SmartDashboard

Last Updated: 01-11-2019

Table of Contents

SmartDashboard

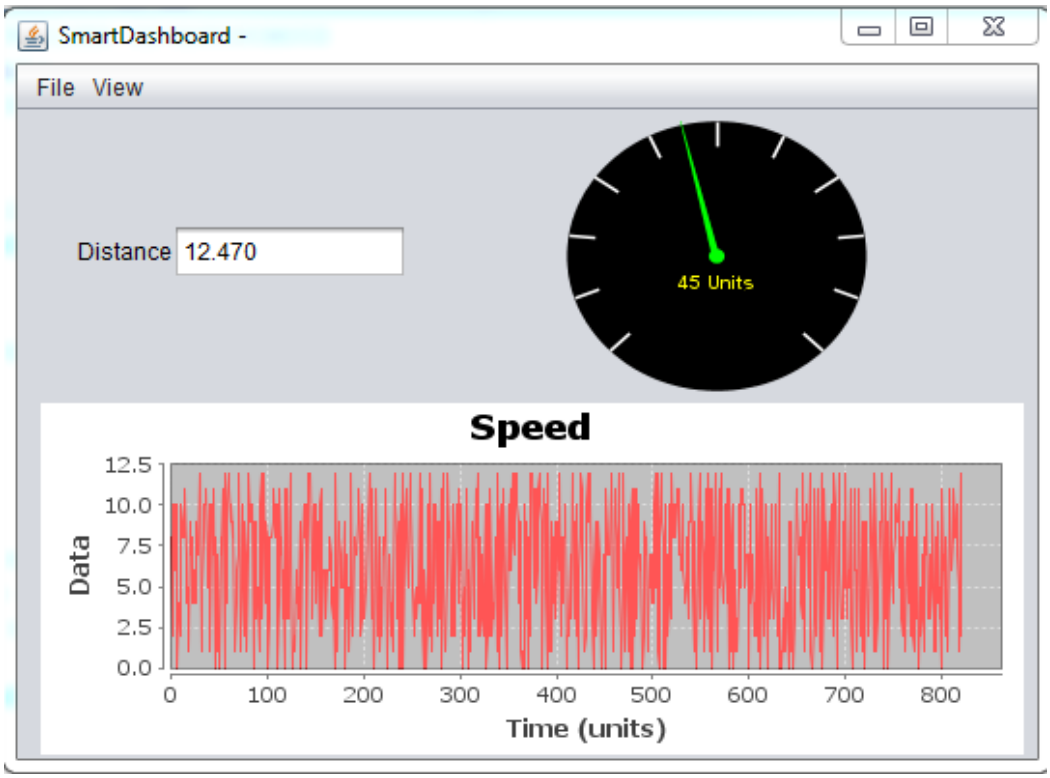
Getting Started with the SmartDashboard

The SmartDashboard typically runs on the Driver Station computer and will do two functions:

1. View robot data that is displayed as program status as your program is running.
2. View sensor data and operate actuators in Test mode for robot subsystems to verify correct operation.

The switch between program status and test modes are done on the Driver Station.

What is the SmartDashboard?



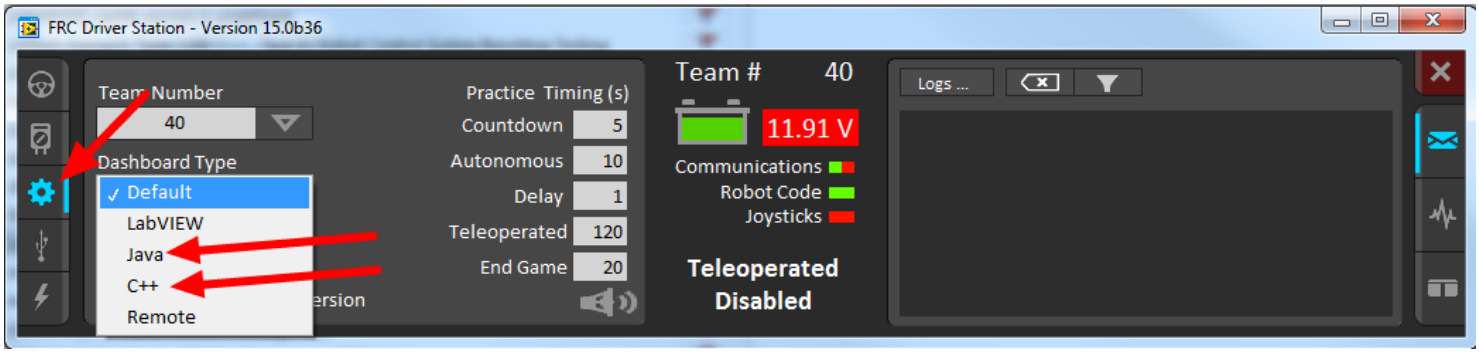
The SmartDashboard is a Java program that will display robot data in real time. The SmartDashboard helps you with these things:

SmartDashboard

- Displays robot data of your choice while the program is running. It can be displayed as simple text fields or more elaborately in many other display types like graphs, dials, etc.
- Displays the state of the robot program such as the currently executing commands and the status of any subsystems
- Displays buttons that you can press to cause variables to be set on your robot
- Allows you to choose startup options on the dashboard that can be read by the robot program

The displayed data is automatically formatted in real-time as the data is sent from the robot, but you can change the format or the display widget types and then save the new screen layouts to be used again later. And with all these options, it is still extremely simple to use. To display some data on the dashboard, simply call one of the SmartDashboard methods with the data and its name and the value will automatically appear on the dashboard screen.

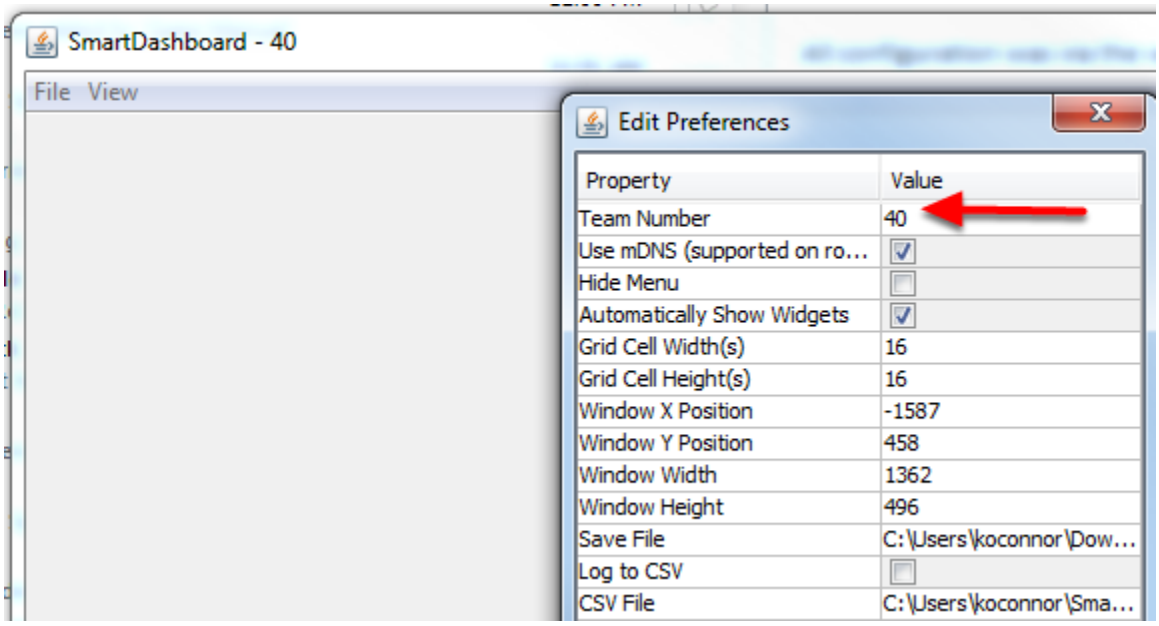
Installing the SmartDashboard



The SmartDashboard is now packaged with the C++ and Java language updates and can be launched directly from the Driver Station by selecting the **Java** or **C++** buttons on the **Setup** tab.

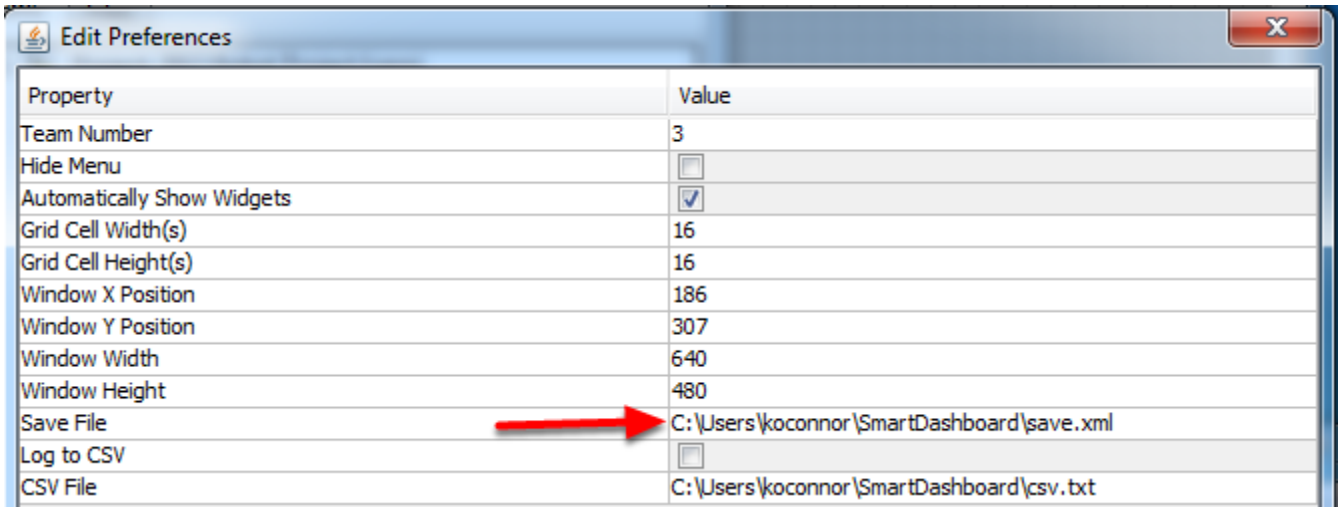
Note: If using the Classmate PC or other PC where the DS is run from a separate account from the C++\Java tools install (e.g. Driver and Developer) the buttons shown above may not work. You can utilize the SmartDashboard in this type of setup in one of two ways. The first way is to set the type to **Default** and modify the DS INI file to launch the appropriate dashboard, details can be found in this article. The second option is to copy the C:\Users\Developer\wpilib\tools directory to C:\Users\Driver\wpilib\tools. When using this second method it is recommended to make sure that the Dashboard under both the Driver and Developer accounts point to the same save file (see Locating the Save File) below.

