This is a complete rewrite of the previous draft of this manual. This iteration attempts to outline more of the components of robot control system while building and reorganizing the existing sections.

This reorganizing and writing is not complete. Comments about this manual are welcome especially about repetitive, unnecessary or desired explanations are welcome.

FRC 4513 Robot Manual

Table of Contents

The purpose of this manual is to introduce students and mentors to the FRC 4513 robot software. This provides an overview of the robot architecture and how that relates to the software architecture and the file structure of the source code files. It also includes a brief overview of the Java programming language and its characteristics as an Object Oriented Language.

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

Robot Hardware Architecture

A more detailled 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.

Required Hardware Components

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

- 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.
- **Main breaker** to protect all of the components against accidental shorting. This is a 120 A breaker so it will allow a dangerous current flow which can arc and melt metal. The main supply wires should ALWAYS be treated with respect and care.
- **Power distribution** hub has breakers to protect individual circuits against over current. Every motor in the robot has its own breaker.
- **roboRIO** is the main controller for the robot. It has almost all of 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. Some auxiliary co-processors are allowed in things like vision and navigation subsystems. The r**oboRIO** should ultimately be in control of all co-processors and the devices that they may control.
- Every robot must have a clearly visible **Robot Status Light (RSL)**. It shows:

Solid On	Robot powered up and disabled
Blinking	Robot powerer up and enabled
Off	Robot off, roboRIO not powered or RSL, is not wired properly

- A **WiFi radio** provides communication between the robot and the field management system and through it, the driver station. This radio encrypt the communication to ensure that only the field management system can communicate to the robots on the field and 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 **Robot Power Module (RPM)** specifically to power the **WiFi Radio** using power over ethernet.
- A **voltage regulator** to convert the 12V robot power to 5V power used by some components in the robot

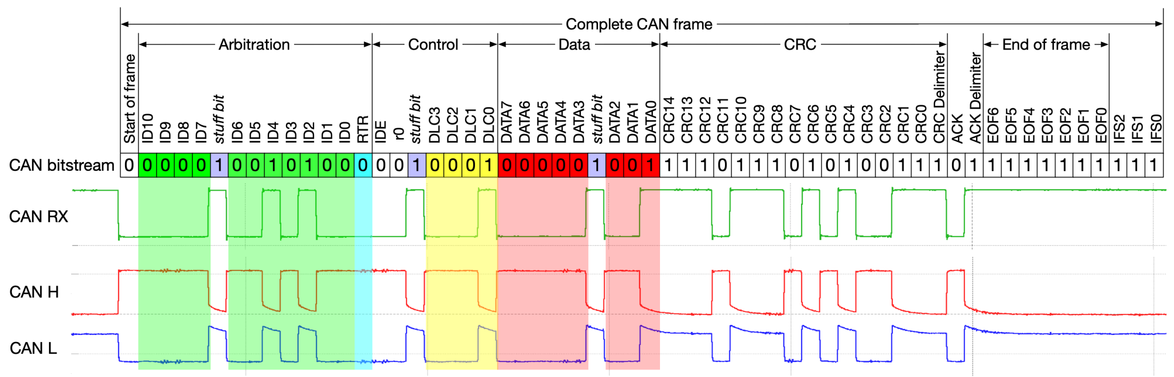
CAN Bus

The roboRIO communicates with many of the hardware components over the Controller Area Network, 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 architecture that can have many senders and receivers. It also has a protocol that is used to control access to the bus. The

CAN bus is wired as a “daisy chain,” where by the each component is a link in a chain and there is a CAN bus input and CAN bus output on every device. The chains connect the output CAN bus of one component to the input CAN bus on another. The last device on each end of the CAN bus daisy chain must be connected to a terminating resistor to ensure the signal integrity of the entire bus.

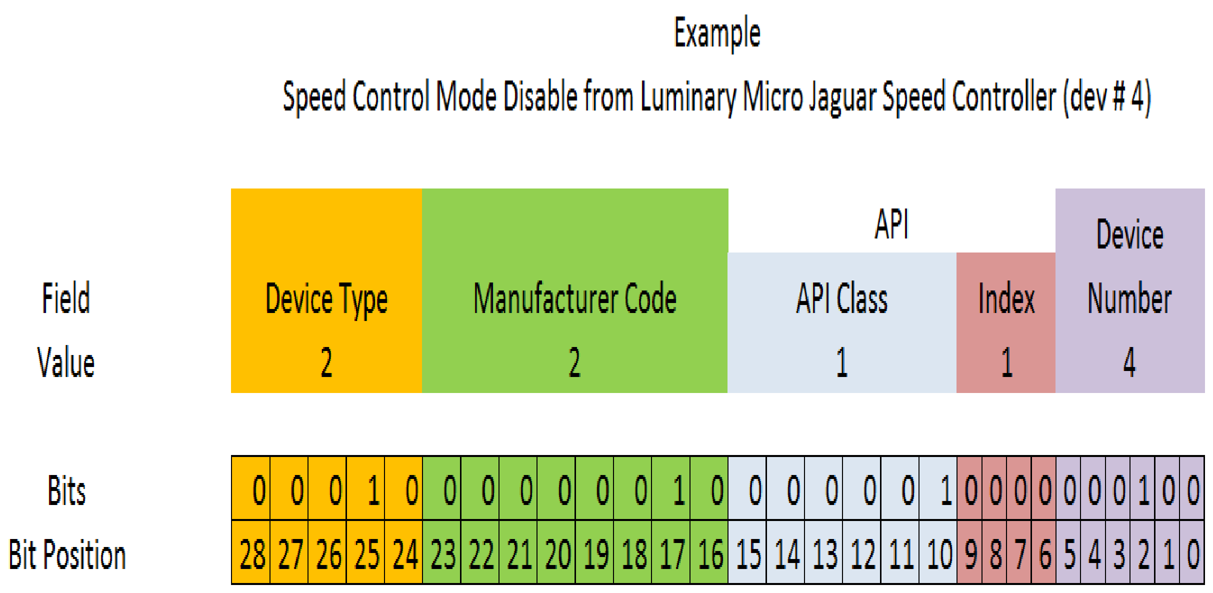
Some features of the CAN bus:

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



Basic CAN Bus frame *By Ken Tindell, Canis Automotive Labs Ltd, CC BY-SA 4.0*

- The protocol allows multiple masters (nodes which can initiate a data transfer). This allows bi-directional data transfers.
- It has a bus architecture, i.e., it uses a daisy chain with extremely short stubs.
- The CAN bus needs to have a single termination resistor at the two ends of the daisy chain. This is to prevent reflections on the bus which degrade its performance.
- Bits are sent using a differential voltage across a pair of wires.One wire is called CAN H and the other is called CAN L. A logical “1” is passive, not actively driven, with a low voltage on both the CAN H and CAN L leads. A logical “0” actively drives a voltage on both CAN H and CAN L leads. The “0” can overpower a “1.”
- This diffential voltage is fairly immune to electrical noise from motors, because motor noise induces a similar voltage into both leads equally and there is very little diffential component.
- Collisions are detected when a node sends a non-dominant “1” value and detects a dominiant “0” value. Since the identifier is the first data sent in the protocol, this give priority to lower identifier values.
- 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. This works out to 1000 messages every 20ms.
- 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]



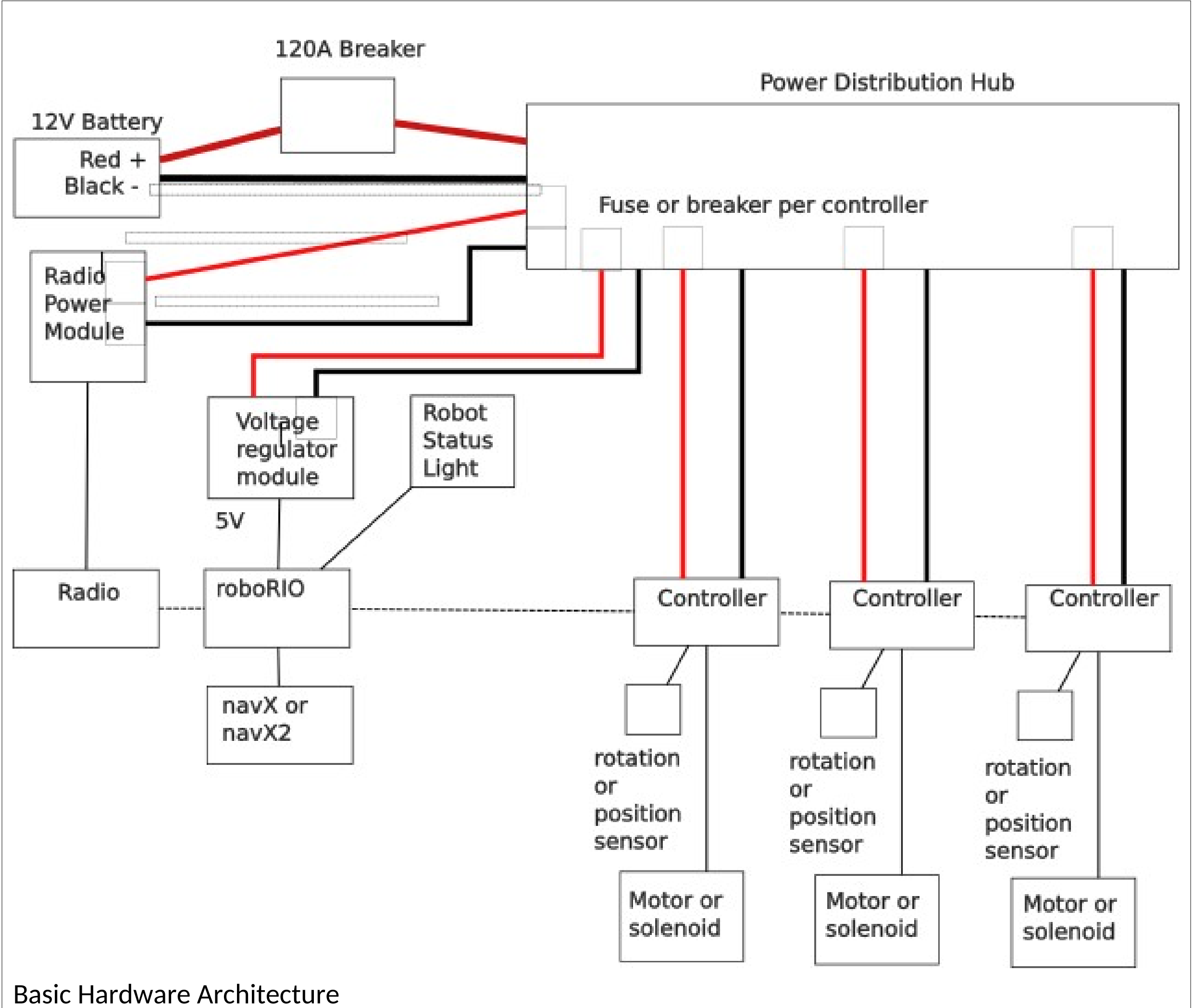
Example

Speed Control Mode Disable from Luminary Micro Jaguar Speed Controller (dev # 4)

														API										Device					
Field	Device Type					Manufacturer Code								API Class						Index				Number					
Value	2					2								1						1				4					
Bits	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
Bit Position	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Example of coding of a CAN Bus command carried in the CAN bus data payload
This appears to be using the extended CAN bus encoding....need to do more research in this.

