damage. The three cathode wires of the LED then connect to the digital connections of the Arduino, labeled JP2. When the pins are set to HIGH (i.e., 5 volts), no current flows through the LED. When a pin is set to LOW (i.e., 0 volts), current flows through the LED and the LED illuminates.

The phototransistor is also shown in the circuit diagram. It uses a "pull-down" resistor configuration where the 5 volt supply is connected to the phototransistor, which is then connected to the resistor. The phototransistor and resistor act like a voltage divider, where the voltage at their junction is connected to the A0 analog input of the Arduino. This is on the JP1 header as shown. When the phototransistor is not exposed to light (i.e., in darkness), the transistor does not allow current to pass and A0 measures ~0 volts. When exposed to light, however, the phototransistor allows current to flow, which raises the voltage measured at A0. The brighter the light, the higher the voltage measured at A0.

The RGB LED in this example is a common anode design. There is also a common cathode RGB LED. Anode means the positive side of the LED, cathode means the negative side. If you are using a common anode LED, as the diagram shows and like we do in the lab procedures, the long lead is the positive side of the LED and needs to be connected to 5V. If you are using a common cathode LED the long lead is the negative side and needs to be connected to GND. These two different types of LEDs are programmed opposite from each other because the power connections are opposite. With a common anode LED you set the digital pins to LOW in order to turn the LED on. With a common cathode LED you set the pins to HIGH to turn the LED on.

Also be careful with which way you wire the phototransistor. Like an LED, if you wire it backwards it won't work properly. An indicator that a phototransistor isn't working properly is the readings you get from it are always low regardless of how much light is shining on it (less than 1V, or an analog reading on the Arduino of less than 100). When you shine a light on a properly functioning phototransistor you should get a voltage reading of around 5V (or around 1000 on the Arduino). One way to troubleshoot to see if your phototransistor is working is to connect a multimeter to the point between the phototransistor and resistor, apply power and GND to the system, then read the voltage when you shine a light on the phototransistor You should read close to 5V when a bright light is placed right near it and should read about 0V if you

cover it up in darkness. If you don't get these readings, it is possible the phototransistor is wired backwards or is broken.

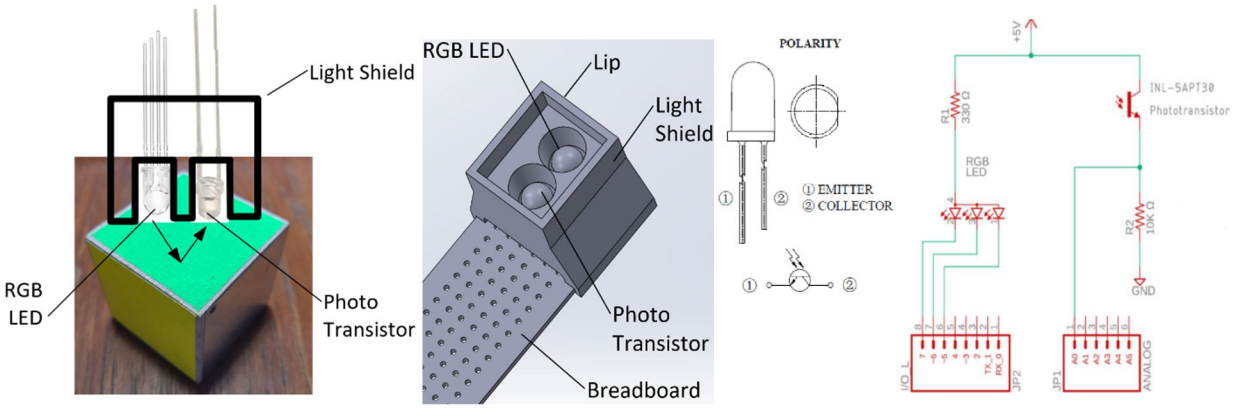

Figure 14. Illustration of color sensor (left), the color sensor light shield (middle), the color sensor electrical connections to an Arduino (right).

One last consideration is if you create a shroud to go around your sensor, use a dark colored material that doesn't reflect much light. If you 3D print a white housing it will light up and affect the phototransistor readings.

Motor Encoders

The motors used in this course feature three parts: the motor, the gear box, and the encoder. On the 37mm diameter motors the encoders are directly attached to the DC motor shaft (underneath the black plastic cap on the end). On the other motors you will need to purchase the encoders separately and attach/solder them to the metal tabs and small motor shaft on the back of the motor. An Arduino library has been developed by the company PJRC to count encoder pulses from encoded signals (www.pjrc.com/teensy/td_libs_Encoder.html).

This library is used to create an encoder object in your program with which you can keep track of how many counts the encoder has reported. Using this library is important because the speed at which the counts from the encoder show up is generally faster than the processing time of your code. The library implements interrupts to keep track of the counts without losing any. You will see during the encoder lab that if you don't use interrupts you will not get valid readings from your encoders. For the Arduino Mega, the dedicated interrupt pins are: 2, 3, 18, 19, 20, and 21. You only need 1 interrupt per encoder, using 2 interrupts per encoder is useful but not necessary.

Once you know how many counts the encoder has read, you can convert that number into the position of the gearbox output shaft by using the number of counts per revolution of the encoder (found in the specs for any encoder) and the gearbox ratio. For example:

$$100 counts \times \frac{1 MotorRev}{64 CountsPerRev} \times \frac{1 GearboxRev}{50 MotorRevs} = 0.03125\ OutputShaftRevs$$

Note: The encoder is a relative position measurement device, rather than absolute. This means that every time you turn on the encoder, whatever position the motor is currently in is considered the 0 position. Then when the motor moves, the encoder tells you the movement *relative* to that initial position. If you have an encoder on a motor that moves an arm and you turn on the system while the arm is pointing straight upwards, then straight upwards becomes 0 degrees and horizontal becomes 90 degrees. But if you turn off the system, move the arm manually to a horizontal position, and then turn the system back on, now horizontal is 0 degrees and pointing upwards is 90 degrees. You need to account for this either in your design, or in your program. One way to do so is to say "The specific position doesn't matter as long as it rotates 360 degrees" like with a conveyor belt. Another way would be to put in a limit switch and say "every time the arm rotates downwards and touches the limit switch, that position will be our home position of 0 degrees" and in your code you would set your encoder to 0 whenever the limit switch is triggered.

Limit Switches

Limit switches should be placed in a way that they won't be crushed if a component fails to stop moving in time. As an example, if there is a limit switch on the front of a driving system to detect when it bumps into a wall, the limit switch should be recessed into the bumper of the system so that if the system doesn't stop when it hits the wall the limit switch doesn't get crushed. Another example would be for a rack and pinion mechanism that uses a limit switch to determine when the rack has moved its full length. Placing the limit switch off to the side, and having a small tab move past the limit switch to trigger it makes sure that if the rack continues to move for some reason the limit switch won't get crushed.

Custom Sensor Breakout Boards

Building a custom sensor breakout board can be helpful in minimizing the amount of wiring on your system. This can reduce the possibility of a "rat's nest" of wires, which is hard to work with, hard to troubleshoot, and can be visually unappealing. The point of a sensor breakout board is to provide the sensors with self-contained landing locations on a circuit board. Each landing location has a set of header pins that provide power, ground, and a command signal to the sensor, while receiving the sensor reading from it. This way all the wires for a sensor can all land on the board in the same place, keeping them tidy, rather than splitting up the wires and sending the power wires to one location and the signal wires to another location. Figure 15 (left) shows an example of a sensor breakout board.