Configuring the Team Number



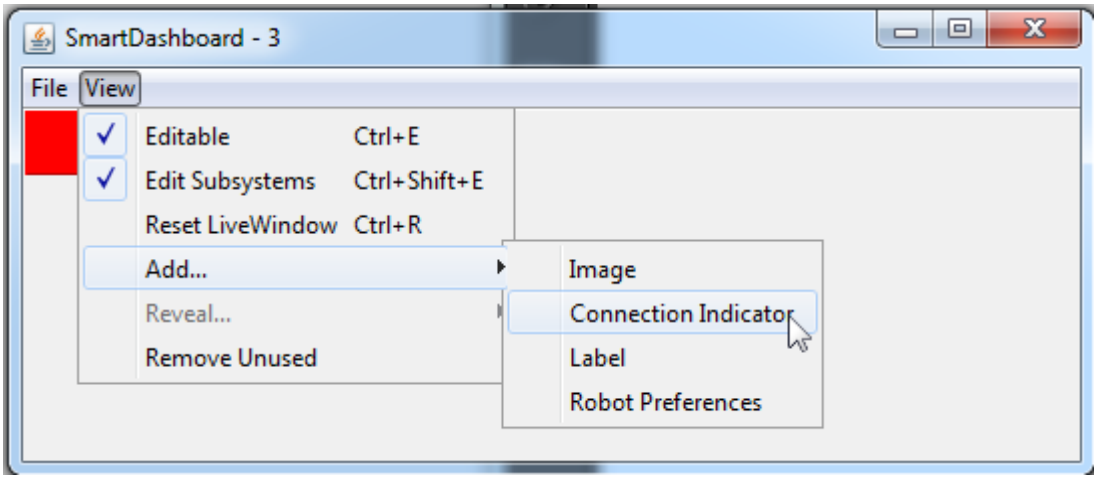
The first time you launch the SmartDashboard you should be prompted for your team number. To change the team number after this: click **File > Preferences** to open the Preferences dialog. Double-click the box to the right of **Team Number** and enter your FRC Team Number, then click outside the box to save. **Note that the SmartDashboard will take a moment to configure itself for the team number, do not be alarmed.**

Locating the Save File



Users may wish to customize the save location of the SmartDashboard. To do this click the box next to **Save File** then browse to the folder where you would like to save the configuration. **It is recommended that this folder be inside the users home directory and not inside the sunspotfrcsdk or workbench directories.** Files saved in the installation directories for the WPILib components will likely be overwritten on updates to the tools.

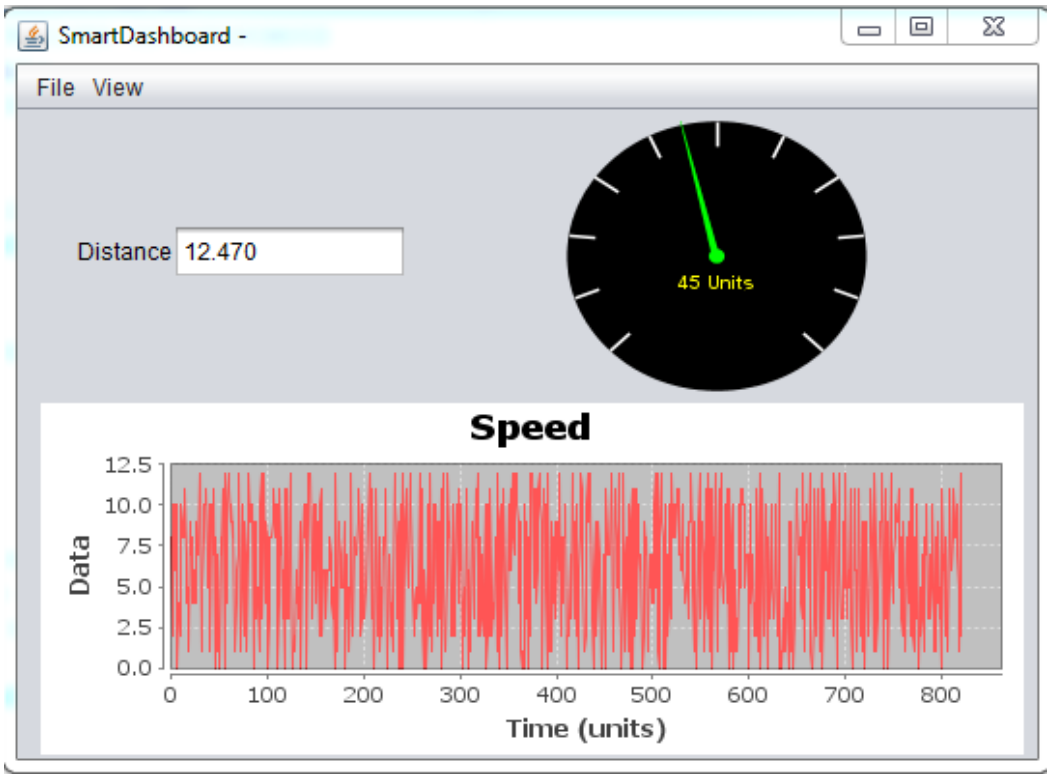
Adding a Connection Indicator



SmartDashboard

It is often helpful to see if the SmartDashboard is connected to the robot. To add a connection indicator, select **View** > **Add** > **Connection Indicator**. This indicator will be red when disconnected and green when connected. To move or resize this indicator, select **View** > **Editable** to toggle the SmartDashboard into editable mode, then drag the center of the indicator to move it or the edges to resize. Select the **Editable** item again to lock it in place.

Adding Widgets to the SmartDashboard



Widgets are automatically added to the SmartDashboard for each "key" sent by the robot code. For instructions on adding robot code to write to the SmartDashboard see Displaying Expressions from Within the Robot Program.

Displaying Expressions from Within the Robot Program

Often debugging or monitoring the status of a robot involves writing a number of values to the console and watching them stream by. With SmartDashboard you can put values to a GUI that is automatically constructed based on your program. As values are updated, the corresponding GUI element changes value - there is no need to try to catch numbers streaming by on the screen.

Writing values to the SmartDashboard

```
protected void execute() {
    SmartDashboard.putBoolean("Bridge Limit", bridgeTipper.atBridge());
    SmartDashboard.putNumber("Bridge Angle", bridgeTipper.getPosition());
    SmartDashboard.putNumber("Swerve Angle", drivetrain.getSwerveAngle());
    SmartDashboard.putNumber("Left Drive Encoder", drivetrain.getLeftEncoder());
    SmartDashboard.putNumber("Right Drive Encoder", drivetrain.getRightEncoder());
    SmartDashboard.putNumber("Turret Pot", turret.getCurrentAngle());
    SmartDashboard.putNumber("Turret Pot Voltage", turret.getAverageVoltage());
    SmartDashboard.putNumber("RPM", shooter.getRPM());
}
```

You can write Boolean, Numeric, or String values to the SmartDashboard by simply calling the correct method for the type and including the name and the value of the data, no additional code is required. Any time in your program that you write another value with the same name, it appears in the same UI element on the screen on the driver station or development computer. As you can imagine this is a great way of debugging and getting status of your robot as it is operating.

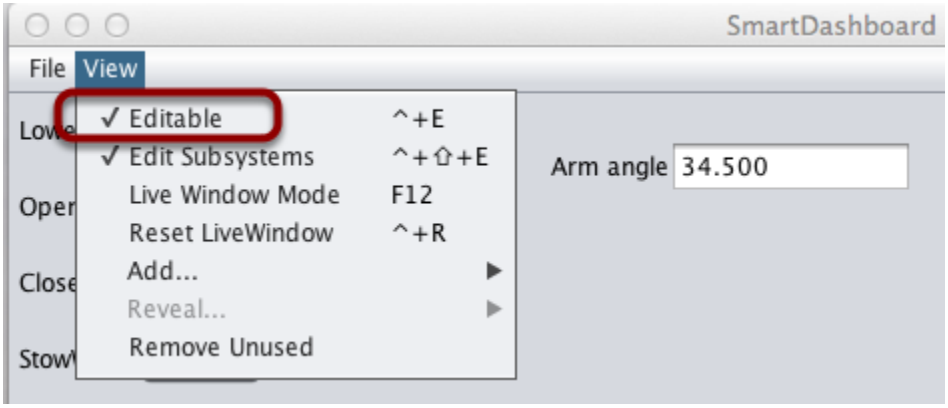
Creating widgets on the SmartDashboard

Widgets are populated on the SmartDashboard automatically, no user intervention is required. Note that the widgets are only populated when the value is first written, you may need to enable your robot in a particular mode or trigger a particular code routine for an item to appear. To alter the appearance of the widget, see the next two sections Changing the Display Properties of a Value and Changing the Display Widget Type for a Value.

Changing the display properties of a value

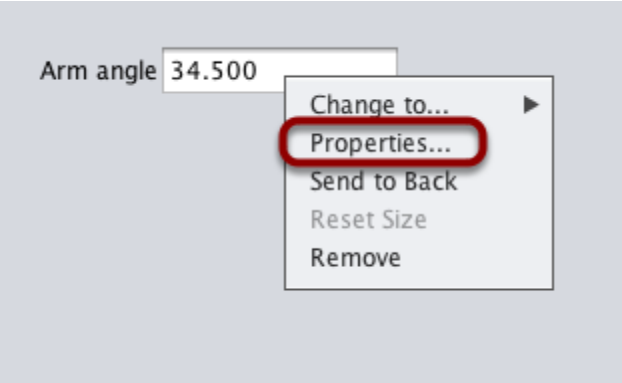
Each value displayed with SmartDashboard has a a set of properties that effect the way it's displayed.

Setting the SmartDashboard display into editing mode



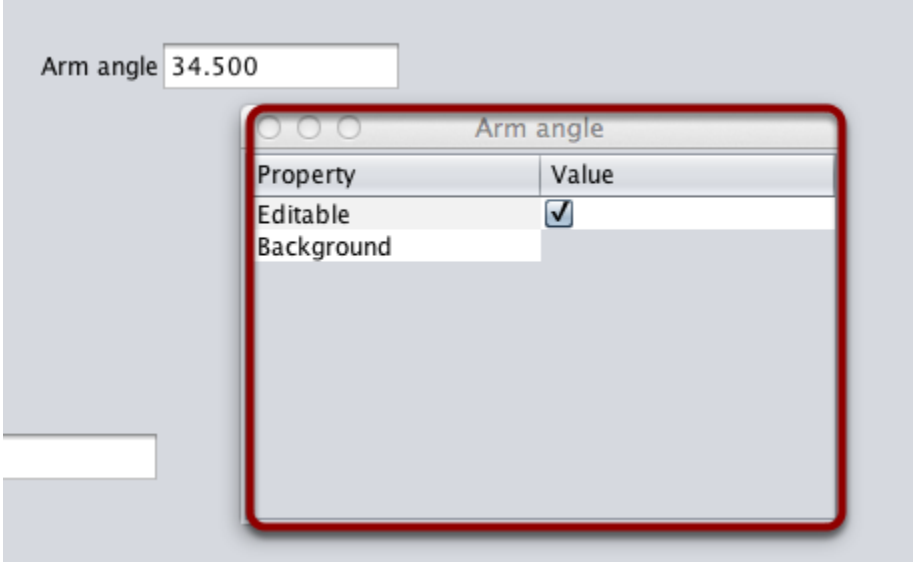
The SmartDashboard has two modes it can operate in, display mode and edit mode. In edit mode you can move around widgets on the screen and edit their properties. To put the SmartDashboard into edit mode, click the "View" menu, then select "Editable" to turn on edit mode.

