

SMARTDASHBOARD

Table of Contents

SmartDashboard.....4

- Getting Started with the SmartDashboard.....5
- Displaying Expressions from Within the Robot Program 10
- Changing the display properties of a value 11
- Changing the display widget type for a value 15
- Testing commands 17
- Choosing an autonomous program from SmartDashboard..... 19
- Displaying the status of Commands and Subsystems 22
- Setting robot preferences from SmartDashboard 26
- Verifying SmartDashboard is working..... 29

SmartDashboard 2.0 (SFX)..... 33

- The new SmartDashboard (SFX) 34
- Setting SFX to Launch with the DS 46
- Creating a custom control using FXML 49
- Creating a custom control using Java..... 54

Test mode and LiveWindow 60

- Enabling Test mode (LiveWindow)..... 61
- Displaying LiveWindow values 63
- PID Tuning with SmartDashboard 65

SmartDashboard details 68

- Stale data and SmartDashboard..... 69

SmartDashboard

SmartDashboard namespace	70
Using the SmartDashboard Vision installer.....	74
Viewing the RoboRealm output in SmartDashboard	75

SmartDashboard

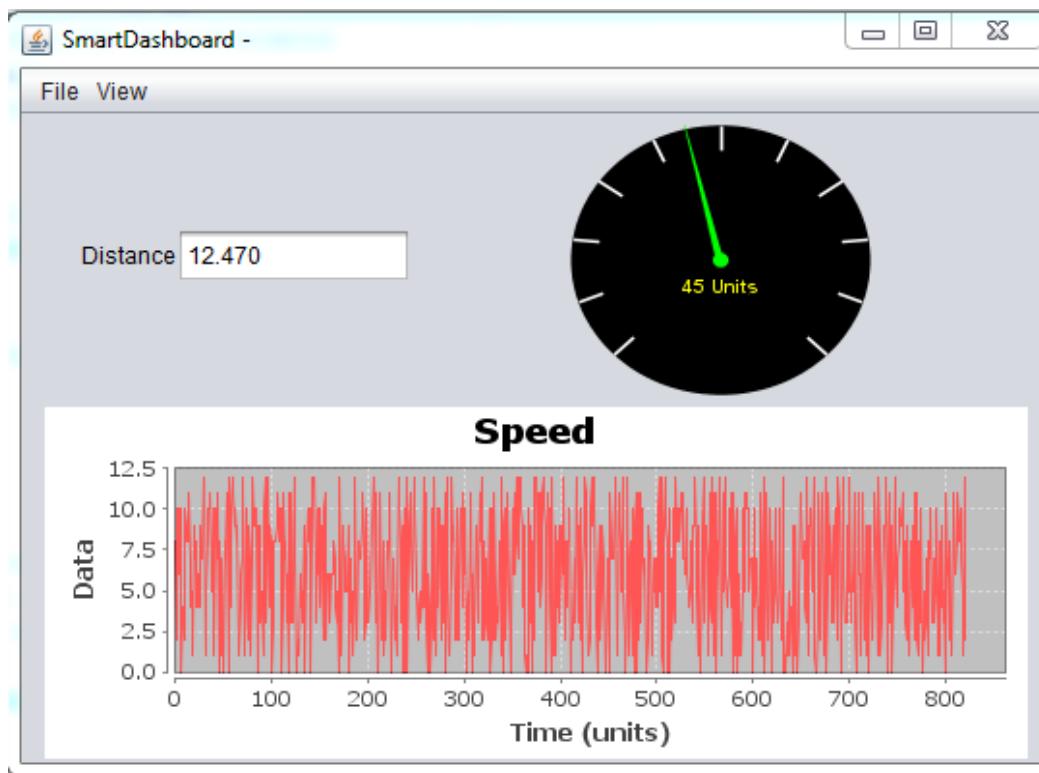
Getting Started with the SmartDashboard

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:

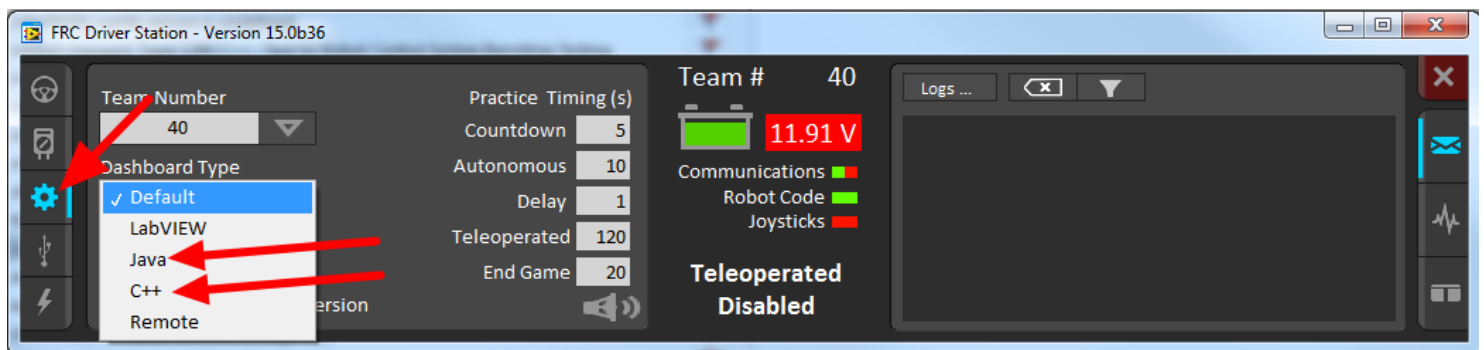
- Displays robot data of your choice while the program is running. It can be displayed as simple text fields or more elaborately in many other display types like graphs, dials, etc.
- Displays the state of the robot program such as the currently executing commands and the status of any subsystems

SmartDashboard

- 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



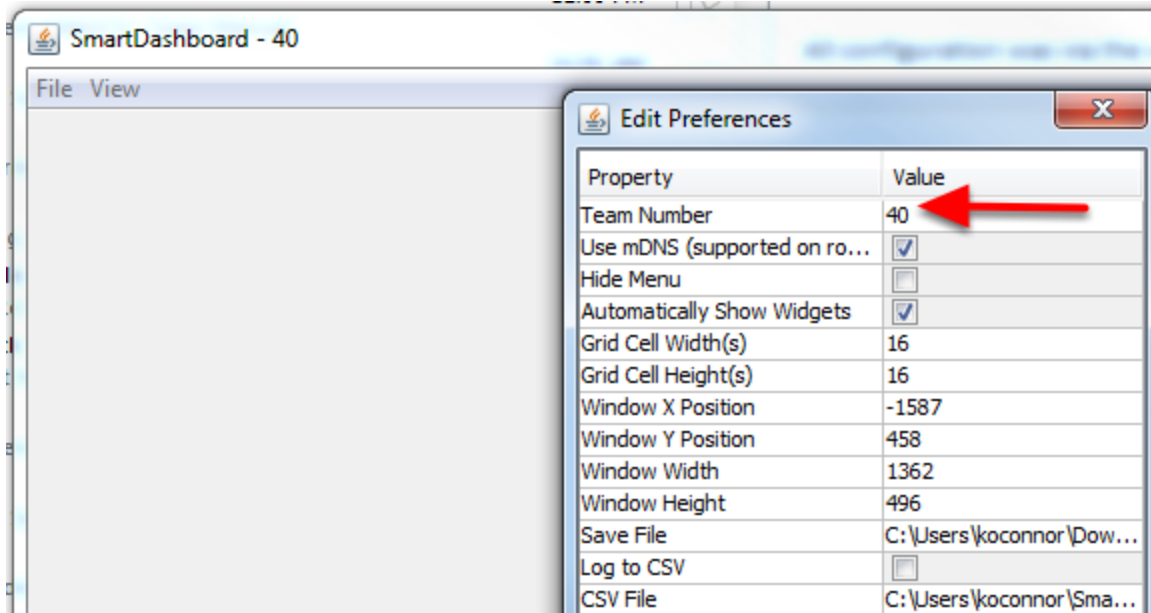
The SmartDashboard is now packaged with the C++ and Java language updates and can be launched directly from the Driver Station by selecting the Java or C++ buttons on the Setup tab.

Note: The 2015 Kickoff release of the Driver Station contains a bug which prevents this setting from sticking across restarts of the DS Software. See [this article](#) for details on how to get the DS to launch the SmartDashboard when it starts.

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.

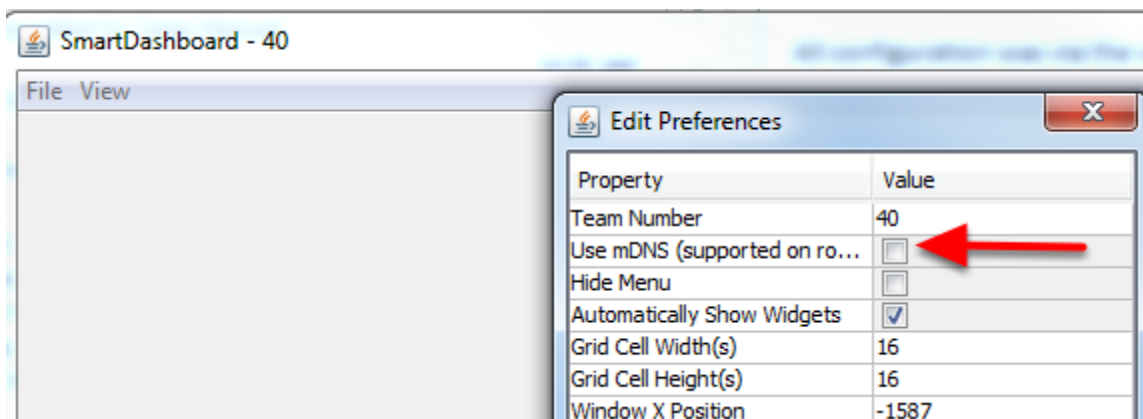
SmartDashboard

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.

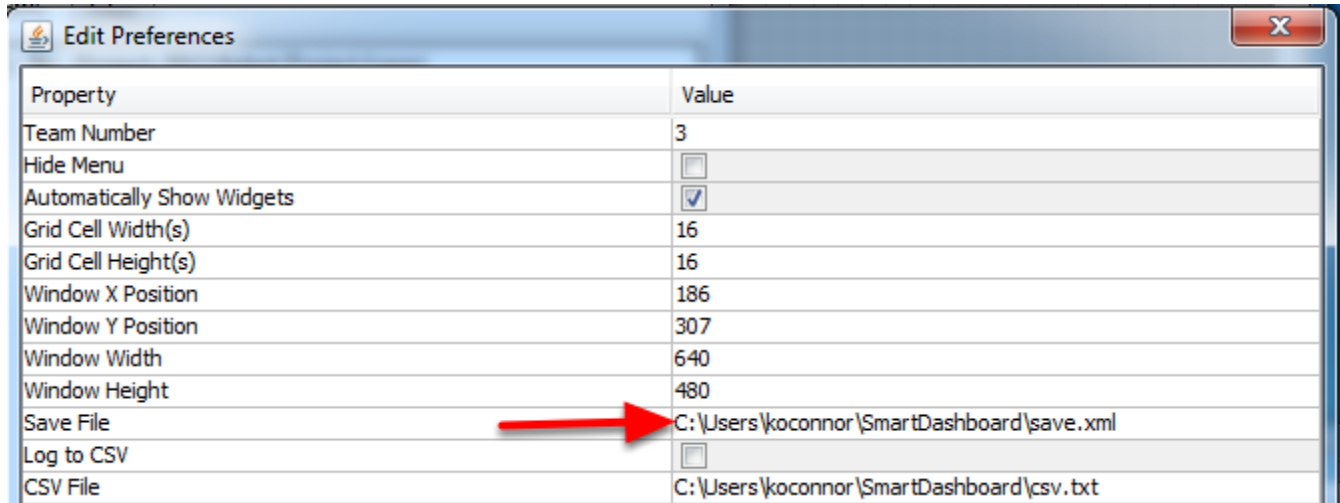
mDNS



By default the SmartDashboard is configured to use mDNS to construct the address (roboRIO-TEAM.local) for use with the roboRIO system. To instead attempt to connect to an IP of the form 10.TE.AM.2 click File -> Preferences and uncheck the Use mDNS box.

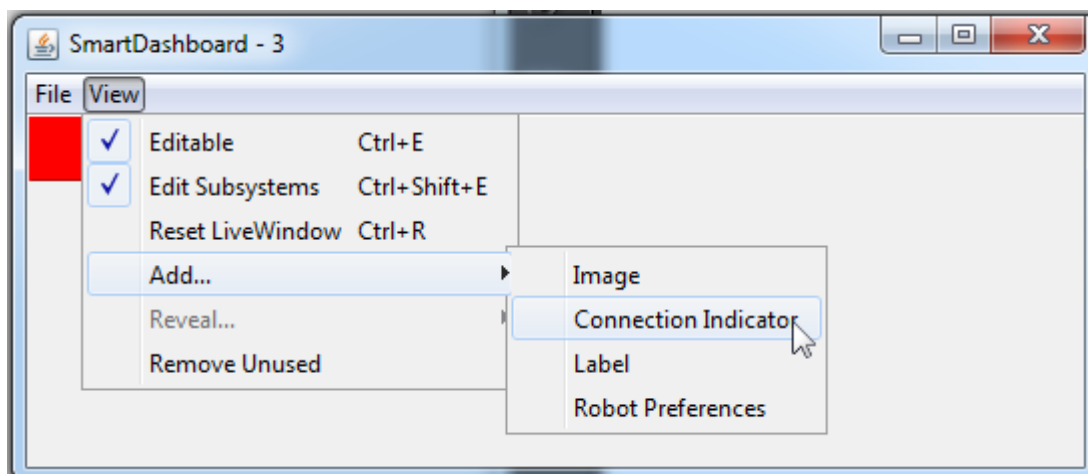
SmartDashboard

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.