-
    - 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, 0x3F may be reserved for device specific broadcast messages.
- Actuators that control motors must ensure that the frame originated from the roboRIO.
- A Universal Heartbeat message is used to enable motors. It this message is not seen for 100ms, controllers should assume that the system is disabled.f



Basic Hardware Architecture

Optional Restricted Hardware Components

Optional components can be used as needed for particular robot designs with some restrictions on the number and supplier:

- Only certain motors can be used and 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, ...)
- Pneumatics may be used (as they are in the show bot), 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.
  - there can be a number of solenoids to release pressure to particular pneumatic devices.
  - it must have a **pressure sensor** on the pressure holding tank connected to the **pneumatic hub** to prevent the system from building up dangerous pressures.

- **Cameras** can be used to send video back to the drive station as long as the image quality is restricted to what the roboRIO will allow.

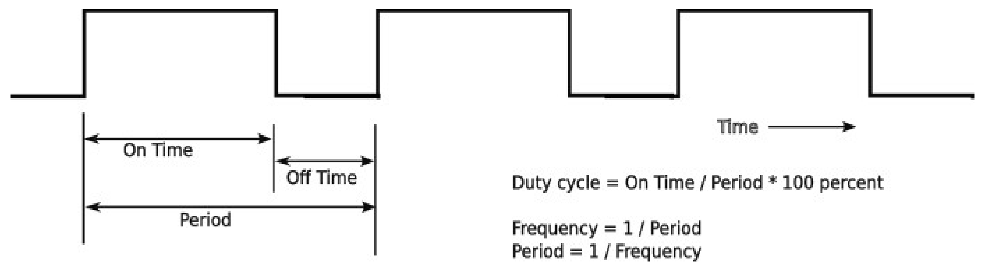
Optional Unrestricted Hardware Components

Some components can be used in robots without restriction:

- An ethernet router or switch to distribute ethernet to various components within the robot like cameras.
- Limit switches on elevators, arms, and other moving devices within the robot.
- An integrated navigation subsystem such as the **navX** or **navX2** which connects directly to the roboRIO.
- An independent navigation subsystem such as the **Pigeon** which connects via the CAN bus.
- There can be co-processor devices to assist the roboRIO. These have been used for vision processing, providing inertial guidance information and tighter motor speed controllers that include extra functionality like PID (Proportional Integrated Differential).

Pulse Width Modulation (PWM)

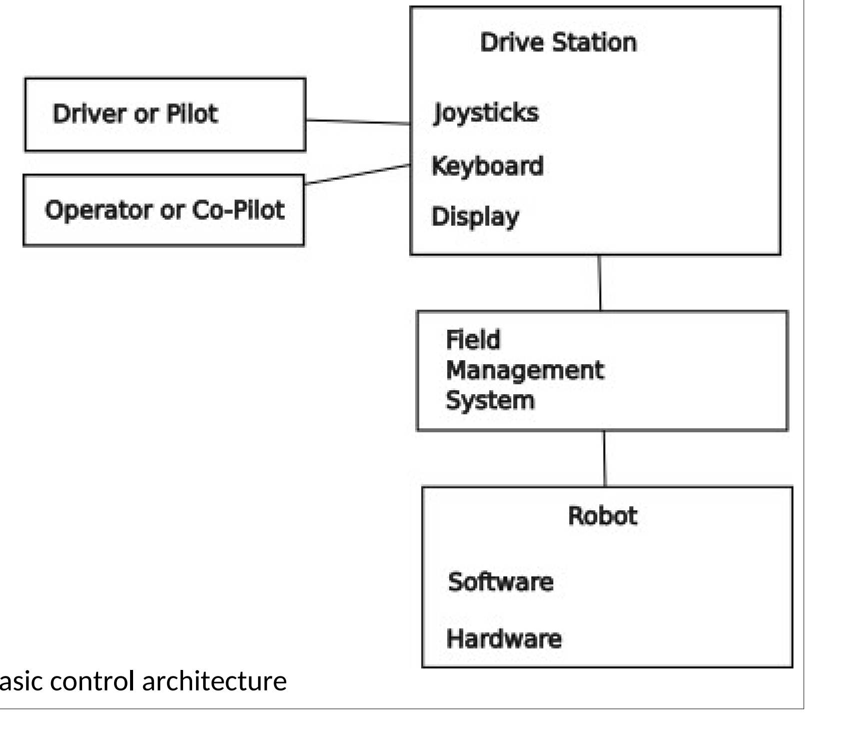
Some motors and lights want to be controlled with a **Pulse Width Modulation (PWM)** signal. A pulse is sent at a fixed rate and the duty cycle of the pulse is varied. The percentage of on time to pulse period determines how fast a motor can go or how bright a light will be.



Duty Cycle of a Voltage Waveform for use as Pulse Width Modulation

Robot Control Architecture

Control of the robot is central to robot competition as is the safe operation of the robots for all 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 they insert control between the driver station and the robot, so that they can intercept any command at any time. This position also allows it to control the periods of the competition: prematch, autonomous, teleop, and postmatch.



Basic control architecture

- In prematch 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 they 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).
- 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 though the **field management system**. This system does a couple of 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.
- stops 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 WiFi 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 WiFi access point must be reprogrammed for each competition.

Robot Software Architecture

The software for a robot can be overwhelming at first, however, by breaking its functionality down into smaller and smaller pieces, it becomes easy to understand and maintain. At the highest level each robot has software that controls various hardware elements on the robot.

In a nutshell, a robot consists of a set of subsystems. Each subsystem controls an aspect of the robot, say an arm, an elevator, a drive train, a climbing mechanism, a throwing mechanism. A subsystem can only do one thing at a time. An elevator can go up or it can go down. It cannot go up and down at the same time with meaning full results. A general way to think about it is that a subsystem controls a single motor. This can be extended to orchestrate a set of subordinate subsystems. The drive train for a tank drive can control two motors. It can go forward, backward, turn to the right, turn to the left, spin right or spin left. In this case the drive train subsystem is simplifying the commands necessary to move the robot by handling the interactions between the two motors. The drive train for a swerve drive is much more complex management of the swerve modules but it has similar commands to the tank drive, except that it can drive forward in any direction.

Commands for the various systems originate mainly from the drive station joystick controllers. A button will request the elevator to rise, stop, or lower. A joystick is generally used to request the drive train to move in a particular direction and at a particular speed. Some commands may originate from events detected by the robot. If the robot detects that the elevator has exceeded its limit, it can command the elevator to immediately stop.

So some coordination is required. The robot should be able to move while raising the elevator and extending its arm and flashing a light and checking on the elevator limit switch. Subsystems can only do one thing a time, but multiple subsystems can operate at the same time. This coordination is handled with two basic concepts. First a robot command is broken down into components that are executed in a fixed way for all command:

- **initialize** is executed one time to set up the processing of the command.
- **execute** is executed periodically during the processing of the command to make changes as necessary over time to ensure the command is executing properly. This can run as many times as necessary.
- **isFinished** is executed immediately after each execute to test of the command has reached its stated objective.
- **interrupted** allows an external process to stop the command cleanly.
- **end** is executed when **isFinished** is true to clean up the processing of the command.

These components need to execute quickly and return control to the operating system so that other commands can simultaneously execute.

The second part of the coordination comes from a scheduler. The scheduler has a queue of commands waiting for their **execute** and **isFinished** methods to be periodically executed. The scheduler essentially has a loop so that it executes everything in its queue every 20 ms. There are other things going on in the roboRio in parallel with the scheduler. It manages several processes of the Linux operating system. It runs the WiFi communication to get commands from the joystick and to send status and possibly video to the shuffle board display.

Background on 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 graphical programming language), Java (a modern object oriented language), and C++ (a version of the traditional C language that supports object oriented programming). FRC 4513 has chosen Java because it is a textual language and not as complex as C++.

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.

Wooster Polytechnic Institute Library for FIRST Robotics or WPI

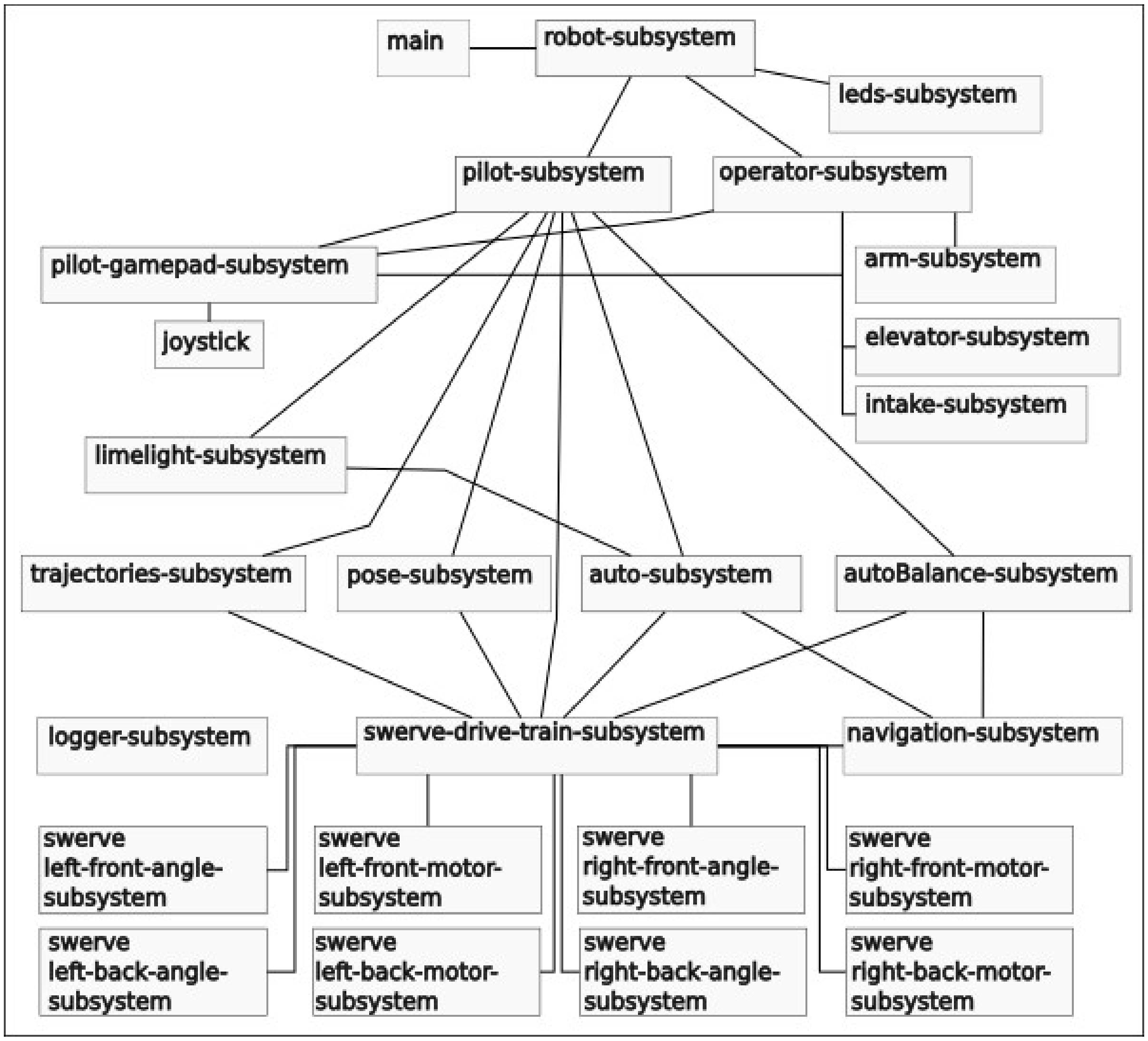
Programmers use libraries of previously coded and debugged software so they do not have to recreate all of that functionality for themselves. Java comes with libraries for basic input and output functions as well as mathematic functions for square roots and trigonometry. Wooster Polytechnic Institude has written a library (WPILIB) for the basic functions of a robot so we 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.

