ROBOTBUILDER

Last Updated: 10-03-2019

Table of Contents

The basic steps to create a robot program

Overview of RobotBuilder

Creating a program with RobotBuilder is a very straight forward procedure by following a few steps that are the same for any robot. This lesson describes the steps that you can follow. You can find more details about each of these steps in subsequent sections of the document.

See How to write an easy to test robot program for more information about the basic steps in creating a program to control your robot.

In addition to the text documentation provided here, a series of videos about Robot Builder are also available.

Divide the robot into subsystems

Divide the robot into subsystems

Your robot is naturally made up of a number of smaller systems like the drive trains, arms, shooters, collectors, manipulators, wrist joints, etc. You should look at the design of your robot and break it up into smaller, separately operated subsystems. In this particular example there is an elevator, a minibot alignment device, a gripper, and a camera system. In addition one might include the drive base. Each of these parts of the robot are separately controlled and make good candidates for subsystems.

For more information see: Defining Robot Subsystems

Add each subsystem to the RobotBuilder project

Add each subsystem to the RobotBuilder project

Each subsystem will be added to the "Subsystems" folder in the RobotBuilder and given a meaningful name. For each of the subsystems there are several attributes that get filled in to specify more information about the subsystems. In addition there are two types of subsystems that you might want to create:

1. PIDSubsystems - often it is desirable to control a subsystems operation with a PID controller. This is code in your program that makes the subsystem element, for example arm angle, more quickly to a desired position then stop when reaching it. PIDSubsystems have the PID Controller code built-in and are often more convenient then adding it yourself. PIDSubsystems have a sensor that determines when the device has reached the target position and an actuator (speed controller) that is driven to the setpoint.
2. Regular subsystem - these subsystems don't have an integrated PID controller and are used for subsystems without PID control for feedback or for subsystems requiring more complex control than can be handled with the default imbedded PID controller.

As you look through more of this documentation the differences between the subsystem types will become more apparent.

For more information see: Creating a subsystem, Writing Java code for a subsystem and Writing C++ code for a subsystem

Add components to each of the subsystems

Add components to each of the subsystems

Each subsystem consists of a number of actuators, sensors and controllers that it uses to perform its operations. These sensors and actuators are added to the subsystem with which they are associated. Each of the sensors and actuators comes from the RobotBuilder palette and is dragged to the appropriate subsystem. For each, there are usually other properties that must be set such as port numbers and other parameters specific to the component.

In this example there is an Elevator subsystem that uses a motor and a potentiometer (motor and pot) that have been dragged to the Elevator subsystem.

Add commands to describe the goals for each subsystem

Add commands to describe the goals for each subsystem

Commands are distinct goals that the robot will perform. These commands are added by dragging the command under the "Commands" folder. When creating a command, there are 3 primary choices (shown on the palette on the left of the picture):

- Normal commands - these are the most flexible command, you have to write all of the code to perform the desired actions necessary to accomplish the goal.
- Command groups - these commands are a combination of other commands running both in a sequential order and in parallel. Use these to build up more complicated actions after you have a number of basic commands implemented.
- Setpoint commands - setpoint commands move a PID Subsystem to a fixed setpoint, or the desired location.

For more information see: Creating a command, Writing the code for a command in Java and Writing the code for a command in C++

Test each command individually by starting it from the SmartDashboard

Test each command individually by starting it from the SmartDashboard

Each command can be run from the SmartDashboard. This is useful for testing commands before you add them to the operator interface or to a command group. As long as you leave the "Button on SmartDashboard" property checked, a button will be created on the SmartDashboard. When you press the start button, the command will run and you can check that it performs the desired action.

By creating buttons, each command can be tested individually. If all the commands work individually, you can be pretty sure that the robot will work as a whole.

For more information see: Adding a button to SmartDashboard to run a command

Add Operator Interface components

Add Operator Interface components

The operator interface consists of joysticks and devices connected through the extended I/O device (Cypress module). You can add operator interface components (joysticks, joystick buttons, and analog and digital inputs) to your program in RobotBuilder. It will automatically generate code that will initialize all of the components and allow them to be connected to commands.

The operator interface components are dragged from the palette to the "Operator Interface" folder in the RobotBuilder program. First (1) add Joysticks to the program then put buttons under the associated joysticks (2) and give them meaningful names, like ShootButton.

Connect the commands to the Operator Interface

Connect the commands to the Operator Interface

Commands can be associated with buttons so that when a button is pressed the command is scheduled. This should, for the most part, handle most of the tele-operated part of your robot program.

This is simply done by (1) adding the command to the JoystickButton object in the RobotBuilder program, then (2) setting the condition in which the command is scheduled.

For more information see: Connecting the operator interface to a command

Develop one or more Autonomous commands

Develop one or more Autonomous commands

Commands make it simple to develop autonomous programs. You simply specify which command should run when the robot enters the autonomous period and it will automatically be scheduled. If you have tested commands as discussed above, this should simply be a matter of choosing which command should run.

Select the robot at the root of the RobotBuilder project, then edit the Autonomous Command property to choose the command to run. It's that simple!

For more information see: Setting the default autonomous command

Generating code for the program

Generating code for the program

At any point in the process outlined above you can have RobotBuilder generate a C++ or Java program that will represent the project you have created. This is done by specifying the location of the project in the project properties (1), then clicking the appropriate toolbar button to generate the code.

For more information see: Generating Netbeans project files and Generating C++ code for a project.

Starting RobotBuilder

RobotBuilder is a Java program and as such should be able to run on any platform that is supported by Java. We have been running RobotBuilder on Mac OS X, Windows 7, and various versions of Linux successfully.

Locating the RobotBuilder .jar file

Locating the RobotBuilder .jar file

RobotBuilder is shipped as a .jar file (Java archive). In most cases you can simply double-click on the file in a graphical file browser for your operating system and it will start.

- For java users RobotBuilder is located in the sunspotfrcsdk/tools directory. This directory is in your user home directory, usually something like /Users/<username>/sunspotfrcsdk.
- For C++ users RobotBuilder is located in the C;\WindRiver\WPILib folder (shown in the example above).

Starting RobotBuilder from the command line

Starting RobotBuilder from the command line

In some cases Java and your file browser might not be properly configured to run the .jar file by double-clicking. Simply type "java -jar RobotBuilder.jar" from the directory that contains RobotBuilder. Be sure to append the correct version number to the RobotBuilder name as shown in the example above.

The RobotBuilder user interface

RobotBuilder User Interface

RobotBuilder User Interface

RobotBuilder has a user interface designed for rapid development of robot programs. Almost all operations are performed by drag and drop or selecting options from drop-down lists.

Dragging items from the palette to the robot description

Dragging items from the palette to the robot description

You can drag items from the palette to the robot description by starting the drag on the palette item and ending on the container where you would like the item to be located. In this example, dropping a potentiometer to the Elevator subsystem.