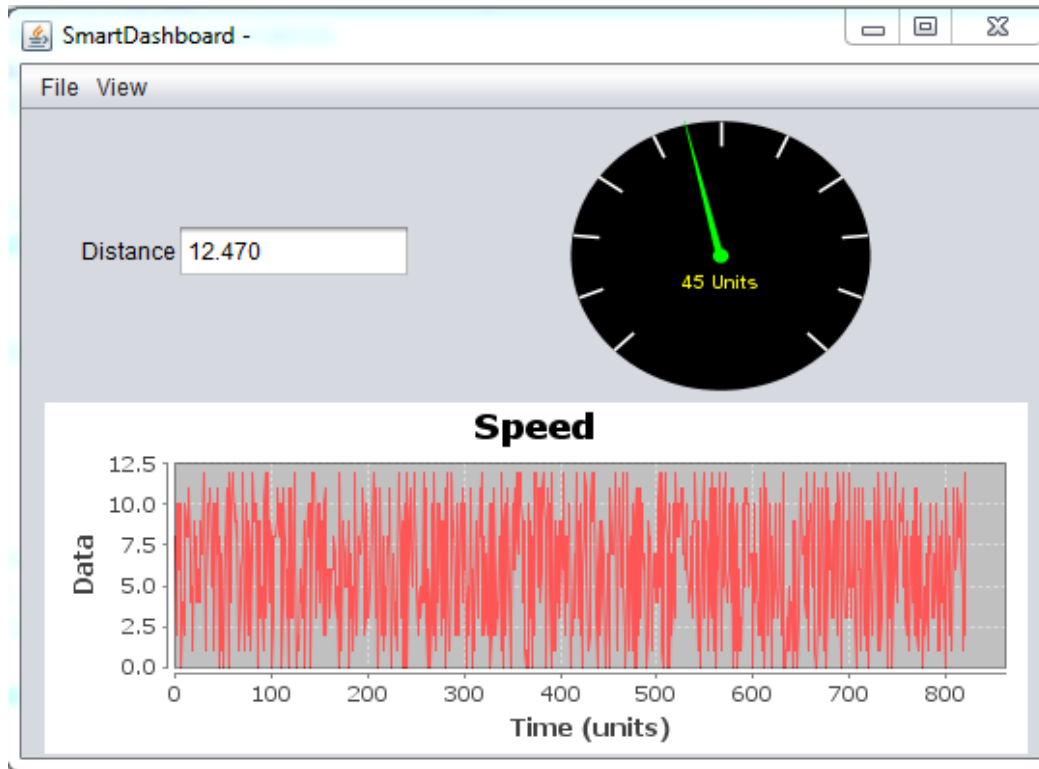
Adding a Connection Indicator



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.

SmartDashboard

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.

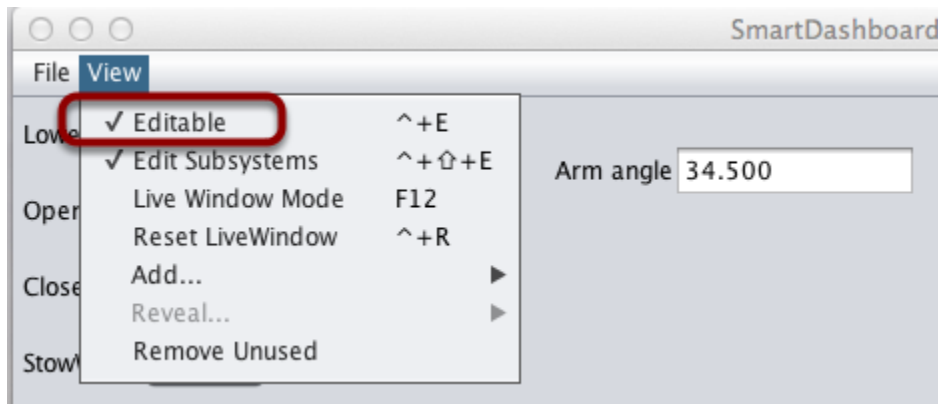
Creating widgets on the SmartDashboard

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

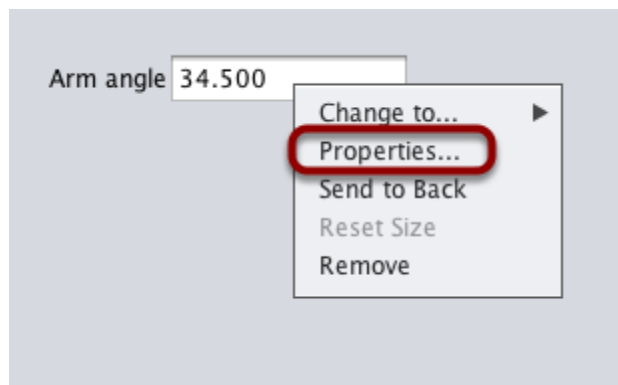
Each value displayed with SmartDashboard has a set of properties that effect the way it's displayed.

Setting the SmartDashboard display into editing mode



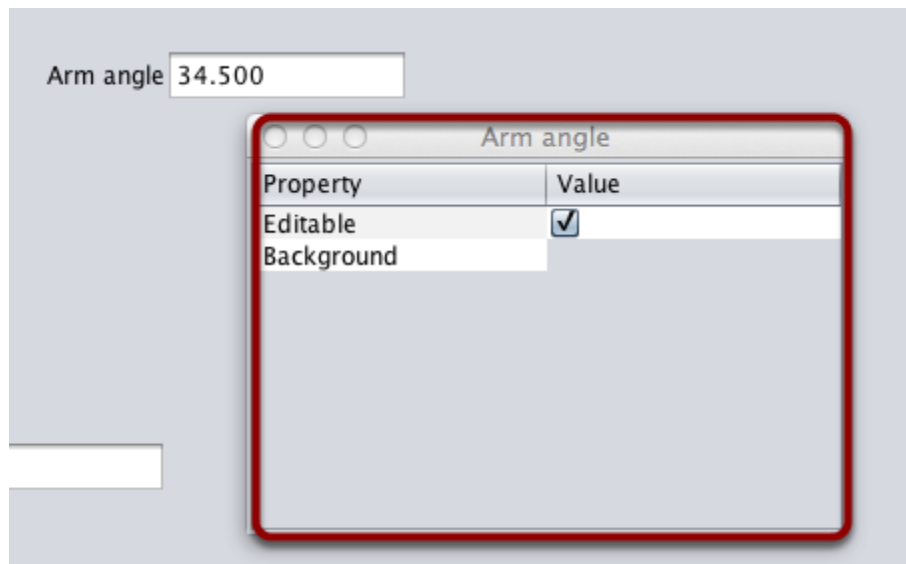
The SmartDashboard has two modes it can operate in, display mode and edit mode. In edit mode you can move around widgets on the screen and edit their properties. To put the SmartDashboard into edit mode, click the "View" menu, then select "Editable" to turn on edit mode.

Getting the properties editor for a widget



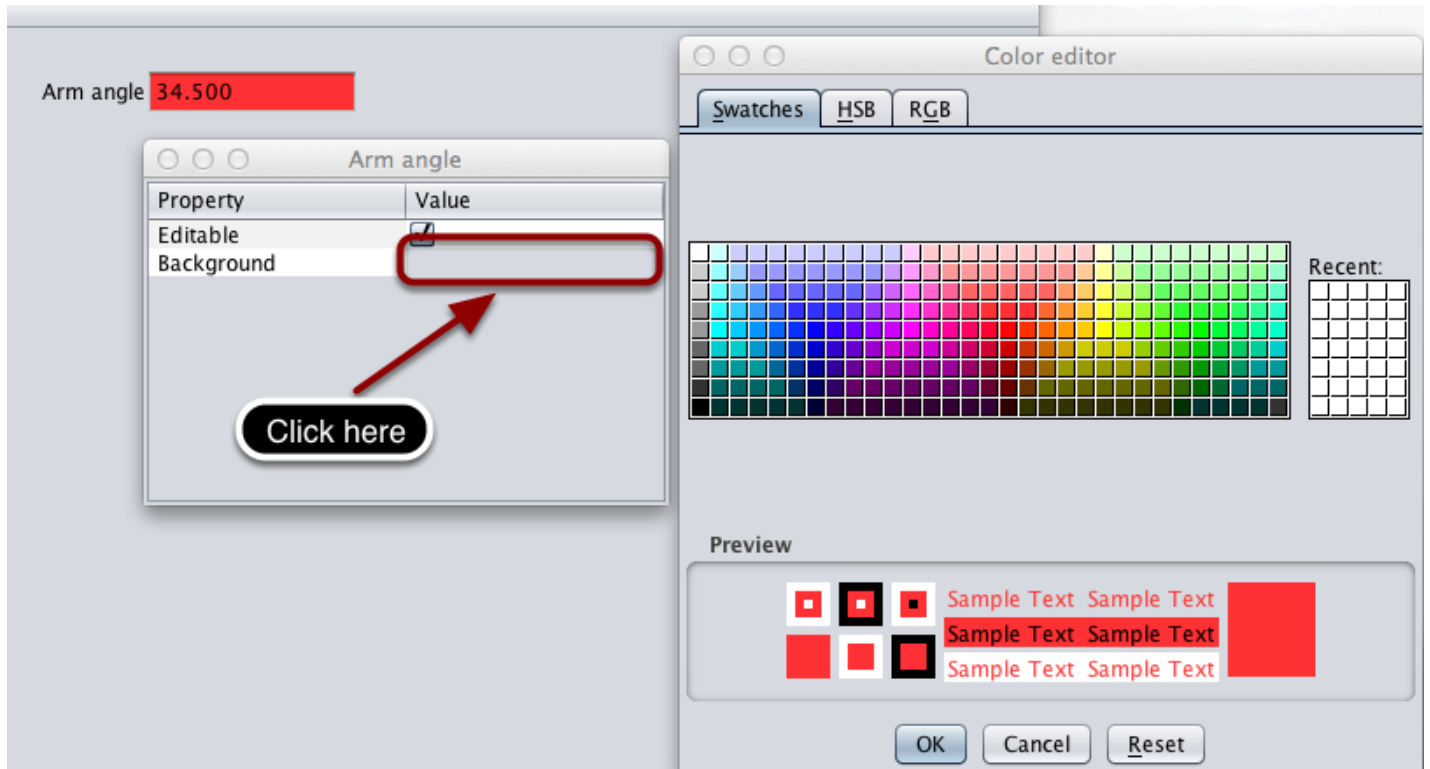
Once in edit mode, you can display and edit the properties for a widget. Right-click on the widget and select "Properties...".

Editing the properties on a field



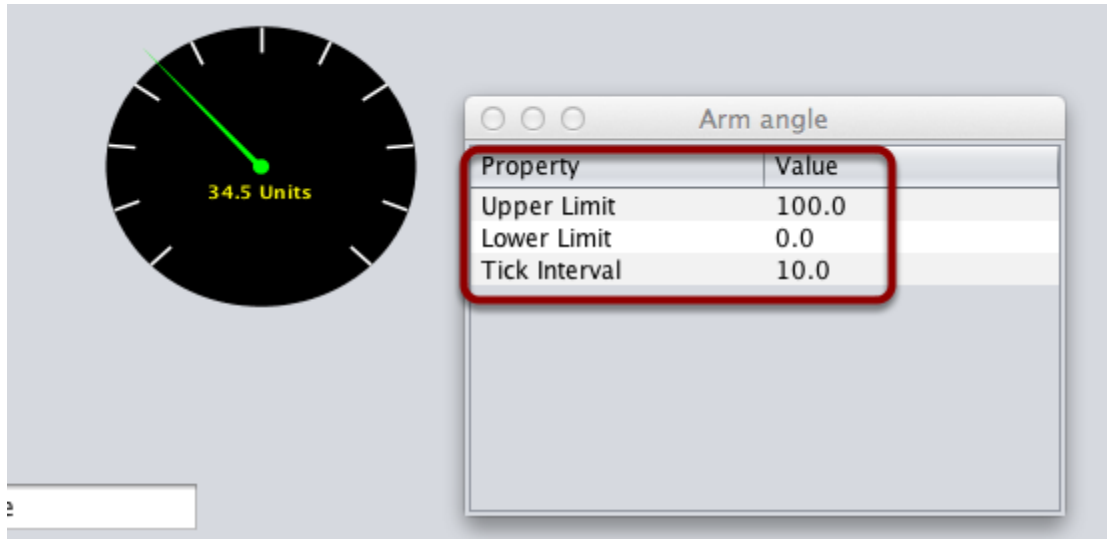
A dialog box will be shown in response to the "Properties..." menu item on the widgets right-click context menu.

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

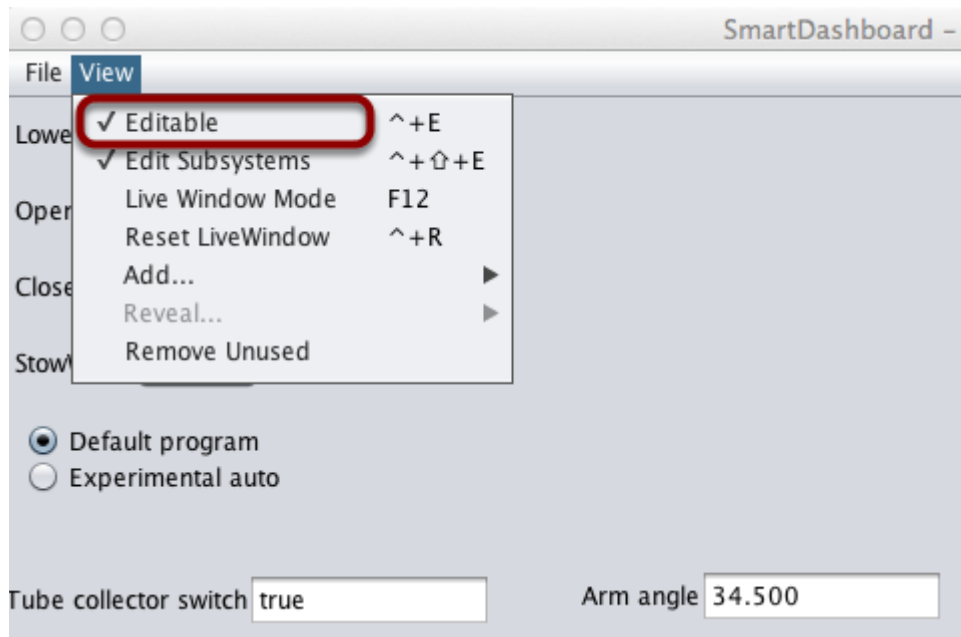


Different widget types have different sets of editable properties to change the appearance. In this example, the upper and lower limits of the dial and the tick interval are changeable parameters.

Changing the display widget type for a value

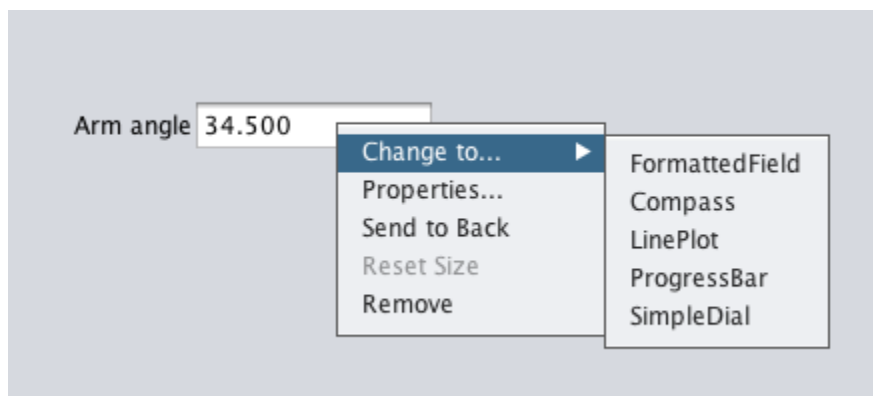
You can change the type of widget that displays values with the SmartDashboard. The allowable widgets depend on the type of the value being displayed.

