Koalafied 2023 Robot Software

This document provides an overview of the code for Team #6996 Koalafied's 2023 Charged Up FRC robot.

Code Structure

The code consists of the following parts. The vast majority is C++ RoboRIO code with a small amount of Arduino code to control the LED strips.

- Robot Class - There is a single Robot class that owns the subsystems, but does very little.
- Robot Configuration - The RobotConfiguration.h header file contains lots of constant values for different parameters of the robot.
- Drivebase - The SwerveDrivebase subsystem class implement control of the swerve drive base This includes each of the four swerve modules.
- Manipulator - The Manipulator subsystem class implements control of all the robot mechanisms for dealing with the game piece. This includes the Pivot, Arm, Wrist and Intake.
- Leds - The Leds subsystem is very simple and handles sending commands to the Arduino that controls the LED strips on the robot.
- Autonomous Commands - In addition to the autonomous routines for the actual game we use autonomous to run a range of testing and tuning operations.
- Utility code - We have several helper classes in the 'util' directory.

Autonomous

Setup and creation of the command to run in autonomous is handled by functions in the AutonomousCommand namespace. During development, it is useful to use autonomous for a range of tasks, such as tuning, characterising and others. To decouple the various development tasks we want to run we have an IAutonomousProvider interface. The system works as follows:

- For development, AutonomousCommand creates a shuffleboard tab called 'Auto' which contains a chooser control to select which IAutonomousProvider will create the auto command.
- Each IAutonomousProvider creates its own shuffleboard tab for its parameters or places them on the 'Auto' tab if it only needs a bit of space.
- For actual competition, only the IAutonomousProvider for that year's auto routines is created and the UI for it goes on the main 'Drive' tab. (NOTE: This is not implemented yet because this stuff was all done after the last comp.)

LED Control

The LEDs on our robot are controlled by a separate Arduino board. This removes the burden of updating the LEDs from the RoboRIO, which has better things to be doing.

We use an Arduino Nano with the Atmega328 processor, which has 2kB of RAM, which is necessary to have large enough buffers for the LED (ATmega168 with 1kB RAM is not sufficient). The Arduino implements some different LED patterns using the FastLED library. The pattern is set by the RoboRIO via a very simple I2C message.

Instrumentation with Shuffleboard

We use the shuffleboard on the driver station to display information for normal robot operation and for debugging and development. To keep the output consistent and nicely formatted we use the ability to layout controls using code rather than editing them on the driver station (see WPILib documentation).
The shuffleboard layout and controls are set up when the drivebase is initialised in the Setup() function. Either the frc::SimpleWidget* or the nt::NetworkTableEntry can be recorded. The values are then updated in the drivebase Periodic() function.
One awkward issue is that the gyro widget will not work with this method. Instead, a Sendable class must be used. We use a very simple Sendable class that allows us to set the current angle for the gyro widget. This trick was copied from the Team 7443 code. Note that getting this right is very important. If it is not done correctly it is not just the giro that does not work, but often most of the rest of that shuffleboard tab will fail, making it hard to track down the problem.
Note that the position of the controls on the shuffleboard is relative to a notional ‘tile’ grid. By default, the tile size is 128, which is very large and makes it hard to fit a lot of controls. We change the value to 32, which is done from the Shuffleboard **File > Preferences** menu.
All the shuffleboard code is in a separate class for each of the two major subsystems - so there are SwriveDrivebaseShuffleboard and ManipulatorShuffleboard classes. Each subsystem creates and owns the corresponding shuffleboard class and passes it a constant pointer to itself. For development and testing, there are a large number of tabs that can be created. For actual competition most would be commented out to avoid driver confusion. This would also save bandwidth, but that is probably not too important as testing has shown that NetworkTables communication does not use a lot of bandwidth.