Adding components using the right-click context menu

Adding components using the right-click context menu

A shortcut method of adding items to the robot description is to right-click on the container object (Elevator) and select the item that should be added (Potentiometer). This is identical to using drag and drop but might be easier for some people.

Editing properties of robot desciption items

Editing properties of robot desciption items

The properties for a selected item will appear in the properties viewer. The properties can be edited by selecting the value in the right hand column.

Using the menu system

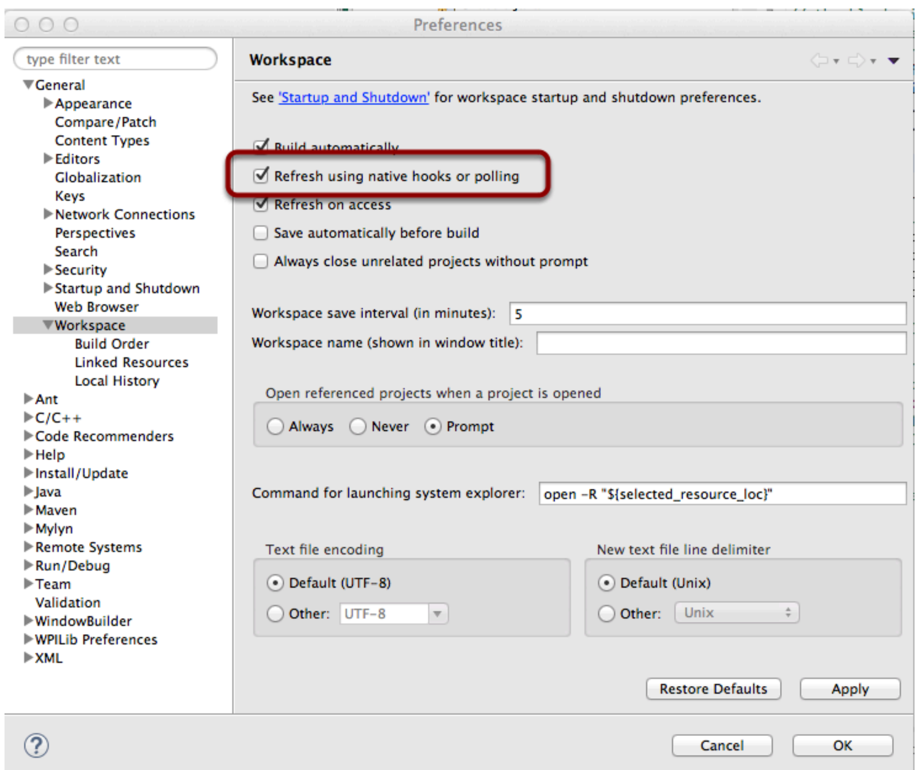
Using the menu system

Operations for RobotBuilder can either be selected through the menu system or the equivalent item (if it is available) from the toolbar.

Setting up the robot project

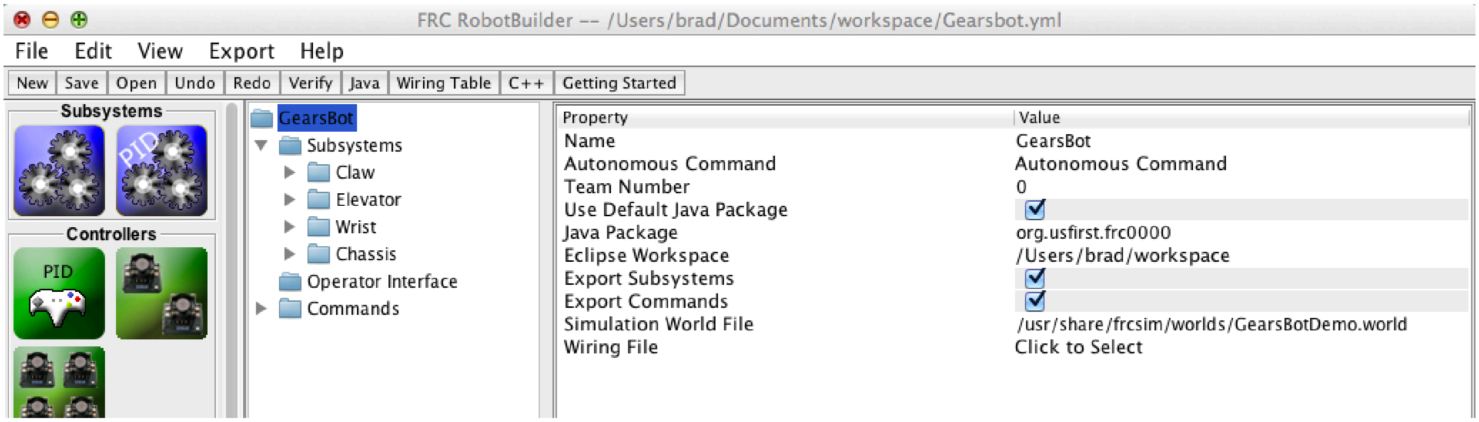
The RobotBuilder program has some default properties that need to be set up so the generated program and other generated files work properly. This setup information is stored in the properties for robot description (the first line).

Using RobotBuilder with Eclipse



When using RobotBuilder files are saved inside the RobotBuilder interface and also from the Eclipse interface. It is important to keep the files in sync. To get eclipse to automatically notice that RobotBuilder has saved new versions of the project files and automatically load them check the "Refresh using native hooks or polling" option in the Workspace preferences under General as shown here. With this option checked, the source files open in the Eclipse editor will automatically be refreshed when RobotBuilder generates new files.

Robot project properties



The properties that describe the robot are:

Name - The name of the robot project that is created

Autnomous Command - the command that will run by default when the program is placed in autonomous mode

Team Number - the team number is used for creating the package names

Java Project - The folder that the java project is generated into when Export to Java is selected

Use Default Java Package - If checked RobotBuilder will use the default package (org.usfirst.frc####). Otherwise you can specify a custom package name to be used.

Java Package - The name of the generated Java package used when generating the project code

Eclipse Workspace - The location of the Eclipse workspace that your project should be saved to