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 some mapping between 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 Docs: https://docs.wpilib.org

The WPI library dictates the structure of the robot code to some extent. FRC 4513 has adopted the changes to the file structure used by the Spectrum FRC team 2847 to better organize of the code. They used a directory for each subsystem so the code is kept more separate and modular. It also makes it easirer to add and delete subsystems from a robot as needed by each season's particular challenge.

Subsystem Software Architecture


main
robot-subsystem
leds-subsystem
pilot-subsystem
operator-subsystem
pilot-gamepad-subsystem
arm-subsystem
joystick
elevator-subsystem
intake-subsystem
limelight-subsystem
trajectories-subsystem
pose-subsystem
auto-subsystem
autoBalance-subsystem
logger-subsystem
swerve-drive-train-subsystem
navigation-subsystem
swerve left-front-angle-subsystem
swerve left-front-motor-subsystem
swerve right-front-angle-subsystem
swerve right-front-motor-subsystem
swerve left-back-angle-subsystem
swerve left-back-motor-subsystem
swerve right-back-angle-subsystem
swerve right-back-motor-subsystem

Subsystem Software Architecture

Robot Module

The entry point for Java programs is a function called **main**. Java programs are compiled into a block of byte codes. Then the byte codes are executed by an interpreter on the target machine, control is transferred the entry point defined by the **main** function. In this architecture, the **main** module just passes control to the robot module, which stands at the top of the robot software architecture. It creates instances of the major subsystems.

It provides methods for major operation modes such as the **autonomous** mode where the robot is entirely controlled by the computer and the **teleop** mode where human pilots drive the robot with computer assist. The **autonomous** mode may have various options to fit particular scenarios so that the team can cooperate with other teams in its alliance during the autonomous period including selecting different starting positions. It also includes various modes for testing the robot and for simulating robot functions.

Subsystem Software Architecture

A subsystem controls a a device or set of devices within a robot as a single process running one command at a time. Each subsystem is managed with a common set of methods:

- **setupDefaultCommand** is a method to establish a default command within the subsystem.
- **periodic** is a method that is called periodically to poll the status of various switches and sensors pertaining to the particular subsystem.
- **commands** is a set of methods that control the subsystem. Each method does only one thing like start the motor, stop the motor, etc.
- A **telemetry** module within a subsystem is used to convey information about the subsystem to a tab in the **shuffleboard** display at the drive station.

Subsystems

The following is an alphabetical list of the current subsystems used by the robot.

- **arm** controls and monitors the position of the arm.
- **auto** controls the robot during the autonomous period.
- **autoBalance**-controls the robot to balance the robot on the charge platform without human intervention. *(This does not appear to be hooked up in the current 4513 code.)*
- **elevator** controls and monitors the position of the elevator.
- **intake** controls and monitors the intake mechanism to load, process and unload game pieces.
- **leds** controls LEDs on the robot. *(This does not appear to do anything in the current 4513 code.)*
- **limelight** controls and monitors a vision system capable of reporting certain location and game piece information. *(This does not appear to be hooked up in the current 4513 code.)*
- **logger** log various events for debugging and performance evaluation. Most subsystem generate events for logging. *(This appears to not be hooked up in the current 4513 code.)* 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 command 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
    - *current if known.*
    - lime light info
    - other telemetry information
- **joystick controller** interfaces a joystick by normalizing its inputs and applying those inputs to a set of commands passed to the robot.
- **operator** provides control and feedback interfaces to a human co-pilot operating a joystick controller.
- **pilot** provides control and feedback interfaces to a human driver operating a joystick controller.
- **pose** provides an estimated x and y coordinate of the robot as well as the angle of the robot with respect to the field. *(This is only used by the swerve subsystem in the current 4513 code.)*
- **swerve drive train** controls the motion of the motion by specifying a desired rotational angle velocity, x velocity and y velocity, all with respect to the field.
- **swerve module** controls the drive velocity and the angle with respect to the robot of one swerve module.
- **trajectories** methods for driving the robot over paths determined on the fly to minimize unnecessary movement to maximize speed and agility.
- **climber** is a subsystem used in games with a climbing challenge to lift the robot off the floor. *Ficticious*
- ***pickup** is a set of methods used to autonomously pickup a game piece from a loading station or off the floor. *Ficticious!*
- **score** is a set of methods used to autonomously place or shoot a game piece. *Ficticious!