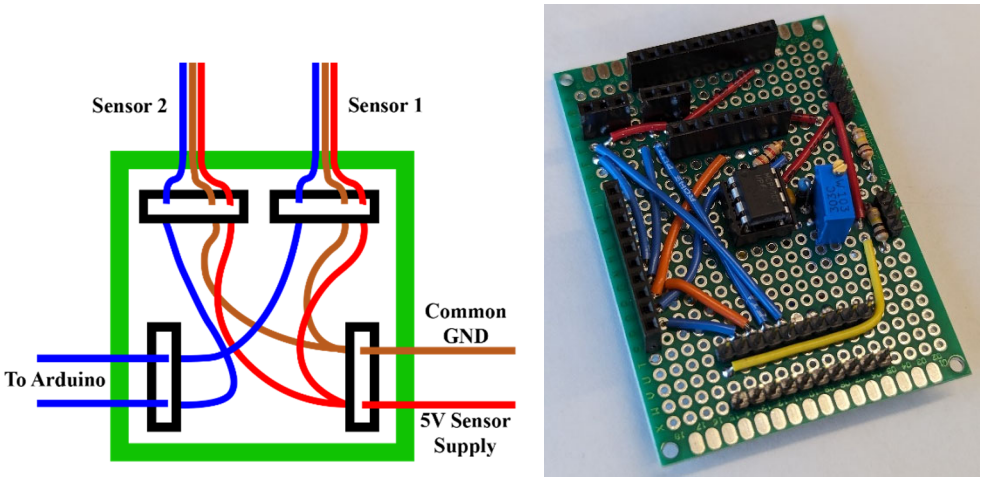

Figure 15. Left: Diagrammatic sensor breakout board example for two sensors. Right: Assembled breakout board for multiple sensors (with onboard amplification circuitry).

Power and ground are landed on the board on header pins on the bottom right, then distributed to headers on the top where the sensors connect. The signals from the two sensors are wired from the sensor header pins to header pins on the bottom left which have a pair of wires running back to the Arduino. This way the complex wire routing of power and signal to and from the sensors is all done on the circuit board where it is nice and contained. The benefits of a sensor breakout board increase with more sensors because you can run a ribbon cable from the sensor board back to the Arduino with up to 10 sensor signals at once. Although with more sensors the wire routing on the board can become challenging. Lastly, this helps with troubleshooting because by using headers for landing your sensor wires it's easy to connect/disconnect any sensor at will. And to troubleshoot electrical problems you only need to inspect your breakout board rather than chase down stray wires that run all over your system. A picture of a fully assembled sensor breakout board is provided in Figure 15 (right).

Wiring

Heat Shrink

Include heat shrink on each soldered connection made between wires, headers, or a combination of the two. This prevents any exposed metal/wires from accidentally touching something and shorting out any of your components. It also provides extra strength to your connection (which is called wire strain relief). In case anything tugs on the wire, the heat shrink may prevent the solder joint from being ripped apart. An example of wires soldered to a row of male headers, with heat shrink on each solder joint is shown in Figure 16.

Crimp Connectors

There are crimp-on wire connectors available for use. These connectors crimp onto the end of a wire by using a crimping tool. If you crimp a male connector to the end of one wire, and a female connector to the end of another wire, you can then quickly and easily connect/disconnect those wires whenever you want. This is a much more robust way to connect two wires than soldering/twisting/taping them together. These connectors are shown in Figure 16.

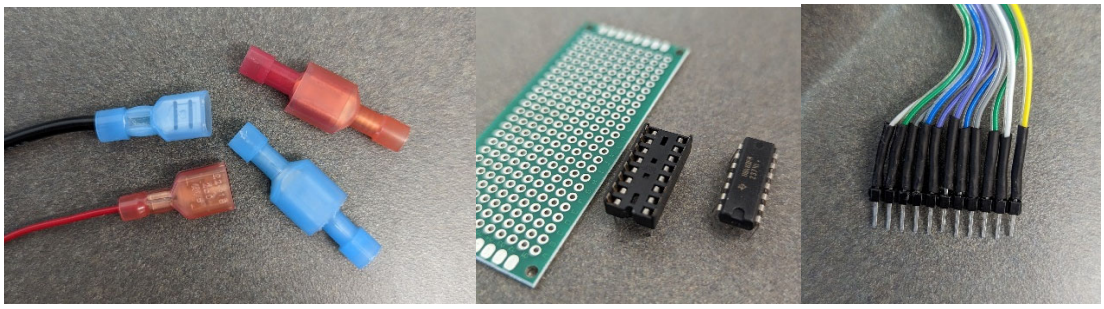

Figure 16. Left: Crimp on wire connectors. Middle: A protoboard, a chip socket, and an IC chip. Right: Soldered connections between wires and a row of male headers.

The female crimp connectors also fit the blade connections on the power switches and fuses. Therefore, you can connect a female crimp connector to a wire, and then connect that wire to the switch or fuse. This is the recommended way to connect your fuses and switches to your system. Soldering a wire directly to a switch or fuse allows the possibility of damaging the component with the heat of the soldering iron. Using crimp connectors removes that danger. It also allows you to remove the switch/fuse in case you want to move the component or replace it (like if you blow a fuse).

Protoboards

Protoboards can be used, and are recommended, for things like a sensor breakout board, a power distribution board, or a custom hall effect & color sensor circuit. There are IC chip sockets that work as holders for electrical chips like op-amp chips that will be used for amplification circuits. These sockets can be soldered to protoboards to provide a location for your IC chips, while reducing the danger of damaging the chip by directly soldering it. They also allow you to pop the chips in and out of your circuit in case you want to switch chips, or you damage the one you're using. An image of a protoboard, an IC socket, and an IC chip are shown in Figure 16.

You should mount any circuit board (including your Arduinos!) to a frame/base using the four mounting holes around the perimeter of the board. Use spacers to lift the board slightly off the surface it is mounted to so the back of the board (with its exposed metal solder joints) doesn't touch anything and accidentally short out any of your connections or damage the board. Also make sure to mount/place circuit boards (especially your Arduinos!) in locations where you can easily access the pin connections (and USB port) when your system is fully assembled. There's no need to design your system in such a way that it is hard to access your electrical connections.

Headers

Using male and female headers will improve the robustness, connection strength, and adjustability of your electrical system. Female headers can be soldered into protoboards to provide your custom circuitry with similar pin connections as what are seen on Arduinos. Male headers can be soldered to wires and to provide a stronger pin to insert into a female header. If you solder multiple wires to male headers, like a ribbon cable with 8 connections coming out of a line sensor, solder them to a set of male headers that are all still attached to each other, rather than soldering each individual wire to an independent male header. By having a row of male headers that are all still connected to each other, they are more structurally sound and will give you better connections to your female headers. This is shown in Figure 16.

Strain Relief

The wires that come out of the 37mmD motors can be zip-tied to the body of the motor to provide strain relief to the wires. If something tugs on the wires, it will tug on the zip-tie instead of ripping the wires out of the encoder. This is another form of strain relief that can save your electrical connections from coming undone. Whenever possible we recommend implementing strain relief to your wires.

Soldering

A few notes about wiring and soldering procedures. When stripping wires, strip the minimum amount necessary. You don't need to strip two centimeters off the end of the wire just to solder a millimeter of it into a protoboard. Stripping more wire than necessary then requires more heat shrink to cover up or can result in exposed wire that can short out your components. Also, when you solder wire or components to a protoboard you should snip off the excess length of the leads sticking out from the bottom of the protoboard. This once again results in less places that can accidentally contact something and short out your system.

Other

Label your wires somehow. This way if a connection comes loose from your Arduino, or power distribution board, or wherever, it will be easy to know which wire it is and where it needs to be plugged back in.

Figure 17 shows some examples of wiring, protoboards, heat shrink, using header pins, using IC chip sockets, and making soldered connections when building custom circuitry on protoboards.