Getting the properties editor for a widget



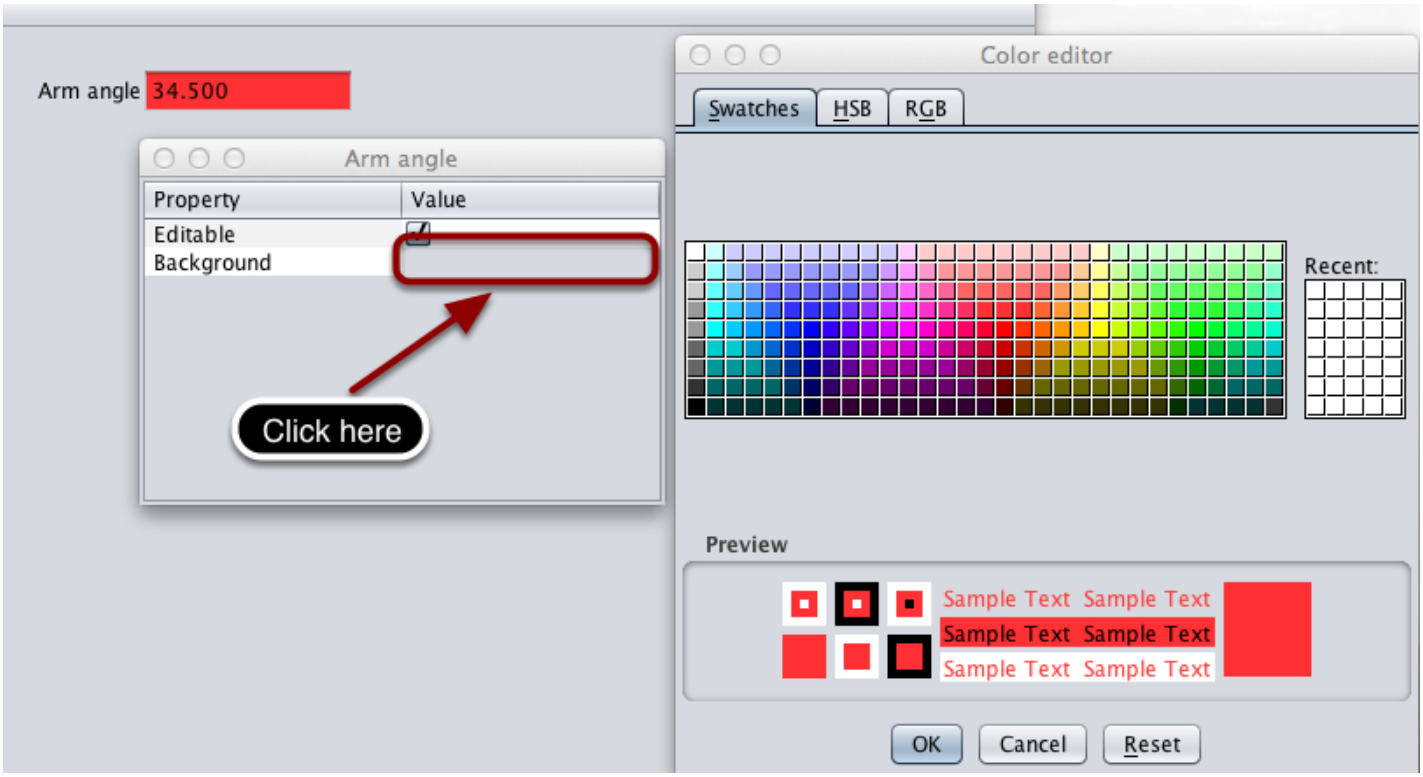
Once in edit mode, you can display and edit the properties for a widget. Right-click on the widget and select "Properties...".

Editing the properties on a field



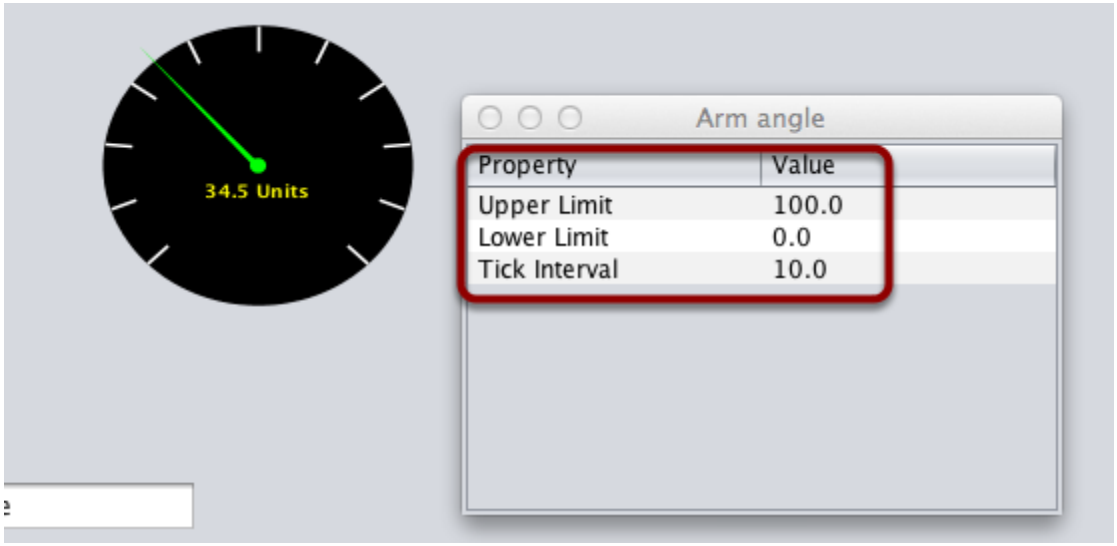
A dialog box will be shown in response to the "Properties..." menu item on the widgets right-click context menu.

Editing the widgets background color



To edit a property value, say, Background color, click the background color shown (in this case grey), and choose a color from the color editor that pops up. This will be used as the widgets background color.

Edit properties of other widget types

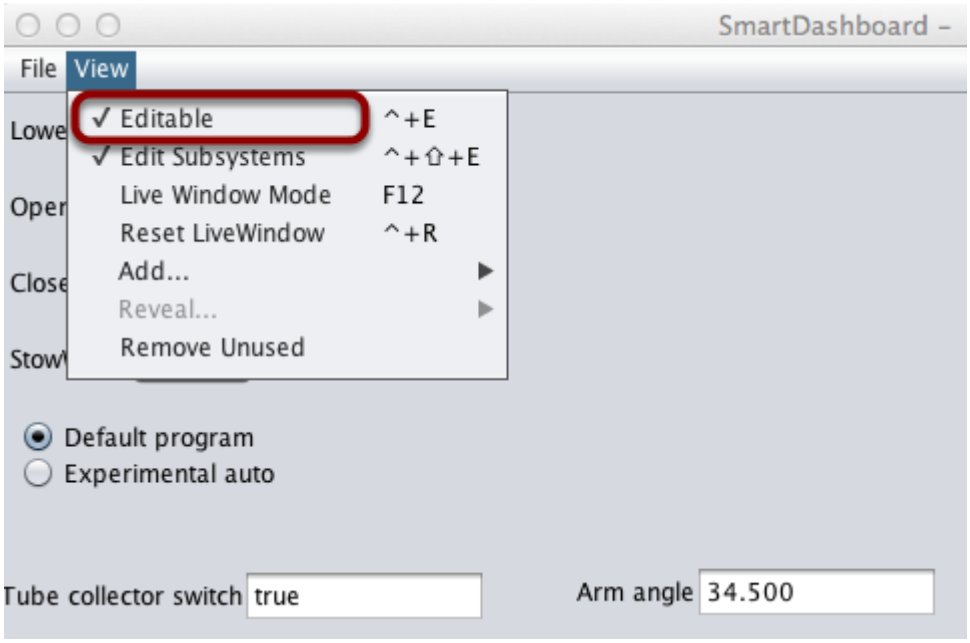


Different widget types have different sets of editable properties to change the appearance. In this example, the upper and lower limits of the dial and the tick interval are changeable parameters.

Changing the display widget type for a value

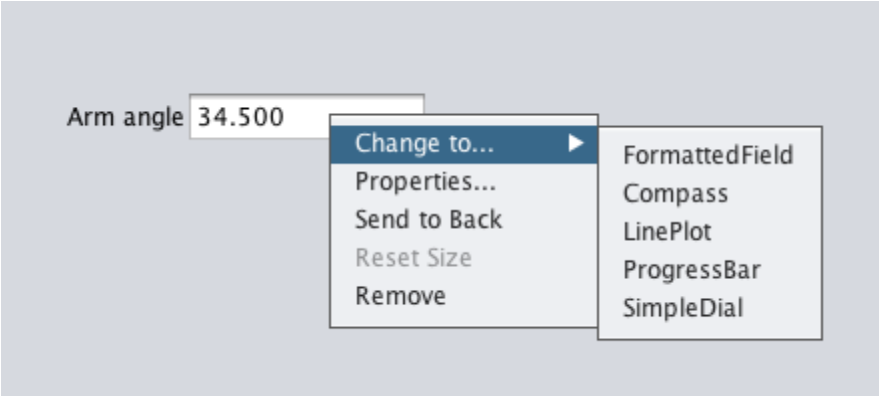
You can change the type of widget that displays values with the SmartDashboard. The allowable widgets depend on the type of the value being displayed.

Set edit mode



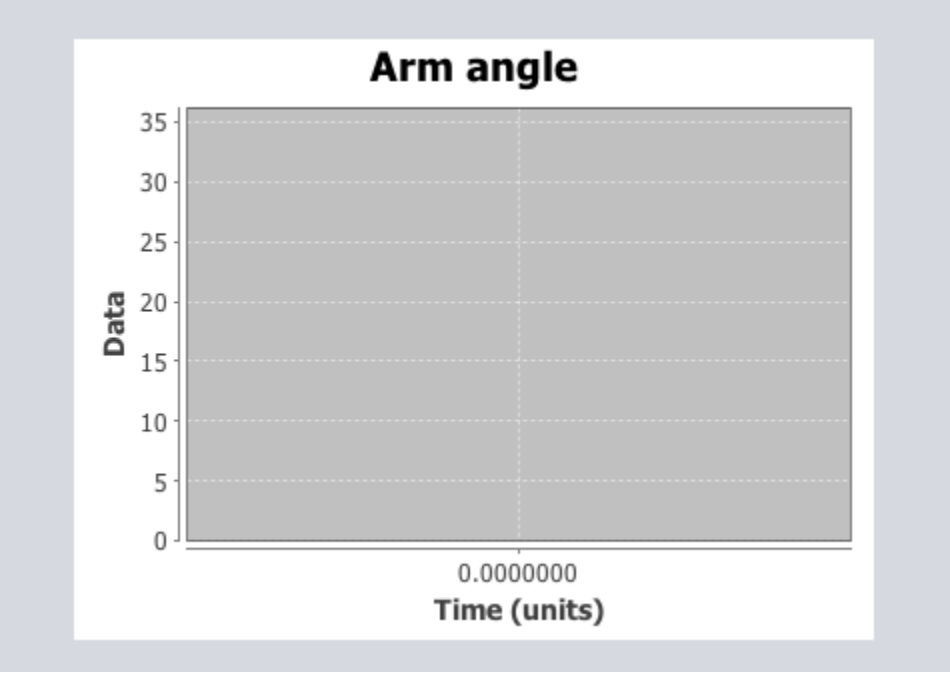
Make sure that the SmartDashboard is in edit mode. This is done by selecting "Editable" from the "View menu"

Choose the new widget type



Right-click on the widget and select "Change to...". Then pick the type of widget to use for the particular value. In this case we choose **LinePlot**.