Simulation

WPILib does not currently support simulation of swerve drivetrains so we have implemented a simple simulation directly. This simulation is not intended to be realistic. The aim is to have the drivebase work in the simulator so we can test code, but we do not expect to predict exact robot behaviour correctly. In particular, the simulation does not take into account the physics of robot behaviour, such as the momentum of the robot. Instead it simply uses the DC motor behaviour equation to calculate the motor speed from applied voltage. See SwerveModule::UpdateSimulation() for details.
We have found that this simulation is sufficient to allow the development of a lot of code without needing a robot.

Utilities

We have several utility helper classes located in the ‘util’ directory.

Haptic Feedback

X-Box controllers have a mechanism to provide haptic feedback. This is very useful for providing information to the driver and operator when they are concentrating on the field. For example, it can be used to indicate that an arm has reached a selected preset position, or that the intake has detected that a game piece has been grabbed.
To make haptic feedback easier to use we have a HapticController class that can ‘play’ haptic patterns. This means that when an event occurs we can play a pattern without having to worry about updating it.

POV Filter

The X-Box POV control is useful for operations like selecting preset operator positions (like levels 1, 2 and 3 in this year's game). However, it is easy for the user to accidentally select a diagonal direction when they mean to select one of the 4 main directions. The PovFilter class is a very simple way to deal with this. It filters input from the POV so that it only returns the four major directions, but also returns the previous major direction if the user slips into a diagonal direction.

Logging

Logging is useful for a lot of different development and tasking tasks. The Logging::CsvFile class provides a very simple, but easy-to-use interface for writing CSV log files. It handles quoting the values and the different directory layouts on the RoboRIO and desktop.

KoalafiedUtilities

The KoalafiedUtilities namespace has a range of useful functions for the following:

- Applying a sign preserving ‘power adjust’ for joystick input (usually used for squaring)
- Converting CTRE Talon SRX and Talon FX velocities between the native units and RPM.
- Performing ‘tuning’ driving of CTRE Talon SRX and Talon FX controllers. This is open and close loop control with automatic calculation and logging of the F and P values according to the CRTE velocity close loop tuning instructions.
- Mathematical utilities for mathematical modulus, clamping and angle normalisation.

Notes

We do not put .h and .cpp files in separate directories. Although this is often done on *nix systems it is not necessary and we feel it makes the code harder to navigate, especially for novice programmers.

SwerveDrivebase

The SwerveDrivebase class is the subsystem that controls the drivebase.
Each serve module is controlled by an instance of the SwerveModule class. This code structure, copied from the WPILib example, is a neat way to partition the code, which is particularly effective because of the ‘narrow’ interface between the drivebase and each module.
Our swerve drive uses Swerve Drive Specialties MK4i Swerve Modules with the L1 gear ratio and Falcon 500 motors. We also use CANcoders to measure the absolute position of the steering shaft of the module.
The following sections describe the main classes and different functionality of the code.

SwerveDrivebase

The SwerveDrivebase has the following key functions.

- Setup() function that initialises the drivebase code
    - Creates and initialises the four SwerveModules, the Pigeon IMU and the Xbox controller
    - Creates and initialises the swerve kinematics and odometry classes.
    - Loads the steering calibration data and passes it to the swerve modules
    - Sets up a load to shuffleboard reporting
- Periodic function that deals with updates
    - Updates the odometry from the Pigeon IUM and swerve module states (speed and direction).
    - Updates the display of shuffleboard data
- Drive() takes x and y speeds and a rotation speed and updates the swerve modules to achieve it. This uses the swerve drive kinematics class.
- DoJoystickControl() reads the inputs from the Xbox controller and updates the robot's movement accordingly. We use several buttons for testing functions but the main operation of the robot is as follows:
    - A deadband is applied to each joystick axis, using a deadband value measured for the controllers we use.
    - A maximum rate of change to the input is applied using a frc::SlewRateLimiter. This means we do not need to use a ramp rate in the motor controller.
    - A ‘slowdown’ factor, using a value from the dashboard, is applied, which we use on all robots to have a way to test them in a controlled manner in confined spaces.
- Functions to deal with CANcoder steering calibration as described below.

