This is a draft. It still should be useful for students.

As a work in progress, **it contains errors, inconsistencies, and omissions and dreaded repetitions**.

Please post issues on https://github.com/FIRST4513/Robot-Manual/issues.

Download PDF at https://first4513.github.io/pdfs/robot-manual.pdf.

FRC 4513 Robot Manual

Some technologies underlying FRC robots

Table of Contents

List of Figures

List of Tables

Dedication

This paper is dedicated to the memory of Woody Flowers (1943-2019) who was dedicated to hands on learning engineering and science to as many as he could. He coined the word coopertition, the spirit of helping competitors to make everyone better. He also coined “gracious professionalism” to help remind us that we are all in this world together, that we succeed without boasting and come to the needs of those around us.

His short class on “How Things Work” inspired this paper.

1 Introduction

1.1 Purpose

The purpose of this manual is to introduce students and mentors to the FIRST Robotics Challenge (FRC) robots. This is a fairly general discussion and applies to the general architecture of FRC robots. It has information which may not be useful for your particular robot. The intent is to provide a discussion of various technologies, so that you may want to try to use them in your robot. The corollary is that it may not have the particular information that you need.

This manual does not simply tell you what you need to know to deploy an FRC robot, it tells you what is under the cover. It is not like a car manual that tells you how to use the various features of a car, it tells you how the various subsystems work. This allows you to understand how the components of a robot work together to perform the objectives set for the robot. You can drive a car without knowing how the engine works, but if you want to race that car, you better know as much as you can about the subsystems in that car.

This document is primarily about the electrical and control systems in the robot. It provides an overview of the robot architecture and how that relates to the control architecture and software architecture. It also includes a brief overview of the Java programming language and its characteristics as an object-oriented language. The rule book for a particular year's challenge will provide the general constraints on the robot's mechanical and physical characteristics.

The goal is to lift all teams to make competitions a bit more competitive. Hopefully it brings some explanations to teams that do not have all the resources and mentors of the larger teams. You don't "need" to know everything in this manual to work with robots, but the more that you know, the more that you can modify, adapt and improve them.

Most of this material is directed to students with little or no background in the subject. Seventh and eighth grade students should be able to read and understand most of this manual, except the sections using advanced math. Some sections assume a knowledge of algebra which also may be beyond some middle school students.

1.2 How To Use This Manuals

This paper develops the basic skills and theories for working on a robot first. It builds on those concepts toward more advanced topics toward the end of the paper. If you (think) know something, skip it. This is more of a manual than a text book. It just tries to introduce information that may be missing from other sources. It leads you to more detailed information on the Internet and other sources.

More detailed general information can be found at https://docs.wpilib.org.

1.3 Acknowledgments

This project is indebted to the documentation provided by WPILib. Some material is shamelessly copied and some is presented differently. The intent is to compliment the official WPILib documentation. Many other teams provide instructional videos on YouTube and elsewhere on the Internet for those who want lecture style instruction. This manual is definitely old school: no access device required.

This manual is not possible without the public works of other teams: Team 2910 Jack in the Bot (swerve drive improvements and overall design inspiration), Team 3847 Spectrum (software structure, joystick

abstractions, exponential mapping for joysticks), Team 3015 Ranger Robotics (PathPlanner) and Team 6328 Mechanical Advantage (AdvantageKit logging and AdvantageScope log analyzer). Many of the explanations were patiently provided by Dennis Schweikhardt, mentor emeritus of Team 4513.

2 Basic Skills and Knowledge Required

Working on the robot requires skills. Some you can learn by doing "on the job" and some may have to wait until you are taught the subject in a class. Most you can learn on your own. Some knowledge may be out of reach for some students and even some mentors. Don't worry, there are still plenty of ways that you can contribute to your team. This section goes into the basic knowledge required.

The most basic skill is a willingness to dig in and learn. Whether this is riveting together the chassis, making bumpers, advertising the team's accomplishments, or figuring out how to use vision subsystems to the greatest advantage. This desire to learn and do is essential to working on the robot and contributing to the team effort.

Robots have a lot of wiring to deliver power and control to various components. Some knowledge of electricity is useful in doing these tasks correctly. A working knowledge of physics is useful to understand the effect of mass on acceleration and speed both in the robot drive train and in various other robot subsystems. An understanding of the basic machines explains the use of gear trains and constant force springs to achieve an objective.

Parts of software use a lot of math. In general the more math you know, the more you and your robot can do. There are still a lot of the software that does not require any math skills.

2.1 Some Basic Physics

A robot is primarily about basic physics. This is not necessarily about the formulas used, but the concepts. The ways that a robot moves or crashes is dictated by physics.

FRC robots are about speed. Speed is the change in distance over a period of time. You are used to miles per hour or kilometers per seconds, the software expresses speed as meters per seconds (m/s). How fast you change speeds is acceleration and that is expressed as meters per second (m/s^2). You can feel acceleration as your family car or school bus accelerates to reach a particular speed. Likewise, you can feel deceleration as the vehicle puts on its brakes. When acceleration or deceleration is rapid, you fell that as well as a jerk (Yes, jerk really is a term in physics). Jerk is measured as meters per second^3 (m/s^3).

2.1.1 Newton's First Law of Mechanics

Newton's First Law of Mechanics has the following two parts:

- An object at rest will remain at rest until acted up by an external force.
- An object in motion remains in motion in a straight line and constant velocity until acted upon by an external force.

This means that an external force is required to affect change in the movement of an object. This is expressed with the formula:

$$Force = F$$

$$Mass = M$$

$$acceleration = a$$

$$F = M \cdot a$$

Which using algebra (dividing both sides by M) can be rewritten as follows to solve for the acceleration.

$$a = F / M$$

This formula means that with a fixed amount of force, acceleration is adversely impacted by mass. The force available to a robot is largely fixed by the rules that limit the size of the battery and the particular motor types used. So the only variable on the right side of the equation is the mass of the robot. The heavier the robot, the slower its acceleration. The shorthand way of saying this is that the robot acceleration is inversely proportional to the mass.

2.1.2 Newton's Second Law of Mechanics

Newton's Second Law of Mechanics states;

- The force acting upon an object is proportional to its mass times the square of its velocity or as expressed in the following formula

$$F = M \cdot v^2$$

What that means to robots is the faster you go, the harder you will hit (and the more damage will be caused).

2.1.3 Newton's Third Law of Mechanics

Newton's Third Law of Mechanics states:

- For every action there is an equal, but opposite reaction.

This has a lot of implications to robots. To accelerate, the robot exerts a force onto the carpet through its wheels. The carpet in turn exerts an equal but opposite force to the robot wheels. If there is no slip, the robot can be propelled in the direction that it desires.

The second and third laws combine and have catastrophic implications to robots. When a robot collides with another robot, all of its momentum and velocity are converted into a force. This force applies to both robots or to the robot and a field piece and can cause great damage to robot that is unable to absorb that force. Field pieces, in general, should be thought to have a large mass as they increase their mass by having a firm attachment to the carpet.

Most of this discussion has been about the forces of moving robots about the field. Many robots have arms and elevators that have to fight the force of gravity. The basic concepts are the same except that the acceleration is replaced with another acceleration, the gravitational constant G.

$$F = M \cdot g$$

where

$$g \approx 9.8\, m/s^2$$

Gravity always pulls down on elevators and arms and this force must be accounted for in motor controllers usually as a feed forward term.

2.2 Some Simple Machines

Physics describes a few simple machines for adapting forces. This allows a small force to become a larger force. This is done by requiring more motion. Work, which is the product of force over distance, remains constant. Some simple machines discussed in the following sections.

2.2.1 Lever

Levers are used to multiply the force applied by a beam. The ratio of the length of the beam on each side of a fulcrum determines the factor to apply to the force. If a force is applied to the long side of the beam and the load is applied to the short side of the beam and the ratio of the two sides is 4:1, the force applied to the load is 4 times as big as the force applied to other end of the beam. The flip side is that the force side of the beam has to move 4 times as far as the load moves.

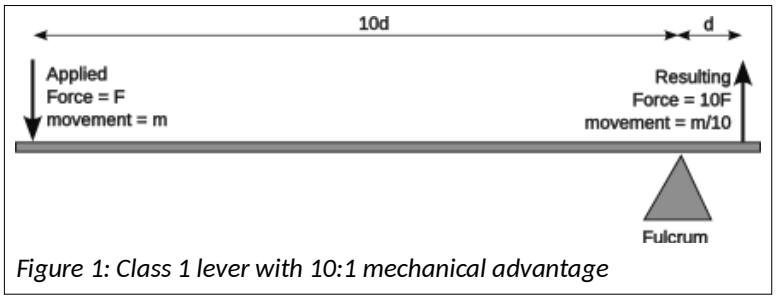


Figure 1: Class 1 lever with 10:1 mechanical advantage

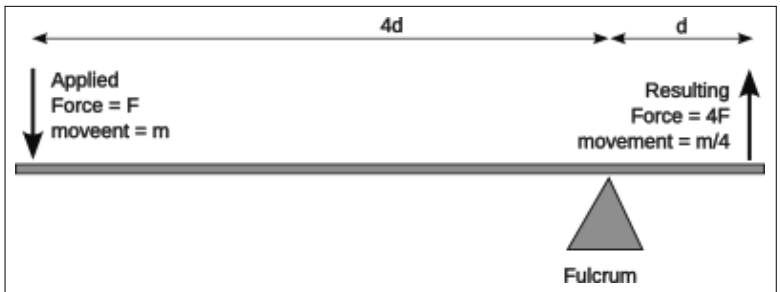


Figure 2: Class 1 lever with 4:1 mechanical advantage

There are three classes of levers:

Class 1: force is applied to one end of a beam and an opposite force is created on the far side of a fulcrum or pivot point. This is the classic use a bar to pry out or lift something. The forces multiply according to the ratios of the length of the beam on either side of the fulcrum. Normally the ratio is not even and the applied force is multiplied. An exception is a teeter totter (when those used to be a thing) can have a more equal forces on each side of a fulcrum, although it will accommodate unequal weights, if the lighter person moves further out on the teeter-totter (or the heavier person can move closer to the fulcrum).

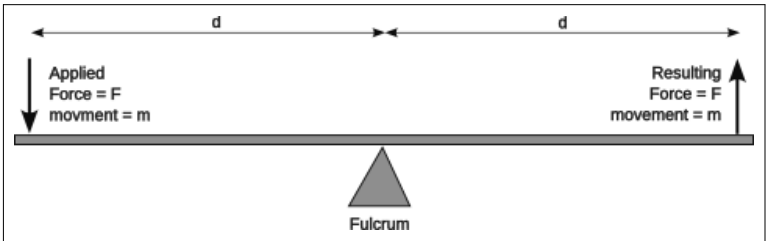


Figure 3: Class 1 lever with no mechanical advantage

Class 2: force is applied to the end of a beam and a magnified force is applied in the same direction closer to the pivot point. An example of this type of lever is a wheel borrow.

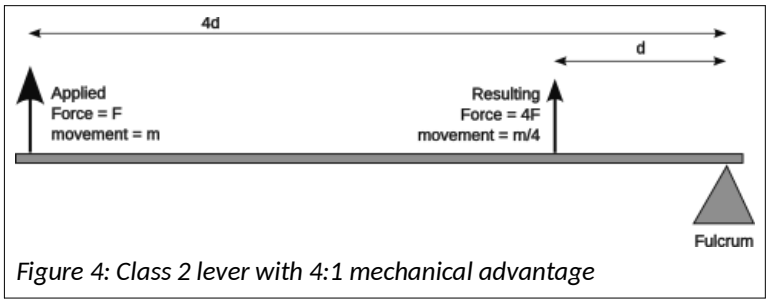


Figure 4: Class 2 lever with 4:1 mechanical advantage

Class 3: a force is applied to a beam close to the fulcrum. The end of the beam is moved through a greater distance, but with less force. This type of lever is typically used to increase movement velocity. Examples of this type of lever are a fly swatter, a baseball bat, tennis racket, catapult, and a fishing rod.

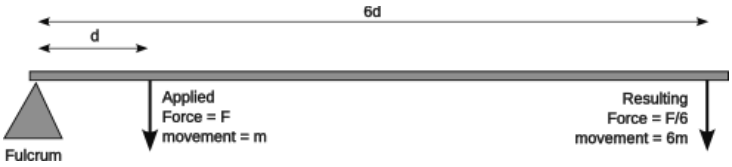


Figure 5: Class 3 lever with 1:6 mechanical disadvantage

2.2.2 Pulleys

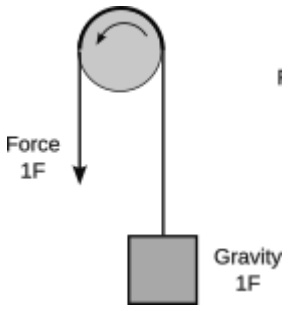


Figure 6: Basic pulley to change direction of force

Pulleys are used to change the direction of a force without any force multiplication.

Pulleys can be arranged with a block (moveable part) and tackle (fixed part) to multiply the force applied. A simple block and tackle will result in a 2:1 mechanical advantage.

By using multiple pulleys in a block and tackle gain even more mechanical advantage to lift heavier loads.

Pulleys are used in some robot arms and elevator mechanisms to extend their length.

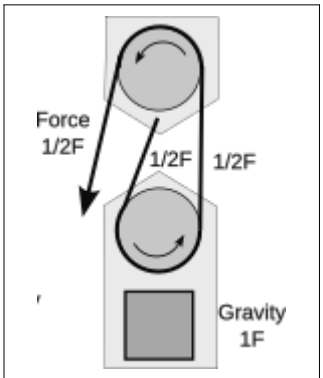


Figure 7: Basic compound pulley or block and tackle

A pulley can be thought of as a lever rotating about a fixed fulcrum or axel.

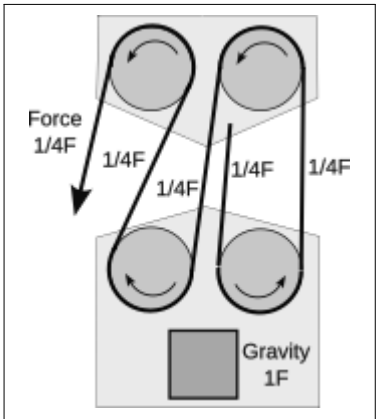


Figure 8: Double compound pulleys in a block and tackle

2.2.3 Gears

The forces of rotary motion can be modified with gears. Their direction can be changed, or the forces can be transferred with gears. Using gears of different ratios allows the forces to be multiplied or reduced. Rotary forces are increased at the expense of speed. Conversely, speed can be gained at the expense of force. The total work, force times distance, remains unchanged (ignoring friction).

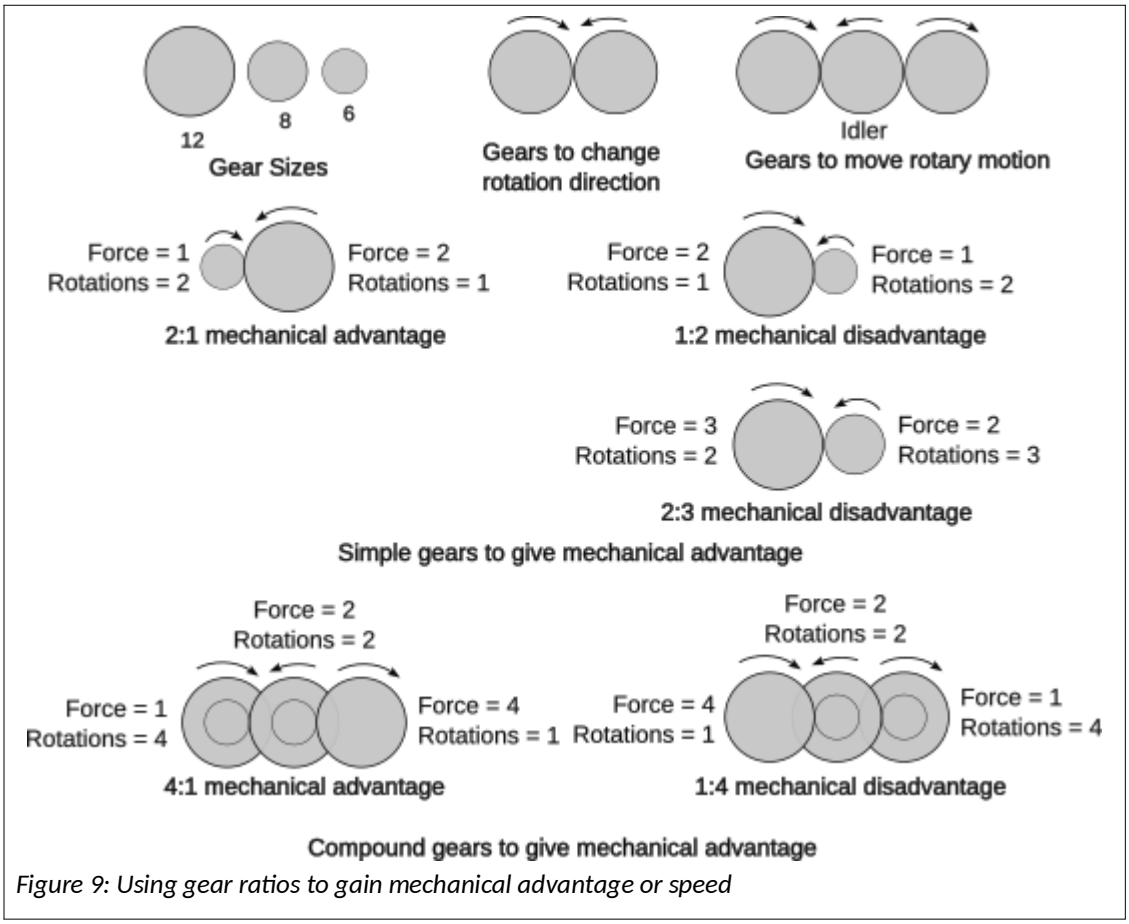


Figure 9: Using gear ratios to gain mechanical advantage or speed

Gears can be compounded by having more than one gear fixed to an axel. Gear trains of two or more axles can multiply the individual gear ratios for even more advantage.

The mechanical advantage is determined by the ratio of the sizes of the gears. Gear sizes may be determined by diameter, radius, tooth count (of fixed size teeth). It does not matter how the size of the gears are determined as long as it is consistent for a given gear train.

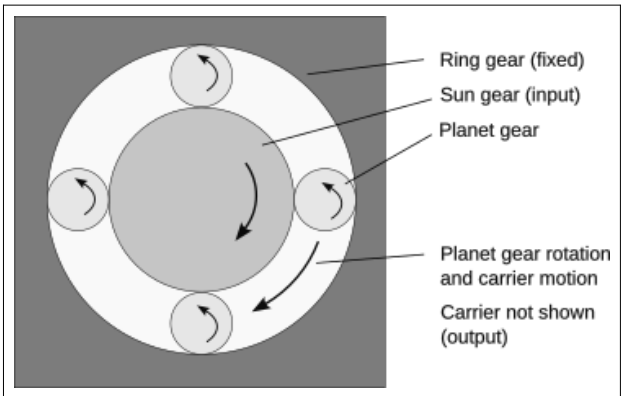


Figure 10: Planetary gear module

Many FRC motors use a system of planetary gear transmission modules to build a gear train. Each module provides a particular gear ratio and its accompanying mechanical ratio. These modules are compact and because the output shaft is in line with the input shaft, they are stackable. This allows building gear train with several modules and the mechanical advantages multiply. Using a 5:1 and a 4:1 module produces a transmission with a 20:1 gear ratio. Internally these modules use a planetary gear. The input shaft is connected to the sun gear. It turns a set of 2 or more planet gears. The planet gears are also messed in an outer ring gear that is fixed to the frame of the module. So as the sun gear turns, the planet gears turn and move their axis of rotation in the same direction as the sun's rotation. The planet gears are connected to a carrier which rotates on the same axis as the sun and also serves as the module output.

Only certain combinations of gear ratios (or gear diameters) will work. The following equation must be satisfied:

$$\frac{N_s + N_r}{n_p} = A$$

where:

N_s is the number of teeth on the sun gear
N_r is the number of teeth on the ring gear
n_p is the number of ring gears
A is a whole number

With the ring gear fixed to the frame, the gear ratio for a planetary gear becomes:

$$\frac{\omega_c}{\omega_s} = \frac{N_s}{N_s + N_r}$$

where:

N_s is the number of teeth on the sun gear
N_r is the number of teeth on the ring gear
ω_c is the angular velocity of the carrier
ω_s is the angular velocity of the sun gear

In robots the gear ratio may be something to consider in control systems if the driving motor shaft is not the same as the monitored shaft. If there is a fixed gear ratio between the two, this ratio should be incorporated into the control parameters for the motor.

2.2.4 Wedge, Ramp, and Screw

Another simple machine is a wedge. As a tool it is for inserting a force between two faces, commonly used to split wood. It has other forms. As an incline or ramp, it allows exchanging a horizontal force with

vertical force. A ramp lifts or lower a vehicle or person. Sometimes a wedge can be driven horizontally almost how a dust pan can be used to lift dirt and other things off the floor. Take a wedge and wrap it around an axle, and you have a screw which converts a rotary motion into a linear motion.

More detail about these machines can be found at https://en.wikipedia.org/wiki/Simple_machine. Consideration of these various machines go into the mechanical design of the robot.

2.3 Forces

There are many forces that act upon robots:

- Rotary forces from motors.
- Electro-magnetic and magnetic forces that turn motors.
- Inertia that resist any changes to motion (whether starting or stopping movement).
- Electrical forces provided by a battery to run the components of the robot.
- An electrical force that is generated by unpowered spinning motors (called a back electromotive force).
- Gravity that pulls down on elevators and arms (and sometimes a robot going up or down a ramp or pulling it over).
- Centrifical forces when a robot takes a corner too fast.
- Magnetic forces used in some shaft encoders.

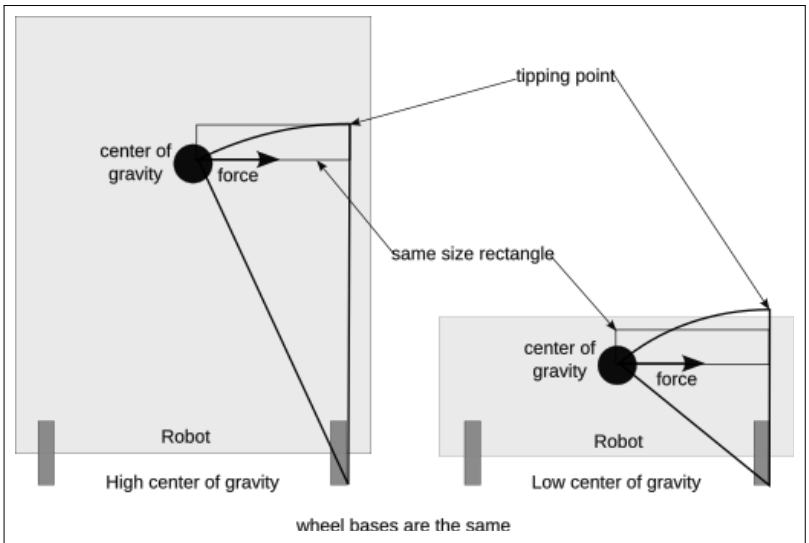


Figure 11: Effect of high center of gravity

Sometimes there are unexpected forces which can have an unwanted effect. The center of gravity of an object is the point within the object where the mass of the object is perfectly balanced: front-to-back, side-to-side and top-to-bottom. Building mechanisms on the robot to increase its height, move the center of gravity higher and higher. Having high and heavy mechanisms lift that center even higher. This creates a problem in that the center of gravity exerts a force that can act as a lever against the robot wheels when the robot changes motion. When a robot accelerates or decelerates the center of gravity exerts an inertial force to oppose the change in motion. When the robot makes a sharp turn the center of gravity exerts a centrifugal force to oppose the turn. Either force acts upon the center of gravity and levers off a wheel as a fulcrum to lift the robot onto that wheel. If the force is strong enough it can lift the robot to the tipping point where the center of gravity of the robot is over the outer edge of the fulcrum wheel and the robot will topple. Robots

with a lower center of gravity and the same wheelbase have to lift the center of gravity higher to reach the tipping point and are therefore more stable. The kicker is that robots with higher centers of gravity also have a longer lever to lift the weight (the distance between the wheel edge and the center of gravity), so less force is needed to lift the robot.

2.4 Some Basic Electrical and Electronics

An electrical circuit is a loop which moves electrons from a source to a component which consumes power and back to the source. Think of a circuit as a closed circle and electrons moving around that circle. Break it anywhere and the current stops. A source is typically a battery, but it can be a power supply connected to power lines. The consuming component may be things like a resistive heating element, a light, a motor. For simplicity these diagrams show a simple resistive load.

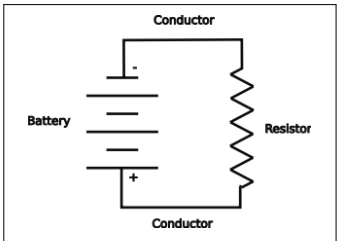


Figure 12: A simple circuit

The battery could be "shorted" where its positive terminal is connected directly to the negative terminal. This is not a good idea, because the interconnecting wire was little resistance and a large flow of electrons will occur rapidly heating the wire. With a large enough battery, like that used in FRC robots, this heat can burn body parts or the conductor itself. This is one of the most dangerous parts of the robot, so it should be always treated with respect.

In this circuit the current flows from the positive terminal of the battery through the conductors and the resistor to the negative terminal of the battery. The resistor resists the current flow and heats up. In this circuit the current is called **Direct Current** or DC as it flows in only one direction, positive to negative.

There are few concepts about this circuit. The battery supplies power. It has a **voltage** which is the pressure that moves the electrons. Higher voltages have higher pressures and cause more current through a given resistance. The **current** flow is measured in **amperes** or amps. **Resistance** is measured in **ohms**. A resister with more resistance has more ohms. Power is supplied by the battery and consumed by the resistor. Power is measured in watts which is the current multiplied by the voltage. Energy is the power used over a period of time. In homes this is measured in thousands of watt-hours or kilo watt-hours (kWHr). Robots use a battery with a fairly constant voltage, so energy is measured in amp-hours. Batteries have an amp-hour rating to tell you how much energy can be stored in the battery.

2.4.1 Parallel Circuit

In a parallel circuit the components are hooked up in parallel. Each component has the same voltage. If the component is a load like a resistor and the resistance is equal, the current is divided equally in each component.

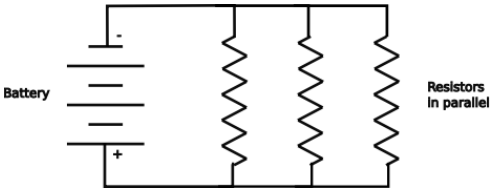


Figure 13: Resistors hooked up in parallel

2.4.2 Series Circuit

In a series circuit the components are hooked up end to end. Each component has the same current. If the component is a load like a resistor and the resistance is equal, the battery voltage is divided equally between each component.

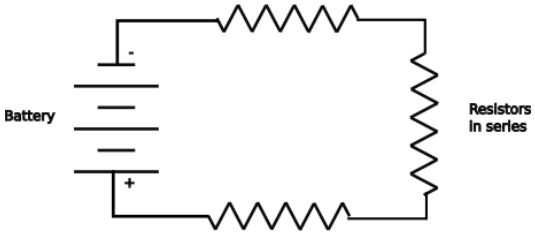


Figure 14: Resistors hooked up in series

2.4.3 Measuring Current and Voltage

Current is measured with an ammeter and voltage is measured with a volt meter. The circuit diagram shows both to measure the current and voltage applied to the resistor.

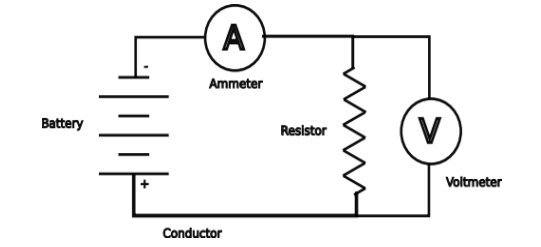


Figure 15: Circuit with an ammeter and a volt meter

The ammeter is wired in **series** with the resistor so that the same current passes through both. To measure the current of a circuit, the ammeter must be inserted into the circuit. In this circuit you may note that some current will also flow through the volt meter. Most volt meters are designed with high impedance (nearly the same as resistance), so that very little current flows through the volt meter and its measurements do not affect the circuit under test.

The volt meter is connected in **parallel** to the resistor so that it measures the same voltage as the voltage applied to the resistor. The volt meter could be placed across the battery to measure the battery voltage. Although they would look nearly the same on a circuit diagram, the voltages may be different because the conductor is not perfect and has some resistance of its own. Measuring across the load or resistor measures the voltage that is delivered to the load.

It should also be noted that all batteries have some internal resistance, so the voltage measured at the battery will drop under high current load. This resistance will increase as the battery ages. So measuring the battery voltage with no load will be higher than when measured under a load. The internal resistance will heat up and can be dangerous for batteries under high load. This is another reason why you should never short out a battery.

2.4.4 Relationships Between Voltage, Current, Resistance and Power

The mathematical relationship between voltage (E), current (I) and resistance (R) is show in the following:

$$E = I \cdot R$$

What this means for most components with a fixed voltage, that higher resistance results in a lower current or a lower resistance results in a higher current.

This may be rewritten as the following to solve for current or resistance

$$I = E / R$$

$$R = E / I$$

A fourth electrical measurement is power (P) expressed as watts. It is the product of voltage and current:

$$P = E \cdot I$$

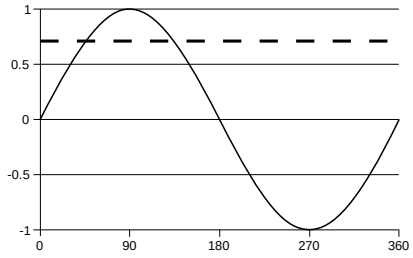
This can be rewritten when combined with the previous expressions as:

$$P = I^2 \cdot R$$

$$P = E^2 / R$$

2.4.5 Alternating vs. Direct Current

Power that comes out of an outlet in your home or school is called **alternating current** or AC. The voltage traces a sine wave and sends the current in one directed and then sends the current in the opposite direction. In the US and Canada this happens 60 times a second, so it has a frequency of 60 Hz. AC is used in power distribution because it has been historically easy to change its voltage up or down using transformers. A transformer has two coils where the magnetic field of one coil is coupled to the other coil. This coupling allows current in one winding be induced into the winding of the other coil and hence the current is able to pass through the transformer. The ratio of the number of windings of the two coils determines the ratio of the input voltage to the output voltage. So if the input coil has twice as many windings as the output coil, the output will have half the voltage as the input. This is a step-down transformer. If the ratios were reversed, the output would have twice the voltage as the input. This is a step-up transformer. AC current measurement may be measured like DC currents, or it can use a “clamp” device to encircle a conductor and sense the current in the wire inductively just like a transformer. This works only for AC because the electrical inductance requires that the voltages be changing for currents to be induced.



(Dashed line represents the root-mean-square value read by volt meters.)

(From Wikipedia)

Figure 16: AC voltage over time

These last two relationships show something interesting when trying to move power from one place to another. All real world conductors (which rule out super conductors in a laboratory environment) have resistance and that resistance means some power is lost in delivering power. Bigger conductors of the same material have less resistance than smaller conductors. Some materials like gold and silver conduct with less resistance than copper or aluminum, but are also more expensive. For a given power line with a fixed amount of resistance, more power can be delivered with less loss using a higher voltage than a lower voltage. Also, the same power can be more efficiently delivered with a higher voltage and a lower current and a smaller more economical wire. Transmission lines have traditionally used alternating current because of its ease of transforming its voltage. Transmission lines use a higher voltage, but this voltage is lowered with a transformer to a lower safer voltage at various stages of distribution.

2.4.6 Energy

Energy is power used over time and is typically measured in watt-hours or kilowatt-hours. Batteries take a slightly different approach and use amp-hours or ampere-hours to measure battery energy capacity as they have a relatively fixed voltage. Amp-hours should be multiplied by the battery voltage to get the true measure of energy, watt-hours.

A 12 volt robot battery can supply around 120 amps. This is 1440 watts. A direct short circuit moves all the available current through the circuit caused by a short. This power is dissipated by heating the wiring of the circuit. It will get hot enough to melt insulation, conductors and cause severe burns to human body parts.

Robots can monitor their use of energy by monitoring the amount of current used and multiplying by voltage and the sample time. By adding up its energy usage over time (integrating), you can find how much energy has been used and subtracting that from the battery capacity you can find how much energy is left in the battery. Further monitoring the voltage can indicate when the battery capacity is failing prematurely as it will as it ages.

2.4.7 Polarization

Polarized components must be connected properly. The positive terminal must be connected to positive wires. The negative terminal must be connected to negative wires. The battery is a polarized component as are many electronic controllers. FRC requires that positive wires be colored red and negative wires be colored black.

2.4.8 Switched Circuits

So far we have described circuits that are always on. These have some uses, but robots are more interesting when we control power to the various loads and motors in the robot. The simplest way to control power is with a mechanical switch. In a schematic is normally shown as in the diagram to the right with the switch open.

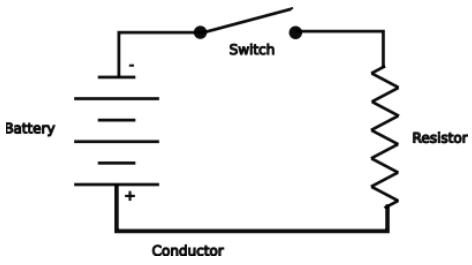


Figure 17: Switched circuit with the switch open and no current flow.

This switch may also be closed as in the next diagram to complete the circuit and allowing current to flow through the load resistance.The switch in the schematic is a generalized form of any switch, although it is generally thought of as a toggle switch: flip it one way to turn on and flip it the other way to turn off.

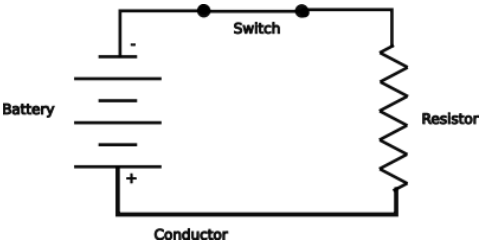


Figure 18: Switched circuit with the switch closed and current flow through load resistor

A push button or intermittent switch may be depicted as in the following diagram to the left. The button is press to complete the circuit and released to open the circuit.

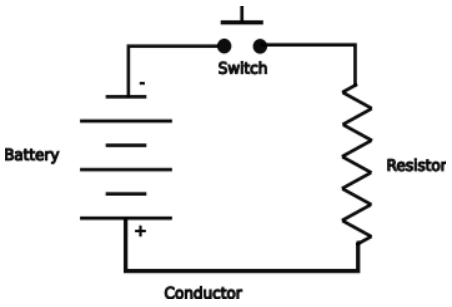


Figure 19: Simple switched circuit with a momentary (push-button) switch

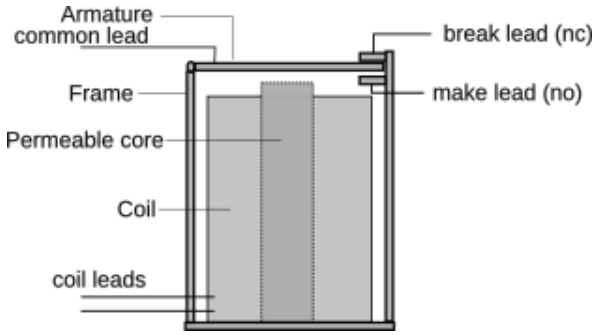


Figure 20: Electrical relay

The switch could be replaced by a relay or other control device to allow the switch to be electrically controlled. Relays are used in high voltage or high current circuits. They are controlled by a low voltage, low current coil. The relay has a coil wound around a permeable core to concentrate the magnetic pull of the coil acting as an electromagnet. The energized magnetic field pulls down on the armature. The armature is the common lead and there are two other possible leads: a make lead that is normally open (n.o.) and a break lead that is normally closed (n.c.). Relays have lost favor in some electronic circuits because they have a back electromotive force (EMF) which can be harmful to modern integrated circuit devices like microcontrollers.

Solenoids are a similar to relays and are used where even bigger currents or heavier mechanical mechanism like a pneumatic valve is needed. A solenoid is a coil wound around a hollow core. A permeable plunger is pulled into the solenoid when its field is energized. This movement is used to control a valve or switch. FRC robots use solenoids to control pneumatics. Your car probably uses a solenoid to control the power going into the starter motor.

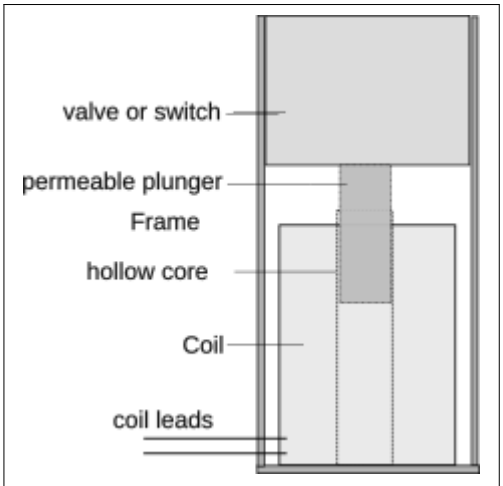


Figure 21: Solenoid

2.4.9 Electrical Schematic Diagrams

Many of the examples of simple circuits have employed a diagram with some symbols. There are many symbols for electric and electronic components, some of which are summarized in the figure below. A circuit diagram is usually called a schematic. A schematic assists when debugging problems in complicated circuitry to figure out what wire goes where.

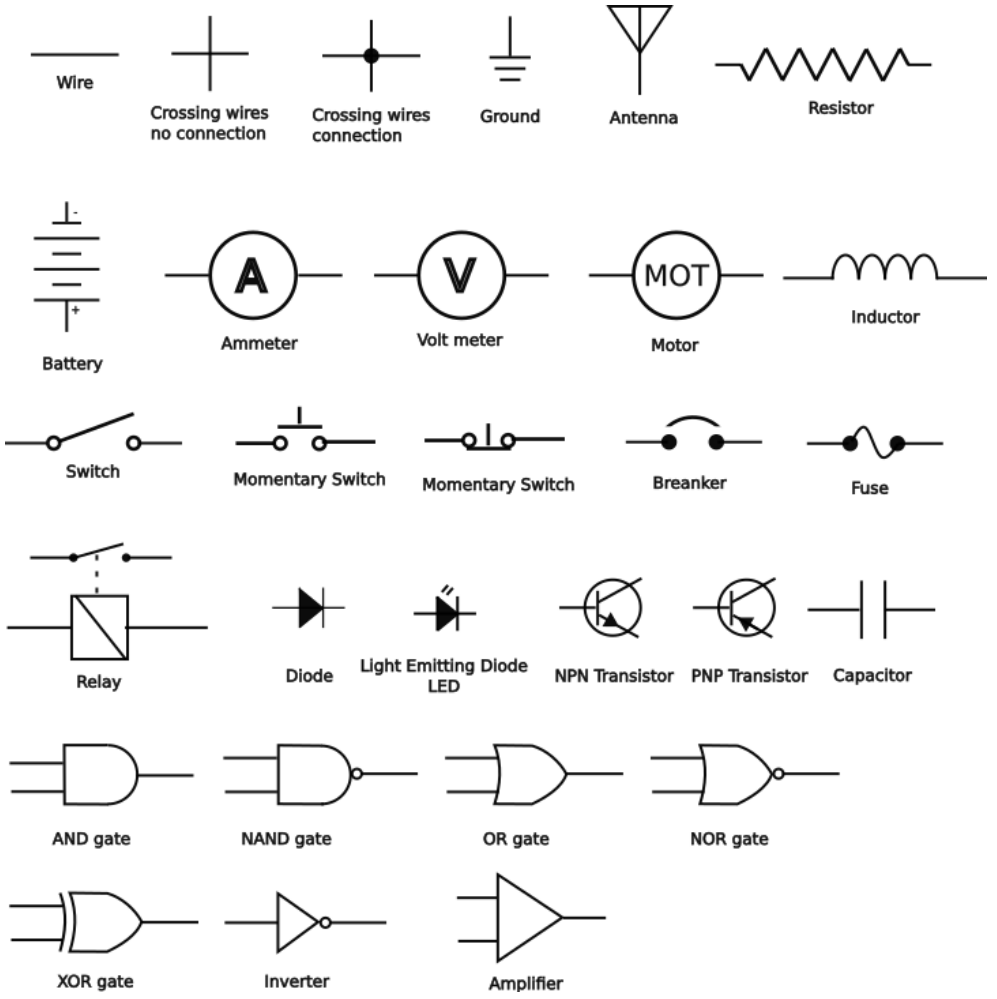


Figure 22: Symbols for electrical schematics

2.4.10 Circuit Breakers and Fuses

Another specialized switch is a circuit breaker. A circuit breaker can be manually opened and closed like a switch, but it is not designed to do this, so it will wear out fairly quickly (and it is more expensive than a switch). The special thing about a circuit breaker is that it will open or trip when an excessive amount of current flows through it. The tripped breaker can be manually turned on again without other action.

Another type of circuit current protection is a fuse that will melt and physically open the circuit when excessive current flows through it preventing further current flow. A "blown" fuse must be replaced before the circuit will operate again.

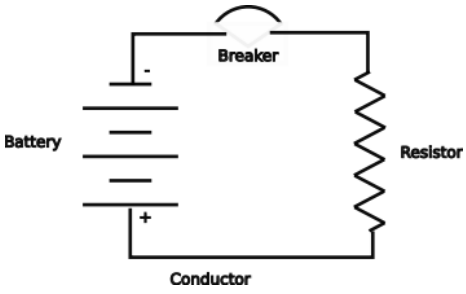


Figure 23: Simple circuit with a circuit breaker

2.4.11 Power Distribution

Power distribution is a method for safely distributing power to various components while protecting each component with over-current protection. The battery in an FRC robot is capable of supplying around 180 amps. This requires heavy wires and caution when working around the leads coming from the battery as a short circuit across the batter will unleash the full power of the battery. The power distribution panel or hub is used to take the full battery current and break it up into smaller safer branch circuits. Each of these branch circuits has its own protective fuse or circuit breaker to limit the current that circuit can use. With current limiting smaller wiring can be used safely. Using a fuse or circuit breaker that is in excess of what the branch circuit can handle is also dangerous as it would allow enough current to physically burn out the circuit or its components.

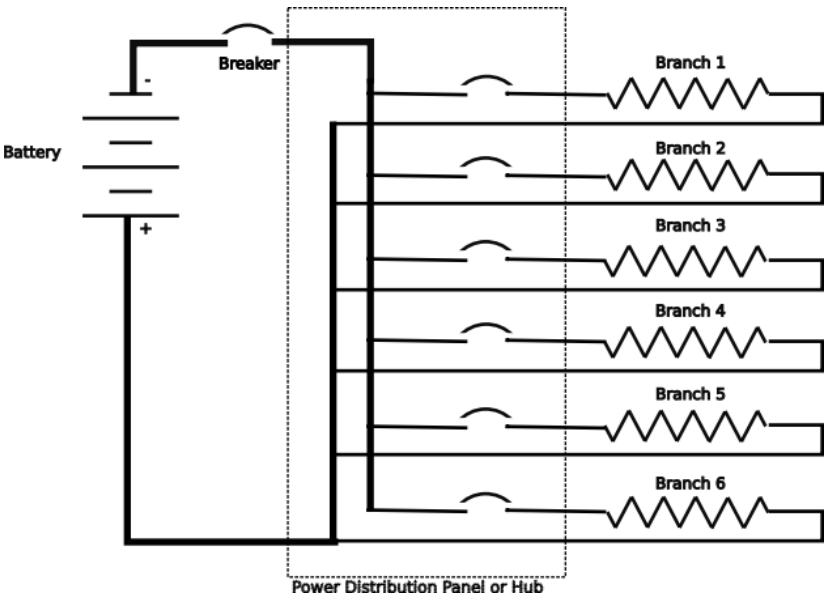


Figure 24: Power distribution in a robot

Each circuit has a limit as to how much current it can carry. Motors use 30 to 40 amps of current. LED strings and some coprocessing units may use only 5 amps.

Only one battery can be used in a robot, so the total amount of current available is limited and must be budgeted so that each circuit can get the amount of power that it needs. If more current is requested than is available, a brown out condition will occur where the supplied voltage may fall below the minimum operating voltage causing some components to temporarily fail. A robot may monitor the amount of current being requested in real time and apply current budgeting dynamically by further limiting how much current a device is using to ensure the brown out condition is avoided.

The FRC rules limit one motor per branch circuit. This allows the limiting current to be closer to the current that can damage an individual motor.

FRC rules also require that a separate wire be used to return the current to the distribution panel. If wires are joined, the resulting wire carries the current from all participating branches and may exceed the capacity of the wire.

This scheme is not perfect. A stalled motor will draw as much current as it can. Sometimes this is enough to burn up the motor without tripping the circuit breaker. Software should monitor the current used in a motor circuit to detect the stalled condition and limit how much current is allowed to the motor by the motor’s controller to prevent motor burn-out.

2.4.12 Analog vs. Digital

A robot has both analog and digital circuits. Analog circuits are characterized by a time varying voltage and information is conveyed with that voltage. Traditionally analog voltages controlled the speed of motors and the brightness of lights. Now analog circuits are limited to a few sensors inputs like some distance sensors and shaft encoders. Digital circuits are characterized by sending information as a series of symbols. A symbol usually is a pattern of timed pulses between two voltages, typically on and off. The digital circuits are typically communication channels like:

- Ethernet
- USB (Universal Serial Bus)
- CAN bus (controller area network)
- I^2C
- LED strings

The symbols carry bits of information and in many protocols carry timing information to properly interpret the meaning of a time varying signal. Each protocol has its own specification for its symbols. Bits have two distinct values: 1 and 0. These values may be expressed as a voltage, a current, or a differential voltage. A voltage below a specified ceiling is a 0 and a value over a specified floor is a 1. Values between the ceiling and the floor are indeterminate. Recovering the timing signal or clock is necessary to sample the signal when it is stable and not when the signal is transitioning between two symbols.

Ultimately all signals have analog characteristics. They all have a time varying voltage or current. A CAN bus, a high speed digital signal, must be properly terminated to eliminate analog echos that wipe out the meaning of the conveyed digital signal.

2.4.13 Pulse Width Modulation (PWM)

Some motors and lights can be controlled with a **Pulse Width Modulation (PWM)** signal. In PWM the voltage is turned on and off at a set frequency or period. The duty cycle of the on pulse is varied. The percentage of on time to pulse period is the duty cycle and that cycle determines how fast a motor can go or how bright a light will be. A 100% duty cycle is on all the time; 50% is on half the time; 10% is on 10% of the time. This means that a 50% duty cycle 12 volt motor will run at the same speed if it were supplied 6 volts. Some care must be used with PWM to ensure that the correct base frequency is used. The roboRIO uses three periods 5, 10 and 20 ms (or the frequencies 200, 100 or 50 pulses per second (pps) respectively). Motors have a lot of mass compared to most electrical components so they treat a PWM voltage almost the same as the equivalent analog voltage. It is easier for computer controlled circuits to provide the PWM voltage than the equivalent analog voltage.

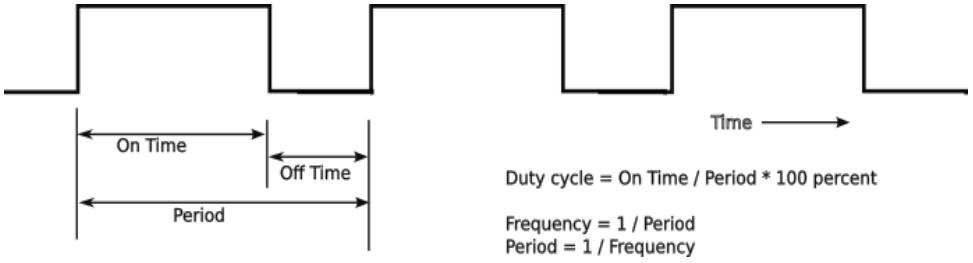


Figure 25: Duty cycle of a voltage waveform for use as Pulse Width Modulation

In the roboRIO the PWM signals are generated by its Field Programmable Gate Array, see the WPILib documentation for setting up the software to program the FPGA to generate the PWM signal.

2.4.14 Field Programmable Gate Array (FPGA)

The roboRIO has an FPGA which is an array of logic gates that can be programmed to provide a number of functions in hardware rather than take up processor time to do them in software. Specifically for FRC robots, it can control PWM channels and a channel for LED lights.

See the WPILib documentation to configure the roboRIO to us the FPGA to generate the serial protocol used for WS2812 LED strings.

2.4.15 Light Emitting Diodes (LEDs)

The WPILib allows the FPGA in the roboRIO to control a set of serial WS2812 RGB LEDs. These LEDs may be in one or more strips or matrices. The number of LEDs is limited to the supply current available and the time it takes to shift out all the LEDs.

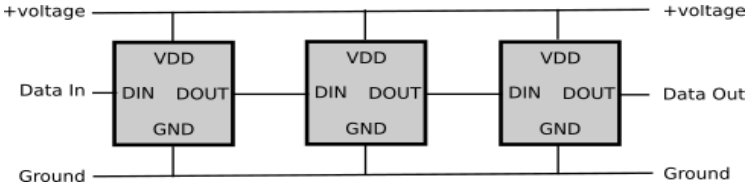


Figure 26: WS2812 LED cell interconnection

Each cell of a WS2812 consists of three LEDs which are controlled by a PWM signal that uses an 8-bit input to control the brightness of that one color. The value of the PWM is latched so it need not be sent continuously. Each cell is addressable by its position in the string. The string is accessed serially, so only one signalling connection is needed. The first LED in the string takes the first 24 bits to control its LED and passes the remaining bits down the string. The second LED in the string takes the first 24 bits that it sees (really the second set sent) to control its LED and passes the remaining bits down the string. This is repeated for all LEDs in the string.

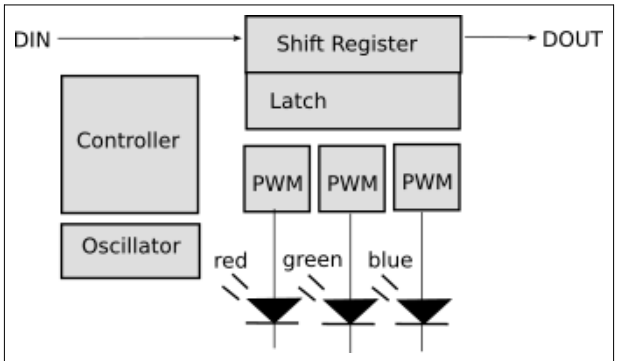


Figure 27: WS2812 cell block diagram

The LED string has three basic connections: +5v power, ground and a control signal sent serially over DIN (data in). The serial signals are daisy-chained to the next cell in the series by connecting the DOUT of one cell to the DIN to the next cell in the chain. Each cell consists of an integrated controller, a shift register and three channels of PWM to control the brightness of the individual red, green and blue LEDs for that cell. Once the PWM values for a cell are received, they are latched, it continues controlling the brightness of its LEDs.[12]

1 Talk is to add a 220 to 470 ohm resister to the data line and a 100uF to 1000uF capacitor across the supply of the strip to smooth out power and reduce noise.

2 WPILib use 4 bytes per cell but does not support RGBW LED strings, each cell of an RGBW string has four controllable LEDs including a white cell. (as of 2024)

Signals to the WS2812 are sent serially as a sequence of one and zero symbols. Each digit signal takes the same total amount of time, but the timing differences allow for distinguishing between a one and a zero as well as to recover the timing signal. Between the sequences for the entire string of LEDs, a reset symbol must be sent to start the process over again. Timing of the symbols is in the figures.

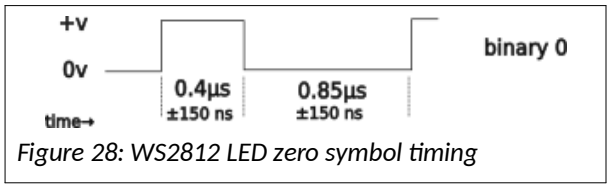


Figure 28: WS2812 LED zero symbol timing

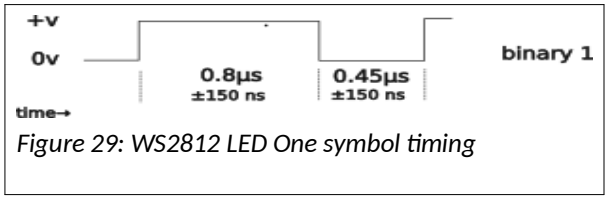


Figure 29: WS2812 LED One symbol timing

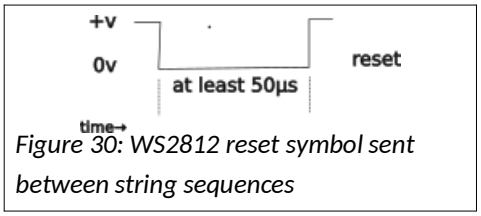


Figure 30: WS2812 reset symbol sent between string sequences

Each LED cell is sent 8 bits for each color with the high bit sent first for each color and the colors sent in the order green, red, blue.

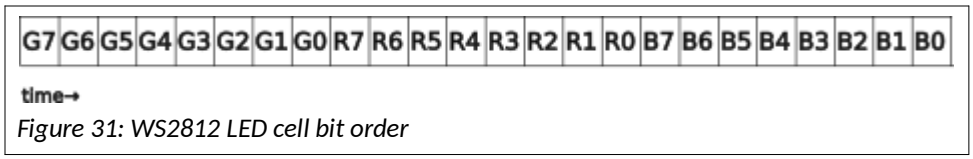


Figure 31: WS2812 LED cell bit order

The overall timing for sending a string of LEDs looks like the following figure:

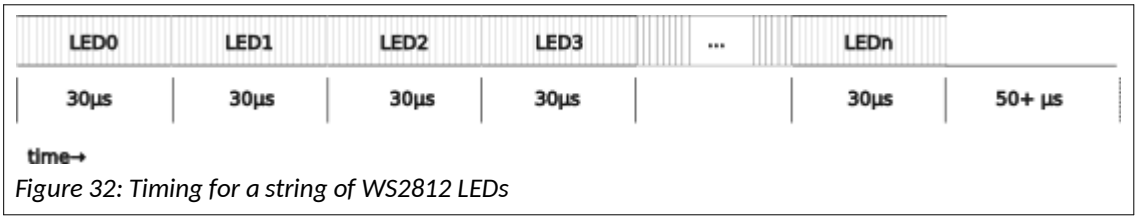


Figure 32: Timing for a string of WS2812 LEDs

Strip timing can be computed by the following formula:

$$stripTime = 30n + 50\,\mu s \qquad (\text{where } n \text{ is number of LED cells})$$

Each cell draws 50 ma, so a dense strip of 144 cells draws 7.2 amps at 5 volts which is a huge draw on a robot with other higher priority uses for power. Power consumption can be decreased by:

- Decreasing the brightness.
- Avoiding white which uses all three LED colors.
- Use patterns that do not use every LED.
- Use flashing or other periodic technique to decrease the average on time.
- Monitor both the overall power usage and the LED power usage to be aware of its impact.

2.4.16 Electric Motors

Electric motors convert electrical energy into motion or kinetic energy. They work with an interaction between electromagnets and fixed magnets. They fall into two basic types:

- Brushed DC motors
- Brushless DC motors

2.4.16.1 Brushed DC Motors

A brushed DC motor has electromagnets mounted to the part that turns called an armature or rotor. Power for these electromagnets is delivered through a pair of brushes contacting the commutator on the axel of the rotor. The commutator is insulated from the rotor shaft and has conductive segments to connect to the right electromagnet depending on the orientation of the shaft. As the rotor turns, different commutator segments are contacted by the brushes causing the different rotor electromagnets to turn on, turn off, and change polarity.

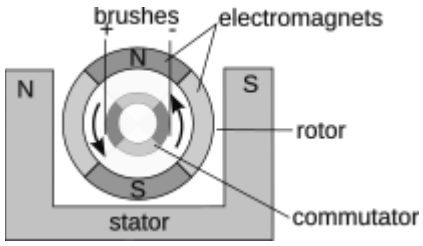


Figure 33: Basic brushed DC motor

The energized electromagnets interact with permanent magnets mounted to a fixed frame called the stator. These fields alternately repel and attract the stator magnets causing the rotor to turn. The commutator is designed to avoid powering the electromagnet where the direction of rotation would be ambiguous thereby preventing a stalled motor.

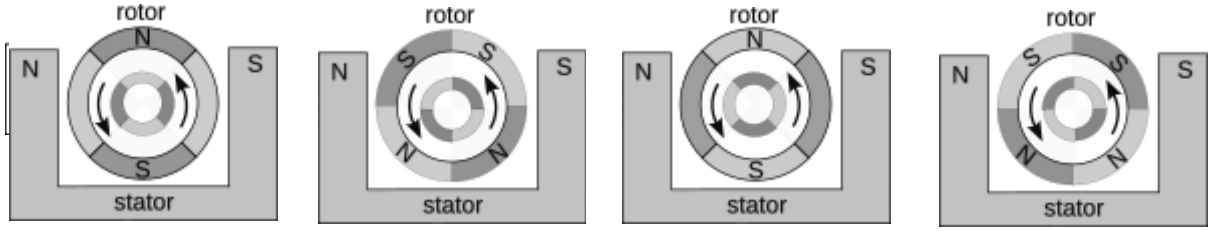


Figure 34: Time sequence of a brushed DC motor movement

The contact of the brushes with the commutator causes arcs which in turn is the source of electrical noise. The arcing also slowly destroys both the brushes and the commutator. These motors are simple and cheap.

2.4.16.2 Brushless DC Motors

Brushless motors flip the design of the motor around so that the fixed magnets are mounted to the rotor and the electromagnets are placed in the stator. There may be more than one pair of fixed magnets in the stator and more than one pair of electromagnets in the stator. A controller turns the electromagnets on to attract or repel the stator at more precise times allowing the motor to develop more torque and to turn at a commanded speed. Since there are no brushes, there is nothing to ware out. These motors do require a controller and that makes them more complex and expensive. In general a second motor controller is not required.

2.4.17 Controllers and Control Theory

Many devices have controllers, but especially motors. A controller controls a motor to aid in performing its specific function. In control theory, controllers can be one of two basic types: open-loop or closed-loop.

An open loop controller controls the device in a fixed way. A timer can turn a device on for an amount of time and then turn it off. Microwave ovens use mostly open loop timers to control them. Flashing LEDs are controlled by open loop controllers. A common switch may also be used as an open loop controller. A closed loop controller includes a sensor to determine when the device needs power or not.

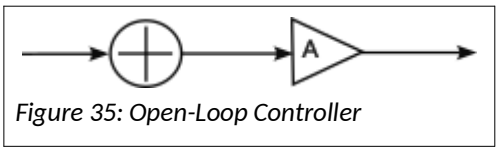


Figure 35: Open-Loop Controller

An oven with a thermostat is an example of a closed loop system. The oven heats up to the set temperature and then the temperature is reached, the oven is switched off. When the temperature drops, the oven will be turned on again. The oven temperature is used to control the oven heating element. Block B may contain a control algorithm which can be used to "shape" the feedback signal, so that provides a stronger signal when the error (target – current) is great and small when the error approaches zero. This is a proportional controller. Controllers using appropriate proportional feedback have less tendency to overshoot the target goal because they back off power as they get closer to the goal.

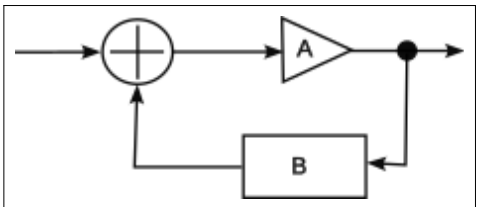


Figure 36: Closed-Loop Controller

Many motors in current robots have built-in controllers that use the PID (proportional-integrated-differential) or similar motion control algorithm. Brushless DC motors must have a built-in controller just to turn, and this controller may be extended to provide other types of motion control.

More information about control theory can be found at https://file.tavsys.net/control/controls-engineering-in-frc.pdf.

2.4.17.1 Feed Forward

A feed forward term may be applied in some control loops. In general the feed forward term relies on a model of how the controlled system will behave. An elevator may have a feed forward term because it is predictable how gravity will act upon the elevator when a control input is removed. An elevator may need a fixed amount of power to hold its position as it fights gravity, even when it is commanded to be stopped. The amount of the feed forward parameter can be dependent on the perceived or predicted state of the system. An arm affected by gravity may use a feedforward term that accounts for the force of gravity as a function of the current angle of the arm. The control applied to hold an extended arm is greater than the control used to hold a vertical arm unaffected by gravity. The feedforward term may include factor for when the arm is loaded vs. being empty or when it is starting to fight inertia.

2.4.17.2 PID Controller

A PID (proportional-integrated-differential) controller uses an algorithm which uses three types of feedback:

- Proportional to the value of the current error, which is equal to the desired target state minus the current state. The state can be a position, velocity, acceleration, etc.
- Integral or proportional to the accumulated error over time, which should be zero. This term gives weight to a non-zero accumulation to help move it back to zero.
- Differential or proportional to the immediate rate of change of the error. This can be as simple as the difference of the current error value and the last error value ($error_{time=n} - error_{time=n-1}$).