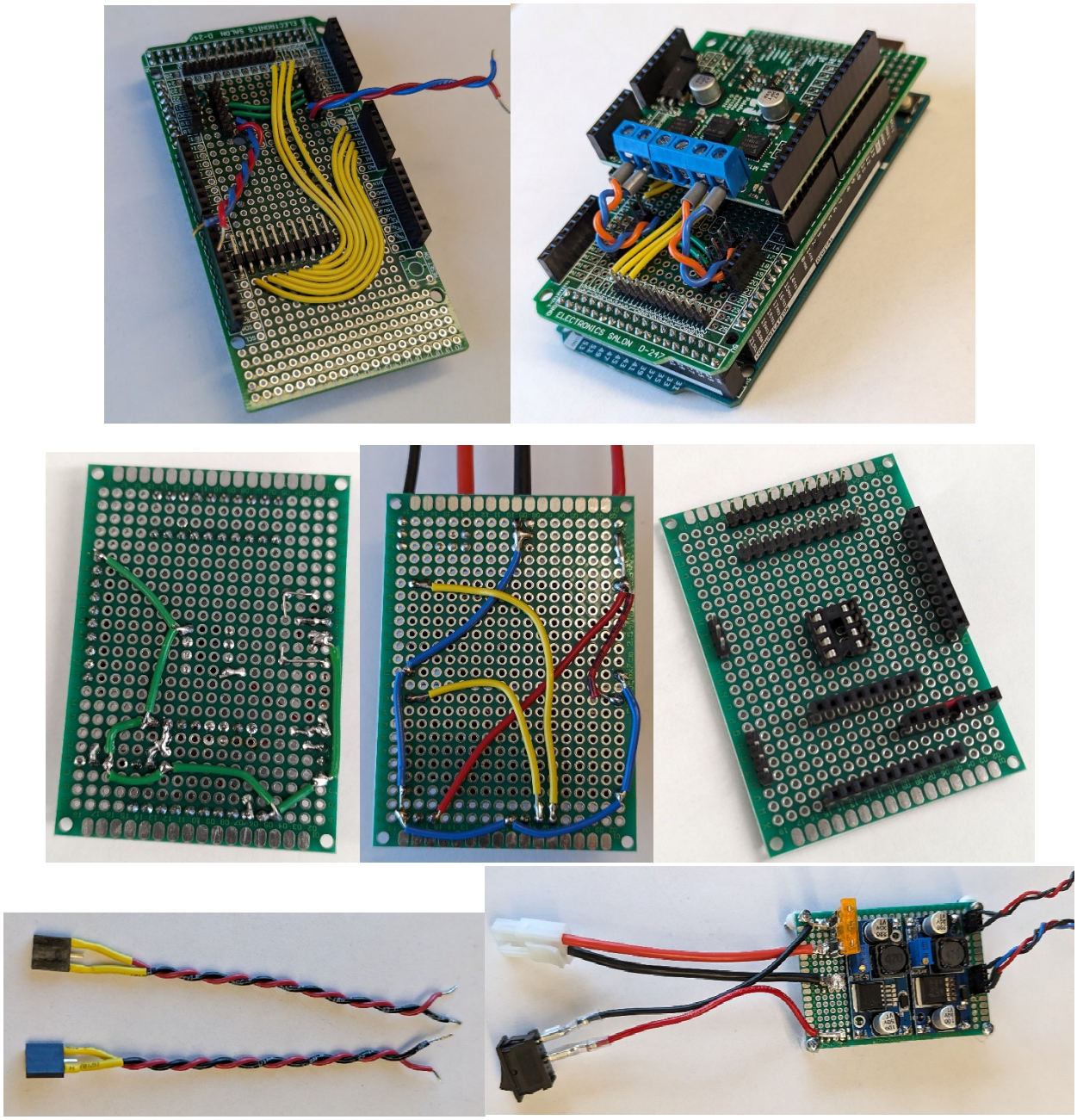

Figure 17. Electrical examples.

Pin Considerations when using the Arduino Mega

PWM Pins: 2-13, 44-46

These pins are capable of outputting a PWM signal for commanding motor drivers and servos. The DualTB9051 Motor Driver Shields are wired to use pins 9 and 10 for PWM (which are associated with Timer 2 when using the *DualTB9051FTGMotorShieldUnoMega.h* *DualTB9051FTGMotorShieldMod3230.h* Arduino Libraries), while using pins 2,4,6,7,8,10,12 for direction/enable/diagnostics. These latter functions don't require PWM so it's unfortunate they are using up PWM pins. You can remap them to non-PWM pins if necessary by rewiring your motor shield (see the Project Construction Guide on Canvas). If you use a 2nd Motor Driver shield, we recommend using pins 45 and 46 for PWM (which are associated with Timer 5 when using the *DualTB9051FTGMotorShieldMod3230.h* Arduino Libraries), and using non-PWM pins for direction/enable.

Table 2 below shows potential conflicts when using the various PWM pins. For example, if you use Timer 5 for a 2nd TB9051 Motor Shield, then you will have a conflict with the *servo.h* library. We recommend using the *PWMservo.h* library instead, which allows you to connect servos to pins 11,12 and 13. The *PWMservo.h* library also has the advantage of not using interrupts, so it will not conflict with other processes that use interrupts. Therefore, we recommend using *PWMservo.h* for controlling servos in any case.

Table 2: PWM pins and Timers on the Arduino Mega

Timer	Bits	Pins	Conflicts
0	8	4,13*	Timer 0 is used by time functions such as *millis()*, *delay()*
1	16	11,12	Timer 1 is used by the *PWMservo.h* library
2	8	9,10	Timer 2 is used by sound functions such as *tone()*
3	16	5,2,3	Pins 2 and 3 are useful for hardware interrupts
4	16	6,7,8	These pins are used by the TB9051 motor shield by default
5	16	46,45,44	Timer 5 is used by the *servo.h* library

*Also note that Pin 13 is connected to the LED that flashes when the Arduino resets, so this may cause a jerk in your motor if you put a motor on this pin

Interrupt Pins: 2,3,18-21

These pins are capable of triggering interrupts, and should be reserved for encoders if possible. This will ensure you don't miss encoder counts. Each encoder can either use 1 interrupt pin + 1 non-interrupt pin

(good performance) or 2 interrupt pins (best performance). The *encoder.h* Arduino Library will use interrupts if you wire your encoders to interrupt pins, and will do all the counting for you.

Table 3: Interrupt pins on the Arduino Mega

Interrupt	Pin
0	2
1	3
2	21
3	20
4	19
5	18

Serial Communication Pins: 0,1,14-19

These pins can be used for serial communication. Serial channel 2 (pins 17,16) or Serial channel 3 (pins 15,14) would be a good choice for wireless communication with the XBees. The *SoftwareSerial* library allows you to use any digital pin for serial communication, but it interferes with interrupt usage, so it is recommended not to use *SoftwareSerial.*

Table 4: Serial Pins on the Arduino Mega

Serial Channel	Pins (Rx,Tx)	Conflicts
0	0,1	Used by USB port (serial comm with your PC)
1	19,18	These pins are useful for interrupts (save for your encoders)
2	17,16	
3	15,14	

Miscellaneous Digital Pins: 22-43, 47-53

These pins will be useful for non-PWM motor driver functions, reflectance sensor arrays, and other miscellaneous digital I/O. Pins 50-53 are used for SPI (Serial Peripheral Interface) devices, but it's unlikely you'll have any, so they're typically up for grabs.

Analog Pins: A0-A15

These pins are useful for connecting analog sensors. The TB9051 motor driver is wired to use A0 and A1 for current sensing. If you use a 2nd motor driver, you can wire it to pins A2 and A3 for current sensing, but

it’s optional to use current sensing. Either way, there should be plenty of remaining analog pins for distance, color, and magnetic sensors.