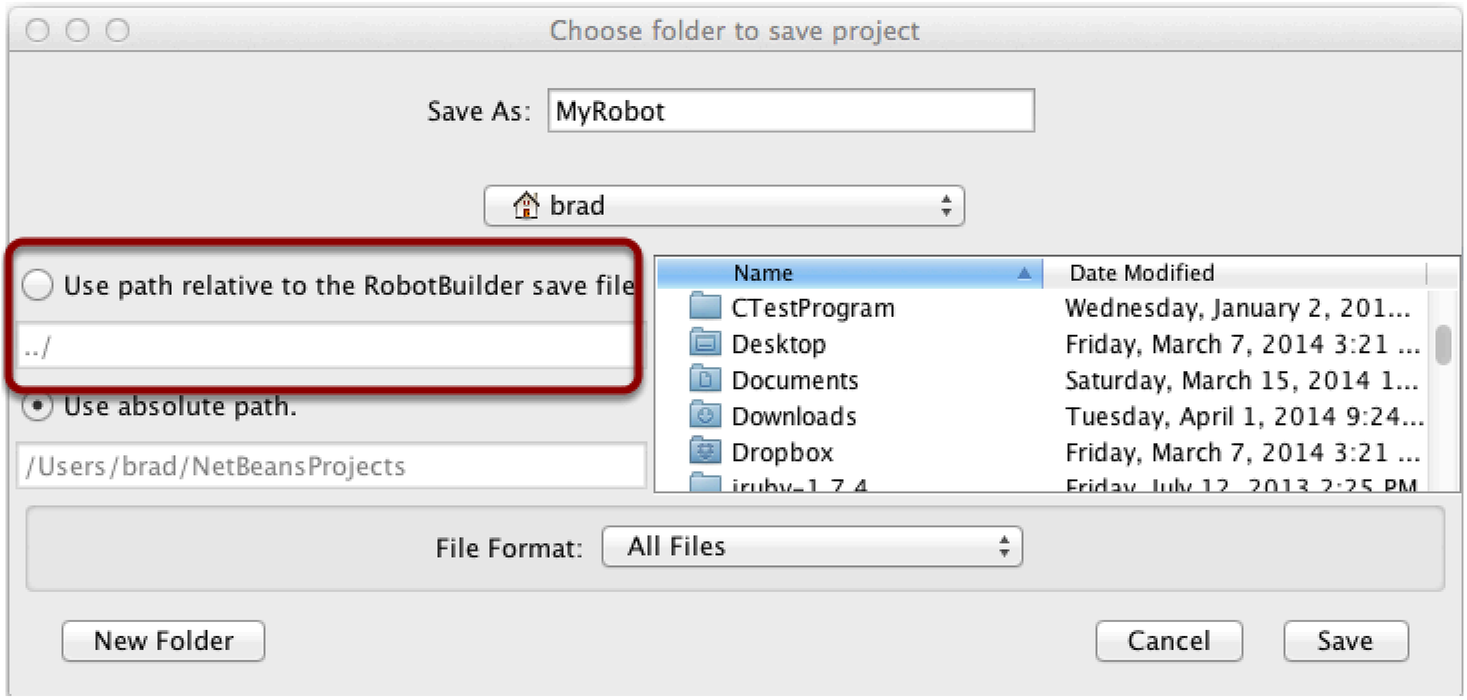
Export Subsystems - Checked if RobotBuilder should export the Subsystem classes from your project

Export Commands - Checked if RobotBuilder should export the Command classes from your project

Simulation World File - The World File that is used for simulation of your robot project

Wiring File - the location of the html file that contains the wiring diagram for your robot

Using source control with the RobotBuilder project



When using source control the project will typically be used on a number of computers and the path to the project directory might be different from one users computer to another. If the RobotBuilder project file is stored using an absolute path, it will typically contain the user name and won't be usable across multiple computers. To make this work, select "relative path" and specify the path as an directory offset from the project files. In the above example, the project file is stored in the folder just above the project files in the file hierarchy. In this case, the user name is not part of the path and it will be portable across all of your computers.

Creating a subsystem

Subsystems are classes that encapsulate (or contain) all the data and code that make a subsystem on your robot operate. The first step in creating a robot program with the RobotBuilder is to identify and create all the subsystems on the robot. Examples of subsystems are grippers, ball collectors, the drive base, elevators, arms, etc. Each subsystem contains all the sensors and actuators that are used to make it work. For example, an elevator might have a Jaguar speed controller and a potentiometer to provide feedback of the robot position.

Creating a subsystem by dragging from the palette

Creating a subsystem by dragging from the palette

Drag the subsystem icon from the palette to the Subsystems folder in the robot description to create a subsystem class.

Creating a subsystem by using the context menu on the Subsystem folder

Creating a subsystem by using the context menu on the Subsystem folder

Right-click on the Subsystem folder in the robot description to add a subsystem to that folder.

Name the subsystem

Name the subsystem

After creating the subsystem by either dragging or using the context menu as described above, simply type the name you would like to give the subsystem. The name can be multiple words separated by spaces, RobotBuilder will concatenate the words to make a proper Java or C++ class name for you.

Drag actuators and sensors into the subsystem

Drag actuators and sensors into the subsystem

There are two steps to adding components to a subsystem:

1. Drag actuators or sensors from the palette into the subsystem as required.
2. Give the actuator or sensor a meaningful name
3. Edit the properties such as module numbers and channel numbers for each item in the subsystem.

RobotBuilder will automatically use incrementing channel numbers for each module on the robot. If you haven't yet wired the robot you can just let RobotBuilder assign unique channel numbers for each sensor or actuator and wire the robot according to the generating wiring table.

This just creates the subsystem in RobotBuilder, and will subsequently generate skeleton code for the subsystem. To make it actually operate your robot please refer to: Writing the code for a subsystem in Java or Writing the code for a subsystem in C++.

Creating a command

Commands are classes you create that provide behaviors or actions for your subsystems. The subsystem class should set the operation of the subsystem, like ElevatorUp to start the elevator moving up, or ElevatorToSetPoint to set the elevator's PID setpoint. The commands initiate the subystem operation and keep track of when it is finished.

Drag a command to the robot description Commands folder

Drag a command to the robot description Commands folder

Simple commands can be dragged from the palette to the robot description. The command will be created under the Commands folder.

Creating commands using the context menu

Creating commands using the context menu

You can also create commands using the right-click context menu on the Command folder in the robot description.

Give the command a name and set the required subsystem

Give the command a name and set the required subsystem

Name the command with something meaningful that describes what the command will do. Then set the subsystem that is used by this command. When this command is scheduled, it will automatically stop any command currently running that also requires this command. If a command to open the gripper is currently running (requiring the gripper subsystem) and the close gripper command is scheduled, it will immediately stop opening and start closing.

Setting the default command for a subsystem

Once you have some commands created, you can set one of them to be the default command for a subsystem. Default commands run automatically when nothing else is running that requires that subsystem. A good example is having a drive train subsystem with a default command that reads joysticks. That way, whenever the robot program isn't running other commands to operate the drive train under program control, it will operate with joysticks.

Create the command that should be the default for a subsystem

Create the command that should be the default for a subsystem

Here a command is created called "Drive with joysticks" that would read the joystick values and set them in the Drive Train subsystem. This is what the Drive Train should be doing if it isn't being asked to do anything else.

Set the command as the default for the subsystem

Set the command as the default for the subsystem

The "Drive with joysticks" command is set as the default command for the Drive Train subsystem.

You can also set the default Autonomous command, that is the command that runs when the robot enters the Autonomous state.

Setting the default autonomous command

Since a command is simply one or more actions (behaviors) that the robot performs, it makes sense to describe the autonomous operation of a robot as a command. While it could be a single command, it is more likely going to be a command group (a group of commands that happen together).