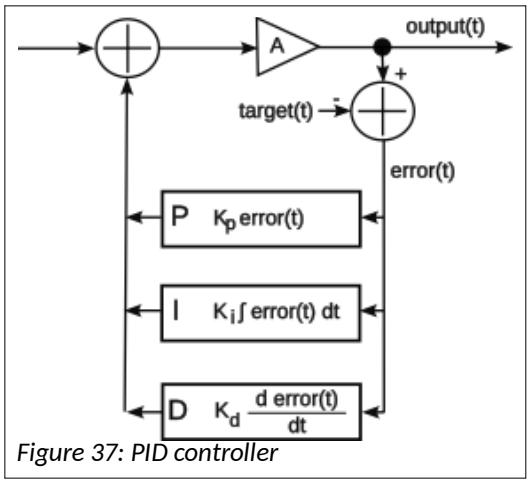


Figure 37: PID controller

Each of the three feedback components has a constant to control the amount of that type of feedback to use. A constant factor called K_p is multiplied against the error and fed as a proportional input to control the motor. A low K_p will be slow to reach the goal and a high K_p makes the control over reactive and prone to overshoot the target. A K_p that is just right will quickly reach the goal and reduce the error to near zero. K_i is the constant multiplied against the integral or accumulation of past error values. A successful K_i value will reduce the accumulated error to zero over time. The K_d term is multiplied by the difference of the past error value. This anticipates future values based on the current rate of change. Tuning is the process of selecting the correct values of K_p, K_i, and K_d often using a trial and error technique. This is usually done in the order K_p first, then K_d, and finally K_i. The constants K_d, and K_i are much smaller than K_p.

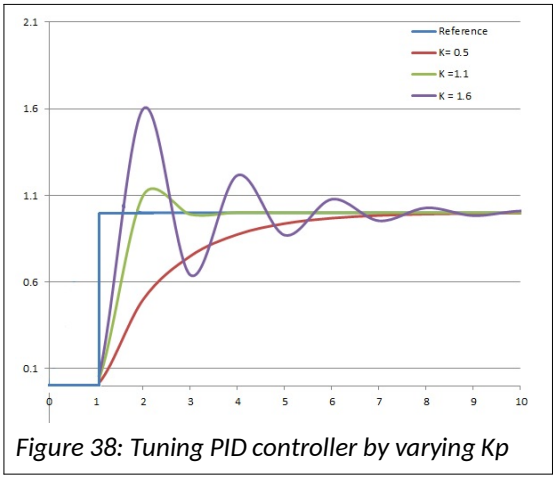


Figure 38: Tuning PID controller by varying Kp

More information can be found at https://pidexplained.com/how-to-tune-a-pid-controller.

2.4.17.3 Trapezoidal Controller

Some FRC motor controllers may provide "trapezoidal" shaping to the velocity target. When the motor is started it is spun up with a fixed acceleration until it reaches the target velocity. This velocity is held until released. When the motor is slowed, it slows with a fixed deceleration until it is essentially at zero velocity.

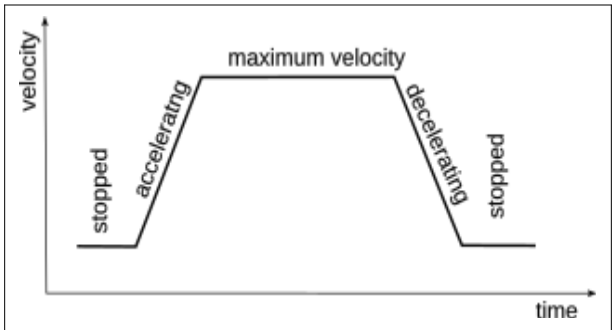


Figure 39: Trapezoidal envelope of motor velocity

The trapezoidal shaping may be enhanced by limited the maximum jerk allowed. This limits the rate of change of the acceleration and deceleration and effectively rounds off the corners of the trapezoidal control curve. A limited jerk factor eases motion transitions to operate the mechanisms more smoothly. Limiting jerk too much makes the controls "mushy" or less responsive.

2.4.18 Gyroscope and the Inertial Measurement Unit

Guidance computers have been using gyroscopes in their Inertial Measurement Units (IMUs) since World War II. The bulky and expensive physical gyroscopes used in early navigation systems have been replaced with a miniaturized part built with a Micro Electromechanical Systems (MEMS). These parts can fit into a small package and are included in fitness trackers, cell phones, and personal navigation systems. MEMS devices are made to have a small part that moves, and that motion changes its capacitance slightly which is measured as a change of voltage. The modern parts have up to nine axes:

- Accelerometer in the x-, y- and z-axes to measure acceleration in each of the separate axes relative to the device. Note that the normal z-axis will measure the earth's gravitational pull at all times.
- Gyroscope roll, pitch, and yaw angles to measure the current orientation of the device. For FRC robots the roll and pitch should normally be zero, except for overturned robots. The yaw is useful to know which direction the robot chassis is pointing.
- Magnetic compass measures the magnetic field strength in the x-, y-, and z-directions to measure the location of the earth's magnetic pole relative to the device. Note that this will measure magnetic inclination. This is not used much in FRC robot competitions because the robots have many motors that interfere with the detection of the weak earth magnetic field.

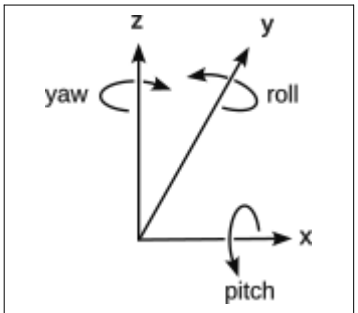


Figure 40: Axes of an Inertial Measurement Unit

IMUs measure acceleration. These accelerations are added up to determine a velocity and the velocities are added up to get a displacement. The displacement may be a distance or an angle depending on the sensor.

Note that the figure shows the y-axis as the forward direction rather than the x-axis as used in FRC. This difference should be accounted for in the software interface to the IMU.

2.4.19 Frequency, Oh, That Baud Hertz!

Signals that occur periodically have a frequency expressed as cycles per second or the unit Hertz (Hz). The speed of digital communication signals are expressed as number of symbols per second or Baud. For many signals there is exactly one bit per symbol, so the bits per second (bps or b/s) is the same value as the Baud. The number of bits per symbol can vary.

2.4.20 Shaft Encoders

A shaft encoder is a device for measuring the rotation of a shaft. This is useful for moving an arm to a specified angle or knowing the position of the robot on the field based on how far its wheels have turned. Shaft encoders come in two flavors: incremental and absolute. Incremental shaft encoders only report on relative measurements, so the computer must keep track of a reference point and accumulate movements to know the current absolute position of the shaft. Absolute encoders just report the position of the shaft and will report the same position without correction, calibration, or accumulation.

Primitive shaft encoders used optics to read lines on the shaft which would be input to a computer as a set of ticks as the shaft is turned. Some of these would not report the direction of shaft rotation, only that the shaft is being rotated. The resolution is limited by the ability to resolve individual lines.

A “Gray code” shaft encoder uses an optical sensor to resolve a “Gray code” etched into the shaft as patterns of black and white. A Gray code is a binary code where only one bit changes for each change in incremental value. This allows the output to report the current position of the shaft as well as the direction that the shaft isd turning. If monitored continually the computer can tell if the reading has errors by skipping values. A four-bit gray code is shown on the right. The resolution is limited by the ability to resolve the coding on the shaft.

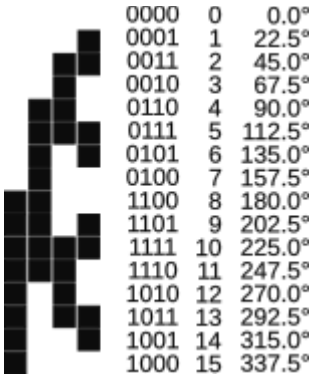


Figure 41: Gray Code (physical, binary, x/16 rotations, degrees)

A Hall effect shaft encoder uses a magnetic sensor to measure the field coming from a magnet permanently fixed to the shaft. This analog measurement varies as the shaft is rotated and is converted to a digital measurement. Some Hall effect shaft encoders can resolve a shaft rotation to 12-bit precision or 1/4096 of a turn or 0.0879 degrees. The accuracy of a Hall effect device may be affected by other magnetic fields or the presence of magnetic materials, like steel or iron, which may distort the measured fields.

2.4.21 A Word on Common, Ground, and Return

All circuits must return current from the load back to the power source or battery. Many real world systems take advantage of this and connect these returns all together. This can be done, but there are some “rules” to remember to keep the circuits safe.

First a little vocabulary:

- Return – that part of a circuit that returns current from a load to the power source.
- Common – another name for return.
- Neutral – a return in household AC wiring. (Household AC systems have two “phases” of power. Using either one and neutral provide normal 120 VAC. Using both without the neutral, the phases are additive and provide 240 VAC. This is really beyond the scope of this manual.) In household

wiring, the neutral wire is white and power wires are any color other than white, green or bare copper.

- Ground – Normally a connection to earth. Some long distance circuits use this as a return. In household wiring the ground wire is green or bare copper. The circuit symbol for a ground is shown to the right.

Figure 42: The ground symbol

For safety circuits include a conductor, called a common or return, as the path to return current of a circuit or branch circuit back to the power source. The conductor is sized to carry the current of the circuit or branch. The earth itself, or ground, is a conductor, and it is sometimes it is used as the return path. This is discouraged for a number of reasons. The connection to the earth may dry out and be lost. When underground pipes are used to make the connection to the earth, they are susceptible to electrolysis which corrodes the pipes.

Household AC power has both a common and a ground. This is done for safety. The common or neutral wire is the normal path to return current, and the ground provides a safe alternate path to return current. Appliance chassis are connected to ground, so that if they are in contact with a “hot” wire, the stray current can be returned safely. This prevents a chassis from ever being connected to the power or “hot” lead. By the way, carrying the return current via the ground lead versus the common lead is a ground fault and there are special devices that detect ground faults and disconnect power from that branch.

In household wiring the common is connected to an earth ground at the main distribution panel. This is done to make sure that the common has the same potential or voltage as the ground. This connection to ground is only done once for each home to prevent ground loops (which have inadvertent current flows and may generate electrical noise).

FRC rules require that the robot chassis cannot be part of a circuit. It cannot be a ground, a neutral, a common, or a supply rail. This is done for so that should a wire come in contact with the chassis it will not complete or short out an existing circuit. Each branch circuit must provide its own return wire that is properly sized to carry the current for that branch. This rule also prevents electrical arcing if a robot component like an arm should be disconnected from the rest of the robot.

2.4.22 Using a Multimeter

A multi-meter is a handy way to trouble shoot electrical problems in a robot. Modern multimeters have a digital display. Older analog instruments have a meter and work on the same principles. The multimeter has a switch to select the desired measurement and range. There is an on-off switch to turn off the internal battery. This switch may be incorporated into the selection switch, or it may be separate. There are two probes: one red and one black. Connect the black probe to the ground jack (three stacked horizontal lines of decreasing length) or common jack of the meter. Connect red probe to the jack appropriate to the desired measurement. Current measurements typically have alternate input jacks because these inputs must be fused to avoid damage to the circuit or the multimeter. Remember that to measure current the meter must be in series with the rest of the circuit. Never measure the current across a power

(From Wikipedia)

Figure 43: A digital multimeter

source like the battery terminals, because this is a dead short and will likely blow the fuse internal to the multimeter.

Some multimeters have a current clamp. These can be used to measure current on Alternating Current (AC) circuits like those in your home or school as the sensor measures the rising and falling magnetic field around a conductor. A current clamp with a Hall effect device can measure direct current amperage, but these are less common.

A hand held multimeter is a useful device for troubleshooting electrical problems on the robot.

2.4.22.1 Using a Multimeter Safely

When using a multimeter there are some safety considerations to remember.

- Measuring voltage or current is working on a live system and the meter may put the system's voltages or current where they are not expected.
- Make sure that the probes are properly connected to the multimeter before attempting to probe connections within a device.
- Make sure that the selection switch is appropriate for the desired measurement. Some meters may be damaged if this is not done correctly.
- When probing make sure that the probe is not causing an unintended short circuit by bridging two conductors.
- Make sure that you are not touching any part of the device. You should only be touching the probe handles.
- Do not touch the ends of multimeter probes while probing a circuit as the circuit voltage may be transferred to the other probe.

2.4.22.2 Measuring Voltage

To measure a voltage:

- Adhere to the safety practices outlined in 2.4.22.1 Using a Multimeter Safely.
- Select the desired range of the measurement and the appropriate AC or DC type of voltage being measured. Some meters are auto-ranging, meaning that they select the range automatically. The type of measurement still has to be selected.
- Turn on the meter (if not done in the previous step).
- Taking care not to cause short circuits, to access dangerous voltages, or to touch any part of the circuitry being examined; put the end of the black ground probe to the circuit ground.
- Put the end of the red probe to the point in the circuit where the voltage is to be measured.
- Read and note the voltage measurement on the display.
- Turn off the meter.

2.4.22.3 Measuring Current

To measure a current:

- Adhere to the safety practices outlined in 2.4.22.1 Using a Multimeter Safely.
- Determine where in the circuit you would like to measure the current.
- Break the circuit at that point.
- Set up the meter for a current measurement by selecting the proper range and appropriate AC or DC type of current. You may need to change the red probe to a different input. Note and heed the limitations of meter.
- Turn on the meter (if not done in the previous step).
- Insert the meter into the circuit where it was broken above.
- Read and note the current measurement on the display.
- Turn off the meter.
- Restore the circuit to its original state.

2.4.22.4 Measuring Resistance

Measure resistance only on unpowered components. Attempting to measure resistance in active circuits may damage the meter. Attempting to measure resistance of wired components may give unexpected results. Some components like diodes, transistors, and LEDs are polarized and will read different resistances in when hooked up in reverse. Resistance measurements allow color-blind people to read resistor values.

- Adhere to the safety practices outlined in 2.4.22.1 Using a Multimeter Safely.
- Select the range of resistance that seems appropriate. Some meters are auto-ranging, meaning that they select the range automatically, but you still have to select a resistance measurement.
- Turn on the meter (if not done in the previous step).
- Put one probe on one end of the component.
- Put the other probe on the other end of the component.
- Read and note the resistance measurement on the meter.
- Turn off the meter.

2.4.22.5 Measuring Continuity

Continuity is really just an electrical connection. You want to verify that a wire is electrically connected you test its continuity from one end to the other. You want to make sure that a return circuit is not connected to the chassis, so you verify there is no continuity between the return and the chassis.

Continuity measurements by their nature are done on wired systems. Do not test continuity on powered up circuits to avoid damage to the meter. Be aware that continuity measurements inject voltage and current into the circuit and may damage sensitive components.

You can measure continuity in two ways. One way is to measure the resistance. No continuity means a very high resistance like several megohms. Continuity means a few ohms of just the interconnection wires. The easier way is to use the continuity setting on the multimeter and let the meter beep when

continuity is found. You can test the meter by connecting the two probes together. The beep is handy because you can focus on the placement of the probes and not worry about the actual meter reading.

- Select the continuity measurement.
- Turn on the meter (if not done in the previous step).
- Verify the meter is set up properly and touching the probe together and hearing the beep.
- Put one probe on one end of the test.
- Put the other probe on the other end of the test.
- Listen for the beep to verify continuity or confirm a low resistance.
- Turn off the meter.

2.4.23 FRC Wiring Color Code

FRC rules require that wires are colored to reflect their function and usage. Wires of a particular color only connect to wires of the same color.

- Red-- wires with positive voltage.
- Black – wires with common (or negative) voltage.
- White –wires that carry a signaling component such as a PWM or cable.
- Yellow – wires that are the high side of the CAN bus (CAN-H).
- Green – wires that are the low side of the CAN bus (CAN-L).

More information on FRC robot wire is found in the WPILib documents.

Connecting components to the wrong polarity may damage or destroy the component.

2.4.24 Capacitance

Another electrical component is a capacitor. It is basically two plates separated by a dielectric which may be air or a polymer. The larger the surface area of the plates the larger the capacitance. Capacitance is the ability to store a charge. So you can think of this as a battery, and it does find this use in some circuits. It has been used for small timing circuits because a capacitor could be quickly charged up, and that charge could be bled off with a resistor in parallel to it. The amount of time that it takes to discharge to 1/5 of its original voltage is a product of the capacitance the resistance, the so-called RC time constant.

You will find many small capacitors in digital electronics boards. Capacitors have the property of passing high frequencies and blocking DC. So they can couple high frequency signals, or they can be used to shunt high frequency noise to ground, thereby quieting the circuit. Digital signals with abrupt changes from a 0-condition to a 1-condition, generate high frequency noise. The squarer the signal, the more noise is generated. The shunting capacitors eliminate that noise, but they “round off” the corners of the square signals. Round off too much and you lose the signal.

Capacitance is measured in Farads. One Farad is very large, so most capacitors are measured in micro Farads (µF or 10^{-6}) or pico Farads (pF or 10^{-12}).

2.4.25 Inductance

A coil of wire has inductance. Passing a current through the coil will produce a magnetic field, so on the onset of the current, energy is used to build up that field. When the current is removed, the magnetic field collapses and this produces a voltage, called a back electromotive force or back EMF. If the current is controlled by a switch, this back EMF is strong enough to produce arcing on the switch contacts. This arcing corrodes the contacts and eventually will form pits and "wear out" the contacts.

A motor is a form of an inductor, but the electromagnetic energy is used to turn a shaft. When the voltage is no longer applied, the motor becomes a generator as long as the shaft is still turning. This is another form of the back EMF. In robotics, there is a choice what to do with this back EMF. It can be left open, and the motor spins down slowly based on its internal friction. It can be shunted to a resistor and the back EMF heats up the resistor consuming the kinetic energy of the motor and the momentum of what it is attached to like a chassis, elevator, arm, etc. This effectively is a brake on the motor. This braking is desirable in some cases like drive motors and unwanted in other cases like shooter motors.

Inductance is measured in units of Henries.

Transmission lines like Ethernet cables and CAN buses are designed to minimize their external inductance. This is primarily done by twisting the wires so that currents in the cable do not induce currents in adjacent cables and vise versa. An Ethernet cable internally consists of four twisted pairs, each with a different twisting rate. This is to prevent each pair from interfering with signals in the other pairs. This twisting balances the capacitance and inductance of the wires that make capable of carrying high frequency signals. Transmission lines need to be terminated in a resistor to prevent reflections.

An inductor will have low impedance or resistance to low frequency AC or DC circuits. At higher frequencies, the inductor is alternately building and releasing electromagnetic energy and has a higher impedance. It can act as a block to high frequencies. Some cables employ ferrite beads wrapped around them to block the transmission of high frequencies. This is to block interference from other signal sources and to prevent the cable from interfering with others.

The difference between impedance and resistance is that the impedance is frequency dependent and the voltage and currents may be out of phase with each other. Power in circuits is measured in volt-amps rather than watts to correctly measure the apparent power consumed by the circuit. In inductive circuits, the voltage and current will have a phase difference. This usually does not apply to FRC robots.

2.4.26 Black Box

A black box is a theoretical device that is an engineering tool to understand how a system behaves. All of its inputs and outputs are measured and monitored. From those observations the functionality of the black box can be ascertained.

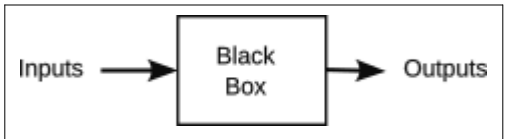


Figure 44: Theoretical black box

FRC robots use the concept of a black box in a couple of ways. One way is the simulator which is used to replace the hardware. If done well the software controls under test cannot tell whether the underlying system is the real physical devices or a software simulation. This uses a Hardware Abstraction Layer (HAL) to form an interface between the hardware and the control system. Either the simulator or the real system can be hooked up to the HAL.

The AdvantagieKit logging system takes this one step further. It captures all the inputs and outputs to the robot, thereby treating the entire robot as a black box. By recording those inputs, the same conditions can

be replayed to a robot with modified software and the outputs compared. To learn more about AdvantageKit there is documentation on their GitHub site: https://github.com/Mechanical-Advantage/AdvantageKit?tab=readme-ov-file.

Airplane black box controllers record inputs similar to the AdvantageKit recordings, but here they are not acting as an engineering black box, but rather recording what happens to an airplane as if it were a black box. By analyzing the recorded inputs and outputs, failures can be traced backwards to a cause.

2.5 Some Basic Math

FRC robots use a lot of math to make them work. A lot of that math is hidden so you don't have to deal with it, but the more you know about what is going on, the more you can take advantage of it. Besides that the math is fun to use.

This starts off simple with things that you should know from middle school like measurements and units. The coordinate systems should start off being familiar, but gets more complicated when having to translate between different reference systems because trigonometry in three dimensions is sometimes involved. Trig is also used in kinematics and odometry. Odometry really is simple calculus integration using arithmetic sums of small incremental movements. These skills all build on each other. If you don't get it at first, don't worry about it, but come back later and try again.

2.5.1 Measurements

Measurement units are often abbreviated. The following table lists some measurements and their units.

Table 1: Common measurements, their units and abbreviation

Measurement	Unit	Abbreviation
8-bit units	Byte	B
Angle	Degree	°
Angle	Radians	rad
Angle	Rotation	rot
Capacitance	Farad	F
Current	Ampere	a
Digital information	bit	b
Distance	Feet	ft
Distance	Inches	in
Distance	Meters	m
Distance	Miles	mi
Frequency	Hertz	Hz
Inductance	Henries	H
Resistance	Ohms	Ω
Time	Hour	hr
Time	seconds	s
Voltage	Volts	V

2.5.2 Powers of 10

Many of the units of measurement use a simple notation to increase the range of the number while keeping the number itself fairly simple. The following table shows many of the common abbreviations and some examples that are in this document and beyond.

Table 2: Powers of 10

Factor	Exponential Notation	Prefix	Abbreviation	Example with Units
1	10^0			a, v, s, m, Ω, b, B
1,000	10^3	kilo	k	kΩ, km, kb
1024	2^{10}	Kilo	K	KB, (maybe kb)
1,000,000	10^6	mega	m	mΩ, mb
1,048,576	2^{20}	mega	M	MB, (maybe mb)
1,000,000,000	10^9	giga	g	gb
1,073,741,824	2^{30}	Giga	G	GB
0.001	10^{-3}	milli	m	ma, ms
0.000 001	10^{-6}	micro	μ	μs, μF
0.000 000 001	10^{-9}	nano	n	ns
0.000 000 000 001	10^{-12}	pico	p	pF. ps

You may have noticed there are a few odd numbers in the table that are not powers of ten, but are rather powers of 2. The computer industry uses powers of 2 to indicates the size of disk drives and memory because that is reflective of the underlying technology. However, sometimes marketing uses the value that makes a product look better than its competition, so it is important to know the difference and to look for it in advertising claims.

A note of trivia: the root of giga is the same as giant and gigantic and should be pronounced that way. A common way of saying giga is to extend the word gig with an ah, but that belies the root of the word. The jig is up.

2.5.3 Accuracy vs. Precision

Precision is how many digits are used to express a measurement. 1 inch is less precise than 1.000 inches. Accuracy is how repeatable is a measurement. If a measurement is accurate to, let's say, a tenth of an inch, then specifying more digits does not make the measurement more accurate. 1.0 inch is somewhere between 0.95 and 1.05 inches. Saying that it is 1.000 may seem more precise, but precision should not exceed the accuracy of a measurement.

Java math libraries tend to use doubles. A Java float variable has 32 bits which is 7 decimal digits of precision. A Java double variable has 64 bits which is 16 decimal digits of precision. Since most measurements for a robot are only accurate to 3 digits, expressing a measurement in more digits does not increase its accuracy.

2.5.4 Modulo Arithmetic

In elementary school we learn some basic arithmetic operators like add, subtract, multiply and divide. Most computer languages provide one more: modulo. It just returns the remainder of a division operation. This is the fractional part of the division and is useful for several things in a robot, but especially in angular computation. If you want to find whether a number is even or add you say:

$$number \, mod \, 2$$

For positive integer numbers, it will return 1 if odd and 0 if even. If you allow negative integers it will return -1 for negative odd numbers. This may not be what you want. So if you say:

$$((number \, mod \, 2) + 1) \, mod \, 2$$

It will return 0 for odd integers and 1 for even odd integers. The inside modulo gets the number in the range -1 to 1 and the outside modulo gets the number to be 0 or 1.

Angles can be accumulated to be more than one rotation positively or negatively. This is useful when you want to know how far a wheel has turned. But if you want to know the current position of a wheel or shaft, you want to convert the accumulated angular measurement into the fractional rotation. For example if a wheel has turned 900° (2.5 rotations):

$$900° \, mod \, 360° = 180°$$

For a wheel turning -900°, you get a negative number, -180, which may not be what you want, as in:

$$-900° \, mod \, 360° = -180°$$

To make it positive you need to add one rotation or 360° to negative numbers to make them positive:

$$-180° + 360° = 180°$$

These operations can be generalized to convert an angle a to a position a:

$$Position \, a = ((a \, mod \, 360°) + 360°) \, mod \, 360°$$

The inner mod gets the number in the range -360° to +360° and the outer mod gets the number into the range 0 to 360°.

2.5.5 Binary, Base-2, and Base-16 Numbers

Most math works on numbers in base-10. Each digit is from 0 to 9. Each digit place in a number is worth 10 times the digit place value to its right. For example:

The number 879 is 9 + 7×10 + 8×100 or using exponential notation to keep track of how many places we are to the left of the decimal point:

$$879 = 9 \times 10^0 + 7 \times 10^1 + 8 \times 10^2$$

In the binary number system (also called base-2 numbers), a digit can only be a 0 or a 1. This is handy for computers because they are dumb and can't understand anything except off and on. Each digit place is worth 2 times the digit place to its right. So if we apply the same technique that we did above to a binary number:

$$1101_2 = 1\times2^0_{10} + 0\times2^1_{10} + 1\times2^2_{10} + 1\times1^3_{10}$$
$$1101_2 = 1\times1_{10} + 0\times2_{10} + 1\times4_{10} + 1\times8_{10}=13_{10}$$

Note that exponents are superscripted numbers and the base is a subscripted number.

The problem with binary numbers is that they become long and hard to remember for us mortal humans. A common representation is to use base-16 or hexadecimal notation (also called hex). The relationship between groups of four-digit binary numbers and a hex digit is shown in the following table:

Table 3: Binary, hexadecimal, decimal, and octal equivalents

Binary number (base-2)	Hexadecimal (base-16) digit	Decimal (base-10) value	Octal (base-8) value[3]
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	10
1001	9	9	11
1010	A	10	12
1011	B	11	13
1100	C	12	14
1101	D	12	15
1110	E	14	16
1111	F	15	17

An integer in Java holds a 31-bit number and a sign bit. It is common to use base-10 numbers, but you can use binary or hexadecimal numbers as well for the same value:

```
int variable = 14;      //base 10 is default number base
    variable = 0b1110; // binary or base 2
    variable = 0xe;    // hexadecimal or base 16 (lower case is OK)
    variable = 0xE;    // hexadecimal or base 16
    variable = 016;    // octal or base 8...BEWARE of numbers with leading 0!
```

3 Octal is no longer in common use, but it is supported by Java.

2.5.5.1 Binary Arithmetic

Numbers add and subtract just like decimal numbers. The problem is a number on a computer usually has a finite length. An integer in Java has 31 bits. The most significant bit is the sign bit. If your operation is close to the overflow value, $\pm 2^{32}$-1, the result may affect the sign bit.

2.5.5.2 Ones-Complement and Twos-Complement Numbers

A ones-complement number is when you just change all one-bits to zero-bits and all zero-bits to one-bits. This is what happens with the bitwise negate operator.

Computers normally use twos-complement numbers for their signed integers. When you negate a number, you just don't change the sign, but you negate all the bits including the sign and then add 1 to the resulting number. Let's say you have a 4-bit signed number just to make it easier to work with. This is really a 3-bit number and a sign bit: 0 for positive values and 1 for negative values. Let's say we start with a value of 5 which has the 4-bit value of: 0101, we want -5 so we negate all the bits to get 1010; and we add 1 to that for 1011. If we add the two values 5 and -5 together we get 0 as expected. 1 is 0001 and -1 is 1111. All the 4-bit numbers for this simple example are shown below:

Table 4: Ones- and twos-complement binary numbers

Signed decimal value	Binary number	Ones-complement value	Twos-complement value
0	0000	1111	1110 (there really is not a -0)
1	0001	1110	1111
2	0010	1101	1110
3	0011	1100	1101
4	0100	1011	1110
5	0101	1010	1011
6	0110	1001	1010
7	0111	1000	1001
-7	1001	0110	0111
-6	1010	0101	0110
-5	1011	0100	0101
-4	1100	0011	0100
-3	1101	0010	0011
-2	1110	0001	0010
-1	1111	0000	0001

2.5.5.3 Big Endian and Little Endian

Binary numbers can be stored with their low order part on either end of storage media and there is no agreement between processors as to which order to use. Big Endian stores the high order part first and Little Endian stores the low order part first. This applies to storage medium or the transmission of bits over a serial interface. High level languages take care of most of this problem by hiding the underlying processor from the user. Examine the bit order for serial interfaces carefully when trying to implement or interpret messages on that interface.

2.5.6 Boolean Algebra

A Boolean is a quantity that can either be **true** or **false**. Boolean algebra defines a set of logic operations on Boolean values:

- **AND** as in A AND B is **true** if-only-if A and B are both **true**. If either A or B is **false**, the result is **false**. This is shown in the following logic table:

  Table 5: Boolean AND operation

A	B	A AND B
false	false	false
false	true	false
true	false	false
true	true	true

- **OR** as in A OR B is **true** if-either A or B are **true**, but is **false** if both A and B are **false**. This is shown in the following logic table:

  Table 6: Boolean OR operation

A	B	A OR B
false	false	false
false	true	true
true	false	true
true	true	true

- **EXCLUSIVE OR** (XOR) as in A XOR B is **true** if A and B are different, but is **false** if A and B are the same. This is shown in the following logic table:

Table 7: Boolean Exclusive OR (XOR) operation

A	B	A XOR B
false	false	false
false	true	true
true	false	true
true	true	false

- **NOT** is a unary operator as in NOT A. If A is True the result is **false** or if A is **false** the result is True. This is shown in the following logic table:

Table 8: Boolean NOT operation

A	NOT A
false	true
true	false

Using words for the operator is a little cumbersome for equations, so various notations are used.

Table 9: Boolean operators and the symbols used to represent them

Operation	Math Symbol	Java Symbol for bitwise operations	Java Symbol for boolean expressions
AND	∧	&	&&
OR	∨	|	||
XOR	⊻ or ⊕	^	^^
NOT	¬ or placing a bar over a value symbol	!	!

The Java bitwise operations work on integer, byte, and short data types and apply to the corresponding bits of the two values in the operation. The Java boolean expression operators are usually used in conditional statements to avoid nested “if” statements by testing two or more expressions in the same statement.

Using the Java logic notation (since it is simpler to type, and it is what is used by FRC robots)

A & !A = **true** (always)

A | !A = **false** (always)

2.5.6.1 De Morgan's Law

De Morgan's Law allow the transformation of an OR statement into an AND statement using grouping and negation. This is useful in logic circuit design.

!(A | B) = !A & !B

!(A & B) = !A | !B

2.5.7 Distance Measurements

Most of the math for a robot has to do with knowing the robot's position and orientation. This may be how far a wheel has turned, the position of an arm or the speed of a shooter. Distances in the field are measured in two units: inches when you read field measurement and CAD drawings and meters when using the code libraries.

$$1\,meter = 39.37\,inches = 3.3\,feet$$

The distance measurement conversions are shown in the following table.

Table 10: Distance measurement conversions

From	To	Conversion
Feet	Inches	12*x*
Feet	Meter	*x*/3.3
Inches	Feet	*x*/12
Inches	Meters	*x*/39.37008
Meters	Feet	*3.3x*
Meters	Inches	39.37008*x*

Keep in mind that the WPILib APIs change from one season to the next, so unit conversions must typically ripple through any carried over code base to comply with the new interface requirements.

Distance is a major factor during the autonomous period, but during the teleop period, movements are specified in velocities rather than distance. You use a joystick to control the velocity of the robot. Velocities are measured in meters per second (mps or m/s). When you get into the controllers you will find accelerations or changes in velocity measured in meters per second per second (m/s^2). Advanced controllers worry about the smoothness of the control and uses a measurement of the changes of acceleration or jerk as meters per second per second per second (m/s^3).

Table 11: Distance and change of distance measurements

Measurement	Name	Example units
distance	distance	meters (m), inches (in), kilometers (km), miles (mi)
change of distance over time	velocity or speed	m/s, km/hr, mi/hr
change in velocity over time	acceleration	m/s^2
change in acceleration over time	jerk	m/s^3

2.5.8 Angular Measurement

Angular measurement is a bit more difficult. We have been trained to use degrees, and we intuitively know what 360, 180, 90 and 45 degrees mean. So the human interface usually uses degrees to express an angular measurement. Math routines in software like angles expressed in radians, where a full rotation has 2 radians. Shaft encoders report angular measurement as ticks where the resolution of a tick is dependent on the device, an analog voltage, or a number between 0 and 1. To simplify the confusion, some Java software libraries use the notion of rotations. It is convenient for calculations, and it is easy to convert what every angle measurement you have to other angular measurement systems as required.

As humans, we normally express angles in terms of degrees. We know that there are 360 degrees in a circle and what a 45° angle looks like. Java methods for dealing with angles typically use radians. This is common for many mathematical functions and their software counterparts. Lately the WPILib has been using “rotations” as an angular measurement. The conversion between different units is fairly simple, so starting with the base equality:

$$1 rotation = 360° = 2\pi \ radians$$

The angular conversions are shown in the following table.

Table 12: Angular unit conversions

From	To	Conversion
Rotations	Degrees	$360x$
Rotations	Radians	$2\pi x$
Degrees	Rotations	x / 360
Degrees	Radians	(x / 360) * 2 π = πx/180
Radians	Rotation	x/(2π)
Radians	Degrees	*360* x/(2π) = 180 x/π

In addition to these some shaft encoders work their own angular measurement. Some use 4096 ticks per rotation, so you would have to convert their measurement to rotations (by dividing by 4096) and then converted to the units desired. Your vendor(s) may use different angular measurements.

Many APIs provide decorator methods for converting units, so you should not risk hard coding the conversions and making a typographical error.

$$1\,rotation = 2 \cdot \pi\, radians = 360\,° = so\ many\ ticks\ of\ a\ shaft\ encoder$$

This paper will use degrees as the measure of angles even though the actual robot code will use something else. This is just to make the paper easier to understand.

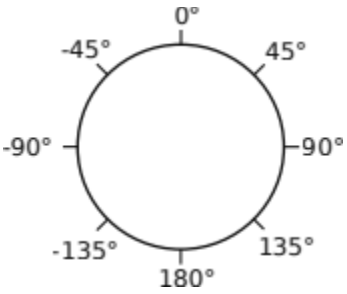


Figure 45: Angular measurement of a relative heading.

The expression of an angle may be expressed as an absolute number which for degrees would be between 0° and 360° or a relative measurement which would be a number between -180° and +180° degrees. These both have a place in robotics. Relative angles are used when you want to use an angle relative to the current orientation or heading. This is like turn right 45°, or turn left 35° from the current orientation of 0°. Positive angles are clockwise from the starting orientation and negative angles are counterclockwise.

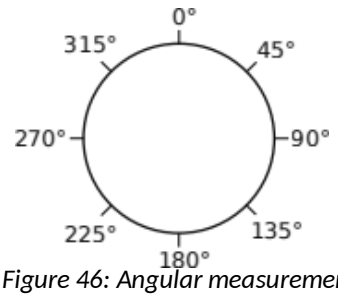


Figure 46: Angular measurement of a magnetic heading.

You may have used a magnetic compass. It expresses an absolute angle with north being 0° and the angles increase clockwise with east being 90°, south 180° and west 270°. In Cartesian coordinates as you will learn in the next section measures absolute angles a little differently. It uses 0° for the positive x-axis and then rotates the angle counterclockwise 90° to the positive y-axis. It is important to know where the zero angle is for an absolute angular measurement and whether angles increase positively clockwise or counterclockwise. Robots use a variety of these in different frames of reference.

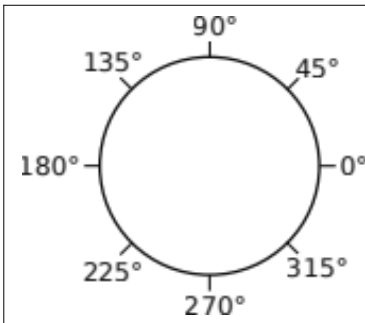


Figure 47: Angular measurement of a mathematic heading.

Some measurements, like knowing how far a robot has moved is dependent on the number of rotations that a wheel has turned. Here it is necessarily to accumulate all the relative angle changes to get the total number of rotations (whole and fractional). Orientation does not want to unwind the accumulated turns, so it “rolls over” when more than 360° have been accumulated. The same is true of relative angles greater than 180° or less than -180°. This causes an anomaly at the transition point: 359.999… and 0 for absolute angles and -179.99… and 180 for relative angles. The anomaly must be accounted for when a comparison is made between two angles which cross the anomaly point. This can be done by adding the equivalent of 360° to one or both angles to force the calculation to not cross the anomaly.

2.5.8.1 Calculating the Difference of Two Relative Angles

If you take the difference between two relative angles and the result is positive, that indicates that the measurement is in the opposite rotation reference. So the angular difference between two angles must be negated to get the difference to correspond to the reference direction.

The difference of $50°$ and $30°$ is $20°$ clockwise ($-20°$):

$$-(50°-30°)=-20°$$

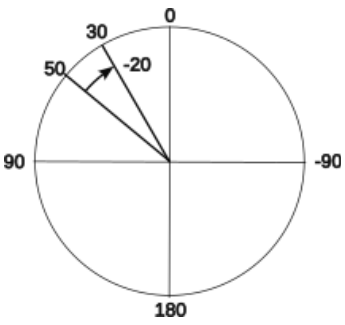


Figure 48: Difference of 50 and 30

The difference of $30°$ and $50°$ is $20°$ counterclockwise:

$$-(30°-50°)=20°$$

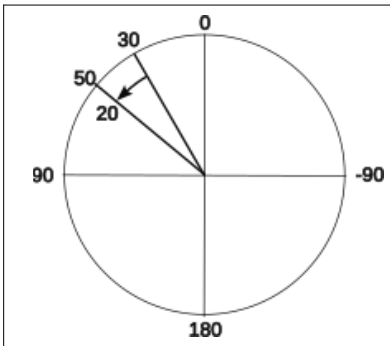


Figure 49: Difference of 30 and 50

Now test the boundary condition around zero. The difference between relative $10°$ and $-10°$ is $20°$ clockwise ($-20°$):

$$-(10°-(-10°))=-20°$$

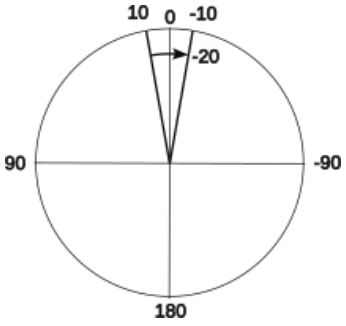


Figure 50: Difference of 10 and -10

The difference between relative $-10°$ and $10°$ is $20°$ counterclockwise:

$$-(-10°-10°)=20°$$

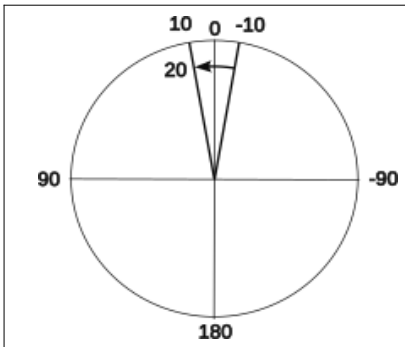


Figure 51: Difference of -10 and 10

Test the other boundary condition around the 180° discontinuity. The difference between relative 170° and -170° is 20° counterclockwise, but:

$$-(170° - (-170°)) = -340°$$

Since the absolute value is greater than 180°, it must be increased or decreased by one rotation or 360°:

$$-340° + 360° = 20°$$

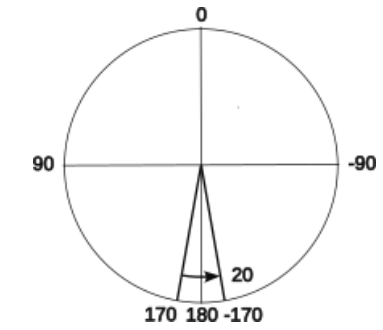


Figure 52: Difference of 170 and-170

The difference between -170° and +170° is 20° counterclockwise (-20), but:

$$-(-170° - 170°) = 340°$$

Since the absolute value of 340° is greater than 180°, add or subtract one revolution or 360° to get it in the right range:

$$340° - 360° = -20°$$

The formula for taking the difference of two relative angles can be generalized using modulo math as:

$$Difference = (-(a - b) + 360°) \bmod 360° - 180°$$

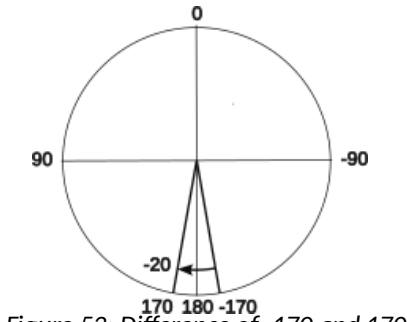


Figure 53: Difference of -170 and 170

Adding 360° to the difference forces it to be positive.

"mod 360°" means divide by 360° and use the remainder. This puts the number in the range 0° to 360°

Subtracting 180° changes the range 0° to 360° to the range -180° to 180°.

To find the smallest angle between two angles, you have to take both differences of the two angles and take the result with the smallest absolute value.

2.5.9 Coordinate Systems

A coordinate system is way of locating an object in space numerically. Robots use many different coordinate systems, sometimes at the same time. Some coordinate systems find locations with distances. Others do it with angles. Still others do it with some combination of both. All coordinate systems have:

- An origin or a point at the center of the coordinate system. Each dimension or coordinate of a coordinate system has a zero or origin.
- An orientation used for measurements of that coordinate are fixed by convention.
- Some relationship between the coordinates. Most coordinate systems use coordinates that are 90°, or orthogonal, to each other.
- A convention for the measurement of each coordinate.

2.5.9.1 Cartesian Coordinate System

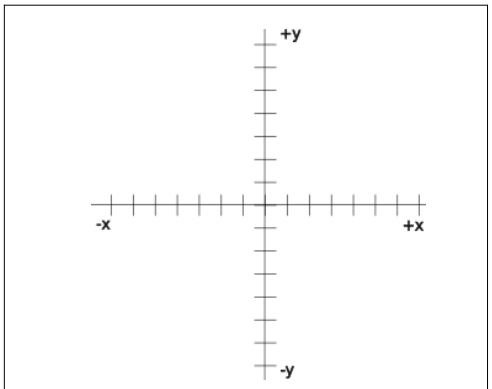


Figure 54: Cartesian coordinate system

One basic coordinate system is the Cartesian coordinate system. A point on a plane can be located with its x- and y-coordinates. A point can be located in three-dimensional space by adding a z-coordinate to express the height of the point above (or below) the reference plane.

Cartesian coordinate systems are used in robotics to locate a robot on a field or a subsystem on a robot. There are several nested reference frames to allow locating a point on a robot to a specific point on the field based on the robot's location on the field.

Positions within a robot are assigned on a Cartesian coordinate system based on the physical center of the robot and are measured in inches from the center point. The front and back of the robot are a little ambiguous, and must be defined by agreement. This agreement must stick for the life of the robot to avoid confusion as it is the basis for many calculations involving components of the robot. Note that the front of the robot is 0°, so it is at the positive end of the x-axis and y-values increase toward the left side.

2.5.9.2 Polar Coordinate System

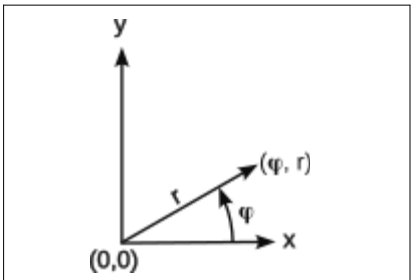


Figure 55: Polar coordinate system

Polar coordinates allow a point to be specified on a two-dimensional plane as an angle and a radial distance. Converting between Cartesian and polar coordinates is straight forward:

$$r = \sqrt{x^2 + y^2}$$

$$\boldsymbol{\varphi} = atan\left(\frac{y}{x}\right)$$ (adjusted for the particular quadrant)

$$x = r\cos(\boldsymbol{\varphi})$$

$$y = r\sin(\boldsymbol{\varphi})$$

2.5.9.3 Spherical Coordinate System

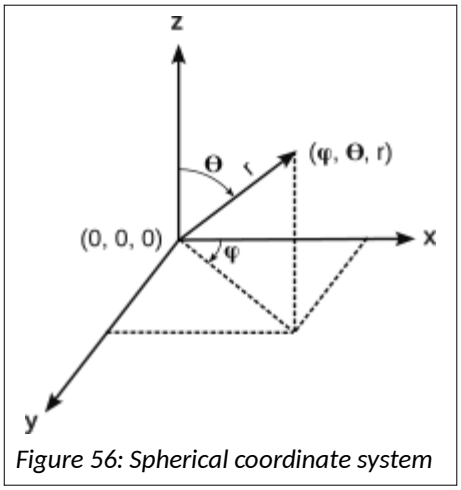


Figure 56: Spherical coordinate system

Another way of expressing a position in three-dimensional space is the spherical coordinate system. It starts with an origin at the center of an imaginary sphere. A position is located by an angle φ in the x-y plane and a second angle Θ from the axis perpendicular to the x-y plane (i.e., the z-axis). The radial distance r fixes a point on the line defined by the two angles.

This coordinate system is similar used to locate places on Earth. The first angle is longitude and the second is latitude. 0° longitude goes through Greenwich, England home of the British Royal Navy that originally set this up. Positive longitude angles proceed to the east and negative angles proceed to the west. 180° is the International date line. 0° latitude is at the equator, +90° is at the

north pole and -90° is at the south pole. For earth coordinates an altitude forms the r-component, but it is relative to the agreed upon radius of sea level.

Spherical coordinates are useful for the cameras on robots. A camera may be pointed nearly anywhere on the field. Normally a camera views a world at fixed angles which can be expressed as polar coordinates relative to the robot frame of reference. The pitch angle, used for the Θ angle, has its zero point on the horizontal plane.

$$r = \sqrt{x^2+y^2+z^2}$$

$$\boldsymbol{\varphi} = atan\left(\frac{y}{x}\right) \text{ (adjusted for the particular quadrant)}$$

$$\Theta = atan\left(\frac{\sqrt{x^2+y^2}}{z}\right)$$

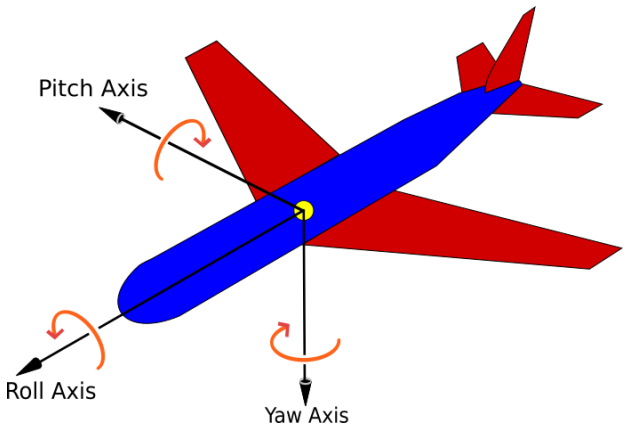
$$x = r\sin(\Theta)\cos(\boldsymbol{\varphi}) \quad , \quad y = r\sin(\Theta)\sin(\boldsymbol{\varphi}) \quad , \quad z = r\cos(\Theta)$$

2.5.9.4 Vehicle Orientation Reference System

Aircraft and boats use three angles to express its orientation, usually in relation to gravitational planes or inertial reference planes:

- **Yaw** is the angle that the vehicle is pointing on a horizontal plane.
- **Pitch** is the front-to-back angle of the vehicle relative to a horizontal plane. Positive angles are up and negative down.
- **Roll** is the side-to-side angle relative to a horizontal plane. Positive angles are to the right.

This is a variation of a spherical coordinate system that uses angles to express an orientation relative to the vehicle frame of reference, but providing no location information.



(From Wikipedia ros-robotics.blogspot.com*)*

Figure 57: Vehicle orientation reference system

The vehicle orientation angles are also a variation of Euler angles. The difference is that Euler angles measure the pitch from the positive z-axis with a range of 0 to 180° rather from the x-y plane with a range of 0 to ±90°.

2.5.10 Frames of Reference

A frame of reference basically is a coordinate system like the familiar Cartesian coordinate system. In FRC robotics several frames of reference are used, some with the same origin, some with different origins, some with different orientations, and some have different rotations. These allow calculations be based on different perspectives like a field perspective, a driver's perspective or the robot's perspective. Each is useful for certain calculations and translating between the different perspectives is essential to efficient robot control. Moving to a fixed point on the field makes it easy to do autonomous movements. Knowing where the wheels of the robot allow calculations to estimate the robot position based on wheel movements. Telling a robot to move right with respect to the field makes driving the robot similar to an arcade game.

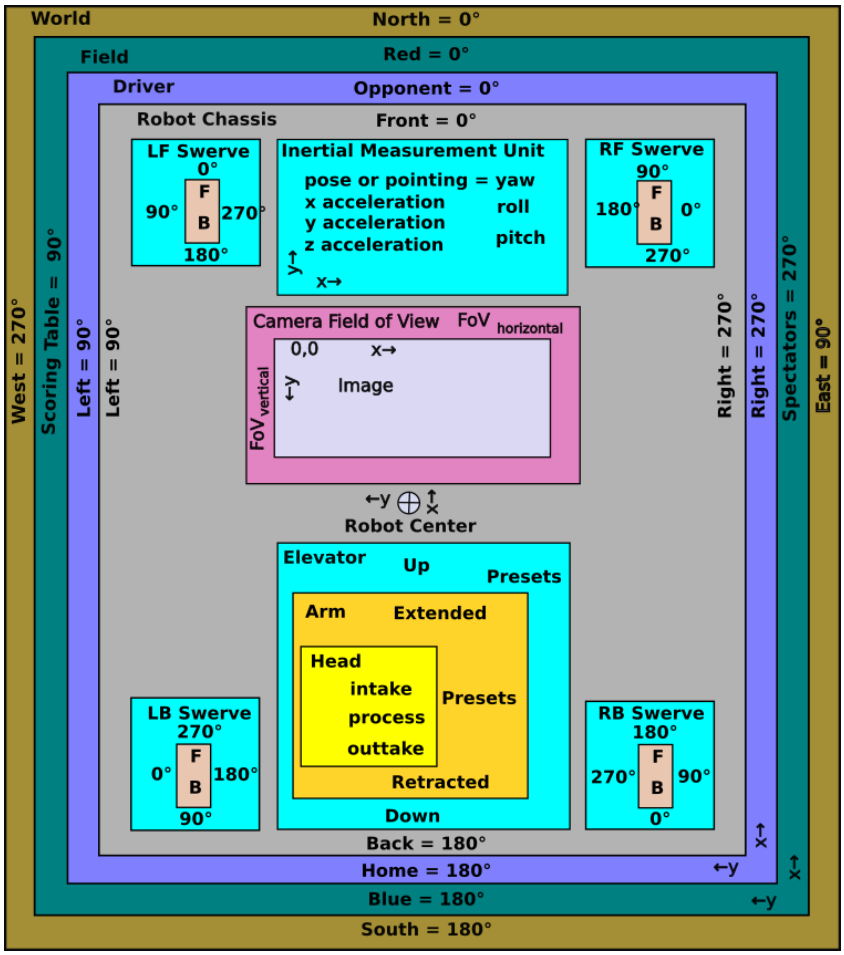


Figure 58: Robot frames of reference

A frame of reference defines how to specify a point on the field in terms of a set of coordinates or a set of angles. A given point on the field can be described by several frames of reference. It is possible to translate between these frames as some frames are better for some calculations than others. A camera has a fixed location on a robot, but that robot can be just about anywhere on the field, and it can be oriented in any angle. This means that the camera can be located on the field and its orientation be determined based on where the robot is located and where it is pointing.

The robot frame of reference is allowed to move freely on the field, so as it moves so does it location in the world, field, and driver's frame of reference. The wheels of the robot remain in fixed positions on the robot (normally), so their coordinates in the robot frame of reference do not change even though the robot is moves down the field and turned in different directions. As the robot moves the distances from the robot to field elements, like a scoring location, changes. The robot control software may need to know how far away the robot is from the field element and may need to know at what angle is the field element with respect to the robot. To answer questions like this on must translate from one frame of reference to another.

2.5.10.1 World Frame of Reference

The world frame of reference is how one would normally locate a position on earth. It usually uses a coordinate system like latitude or longitude and directions north, south, east, and west. The Global Positioning System (GPS) returns your position in a world frame of reference. GPS measurement accuracy in cellphones has increased in the last twenty years by using a variety of techniques involving math operations on the received signals and have boosted the accuracy from 30 meters down to about 2 meters. Unfortunately it is device and environment dependent and still is not accurate enough for FRC robots applications, which need an accuracy to a few inches. Some high accuracy applications like farming and construction get additional accuracy, but they use an auxiliary transmission of an error correction signal. Neither GPS nor the correction signal is not allowed by current FRC rules.

Table 13: World frame of reference attributes

Reference	Description	Robot usage
0 angle	True north	
Angles increase	Clockwise like a magnetic compass	
Origin	Corner of red alliance and scoring table	
x increases	To the east	
y increases	To the north	

2.5.10.2 Field Frame of Reference

The field frame of reference is used to find a robot's location on the field. It is like the Cartesian coordinate system where x and y values are all positive. The origin is in the corner of the blue alliance and the scoring tables. Positive y values move from the blue side to the red side. Positive x values move from the scoring tables to the audience side. The field CAD drawings show the locations of elements in terms of inches. The WPILib uses locations based on the metric system in meters.

Each robot keeps track of its position through kinematics (the movement of individual drive wheels) and odometry (the movement of the robot). Robots may also use vision systems to resolve their location based on the known locations of April tags.

Locations of field elements are positioned in this frame of reference including April Tags, scoring locations, game piece loading stations, and other obstacles.

ID	X	Y	Z	Rotation
1	593.68	9.68	53.38	120°
2	637.21	34.79	53.38	120°
3	652.73	196.17	57.13	180°
4	652.73	218.42	57.13	180°
5	578.77	323.00	53.38	270°
6	72.50	323.00	53.38	270°
7	-1.50	218.42	57.13	0°
8	-1.50	196.17	53.13	0°
9	14.02	34.79	53.38	60°
10	57.54	9.68	53.38	60°
11	468.69	146.19	52.00	300°
12	468.69	177.10	52.00	60°
13	441.74	161.62	52.00	180°
14	209.48	161.62	52.00	0°
15	182.73	177.10	52.00	120°
16	182.73	146.19	52.00	240°

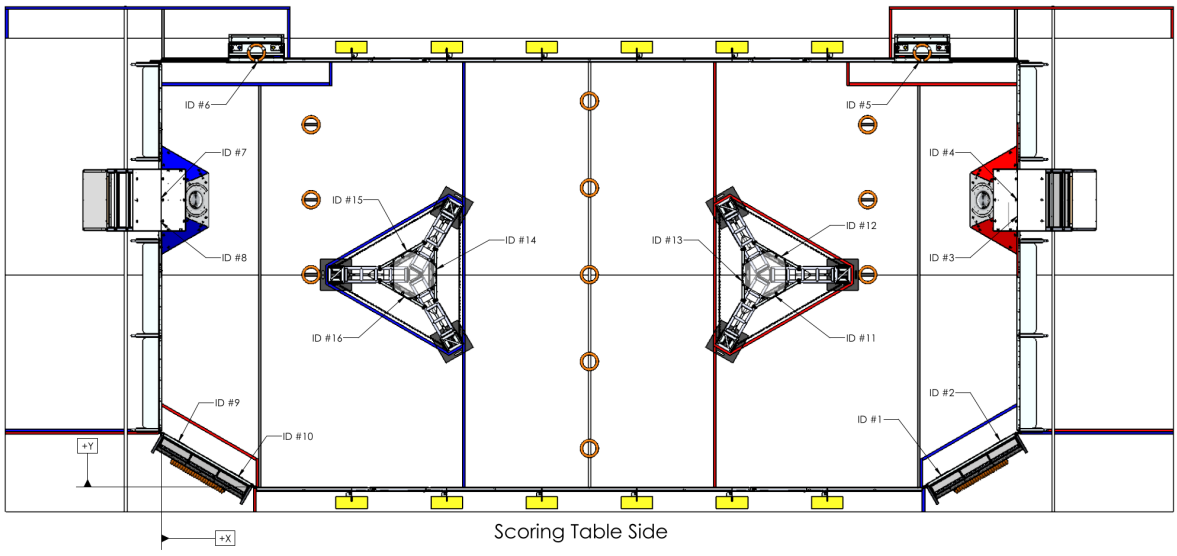
Table 14: 2024 April tag poses (in inches)

Table 15: Field frame of reference attributes

Reference	Description	Robot usage
0 angle	Toward red alliance along scoring table wall	Kinematics, odometry and vision systems.
Angles increase	Counterclockwise	Kinematics, odometry and vision systems.
Origin	Corner of blue alliance and scoring table	Kinematics, odometry and vision systems.
x increases	From scoring table to audience	Kinematics, odometry and vision systems.
y increases	From blue to red alliance	Kinematics, odometry and vision systems.

2.5.10.3 Driver Frame of Reference

The driver just sees the end of the field and drives to what he or she sees. The home end is toward the driver and the opponent alliance is away from the driver. The directions home, opponent, right, and left are from the driver's perspective and the joystick controls move the robot in those directions. Software controls that pick out the home alliance field elements are changed to reflect the driver's current alliance, so that "go to load station" sends the robot to the current alliance load station. If there are more than one, it may be something like the "go to the right load station" that is the right load station from the driver's perspective. The idea is to keep things intuitive for the driver.



(figure from 2024 Cresendo game manual.)

Figure 59: Field layout for the 2024 Crescendo game.

Table 16: Driver frame of reference attributes

Reference	Description	Robot usage
0 angle	Toward opponent alliance along scoring table wall	Joystick mapping
Angles increase	Counterclockwise	Joystick turning and spin controls
Origin	Corner of home alliance and scoring table	Joystick turning and spin controls
x increases	From home to opposition alliance	Joystick mapping
y increases	From scoring table to audience	Joystick mapping

2.5.10.4 Global Frame of Reference

The WPILib refers to a “Global Frame of Reference.” In this context they are referring either to the Field Frame of Reference or to the Driver’s Frame of Reference dependent on the choice of what the team is using as their main frame of reference.

2.5.10.5 Robot Frame of Reference

The origin for the robot frame of reference is normally the center of the robot. The “front” of the robot is a somewhat arbitrary decision, but it is generally in the scoring orientation of the robot. Whatever you pick communicate the choice and stick with it. This determines the direction that the robot moves when issued a “forward” command.

There may be occasions when a driver wants to drive from the robot frame of reference and may use vision to gain the perspective of the robot.

There are various components and subsystems mounted on the robot. These subsystems are located on the robot frame of reference, so that their relative location on the robot can be translated to a field or driver frame of reference. These locations are used to translate drive wheel movements into robot chassis movements, and vision subsystem position estimates can be translated into robot position estimates. The locations of other subsystems may be required for specific translations and transformations.

Table 17: Robot frame of reference attributes

Reference	Description	Robot usage
0 angle	Toward “front” of robot based on the “front” as chosen by the team	
Angles increase	Counterclockwise	
Origin	Center of the robot at the floor level	
x increases	Toward front of robot	Kinematics, odometry
y increases	Toward left side of robot	
z increases	From the floor to the top of the robot	Vision subsystems

2.5.10.6 Turret Frame of Reference

Some robots have a turret that rotates independently of the chassis. This allows the turret to track the target and drive train motion is not wasted to align to the target. The angle of the turret to the chassis is call azimuth. The turret may include many subsystems including elevators, arms, shooters, etc.; and it may include cameras for vision processing of the target. This additional frame of reference would have to be factored into the calculations performed by the robot involving subsystems mounted to the turret.

2.5.10.7 Drive Wheels

The location of the drive wheels with respect to the robot center is used by kinematics and odometry to estimate the current position of the robot chassis based on the incremental movements of individual wheels.

Additionally, swerve drive modules are usually mounted in a corner of a robot and have a forward direction that is out of alignment of the other three modules. The modules need to be initialized so that all drive modules agree on a common forward direction, both in module steering orientation and drive motor direction. This direction should be aligned to the robot axis so that the robot drives straight and not skewed.

2.5.10.8 Vision Subsystem Camera(s)

The location of cameras with respect to the robot center is used with vision subsystems. This location is used to updating the robot position of the field from measurements by the vision subsystem. The x-, y- and z-coordinates of the camera with respect to the robot must be known as well as the orientation of the camera. Vision sub-systems convert image coordinates into an angular measurement. This measurement is transformed to the robot center.

Table 18: Camera frame of reference attributes

Reference	Description	Robot usage
0 angle	Toward “right side” of robot based on the “front” as chosen by the team	
Angles increase	Counterclockwise	
Origin	Upper left corner of image	
x increases	Toward right side of image	Vision subsystems
y increases	Toward bottom of image	Vision subsystems.
Field of View horizontal	Size of field of view in terms of angular degrees and number of pixel	Vision subsystems
Field of View vertical	Size of field of view in terms of angular degrees and number of pixel	Vision subsystems
Yaw	Angle of the camera in the robot z-axis relative to the front-back (x) axis of the robot	Vision subsystems
Pitch	Angle of the camera on the robot y-axis relative to horizontal (the robot x-y-plane and usually the field plane).	Vision subsystems
Roll	Angle of the camera in the robot x-axis, normally = 0.	Vision subsystems

2.5.10.9 Shaft Encoder Frame of Reference

If a shaft encoder is on a motor and those rotations are changed by the gearing, the gear ratio needs to be applied to the calculations for the target shaft rotation. Let’s say you have a motor with a shaft encoder that drives an arm through a 10:1 reducing gear. If you want to turn the arm 45°, you need to turn the motor 450°. That difference should be applied consistently through your commands and human interface to reflect the arm angle and not the indicated shaft encoder angle. Also keep in mind that the absolute angle with respect to vertical may be subject to the addition of several angles, e.g., shoulder angle, elbow angle, and wrist angle.