File Structure

The beauty of Spectrum Team file structure is that it mirrors the subsystem architecture. This means that the structure breaks down into individual subsystems..

```
~/dev/4513/Robot2023SwerveVer03
├── 3rd Part Library Links.txt
├── Autonomous.png
├── bin
│   └── main
│       └── frc
│           ├── lib
│           │   ├── drivers
│           │   │   ├── LimeLight$PeriodicRunnable.class
│           │   │   ├── LimeLight.class
│           │   │   ├── LimeLightControlModes$Advanced_Crosshair.class
│           │   │   ├── LimeLightControlModes$Advanced_Target.class
│           │   │   ├── LimeLightControlModes$CamMode.class
│           │   │   ├── LimeLightControlModes$LedMode.class
│           │   │   ├── LimeLightControlModes$Snapshot.class
│           │   │   ├── LimeLightControlModes$StreamType.class
│           │   │   ├── LimeLightControlModes.class
│           │   │   ├── LinearServo.class
│           │   │   ├── SpectrumDigitalInput.class
│           │   │   ├── SpectrumDigitalRelay.class
│           │   │   └── SpectrumSolenoid.class
│           │   ├── gamepads
│           │   │   ├── AxisButton$ThresholdType.class
│           │   │   ├── AxisButton.class
│           │   │   ├── Dpad.class
│           │   │   ├── Gamepad.class
│           │   │   ├── mapping
│           │   │   │   ├── Curve.class
│           │   │   │   ├── ExpCurve.class
│           │   │   │   ├── LinCurve.class
│           │   │   │   ├── SplineCurve$SegmentType.class
│           │   │   │   ├── SplineCurve.class
│           │   │   │   └── SplineType.class
│           │   │   ├── SpectrumButton.class
│           │   │   ├── ThumbStick.class
│           │   │   ├── Triggers.class
│           │   │   ├── XboxGamepad$XboxAxis.class
│           │   │   ├── XboxGamepad$XboxButton.class
│           │   │   ├── XboxGamepad$XboxDpad.class
│           │   │   └── XboxGamepad.class
│           │   ├── logger
│           │   │   └── Logger.class
│           │   ├── motorControllers
│           │   │   ├── TalonFXConstantsExample.class
│           │   │   ├── TalonFXSetup.class
│           │   │   └── TalonSRXSetup.class
│           │   ├── preferences
│           │   │   ├── SpectrumPreferences.class
│           │   │   └── TunablePrefrence.class
│           │   ├── sim
│           │   │   ├── PhysicsSim$SimProfile.class
│           │   │   ├── PhysicsSim.class
│           │   │   ├── TalonFXSimProfile.class
│           │   │   ├── TalonSRXSimProfile.class
│           │   │   └── VictorSPXSimProfile.class
│           │   ├── subsystems
│           │   │   ├── angleMech
│           │   │   │   ├── AngleMechConfig.class
│           │   │   │   └── AngleMechSubsystem.class
│           │   │   ├── CommandExamples.class
│           │   │   ├── FollowerFalconFX.class
│           │   │   ├── FollowerTalonSRX.class
│           │   │   ├── linearMech
│           │   │   │   ├── LinearMechConfig.class
│           │   │   │   └── LinearMechSubsystem.class
│           │   │   ├── MotorFXSubsystem.class
│           │   │   ├── MotorFXSubsystemConfig.class
│           │   │   ├── MotorSRXSubsystem.class
│           │   │   ├── MotorSRXSubsystemConfig.class
│           │   │   ├── pneumatic
│           │   │   │   └── PneumaticSubsystem.class
│           │   │   └── rollerMech
│           │   │       ├── RollerMechConfig.class
│           │   │       └── RollerMechSubsystem.class
│           │   ├── swerve
│           │   │   ├── CTREModuleState.class
│           │   │   ├── SwerveModuleConfig.class
│           │   │   └── SwerveUtil.class
│           │   ├── sysid
│           │   │   ├── PolynomialRegression.Pblmjava
│           │   │   ├── SysIdDrivetrainLogger.class
│           │   │   ├── SysIdGeneralMechanismLogger.class
│           │   │   ├── SysIdLogger.class
│           │   │   └── VelocityProfiler.class
│           │   ├── telemetry
│           │   │   ├── Alert$AlertType.class
│           │   │   ├── Alert$SendableAlerts.class
│           │   │   ├── Alert.class
│           │   │   ├── AlertCommand.class
│           │   │   ├── CustomLayout.class
│           │   │   ├── TelemetrySubsystem$1.class
│           │   │   ├── TelemetrySubsystem.class
│           │   │   └── WidgetsAndLayouts.class
│           │   ├── util
│           │   │   ├── Conversions.class
│           │   │   ├── GeomUtil.class
│           │   │   ├── Network.class
│           │   │   ├── Rmath.class
│           │   │   └── Util.class
│           │   └── vision
│           │       ├── CircleFitter.class
│           │       ├── Controller.class
│           │       └── LLDistance.class
│           ├── Rmath.class
│           └── robot
│               ├── arm
│               │   ├── ArmConfig.class
│               │   ├── ArmSRXMotorConfig.class
│               │   ├── ArmSubSys$ArmStates.class
│               │   ├── ArmSubSys.class
│               │   ├── ArmTelemetry.class
│               │   └── commands
│               │       ├── ArmCalibrateCmd.txt
│               │       ├── ArmCmds.class
│               │       ├── ArmDriveForSecondsCmd.class
│               │       ├── ArmHoldPositionCmd.txt
│               │       ├── ArmReleaseCmd.txt
│               │       └── ArmToStowPosCmd.class
│               ├── auto
│               │   ├── Auto.class
│                   │   ├── AutoConfig$AutoPosition.class
│                   │   ├── AutoConfig.class
│                   │   └── commands
│                   │       ├── ArmElevDriveCmd.txt
│                   │       ├── AutoBuilder.class
│                   │       ├── AutoCmds.class
│                   │       ├── CenterDriveOnCmd$DRIVE_STAGE.class
│                   │       ├── CenterDriveOnCmd.class
│                   │       ├── DelayCmd.class
│                   │       ├── PrintAutoTimeCmd.class
│                   │       └── TaxiSimplePCmdGrp.class
│                   ├── autoBalance
│                   │   ├── AutoBalanceConfig.class
│                   │   └── commands
│                   │       └── AutoBalanceCommand.class
│                   ├── elevator
│                   │   ├── commands
│                   │   │   ├── Elev2StageSafeMove.txt
│                   │   │   ├── ElevatorCmds.class
│                   │   │   ├── ElevatorHoldPosCmd.class
│                   │   │   ├── ElevReleaseArmCmd.txt
│                   │   │   └── ZeroElevator.class
│                   │   ├── ElevatorConfig.class
│                   │   ├── ElevatorSubSys.class
│                   │   ├── ElevatorTelemetry.class
│                   │   └── ElevFXMotorConfig.class
│                   ├── intake
│                   │   ├── commands
│                   │   │   ├── IntakeCmds.class
│                   │   │   ├── IntakeConeMainCmd.txt
│                   │   │   └── IntakeConeSecondaryCmd.txt
│                   │   ├── IntakeConfig.class
│                   │   ├── IntakeSubSys.class
│                   │   └── IntakeTelemetry.class
│                   ├── leds
│                   │   ├── commands
│                   │   │   └── LedCmds.class
│                   │   ├── LedConfig.class
│                   │   ├── LedSubSys.class
│                   │   └── LedTelemetry.class
│                   ├── limelight
│                   │   ├── commands
│                   │   │   ├── CameraFrontCmd.class
│                   │   │   ├── CameraRearCmd.class
│                   │   │   ├── LEDOffCmd.class
│                   │   │   ├── LEDOnCmd.class
│                   │   │   ├── LLCamModeDriverOnCmd.class
│                   │   │   └── LLCamModeVisionOnCmd.class
│                   │   ├── limelightSubSys$LedMode.class
│                   │   ├── limelightSubSys$LLSmardashState.class
│                   │   └── limelightSubSys.class
│                   ├── logger
│                   │   ├── commands
│                   │   │   └── LoggerCmds.class
│               │   └── Logger.class
│               ├── Main.class
│               ├── operator
│               │   ├── commands
│               │   │   ├── ArmElevComboMoveCmds.class
│               │   │   ├── ArmElevDriveSafeCmd$CmdState.class
│               │   │   ├── ArmElevDriveSafeCmd.class
│               │   │   └── OperatorGamepadCmds.class
│               │   ├── OperatorGamepad.class
│               │   ├── OperatorGamepadConfig.class
│               │   └── OperatorGamepadTelemetry.class
│               ├── pilot
│               │   ├── commands
│               │   │   └── PilotGamepadCmds.class
│               │   ├── PilotGamepad.class
│               │   ├── PilotGamepadConfig$MaxSpeeds.class
│               │   ├── PilotGamepadConfig.class
│               │   ├── PilotGamepadTelemetry$gamepadLayout.class
│               │   └── PilotGamepadTelemetry.class
│               ├── pose
│               │   ├── commands
│               │   │   └── PoseCmds.class
│               │   ├── Pose.class
│               │   ├── PoseConfig.class
│               │   ├── PoseTelemetry.class
│               │   └── README.md
│               ├── Robot.class
│               ├── RobotConfig$AnalogPorts.class
│               ├── RobotConfig$Encoders.class
│               ├── RobotConfig$LimitSwitches.class
│               ├── RobotConfig$Motors.class
│               ├── RobotConfig$PWMPorts.class
│               ├── RobotConfig.class
│               ├── RobotTelemetry.class
│               ├── swerve
│               │   ├── commands
│               │   │   ├── DriveToMetersCmd.class
│               │   │   ├── LockSwerve.class
│               │   │   ├── SetModulesToAngleCmd.class
│               │   │   ├── SpinMoveCmd.class
│               │   │   ├── SwerveCmds.class
│               │   │   ├── SwerveDrive2Cmd.class
│               │   │   ├── SwerveDriveCmd.class
│               │   │   ├── SwerveDriveConfirmCmd.class
│               │   │   └── TurnToAngleCmd.class
│               │   ├── Gyro.class
│               │   ├── Odometry.class
│               │   ├── Swerve.class
│               │   ├── SwerveConfig$BLMod2.class
│               │   ├── SwerveConfig$BRMod3.class
│               │   ├── SwerveConfig$FLMod0.class
│               │   ├── SwerveConfig$FRMod1.class
│               │   ├── SwerveConfig.class
│               │   ├── SwerveModule.class
│                   │   └── SwerveTelemetry.class
│                   └── trajectories
│                       ├── commands
│                       │   ├── FollowOnTheFlyPath.class
│                       │   ├── FollowTrajectoryCmd.class
│                       │   ├── GeneratePathForScoring.class
│                       │   ├── PathBuilder.class
│                       │   ├── PathBuilderEstimatedPose.class
│                       │   └── TrajectoriesCmds.class
│                       ├── README.md
│                       ├── Trajectories.class
│                       └── TrajectoriesConfig.class
├── build.gradle
├── config.json
├── gradle
│   └── 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
├── settings.gradle
├── shuffleboard.json
├── src
│   ├── main
│   │   ├── deploy
│   │   │   ├── example.txt
│   │   │   └── pathplanner
│   │   │       ├── 215Meter.path
│   │   │       ├── Blue1Meter.path
│   │   │       ├── BlueConeShortCrossScale1.path
│   │   │       ├── BlueLeftCubeShort.path
│   │   │       ├── BlueRightCubeLong.path
│   │   │       ├── BlueRightLongScale1.path
│   │   │       ├── BlueRightLongScale2.path
│   │   │       ├── BlueShortCrossScale1.path
│   │   │       ├── BlueShortCrossScale2.path
│   │   │       ├── CtrScale1.path
│   │   │       ├── CtrScale2.path
│   │   │       ├── CtrScale.path
│   │   │       ├── RedConeShortCrossScale1.path
│   │   │       ├── RedLeftCubeLong.path
│   │   │       ├── RedRightCubeShort.path
│   │   │       ├── RedShortCrossScale1.path
│   │   │       ├── RedShortCrossScale2.path
│   │   │       ├── TestTurn2.path
│   │   │       └── TestTurn.path
│   │   └── java
│   │       └── frc
│   │           ├── lib
│   │           │   ├── drivers
│   │           │   │   ├── LimeLightControlModes.java
│   │           │   │   ├── LimeLight.java
│   │           │   │   ├── LinearServo.java
│   │           │   │   ├── SpectrumDigitalInput.java
│   │           │   │   ├── SpectrumDigitalRelay.java
│   │           │   │   └── SpectrumSolenoid.java
│   │           │   ├── gamepads
│   │           │   │   ├── AxisButton.java
│   │           │   │   ├── Dpad.java
│   │           │   │   ├── Gamepad.java
│   │           │   │   ├── mapping
│   │           │   │   │   ├── Curve.java
│   │           │   │   │   ├── ExpCurve.java
│   │           │   │   │   ├── LinCurve.java
│   │           │   │   │   ├── SplineCurve.java
│   │           │   │   │   └── SplineType.java
│   │           │   │   ├── SpectrumButton.java
│   │           │   │   ├── ThumbStick.java
│   │           │   │   ├── Triggers.java
│   │           │   │   └── XboxGamepad.java
│   │           │   ├── logger
│   │           │   │   └── Logger.java
│   │           │   ├── motorControllers
│   │           │   │   ├── TalonFXConstantsExample.java
│   │           │   │   ├── TalonFXSetup.java
│   │           │   │   └── TalonSRXSetup.java
│   │           │   ├── preferences
│   │           │   │   ├── SpectrumPreferences.java
│   │           │   │   └── TunablePrefrence.java
│   │           │   ├── sim
│   │           │   │   ├── PhysicsSim.java
│   │           │   │   ├── TalonFXSimProfile.java
│   │           │   │   ├── TalonSRXSimProfile.java
│   │           │   │   └── VictorSPXSimProfile.java
│   │           │   ├── subsystems
│   │           │   │   ├── angleMech
│   │           │   │   │   ├── AngleMechConfig.java
│   │           │   │   │   └── AngleMechSubsystem.java
│   │           │   │   ├── CommandExamples.java
│   │           │   │   ├── FollowerFalconFX.java
│   │           │   │   ├── FollowerTalonSRX.java
│   │           │   │   ├── linearMech
│   │           │   │   │   ├── LinearMechConfig.java
│   │           │   │   │   └── LinearMechSubsystem.java
│   │           │   │   ├── MotorFXSubsystemConfig.java
│   │           │   │   ├── MotorFXSubsystem.java
│   │           │   │   ├── MotorSRXSubsystemConfig.java
│   │           │   │   ├── MotorSRXSubsystem.java
│   │           │   │   ├── pneumatic
│   │           │   │   │   └── PneumaticSubsystem.java
│   │           │   │   └── rollerMech
│   │           │   │       ├── RollerMechConfig.java
│   │           │   │       └── RollerMechSubsystem.java
│   │           │   ├── swerve
│   │           │   │   ├── CTREModuleState.java
│   │           │   │   ├── SwerveModuleConfig.java
│   │           │   │   └── SwerveUtil.java
│   │           │   ├── sysid
│   │           │   │   ├── PolynomialRegression.Pblmjava
│   │           │   │   ├── SysIdDrivetrainLogger.java
│   │           │   │   ├── SysIdGeneralMechanismLogger.java
│   │           │   │   ├── SysIdLogger.java
│   │           │   │   └── VelocityProfiler.java
│   │           │   ├── telemetry
│   │           │   │   ├── AlertCommand.java
│   │           │   │   ├── Alert.java
│   │           │   │   ├── CustomLayout.java
│   │           │   │   ├── TelemetrySubsystem.java
│   │           │   │   └── WidgetsAndLayouts.java
│   │           │   ├── util
│   │           │   │   ├── Conversions.java
│   │           │   │   ├── GeomUtil.java
│   │           │   │   ├── Network.java
│   │           │   │   ├── Rmath.java
│   │           │   │   └── Util.java
│   │           │   └── vision
│   │           │       ├── CircleFitter.java
│   │           │       ├── Controller.java
│   │           │       └── LLDistance.java
│   │           ├── Rmath.java
│   │           └── robot
│   │               ├── arm
│   │               │   ├── ArmConfig.java
│   │               │   ├── ArmSRXMotorConfig.java
│   │               │   ├── ArmSubSys.java
│   │               │   ├── ArmTelemetry.java
│   │               │   └── commands
│   │               │       ├── ArmCalibrateCmd.txt
│   │               │       ├── ArmCmds.java
│   │               │       ├── ArmDriveForSecondsCmd.java
│   │               │       ├── ArmHoldPositionCmd.txt
│   │               │       ├── ArmReleaseCmd.txt
│   │               │       └── ArmToStowPosCmd.java
│   │               ├── auto
│   │               │   ├── AutoConfig.java
│   │               │   ├── Auto.java
│   │               │   └── commands
│   │               │       ├── ArmElevDriveCmd.txt
│   │               │       ├── AutoBuilder.java
│   │               │       ├── AutoCmds.java
│   │               │       ├── CenterDriveOnCmd.java
│   │               │       ├── DelayCmd.java
│   │               │       ├── PrintAutoTimeCmd.java
│   │               │       └── TaxiSimplePCmdGrp.java
│   │               ├── autoBalance
│   │               │   ├── AutoBalanceConfig.java
│   │               │   └── commands
│   │               │       └── AutoBalanceCommand.java
│   │               ├── elevator
│   │               │   ├── commands
│   │               │   │   ├── Elev2StageSafeMove.txt
│   │               │   │   ├── ElevatorCmds.java
│   │               │   │   ├── ElevatorHoldPosCmd.java
│   │               │   │   ├── ElevReleaseArmCmd.txt
│   │               │   │   └── ZeroElevator.java
│   │               │   ├── ElevatorConfig.java
│   │               │   ├── ElevatorSubSys.java
│   │               │   ├── ElevatorTelemetry.java
│   │               │   └── ElevFXMotorConfig.java
│   │               ├── intake
│   │               │   ├── commands
│   │               │   │   ├── IntakeCmds.java
│   │               │   │   ├── IntakeConeMainCmd.txt
│   │               │   │   └── IntakeConeSecondaryCmd.txt
│   │               │   ├── IntakeConfig.java
│   │               │   ├── IntakeSubSys.java
│   │               │   └── IntakeTelemetry.java
│   │               ├── leds
│   │               │   ├── commands
│   │               │   │   └── LedCmds.java
│   │               │   ├── LedConfig.java
│   │               │   ├── LedSubSys.java
│   │               │   └── LedTelemetry.java
│   │               ├── limelight
│   │               │   ├── commands
│   │               │   │   ├── CameraFrontCmd.java
│   │               │   │   ├── CameraRearCmd.java
│   │               │   │   ├── LEDOffCmd.java
│   │               │   │   ├── LEDOnCmd.java
│   │               │   │   ├── LLCamModeDriverOnCmd.java
│   │               │   │   └── LLCamModeVisionOnCmd.java
│   │               │   └── limelightSubSys.java
│   │               ├── logger
│   │               │   ├── commands
│   │               │   │   └── LoggerCmds.java
│   │               │   └── Logger.java
│   │               ├── Main.java
│   │               ├── operator
│   │               │   ├── commands
│   │               │   │   ├── ArmElevComboMoveCmds.java
│   │               │   │   ├── ArmElevDriveSafeCmd.java
│   │               │   │   └── OperatorGamepadCmds.java
│   │               │   ├── OperatorGamepadConfig.java
│   │               │   ├── OperatorGamepad.java
│   │               │   └── OperatorGamepadTelemetry.java
│   │               ├── pilot
│   │               │   ├── commands
│   │               │   │   └── PilotGamepadCmds.java
│   │               │   ├── PilotGamepadConfig.java
│   │               │   ├── PilotGamepad.java
│   │               │   └── PilotGamepadTelemetry.java
│   │               ├── pose
│   │               │   ├── commands
│   │               │   │   └── PoseCmds.java
│   │               │   ├── PoseConfig.java
│   │               │   ├── Pose.java
│   │               │   ├── PoseTelemetry.java
│   │               │   └── README.md
│   │               ├── RobotConfig.java
│   │               ├── Robot.java
│   │               ├── RobotTelemetry.java
│   │               ├── swerve
│   │               │   ├── commands
│   │               │   │   ├── DriveToMetersCmd.java
│   │               │   │   ├── LockSwerve.java
│   │               │   │   ├── SetModulesToAngleCmd.java
│   │               │   │   ├── SpinMoveCmd.java
│   │               │   │   ├── SwerveCmds.java
│   │               │   │   ├── SwerveDrive2Cmd.java
│   │               │   │   ├── SwerveDriveCmd.java
│   │               │   │   ├── SwerveDriveConfirmCmd.java
│   │               │   │   └── TurnToAngleCmd.java
│   │               │   ├── Gyro.java
│   │               │   ├── Odometry.java
│   │               │   ├── SwerveConfig.java
│   │               │   ├── Swerve.java
│   │               │   ├── SwerveModule.java
│   │               │   └── SwerveTelemetry.java
│   │               └── trajectories
│   │                   ├── commands
│   │                   │   ├── FollowOnTheFlyPath.java
│   │                   │   ├── FollowTrajectoryCmd.java
│   │                   │   ├── GeneratePathForScoring.java
│   │                   │   ├── PathBuilderEstimatedPose.java
│   │                   │   ├── PathBuilder.java
│   │                   │   └── TrajectoriesCmds.java
│   │                   ├── README.md
│   │                   ├── TrajectoriesConfig.java
│   │                   └── Trajectories.java
│   └── shuffleboard.json
├── sysid_data20230225-174837.json
├── sysid_data20230225-175221.json
├── sysid_data20230225-182314.json
├── sysid.json
├── SysID Notes.txt
├── vendordeps
│   ├── NavX.json
│   ├── PathplannerLib.json
│   ├── Phoenix.json
│   ├── PhoenixProAnd5.json
│   ├── PhoenixPro.json
│   ├── REVLib.json
│   └── WPILibNewCommands.json
├── WPILib-License.md
└── Xbox Ctrlrs Mapping.vsdx
```