To designate a command that runs when the robot starts during the autonomous period of a competition:

1. Select the robot in the robot program description
2. Fill in the Autonomous command field with the command that should run when the robot is placed in autonomous mode. This is a drop-down field and will give you the option to select any command that has been defined.

When the robot is put into autonomous mode, the defined Autonomous command will be scheduled.

See: Creating a command and Setting the default command for a subsystem.

Adding a button to SmartDashboard to test a command

Commands are easily tested by adding a button to the SmartDashboard to trigger the command. In this way, no integration with the rest of the robot program is necessary and commands can easily be independently tested. This is the easiest way to verify commands since with a single line of code in your program, a button can be created on the SmartDashboard that will run the command. These buttons can then be left in place to verify subsystems and command operations in the future.

This has the added benefit of accommodating multiple programmers, each writing commands. As the code is checked into the main robot project, the commands can be individually tested.

Creating the button on the SmartDashboard

Creating the button on the SmartDashboard

The button is created on the SmartDashboard by putting an instance of the command from the robot program to the dashboard. This is such a common operation that it has been added to RobotBuilder as a checkbox. When writing your commands, be sure that the box is checked, and buttons will be automatically generated for you.

How to operate the buttons on the SmartDashboard

How to operate the buttons on the SmartDashboard

The buttons will be generated automatically and will appear on the dashboard screen. You can put the SmartDashboard into edit mode, and the buttons can then be rearranged along with other values that are being generated. In this example there are a number of commands, each with an associated button for testing. The button is labeled "Start" and pressing it will run the command.

As soon as it is pressed, the label changes to "Cancel" and pressing it will interrupt the command causing the Interrupted() method to be called.

Adding commands manually

Adding commands manually

Commands can be added to the SmartDashboard manually by writing the code yourself. This is done by passing instances of the command to the PutData method along with the name that should be associated with the button on the SmartDashboard. These instances are scheduled whenever the button is pressed. The result is exactly the same as RobotBuilder generated code, although clicking the checkbox in RobotBuilder is much easier than writing all the code by hand.

Connecting the operator interface to a command

Commands handle the behaviors for your robot. The command starts a subsystem to some operating mode like raising and elevator and continues running until it reaches some setpoint or timeout. The command then handles waiting for the subsystem to finish. That way commands can run in sequence to develop more complex behaviors.

RobotBuilder will also generate code to schedule a command to run whenever a button on your operator interface is pressed. You can also write code to run a command when a particular trigger condition has happened.

Set up a command to be run by the button press

Set up a command to be run by the button press

In this example we want to schedule the "Elevator to top" command to run whenever joystick button 1 is pressed.

1. The command to run is called "Elevator to top" and its function is to move the elevator on the robot to the top position
2. Notice that the command requires the Elevator subsystem. This will ensure that this command starts running even if there was another operation happening at the same time that used the elevator. In this case the previous command would be interrupted.

Adding the Joystick to the robot program

Adding the Joystick to the robot program

Add the joystick to the robot program

1. Drag the joystick to the Operator Interface folder in the robot program

2. Name the joystick so that it reflects the use of the joystick and set the USB port number

Add a button and link it to the Elevator to top command

Add a button and link it to the Elevator to top command

Add the button that should be pressed to the program

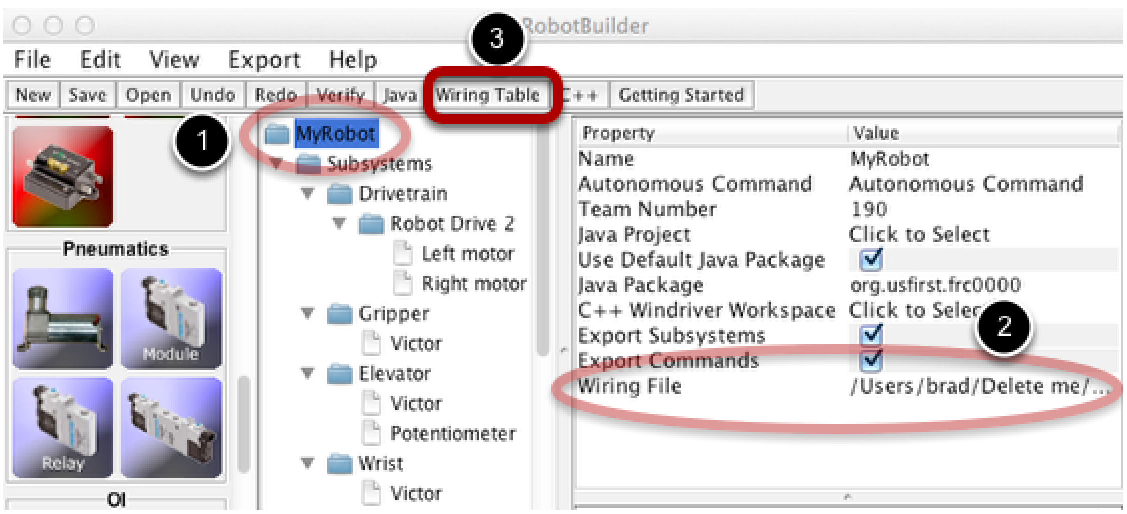
1. Drag the joystick button to the Joystick (Driver gamepad) so that it's under the joystick
2. Set the properties for the button: the button number, the command to run when the button is pressed, and the "When to run" property to "whenPressed" to indicate that the command should run whenever the joystick button is pressed.

Note: **Joystick buttons must be dragged to (under) a Joystick**. You must have a joystick in the Operator Interface folder before adding buttons.

Producing a wiring diagram for your robot

Once all the subsystems are defined and filled with sensors and actuators you can produce a wiring diagram that will help the team wiring the robot to make sure that the electrical connections between the components and the cRIO robot controller are correct.

Filling in the wiring file location

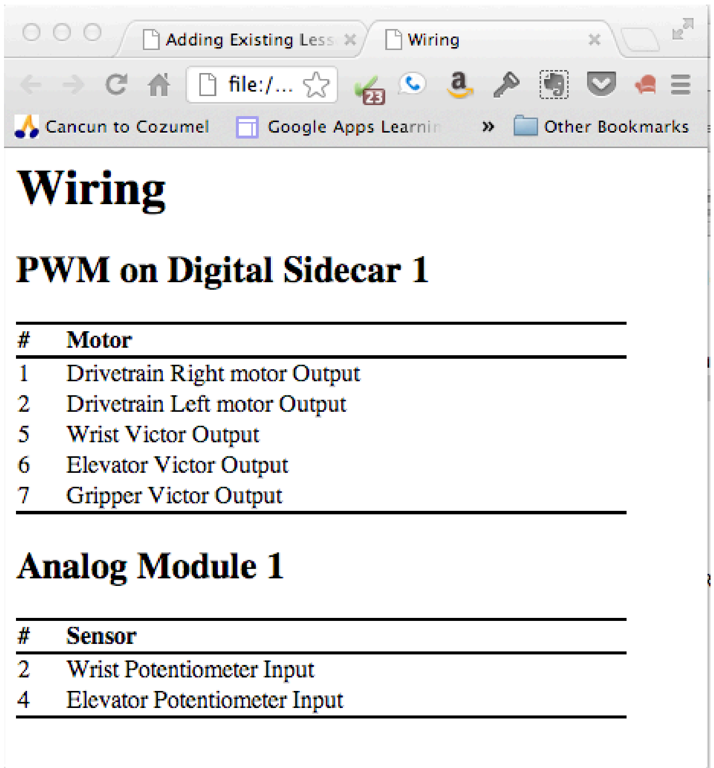


There are three steps to getting a wiring diagram for your robot.