New widget type is shown for the value

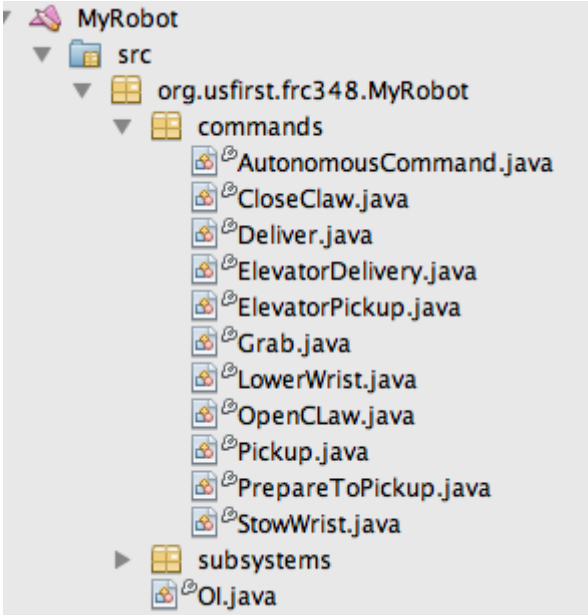


The new widget type is displayed. In this case, a Line Plot, will show the values of the Arm angle value over time. You can set the properties of the graph to make it better fit your data by right-clicking and selecting "Properties...". See: Changing the display properties of a value.

Testing commands

Commands represent robot behaviors such as moving an elevator to a position, collecting balls, shooting, or other tasks. It is desirable to test commands on the robot as they are written before combining them into more complex commands or incorporating them into other parts of the robot program. With a single line of code you can display commands on the SmartDashboard that appear as buttons that run the commands when pressed. This makes robot debugging a much simpler process than before.

Robot project with a number of commands that need testing



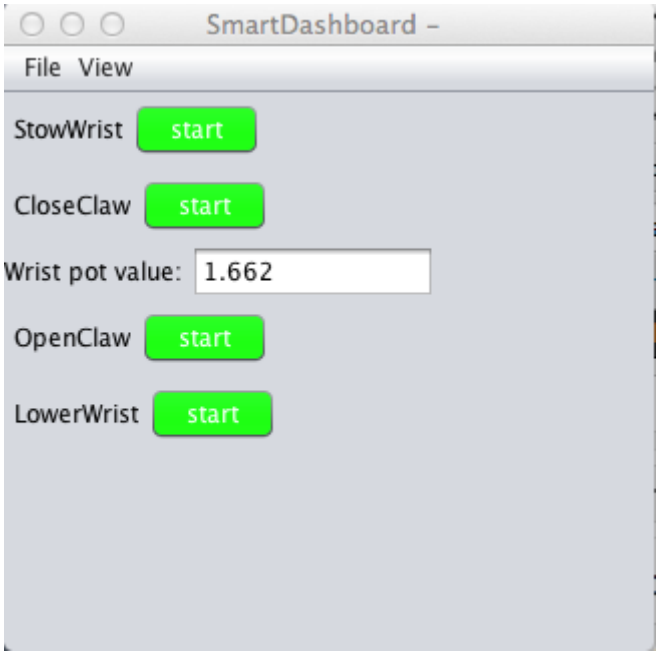
One of the features of commands is that it allows the program to be broken down into separate testable units. Each command can be run independently of any of the others. By writing commands to the SmartDashboard, they will appear on the screen as buttons that, when pressed, schedule the particular command. This allows any command to be tested individually by pressing the button and observing the operation.

Adding command instances to the SmartDashboard

```
73              joystickButton2 = new JoystickButton(gamePad, 2);
74              joystickButton2.whenPressed(new OpenCLaw());
75              joystickButton = new JoystickButton(gamePad, 1);
76              joystickButton.whenPressed(new CloseClaw());
77
78          // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
79
80              SmartDashboard.putData("OpenClaw", new OpenCLaw());
81              SmartDashboard.putData("CloseClaw", new CloseClaw());
82              SmartDashboard.putData("StowWrist", new StowWrist());
83              SmartDashboard.putData("LowerWrist", new LowerWrist());
84          }
85      }
86
```

Commands are written to the SmartDashboard simply by adding a line for each command to your program. In the above example 4 commands are written to the SmartDashboard by specifying the command name and an instance of the command.

Commands in the SmartDashboard



SmartDashboard

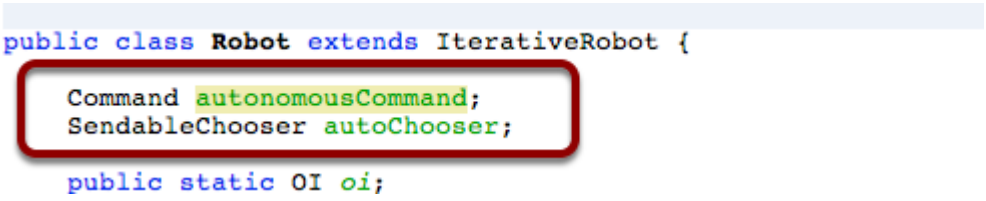
Here you can see the resultant buttons on the SmartDashboard that appear corresponding to each command that was sent. Pressing the button will run the command until the isFinished() method in the command returns true. While the command is running, the "start" button will change to "cancel" giving you an opportunity to cancel the command if it isn't working properly. The "Wrist pot value" is just a numeric value that was written to the dashboard and is not part of the output from writing the commands.

Choosing an autonomous program from SmartDashboard

Often teams have more than one autonomous program, either for competitive reasons or for testing new software. Programs often vary by adding things like time delays, different strategies, etc. The methods to choose the strategy to run usually involves switches, joystick buttons, knobs or other hardware based inputs.

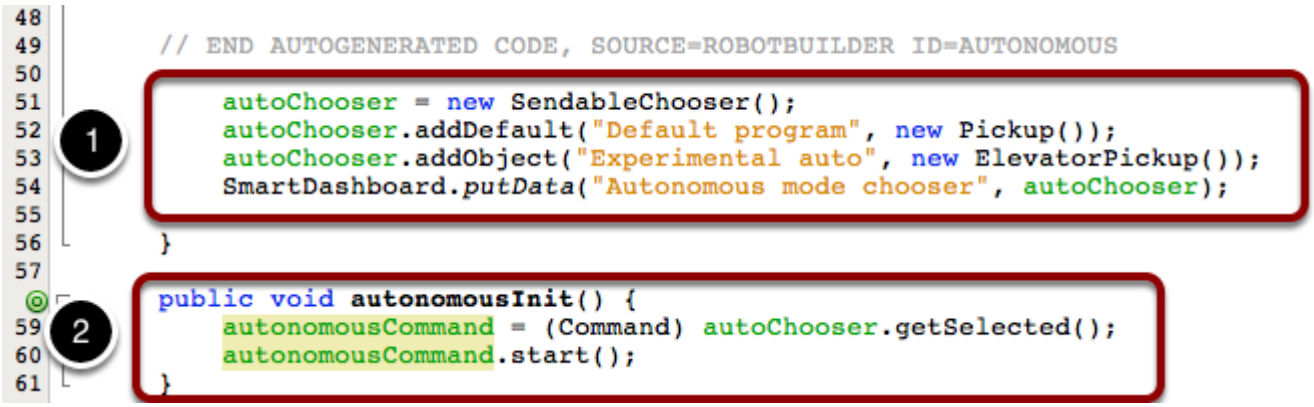
With the SmartDashboard you can simply display a widget on the screen to choose the autonomous program that you would like to run. And with command based programs, that program is encapsulated in one of several commands. This article shows how to select an autonomous program with only a few lines of code and a nice looking user interface.

Creating the SendableChooser object in Robot.java



Create a variable to hold a reference to a SendableChooser object. This example also uses a RobotBuilder variable to hold the Autonomous command.

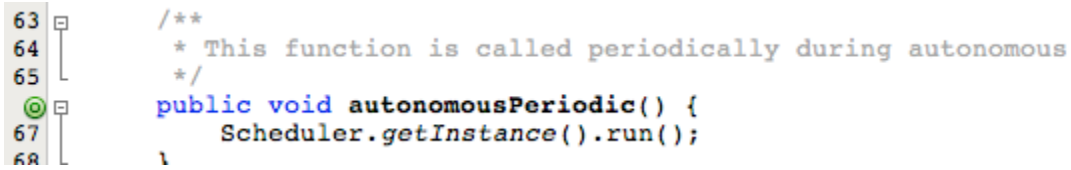
Set up the SendableChooser in the robotInit() method



Imagine that you have two autonomous programs to choose between and they are encapsulated in commands Pickup and ElevatorPickup. To choose between them:

1. Create a SendableChooser object and add instances of the two commands to it. There can be any number of commands, and the one added as a default (addDefault), becomes the one that is initially selected. Notice that each command is included in an addDefault() or addObject() method call on the SendableChooser instance.
2. When the autonomous period starts the SendableChooser object is polled to get the selected command and that command is scheduled.

Run the scheduler during the autonomous period



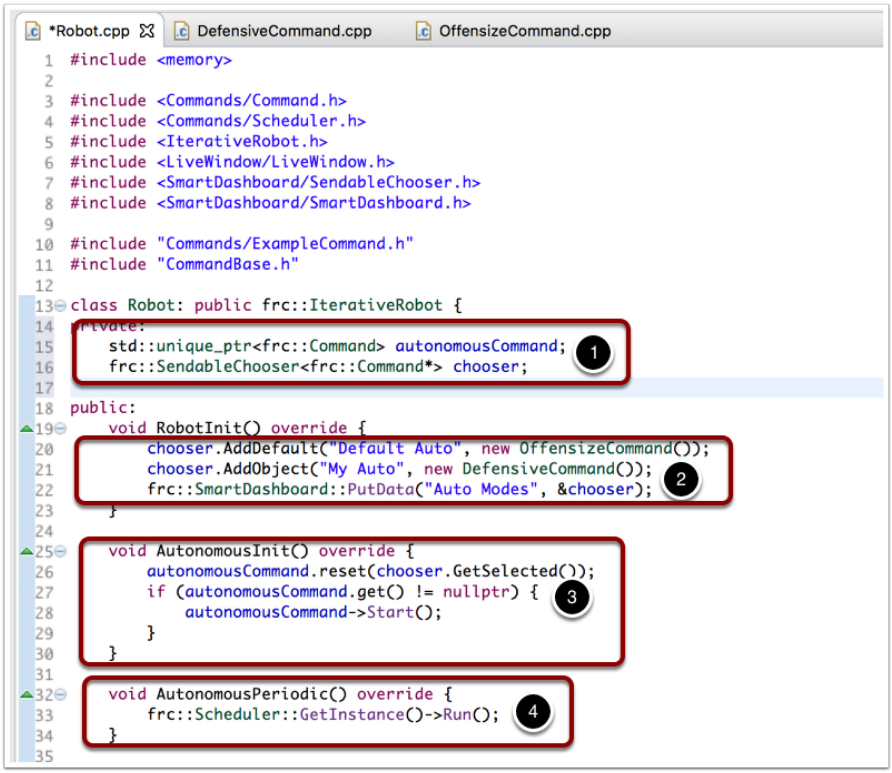
RobotBuilder will generate code automatically that runs the scheduler every driver station update period (about every 20ms). This will cause the selected autonomous command to run.