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 for a season 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. Thinks like the CAN bus identifer of every controller, input ports for limit switches, analog sensors,   and pulse width modulator for LED controller.
- **RobotTelemetry.java** creates instances of subsystem telementry objects and maps information known to the robot to specific tabs in the shuffleboard display at the driver station.
- Various subsystem directories to control individual subsystems.

Each subsystem directory usually contains (note that subsystemName in the following is replaced by the actual subsystem name):

- **subsystemNameConfig.java** which holds the configuration data for the subsystem.
- **subsystemNameSubSys.java** defines the class for the subsystem.
- **subsystemNameTelemetry.java** defines the class for the subsystem telemetry attributes.
- **commands** is a directory containing code that defines the individual commands for the subsystem. This may be a single file with a bunch of short commands or individual files for more complex commands.

Command Architecture

Each subsystem has a set of commands. A subsystem can execute only one command at a time. A command may run until it meets a particular criteria (like time, position, angle, count). Commands may also be interrupted or stopped by other conditions. The general architecture of each command is composed of the following methods:

Commands require special attention. In robot control code it is desirable to be able to control various subsystems at the same time. Running a motor typically takes some time to reach its desired end point. Commands are defined as objects so that they can be scheduled either as a one time event or a command that is executed ove**initialize** is a method to initialize and setup the parameters, attributes or states necessary for the command to run.

- **execute** is a method invoked to start or continue the command every 20 ms. This has a very short duration and must return control to allow other commands to operate. This typically is to start a motor and may be used to dynamically control to the motor to vary its speed.
- **isFinished** a method that is called after every **execute** to determine if the command is complete by returning a boolean: **true** for finished and **false** for not-finished.
- **interrupted** a method to handle a request to interrupt a command in progress.
- **end** a method to stop or abort 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.r time. To do this efficiently with a common scheduler, all commands have the following methods.

A couple of examples may help clarify the use of commands. Let’s say the robot has an elevator. A command may be to raise the elevator to the top.

- **initialize** sets up sets the speed and desired end point for the elevator.
- **execute** is starts the elevator motor running. As the elevator nears the end point, it will slow the motor to ensure a smooth stop. Since the elevator is fighting gravity, it may also hold the elevator at the desired end position indefinitely.
- **isFinished** is checked to determine if the elevator has reached its desired height. It also checks the elevator limit switch just in case the height sensor is not working properly.

  interrupted allow the raising to be interrupted. The driver may have changed his mind about which level to raise the elevator to or has decided to do something else entirely. The elevator command could be ended.
- **end** is invoked to stop the elevator motor.

A second example is the drive chain commands.

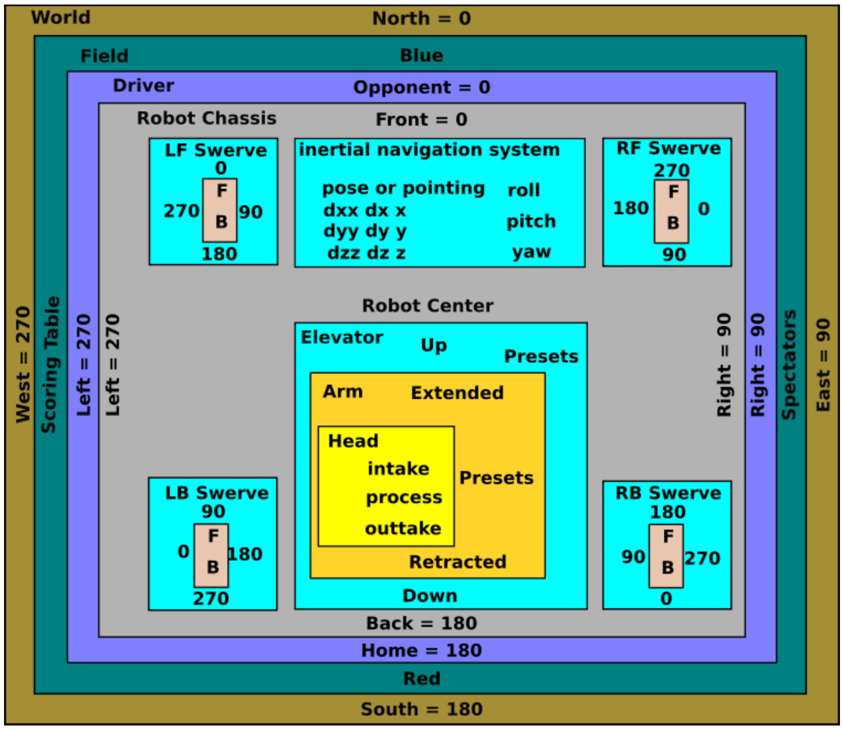
- **initialize** sets up the joystick to be used to control the speed and direction of the robot.
- **execute** goes off and reads the current value of the selected joystick and applies it to the x and y velocities of the robot. This may apply algorithms to limit the acceleration and path of the robot so that it drives smoothly without tipping over.
- **isFinished** is will always return true as this command is never really finished.

  interrupted allows the driving to be interrupted. This could be an automated routine that takes over the manual driving when the robot approaches a loading or scoring station.
- **end** is invoked to stop the drive train and possibly to lock the wheels.

Frames of Reference

A frame of reference defines where angles are measured from and distances are measured from. So it changes depending on which frame of reference you are using and there are measurements between each frame of reference and the containing one so that measurements can be used in any frame of reference.

Nearly every device on the robot has at least one angle or position that it is concerned about. In simple robots, everything can reference the robot chassis. Its front is at 0 degrees and angles increase clockwise looking down from above. Driving the robot is much easier when looking at the field from the driver perspective. Down field is 0 degrees, right is 90 degrees and left is 270 degrees.



Robot Frames of Reference

In FIRST competitions the driver may be on a red alliance or a blue alliance. The operation of the robot really does not change, but the placement of field pieces and game pieces does change when referenced from the perspective of the drive station. This has an impact to automated sequences during both the autonomous period and the teleop period. Beyond the field setup is the world. This can be found with a magnetic compass found in some navigation sensors, again with north being 0, degrees, east being 90 degrees, south 180 degrees and west 270 degrees. This could be useful in some situations to determine the bearing of the robot, but the relation ship of the magnetic compass and the field must be known or determined. Rarely will a field be laid out on a north-south line, but will be on some other angle. Luckily this can be determined when calibrating the robot.

The components within a robot also have frames of reference in their relationship to the robot chassis. In a swerve drive train there are four swerve drive assemblies. These each have a different positional relationship with respect to the chassis. Not only is their x and y coordinate different for each corner, but the orientation of each drive is different. Further it must be noted which direction if forward for each motor, as in a swerve drive it is hard to tell by looking. This can be expressed as a x, y measurement from the center of the robot to the vertical axis of the swerve module. It is also important to know at what angle forward is within the robot. Luckily this calibration normally only has to happen once during the build, although it may have to be recalibrated in the field as a result of a collision.

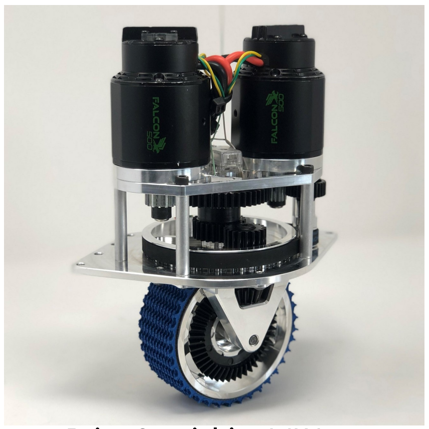
The inertial *navigation* system determines the pose of the robot. The pose is a tuple of three attributes: *x*, *y* and *theta*. *x* and *y* should be relative to the field. When initialized the x and y are set to 0,0 based on the location of the robot when it is started up. This should be offset to give values relative to the field. Theta Provides two more frames of reference. It provides a magnetometer that shows the roll, pitch and yaw of the robot with respect to magnetic north. This could help align the robot with respect to the world frame of reference as long as the correlation between the magnetic direction and the field direction are known. The inertial measurements provide a roll, pitch and yaw angles of the robot with respect to its start up position (or subsequent resets). The system also provides a displacement estimate that is good to about 4 inches over 15 seconds or about 3 feet during a match (unless accurately reset during the match). For the inertial measurements to be useful for field displacement estimates and angular measurements, they must be aligned to the field frame of reference. Since robots may be started or reset from various points on the field, the point where the reset is occurring must be supplied manually or determined by a vision system against field land marks or inferred by displacement estimate and commands being performed (e.g., scoring or loading a piece from a station).

