FRC C++ Programming How-To’s

Contents

[Projects 2](#_Toc466985023)

[Creating a new project 2](#_Toc466985024)

[Moving a project to the Git repository 2](#_Toc466985025)

[Operator Interface – Joysticks 3](#_Toc466985026)

[Subsystems 4](#_Toc466985027)

[Commands 6](#_Toc466985028)

[Creating a new command 6](#_Toc466985029)

[Configuring the command 6](#_Toc466985030)

[Running the command 8](#_Toc466985031)

[Operator Interface – Buttons 9](#_Toc466985032)

[Command Groups 10](#_Toc466985033)

[Creating a new command 10](#_Toc466985034)

[Sequential vs. parallel command execution 10](#_Toc466985035)

[Adding commands to the group 10](#_Toc466985036)

[Detailed Examples 10](#_Toc466985037)

[Cheat Sheet 13](#_Toc466985038)

[New Subsystem 13](#_Toc466985039)

[New Command 13](#_Toc466985040)

[Tying a Command to a Joystick Button 13](#_Toc466985041)

[New Command Group 14](#_Toc466985042)

# Projects

## Creating a new project

To try out new ideas or functionality away from the clutter of existing code, create a new project.

1. Open Eclipse.
2. *File 🡪 New 🡪 Project*…
3. *WPILib Robot C++ Development -> Robot C++ Project*.
4. Team Number = 4917 (if asked).
5. Choose a name.
6. Select *Command-Based Robot*.
7. *Finish*.

The project is now ready to use. You may have noticed that the project template includes an ExampleCommand and ExampleSubsystem. Once you have your own commands and subsystems defined, you can remove these example files.

## Moving a project to the Git repository

By default, new projects are created in your workspace directory. This may not be where you want it to live. For example, maybe you want it in your Git repo so you can share it with the team. To move an existing project, do the following.

1. In Eclipse, right-click the project you want to move and select *Delete*.
2. Make sure that *Delete project contents on disk* is **NOT** checked.
3. Click *OK*.
4. Using Windows Explorer (or similar), move your project folder to the new location.
5. In Eclipse, go to *File 🡪Import*.
6. Select *General 🡪 Existing Projects into Workspace*.
7. Click *Next*.
8. Click *Browse…* next to the *Select* *root directory* box.
9. Make sure your project is checked and click *Finish*.

# Operator Interface – Joysticks

Other than running Auto, you can’t do much with the robot without configuring a joystick so it’s good to set up the basics early on. The joystick is handled in the OI class. OI stands for Operator Interface.

1. Open your project’s src/OI.h from the Project Explorer pane
2. Just below the includes, add a constant that links a user-friendly name to the joystick’s number (0 to 5)

e.g. const int DRIVER\_CONTROLLER\_PORT = 0;

1. In the class’s private section, add a Joystick pointer

e.g. std::unique\_ptr<Joystick> driverController;

1. Open src/OI.cpp
2. Create the joystick in the class’ constructor (OI:OI())

e.g. driverController.reset(new Joystick(DRIVER\_CONTROLLER\_PORT));

1. Save all of the files\* and build the project\*\*

\*To save all files, do a CTRL-SHIFT-S.

\*\*To build the project, right-click your project in the Project Explorer and click Build Project. Make sure that the build didn’t generate errors by clicking on Eclipse’s Console tab.

That’s it. You’re now ready to make use of the joystick(s).

# Subsystems

A “subsystem” is a group of all of the actuators and sensors that make something work. For example: the drivetrain, the box stacking mechanism, the ball shooting mechanism, the shot-alignment mechanism.

### Defining a new subsystem

1. Right-click the project folder in Eclipse’s Project Explorer.
2. Choose *New 🡪 Subsystem.*
3. For the Class Name, enter a descriptive name followed by *Sub*
   * e.g. DrivetrainSub
4. Click *Finish.*

The subsystem is now defined (we know what it looks like) but it doesn’t actually exist in a way that the robot can use.

### Adding the subsystem to the robot

1. Open src/CommandBase.h
2. Add an include for the new subsystem

e.g. #include “Subsystems/DrivetrainSub.h”

1. Create a pointer to the new subsystem by adding a line to the public section of the *CommandBase* class (right above the OI entry).

e.g. static std::unique\_ptr<DrivetrainSub> drivetrainSub;

1. Open src/CommandBase.cpp
2. Create the subsystem by adding a line in CommandBase::init().

e.g. drivetrainSub.reset(new DrivetrainSub());

1. Open the subsystem’s cpp file

e.g. src/Subsystems/DrivetrainSub.cpp

1. Change Subsystem(“ExampleSubsystem”) to Subsystem(“*YourNewSubName*”)

e.g. DrivetrainSub::DrivetrainSub() : Subsystem("DrivetrainSub")

The robot can now use the new subsystem, but it doesn’t do anything yet.