Set edit mode



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

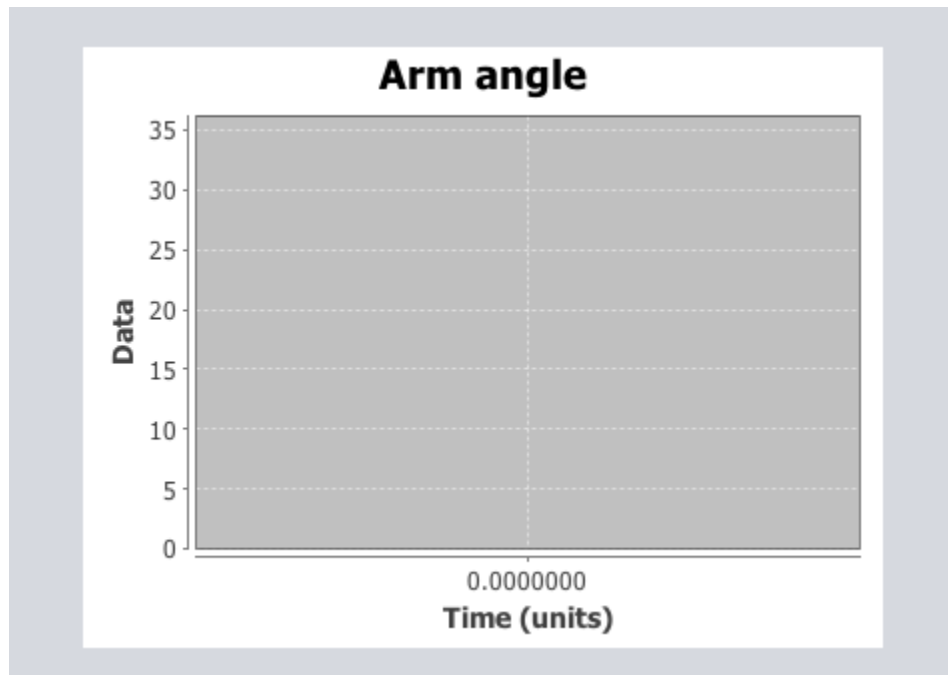
Choose the new widget type



SmartDashboard

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

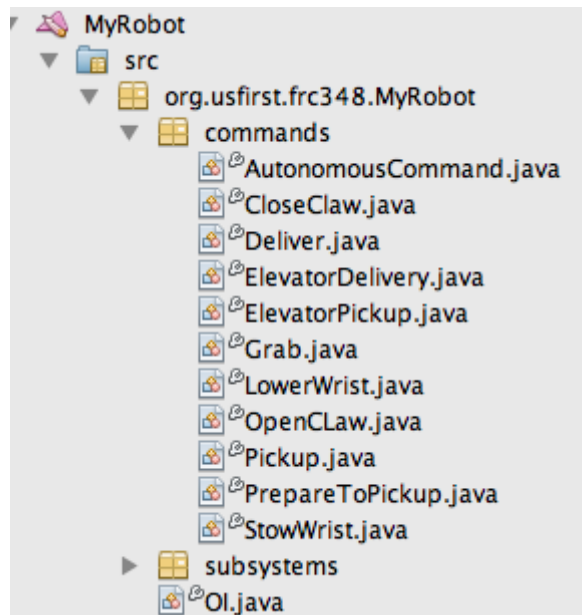


The new widget type is displayed. In this case, a Line Plot, will show the values of the Arm angle value over time. You can set the properties of the graph to make it better fit your data by right-clicking and selecting "Properties...". See: [Changing the display properties of a value](#).

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.

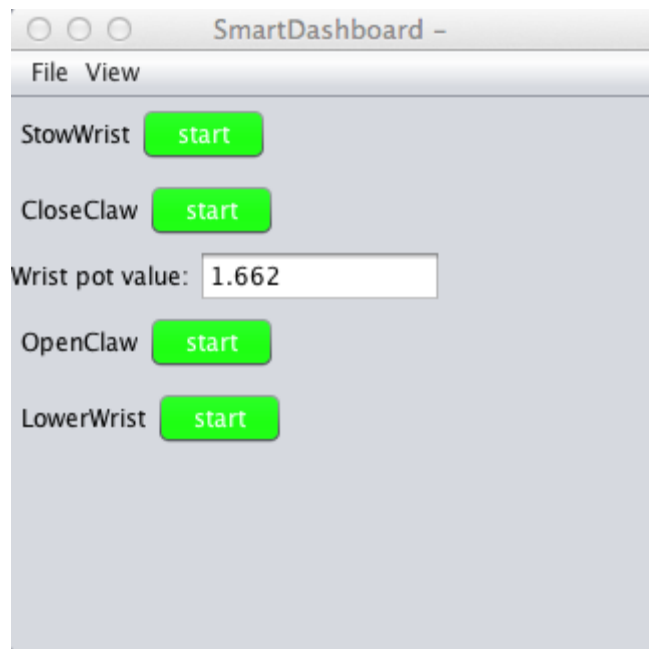
SmartDashboard

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



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

```
public class Robot extends IterativeRobot {  
    Command autonomousCommand;  
    SendableChooser autoChooser;  
  
    public static OI oi;
```

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

```
48  
49 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS  
50  
51  
52 1 autoChooser = new SendableChooser();  
53   autoChooser.addDefault("Default program", new Pickup());  
54   autoChooser.addObject("Experimental auto", new ElevatorPickup());  
55   SmartDashboard.putData("Autonomous mode chooser", autoChooser);  
56 }  
57  
58  
59 2 public void autonomousInit() {  
60     autonomousCommand = (Command) autoChooser.getSelected();  
61     autonomousCommand.start();  
62 }
```

SmartDashboard

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

```
63 |  
64 |  
65 |  
66 |  
67 |  
68 |
```

```
/**  
 * This function is called periodically during autonomous  
 */  
public void autonomousPeriodic() {  
    Scheduler.getInstance().run();  
}
```

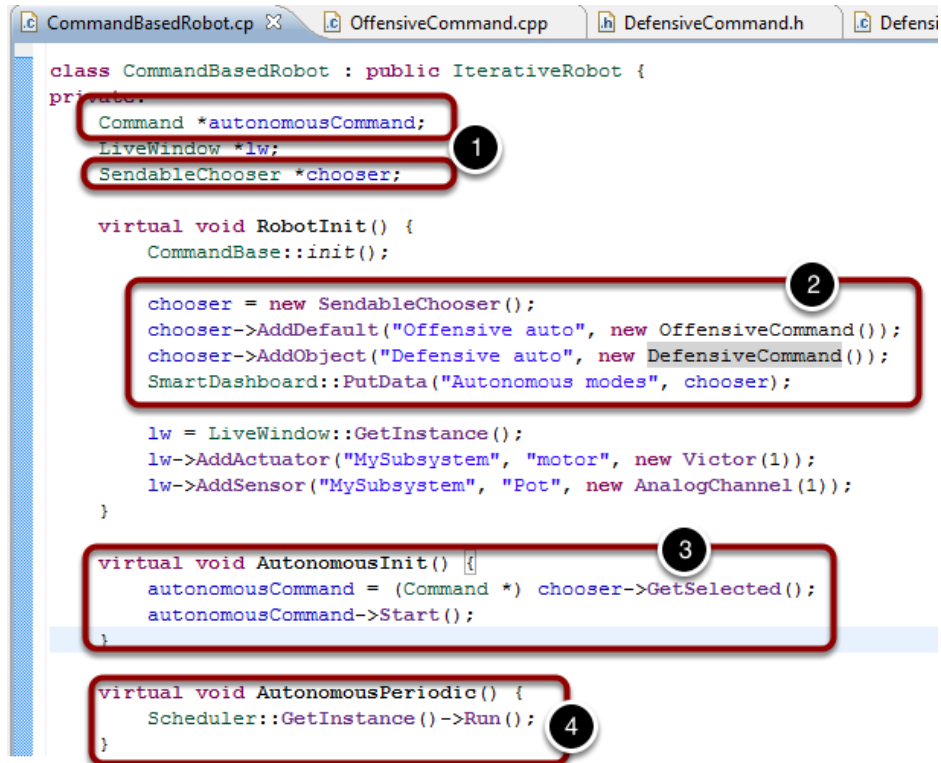
RobotBuilder will generate code automatically that runs the scheduler every driver station update period (about every 20ms). This will cause the selected autonomous command to run.

SmartDashboard 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++



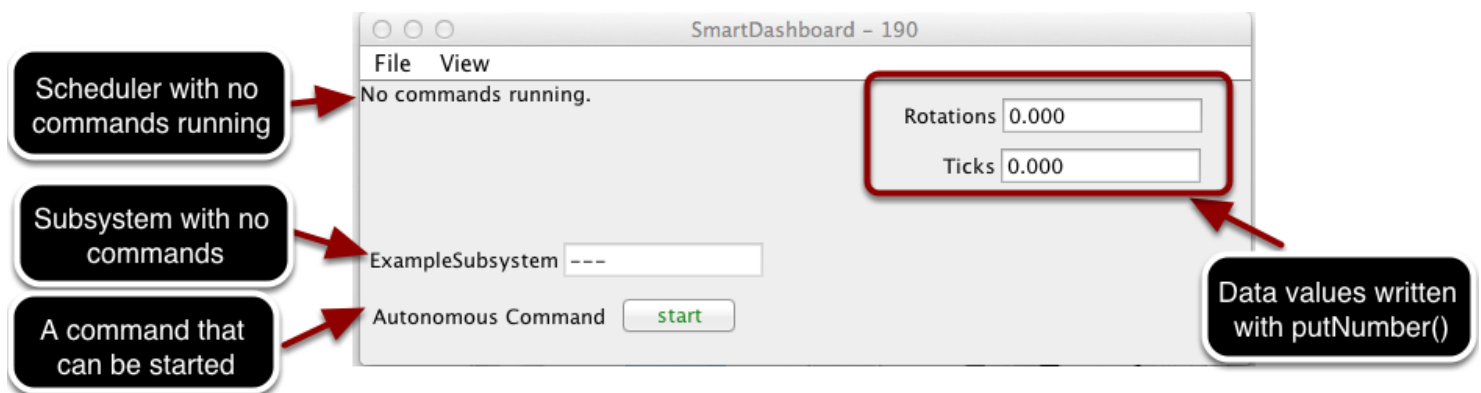
This is an example of creating a SendableChooser object and using it to select between a Defensive and Offensive autonomous command to run when the autonomous period of the match starts. Just as in the Java example:

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 chosen 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



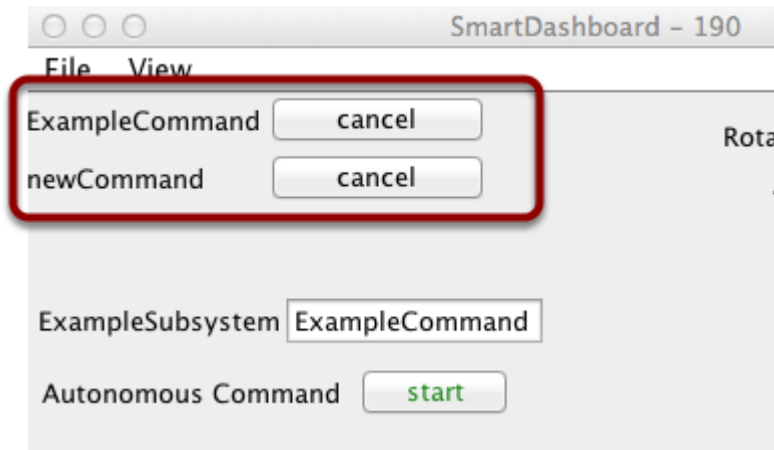
With SmartDashboard you can display the status of the commands and subsystems in your robot program in various ways. The outputs should significantly reduce the debugging time for your programs. In this picture you can see a number of displays that are possible. Displayed here are:

- 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.

SmartDashboard

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.

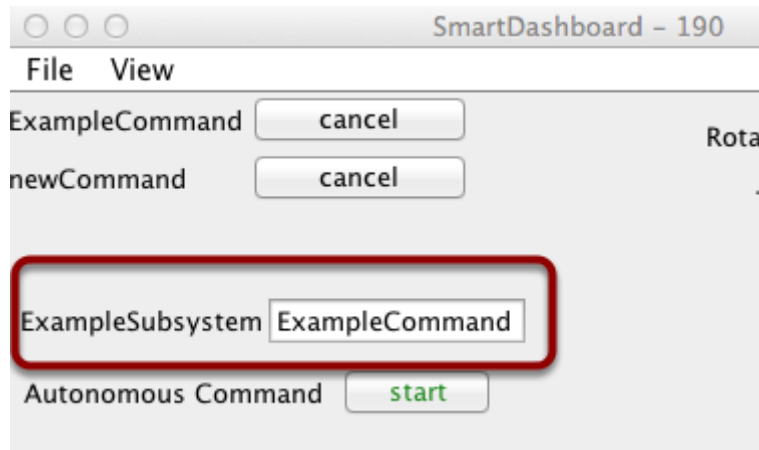
Displaying the Scheduler status

```
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.

SmartDashboard

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".

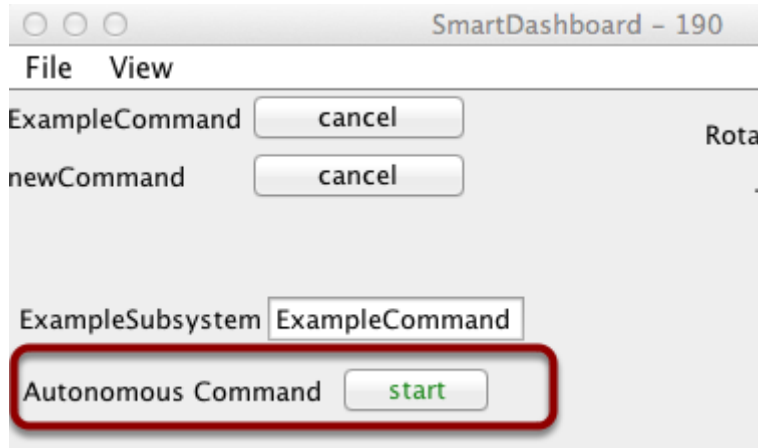
Writing the code to display a subsystem

```
7
8
9
10 public abstract class CommandBase extends Command {
11
12     public static OI oi;
13     // Create a single static instance of all of your subsystems
14     public static ExampleSubsystem exampleSubsystem = new ExampleSubsystem();
15
16     public static void init() {
17         oi = new OI();
18         SmartDashboard.putData(exampleSubsystem);
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.