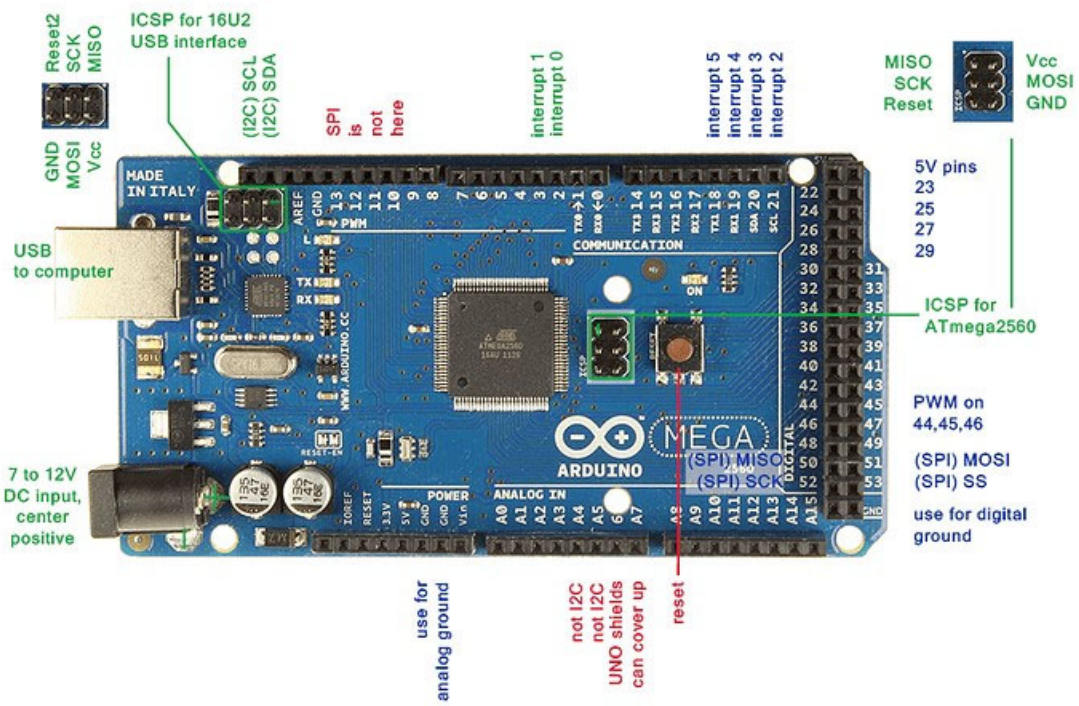

Figure 18. Arduino Mega with pin information showing.

Communications & Teleoperation

XBee Wireless Shield Assembly

We will be using XBee wireless transmission[2] to communicate between the robot and either a remote controller or your laptop (depending on the semester project). See Appendix A or WebCT for links to XBee information along with XBee Shield instructions and setup.

1. Solder the stackable headers to the XBee shield. DO NOT trim the header leads, as these are needed for the interface with the Arduino.
2. Place your XBee chips in the designated shield slots with the antenna towards the outside.

Ensure that the switches on both shields are set to DLINE, so that the digital pins 2 and 3 are used for serial communication rather than the USB lines of the Arduino.

Basic Communication

Because the Xbee chips are already paired, you will be able to communicate between the two devices immediately. Specifically, you can use the code that was prepared during Lab 01 to transmit data between two Arduino devices. While this code is a good starting point, there are a few points that should be clarified before you begin formulating the communication protocol of your mechatronic system

1. The Arduino Uno will require the use of the SoftwareSerial library to use two of the digital pins as an alternate communication method. Because this is the primary functionality of the Uno, you can simply use pins 2 and 3 onboard for this new serial communication line.
2. While SoftwareSerial will also work for your Arduino Mega, the library utilizes interrupt pins on the microcontroller. If all you are doing is wireless communication, this is not an issue. However, if your robot is designed to use servo motors, these interrupts can hinder the effectiveness of these motors. Therefore, the Mega should instead use one of the extra built-in serial ports. Specifically, port two, which is attached to digital pins 16 and 17. You should use jumper wires to connect these pins to the communication pins of your Xbee shield (pin 17 of the Mega should connect to pin 2 of the xBee shield, pin 16 of the Mega should connect to pin 3 of the xBee shield). The header pins (digital 2 and 3) on your Mega's Xbee shield should be bent out (or clipped) so that they do not connect to the shield or microcontroller below. This ensures that these pins are available for use, likely by your motor driver shield.
3. Now that the shields are correctly wired and configured, check to see if the two Xbees are communicating properly. Use the code you wrote for Lab 01 to blink an LED on the Mega and print a sine wave to the serial monitor. **TIP:** you can use the Serial Plotter tool in the Arduino IDE (either Ctrl+Shift+L or go to Tools → Serial Plotter) to visualize data.

[2] Additional resources for Xbee devices and ZigBee communication are available online.
ZigBee Information: http://www.zigbee.org/
XBee Shield Information: http://www.sparkfun.com/products/9841
XBee Shield Schematic: https://cdn.sparkfun.com/datasheets/Wireless/Zigbee/Xbee_shield_v15.pdf

Coding

The following pseudocode illustrates a very simple version of taking an input from the serial monitor and sending it out over the XBee. This would be code used on the Uno to send a couple character/number commands to the Mega wirelessly (something like 'a' or 5 or 'rgm'). This code does not handle strings, it doesn't handle large numbers, and it only sends data when the exact correct number of characters is typed into the serial monitor. Again, this is a very simple code. However, this code can get you started for wireless communication.

```
If Serial.available()
        Delay(20)
        If Serial.available() == numOfDesiredInputChars + 1
                xbeeSerial.write(Serial.read())
                Do this numOfDesiredInputChars times
                Serial.read()
        Else
                Print("Wrong number of inputs, please input your command again")
                While Serial.available()
                        Serial.read()
```

There are a couple of things to point out about this pseudocode. First, notice there are two if statements and one is nested inside the other. The reason for the first/outer if statement is so that this code will only happen when something is input from the serial monitor. If nothing is typed into the monitor, nothing will happen. The second/nested if statement then checks how many pieces of information were provided from the serial monitor. If we plan on sending a single character, like 'a', then numOfDesiredInputChars will be 1. If we want to send a couple pieces of information, such as "red green blue" indicated by 'rgb' then numOfDesiredInputChars will be 3.

When receiving information from the Serial Monitor, be aware that by default the Serial Monitor adds a newline character to your data after you type something and send it. This means if you type three characters and hit enter, the Serial.available() function will return 4 instead of 3. You can change this in your serial monitor, but more likely you should simply make your code handle this extra byte of data. The way this is handled in the pseudocode is by checking if numOfDesiredInputChars + 1 is available. The + 1 accounts for the newline character that was added by the serial monitor.

If the correct amount of information was input, then we are going to use that information. If a different amount of information is input, then this code will output a small error message and clear the serial buffer. The way we clear the serial buffer is with a small while loop that continues to read and discard anything in the buffer until nothing is available anymore. If we received the correct amount of information then we are going to read that information one byte at a time, and write it across the XBee communication as we go. If we have 3 pieces of information then we will call this write(read()) line three times, and so on. Once we have done that we call read() one final time just to discard the newline character that was put into the buffer by the serial monitor. In this way we can type a couple characters into the serial monitor and send them wirelessly across the XBees.

A final note is about the delay. Delays are not encouraged in this course as they stop your code from executing anything at all until the delay is finished, and that is bad. However, the read() and write() functions execute at different speeds and it is a good idea to place a small delay just to make sure all the information that was entered into the serial monitor finds its way into the buffer before we start checking how much info is in the buffer. Without this delay it is possible that the first character of, say, 10 will get put into the buffer, the outer if statement will become true, but before all 10 bytes are put into the buffer the second if statement will be checked. Since only some information got in at that point, the second if statement will return false and the buffer will be cleared instead of reading true and sending the information. Be sure to only use a very small delay here though as to not pause the rest of your code from running for too long.