### Setting up the subsystem’s hardware

1. Open the subsystem’s h file

e.g. src/Subsystems/DrivetrainSub.h

1. In the private section of the class, add a pointer for each piece of hardware that this subsystem uses. If you’re not sure what the subsystem will eventually use, add what you know and come back to update it later.

e.g.

std::unique\_ptr<Talon> leftMotor1;

std::unique\_ptr<Talon> leftMotor2;

std::unique\_ptr<Talon> rightMotor1;

std::unique\_ptr<Talon> rightMotor2;

1. In the public section of the class, add a list of methods that can be used for controlling the subsystem. You can modify this list at any time.

e.g. void drive(float rSpeed, float lSpeed);

1. Open the subsystem’s cpp file again.
2. In the constructor (e.g. DrivetrainSub::DrivetrainSub()), initialize all the actuators and sensors that were defined in the h file. If the h file ever changes, don’t forget to update this section too.

e.g.

leftMotor1.reset(new Talon(LEFT\_MOTOR1\_PORT));

leftMotor2.reset(new Talon(LEFT\_MOTOR2\_PORT));

rightMotor1.reset(new Talon(RIGHT\_MOTOR1\_PORT));

rightMotor2.reset(new Talon(RIGHT\_MOTOR2\_PORT));

Note: This example assumes you’ve already defined the motor ports in src/RobotMap.h

1. Add the code for every method that was defined in the h file. Add, remove or modify them whenever the h file changes.

e.g.

void DrivetrainSub::drive(float rSpeed, float lSpeed)

{

leftmotor1->Set(lSpeed);

leftmotor2->Set(lSpeed);

rightmotor1->Set(rSpeed);

rightmotor2->Set(rSpeed);

}

1. Save all the files and build the project.

Your subsystem is now ready to be used.

# Commands

Commands are a list of one or more actions (really just calls to methods) that accomplish something over time. For example:

**Suck-in a ball**

1. Set the intake motor to rotate inwardly.
2. Keep checking the limit switch to determine if it has detected the ball.
3. Once it has detected the ball, stop the motor.

Commands are usually triggered by the joysticks.

## Creating a new command

1. Right-click the project folder in Eclipse’s Project Explorer.
2. Choose *New 🡪 Command.*
3. For the Class Name, enter a descriptive name followed by *Cmd*
   * e.g. BallIntakeCmd
4. Click *Finish.*

The command is now runnable … but doesn’t do anything yet.

## Configuring the command

Commands are made up of 6 methods and each of these methods lets you configure a different part of how the command works. When creating a new command you should think about what, if anything should go into each of these methods. Open src/Commands/*Name*Cmd.cpp and fill in the code as follows.

1. **The Constructor (i.e. the method that has the same name as the command)**

Here is where you list all of the subsystems that this command requires. If you will be calling a method from a subsystem anywhere in this command, then you should add a *Requires* entry for it. This is important because the robot has the ability to run multiple commands in parallel but doesn’t want to run two commands that use the same subsystem at the same time. For example, you don’t want to try going forward at the same time that you reverse.

e.g If this command will use the intake subsystem to move the robot, add Requires(intakeSub).

Note: If you get a casting error when using *Requires* with a unique\_ptr, use this format: Requires(intakeSub.get());

1. **Initialize()**

When a command is executed, the first thing it does is run this code. This is a good place to do any prep work for your command. For example, if you need to unlock your wheels before you can move, you would call the wheel-unlock method here.

For simple commands that don’t do something repeatedly and that don’t need to wait until something else happens to finish, you may only need to put code in this method. For example, if the command opens your robot’s hand with a pneumatic actuator, you would just call something like armSub->openHand() and your command is complete.

1. **Execute()**

This method is called continuously (several times a second). Here is where you put code that handles continuously handles input changes.

For example, if you are driving using a joystick, you want the motor speed to change based on how far the joystick is moved forwards or backwards. To do this, you’d add something like drivetrainSub->setSpeed(oi->getStickValue()) . That way, every time *Execute()* is called, the motor speed would get updated with the latest value from the joystick.

1. **IsFinished()**

This method determines when the command is finished. It’s called after every Execute() to determine if the command should keep running of not.

When it comes to this method, there are three types of commands.