SmartDashboard

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  
18  
19 public void robotInit() {  
20     autonomousCommand = new ExampleCommand();  
21  
22     CommandBase.init();  
23     SmartDashboard.putData(Scheduler.getInstance());  
24     SmartDashboard.putData("Autonomous Command", autonomousCommand);  
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;

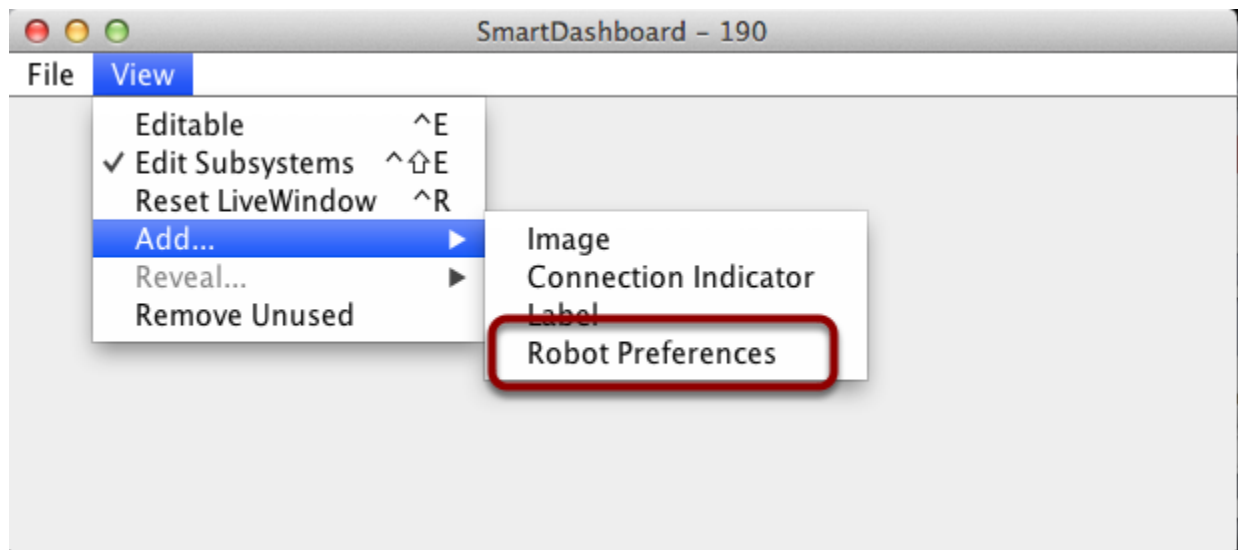
    public void robotInit() {
        prefs = Preferences.getInstance();
        armUpPosition = prefs.getDouble("ArmUpPosition", 1.0);
        armDownPosition = prefs.getDouble("ArmDownPosition", 4.);
    }
}
```

SmartDashboard

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.

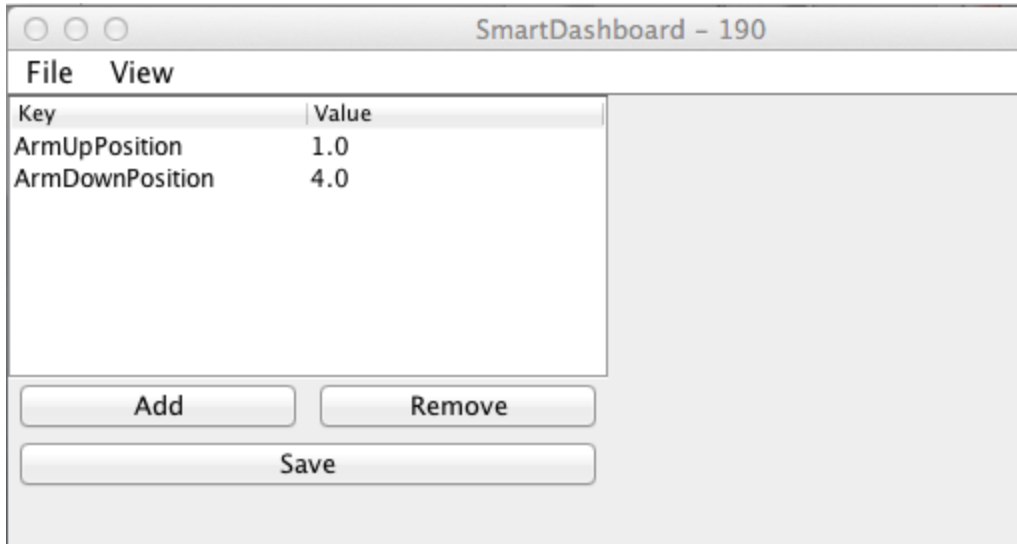
Displaying the Preferences widget in SmartDashboard



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.

SmartDashboard

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

```
1 package org.usfirst.frc.team40.robot;
2
3
4 import edu.wpi.first.wpilibj.SampleRobot;
5 import edu.wpi.first.wpilibj.Timer;
6 import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
7
8
9 public class Robot extends SampleRobot {
10
11     public void operatorControl() {
12         double counter = 0.0;
13         while (isOperatorControl() && isEnabled()) {
14             SmartDashboard.putNumber("Counter", counter++);
15             Timer.delay(0.10);
16         }
17     }
18 }
```

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.

SmartDashboard

Minimal C++ robot program

```
#include "WPIlib.h"

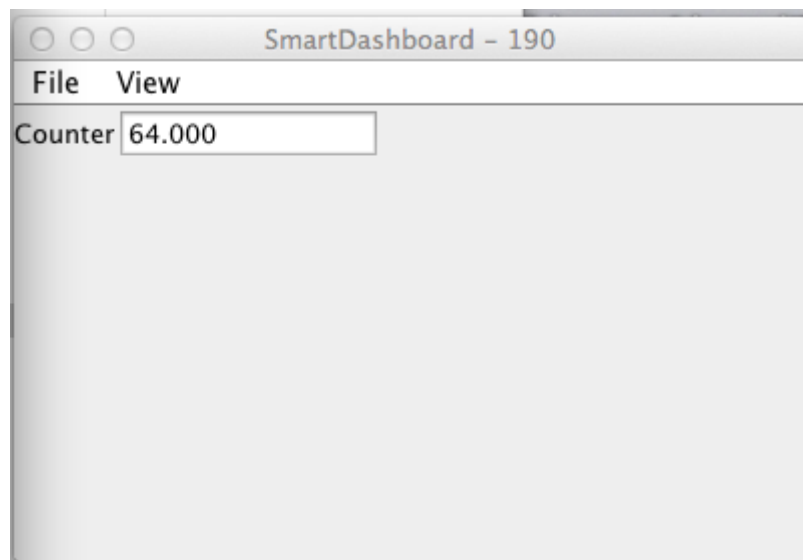
class Robot: public SampleRobot
{
public:
    Robot() {
    }

    void OperatorControl()
    {
        float counter = 0.0;
        while (IsOperatorControl() && IsEnabled()) {
            SmartDashboard::PutNumber("Counter", counter++);
            Wait(0.10);
        }
    }
};