SmartDashboard

SmartDashboard display

When the SmartDashboard is run, the choices from the SendableChooser are automatically displayed. You can simply pick an option before the autonomous period begins and the corresponding command will run.

Creating a SendableChooser in C++



SmartDashboard

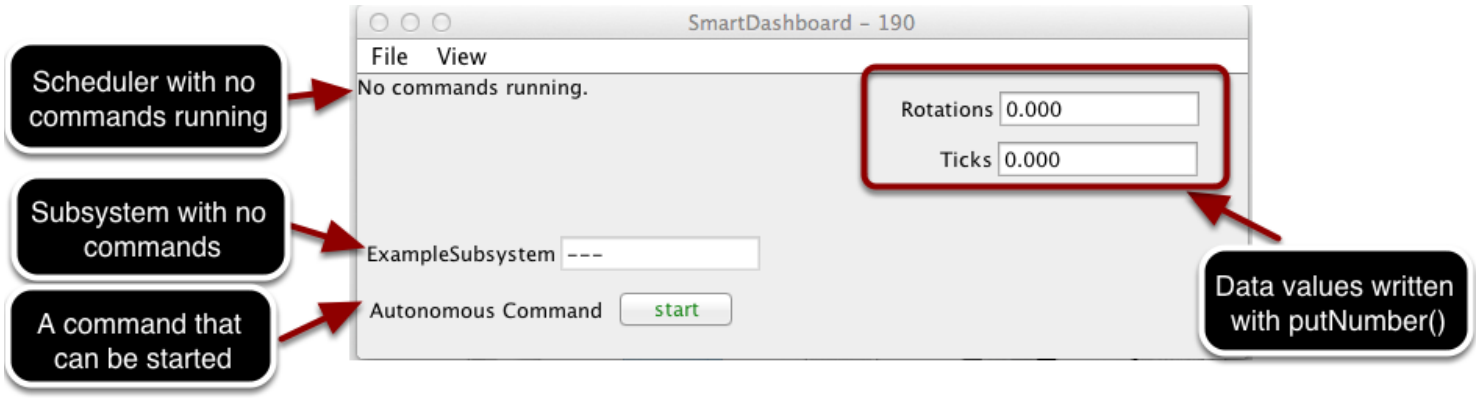
This is an example of creating a SendableChooser object and using it to select between a Defensive and Offensive autonomous command to run when the autonomous period of the match starts. Just as in the Java example:

1. Create variables to hold the autonomousCommand pointer and the SendableChooser pointer.
2. the SendableChooser is created and initialized in the RobotInit() method.
3. In the AutonomousInit() method just before the Autonomous code starts running, the choosen command is retrieved from the SmartDashboard and scheduled.
4. In the AutonomousPeriodic() method, the scheduler is repeatedly run.

Displaying the status of Commands and Subsystems

If you are using the command-based programming features of WPILib, you will find that they are very well integrated with SmartDashboard. It can help diagnose what the robot is doing at any time and it gives you control and a view of what's currently running.

The SmartDashboard command system displays

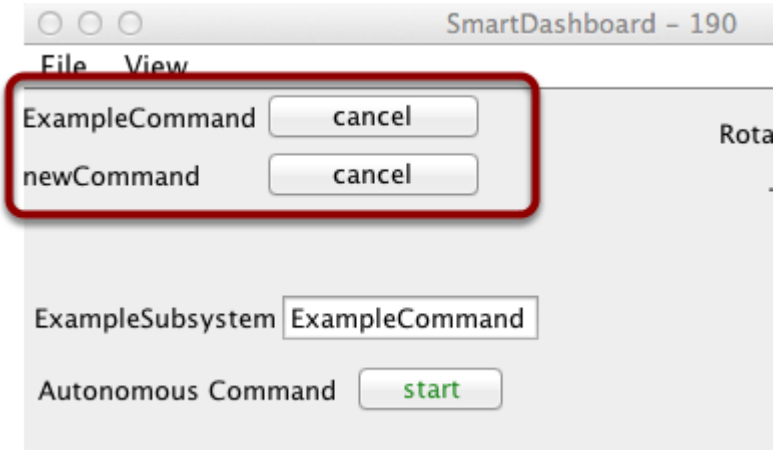


With SmartDashboard you can display the status of the commands and subsystems in your robot program in various ways. The outputs should significantly reduce the debugging time for your programs. In this picture you can see a number of displays that are possible. Displayed here are:

- The Scheduler currently with "No commands running". In the next example you can see what it looks like with a few commands running showing the status of the robot.
- A subsystem, "ExampleSubsystem" that indicates that there are currently no commands running that are "requiring" it. When commands are running, it will indicate the name of the commands that are using the subsystem.
- A command written to SmartDashboard that shows a "start" button that can be pressed to run the command. This is an excellent way of testing your commands one at a time.
- And a few data values written to the dashboard to help debug the code that's running.

In the following examples, you'll see what the screen would look like when there are commands running, and the code that produces this display.

The scheduler display showing a few commands running



This is the scheduler status when there are two commands running, "ExampleCommand" and "newCommand". This replaces the "No commands running." message from the previous screen image. You can see commands displayed on the dashboard as the program runs and various commands are triggered.

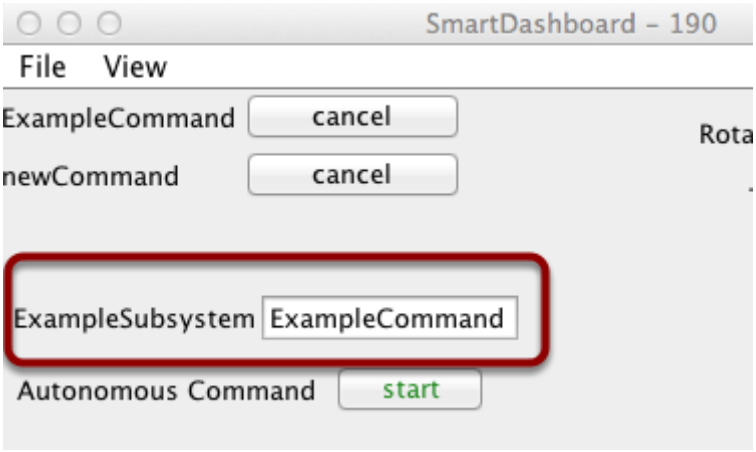
Displaying the Scheduler status

```
14  public class TestScheduler extends IterativeRobot {
15
16      Command autonomousCommand;
17
18      public void robotInit() {
19          autonomousCommand = new ExampleCommand();
20
21          CommandBase.init();
22          SmartDashboard.putData(Scheduler.getInstance());
23      }
```

You can display the status of the Scheduler (the code that schedules your commands to run). This is easily done by adding a single line to the RobotInit method in your RobotProgram as shown

here. In this example the Scheduler instance is written using the putData method to SmartDashboard. This line of code produces the display in the previous image.

Displaying the status of a subsystem



Running commands will "require" subsystems. That is the command is reserving the subsystem for its exclusive use. If you display a subsystem on SmartDashboard, it will display which command is currently using it. In this example, "ExampleSubsystem" is in use by "ExampleCommand".

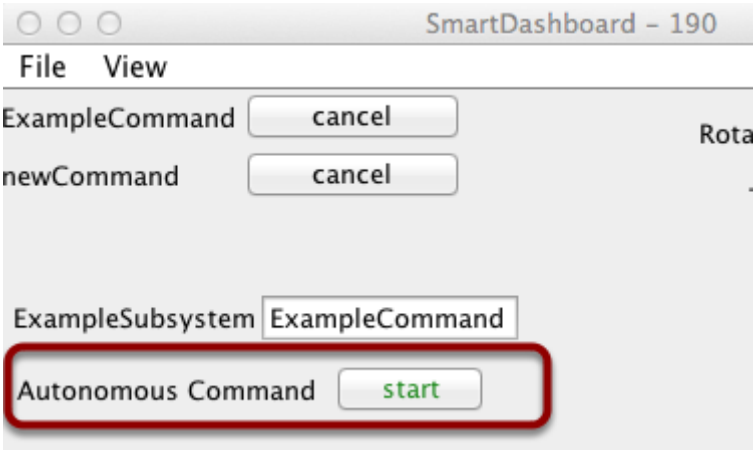
Writing the code to display a subsystem

```
7
public abstract class CommandBase extends Command {
9
10      public static OI oi;
11      // Create a single static instance of all of your subsystems
12      public static ExampleSubsystem exampleSubsystem = new ExampleSubsystem();
13
14      public static void init() {
15          oi = new OI();
16
17          SmartDashboard.putData(exampleSubsystem);
18      }
19
```

In this example we are writing the command instance, "exampleSubsystem" and instance of the "ExampleSubsystem" class to the SmartDashboard. This causes the display shown in the previous

image. The text field will either contain a few dashes, "---" indicating that no command is current using this subsystem, or the name of the command currently using this subsystem.

Adding a button to activate a command



In this example you can see a button labeled "Autonomous Command". Pressing this button will run the associated command and is an excellent way of testing commands one at a time without having to add throw-away test code to your robot program. Adding buttons for each command makes it simple to test the program, one command at a time.

Code required to create a button to run a command

```
17
    public void robotInit() {
19      autonomousCommand = new ExampleCommand();
20
21      CommandBase.init();
22      SmartDashboard.putData(Scheduler.getInstance());
23      SmartDashboard.putData("Autonomous Command", autonomousCommand);
24  }
25
```

This is the code required to create a button for the command on SmartDashboard. RobotBuilder will automatically generate this code for you, but it can easily be done by hand as shown here. Pressing the button will schedule the command. While the command is running, the button label changes from "start" to "cancel" and pressing the button will cancel the command.

Setting robot preferences from SmartDashboard

The Robot Preferences class is used to store values in the flash memory on the roboRIO. The values might be for remembering preferences on the robot such as calibration settings for potentiometers, PID values, etc. that you would like to change without having to rebuild the program. The values can be viewed on the SmartDashboard and read and written by the robot program.