The difference between the magnetic frame of reference and the inertial frame of reference may be a way to measure the drift of the inertial frame of reference.

Accessory attachments to the robot chassis such as a elevator and or arm mechanisms have a (hopefully) fixed relationship to the chassis. This allows the chassis to be oriented in a way that the elevator and arm can effectively interact with the game elements.

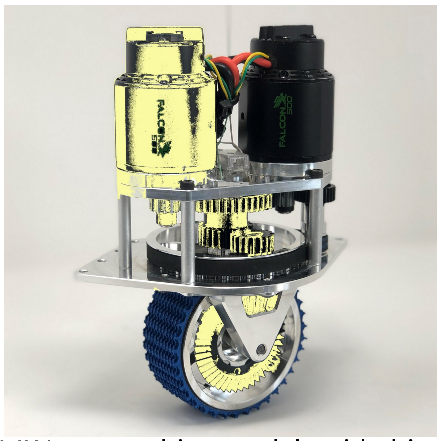
Swerve Drive Modules

The swerve drivetrain uses four swerve drive modules. Each module has a drive motor connected to a wheels 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.

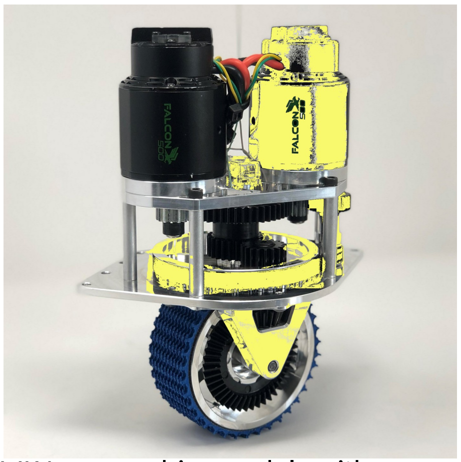


Swerve Drive Specialties MK4 swerve drive module

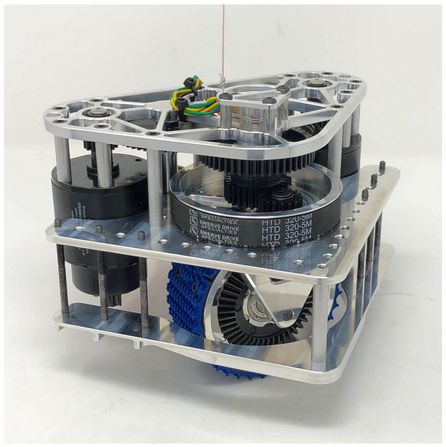
The drive transmission components are highlighted in yellow in the figure below. The drive motor is on the left hand side. It drives a gear train which eventually sends power down a shaft along side of the wheel to engage a ring gear with a bevel gear. The intermediate gears are on the outside of the inner axle for the wheel axel assembly. This turns the wheel in the horizontal axis and is independent of where the wheel is pointing.


MK4 swerve drive module with drive gear train highlighted in yellow.

The steering motor is on the left side. It transmits power to a belt that turns the wheel axle assembly about a vertical axis. The position of the wheel is monitored by a sensor at the top of the wheel axle assembly as the vertical axel has bearings in both the top and bottom mounting plates and is inside of the intermediate drive gears.

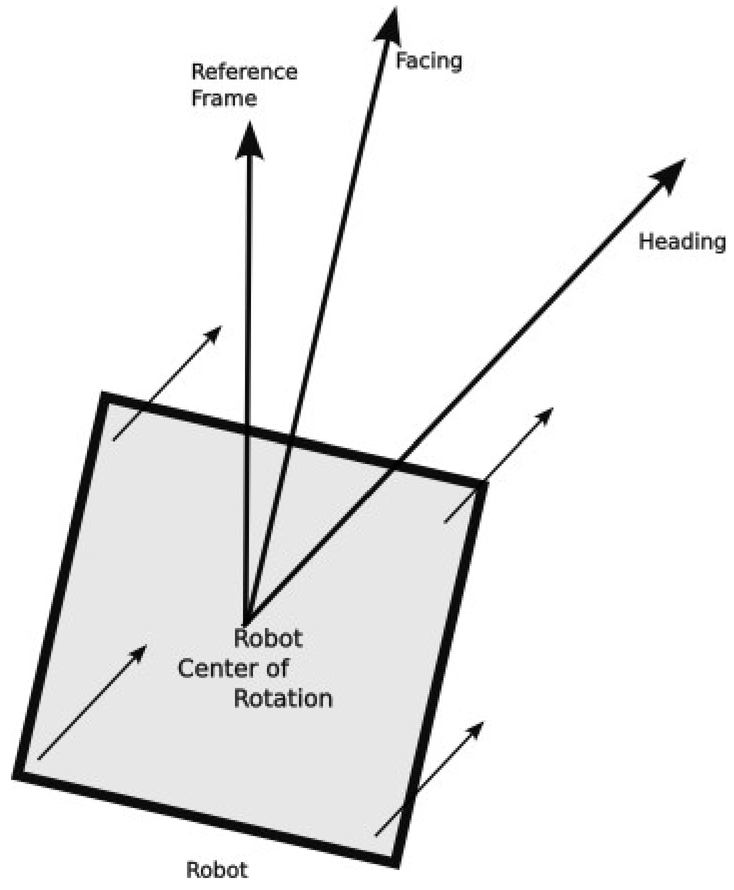

MK4 swerve drive module with steering gear train highlighted in yellow.

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


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

Drive Train

The Drive Train is central to the functionality of the robot as this is how the robot moves. The drive train controls the motors that propel the robot. This level of indirection allows the robot to be driven from a field frame of reference rather than a robot frame of reference. It also hides some of the complexity of interfacing joysticks to the actual drive motors.

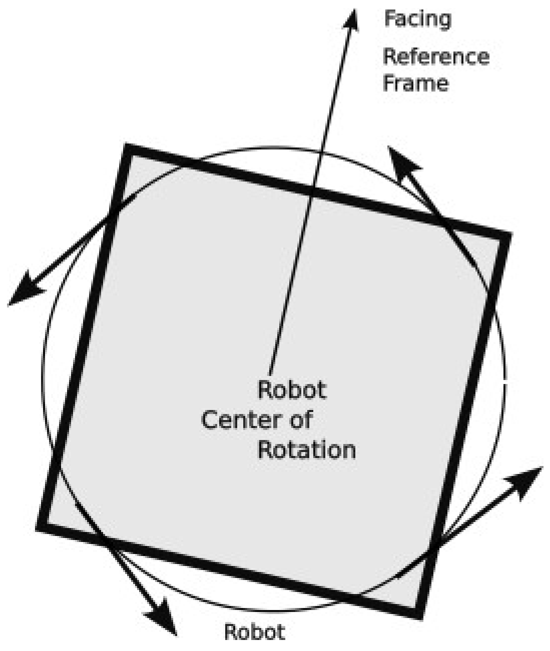


Swerve Drive Angles

Traditionally robots in the FRC have had tank drives. A tank drive has one or more wheels on each side of the robot. Each side was driven independently of the other. Steering was accomplished by driving each side a different velocity. The faster side would curve torward the slower side. A robot could turn in place by driving one side forward and the other side backward. This type of drive is also called differential drive because its steering is dependant on being driven at different speeds. A differential drive robot cannot move sideways. It must turn to the direction that it wants to move toward (or away from).

Most teams now use swerve drive. There are two or more (but usually four) swerve modules in a robot. Each module has two motors. One to steer the wheel and the other to drive the wheel. Each swerve module is steered independently of the others, but in concert so that the robot can move in any direction or even spin while moving.

A robot has two angles relative to the field. It has a heading angle or direction of travel and it has a facing angle which is the angle that the robot is facing. In tank drives these two angles are the same. In swerve drives these angles are independent. The robot has a pose angle which is the facing angle as reported by the pose subystem. This angle will drift from the actual facing angle. The pose angle and the yaw angle reported by the navigation subsystem should be the same although there may be differences if the field floor is not level.



Rotating a Swerve Drive Robot

Swerve Drive Train

To understand how swerve drive trains work requires a little math background. Vectprs are a way of expressing a direction and a magnitude. This could be something like go west at 30 MPH or the house is ¾ of a mile southwest of the town center. For the swerve drive train, these vector are a direction and a velocity. This vector starts as a simple joystick command where the driver pushes the joystick which produces an x and a y value. These two values are combined as a vector to express in what direction the driver want the robot to move and how fast. That's simple enough and the swerve drive train library has a move command with the parameters for the x and y components of movement with a boolean to select whether the coordinates are in the field frame of reference or the robot frame of reference. Normally the driver uses the field frame of reference because that is what he or she sees. The driver can select to drive using the robot frame of reference if the robot is driven with a video feed from the robot and the driver wants to move relative to what is shown on the video. The movement commands are sent as velocities rather than as coordinates, because that is more intuitive to the driver. There may be some adjustments to the raw joystick values to the values used by the robot because the joystick gives a normalized value between -1 and 1 and the drive train uses a velocity expressed in units of meters per second. Also the raw joystick values may be altered to fit a non-linear curve to give the driver more control over lower speeds than high speed.

Spinning the Robot

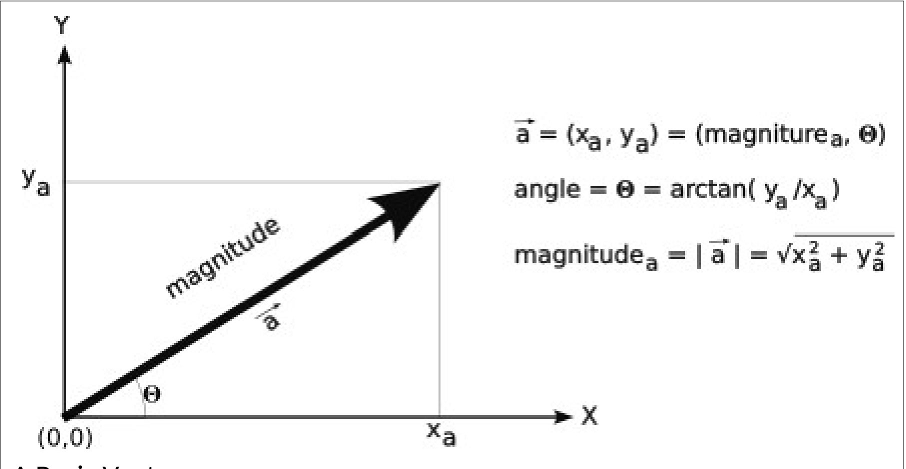
So simple enough. Push the joystick in the direction you want to move and the robot moves in that direction. What about rotation or spinning? Normally the robot rotates about its center and that is what we will discuss for the moment. The angle of rotation for each swerve module is tangential relative to a circle centered on the robot and passing through the center of the swerve module. This is illustrated in the figure to the right.

A minor problem is that each swerve drive has its own frame of reference for the rotation of its wheel. The angle of the wheel for rotation of the robot is dependent on the x and y distances of the center of the swerve module to the center of the robot. Robots need not be a shaped like a square, nor do the swerve modules need to be positioned symmetrically in the robot. These adjustments are made in the configuration constants used as swerve drive attributes.