2.5.10.10 Inertial Measurement Unit

The inertial measurement unit (IMU) of the robot should be close to the x-y center of the robot. If it is offset, its measurements need be translated to reflect what they would be at the robot center. See vector addition in a following section. As mentioned in section 2.4.18 Gyroscope and the Inertial Measurement Unit, the IMU provides a roll, pitch, and yaw angle and the acceleration in the x-, y- and z- axes.If the Inertial Measurement Unit is mounted at the center of the robot, its measurements are the same as the robot frame of reference, otherwise some translation is needed to use the measurements accurately.

2.5.10.11 Robot Mechanical Subsystems

The location of arms, elevators, climbers, and other mechanical subsystems is not necessary for the operation of the robot beyond their state (up, down, extended, retracted, position, etc.). Location of these components becomes critical if other subsystems like vision subsystem camera(s) are mounted on a moving component.

Advanced teams may keep track of the robot subsystems to monitor and control the behaviors of the robot as the center of gravity is moved. This information may be useful in post match playback of recorded logging data.

2.5.10.12 Mirroring of Field Coordinates for Drivers Coordinates

In recent competitions the layout of the field from the red alliance perspective is a mirror image of the blue alliance perspective. The math for mirroring blue alliance coordinates to red alliance coordinates is:

$$x_{red} = x_{blue}$$

$$y_{red} = fieldLength - y_{blue}$$

$$angle_{red} = 180° - angle_{blue}$$

This has an impact to automated sequences during both the autonomous period and the teleop period. The path planning routes are supposed to be reflected about the center, so that a single path can be used on either the red or the blue alliance side of the field. [We did not have much luck doing this in 2024.]

The odometry subsystem keeps track of where the robot is during a competition. It keeps track of how many turns a wheel turns and in what direction. Specifically it looks at each wheel angle with respect to the field and adds in the average wheel velocity for a period times the length of the period. All four wheels are accounted for including their position with respect to the robot center (as recorded in the robot kinematics configuration). These incremental movements are added as a vector sum to determine the x-y motion of the robot and the robot's angular rotation. Luckily this is all done under the covers of the robot library. These measurements are done with as small of an increment as possible to increase its accuracy. Even so this is susceptible to errors introduced by spinning wheels, flying robots and collisions. Vision processing of April Tags is used by some teams to reset the odometry to keep it accurate.

2.5.10.13 Pose

The WPILib uses a construct called Pose class to express the position of the robot (and other objects) on the field from the field frame of reference. The common robot positiom uses a Pose2d class to define objects consisting of three attributes:

- x-coordinate expressed in meters (although field drawing use inches)
- y-coordinate express in meters.
- Facing angle expressed as rotations (but easily converted to degrees or radians).

This if fine for things that are on the floor of the field, but a lot ot things like April Tags, cameras, shooters, and scoring targets are off the floor. To express their location WPILib uses the Pose3d object which consists of six parts:

- x-coordinate expressed in meters.

- y-coordinate express in meters.
- z-coordinate expressed in meters.
- Yaw or facing angle expressed as rotations.
- Pitch or angle about a horizontal axis perpendicular to the facing expressed as rotations.
- Roll or angle about a horizontal axis parallel to the facing angle expressed as rotations.

2.5.11 Geometry

Geometry is one of those things that after a while you take for granted and do not know that you are using it. The geometric theorems may be useful at times to solve some angle problems. In general most robot math does not require this. Some of the most important things to remember is just some basic rules like:

- The sum of the angles that form a straight line total 180 degrees.
- The sum of the angles in a triangle is 180 degrees.
- The Pythagorean theorem for finding the sides of a right angle triangle: $a^2+b^2=c^2$

  A simple proof of the theorem (not really necessary for robotics, but fun):

  1. Construct a square with sides a + b
  2. Connect the a-sides with the b-sides to form four triangles a-b-c and opposite angles A, B, C.
  3. The angles of the triangles are A, B, and C, so:
     $A+B+C=180°$
  4. Since C is both a corner of the original square and the corner of the triangle a-b-c:
     $C=90°$
  5. This means that since:
     $A+B+C=180°$ and
     $C=90$ that:
     $A+B=90$
  6. Since a straight line has 180°,
     $A+B+D=180°$ and
     $A+B=90°$, so:
     $D=90°=C$ and the inner area c-c-c-c is a square
  7. The area of the inner square is just:
     $Area\ inner\ square=c\cdot c=c^2$

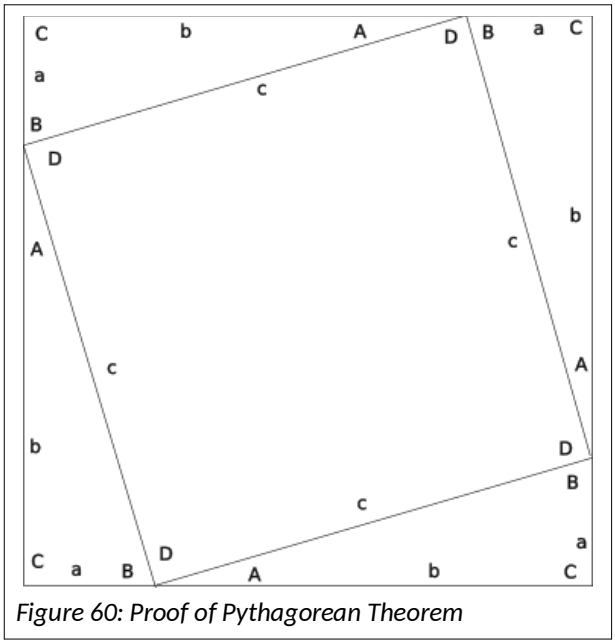


Figure 60: Proof of Pythagorean Theorem

8. The area of each triangle is:
   $$Area\ triangle = (base \cdot height)/2 = (a \cdot b)/2$$
9. The relationship between the area of the outer square and its constituent triangles and inner square is:
   $$Area\ outer\ square = 4 \cdot ((ab)/2) + c^2 = 2ab + c^2$$
10. Just squaring the side to find the area of the outer square and reducing:
    $$Area\ outer\ square = (a+b)(a+b) = (a+b)^2$$
    $$Area\ outer\ square = a(a+b) + b(a+b)$$
    $$Area\ outer\ square = a^2 + ab + ab + b^2$$
    $$Area\ outer\ square = a^2 + 2ab + b^2$$
11. Combining 9 and 10 above the area of the outer square is:
    $$Area\ outer\ square = a^2 + 2ab + b^2 = 2ab + c^2$$
12. Subtracting 2ab from both sides of the equation leaves:
    $$a^2 + b^2 = c^2$$
    $$Q.E.D.$$

- The Pythagorean theorem can be rewritten as follows to solve for the length of the (c) or one of the short sides (a):
  $$c = \sqrt{a^2 + b^2}$$
  $$a = \sqrt{c^2 - b^2}$$

2.5.12 Trigonometry

Trigonometry is used quite a bit in kinematics and odometry, swerve drive subsystems and in April Tags, so a short discussion is warranted. Trigonometry, or trig for short, allows finding the length of sides and angles of triangles when only three are known (i.e., three sides, three angles, an angle and two sides, or two angles and a side).

The fundamental observation of trigonometry is that the ratio between the lengths of the sides of triangle with the same angles does not change as the lengths of the sides are changed. If you double the length of the sides of a triangle the ratios between the sides remain the same.

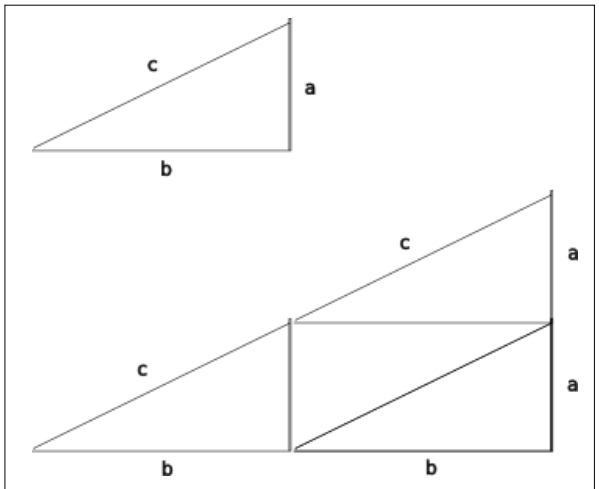


Figure 61: Intuitive Trig Proof

$$a^2 + b^2 = c^2$$

$$3^2 + 4^2 = 5^2$$

$$area = \frac{ab}{2} = 3 \cdot 4/2 = 12/2 = 6$$

Double each side of the triangle, and the calculations become:

$$a^2+b^2=c^2$$

$$(2\cdot3)^2+(2\cdot4)^2=6^2+8^2=36+64=100=10^2=(2\cdot5)^2$$

$$area=\frac{2\cdot a\cdot 2\cdot b}{2}=\frac{2\cdot3\cdot2\cdot4}{2}=\frac{6\cdot8}{2}=\frac{48}{2}=24=2^2\cdot6$$

Trigonometry tables are made for right angle triangles. Geometry can be used to convert any triangle into a series of right angle triangles. Trigonometry refers to an angle as being adjacent or opposite to an angle (and it ignores the right angle), so a sine (abbreviated sin) is the ratio between the opposite side and the long side or hypotenuse. Angles are expressed as upper case letters, and angles are expressed with lower case letters. An angle is associated the side opposite it (not touching it) and they use the same letter.

$$\sin A=\frac{opposite}{hypotenuse}=\frac{a}{c}$$

Cosine (abbreviated cos) uses the ratio between the adjacent side and the hypotenuse:

$$\cos A=\frac{adjacent}{hypotenuse}=\frac{b}{c}$$

Tangent (abbreviated tan) uses the ratio between the opposite and adjacent sides:

$$\tan A=\frac{opposite}{adjacent}=\frac{a}{b}$$

Cotangent (abbreviated cot) uses the ratio between the adjacent and opposite sides:

$$\cot A=\frac{adjacent}{opposite}=\frac{b}{a}$$

When you know the angles, but not the angle there is an inverse set of functions. arc sine (abbreviated asin or $\sin^{-1}$) is the angle derived from the ratio between the opposite side and the long side or hypotenuse:

$$asin\left(\frac{opposite}{hypotenuse}\right)=asin\left(\frac{a}{c}\right)=A$$

Arc cosine (abbreviated acos or $\cos^{-1}$) uses the ratio between the adjacent side and the hypotenuse to find the angle:

$$acos\left(\frac{adjacent}{hypotenuse}\right)=acos\left(\frac{b}{c}\right)=A$$

Arc tangent (abbreviated atan or $\tan^{-1}$) uses the ratio between the opposite and adjacent sides to find the angle:

$$atan\left(\frac{opposite}{adjacent}\right)=atan\left(\frac{a}{b}\right)=A$$

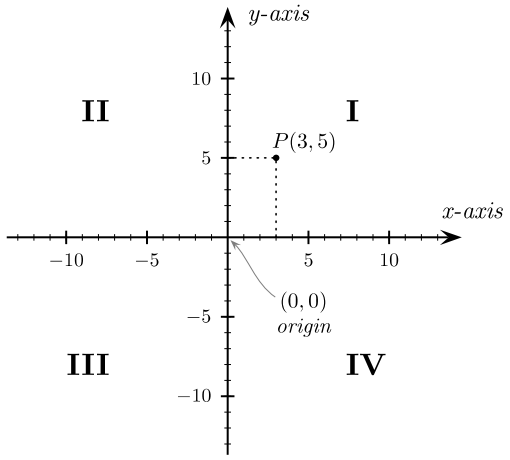
There isn't really an arc cotangent as you can just work with the other angle and an arc tangent function to find both angles.

2.5.13 More on Arc Tangents and Quadrants

The arc tangent function really only works in the range -90° to +90°. To get it work in the range -180° to +180° or the range 0° to 360°, software must account for the quadrant where the endpoint lies. Back to the Cartesian coordinates, the table below lists the four quadrants and their characteristics:

Table 19: Quadrant and arc tangent angle

Quadrant	x-value	y-value	angle
I	positive	positive	0 to 90°
II	negative	positive	90 to 180°
III	negative	negative	180 to 270° or -90 to -180°
IV	positive	negative	270 to 360° or -90° to 0



(figure from Wikipedia)

Figure 62: Cartesian coordinates and quadrants

Use the quadrant to adjust the return arc tangent value to the angle appropriate for that quadrant. Java provides an **atan2** function in its math module to do that.

2.6 Some Advanced Math Topics

2.6.1 Vectors

Vectors simplify the discussion of how forces, motions, and observations in robots are calculated. Vectors can be used in translating a desired movement of the robot to the motions required of its drive train, whether it is a tank drive or a swerve drive. Vectors can also be used in translating or moving within a frame of reference or transforming between frames of reference.

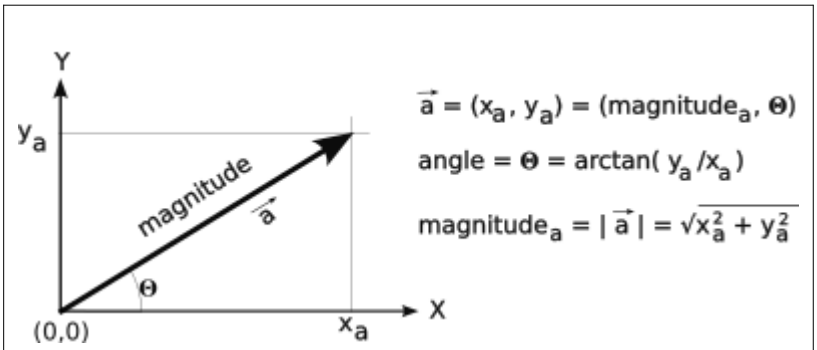


Figure 63: Basic vector

In its simplest form a vector is a direction and a magnitude. It could be something list a 30 MPH wind from the northwest. The magnitude in this case is 30 MPH and the direction is from the northwest (or toward the southeast). The magnitude may be a force, distance, velocity, or acceleration. For most robot application the direction is a two-dimensional direction expressed as an angle to a frame of reference or the x- and y-components of the direction in the frame of reference. (The vectors for vision processing use three dimensions and have to specify the direction as either the x-, y-, and z- components or the roll, pitch, and yaw angles in the frame of reference. We will stick to two-dimensions for now but everything applies in the three-dimensional world as well.) For robot movements in the WPILib, the magnitude is expressed in units of meters per second. Rotational speeds are expressed in radians per second or rotations per second. Radians are used by the Java (and other programming languages) math routines.

$$2 \cdot \pi \cdot radians = 360° = 1 rotation$$

$$2 \cdot \pi \cdot radians = 360° = 1 rotation$$

A vector can be expressed as its angle and magnitude or as its x and y components as shown in the figure above.

$$\vec{a} = |\vec{a}| \angle \Theta = (x_a, y_a) \text{ where } \vec{a} \text{ is the shorthand notation for vector a}$$

where:

$\vec{a}$ *is the shorthand notation for vector a*

$|\vec{a}|$ *is the notation for the magnitude of vector a*

$\angle \Theta$ *is the notation for the angle* Θ *of vector a*

(x_a, y_a) *is the notation for the x , y components of vector a*

A third way of expressing a vector is as a matrix, as in: $\vec{a} = \begin{bmatrix} x \\ y \end{bmatrix}$

2.6.1.1 Unit Vectors

Unit vectors are another way to specify a point in a coordinate system. A unit vector has a length of 1. Each unit vector used to specify a space is orthogonal (perpendicular, but in any number of dimensions) to other unit vectors. A two-dimensional space uses two unit vectors: $\hat{x}$, a unit vector in the x-direction, and $\hat{y}$, a unit vector in the y-direction. A three-dimensional space uses these units vectors plus a third: $\hat{z}$, a unit vector in the z-direction.

We have seen two notations to describe a vector using the x and y components.

$$\vec{a} = (x_a, y_a) = \begin{bmatrix} x_a \\ y_a \end{bmatrix}$$

This latter notation is really a shorthand for $\vec{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix} [\hat{x}\ \hat{y}]$.

Multiplying out the matrix results in the following vector addition:

$$\vec{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix} [\hat{x}\ \hat{y}] = x_a\hat{x} + y_a\hat{y}$$

The figure at the right shows unit vectors in three dimensions.

$$\vec{a} = 4\hat{x} + 3\hat{y} + 5\hat{z} = \begin{bmatrix} 4 \\ 3 \\ 5 \end{bmatrix} [\hat{x}\ \hat{y}\ \hat{z}]$$
$$= 4\hat{x} + 3\hat{y} + 5\hat{z}$$

The projection of the vector onto the x-y plane is:

$$a = 4\hat{x} + 3\hat{y} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} [\hat{x}\ \hat{y}] = 4\hat{x} + 3\hat{y}$$

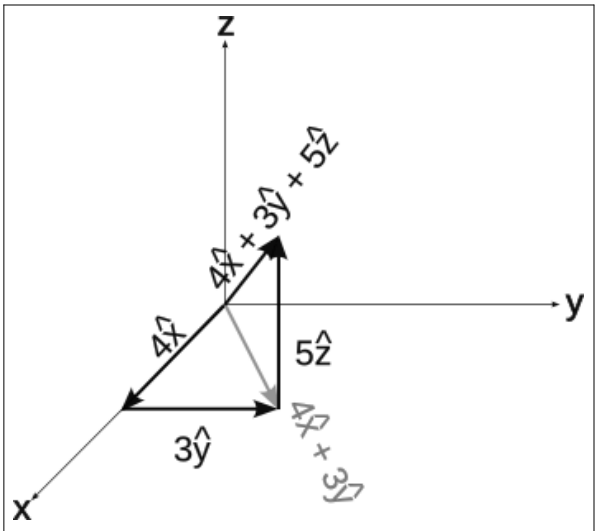


Figure 64: Unit vectors in three dimensions

2.6.1.2 *Vector* Addition

An example of the use of vectors and vector addition is let's say you are hiking in the woods and come upon a fast moving river. You want to swim to the other side and swim straight across the river, but the current carries you downstream. The figure at the right shows your swimming effort as a vector perpendicular to the water flow, the effect of the current flow is shown as a second vector in parallel with the water flow. These two are added together to get your resulting path. Hopefully you swim quick enough to get to the other side before the waterfall.

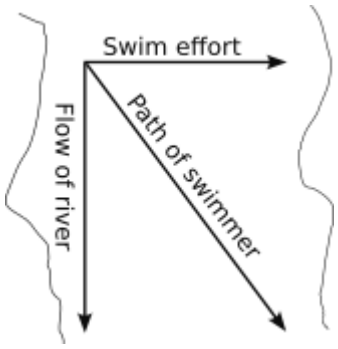


Figure 65: Example of a real-life vector addition

There are two vectors in play here: the force of you swimming perpendicular to the river and the force of the flowing river parallel to the river. If you add these two vectors you can find your actual path across the river.

A vector sum basically is to place the tail of the first vector at the origin and the tail of the second vector at the head of first vector. The resulting vector is the vector from the origin to the head of the second vector. This is the same thing as adding the x components of the two vectors to form the x component of the resulting vector. Do likewise for the y component. That is:

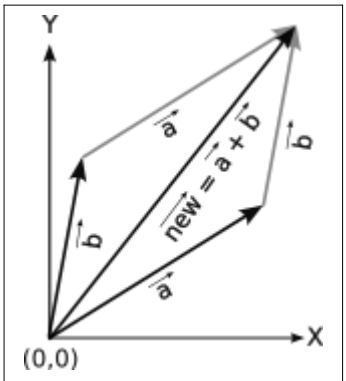


Figure 66: The vector sum of the vectors a and b

$$\vec{x_{new}} = x_a + x_b$$

$$\vec{y_{new}} = y_a + y_b$$

The resulting vector angle can be found by the formula:

$$angle_{new} = \arctan\left(\frac{y_{new}}{x_{new}}\right)$$

The resulting vector magnitude can be found using the Pythagorean theorem:

$$magnitude_{new} = \sqrt{x_{new}^2 + y_{new}^2}$$

You will sometimes see this written in the more compact vector notation as:

$$\vec{a} + \vec{b} = \vec{new}$$

2.6.1.3 Inverting a Vector

The inverse of a vector is a vector with the same magnitude and opposite direction.

When adding some vectors it is sometimes necessary to invert a vector so that the vector can be placed tail-to-head.

Adding an inverted vector is the same thing as subtracting a vector.

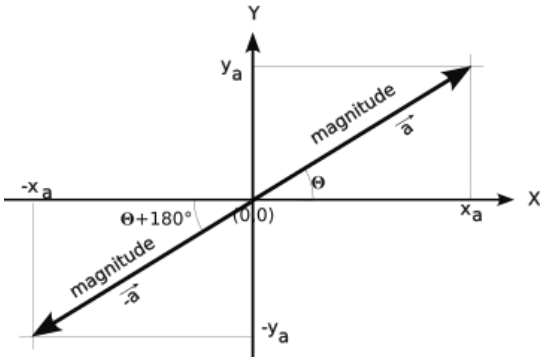


Figure 67: Vector a and its inverse vector -a

2.6.1.4 Vector Multiplication

Vector multiplication has three forms. If you multiply a vector by an undirected quantity or number, you are multiplying by a scalar and the vector is "scaled" by the amount of the scalar.

$$\vec{a} = [1 \quad 2 \quad 3]$$
$$3\vec{a} = 3[1 \quad 2 \quad 3] = [3 \quad 6 \quad 9]$$

There is a dot product which is the sum of the products of individual coordinates. A three-dimensional dot product is:

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$$

The cross product of two vectors is a vector perpendicular to the plane containing the two vectors with a length = the area of the parallelogram bounded by the two vectors.

$$\vec{a} \times \vec{b} = [a_x a_y a_z] \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_x & a_y b_x & a_z b_x \\ a_x b_y & a_y b_y & a_z b_y \\ a_x b_z & a_y b_z & a_z b_z \end{bmatrix}$$

$$\vec{a} \times \vec{b} = [a_x a_y a_z] \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_x & a_y b_x & a_z b_x \\ a_x b_y & a_y b_y & a_z b_y \\ a_x b_z & a_y b_z & a_z b_z \end{bmatrix}$$

2.6.2 Converting a Point Between Frames of Reference

If the frames are orthogonal to each other:

1. Offset the x in the new coordinate system by the offset of the new coordinate system by the offset of the new coordinate origin to the origin of the old coordinate system.
2. Offset the y in the new coordinate system by the offset of the new coordinate system by the offset of the new coordinate origin to the origin of the old coordinate system.
3. Change the sign of rotations if the rotation is between the two frames of reference are different.

Example:

Convert a point in a driver frame of reference to the field frame of reference. If the driver is on the blue alliance, you are done. Otherwise:

a) x_{driver} = field length – x_{field}

b) y_{driver} = field width – y_{field}

c) $angle_{drver}$ = $angle_{field}$ +180 mod 360

Example:

If the frames do not line up:

a) Find the vector from the old frame of reference to the new frame of reference, i.e. a translation

b) Add the angular difference between the two frames of reference to the translation vector to form a transform.

c) Transform the old frame of reference to the new frame of reference, i.e., add the transform to the old vector.

Example:

Convert a point on the robot to the field frame of reference (2d).

a) Find the length of the vector from the robot center to the point.

$$length = \sqrt{x_{robotToPoint}^2 + y_{robotToPoint}^2}$$

b) find the angle of that vector in the robot frame of reference:

$$angle = \arctan\left(\frac{y_{robotToPoint}}{x_{robotToPoint}}\right)$$

c) find the field x- and y-components of the point in field orientation.

$$x_{fieldRobotToPoint} = length \cdot \cos(yaw_{robot} + angle)$$
$$y_{fieldRobotToPoint} = length \cdot \sin(yaw_{robot} + angle)$$

d) add those components to the x- and y-components of the robot field position

$$x_{FieldToPoint} = x_{fieldToRobot} + x_{fieldRobotToPoint}$$
$$y_{FieldToPoint} = y_{fieldToRobot} + y_{fieldRobotToPoint}$$

2.6.3 Converting a Vector Between Frames of Reference

To convert a vector from one frame of reference to another:

1. If both frames measure angles in different directions, change the sign of the vector angle
2. Add the angular difference between the two frames of reference to the vector angle.
3. Convert the resulting vector to its x and y components in the new frame of reference

Example:

Convert a robot motion in its frame of reference to the field frame of reference (2D).

a) the heading is found by adding

$$angle_{robotFieldHeading} = robot_{yaw} + angle_{robotHeading}$$

b) the velocity stays the same.

c) The x- and y- components of the motion may be found as:

$$velocity_{RobotFieldX} = velocity_{robot} \cdot \cos(angle_{robotFieldHeading})$$
$$velocity_{RobotFieldY} = velocity_{robot} \cdot \sin(angle_{robotFieldHeading})$$

Example:

Convert a robot motion downfield and to the right from the driver frame of reference on the red alliance to the red team to the robot frame of reference.

a) Change the heading of the robot to 315° in the drive frame of reference. If the driver is on the blue alliance this is 315° and if the driver is on the red alliance this is 135° (315+180 mod 360).

b) For non-holonomic drives like differential, tank, and West Coast drives, command the robot to drive toward the above angle turning as necessary.

$$angle_{desiredRobotYaw} = angle_{headingDesired}$$

For holonomic drives like swerve and Mecanum, command the robot to drive toward the above heading without changing the yaw of the robot. The command angle would be:

$$angle_{robotToHeading} = angle_{headingDesired} - angle_{robotYaw}$$

2.6.4 WPILib Classes for Vector Manipulation

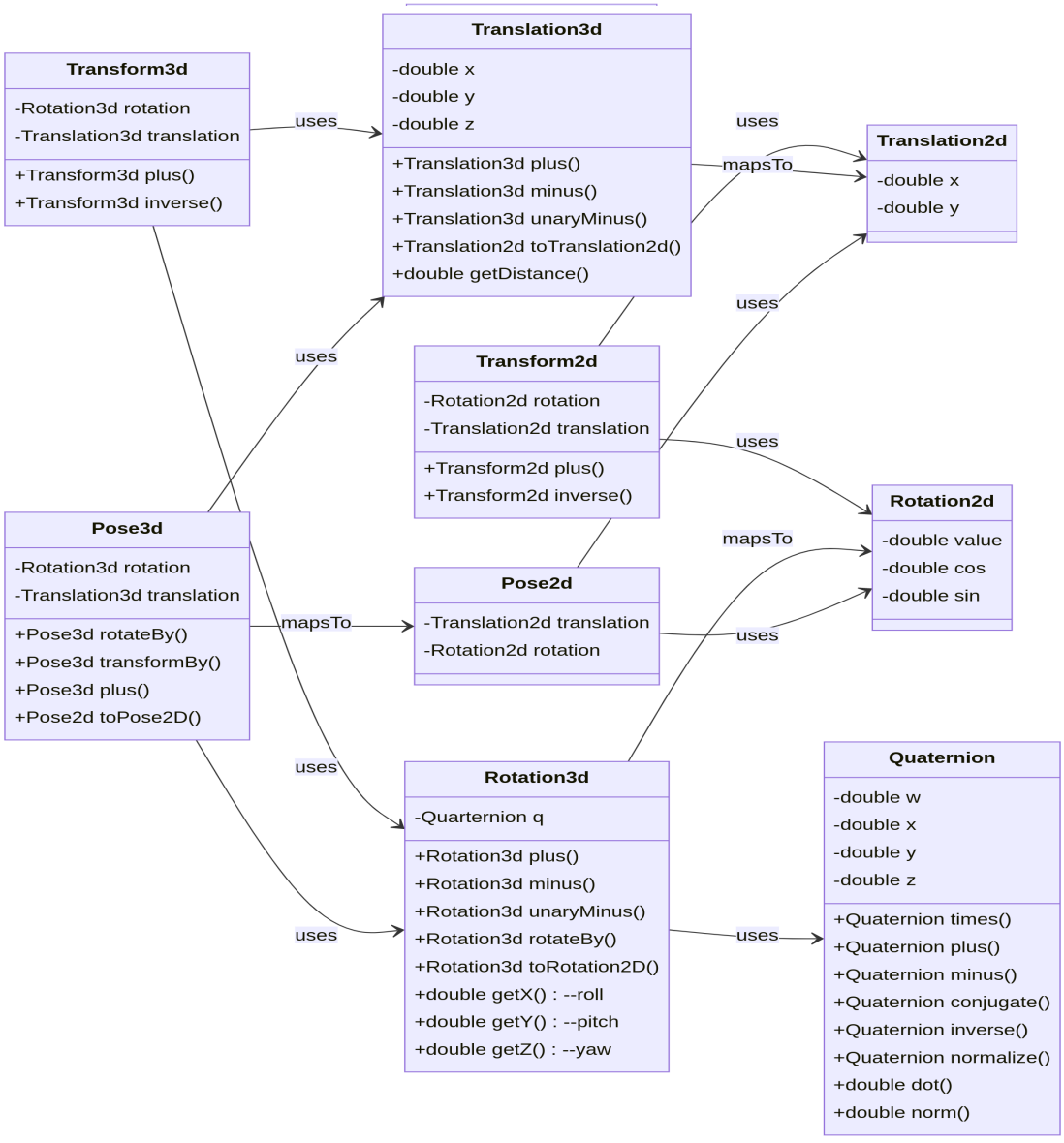


Figure 68: Class diagram for pose translation and rotation

The WPI has classes for expressing vectors, location, and orientation. These three uses can be confusing, so let's sort out the basic functions of the classes.[4]

Translation2d is either a point (x, y) or a vector from an origin (0,0): $\begin{bmatrix} x \\ y \end{bmatrix}$

4 Only some attributes and methods are listed. Each includes its type and access modifier: "-" for private and "+" for public.

rotateBy(φ) rotates the x and y translations by angle **φ**: $\begin{bmatrix} x \\ y \end{bmatrix} \times \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix}$.

plus() adds two Translation2d objects results in: $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \end{bmatrix}$.

Translation3d is a point (x, y, z) in a three-dimensional space: $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$.

getDistance() returns the distance between another Translation3d object: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$.

getNorm() returns the distance of the translation: $\sqrt{x^2 + y^2 + z^2}$.

rotateBy(Rotation3d) returns the current translation p rotated by the Rotation3d q using a quaternion rotation: $p_{current} q_{other} \overline{p_{current}}$. (See 2.6.5 Quaternions.)

toTranslation2d() returns a Translation2d object using only the x- and y-components.

plus() adds a Translation3d object to the current: $\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{bmatrix}$.

minus(): subtracts a Translation3d object from current: $\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} - \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{bmatrix}$

Rotation2d defines an angle in radians. Its reference angle is determined by context.

rotateBy() rotates a Rotation2d object by another.

Rotation3d defines a three-dimension angle from a reference angle with roll (about the x-axis), pitch (about the y-axis) and yaw (about the z-axis) components. (This assumes the forward direction on on the x-axis.) Internally it uses a quaternion to represent the angles and converts only on request. See 2.6.5 Quaternions.

plus() add two Rotation3d objects. This is same as rotateBy()

minus() subtracts one Rotation3d object from another. This is the same as rotateBy(unaryMinus(other Rotation)

unaryMinus() returns the rotation in the opposite direction

rotateBy() rotates a Rotation3d object by another.

getX() returns the rotation about the x-axis or roll angle.

getY() returns thte rotation about the y-axis or pitch angle.

getZ() returns the rotation about the z-axis or yaw angle.

Pose2d expresses a Pose2d location on a plane with x- and y-coordinates, and a Rotation2d to express the orientation.

plus(Transform2d) returns the Pose2d representing the current pose displaced and rotated by the Transform2d. **φ**:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ \varphi_{new} \end{bmatrix} = \begin{bmatrix} x_{pose} \\ y_{pose} \\ \varphi_{pose} \end{bmatrix} + \begin{bmatrix} \cos(\varphi_{pose}) & -\sin(\varphi_{pose}) & 0 \\ \sin(\varphi_{pose}) & \cos(\varphi_{pose}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{transform} \\ y_{transform} \\ \varphi_{transform} \end{bmatrix} .$$

minus(Pose2d) returns a Transform2d of the other pose mapped onto the current pose.

rotateBy(Rotation2d) returns the current Pose2d's translation and orientation rotated by a Rotation2d about the origin.

transformBy(Transform2d) adds the rotated Transform2d's translation to the current translation and adds the Transform2d's rotation to the current orientation.

relativeTo(Pose2d) returns the error between the current Pose2d and the reference Pose2d.

Pose3d expresses a location in a three-dimension space with x-, y-, and z-coordinates, and an orientation expressed with a Rotation3d.

plus(Transform3d) returns the Pose3d representing the current pose displaced and rotated by the Transform3d. The displacement x-y-z coordinates are rotated with quaternion rotation:

$$q' = pq\bar{p} \text{ where } q' \text{ is the rotated quaternion}$$
$$p \text{ is the rotation quaternion},$$
$$\bar{p} \text{ is the conjugate of } p,$$
$$q \text{ is the quaternion of the displacement}.$$

$$\begin{bmatrix} x_{new} \\ y_{new} \\ z_{new} \end{bmatrix} = \begin{bmatrix} x_{pose} \\ y_{pose} \\ z_{pose} \end{bmatrix} + \overrightarrow{transform\ displacement\ vector\ rotated\ by\ rotation_{transform}} .$$

$$rotation2d_{new} = rotatation3d_{pose} + rotation_{transform}$$

minus(Pose3d) returns a Transform3d of the other pose mapped onto the current pose.

rotateBy(Rotation3d) returns the current's translation and orientation rotated by Rotation3d about the origin.

transformBy(Transform3d) reruns adds the rotated Transform's translation to the current translation and adds the Transform3d's rotation to the current orientation.

relativeTo(Pose3d) returns the error between the current Pose3d and the reference Pose3d.

toPose2d() returns a Pose2d representation of current pose projected onto the x-y plane.

Transform2d is a representation of a movement (translation and rotation) in two-dimensions.

plus(other) returns the combined transform, essentially add the translations and add the rotations.

inverse() returns the opposite translation and opposite rotation.

Transform3d is a representation of a movement (translation and rotation) in three-dimensions.

plus(other) returns the combined transform, essentially add the translations and add the rotations.

inverse() returns the opposite translation and opposite rotation.

2.6.5 Quaternions

WPILib uses quaternions to handle rotations in 3D space. Quaternions are an extension to the complex numbers and are expressed with four components to cover one real axis and three imaginary axes to form a vector.

These may be a little hard to grasp because it is four dimensions and three of those are imaginary. The basic equations for the derivation of quaternions is presented here, but some video lectures on the Internet may be more informative or more intuitive.

Basic quaternions using a narrative description: https://www.youtube.com/watch?v=d4EgbgTm0Bg

Quaternion rotations: https://www.youtube.com/watch?v=zjMuIxRvygQ&t=0s

Rotations: https://www.youtube.com/watch?v=jTgdKoQv738

A playground for quaternions: https://eater.net/quaternions. This is an interactive video so you get and explanation and demonstration of changing various variables which you can pause at any time and experiment with your own changes.

2.6.5.1 Complex Numbers

Quaternions are an extension of complex numbers, so this is a quick explanation of complex numbers and some of their properties. Complex numbers are represented by a real part and an imaginary part. This is expressed as:

$$a + bi \text{ where } i = \sqrt{-1} \ .$$

Graphically this can be represented in two-dimensional space with the real part on the horizontal axis and the imaginary part on the vertical axis as shown the figure to the right. This is like a vector, so the magnitude and angle of the complex number is:

$$|a+bi| = \sqrt{a^2+b^2}$$

$$\angle(a+bi) = atan\frac{b}{a}$$

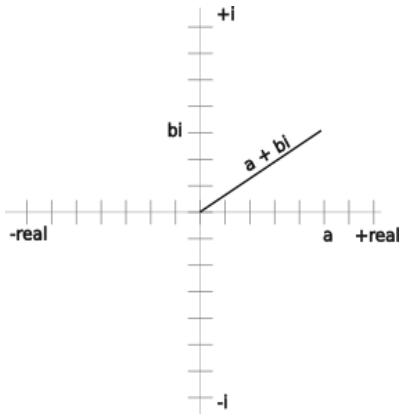


Figure 69: Graphic representation of a complex number

Two imaginary numbers can be added together just by adding the real and imaginary components of the two numbers. This looks like normal vector addition. Add the separate like components together or graphically put the tail of the second vector on the head of the first.

$$(4+i) + (2+2i) = (6+3i)$$

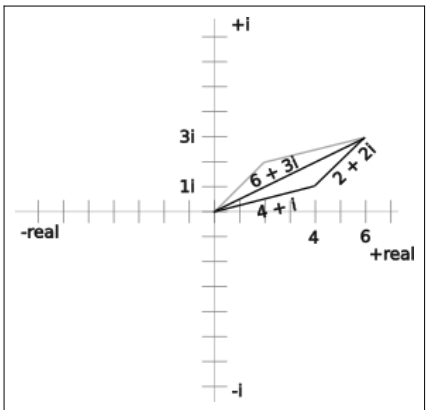


Figure 70: Complex number addition

A normalized vector has a magnitude of 1. A complex unit vector is normalized and the real part is equal to cos(Θ) and the imaginary part is equal to sin(Θ) for any angle Θ.

$$(\cos(\Theta) + \sin(\Theta)i)$$

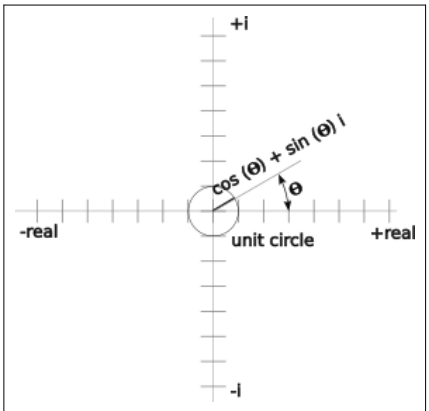


Figure 71: A unit complex number

Now let's create another imaginary number (1+ 4i) and multiply it by the unit imaginary number:

$$(a + bi)(\cos(\Theta) + \sin(\Theta)i) \quad \text{the general form}$$

$$(1 + 4i)(\cos(\Theta) + \sin(\Theta)i)$$

Using (1+4i):

$$\cos(\Theta) + \sin(\Theta)i + 4\cos(\Theta)i + 4\sin(\Theta)i^2$$
$$(\cos(\Theta) - 4\sin(\Theta)) + (\sin(\Theta) + 4\cos(\Theta))i$$

Set Θ to 30°:

$$\cos(30°) - 4\sin(30°) + (\sin(30°) + 4\cos(30°))i$$
$$0.866 - 4 \cdot 0.5 + (0.5 + 4 \cdot 0.866)i$$
$$-1.134 + 3.964i$$

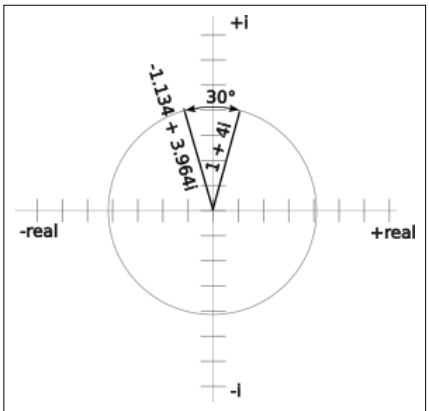


Figure 72: (1 + 4i) rotated 30°

The magnitude and angle of the complex number vector is:

$$|-1.134 + 3.964i| = \sqrt{-1.134^2 + 3.964^2} = 4.123$$
$$\angle(-1.134 + 3.964i) = atan(\frac{3.964}{-1.134}) = -74.03°$$

Correcting the arc tangent for quadrant II (See 2.5.13 More on Arc Tangents and Quadrants):

$$-74.03° + 180° = 105.97°$$

The magnitude and angle of the original complex vector was:

$$|1 + 4i| = \sqrt{1^2 + 4^2} = \sqrt{17} = 4.123$$
$$\angle(1 + 4i) = atan(\frac{4}{1}) = 75.96°$$

The magnitude is unchanged, and the angle has changed by 30° to 105.96°. This shows that the multiplication of a complex number by a normalized complex number just rotates the complex number about the origin of a complex plane.

Let's do this for the another complex number:

$$|4 + i| = \sqrt{4^2 + 1^2} = \sqrt{17} = 4.123$$
$$\angle(4 + i) = atan(\frac{1}{4}) = 14.04°$$

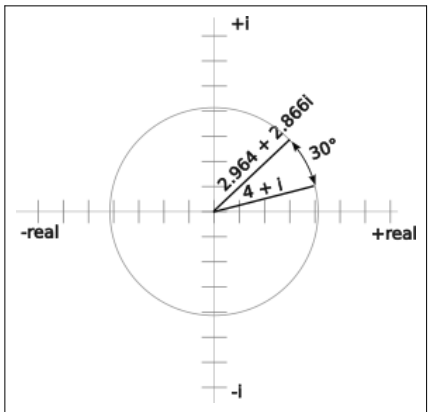


Figure 73: (4 + I) rotated 30°

Multiplying the complex number by a unit complex number at 30°:

$$(4 + i)(\cos(\Theta) + \sin(\Theta)i)$$
$$(4\cos(\Theta) + 4\sin(\Theta)i + \cos(\Theta)i + \sin(\Theta)i^2)$$
$$(4\cos(\Theta) - \sin(\Theta)) + (4\sin(\Theta) + \cos(\Theta))i$$

Set Θ to 30°:

$$4\cos(30°) - \sin(30°) + (4\sin(30°) + \cos(30°))i$$
$$4 \cdot 0.866 - 0.5 + (4 \cdot 0.5 + 0.866)i$$
$$2.964 + 2.866i$$

The magnitude and angle of this complex number is:

$$|2.964 + 2.866i| = \sqrt{2.964^2 + 2.866^2} = 4.123$$
$$\angle(2.964 + 2.866i) = atan(\frac{2.866}{2.964}) = 44.04°$$

Again the magnitude is unchanged and the angle changed by 30°.

Generalizing, multiplication of a complex number by a unit complex number rotates the first by the angle of the second.

2.6.5.2 Properties of Quaternions

But why quaternions? Quaternions were discovered in 1840 and languished as a mathematical curiosity until it was discovered that they were useful for understanding the quantum mechanics of subatomic particles and doing rotations for three-dimensional video games. The advantage of the quaternions is that they do not have the "gimble lock" problem that other computational methods have. Robots need to do a lot of vector addition for odometry and changing of reference frames for adding some vectors such as those used in April Tag calculations. This makes robots a prime candidate for quaternions.

Complex numbers are expressed in a form like:

$$a+bi \text{ where } i=\sqrt{-1}$$

Quaternions are similarly expressed as:[5]

$$a+bi+cj+dk \text{ where } i^2 = j^2 = k^2 = ijk = -1$$

you may be thinking that the quaternion i, j, and k are all the same as the imaginary number i, but they are not quite the same thing. The above equality equation is the foundation of quaternions and is given, even if it doesn't make algebraic sense.

$i=j=k=\sqrt{(-1)}$ because $i^2 = j^2 = k^2 = -1$

$i=j=k=\sqrt[3]{-1}$ because $ijk = -1$ and $i^2 = j^2 = k^2$

$\sqrt{(-1)} \neq \sqrt[3]{-1}$ combining the preceding statements just doesn't make sense.

A quaternion has a real part: a and an imaginary part : bi +cj +dk. The imaginary part is a vector into an imaginary space with orthogonal (perpendicular in n-dimensions) axes i, j and k.

Quaternions have some interesting properties. You can easily add them, subtract them, negate them, and multiply them. There is a catch with multiplication. It is not communicative. If you have two quaternions q_1 and q_2, q_1q_2 is usually not equal to q_2q_1. The same thing happens when you travel around a sphere, like the earth. Let's say you start at the prime meridian at the equator and go north for a quarter rotation of the earth to the north pole. Then turn right to go east and move another quarter of a turn. You end up 90° east if the prime meridian at the equator. Do the changes in the other order: go east a quarter of a rotation of the earth, then turn north and go a quarter of a turn. You end up shivering at the North Pole with Santa.

2.6.5.3 The Math of Quaternion Operations

Let's look at the addition of two quaternions at a mechanical algebraic and then at more detail. \Let's add q_1 and q_2:

$$q_1=a+bi+cj+dk$$

$$q_2=e+fi+gj+hk$$

$$q_1+q_2=(a+e)+(b+f)i+(c+g)j+(d+h)k$$

5 This document uses the informal notation or I, j and k rather than the more formal $\hat{i}$, $\hat{j}$, and $\hat{k}$ which express the notion that these are orthogonal unit vectors, rather than an attribute of a matrix. $\hat{i} \neq i=\sqrt{(-1)}$.

Subtracting is similar:

$$q_1 - q_2 = (a-e) + (b-f)\,i + (c-g)\,j + (d-h)\,k$$

The dot product is:

$$q_1 \cdot q_2 = ae + bf + cg + dh$$

Multiplication, the cross product of two vectors, is:

$$q_1 \times q_2 = ae + afi + agj + ahk + bei + bfii + bgij + bhik + cej + cfji + cgjj + chjk + dek + dfki + dgkj + dhkk$$

There are a lot of terms coming out of the cross product: i^2, ij, ik, ji, j^2, jk, ki, kj and k^2. It would be nice to eliminate or combine some of those terms. With a little algebraic manipulation of the fundamental quaternion equation $i^2 = j^2 = k^2 = ijk = -1$ there are the following six algebraic derivations of simplifications.

$ijk = -1$ given, and multiplying both sides on the left by i

$iijk = -i$ but ii = -1

$-jk = -i$ multiply both sides by -1

$jk = i$

$ijk = -1$ given and multiplying both sides on the right by k

$ijkk = -k$ since kk = -1

$-ij = -k$ multiplying both sides by -1

$ij = k$

$ij = k$ from above and multiplying both sides on the left by i

$iij = ik$ since ii is = -1

$-j = ik$ reorder

$ik = -j$

$ij = k$ from above and multiply both sides on the right by j

$ijj = kj$ since jj = -1

$-i = kj$ reorder $ki = j$

$kj = -i$

$jk = i$ from above and multiplying both sides on the left by j

$jjk = ji$ since jj = -1

$-k = ji$ reorder

$ji = -k$

$ji = -k$ from above and multiplying both sides on the right by i

$jii = -ki$ since ii = -1

$-j = -ki$ multiply both sides by -1 and reorder

$ki = j$

These simplifications can be summarized with the following multiplication table. The values in the left column are multiplied by the values on the top row The figure to the right shows positive results of multiplying two quaternions clockwise and negative results of multiplying two quaternions counterclockwise.

Table 20: Quaternion multiplications (row times column)

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

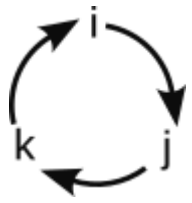


Figure 74: Quaternion multiplication

This shows that the multiplication is not commutative and changing the order of the operands changes the sign of the result, for example: $ij = k$ and $ji = -k$.

Now we have the tools to reduce the cross product equation.

$$q_1 \times q_2 = ae + afi + agj + ahk + bei + bfii + bgij + bhik + cej + cfji + cgjj + chjk + dek + dfki + dgkj + dhkk$$

$$q_1 \times q_2 = ae + afi + agj + ahk + bei - bf + bgk - bhj + cej - cfk - cg + chi + dek - dfj - dgi - dh$$

Collecting similar terms:

$$q_1 \times q_2 = (ae - bf - cg - dh) + (af + be + ch - dg)i + (ag - bh + ce - df)j + (ah + bg - cf + de)k$$

Expressed in matrix notation:

$$q_1 \times q_2 = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \times \begin{bmatrix} e & f & g & h \end{bmatrix} = \begin{bmatrix} ae-bf-cg-dh \\ af+be+ch-dg \\ ag-bh+ce+df \\ ah+bg-cf+de \end{bmatrix} = \begin{bmatrix} ae-bf-cg-dh \\ af+be+ch-dg \\ ag+ce-bh+df \\ ah+de+bg-cf \end{bmatrix}$$

This is a much easier equation than the original cross product. The last matrix is a rearrangement of terms to agree with the code.

$$q_1 \times q_2 = (ae-bf-cg-dh) + (af+be+ch-dg)i + (ag-bh+ce-df)j + (ah+bg-cf+de)k$$

2.6.5.4 The Implication of Quaternion Multiplication

A unit quaternion has a magnitude is 1 or

$$|a+bi+cj-dk| = \sqrt{a^2+b^2+c^2+d^2} = 1$$

When unit quaternions are multiplied together, the result is another unit quaternion rotated about the center. This is similar to how unit complex numbers rotate when multiplied by a unit complex number (See 2.6.5.1 Complex Numbers). A way to think of it is that the first quaternion is operating on the second. This allows rotations to be accumulated without worrying about what happens near the poles of an axis. This rotation allows for easy conversion between different reference points.

Form the complex conjugate of q by negating the imaginary components of a quaternion:

$$q = a+bi+cj+dk$$
$$\bar{q} = a-bi-cj-dk$$

If you multiply q and $\bar{q}$ we get the following:

$$q\bar{q} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \times \begin{bmatrix} a & -b & -c & -d \end{bmatrix} = \begin{bmatrix} aa--bb--cc--dd \\ -ab+ab-cd--dc \\ -ac--bd+ac-bd \\ -ad+-bc--bc+ad \end{bmatrix} = \begin{bmatrix} a^2+b^2+c^2+d^2 \\ -ab+ab-cd+cd \\ -ac+bd+ac-bd \\ -ad+-bc+bc+ad \end{bmatrix} = \begin{bmatrix} a^2+b^2+c^2+d^2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The length of a quaternion is $|q| = \sqrt{q\,\bar{q}} = \sqrt{a^2+b^2+c^2+d^2}$

If we divide $\bar{q}$ by the square of its magnitude we get the inverse of q:

$$q^{-1} = \frac{\bar{q}}{|q|^2} = \frac{a}{a^2+b^2+c^2+d^2} + \frac{bi}{a^2+b^2+c^2+d^2} + \frac{cj}{a^2+b^2+c^2+d^2} + \frac{dk}{a^2+b^2+c^2+d^2}$$

The q^{-1} is a unit vector and it works the same as the unit vector did for imaginary numbers, multiplying it just rotates the quaternion that it is operating on.

To rotate $q^{-1} = \frac{\bar{q}}{|q|^2} = \frac{a}{a^2+b^2+c^2+d^2} + \frac{bi}{a^2+b^2+c^2+d^2} + \frac{cj}{a^2+b^2+c^2+d^2} + \frac{dk}{a^2+b^2+c^2+d^2}$ around a point (x, y, z), define a quaternion as that point with no real part:

$$p = (p_w, p_x, p_y, p_z) = (0, x, y, z)$$

To rotate p, multiply it by q on the left and q^{-1} on the right to get a p rotated by the angle represented by q:

$p' = qpq^{-1}$ for the active direction

$p' = q^{-1}pq$ for the inactive direction

There can be two results for the same resulting rotation. This gives the option of going around the sphere the short way or the long way.

2.6.5.5 Alternative Way of Rotating Quaternions

Using a quaternion in the following form $q = (\cos(\Theta), \sin(\Theta)(bi, cj, dk))$ allows an axis to be defined by b, c, and d and the rotation about that axis to be the angle Θ.

This is not supported by the 2024 WPILib.

2.6.5.6 Conversion Between Quaternion and Vehicle Orientation

The WPILib converts between real world vehicle orientations like roll, pitch, and yaw and the quaternions with the following equations (they use quaternion tuple with w, x, y, and z values to stand in for the mathematical a, b, c, and d values above. The quaternion values are represented here as q_w, q_x, q_y, and q_z to avoid confusion with the Cartesian x, y and z values):

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos(\frac{roll}{2})\cos(\frac{pitch}{2})\cos(\frac{yaw}{2}) + \sin(\frac{roll}{2})\sin(\frac{pitch}{2})\sin(\frac{yaw}{2}) \\ \sin(\frac{roll}{2})\cos(\frac{pitch}{2})\cos(\frac{yaw}{2}) - \cos(\frac{roll}{2})\sin(\frac{pitch}{2})\sin(\frac{yaw}{2}) \\ \cos(\frac{roll}{2})\sin(\frac{pitch}{2})\cos(\frac{yaw}{2}) + \sin(\frac{roll}{2})\cos(\frac{pitch}{2})\sin(\frac{yaw}{2}) \\ \cos(\frac{roll}{2})\cos(\frac{pitch}{2})\sin(\frac{yaw}{2}) - \sin(\frac{roll}{2})\sin(\frac{pitch}{2})\cos(\frac{yaw}{2}) \end{bmatrix}$$

To get the vehicle orientations from a quaternion the following equations are used:[6]

$$roll = atan2(\frac{2(q_w q_x + q_y q_z)}{1 - 2(q_x^2 + q_y^2)})$$

except at the poles where $q_x^2 + q_y^2 < 10^{-20}$ where

$$roll = 0$$

$$pitch = asin(2(q_w q_y - q_z q_x))$$

except at the poles where $2(q_w q_y - q_z q_x) \geq 1$:

6 Note the atan2 function is used to represent a special atan function that returns a value in the normal angular range of 0 to 360° or 0 to 2π radians.

$$pitch = 2(q_w q_y - q_z q_x)$$

or at the extremes where $2(q_w q_y - q_z q_x) \le 1$:

$$pitch = -2(q_w q_y - q_z q_x)$$

$$yaw = atan2\left(\frac{2(q_w q_z + q_x q_y)}{1 - 2(q_y^2 + q_z^2)}\right)$$

except at the poles where $(1 - 2(q_y^2 + q_z^{2)})^2 + 4(q_w q_z + q_x q_y)^2 < 10^{-20:}$):

$$yaw = atan2\left(\frac{2 q_w q_z}{q_w^2 - q_z^2}\right)$$

A note about the WPILib code. There are some differences in the variable names used by the WPILib and those used in this discussion and elsewhere. These differences are shown in the following table.

Table 21: Differences in notations for quaternions

WPILib Rotation Class	Math	Here
w	a	a or q_w
x	b	b or q_x
y	c	c or q_y
z	d	d or q_z
getX()	φ	roll
getY()	θ	pitch
getZ()	ψ	yaw

More information and lectures about quaternions are on the Internet. More details and derivations of the quaternion vehicle orientation conversions are at https://github.com/wpilibsuite/allwpilib/blob/main/wpimath/algorithms.md.

2.6.6 Odometry and Kinematics

Kinematics convert robot chassis x-, y-velocities and a rotational velocity to the required individual wheel velocities (and individual directions for swerve). The calculations vary between drive trains. For deferential drives, it provides the right and left side wheel speeds, for Mecanum, it provides the speeds of all four drive motors. For swerve drives it provides the speed and direction for each module.

In the WPILib the odometry classes are used to update and reset the estimation of robot pose (x, y, and orientation in the field frame of reference). To do this, odometry adds up the x, y and rotational movements of the robot chassis every 20ms. This movement is calculated from the movement of the

individual robot wheels using inverse kinematics. This converts the individual wheel movements back into the movement of the chassis.

2.6.6.1 Odometry

Odometry provides an estimate of the robot pose (x, y and rotation) in the field frame of reference. It is useful to estimate the robot pose for accurate movements during the autonomous or teleop period. This pose can be used with commands to send the robot to a known field position and orientation. It can also be used to accurately score to a target.

Odometry does the classic kinematics by summing up the incremental movements of the robot periodically. Classically this is represented as the equations:

$$x_{robot} = x_{\text{robot-start}} + \sum velocity_x \cdot t$$

$$y_{robot} = y_{\text{robot-start}} + \sum velocity_y \cdot t$$

Where *t* is the incremental time. Since we are measuring the motion based on the rotation of a wheel, we don't need to calculate the instantaneous velocity as in the previous equations and may use the x- and y-components of the movement directly:

$$x_{robot} = x_{\text{robot-start}} + \sum x_{incremental}$$

$$y_{robot} = y_{\text{robot-start}} + \sum y_{incremental}$$

$$rotation_{robot} = rotation_{\text{robot-start}} + \sum rotation_{incremental}$$

The $x_{incremental}$ and $y_{incremental}$ are measured in the robot frame of reference and converted to the field frame of reference before being accumulated. $Rotation_{incremental}$ is the same in any frame of reference. Since the gyroscope also measures the rotation, it is not necessary to accumulate it.

The **updating** process just adds the incremental movements to the accumulated totals. This is done in the robot periodic class.

The **reset** process sets the accumulated values to a known pose or x, y and rotation in the field frame of reference. The known position may be reported from the vision system processing April tag or by "bumpering up[7]" to a field element in a known location. (See also 2.8 Correcting Gyroscope Yaw Angle with April Tags.)

Accuracy is predicated on having the sampling time small enough. WPILib periodic routines typically run every 20ms. For a robot capable of moving 20 ft/s, this is measuring increments of 0.4 inches or about 10 mm. A problem with odometry based on wheel rotations is that it cannot account for wheel slippage or sliding. This error may be small during the autonomous period where all movements are controlled, and the time period is small. This error is larger and accumulative during the teleop period.[8]

7 To move the robot so that its bumper touches a known field element and position, like a loading station or a shooting position.

8 Another technique for location measurement is inertial guidance that sums up the incremental movements in three dimensions using the x, y and z acceleration and the yaw, pitch and roll values supplied by the gyroscope. This technique is used for missile, aircraft, submarine and quadcopters, but is not known to be used in FRC robotics. It seems like this should be possible, but the gyroscopes may not be able to handle the instantaneous velocity changes caused by collisions. Even so it seems like it may be better than dealing with wheel slips and slides.

2.6.6.2 Kinematics

Kinematics takes a chassis velocity (x, y and rotation) to find the required wheel velocities.

Since the required motion is normally stated in field frame of reference, these movements need to be converted to the robot frame of reference. Then the desired wheel velocities (and orientation) can be derived.

2.6.6.3 Differential or Tank Drive Kinematics

Relative to the robot frame of reference a tank drive has no y-velocity component. It can only move forward or backward or rotate by having a difference between the drives on the left and right sides.

The rotation vector is tangential to the wheel and forward on the right side and backward on the left side, and hence the name differential. It is the product of the effective radius of the turn, r, and the angular velocity, ω. The radius of the turn is one half of the wheelbase.

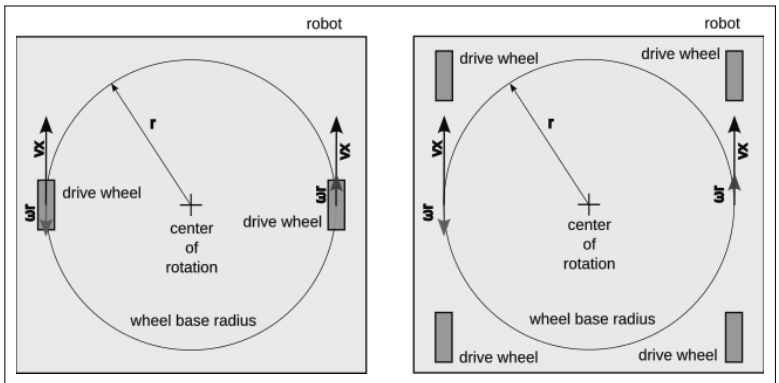


Figure 75: Robot differential drive kinematics

There is no y-velocity relative to the robot frame of reference in a differential drive.

If the joystick is set up for driving from the robot frame of reference, forward and backward movement on the joystick moves the robot forward and backward. Moving the joystick side to side controls the rotation of the robot. These can be applied simultaneously.

For arcade drive the joystick commands are interpreted as movements in the field frame of reference. The robot must be oriented to the intended movement direction. To make the movement smooth, the rotation of the robot can occur while the robot moves forward. Reversing the orientation of the robot takes special consideration and control.

2.6.6.4 West Coast Drive Kinematics

The kinematics for West Coast drives is similar to differential drives except that there are two centers of rotation which change as the center of gravity shifts forward and backward. The center of gravity changes only slightly to cause this change. The center of rotation is affected by the center of gravity, so the diagram to the left greatly exaggerates the difference in centers of rotation. The actual centers of rotation should be determined experimentally.

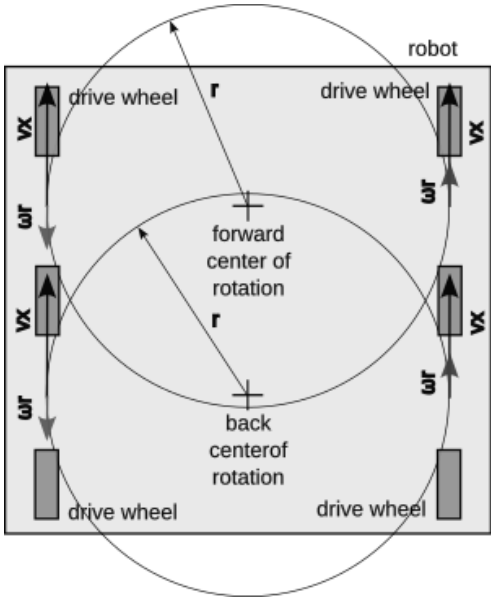


Figure 76: West coast drive kinematics

2.6.6.5 Mecanum Drive Kinematics

Explaining the kinematics for Mecanum drives in any detail is beyond the scope of this paper because the math is beyond the reach of the author.

The graph to the right shows how the drive motors would have to be controlled to achieve the desired robot velocity. This does not account for the position of the motors on the robot, not does it consider any other angle than those shown.

Mecanum wheels apply a force perpendicular to the rollers on their wheels that is 45° off of the wheel direction. The wheels are arranged so that adjacent wheel apply an opposing force when the wheel turn in the same direction.