START_ROBOT_CLASS(Robot);
```

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.

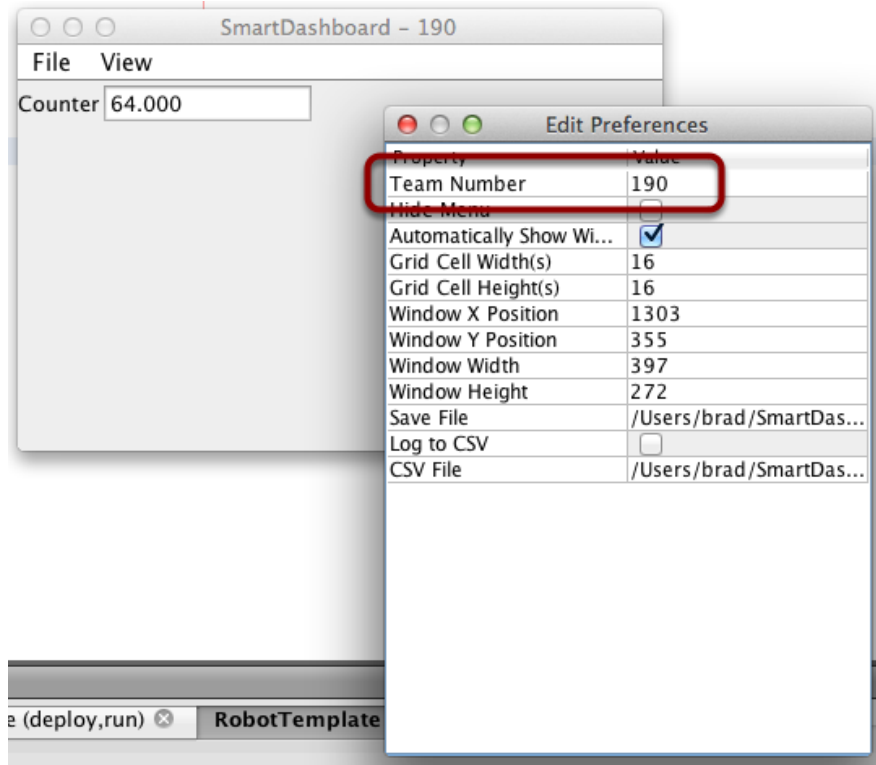
SmartDashboard output for the sample program



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.

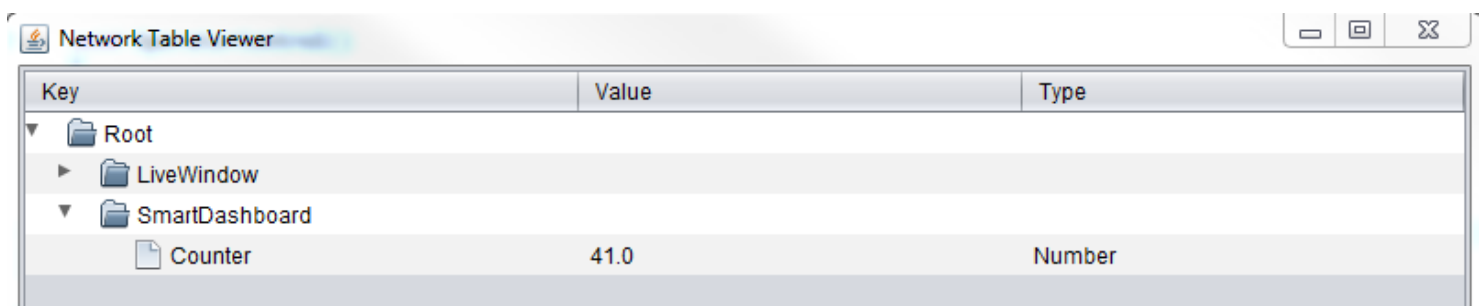
SmartDashboard

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



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"

SmartDashboard

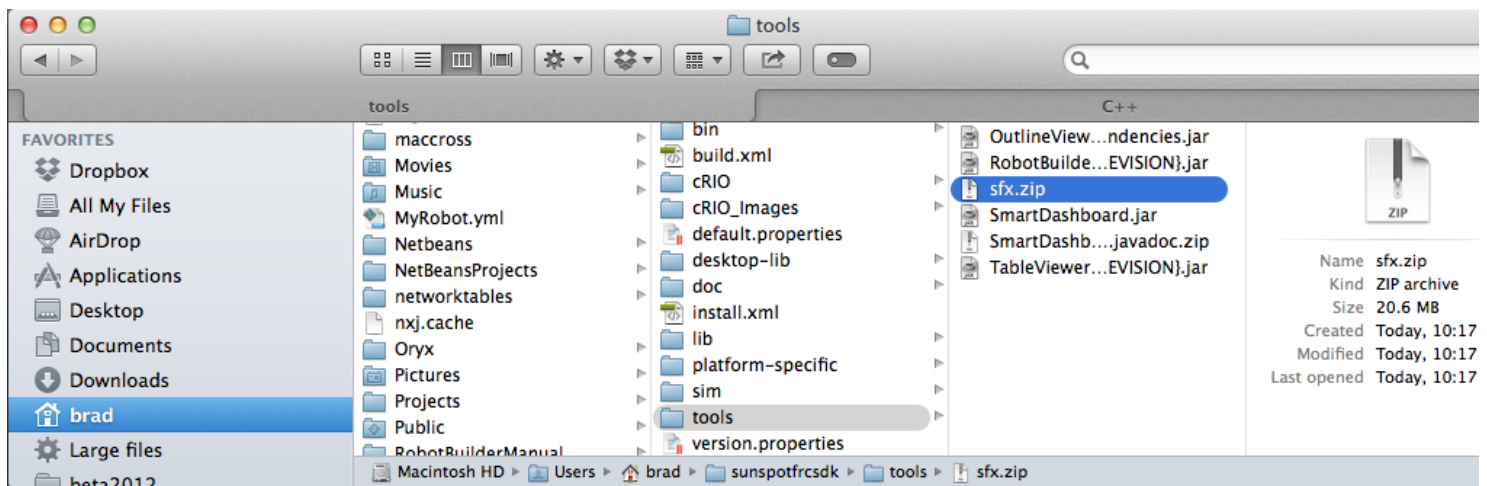
Look at the second row in the table, the value "SmartDashboard/Counter" is the variable written to the SmartDashboard via NetworkTables. As the program runs you should see the value increasing (41.0 in this case). If you don't see this variable in the OutlineViewer then you should look for something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

SmartDashboard 2.0 (SFX)

The new SmartDashboard (SFX)

We have a new SmartDashboard that uses the more modern JavaFX for it's user interface. This has the promise of much richer user interfaces since it allows the widgets to have style sheets applied to them and the library is much newer. The new dashboard requires a current version of Oracle Java 7 (minimum release 7u6) to be installed on your system to get JavaFX.

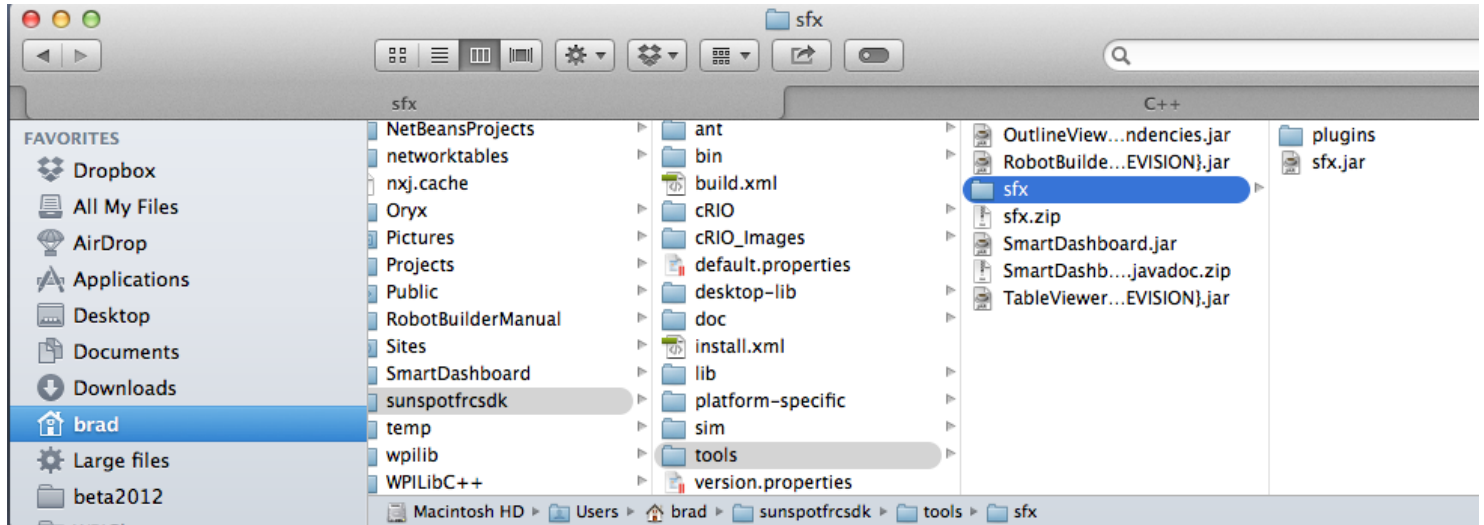
Installing SFX



The new dashboard is shipped as a zip file called sfx.zip. You must first unzip the file in <your-home-directory>/wpilib/tools as shown in here. If the sfx folder already exists from a previous install, delete it before unzipping the newer version.

SmartDashboard

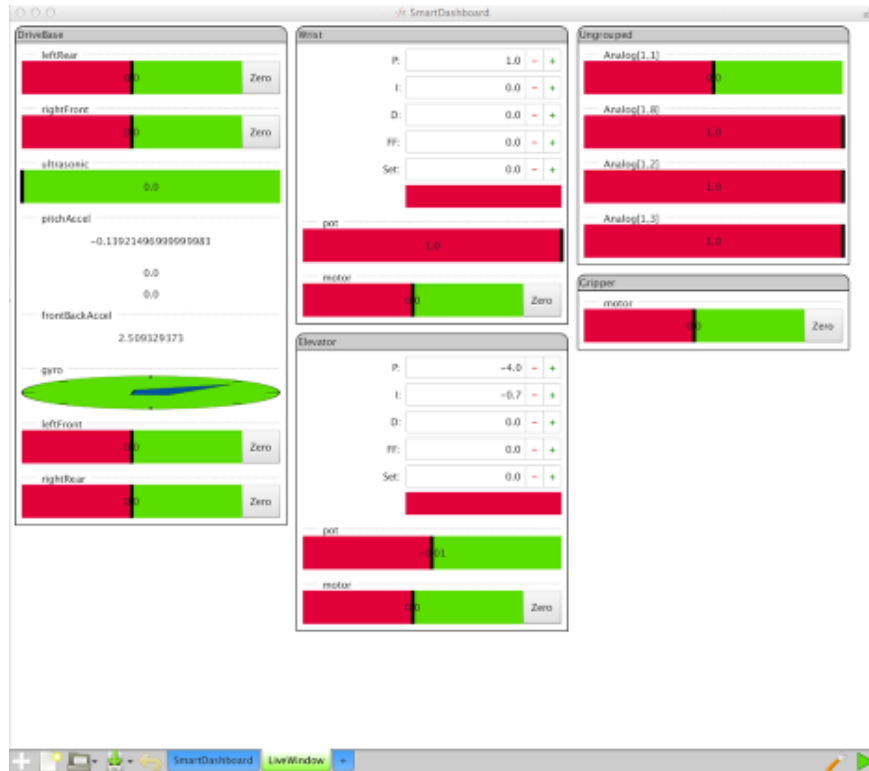
Running SFX



After unzipping the dashboard, you run it by double-clicking on sfx.jar. If the .jar file extension isn't defined on your system (depending on the platform and the java installation) it can also be run with command line, "java -jar sfx.jar".

SmartDashboard

The SFX user interface



SFX will start up and automatically display any values written to the dashboard in Autonomous or Teleoperated modes and will display subsystems or individual controls in Test mode. In the above example the robot is in Test mode and there are a number of subsystems shown (DriveBase, Wrist, Elevator, and Gripper) as well as a number of Ungrouped analog inputs that are allocated but not in any subsystem.

SmartDashboard

SFX controls

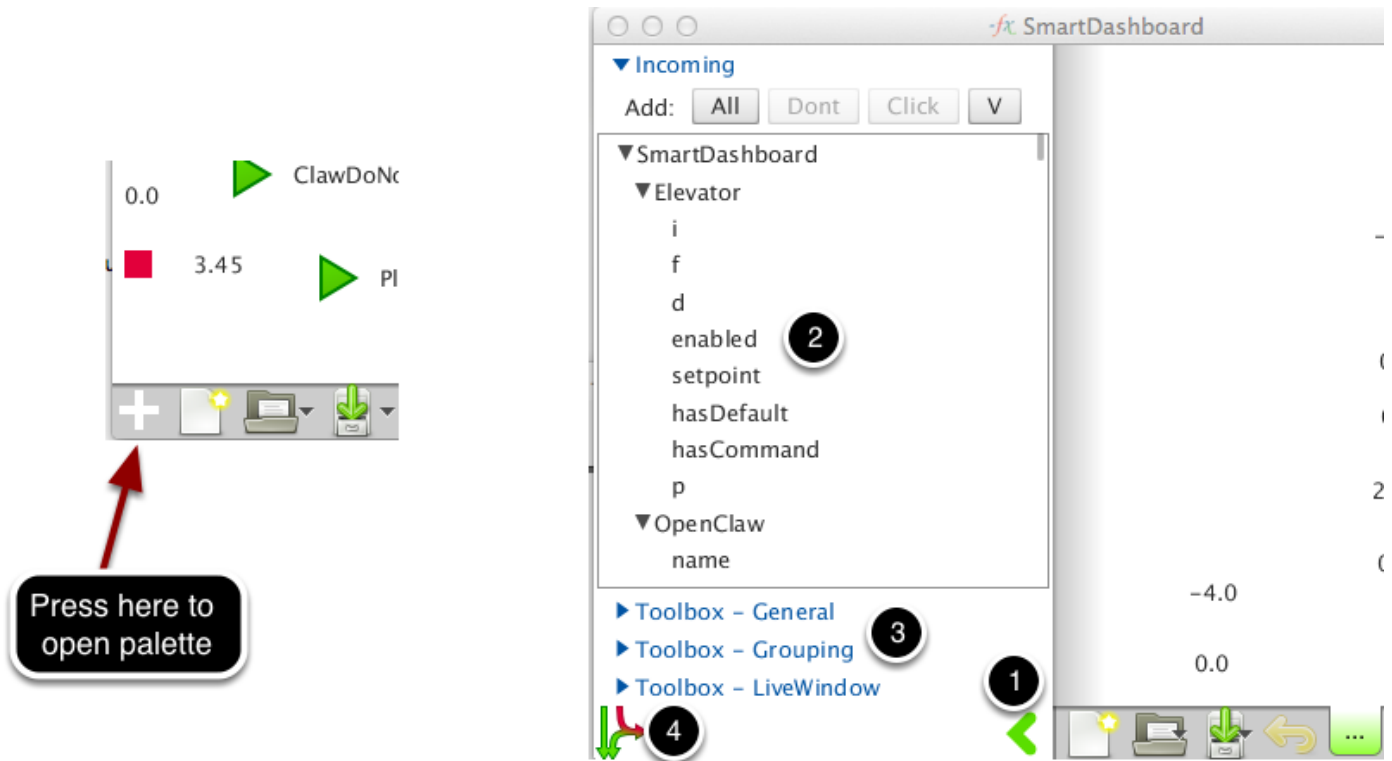


The user interface consists of a number of controls:

1. These controls are for opening the widget palette, creating a new layout, and saving the layout.
2. Tabbed windows are predefined for SmartDashboard and LiveWindow displays. You can also create your own tabs by pressing the plus (+) button. In the future you'll be able to route widgets to each of the windows through pattern matching of the name string for the corresponding values.
3. Settings and the Play button (takes the dashboard in and out of edit mode).

SmartDashboard

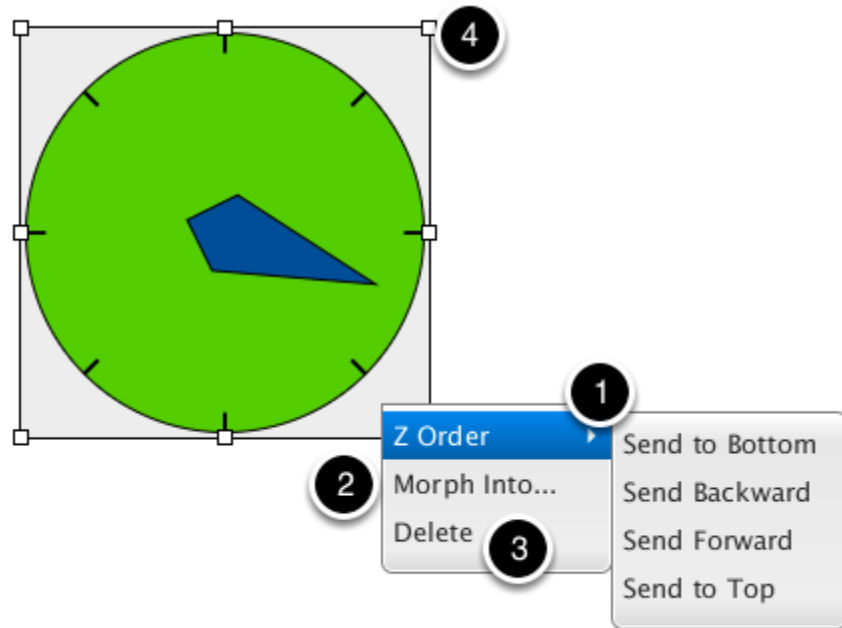
The palette



The palette area is accessed by pressing the plus (+) button and closed with the close button (1). There is a hierarchical list of values (2) that come from the data source (the running program SmartDashboard class methods) that can be dragged into onto the screen. Doing this will create a widget for that value. In addition, you can place widgets on the dashboard and associate them with values later by selecting from the toolbox sections (3).

The data source button (4) opens a dialog that allows you set the IP address for the robot as well as map data sources to the dashboard. This is currently unsupported and will be more completely implemented in a future release.

Editing currently displayed values in the dashboard

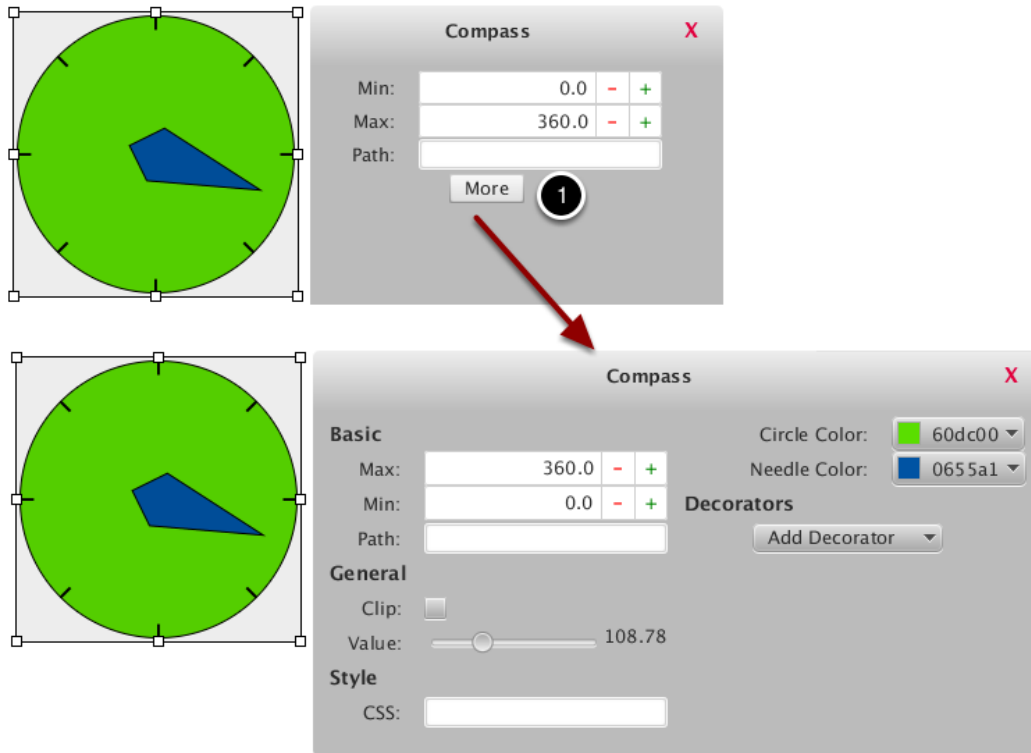


Right-clicking on a value brings up a context menu with a number of options for the value being displayed. You can:

1. Change the stacking order of the widget with respect to other widgets above or below this one.
2. Change this widget into another type of widget (morph) that can display the same value with a different graphical representation.
3. Delete the widget from the layout. Widgets deleted can be added back in later by selecting them from the palette.

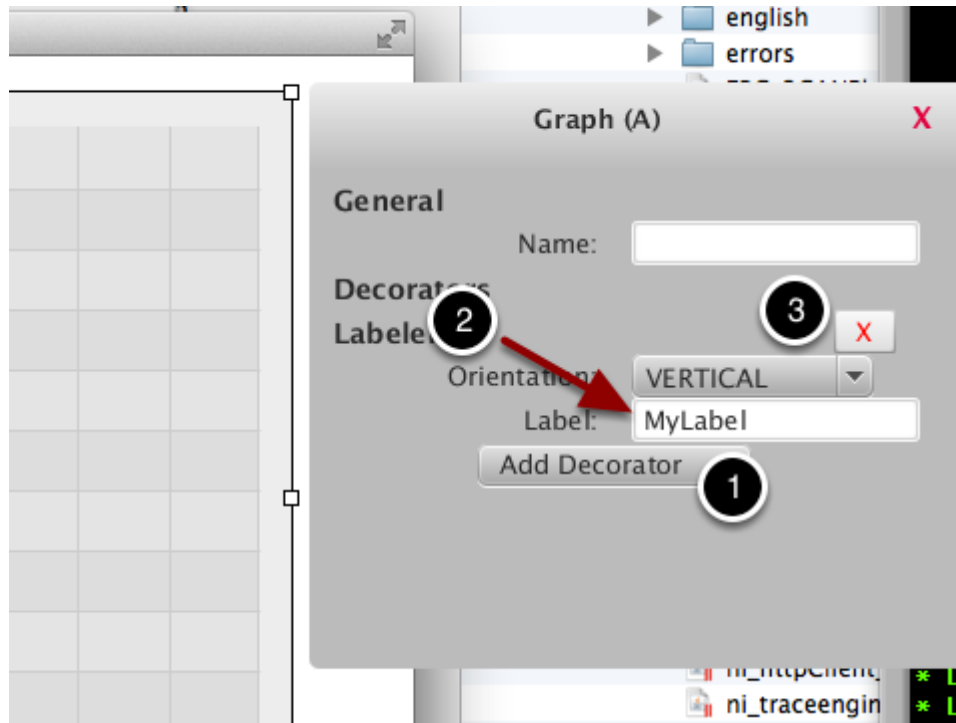
The widget may also be resized by dragging the resize handles (4) to change the width or height of the value.

Edit Properties



Clicking on the control selects it and brings up a properties list. Most controls only show a few basic properties by default, but can show all properties by clicking the "More" button (1) This shows all properties grouped by type, as well as the decorators section, which is explained next. If you don't know what a property is, hover over its name for a short description of what it does. Properties are updated instantly, no need to save.

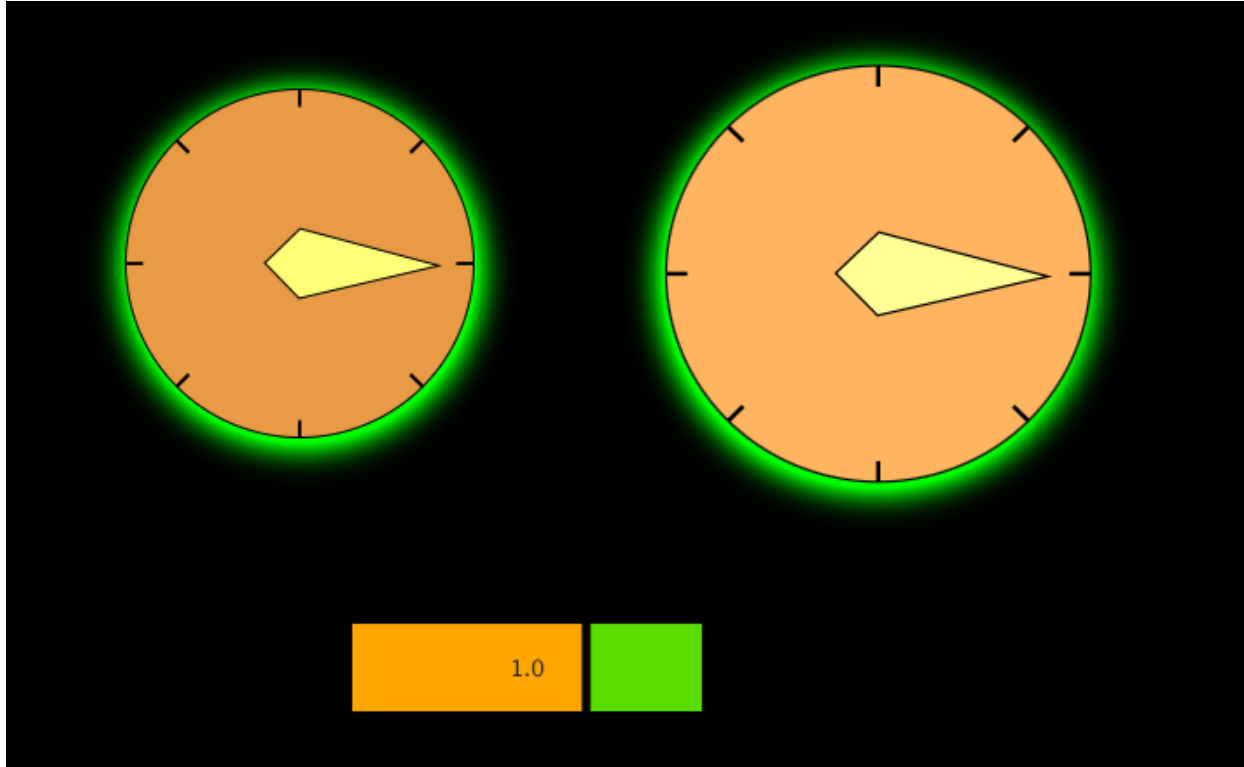
Adding labels to dashboard widgets



You can "decorate" a widget with a label. To add a label, click on the widget then click on the "More" button to view the extended properties. From there, click the "Add Decorator" button (1) and select "labeler" to add a label to the widget. You can then type the label text into the label field (2) in the property panel for the widget. Select Horizontal to place the label before the control and vertical to place the label underneath the control.

To later delete the label from the widget, press the red X button to the right of the decorator to be removed (in this case, the labeler) (3)

Using css styles to modify the look of widgets



Most controls support CSS styling as they use JavaFX primitives. At the present time, you must enter css rules in a text box, but this will change with a full designer in a future release. All CSS is JavaFX CSS and is NOT the same as browser CSS. See the [JavaFX CSS reference page](#) for commands.

The above layout was made with a canvas with