1. Select the top of the robot description to view the robot program properties
2. Click on the wiring file name (initially not filled in) to set where the wiring file should be generated
3. Click on the Wiring Table toolbar item to create the wiring diagram

It is only necessary to do steps 1 and 2 the first time, the RobotBuilder will remember the location of the wiring file once it is set for the project.

View the wiring file your web browser



The wiring file may be opened with the system web browser and printed to hand off to the robot wiring team. You should keep the final version of the wiring diagram with the robot in case connections are pulled out during a competition so that it can be quickly rewired.

RobotBuilder generated code

Main robot program

Main robot program

This is the main program generated by RobotBuilder. There are a number of parts to this program:

1. This class extends IterativeRobot. IterativeRobot will call your autonomousPeriodic() and teleopPeriodic() methods every 20ms (each time the driver station exchanges Joystick and other data with the robot).
2. Each of the subsystems is declared here These are public static variables so that they can be referenced from throughout your robot program by writing Robot.<*subsystem-name*>.method(), for example Robot.elevator.setSetpoint(4).
3. The subsystems are instantiated in the robotInit() method that is called after the construtor runs for this class. It is important to be create the subsystems after the constructor to avoid recursive loops. Also instance of the OI() class (for your operator interface) and the autonomous command are created here.
4. In the autonomousInit() method which is called every 20ms, make one scheduling pass. That is call the isFinished() and execute() methods of every command that is currently scheduled.
5. In the teleopPeriodic method which is called every 20ms, make one scheduling pass.

Writing C++ code for your robot

Generating C++ code for a project

Adding code to create an actual working subsystem is very straightforward. For simple subsystems that don't use feedback it turns out to be extremely simple. In this section we will look at an example of a Claw subsystem that operates the motor for some amount of time to open or close a claw on the robot arm.

Generate the code for the project

Generate the code for the project

Verity that the C++ WindRiver workspace location is set properly (1) and generate code for the C++ robot project (2).

Import the project into WindRiver Workbench

Import the project into WindRiver Workbench

Right-click in the Project Explorer and import your project from the location set in RobotBuilder. Ideally the project has been saved in your workspace.

Viewing the imported project

Viewing the imported project

You can view the project in the project explorer by double-clicking on the project name in the project explorer. From there you can see all the project files. Your subsystems are in the Subsystems folder and the commands are in the Commands folder.

Writing the C++ code for a subsystem

Writing the code for a command in C++

Writing the code for a PIDSubsystem in C++

PIDSubsystems use feedback to control the actuator and drive it to a particular position. In this example we use an elevator with a 10-turn potentiometer connected to it to give feedback on the height. The skeleton of the PIDSubsystem is generated by the RobotBuilder and we have to fill in the rest of the code to provide the potentiometer value and drive the motor with the output of the imbedded PIDController.

Setting the PID constants

Setting the PID constants

Make sure the Elevator PID subsystem has been created in the RobotBuiler. In the case of our elevator we use a propertional constant of 6.0 and 0 for the I and D terms. Once it's all set, generate C++ code for the project using the Export menu or the C++ toolbar menu.

Add constants for the Elevator preset positions

Add constants for the Elevator preset positions

Elevator constants define potentiometer voltages that correspond to fixed positions on the elevator. These values can be determined using the print statements, the LiveWindow or SmartDashboard.

Initialize the elevator position in the Elevator constructor

Initialize the elevator position in the Elevator constructor

Set the elevator initial position so when the robot starts up it will move to that position. This will get the robot to a known starting point. Then enable the PIDController that is part of the PIDSubsystem. The elevator won't actually move until the robot itself is enabled because the motor outputs are initially off, but when the robot is enabled, the PID controller will already be running and the elevator will move to the "STOW" starting position.

Set the ReturnPIDInput method to return voltage

Set the ReturnPIDInput method to return voltage

By default the RobotBuilder generated ReturnPIDInput() method returns the potentiometer value in raw units (a value between 0-1023). Since the setpoints are all in voltages (0.0 - 5.0V) the ReturnPIDInput() method must be changed to return volts to match.

That's all that is required to create the Elevator PIDSubsystem in C++. To operate it with commands to actually control the motion see: Operating a PIDSubsystem from a command in C++.

Writing Java code for your robot

Generating Netbeans project files

After you start getting a significant part of your robot designed in RobotBuilder you can generate a Java project for use with Netbeans. The code that is generated includes project files that will let you just open the project and start adding your robot specific code. In addition, if you later make changes in RobotBuilder, you can regenerate the project again and it will not overwrite your changes. This process is described in detail below.

Setting up the project properties for export

Setting up the project properties for export

Here is the procedure for setting up the project for Java code generation (export).

1. Select the project name in the top of the robot description to see the project properties.
2. Set the project name to something meaningful for your teams robot.
3. Set the directory where the project should be saved. This might be inside your NetbeansProjects directory or some other folder.

Generate the project files

Generate the project files

Once the location of the exported project files is defined (previous step) either click on Java from the Export menu or use the "Java" item in the toolbar to generate code to the correct location. This will generate a full project the first time the button is pressed, or it will update the project with changes on subsequent exports.

Open the project in NetBeans

Open the project in NetBeans

In NetBeans, select "File" from the menu bar, then "Open project..." and select the location where the file was saved from RobotBuilder. The project will be opened and you will see it in the "Projects" tab on the left side of the NetBeans window. **The project name will be the same as the name of the top folder in RobotBuilder.**

Writing the code for a subsystem in Java

Adding code to create an actual working subsystem is very straightforward. For simple subsystems that don't use feedback it turns out to be extremely simple. In this section we will look at an example of a Claw subsystem that operates the motor for some amount of time to open or close a claw on the robot arm.

Create the subsystem

Create the subsystem

Be sure that the subsystem is defined in the RobotBuilder robot description. The Claw subsystem has a single Victor and no sensors since the claw motor operates for one second in either direction to open or close the claw.

Generate code for the project

Generate code for the project

Verify that the java project location is set up (1) and generate code for the robot project (2).

Open the project in Netbeans

Open the project in Netbeans

Open the generated project in Netbeans and notice the subystems package containing each of the subsystem files. Open the Claw.java file to add code that will open and close the claw.