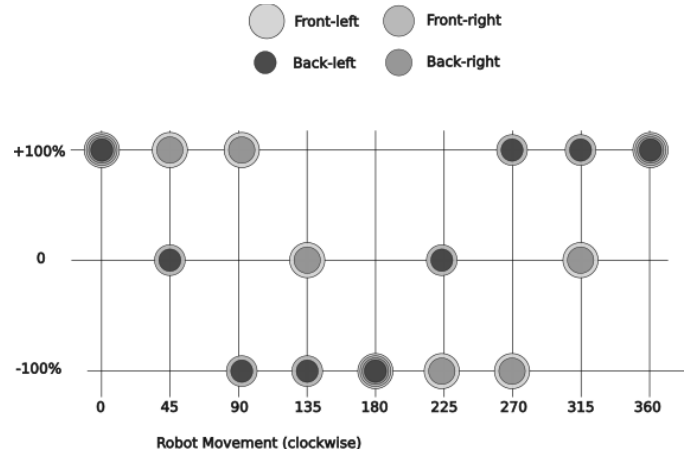


Figure 77: Graph of motor velocities vs. desired robot direction

The rotational component for Mecanum wheels is when each side moves in the same direction, but opposite to the other side. This is fairly easy to add or subtract this component from the normal translation movements.

2.6.6.6 Swerve Drive

Each drive is commanded with the vector sum of the x-component, y-component and rotational component. The latter is tangential to the radial from the center of the robot to the particular wheel. The x- and y-components are applied to all wheels.

The radius of wheel location used for the rotation calculation is normally the based in the wheelbase L and the track width W.[9]

$$r = \frac{\sqrt{(L^2 + W^2)}}{2}$$

$$v_{1x} = v_x + (\omega r)_x = v_x + \omega L/2$$

$$v_{1y} = v_y + (\omega r)_y = v_y - \omega W/2$$

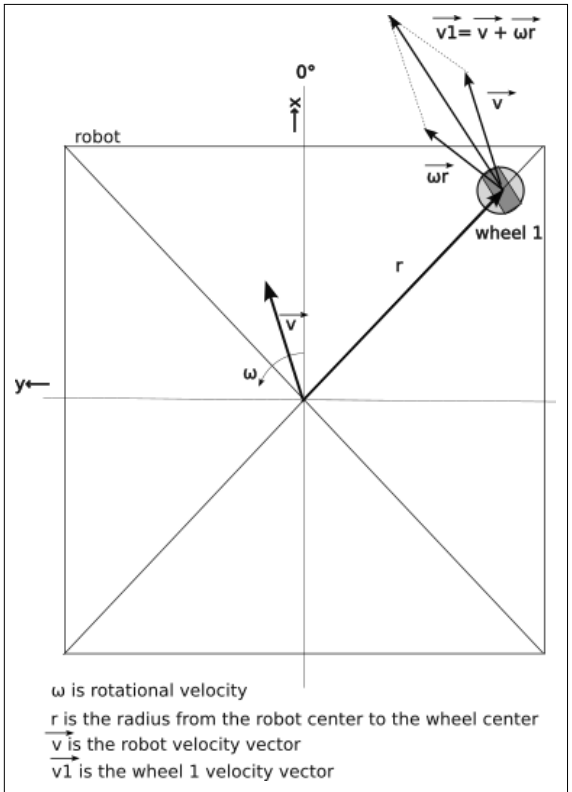


Figure 78: Vectors for kinematics for one swerve drive wheel

9 The derivation of the swerve drive kinematics is from an unsigned paper on the Internet. It used clockwise rotations. This has been modified to be counterclockwise to match my understanding of the robot frame of reference. The direction of the rotation can be reversed by negating the y-component.

Solving for the velocity and angle of each wheel in the robot frame of reference:[10]

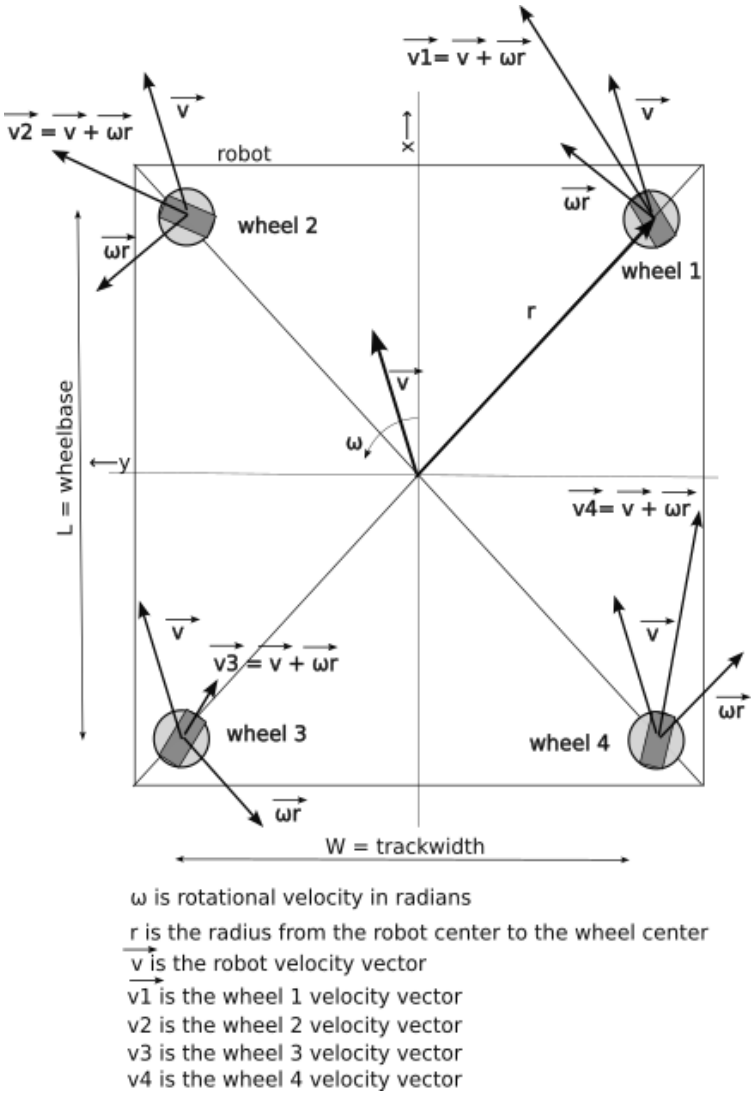


Figure 79: Kinematic vectors for all four swerve drive modules

$$velocity_{wheel1} = \sqrt{(v_x + \omega L/2)^2 + (v_y - \omega W/2)^2}$$

$$angle_{wheel1} = atan\left(\frac{v_y - \omega W/2}{v_x + \omega L/2}\right)$$

$$velocity_{wheel2} = \sqrt{(v_x + \omega L/2)^2 + (v_y + W/2)^2}$$

$$angle_{wheel2} = atan\left(\frac{v_y + \omega W/2}{v_x + \omega L/2}\right)$$

10 Note that the atan (arc tangent) functions are assumed to return values in the range -180° and +180°

$$velocity_{wheel3} = \sqrt{(v_x - \omega L/2)^2 + (v_y + \omega W/2)^2}$$

$$angle_{wheel3} = atan\left(\frac{v_y + \omega W/2}{v_x - \omega L/2}\right)$$

$$velocity_{wheel4} = \sqrt{(v_x - \omega L/2)^2 + (v_y - \omega W/2)^2}$$

$$angle_{wheel4} = atan\left(\frac{v_y - \omega W/2}{v_x - \omega L/2}\right)$$

2.6.7 Inverse Kinematics

The goal of inverse kinematics is to find the robot velocity from motion of its wheels. The gyroscope provides the yaw angle of the robot the incremental change in yaw is ω.

For each wheel subtract the rω contribution for that wheel (or Wω/2 from the x-component of the wheel velocity and Lω/2 from the y-component of the wheel velocity.) What remains should be the velocity of the robot.

2.6.8 Vision and April Tags

Many teams use vision to augment and automate many commands within the robot. These vision subsystems can use April Tags locate the position of the robot on the field to a few inches and can find targets or game pieces. Most vision systems are based on an open source vision library called OpenCV or Open Computer Vision. This library has tools to separate colors and to find particular shapes within the field of vision. This finding of shapes takes a lot of computing power. To decrease processing load on the roboRIO this processing is done on a coprocessor and likely a separate coprocessor for each camera used for computer vision and may be even separate coprocessors to separate April Tag processing from other vision tasks.

These coprocessors generally use one of two different libraries for vision processing: the proprietary LimeLight OS and the open source PhotonVision. Each has particular advantages and disadvantages over the other. Many teams use Raspberry Pi or Orange Pi single board processors to host the coprocessing.

The camera may connect to the coprocessor using the board's built-in camera interface or by using one of its USB ports. This opens some choices for the camera to be used. More pixels mean more resolution and more processing power required to process the images and in turn may mean more latency to process an image. This latency is an issue not only in the delay in measurement, but in the processing of an image itself. Most cameras scan the image from top to bottom, so there can be some skew in the image due to motion (and all robots want to be in motion). This can be overcome with a camera with a global shutter where the entire image is locked before scanning, so it does not have the skewing issue. These cameras are less common and offer less choices.

Another use of vision is to stream the video to the drive station. This lets the driver and operator see things from the robot perspective to see game pieces, load stations, and targets with a closer than direct view.

Odometry has a couple of inherent problems. It cannot accurately account for all robot movement, like when the wheels spin, the robot slides due to a collision, or the robot is airborne (like crossing a cable

protector or jumping off the charge station in the 2023 game). There needs to be a reliable periodic way to reset the odometry pose quickly and accurately.

One solution is April Tags and vision processing. The basic idea is that from the vision system can provide information to reset the robot x and y field position as the robot moves about the field. It does this by computing the angles between the robot and the tag to ascertain the robot's position.

2.6.8.1 Usefulness of April Tags

Odometry calculates a Pose2d object for the robot. This object has three attributes, that form a vector to locate a robot and its orientation on the field:

- x
- y
- orientation angle or yaw.

For odometry these attributes are with respect to the field frame of reference, but other poses may not be.

The problem is that the odometry data is good only for short durations of time. The wheels may slip or slide and that affects the estimates of the robot x- and y-coordinates. The gyroscope also drifts, especially when the robot is moving, to so the rotational angle may not be precise. Using vision systems to process April tags may update the robot odometry data so that it is useful most of the time. Knowing the precise location of the robot can be used to aid in aiming shooters, aligning the robot, finding selective traffic routes, avoiding game obstacles, etc.

2.6.8.2 How April Tags Work

The vision processing for April Tags starts by locating a general April Tag shape in an image. Once a tag is located generally its corners are determined. These corners allow the actuarial value to be read. This information will indicate to the robot where a particular tag is located and oriented. The corners are averaged to find the center yaw and pitch of the tag from the perspective of the camera. Differences in the length of the top and bottom segments of the tag give a measurement of pitch from the tag to the camera. Differences in the length of the left and right side segments of the tag give a measurement of yaw from the tag to the camera.

The measurements of the tag corners start out as x- and y- coordinates on a flat grid representing the camera image. These positions are converted internally to angles by the vision processing software based on calibration data. The resulting measurement is angular and not linear.

2.6.8.3 April Tags Calibration

The April Tag system is calibrated by moving a 1-inch checkerboard pattern in from of the camera. The vision processing does not know how far away the pattern is or its orientation. It can tell when it is perpendicular to the camera because the grid will become square at the center of focus with foreshortening radiating from the focused square. The amount of foreshortening suggests an angle and that in turn suggests a distance. This process allows the software to map what the camera sees to a set of definitive angles in every part of the field of view. This may be non-linear because angular measurements are only linear on a curved surface with a common radius to a focal point. The camera imaging device is a flat plane. This is complicated by the use of a lens which attempts to apply its own corrective factors to the image. The more points that are calibrated during the process will result in more accurate April Tag processing.

2.6.8.4 April Tag Calculations on the Image

April Tab calculations rely on configuration data, including:

- April Tag field layout, for each tag:
  - Pose3d: x, y, z, roll (normally 0), pitch (usually 0), and yaw relative to the field frame of reference. (See 2.5.9.4 Vehicle Orientation Reference System for explanation of roll, pitch and yaw.)
  - Actuarial value of the tag (its number or identity)
- Camera location on robot:
  - Pose3d: x, y, z, roll, pitch, and yaw relative to the robot frame of reference.
- Camera calibration data. The vision system calibrates the camera to be able to map from an image pixel location to an angular measurement. The calibration process is deep within the vision subsystem and is a way to account for the non-linear mapping of a spherical world through spherical lenses onto a flat image detector. Think of this process as breaking the surface of a sphere into a bunch of flat patches that behave linearly, just as a circle can be described as a set of short straight segments. As a general statement, the better a camera is calibrated, the smaller these patches and the better its measurements will be.

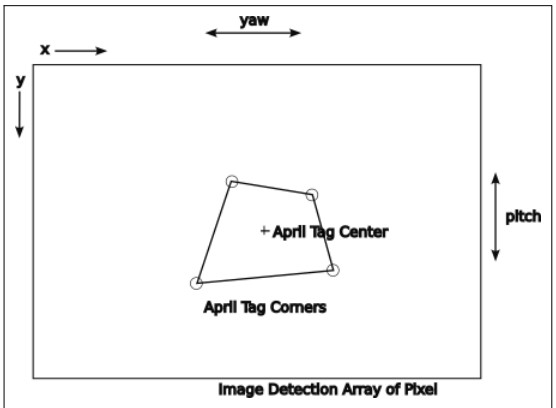


Figure 80: Camera image frame of reference

The calculations also require some real time sensor information, such as

- The yaw and pitch angles of the tag with respect to the camera frame of reference.
- The robot yaw angle from the gyroscope with respect to the field frame of reference.
- The current Pose (x, y and rotation) of the robot (for filtering computed pose estimates).

April Tags have to work in three-dimensions, because the tags are not on the floor, the tags have various mounting angles with respect to the field axes and the robot cameras that “see” them are also not on the floor. Solving the measured angles in two dimensions is not enough.

The roll angle of the April Tags is normally zero. With careful installation the roll angle of the camera should also be zero. We will work with that assumption through the following calculations.

Let’s work through the processing of an April Tag one step at a time

1. First the robot vision system “sees” an April Tag.
2. The image is processed to detect the corners of the April Tag in terms of pixels. The pixels addresses are converted from x and y pixel coordinates to yaw and pitch angles before other calculations are performed because the angular size of the pixels is not uniform.

3. The “actuarial” value (its identifying number) of the tag is determined by its pattern of black and white squares within the tag.
4. The center angles of the April tag are found by taking the averages of the corner pixel yaw angles and corner pitch angles.

$$ImageToTag_{yaw} = \frac{\sum_{n=0}^{3} ImageToTagCorner_{yaw}(n)}{4}$$

$$ImageToTag_{pitch} = \frac{\sum_{n=0}^{3} ImageToTag_{pitch}(n)}{4}$$

2.6.8.5 Calculating Robot Pose From an April Tag Sighting

The April Tag sighting in the previous section can now be used to find the robot’s Pose (x, y and rotation) in the field frame of reference. The figure to the right shows the various vectors used in this calculation. The basic idea is to find the location of a camera based on a known April Tag location and then based on the camera location on the robot and the robot’s orientation to find the location of the robot. This is solved with the following vector sum (the FieldRobotToCamera is in the wrong direction, so it must be inverted in the sum):

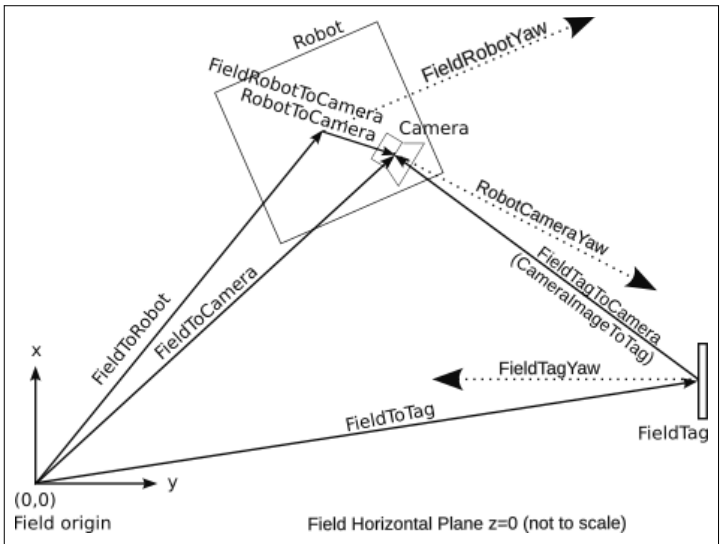


Figure 81: Vectors for locating a robot with a April Tag

$$\overrightarrow{FieldToRobot} = \overrightarrow{FieldToTag_{2D}} + \overrightarrow{FieldTagToCamera_{2D}} - \overrightarrow{FieldRobotToCamera_{2D}}$$

Note that the sum is of two-dimensional vectors, where the actual three-dimensional vectors are projected on to horizontal plane of the field carpet. The complexity of the projection can be hidden with the software libraries. This explanation shows the math using trigonometry to explain the basic calculations.

Naming convention: Vectors are named from tail to head. The frame of reference is included when it is not integral to the tail object. Attributes of a vector are shown as subscripts.

Now solve one step at a time.

1. The $FieldToTag_{2D}$ vector is given as part of the field layout:

$$\overrightarrow{FieldToTag_{2D}} = \begin{bmatrix} FieldToTag_x \\ FieldToTag_y \end{bmatrix}$$

2. The vision system finds the pitch of the target within the field of view of the camera and adds the camera pitch to that to get the CameraToTagPitch.

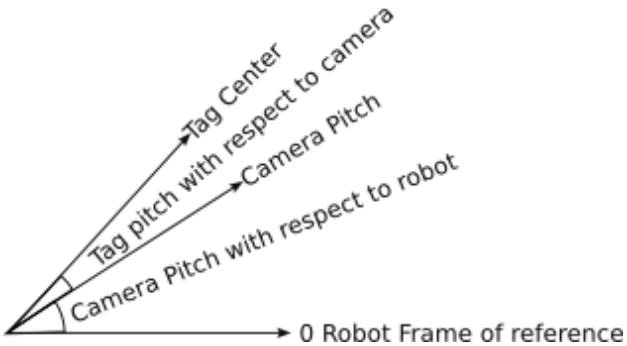


Figure 82: Pitch camera to tag

3. From the CameraToTag pitch compute the CameraTagHorizontalDistance:

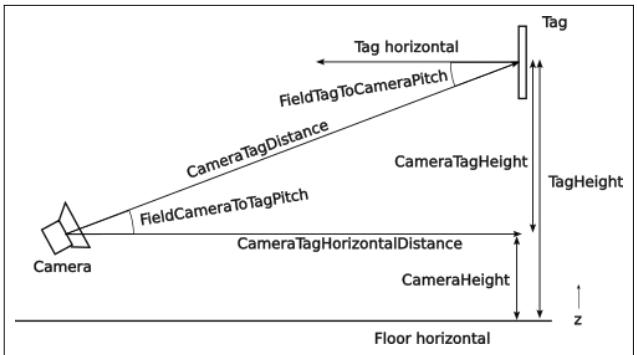


Figure 83: Camera tag vertical plane

$$CameraTag_{horizontalDistance} = (Tag_{height} - Camera_{height}) \cdot \tan(FieldCameraToTag_{pitch})$$

4. The vision system will also provide a CameraToTagYaw angle which can be added to the RobotToCameraYaw and the FieldToRobotYaw to get a FieldCameraToTagYaw angle:

$$FieldCameraToTag_{yaw} = CameraToTag_{yaw} + RobotToCamera_{yaw} + FieldToRobot_{yaw}$$

5. From this angle the FieldTagToCamera vector can be found:

$$\overrightarrow{FieldCameraToTag_{2D}} = \begin{bmatrix} CameraTag_{horizontalDistance} \cdot \cos(FieldCameraToTag_{yaw}) \\ CameraTag_{horizontalDistance} \cdot \sin(FieldCameraToTag_{yaw}) \end{bmatrix}$$

$$\overrightarrow{FieldTagToCamera} = -\overrightarrow{FieldCameraToTag}$$

6. The distance from the robot center to the camera is found as:

$$RobotToCamera_{distance} = \sqrt{(RobotToCamera_x^2 + RobotToCamera_y^2)}$$

7. The angle between the robot axis and the location of the camera is found with:

$$RobotToCamera_{yaw} = atan(\frac{RobotToCamera_y}{RobotToCamera_x})$$

8. The field relative angle from the robot the the camera is:

$$FieldRobotToCamera_{yaw} = FieldToRobot_{yaw} + RobotToCamera_{yaw}$$

9. The field relative vector from the robot to camera can now be computed:

$$\overrightarrow{FieldRobotToCamera}_{2D} = \begin{bmatrix} RobotToCamera_{distance} \cdot \cos\left(FieldRobotToCamera_{yaw}\right) \\ RobotToCamera_{distance} \cdot \sin\left(FieldRobotToCamera_{yaw}\right) \end{bmatrix} \quad \cos$$

10. Solving the original equation gives the FieldToRobot vector.

$$\overrightarrow{FieldToRobot} = \overrightarrow{FieldToTag}_{2D} + \overrightarrow{FieldTagToCamera}_{2D} - \overrightarrow{FieldRobotToCamera}_{2D}$$

The above sequence can be simplified by using vector addition and rotation methods in the WPILib (see 2.6.4 WPILib Classes for Vector Manipulation.) Be mindful of the angles used for rotations.

2.7 Selecting Best Estimated Robot Pose

One of the issues with vision processing of April Tags is that the estimate may not be correct and there may be multiple estimated poses based on a single time instance. The results can be screened with a combination of:

- The tag with the lowest ambiguity a multiple tags are seen at once. Ambiguity is a measure of the probability of the measurement being wrong. Some key stoned shapes of the April tag may be interpreted as one of two (or more) possible poses. Higher uncertainty means more ambiguity.
- A tag with a low enough ambiguity.
- The vision estimated robot pose is closest or close enough to the robot's current estimated pose. This screening allows only "small" changes to be made and does not allow larger changes even when they are necessary.
- Correct the vision estimate by adding the robot movement since the pose was taken to correct for the vision system lag time. (Robot velocities in the x- and y- directions multiplied by the time difference.)

2.8 Correcting Gyroscope Yaw Angle with April Tags

A problem with the April Tag location calculation in 2.6.8.5 Calculating Robot Pose From an April Tag Sighting is its dependency on a gyroscope for the $FieldToRobot_{yaw}$ measurement. This calculation feeds back into the pose estimate that is offered by the vision system which includes corrections for the same $FieldToRobot_{yaw}$, making it a circular reference. One solution would be for the vision system to supply the yaw and pitch of tag to camera vector directly. Another solution is use triangulation to locate the robot and to correct the gyroscope yaw. The steps are along the lines of:

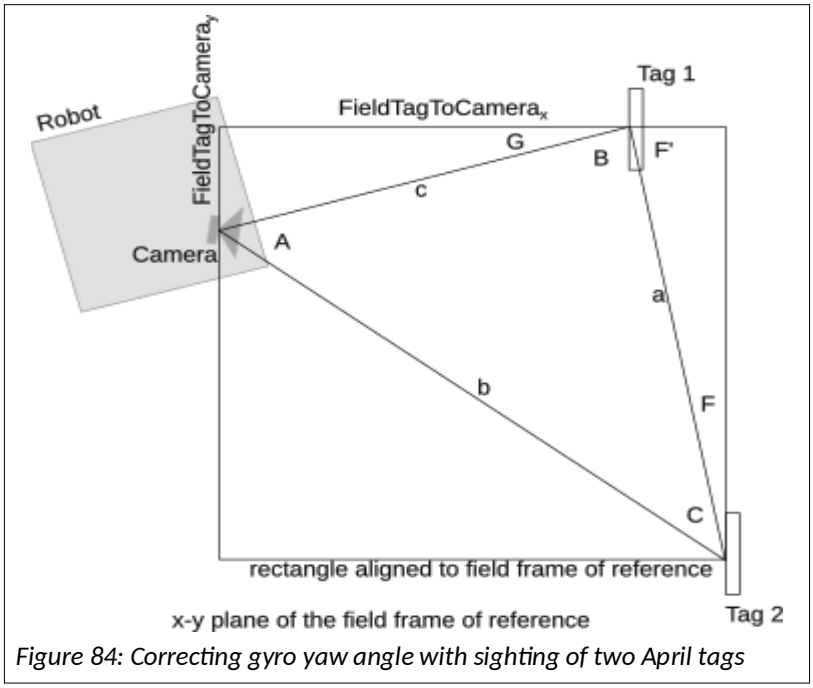


Figure 84: Correcting gyro yaw angle with sighting of two April tags

1. Locate two tags simultaneously. Let's call them Tag 1 and Tag 2. Having two tags at the same time eliminates the latency issue between sightings.
2. Calculate the distances between the tags and the camera using the $CameraToTarget_{pitch}$. Let's call these distances c and b.

$$b = (Tag\,1_{height} - RobotCamera_{height})\ \tan(CameraTag\,1_{pitch})$$
$$c = (Tag\,2_{height} - RobotCamera_{height})\ \tan(CameraTag\,2_{pitch})$$

3. Calculate the static distance between the two tags.

$$a = \sqrt{(Tag\,1_x - Tag\,2_x)^2 + (Tag\,1_y - Tag\,2_y)^2}$$

4. Solve for the angle B using the Law of Cosines:

$$B = acos\left(\frac{a^2 + c^2 - b^2}{-2ac}\right)$$

5. Find the static angles F and F':

$$F = atan\frac{Tag\,1_x - Tag\,2_x}{Tag\,1_y - Tag\,2_y}$$
$$180° = F' + F + 90° \qquad (\text{sum of angles in a triangle})$$
$$F' = 90° - F$$

6. Solve for angle G:

$$G = 180° - F' - B \quad (\text{sum of angles in a straight line})$$

7. Solve for the FieldCamera_{yaw} angle:

$$FieldCamera_{yaw} = -G$$

8. Find the correct robot yaw angle:

$$FieldRobot_{yaw} = FieldCamera_{yaw} - RobotCamera_{yaw}$$

This correction is free from the yaw from the gyroscope, but it is dependent upon the accuracy of the pitch measurements of the camera mounting and vision system. The accuracy can be improved by careful calibration of the angle measured by the vision system and the angle computed with direct measurement. The solution is also not generalized to work with any (reasonable) pair of April tags. The algorithm should be adjusted for the location of the April tags and the location of the robot.

2.9 Networking Basics

FRC robots take advantage of the networking provided by the Internet Protocol (IP) family of protocols, devices, and services. You have used these technologies when you use your computer or phone to access messaging, email, or web pages. This paper distinguishes between the Internet (capitalized), which provides worldwide network, and an intranet or internet (lower case), which is a localized private network. The FRC robot competition system is an internet as is the robot in practice mode.

2.9.1 Internet Protocol

The Internet Protocol (IP) is the procedures and signaling sequences that allow for the transfer of information on a de-centralized network. It is not a singular protocol, but rather a suite of protocols designed to carry a specific kind of information. It uses a stack of protocols similar to the International Standards Organization (ISO) Open System Interface (OSI) model (see https://en.wikipedia.org/wiki/OSI_model). Each layer provides services to the layer above. This allows a protocol on one layer be used by different protocols on the layer above it and it in turn can use the protocols on the layers below it. This architecture allows a lot of flexibility to provide network services. FRC robots take advantage of that flexibility and wide availability of devices to interconnect a robot to a drive station, Field Management System, and other devices within the robot.

2.9.2 Physical Access

Physical access is at the bottom of the stack. Networking allows a variety of connection options to use common communication protocols and methods. It provides a means for transmitting symbols, really bits, over a particular type of physical medium. The two primary mediums used in robotics is wired connections using the Ethernet protocol and wireless connections using the Wi-Fi Protocol.

FRC robots use three physical media for internet access: though a wired Ethernet connection or a wireless Wi-Fi connection. There are other methods that can be used. Cameras can connect to the roboRIO using an internet over USB connection.

Ethernet has evolved from coaxial cables to faster and faster versions of twisted-pair cables and even faster fiber optic cables. The latest and fastest widely used cables are Cat 5E and Cat 6 cables. These cables use 4 pairs of wires to carry signals. The twisting rate of each pair and the spacing between pairs is

engineered to maximize the transmission speed of the data signals in the cable. The cables connect with RJ-45 jacks and plugs. The selection of a specific cable type is beyond this paper, but make sure that the cables are rated for the bandwidth they need to carry. When making cables or connectors, try to maintain the twisting as much as possible.

Wi-Fi has also evolved overtime to where it uses spectrum in three bands for wireless transmission. This spectrum is shared with many Wi-Fi devices and other non-Wi-Fi devices. The Wi-Fi protocol seeks the least-used channel within its spectrum and has an etiquette to back off when it senses interference.

There are three drive station to robot physical connection scenarios:

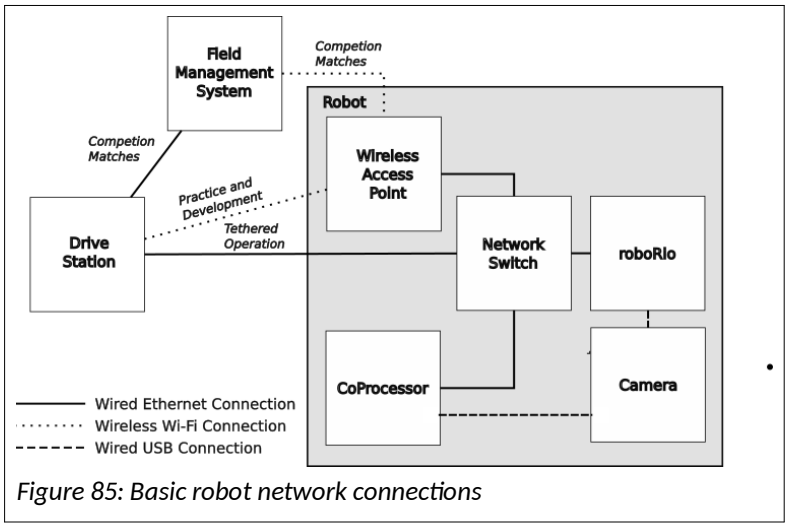


Figure 85: Basic robot network connections

1. Wireless Wi-Fi connection between drive station and wireless access point on the robot. This is used during development and most practice sessions on your home turf.
2. Wired Ethernet connection between the drive station and the Field Management System and then a wireless Wi-Fi connection between the Field Management System and the wireless access point or radio on the robot. This is the normal connection for competition robots.
3. A direct wired Ethernet connection, or tethered connection, between the drive station and the roboRIO (optionally through a network switch). This is used for practice sessions at a competition to prevent interference with the wireless used for the matches.

There is a fourth type of physical connection on some FRC robots. Some cameras connect using a USB (Universal Serial Bus) connection. This connection carries Internet Protocol packets on top of the USB protocol. This allows the camera images to be carried over the internet to the drive station or a vision coprocessing device.

2.9.3 Link Layer

The link layer handles the transfer of information in groups called packets. The Internet uses primarily two link layer protocols: UDP and TCP. UDP or User Datagram Protocol, is a simpler faster protocol, but does not guarantee delivery, meaning that it does not detect failures and does not re-transmit failed packets. TCP or Transfer Control Protocol does provide guaranteed delivery. UDP is used for the transfer of voice and video and may be used when multiple destinations are needed. TCP is used for transferring web pages, file transfers, and email. Robots use both.

2.9.4 Network Layer

The network layer routes packets to their intended destination.

2.9.4.1 Addressing

Every node on an interconnected network has an address. A node may be a computer, a server, a cell phone, a laptop, or any other device. This allows a packet to be sent from any node and for the response to be sent back to the requesting node. The packets used on the network contain both the sender's return address and the destination address. This is flipped at the receiving end to return the response to the sender.

The addresses may be public or private. Public node like google.com can be accessed by any node on the public Internet. Private node exist on either stand-alone networks or private networks behind a firewall. A node on the public network cannot directly access a node on an interconnected private network, but a node on a private network can access nodes on the public Internet. This connection is managed by a router at the interface of the two networks. One side of this router has a public Internet address. The other side of the router has a set of nodes with private addresses. This setup is used with your home router and with your school router. The router runs a protocol called Network Address Translation (NAT) to route packets between the public and private networks (see 2.9.4.5 Network Address Translation (NAT)).

2.9.4.2 IP v4 Addresses

The roboRIO mostly uses the older IPv4 network addresses, since it is a little easier to use and the cost of moving on. These addresses are specified in the **dotted quad** or **dotted decimal** notation, which is four groups of numbers separated by a period. Each of the four numbers is between 0 and 255.0 The first group specifies the network group.

To avoid conflicts with the public Internet addresses, there are special number ranges which are set aside for private use, as shown in the following table.

Table 22: Special IPv4 network address ranges

Range beginning	Range end	Range use
127.0.0.0	127.0.0.255	Local computer. This address is used internally to the computer to access locally defined services, servers, etc.
10.0.0.0	10.255.255.255	Corporate private address space. Addresses in this space are used by some routers for their private addresses.
169.254.0.0	169.254.255.255	Private address space used by Microsoft computers
172.22.11.0	172.22.11.255	Private address assigned by a USB connection to the roboRIO.
192.168.0.0	192.169.255.255	Home or small business private address space. This is used by some routers for their private addresses.

Computers use 127.0.0.1 as a loop back address and is also known as "home." Most routers use either the 10.x.x.x or the 192.168.x.x addresses by default. The roboRIO uses the addresses in the 10.x.x.x range for its uses. Each robot is assigned an address made of the teams number. 10.TE.AM.xx, where "TE" is the first two or three digits (range 0 to 255) of the team number and "AM" is the last two digits of the team number (range 00 to 99). The team numbers are padded on the left with zeros, so Team 20 is assigned the

IP address 10.0.20.xx. The IP addresses assigned for Team 4513 are 10.45.13.xx. Common address assignments are shown in the following table:

Table 23: FRC robot IPv4 private network addresses

Device	Low order network address	Network address	Team 4513's Network Address
Open Mesh Radio	1	10.TE.AM.1	10.45.13.1
roboRIO	2	10.TE.AM.2	10.45.13.2
Driver Station	5	10.TE.AM.5	10.45.13.5
IP Camera	11	10.TE.AM.11	10.45.13.11

2.9.4.3 IPv6 Addresses

The Internet is running out of IPv4 address, and it is slowly incorporating longer IPv6 address. These addresses are consumed quickly by devices like cell phones and Internet of Things (IoT) that are directly connected to the public Internet. IPv6 addresses are 128 bits long as opposed to the 32-bit IPv4 addresses. The Internet currently supports both address types. An IPv6 address is represented by 8 groups of 4 hexadecimal digits separated with colons. Each digit represents four bits, so the group is 16 bits. The address representation can be shortened by omitting the lead zeros in each group and skipping one or more all zero groups with a double colon. For example the address representation 2001:0db8:0000:0000:0000:8a2e:0370:7334 can also be represented as 2001:db8::8a2e:370:7334.

Because both address schemes are in use, you can find both in tools that return network addresses. Over time IPv6 addresses will be more common.

2.9.4.4 Port Addresses

Each node on an internet or the Internet also has a port address which acts as an extension the IP address. This allows routing packets for a particular service a specific application within the node which can interpret and act upon the packets for that service. A port may be assigned for UDP or TCP packets or both.

The Field Management System also limits what ports can be used during competitions. A port is an address within an Ethernet endpoint, which generally supports a single specified service. The FMS allows:

Table 24: Port addresses reserved for specific FRC robot services

Protocols	Port(s)	Usage
TCP, HTTP, and WebSockets	80	Camera/web interface. This is also used for WebSocket services like Network Tables.
UDP	140	Robot-to-Drive-Station status data

TCP, HTTP, and WebSockets	443	Camera/web interface (secure) This is also used for WebSocket services like Network Tables.
UDP/TCP	554	Real-Time Streaming Protocol for h.264 camera streaming
UDP	1130	Drive-Station-to-Robot control data
UDP/TCP	1180–1190	Camera Data
TCP	1735	SmartDashboard
UDP/TCP	5800-5810	Team Use

The following table lists other ports and protocols that may be used in FRC robots when not competing.

Table 25: Port addresses reserved for Internet services

Protocols	Ports	Usage
UDP and ICMP	--	Ping uses Internet Control Messaging Packets (ICMP) to request an echo from a remote address to determine if that network node is connected and alive.
TCP and ftp	20 and 21	File Transfer Protocol (FTP) for transferring files to and from the roboRIO. The Linux command line program for FTP is ftp.
TCP and ssh	22	Secure shell for terminal access on the roboRIO. The Linux command line program for SSH is ssh.
UDP and telnet	23	An obsolete insure protocol for terminal access to a computer. The Linux command line program for telnet is telnet. Use the more secure ssh for replacement.
UDP/TCP	53/853	Domain Name Server (DNS) is used to translate a common language name like “Google.com” to a network IP address. The last part of the name, like “.com” in the example is a domain name. Devices local to a router can be accessed with names in the “.local” domain.
UDP	67 and 68	Dynamic Host Configuration Protocol (DHCP) is used by a network router to assign an IP address to a new device connected to the network from a range of IP addresses. Static or self-assigned IP addresses may also be used (assigning addresses outside the DHCP reserved range). (see 2.9.4.6 Dynamic Host Configuration Protocol (DHCP)).
UDP/TCP	1250	CTRE Diagnostic Server

2.9.4.5 Network Address Translation (NAT)

A router that sits between a private network and a public network uses Network Address Translation to route packets between the two networks. Outbound messages from the private network are modified to use the public address of the router as the return address, so that the response from the public network can be routed back to the router. Additionally, the return address includes a high port number that is mapped to the originating node's private IP address and port number. The destination address of the response is modified to be the saved originating address. Some routers enforce that only the original destination address can send the response packet to prevent hijacking the NAT mapping.

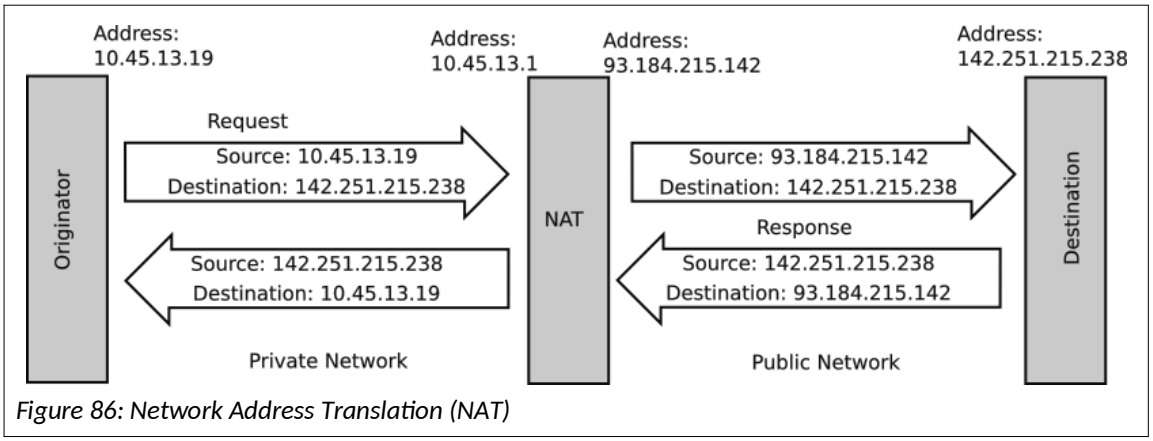


Figure 86: Network Address Translation (NAT)

2.9.4.6 Dynamic Host Configuration Protocol (DHCP)

Network addresses can be dynamically assigned by a protocol called DHCP, or the **Dynamic Host Configuration Protocol**. This allows connecting previously unknown devices to a local network without having to manually manage their network addresses (also known as static IP addresses).

The downside of DHCP is that sometimes a network address of a device is needed, but since it was assigned by DHCP, it is unknown. A tool such as Angry IP may be used on the private network to find all the devices connected.

2.9.5 Network Devices

2.9.5.1 Radio or Wireless Access Point

A Wireless Access Point provides Wi-Fi access. It broadcasts its system identifier (SID) which is used by other devices to select the particular Wi-Fi network. FRC devices use a SID of "FRC-xxxx" where xxxx is the Team number. This is the normal usage mode of the FRC radio. You must use the "FRC Radio Configuration Utility" to program the radio. The Radio imposes a bandwidth restriction of 4 Mbps.

Real world examples of a Wireless Access Point are the Wi-Fi router in you home or school, a cell phone operating as a hot spot.

2.9.5.2 Wireless Bridge

A Wi-Fi connection has two endpoints: a Wireless Access Point and a Wireless Bridge. The Access Point was described in the preceding paragraph. A Wireless Bridge is used to access the Wi-Fi services and network connections provided by a Wireless Access Point. Think of this like your laptop. It can access a

Wi-Fi network using a Wireless Bridge. As long as the laptop is not acting as a hot spot, other devices cannot connect to the laptop using Wi-Fi.

The FRC radio can be configured as a Wireless Bridge.

Real world examples of devices employing a Wireless Bridge is your laptop, Alexa, or other device that accesses the Internet or internet wirelessly.

2.9.5.3 Firewall

A firewall is a device on the edge of a network or node that limits access to protected network or node. The firewall limits which IP addresses and ports that are allowed to pass through it. The Radio can be configured to be a firewall. The Field Management System is a firewall.

Most home and school Wireless Access Points include router and firewall functionality.

2.9.5.4 Router

A router is used to connect a group of wired internet connections. Router repeat incoming packets to the destination interface.

A router provides other services such as DNS, DHCP, and Network Address Translation (NAT).

2.9.5.5 Network Switch

A network switch is used to connect a group of wired internet connections. Simple switches merely repeat packets from any port to all other ports. Advanced switches repeat incoming packets to the destination interface.

A network switch does not provide other services such as DNS or DHCP that would be provided by a router.

2.9.5.6 Servers

A server is a network endpoint that provides a service by interpreting the contents of incoming packets and generating an appropriate response. The roboRIO, Drive Station, and Field Management System provide network services and are servers for specific services.

2.9.6 Network Tools

Network tools are used to diagnose network problems.

- **ping** on the roboRIO and other Linux devices is used to test if a particular address is connected and active.
- **Angry IP** is a tool on a laptop that scans its local network for the IP address of connected devices. This is useful when a device connects and uses DHCP to provide its IP address. Angry IP can find that IP address.
- **traceroute** on the roboRIO and other Linux devices is used to trace the route that a packet is taking to verify that the packet routing and fire walls are set up correctly.

3 Robot Architecture

The robot architecture is divided into several areas. This allows for division of the responsibility of the components. At the highest level the robot breaks down into the following layers:

- **Mechanical** defines the physical robot components from the chassis, drive train, wiring harnesses, subsystem mechanisms, nuts, bolt, rivets, tie wraps, baling wire, chewing gum, duct tape, etc. The intent of this architecture is hold the parts of the robot together and survive the hardships of competitions while supporting moving components to perform the actions required of the robot.
- **Hardware** defines the electrical and pneumatic motors and actuators that move the mechanical components. Hardware also includes the LEDs that dress up a robot.
- **Electrical** defines the distribution of power through the robot to power motors and lights. This is separate and distinct from the control signals (which are electrical by their nature, but in general are not conveying power).
- **Control** defines the systems that control the speed and direction of the hardware components. This include the Pulse Width Modulation (PWM) signals, CAN bus, and the LED serial bus and control leads for relays and solenoids. It also includes the wiring for sensors and interconnecting network devices.
- **Software** is the programs that run on the roboRIO, Drive Station, and Field Management Systems used to drive the controls of the robot.

3.1 Robot Mechanical Architecture

Robots are highly varied and original for each team. This individual nature of robots precludes very much discussion of the mechanical architecture other than to refer you to the game rules for a particular year that specify size, height, bumper, and other constraints.

The challenge in a mechanical design is finding the winning combination of trade-offs between the following factors:

- cost
- weight
- speed
- robustness (ability to survive crashes)
- stability (ability to maintain control)
- functionality
    - complexity (increased functions, less focus)
    - simplicity (decreased function, more focus)
- time to repair
- capabilities of the team

Robots come in many styles, shapes, and sizes. All start with a chassis, a frame to which other components are attached. It must be robust to stand up to collisions with other robots and field elements. Bumpers are added to the chassis to help mitigate the effect of collisions, but not as much as you might think. Other components are added to give the robot its functionality and individuality as dictated by the rules of the game for a given year and choices made by the team.

Mechanical components of a robot may include depending on the objectives of the game:

- Chassis
- Drive train
- Pickup to get game pieces from floor or loading station
- Shooter to launch game pieces toward a target
- Processor to get game pieces into desired position
- Grinder to grind game piece or opposition into dust. Violates the rules if done intentionally or repetitively.
- Elevator to lift other components
- Arm to extend other components
- Turret to spin components independently of the robot chassis
- Climber to lift the robot off the floor.
- Claw to grab things.
- LEDs to dress up the robot and communicate status.

3.2 Robot Hardware Architecture

The hardware architecture includes the electrical and moving elements (motors, relays, and solenoids) of a robot. A more detailed overview of the hardware components including photographs of those components can be found at https://docs.wpilib.org/en/stable/docs/controls-overviews/control-system-hardware.html. That discussion includes components that are not used in every evolution of the robot and is a great reference source.

3.3 Robot Electrical Architecture

The electrical architecture of the robot safely distributes electrical power to the various components and subsystems of the robot.

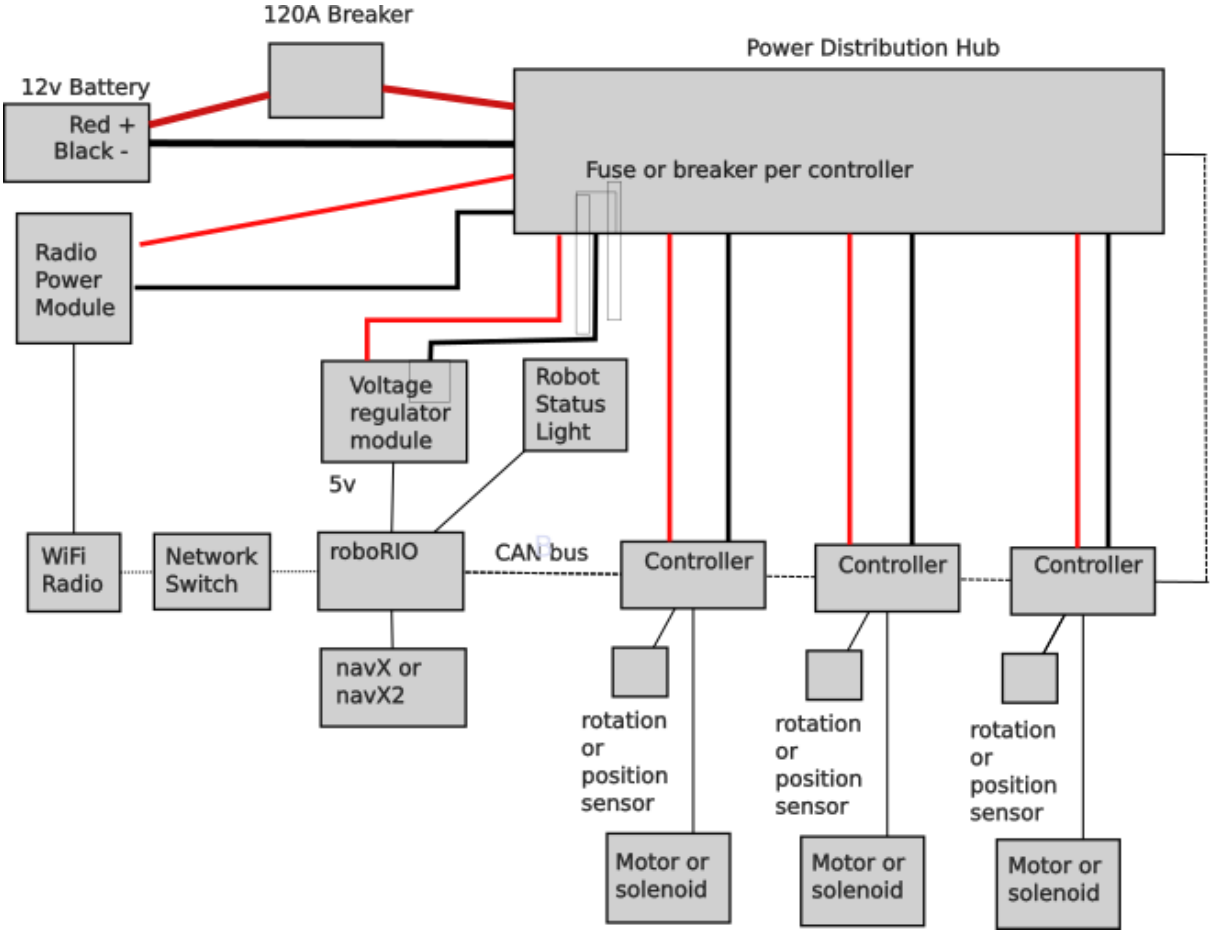


Figure 87: Basic robot electrical architecture

3.3.1 Required Electrical Components

FRC requires that all robots have the following components, so the robots can compete safely:

3.3.1.1 Battery

One and only one gel cell **battery** can be used. Some peripheral devices may have independent battery power as long as they meet restrictions as specified in the rules for a given season.

3.3.1.2 Main Breaker

Main breaker to protect all the components against accidental shorting. This is a 120A breaker. 120A allows a dangerous current flow capable of arcing and melting metal. The main supply wires should ALWAYS be treated with respect and care.

3.3.1.3 Power Distribution Hub

The Power Distribution Hub has circuit breakers to protect individual circuits against taking too much power by limiting how much current the circuit can take. Every motor in the robot has its own breaker, so that the breaker can protect each motor against too much current.

3.3.1.4 roboRIO

roboRIO is the main controller for the robot. It has almost all the team written code for the robot. It's operating system is also part of the Field Management System that controls the start of the robot software for the autonomous and teleop periods as well provides a way to shut down the entire robot in the event its operation is deemed to be unsafe by the field judges. Auxiliary coprocessors are allowed to assist the roboRIO to off load processing for things like vision and navigation. The **roboRIO** should ultimately be in control of all coprocessors and the devices that they may control.

3.3.1.5 Robot Status Light

Every robot must have a clearly visible **Robot Status Light** (**RSL**) that displays important status information of the robot.

Table 26: Meaning of RSL flashing patterns

RSL Flashing Pattern	Meaning
Solid on	Autonomous period enabled
Solid on with off every 1.5 seconds	Teleop period enabled
Slow blink	Disabled by system watchdog or drive station set to disabled
Fast-slow	Battery failing (less than 12v) and system disabled
Fast	System error

3.3.1.6 Wi-Fi Radio

A **Wi-Fi radio** provides communication between the robot and the Field Management System and through it, the driver station. This radio ensures that only the Field Management System can communicate to the robots on the field and that it can enforce bandwidth restrictions so that all robots on the field have access to the same amount of bandwidth. The radio should be mounted where it has minimal electrical interference from other components such as motors.

A new Wi-Fi Radio was introduced at the very end of the 2024 competition season. These radios have some reverse compatibility with the existing radios, but also use the 6GHz band. This new band offers more channels and should have less congestion and interference issues (at least in the beginning). It

suffers from a poor thermal design and runs hot. It must be attached to a metal frame for sinking the heat generated.

3.3.1.7 Robot Power Module

A **Robot Power Module (RPM)** specifically to power the **Wi-Fi Radio** using power over Ethernet (POE).

3.3.1.8 Voltage Regulator

A **voltage regulator** to convert the 12v robot power to 5v power used by some components in the robot.

3.3.2 Status Lights

Beyond the required Robot Status Light, there are many other lights on electrical components to indicate their status. Rather than repeat the list here see https://docs.wpilib.org/en/stable/docs/hardware/hardware-basics/status-lights-ref.html.

3.4 Hardware Architecture

The hardware architecture of the robot fits between the electrical system. and the mechanical systems. Hardware converts electrical power into motion and includes the drive train motors and the motors and actuators of the robot payload.

3.4.1 Drive Train

The drive train provides locomotion to the robot and is central to the functionality of the robot. Currently, there are four major styles of drive train possible (in historical order):

- Tank drive
- West coast drive
- Mecanum drive
- Swerve drive

The default mode for the drive train is the arcade mode which controls the robot from the driver frame of reference. This gives an intuitive way to drive the robot. An alternative mode of control is to control the robot from the robot's frame of reference. While in this mode the driver has to think about directions from the robot's point of view. In either mode, the drive train abstracts the underlying drive mechanism from the controls required to request robot movements.

These drives are discussed in more details in the following sections.

3.4.1.1 Tank Drive or Differential Drive

A tank drive robot is driven by two sets of wheels, one on each side. Most tank drives only have two wheels on each side, but may have more. All of the wheels on a side may be tied together with a chain or belt. Steering is performed by changing the speed of one side versus the other differentially, leading to its other name: differential drive. Tank drive robots are the simplest form of FRC robot drive system. The disadvantage of a tank drive is that it must drag some wheels sideways when it turns. This leads to wheels with less grab and less ability to accelerate quickly or push other robots. The other disadvantage is that the robot can only move in the direction that it is pointed. While it can turn on the move, it may need to perform special maneuvers or stop to change the leading end of the robot.

3.4.1.2 West Coast Drive

A West Coast drive is a variation of the tank drive where the center wheels on each side are set about an 1/8” lower than the outside wheels. This means that the balance point of the robot can change depending on past momentum shifts. Acceleration can move the balance point back, while deceleration will move the balance point forward. Each side has 3 or 4 wheels. Like a tank drive a West Coast drive is steered differentially. The difference is that it has less turning resistance because it does not drag the outer wheels if at all or as much as a tank drive. Another advantage is that it effectively has a shorter wheelbase for tighter turns while having the stability of a longer wheelbase. The outer wheels may be free turning casters or balls to reduce their drag when the robot is turning.

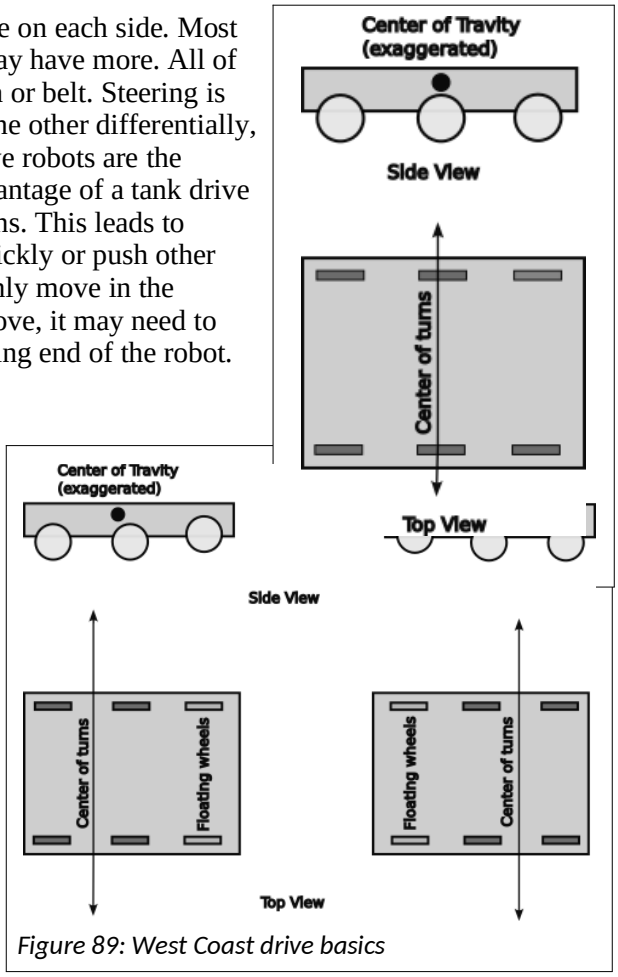


Figure 89: West Coast drive basics

3.4.1.3 Mecanum drive

(photo by Gwpcmu CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=11440618)

Figure 91: Uranus omni directional robot

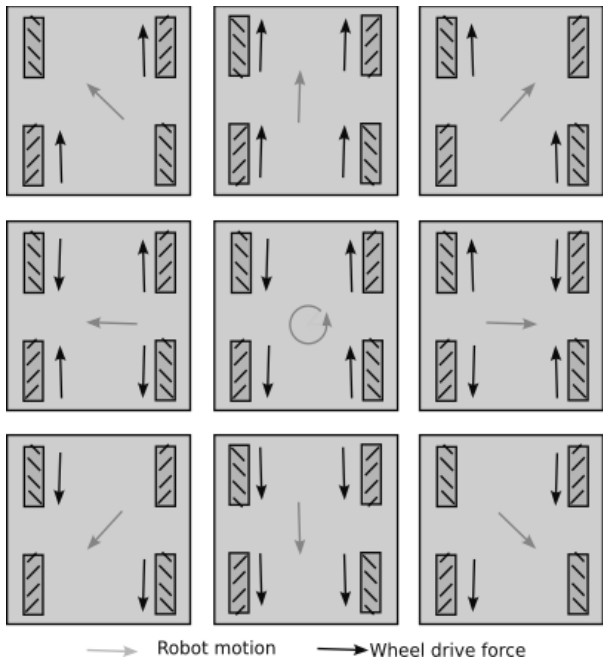


Figure 90: Mecanum drive basic movements

drive uses four special wheels each with an independent drive. Each wheel has rollers attached at 45° to the direction of rotation. As the wheel turns it applies a force perpendicular to the orientation of the roller. By controlling the relative speeds of the motors the robot can be moved in any direction or turn with any center point. It offers the flexibility of a swerve drive with fewer motors. However, the option of tire tread is limited to the roller, so it has less grip on the competition carpet and the robot is pushed around by competitors.

3.4.1.4 Swerve Drive

Swerve drive uses two or more swerve drive modules to power the robot. Each module is like a caster using one motor to steer the wheel and another motor to drive the wheel. The center of rotation for turns can be anywhere. Robots with a swerve drive can spin or twist while moving forward, providing for more fluid movement than possible with a tank or West Coast drive. It has various wheel, motor, and gearing options to suit the requirements of each team.

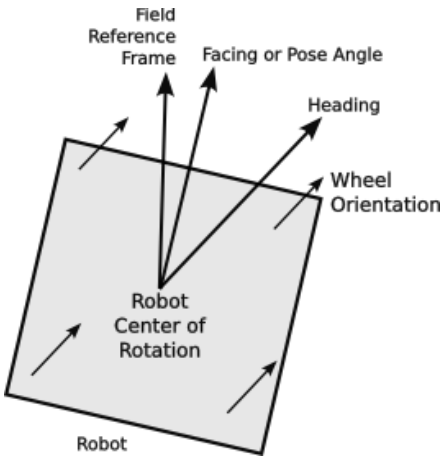


Figure 93: Swerve drive angles

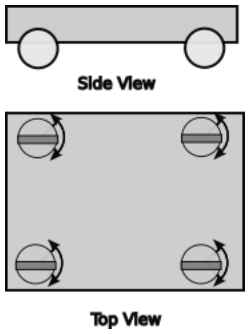


Figure 92: Swerve drive basics

The swerve drivetrain uses four swerve drive modules. Each module has a drive motor connected to a wheel and a steering motor to change the direction of the drive wheel. A swerve drive from Swerve Drive Specialties looks like the photograph to the right.

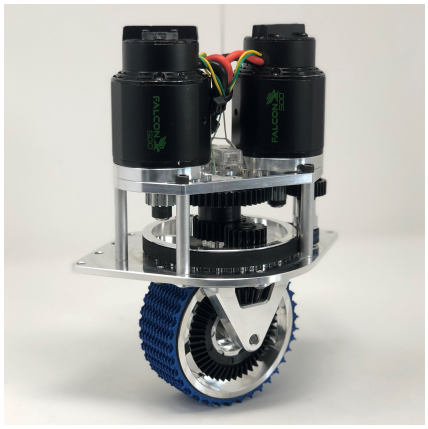


Figure 94: Swerve Drive Specialties MK4 swerve drive module

Figure 95: MK4 swerve drive module emphasizing the drive mechanism.

The drive transmission components are highlighted in the figure to the left. The drive motor is on the left-hand side. It drives a gear train which sends power down the outer shaft of the steering axel to another shaft along the side of the wheel to engage a ring gear on the wheel with a bevel gear. This turns the wheel on its horizontal axis and is independent of where the wheel is pointing.

The steering motor is on the right-hand side. It transmits power to a belt that turns the wheel axle assembly about a vertical axle. The position of the wheel is monitored by a sensor at the top of the wheel axle. The vertical axel has bearings in both the top and bottom mounting plates.

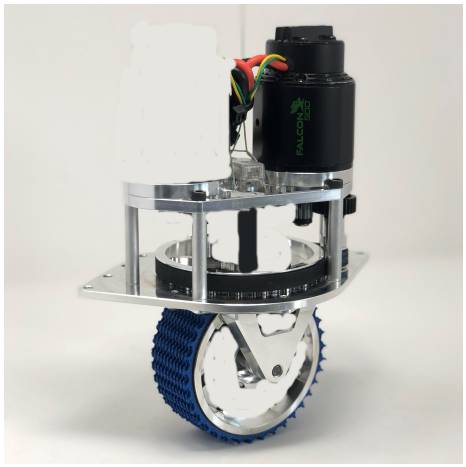


Figure 96: MK4 swerve drive module emphasizing the steering mechanism and inner shaft to shaft encoder.

FRC 4513 used a variation of the MK4, called the MK4i, which inverts the motors for a lower profile. It is shown at the right.

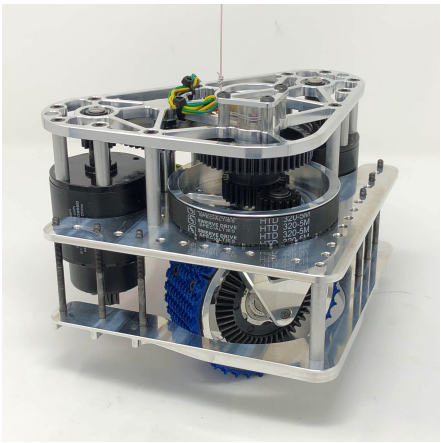


Figure 97: MK4i swerve drive modules with the motors inverted from the Mark 4 modules.