1. Do something in Initialize() and that’s it. These commands simply return true; (*I’m done*). Continuing with the open-hand example, once you’ve told the robot to open the hand, you are done. You don’t have to check if to see if it’s opened far enough or if a limit switch has been hit.
2. Keep doing something until something else kills the command. These commands return false; (*I’m never quitting on my own*). An example of this is driving using a joystick. This command should always respond to changes in the joystick so it needs to execute all the time. If the joystick is centered, it will set the motor speed to zero and as soon as it is not zero it will set the speed accordingly.
3. Do something and then wait until something else happens. These commands will run code that checks to see if what you are waiting for has happened yet or not. If it hasn’t happened yet, it returns false. When it does happen, it returns true and the command ends.

For example, let’s say you wanted to suck a crate into your robot and that you know that the crate is safely inside when it hits a limit switch. In *Initialize()*, set the intake motor to turn inwards. In *Execute()*, you don’t need to do anything because nothing is changing during the command. In *IsFinished()* you add something like

return intakeSub->isLimitSwitchHit(). This means that the command will end once the limit switch is hit.

1. **End()**

The method handles any cleanup that needs to be done before the command really quits. It is kind of the opposite of *Initialize()*. It only happens once after *IsFinished()* returns true.

With the crate example above you may have noticed that we never actually turned the intake motor off after we determined that we had hit the limit switch. That’s because you would do that here.

1. **Interrupted()**

This method is like a special case of *End()*. If another command that starts running requires one or more subsystem that the current command requires, this command will end immediately after this method is executed.

Often, all you do here is call *End()*. That would be the case with the crate-intake example above. If another command wants to use the intake subsystem, just stop the motor and let them take over.

On the other hand, if leaving a crate half-in the robot would be really bad, you could add code to expel or release the crate before the command quits.

Now that the command’s behavior is defined, it’s ready to be used.

## Running the command

There are multiple ways to run a command.

1. Triggered in response to something happening on the joysticks. This is the most common way that commands are triggered. See “Operator Interface – Buttons” for instructions on how to tie joystick actions to commands.
2. Called as part of a command group. A command group is just a command that calls any number of other commands. Command groups are covered later in this document.
3. Called explicitly by running the commands Start() method. This is how Autonomous mode works. You pick which command or command group to run and then call its Start() method.
4. Set as the default command for a subsystem. You can set a default command for every subsystem by adding SetDefaultCommand(new *name*Cmd()); to the subsystem’s InitDefaultCommand() method. When the subsystem isn’t being used by any other commands, it will run this command. For example, as a default, the drivetrain subsystem could update its drive motor speed based on the joystick values. That way, if you’re not doing anything special/automatic, you are manually driving the robot.

# Operator Interface – Buttons

Buttons can trigger commands in multiple ways. Here’s a list of the methods that you can use:

* WhenPressed: The command is scheduled once when the button is pressed
* WhenReleased: The command is scheduled when the button is released
* WhileHeld: The command is scheduled repeatedly while the button is pressed
* ToggleWhenPressed: Tap the button once to run the command and a second time to stop it
* CancelWhenPressed: Cancels a command that is currently executing

The steps to connect a command to a button are listed below. Note that this assumes that you’ve already created and configured a Joystick (see Operator Interface – Joysticks). For the examples, let’s call the joystick *operatorController*.

1. Open src/OI.h
2. Below the includes, add a constant that links a user-friendly name to a joystick button number

e.g. const int BALL\_INTAKE\_BTN = 2;

1. In the private section of the class, add a JoystickButton pointer

e.g. std:unique\_ptr<JoystickButton> ballIntakeBtn;

1. Open src/OI.cpp
2. At the top, add an include for the command

e.g. #include “Commands/BallIntakeCmd.h”

1. Go to the constructor (OI::OI()) and create the joystick button

e.g. ballIntakeBtn.reset(

new JoystickButton(OperatorController.get(), BALL\_INTAKE\_BTN));

1. Also, link the button to the command

e.g. ballIntakeBtn->WhenPressed(new BallIntakeCmd);

The button is now linked to the command. Anytime that button is pressed, that command will execute.

# Command Groups

Command groups combine any number of commands into one super-command. All they consist of is a constructor. In this constructor, you add a list of commands that you want to run as part of the group.

This is most often used in Autonomous mode where the robot has to execute complicated sets of maneuvers. But it can also be useful in Teleoperation to simplify sets of actions that are repetitive or prone to operator error.

For example, for the 15 second Autonomous the robot may run a command group that:

* Intakes the ball until the limit switch is hit
* Goes forward 2000mm
* Rotates right until the IMU measures a 45% turn
* Starts the shooting motor
* Adjusts the shooting angle based on what the camera sees
* Moves the ball into the shooter to take the shot

## Creating a new command

1. Right-click the project folder in Eclipse’s Project Explorer.
2. Choose *New 🡪 CommandGroup.*
3. For the Class Name, enter a descriptive name followed by Grp
   * e.g. AutoCrossRoughTerrainAndShootGrp
4. Click *Finish.*

## Sequential vs. parallel command execution

Each command in the group can be set to run sequentially or in parallel. When a command is added sequentially (AddSequential()), it will execute completely before the next command in the group starts. When a command is added in parallel (AddParallel()), it gets started but the next command in the group is also immediately started. If that next command was also added in parallel, then the third command is also started. And so on until the next sequential command is started or there are no more commands in the group.