Add methods to open, close, and stop the claw

Add methods to open, close, and stop the claw

Add methods to the claw.java that will open, close, and stop the claw from moving. Those will be used by commands that actually operate the claw. The comments have been removed from this file to make it easier to see the changes for this document. Notice that a member variable called "victor" is created by RobotBuilder so it can be used throughout the subsystem. **Each of your dragged-in palette items will have a member variable with the name given in RobotBuilder**.

See: Writing the code for a command in Java to see how to get this Claw subsystem to operate using commands. See: Writing the code for a PIDSubystem in Java to write the code for a more complex subsystem with feedback (PIDSubsystem).

Writing the code for a simple command in Java

Subsystem classes get the mechanisms on your robot moving, but to get it to stop at the right time and sequence through more complex operations you write Commands. Previously in Writing the code for a subsystem in Java we developed the code for the Claw subsystem on a robot to start the claw opening, closing, or to stop moving. Now we will write the code for a command that will actually run the Claw motor for the right time to get the claw to open and close. Our claw example is a very simple mechanism where we run the motor for 1 second to open it or 0.9 seconds to close it.

Adding a command

Adding a command

To create a command drag the command icon from the palette to the Commands folder in the robot description. This will create a command with a default name, then rename the command to be something meaningful, in this case "Open claw" or "Close claw".

Setting the Requires property to the correct subsystem

Setting the Requires property to the correct subsystem

Set the Requires propert to the subsystem that this command is controlling. In this case, the "Close claw" command controls the Claw subsystem. If the "Close claw" command is scheduled while another command that uses the Claw is also running, the "Close claw" command will preempt the other command and start. For example, if the "Open claw" was running, then the robot operator decided that they really wanted to close it, since they both require the Claw, the second command (Close claw) would cancel the running open command.

Generate code for the project

Generate code for the project

Using either the Export menu (1) or the Java toolbar item (2), generate the code for the project.

Write the code to close the claw

Write the code to close the claw

Add these 5 lines of code to the Closeclaw command to it can be used:

1. The claw needs to run in the close direction for 0.9 seconds to completely get closed. Setting a timeout initializes the timer for this command. Each command can have a single timer that can be used for timing operations or timeouts to make sure that commands don't get stuck in the case of a broken sensor. Then start the claw closing. Notice that we only need to start the claw closing once, so having it in the initialize method is sufficient.
2. The claw should continue closing until the timer runs out. The command has an isTimedOut() method that returns true if the timer that was set in the initialize() method is done.
3. In the end() method stop the claw from moving. This is called when the isFinished() method returns true (the timer runs out in this case).
4. The interrupted() method is called when this command is preempted by another command using the Claw subsystem. In this case, we just call the end() method to stop the claw from moving.

That's all that's required to get the claw to run when the command is run. Notice that **you can refer to any subsystem by using the class name Robot** since subsystem references are automatically generated as static variables. For example, "Robot.claw.closeClaw()" gives you access to the claw class and it's methods.

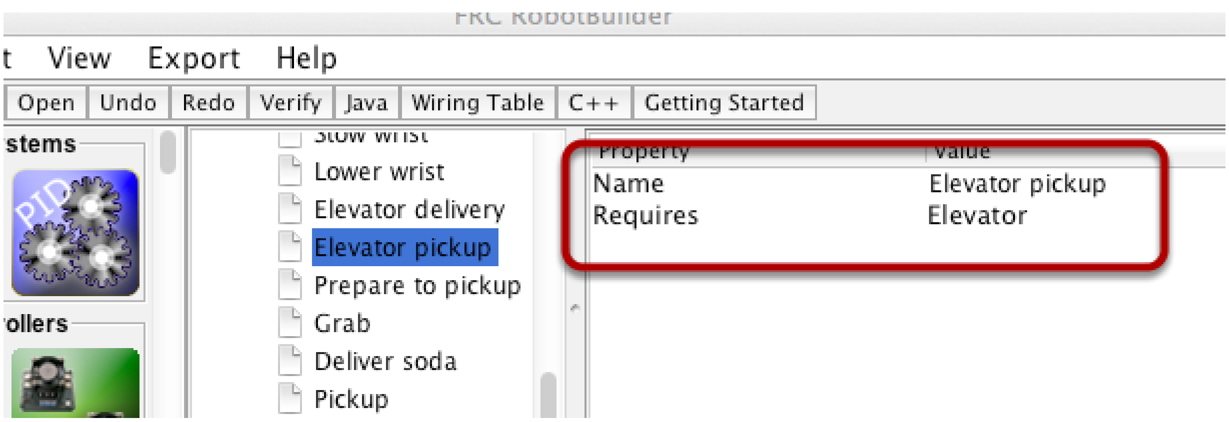
This command can be part of a more complex Command Group (see: Creating a command that runs other commands) or run from an operator interface button such as a joystick button (see: Connecting the operator interface to a command).

Writing the code for a PIDSubystem in Java

Operating a PIDSubsystem from a command in Java

A PIDSubsystem will automatically control the operation of an actuator with sensor feedback. To actually set the setpoints for the subsystem use a command since commands can be controlled over time and put together to make more complex commands. In this example we move the Elevator subsystem to the pickup (BOTTOM) position. To create the PIDSubsystem for the elevator see: Making a subsystem with feedback from sensors and Writing the code for a PIDSubystem in Java

Create the Elevator pickup command



The elevator pickup command moves the elevator to the pickup (BOTTOM) position. Notice that the Command requires the Elevator subsystem. By requiring the elevator, the command scheduler will automatically stop any "in progress" elevator commands when the Elevator pickup command is scheduled.

Export to Java to generate code for the robot program including the new Elevator pickup command.