Spinning the Robot While Moving

To spin the robot while moving requires the angle and speed of each swerve module be coordinated by the drive train to have a smooth and efficient movement. To do this, the software does a vector sum of the commanded movement vector and a rotational vector for each swerve module. It also has to translate the resulting vector relative to which ever movement reference frame is used to the reference frame used by each individual swerve module.

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 diection is from the northeast (or toward the southwest). For robot movements in the WPILIB the magnitude is expressed in units of meters per second. Rotational speeds are expressed in radians per second. Radians are used by the Java (and other programming languages) math routines.


Y
y_a
magnitude
$\vec{a}$
Θ
(0,0)
x_a
X
$\vec{a} = (x_a, y_a) = (\text{magniture}_a, \Theta)$
$\text{angle} = \Theta = \arctan(y_a / x_a)$
$\text{magnitude}_a = |\vec{a}| = \sqrt{x_a^2 + y_a^2}$
A Basic Vector

$$2 \cdot \pi radians = 360°$$

A vector can be expressed as the x and y components relatve to a reference frame or as the angle and magniture as shown in the figure above.

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:

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

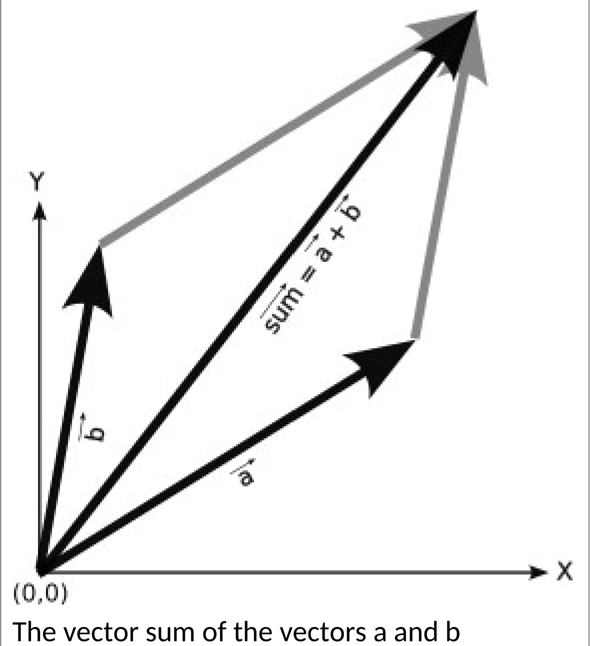
$$y_{new} = y_a + y_b$$

The resulting vector angle can be found by the formula:

$$angle_{new} = \arctan(y_{new} / x_{new})$$

The resulting vector magnitude can be found by the formula:

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



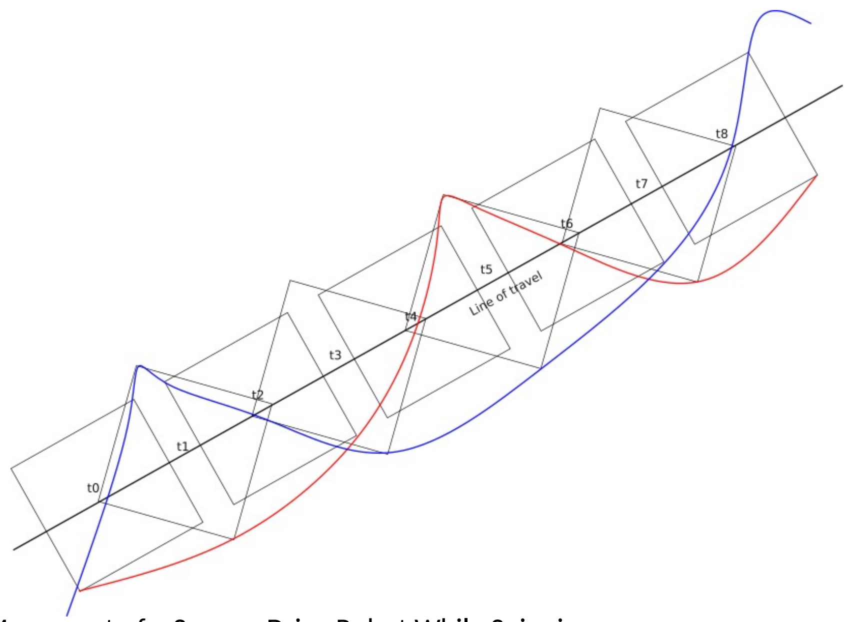
The vector sum of the vectors a and b

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

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

The swerve drive train interface uses four parameters: X velocity, Y velocity, rotation velocity, and a boolean to select either the field frame of reference or the robot frame of reference. The rotation velocity is the same for either the robot frame of reference or the field frame of reference. However, intenally it must convert the angular velocity to the same units as the X and Y velocities, i.e. to meters per second. Multiply the angulary velocity in radian per second by the distance (radius) of the swerve drive to the robot center. The angular velocity is perpendicular to the radial between the robot center and the swerve drive. The X velocity and Y velocity together define a robot velocity vector which is an angle and a magnitude.

The figure to the right shows a robot spinning as it moves forward. This shows two things: as the robot spins its apparent width changes and the speed of the robot is diminished to allow one side of the robot to spin. This is the default way to spin the robot while moving forward. The path of two corners is shown, one in red and one in blue. Time is marked along the line of travel. Because one side of the robot must slow down for the robot to spin, the overall speed of the robot is reduced. Spinning in general is not effiecient.

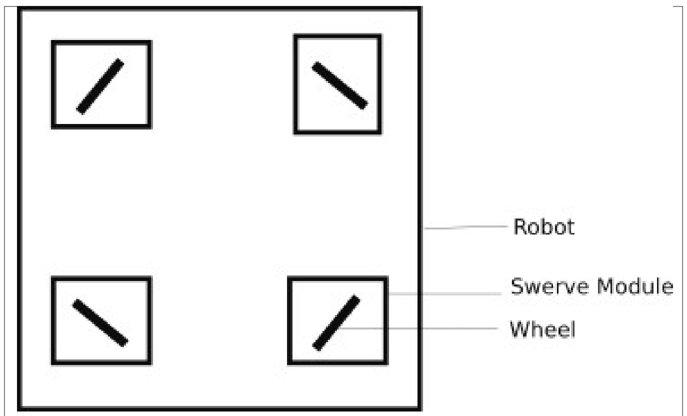


Movement of a Swerve Drive Robot While Spinning

Note that the arc tangent is a little more complicated than pure math. The normal math function will return an angle between -π/2 radians (-90°) and π/2 radians (+90°). We want the angle between 0 and 2π radians (360°), so the signs of the x and y determine the quadrant of the Cartesian coordinates which is used to extend the range of the returned angle in the Java robot math (Rmath) class.

Locking the Wheels on a Swerve Drive

To lock the wheels on a swerve drive to make it harder to move, turn all wheels to be "tangentially aligned" (i.e.. aligned perpendicular to the radial connecting a wheel to the center of the robot.) as in the figure to the right:



Position of Wheels When Locked in a Swerve Drive

Vocabulary

argument a positional parameter passed to a method, function or subroutine

attribute a variable associated with the instance of an object.

block a group of statements bound by curly braces ‘{…}’

CAN bus ia the Controller Area Network used for automobiles and FIRST robots.

controller a device that controls something. For example, this can be the software 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**.

declare define the name and type of a variable or attribute.

driver the human driving the robot and the same as a pilot. The low level code used to control a piece of hardware.

encapsulation really need to define this here.

FMS is the Field Management System which allows the field referees to control robots for simultaneous starts and safety.

heading is the angle that the robot is moving toward with respect to the field.

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**.

instantiation the creation of an object from a class template.

initialize set the initial value of a variable or attribute.

Lime Light is the brand name of a FIRST robotics vision co-processor.

operator is the human who controls accessories on the robot. Also called the co-pilot.

method a function or subroutine that performs an operation on the containing object

navX of **navX2** is the brand name of a FIRST robotics navigation co-processor.

operator is the human who controls accessories on the robot. Also called the co-pilot.

overloading is using the same method name with different signatures to do essentially the same thing but with different arguments.

parameter see argument

PID proportional-integrated-differential control, a mathematical procedure 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 yaw.)

pointing or **pose** is the angle of the robot with respect to the field.

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

public accessible outside of the object

PWM is Pulse Width Modulation which uses a timed wave form to dynamically change the amount of power delivered to a device.

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 yaw.)

RSL is the Robot Status Light which communicates the Field Management Status of the robot.

scope is where an object, variable, or parameter can be accessed. In Java scope is normally confined to the containing **block**.

shuffleboard is the display of telemetry data on the drive station.

signature is the returned primitive data types or classes and the primitive data types or classes of the passed arguments. For example:

int foo(int a, int b) is the same signature as int foo (int c, int d)

(int, int, int) = (int, int, int)

int foo(float a, float b) is a different signature than int foo(int a, int b)

(int, float, float) != (int, int, int)

static retained in memory. It is not necessary to create an object for a static class.

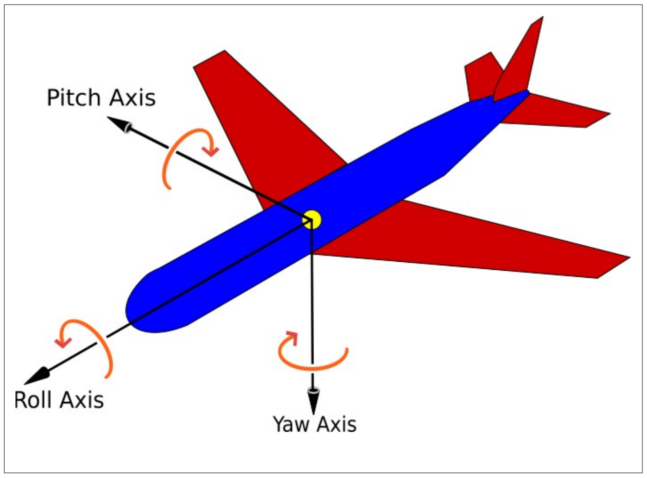
subroutine a routine that does not does not return a value.

type the primitive type (integer (int), float, double, boolean, char) or class (String or a locally defined class)

variable a named value with a specified type

WPILIB is the Wooster Polytechnic Institude (WPI) library for FIRST robotics

yaw is the angle of the robot (really inertial measurement device) about the z-axis and the initial inertial angle.



Java and Object Oriented Languages

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. In general computer languages fall into two major types: compiled and interpreted. Compiled languages translate source code into the native machine codes for the target computer. Interpreted languages are parsed at run time to perform the actions directed by the programming. Java is a hybrid type as it is compiled into a set of "byte codes" which are then stored in a ".jar" file. The ".jar" file is interpreted on a target machine by a Java runtime. Since the Java runtime is written specifically for the target machine, the .jar code is portable between different machines. The Java runtime is a subset of a normal Java runtime as it contains only the features needed by robot systems.

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, but 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.

A list of the key words (the words used by Java language and should not be used for the names of anything defined by the user) can be found with a short definition at https://www.w3schools.com/java/java_ref_keywords.asp. This does not include the use of "decorators", words preceded with an at '@' character.

Much of the syntax of Java is similar to C, Javascript, and a few other languages in that it:

- blocks of code are fenced off with braces '{' and '}'.
- expressions in a conditional are enclosed in parenthesis '(' and ')'.
- members of an array are accessed with an index enclosed in brackets '[' and ']'. For example to access the second element the array 'cars' one can write '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.

```
var robot1 = 1; // this comment go to the end of the line
var robot2 = 2;
/* this is a multi line comment
var = robot 3 = 3;
the above line is commented out, as is this one /*
```

- A statement must end with a mandatory semicolon ';'.