SwerveModule

The job of each swerve module is to point the wheel in the right direction and drive it at the correct speed. The speed and state for a swerve module are wrapped up in the frc::SwerveModuleState struct. The key functions are:

- Functions to get the current state of the swerve module and to set the desired state. Note that it takes time for the motors to adjust the speed and direction of the module, so the current state does not become the desired state immediately, and in fact it will never be the desired state exactly.

- A function to get the absolute encoder angle and to set the absolute encoder angle for the zero point. These are used in calibration.
- A bunch of functions for testing

Swerve Module Speed Control

Control of the module speed motor uses the velocity close loop PID controller in the Falcon FX motor controller.
We tune the parameters for the PID control using some utility tuning code that we use for all our mechanisms (see class KoalafiedUtilities). Essentially this performs the operations suggested in the CTRE Phoenix documentation, which in summary is as follows

1. Run the motor in open-loop and calculate F, the feed-forward gain.
2. Run the motor in velocity closed-loop and calculate P to reduce the error to an acceptable value.
3. Leave I and D at 0 unless there are problems, which we have not had to do for a drivebase.

Swerve Module Direction Control

Control of the module steering motor uses the motion magic control mode PID controller in the Falcon FX motor controller. This mode moves the motor to a given encoder position but with control of the maximum speed and acceleration of the movement.
Note that the controller on the motor uses the encoder on the motor for the control loop, but we want to make sure that the position is correct according to the CANcode which knows the absolute position. To achieve this we work out the delta that the steering needs to turn using the CANcode to measure its current absolute position. Then we scale the delta by the steering gear ratio and use the Falcon controller to move by that delta from its current position. See SwerveModule::SteerToAngleMotionMagic() for details.
Note that this method allows us to take advantage of the FalconFX controller, with its fast 1ms update cycle, while still using the CANcoder for the correct absolute positioning of the steering.
Note that we currently do not have code for tuning the steering motor. Some initial guesses for PID parameters seem to be doing a good job.

Calibration

The CANcoder gives an accurate measurement of the absolute angle of the swerve module angle, however, the zero point for each module is unknown because it depends on the angle at which the magnet is glued into the vertical module shaft. For this reason, we need to measure the zero point by aligning the modules to the 0-angle position and recording the absolute angle from the CANcoder.

The zero point angle is then stored in a file on the RoboRIO. Each time the robot code starts the calibration file is read and the zero point angles are used to convert the CANcoder angle to the true position of the module.

Note that the 0 angle position does not just require that the wheels be aligned front to back on the robot as there are two angles (0 and 180) that this could be. The 0 angle is the direction such that the drive motor moves the robot forward. This angle was determined by testing the robot on blocks and marking on the robot which side the bevel gear on the swerve motor wheel needs to be.

Manipulator

The manipulator subsystem handles the parts of the robot that are controlled by the operator, which is everything except the drivebase.

Division into Mechanisms

To simplify and structure the code we have a number of mechanism classes, each of which performs a specific action. The overarching manipulator class helps to manage the potentially complex interactions between mechanisms and simplifies the command structure. The 2023 robot has four mechanisms: the pivot, arm, wrist and intake.

Note that in many WPILib examples, these mechanisms would be subsystems (i.e. derived from frc2:SubsystemBase), however, that is not an appropriate or helpful structure. A subsystem should be something that can be the target of a command, which is appropriate for the Manipulator as a whole, but not for the individual mechanism. There are many complex interactions between the mechanisms that the Manipulator class needs to enforce. For example, the pivot should be moved more slowly when the arm is extended. If the mechanisms were subsystems then such interactions would need to be enforced in complex parallel commands, which is difficult or impossible and means important behaviour is spread around instead of being encapsulated in one place. We have used this approach in our code for many years and found it a ‘good idea’™.