```
-fx-background-color: black;
```

a ValueMeter with

```
-fx-background-color: orange;
```

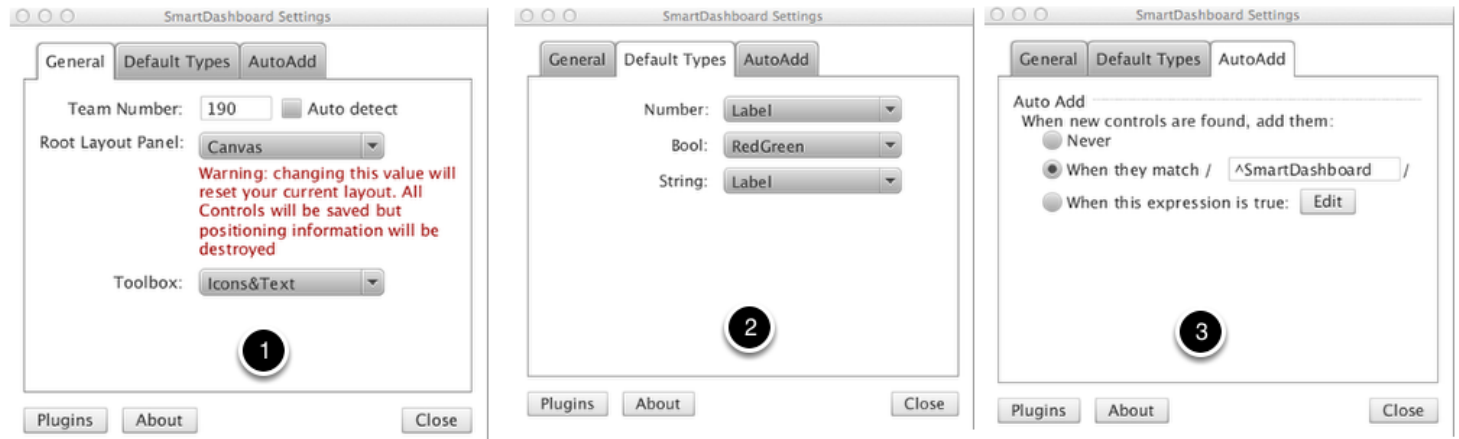
and two Compasses with