Once the code is set up on the Uno to send information, a script can be set up on the Mega to receive that information. The sending and receiving codes need to be designed to match each other. If strings are being sent, then the receiving code needs to be capable of accepting strings. If multiple numbers are being sent with spaces between them, the code needs to receive that data in that format. In our very simple sending code we send a couple characters or small numbers. Therefore, the following pseudocode is capable of receiving such data.

```
If xbeeSerial.available()
    Delay(20)
    If xbeeSerial.available() == numOfDesiredInputChars
        Variable1 = xbeeSerial.read()
        Call this line numOfDesiredInputChars times, using multiple variables or an array to store
        the information
    Else
        Print("Something went wrong, I received the wrong amount of information")
        While xbeeSerial.available()
            xbeeSerial.read()
```

This code will store the information that was sent across the XBees. Remember that this is only a simple example that is designed specifically to receive the simple information that the Uno pseudocode was set up to send. Once again, we see nested if statements, a small delay to make sure all our data gets into the buffer before we start checking it, and a small contingency incase some of our data doesn't make it to us. Once the mega has this information it can decide what to do, maybe with if statements or a switch statement, maybe in some other way.

There are many other ways to send and receive data, and often times you will structure that data in desired ways. This is called packaging your data, or sending a data packet. One typical way to do this is to have a starting piece of information indicate the beginning of the information. This is sometimes called an indicator or flag. For the Lab 01 exercise, the integer 255 was used. Whatever value is selected, you **MUST** ensure that this value is unique and not achievable by any of the other variables present in the data packet. That is, no other piece of information in the data being sent is allowed to be 255. Otherwise, this structure will not work correctly. Once you send the flag you send the rest of the information. This way the receiving side can wait until it sees the flag, and then read all the data that comes after it while feeling confident it is reading it correctly. The benefits of packaging data like this include protection against reading data in the

wrong order. Say for example the two communicating devices turn on at different times and the receiving side starts receiving information in the middle of the message. Without an indicator of the start it would have no idea that it is reading the middle of the message instead of the beginning. But by waiting until it sees the start flag, the receiver knows for sure it is starting at the beginning of the data.

Decoding this particular data packaging structure is achieved thusly. Your receiving code will wait until the necessary number of bytes becomes available. For example, if you are sending three pieces of information (including the start bit), the receiving code will wait until three or more bytes are available to process. Next, the first byte is observed and compared to the known starting byte. If this is not the expected value, this data should be discarded, and the next byte investigated. Once the expected starting byte is found the remainder of the data packet can be sequentially analyzed in the order which it is expected to be seen.

Wiring Diagrams

A wiring diagram is a visual representation of your system used both to aid you in your building process, and to illustrate your connections to others. In order to be effective at these purposes it must clearly and accurately represent all the electrical connections in your system. This needs to include each component in your system, both large components like an Arduino and also small components like resistors and LEDs. For components that have multiple connections it needs to indicate which connections are being made. The power of the system, and how the power reaches every component, must also be portrayed. An example of a simple wiring diagram is shown in Figure 19.

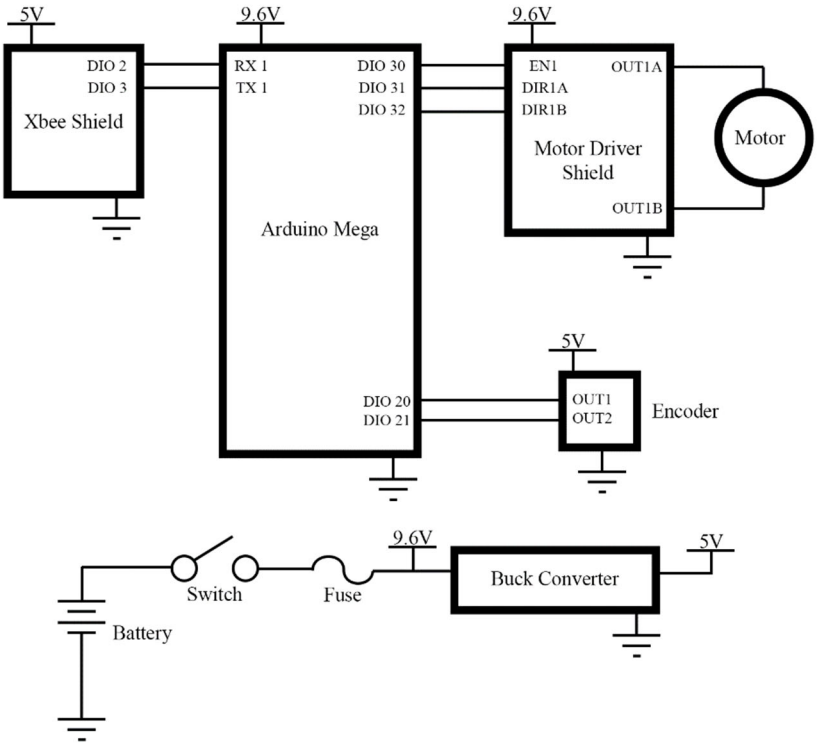

Figure 19. Example wiring diagram including an Arduino Mega, an xBee shield, a motor driver shield, a motor, an encoder, and power components.

As you can see from this example every component is clearly labeled. Each connection is also labeled on the components to indicate which pin or header the connection is made to.

A subsystem has been created for the power to indicate how the power comes from the battery and is regulated to different voltages. However, there are no lines going across the diagram for the power running to each component. Instead, the power subsystem indicates which voltages are available, and then each component has a small indicator of which power voltage is supplied to it. Also, each component has an indication that it is connected to ground, without needing to draw a line from every component to the ground of the battery. In this way a subsystem for your power can clear out cluttered lines that might run all over

your diagram. Just be sure to indicate for every component which supply voltage it is receiving, and that it is connected to ground.

In this example there is a motor driver shield and an XBee Communication shield. These are components which physically mount on top of an Arduino and have header pins that connect to every pin on the Arduino board. However, just because every pin is physically connected between the Arduino and the shields doesn't mean every pin is actually being used. In an electrical diagram you should show the shields in their own independent location (not on top of the Arduino) and indicate only the connections that are actually being used between the shield and the Arduino. Do not show every single pin on the Arduino being connected to every pin on the shield.

This exemplifies that just because two components are physically connected doesn't mean they are electrically represented that way. Another example of this is the encoders on motors. Often encoders are mounted directly on the motors, and in the case of the 37mm D motors in this course the wires for the motors are connected to the wires of the encoders in a single ribbon cable that comes out of the back of the motor. However, just because the cables are all together, and the encoder is physically on the motor doesn't mean they are electrically the same component. A motor has its own power supply, as does the encoder. In the example above the motor and encoder are correctly represented as two independent components, both with their own unique connections.

Autodesk Eagle Software

The software recommended in this course for creating complete, organized, and professional wiring schematics is Autodesk Eagle. This software is free to download and use, all you need to do is create an account using an email address. There are many quick guides and start up tutorials online, such as SparkFun's: https://learn.sparkfun.com/tutorials/how-to-install-and-setup-eagle/introduction. You can investigate other guides for a more thorough tutorial. Our project guide will cover the basics that will get you going.

Creating Your First Schematic

First, download Autodesk Eagle and create an account. Open Eagle, and select File->New Project. A new project will show up in the file tree on the left side of the screen. Name your project, then right click on it and select new schematic. This will open up a new window where you can create your wiring schematic.

To add components, click the "Add Part" tool on the left side of the screen:  . A new window will open up where you can search through your library to find the part you want to place. Next you will need to add the lines between the components. In Eagle these are called "Nets". To add a new net, click the Net tool on the left side of the window:  . Then click on the connections points of the components you want connected and a net will be drawn for you. Make sure your schematic is clear and comprehendable, each component is included , and that all connections you need to include are shown.

For further reading on how to create schematics in Eagle you can check out Sparkfun's tutorial: https://learn.sparkfun.com/tutorials/using-eagle-schematic/introduction

Adding the ME3230 Library

We have developed a rudimentary Eagle library full of components that are used in this course. To use this library, begin by downloading the "ME3230ElectricalDiagramLibrary.lbr" file from canvas. Move the file to your "…/EAGLE/libraries" directory which is likely located in your Documents folder. Once you have added the library file to your library directory you should have access to it within Eagle.