## Adding commands to the group

1. Open src/Commands/*Name*Grp.cpp
2. In the constructor, add one or more commands
   1. AddSequential(new *Name*Cmd());
   2. AddParallell(new *Name*Cmd());

## Detailed Examples

### Example 1

As a simple example, let’s say the robot moves forward 2000mm and then turns 45 degrees. These commands need to run sequentially.

AddSequential(new MoveForwardCmd(2000));

AddSequential(new TurnCmd(45));

Now let’s look at mixing parallel and sequential with these commands:

1. ResetWheelEncodersCmd – requires drivetrainSub, a fraction of a second
2. OpenShooterGateCmd – requires shooterSub, takes 2 seconds
3. MoveForwardCmd(2000) – requires drivestrainSub, takes 4 seconds

#1 and #3 can’t run in parallel since they both use the drivetrain subsystem. So that leaves you with:

AddParallel(new OpenShooterGateCmd());

AddSequential(new ResetWheelEncodersCmd());

AddSequential(new MoveForwardCmd(2000));

The graph below shows what times each command is active.

### Example 2

You have to be careful when running commands in parallel because if a later command starts executing before it’s done and both need the same subsystem(s), the first command will get interrupted. For example, if we use these commands:

1. MoveForwardCmd(2000) – requires drivestrainSub, takes 4 seconds
2. OpenShooterGateCmd – requires shooterSub, takes 2 seconds
3. TurnCmd(45) – requires drivetrainSub, takes 1 second

added this way:

AddParallel(new MoveForwardCmd(2000));

AddSequential(new OpenShooterGateCmd());

AddSequential(new TurnCmd(45));

The code will not work as expected. What actually happens is that the turn command interrupts the move command as shown in the graph below.

The result is that the robot will not have moved far enough before it turns.

In this case, the correct order would be:

AddParallel(new OpenShooterGateCmd());

AddSequential(new MoveForwardCmd(2000));

AddSequential(new TurnCmd(45));

And it looks like this, graphically:

Now the movement is completed before the robot turns.

# Cheat Sheet

## New Subsystem

* Create a new Subsystem
* Class name is *Name*Sub (where *Name* is a descriptive name)
* In src/CommandBase.h
  + Add #include “Subsystem/*Name*Sub” near top
  + Add static std::unique\_ptr<*Name*Sub> *name*Sub; to public in CommandBase
* In src/CommandBase.cpp
  + Add std::unique\_ptr<*Name*Sub> CommandBase::*name*Sub = NULL; after includes
  + Add *name*Sub.reset(new *Name*Sub()); in CommandBase::init()
* In src/Subsystems/*Name*Sub.h’s
  + In the private section, add pointers for each of the subsystem’s actuators and sensors
    - std::unique\_ptr<*I/O-Class-Type*> *descriptiveName*;
  + In the public section, add methods for controlling the subsystem
* In src/Subsystems/*Name*Sub.cpp
  + Change Subsystem(“ExampleSubsystem”) to Subsystem(“*Name*Sub”)
  + In the constructor, initialize all of the actuators and sensors defined in the h file
    - *descriptiveName*.reset(new *I/O-Class-Type*(*parameters*));
  + Add the code for every method that was defined in the h file

## New Command

* Create a new Command.
* Class name is *Name*Cmd (where *Name* is a descriptive name)
* In Commands/*Name*Cmd.cpp
  + Add code to constructor, Initialize(), Execute(), IsFinished(), End() and/or Interrupted()

## Tying a Command to a Joystick Button

* In src/OI.h
  + Add a constant for the button number below the includes
    - const int *NAME*\_BTN = 2
  + In the class’ private section, add a pointer for the button
    - std:unique\_ptr<JoystickButton> *name*Btn;
* In src/OI.cpp
  + Add an include for the command
    - #include “Commands/*Name*Cmd.h”
  + In the constructor, create the button
    - *name*Btn.reset(new JoystickButton(*controller*, *NAME*\_BTN));
  + Also link the button to the command
    - *name*Btn->WhenPressed(new *Name*Cmd());

## New Command Group

* Create new CommandGroup.
* Class name is *Name*Grp (where *Name* is a descriptive name).
* In src/Commands/*Name*Grp.cpp, in the constructor
  + Add commands using
    - AddSequential(new NameCmd())
    - AddParallel(new NameCmd()))