Sample program that reads and writes preference values

C++

```
class Robot: public SampleRobot {

        Preferences *prefs;

        double armUpPosition;
        double armDownPosition;

        public void RobotInit() {
                prefs = Preferences::GetInstance();
                armUpPosition = prefs->GetDouble("ArmUpPosition", 1.0);
                armDownPosition = prefs->GetDouble("ArmDownPosition", 4.);
        }
}
```

Java

```
public class Robot extends SampleRobot {

        Preferences prefs;

        double armUpPosition;
        double armDownPosition;

        public void robotInit() {
                prefs = Preferences.getInstance();
```

SmartDashboard

Retrieves “ArmUpPosition” value from preferences table on roborio memory;

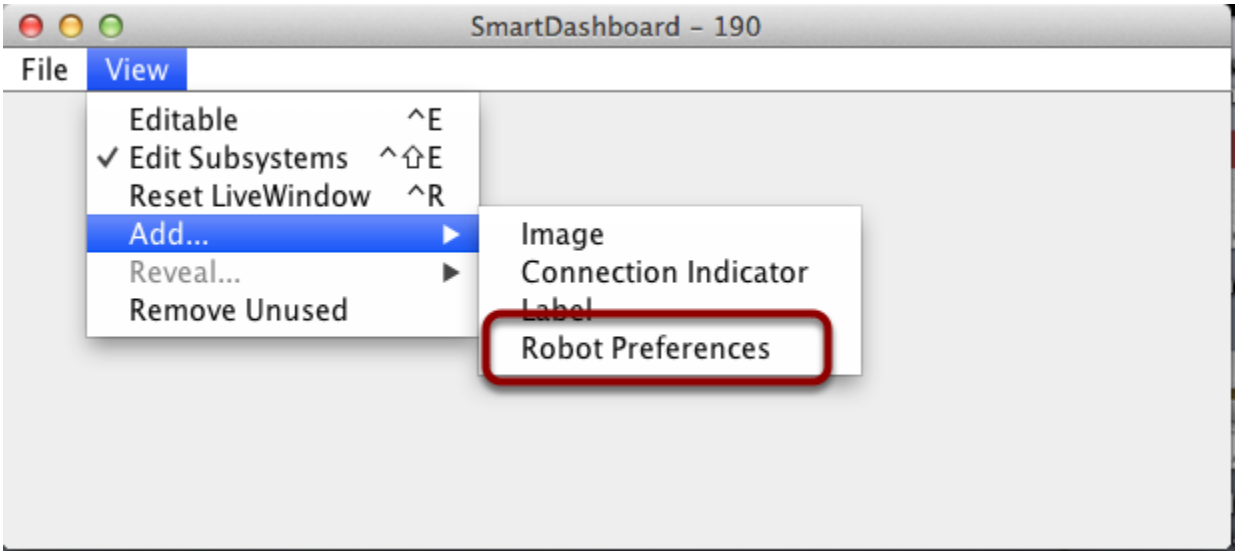
If none exists, returns the backup value and sets a new key

```
                armUpPosition = prefs.getDouble("ArmUpPosition", 1.0);
                armDownPosition = prefs.getDouble("ArmDownPosition", 4.);
        }
}
```

Often potentiometers are used to measure the angle of an arm or the position of some other shaft. In this case, the arm has two positions, "ArmUpPosition" and "ArmDownPosition". Usually programmers create constants in the program that are the two pot values that correspond to the positions of the arm. When the potentiometer needs to be replaced or adjusted then the program needs to be edited and reloaded onto the robot.

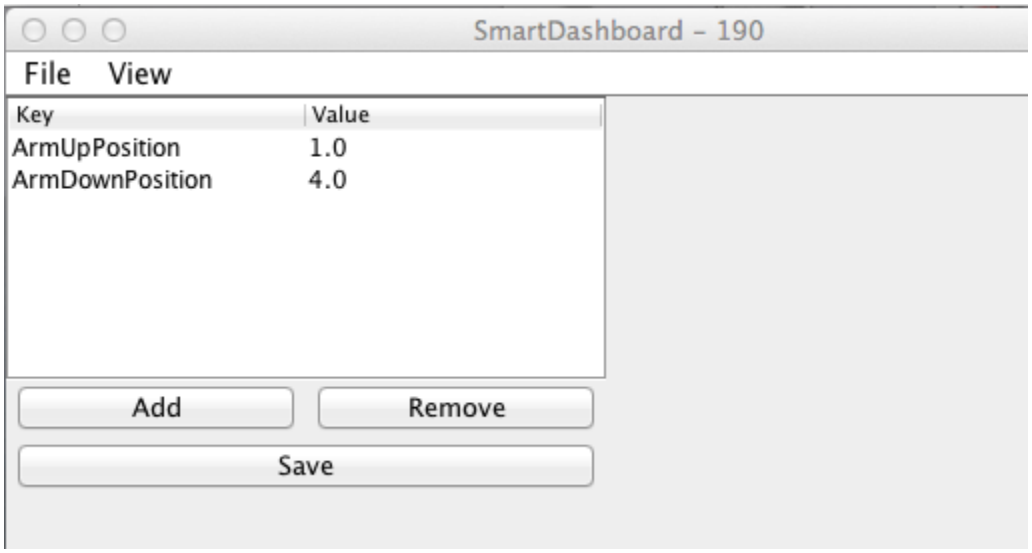
Rather than having "hard-coded" values in the program the potentiometer settings can be stored in the preferences file and read by the program when it starts. In this case the values are read on program startup in the robotInit() method. These values are automatically read from the preferences file stored in the roboRIO flash memory.

Displaying the Preferences widget in SmartDashboard



In the SmartDashboard the Preferences display can be added to the display revealing the contents of the preferences file stored in the roboRIO flash memory.

Viewing and editing the preference values



The values are shown here with the default values from the code. This was read from the robot through the NetworkTables interface built into SmartDashboard. If the values need to be adjusted they can be edited here and saved. The next time the robot program starts up the new values will be loaded in the robotInit() method. Each subsequent time the robot starts, the new values will be retrieved without having to edit and recompile/reload the robot program.

Verifying SmartDashboard is working

Minimal Java robot program

Minimal Java robot program

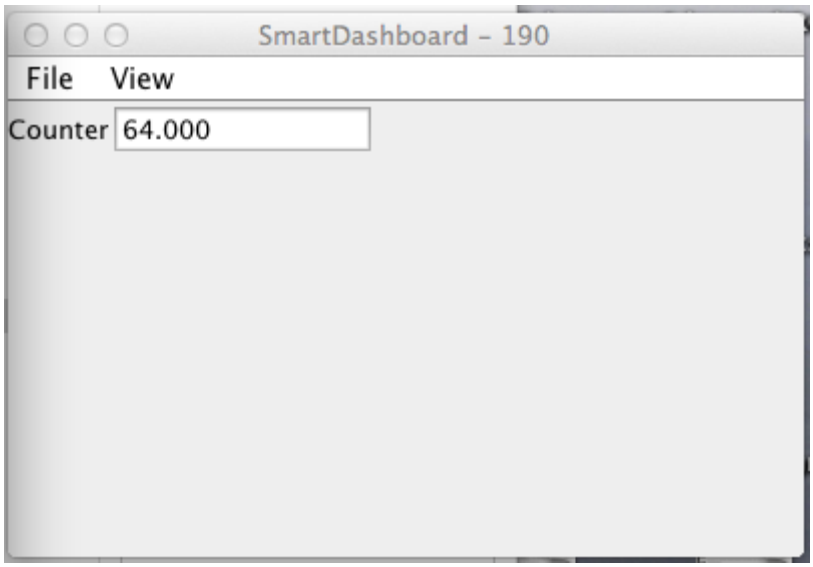
This is a minimal Java robot program that writes a value to the SmartDashboard. It simply increments a counter 10 times per second to verify that the connection is working.

Minimal C++ robot program

Minimal C++ robot program

This is a minimal C++ robot program that writes a value to the SmartDashboard. It simply increments a counter 10 times per second to verify that the connection is working.

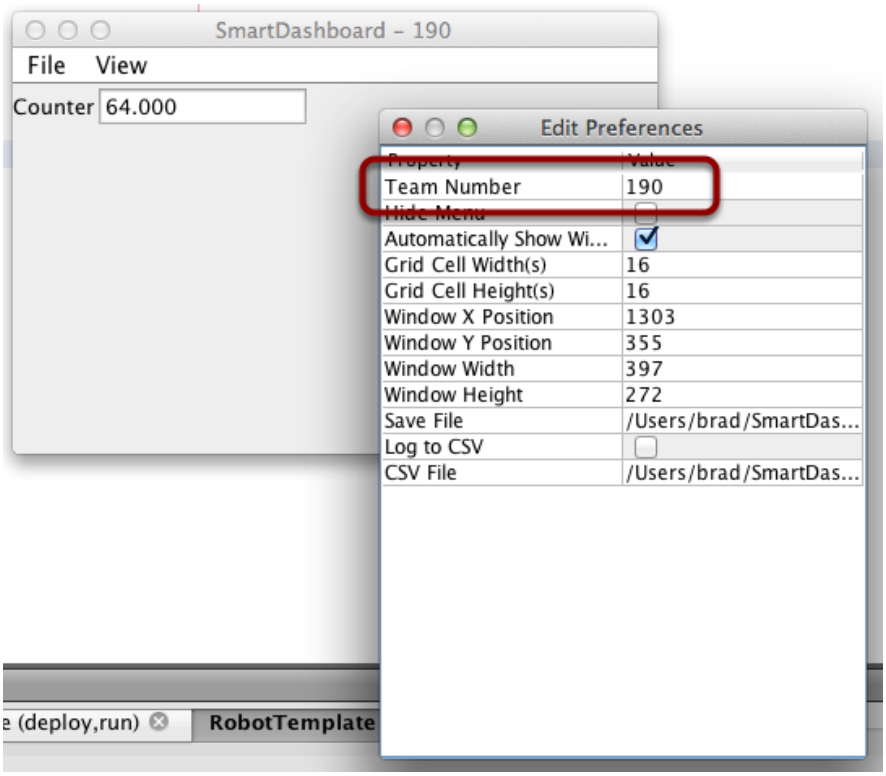
SmartDashboard output for the sample program



SmartDashboard

The SmartDashboard display should look like this after about 6 seconds of the robot being enabled in Teleop mode. If it doesn't then you need to check that the connection is correctly set up.

Verifying the IP address in SmartDashboard



If the display of the value is not appearing, verify that the team number is correctly set as shown in this picture. You get to the preferences dialog by selecting File, then Preferences.

Using OutlineViewer to verify that the program is working

Using OutlineViewer to verify that the program is working

You can verify that the robot program is generating SmartDashboard values by using the OutlineViewer program. This is a java program, OutlineViewer.jar that is located in the USERHOME\wpilib\tools folder. It is run with the command: java -jar OutlineViewer-with-dependencies.jar. In

the host box, enter your roboRIO hostname (roboRIO-####.local where #### is your team number with no leading zeroes). Then click "Start Client"

Look at the second row in the table, the value "SmartDashboard/Counter" is the variable written to the SmartDashboard via NetworkTables. As the program runs you should see the value increasing (41.0 in this case). If you don't see this variable in the OutlineViewer then you should look for something wrong with the robot program or the network configuration.

Test mode and LiveWindow

Enabling Test mode (LiveWindow)

You may add code to your program to display values for your sensors and actuators while the robot is in Test mode. This can be selected from the Driver Station whenever the robot is not on the field. The code to display these values is automatically generated by RobotBuilder and is described in the next article. Test mode is designed to verify the correct operation of the sensors and actuators on a robot. In addition it can be used for obtaining setpoints from sensors such as potentiometers and for tuning PID loops in your code.

Setting Test mode with the Driver Station

Setting Test mode with the Driver Station

Enable Test Mode in the Driver Station by clicking on the "Test" button and setting "Enable" on the robot. When doing this, the SmartDashboard display will switch to test mode (LiveWindow) and will display the status of any actuators and sensors used by your program.