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 in another.

Java uses types extensively. They are part of the declaration of variable, attributes, arguments, and methods. They are checked when these are used in other parts of the program. There are some studies that conclude that highly typed languages are more reliable because the right things are being used and acted upon. So just what is a type. Java has four primitive types:

- **int** or integer for positive and negative whole numbers.
- **float** for positive and negative fractional numbers.
- **double** for more precise fractional numbers.
- **char** for single characters. Characters also use single quotes like ‘a’. Single quotes cannot be used to delimit a string as in other languages.
- **boolean** for values that are either true or false.

In the robot code, there are many uses of **int**, **double**, and **boolean** types. **float** and **char** are not used often, if at all.

Types are extended by classes. **String** is a built-in class to handle arrays of characters. User defined classes also extend type and robot code uses many of these classes.

An object oriented language allows the software to be written as a set of reusable components or modules. An object can be thought of as a “thing.” An object may be physical thing like a robot or a logical or abstract thing like an address. An object has a set of attributes which are used to define its characteristics. An object has a set of methods that manipulate the object in some way, like

- setting the value of an attribute.
- getting the value of an attribute.
- manipulating the attributes of the object.
- controlling the physical thing represented by the object. (e.g., move the robot forward)

Objects are defined as a “class.” There can be many objects defined with the same class. 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.

Different objects may have similar methods with the same name. This allow for treating objects similarly even if they are in different classes. For example, many objects have an initialization method called ‘init.’

Let’s work through a simple example for a normal street address.

```
Class Address (){
	String streetNumber; // allow numbers like ‘234 1/2’
	String streetName;
	String city;
	String stateAbbreviation;
	int zipCode;

	Address () {
		streetNumber = 0;
		streetAddress = “”;
		city = “”;
		stateAbbreviation = “”
	}

	Address ( String number, String street, String city, String state, int zip ) {
		streetNumber = number;
		streetName = street;
		city = city;
		stateAbbreviation = state;
		zipCode = zip;
	}

	setStreetName( String street) {
		streetName = street;
	}

	getFormattedAddress() {
		String line1 = streetNumber + “ “ + streetName
		String line2 = city + “ “ + stateAbbreviation + “ “ + zipCode
		return (line1 +”\n” + line2)
	}
}
```

This whole section needs to be re-worked. It is desireable to have examples of each topic and possibly graphics to highlight different things in the code.

What needs to be covered include:

- end of line comment
- multiple line comment
- partial line comment
- variable declaration
- variable assignment
- variable reference
- class declaration
- object instantiation (class reference)
- method declaration
- method reference within object
- method reference outside object
- differences between methods, functions, procedures, subroutines, etc.
- attribute declaration
- attribute assignment within defining object
- attribute assignement outside defining object
- attribute reference within defining object
- attribute reference outside defining object
- differerences between variables, constants, attributes, parameters, arguments, etc.
- use of a getter method
- use of a setter method
- use of modifiers in a declaration.
    - private
    - public
    - static
- passing parameters to a method
    - passed by value
    - passed by reference
    - when are parameters evaluated
- overlaying parameters for a method (polymorphism)
- what is a lambda function

- example of lamda function usage

Computer Science Topics

These probably should be covered by other sections (examples and definitions), but am including here just to make sure that each is discussed enough.

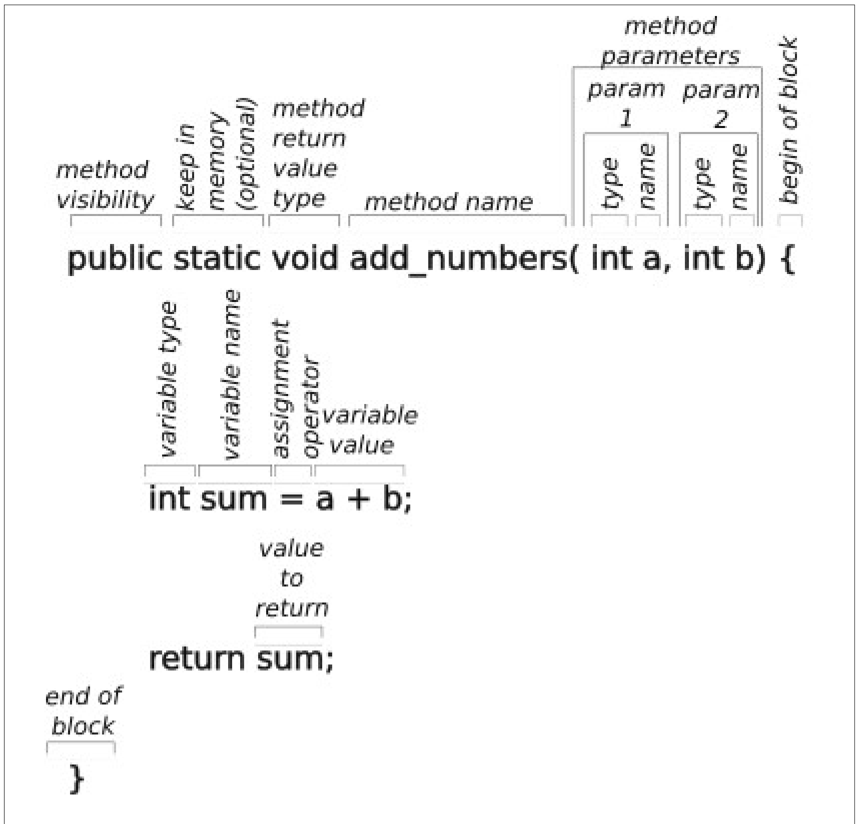
- Encapsulation
- Modiularization
- Inheritance
- Polymorphism

```
Class Address (){
       String streetNumber; // allow numbers like ‘234 1/2’
       String streetName;
       String city;
       String stateAbbreviation;
       int zipCode;

       Address () {
              streetNumber = 0;
              streetAddress = “”;
              city = “”;
              stateAbbreviation = “”
       }

       Address ( String number, String street, String city, String state, int zip ) {
              streetNumber = number;
              streetName = street;
              city = city;
              stateAbbreviation = state;
              zipCode = zip;
       }

       setStreetName( String street) {
              streetName = street;
       }

       getFormattedAddress() {
              String line1 = streetNumber + “ “ + streetName
              String line2 = city + “ “ + stateAbbreviation + “ “ + zipCode
              return (line1 +”\n” + line2)
       }
}
```

Methods

A method is a block of code that is used to manipulate an object.

An example of a method declaration is as follows:



Parts of a Java Method Declaration

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 github.com site should use markdown formatting and have the '.md' extension.
- Directories use lower case names.
- Names for variables, methods, attributes, classes, objects are usually descriptive and use camel case (start with lower case (except class names) and begin each new word with a capital letter. Names cannot contain spaces. Underscores are rarely used except to separate words that are capitalized, like WPILIB really should be WPI_LIB to separate the two words, but this was their design choice and it is pervasive.

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.

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.
- 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)'
- Conditionals should ALWAYS be written the same way with the blocks indented, so you don't have to think.
    - e.g., "if (foo) return true; return false;' should be written in the more general form as:

```
if (foo){
   return true
} else {
   return false
}
```

Don't waste a lot of time with style wars. Messy code shows a lack of thought and care and probably has reliability problems. Code that is straight forward and easy to read is probably more reliable.

Code Development

Source Code Editing

Source code is written on an editor or integrated development system. 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.

Debugging

All code that is writted 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. Test it and make sure the code is correct.

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.

Source Code Management (git)

A source code manament system keeps track of the changes to the software over time. It allows mulitple 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. Saving code to a remote repository is also a way to maintain a backup of the source code during development.

The basics to git (as used by github.com or gitlab.com) is to **add** the files that you want to include in a group of files which have been modified/

There are three phases:

- staging where modified files are **add**ed to a group of files to be included in a committed change. s**tatus** can be used to list the files which have been modified as well as the files staged for commit.
- **commit** where the modified files are marked as changed along with a reason for the particular change. In general commits should cover as small of a set of changes as possible to give granularity to the change.
- **push** is to change a remote repository with the committed changes on a local machine.
- (A pull is a a requested change outside of the normal source code control change. This lets outside developers propose changes to fix particular problems. This probably will not be used I robot code development, except to propose changes to the WPILIB code libraries.

Ideas

Collision Detection

The navigation subsystem will detect collisions. You can take the impact vector and subtract the last known robot momentum (E = mvv) vector to find the direction of the colliding robot. For automatic bump and run...check to see if the robot is clear of game structures, if it is, roll away from the colliding robot.

Defensive Rolling

The bump and run roll holds a corner of the robot against the other robot. The center of the robot will trace a scallop curve with the radius of the center of the robot rotating around a corner until another corner makes contact with the opposing robot. If the movement is to the left of the other robot, the spin is clockwise, otherwise it is counter clockwise. The rate of the spin is to keep the corners of the robot stationary. The swerve drive chain provides for the translation of the center of rotation to allow this.

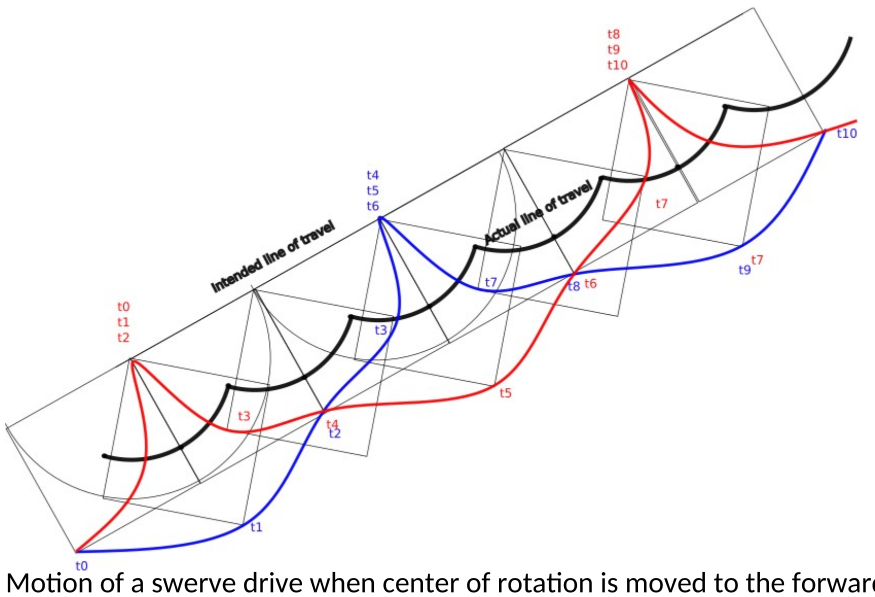
Logging Data for Analysis

Should turn on the logger code. This should be used to output timestamped pose estimates which can be used to create a heat map to point out where too much time is being spent. It can also more accurately report instantaneous velocities. It can also point to wheel slippage and other problems. It can help identify total cycle times and conponents of cycle times to point out areas for improvement.

Defensive Spinning

In the figure to the fight. 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 for defense as it does not “give up” any territory while spinning and moving forward. Again speed is sacrificed to spin while moving.

The path of two corners is shown, one in red and one in blue. The times for each corner is shown independently one corner will stop and the robot rotates about that corner.



Motion of a swerve drive when center of rotation is moved to the forward right corner and swung through 90 degrees for each corner of the robot.