Add the methods to finish the command

```java
 *
 */
public class  Elevatorpickup extends Command {

    public Elevatorpickup() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
    }

    protected void initialize() {
        Robot.elevator.setSetpoint(Elevator.BOTTOM);
    }

    protected void execute() {
    }

    protected boolean isFinished() {
        return Math.abs(Robot.elevator.getSetpoint() - Robot.elevator.getPosition()) < 0.1;
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```

There are two changes that need to be made to make the Command work properly:

1. Set the setpoint on the Subsystem PID controller so that it starts the elevator moving to the right position.
2. Add code to the isFinished() method so the command can finish when the elevator has moved to its target position. This way, other commands that run after this command will start when the elevator has reached its target position.

Advanced techniques

Creating a command that runs other commands

Often you will want to run multiple commands, one after another to enable more complex behaviors in your program. Once each of the individual commands have been debugged, you can create a CommandGroup. A CommandGroup is a named set of commands that may be executed sequentially or in parallel.

Creating a Command Group

Creating a Command Group

To create a CommandGroup

1. Drag the command group from the palette to Commands folder in the robot description
2. Name the command group so that is has a meaningful name

Edit the the generated code in the command group (Java)

Edit the the generated code in the command group (Java)

Add each command to command group that should be sequentially scheduled when the command group is scheduled. This allows you to build up complex commands based on simpler and tested commands. For each command that should run, call the addSequential() method with a reference to the instance of the command.

Edit the generated code in the command group (C++)

Edit the generated code in the command group (C++)

Add each command to command group that should be sequentially scheduled when the command group is scheduled. This allows you to build up complex commands based on simpler and tested commands. For each command that should run, call the addSequential() method with a reference to the instance of the command.

Using PIDSubsystems to control actuators with feedback from sensors

More advanced subsystems will use sensors for feedback to get guaranteed results for operations like setting elevator heights or wrist angles. The PIDSubsystem has a built-in PIDController to automatically set the correct setpoints for these types of mechanisms.

Create a PIDSubsystem

Create a PIDSubsystem

Creating a subsystem that uses feedback to control the position or speed of a mechanism is very easy.

1. Drag a PIDSubsystem from the palette to the Subsystems folder in the robot description
2. Rename the PID Subsystem to a more meaningful name for the subsystem

Notice that some of the parts of the robot description have turned red. This indicates that these components (the PIDSubsystem) haven't been completed and need to be filled in. The properties that are either missing or incorrect are shown in red.

Adding sensors and actuators to the PID Subsystem

Adding sensors and actuators to the PID Subsystem

Add the missing components for the PIDSubsystem

1. Drag in the actuator (a speed controller) to the particular subsystem - in this case the Elevator
2. Drag the sensor that will be used for feedback to the subsystem, in this case the sensor is a potentiometer that might give elevator height feedback.

Fill in the PIDSubsystem parameters to get the correct operation of the mechanism

Fill in the PIDSubsystem parameters to get the correct operation of the mechanism

There a number of parameters for the PIDSubsystem but only a few need to be filled in for most cases

1. The Input and Output compents will have been filled in automatically from the previous step when the actuator and sensor were dragged into the PIDSubsystem
2. The P, I, and D values need to be filled in to get the desired sensitivity and stability of the component

See: Writing the code for a PIDSubystem in Java and Writing the code for a PIDSubsystem in C++

Setpoint command

A common use case in robot programs is to drive an actuator to a particular angle or position that is measured using a potentiometer or encoder. This happens so often that there is a shortcut in RobotBuilder to do this task. It is called the Setpoint command and it's one of the choices on the palette or the right-click context menu that can be inserted under "Commands".

Start with a PIDSubsystem

Start with a PIDSubsystem

Suppose in a robot there is a wrist joint with a potentiometer that measures the angle. First create a PIDSubsystem that include the motor that moves the wrist joint and the potentiometer that measures the angle. The PIDSubsystem should have all the PID constants filled in and working properly.

It is important to set the Tolerance parameter. This controls how far off the current value can be from the setpoint and be considered on target. This is the criteria that the SetpointCommand uses to move onto the next command.

Creating the Setpoint Command

Creating the Setpoint Command

Right-click on the Commands folder in the palette and select "Add Setpoint command".

Setpoint Command parameters

Setpoint Command parameters

Fill in the name of the new command. The Requires field is the PIDSubsystem that is being driven to a setpoint and the Setpoint parameter is the setpoint value for the PIDSubsystem. There is no need to fill in any code for this command, it is automatically created by RobotBuilder.

Whenever this command is scheduled, it will automatically drive the subsystem to the specified setpoint. When the setpoint is reached within the tolerance specified in the PIDSubsystem, the command ends and the next command starts. It is important to specify a tolerance in the PIDSubsystem or this command might never end because the tolerance is not achieved.

Driving the robot with tank drive and joysticks

A common use case is to have a joystick that should drive some actuators that are part of a subsystem. The problem is that the joystick is created in the OI class and the motors to be controlled are in the subsystem. The idea is to create a command that, when scheduled, reads input from the joystick and calls a method that is created on the subsystem that drives the motors.

In this example a drive base subsystem is shown that is operated in tank drive using a pair of joysticks.

Create a Drive Train subsystem

Create a Drive Train subsystem

Create a subsystem called Drive Train. Its responsibility will be to handle the driving for the robot base. Inside the Drive Train is a Robot Drive object for a two motor drive robot (in this case). There is a left motor and right motor as part of the Robot Drive 2 class.

Add the joysticks to the Operator Interface

Add the joysticks to the Operator Interface

Add two joysticks to the Operator Interface, one is the left stick and the other is the right stick. The y-axis on the two joysticks are used to drive the robots left and right sides.

Note: be sure to export your program to C++ or Java before continuing to the next step.

Create a method to write the motors on the subsystem

Create a method to write the motors on the subsystem

Create a method that takes the joystick inputs, in this case the the left and right driver joystick. The values are passed to the RobotDrive object that in turn does tank steering using the joystick values. Also create a method called stop() that stops the robot from driving, this might come in handy later.

Note: the extra RobotBuilder comments have been removed to format the example for the documentation.

Create a command that reads the joystick values and calls the subsystem method

Create a command that reads the joystick values and calls the subsystem method

Create a command, in this case called DriveWithJoysticks. Its purpose will be to read the joystick values and send them to the Drive Base subsystem. Notice that this command Requires the Drive Train subsystem. This will cause it to stop running whenever anything else tries to use the Drive Train.

Note: be sure to export your program to C++ or Java before continuing to the next step.

Add the code for the command to do the actual driving

Add the code for the command to do the actual driving

Add code to the execute method to do the actual driving. All that is needed is to get the Joystick objects for the left and right drive joysticks and pass them to the Drive Train subsystem. The subsystem just uses them for the tank steering method on its RobotDrive object. And we get tank steering.

We also filled in the end() and interrupted methods so that when this command is interrupted or stopped, the motors will be stopped as a safety precaution.

Make the command the "default command" for the subsystem

Make the command the "default command" for the subsystem

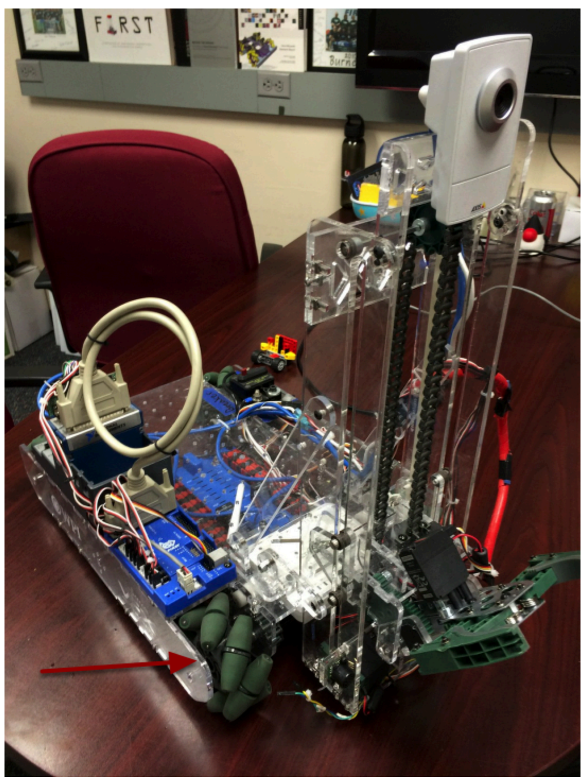
The last step is to make the DriveWIthJoysticks command be the "Default Command" for the Drive Train subsystem. This means that whenever no other command is using the Drive Train, the Joysticks will be in control. This is probably the desirable behavior. When the autonomous code is running, it will also require the drive train and interrupt the "DriveWithJoystick" command. When the autonomous code is finished, the DriveWithJoysticks command will restart automatically (because it is the default command), and the operators will be back in control. If you write any code that does teleop automatic driving, those commands should also "require" the DriveTrain so that they too will interrupt the DriveWithJoysticks command and have full control.

Note: be sure to export your program to C++ or Java before continuing.

Driving a robot using Mecanum drive

Mecanum drive is a method of driving using specially designed wheels that allow the robot to drive in any direction without changing the orientation of the robot. A robot with a conventional drivetrain (4 or six wheels) must turn in the direction it needs to drive. A mecanum robot can move in any direction without first turning and is called a holonomic drive.

Mecanum wheels



The wheels shown in this robot have rollers that cause the forces from driving to be applied at a 45 degree angle rather than straight forward as in the case of a conventional drive. You might guess that varying the speed of the wheels results in travel in any direction. You can look up how mecanum wheels work on various web sites on the internet.

Code for driving with mecanum wheels