Hiding other libraries

By default, Eagle has a bunch of libraries and components that you may have to wade through to find the components you want. If you don't want to see all of these components you can elect to not "Use" them, which will hide them from your active libraries when making a schematic. To do so, in the main Eagle Control Panel (Home) window click the expand arrow next to "Managed Libraries" to expose the "Eagle PCB" library folder. Right click on "Eagle PCB" and then click "Use None". This will hide all of the default libraries, and the only library remaining should be the course library you downloaded.

Final Notes

Eagle wiring is designed to help people create custom printed circuit boards (PCBs). Because of this it has dual functionality. First is the functionality of building electrical schematics. These are abstracted representations of circuits (with boxes, shapes, and symbols). These schematics allow a person to view all the components and connections easily. The second functionality is designing the physical board layout of actual circuitry and components. This involves the dimensions of components, and using the appropriate footprint for each part, so that when a circuit board is printed it has all the physical requirements and connections.

For our purposes we will not get into the board design, nor the actual footprints of any of our components. We will only deal with the schematics, which are symbolic/abstract representations of our systems. Therefore, the library provided for this course does not have accurate footprints for its components. If you are planning to build/print your own PCB, do not use the footprints found in the ME3230 Eagle library.

State Transition Diagram

State transition diagrams are discussed in class and should be created as explained. The following section provides some additional tips and tricks for making a good diagram.

Every state is something that happens for some indeterminant amount of time. The state continues to occur until the event eventually happens and forces the system to transition out of that state. Therefore, a single state can't depict multiple different actions/operations that happen sequentially. To be clear, driving forward while detecting a line and also checking a distance sensor can all be a single state because all these things happen together, in parallel, for some indeterminant amount of time. But driving forward until a button is pushed and then driving backwards can't be a single state because the forward motion and the backward motion happen sequentially, and the trigger is a button being pushed. In the examples shown below you will see how single tasks such as "Drive to the next location" are split into all the sequential components that make up that task.

Keep in mind that things that occur only for a instant and immediately move onto something else does not qualify as a state. A state occurs for some nonzero length of time. As an example, if a system drives forward, then stops and turns, you don't need to include a "Stop" state. It only stops for a split second, and it immediately begins to turn. Therefore the "Drive Forward" state would go right into the "Turn" state once the desired position was reached.

A quick tip is to avoid using time as a condition for your events. Rather than saying "drive forward for 3 seconds", say something like "Drive forward until final position is reached" which indicates that you are using your encoders to detect positional movement. The reason for this is that with a battery-operated system, certain things will happen differently based on how much charge the battery has. A fully charged

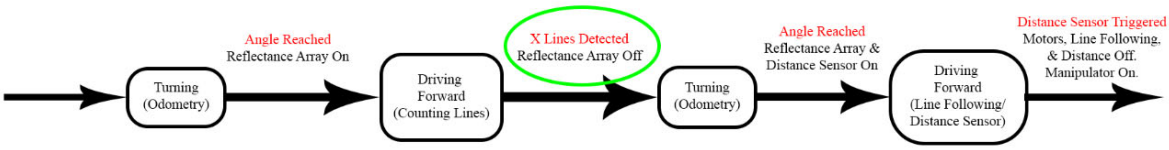

Figure 20. Example of states in series with each other, and a transition that is based on a variable.

battery might make your system drive 10cm in 3 seconds, but a half-charged battery might only allow you to go 7cm in 3 seconds. Generally, your events will all be based on sensors (encoders, distance sensors, limit switches, etc.) rather than time.

The conditions of an event can be based on a variable, which alters the operation of the system. In the following example the system turns, drives forward until a desired number of lines have been crossed, then turns and drives forward again. The variable desired number is shown in the diagram as "X" and is circled in green. Based on what final position the system is driving to the number of lines to be crossed will be different. This way the same four blocks can portray driving to, say, Zone 2, 4, or 5 instead of duplicating these four states in the diagram for each possible final location.

A different way of including repeated states that occur during different times of operation is to simply include them multiple times in the diagram. The following diagram shows the Extend/Retract states twice. These states are identical to each other and have identical events and services. However, these states occur at different times throughout system operation, so they are shown multiple times in the appropriate positions along the pathways of states.

Another common structure is to have parallel pathways that maybe diverge and converge from a single location. The following image shows an example of such a structure. If driving away from Zone 2 the system only needs to drive forward and then turn, but if driving away from any other Zone the system needs to complete two extra tasks (turning and driving forward). In this case a flag variable would keep track of which Zone the system is driving away from, and decide which path to follow based on that flag variable.

This previous diagram showing parallel pathways also demonstrates that the conditions that trigger events can have multiple requirements. That diagram showed that both the distance sensor needs to be triggers AND the previous Zone needs to be 2 in order to follow the shorter path. Having multiple conditions for an

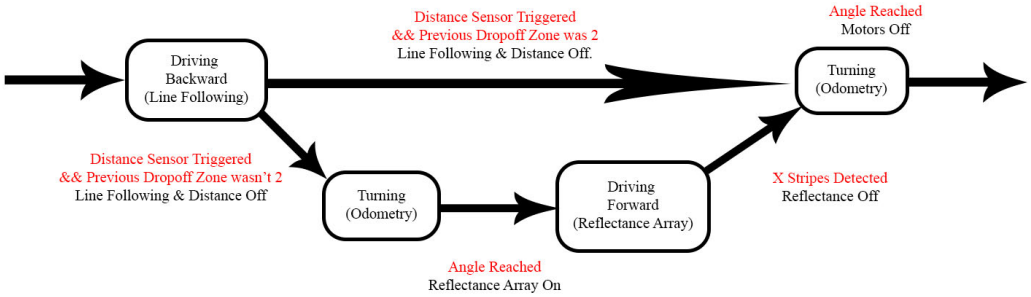

Figure 22. Example of multiple/parallel alternate paths.

event is common, so make sure to keep track of all the requirements and include them.

The final thing that the parallel pathway diagram example shows is another example of a variable condition. The final turning state operates until a desired angle is reached. That angle is variable depending on how the system is currently operating. In that example, if the system came from the "Driving Backwards" state then it needs to turn 180 degrees, but if it came from the "Driving Forward" state then it only needs to turn 90 degrees. This is another example of generalizing a state or an event so that it is applicable to multiple scenarios.

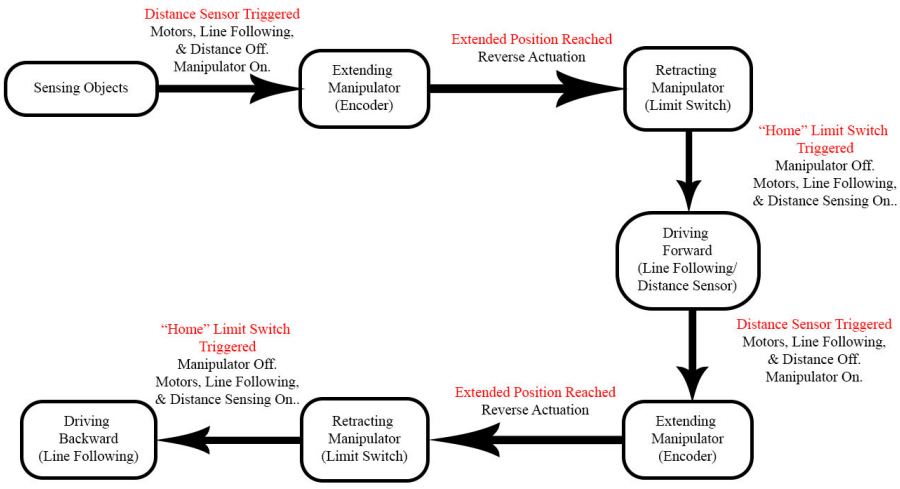

Figure 21. Example of repeated states. The extend/retract states are repeated here.