3.4.2 Optional Restricted Hardware Components

Optional components can be used as needed for particular robot designs with some restrictions on the number and supplier (see the rules for the current year competition as they do change over time):

- Only certain motors can be used, they cannot be modified, and the total number is restricted.
- Only certain motor controllers can be used (Talon SRX, Victor SPX, Spark Max…)
- Some motors with integrated speed controllers may be used (Falcon 500, etc.)
- Pneumatics may be used, but
    - There has to be a **pneumatic hub** which provides electrical controls to the other pneumatic components either directly or via CAN bus commands.
    - There can be only one **compressor**, and it must be connected to the pneumatic hub.
    - Solenoids may be employed to pressurize particular pneumatic devices.
    - It must have a **pressure sensor** mechanism on the pressure holding tank connected to the **pneumatic hub** to prevent the system from building up dangerous pressures.
- **Cameras** can be used to stream video back to the drive station. The radio enforces bandwidth restrictions on the overall transmission.

3.4.3 Optional Unrestricted Control Components

The following components can be used in robots without restriction:

- An Ethernet router or switch to distribute Ethernet to various components within the robot like cameras.
- An accessible and dedicated Ethernet jack for tethered operations.
- Limit switches and sensors on elevators, arms, and other moving devices within the robot.
- An inertial measurement unit (IMU) such as the **navX** or **navX2** which connects directly to the roboRIO.
- An independent inertial measurement unit (IMU) coprocessor such as the CTRE **Pigeon** which connects via the CAN bus.
- Using a CTRE **CANivore** to control a separate CAN FD bus to allow faster control communication with drive motors to improve their responsiveness and the accuracy of odometry data. Note that the CAN bus and CAN FD bus are incompatible and will support only compatible devices. This means that if a CAN FD bus is desired, it is used in addition to the base CAN bus.
- There can be any number of coprocessor devices to assist the roboRIO. These have been used for vision processing to locate April Tags or game pieces, providing inertial guidance information, and tighter motor speed controllers. The motor speed controllers may include extra functionality like PID (Proportional Integrated Differential) algorithms, trapezoidal motion controller or Motion Magic.

3.5 Robot Control Architecture

Control of the robot is central to robot competition to ensure the safety of the robots, the competitors and other participants. FRC rules maintain that the match judges have ultimate control over the robot and may deactivate it at any time that they deem that its continued operation is a danger to other robots or participants. To do this the Field Management System or FMS inserts control between the driver station and the robot, so that they can intercept any command at any time and it can disable the robot. This position also allows it to control the periods of the competition: pre-match, autonomous, teleop, and post-match.

- In pre-match they allow the robot to be connected and powered up. The drive teams can communicate with their robots to ensure that communications are established and to select the particular autonomous routine that they want to perform.
- In the autonomous period, the FMS invokes the autonomous method for every robot on the field. This makes all robots start at the same time. The autonomous period continues until it is done for a time determined by the game rules (e.g., 15 seconds).

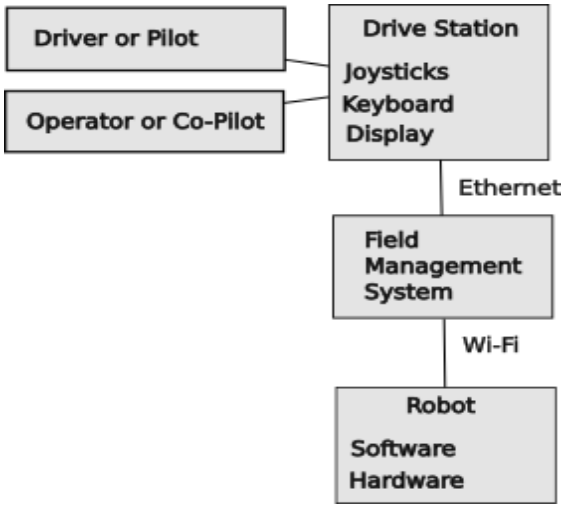


Figure 98: Basic overall control architecture

- After completion of the autonomous period, the teleop period begins and the FMS allows communications from the drive stations allowing the drive teams to fully control the operations of the robot. This period last for a time determined by the game rules (e.g, 130 seconds).
- After completion of the teleop period, power and control is removed from the robot so that the teams can safely remove their robots from the field.

The **drive station** where one or more humans interact with joysticks and other controls to send commands to the robot. They also monitor performance of the robot and take corrective action as necessary. The **driver or pilot** controls the basic movement of the robot about the field. Most teams use a second human called the **operator or co-pilot** who is responsible for the auxiliary functions of the robot.

The robot commands from the **drive station** are required to go through the **Field Management System**. This system does the following things:

- Starts all competing robots at the same time.
- Controls the starting of the autonomous period.
- Controls the starting of the teleop period and the enabling of the drive stations.
- Stops all robots at the end of the teleop period.
- Disables any robot that is exhibiting unsafe operation.

The **robot** is required to comply with a set of requirements to manage the safety of the robots and their accompanying humans at all times. This includes restrictions on the **hardware** components that can be used as well as the specific way that the controlling **software** is written. More details about this can be found in the robot rules for a given season.

The FMS also controls access over the Wi-Fi network. The drive stations are tied to the FMS over a hardwired Ethernet connections. The FMS then sends commands from a team's drive stations to the team's robot using the team number as an identifier. Some encryption is used to prevent additional control stations from controlling the robot. Because of this encryption, the robot Wi-Fi access point must be reprogrammed for each competition.

The control architecture within a robot is shown in the figure below. It includes a local internet network which connects the roboRIO with the radio, auxiliary coprocessors via a network switch. There is also the CAN bus, a serial bus for LEDs, and discrete analog or digital sensor inputs. An optional CAN FD bus may also be included.

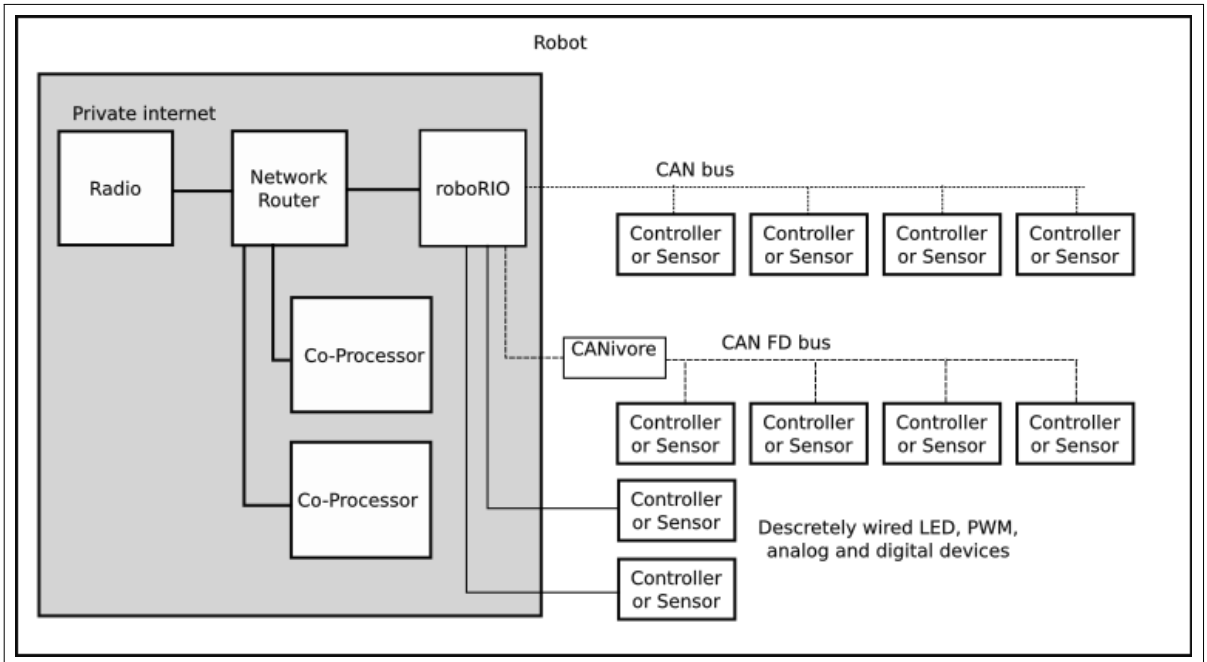


Figure 99: Control Architecture within a robot

3.5.1 CAN Bus

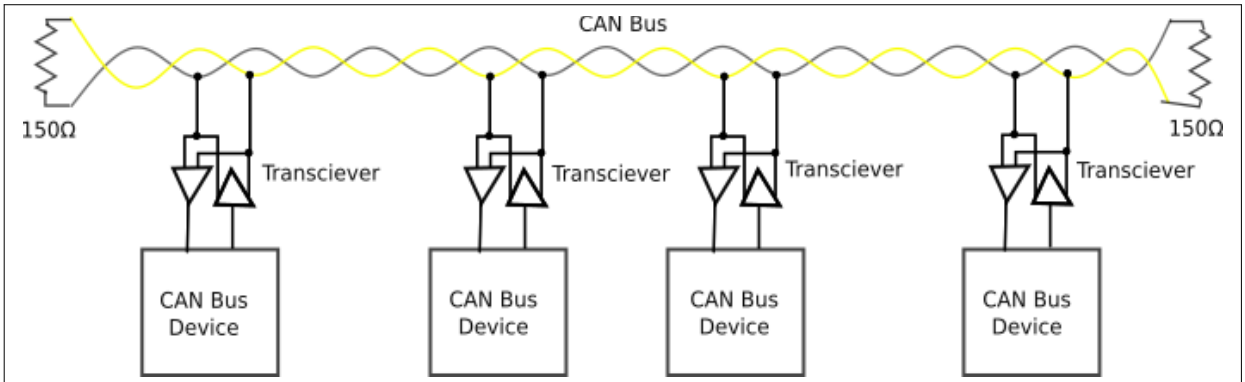


Figure 100: General CAN bus topology

The roboRIO communicates with many of the hardware components over a **C**ontroller **A**rea **N**etwork, more commonly know as the CAN bus. This was initially designed to be a control network for vehicles and has been adopted as a robot communication bus. A bus is a signaling network topology that can have many senders and receivers attached to a single transmission line. The CAN bus is wired as a "daisy chain," where each component is linked to the next component in a chain. There is a CAN bus input and output on every device. The CAN bus is a "transmission line" which is an electrical system for sending

signals without loss or distortion. This type of transmission line uses a twisted pair of wires to prevent it from picking up interference. The transmission line is “terminated” at its ends with a 120 Ω resistor which “eats” the signal to prevent reflections or echos of signals. This termination resistor is built into the roboRIO and the Power Distribution Panel. If these two devices are at the ends of the CAN bus, they can terminate the bus when the built-in terminating resistor is enabled. If these devices are elsewhere in the daisy chain, their built-in resistor must be disabled and a terminating resistor must be added to the actual end(s) of the daisy chain.

In FRC robots the twisted pair is typically a green and yellow wire twisted together. One wire is called CAN high or CAN-H and the other wire is called CAN low or CAN-L. It is important to connect yellow wires to yellow wires and green wires to green wires to keep the polarity straight. The CAN bus signal is differential signal which uses opposite polarity to communicate digital ones and zeros.

Some features of the CAN bus:

- It is a serial protocol meaning that messages are sent as a frame containing a number of bits and each frame is sent one at a time

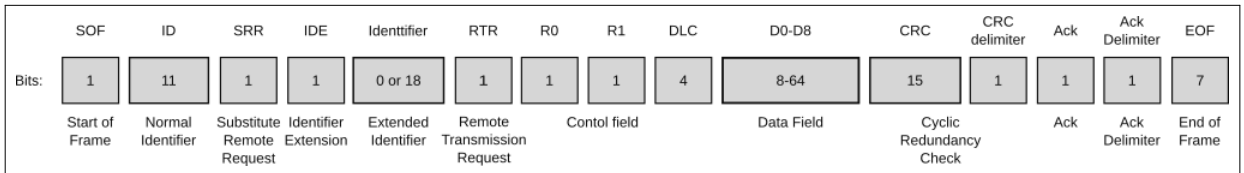


Figure 101: Basic CAN bus frame (RFC robots use extended identifier for the full 29 bits)

- The protocol allows multiple masters (nodes which can initiate a data transfer). This allows bidirectional data transfers.
- It has a bus topology, i.e., it uses a daisy chain with extremely short stubs. There is some discussion about using longer stubs, but this will require that each stub include a termination resistor of a slightly lower value.
- Bits are sent using a differential voltage across a pair of wires. The “dominant” state is driven with the voltage CAN-H > CAN-L. The “passive” state is not driven but pulled by passive resistors to a voltage where CAN-H ≤ CAN-L. The dominate state can override the passive state. Zeros are sent with the dominant state, and ones are sent in the passive state. Since the address bits are sent first, lower numbered addresses have a higher priority than high numbered addresses.
- This differential voltage is fairly immune to electrical noise from motors, because motor noise induces a similar voltage into both leads equally and there is very small differential component.
- Collisions are detected when a node sends a non-dominant “1” value and detects a dominant “0” value in the same time slot. Since the identifier is the first data sent in the protocol, this gives priority to lower identifier values. There should not be conflicts beyond the identifier bits in a properly configured system.
- The CAN data frame has 44 bits of overhead before bit-stuffing and may transfer up to 8 bytes of data.
- The protocol uses bit-stuffing to aid in message synchronization by adding a passive “1” stuff-bit whenever there are 5 active “0” bits sent in a row. Up to 25 stuff bits may be sent in a CAN frame with 8 data bytes. This is a maximum of 133 bits for a frame carrying 8 bytes. The base rate for

the CAN bus is 1Mbps or 1 million bits per second. This works out to 1000 messages every 20ms period of the robot control loop.

- The Data portion of the frame is further encoded using the FRC specification as described in https://docs.wpilib.org/en/stable/docs/software/can-devices/can-addressing.html This leaves 34 bits for devices to use. [Not verified]

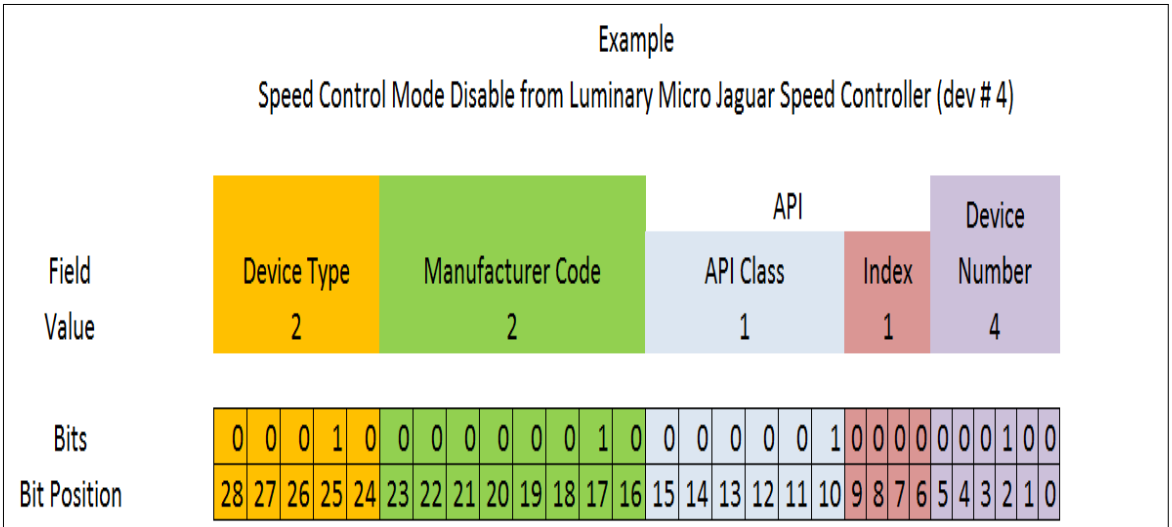


Figure 102: Example of CAN bus ID coding in the data payload (from WPILib docs)

- In the above example:
    - Device type is the type of device
    - Manufacturer code reflects the company who made the device
    - The API is the message identifier, it has a class for certain controls and an index for individual parameters within the API.
    - The device number is the number of a device of a particular type. This should default to zero, although 0x3F may be reserved for device specific broadcast messages.

      Some verification of the identifier field for FRC robots is required.
- Actuators that control motors must ensure that the frame originated from the roboRIO.
- The device number is configured for individual devices on the CAN or CAN FD bus with a configuration tool like the CTR Phoenix tuning tool. It will detect when more than one device is using the same device number. This error must be corrected for everything to work correctly.
- A Universal Heartbeat message is provided by the roboRIO. It used to enable motor controllers. It is normally sent every 20ms. If this message is not seen for 100ms, controllers should assume that the system is disabled. The Universal Heartbeat is encoded as follows:

Table 27: Encoding of the Universal Heartbeat message on the CAN bus.

Description[11]	Byte	Width (bits)
Match time (seconds)	8	8
Match number	6-7	10
Replay number	6	6
Red alliance	5	1
Enabled	5	1
Autonomous mode	5	1
Test mode	5	1
System watchdog[12]	5	1
Tournament type	5	3
Time of day (year)	4	6
Time of day (month)	3-4	4
Time of day (day)	3	5
Time of day (seconds)	2-3	6
Time of day (minutes)	1-2	6
Time of day (hours)	1	5

3.5.2 CAN FD Bus

CAN FD is supported by some devices that can be used with FRC robots. It is like the CAN bus, but has a high speed and can carry a bigger payload. It can use an 11-bit node identifier or the 29-bit extended identifier. Each message can carry a payload of up to 64 bytes (512 bits) of data. CAN FD is supported by the CTRE CANivore device and the CTRE Phoenix Turner configuration tool. The CTRE CANivore can support up to 10Mbps and connects to the roboRIO via its USB port.

The CAN FD bus is not compatible with the CAN bus and some devices use only the CAN bus, so if the CAN FD bus is used, a separate CAN bus must also be used. Both buses are wired with the yellow and green twisted pair wiring. Both buses need to have a 120 Ω terminating resistor at both ends. The CAN FD bus, being high speed is much more sensitive to the required terminating resistor.

One advantage of using the CAN FD bus is that the common motors and shaft encoders for swerve drive can use the CAN FD bus so that the Pose update process can be done more often and with more accuracy.

11 From WPI docs
12 A watch dog is a system which needs to be probed periodically. If the probe is not received, the device should go into a safe mode of operation, like disabling the controller.

4 Robot Software Architecture

Robot software is a complex undertaking. There are many pieces and perspectives on the software and its development process. This section attempts to bring all the pieces together. This section explains how various parts of the robot system work. The intent is to better prepare students by understanding what is going on in the libraries so that they can improve their software and development practices. It certainly does not contain everything you need to know for your team or your understanding, but it provides some details not found elsewhere.

4.1 Design Patterns

Design patterns are a technique used by software developers to reduce the overall complexity of a system. Rather than “reinvent the wheel” each time, the developer uses a familiar pattern over and over to accomplish the task at hand. Once you know the basic pattern for doing something for one instance, it is easy to understand other instances that use the same design pattern. Using the same design pattern also reduces the possibility of introducing bugs, because similar code is written similarly and differences stand out.

There are two fundamental design patterns that are fundamental to robot software. One design pattern is used for defining the software to define and control a subsystem. Another design pattern is used for robot commands.

4.1.1 Subsystem Design Pattern

The functionality of a robot is broken down into a set of subsystems. Each subsystem corresponds to a mechanical feature of the robot, like an arm, an elevator, the drive train, a climbing mechanism, or a throwing mechanism. A subsystem can only do one thing at a time or more specifically process only one command at a time. An elevator can either go up or it can go down. It cannot go up and down at the same time without self-destructing. A general way to think about it is that a subsystem controls a single motor. If an arm has multiple joints, like a shoulder, elbow and wrist, each of these will likely have its own motor and be its own subsystem.

Each subsystem should be independent, so that its operations does not interfere with other subsystems.

Subsystems can be implemented in a hierarchy where one subsystem controls one or more other subsystems. An arm subsystem can control the movements of the individual shoulder, elbow and wrist subsystems. A drive train subsystem can control the movements of the robot with commands to the individual motor subsystems in the drive train.

This hierarchical control scheme can generalize or abstract the movement commands to make simpler commands. For example, you can issue a command to extend or retract the arm, rather than having to issue individual commands to rotate the wrist to 180°, rotate the elbow to 180°, and rotate the shoulder to 90°. Because the detailed movements are defined in a controlling subsystem, then every time the arm is extended, exactly the same set of dependent movements are performed. Additionally, it may be possible to move the shoulder, elbow, and wrist at the same time (provided there are no physical constraints).

Another example is the drive train subsystem which can be written for different types of underlying drive trains, like swerve, tank or Mecanum. The commands to the drive train are the same, but what is does is different.

4.1.1.1 Subsystem Object Pattern

All subsystem objects extend the “SubsystemBase” class to inherit the common attributes and methods of all subsystems. This object is then extended with the attributes and methods that are specific to the particular subsystem. Base definitions may be overridden for subsystem specific controls.

Each subsystem has:

- **Commands** for controlling the movement (or operation) of the subsystem. Each command is a single method does one basic movement, like move to position 3, or go in the direction and speed indicated by the joystick.
- A **default command** which is run when other commands are not invoked.
- An optional **periodic** method which runs every cycle. Some subsystems may work better as a modal operation rather than command driven operation. For example: an elevator may have a number of set positions, so it can be in one of those stable positions or it can be transitioning between positions. The commands only select the desired position and the periodic maintains the elevator position depending on the currently selected position.
- A **telemetry** method that sends its current status information to the drive station dashboard.
- **Configuration** data that defines the fixed attributes of the subsystem, like elevator stop positions, maximum motor velocity, motor control parameters, etc.

4.1.1.2 Subsystem Directory Structure Design Pattern

Team 4513 Circuit Breakers follows the lead proposed by Team 3847 Spectrum in organizing files. The basic philosophy is that form follows function. The WPILib approach was to put all subsystems into a common directory. This gets confusing when you have a lot of subsystems. Team 3847 suggested that each subsystem get its own directory and that each subsystem directory have a set of pre-specified files and directories as another design pattern. This organization makes it easy to remove a subsystem that you had last season, but don’t need this season. It also makes it easy to copy an existing subsystem for that special mechanism that you created this year that is only a slight variation of another subsystem.

A subsystem (XXX) may have the following files (depending on the particular subsystem):

Table 28: Naming conventions for robot subsystem files

Template	Example name	Use
XXXSubSys.java	ArmSubSys.java	Contains the extended class definition for the XXX subsystem. This class is normally instantiated in Robot.java file.
XXXCommands.java	ArmCommands.java	For subsystems with a small number of small commands, all commands may be contained in a single file
commands/yyy.java	commands/ ArmStow.java	For subsystems with a large number of commands, or more complex commands; individual commands are contained in separate files within a command directory.
XXXConfig.java	ArmConfig.java	Contains the configuration data for the XXX subsystem. Hard coding configuration parameters in other source files is discouraged.
XXXMotorConfig.java	ArmFalconConfig.java	Contains configuration data for a particular type of motor used, but specific to this instance.
XXXTelemetry.java	ArmTelemetry.java	Defines the class for the XXX subsystem data attributes used in drive station dashboard and logging.
XXXPeriodic.java	ElevatorPeriodic.java	For subsystems that are more modal than command driven, there may be a periodic method that is called periodically by the main robot loop.

4.1.2 Command Design Pattern

All subsystems may each be running a command at the same time. The command design pattern enables a simple multiprocessing system. The pattern breaks down what any command must do at various phases of its existence. All commands use the same design pattern for their unique phases. The secret is that the processing for each phase has to be very quick. The scheduler executes every command in its queue every 20 ms, so there is not much time for individual processing. It does allow ample time for each subsystem to check on its progress, insert midcourse corrections, etc. While the commands are not literally running at the same time, the command scheduler gives the appearance that they are.

Each command is an object defining a set of methods for the processing for each phase of that command. The names for the phase methods of a command are the same for all commands. The scheduler executes these methods.

- **initialize** is a method to initialize the attributes and states necessary for the command to run. It may set up the conditions to be met for completing the command.
- **execute** is a method invoked to continue the processing of a command every 20 ms. This has a very short duration and must return control to allow other commands to operate. This typically is used to dynamically control a motor within the subsystem.
- **isFinished** is a method that is called after every **execute** method to determine if the command is complete. It returns a Boolean: **true** for finished and **false** for not-finished. If it is finished, the **end** method is invoked. Some commands, like "driveByJoystick" are never finished by an internal criteria, so they would always return **false**.
- **interrupted** is a method to handle a request to interrupt a command in progress and usually invokes the **end** method. If the robot detects that the elevator has exceeded its limit, say in its periodic method, it can interrupt the elevator subsystem command in progress to immediately stop the elevator.

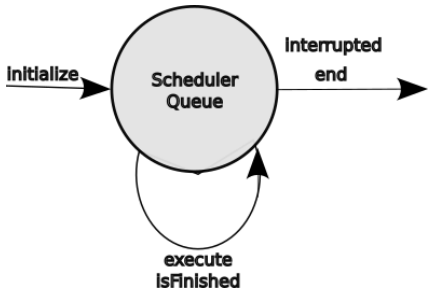


Figure 103: Command state transitions

- **end** is a method to complete a command in progress either during or at the end of its execute cycle. This method brings the subsystem to a stable and known state. It typically stops the associated motor.

A command is defined by instantiating the Command class and overriding the default methods with the methods appropriate for the command being defined. Additionally there are three getter methods for command properties:

- **getRequirements()** define what subsystem(s) the command effects so that the scheduler can enforce the "only one command at a time per system" rule.
- **runsWhenDisabled()**: returns True if the command can run when the robot is disabled. Motors are prevented from running when the robot is disabled.
- **getInterruptionBehavior()**: defines what should happen if another command is started using the same subsystem(s) as this command.

More information is in the WPILib documentation.

4.1.2.1 Command Composition

Some simple one-time commands may be constructed or composed by using classes that populate different parts of a command without having to do all of the details. The Command class itself defaults most of its methods, so instantiating commands do not need to do it themselves. The InstantCommand class accepts a runnable to allow constructing of commands on the fly with lambda expressions. For example the following code implements a new command from part of an old command.

```
Command driveStop = new InstantCommand( () -> driveCommand.end());
```

Command composition allows commands to be strung together in parallel or sequential groups. Controls commands may be inserted in the group to delay execution until a specific condition is met. Similarly a control may keep a group running while a condition is true. Firing a game piece fo a shooter may be dependent on a game piece loaded in the pre-shoot position, the shooter tilted to a specific angle and the shooter spun up to a particular speed. This sort of subsystem interaction and dependence can be handled by the command scheduler. (See 8.7.5 Autonomous Routine Example or 4.1.2.4 Example of Command Compositionfor examples of command composition.)

4.1.2.2 Command Group Classes

WPILib allows commands to be composed with a fairly robust language with the Command class and its methods which may be chained together as decorators. The basic command methods: initialize(), execute(), isFinished(), interrupted() and end() have been discussed previously as part of the command design pattern (see 4.1.2 Command Design Pattern). The compositions are a series of decorators which build new Command objects by adding new methods and attributes to the base. (see https://java-design-patterns.com/patterns/decorator/ for more information about decorators.

A fairly obscure syntax is used to build lists of commands. It is **Command…** which is functionally equivalent to **Command[]**. When used as an argument it can be replaced by an undelimited list of Command objects, which makes it handy for composing a command which is a sequence of commands.

The Command class decorators build a set of new classes with enhanced attributes including:

ParallelGroup(Command…) class for a group of commands that are run as separate command threads)

SequentialGroup(Command…) class for a group of commands to be executed in sequence).

ParallelRaceGroup (Command...) class for a group of commands running in another thread.

RepeatCommand(Command…) class of command or group of commands that is run repeatedly.

WrapperCommand() class for a command or group of commands with a name.

ConditonalCommand(Command…, condition) a command or group of commands that is run if the condition is true.

4.1.2.3 Command Group Methods

The composition methods of the Command class (and associated classes) include the following:

withTimeout() starts a parallel timer to interrupt the base ParallelCommandGroup. Make sure the timeout is removed if the base command completes on time.

until (BooleanSupplier) continue running ParallelCommandGroup while the BooleanSupplier is false.

onlyWhile(BooleanSupplier) continue running ParallelCommandGroup while the BooleanSupplier is true.

beforeStarting(Runnable or **Command...)** adds a method or commands to be run before a SequentialCommandGroup.

andThen(Runnable or **Command...)** adds a method or commands to be run after the the SequentialCommandGroup has finished.

deadlineWith(Command…) adds a ParallelCommandGroup that runs until the calling command completes.

alongWith(Command…) adds a ParallelCommandGroup that runs to completion.

raceWith(Command…) adds a ParallelRaceGroup that runs until the first command completes.

repeatedly() decorates a command or command group to repeat until interrupted.

asProxy() forks a ProxyCommand to not be bound to the current command's subsystem requirements.

unless(BooleanSupplier) runs the command if the BooleanSupplier is false.

onlyIf(BooleanSupplier) runs the command if the BooleanSupplier is true.

ignoringDisable(boolean) overrides the runsWhenDisabled() method to return the boolean.

withInterruptBehavior(InterruptionBehavior) modifies the interrupt behavior of the command or command group.

finallyDo(BooleanConsumer or **Runnable)** runs the specified BooleanConsumer or Runnable after the rest of the command group has executed or interrupted.

handleInterrupt(Runnable) runs the specifed Runnable if the command is interrupted.

schedule() is a method to schedule the current command.

cancel() is a method to cancel the current command without regard to the interrupt behavior.

isScheduled() checks if the command is scheduled.

hasRequirements() checks if the command requires specified subsystems.

getInterruptionBehavior() modifies the interruption behavior of the command.

runsWhenDisabled() returns a boolean to indicate if the command runs when disabled. The command default can be overridden by the ignoringDiable() method.

withName(String) changes the name of the command.

There are many subtleties with command composition, so it would be wise to study the WPILib documentation on the topic as well a reading the library code for various warnings. Of course the library may change between seasons as well. A thorough test of code is also required to ensure that all branches have been covered.

Commands in the command queue may be manipulated.

CommandScheduler.removeComposedCommand(Command) to remove a scheduled command explicitly.

4.1.2.4 Example of Command Composition

The following code is from the Team 2910 2023 code base (any yes, this is more complex than probably 99% of other team's use of command composition).

```
new Trigger(primaryController::getRightBumper)
        .whileTrue(new InstantCommand(
                () -> intakeSubsystem.setTargetPiece(operatorDashboard.getSelectedGamePiece()))
                .andThen(new SequentialCommandGroup(new WaitUntilCommand(() -> Math.abs(drivetrainSubsystem
                        .getPose()
                        .getRotation()
                        .minus(getScoringRotation(true))
                        .getRadians())
                        < SnapToAngleCommand.ALLOWABLE_ANGLE_ERROR)
                .andThen(new InstantCommand(() ->
                        armToPoseCommand.setTargetPoseSupplier(() -> getArmScoringPosition(false))))
                .andThen(armToPoseCommand.alongWith(new WaitUntilCommand(
                        () -> (primaryController.getRightTriggerAxis() > 0.5
                                || ((operatorDashboard.getShouldUseAutoScore()
                                        && drivetrainSubsystem.isGoodToEject())
                                && armSubsystem.atTarget()))))
                .andThen(new SetIntakeCommand(
                        intakeSubsystem,
                        primaryController,
                        () -> IntakeSubsystem.TargetIntakeStates.EJECT,
                        intakeSubsystem::getTargetPiece)
                        .alongWith(new WaitCommand(0.0)
                                .andThen(new InstantCommand(
                                        () -> armToPoseCommand.setTargetPoseSupplier(
                                                () -> getArmScoringPosition(true)))))))))
                .alongWith(new DriveToScoringLocationCommand(
                        drivetrainSubsystem,
                        operatorDashboard,
                        () -> getScoringRotation(true),
                        () -> -adjustJoystickValue(primaryController.getLeftY())
                                * drivetrainSubsystem.getMaxVelocityMetersPerSec(),
                        () -> -adjustJoystickValue(primaryController.getLeftX())
                                * drivetrainSubsystem.getMaxVelocityMetersPerSec()
                                * drivetrainSubsystem.getMaxVelocityMetersPerSec(),
                        primaryController,
                        primaryController::getAButton,
                        primaryController,
                        primaryController::getAButton,
                        () -> primaryController.getRightTriggerAxis() > 0.5,
                        vision,
                        intakeSubsystem,
                        () -> ((operatorDashboard.getShouldUseAutoScore()
                                        && drivetrainSubsystem.isGoodToEject())
                                && armSubsystem.atTarget()),
                        ledSubsystem));
```

Figure 104: Example of a composed command from Team 2910

Another example of command composition is in section 8.7.5 Autonomous Routine Example.

4.2 Subsystem Software Architecture

Most of the code for robots is located in subsystems. Each subsystem controls a device, usually a motor or a motion. At a high level these subsystems can be grouped together in five simple groups:

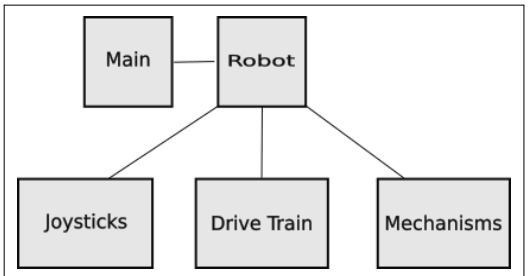


Figure 105: Overall software architecture

- **Main** is the entry point for starting any Java program. It trivially passes control to the Robot Class.
- **Robot** starts up all subsystems in the robot and performs periodic actions for the robot including running the scheduler.
- **Drive Train** is the set of subsystems that control the movement of the robot around the field.
- **Mechanisms** is the set of subsystems that perform various actions for the robot.
- **Joysticks** is the set of subsystems that monitor the driver and operator joystick controllers and turn those triggers into commands that control the drive train and mechanisms.

Breaking the robot down in this fashion isolates the changes needed each season to accommodate various subsystem changes:

- **Main** does not change.
- The **Robot** module is updated with the current set of subsystems in the mechanisms.
- In a perfect world, the **Drive Train** would not change, but teams always want to make improvements for the handling and performance of their robot.
- **Mechanisms** need to be updated with the set of subsystems needed for a season. Many of these changes are cut and paste from an existing subsystem and modifying to reflect the particular subsystem, motors used, etc.
- The **Joystick** triggers are mapped to the Mechanisms and Drive Train commands as desired for the current drive team and robot.

4.2.1 Main Class

The entry point for Java programs is a method called **main** in a static class Main. Java programs are compiled into a block of byte codes. These byte codes are executed by an interpreter on the target machine, in this case the roboRIO. Control is transferred the entry point defined by the **main** method in the **Main** class. In this robot implementation, the main method just passes control to the robot method in the Robot Class.

4.2.2 Robot Class

The Robot Class stands at the top of the robot software implementation. Its main jobs are:

- Create instances of all subsystems.
- Initialize attributes like the **alliance** color and set up driver field of reference.

- Select the particular routine to be run during the **autonomous** period begins.
- Perform periodic actions which include running the scheduler to execute individual commands and log specified items.

4.2.3 Drive Train

The drive train is a hierarchal subsystem that controls the motion of the robot chassis by controlling the underlying drive motors and maintains the position of the robot through odometry and vision subsystem processing.

The drive train abstracts commands to move the robot to two main modes. One mode is called an arcade mode where the robot is moved to a field position based on the direction of the joystick. Push the joystick right, the robot moves to the driver's right. Move the joystick forward, the robot moves away from the driver.

The other mode is robot perspective. The driver has to envision being on the robot. A command to move to the right, moves the robot to its right. A command to move the robot forward moves the robot towards its front. The driver may be assisted in this perspective by a streaming video from the robot pointed forward.

The arcade mode commands are translated into robot perspective commands, so the drive subsystem only is concerned the moving the robot relative to its own frame of reference.

4.2.3.1 Odometry

Odometry has two jobs in the drive train: maintain the yaw or azimuth angle of the robot and keep track of where the robot is on the field.

Odometry is updated by adding up the incremental movements of the wheels to determine their contribution to the movement of the robot chassis, both in its position on the field and its rotation or yaw with respect to the field. This information has a low degree of confidence over a long period of time because it does not account for wheel slippage in the direction of travel and sliding in any direction. Odometry estimates may be updated by vision system estimates of robot position based on the analysis of April tag sightings.

4.2.3.2 Gyro

This object contains the methods for interfacing the gyroscope. Specifically it gets the gyroscope yaw angle and has the ability to reset it to a new angle. There may be different versions of this module to interface different types of gyroscopes (e.g., navX, navX2, and Pigeon).

The yaw angle accuracy drops over time as it drifts and accumulates small errors. The yaw angle can be updated during a match with vision system estimates.

4.2.3.3 Drive Subsystem

Commands to the drive subsystems are from the robot perspective. These commands are a velocity in the x (forward positive) and y-directions (left positive) and a separate target yaw angle (always with respect to the field frame of reference).

The drive train subsystem abstracts the underlying drive subsystem which maybe holonomic for swerve or Mecanum drives or non-holonomic for tank and West Coast drives. All the underlying drive systems

are commanded with the same chassis motion commands and it changes those commands into commands for the individual motors that it controls.

4.2.3.4 Typical Drive Motor Subsystem

The subsystem for any drive motor in any drive train is similar for all drive trains. Drive motors are given commands to request changes to their velocity. Modern drive motors use a dedicated controller that actually controls the motor, so this does not have to be managed directly by robot software. The motor control algorithms are selected and controlled by a set of attributes.

One of the attributes of the motor controllers is braking/non-braking. When a motor is set to braking and the motor is no longer being commanded to move, the windings of the motor are intentionally shorted to produce a back Electromotive Force (EMF) or voltage which counters any movement. This effectively puts the brakes on any movement. When braking is not applied the motor is allowed to free wheel. This is not desirable on elevators which will slam downward when power is not applied. Drive wheels will tend to coast when not braked. We experienced this in practicing our autonomous routine when the robot ran halfway across the field without applied power. Some motors may not require breaking, like the motors on a shooter.

These modern controllers have an additional benefit: they simplify the wiring of a robot. Each motor needs two power leads (on a separate circuit to comply with FRC rules) and a connection to a CAN bus for control and status. Some motors may be connected to a CAN FD bus, separate from the normal CAN bus, to allow more frequent updating of odometry information.

4.2.3.5 Tank Drive Subsystem

The desired chassis movement is specified as v_x, v_y and a target rotation angle in the robot frame of reference. The v_y is the requested forward velocity and the v_x is a requested turning velocity. The turning velocity is added to one side of the drive and subtracted from the other. If v_y is zero, the robot will turn in place. If the robot is moving forward with velocity v_y, one side will be slower than the other causing the robot to turn in that direction. A control of this type may want a non-linear response to the joystick axis controlling v_x.

With tank drive you may want to make a decision to optimize movements. How do you handle pulling back on the joy stick? Is that a command to reverse the robot, or is it a command to change the heading of the robot by 180°. Do you want a separate command to change the yaw by 180° and leave the heading in the same field orientation?

4.2.3.6 West Coast Drive Subsystem

West Coast drives work similar to tank drives. Odometry is complicated by the robot shifting between a front pair and a back pair of wheels. This may be further complicated by the robot physically turning nearer to its center of gravity rather than the center of its wheels. The center of gravity must be close to the center wheels to allow the robot to rock between the front and rear pair of wheels.

4.2.3.7 Mecanum Drive Subsystem

Translate drive train commands into motor speed commands for the four motors to achieve the desired robot motion. Additionally add in the rotational speed vector.

The calculations for Mecanum drives is beyond the math knowledge of the author.

4.2.3.8 Swerve Drive Subsystem

Swerve drive control relies on the vector sum of desired forward velocity vector, specified as $velocity_x$ and $velocity_y$, and a rotation velocity. The forward velocity vector contribution is the same for all wheels. The rotational vector is tangential to the radius of the wheel to the robot center point, so it is different for each wheel. This vector sum results in a velocity and turn angle velocity for each wheel in the swerve drive.

The vector sum is discussed more thoroughly in 2.6.1.2 Vector Addition.

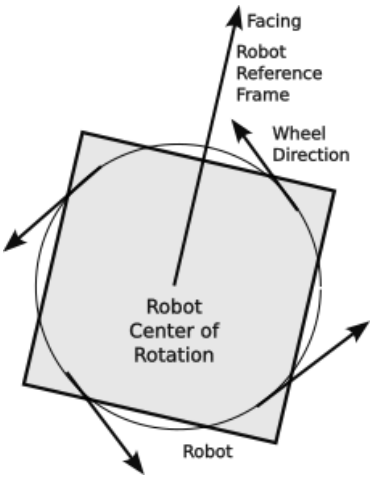


Figure 106: Rotating a swerve drive robot

4.2.3.9 Holonomic Rotation Controller

Rotation of holonomic drives like swerve and Mecanum drives is independent of the requested motion vector. It may be desirable to rotate the robot to align it with a pickup station, scoring target, or to perform an offensive or defensive movement. Since rotation is independent of robot movement, this rotation can happen at any time (barring obstacles).

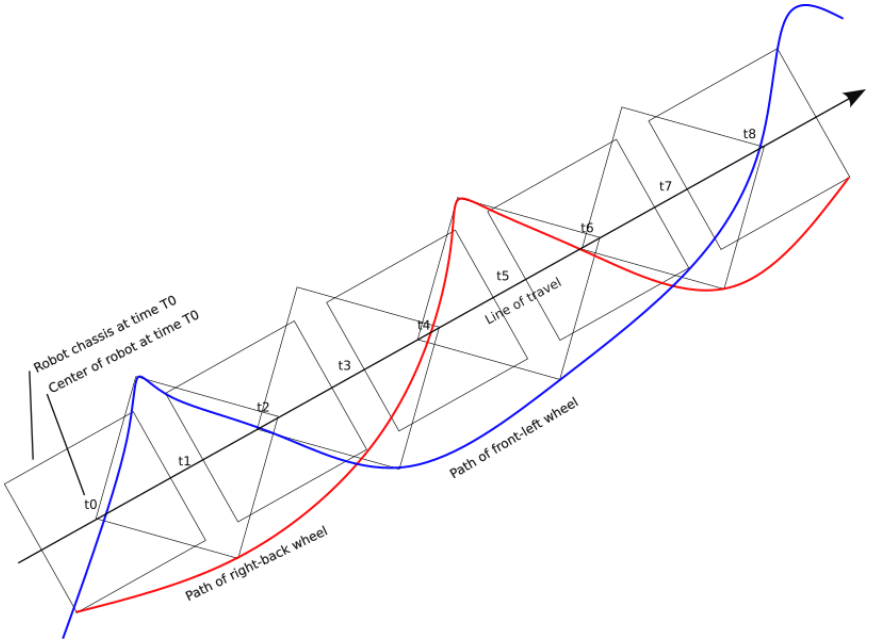


Figure 107: Movement of a swerve drive robot while spinning

There can be commands to snap to specific angles to align with field elements or a command to rotate with a velocity provided by the joystick trigger axes.

To spin the robot while moving requires 1) varying the speed of the drive motor and 2) changing the steering angle of each drive module. This rotation will slow the robot's forward progress. This can be seen in the exaggerated diagram on the right.

The other thing apparent from this drawing is the **aspect** or apparent width of the robot changes with the angle between the heading (angle of travel) and yaw or azimuth. This can be useful in positioning for defense to maximize the covered area.

4.2.3.10 Defensive Spinning of a Swerve Drive

In the figure to the right, a robot is spun while moving forward, but the point of rotation is moved to the upper forward corner of the robot. This allows the robot to move in a straight line with respect to its side. This maneuver may be useful to maintain contact with a robot while moving (t0, t1, …) for each corner is shown independently one corner will stop and the robot rotates about that corner.

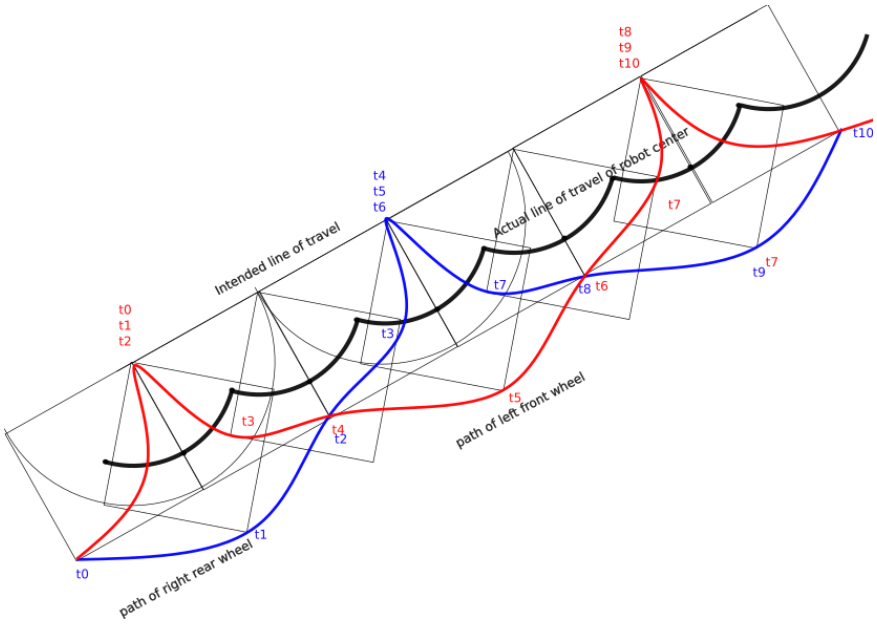


Figure 108: Motion of a spinning swerve drive robot with maintained edge

4.2.3.11 Locking the Wheels on a Swerve Drive

At times, it may be desirable to make a robot harder to move or push. This is done by locking the wheels by turning all wheels to be “tangentially aligned” (i.e., aligned perpendicular to the radial connecting a wheel to the center of the robot). This orientation offers little protection for radial movement of the robot. Radial protection can be give by aligning the wheels radially.

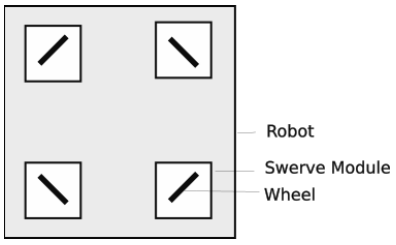


Figure 109: Tangentially Locked Wheels in a Swerve Drive

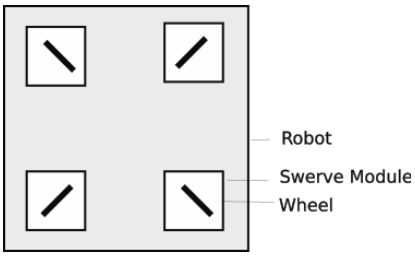


Figure 110: Radially Locked Wheels in a Swerve Drive

4.2.3.12 Swerve Alignment Controller

The swerve alignment controller contains a command for setting the forward angle for all of the swerve modules during an alignment procedure. In this procedure the steering motors are turned off to allow free manual turning, all the drive wheels are then turned to a forward position aligned with the chassis. The command then resets all steering angle encoders to the 0° angle in the robot frame of reference.

4.2.4 Mechanisms

The mechanisms or payload of the robot are all subsystems that control motions other than chassis movements. It also includes any LED displays on the robot. Examples of mechanism subsystems are:

- The **arm** subsystem controls and monitors the position of the arm.
- The **elevator** subsystem controls and monitors the position of the elevator.
- The **intake** subsystem controls and monitors the intake mechanism to load, process and unload game pieces.
- The **leds** subsystem controls the LEDs on the robot.
- The **climber** subsystem controls a mechanism to lift the robot off the floor.
- The **shooter** subsystem is used to propel a game piece toward a target.
- The **tilter** subsystem is used to change the elevation angle of a shooter.

4.2.4.1 LED Subsystem

Team 4513 has moved to a modal system for controlling strings of LEDs (see 2.4.15 Light Emitting Diodes (LEDs)). In this system, the LEDs are in one of a menu of possible modes. There can be modes for setting up, autonomous, teleop, loaded, climbing, etc. Within each mode the LEDs are divided into contiguous groups called sections. Each section can be separately controlled to provide an interesting display like a rainbow, rainbow wave, marquee, the Kit car display, or more boring like a solid color, a flashing color, or a changing color. The color may be specified, or it may be the alliance color. Any of these sections can be conditional upon the status of something with in the robot: elevator at high limit, game piece loaded, game piece ready, aligned, etc. In general the idea is to give the team the tools to use LEDs both to dress up the robot and to provide easily seen heads-up robot status information.

The overall LED periodic must:

- Clear the LED buffer, a staging area for the RGB values for each LED.
- Update the LED buffer for each active section.
- Write the LED buffer to the LED string.

Timing for the changes of the display (like flashing, marquee, etc.) is done by doing modulo arithmetic on a count of routine calls. Using the modulo count lets the pattern repeat smoothly for any repeat period needed. For example: you want a 0.5s flashing light with a 50% duty cycle:

- The periodic method is called every 20ms, so the period of 0.5s is 25 counts.
- Take the remainder of the division of the count by 25 (i.e., count % 5).
- If the result is less than 12 turn it on, else turn it off.

Sections are defined as an **enum** object to give them a name, a beginning LED offset, and an ending LED offset. There can be as many sections as desired, but Team 4513 found it useful to have a section for each physical segment mounted to the robot. Dividing these segments more is fairly hard to see across the field.

4.2.5 Joysticks

Human control of the robot has evolved to use Xbox controllers. These are well known and have plenty of controls at a user's fingertips. During the teleop period, the pilot, and operator uses their respective Xbox controllers to send commands to the robot to move it across the field and to use various mechanisms. Many of the controls are a digital on/off button, but the two joysticks and the two triggers provide for continuous analog inputs in the range -1 to 1.

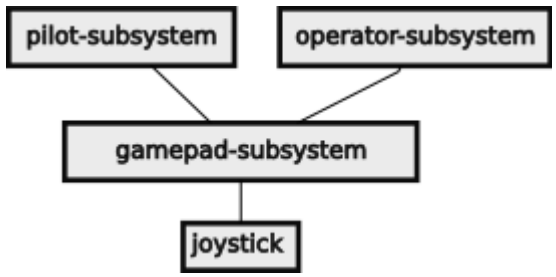


Figure 111: Joystick Controllers

The WPILib, Team 3847 Spectrum and Team 2363 abstract the joystick inputs to use simple declarative programming to bind a control to a command. They also provide filters to transform the raw joystick outputs into values more suited for robot controls:

- Use declarative style programming for binding joystick controls to commands.
- Map a linear analog value onto an exponential curve, spline curve or an arbitrary curve to give finer control at the beginning of movement.
- Provide a dead zone on the joysticks so that nothing is output when the joystick is released and there is a small value present.
- Map a trigger analog value to an on/off event.
- Compose triggers to launch commands based on specific button presses using
  - onTrue(Command) to apply the command when the trigger is first made true.
  - onFalse(Command) to apply the command when the trigger is first made false.
  - whileTrue(Command) to apply the command while the trigger is true.
  - whileFalse(Command) to apply the command while the trigger is false
  - toggleOnTrue(Command) to change whether the command is applied or not on appearance of true.
  - toggleOnFalse(Command) to change whether the command is applied or not on appearance of false.
- Compose triggers using:
  - .and(Trigger) to make a new trigger when the current trigger and another are both true. This is useful for making a button a "shift key" to make more command combinations.
  - .or(Trigger) to make a new trigger when the current trigger or another is true.
  - .negate(Trigger) to make a new trigger that is not the value of the current trigger.
  - .debounce(seconds) to require that the trigger be present for a period of time.

It is important to document the button assignments as they are updated. This is to keep all members of the team that drive or operate the robot up-to-date.

The following table lists the controls available on an Xbox controller:

Table 29: Xbox button names and native ranges

Raw Xbox name	Native range	Name in code	On Xbox controller
Button[0]]	off-on	A_BUTTON	A
Button[1]	off-on	B_BUTTON	B
Button[2]	off-on	X_BUTTON	X
Button[3]	off-on	Y_BUTTON	Y
Button[4]	off-on	LEFT_BUTTON	Left bumper
Button[5]	off-on	RIGHT BUTTON	Right bumper
Button[6]	off-on	VIEW_BUTTON	Overlapped squares
Button[7]	off-on	MENU_BUTTON	Hamburger strips
Button[8]	off-on	XBOX_BUTTON	Xbox logo
Button[9]	off-on	LEFT_JOY_BUTTON	Button in left joystick
Button[10]	off-on	RIGHT_JOY_BUTTON	Button in right joystick
Axis[0]	-1..1, 0 neutral		Left joystick y-axis
Axis[1]	-1..1, 0 neutral		Left joystick x-axis
Axis[2]	-1..1, -1 neutral		Left trigger
Axis[3]	-1..1, 0 neutral		Right joystick y-axis
Axis[4]	-1..1, 0 neutral		Right joystick x-axis
Axis[5]	-1..1, -1 neutral		Right trigger
Hat	[(-1,0,1), (-1,0,1)]	HAT	Hat cross
Hat [-1,0]	off-on	f	
Hat [1,0]	off-on	b	
Hat [0,-1]	off-on	l	
Hat [0,1]	off-on	r	
Hat [-1,1]	off-on	fr	
Hat [-1,-1]	off-on	fl	
Hat [1,1]	off-on	br	
Hat [1,-1]	off-on	bl	

4.3 Software Infrastructure

Software does not exist in a vacuum, especially robot software. It requires a support systems, or infrastructure, to make all the parts work together. Infrastructure includes:

- The operating system on the target machine, also known as the robot.
- Background support services running on the roboRIO.
- Software running on the Drive Station.
- Software running on the Field Management System.
- Embedded Motor controllers.
- Support tools.

This section explains the robot software infrastructure.

4.3.1 Linux Operating System

The roboRIO uses the Linux operating system. This is largely invisible to most robot users, but may be useful in some contexts. It is primarily used to manage the files and processes on the roboRIO. Linux, in general, is not known as a real time operating system. However, the operations on robots are relatively slow (in computer terms), it is possible for Linux to mimic real-time services for the robot operations. Linux also provides a host of services behind the scenes.

4.3.1.1 Linux Core Services

The Linux core, like that used in the roboRIO, provides basic services. It provides a file management system so that files can be written, read and organized so that they can be quickly found. It allows both access to its internal mass storage device, but it can also access USB thumb drives to save images or logging data. It provides process management so that the various tasks being preformed by the computer do not interfere with each other and that critical tasks are given more priority that less critical tasks.

At a low level Linux provides interface drivers for various protocols to allow tasks on the roboRIO to communicate with a wide variety of devices on the robot. These drivers include:

- Wired Ethernet
- Wi-Fi Client services (to initiate access to other Wi-Fi networks)
- Wi-Fi Host services (allows access by drive stations and development systems via Wi-Fi)
- Universal Serial Bus (USB) for thumb drives and a few other devices. Physical access is limited. The thumb drives are used as an alternate destination for log files.
- CAN bus
- CAN FD bus over USB, used by the CTRE CANivore
- SPI bus
- I^2C bus, used by some auxiliary devices like the navX inertial measurement unit
- Ethernet over USB for some USB cameras

Wireless networking and other interface drivers consumes computational resources on the roboRIO. These services provided by Linux in a time-sharing arrangement.

4.3.1.2 Linux Services

Linux offers many services that may be used to support a particular application including some command line functions.

- **DHCP** assigns IP addresses dynamically to devices (like the drive station or a development system) accessing the roboRIO via the Wi-Fi access point or directly via Ethernet. (During competitions, this function is provided by the Field Management System.)
- **ssh** allows remote access over Ethernet to a terminal shell within the roboRIO. This allows a development system to access logs or any installed Linux command line function. The access may be from a terminal window on a Linux or Macintosh systems and PUTTY or a terminal window in Windows machines. You can set up that access to use a public key exchange rather than passwords to be more convenient (ssh-keygen).
- **scp** is a secure copy function that uses an ssh connection to transfer files securely from a terminal window without having to log into the remote system if the ssh keys are set up.
- **rsync** is a secure copy function that only copies the files that have changed. It is useful for copying log files and other data between the roboRIO and the development system.
- **tree** is a Linux command line function for listing the files and subdirectories of a particular directory. This is useful for visualizing the design patterns used in subsystems.

4.3.2 Background Services Running on the roboRIO

4.3.2.1 Basic Robot Operations

- Instantiate the Main Class which in turn instantiates the Robot Class which instantiates all subsystems.
- Use the Linux time services to run the robot-periodic loop to run the Scheduler and other periodic events.

4.3.2.2 Network Tables

Network Tables is a National Instruments implementation of a publish-and-subscribe service[13] on the roboRIO. Publishers send information to the server which forwards the information to subscribers who have previously requested the information. In this way the publishers do not know or care about the consumers of their data, and vice versa.

Data has two fields: topic and value. The topic is a string that identifies the data. It uses a pseudo file-structure-like format to allow hierarchical specification of the data. Classifications are separated with slash characters. So a topic might be “/swerve/right-front-steer/angle” for the right front steering motor angle. The value can be any Java type, like double, int or String. The type of the value is set by the first publisher and cannot change for the life of that topic. The server stores a value of each unique topic (called an instance) and overwrites that value when it receives a new value for a that topic. A subscriber

13 Network Tables is similar to the MQ Telemetry Transport (MQTT) originally developed by IBM. Network Tables is a simpler protocol as it does not support wildcards in the subscribe requests nor quality of service (QoS) transmission controls. MQTT is popular with Internet of Things (IoT) networks.

requesting a topic that already exists, is returned the current value of that topic and any updates to that topic.

Network Tables runs as a WebSockets application which shares the port used for HTTP: Port 80 for unencrypted transactions and Port 443 for secure communications (probably not applicable for robots).

There is much more information about Network Tables in the WPI Documentation. WPI uses Network Tables to populate the Shuffleboard and can be used for primitive logging of data either on the roboRIO or on the Drive Station.

4.3.2.3 WPI Logging

This is the normal logging supplied by the WPILib easily records data sent on the Network Tables or to the Shuffleboard, but may include other data. This should include:

- Robot x, y field coordinates as they change.
- Robot pointing or pose angle as it changes.
- Robot x, y, and rotational (yaw) velocities (although can be derived from the x and y)
- Robot roll and pitch.
- Robot roll and pitch velocities to help detect collisions.
- Robot drive commands as they occur.
- Elevator position as it changes.
- Elevator commands as they occur.
- Arm position as it changes.
- Arm commands as they occur.
- Intake state as it changes.
- Intake commands as they occur.
- Other subsystem states as they change.
- Other subsystem commands as they occur.
- Robot voltage
- Total robot current if known.
- LimeLight, PhotonVision, or other vision processing subsystem info.
- Other telemetry information.

4.3.2.4 AdvantageKit Logging

AdvantageKit logging, developed by Team 6328 Mechanical Advantage, logs all inputs to the robot system and have the capability of replaying the events in a simulation. The interface to the robot is abstracted to give a way to intercept the input signals during normal operations and to substitute the real signals with recorded signals in playback operation. In real-time mode the inputs from the joystick

controllers, encoders, gyroscopes, etc. are sent to the robot and are processed normally. These inputs are also recorded to the AdvantageKit logging. In a playback mode, the inputs come from the log file. Time can be suspended to better understand what happened during a match and to propose changes to fix problems. The fixes can be implemented and deployed to the robot and then tested by replaying the log data again and monitoring that the problem was corrected as anticipated.

AdvantageKit puts some metadata into the log file when it starts up. This includes:

- Date and Time
- Competition information
- Source code used: committed changes and uncommitted changes. This allows the tool to get the specific code used in a particular match for an accurate replay of the log data.

  More information about the meta data can be found at: https://github.com/Mechanical-Advantage/AdvantageKit/blob/main/docs/VERSION-CONTROL.md

They use the CAN FD bus to get finer resolution data from the drive train modules on a separate thread. This thread can run on the second core of the roboRIO to record the data without adverse affecting the normal robot operations.

More information about AdvantageKit may be found on their GitHub site at https://github.com/Mechanical-Advantage/AdvantageKit?tab=readme-ov-file.

4.3.3 Services Running on the Drive Station

- Runs exclusively on the Windows operating system.
- Interconnects joysticks and other controllers used control the robot.
- Dashboard to display the status of the robot to the driver. This display should be large and easy to read.
- Shuffleboard is an optional dashboard with multiple pages that is useful for competitions and for debugging robot behavior during practice sessions. Shuffleboard has two-way communications so it can be used to set parameters in a robot such as PID parameters.
- Network communications with the Field Management System during competition matches via a wired Ethernet connection.
- Network communications with the robot via a wired Ethernet connection during tethered operations or via a Wi-Fi Connection during software development and practice sessions.,

4.3.4 Services Running on the Field Management System

The Field Management System has the following responsibilities for during competition matches.

- Communicate with the Drive Stations of the six teams participating in the match using wired Ethernet connections.
- Pass robot control communications on to the robots using wireless Wi-Fi network connections.
- Ensure that the robots are disabled for field load-in and load-out.

- Ensure that all robots get the enable command at the same time at the beginning of match and get the disable command at the end of the match.
- Allow officials or teams to disable individual robots which may be misbehaving.

4.3.5 Dedicated Motor Controllers

In modern robots, almost all motors have a dedicated controller. This controller off-loads the details of controlling the motor and provides tighter closed loop control system such as PID. The characteristics of these controllers is set up with attributes passed by instantiating an interface or updating it with an interface method.

4.3.6 Support Tools

4.3.6.1 PathPlanner

PathPlanner is a tool that generates the controls for driving the robot over specific paths for autonomous routines. It was developed by Team 3015 Ranger Robotics. The paths have smooth curves for more efficient movement and can be adapted for Red or Blue alliance starting points. In addition to controlling the x-y position of the robot, the rotation of the robot can be controlled to orient the intake mechanism to game pieces on the floor or to align a shooter to a target. The design tool allows the path to be simulated so that its efficiency can be improved.