```
-fx-effect: dropshadow(gaussian, lime, 30, 0.5, -1, 3);
```

SmartDashboard

and different fill and needle colors.

Settings in SFX

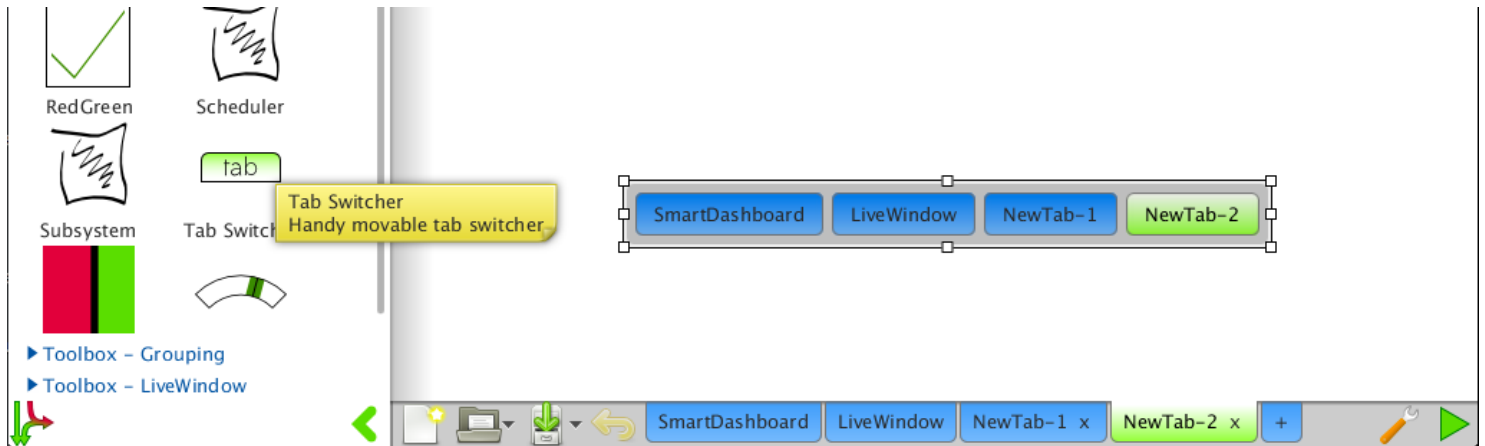


There are a number of settings for SFX to set the behavior to match your requirements. The settings are in 3 panels as shown above. Described below are the options that can be set:

1. **General settings** - This sets the team number which is used for communications with the robot. The root panel layout is the default JavaFX layout type that is used for laying out the root panel in a tab. Changing this will change the way the widgets are placed on the panel when automatically added as well as your ability to manipulate the widgets once they are placed. The toolbox option sets how toolbox items will be displayed in the palette (when clicking on the plus (+) in the lower corner of the screen).
2. **Default types** - these are the types of widgets that will be automatically created when the SmartDashboard values are loaded in the "SmartDashboard" tab in sfx. For each type of data, numeric, boolean, or string select the widget type that should be created.
3. **AutoAdd settings** - widgets are automatically added to the "SmartDashboard" panel when they meet the criteria shown in this editor. By default, you can see that names that start with SmartDashboard in the name are automatically placed.

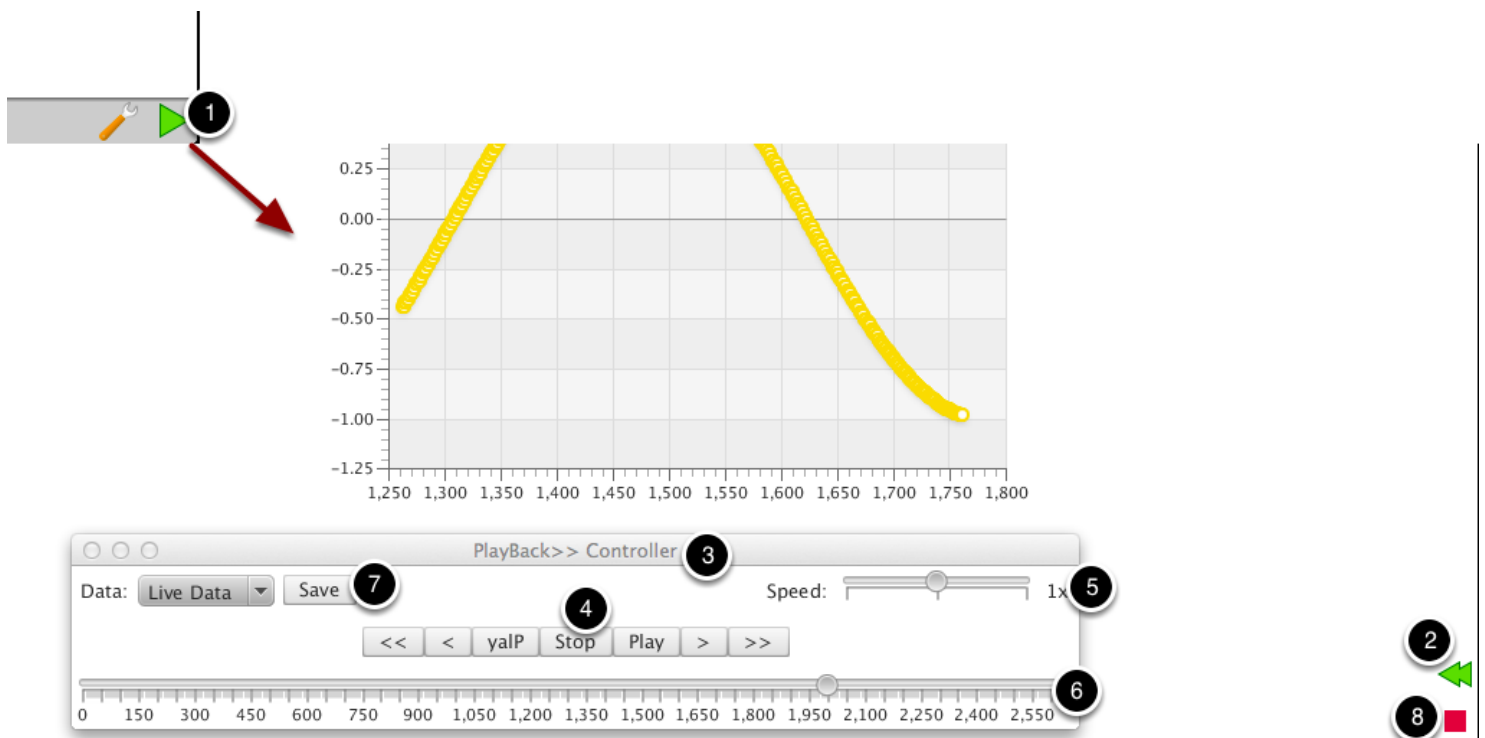
SmartDashboard

Tab Switcher



When in run mode, tabs are hidden. LiveWindow will automatically show and hide, however it is difficult to move between custom tabs. The tab switcher control enables you to switch tabs even when running. Simply drag it onto a tab from the General toolbox and run. Note you should probably add it to multiple tabs.

Running & Playback



SmartDashboard

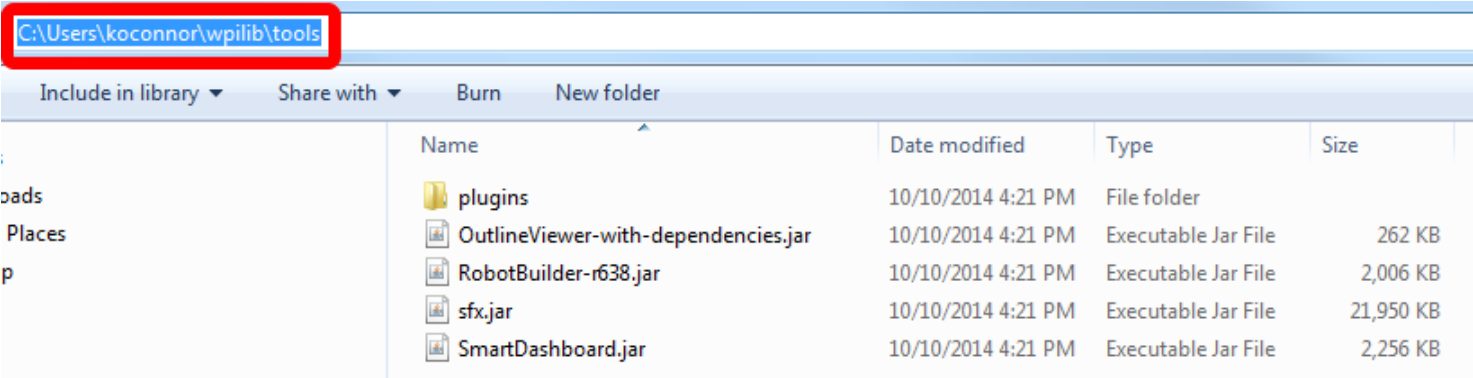
All controls display data in any mode, but you can only interact with controls in run mode. Hit the green run button (1) in the bottom right to enter run mode.

Once in run mode, you can either use the dashboard, exit (8), or open playback (2). Playback continually records changes in data over time and allows you to play these changes back at any time. When you open playback (3), it pauses all data (though it continues to record in the background). You can use the media controls (4) to step around the data, and play at speed (which can be controlled by 5), or drag the slider (6) manually to scrub around. Data can also be saved using (7) and loaded at another time to save matches, for example. To exit playback, just close the window (3)

Setting SFX to Launch with the DS

The C++ and Java Dashboard buttons in the Driver Station link to the previously existing SmartDashboard for the 2015 season. This article details how to set up the DS to launch SFX instead.

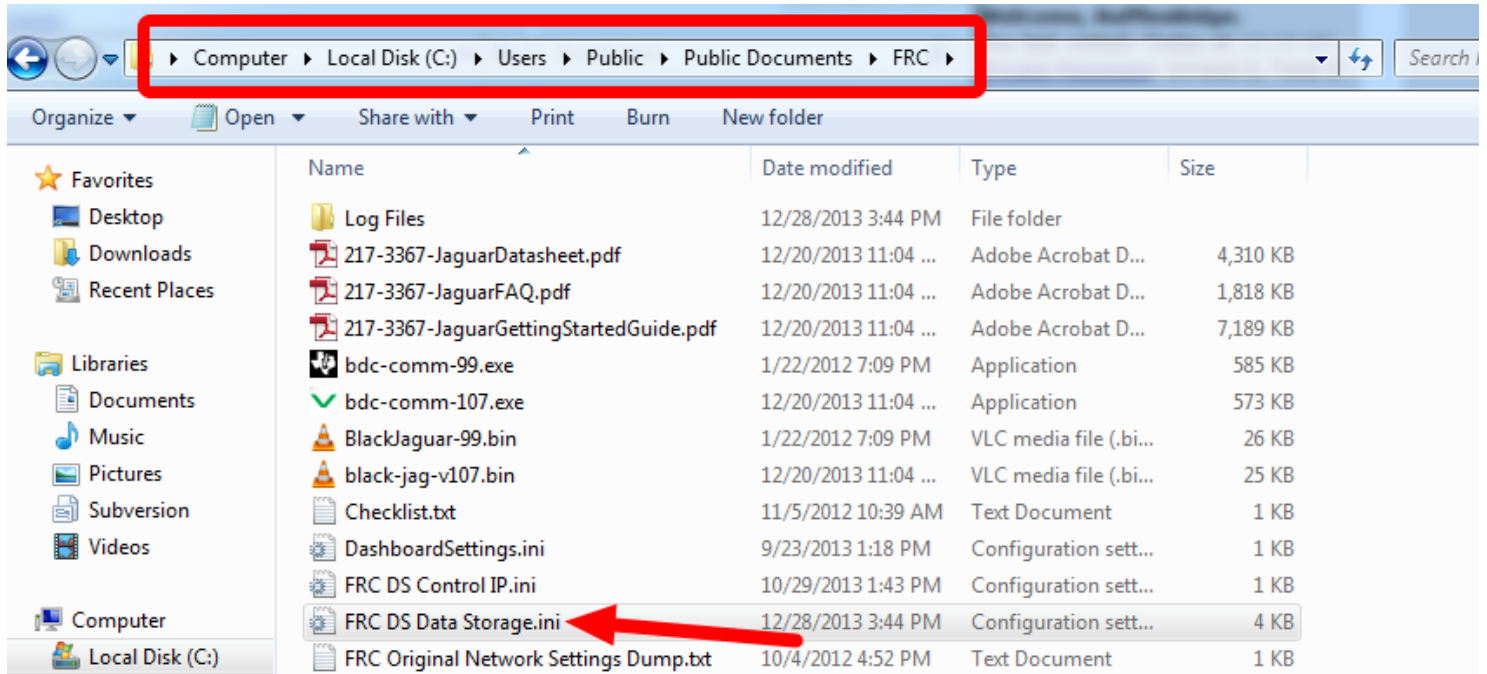
Locate SFX



Locate the path to the SFX jar file on your system. It is likely `C:\Users\USERNAME\wpilib\tools`. Note the full path to the `sfx.jar` file this will be needed later.

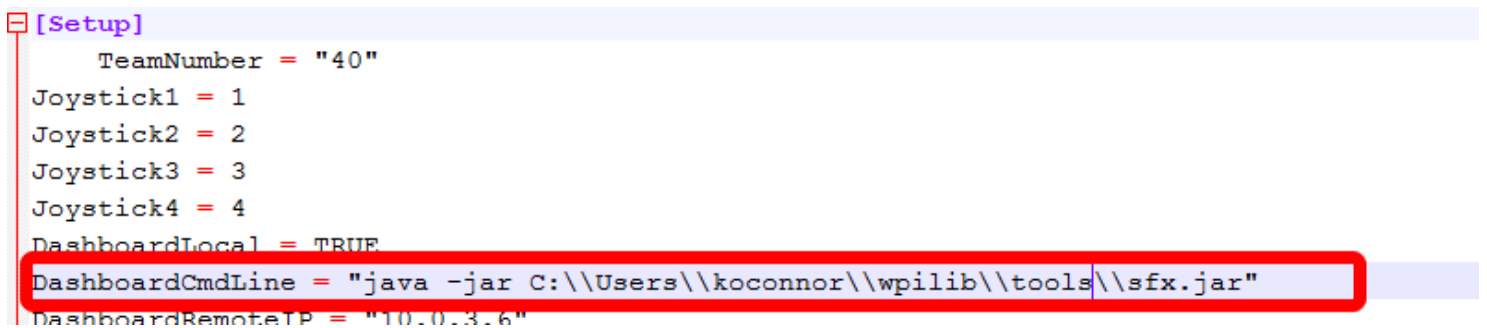
SmartDashboard

Open FRC DS Data Storage



Make sure the Driver Station is not open, then locate and open FRC DS Data Storage.ini. On Windows 7 machines this will be in the C:\Users\Public\Documents\FRC folder.

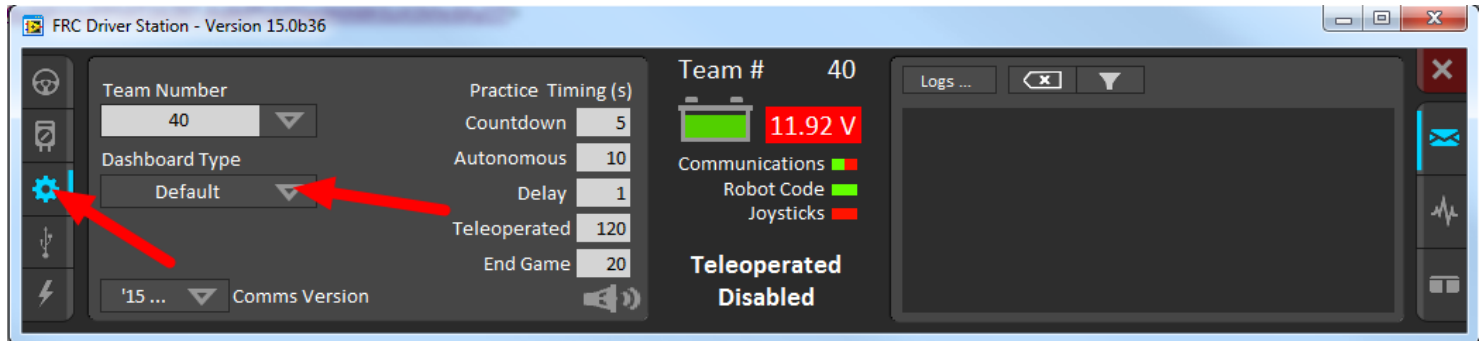
Set Dashboard path



Locate the line that starts with `DashboardCmdLine =` and replace the value in quotes with `"java -jar SFXPATH"` where SFXPATH is the path you noted earlier with all backspaces doubled. An example of this for the user "koconnor" is shown in the image above. Save and close the document.

SmartDashboard

Set the DS Dashboard Setting



Launch the Driver Station and select the Setup tab. Change the Dashboard Type setting to Default. This should launch the SFX dashboard. If it does not try closing the DS and re-opening.

Troubleshooting

If the SFX Dashboard does not launch with the DS after following the above steps, try the following:

1. Verify that SFX launches when you double click on the JAR file.
2. Return to the DS Data Storage file and verify that the path there is correct.
3. Open a Command Prompt by clicking Start and typing "cmd" in the box, then clicking Enter. Copy the path from the DS Data Storage file (including the "java -jar" part) and paste it into the Command Prompt by right-clicking and selecting Paste. Press Enter and verify that SFX launches. If it does not you may have an issue with your Java version or the environment variables which point to the Java install. If this is the case it is recommended to uninstall Java and complete a fresh install of [Java Runtime Engine version 7](#).

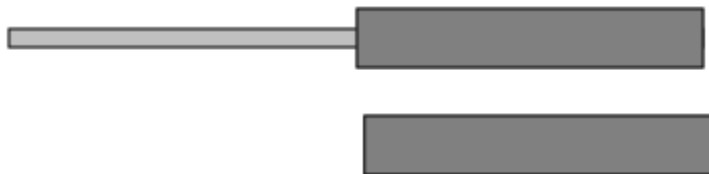
Creating a custom control using FXML

sfx comes with a palette of built-in controls that feature a wide range of use cases. But sometimes you would like to further customize your robot dashboard with controls that you create yourself. There are two strategies for creating custom controls, either:

1. FXML - a XML-based markup language for describing your own controls using a declarative language without needing programming
2. Java-based controls can have more complex requirements and behaviors

In this lesson we'll look at creating FXML-based controls. For creating Java controls, see the [Java tutorial](#).

Creating a pneumatic piston position indicator using FXML



Suppose you need to display the position of a pneumatic piston to make it clear to the operators the state of the piston. Ideally you would draw the piston and show the piston rod either in or out of the case. Showing it graphically might be much clearer than just having an indicator that was either red or green. FXML allows you to create more complex drawings and animate them based on, in this case, a boolean value. The illustration above shows the piston in the in and out position.

In this example we use a 2x3 grid pane to model the piston. When it is extended, the left middle cell contains a light grey panel that is visible. When it is retracted, that panel is hidden.

Creating the FXML file

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import java.lang.*?>
4  <?import java.util.*?>
5  <?import javafx.geometry.*?>
6  <?import javafx.scene.*?>
7  <?import javafx.scene.control.*?>
8  <?import javafx.scene.layout.*?>
9  <?import javafx.scene.shape.*?>
10
11 <dashfx.controls.bases.BooleanControlBase fx:id="base" xmlns:fx="http://javafx.com/fxml">
12   <ui>
13     <GridPane prefHeight="63.0" prefWidth="488" xmlns:fx="http://javafx.com/fxml">
14       <children>
15         <Pane visible="{base.value}" prefHeight="200.0" prefWidth="200.0" style="-fx-back
16         <Pane prefHeight="200.0" prefWidth="200.0" style="-fx-background-color: gray;-fx-b
17       </children>
18     </GridPane>
19     <columnConstraints>
20       <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
21       <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
22     </columnConstraints>
23     <rowConstraints>
24       <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
25       <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
26       <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
27     </rowConstraints>
28   </ui>
29 </dashfx.controls.bases.BooleanControlBase>
30
```

FXML is a declarative markup language for describing the graphical properties of your widget. It lets you specify values for shape, color, position or other properties to describe your widget.

FXML controls have the following structure:

1. Header
2. Base
3. UI & Bindings

The header is made of the XML header and JVM imports to avoid fully qualifying all nodes

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
```

SmartDashboard

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.shape.*?>
```

The **base** is required for SFX controls and abstracts the data management away. The following are valid bases:

- BooleanControlBase - any boolean-like values (true/false, 1/0, etc...)
- NumberControlBase - any floating point or integral value. Also coerces numbers in strings.
- RangedNumberControlBase - The same as a NumberControlBase but with a min and max value
- StringControlBase - any string value or the toString of any other values

In this case we need a boolean so we use the boolean control base (note we assign it an ID so we can bind to it later):

```
<dashfx.controls.bases.BooleanControlBase fx:id="base" xmlns:fx="http://javafx.com/fxml">
</dashfx.controls.bases.BooleanControlBase>
```

All current bases have a **ui** property that describes what to show on screen, and this is where we put all the **ui**:

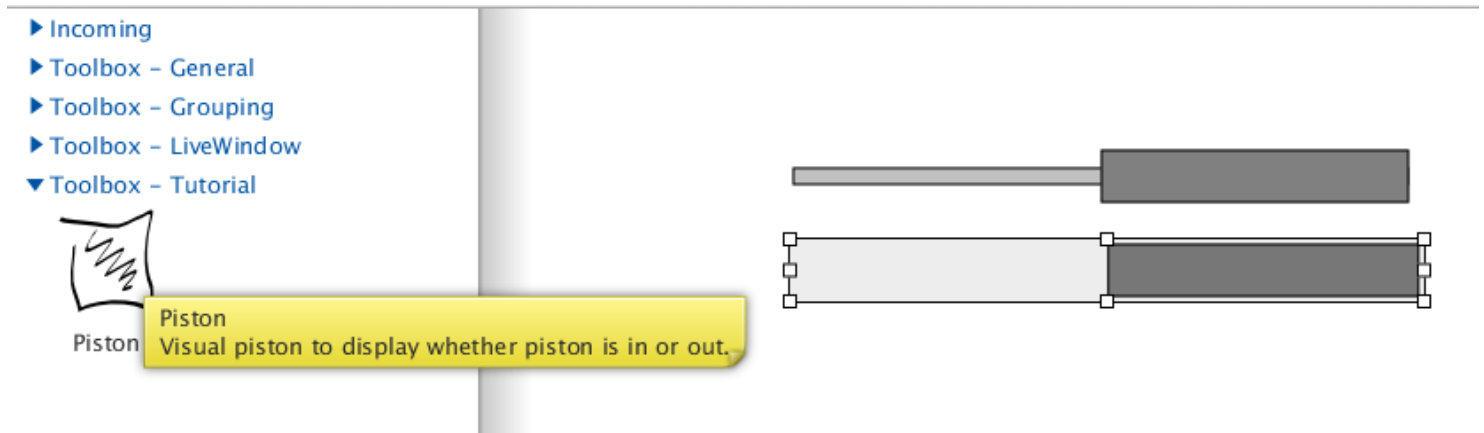
```
<ui>
  <GridPane prefHeight="50" prefWidth="500">
    <children>
      <Pane visible="{base.value}"
        prefHeight="200.0" prefWidth="200.0"
        style="-fx-background-color: silver;-fx-border-color: black;"
        GridPane.columnIndex="0"
        GridPane.columnSpan="2147483647"
        GridPane.rowIndex="1" />
      <Pane prefHeight="200.0" prefWidth="200.0"
        style="-fx-background-color: gray;-fx-border-color: black;"
        GridPane.columnIndex="1"
        GridPane.rowIndex="0"
        GridPane.rowSpan="2147483647" />
    </children>
    <columnConstraints>
      <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
      <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
    </columnConstraints>
  </GridPane>
</ui>
```

SmartDashboard

```
</columnConstraints>
<rowConstraints>
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
</rowConstraints>
</GridPane>
</ui>
```

This contains normal FXML and was built with the JavaFX Scene Builder. Note the first pane's visibility property is bound to the base's value, thus enabling the piston to appear retracted when the value is false, and extended when true. This is all the FXML needed to define this control.