General Coding Practices

Main Structure

One aspect of state transition diagrams that can be extremely helpful is that they can be directly related to the structure of your code. You can write you code based on the states of your system, implement transitions between these states, and run services. The basic idea can be implemented as follows:

```
Switch(State)
    Case "First State"
        Call Internal State Functions
        If (event)
            Transition
            Call Services
    Case "Second State"
        Call Internal State Functions
        If (event)
            Transition
            Call Services
```

We can see that the states, events, services, and transitions from our state diagram directly show up in the structure of the code if it is written this way. This helps us keep track of our code and what the system is doing by associating it to the visual representation of our state transition diagram.

In this structure the Transition would be to change your state variable from one state to another so that the next time your code comes through this switch statement it goes into a new case. The internal state functions are functions that operate while in that state. These could be functions like driving forwards, line following, or sensing objects. The events will likely be triggered by variables that are updated by your internal state functions. One example of this could be a driving function which updates the current position of your system, and the event is triggered when a global position variable reaches its desired position. Another example could be a sensing function that reads a sensor, and the event is triggered when the returned value crosses a desired threshold. The services that are called during the transition are things that need to happen a single time during the transition from one state to another. This can include things such as setting the motor command to stop, turning a sensor on, or setting a baseline variable to its initial value.

Functions

It is good practice to create functions to perform portions of code that will be called multiple times in multiple places. A good example of this would be a driving function which can be called any time your code reaches a state where driving around is involved. When you create these functions there are a few things to keep in mind so that your code will run smoothly.

Do not use loops in your functions. Use the main loop of your code to do the looping functionality for all your code. If you put a loop into a function your code will get trapped in that function until the loop completes and won't be able to do anything else. As an example, imagine you want your system to drive forward a certain distance, but if it detects something in front of it it should stop. So, you create a driving

function that has a while loop to drive forward until your desired position is reached. However, your code is now stuck in that while loop and will never be able to check your sensor and will run into objects. Therefore, you should not use loops inside your functions.

Note: This is not an absolute rule, there are times, although very rare, where it is perfectly fine to use a loop in your functions. A great example is when reading all the sensors on your line following sensor array. A small loop is used to check each sensor along the array.

Rather than using loops in your functions you should create them in such a way that they run once and update any variables they need to update, and then finish. If they are supposed to run more than once the main loop function of your code will call them multiple times. For a driving function this would mean it gets called for a single time step: the current position is compared to the desired position, a motor command is sent to the motors, and then the function concludes. The main loop will then call this multiple times until eventually the position of the system reaches a desired location, or some other event triggers the main loop to start calling a different function.

Some of your functions will output nothing, some will output values, some will output Booleans. It depends on what the function is doing, and how you want that function to interact with your switch statement. A driving function could either update a global position variable, and therefore not output anything, or determine the position locally and output that value to be used in the main loop. A line following function would likely not output anything because other factors would be the events triggering the system to leave the line following state. A sensor reading function would likely output the reading. A function could also output a Boolean to tell the main code whether it has completed. These are all different ways your functions can interact with your main switch statement.

The following are two examples of functions that essentially do the same thing: move until a desired position is reached. The DrivingForward() function updates a global variable and does not return anything, the Turning() function updates a local position variable and returns it.

```
void DrivingForward(float finalPosition) {
  long t_ms = millis();
  float t = t_ms / 1000.0;
  if (currentDesiredPosition < finalPosition) {
    currentDesiredPosition += desiredSpeed * (t - t_old);
  }
  globalPositionVariable = UpdateCurrentPosition();
  float error = currentPosition - currentDesiredPosition;
  float motorCommand = Kp * error;
  driverObject.setSpeed(motorCommand);
  t_old = t;
}
```

```
float Turning(float finalAngle) {
  long t_ms = millis();
  float t = t_ms / 1000.0;
  if (currentDesiredAngle < finalAngle) {
    currentDesiredAngle += desiredOmega * (t - t_old);
  }
  float currentAngle = UpdateCurrentAngle();
  float error = currentAngle - currentDesiredAngle;
  float motorCommand = Kp * error;
  driverObject.setSpeed(motorCommand);
  t_old = t;
  return currentAngle;
}
```

Based on these functions the main switch statement might look something like this:

```
void loop() {
  switch (stateVariable) {
    case 1:
      fcnOutputStorageVariable = Sensing();
      if (fcnOutputStorageVariable == true) {
        stateVariable = 2;
        t_old = millis() / 1000.0;
      }
      break;
    case 2:
      DrivingForward(desiredFinalPosition);
      if (globalPositionVariable >= desiredFinalPosition) {
        stateVariable = 3;
      }
      break;
    case 3:
      returnedCurrentAngle = Turning(desiredFinalAngle);
      if (returnedCurrentAngle >= desiredFinalAngle) {
        stateVariable = 4;
      }
      break;
```

We can see in this structure that we have internal functions being called while we are in each state, and then when the event occurs we transition by changing the state variable to the next state.

Variables

You can define a variable to either be global or local. Global variables are declared at the top of your main script, outside of any functions. Local variables are declared inside functions and are only used in those functions.

Local variables are useful when you are only using a variable inside a function. In the previous function examples the error variable is a local variable. It gets set each time the function is called, meaning it doesn't need to remember it's previous value, and it is only used inside this function as an intermediate variable for some internal calculations. Since we don't need to look at that error variable anywhere else in the code, nor do we need to store its value from one call of the function to the next, it is perfect for a local variable.

On the other hand, some variables need to be global variables. Variables that are used outside of your functions, like the globalPositionVariable which gets used inside the function but checked in the main loop code, need to be global so that both the function and the main loop have access to that variable. Another reason a variable will need to be global is if it needs to remember its value from one call to the next. A great example is t_old. If t_old was a local variable it would get declared and initialized every single time the function was called, which is not what we want. We want to update t_old and then next time we call the function remember what value t_old is. Therefore, it needs to be a global variable. Same thing with the currentDesiredPosition variable in the previous examples.

Another important thing to remember about global variables is that you need to set them appropriately, and at appropriate times. The t_old variable is a great example again. Notice that when transitioning from sensing to the driving state, t_old is set equal to the current time. This is because the driving function is

going to use t_old. If you don't do this then when the driving function and goes to update currentDesiredPosition based on a desired speed and time, t_old will be 0 and the time difference will be huge which causes the initial update of the desired position to make a huge jump. Therefore, it's very important to set your global variables to their appropriate values during your state transitions.

Notes

Be careful of integer division in your code. Int type variables do not have decimal points, so if you divide something (like when dividing time in milliseconds from the millis() function by 1000) you can potentially lose the precision provided by having a decimal result. You can get around this by using float or double type variables, and by placing a ".0" at the end of values in your equations (as seen in the example above).

When using time in your functions be sure to record the initial time that the function starts. There are some times when knowing the time of the previous loop iteration is important (t_old, like seen in the above examples). In these instances it is necessary to have an accurate value in the t_old variable, rather than a value of 0 which is likely what the variable is initialized to. If the system turns on, waits a while, and then performs a command, it could act differently than if it turned on and performed the command immediately if you don't accurately record the starting time of the function.

Motor Selection Exercise

Note: Experimental characterization of DC motors will be conducted in Lab 10, however, we are providing this information early to help guide your robot designs.