Dynamic path planning allows automatic controlling the robot from the robot's current position to a known point on the field, like a pickup station or scoring location. This routine is run on the fly, but it requires that the robot has accurate Pose data.

- Tool for laying out path segments for autonomous routines
- Can lay out dynamic paths to reach particular poses on field (e.g., pickup or scoring)
- It uses spline curves like Bézier curves to have smooth path
- It also uses acceleration and jerk profiles to make the motion smooth with controlled velocity and rotation specified along the defined path.
- Can use way points for more complex paths or to avoid fixed field elements and other static obstacles.

For more information visit https://pathplanner.dev/.

4.3.6.2 AdvantageScope

AdvantageScope is a data visualization tool developed by Team 6328 Mechanical Advantage. Raw log data is hard to read and understand. AdvantageScope creates graphs of the data so that you can quickly see variations and make linkages to other events that may be inadvertently causing those changes. AdvantageScope works with WPILib logs or with AdvantageKit logs. It has modes for displaying the robot pose in 2D or 3D field representations and indicating pose updates from vision subsystems.

AdvantageScope was also developed by Team 6328 Mechanical Advantage.

4.4 Software File Organization

A robot has a lot of code files and other files that it uses to defines its operations. Keeping these files organized makes it easier to find particular pieces of the code to fix problems or to add new features.

The file organization is the same for the development system(s) and for the code repository. Code can be developed on devices chosen by the team: Windows, Mac, Chrome, or Linux. The repository holds all versions of a file, so that past versions can be obtained, and the current version has a documented list of changes.

4.4.1 Software Libraries

Libraries allow teams to use software written and debugged by others. This saves development time and allows use of complex computations without having to understand how everything works. A library can be used without understanding exactly how it works inside. Libraries describe their functionality as an Application Program Interface (API) that defines the arguments of a function and what it returns.

Libraries are used to hold code that is stable and (hopefully) not in need of maintenance. These libraries come from several sources:

- Java has libraries that extend the language like Strings, ArrayLists, WrapperClass and some mathematic objects like Point2d, Point3d, Rotation3D, translation2D and translation3d and transform3d.
- WPILib has the basic software for robots including the major design patterns for subsystems and commands.
- Team 6328 Mechanical Advantage has libraries for AdvantageKit Logging and AdvantageScope for visualizing log data.
- Teams may use other vendor libraries to interface particular hardware used on their robot (e.g., for navX IMU, phoenix drivers, pigeon IMU).
- Team may use pure software libraries developed by others (e.g., PathPlanner, PhotonVision, LimeLight).
- Teams may have their own libraries for code which is (relatively) stable between competition seasons for things like telemetry, logging, drive train, etc.

4.4.1.1 Wooster Polytechnic Institute Library for FIRST Robotics or WPILib

Programmers use libraries of previously coded and debugged software so they do not have to recreate all of that functionality themselves. Java comes with libraries for basic input and output functions as well as mathematic functions for square roots and trigonometry as well as to support data types useful for robotics (Pose2d, Pose3d, Transform2d, Transform3d, Translation2d, Translation3d, Rotation, and Rotation3d; see 2.6.4 WPILib Classes for Vector Manipulation). Wooster Polytechnic Institute has written a library (WPILib) for the basic functions of a robot, so teams don't have to write that code. Vendors have added code to that library to interface to the hardware components, again saving us the time of figuring out how these subsystems work at a low level and writing code to control them. (It still does take some time to find and understand the use of the APIs and libraries.)

WPILib defines a robot as a set of subsystems. Each subsystem is largely independent of other subsystems. The majority of the initial programming is to select what software subsystems your robot needs based on the design of the robot. Once you have the subsystems in place, there is binding the joystick controls to the commands that request some action from a subsystem. From there you add code to reflect how you want to change the robot or how you specifically want it to behave.

FRC Programming WPILib documents are at: https://docs.wpilib.org.

4.4.1.2 Coprocessor Libraries

Many coprocessors have their own libraries for their stand alone operating system and application(s). This code is part of the robot code and should be managed as a separate library integrated into the robot library scheme. This allows all the code used in a robot can be managed by version control and a specific version can be replicated to investigate future problems.

4.4.1.3 Team Libraries

Individual teams have developed libraries of their own, such as:

- Team 6328 Mechanical Advantage AdvantageKit for logging everything. This is the current (2024) gold standard of logging because it logs all inputs used for every command. It allows accurate replaying of a recorded practice session or competition match. The software is technically challenging to introduce into an existing system.
- Team 6328 Mechanical Advantage AdvantageScope for visualizing data. They convert a string of numbers into a graph so that you can see just when voltage spikes or drop-outs occur. They also do 2D and 3D visualizations of the robot pose, so you can see how a robot moves through a practice session or even replay a match. This capability has proven useful to try out robot maneuvers while the robot is on blocks, however it may be limited to cases where the gyroscope is not depended upon.
- Team 3847 Spectrum extensions and organization of software modules.

Looking at other teams repositories is a good way to get ideas for your robot and for examples of how others write their robot code.

4.4.2 Software File Structure

Team 4513 Circuit Breakers follows the lead proposed by Team 3847 Spectrum in organizing files. The basic philosophy is that form follows function. The WPILib approach was to put all subsystems into a common directory. This gets confusing when you have a lot of subsystems. Team 3847 suggested that each subsystem get its own directory and that each subsystem directory have a set of pre-specified files and directories as another design pattern. This organization makes it easy to remove a subsystem that you had last season, but don't need this season. It also makes it easy to copy an existing subsystem for that special mechanism that you created this year that is only a slight variation of another subsystem.

4.4.2.1 Robot Project Directory

The directory hierarchy to a specific robot is **/root/project/robot.**

The **root** is the root directory for all robot projects of Team 4513.

A **project** directory in the root directory contains the code for a robot competition season or for special robots like a second robot or for the show bot.

The **robot** directory contains:

- **Main.java** just kick-starts robot.java
- **Robot.java** instantiate all subsystems and provide methods for initializing the robot, running periodic commands
- **RobotConfig.java** configuration parameter for robot components. This includes things like
  - The CAN bus identifier of every controller.
  - Input ports for limit switches and other digital inputs.
  - Input ports for analog sensors
  - Pulse width modulator port for motors and for the LED controller.
- **RobotTelemetry.java** creates instances of subsystem telemetry objects and maps information known to the robot to specific tabs in the Shuffleboard display at the driver station.
- Various subsystem directories contain code to control individual subsystems. See 4.1.1.2 Subsystem Directory Structure Design Pattern for description of subsystem file organization design pattern.

4.4.2.2 File Naming Conventions and File Extensions

The type of a file is indicated by a file extension. File extensions in Linux are a convention to communicate to human programmers unlike the Windows operating system which uses extensions to select the program appropriate for a particular file type. A file extension is a period and few letters at the end of the file name. Some common extensions are:

.bat is a batch file on a Windows system.

.class compiled Java byte codes defining a class.

.cnf a configuration file, usually plain text.

.conf a configuration file, usually plain text.

.dat a data file, usually binary data

.jar Java archive file containing they executable byte codes.

.java Java source file.

.jpg a graphic file in the joint picture group format.

.json general JSON formatted file. JSON is based on JavaScript Object Notation which is loosely based on the JavaScript syntax for integers, floating point numbers, strings, arrays (lists of values) and dictionaries (attribute-value pairs). (See 11 JSON Syntax.)

.md a text file in the mark down format. A mark down file is usually in the GitHub.com version of mark down. This allows for simple formatting of a plain text file.

.path JSON formatted file used by the PathPlanner.

.pdf a stand alone document

.png a graphic file in the portable network graphics format.

.sh a Linux shell program that is also plain text. It is usually a bash shell program. Check the first line for a she bang '#!' to indicate the program that should execute the file.

.txt a plain text file. This is sometimes used to comment out a whole Java file.

.zip a compressed file made by the zip utility.

4.4.2.3 Hidden Files

Files in Linux and Macintosh may be hidden. On Linux such files start with a period. On the Macintosh hidden files normally cannot be viewed. When copying whole directories between systems, these files will also be copied and may not be useful to the target system. They may contain things like a thumbnail of an image file used by the operating system to display when listing the files.

4.4.2.4 Example File Structure

The following is a portion of the Team 4513 file structure to show its organization for subsystems. Comments about individual files is highlighted.

```
~/dev/4513/Robot2023SwerveVer03
├── 3rd Part Library Links.txt
├── Autonomous.png      image of paths for autonomous period for alliance consultations
├── bin                 directory for compiled byte codes
│   └──                 … generated directory that mimics the source code directory
├── build.gradle        build configuration file
├── config.json
├── gradle              directory of gradle files
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── networktables.json
├── pathplanner
│   └── data
│       ├── flutter_assets
│       │   ├── AssetManifest.json
│       │   ├── FontManifest.json
│       │   ├── fonts
│       │   │   └── MaterialIcons-Regular.otf
│       │   ├── images
│       │   │   ├── field22.png
│       │   │   ├── field23.png
│       │   │   └── icon.png
│       │   ├── NOTICES.Z
│       │   ├── packages
│       │   │   ├── cupertino_icons
│       │   │   │   └── assets
```

```
│       │   │   │       └── CupertinoIcons.ttf
│       │   │   └── window_manager
│       │   │       └── images
│       │   │           ├── ic_chrome_close.png
│       │   │           ├── ic_chrome_maximize.png
│       │   │           ├── ic_chrome_minimize.png
│       │   │           └── ic_chrome_unmaximize.png
│       │   └── shaders
│       │       └── ink_sparkle.frag
│       └── icudtl.dat
├── README.md                       Markdown file for GitHub.com repository
├── settings.gradle
├── shuffleboard.json               configuration for Shuffleboard
├── src                             source code directory
│   ├── main                        the real source code directory, ill named
│   │   ├── deploy                  files to be downloaded to roboRIO without changes
│   │   │   ├── example.txt         description of files in this directory
│   │   │   └── pathplanner         directory for PathPlanner files
│   │   │       ├── ...
│   │   └── java                    source code directory for Java code
│   │       └── frc                 source code directory for FRC Java code
│   │           ├── lib             source code directory for WPILIP Java library code
│   │           │   ├── ...         usually one directory per subsystem
│   │           ├── Rmath.java      source code for 4513 robot math library code
│   │           └── robot           source code for 4513 robot
│   │               ├── auto        directory for autonomous period code
│   │               │   ├── AutoConfig.java
│   │               │   ├── Auto.java
│   │               │   └── commands
│   │               │       ├── AutoCmds.java
│   │               │       └── DelayCmd.java
│   │               ├── drivetrain       directory for drive train subsystem code
│   │               │   ├── AlignmentController.java
│   │               │   ├── commands
│   │               │   │   ├── DrivetrainCmds.java
│   │               │   │   └── SwerveDriveCmd.java
│   │               │   ├── config
│   │               │   │   ├── AngleFalconConfig.java
│   │               │   │   ├── CanCoderConfig.java
│   │               │   │   ├── DriveFalconConfig.java
│   │               │   │   └── DrivetrainConfig.java
│   │               │   ├── DrivetrainOdometry.java
│   │               │   ├── DrivetrainSubSys.java      source for Drivetrain subsystem
│   │               │   ├── DrivetrainTelemetry.java
│   │               │   ├── OdometryThread.java
│   │               │   ├── PigeonGyro.java
│   │               │   ├── RotationController.java
│   │               │   └── SwerveModule.java      source for SwerveModule class
│   │               ├── Main.java                  Java source code for Main class
│   │               ├── mechanisms
│   │               │   ├── climber                   directory for Climber subsystem
```

```
│   │                   │   │   ├── ClimberConfig.java
│   │                   │   │   ├── ClimberSubSys.java   source for Climber subsystem
│   │                   │   │   ├── ClimberTelemetry.java
│   │                   │   │   └── commands
│   │                   │   │       └── ClimberCmds.java    source for Climber commands
│   │                   │   ├── others...           directory for other mechanism subsystem
│   │                   ├── RobotConfig.java          source for RobotConfig attributes
│   │                   ├── Robot.java                source for Robot class
│   │                   ├── RobotTelemetry.java       source for RobotTelemetry attributes
│   │                   ├── vision                    source for Vision subsystem
│   │                   │   ├── VisionConfig.java
│   │                   │   ├── VisionPhotonCamera.java
│   │                   │   ├── VisionSubSys.java
│   │                   │   └── VisionTelemetry.java
│   │                   └── XBoxCtrlrs        source for XBoxControllers
│   │                       ├── Layouts.md    human readable controller assignments
│   │                       ├── operator
│   │                       │   ├── commands
│   │                       │   │   ├── ArmElevComboMoveCmds2.txt
│   │                       │   │   ├── ArmElevDriveCmd.txt
│   │                       │   │   └── OperatorGamepadCmds.java
│   │                       │   ├── OperatorGamepadConfig.java
│   │                       │   ├── OperatorGamepad.java
│   │                       │   └── OperatorGamepadTelemetry.java
│   │                       └── pilot
│   │                           ├── commands
│   │                           │   └── PilotGamepadCmds.java
│   │                           ├── PilotGamepadConfig.java
│   │                           ├── PilotGamepad.java
│   │                           └── PilotGamepadTelemetry.java
```

4.4.3 Code Repository

A code repository simplifies code development as it allows team members access to the code files, whether they are reading the files or developing new code. Most modern repositories also include some form of **source code management** to keep track of files changes: why was the file changed, who changed the file, when was the file changed, what other files were changed with this file. A common source code management system is Git, and it is discussed more in the software development section of this document or online. Git can be used in a stand-alone mode on a development system, or it can work as part of a repository check-out and check-in system.

FRC rules require that any code that is used from one season to the next be placed in a public repository before the start of a season. This means that anyone can access code on that repository. It is probably a good idea to keep current development in a private repository and change its access to public at the beginning of a new season.

Two examples of public code repositories are: GitHub.com and GitLab.com.

5 Software Development Process

5.1 Software Development Steps

- Study: understand how the existing code works
- Plan: on the changes to be made (see next section on strategies)
- Design: figure out the major design changes to subsystems, commands, etc.
- Write the code:
  - Edit.
  - Compile cleanly.
  - Deploy.
  - Debug to find bugs or code that does not work as expected. This may involve adding some temporary code to print out values or progress. Studying the results to infer why things are working they way they are and develop a fix to make things work as expected. Some bugs can be found by analyzing the log data kept by the robot.
  - Loop back to writing
- Accept the code when it works as expected (and the documentation is done)

5.2 Project Management

Most teams suffer from poor time management. There is not enough time do everything that you want or need to do. In the work world there is Project Management to keep a project on task and on schedule. It does take some effort to set up and manage. It is worthless if it is not followed. But if it used, it can detect problems early on so that they can be addressed. Basically if a team elects to do project management, it requires a buy in of the entire team and persistent follow through to keep the tasks on schedule (or adapt when something runs into unanticipated difficulties).

5.2.1 Definition Phase

In the definition phase a team needs to identify a set of tasks that it wants or needs to accomplish. Each task has the following attributes:

Name	Name of the task.
Description	What does the task entail including deliverables which must be completed by the end of this task.
Duration	How many days are required to complete this task.
People	How many people will this task consume. If this is a variable number then think about breaking down the task into its constituent sub-tasks. Identifying specific individuals is important to identify dependencies. For the most part the tasks break down into the responsible teams.

Dependencies What deliverables does this task require to start its work. What skills are required to complete this task.

Deliverables What things or deliverables must be completed by this task. Any rework on a deliverable delays the completion of the task. Being sloppy here will likely mean downstream delays because it set expectations about what is available and when.

A Gantt chart shows the interrelationships between tasks and resources on a timescale. For a build season, there are only 7 weeks.

Another thing to be documented is what days is the team going to work and what days are they going to rest. No one should be expected to work long days or every day. Keep in mind that build season includes some school holidays and may include in-service days which students treat as a holiday. In northern climates anticipate a number of snow days randomly appear in your schedule. It is better to plan for them than be surprised by their delaying effect.

A team needs to build a set of tasks and milestones. For typical build season the starting milestone is the game reveal in January and the ending milestone is the first competition. Tasks may include things like:

- Get the new game rules.
- Decide on what the robot needs to do by the first competition.
- Decide on “what” the robot needs to do by the second competition.
- Decide “how” the robot will implement the “what” decided above.
- Test concepts for new mechanisms.
- Complete CAD for mechanisms.
- Complete CAD for whole robot.
- Build the robot mechanical systems.
- Build the fixed robot electrical systems.
- Wire the electrical components of the robot.
- Commission the robot.
- Test the robot functionality.
- Build the field elements.
- Get the game pieces.
- Train a drive team on the robot.
- Test new software on a second robot without all mechanisms.
- Test new software on the competition robot.

You need to keep this information in an easy to read and understand format. Some in the industry use Gantt charts to keep track of project management. The mermaid graphing tool https://mermaid.js.org/syntax/gantt.html can produce simple Gantt charts that are sufficient for the build season. The source code for the above looks like:

```
1.  gantt
2.  title Team 4513 2025 Project Management
```

```
3.  dateFormat YYYY-MM-DD
4.  excludes Friday Sunday
5.
6.  section Base design
7.  Rules released :bas1, 2025-01-04, 1d
8.  Decide what :bas2, after bas1 , 1d
9.  Decide how :bas3, after bas2 , 5d
10.
11. section Mechanical
12. mechanical CAD : mech1, after bas3 , 5d
13. robot CAD : mech2, after mech1, 5d
14. build base robot : mech3, after mech2, 3d
15. build mechanisms : mech4, after mech3, 4d
16.
17. section Electrical
18. place components :ele4, after mech4, 2d
19. wire robot :ele5, after ele4, 5d
20.
21. section Field
22. build target :field1, after bas1, 5d
23. built load station : field2, after field1, 5d
24. build end station : field3, after field2, 5d
25.
26. section Software
27. setup library and IDEs : soft1, after bas1,5d
28. setup mechanism folders : soft2, after bas2, 4d
29. write new code : soft3, after soft2, 10d
30. test code : soft4, after soft3, 5d
31. commission robot : soft5, after ele5 soft4, 1d
32. test new robot code : soft6, after soft5, 2d
33. test robot with driver : soft7, after soft6, 5d
34. new joystick commands : soft8, after soft6, 5d
35.
36. section Salem Competition
37. load : comp1, 2025-02-26, 1d
38. travel : comp2, after comp1, 1d
39. compete : comp3, after comp2, 2d
```

Comments:

1. tells mermaid what sort of diagram to produce

2-4 set up the title, date format and what days are excluded

6, 11, 17, 21, 26, 36 define the sections reflecting the various sub-teams

The rest are tasks formulated one per line as: task words ":" name "," start date or "after" dependency "," end date or duration

Mermaid produces the diagram on the next page:

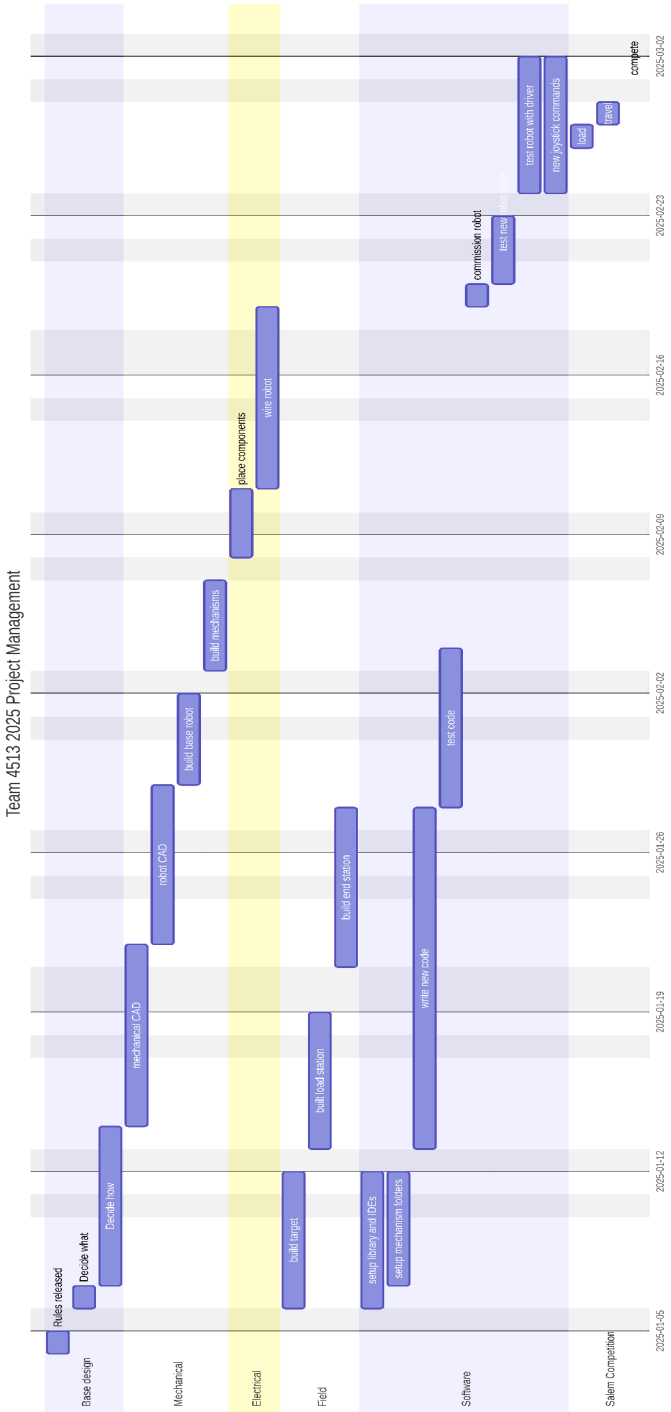


Figure 112: Gantt chart for a simple build season

5.2.2 Management Phase

During the build season the designated project manager must query each subteam to ascertain progress on the assigned tasks. This must be done every week and probably should be done on a consistent day, like Monday or Friday. The manager can mark tasks complete or work with the teams that are falling behind to plot a path forward to get back on schedule with the least impact to down stream tasks. This is the hard part of project management. The goal is to keep the project on track and there has to be some give and take because no matter how good the initial plan was, Inevitably there will be unforeseen problems during the project.

5.3 Software Development Strategies

Try to work incrementally.

- Do small steps with minimal changes to the existing code base.
- Small changes are easier to test and debug.
- Small changes are easier to understand.
- Small changes have smaller risk.
- Quicker cycles can be more rewarding and less frustrating.
- Document the changes made with a meaningful comment when the code is committed. “Yesterday’s changes” only means something in brief instant of time. “Corrected motor shuttering” identifies the problem being fixed.
- Complete each step before moving to the next step.
- Large steps inherently have a lot of problems and risk both the base and the schedule.

This is also known as agile or cascaded software development.

5.4 Software Development Tool Chain

A tool chain is the set of tools used to develop software programs. It can be as simple as a run time environment that allows entry and direct running of a programming. Such an environment limits the complexity of the code that can be developed. Robotics code requires a more complex environment.

5.5 Integrated Development Environment

WPI recommends the use of Visual Studio Code for code development. It integrated various software development tools which can be accessed from the same human interface.

- Visual Studio Code is an “Integrated Development Environment” or IDE which includes access to several software tools from a single window.
  - An editor for entering the text of the program. This is flexible allowing various key maps to simulate may editors, including a Linux favorite for 50 years, **vi or vim**.
  - A syntax checker to point out syntax errors in near real time as you edit the program.
  - An easy interface to compile all of the modules of a program to build the Java .jar file for deployment

    - An easy interface to deploy the compiled code to a target robot.

- WPI integration with Visual Studio Code keeps different versions based on different libraries separate. The WPI libraries change each season. The libraries and source code for a robot for a year are kept together, so that they will compile and simultaneously are kept in separate trees. This allows code from past year(s) be run at any time to check if a bug existed in a prior year. It allows the current code to use the current library.
- Compiler converts the Java source code into Java byte codes and eventually into a Java archive or .jar file
- Gradle… automates the build (compiling, linking, and building load files consistently) and download files to the target robot.
    - Link in the vendor dependencies.
    - Build the robot.jar file with all the compiled Java byte codes.
    - Include the PathPlaner .json files.
    - Include any other files needed by the robot.
- deploy. Done by Gradle in the preceding step.

5.6 Source Code Editing

Source code is written on an editor or integrated development environment. The later includes many tools so that a developer can see the file structure, code and other tools at the same time and use the same interface to do many tasks.

5.7 Source Code Management (Git)

A source code management system keeps track of the changes to the software over time. It allows multiple people to work on the same code base at the same time and provides tools for resolving conflicts when two people change the same code module. It allows an unsuccessful or experimental code change to be backed out. If used properly it can keep track of who made what changes and why the changes were made. It also helps in resolving the conflicts when two people modify the same file during the same period of time. A remote repository such as GitHub.com or GitLab.com to allow others to access or, if trusted, to modify the code repository.

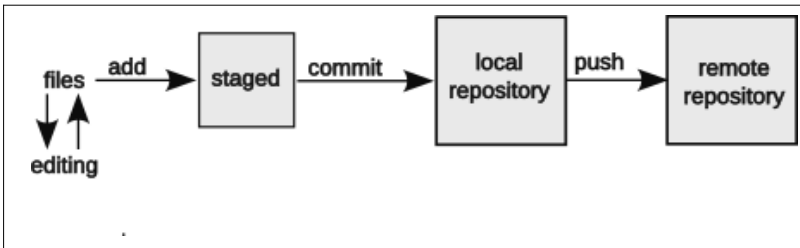


Figure 113: Git source code management

Basic code management has the following phases:

- editing where files are created and modified. Files are edited on the development system. Files may be deployed to the target system during this phase as part of testing.

- staging where modified files are **add**ed to a group of files to be included in a committed version. A version includes many files. The s**tatus** command an be used to list the files which have been modified as well as the files staged for commit. Staging is done on the development system.
- The **commit** command saves a copy of the staged files as a version into the repository, so that the set of files can be reproduced should future changes do not achieve their objective or to retrieve the current version of files. It. also changes the state of the staged files to unchanged for future staging. In general, commits should cover as small of a set of changes as possible to give granularity to the change and to implement a particular function. A commit is done on the system managing the repository which may be local or remote to the development machine.
- The **push** command uploads one or more commits to a remote repository. While local repository are possible and useful in some cases, a remote repository allow all members of the coding and review teams to access the latest software.
- (A **pull** request is a change outside of the normal source code control change. This lets outside developers propose changes to fix a particular problem or problem. This probably will not be used in robot code development, except to propose changes to the external code libraries.)

5.8 Debugging

All code that is written must be tested to ensure that it works as the programmer intended. In the course of that testing, it is inevitable that anomalies or bugs will be found. These bugs need to be corrected so that the program works as it is intended to do. Never assume that a change will work the way that you think that it will.[14] Test it and make sure the code is correct.

This takes a skill to problem solve. First recognize that there is a problem. This usually manifests itself as a set of symptoms (e.g., sluggish, wheels chatter, elevate move uncontrolled, wrong or unexpected message on the drive station). For each symptom, find the root cause and fix it. Don't just try to fix the symptom, but get down to the underlying root problem. If you don't do this, the problem will keep on biting you. Of course this is a symptom that you haven't dug deep enough to find the root cause.

The Drive Station can access the roboRIO log files in real time and once the Drive Station has access; a window on the Visual Studio Code system can be opened to view data. AdvantageScope may also be accessed on the development system.

5.9 Simulation

Simulation is a way to run the code without having to load it on the physical robot hardware. In simulation mode the various physical subsystems are replaced by software that acts the same way as the hardware that it represents. While simulation my not be perfect, it does offer a way to test software before hardware is ready or in use by other team members.

As of the 2024 season, swerve drives were not simulated. An approximation to simulation was to use the AdvantageScope to monitor a robot's movement while the robot was on blocks. This worked unless the yaw of the robot needed to change.

14 See Murphy's Law in 9.7 Just Saying…

5.10 Deploying Code to the Robot

The WPILib/VS Code environment has a method for deploying code to the robot. This looks for an attached robot and interacts with it to download the robot control code. After the download is complete, the new program is immediately executed.

5.11 Log Analysis

Logs kept by the robot allow for analysis of the performance of the robot after a practice session, match or competition. The goal of this analysis is to uncover the underlying problem and to help formulate the corrections to alleviate such problems in the future. Keeping robot logs without analyzing them is pretty much a waste of time, effort and money.

- Logging
  - Log Shuffleboard data.
    - Display data on the Drive Station.
    - Mostly used for debugging.
    - Some logs could be analyzed during competitions to find areas of improvement, but remember the drive team has a limited bandwidth.
    - May log activity on the Network Tables.
- AdvantageScope
  - Visualization of logged data. This takes a table of numbers and displays it as a graph so you can see when and where the voltage dips or a current spikes. Looking at other recorded data, you can search for the culprit.
  - Read odometry data to display on a two-dimensional or three-dimensional display.
  - Show data in real time.
  - Show current usage by the robot overall or individual motors.
  - Show temperature readings.
- AdvantageKit logging
  - log “everything”.
  - All inputs to processes pass through an abstract “interface”.
  - Possible to replay a match using the real inputs.
  - Keep track of the software version.
    - Know the current version of the code as known to git or other source code management system.
    - Know the versions of code that has been deployed event that that code has not been staged or committed.

- This information is used in looking for bugs in postmortem situations and things did not go as expected. It is important that you are looking in the actual code that was running at the time of the failure.

5.12 Carry Over and Re-Use

An important part of software development is to reuse software from past seasons. FRC rules allow you to reuse code from prior seasons as long as it is posted to a public repository **BEFORE** the season begins. This is an opportunity that you should not miss out on. Also, it is a great resource to see how other teams solve problems and perhaps use some of their code in your software.

6 Computer Science Topics

Java programming uses a lot of Computer Science jargon and concepts. This section briefly explains some of those concepts.

6.1 Types of Computer Languages

Computer languages come in types based upon how the computer handles the language:

Machine language is the codes used within the processing unit. This code is the closest you can get to the machine and can be the fastest possible code. This code is dependent on the processing unit on which it runs.

Assembly language is a human-readable form of machine language using mnemonic codes and simple arguments. Like machine language, assembly language is dependent on the processing unit on which it runs.

Compiled language is a human-readable language that is compiled into a machine-readable language, usually machine language. An example of a compiled language is C or C++.

Interpreted language is a human-readable language which can be indistinguishable from a compiled language (or a compiled form of the interpreted language). The difference is that the interpreter language is interpreted at execution time. This language form is typically the slowest of all language forms. An example of an interpreted language is the JavaScript code that runs in most browsers, Python or Basic.

Tokenized language is a human-readable language which is compiled into an intermediate machine-readable form, which in turn is interpreted at run time. This is normally quite a bit faster than pure interpreted language because the language elements do not have to be interpreted and analyzed. There must be a run time environment on the target machine to interpret the tokenized code. An example of a tokenized language is Java.

Graphical languages are defined in terms of graphic blocks that are arranged to form programs. Because each block is a complete statement, most syntax errors are eliminated. The graphics are saved either as tokenized code which can be interpreted on the target machine or compiled and deployed to the target machine.

It is possible to build a compiler for a normally interpreted language. It is also possible to write a interpreter for a normally compiled language, such as some websites that use JavaScript interpreters for other languages including Java.

6.2 Language Orientation

High level languages in general have different orientations which is the mind set used when writing code in such languages. Some of the orientations are:

Procedurally-oriented languages use procedures to process inputs. Examples are C, COBOL, BASIC and most graphical languages. C introduced structs (structures) to build composite data types of primitive data types. This allowed procedural code to operate on arrays of structs that represented separate, distinct, and similar objects.

Functionally-oriented languages focus on functions that transform input values into other more useful values. Examples are FORTH, Clojure, LISP.

State-oriented languages centers on managing the state of a set of elements. An example is Erlang (used for telephone switching and some protocol processing).

Object-oriented languages center on objects with inherited characteristics from classes. Examples are SmallTalk and Java.

Hybrid languages allow a mix of object-oriented and procedural-oriented styles. Examples are C#, C++, Python and JavaScript. Python is largely object-oriented, but is not "pure", because it allows methods outside of classes to support procedurally oriented code.

Declarative languages define the characteristics of a set of elements. These definitions may be hierarchal. Examples are HTML, CSS, and JSON. WPILib uses declarative styled code to bind commands to joystick inputs and to build commands.

6.3 FRC Language Selection

FRC robots are supported by a software library called WPILib. It was developed by the Worcester Polytechnic Institute (WPI) and it has libraries for most of the major components of a FRC robot. The WPILib supports three languages: National Instruments LabView (a procedurally oriented graphical programming language), Java (a modern object-oriented language), and C++ (a version of the traditional C language with extensions to support object-oriented programming). FRC 4513 chose Java because it is a textual language, is widely supported, and is not as complex to teach as C++.

6.4 Strongly Typed Languages vs. Auto-Typed Languages

All data elements have a type. Some languages like JavaScript and Python are "auto-typed" meaning that the type of the data element is derived from its context and usage. The type of an element may change during the life of the program. For example "add" a number to a string and the number is automatically converted to a string and the result is concatenated to the first string. This makes for easy-to-use friendly languages.

Other languages are "strongly typed" which means that the type of a variable must be declared before it is used or even initialized. Every use of the variable is checked to see if it matches the type that is expected for that variable. Adding a number to a string doesn't make sense because the two data types are different, The number must be converted to a string and then the two strings can be concatenated. These languages feature a "type casting" mechanism to explicitly convert one type to another. Java is such a language. It seems to go one step further in converting a float to an integer, one must first convert the float to a whole number through a rounding, floor, or ceiling function before the casting can take place.

For most of us who have experienced auto-typed languages, strongly type languages are harder to use as they generate a lot of error messages just getting code compiled. On the other hand the strongly typed languages promise less buggy code that is more robust and reliable.

6.5 Stacks, Queues, and Deques

Stacks, queues, and deques are special data structures used to hold information. Data is stored in a stack by "pushing" it onto the stack and that data is later retrieved by "popping" it off the stack. The stack is called a LIFO or Last-In-First-Out mechanisms. Queues hold data in the order that it was stored. They use a First-In-First-Out or FIFO mechanism. Queues are useful for holding task objects waiting to be executed. There are some data structures that are a hybrid where data can be stored to either end or removed from either end. This double ended queue is also called a deque (pronounced like a deck of cards).

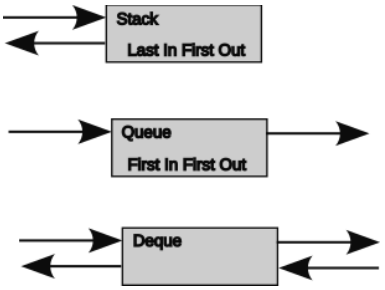


Figure 114: Stack, queue, deque

6.6 Garbage Collection and Memory Leaks

No, garbage collection in software isn't the trucks that come by your home or school to pick up the trash bins. This is a background process in modern languages to manage memory. When an object is created, memory is used to hold attributes of that object. When the object is no longer needed, that memory can be released and reused within the program. When a method is invoked, the method arguments are pushed onto the stack. Variables created by the method are also pushed on the stack. The method can access the arguments and the variables while it is running, but when the method returns control; the stack is popped and the memory is released. That is the easy part.

The harder part are the object attributes. These live as long as the object lives, which is determined if the object is referenced somewhere in the program. The problem is that the memory used by objects is a chunk here and and a chunk there. As objects come and go, memory becomes fragmented into smaller and smaller memory chunks. To make sure there are enough big chunks, a process called garbage collection, moves the active chunks into a contiguous area thereby causing the unused chucks to form a bigger contiguous pool. Garbage collection takes processing time. The good news is that the process can be incremental so that it can work a little at a time as long as the data copying is an atomic action. An atomic action is one that is completed in its entirety without interruption. The atomic action guarantees the integrity of the object attributes and prevents the program from writing attribute updates into an abandoned object store or accessing attributes from a partially written object store. Garbage collection is part of the language and is not really under the control of the programmer, even though there may be directives to give that impression. Garbage collection does affect execution time.

A memory leak is memory usage over time. These are caused by abandoning an object and its attributes without the language noticing. So that as a program runs, it uses more and more memory until it uses all memory allocated to that program. The program has to be violently put out of its misery. Obviously for a reliable system, memory leaks are bad. Testing of a program should include monitoring of memory usage over time to detect memory leaks.

6.7 Methods, Functions, Procedures, and Subroutines

In general methods, functions, procedures, and subroutines are very similar. All are a group of statements that are invoked or called to perform a task. Usually this is a task that is performed repeatedly. All may be passed zero or more parameters as input arguments. Not all tasks require an argument. A function returns a value which might be mathematical, like return the square root of an input argument. Procedures and subroutines usually do not return any values. A method is similar in that it performs a task, but it operates within the scope of a class or an object. It can use and manipulate attributes of that object. It can return

one or more values to the invoking code like a function. Java has only methods tied to a class or object, although some of its methods are also functions.

6.8 Variables, Constants, Attributes, Parameters, and Arguments

Values are stored and passed in named data elements. In Java there are:

- A **variable** is a temporary-declared value whose value can be changed by a method. It lives only as long as the method executes. A variable can only be accessed in the method or block scope in which it is defined.
- An **attribute** is a value tied to an object or its defining class.
- A **parameter** is a value passed to a method, function, procedure, or subroutine. This value may or may not have a name as it can be an expression or a literal value. The type of parameter must match the type of the corresponding argument of the method being invoked.
- An **argument** is a defined value with a type and is an input to a method. For example, let's say you have a function that returns the square root of the value *x*. The argument of the function is x. When the function is invoked, it is passed a parameter value that is set to the argument x. The value of could be anything: a literal, the result of an expression, a method variable, or another named value. Arguments in Java are positional, meaning that a value in a particular position within the method parentheses is treated a particular way. Java does not have a built-in way to name parameters to associate them with a specific argument.
- A **constant** is a variable or attribute whose value cannot be changed.

6.9 Scope and Encapsulation

Scope is where a variable, method, or attribute can be accessed.

Encapsulation allows for a variable name to be reused in different parts of a program without conflict.

Java provides a couple of mechanisms to limit scope and encourage encapsulation. Variables can be accessed only within the scope in which they are defined which is limited to the defining method or block. Iteration variables associated with a for statement, can only be accessed within that statement.

An attribute or method can be marked with the **private** modifier, which means it can only be accessed within the defining class or its instantiated objects. Conversely, an attribute of method can be marked with a **public** modifier, allowing it to be accessed anywhere. The **member selection** operator must be used when accessing outside the defining class or object to ensure that the reference is to the right static class or object instance.

6.10 Pass By Value

Pass by Value is when a passing an argument to a method, only the value is passed. Changes to the argument internally to the method do not affect the value of the original parameter in the calling method. For example:

```
int fum (int fumIn) {
      fumIn = fumIn + 1;
      return fumIn
```

```
}

int foo = 2;
fe = fum (foo);
// fe is 3
// foo still is 2
// fumIn no longer exists, the scope of fum no longer exists
```

In Java when an object is passed as an argument, it is passed by value. Its value is copied, not a pointer or reference to the object.

6.11 Pass By Reference

Pass by Reference is when passing an argument to a method, the pointer or reference to the argument is passed. If the called method modifies the passed argument, the value of the original argument is also modified.

Java normally does can pass by reference, although there are cases where it does or at least simulates it. Lambda expressions are copied and not evaluated when encountered. The invoked method can save the expression as a Runnable class of object. This evaluation is delayed until some point internal to the called method. This delayed evaluation may include the invocation of a method, which effectively treats it as a passed pointer. The evaluation also produces a value which can be used immediately or saved. Lambda expressions themselves cannot not modify the accessed method or attribute, although it may be possible by passing the delayed lambda expression a value that is "evaluated" by invoking a setter method with the argument. (See also 7.18 Lambda (or Anonymous) Expressions.)

6.12 Modularization

Modularization allows a program to be broken up into several files for ease of maintenance. A program that needs the code of another file may **import** that file. By convention each class is defined in its own file. For robots that means every subsystem has its own file, but there are separate files for commands, configuration parameters and telemetry data for each subsystem. Using a directory for each subsystem increases the modularity of the subsystem.

A **package** can be created by grouping a set of class definitions in a common directory, and each class declares the package name in the file header. The package can then be imported with either all the defined classes in the package or a single named class.

6.13 Inheritance and Overriding

Inheritance allows the methods and attributes of a class to be inherited by a subordinate class. In Java a superclass is inherited from a parent or superclass by including the keyword **extends** and the name of the superclass in the declaration of a subclass. The new subclass can use the methods and attributes of its superclass.

A subclass can override the definition of a method by including the compiler directive **@override** before the redefinition of method. The **@override** just tells the compiler that you are doing this intentionally and should not generate an error for doing so. The argument and return types are the same for overridden methods.

WPILib uses inheritance in the definition of commands. The specific instance of a command may override the original version to do a specific action. The **init**, **execute**, **isFinished**, **interrupted**, and **end**

methods of a command class can be modified to do what is required for specific commands, rather than the default action of not doing anything.

6.14 Polymorphism, Overloading, and Signature

Overloading is using the same method name with different input types or return type in the same class. This is also called **Polymorphism**. Each method instance has a different **signature** which is the tuple of the return value types, method name, and input argument types to a particular method instance.

6.15 Abstractions

An abstraction is a way to define a structure of a class or object without specifically defining the class or object. In Java an abstract class cannot be instantiated, it can only be extended. The abstract class defines what needs to be in all classes using it. The extension of the abstract class fill in the details for each class using the abstract class. This allows definition of a design pattern to be followed by a class of classes.

An abstract interface takes this one step further. It includes an abstract method with no body. When the interface is extended, the method must be given a body, so that it can actually do something.

A supplier is an abstract class that gets an attribute. The specific attribute has a type. When the specific supplier is extended, it is specified to use the particular type, and it is supplied a lambda expression that returns the desired attribute value (sometimes invoking a getter method). When the specific supplier is used to specify the class of an argument to a method; the method is stating that it wants to get this attribute at run time and the data type is the type returned by this class of supplier. There are two types in this argument specification: the type of argument of method is a method reference to the specific class of supplier, usually a lambda expression, and the type provided by the method reference should match the type specified for the class of supplier.

See 7.12.3 Getting a Value with a Supplier and a Lambda Expression for a discussion on the use of a lambda expression as a supplier as an argument to a method.

There is a step missing from this process. One needs to define a separate class for each type or class of value that needs to be supplied. This could be tedious to define each specific extension. An abstract class allows a class to be passed to the abstract class to define a specific class by enclosing it in angle brackets. In the following code snippet, the class returned by the abstract Supplier or Consumer class is passed enclosed in angle brackets.

```
  public static void configureHolonomic(
      Supplier<Pose2d> poseSupplier,
      Consumer<Pose2d> resetPose,
      Supplier<ChassisSpeeds> robotRelativeSpeedsSupplier,
      Consumer<ChassisSpeeds> robotRelativeOutput,
      HolonomicPathFollowerConfig config,
      BooleanSupplier shouldFlipPath,
      Subsystem driveSubsystem) {
    if (configured) {
      DriverStation.reportError(
              "Auto builder has already been configured. This is likely in
error.", true);
    }
```

In these declarations the “Supplier<type>” is an abstract class that is extended the specific class, e.g. Supplier<Pose2D> builds a specific class SupplierPose2D for a method reference to return a Pose2D value. All the specific supplier classes need not be defined individually. The abstract class Supplier takes care of the grunt work. The Supplier abstract functional interface is defined as:

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

In line 2 “T” is a generic type enclosed by the angle brackets. This means that it can be any type passed to the Supplier class when it is used.

In line 3 “T” is referencing the type passed as a generic type to the declaration of the get() method. This means that the get method has no arguments and that it will return a value of type T. This method has no body. The body must be defined when the Supplier is used. This is normally done by passing the Supplier a formal method reference or a lambda expression. This method reference is assigned to the method without arguments “get()”.

When the pose is needed in code, the code references “poseSupplier.get()”. Remember that “poseSupplier” is a supplier object and poseSupplier.get() is the method within that object that actually retrieves the desired attribute.

To reiterate all of the steps in an abstract interface class:

1. Define the interface class, e.g., Supplier as a generic type that gets a particular type of variable.
2. Extend the interface class with a specific class or type instance, e.g., BooleanSupplier
3. Define a method with an argument of the extended interface class, e.g., BooleanSupplier shooterReady.
4. Pass a method reference or lambda expression to the method defined in 3, e.g., access the shooter.shooterReady getter which returns a Boolean value with a lambda expression “() → shooter.shooterReady” or “shooter::shooterReady”
5. Within the method defined in 3 above (or the methods that it invokes), get the attributer by invoking the shooterReady class method get with shooterReady.get(). This invokes the method reference passed in 4 above which in turn returns a Boolean value.

6.15.1 Abstractions for Design Patterns

Subsystem and Command are abstract classes.

All subsystems extend the SubsystemBase class as discussed 4.1.1.1 Subsystem Object Pattern. This ensures that all subsystems the same basic functionality.

Individual commands extend the Command class as discussed in 4.1.2 Command Design Pattern. This ensures that all commands are invoked with the same methods which allows the scheduler to be much simpler. No matter what the command does, its execution phase is run by the scheduler as “execute().”

The Command class itself is an extension of the Sendable class which allows the object to be sent over a network.

6.15.2 Common Abstract Classes

Java has a ArrayList that can be an array of any class of objects. The abstract class has a set of methods that operate on the class, so there is no need to build a separate mechanism for every class that you may want to include in an array.

A note on the ArrayList is that it works only on objects. To use this class for primitive data types, a Wrapper class must be used to make the primitive data type look like an object of a particular class.

7 Basic Java Coding

The code for the FRC 4523 robot is written in Java which is a modern object-oriented language. Java was developed by Sun Microsystems in the 1980s to be a portable language that could run on any processor and any operating system. Source code is compiled into byte codes. Intermediate files of the byte codes are found in the .java files in the bin directory of the development file tree. These intermediate files are combined into a java archive file with a .jar extension. The .jar file is interpreted by a runtime program on the target machine, in this case the roboRIO.

Learning your first computer language is a journey of exciting new concepts and understandings. There are many courses available in books or online to teach the basics of many languages including Java. Many of the concepts apply across many languages, so once you know one language it is relatively easy to move to other languages. The intent of this handbook is not to teach you everything you need to know about Java. It does have a few things to remind you of some concepts that are important to understand when reading or writing FRC Java based robot control code. Sometimes this will serve as a signpost to direct you to further study in books or on the Internet.

Much of the syntax of Java is similar to C, JavaScript, Rust, and a few other languages in that it:

- Blocks of code are fenced off with braces ‘{‘ and ‘}’. These blocks also control the scope, although the scope rules vary between languages.
- Expressions in a conditional are enclosed in parenthesis ‘(‘ and ‘)’.
- Members of an array are accessed with an index enclosed in brackets ‘[‘ and ‘]’. For example the second element the array ‘cars’ can be accessed with ‘cars[1]’. Note that array indexes are zero-based and not one-based.
- Long comments are set off by slash-asterisk ‘/*’ and asterisk-slash ‘*/’. Comments at the end of the line are set off by double-slash ‘//’ and extend to the end of the line. Comments are ignored by the compiler and are used to provide information about the program to human readers.

```
int robot1 = 1; // this comment goes to the end of the line
int robot2 = 2;
/* this is a multi line comment
int robot3 = 3;
the above line is commented out, as is this one */
```

- A statement must end with a mandatory semicolon ‘;’.
- Control statements like if, if-else, and loops use a fairly common form, but there are differences.

These similarities allow one to move easily between languages. However, there are some differences and these differences sometimes cause confusion and difficulties because what works in one language may not work the same way in another. This is especially true when using libraries as they have different usage.

Types are extended by classes. A class is a definition of a object with a set of attributes and a set of methods that use or manipulate those attributes. A class can be thought of as a thing, like an elevator or drive train or it can be thought of as a data type like a Point or a String. A **String** is a built-in class to handle arrays of characters. User defined classes also extend type. Robot code uses many of these classes, e.g., Pose2d, Pose3d, Rotation, and Rotation3d. Many of these object classes provide conversion methods. For example the Rotation class provides methods to convert rotations to degrees or radians and vice versa.

A class may be used in methods to create, or instantiate, an instance of the defined class. This instance is a named object and may be used as its own entity. A class may be instantiated any number of times. All objects in a class have the same set of attributes, although the values for those attributes will likely be different. Likewise all objects of a class have the same set of methods to manipulate the individual object.

A special case exists for some classes that will only have one instance, like Main or Robot. These classes may be declared as **static** and used without instantiating them.

Different objects may have similar methods with the same name. This allows for treating objects similarly even if they are in different classes. For example, many objects have an initialization method called ‘init.’

There are many ways to learn Java programming. One way is to use the Team 20 Java video series at Team 20 Intro to Java Programming. W3C schools has a quick introductory Java course at https://www.w3schools.com/java/default.asp. This goes through the basic syntax and operations for Java, but light on the details and nuances. There are more complete courses on the Internet and in books.

7.1 Introduction to Java Syntax

The intent of this section is not to be a complete Java primer or a Java reference. It is intended to introduce the language and some of its esoteric uses within WPILib robot code libraries.

Java is a language and all languages have a syntax. The English syntax is grammar, and it dictates how words are put together in sentences so that they can be understood by a listener or reader. Java is similar. Java is used to communicate step-by-step instructions to a computer. To simplify this communication, tasks use a standardized way of putting words together to minimize the ambiguity between the programmer’s intent and the computer’s actions.

English is built on a hierarchy of punctuation, words, sentences, paragraphs, chapters, stories, books, etc. Java is built on a similar hierarchy:

- **Punctuation** separates and joins other parts of the language. Some punctuation is used to select different encodings of literal values.
- **Comments** are messages to humans that are ignored by the computer.
- **Keywords** tell the computer what the statement is intended to do.
- **Literal values** is a way to communicate known values to the computer.
- **Variables** is a way to keep track of values with a name.

- **Operands** is a way to combine values, like addition, subtraction, etc.
- **Expressions** is a way to produce a value from other values, variables and operators.
- **Condition** is a boolean expression used to control the flow of a program.
- **Statement** is a single (high level) instruction to a computer.
- **Block** is a group of statements usually enclosed by a pair of curly braces '{' and '}'. Block comments is a comment between slash-star '/*' and star-slash '*/'.
- **Class** defines the attributes and methods shared by a set of objects.
- **Object** is a member of a class that has the attributes and methods of the class.
- **Attribute** is a value associated with an object.
- **Method** is a function associated with an object that works with the objects attributes.
- **Arguments** are values passed to an method. **Parameters** or **expressions** are passed to a method which references them as arguments.
- **Return Value** is a value produced by a method.
- **Constructor** is a special method in a class used to initialize the attributes of an object when the object is created or instantiated.
- **Modifiers** are used to modify the accessibility or characteristics of classes, attributes or methods.

7.1.1 Java Punctuation

Java uses punctuation as separators and to make groups. (Some other characters are operators that change the value of an expression, see 7.6.1 Operators.) The basic rules of punctuation of Java are:

- Space characters ' ' are required between keywords and names of variables, arguments, attributes, etc. They are not needed when other punctuation is required.
- Spaces are optional between other syntactic elements. They often make code clearer and easier to read. (see 7.20.2 Coding or Style Conventions).
- Ignore everything after a double slash '//' (end-of-line comment)
- Ignore everything between a slash star '/*' and the first star slash '*/' after that. (block comment)
- End a statement with a semicolon ';'.
- A colon ':' is used to separate a case value from its block of code in a **switch-case** statement.
- A colon ':' is used to separate a looping variable from an array in a **for-each** statement.
- Any statement can be replaced with a block of statements surrounded by curly braces '{' and '}'.
- Multiple declarations can be separated with commas ',' in the same statement.

```
int a, b, c;
int a = 1, b = 0, c = 0;
```

- Control statements enclose condition expressions in parentheses '(' and ')'.
- **for** statement:
  - Encloses three sub-statements within parentheses '(' and ')' separated with semi-colons ';'.
  - The first sub-statement declares and initializes the iteration control variable. This may be more than one statement separated by commas.
  - The second sub-statement is a boolean expression that determines whether the code block is executed or not. Only a single expression is allowed.
  - The third sub-statement is one or more assignment statements to update the iteration variable(s). These statements are separated with commas.
- Operator precedence can be modified by enclosing sub-expressions in parentheses '(' and ')'.
- The arguments in a method declaration are listed with their type between a pair of parentheses '(' and ')'.
- The parameters in a method invocation are listed between a pair of parentheses '(' and ')'.
- No parameter or argument to a method is indicated with an empty pair of parentheses '()'.
- Multiple parameters or arguments to a method are separated with commas.
- An array is declared using empty square brackets '[]' after a data type or Class and before its name, e.g.:

```
int [] counters;
Foo [] foos;
```

- An element of an array is accessed by the array name followed by a positive integer value enclosed in square brackets '[' and ']'.
- Arrays can be initialized with a list of values enclosed in curly braces '{' and '}' and separated with commas ','.
- An array of a variable number of objects may be passed as an argument in two equivalent ways:

```
Void foo (Car[] cars){
    for car in cars{ /*unpack cars (/};
}
Void foo (Car... cars);
    for car in cars{ /*unpack cars (/};
}
//foo may invoked fairly simply as:
foo( ford, chevy, dodge); // ford, chevy and dodge are Car objects
                          // note that enclosing {} are not required
```

- A statement may use more than one line, usually to improve its readability. There is no line continuation character as there is in some other languages.
- A line may have more than one statement, although this is discouraged by best practices.
- Binary or base-2 whole numbers can be preceded with a ‘0b’ prefix.
- Hexadecimal or base-16 whole-numbers can be preceded by a ‘0x’ prefix. The digits a through f may be lower or upper case.
- Octal whole-numbers start with a leading zero and are limited to the digits 0 through 7, e.g., 017 is octal (base 8) for 15 base 10.
- Octal whole-numbers may start with a backslash and leading zero and are limited to the digits 0 through 7, e.g., \017 is octal (base 8) for 15 base 10.
- A floating point number (with or without a decimal point) ends in ‘f’.
- A long integer ends in a upper case ‘L’ to avoid confusion with 1.
- Backslash characters ‘\’ may be used in string to give an alternate or “escaped” value for the character(s) that follow.
- A double-quote ‘”’ within a double-quoted string may be escaped with a backslash ‘\”’.
- A single-quote “’” within a double-quoted string may be escaped with a backslash “\’”.
- Strings may use a tab characters ‘\t’.
- Strings may use a backslash character ‘\\’.
- Strings may use new line characters ‘\n’.
- Strings may use carriage return characters ‘\r’.
- Unicode character values are ‘\u’ followed by four hexadecimal digits
- Octal codes are a backslash and up to three octal digits (0-7).
- Octal codes are literal integer values with a leading zero ‘0’ digit e.g., 017 is octal (base 8) for 15 base 10.
- The at symbol @ prefixes compiler annotations like @override and @functionalInterface; and Javadoc directives like @param, @author, @version, @since (see https://en.wikipedia.org/wiki/Javadoc).

7.2 Comments

Comments are your friend. They offer no functionality to a program. When done correctly, they tell the intent of the original programmer and provide guidance for anyone reading or maintaining the code. Java offers two basic ways to comment: an end-of-line comment and a block comment.

7.2.1 End of Line Comment

End of line comments are used to explain the intent of a particular line of code.

```
foo = 16; // this is an end of line comment
```

7.2.2 Block or Multiple Line Comment

Multiple line comments are well suited to explanations of the code that go beyond a particular line of code. It is sometime used to "comment out" a block of code that is used only for testing or that is no longer needed.

```
/* this is a multi-line comment
in a general form.
and another line */

/*
 * This is a format used to set off a block of code
 * and it can have as
 * many lines as it
 * wants
 * or needs.
 */
```

7.2.3 Block Comment Within a Statement

A comment may be provided within a statement to provide information about that statement. This usage is rare because it breaks the normal flow of the code, making it harder to read.

```
foo /* a comment within a statement */ = 19;
```

7.3 Java Keywords

The following is a list of keywords used with the Java language. These words have special meaning to the Java compiler to signal what the programmer wants to do. In turn the programmer can not the keywords in the names of programming attributes, variables, arguments, methods, classes, packages, etc.

Table 30: Java keywords and usage

Keyword[15]	Usage
abstract	A non-access modifier. Used for classes and methods: An abstract class cannot be used to create objects (to access it, it must be inherited from another class). An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
assert	Used for statements that test code that checks the results of a calculation are what is expected.
boolean	A primitive data type whose values are either true or false.
break	A statement used to immediately exit a loop or **switch-case** statement.
byte	A primitive data type whose value is signed whole numbers between -128 and 127.
case	Begins a block of code within a **switch-case** statement for a specified value that is ended with a break statement.
catch	Begins a block of code within a **try** statement for handling specific exceptions.

15 This list was expanded from a list found at https://www.w3schools.com/java/java_ref_keywords.

Keyword	Usage
char	A primitive data type whose value is a 16-bit Unicode character between \u0000 and \ uffff.
class	Used in a statement to define a class.
continue	Statement used to jump to the next iteration of a loop.
const	Statement used to define. No longer used - use **final** instead.
default	Sets off a block of code within a **switch-case** statement to handle values not specified by other cases. Traditionally this is the last code block in the **switch-case** statement.
do	Begins a **do-while** loop statement.
double	A primitive data type whose value is a signed floating point number between 1.7e−308 to 1.7e+308 (64-bits).
else	Used in a **if-else** statement.
enum	A non-primitive data type with a set of defined ordered values.
exports	Exposes a package in a module for use. New in Java 9.
extends	Inherits the named class, part of a class declaration.
false	The literal boolean false value.
final	A non-access modifier used for classes, attributes and methods to make them non-changeable (impossible to inherit or override). Use for constants.
finally	Sets off a block of code within a try statement that will be executed in every execution.
float	A primitive data type whose value is a signed floating point number between 3.4e−038 and 3.4e+038 (32-bits).
for	Begins a for loop statement.
goto	No longer in use.
if	Begins an if or if-else statement.
implements	Sets off a class using a specified interface. The class must define the methods required by the interface.
import	Starts statement to import a package, class or interface.
instanceof	A unary operator that has a boolean value indicating an object is an instance of a specified class or interface.
int	A primitive data type whose values are signed whole numbers between -2147483648 and 2147483647 (32-bit).
interface	Defines a special class that contains only abstract methods (which must be specified when the interface is instantiated or implemented.
long	A primitive data type whose values are signed whole numbers between -9223372036854775808 to 9223372036854775808 (64-bit).
module	Statement to declare a module. New in Java 9.
native	A non-access modifier for a method is not implemented in Java.
new	Creates new object of the specified class.
package	Statement to declare a package.
private	An access modifier used for attributes, methods and constructors, making them only accessible within the declared class.
protected	An access modifier used for attributes, methods and constructors, making them accessible in the same package and subclasses.
public	An access modifier used for classes, attributes, methods and constructors, making them accessible by any other class.
requires	Specifies required libraries inside a module. New in Java 9.

Keyword	Usage
return	A statement that ends method process and returns a specified value. This allows methods to be functions.
short	A primitive data type whose values are signed whole numbers between -32768 and 32767 (16-bits).
static	A non-access modifier used for methods and attributes that can be accessed without creating an object of a class.
strictfp	No longer used. Restrict the precision and rounding of floating point numbers.
super	Refers to superclass (parent) objects.
switch	Begins a **switch-case** statement which selects one of several **case** or **default** blocks to be executed.
synchronized	A non-access modifier, which specifies that methods can only be accessed by one thread at a time.
this	Refers to the current object in a method or constructor.
throw	Statement to creates a custom error.
throws	Indicates what exceptions may be thrown by a method.
transient	A non-access modifier for an attribute to prevent its storage when serializing an object, used to prevent exposure of sensitive information.
true	The literal boolean true value.
try	Starts a **try**...**catch** statement
var	Declares a variable whose type is inferred. New in Java 10.
void	Used in place of a method data type to specify that the method should not **return** anything.
volatile	A non-access modifier for an attribute to read its value from main memory and not the cached value.
while	Begins a while loop statement or ends a do-while statement.

7.4 Data Types

Java uses data types extensively as it is a strongly typed language. A type is specified as part of the declaration of variables, attributes, arguments, and methods. They are verified when these are used in other parts of the program. Java has the following primitive types:

- **boolean** Values are either true or false.
- **byte** Signed value is whole numbers (integers) between -128 and 127.
- **char** 16-bit Unicode character between \u0000 and \uffff.
- **double** Signed floating point number between 1.7e−308 to 1.7e+308 (64-bits).
- **float** Signed value is from from 3.4e−038 to 3.4e+038 (32-bits).
- **int** Signed whole numbers (integers) between -2147483648 and 2147483647 (32-bit).
- **long** Signed whole numbers (integers) between -9223372036854775808 to 9223372036854775808 (64-bit).
- **short** Signed whole numbers (integers) between -32768 and 32767 (16-bits).

In the robot code, there are many uses of **boolean**, **double**, and **int** data types (and **void** when a method does not return a value). Less common in robot code are byte, char, float, long and short.

7.4.1 Non-primitive Data Types

Data types can also be non-primitive. These data types are objects created using classes. Java has two built-in non-primitive data types: array, String, and enum.

- **array** an ordered list of values of the same type. Java has a library of methods for manipulating arrays. The built-in array has a fixed number of members set when the array is created.
- **Array List** a class for a variable length array of objects of the same type. The number of elements in the Array List may change during the execution of the program.
- **String** a built-in class for an array of chars (characters). Java has a library of methods for manipulating strings.
- **enum** defines a class for a set of defined ordered values (final variables). An enum may not be inherited or extended. It is static, public, and final by default. Enums are useful when defining a set of constants that need to be unique, like the values used for a switch-case statement or defining the sections of an LED string. The names used with enum values are spelled in all capitals letters by convention. For example:

```
enum TrafficLights {  // TrafficLights is the class name
    RED,              // RED is a value of TrafficLights
    YELLOW,
    GREEN
};
```

The individual values are accessed as like an attribute of a class and the type of the variable is the class of the **enum**. E.g.:

```
TrafficLights lightColor = TrafficLights.RED;
```

Enums may have their own methods and a specialized constructor.