```
#include "WPILib.h"
/**
 * Simplest program to drive a robot with mecanum drive using a single Logitech
 * Extreme 3D Pro joystick and 4 drive motors connected as follows:
 *   - Digital Sidecar 1:
 *     - PWM 1 - Connected to front left drive motor
 *     - PWM 2 - Connected to rear left drive motor
 *     - PWM 3 - Connected to front right drive motor
 *     - PWM 4 - Connected to rear right drive motor
 */
class MecanumDefaultCode : public IterativeRobot
{
        RobotDrive *m_robotDrive;                // RobotDrive object using PWM 1-4 for
drive motors
        Joystick *m_driveStick;                        // Joystick object on USB port 1
(mecanum drive)
public:
        /**
         * Constructor for this "MecanumDefaultCode" Class.
         */
        MecanumDefaultCode(void)
        {
                // Create a RobotDrive object using PWMS 1, 2, 3, and 4
                m_robotDrive = new RobotDrive(1, 2, 3, 4);
                // Define joystick being used at USB port #1 on the Drivers Station
                m_driveStick = new Joystick(1);
                // Twist is on Axis 3 for the Extreme 3D Pro
                m_driveStick->SetAxisChannel(Joystick::kTwistAxis, 3);
        }
        /**
         * Gets called once for each new packet from the DS.
         */
        void TeleopPeriodic(void)
        {
                m_robotDrive->MecanumDrive_Cartesian(m_driveStick->GetX(), m_driveStick-
>GetY(), m_driveStick->GetTwist());
```

```
        }
};
START_ROBOT_CLASS(MecanumDefaultCode);
```

Here's a sample program that shows the minimum code to drive using a single joystick and mecanum wheels. It uses the RobotDrive object that is available in both C++ and Java so even though this example is in C++ similar code will work in Java. The idea is to create the RobotDrive object with 4 PWM ports for the 4 speed controllers on the robot. The joystick XY position represents a direction vector that the robot should follow regardless of its orientation. The twist axis on the joystick represents the rate of rotation for the robot while it's driving.

Thanks to **FRC Team 2468** in Austin, TX for developing this example.

Updating the program for field-oriented driving

I would be remiss in not mentioning that is a 4th parameter to the MecanumDrive_Cartesian() method that is the angle returned from a Gyro sensor. This will adjust the rotation value supplied, in this case, from the twist axis of the joystick to be relative to the field rather than relative to the robot. This is particularly useful with mecanum drive since, for the purposes of steering, the robot really has no front, back or sides. It can go in any direction. Adding the angle in degrees from a gyro object will cause the robot to move away from the drivers when the joystick is pushed forwards, and towards the drivers when it is pulled towards them - regardless of what direction the robot is facing!

The use of field-oriented driving makes often makes the robot much easier to drive, especially compared to a "robot-oriented" drive system where the controls are reversed when the robot is facing the drivers.

Just remember to get the gyro angle each time MecanumDrive_Caresian() is called.

Sample Robot Walkthru

Constructing the robot program

Wiring file

Wiring file

The wiring file is automatically generated using the "Wiring Table" toolbar button based on the actuators and sensors added to each subsystem.

RobotDrive object

RobotDrive object

All the motors are inverted based on the gearing and mounting of the motors. Adding gears reverses the direction of the motor for each gear added.

Elevator

Elevator

The elevator is controlled by a potentiometer and driven with a motor connected with a Victor speed controller. Since this is a PIDSubsystem, it has P, I, and D constants and a Tolerance value that determines when the elevator has reached it's target. The target is used by commands to know when they are finished for sequential operations.

Wrist

Wrist

The wrist is a PIDSubsystem with a Victor speed controller motor and a potentiometer to measure the wrist angle. The P, I, and D values are used with the built-in PID controller and the tolerance is used to determine if the wrist has reached the desired angle.

Gripper

Gripper

Subsystems and commands

Subsystems and commands

There are a number of commands that are used to implement this robots operation, but each one only requires writing a few lines of code to make them operate. In most cases the code sets a timeout for timed commands, or sets a PID target for PID controlled commands.

Claw commands

Claw commands

There are commands that open and close the claw. It has no sensor and is operated for a fixed time rather than from sensor feedback. It is generally better to have some sensor feedback to control the subsystem rather than relying on time, but this is simply a Vex motor and running it for a longer time than necessary won't hurt anything. For larger FRC motors, the motor could be easily damaged if it runs open-loop like this.

DriveForward

DriveForward

This command has the robot driving forward for a fixed period of time, in this case about 0.5 seconds. The command starts the motors running, then uses a timeout to stop them after driving for a while. The end() method stops the robot from driving and the interrupted() method simply calls the end() method.

Operator Interface

Operator Interface

The operator interface in this robot is fairly simple, just a logitech gamepad with a few buttons to operate some commands.

DriveWithJoystick

DriveWithJoystick

DriveWithJoystick calls the method on the drive base to drive the robot using the joystick that is created as part of the operator interface. This command never finishes because it's the default command for the drive base subsystem so the robot can drive using the joystick when it's not doing anything else.

Drive base code for mecanum drive

Drive base code for mecanum drive

the DriveBase subsysem has a default command (DriveWithJoystick) that allows the user to drive the robot when no other command is using the drive base. It drives using the mecanum drive methods as part of the RobotDrive class. The speed is inverted for forward driving because the joystick has positive (forward) values when the stick is pulled backwards.

Elevator Commands

Elevator Commands

Elevator commands the move the elevator from the upper (drop off) position to the lower (pickup) positions. Both use the potentiometer for feedback.

Wrist Commands

Wrist Commands

Wrist PIDSetpoint commands that cause the wrist to go to the horizontal and up (stowed) positions.