The objective of this section is to analyze the provided motors to select the "best" motor for a team's strategy. This begins by classifying the provided motors[3]. Part of your team's strategy that leads directly toward design involves how fast or strong the robot is and how the robot moves. This involves the drivetrain design of the robot and ultimately motor selection; for most designs this will constitute selecting a motor/gearbox and wheel to attach to the motor. There are three scooter wheel sizes to choose from: 70, 84, and 100 mm. These scooter wheels will fit on the 37 mm Pololu motors with the appropriate hubs. There is only one size of wheels that will fit on the Micro Metal Gearmotors. These wheels are 32mm in diameter. Certain brands of motors look the same, but the gearboxes vary in their ratios. For the 37mm Pololu motors, the ratios are as follows: 50:1, 70:1, 100:1, and 131:1. For the Micro Metal Gearmotors, the ratios are as follows: 100:1, 150:1, 210:1, 298:1, 380:1, and 1000:1. You will need two motors of the same type, one for each drive wheel. You should take time to discuss with your team the advantages and disadvantages for each motor type. **As you are analyzing each motor/gearbox enter the data into a table with columns like the one below for each motor and wheel pair.**

[3] For another method of motor selection you can look at the "MICROMO – How To Select a DC Motor" guide online at http://www.micromo.com/how-to-select-a-dc-motor.aspx

Table 5. Motor table example for a 50:1 motor with 70mm wheel attached.

Gearbox	**50**	rad/rad (out/in)
Wheel diameter	**70**	mm
Gearbox output RPM at 12V		RPM
Motor RPM at 12V		RPM
No load translational speed at 12V		m/s
No load translational speed at 1V		m/s
No load translational speed at 5V		m/s
No load translational speed at 9V		m/s
Stall torque at gearbox output at 12V		oz-in
Stall torque at gearbox output at 1V		oz-in
Stall torque at gearbox output at 5V		oz-in
Stall torque at gearbox output at 9V		oz-in
Robot holding force at 9V		lbf

Note: *While the motors are capable of operating at 12V, the robots are limited by the batteries which supply 9.6V. It is safe to assume that the maximum voltage you can supply will be about 9V since there is a voltage drop across the H-Bridge used to control the motors.*

Assuming a linear relationship between motor speed and torque, calculate the following using the information from the datasheets[4], answer the following questions. It may be helpful to use Excel for this exercise, as it will better organize the calculations outlined below. By answering the following questions, you should determine the theoretical speed and force achievable from your robot. From these values, your team can select a motor/wheel configuration that is suitable for your desired performance and strategy for the competition.

Motor Speed

1. Using the datasheet for each motor and the free-run RPM of the gearbox at 12V, what is the Motor RPM (Use the gear ratio)?

2. What is the minimum translational speed that each motor and wheel can ideally produce? This translates to how quickly the robot would move utilizing the motor, assuming no load on the motor. Perform this calculation for each of the following applied voltages: 1V, 5V, 9V, and 12V.

Motor Torque (note stall torque is the torque at zero RPM)

3. Using the online datasheets, what is the stall torques at the output of the gear box at 12V?

4. As a result what is the output stall torque at 1V, 5V and 9V?

[4] Motor specs can also be found online from https://www.pololu.com/product/4753 and wheel specs from https://www.pololu.com/product/3272.

5. Using the datasheet for the motors and knowing the diameters for the wheels, calculate the theoretical maximum holding force for a robot utilizing two motors (or as many drive motors are on your design) at 9V, *i.e.*, how much force can the robot push or pull using two motors, if 9V is applied to the motors?

Motor Selection

6. Given the knowledge of these motors, which would yield the fastest robot (Think of translational speed)?

7. Which motor selection would yield the robot with the greatest holding force?

8. Based on your team strategies, select the desired motor for your team to use and justify your decision.

Using Two (2) TB9051FTG Motor Driver Shields

If you plan on using additional DC motors for your mechanism(s), or other similar equipment which functions similarly as these motors, you may consider using a second motor driver shield. The shields used in lab are stackable and can be set up such that more than one can be used. When using two shields, be sure to use the modified DualTB9051FTGMotorShieldMod3230 library found in canvas. Here, we will outline a procedure to achieve a dual configuration. You can also see https://www.pololu.com/docs/0J78/3.e for the official documentation.

1. Solder two (2) Pololu Dual TB9051FTG Motor Driver Shields as outlined above. One of these will be used in the standard configuration, and thus should not be altered. The second shield will require remapping of pins to new locations on the Arduino. The ARDVIN=VM header pin is only necessary for one of the shields, so these pins are not necessary for the second shield.
2. Use your X-Acto knife to carefully cut the traces on the topside of your second motor driver shield. The traces shown in Figure 23 connect the default pins of the motor driver. You can use the continuity checking functionality of your digital multimeter to test whether the traces are completely cut. If you don't want to cut the traces you can just bend the header pins so that they don't connect to the first shield. Be sure to insulate these bent pins so they don't short out on anything. Also remember to insulate the bottom of the blue terminal connectors.
3. You will be jumping the now floating pins on the motor shield to new pins on your Arduino Mega. There are two locations where you can break out: the row of black header pins at the edge of the board, or from the connection points where you cut the traces. It is recommended that you use solid-core wire with enough length to reach any pin on the Arduino from these through-holes, since you may need to iterate through the pin mapping process.

4. The DualTB9051FTGMotorShieldMod3230 library should be used when stacking two motor driver shields. It uses the pins in Table 6 by default. If you want to change the pins for motors 3 and 4 you can use the following line when creating your object. MxEN, MxDIR, …, etc. should be updated accordingly.

Table 6. Default pins when using DualTB9051FTGMotorShieldMod3230 motor driver library.

	Driver Shield Pin	Arduino Pin
Motor 1	M1EN	Digital 2
	M1DIR	Digital 7
	M1PWM	Digital 9
	M1DIAG	Digital 6
	M1OCM	Analog 0
Motor 2	M2EN	Digital 4
	M2DIR	Digital 8
	M2PWM	Digital 10
	M2DIAG	Digital 12
	M2OCM	Analog 1

	Driver Shield Pin	Arduino Pin
Motor 3	M1EN	Digital 23
	M1DIR	Digital 27
	M1PWM	Digital 45
	M1DIAG	Digital 31
	M1OCM	Analog 2
Motor 4	M2EN	Digital 25
	M2DIR	Digital 29
	M2PWM	Digital 46
	M2DIAG	Digital 33
	M2OCM	Analog 3

If you change the pins for motors 3 and 4, the main pins of interest are the enable, direction, and PWM (speed) pins. Specifically, the PWM signal is restricted to only the PWM capable pins on your Arduino. Run wires to connect the appropriate pins on your motor driver shield to their matched Arduino pins. If you are not using the DIAG and OCM functionality it is not necessary to physically rewire them.

You should now be able to stack the two motor shields together onto your Arduino Mega and run four separate motors. For this class, it is not recommended to use a third motor shield. While the hardware is

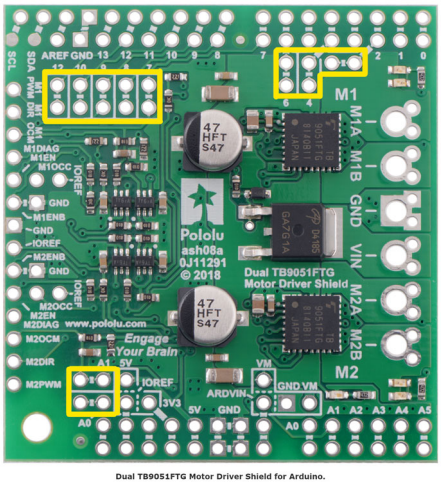

Figure 23. Cuttable traces of the motor driver shield. Traces are on the (shown) topside of the circuit. Taken from https://www.pololu.com/docs/0J78/3.e, for educational purposes only.

capable of this configuration, significant work is needed to acquire additional PWM pins with correct timers, and more advance knowledge of coding in C is recommended. Instead, teams with 5 or more DC motors should either consider redesigning to decrease this count, or investigate alternate hardware (L298 driver, or out of lab) to run the additional motors.

For further information on these motor shields, look at the user guide on the Pololu website: https://www.pololu.com/docs/0J78/all