- **Wrapper** A set of wrapper classes is provided so that primitive data types can be treated as a class when required, such as in the Array List class. The defined wrapper classes are in the following table:

Table 31: Java wrapper classes

Wrapper Class	Primitive Data Type
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean
Character	char

- **Point** WPILib class for an x, y point on the field.
- **Pose** WPILib class for an x, y point on the field and a Rotation2d for the yaw.
- **Pose2d** WPILib class for an x, y point on the field and a rotation for the yaw.
- **Pose3d** WPILib class for an x, y and z point on or above the field and a Rotation3d for the roll, pitch and yaw angles.
- **Rotation** WPILib class for an angular measurement in full rotations (360°).
- **Rotation3d** WPILib class for an angular measurement in full rotations (360°) for roll, pitch and yaw angles.
- **Translation2d** WPILib class for a point or vector consisting of an x, and y for a point or displacement.
- **Transform2d** WPILib class for a vector consisting of an x, and y for displacement and an angle of rotation.
- **Translation3d** WPILib class for a point or vector consisting of a x-, y-, and z-displacements.
- **Transform3d** WPILib class for a point or vector consisting of a x-, y-, and z-displacements and a Rotation3d for rotation.

7.5 Literal Values

The following are example of various type of literal values that can be used with Java. These are used when the value is known. Many of these find their way into configuration files.

```
2                   // the literal integer value 2
-1                  // the literal integer value -1
0xf                 // the literal integer value of hexadecimal f = 15
0b1001              // the literal integer value of binary 1001 = 9
\u0041              // the literal char value of hexadecimal 41 = 'A'
                    // the above form is used with Unicode characters
\017                // the literal integer value of octal 17 = 15
017                 // the literal integer value of octal 17 = 15
true                // the literal boolean value true
false               // the literal boolean value false
3.5f                // the literal floating point value 3.5
2f                  // the literal floating point value 2.0
100L                // the literal long integer value of 100
```

7.6 Expressions

An expression produces a value. It may use literal values, stored data elements and multiple operations. Expressions may appear within a statement where ever a value or condition is expected. Except for assignment operators, they cannot be a statement by themselves. The resulting value may be any data type or defined Class.

```
3 * daysPerWeek   // an integer expression with multiply operation
counter > 3       // a relational expression evaluating to true or false
2 + 3 * 4         // expression evaluating to 14 (see precedence order)
!true             // false
-foo              // negative foo, integer of float variable
```

Expressions may be contained within a pair of parentheses. The order of precedence for the operators can be controlled by using pairs of parentheses around those sub-expressions that are to be evaluated first.

7.6.1 Operators

Expressions may contain operators to combine or modify the value of operands. Operands may be literal values or stored values (variables, attributes, or arguments).

7.6.1.1 Operator Precedence

The following table lists the operators that can be used with Java in the order of precedence. The order can be modified by using a pair of parentheses which have the highest order of precedence. The use of each operator is explained in the following sections.

Table 32: Java operator precedence and usage

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left

Precedence	Operator	Type	Associativity
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	

There are several operators in Java that do not have precedence as above, but are an important part of the language: These operators are shown in the following table:

Table 33: Java operators with no precedence

Operator	Type
<type>	For passing a data type or class as a parameter to an abstract method.
<>	For declaring an implied data type of class in an abstract method.
() -> expression	For declaring the use of a parameterless lambda expression
(arg1, arg2...) -> expression(arg1, arg2...)	For declaring a lambda expression with one or more parameters
() - > { block }	For declaring a lambda expression with a block of one or more statements. Block cannot contain control statements and must contain a **return** statement if a value is expected.
class :: staticMethod	A method reference to a static method
object :: instanceMethod	A method reference to an objects's method
typeClass :: classMethod	A method reference to a type class's method
class :: new	A method reference to a class constructor
:	In the for-each statement to separate an iterated array member from its containing array.
:	In a switch case statement to separate a case expression from an undelimited block of statements.

7.6.1.2 Grouping Operators Precedence Group 15)

- **()** Parentheses grouping

```
(1 + 3) * 4  // force addition to go ahead of multiplication
```

- **[]** Array subscript

```
int [] cards = {0,1,2,3,4,5} ;   // declare cards as in integer array
                                 // and initialize array
cards [3]          // to access the fourth element of the cards array
```

- **.** member selection (like an attribute or method of a class).

```
System.out.println(Math.random());
    // println() is a method of the out class
        // out class is a class of the System class
    // random() is a method of the Math class
```

7.6.1.3 Unary Operators (Precedence Group 14 and 13)

The first four operators can be statements by themselves or part of an expression for another value. In either case they do an assignment operation on the value being incremented or decremented.

- a++ unary post increment of a (increment and assignment of a is done after the current value of a is used). This is an assignment operator as it is the equivalent of a = a + 1;
- a-- unary post decrement of a (decrement and assignment of a is done after the current value of a is used). This is an assignment operator as it is the equivalent of a = a - 1;
- ++a unary pre increment of foo (increment and assignment of a is done before the new value of a is used). This is an assignment operator as it is the equivalent of a = a + 1;
- --a unary pre decrement of foo (decrement and assignment of a is done after the new value of a is used). This is an assignment operator as it is the equivalent of a = a – 1;
- +a unary plus for floating point and integer values
- -a unary minus for floating point and integer values
- **!**a logical negation for Booleans.
- ~a bitwise complement for bitwise integers (one's complement)
- **(type)** a for changing the type or class of value a to the type or class enclosed with the parentheses (also called type casting to promote confusion with the movie industry).

7.6.1.4 Math Operators (Precedence Group 12 and 11)

- a*b the result of multiplying two floating point or integer values a and b.
- a/ b the result of dividing a floating point or integer value a by another value b of the same type
- a% b the remainder of the division of a floating point or integer value a by another value b of the same type. This is the modulo operator.
- a+b the result of adding two floating point or integer values a and b.
- a-b the result of subtracting a floating point or integer value b from another value a of the same type.

7.6.1.5 Bitwise Shift Operators (Precedence Group 10)

- a << b Shift the integer a integer b places to the right. This is equivalent to multiplying a by 2^b.
- a >> b Shift the integer a integer b places to the right. The sign bit is copied right during the shift. This is equivalent to dividing a by 2^b.
- a >>> b Shift the integer a integer b places to the right, but fill the high order bits, including the sign bit, with zeros.

```
0b101 << 2;          // 0b10100
0b1001 >> 3;         // 0b0001
```

7.6.1.6 Relational Operators (Precedence Group 9 and 8)

- a < b returns a boolean true the value of a is less than the value of b, else it returns false.
- a <= b returns a boolean true the value of a is less than or equal to the value of b, else it returns false,
- a > b returns a boolean true the value of a is greater than the value of b, else it returns false.
- a >= b returns a boolean true the value of a is greater than or equal the value of b, else it returns false.
- a **instanceof** b returns a boolean true if the value b is an instance of class a, else it returns false.
- a == b returns a boolean true if the value a is equal to the value b, else it returns false.
- a **!**= b returns a boolean true if the value a is not equal to the value b, else it returns false.

7.6.1.7 Bitwise Operators (Precedence Group 7, 6, and 5)

Bitwise operators work on integers, but operating on individual binary digit positions. A one digit is treated as true and and a zero-digit is treated as false. All the bits except for the sign bit (the most significant bit) can be used. Right shifts can either copy the sign bit or fill in zeros into the high order bits.

- a **&** b returns a bitwise integer value where each bit n is true if bit n of a and bit n of b is true, else bit n is false. (This is a bitwise AND)
- a ^ b returns a bitwise integer value where each bit n is true if bit n of a equals bit n of b is true, else bit n is false. (This is a bitwise EXCLUSIVE OR.)
- a | b returns a bitwise integer value where each bit n is true if bit n of a or bit of b n is true, else bit n is false. (This is a bitwise OR.)

```
0b1100 & 0b1000;     // 0b1000
0b1100 ^ 0b1000;     // 0b0100
0b1000 | 0b0001;     // 0b1001
```

7.6.1.8 Logical Operators (Precedence Group 4 and 3)

Logical operators work on Booleans. These can be used in condition of conditional statements.

- a **&&** b returns a boolean true if the boolean value a is true and the boolean value b is true, else it returns false.
- a || b returns a boolean true if the boolean value a is true or the boolean value b is true, else it returns false.

```
true || false       // true
true && true        // true
false && false      // false
```

7.6.1.9 Ternary Conditional Operator (Precedence Group 2)

- a **?** b **:** c if the expression **a** is true, use the value of expression **b**; else use the value of the expression **c**.

7.6.1.10 Assignment Operators (Precedence Group 1)

These operators are complete statements by themselves.

- a = b (assignment) the variable or attribute a is set to the value of b.
- a += b (addition assignment) the variable or attribute a is set to the value of a plus the value of b. This is equivalent to a = a + b;.
- a -= b (subtraction assignment) the variable or attribute a is set to the value of a minus the value of b. This is equivalent to a = a - b;.
- a *= b (multiplication assignment) the variable or attribute a is set to the value of a multiplied by the value of b. This is equivalent to a = a * b;.
- a /= b (division assignment) the variable or attribute a is set to the value of a divided by the value of b. This is equivalent to a = a / b;.
- a %= b (modulus assignment) the variable or attribute a is set to the value of the remainder of the value a divided by the value of b. This is equivalent to a = a % b;.

7.6.1.11 Operator Trickiness With Assignment vs. Equivalence

There is confusion for some first time programmers with the assignment operator = and the equivalence operator ==. In many languages including Java, a single equals sign is used to assign a value to a variable or attribute. The left side of the equal sign ‘is set to’ the value of the right side. The same variable can be set to another value at any time (unless the **final** modifier is applied to the variable. The double == is used to test for equality. So the left side of a double equal sign is tested for being equivalent to the right side.

7.6.2 Names of Data Elements and Methods

There is a great deal of flexibility of choosing names for data elements and methods, but there are the following restrictions:

- Cannot use a reserved word.
- Must start with a non-numeric character to distinguish it from numbers. This is usually a letter or an underscore ‘_’ or a currency symbol ‘$’.
- Characters after the first character may use any character (including Unicode characters), except a space character ‘ ‘ and those used for punctuation or operators. Most commonly this is restricted to unaccented Latin letters, numerical digits and the symbols ‘_’ and ‘$’.
- May have any number of characters.
- Name should reflect the usage of the data element or method to make the code easier to understand.

7.7 Variables and Attributes

A data element is a value that can be accessed used by a method. The data element itself can be called several names depending on the context of its usage, even though some data elements are not named. The data element may be any primitive data type or defined class and it value is within the constraints of its type or class. Some data elements are:

- **variable** a named data element used only within the scope of a method. It is created when the containing method is executed and is released when that execution is complete.
- **attribute** a named data element that is a characteristic of an object. It is declared with other definitions of the class. It exists for every object created with that class or with the class itself for static classes.
- **argument** a named data element that is an input passed to a method or constructor. It lives only during the lifetime of the invoked method, but it can be used by the method in any way including being saved as a attribute of the method's parent object.
- **parameter** a value that is passed to an method or constructor. It exists in the instant that the method is invoked and then copied to a corresponding argument of that method.
- **return value** a value that is returned by some methods. If it is not immediately used or saved, its value is lost forever.
- **constant** a named variable or attribute declared with the **final** modifier. The value can be set only once and cannot be changed.

An example of these values is shown in the following example:

```
class Main {
    int last = 0;             // "last" is declared as an integer
                              // attribute of the class "Main"
                              // initialized to zero

    int square (int a) {      // the method "square" is declared as
                              // returning an integer value and
                              // has one integer argument "a"
        int temp = a * a;     // the variable "temp" is declared as an
                              // integer and is assigned the value a*a
        return temp;          // the value of temp is returned
                              // "square"" dies, as does "a" and "temp"
    }

    void main() {             // the method "main" with no arguments
        last = 3;             // Main attribute "last" set to 3
        last = square( 2)     // Main attribute "last" set to the
                              // value returned by the method square
                              // when passed a literal parameter of 2
                              // which is the integer value 4.
    }
}
```

7.7.1 Variable or Attribute Declaration

The declaration of a variable or attribute within a method defines it name and data type and is used by the language to allocate memory space for its value.

```
int foo;     // foo is declared as an integer, but not initialized
```

7.7.2 Variable or Attribute Initialization and Assignment

Assignment of a variable or attribute is to give it a specific value. The variable or attribute must be declared before it is assigned. A variable or attribute must be initialized before it is used. A variable or attribute may be reassigned a different value at any time, unless the variable or attribute was declared with the **final** modifier.

```
                // foo was previously declared as an integer
foo = 19;       // now foo is initialized to 19
                // this form could be used for normal assignments
```

7.7.3 Variable or Attribute Declaration and Initialization

A variable or attribute can be declared and initialized in one statement

```
int foo = 19;   // foo is declared as an integer, and initialized to 19
```

7.7.4 Scope Rules for Variables

The scope rules limit the extent that a variable can be accessed, so the same name can be used in multiple methods of an object or multiple blocks within a method without collisions. Any variable can only be access by the method in which it is declared. This scope can be further limited by declaring the variable inside of a code block bounded by curly braces ‘{ and ‘}’ and the variable can only be accessed within that block. The scope of the iteration variable(s) used by a for statement are restricted to that fore statement.

7.7.5 Scope Rules for Attributes and Methods

Attributes and methods are declared in the context of the class, so they can be accessed by any method in the class. If an attribute or class is declared with the **private** modifier, it can only be accessed from within the class with its name. Without the **private** modifier any class with access to the attribute’s defining class can access the attribute with the notation objectName.attributeName or the method with the notation objectName.methodName(). If the attribute was in a **static** class, the attribute can be accessed with ClassName.attributeName. If the method was in a **static** class, the method can be accessed with ClassName.methodName().

Table 34: Scope of attributes and methods as affected by modifiers

	Class	Package	Subclass (same package)	Subclass (different package)	World
public	yes	yes	yes	yes	yes
protected	yes	yes	yes	no	no
default, no modifier	yes	yes	yes	no	no
private	yes	no	no	no	no

7.8 Control Statements

Statements are the heart of any programming language. So far we have seen statements for declaring and initializing variables and attributes and there were statements that assign a new or changed value to a variable or attribute. Control statements allow for code to be selectively or repetitively executed based on a boolean that is computed on the fly. These control statements are the most powerful aspects of any programming language, and they are detailed in the following sections.

Java has other statements. Some declare the statements included in a method or in a special method, a constructor, used to initialize a newly created object (see 7.9 Methods). Some for the declaration of classes or the instantiation of a class in the declaration of an object (see 7.11 Classes and Objects). Some statements intercept and generate error conditions (see 7.16 Exceptions, Error Handling, and Debugging Statements). Some statement are used to build packages of code and to refer to those packages (See 7.13 Packages and Import.)

7.8.1 if Control Statement

```
if (condition) statement; // short form, unusual
                         // when condition is true
                         // the single statement is executed.

if (condition) {         // long form, ordinary
    /*block of one or more statements that are executed
      when the condition is true */
};
```

7.8.2 if-else Control Statement

The **if-else** statement is used to execute one of two statements depending on whether a condition is true or false. This leads to a some ambiguity, because the statement after the if and condition must be terminated with a semi-colon. The **else** clause is a new statement and is assumed to be executed on the failure of the previous **if** statement. The following example illustrates the ambiguity.

```
if (condition) statement; // short form, unusual
                         // when condition is true
                         // the single statement is executed.
else statement;          // if the above condition is false
                         // this single statement is executed.
                         // The linkage to the if part of the
                         // statement is ambiguous, see next.

if (condition1) if (condition2) statement1;
else statement2;
// statement1 is executed if condition1 and condition2 are true
// it is ambiguous whether statement2 executes
//   when condition1 is false OR
//   when condition1 is true and condition2 is false.
// Java does pick the latter, but that depends on the first.
// whenever the if statement is changed, the linkage to the
// else clause may also change introducing an error.
```

Always using the curly brace '{' and '}' notation for all code blocks is a better practice even on block containing only a single statement, because it is not ambiguous. Using that as a design pattern also makes **if**, **if-else**, and **if-else-if** statements much easier to read. An example of using the block notation is as follows:

```
if (condition) {          // long form, ordinary, more reliable
    /*block of one or more statements that are executed
      when the condition is true */
} else {
    /* block of one or more statements that are executed
       when the condition is false */
};
```

7.8.3 if-else-if Control Statement

A special syntax is provided to control the depth of nested **if-else** statement. This syntax treats an **if** or **if-else** statement a bit differently than other statements in the **else** clause of an **if-else** statement, The syntax extends the number of clauses in an **if-else** statement to allow as many as necessary to process the conditions. Normally the curly braces '{' and '}' would have to surround the else clause. The syntax basically is as follows:

```
if (condition1) {          // long form, ordinary, more reliable
    /*block of one or more statements that are executed
      when the condition1 is true */
} else if (condition2) {
    /* block of one or more statements that are executed
       when the condition1 is false and condition 2 is true */
} else if (condition3) {
    /* block of one or more statements that are executed
       when the condition1 and condition2 are false
       and condition 3 is true */
//...
} else {
    /* block of one or more statements that are executed
       when all conditions are false */
};
```

7.8.4 for loop Control Statement

A **for** statement typically executes a block of statements a number of times. Its condition has three sub-statements: initialization, conditional for executing and iteration. The initialization sub-statement sets up the iteration variable(s). The conditional is an expression returning a boolean. If it is true, the block of statements is be executed along with the iteration part and the condition is tested again. If the conditional is false, execution passes to the next statement. The iteration sub-statement changes the iteration variable so that the statement advances and should eventually end.

```
for (int i = 0;  /* declare and initialize iteration variable */
     i < 10;     /* test for completion */
     i++) {      /* iterate and begin block*/
    /* block of statements */
    System.out.println( i);
```

```
};   /* end block and end of for statement*/
```

This code is equivalent to the code the next example of a while loop.

7.8.5 while loop Control Statement

A **while** statement executes a block of statements while a condition is true. The condition is tested first, so the block may not be executed if the condition is false at the onset.

```
int i = 0;       /* declare and initialize iteration variable */
while (i < 10) {    /* test for completion, begin block*/
    /* block of statements */
    System.out.println( i);
    i++;      /* iterate */
};             /* end of block and end of while statement*/
```

7.8.6 do-while loop Control Statement

A **do-while** statement is like a while statement in that it executes a block of statements while a condition is true. The difference is that the condition is tested after executing the block, so the block is always executed at least once.

```
int i = 0;           /* declare and initialize loop variable */
do {
                     /* block of statements */
    System.out.println( i);
    i++;             /* iterate loop variable */
} while (i < 10);    /* end of block */
                     /* test for completion */
                     /* end of do-while statement/
```

7.8.7 for-each loop Control Statement

A **for-each** statement loops through all members of an array. It uses a special syntax by introducing a **type member:array** construct, The **type** is the data type of the member and array. The **member** is a variable that represents one member of the array that changes from the first to the last member as the statement progresses. The **array** is an array of zero or more members. For example:

```
int [] primes = { 2, 3, 5, 7, 11, 13, 17, 19};
for (int for-each i : primes) {
                     /* block of statements */
    System.out.println( i);
}                    /* end of block, end of statement */
```

The above example is equivalent to the following for statement:

```
int [] primes = { 2, 3, 5, 7, 11, 13, 17, 19};
for (int i = 0; i < primes.length; i++) {
                     /* block of statements */
    System.out.println( primes [i]);
```

```
}                   /* end of block, end of statement */
```

7.8.8 switch-case-default Control Statement

A **switch** statement selects a group of statements to execute based on a value matching a **case** value associated with a block of statements. The expression and case values must be of the same type. A **default** case block of statements can also be defined for values that do not match any of the other case values.

```
switch (expression) {
    case value1: // begin  block
        /* value 1 statements */
        break;          // end block, go to the end of statement
    case value1: begin block
        /* value 2 statements */
        break;          // end block go to the end of statement
    default:            // matches any value not already matched
        /* default value statements */
}                       // end of default block, end of statement
```

An example of a case statement is:

```
enum TrafficLights { RED, YELLOW, GREEN};
TrafficLights light = RED;
switch (light){
    case YELLOW:
        slowDown();
        break;
    case GREEN:
        goAhead();
        break;
    default:
        stop();
}                   /* end of block, end of statement */
```

7.8.9 return Control Statement

A return statement returns a specified value of the type declared for the method in which it is used.

7.8.10 break Control Statement

A break statement skips other statements to the end of the current block. It is used to mark the end of case blocks in a switch-case statements and to exit the current loop code block.

7.8.11 continue Control Statement

A continue statement skips other statement to the end of the current block, but the iteration or looping test is still performed. It is used to selectively skip loop processing.

7.9 Methods

A method in Java is a callable block of code that may or may not return a value. A method must be defined within the context of a class, so it may be just used in a static class like Math, or it may act upon individual objects created from the class.

Methods are called many names by other languages: routines, function, procedures. A method typically acts upon and within an object. Functions generally applies when a value is returned. In Java and similar languages, the type of the returned value must be declared when the method is declared.

Methods are usually invoked, but they can be called, executed, or run (depending on the background of the speaker).

A method may use a set of named argument values which it uses when it executes. Each of these values has a name and a type.

A method is invoked with a set of parameter values which match up with the type and number of the arguments in the method's declaration.

7.9.1 Method Declaration

A method can only be declared within a class (or nested within a method within a class). The following is an example of a method declaration:

```
static class Main { / the containing class, Main
                    // static indicates the class can be used without
                    // instantiating objects.
    public void nada () {
                          // public let's methods in other classes
                          //    use this method
                          // void means no value is returned
                          // nada, is the method's name
                          //    i.e., Not Applicable for Doing Anything
                          // () indicates a method with no arguments
                          // no statements in block, do nothing at all
    }
}
```

7.9.2 Method Declaration Without a Returned Value

```
static class Main {       // the containing class, Main
    private int foo = 0;  //declare and initialize attribute foo
                          // private means foo can only be accessed by
                          //   the containing class Main
                          // int means foo is an integer.
                          // = 0 initializes foo to zero.

    public void setFoo( int newFoo) {
                          // public means setfoo can be accessed by
                          //   other classes
                          // void means setFoo does not return a value.
                          // int means foo is an integer.
                          // setFoo is the method name.
                          // () indicates a method and arguments.
```

```
                            // int is the data type for first argument
                            //  newFoo, an integer.
                            // newFoo is the name of the first argument
                            // the curly braces hold the method's code
                            //    block
        foo = newFoo;
                            // assign the attribute foo with the value
                            //    of the argument newFoo
    }
}
```

7.9.3 Method Declaration With a Returned Value

```
public class Main {
    private int foo = 0;  //declare and initialize attribute foo

    public int getFoo() {
        return foo;    // return the value of the attribute foo
    }
}
```

7.9.4 Reference a Method Within Scope of Its Containing Class or Object

A method can be referenced anywhere within its defining Class or object. The following example shows a method being referenced within its defining class.

```
static class MyMath { // the containing class
    double square (double in) { // square is method of type double
        return in * in;
    }

    double cube (double in) {
        return square( in) * in; // square is method in same class
    }
}
```

In general a method cannot be defined within another method. There may be exceptions to this, but that is beyond the scope of this paper.

7.9.5 Reference a Method Outside of its Containing Class or Object

Private methods cannot be accessed outside of their containing Class or object. A method defined with the public modifier may be accessed outside of its defining Class or object, but the name of the Class must be used to access it, as shown below:

```
static class MyMath { // the containing class
    public double square (double in) {
                                        // the public modifier allows
                                        // external access
        return in * in;
    }
}

// in a method in another class somewhere
double a = MyMath.square( 4.2f); // uses method square of class MyMath
                                 // and saves results.

    MyMath.square (3); // is an example of calling a method as if it
                       // did not return a value. Perfectly legal.
                       // In this case the value is not saved, so
                       // it is lost forever.
```

7.9.6 Example of a Method Declaration

An example of a method declaration with a lot of options is as shown to the right. All methods start with a return value type. If nothing is to be returned, this type is "void." A method can return any type, like a boolean for a condition that is true or false, an integer, float or double for methods that return a primitive value.

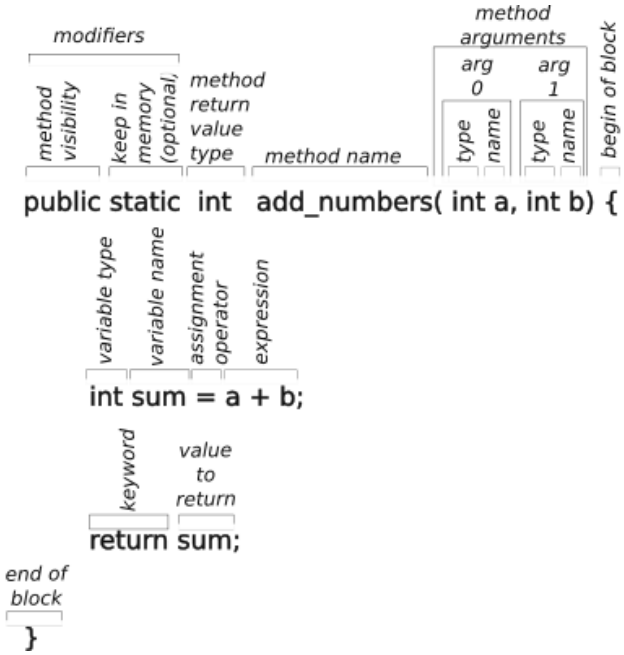


Figure 115: Parts of a method declaration

The method return type is followed by the method name and a set of parameters enclosed in parentheses. A method need not have parameters, which is indicated with just the open and close parenthesis. Each parameter is declared with a type and name. The scope of the name is only within the method which means that the parameter cannot be accessed outside the method.

After the parameters is a block of code set off with an opening and closing curly brace. To be useful there are one or more statements which do something with the parameters and may or may not return a value of the type declared with the method declaration.

7.9.7 Method Overloading

Overloading allow a method name to be used with different inputs and types while generally doing the same thing. For example a math routine to square a number could have the following methods for each data type expected:

```
class MyMath {

    int square (int a) {          // signature is (int, int)
        return a*a
    }

    float square (float a) {      // signature is (float, float)
        return a*a
    }

    double square (double a) {    // signature is (double, double)
        return a*a
    }
}
```

The sequence of return types and argument types for a signature for the method. Each overloaded method must have a different signature.

7.10 Modifiers for Classes, Attributes, Methods, and Constructors

The declaration of classes, attributes, methods and constructors use modifiers to control how they are to be accessed and used. Modifiers are split into two groups: access modifiers and non-access modifiers.

7.10.1 Access Modifiers

Access modifiers control how the Class, attribute, method, or constructor can be accessed. This is discussed in 7.7.5 Scope Rules for Attributes and Methods.

Table 35: Use of modifiers to control scope of methods and attributes

Modifier	Used by	Description
public	Class	The class can be accessed by any other class.
default, no modifier	Class	The class can be access only by classes in the same package.
public	Attributes, methods, and constructors	The item can be accessed by any other class.
private	Attributes, methods, and constructors	The item can be accessed only within the defining class.
default, no modifier	Attributes, methods, and constructors	The item can be accessed by classes in the same package.
protected	Attributes, methods, and constructors	The item can be accessed by classes in the same package and subclasses.

7.10.2 Non-Access Modifiers

Non-access modifiers control how a Class, object, attribute, method, or constructor can be used. The following table describes how the non-access modifiers can be used:

For Classes:

- **final** means that the class cannot be inherited by other classes
- **abstract** means that the class cannot be instantiated. The class can only be inherited.

For attributes of a Class:

- **final** means that the attribute can be assigned a value only one time.
- **static** means the attribute belongs to the class rather than an instantiated object.
- **transient** means the attribute is skipped when serializing the object.
- **volatile** means that the attribute is not cached and is always read from the “main” memory.

For methods of a Class:

- **final** means that the method cannot be overridden.
- **static** means the method belongs to the class rather than an instantiated object.
- **abstract** can only be used in an abstract class. The method only has a return type and arguments, but no body. The body has to be provided by the subclass inheriting the containing abstract class.

- **transient** means the method is skipped when serializing the object.
- **synchronized** means the method can be only accessed by one thread at a time.

7.11 Classes and Objects

The next level in the Java language hierarchy is the Class which is used to create objects. A class consists of:

- A set of data attributes that define the characteristics of an object of this class. All objects created by the class have the same set of attributes, although probably with different values.
- One or more constructor methods used to initialize the object when the object is instantiated. This means initializing the data attributes of the object.
- Zero or more methods that operate on the data attributes and possibly hardware devices like motors and LED strings.

7.11.1 Declaring a Class

A class is declared with the keyword **class**, probably some access modifiers and maybe some inheritance links.

A minimal Java program consists of the Main class and within that a public main method. When Java program is started, control is always passed to the Main.main() method.

```
public class Main {
    // define the class Main. It is public so others can access it.
    // it can be final, because it cannot be inherited.

    public static void main() {
        // declare the method main() as public
        // declare main () as void because it must return nothing.
        // the argument list is: (String[] args)

        System.out.println( “Main.main has executed!”)

    } // end of main() method
} // end of Main class
```

A class must have a matching file name: the “Main” class is in the file named “Main.java”.

7.11.2 Class Constructor

A class normally will have one or more constructors. The constructor is a special method used to initialize the attributes of a class when the class is instantiated as an object. The constructor declaration looks similar to a method declaration except:

- There is no return type. The constructor effectively returns a handle to a newly created object of the type of the defining class.
- It has the same name as the defining class (e.g., the constructor for the class Fee is named Fee()).
- There can be arguments used during the initialization/construction process.

- There can be many constructors for a class for different needs in setting up the object.

An example of a simple class with constructor declaration is below:

```
public class Fee {
    /* class attribute declarations */
    double fee = 0f;

    public Fee(/*may have arguments*/) { /***CONSTRUCTOR***/
        /* constructor code block including attribute initialization*/
        fee = 0f;
     {

     /* class methods */
}
```

7.11.3 Object Instantiation (Class Reference)

An object can be created or instantiated from a class. An example is below:

```
Fee fum = new Fee(); /* class fee is instantiated as object fum */
    /* Fee is a data type reference to the class Fee. */
    /* fum is the new object name */
    /* new indicates that memory should be allocated for object*/
    /* Fee() is the object constructor with parameters as needed. */
```

7.11.4 Referencing a Class Attribute or Method

The difference in syntax between an attribute and a method is:

- An attribute looks like: name.
- A method looks like: name() or name(parameters...)

Within the class definition, a class's methods and names can be accessed directly.

From other classes, to access an attribute or method in a static class:

- The desired method or attribute has to be defined with the **public** modifier,
- The method or attribute name has to be prepended with the class name and the member selection operator '.'

From other classes, to access an attribute or method in a defined object:

- The desired method or attribute has to be defined with the **public** modifier,
- The method or attribute name has to be prepended with the object handle and the member selection operator '.'. The object handle is the attribute holding the object being referenced.

See also:

- 7.9.4 Reference a Method Within Scope of Its Containing Class or Object
- 7.9.5 Reference a Method Outside of its Containing Class or Object

An example of accessing an object's attributes is below:

In the file “Point.java”

```
public class Point () {
    public double x;
    public double y;

    public Point ( double a, double b) {  /*** CONSTRUCTOR ***/
        x = a;
        y = b;
    }
}
```

In the file Main.java

```
static public class Main {

    static public void main(Strings [] args) {
        Point myPoint = new Point (2f,4f);
        System.out.printf( “myPoint is at (%f, %f)”,
                myPoint.x, myPoint.y);
    }
}
```

7.11.5 Example of a Class Definition

Let’s work through a more complete example for a sample street address class declaration.

```
1.  Class Address (){
2.      // list of class attributes
3.      private String streetNumber; // allow numbers like ‘234 1/2’
4.      private String streetName;
5.      private String city;
6.      private String stateAbbreviation;
7.      private int zipCode;
8.
9.      public Address () { /***CONSTRUCTOR***/ // with no arguments
10.         streetNumber = 0;
11.         streetAddress = “”;
12.         city = “”;
13.         stateAbbreviation = “”
14.     }
15.
16.     public Address ( String number, String street, String city, String state,
    int zip) { /***CONSTRUCTOR***/
17.                     // with arguments to initialize all attributes
18.         streetNumber = number;
19.         streetName = street;
20.         this.city = city;  // this.city is the attribute
21.                            // city is the argument
22.         stateAbbreviation = state;
23.         zipCode = zip;
24.     }
25.
26.     public void setStreetName( String street) {
```

```
27.                                     // a “setter” for street name
28.         streetName = street;
29.     }
30.
31.     public void getFormattedAddress() {
32.                                     // a “getter” for formatted address
33.         String zipStr = "00000" + zipCode;
34.                                     //pad zipCode to have leading zeros
35.         zipStr = zipStr.substring(zipStr.length() -5);
36.                                     // limit to 5 digits
37.         String line1 = streetNumber + “ “ + streetName;
38.         String line2 = city + “ “ + stateAbbreviation +
39.                             “ “ + zipStr;
40.         return (line1 +”\n” + line2);
41.     }
42. }
```

This code breaks down as follows:

- Line 1 declares the definition of the class Address. It includes all line through line 42.
- Lines 2 through 7 declares the attributes of the class Address. These attributes are declared as private to prevent outside methods from accessing them. All are of type String except for the zipCode in line 7 which is an int or integer.
- Lines 9 though 14 declares a constructor with no arguments for building a new object of the Address class with attributes set to minimal values.
- Lines 16 through 24 declares a constructor with arguments for building a new object of the Address class with attributes set to the argument values.
- Lines 26 through 29 declares a setter method used for setting the value of the streetName attribute of an Address object to the value of an argument string.
- Lines 31 through 41 declares a method for retrieving and formatting a street address from an Address object.

7.12 Encapsulation, Getters, and Setters

Encapsulation protects the attributes within a object from direct access from outside the object by declaring them with the **private** access modifier.

The attributes can be accessed with a **getter** method, usually named getAttributeName() and changed with a **setter** method, usually named setAttributeName().

This gives the program better control over an object’s attributes and increases the security of the data. If necessary the getters and setters can log or print messages when an attribute is accessed.

7.12.1 A “Getter” Method

Normally the attributes of a method are private to prevent them from being accesses or modified externally. A “getter” method is used to return the value of a private attribute externally. This method can log accesses for debugger purposes, or it can provide additional control measures.

```
public class Main {
    private int foo = 0;  //declare and initialize attribute foo

    public int getFoo() { // declare getFoo method to get foo.
        return foo;    // return the value of the attribute foo
    }
}
```

7.12.2 A “Setter” Method

Normally the attributes of a method are private to prevent them from being accessed or modified externally. A “setter” method is used to modify the value of a private attribute externally.

```
static class Main {       // the containing class, Main
    private int foo = 0;  //declare and initialize attribute foo
                          // private means foo can only be accessed by
                          //   the containing class Main
                          // int means foo is an integer.
                          // = 0 initializes foo to zero.

    public void setFoo( int newFoo) {
                          // public means setfoo can be accessed by
                          //   other classes
                          // void means setFoo does not return a value.
                          // int means foo is an integer.
                          // setFoo is the method name.
                          // () indicates a method and arguments.
                          // int is the data type for first argument
                          // newFoo, an integer.
                          // newFoo is the name of the first argument
                          // the curly braces hold the method’s code
                          // block
        foo = newFoo;
                          // assign the attribute foo with the value
                          // of the argument newFoo
    }
}
```

7.12.3 Getting a Value with a Supplier and a Lambda Expression

This section is fairly complex because it uses indirection, abstraction and very terse syntax. That makes it powerful, but hard to grasp the first couple of times around.

The basic problem is that some commands need to read changing values during their execution which may last an entire match. Since everything in Java is call by value, only values can be passed when a command initially invoked. The solution is to pass a lambda expression that can be evaluated during the execution of the command, and it will produce current values. This means that the command must accept a lambda expression as an argument and that the expression must return a certain data type when the expression is evaluated.

Java has a Supplier class that is an abstract interface with a single abstract method get(). This class is used to define datatype specific suppliers. The constructor for an object of the boolean Supplier class takes a lambda expression as an argument. The constructor sets the abstract get () method of that object to that

argument (a lambda expression). Now whenever the get() method of that object is invoked, it returns the results specified by the original lambda expression.

That was the quick version. Now a little slower, step by step from a developer point of view on a real example.

1. Find an expression that returns the desired value from any context. For example.

```
Robot.intake.getGamepieceDetected())
```

2. Create a lambda expression to return the expression in 1. For example:

```
()-> Robot.intake.getGamepieceDetected())
```

3. Use the above lambda expression as a parameter to a method where it has a runnable argument that supplies the particular data type. This may be an abstract interface. The argument is usually saved as a method in an abstract interface type by the called method. For example:

```
command = new WaitUntilCommand ( ()- > Robot.intake.getGamepieceDetected());
```

This passes the lambda function to the WaitUntilCommand constructor which saves it as a BooleanSupplier in the method handle m_condition[16], as follows:

```
public class WaitUntilCommand extends Command {
  private final BooleanSupplier m_condition;
       // m_… flags the method as a handle to an object that can be executed
       // final allows the method to be set, but not changed thereafter.

  /**
   * Creates a new WaitUntilCommand that ends after a given condition becomes true.
   *
   * @param condition the condition to determine when to end
   */
  public WaitUntilCommand(BooleanSupplier condition) {  //***CONSTRUCTOR***
    m_condition = requireNonNullParam(condition, "condition", "WaitUntilCommand");
       // requireNonNullParam is a runtime check that the condition is not null
       // the two strings would be printed as an error message if condition is null
  }
. . .
}
```

The BooleanSupplier is a functional interface defined as follows (note that the implementation has changed get() into getAsBoolean()):

```
package java.util.function;

@FunctionalInterface
public interface BooleanSupplier {
```

16 A “handle” is a attribute or variable that points at an object or a method reference.

```
   boolean getAsBoolean();  //this is an abstract method with no body
}
```

4. In this example, the WaitUntilCommand follows the command design pattern. It uses the condition to determine when the command isFinished by running the saved lambda function by executing the getAsBoolean() abstract method of the m_condition handle which was set to the lambda expression in step 3 above.

```
public class WaitUntilCommand extends Command {
  private final BooleanSupplier m_condition;
. . .

  @Override
  public boolean isFinished() {
    return m_condition.getAsBoolean();
  }
}
```

7.12.4 Setting a Value with a Consumer and a Lambda Expression

This section is similar to the preceding section and is used for the delayed execution of a setter function. The only examples of consumers in the current code base are used in autonomous routines. The robot pose can be set depending on its starting position. This position is configured into the autonomous routine and is passed to reset the pose through the magic of the autonomous routine.

Java uses a Consumer functional interface to allow attributes to be set at some time in the future similar to the way that the Supplier functional interface works.

A lambda expression is passed as a parameter to a method that accepts it as a Consumer. In this example of setting up the configuration for autonomous operation, a list of Suppliers and Consumers is provided.

```
// Configures the auto builder to use to run paths in autonomous and in teleop
public static void configureAutoBuilder() {
    // Configure the AutoBuilder settings
    AutoBuilder.configureHolonomic(
        Robot.swerve::getPose,
                            // Supplier<Pose2d> ------------> Robot pose supplier
        Robot.swerve::resetPose,
                            // Consumer<Pose2d> ------------> Method to reset odometry
        Robot.swerve::getChassisSpeeds,
                            // Supplier<ChassisSpeeds> -----> MUST BE ROBOT RELATIVE
        Robot.swerve::driveByChassisSpeeds,
                            // Consumer<ChassisSpeeds> -----> Set robot relative speeds
                            // (drive)
        AutoConfig.AutoPathFollowerConfig,
                            // HolonomicPathFollowerConfig -> config for path commands
        () -> false,
                            // BooleanSupplier -------------> Should mirror/flip path
        Robot.swerve
                            // Subsystem: ------------------> required subsystem (swerve)
    );
}
```

This uses the '::' operator as a method reference. A method reference is similar to a lambda expression. The compiler takes care of passing the necessary parameters, so it is more compact. "Robot.swerve::resetPose" is equivalent to "()-> Robot.drivetrain.DrivetrainSysSys.resetPose()".

7.13 Packages and Import

A package is a group of similar classes. The files defining the classes of a package are in a common directory. A program can import either a package including all classes in that package, or a single class from a package, for example:

```
import java.util.*;           /* import all util classes */
import java.util.Scanner;     /*import only the util Scanner class */
```

Some packages are considered to be an Application Program Interface or API. The Java util package and the WPILib packages are APIs.

To create your own package, start with a directory with the intended package name. This name should be lower case to distinguish it from the classes that it contains. In that directory create the Classes for the package starting by identifying the containing package:

```
package mypackage;
class ...
```

The Classes of the package must be compiled and saved to the package directory:

```
javac -d . <list of files defining the classes of the package>
```

7.14 Inheritance and the Extends Keyword

A class (called a subclass) can inherit the attributes and methods of another class (called a superclass) by using the **extends** keyword in the line declaring the class. In the WPILib architecture all subsystems inherit attributes and methods from the SubsystemBase class. The superclass is a class of subclasses.

7.14.1 Inner Classes

Classes can be nested, so inner classes can be declared within a class and can only be instantiated after the outer class is initiated. The inner class can be protected, so that it can only be instantiated or used by the outer class.

It is not known if inner classes are typical in robot code or in the WPILib.

7.15 Abstractions

7.15.1 Abstract Classes

An abstract class cannot be instantiated. It can only be inherited by other classes. Without using the **super** keyword, when the subclass is instantiated, the abstract class constructor will initialize the super class attributes of the subclass.

The WPILib SubsystemBase is an abstract class used by all subsystem classes. Likewise, all command classes are inherited from an abstract Command class.

7.15.2 Abstract Method

An abstract method has no body, and it can only be used in a abstract classes. Its body must be created by the subclass.

7.15.3 Interface and Implements Keyword

An interface is a completely abstract class and all of its methods are abstract.

An interface is accessed with the **implements** keyword. An interface cannot be instantiated and cannot contain a constructor.

7.16 Exceptions, Error Handling, and Debugging Statements

Java produces a number of errors or exceptions, including:

- Compiler errors like common syntax errors. With the modern IDEs, this should be a thing of the past. Check the code before issuing a **build** or **deploy** command.
- Runtime errors like divide by zero or NullPointerException. These are nastier. If they remain in your code they could pop up and stop the program dead in its tracks. Thorough testing will eliminate most if not all of these errors. Special error handling can be developed to catch specific errors from specific instructions. This handling is to do something that allows the program to go on without bring it to a halt.

```
try {
    /* block of code in the scope of the try statement */
} catch (SpecificException se) {
    /* block of code to handle the exception */
} catch (Exception e) { /* any other exceptions */
    /* block of code to handle the other exceptions */
} finally {
    /* block of code executed after any error handling (if any) */
}
```

  Java documentation has a list of the Exceptions enum.

- Programming errors like using the wrong sign of a number.
  - **throw** statement to cause a user error exception.
  - **throws** modifies a method declaration to list the exceptions that could be thrown by the method.
  - Unit testing could use **assert** statements to throw errors when the assertion is not true. This could be a lot of work. A separate unit test method may be easier to implement. The assert statement only works when a compiler switch is applied.
  - Temporary print or printf statements to show what the code is doing.

7.16.1 How to Print Progress Messages

When debugging code, it is sometimes useful to get an indication that the program got to where it was supposed to or got somewhere it was not supposed to get to. This can be accomplished by sending a text message to the console with the following code:

```
System.out.println(“The program got here OK”);
```

7.16.2 How to Print Progress Messages with Values (printf)

Of course that is quite limited so you might want to learn about the printf() method that allows you to output a formatted string with values to let you know if things are calculated correctly. The following is an example of printing out an angle value from within a for loop:

```
System.out.printf(“Program loop %d where angle is:%f”, i, angle);
    // i is some loop iteration counter
    // angle is the angle of the whatever is being monitored
```

The Java **printf()** method is modeled after the c language version of **printf()**. It can easily display and format many data types. The format for a particular value is indicated by a percent sign “%” followed by a character specifying the format desired. “%d” for integers, “%f” for floating point numbers, and “%s” for strings are the most common, %n for inserting a newline character, but the list goes on. The formatted string is followed by a list of values that are formatted in the order that they appear. The command is powerful to format output just about any way desired (padding, precision, number of digits, etc.). This method is worth learning more about and is beyond the scope of this paper.

7.17 Threads

A thread is a separate processing stream in a computer operating system. A program my create multiple threads for various reasons, such as:

- To simplify the overall processing. Several processes are independent of the command scheduling process, such as odometry, telemetry and logging.
- To improve performance on processors with multiple cores. The roboRIO has two cores that can process threads. The operating system is responsible for balancing the load between threads.
- To provide some separation. Lambda function evaluations typically run in a short lived separate stream that returns a value that can be used as needed.

7.17.1 How to Create a Thread

Create an object of the abstract type Runnable. It is run by invoking its run() method.

7.17.2 How to Make Code Thread Safe

Multiple threads running on the same program can cause problems. One process may be partially complete in its writing of data to a common data store and the reading process gets partially old data and partially new data. To prevent this from happening, processes use semaphores to lock the resource, data attribute or method, for their exclusive use. When they are done, the semaphore is unlocked. Processes desiring the resource check the semaphore. The semaphore can be automatically locked if the resource is

available or the process is blocked until it becomes available. The check-lock sequence is indivisible to prevent a race condition where two processes could be granted access to a resource at the same time.

Java uses the **synchronized** modifier for that purpose.

7.18 Lambda (or Anonymous) Expressions

A lambda expression is an anonymous expression or an expression that can be used without formally declaring it and its name. A common use of lambda expressions to pass an expression that will be evaluated during the execution of the method. The expression has a delayed evaluation, so it not evaluated when the method is initially invoked, but later on during its execution.

Other languages use a similar lambda function, but it looks more like a function or method. In Java it looks and operates like an expression.

A common use of a lambda expression is in sort routines. The lambda expression is used to compare two Strings a and b, to determine whether a > b, a = b, or a < b.

If you just compare the character value of each string position, you will get a sorted file that may contain:

```
$
10000
2
450
5
Aardvark
Beaver
anteater
buffalo
-thanks
```

To improve the comparison expression, it needs to know about letters, digits and symbols so that they are grouped together. It would have to know that 5 is before 10 is before 100, etc. It should treat upper and lower case the same. It should group all symbols together. In reality, it is more complex because different sorts have different rules. Street names are sorted with their directional prefixes together and increasing street numbers. Phone books group McDonald and MacDonald together. This requires a very complex sorting routine with a lot of switches to control the various sub-sorts. Alternative pass a comparison expression which simply determines whether a string is before, after, or equal to another string. The lambda expression is a mechanism to pass the comparison expression to a sort method.

This allows the same sort routine to sort numbers in one instance, to sort alphabetically in another, to sort in some other order (alphabetically ignoring case and numerically for digits) in a third instance. The sort routine is the same, only the type of sorting changes. The lambda expression is evaluated for every comparison needed.

To use the lambda, remember that it has two types. In the method argument list, it is a Runnable class object. When it is evaluated or run, it produces a value of another type.

The method would used the passed lambda expression in its internal processing. This can be to access a value that would change over the life of the method, like accessing the state of another subsystem during the execute method of a command.

7.18.1 Lambda Expression and Command Factories

The WPILib constructor for the **InstantCommand** is a fairly simple method. It takes two arguments: a Runnable and requirements. The Runnable is method that is run in the **Initialization** phase of a command and the other phases are basically null. The requirements is a list of subsystems that it needs. The command basically runs entirely in the expression passed as a lambda expression, but an entire command is constructed by instantiating the Instant Command.

```
command = new InstantCommand (()-> Robot.climber.stopMotors(), Robot.climber);
```

A similar command class **WaitUntilCommand** method works with a single parameter which is a BooleanSupplier, meaning that it is a method reference to a method that produces a boolean true of false value. The supplier is set to the **isFinished** phase of a command and the rest of the phases essentially do nothing. The command just waits until the condition is true and the command finishes.

```
command = new WaitUntilCommand ( ()- > Robot.intake.getGamepieceDetected());
```

These command factories make it easy to write simple commands, especially those used in command sequences.

7.19 Modules

Modules are another mechanism for doing modularization and including only the needed parts in the compiled code.

A module is declared with a **module** statement.

A package is exposed with **exports** statements.

Specific libraries can be requested with **requires**.

Anything beyond that is beyond the scope of this document.

7.20 Conventions

7.20.1 Java Naming Conventions

Many programmers use naming conventions to help identify the use of an identifier by its appearance. There are a few conventions which apply to the robot code as follows:

- Class names are capitalized to set them off from object names that are lowercase, sometimes using the same name as the class. The file containing the class definition is name for its class. For example the Robot class is defined in the Robot.java file.
- Files containing Java code should end with the extension ‘.java’.
- Files containing documentation for the remote code repository, e.g. GitHub.com or GitLab.com, should use markdown formatting and have the ‘.md’ extension.
- Directories use lower case names, unless the directory is for a class. Class names should always be capitalized even in a directory name to be consistent.

- Names for classes use upper-camel-case, in which the name starts with a capital letter followed by lower case letters and additional words added without spaces and capitalized. Names cannot contain spaces per the language syntax.
- Names for variables, attributes, methods, and objects are usually descriptive and use lower-camel-case (all lower case unless additional words are needed which are capitalized and appended without spaces.
- Names for constants or final variables have three conventions. Java likes constant names to be upper-snake-case, or all upper case with words separated with underscore characters '_'. Other constants in the WPILib code use a convention of a lower case k (for constant) followed by the camel case name (e.g., kP for the proportional factor in a PID routine). All attributes in a subsystem configuration class are constants which is obvious when the class name is dotted (member selected) to the attribute name.
- Names for handles to a method, usually a Runnable or to an object containing a Runnable as in a Consumer or Producer objects, start with a "m_" prefix.

7.20.2 Coding or Style Conventions

OK, now for the controversial side of coding. Everyone has a style that they like and they basically hate everyone else's style. If you are modifying existing code, it is generally a good idea to mimic the style that is there. The purpose of style conventions is to make your code easier to read. A good start is the Google Java style guide at https://google.github.io/styleguide/javaguide.html.

Here are some rules that I like (that may or may not be reflected in the robot code):

- Use meaningful names so that you don't have to guess or remember its use.
- Similarly, longer names that reflect their usage is better than short names like 'a' or 'b'.
- Add a space where you would pause while reading the statement out loud.
  - e.g., j=a+b should be written j = a + b.
- Add a space after an opening parenthesis when the parenthesis is really part of a name.
  - e.g., function(a) should be function(a), the parenthesis is part of 'function'
  - e.g., 'if(a<b)' should be 'if (a < b)', the parenthesis is used for grouping and is not part of a name.
- Conditional statements should ALWAYS be written the same way with the blocks indented, so the design pattern can be quickly recognized and don't have to think.

  For example, the following code:

```
if (foo) return true;return false;   //breaks normal design pattern
```

  should be written in the more regular design pattern as:

```
if (foo) {
      return true;
```

```
} else {
      return false;
}
```

or because foo is a boolean:

```
return foo;
```

It also could have been written as follows, but the else clause is ambiguous:

```
if (foo) return true; else return false;   //breaks normal design pattern
```

- Avoid using the ternary conditional operator (? :) as it also breaks the normal conditional design pattern which helps to hide bugs. What every cleverness you may feel in using it is far outweighed by the time spent debugging code using it.

```
return x==3 ? 0 : 1;   //breaks normal design pattern

return x==3;  //also so terse, it is easy to overlook
```

Don’t waste a lot of time fighting over style wars. It is somewhat personal, but it is important to adapt to guidelines adopted by the organization , even if you strongly disagree. Using the the style guild lines makes the code easier to read and maintain. Some organizations use ‘lint’ utilities to enforce their coding style guide. Messy code shows a lack of thought and care and increases the probability of introducing and hiding problems. Code that is straight forward and easy to read is usually more reliable and is much easier to maintain.

8 Robot Procedures

This section discusses various procedural things that must be performed before a robot can be used. Some of these things are simple and some require a bit more planning.

8.1 Swerve Drive Alignment Procedure

The purpose of swerve drive alignment is two-fold: make sure all the motor modules agree as to a forward direction and to make sure that alignment is aligned to the chassis. Follow the following steps:

1. Manually execute the alignment command (see 4.2.3.12 Swerve Alignment Controller).
2. Manually turn each drive module so that the gear is on the right-hand side.
3. Align the two wheels on each side using a bar or piece of lumber.
4. Note the angle of each wheel.
5. Put the angles found in step 4 in the swerve configuration file.

8.2 Commissioning Procedure

Commissioning is the procedure to align software configuration data with the installed hardware and controllers. This hits many areas of the system including, but not limited to:

- Reflashing the roboRIO with the roboRIO imaging tool.
- Subsystem configuration data including motor parameters, PID controller constants.
- CAN Bus address assignments on the CAN bus (and CAN FD bus if use on your team's robot.
    - roboRIO CAN bus ID should be 0 to give it the highest priority on the CAN bus.
    - The power distribution panel (PDP) from CTRE requires CAN bus ID 0 to do power monitoring. [believe this means device number and not ID.]
    - The power distribution hub (PDH) from Rev Robotics requires CAN bus ID 1 to do power monitoring. [believe this means device number and not ID.]
    - Ensure that other controllers have a unique ID and that the software is configured to the proper controller. Note that the ID includes all 28 bits including the device type and manufacturer and not just the last 6 bits of the device number.
    - Use one or more of the following tools to perform the checking for this step:
        - CTRE Tuner X for accessing CAN FD bus devices and CTR devices like the Pigeon. It is also used to upgrade the software on devices with CTR controllers.
        - CTRE Phoenix Tuner
        - REV Hardware Client
- Ensure that all absolute encoders are properly aligned.

8.3 Acceptance Test Procedure

An acceptance test procedure is a quick set of simple tests to ensure that all subsystems of a robot are functioning correctly and are properly connected. This procedure should be performed before every match and after every repair operation where ANY component has been electrically or mechanically detached. Mistakes happen, and this testing should be designed to catch them. It should include tests for:

- Drive station to roboRIO connection
- DriveStation control of every subsystem including all subordinate motors.
- Test that every subsystem can be driven to its range limits.
- Verify that all limit sensors work properly.
- Verify the alignment of absolute encoders.

8.4 Match Procedure(s)

Before every match there is a list of things that need to be checked and re-checked. Different members of the team have different roles and different things that they are responsible for. Each team should develop its own checklist for its own team members.

- Drive coach
    - Work with alliance partners to learn their strategies and your team's roles to implement those strategies. The plural form is used to allow a strategy to change in the middle of a match if necessary.
    - Make sure other drive team members know their roles and match strategy.
    - Don't forget defense. The team with the best defense is often the winning team.
    - Avoid conflicts with other teams (personal) and other robots (physical space) during the autonomous period.
- Pilot
    - Make absolutely sure the proper autonomous routine is selected.
    - Make absolutely sure that the correct joy stick mapping is selected.
- Operator
    - Operate the various mechanism of the robot when necessary.
    - Anticipate robot movement so the mechanism is ready when it is needed.
- Technician
    - Make sure robot has a fresh battery and that it is connected and strapped in.
    - There must be a whole check list of things to do when the robot is prepared in the pit.
    - There must be a complete check list of things to do when the robot is brought to the field.
- Human Player

- Stay awake. This is not a joke. Human players can cause unnecessary delays to a scoring cycle time which means lower scoring ability.
- Anticipate robot movements, so the delivery of game pieces can be completed as quickly as possible.
- Signal to the driver and operator successful and unsuccessful loading.

- Scouts
  - Make sure the team knows who their competitors are. What can they do and how quickly.
  - Be on the lookout for ways to improve your team's robot.
- Cheer section
  - Keep the team excited and focused on the job at hand.
- Mentors
  - Try not lose many kids as some parents may become quite upset.[17]
  - Beware of kids that their parents want you to lose.[18]
  - Make sure kids are fed.[19]

8.5 Command Binding

To execute a command on demand, each desired command must be mapped to a control on the driver or co-pilot console. This binding is critical to the timely and accurate human control of the robot. Of course this binding is subject the game's rules, the mechanisms of the robot and drive team preferences.

The following code fragment binds a joystick and one of the triggers to the swerve drive for a "drive by joystick" type of command.

```
1.  // down field and back
2.  public double getDriveFwdPositive() {
3.     return forwardSpeedCurve.calculateMappedVal(
4.                                        this.gamepad.leftStick.getY());
5.  }
6.
7.  // side-to-side across the field
8.  public double getDriveLeftPositive() {
9.      return sidewaysSpeedCurve.calculateMappedVal(
10.             this.gamepad.rightStick.getX();
11. }
12.
13. //Positive is counter-clockwise, left Trigger is positive
14. public double getDriveRotationCCWPositive() {
15.     double value = this.gamepad.triggers.getTwist();
16.     value = rotationCurve.calculateMappedVal(value);
17. return value;
```

17 Just kidding. Don't lose any kids.
18 Just kidding. Don't lose any kids.
19 Not kidding.

```
18. }
19.
20. /** Field Oriented Drive */
21. public static Command FpvPilotSwerveCmd() {
22.     return new SwerveDriveCmd(
23.             () -> Robot.pilotGamepad.getDriveFwdPositive(),
24.             () -> Robot.pilotGamepad.getDriveLeftPositive(),
25.             () -> Robot.pilotGamepad.getDriveRotationCCWPositive(),
26.             true,
27.             false)
28.         .withName("FpvPilotSwerveCmd");
29. }
```

Line by line comments:

2-4. defines the getDriveFwdPositive getter function to retrieve the left joystick y-value modified with a forward speed curve.

8-10. defines the getDriveLeftPositive getter function to retrieve the left joystick x-value modified with a sideways speed curve.

13-17 defines the getDriveRotationCCWPositive getter to retrieve the trigger value modified with a rotational speed curve.

21. This is a basic instantiation of the FpvPilotSwerveCmd object as a Command class object.

22-27. The command object is returned as a new swerve drive object with the joystick controls defined above included.

28. Note that the ".withName" is a decorator to the Command class to add the "FpvPilotSwerveCmd" name.

An example of binding a joystick controller button to a command is as follows.

```
1.  // ----- Gamepad specific methods for button assignments -----
2.  public void setupTeleopButtons() {
3.      gamepad.rightBumper.onTrue(ShooterCmds.shooterSetManualCmd())
4.                         .onFalse(ShooterCmds.stopShooterCmd());
5.      gamepad.aButton.onTrue(ShooterCmds.shooterSetHPIntakeCmd())
6.                     .onFalse(ShooterCmds.stopShooterCmd());
7.      gamepad.bButton.onTrue(ShooterCmds.shooterSetSpeakerCmd())
8.                     .onFalse(ShooterCmds.stopShooterCmd());
9.  }
```

Line by line comments:

2 declares a method to set up the buttons for teleop.

3 assigns the right bumper initial depression to be shoot manually and on release to stop the shooter mechanism.

5 assigns the A button initial depression to do the human play intake and on release to stop the shooter mechanism.

7 assigns the B button initial depression fo shoot and on release to stop the shooter mechanism.

3-8 use decorators to button commands. ".onTrue" specifies that the command is to be performed on initial depression, ".onFalse" specifies that the command is to be performed on button release and ".whileTrue" specifies that the command is to be repeated as long as the button is depressed.

8.6 Command Chaining or Command Composition

Commands may be part of a complex chain of commands where some commands control the execution of other commands using command decorators. For example a shooter may require that it come up to a minimum speed before it is fed a game piece. Some teams use incredibly complex chains of parallel and dependent commands.

The autonomous or auto subsystem issues commands to various subsystem to perform the autonomous routine.

WPILib allows for command composition. This allows single command to actually run multiple commands and may command different subsystems. For instance a command to prepare intake may:

- Run the elevator to a predetermined height
- Move the arm to a predetermined angle
- Start running the intake motor
- These commands may be executed sequentially or in parallel based on the constraints of the robot. Team 2910 has had command chains that defy understanding, but work well in practice.
- See 8.7.5 Autonomous Routine Example for an example of command sequential groups and command parallel groups.
- See 8.5 Command Binding for an example of using decorators with joystick controller button to command assignments.

8.7 Autonomous Period Programming

The autonomous period is a showcase for the skill of the programming team, and it is an excellent way to get points on the board early in a match.

8.7.1 Planning for the Autonomous Period

Teams should plan for different autonomous scenarios so that the team can easily adapt to different routines based on the capabilities and accuracy of other teams in an alliance.

- Be able to work on a Red or a Blue alliance. Make sure both sets of scenarios have been tested.
- Different possible starting positions to avoid problems within an alliance. It is surprising when a team only has one starting position (but it has to be an unlikely starting position).

- For some teams it is necessary to reset the robot Pose to the autonomous starting position. In these cases it is necessary to precisely locate and orient the robot. Small errors can lead to big problems.
- Different strategies
  - Do nothing.
  - Move out of the home area.
  - Score once.
  - Score and leave home area.
  - Score multiple times and leave home area.
  - Other game dependent strategies like moving game pieces to prevent access to them by opponents.
- Be flexible. It is surprising how autonomous events change during a competition. The software team should be aware of those changes through direct observation or reporting by scouts so that they can react and modify the autonomous routines to take advantage of (or defend against) advancements by other teams.