Explicitly vs. implicit test mode display

```
RobotDrive drive = new RobotDrive(1, 2);
Jaguar jag;
Accelerometer accel = new Accelerometer(1, 2);


public void robotInit() {
    jag = new Jaguar(3);
    drive.setSafetyEnabled(false);
    LiveWindow.addActuator("SomeSubsystem", "jag", jag);
    LiveWindow.addSensor("SomeSubsystem", "accelerometer", accel);
    SmartDashboard.putData("TestPID", new PIDController(1, 1, 1, accel, jag));
    SmartDashboard.putNumber("X", 0.0);
}
```

All sensors and actuators will automatically be displayed on the SmartDashboard in test mode and will be named using the object type (such as Jaguar, Analog, Victor, etc.) with the module number and channel number with which the object was created. In addition, the program can explicitly add sensors and actuators to the test mode display, in which case programmer-defined subsystem and object names can be specified making the program clearer. This example illustrates explicitly defining those sensors and actuators in the highlited code.

Understanding what is displayed in Test mode

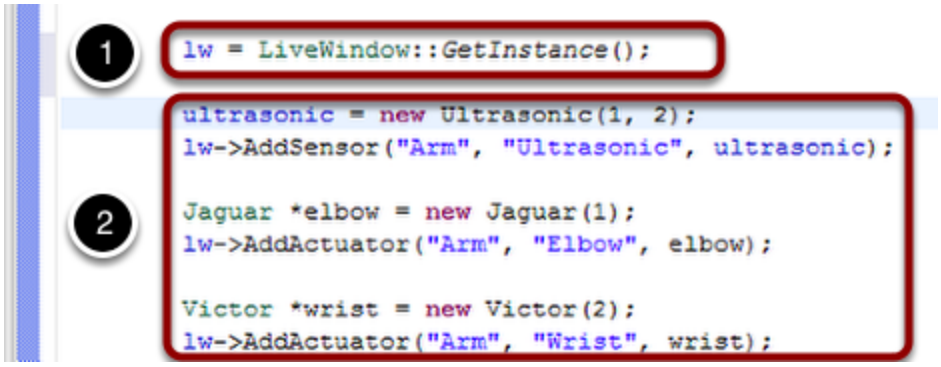
Understanding what is displayed in Test mode

This is the output in the SmartDashboard display when the robot is placed into test mode. In the display shown above the objects listed as Ungrouped were implicitly created by WPILib when the corresponding objects were created. These objects are contained in a subsystem group called "Ungrouped" **(1)** and are named with the device type (Analog, Jaguar in this case), and the module and channel numbers. The objects shown in the "SomeSubsystem" **(2)** group are explicitly created by the programmer from the code example in the previous section. These are named in the calls to LiveWindow.addActuator() and LiveWindow.AddSensor(). Explicitly created sensors and actuators will be grouped by the specified subsystem.

Displaying LiveWindow values

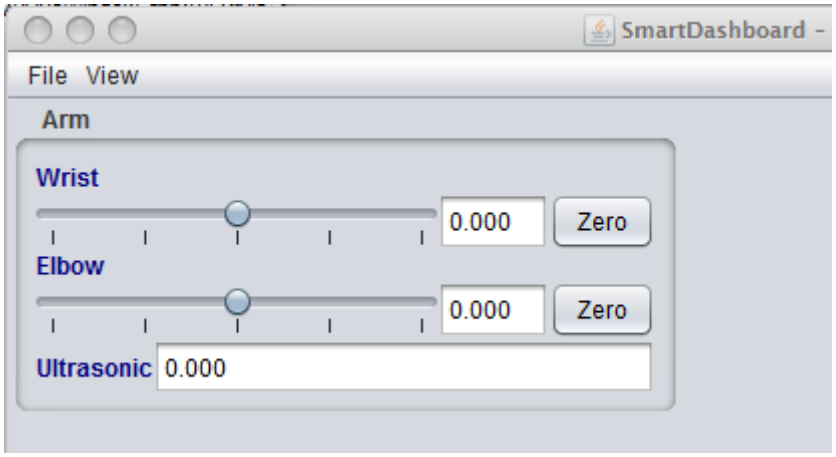
Typically LiveWindows are displayed as part of the automatically generated RobotBuilder code. You may also display LiveWindow values by writing the code yourself and adding it to your robot program. LiveWindow will display values grouped in subsystems. This is a convenient method of displaying whether they are actual command based program subsystems or just a grouping that you decide to use in your program.

Adding the necessary code to your program



Get a reference (in Java) or a pointer (in C++) to the LiveWindow object in your program. Then for each sensor or actuator that is created, add it to the LiveWindow display by either calling AddActuator or AddSensor (addActuator or addSensor in Java). When the SmartDashboard is put into LiveWindow mode, it will display the sensors and actuators.

Viewing the display in the SmartDashboard



The sensors and actuators added to the LiveWindow will be displayed grouped by subsystem. The subsystem name is just an arbitrary grouping the helping to organize the display of the sensors. Actuators can be operated by operating the slider for the two motor controllers.

PID Tuning with SmartDashboard

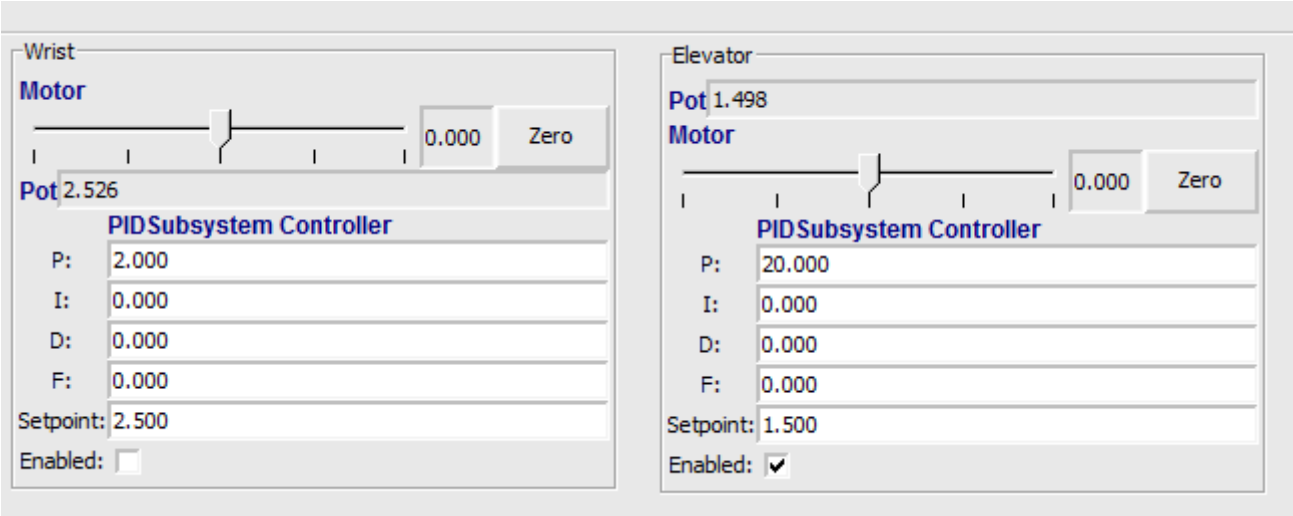
The PID (Proportional, Integral, Differential) is an algorithm for determining the motor speed based on sensor feedback to reach a setpoint as quickly as possible. For example, a robot with an elevator that moves to a predetermined position should move there as fast as possible then stop without excessive overshoot leading to oscillation. Getting the PID controller to behave this way is called "tuning". The idea is to compute an error value that is the difference between the current value of the mechanism feedback element and the desired (setpoint) value. In the case of the arm, there might be a potentiometer connected to an analog channel that provides a voltage that is proportional to the position of the arm. The desired value is the voltage that is predetermined for the position the arm should move to, and the current value is the voltage for the actual position of the arm.

Finding the setpoint values with LiveWindow

Finding the setpoint values with LiveWindow

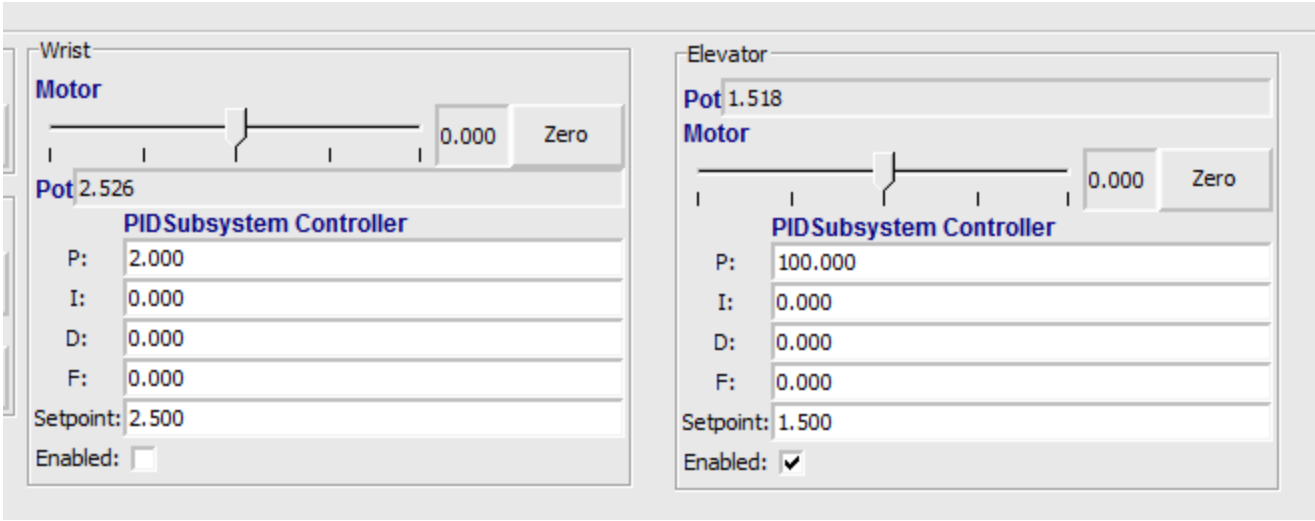
Create a PID Subsystem for each mechanism with feedback. The PID Subsystems contain the actuator (motor) and the feedback sensor (potentiometer in this case). You can use Test mode to display the subsystem sensors and actuators. Using the slider manually adjust the actuator to each desired position. Note the sensor values (2) for each of the desired positions. These will become the setpoints for the PID controller.

Viewing the PIDController in LiveWindow



In Test mode the PID Subsystems display their P, I, and D parameters that are set in the code. The P, I, and D values are the weights applied to the computed error (P), sum of errors over time (I), and the rate of change of errors (D). Each of those terms is multiplied by the weights and added together to form the motor value. Choosing the optimal P, I, and D values can be difficult and requires some amount of experimentation. The Test mode on the robot allows the values to be modified, and the mechanism response observed.

Tuning the PIDController



Tuning the PID controller can be difficult and there are many articles that describe techniques that can be used. It is best to start with the P value first. To try different values fill in a low number for P, enter a setpoint determined earlier in this document, and note how fast the mechanism responds. If it responds too slowly, perhaps never reaching the setpoint, increase P. If it responds too quickly, perhaps oscillating, reduce the P value. Repeat this process until you get a response that is as fast as possible without oscillation. It's possible that having a P term is all that's needed to achieve adequate control of your mechanism.

