Advanced FRC Controls Programming

GeorgiaFIRST Symposium
November 5, 2016

Gary Lewis
Mentor, Kell Robotics, Team 1311

Presenter Background

- Mentor, Kell Robotics, Team 1311
- Professor Emeritus of Computer Science and Physics, Kennesaw State University
- Ph.D., Physics, Georgia Tech
- Industrial experience
- Experience in electronics, chip manufacturing, embedded systems, robotics ...

This Presentation

- Oriented toward people with some experience in FRC programming.
- A copy of this presentation is available at the Kell Robotics GitHub site:

  http://github.com/kellrobotics/

- Look under “FRC-Programming”
- Additional supplementary material will be posted in the next few days.

A Very Important Page

- The FRC documentation is essential, even for advanced topics.
- Currently at:

  https://wpilib.screenstepslive.com/s/4485

- A search on “FRC Control System” should also get you there.

Topics

- Command Based Programming
- PID Control Systems
- Driver Station Communication, Display, and Input (briefly)

I will focus on an orientation in these areas, along with reviewing some of the pitfalls. I intend that this will be a good supplement to the usual references.

Programming Base Types

- When you create a programming project to control an FRC Robot, there are three options:
  - Sample: should be used only for advanced programming projects since a great deal of development is required for a real competition program.
  - Iterative: recommended for starting out, since simple programs can be done within a single file.
  - Command-Based: recommended when moving beyond the basics for almost all teams.

When to Move to Command-Based

- If you are planning autonomous mode operation that is more complex than “drive in a straight line”
- If you are using more than a couple of buttons to control action on the robot.
- If you are using encoders and other feedback devices in more than a very basic mode.

Issues with Command-Based?

- There is no issue in terms of performance or capability – Command-Based is an extension of Iterative, so it can do anything you can do in Iterative.
- There is slightly higher initial work required, but that is offset by having a much more maintainable system once it becomes complex.

Command-Based Overview

- Allows building commands to do things like:
  - Drive forward x inches
  - Turn y degrees left
  - Raise elevator to position z
- Commands are classes and can be combined so they are executed in series or parallel in Command Groups
- Commands and Command Groups can be:
  - Called during autonomous
  - Linked to a button or control at the driver station

Subsystems, etc.

- Commands call subsystems to actually accomplish their tasks.
- Commands are oriented around an action to be taken.
- Subsystems are oriented around specific mechanical parts of the robot – such as drive system, turret rotation, crane, elevator, winch, etc.

Iterative vs. Command-Based

- In Iterative, you initially get a single package with a single class, Robot.java.
  - That class serves as your “main” program (the true main class is part of the underlying structure that you are expanding).
  - You can add packages and classes as you need them.
- Command-based sets up a more complex structure from the beginning (next slide).
  - You can still add packages and classes.

Command-Based Structure

- xx.robot package, containing classes:
  - Robot.java
  - OI.java
  - RobotMap.java
- xx.robot.commands package, containing an example command.
  - Commands and Command Groups go here by default
- xx.robot.subsystems package, containing an example subsystem
  - Subsystems go here by default

New Files in Robot Package

- OI – Operator Interface
  - A location for all of the routines that access the driver station controls – joysticks, buttons, etc.
  - Associate Commands with controls here also.
- RobotMap
  - Single location for port numbers, constants, etc.
  - Any changeable number should go here with a named constant that is used elsewhere in the program.
- These are highly recommended, and set up by default, but you could use other names or locations.

Building the System

- Planning should start with the bottom of the previous list, subsystems:
  - Any equipment involving motion on the robot should be controlled by a subsystem.
  - Separate subsystems should be used for each set of items that could be controlled independently. This allows using the powerful feature that prevents one command from using a subsystem until others are finished.
  - Complex systems can require careful planning. Should a crane with rotate and tilt be one system or two? That does not have an absolute answer.

Subsystem Coding

```
public class ExampleSubsystem extends Subsystem {

    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    public void initDefaultCommand() {
        // Set the default command for a subsystem here.
        //setDefaultCommand(new MySpecialCommand());
  }
}
```

Sensors, etc.