8.7.2 SendableChooser to Get Choices From Drive Station

The WPILib includes a data type class SendableChooser. It is used in the robot code to create a menu on the drive station to allow the drive team to select options for various things, but especially to select the options for the autonomous period. The following example show the setup for two menus with a simple selection of starting position and action desired for the 2014 Crescendo game.

```
1.   public class Auto {
2.       public static final SendableChooser<String> actionChooser = new SendableChooser<>();
3.       public static final SendableChooser<String> positionChooser = new SendableChooser<>();
4.
5.       public static String actionSelect;
6.       public static String positionSelect;
7.       private static Pose2d startPose;
8.
9.       // ----- Autonomous Subsystem Constructor -----
10.      public Auto() {
11.          configureAutoBuilder();
12.          registerNamedCommands();
13.          setupSelectors();               // Setup on screen selection menus
14.      }
15.
16.      public static void setupSelectors() {
17.          // Selector for Robot Starting Position on field
18.          positionChooser.setDefaultOption("Left Speaker",        AutoConfig.kSpkrLeftSelect);
19.          positionChooser.addOption(       "Center Speaker",      AutoConfig.kSpkrCtrSelect);
20.          positionChooser.addOption(       "Right Speaker",       AutoConfig.kSpkrRightSelect);
21.          // Selector for Autonomous Desired Action
22.          actionChooser.setDefaultOption(  "Do Nothing",          AutoConfig.kActionDoNothing);
23.          actionChooser.addOption(         "One Note",            AutoConfig.kActionOneNoteOnly);
24.          actionChooser.addOption(         "Two Note",            AutoConfig.kActionTwoNote);
25.          actionChooser.addOption(         "One Note and Crossline", AutoConfig.kOneNoteCrossOnly);
26.          actionChooser.addOption(         "Crossline Only",      AutoConfig.kCrossOnlySelect);
27.      }
28.
29.      // ------ Get operator selected responses from shuffleboard -----
30.      public static void getAutoSelections() {
```

```
31.           actionSelect =     actionChooser.getSelected();
32.           positionSelect =  positionChooser.getSelected();
33.           Robot.print("Action Select = " +     actionSelect);
34.           Robot.print("Position Select = " +     positionSelect);
35.       }
36.   …
```

Explaining the code a bit:

Line 2 and 3: The <String> specifies a data type for the abstract class SendableChooser. The diamond operator '<>' says to just used the implied data type.

Line 18-20 and 22-26: each SendableChooser can have a set of options, each with a descriptive string and an object to use if that option is selected which in this case is also a String.

Line 31 and 32 the SendableChooser are querried to to see which option was selected. The code then can use those values to control the selection of the code to run.

8.7.3 PathPlanner and Trajectory Planning

- **path:** a sequence of poses for moving the robot. In general make this as short and smooth as possible.
- **trajectory:** a timed sequence of posing for moving the robot at differing velocities.
- **Init and follow**: a path that also initializes the robot pose to a known field location and orientation before following the path where it goes.
- **follow**: a path that defines where the robot is to go.

PathPlanner can be use with teleop: This is used for commands that move to load station, move to shooting position, or move to other specific positions. It is used to assist driving to a specific location and orientation. The driver may override the path to avoid other robots. To be useful in teleop, the robot must have good odometry so that its knows its location precisely to get the destination quickly and accurately without hitting obstacles. Paths may require way points to get around fixed obstacles and some paths may need to be dynamically built.

8.7.4 Compose the Autonomous Command Sequence

While many of the commands need for autonomous routines can be share with normal teleop commands, there are some that have to have variations for autonomous operation since there is no human intervention allowed. Some commands include:

- Aim shooter.
- Bring shooter up to speed.
- Shoot.
- Stop shooting mechanism.
- Pickup game piece from floor.
- Load game piece into holding position before shooter.

- Stop loading mechanism.
- Bring shooter up to speed.

8.7.5 Autonomous Routine Example

An example of an autonomous routine for the 2024 Crescendo game follows. It shows command groups for sequential commands and simultaneous commands. It shows the use of PathPlanner to route the robot. It also shows interaction with various subsystems which is beyond this example other than to show that those interactions occurred and how they were handled.

```
1.  //---------------------- Speaker Shoot Command ----------------------
2.  public static Command SpeakerShootCmd() {
3.      return new SequentialCommandGroup(
4.          new InstantCommand(() -> System.out.println("Auto Shoot Right Speaker")),
5.          OperatorGamepadCmds.readyForBumperShotCmd(),
6.          OperatorGamepadCmds.noAutoPosSpeakerShot()
7.      );
8.  }
9.
10. public static Command followPath(PathPlannerPath path) {
11.     return AutoBuilder.followPath(path);
12.         // Return Cmd to run AutoBuilder on the path
13. }
14.
15. // Get a Command that Follows a Path
16. public static Command followPath(String name) {
17.     PathPlannerPath path = PathPlannerPath.fromPathFile(name);
18.          // Get Path from file by name
19.     return followPath(path);
20.          // Return Cmd controller to follow Path
21. }
22.
23. public static Command initAndFollowPath(String name) {
24.     // Init Robot pose from auto Selections. We can't pull
25.     // initial Pose from path because we only build blue paths,
26.     // and rely on AutoBuilder to flip path at run time as
27.     // needed for red.
28.     Robot.print("Loading path name: " + name);
29.     return new SequentialCommandGroup(
30.     new InstantCommand(
31.         () -> Auto.setStartPose()),  // Init Robot Pose to initial pose
32.         followPath(name)             // Run Path
33.     );
34. }
35.
36. public static Command groundIntakeUntilGamepieceCmd() {
37.     return new SequentialCommandGroup(
38.         PassthroughCmds.setGroundIntakeCmd(),
39.         IntakeCmds.intakeSetGroundCmd(),
40.         new WaitUntilCommand(() -> Robot.intake.getGamepieceDetected()),
41.         new WaitCommand(0.15),
42.         stopAllCmd(),
```

```
43.         new WaitCommand(0.25),
44.         PassthroughCmds.setHPIntakeCmd(),
45.         new WaitUntilCommand(() -> Robot.intake.getGamepieceDetected()),
46.         stopAllCmd()
47.     );
48. }
49.
50. public static Command readyForBumperShotCmd() {
51.     return new ParallelCommandGroup(
52.         PivotCmds.setLowAndRunCmd(),
53.         ShooterCmds.setSpeakerSpeedCmd()
54.     );
55. }
56.
57. //------------------------- Two Note -----------------------------
58. public static Command TwoNoteCmd( String pos, String pathName, String
    pathNameBack ) {
59.     return new SequentialCommandGroup(
60.         new InstantCommand( ()-> Robot.print( "Two Note Cmd ")),
61.         // Shoot pre-loaded note
62.         SpeakerShootCmd(),
63.         new ParallelCommandGroup( // move, load, spin up shooter, move
64.             // Intake note sequence
65.             new SequentialCommandGroup(
66.                 OperatorGamepadCmds.groundIntakeUntilGamepieceCmd(),
67.                 OperatorGamepadCmds.readyForBumperShotCmd()
68.             ),
69.             // First run to-note path, then run to-speaker path
70.             new SequentialCommandGroup(
71.                 initAndFollowPath(pathName),
72.                 followPath(pathNameBack)
73.             )
74.         ),
75.         OperatorGamepadCmds.noAutoPosSpeakerShot() // shoot
76.     );
77. }
```

Line by line comments:

1-57 are definition of commands. Several of the commands are nested.

1-8 is a command sequence to print a message and shoot toward the target.

10-13 is a command to follow a predefined autobuilder path.

15-21 is a command to follow a named path.

23-34 is a command to initialize the robot starting Pose and to follow a path.

36-48 shows a command sequence to load a game piece from the floor.

50-55 shows a command sequence to prepare the shooter for a shot.

57-77 defines a command sequence to print a message, shoot, init and follow a path to a game piece and back; while picking up a game piece off the floor and then spinning up the shooter; and finally taking the second shot.

9 Strategies

This is one of the most opinionated and least scientific sections in this documents. Probably everyone who works on robots (or products of any kind) thinks they "know" the "best" way to do things to develop a winning team and product. There are many factors, and it seems the best teams find a way to balance a set of factors to produce a winning strategy and are not afraid to change that strategy in mid-season, mid-competition and even mid-match. Adaptability seems to be the one thing that really sets teams and robots apart.

9.1 Basic Game Scoring

It is not enough to just say that the team who scores the most will win the match, competition or season. Yes, teams that score more do a much better job than teams that don't.

Part of the problem is deciding how to score. Do you concentrate on just points? Do you concentrate on ranking points? Just scoring points earns a team ranking points. The teams on the winning alliance each win two ranking points. That leaves two more ranking points on the table. There is a metric, ranking points per match, which measures how well teams is earning its ranking points. A perfect score is four. There are a lot of good teams earning more than three. Teams with less than two are losing more than they are winning. This metric is important. It is used to rank points during a competition as the number of matches played is almost always uneven. To make it to the playoff round takes a lot of ranking points and the teams with the highest total ranking points get to be alliance captains and get to pick who they want on their team.

Scoring points is important. Scoring ranking points are more important, because they determine which teams advance and which do not.

The games basically are broken into three parts:

- Autonomous
- Teleop
- End Game

Autonomous is a very key phase of the game because you can quickly score a lot of points. With a little coopertition with your alliance teammates, you can choose autonomous routines that optimize the scoring ability of not just your team, but the alliance as a whole. While rare, defense can play a role in autonomous routines to prevent the other team from scoring shared game pieces. In certain cases it may be desirable to take advantage of a rule that allows ordered placement of robots, so that an alliance can set up defensive or offensive moves even before the beginning of the autonomous period begins.

During the teleop period, most of the concentration is either on scoring or playing defense to disrupt the other alliance from scoring. Defense can be quite effective, but it requires great driver skills to interfere with the strategies of the other team while not incurring a foul. A great defensive driver can shut down a high scoring robot to level out the game play. In the 2024 Crescendo game, a strategy of passing started during the district championships. By the end of the World championships, many teams were doing two

passes to move game pieces down the field, mainly because the defense was so good, that robots were not able to quickly traverse the field. The teams with the best defense ended up winning.

Scoring is important. Alliances (and their teams) cannot win without scoring. Matches early in the season are determined almost solely on scoring. Robots should be able to accurately score with a short cycle time. Cycle time is the time to get a game piece, move to a scoring position, score the piece and move back to pick up another piece. Short cycle times mean that the robot can do this over and over very quickly. Inaccurate scoring generally wastes an entire cycle, although opportunistic robots will pick up the missed piece and score it rather than traversing the field.

Generally, one of the ranking points is awarded to alliances that reach some sort of minimum scoring threshold. This makes scoring points worth even more.

The end game normally has some other way to score and usually involves a fourth ranking point. It might be possible to "solo" this ranking point, but must games are designed to make this difficult to do. The idea is to promote coopertition and not be a show of unicorns (teams that can do it all).

In playoffs a team gets points for being an alliance captain (17 – alliance placement) and points for the order of selection (17 – order selected). A team that get eliminated from semifinals gets 10 points, the teams on the finalist alliance gets 20 points and the teams on the winning alliance get 30 points. (Teams have to play to earn the points, back up teams do not earn points.) Ranking points are not awarded for the accumulation bonus or for the end game in the playoff matches. Only raw score matters.

So roughly 12 matches per competition If you win all 12, that is 24 ranking points. There is another 12 ranking points for the accumulation bonus and 12 more for the end game. The following table shows the breakdown of ranking points for each of the two district events. (Teams can participate in more than two district events, but only the points from two are considered for district ranking.)

Table 36: Ranking points by category

Category	Ranking Points
Winning all 12 matches	24
Accumulation minimum	12
End game	12
Captain	8 to 16
Selection	1 to 16
Playoff	10 to 30
Maximum possible	94
Needed for PNW district cutoff	**Around 35**

There are other points for awards. This hits just the high points for focus (and ignorance of the author). Of course this generalized analysis may change based on the rules for the game of a given season.

9.2 General Strategies

- Communicate. A team that does not communicate is not really a team, but just a gaggle.

    - Mentors should coordinate their activities between themselves and to monitor student activities to keep them focused.
    - Leadership is the group of students elected to run the club organization.
        - They run the meetings.
        - They approve expenditures of the group.
        - They do the business of the club (or see that it is done).
    - Leadership should tell members of the club what is expected of them:
        - What work needs to be done by when.
        - What must be done to qualify for travel.
- Know your competition and alliance team mates with scouting:
    - Qualitative measures
        - Consistency.
        - Ways to score.
        - Reliability.
        - Ability to work well as an alliance member.
        - Leader/follower/not a team player.
        - Play innovations.
        - Autonomous routines.
    - Quantitative measures:
        - Scores of each match.
        - Cycle time.
    - Scouting:
        - Do by hand.
        - Use a scouting tool.
        - Make results easy to use as alliance captain.
- Scoring
    - Raw score counts and helps gain ranking points (usually 2 for winning).
    - Reduce cycle time:
        - Reduce unnecessary movement
        - Reduce waiting around or wasting time

        - Score at a distance
        - Score on the move
        - Pass game pieces to alliance members
        - Have a lean mean machine
    - Reduce build and repair time
        - Work for faster builds and repairs to be quicker in the lab and in the pits
        - Work for faster software development for more improvements
- Defense
    - Always be ready and willing to play defense.
    - Interfere with others ability to move.
    - Interfere with others ability to score.
    - Know the rules! Don’t do things you will regret like being where you should not be.
- Speed
    - Can the mass of the robot be reduced to make the robot faster.
    - Use path planning where possible to increase robot speed.
    - Do not spin unnecessarily.
    - Do not collide unnecessarily.
    - Is speed lost when a robot momentarily loses its balance? Can the center of gravity be lowered? Can the movements of the robot be more controlled to avoid the loss of balance.
    - Sometimes a longer path is actually faster because of less interference and longer runs of full speed.
- Quickness:
    - Has to do mostly with human interactions, like how fast can the team react to a defensive move or dropped game piece, but also has some software and physical dependencies.
    - How well are moves anticipated to reduce the time to wait for a game piece to be dropped?
    - How quickly can a game piece be recognized as loaded and the robot moving on to the next position?
    - How quickly can a team transition from offense to defense when part of the robot malfunctions?
    - How quickly can a team react to a new strategy imposed by an opposing alliance (or in a preceding match)?
    - Can drilling toward specific goals reduce reaction time?

    - How can software improve reaction time?
- Match Negotiations:
    - Don’t overlook cooperation to increase scoring in competitions.
    - Establish lanes and roles to avoid interfering with partners.
    - Assign and keep task assignments.
    - Be willing to take direction from alliance partners.

9.2.1 Avoiding Contact

If you want to be fast and have a short cycle time, avoid contact. This doesn't mean avoid contact at all costs. Team 2910 Jack in the Bot has been effective at a bump and run defense which would disrupt the other alliances, while carrying on its primary mission of scoring with a low cycle time.

The takeaway is that if you must contact, make the contact as short of a time as possible. Don't get blocked in. Don’t foul.

9.2.2 Drive Defensively

Just like when you drive a car, anticipate what the other robots are doing, so that you don't get blocked in, or so that you avoid them blocking you. Always have a path out. This needs to be communicated between driver and coach.

9.2.3 Lane Selection

Good teams establish lanes with their alliance partners so that they can move back and forth and not interfere with each other. This was most important in the congested areas of the scoring grid and the substations. Rather than wait around to gain access they had open lanes for entering and leaving these areas. Sometimes the lanes were dynamic when an alliance member or opponent would block the desired lane, an alternate lane would be chosen by the coach and driver to minimize travel time.

9.2.4 Automation

Certain things like pickup from ground, pickup from station or placing a game piece can be somewhat automated with the addition of vision processing software. A simple sensor may be just to tell the robot that a piece has been loaded or unloaded. This sensor can be used to end one process and begin another to give the robot some inhuman quickness.

9.2.5 Efficiency

Besides staying in lanes and avoiding contact, avoid unnecessary running of motors to keep the voltage higher so that the drive chain has more power available for movement. Some team waste movement in its turret and twisting of its swerve based chassis. A pneumatic compressor must run most the time, placing a load on the battery.

9.3 Reliability

The reliability of a robot can be measured by the number of minutes or number of matches missed. This measure should include the type of breakdowns:

- Something in the intake or delivery system, robot still able to perform in defensive roles.
- Something it the robot sensors.
- Something in the drive chain rendering the robot immobile.
- Lost communication so nothing can be done including troubleshooting.
- A disqualification resulting in removing power from robot.
- A communication breakdown in the drive team.

Software can help with reliability. A stalled motor can be detected and then fed less power to decrease the probability of burnout. It was desirable to maintain enough power to hold a game piece, but not enough to burn out the motor. Software interventions should be visible to the technician and to the pit crew, so that they can make modifications as they deem necessary.

9.3.1 Simplicity Counts

Murphy's Law states that if anything can go wrong, it will. Simplicity eliminates the number of things that can go wrong, thereby improving reliability (for all things equal).

Avoiding complexity has another benefit. It allows the team to focus and refine the capabilities its robot has. Doing a few things very well is much better than trying to do a lot of things and being mediocre or worse in all of them.

Simplicity also reduces the number of **points of failure**. If you have a wire between two points on the robot there are three points of failure: the wire itself and the two connections. Most teams want to add at least one connector to make it easier to disassemble and repair. That adds tree more points of failure for each connector. Add crimp on ferrels to wires can add another point of failure per ferrel. While ferrels tend to strengthen the wire and prevent it from breaking at the terminal, improper crimping can allow the wire to be pulled out of them. Solder the wire to the ferrel to remove that point of failure. Bottom line is to think about every component that you add to the robot because each one adds one or more new points of failure.

9.3.2 Mechanical Soundness

Mechanical soundness refers to how well mechanical connections are made. Can parts shake loose? Murphy says they will. What is lost if something shakes loose? Try to minimize the effect of any failure. As a general rule, if it doesn't look right or feel right, it probably isn't right.

9.3.3 Electrical Soundness

A robot may be properly wired, but connections are lost during a match. This is usually due to some force being applied where it should not be and catching a wire to force a connection apart. In general wiring should be tightly bundled so that it is mechanically stable and that nothing can catch a loose wire. Likewise conductors should be routed so that they are not subject to any rubbing or grinding contact.

9.4 Defense

- Brings good teams down so they operate on your terms
- Increases contact which may increase breakdowns and the probability of a disqualification.

- Can be simply a bump and run. Just slow down a competitor or force a robot out of its lane to make the drive team react.
- Should never interfere with an alliance member

9.5 Human Element

In robot competitions, the human team members have roles that are as important or even more important that the robot engineering. These roles are discussed below.

9.5.1 Pilot

The main thing the pilot is concerned about is moving the robot and decreasing any lag time. Defensive driving should be on mind to avoid disruptions and well as possible defensive moves to disrupt opponents. Smooth driving is preferable to jerky driving to reduce shock and vibration to the robot. Speed and quickness of the driving is all important to reduce cycle time. Being out of control or reckless is harmful to the robot and its performance.

9.5.2 Operator

The main concern of the operator is to control the scoring features of a robot like the elevator, arm and intake device. The operator should also be aware of the height of center of gravity and work to keep it as low as possible during the match.

9.5.3 Coach

The coach is the eyes of the team. The coach has to decide what piece to score next in cooperation with the other teams on the alliance and to help keep the robot in a lane to avoid conflicting with alliance partners.

The coach is aware of the score, the ranking points scored and of the time remaining and the ability of the robot to perform as these aspects will control which activity the robot should do next (defense, attempt to score, or engage with the docking station).

9.5.4 Human Player

The job of the human player is to deliver game pieces to a robot in a manner that does not slow down the robot. Pieces have to be ready and presented in a way desirable to the robot. Bobbles by the human player add seconds to the cycle time.

9.5.5 Technician and Pit Crew

The technician on the field should watch the robot for any problems that will need to be corrected in the pit. The technician may also observe opportunities for improving the robot in the pits or when at home.

9.5.6 Scouting

Train the scouts on our own robot by attending practice sessions and critiquing the robot and drive team performance. Find faults with it or the way it is driven so these things can be corrected. Find areas where the robot can be improved. (sloppy driving, human mishandling of game pieces, missing of game piece pickup or delivery, maneuvers that make the robot tipsy or lose balance.) Remember that these observations are not personal, but are intended to improve the performance of the team and the robot.

Measure the cycle time with a stop watch capable of doing splits. This is the fastest way to separate the great robots from the good robots. Even on a shortened practice field this measurement provides a baseline that should be beaten.

Some teams like to use statistics to help in their alliance choices. The easiest statistic is the team rank in the competition. During the qualification rounds this is calculated by ranking teams in order of their average ranking points per match. This evens out teams which have played different numbers of matches. The problem with this measurement is that it does not reflect a particular team's contribution to the score. There are two statistics that attempt to suss that out: OPR and DPR. OPR, offensive power rating, reflects a team's contribution to their matches scores. DPR, defensive power rating, reflects the team's contribution to the opposing team's score, or basically does a team defend well enough to affect the opposing team's score. These may help in determining which team's score more (or at least help their team score more) or defend better. More information can be found at https://blog.thebluealliance.com/2017/10/05/the-math-behind-opr-an-introduction.

The human part of robotics is also important, but is hard to quantify. Does your team get along with the other team? Does the other team complement your team? Will the other team do what it takes to help the alliance win?

9.6 Fairness of the Competition

FIRST tries to make sure that the contest is a level playing field by regulating size, weight, battery and types of motors. But it cannot regulate the experience, training and desires of the teams and their mentors. It also does not regulate how much time a team may spend on their robots in the off season, build season or competition season. This time can be used to refine designs and to practice until every contingency is automatic through muscle memory. So some teams have better access to fabrication technology than others teams. Some teams have more mentors with a wider variety of experience and expertise. That may make it harder for a small team to be one of the best teams, but it certainly does not prevent you from being the best team you are capable of becoming.

9.7 Just Saying…

These are some sayings picked up from district T-shirts and years of engineering practice. Some may sound contradictory, but in balance most are good things to keep in mind.

Constant Incremental Improvement. Make small changes that are easy to integrate and test before moving on to the next improvement. The industry calls this "agile" or "cascade" development.

Don't Bite Off More Than You Can Chew. This is just another way of saying to compete at the team's level of competence. Make sure that you can finish what you start and don't take on projects that you don't have a reasonable expectation of finishing. A working suboptimal robot is a lot better than a broken optimal robot. Put this another way, say a team has ten units to apply to subsystem. Four units would make a perfect subsystem. Does the team have two perfect subsystems and one mediocre? One perfect and two nearly perfect subsystems? Three nearly perfect systems? Five mediocre subsystems?

Whatever It Takes. If a grunt job needs to be done, do it. Don't be so proud that some tasks are beneath you, either as a team member or as a team as a whole. Sometimes playing defense is necessary to win, so you play defense. It is necessary to clean up the work area, so that the next person can work effectively and not have to work around a messy and unsafe work space. Do the job quickly, without complaint or hesitation. **Just Do It.** This attitude applies to everything:

build, driving, defense, homework, loading in, loading out, being there on time, being attentive to team meetings and to other team's design, effectiveness, methods, strategies, etc.

Fail Fast—Fail Often. Build software so that it fails early. In general is better to discover problems early and correct them. Write code so that it does not mask problems. Don't hide missing inputs by using meaning defaults. Make sure the inputs are there, and if they aren't fix it. It is OK to detect a problem and have a mode to limp along, as long as there is notification in the log and/or drive station.

KISS—Keep It Simple Stupid[20]. This basically reminds you to not overthink a design. A simple solution will be more reliable than a more complex solution. A simple solution is easier and quicker to implement than a complex solution. Spend a little time to consider alternatives when a solution seems too complex, it probably is.

Better is the Enemy of Good Enough. This is an admonishment for creeping featurism or the desire to add more and more features and functions into the robot or product. Focus on the job at hand and be wary of things that are distractions. Features or improvements can easily take so much time and effort that the main mission is not achieved. This can involve the spreading of resources which results in a mediocre performance. This works with incremental improvement: a bunch of small changes is better in general than a big change.

The Worst Decision is Indecision. Putting off a decision always means delay. Many "bad" or "suboptimal" decisions can be lived with, and a truly bad decision can be corrected.

Murphy's Law: Anything That Can Go Bad, Will Go Bad. If a loose wire can catch something, sooner or later it will. If something cannot take the impact of a collision, it eventually will break. If a team member can be distracted, they will be distracted. The thing is to anticipate problems and correct the cause, before the problem has a chance to occur. You may not be able to nail down all potential problems; but the more that you do, the more reliable your team and robot will be. A corollary is that when they go bad, it will happen at the worst possible moment.

If you don't design to win, you won't. Good things don't happen by accident, they have to be designed in. This is things like reliability, speed, repairability, flexibility, robustness, etc. If there is a characteristic that you want in a robot, the robot has to be designed with that characteristic from day one. It cannot be patched in.

Perfect practice makes perfect. Many people say practice makes perfect. That is not necessarily true. If the practice reinforces bad habits and mistakes, it actually makes performance worse. The goal of perfection can be achieved with perfect practice. Practice build up muscle memory. Practice will reduce reaction time only if the practice is perfect.

Be as simple as possible and no simpler. (Attributed to Albert Einstein.) Simplify your designs while achieving their requirements.

Don't ASSUME because it makes an **ASS** out of **U** and **ME**.

Quick and dirty is never quick and always dirty.

20 Some have changed "stupid" to "silly." This misses the point entirely. It is not silly to have a complicated unreliable design, it is a sign that the problem has not be thoroughly thought through. Silly is putting a fun graphic on a robot. It doesn't affect the capabilities of the robot, but it make people smile or laugh. Using the word "stupid" is to remind everyone of the goals, not to insult or poke fun at people or teams who miss opportunities to simplify.

10 Character Encoding and Text File Formats

As a general statement most text files are encoded with a string of ASCII (American Standard Code for Information Interchange) developed in the 1960s. It is a 7-bit code that represents upper and lower case letters, numbers, some punctuation marks and has some control characters for early terminals and teletypewriters. A lot has happened in the last 60 years. Computers have become widespread, not just in North American but throughout the world. ASCII did not even handle English well as it did not do accents and diacritical marks.

Another failing was in the control characters. Teletypes required a line of text to be terminated with a carriage return character (which told the teletype to move the carriage to left side of the page and took a second or more) and a line feed character (to scroll the page up one line). Unix, the operating system on which Linux and Apple Macs (since version 10) are based upon, stores a single line-feed character to mark the end of a line. Windows keep the teletype standard when storing characters in a file. Some other operating systems, including Macs before version 10, use only a carriage return character. This used to lead to a lot of incompatibilities between operating systems, also now everyone can accept the others. Additionally, some operating system and programs used the escape character to signal an unstandardized extension of the character encoding.

Some accent problems were solved by inserting backspace characters to back the carriage up one position and the accent tick mark could overwrite the previously written letter. This didn't always look good, but it was a workaround. It drove line printers nuts, because they didn't know how to handle a backspace. They printed an entire line at a time, so the output had to be buffered as multiple lines without an intervening line feed.

Both Macs and Windows introduced characters using the eighth bit. Of course these uses were not compatible, and the character sets didn't map to each other. Neither attempted to include Asian characters.

Over the years the Unicode Transformation Format evolved.. It has four standard formats:

- UTF-8 which is widely used on the Internet to encode pages. It allows the normal 7-bit ASCII characters, but it uses the eight bit to signal an expansion byte or bytes to follow the first 8 bits. This allows a character to take one byte, two bytes or four bytes as it needs, while using mostly 8-bit characters. It can encode all the world's living languages in its two-byte format. The four-byte format is expanded and expandable for hieroglyphics, mathematics, symbols, and emoji.
- UTF-16 which uses 16-bit characters and can expand to 32-bit characters. UTF-16 is the native format used by Java characters (char).
- UTF-32 stores characters in a fixed length 32-bit format.

The main thing to remember is that different operating systems handle characters in different ways. If you see some weird characters, it may be that somewhere along the line of processing the, the character set was not applied appropriately.

11 JSON Syntax

JSON or JavaScript Object Notation is a commonly used standard for exchanging data between systems. It is based on the data elements used by JavaScript and formalized by a standard ECMA-404. It has a collection of name-value pairs, and it has an ordered list (array) of values. These translate into data structures used by most common languages.

The JSON language is explained below. On the left is a text-based using the Extended Backus Naur Form (EBNF/BNF) syntax for describing a computer language syntaxes with increasing levels of granularity[21][22]. On the right are railroad track diagrams showing the same thing in a graphic format.[23]

```
json = [ object eof ]

object = [ '{' members? '}' ]

members = [ pair [',' pair]* ]

pair = [ string ':' value ]
```

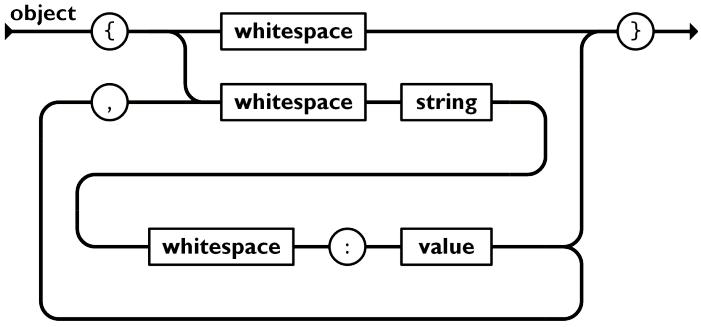


Figure 116: JSON Object and Name-Value-Pair Railroad Track Diagram

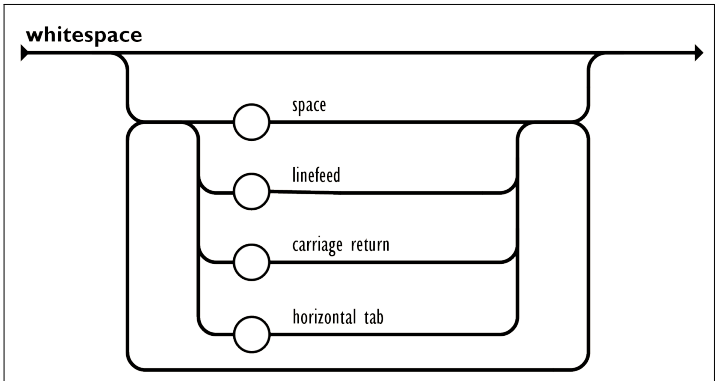


Figure 117: JSON White Space Railroad Track Diagram

21 iamwilhelm.github.io/bnf-examples/json
22 BNF is used to design and build language compilers and interpreters to unambiguously translate the language into its intent. The interpreter can be implemented as a finite state machine where there is a state at every decision point in the language, and the transition to the next state (track) is determined by the next symbol encountered.
23 JSON.org

```
value =
[
    string
    | number
    | object
    | array
    | "true"
    | "false"
    | "null"
]
```

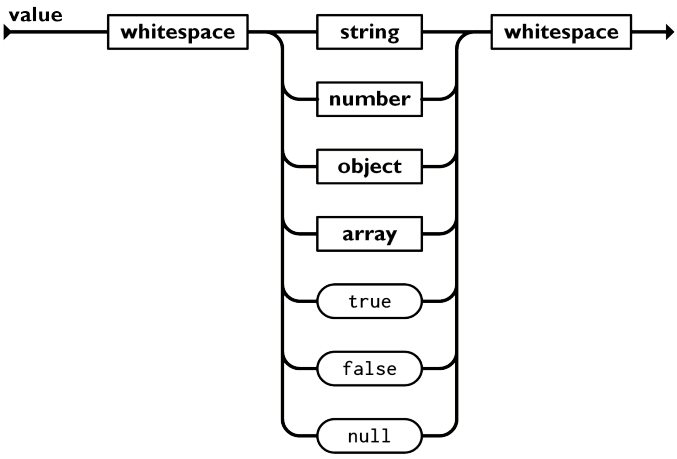


Figure 118: JSON Value Railroad Track Diagram

```
string = '"' [ char [ char ]* ]
'"'

char = any unicode character
except " or \ or control-
character
       | '\"'
       | '\\'
       | '\/'
       | '\b'
       | '\f'
       | '\n'
       | '\r'
       | '\t'
       | '\u' + four-hex-digits
```

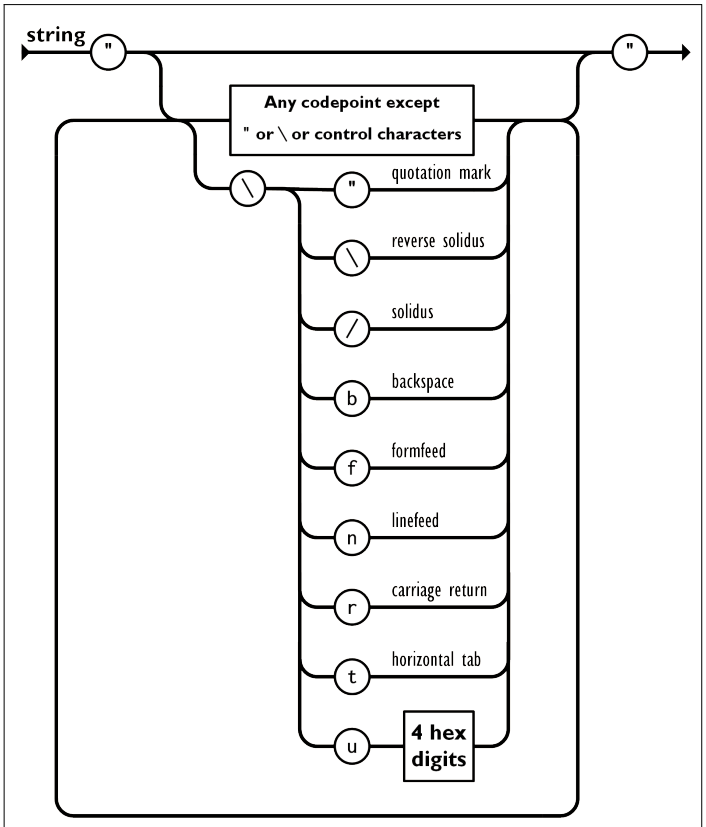


Figure 119: JSON String Railroad Track Diagram

```
array = [ '[' elements? ']' ]

elements = [ value [','
value]* ]
```

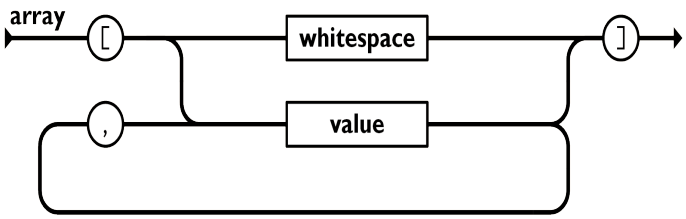


Figure 120: JSON Array Railroad Track Diagram

```
number = [ @ignore("null")
[int frac? exp?] ]

int = [ '-'?
    [
        digit1_9s
        | digit
    ]
]

frac = [ '.' digits ]

exp = [ e digits ]

digit = [ '0'..'9' ]

digit1_9 = [ '1'..'9' ]

digits = [ digit+ ]

digit1_9s = [ digit1_9
digits ]

e = [ ['e'|'E'] ['+'|'-']? ]
```

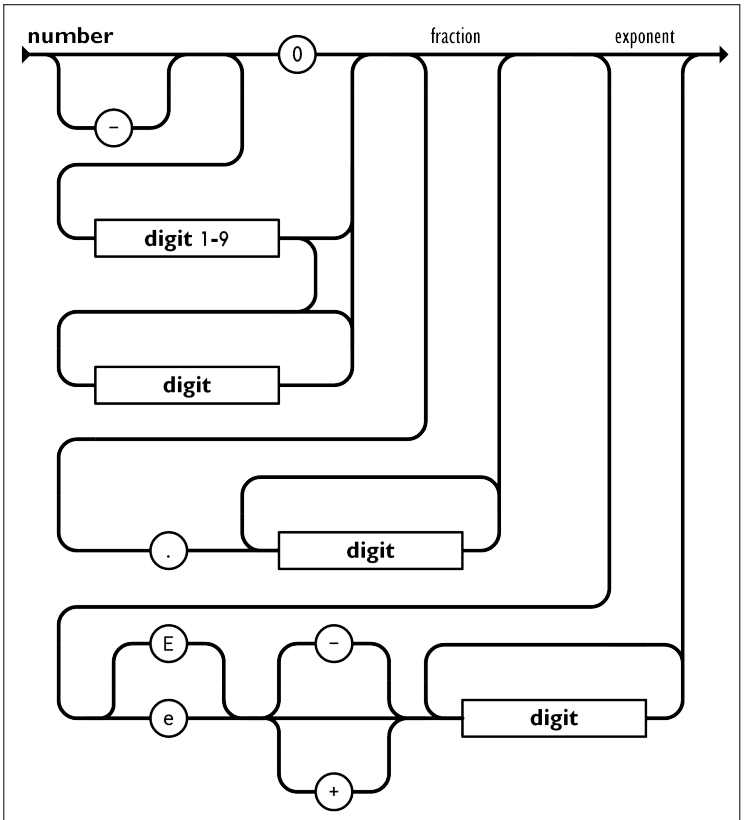


Figure 121: JSON Number Railroad Track Diagram

12 Further Ideas

This section is for half-baked ideas that may or could be implemented on FRC robot.

12.1 Reading Pitch and Yaw Angles from April Tags

Since April tags have zero roll and zero pitch, differences in the lengths between the top and bottom segments indicate a tag-to-camera pitch angle. Differences in the lengths between the left and right side segments indicate a tag-to-camera yaw angle. See Figure 122 and Figure 123. (The longer segment is closer to the camera.) If the vision system would deliver these angles accurately, the dependency on the gyroscope for the robot yaw angle would be lessened.

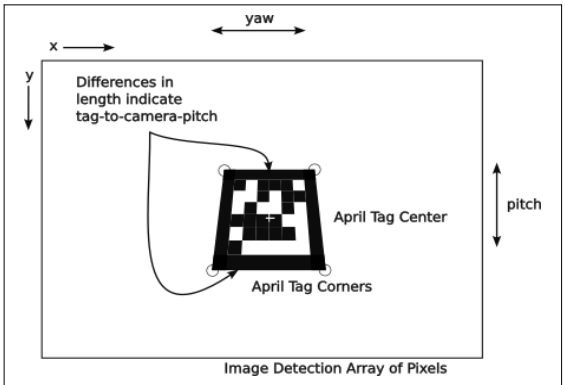


Figure 122: Image frame of reference with tag showing distortion due to tag-to-camera pitch

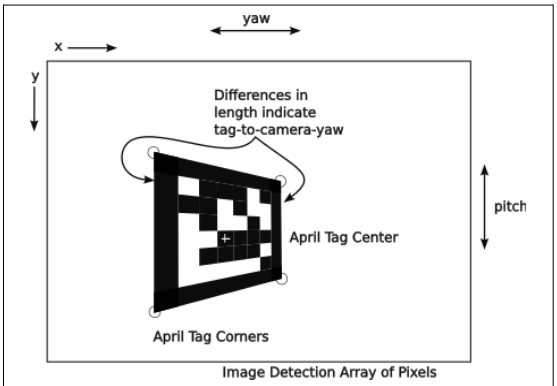


Figure 123: Image frame of reference with tag showing distortion due to tag-to-camera yaw

12.2 Collision Detection

Collisions can be detected by sudden large signals coming from the Inertial Measurement Unit (IMU). It might be useful to attempt to determine the momentum lost in the impact ($E = Mv^2$) and its direction with respect to the robot. Anecdotally, the IMU sensors become overwhelmed in impacts and are not of much use.

12.3 Practice Fencing

Since many smaller teams do not have a full field to practice on, it would be useful to have a way to fence the movement of the robot to prevent the robot from leaving a training area and injuring people or property. A simple rectangular fence could be built using the odometry data to disable the robot if it goes outside of the defined rectangle.

$$FieldFenceMax_x > FieldRobot_x > FieldFenceMin_x$$
$$FieldFenceMax_y > FieldRobot_y > FieldFenceMin_y$$

This could use the testing mode, so that it is not activated during competitions.

Updating of the robot's odometry with April tags would be required, and the proper April tags would have to be in their proper places on field.

There could be two fences set up to represent Red or Blue field positions mapped onto the practice field.

12.4 Battery Monitoring

Some teams maintain a history of battery use to keep track of how many times and when a battery is charged, used and stressed. The idea is to find and use the best batteries in a team's inventory for competition matches, especially the playoff matches.

The basic idea is to have a log or database recording various things about battery usage:

- Battery Identification:
  - When was battery purchased.
  - When was battery put into service.
  - What is its identifier: number, barcode, QR code, etc.
  - Where was battery purchased.
- Charge cycles
  - When.
  - How long.
  - Voltage measured at the end of charging.
  - Battery capacity, if available.
  - Any other measurable quantity.
- Reconditioning cycles:
  - When.
  - How long.
  - Voltage measured at the end of charging.
  - Battery capacity, if available.
  - Any other measurable quantity.
- Matches (practice and competition):
  - Discharge rate over time.
  - Capacity at the end of the match.
  - Duration of brown out condition.
  - Rapid discharges (rate and length of time for each occurrence).
  - Discharging time.
- Summary:
  - Hours of usage:

- Hours of charging.
- Number of charge/discharge cycles.
- Number of abusive discharge cycles.
- Etc.
- Some way to quickly visualize usage and problems over its lifetime.

12.5 Software Current Limiting on Individual Motors

Some motors will stall and use enough current to burn themselves up without tripping the breaker. This may be due to an oversized circuit breaker. Assuming that the breaker is OK, it would be great to detect the stall condition (with shaft encoders or perhaps less reliably with high current use) and do something to limit the current available to that motor. This would have to work with the motor controller as there is no external place for limiting the current used by a motor.

It is rumored that some teams do this in brown out conditions, so that all motors get some current, but not all of the current that they want. The idea is to keep on trucking even at a reduced speed and keep everything, but especially the roboRIO working.

12.6 Use the IMU to do Inertial Navigation

Theoretically inertial navigation can be used to determine the location of the robot on the field free of the inherent errors of odometry and kinematics. The accelerometer values would need to be integrated (accumulated) to find the velocity in each axis. These velocity values would have to be integrated (accumulated) again to find the x, y and z distances traveled and the current x, y, and z position on the field. The problem is that integration (repeated accumulation) sums many small increments, and each increment has a noise error component. These errors accumulate in the result. This would require periodic resetting of the position at known field positions or by using April Tags. Quadcopters are able to use this to return to their starting point. It is believed that no FRC robot uses this technique.

12.7 Missing Topics (TODO list)

Marnie's list

- sensor types: IR prox, lidar, sonar, color, …
- ~~Indicator lights key for roborio and (sparkmax) motor controllers.. and phoenix~~
- cables needed
- ~~software tools~~
  - ~~roboRIO imaging tool~~
  - ~~Phoenix Tuner~~
  - ~~REV Hardware Client~~
    - ~~Burn Flash to same changes~~
- drive station
- ~~carrying software between seasons.~~
- More detailed suggestions from the 3024 manual
- ~~PID tuning steps and diagrams~~
  - ~~https://pidexplained.com/how-to-tune-a-pid-controller/~~
- ~~torque and force… maybe still think about gearings a bit~~
- ~~electric board diagram? … same as schematic?~~
- ~~lock wheels in an X~~
- ~~OPR and DPR as indicators~~

- ~~style = google style guide~~

My list:

- Interface for logging everything (AdvantageKit) …added references to the github page~~.~~
- discussion about transforms and inverse transforms
- look at last copy ???
- System identification… WPI process
- Fix CAN bus protocol issues
  - picture of CAN BUS 2A packet with 11-bit ID (OK in current)….drop this
  - picture of CAN BUS 2B data packet with 29-bit ID extended …. this should be the right one
  - distinguish between 11-bit and 29-bit ID
  - fix figures…
- make the figures reasonably consistent
- ~~Build out the abstract class and methods a bit more ….need to roll it in and integrate it more~~
- ~~Use location fence to war a driver of out of bounds conditions~~
- ~~page of electronic symbols and use the word schematic for circuit diagram~~
- ~~command factory ≠ software factory, make this correction~~
- ~~command composition. Look at the variations allowed in the command class~~
  - ~~have a dictionary of the command possibilities~~
- ~~show the example of the 2910 code, just for grins.~~
- ~~make sure that ther is a .png file for every figure and a .svg file for every .png (a .svg file may generate several .png files.)~~
- ~~make sure every figure has a caption~~
- ~~Strategic design … design to win~~
- ~~strategic design: points of failure~~
-
- ~~unit conversions …added section~~
- ~~external libraries~~
  - ~~AdvantageKit~~
  - ~~naVx~~
  - ~~PathPlanner~~
  - ~~phoenix5~~
  - ~~phoenix6~~
  - ~~PhotonVision library~~
  - ~~WPILib new commands~~
- ~~Logging metadata~~
- ~~talk about Euler angles, especially with vehicle orientation and quanterians~~.
- ~~Built up the discussion about threads more~~
- ~~Hardware Abstraction Layer (HAL)~~
  - ~~a black box technique.~~
  - ~~Used for simulation software~~
  - ~~also used by AdvantageKithex = hexadecimal in reference to base-16 digits~~
- ~~control theory: open loop, closed loop, algorithms, feed forward~~
- ~~restriction on Java names: {letter}{letter|digit)*~~
- ~~first character could be _ or $, no special meaning applied in Java as in other languages~~
- ~~explain commutator better with a drawing of the split.~~
- ~~Feed forward term for control loops: gravity term for elevators, sin()gravity term for arms~~
- ~~basic time management:~~
  - ~~list of tasks~~
  - ~~estimate of time~~
  - ~~estimate of resources~~
  - ~~dependencies~~
  - ~~work days and holidays~~
  - ~~Gantt chart~~

    - ~~reality check.~~
    - ~~Progress check~~
- ~~Explain solenoid and relay.~~
- ~~array notation for vectors~~
- ~~footnote to use gyro yaw to locate robot~~
- ~~atan vs atan2 p50 2.3.13~~

12.7.1 Document Production Checklist

- ~~Do a spell and grammar check pass~~
- ~~Update table of contents and index.~~
- ~~Merge similar entries in index.~~
- ~~Update vocabulary~~
- ~~fix headers and footers~~
- ~~Spell check entire document~~
- ~~check pagination~~
- ~~update table of contents and index~~
- ~~Print it out and review it one more time~~
- Review TODOs to see if any can be quickly fixed.
- Edit in the comments from above.
- ~~Set up project on github to allow commenting and ticket filing~~
- Post images to 4513 GitHub site.
- Complete current pass of changes.
- Spell check entire document.
- Check document for index entries
- Grammar check document.
- Check pagination
- Update table of contents and index
- Make .pdf version and post to 4513 Git Hub.
- Print it.
- Bind it.
- Distribute it for review.

13 Vocabulary and Abbreviations

AC is abbreviation for alternating current, like the power found in the home, school or office. This type of power changes its frequency many times a second, 60 cycles per second in the US and Canada.

AND is a boolean logical operation which returns true if all boolean operands are true.

annotation in Java is a directive to the compiler to modify its normal operation as in @override or @interface or @FunctionalInterface. It can also be used as a directive to javaDocs.

API see Application Program Interface.

Application Program Interface is a set of methods provided by a package that can be used when that package is imported to a program. Documentation describes what parameters a method expects and what results are returned.

argument a positional parameter passed to a method, function or subroutine.

ASCII an abbreviation for American Standard Code for Information Interchange.

attribute a variable associated with the instance of an object.

azimuth is the angle about the field z-axis aligned with the x-axis of the robot. This angle is the same as the yaw of the robot.

Back EMF a electromotive force or voltage generated by a spinning motor or collapse of the magnetic fields around a coil or inductor.

Backus Naur Form is a syntax for representing the syntax of computer languages used in the construction of compilers and interpreters.

Baud is a unit of measurement for symbols per second used in data transmission across some medium.

binary is a number in base-2 expressed with 0 and 1 digits.

binding in the context of WPILib is the linking of a joystick button to a robot command.

bitwise refers to logical operations that use bits in the same position in a value and return the result in the same bit position.

block a group of statements bound by curly braces ‘{…}’. Sometimes a block is one of several pieces of wood used to support a robot above the carpet so that simple movement commands can be safely tested without the robot moving in some unexpected way.

BNF is abbreviation for Backus Naur Form.

boolean a value that can be either **true** or **false**.

bus a connection topology where all connections drop from a common line. Within a processor there is a data bus and and an address bus. In FRC robots there is a CAN bus and may be a CAN FD bus.

byte is a collection of eight bits or an 8-bit binary number.

CAD is an abbreviation of Computer Aided Design (used in mechanical designs) and sometimes Computer Aided Drawing (used in measured drawings like the FRC field drawings).

CAN bus is the Controller Area Network used for automobiles and FIRST robots.

CAN FD bus is the Flexible Data version of the CAN bus with higher transmission speed, longer address space to accommodate more CAN devices on the bus and longer messages. CAN FD is not compatible with CAN, so you cannot mix CAN and CAN FD devices on the same bus.

char is an abbreviation for **character** and a Java primitive data type for a single character.

controller a device that controls something. For example, this can be the software or device that controls a motor.

constructor a method of a class that is used to declare and initialize the attributes of an object of that class.

co-pilot is the human who controls accessories on the robot. Also called the **operator**.

CTRE is an abbreviation for Cross the Road Electronics, the maker of many components used in FRC robots.

DC is an abbreviation for direct current, the type of power furnished by a battery that does not change its polarity.

declare define the name and type of a variable or attribute.

decorator modifies the behaviors of object instances of a class at runtime without affecting other objects of the same class.

DHCP is an abbreviation for Dynamic Host Configuration Protocol used by some network routers to assign IP addresses to new entities joining a network.

DNS is an abbreviation for **Directory Name Service**, which is an Internet service for translating common network resource names, like google.com, into an underlying network address, like 127.0.0.0.

driver 1. the human driving the robot and is also called a pilot. 2 the low level code used to control a piece of hardware.

DPR or **Defensive Power Rating** is a statistic that infers a team's contribution to preventing the other alliance from scoring.

EBNF is abbreviation for **Extended Backus Naur Form**, an enhanced version of the original Backus Naur Form used to describe computer language syntaxes.

EMF is abbreviation for an **electromotive force** is a force that moves electrons. Same as a voltage.

encapsulation allows names to be reused without conflict in different modules. Part of this is controlled with the limited scope of an attribute or method, but these can still be accessed by specifying the containing object name, a separating dot or period, and the attribute or method name.

exclusive OR is a boolean logical operation that is true when the boolean operands are not equal and false when they are equal.

FIRST is abbreviation for **F**or **I**nspiration and **R**ecognition of **S**cience and **T**echnology.

FMS is the Field Management System which allows the field referees to control robots for simultaneous starts and safety.

FPGA is an abbreviation for **Field Programmable Gate Array** an integrated circuit consisting of a set of logic gates that can be configured in the field to provide dedicated high speed logic functionality. This is sometimes called a **Field Programmable Logic Array** or **FPLA**.

frame of reference is a coordinate system for locating an object within its bounds.

FRC is abbreviation for **F**IRST **R**obotics **C**hallenge

function is a routine or method that returns one or more values.

GPS is an abbreviation for **Global Positioning** System, an American satellite based system that determines the location of a receiver on the surface of the earth, usually a latitude, longitude and altitude, but it can use other frames of reference. The more general term is **Global Navigation Satellite System** (**GNSS**) that applies to systems of all countries.

Git is a source code management program used for keeping track of changes to source code and accepting changes from multiple sources. This can run on a local development machine or on a remote commercial server such as those provided by GitHub.com or GitLab.com

gradle is a progam for managing the compiling of various source codes and deploying the code to the target machine, the roboRIO in FRC robots.

handle is a attribute that holds a method reference or reference to an external object. The names of handles begin with "m_."

heading is the angle that the robot is moving toward with respect to the field.

Henry is a unit of inductance.

Hertz is a unit of frequency that corresponds to cycles per second.

hexadecimal is a representation of base-16 numbers using the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E,and F (the letters can be upper or lower case).

hex 1) a spell by a witch or warlock 2) hexadecimal.

holonomic in FRC robotics is a robot that can move in any direction like robots with swerve or Mecanum drives. **Non-holonomic** robots can only move forward and backwards and include robots with differential, tank and West Coast drives.

home run is a connection topology where all connections to devices is from a common point or device. Power distribution on an FRC robot is an example of home run wiring. This is also called a star topology.

Hz is an abbreviation of **Hertz**.

ICMP is and abbreviation for **Internet Control Message Protocol** that is used to test network reachability with network ping commands.

I²C, **I2C**, or **IIC** bus or **I**nter **I**ntegrated **C**ircuit bus is a short distance control bus used by some microcontrollers and devices.

IDE is an abbreviation for **I**ntegrated **D**evelopment **E**nvironment, a suite of software development tools accessed with a single interface, usually editor, syntax checker, compiler, debugger, and deployment tools.

IMU is an abbreviation for **I**nertial **M**easurement **U**nit.

inheritance a class can inherit the attributes and methods of a parent by extending a class. The new class can add additional attributes and methods. This is done with the Java keyword **extends**.

initialize set the initial value of a variable or attribute.

instantiation the creation of an object from a class template.

internet is a local private data network using the internet suite of protocols.

Internet is the global public data network using the internet suite of protocols.

intranet is a local private data network using the internet suite of protocols.

int is an abbreviation and Java data type for **integer**.

IPv4 is an abbreviation for **I**nternet **P**rotocol **v**ersion **4**, usually used in reference to network addresses.

IPv6 is an abbreviation for **I**nternet **P**rotocol **v**ersion **6**, usually used in reference to network addresses.

ISO is an abbreviation for the **I**nternational **S**tandards **O**rganization.

Java is a portable object oriented language that is strongly typed originally developed by Sun Microsystems.

JavaScript is commonly supported by browsers. It is an auto-typed language and supports some object oriented concepts. It has some aspects common to Java, but it is not derived from Java.

JSON is an abbreviation for **J**ava**S**cript **O**bject **N**otation a protocol for transferring structured information.

k is abbreviation for kilo or 1000.

K is abbreviation for a binary kilo or 1024. A K is bigger than k.

$\mathbf{K_d}$ is a constant for controlling the differential component of a PID algorithm.

$\mathbf{K_i}$ is a constant for controlling the integral component of a PID algorithm.

kinematics is the measurement of the motion of a body based on the motions of its locomotion (e.g., feet or drive wheels).

$\mathbf{K_p}$ is a constant for controlling the proportional component of a PID algorithm.

LimeLight is the brand name of a FIRST robotics vision coprocessor.

LimeLightOS is the brand name of a proprietary vision processing system for detecting and processing April Tags and other objects.

operator 1. is the human who controls accessories on the robot. Also called the co-pilot. 2. is a mathematical or logical function for manipulating values.

LED is an abbreviation for **L**ight **E**mitting **D**iode.

m is abbreviation for milli or 0.001.

M is abbreviation for mega or 1,000,000. It is also used for the binary mega or 1,048,576 used by some manufactures when reporting on memory or disk drive capacities.

MEMS is an abbreviation for **M**icro **E**lectro **M**echanical **S**ystem used to build integrated circuits for 3-axis accelerometers and three-axis gyroscopes used in inertial measurement units.

method a function or subroutine that performs an operation on the containing object. Method names are signaled by as set of parentheses which may be empty or contain a set of arguments.

MQTT is abbreviation for **MQ T**elemetry **T**ransport protocol developed by IBM as a publish and subscribe service.

multi-threading is a service offered by some operating systems, like Linux, that allow a program to run as separate threads in separate processes either on the same processor or on a different processor or core. The roboRIO has two cores, so it can do processing in both cores simultaneously.

NAT is an abbreviation for **N**etwork **A**ddress **T**ranslation a protocol for converting between the private network addresses of a local private internet and the network addresses of the public Internet.

Network Tables is publish and subscribe service developed by National Instruments. Each data element is described by an alphanumeric tag and has a value. Publishers establish the tag and may change the value at any time. Subscribers request a set of tags and are informed of changes to the values of the tag to which they subscribe.

navX of **navX2** is the brand name of a FIRST robotics inertial measurement unit (IMU) coprocessor device.

NOT is a boolean logic operation for negating a boolean value.

octal is a number in base-8 using the digits: 0, 1, 2, 3, 4, 5, 6, and 7. It is not in common use anymore, but is used in Java for numbers with a leading zero (0).

odometry is the measurement of the location of a body based on its incremental movements.

operator 1. mathematical, logical, relational, equivalence or assignment operation symbol used in expressions to manipulate values or operands. **2.** is the human who controls accessories on the robot. Also called the co-pilot.

OR is a boolean logic operation that is true if any of its operands are true.

OPR or **Offensive Power Rating** is a statistic that infers a team's contribution to their alliance score.

OSI is an abbreviation for **O**pen **S**ystem **I**nterface usually in reference to the seven layer model.

overloading is using the same method name with different signatures to do essentially the same thing but with different arguments.

parameter is a value passed to a method or constructor. Parameter values are mapped onto the arguments used by the method.

PID proportional-integrated-differential control, a control algorithm used to control a system to smoothly and quickly reach a desired target position or velocity.

pilot is the human driver of the robot.

pitch is the (front-to-vertical) angle of rotation of the robot about the x-axis with respect to initial inertial vertical angle. (see 2.5.9.4 Vehicle Orientation Reference System.)

POE is an abbreviation for **P**ower **O**ver **E**thernet, a method for po1wering devices using the Ethernet data cabling to also carry power.

pointing or **pose** is the angle of the robot with respect to the field.

port is a addressable channel within a device. It is commonly used in networking for subdividing an IP address into functional channels. The roboRIO uses port number to access individual PWM channels (or LED control channel) and analog input channels.

polymorphism the ability to overload a method with different signatures to do essentially the same thing with different data types.

private accessible only inside of the object.

proof is a logical argument showing the deductive steps to arrive at a conclusion.

public accessible outside of the object

PWM is an abbreviation for **P**ulse **W**idth **M**odulation which uses a timed wave form to dynamically change the amount of power delivered to a device.

Q.E.D. is abbreviation at the end of a proof immodestly meaning Quite Exceptionally Done.

RGB stands for **R**ed, **G**reen, and **B**lue, three primary colors for transmitted colors (as opposed to reflected colors). This is also the order of the LED colors in some serial accessing schemes.

RGBW stands for **R**ed, **G**reen, **B**lue and **W**hite, three primary colors plus white for transmitted colors (as opposed to reflected colors). This is also the order of the LED colors in some serial accessing schemes.

roboRIO is the primary robot controller required by FIRST.

roll is the (side-to-vertical) angle of rotation of the robot about the y-axis with respect to the initial inertial vertical angle. (See 2.5.9.4 Vehicle Orientation Reference System.)

RPM is an abbreviation for **R**evolutions **p**er **M**inute used in motor or shaft rotary speeds.

RSL is the **R**obot **S**tatus **L**ight which communicates the Field Management Status of the robot.

rsync is a Linux command for synchronizing the files between two systems using encrypted transfers.

scope is where an object, variable, or parameter may be accessed. In Java scope is normally confined to the containing **block** set off by a pair of curly braces (“{“ and “}”). For example this can be an object, a method within an object, an attribute within an object, a variable within a “for loop” a method.

scp is a secure way to copy files from a Linux operating system such as that used by the roboRIO. scp uses ssh to provide a secure link between systems.

semaphore is a mechanism that is locked and unlocked by a thread to temporily prevent read access to a variable or block of memory.

Shuffleboard is the display of telemetry data on the drive station.

SID is an abbreviation for **S**ystem **ID**entifier used to identify a WiFi access point.

signature is the tuple of the the returned primitive data types or classes and the primitive data types or classes of the passed arguments. For example:

The signature of

```
int foo( int a, int b)
```

is (int, int, int)

The signature of

```
null foo( int c, int e)
```

is (null, int, int)

The signature of

```
float foo( float c, float e)
```

is (float, float, float)

SPI is an abbreviation for a Serial Peripheral Interface used to interconnect microcontroller devices on a short distance bus.

ssh is a secure shell used to access computers running Linux remotely and securely.

static retained in memory. It is not necessary to create an object for a static class.

subroutine a routine that does not return a value.

TCP is an abbreviation for **T**ransfer **C**ontrol **P**rotocol, a network layer internet protocol for guaranteed (acknowledged) delivery of packets between network nodes. This is a base protocol used in the internet suite of protocols. It is sometimes called **TCP/IP**.

thread is a separate process or task in an operating system. In multicore processors threads may be run in different cores.

thread safe is a technique to guarantee that a thread of a process will not read certain values that has been partially written by another thread of the same process. This uses semaphores to lock and unlock read access to such values when they are being written.

topology means study of shapes. In networks the basic topologies are star (or home run), bus, ring, hierarchal, mesh (or graph).

trig is an abbreviation for **trigonometry**, the relationships between the sides and angles of triagles.

typ is stands for typical. It is used to avoid repeating something over and over in a specification.

type the primitive type (integer (int), float, double, boolean, char) or class (String or a locally defined class) assigned to a variable, attribute or return value of a method.

UDP is an abbreviation for **U**nit **D**atagram **P**rotocol, a network layer internet protocol for unguaranteed (unacknowledged) delivery of packets between network nodes. This is a base protocol used in the internet suite of protocols.

Unicode or **Unicode Transformation Format (UTF)** is a numeric encoding of characters for the world's languages, symbols, and emoji. It takes 32 bits for a full encoding, but there are 8- and 16-bit variations to reduce storage requirements.

USB is an abbreviation for **Universal Serial Bus** which is a serial bus for interconnecting external devices such as memory stick, a disk drive, a mouse, a joystick controller, etc. to a personal computer.

UTF-8 is the 8-bit variant of Unicode with expansions for the 16-bit and 32-bit characters.

UTF-16 is the 16-bit variant of Unicode with expansions for the 32-bit characters.

UTF-32 is the full 32-bit variant of Unicode.

variable a named value with a specified type which is used within the method in which it is defined.

WPILib is the Wooster Polytechnic Institute (WPI) library for FIRST robotics.

Xbox is a Microsoft video game platform that has joystick controllers that are commonly used to control FRC robots.

XOR is an abbreviation for e**X**clusive **OR**, a boolean logical operation that is true if its boolean operands are different.

yaw is the angle of the robot (really inertial measurement device) about the z-axis and the initial inertial angle. (See 2.5.9.4 Vehicle Orientation Reference System).

\n is a newline character (traditionally the line feed character)

\r is a return character (traditionally the carriage return character)

\u signals an unsigned hexadecimal value used for a character code.

\\ is a single back slash character.

0 is short hand for a number in octal notation or base-8.

0x is short hand for a number in hexadecimal notation or base-16.

(This page intentionally blank.)

Alphabetical Index