Once you have determined P, I, and D values they can be inserted into the program. You'll find them either in the properties for the PIDSubsystem in RobotBuilder or in the constructor for the PID Subsystem in your code.

The F (feedforward) term is used for controlling velocity with a PID controller.

You can find more information in Operating the robot with feedback from sensors(C++ or Java).

Practice Tuning in FRCSim

With the FRCSim, you can experiment with PID and practice tuning in simulation! This means no broken couplers or time consuming uploads!

SmartDashboard

A video on tuning PID with smartdashboard in simulation can be found on the official WPILib channel here:

https://www.youtube.com/watch?v=yqD9iHiR3j8

SmartDashboard details

Stale data and SmartDashboard

SmartDashboard uses NetworkTables for communicating values between the robot and the driver station laptop. Network Tables acts as a distributed table of name and value pairs. If a name/value pair is added to either the client (laptop) or server (robot) it is replicated to the other. If a name/value pair is deleted from, say, the robot but the SmartDashboard or OutlineViewer are still running, then when the robot is restarted, the old values will still appear in the SmartDashboard and OutlineViewer because they never stopped running and continue to have those values in their tables. When the robot restarts, those old values will be replicated to the robot.

To ensure that the SmartDashboard and OutlineViewer are showing exactly the same values, it is necessary to restart all of them at the same time. That way, old values that one is holding won't get replicated to the others.

This usually isn't a problem if the program isn't constantly changing, but if the program is in development and the set of keys being added to NetworkTables is constantly changing, then it might be necessary to do the restart of everything to accurately see what is current.

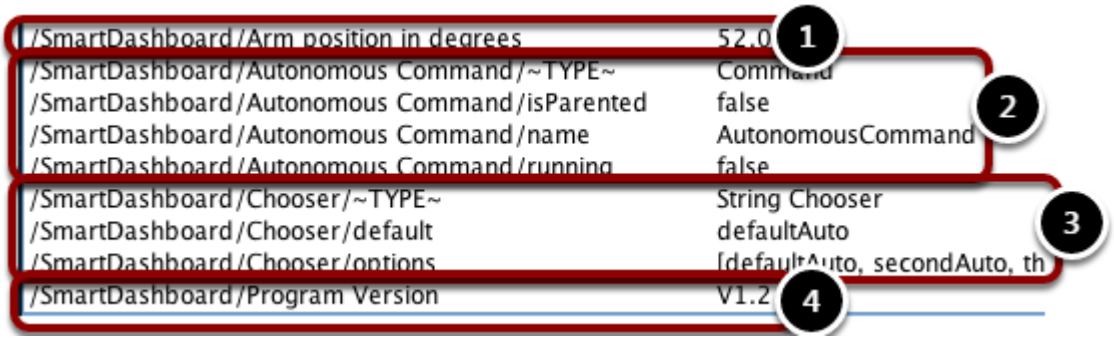
SmartDashboard namespace

SmartDashboard uses NetworkTables to send data between the robot and the Dashboard (Driver Station) computer. NetworkTables sends data as name, value pairs, like a distributed hashtable between the robot and the computer. When a value is changed in one place, its value is automatically updated in the other place. This mechanism and a standard set of name (keys) is how data is displayed on the SmartDashboard.

There is a hierarchical structure in the name space creating a set of tables and subtables. SmartDashboard data is in the SmartDashboard subtable and LiveWindow data is in the LiveWindow subtable as shown below.

For informational purposes the names and values can be displayed using the OutlineViewer application that is installed in the same location as the SmartDashboard. It will display all the NetworkTable keys and values as they are updated.

SmartDashboard data values



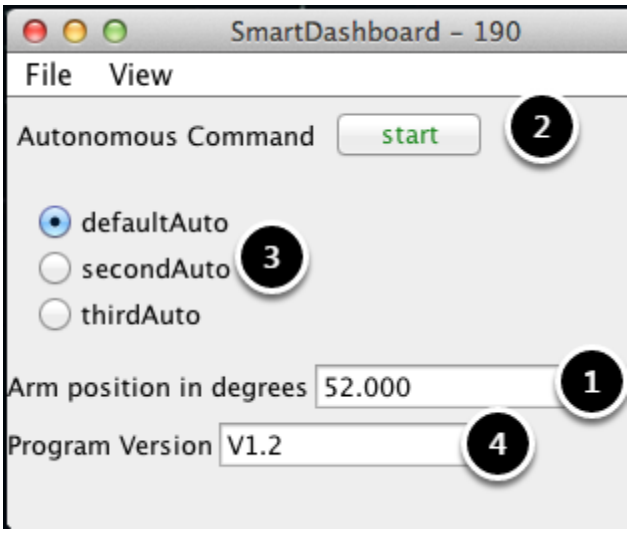
SmartDashboard values are created with key names that begin with "SmartDashboard/". The above values viewed with OutlineViewer correspond to data put to the SmartDashboard with the following statements:

```
        chooser = new SendableChooser();
        chooser.addDefault("defaultAuto", new AutonomousCommand());
```

```
        chooser.addObject("secondAuto", new AutonomousCommand());
        chooser.addObject("thirdAuto", new AutonomousCommand());
        SmartDashboard.putData("Chooser", chooser);
        SmartDashboard.putNumber("Arm position in degrees", 52.0);
        SmartDashboard.putString("Program Version", "V1.2");
```

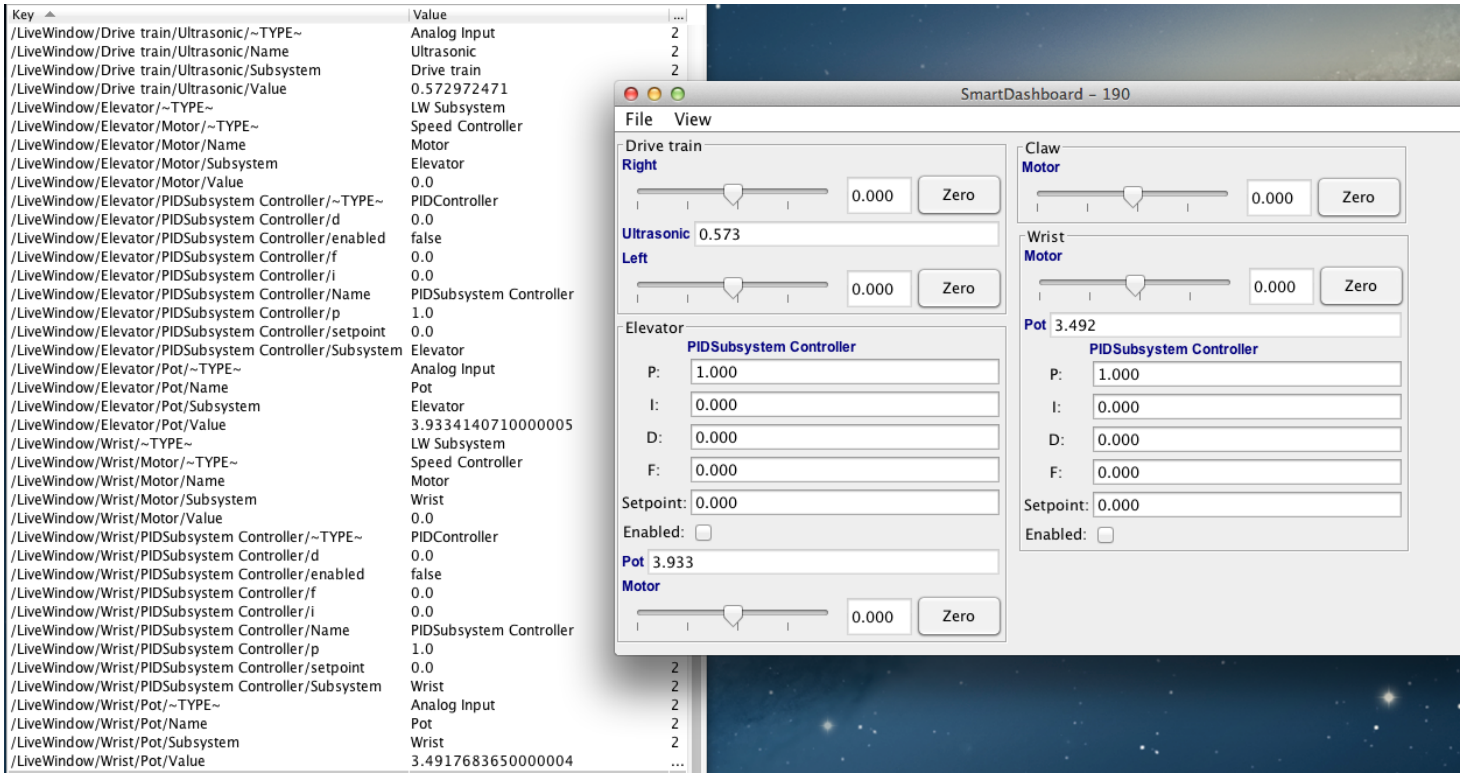
The "Arm position" is created with the putNumber() call. The AutonomousCommand is written with a putData("Autonomous Command", command) that is not shown in the above code fragement. The chooser is created as a SendableChooser object and the string value, "Program Version" is created with the putString() call.

View of the SmartDashboard



The code from the previous step generates the table values as shown and the SmartDashboard display as shown here. The numbers correspond to the NetworkTable variables shown in the previous step.

LiveWindow data values



LiveWindow data is automatically grouped by subsystem. The data is viewable in the SmartDashboard when the robot is in Test mode (set on the Driver Station). If you are not writing a command based program, you can still cause sensors and actuators to be grouped for easy viewing by specifying the subsystem name. In the above display you can see the key names and the resultant output in Test mode on the SmartDashboard. All the strings start with "/LiveWindow" then the Subsystem name, then a group of values that are used to display each element. The code that generates this LiveWindow display is shown below:

```
drivetrainLeft = new Talon(1, 2);
LiveWindow.addActuator("Drive train", "Left", (Talon) drivetrainLeft);
drivetrainRight = new Talon(1, 1);
     LiveWindow.addActuator("Drive train", "Right", (Talon) drivetrainRight);
drivetrainRobotDrive = new RobotDrive(drivetrainLeft, drivetrainRight);
drivetrainRobotDrive.setSafetyEnabled(false);
drivetrainRobotDrive.setExpiration(0.1);
```

```
        drivetrainRobotDrive.setSensitivity(0.5);
        drivetrainRobotDrive.setMaxOutput(1.0);
        drivetrainUltrasonic = new AnalogChannel(1, 3);
            LiveWindow.addSensor("Drive train", "Ultrasonic", drivetrainUltrasonic);
        elevatorMotor = new Victor(1, 6);
            LiveWindow.addActuator("Elevator", "Motor", (Victor) elevatorMotor);
        elevatorPot = new AnalogChannel(1, 4);
            LiveWindow.addSensor("Elevator", "Pot", elevatorPot);
        wristPot = new AnalogChannel(1, 2);
            LiveWindow.addSensor("Wrist", "Pot", wristPot);
        wristMotor = new Victor(1, 3);
            LiveWindow.addActuator("Wrist", "Motor", (Victor) wristMotor);
        clawMotor = new Victor(1, 5);
            LiveWindow.addActuator("Claw", "Motor", (Victor) clawMotor);
```

Values that correspond to actuators are not only displayed, but can be set using sliders created in the SmartDashboard in Test mode.