- If a sensor, such as an encoder, is always used with particular motion devices, then generally it should be included in the same subsystem.
- Encoders on drive wheels should almost always be part of the “drive” subsystem.
- A gyro could be included with the drive subsystem, but is more debatable.
- A complex navigation board probably should not be included with the drive system, since it may have functions outside that system.

Sensors II

- A sensor should not normally be in a subsystem by itself – there is no need. You can just write a regular class with methods that return appropriate values.
- We use subsystems particularly for their ability to block access from multiple commands and for their ability to specify a default command, neither of which is needed for a pure sensor.

Subsystems – Example: Drive

- All communication with the drive motor controllers and their encoders should be done by the drive subsystem.
- Commands should communicate with the subsystem through its methods.
- Avoid public variables in the subsystem – use methods.
- In general, only commands should use the methods of the subsystem. The rest of the program should use commands for access.
  - Possible exception: methods that are read-only, such as reading distance traveled.

Next, Commands

- Once you have a subsystem, you can define commands.
- Start with basic ones, then add others as needed.
- Remember, nothing should control any motion on the robot except by using a command.
- You can now add default commands for subsystems where appropriate.

Default Commands

- For each subsystem, you have the option of defining a default command.
- This command will operate when no other command for that subsystem is active.
- For example, for the drive subsystem, you would normally define a joystick drive command as the default.
- This would cause the joysticks to operate normally, but you could, say, have a button associated with a command to turn 180 degrees. That command would return control to the joysticks as soon as it finished.

Command Coding I

```
public class ExampleCommand extends Command {
    public ExampleCommand() {
 // Use requires() here to declare subsystem dependencies
        requires(Robot.exampleSubsystem);
    }
 // Called just before this Command runs the first time
    protected void initialize() {
    }
 // Called repeatedly when this Command is scheduled to run
    protected void execute() {
    }
```

Command Coding II

```
    // Make this return true when this Command no longer needs to
    run execute()
    protected boolean isFinished() {
        return false;
    }
    // Called once after isFinished returns true
    protected void end() {
    }
    // Called when another command which requires one or more of
    the same
    // subsystems is scheduled to run
    protected void interrupted() {
    }
}
```

Command Group

```
public class AutoMode extends CommandGroup {

    public  AutoMode() {
      addSequential(new DriveDistance(20,0.2));
      addSequential(new Turn(-90,0.3));
      addSequential(new DriveDistance(-10,0.2));
    }
}
```

Putting it Together

- The program Example-Command-Based is a simple example with comments posted on the Kell GitHub site.
- It uses the standard convention for Command-Based Autonomous and calls a command group to execute a series of commands. This requires several statements in the Robot.java class.

Multiple Autonomous Programs

- If you want to select between multiple options for autonomous without having to reload programs, that can be done with command based.
- Look at the FRC documentation under Smart Dashboard and Sendable Chooser.

What is PID?

- The recommended way to set up effective feedback control.
- Typical applications:
  - Controlling angle of a shooter using an encoder
  - Controlling the velocity for a drive system using the wheel encoders
  - Moving to a set distance without overshoot.

What is the Situation?

- We want to move to a particular position using a feedback sensor (encoder, etc.)
- We would like to move rapidly, stop without overshoot, and hold the position.
- Typically, a system with simple feedback will overshoot and oscillate about the target.
- If we slow the system down, we may be able to eliminate overshoot, at the expense of slow motion.

PID Definitions

- So we modify the “correction” signal with the following terms:
  - Proportional – proportional to the “error” (difference between current and desired value)
  - Integral – proportional to the total error (sum of all previous errors)
  - Derivative – proportional to the rate of change of the error.
- Many systems allow for an additional feed forward (FF) term for cases where you may have some starting info that can be applied.