Controller Scheme

To reduce complexity for the operator, the manipulator control interface uses preset positions mapped to controller buttons, in combination with different modes for handling cones and cubes. The current game piece (cone or cube) and a position button (stowed, level 3, level 2, ground and shelf) specify a preset position in the code for the manipulator to drive to.

If manual position control is required, the triggers and joysticks can be used to move each mechanism.

Haptic feedback is used to inform the operator that a preset position has been reached.

Preset Positions

Each preset position in the code specifies three values, a pivot angle, a wrist angle, and an arm extension. The starting position is set to be against one of the limit switches for each positional mechanism so that the position is known at startup. To reduce drift in the positions from repetitive movements, when the manipulator is stowed, each mechanism drives to its limit switch and re-zeros the position values.

For testing, there exists an extra position specifically designed for tuning. Using a controller input, the current position can be saved to the custom position. The current values for the custom position are available on the “Manipulator” shuffleboard tab and can be edited to provide live tuning.

To aid with manual position control, the pivot and wrist are in a virtual 4-bar configuration, meaning that as the pivot moves, the wrist stays at the same angle relative to the ground.

To reduce steady-state error, we implemented a hybrid control system for the positional mechanisms. While the error is larger than a given threshold, motion magic velocity and acceleration control is used to move toward the set position quickly and smoothly. Once the error is below the threshold, the motor controller switches to using positional control with different tuning parameters to provide low steady-state error and no oscillations.

Pivot

The pivot class is responsible for driving the pivot to an angle specified by the manipulator. Because of the large torques on the pivot, the resistance of the motor and gearing is not sufficient to hold the pivot at a specified position. Our solution is to have a member variable equal to the target angle, that the pivot is constantly driving towards, ensuring that the pivot angle does not drift.

To prevent the pivot from being driven beyond the bounds of its desired range of motion, both forward and reverse limit switches are used. The forward limit switch (stowed position) is also used to re-zero the current angle of the pivot, reducing drift in the pivot angle. As an additional safety measure, the pivot is subjected to a slowdown factor that scales with how far the arm is extended, if the arm is extended more than 15 inches.

Arm

Unlike the pivot, the weight of the arm is not enough to back-drive the motor, so there is no need to drive towards the set position once it has been reached.

To prevent the arm from slamming into either end of its range of motion, the input speed for manual control is halved when within five inches of the end it is moving towards. There are also limit switches at each end of the arm's movement to stop it from damaging itself.

Wrist

Unlike the arm and pivot, the wrist only has a limit switch at one end of its range of motion, at the stowed position. To ensure that the wrist does not move too far in the other direction, a soft limit is used.

Intake

The original intake design for this year used two arms with attached rollers, with a winch to open the arms and springs to close them around the game piece.

The winch position relies on a relative encoder count, and it was not possible to add limit switches, so to initialise the position of the winch motor the intake was fully opened when placing the pre-loaded game piece in autonomous. This caused a current spike allowing for the winch encoder value to be recorded, with positions calculated as an offset from the 'fully open' encoder value.

New Intake

Our new intake was a dual roller design similar to many other teams this year.
When 'intaking' a game element the NewIntake class uses a simple state machine that goes through the following states.

1. **Idle** - initial state when not intaking is occurring
2. **Intaking** - runs the rollers, and checks for high current (meaning that a game piece is coming it and causing the motor to work harder). When a high current is detected it moves to the HighCurrent state.
3. **HighCurrent** - Continues to run the roller for a timeout and then moves to the HasGamePiece state.
4. **HasGamePiece** - stops the roller

The roller speed, high current level and high current time are all configurable and are different for cubes and cones.
When dropping a game piece the rollers are simply run at full speed in the opposite direction.
We have a logging system for recording the time, motor output and current to test and debug the operation of the intake.

Acknowledgements

Our swerve code was originally based on the WPILib SwerveBot example, but the code has been almost entirely rewritten. We learnt some useful things about the Shuffleboard from the Jagwires Team 7443 Swerve code.