Correction Factors

- Programs will normally have a constant factor for each of these – Kp, Ki, Kd.
- For slow systems, sometimes Kp alone is enough.
- Kd will tend to damp out the oscillations on a faster system.
- Ki works to offset in built in bias to the system. For example, in an elevator system, the weight will tend to pull it below the set point. The I factor accumulates and compensates for that.

How to Accomplish

- I will talk briefly about three ways we have implemented PID control on FRC robots:
  - Routines are available in the FRC libraries.
  - The new Talon SRX motor controllers have a built-in processor that handles PID and the capability for plugging encoders in directly. They work very well for velocity control.
  - Write your own code.

The Hardest Part?

- Getting the constants (Kp, Ki, Kd) set to work well is often more difficult than the programming.
- You can find quite a number of schemes from just a recommended sequence of changes to a high level analysis of a detailed model of the system (not really feasible for our situations).
- I have a summary sheet of my recommendations for our students that I have posted to the Kell GitHub site.

Using the FRC Routines

- It is possible to use a library class “PIDCommand” to set up a command with the PID function built-in.
- We have generally found that to be too restrictive, and so have used the “PIDController” class, which is based on the same routines and is more general purpose. It can be used either at the command or subsystem level.
- I will post example code in the near future.

PID Input and Output

- The FRC PID routines require specific classes of objects for their input and output routines. Drivers for most of the common input and output devices (encoders, motor controllers, etc.) have the appropriate “hooks” to work with those routines.
- However, if you have something non-standard, i.e., something that gives or gets a straight numeric variable; then you have to dig deeper and gen up the appropriate class structure for those routines.

Talon SRX Controllers

- The Talon SRX controllers have a fairly complete PID system built-in.
- They are connected to the roboRIO over the CAN bus, which is used to control the system as well as reading any encoders plugged into the Talon.
- We've had success using them for:
  - Velocity control of individual wheels
  - Reading both an absolute encoder and a quadrature encoder with one SRX.
- My posted PID summary sheet has some Talon info.

Roll Your Own

- The algorithm for PID is pretty simple. On a regular periodic basis, you execute something like:

  ```
  error = targetValue-currentValue;

  totalError =+ error;

  result = Kp*error + Ki*totalError + Kd*(error-prevError);

  prevError=error;
  ```

- Features can be added, such as having a tolerance value for ignoring error below a certain value.

Custom PID

- When we had a heavy arm we were trying to control, I suggested a custom algorithm that used the previous one with a correction factor that added in a value calculated to roughly correct for the weight based on the sine of the arm angle.
- That simplified tuning significantly.

Movement Application

- If you are trying to move smoothly to a point in Autonomous, then then very good results could be obtained by controlling velocity using PID, then programming a velocity profile calculated to arrive at the destination with zero speed.
- However, either PID control of distance traveled, or just using a profile table will improve the situation.

Robot/Driver Station Communication

- Most communication between robot and driver station is handled by a data structure called NetworkTables, operating over the network.
- Another Kell mentor, Andy Cash, is talking later on vision processing, and he will address using NetWork tables to pass information from an additional processor to the roboRIO.
- It is also possible to add variables to pass back to the driver station. Look at the SmartDashboard routines for that purpose.

SmartDashboard, etc.

- There are two different SmartDashboards that you can run on the Driver Station.
- I’m not too happy with either of them for advanced functions. Last year we wanted to be able to switch between two cameras on the robot and display some custom widgets. Neither of the dashboards had the features we wanted.

Custom Dashboard

- We ended up using a custom version of a system originally developed by team 1418.
- This requires a lot of development and training. Not sure if we will continue.
- That system is available here:

  https://github.com/FRCDashboard/FRCDashboard

Custom Controls

- The standard driver station can use inputs from usb joysticks and game controllers.
- It is also possible to design you own controls, buttons, lights, etc., that work over USB.
- Texas Instruments sells a small “Launchpad” board that can be used to connect your controls to USB. Under $15, so the big cost will be the other parts and the time required. It can be fun.

http://processors.wiki.ti.com/index.php/MSP430_LaunchPad_Operator_Interface_for_FIRST_Robotics_Competition

End