Registering your control with a manifest



To add your control to the dashboard, you need to package it in a plugin. For FXML controls you can simply put it in a folder or pack it in a jar. Either way, we need a manifest file that describes what the plugin contains. Manifests are written in YAML and can contain multiple controls and other options. For our needs, we will start with this file (please generate your own UUID. uuidgenerator.net is where the provided UUID was generated)

```
API: 0.1
Name: Tutorial plugin
Description: Contains all the plugins from the tutorial
Version: 1.0.0
# Please generate your own unique UUID and replace it below
Plugin ID: b673b0fe-716a-40ce-b446-70e34aafc509
Controls:
```

SmartDashboard

```
-  
Name: Piston  
Description: Visual piston to display whether piston is in or out.  
Source: /piston.fxml  
Category: Tutorial  
Defaults:  
  value: false
```

This says:

- we are using plugin API version 0.1 (the current version)
- the name of the plugin is "Tutorial plugin" which can be identified by the UUID
- it has one control, the Piston control, which is in the Tutorial category
- it's described by the given fxml file.

Note most fields are optional and many have been left off this simple example. Save the manifest as manifest.yml. NOTE: YAML expects spaces for indentation, not tabs.

Now register this with SFX. Create a new folder in the sfx/plugins folder in your copy of sfx with the following layout:

```
tutorial  
  manifest.yml  
  piston.fxml
```

Now when you start sfx, you should be able to use the control. It is recommended to launch from the terminal with the command:

```
java -jar sfx.jar
```

in case there are any errors. All plugins can be viewed under settings>plugins.

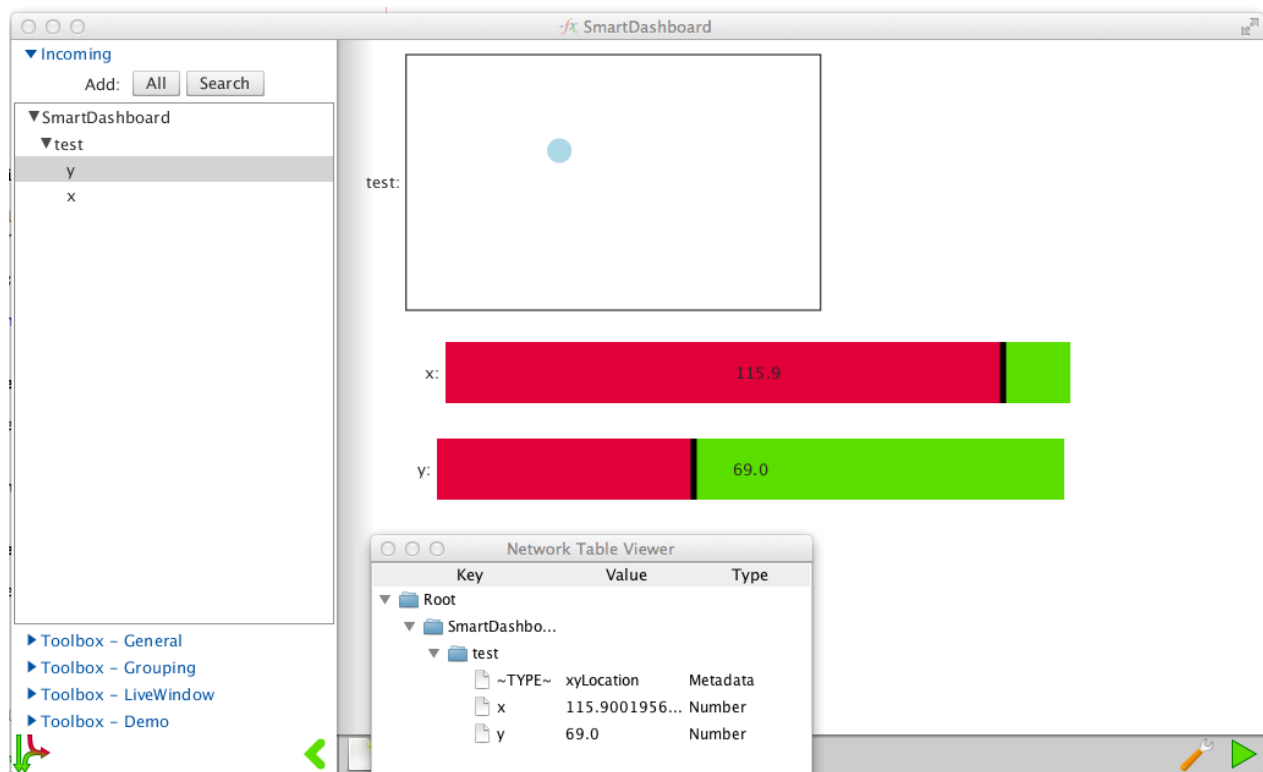
Creating a custom control using Java

sfx comes with a palette of built-in controls that feature a wide range of use cases. But sometimes you would like to further customize your robot dashboard with controls that you create yourself. There are two strategies for creating custom controls, either:

1. FXML - a XML-based markup language for describing your own controls using a declarative language without needing programming
2. Java-based controls can have more complex requirements and behaviors

In this lesson we'll look at creating Java-based controls. For FXML controls see the [FXML tutorial](#).

Creating a X-Y location indicator using Java

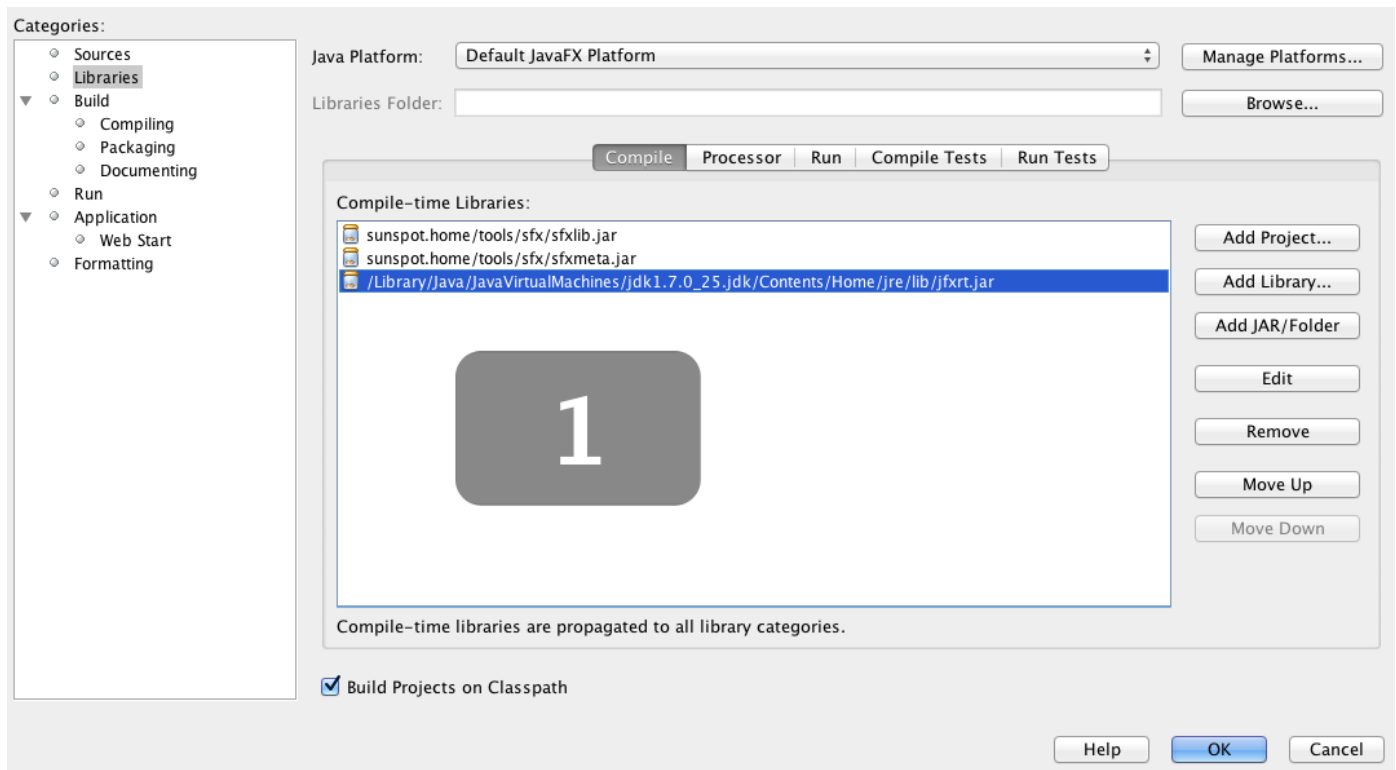


Suppose you need to display some object in 2D space, like a vision target from the camera, robot position on the field, or Joystick position so that field operators can easily see the location. As this has multiple variables, it is much easier to do this with Java-based controls.

SmartDashboard

In this simple example, we will be adding an [Ellipse](#) to a data-enabled [AnchorPane](#) and moving it based on an object with x and y properties.

Create a Netbeans project



In Netbeans, create a new Java Class Library. Once you create the project, right click it and go to properties. Inside the properties, select "Libraries" and add sfxlib.jar and sfxmeta.jar (they will appear in the same directory as sfx.jar after one run). Also add jfxrt.jar, which is system dependent, but is normally found in \$JAVA_HOME/jre/lib/jfxrt.jar

Adding the Control class

Now add a Java source file for your new class (this example will call it xyLocation.java in package com.example). For our example we will extend DataAnchorPane as it is both data-enabled and supports positioning children via x and y coordinates. Since Java classes can have annotations, we can place what FXML files require in manifests in annotations only. Add the following annotation to the class:

```
@Category("Tutorial")
```

SmartDashboard

This marks the class as being in the toolbox category "Tutorial"

```
@Designable(value = "X-Y Location", description = "A control to show x/y position in a range")
```

This describes this class as being designable in SFX, showing it with the given name and description

```
@GroupType("xyLocation")
```

This says that the control designs all groups of type xyLocation. This is implemented in NetworkTables by giving a table a sub-key of ~TYPE~ with value xyLocation

```
@DashFXProperties("Sealed: true, Save Children: false")
```

This adds any arbitrary manifest attributes to the class. These say to treat this as an atomic object, even though we are extending a pane that supports designable children.

As we are extending from a data-enabled class (DataAnchorPane), we can simply call `getObservable()` on ourselves and not worry too much about it. As we are displaying an object, we do need to enable the default name-prepend via `setDataMode(DataPaneMode.Nested)`. This makes all calls to `getObservable("x")` to retrieve the values under `this.getName() + "/x"` instead of just "x".

All controls are given a data source to follow when they are registered with the DataCore, which then calls `registered()`. We need to override this so we can get our keys from the provider at this time.

```
@Override
public void registered(DataCoreProvider provider)
{
    super.registered(provider);
    unwatch();
    // if we are being registered, then we can finally get the x and y variable
    // otherwise just unwatch as we are being unregistered
    if (provider != null)
    {
        xValue = getObservable("x");
    }
}
```


SmartDashboard

```
        yValue = getObservable("y");
        rewatch();
    }
}
```

In order to enable more complex actions later, we will add listeners to the SmartValues

```
private void rewatch()
{
    xValue.addListener(xchange);
    yValue.addListener(ychange);
}
private void unwatch()
{
    // this function un-binds all the variable
    if (xValue != null)
        xValue.removeListener(xchange);
    if (yValue != null)
        yValue.removeListener(ychange);
}
```

xchange and ychange are defined as follows but can easily be extended for multiple other features and/or calculations

```
private ChangeListener ychange = new ChangeListener<Object>() {
    @Override
    public void changed(ObservableValue<? extends Object> ov, Object
t, Object t1)
    {
        ellipse.setCenterY(yValue.getData().asNumber() + 10); //
offset by radius
    }
},
xchange = new ChangeListener<Object>() {
    @Override
    public void changed(ObservableValue<? extends Object> ov, Object
t, Object t1)
    {
```

SmartDashboard

```
        ellipse.setCenterX(xValue.getData().asNumber() + 10); //
offset by radius
    }
};
```

This simply directly sets the position from the values with a constant offset of the radius. Note that this does not scale nor have any limits, so the ellipse can move off the canvas. The units are JavaFX DPI-independent pixels (roughly 1 px at normal dpi with no transforms)

The ellipse is very simple and defined as such:

```
// we are displaying results by moving the ellipse. initialize it here
ellipse = new Ellipse(10, 10, 10, 10);
ellipse.setFill(Color.LIGHTBLUE);
this.getChildren().add(ellipse); // we inherited from DAP so just add it
to ourselves
```

Note that currently there is a small bug in DataPane in that nested mode does not update the correct keys when the name changes. As such, it is currently required to re-bind on each name change, however will not be once this bug is fixed

```
nameProperty().addListener(new ChangeListener<String>()
{
    @Override
    public void changed(ObservableValue<? extends String> ov, String
t, String t1)
    {
        unwatch();
        try
        {
            xValue = getObservable("x");
            yValue = getObservable("y");
            rewatch();
        }
        catch(NullPointerException n)
        {
            //fail, ignore, as we must not be registered yet
        }
    }
});
```

```
    }  
  });  
}
```

Creating the manifest to register the control with SFX

To add your control to the dashboard, you need to package it in a plugin. For FXML controls you can simply put it in a folder or pack it in a jar. Either way, we need a manifest file that describes what the plugin contains. Manifests are written in YAML and can contain multiple controls and other options. For our needs, we will start with this file (please generate your own UUID. uuidgenerator.net is where the provided UUID was generated)

```
API: 0.1  
Name: Tutorial plugin  
Description: Contains all the plugins from the tutorial  
Version: 1.0.0  
# Please generate your own unique UUID and replace it below  
Plugin ID: b673b0fe-716a-40ce-b446-70e34aafc509  
Controls:  
-  
  Class: com.example.xyLocation
```

This says:

- we are using plugin API version 0.1 (the current version)
- the name of the plugin is "Tutorial plugin" which can be identified by the UUID
- it has one control, the xyLocation control, which has more information in its annotations

Save the manifest as manifest.yml in the root of the src/ folder. NOTE: YAML expects spaces for indentation, not tabs.

Now build the project with netbeans, and copy the jar to sfx/plugins/. Now when you start sfx, you should be able to use the control. It is recommended to launch from the terminal with the command:

```
java -jar sfx.jar
```

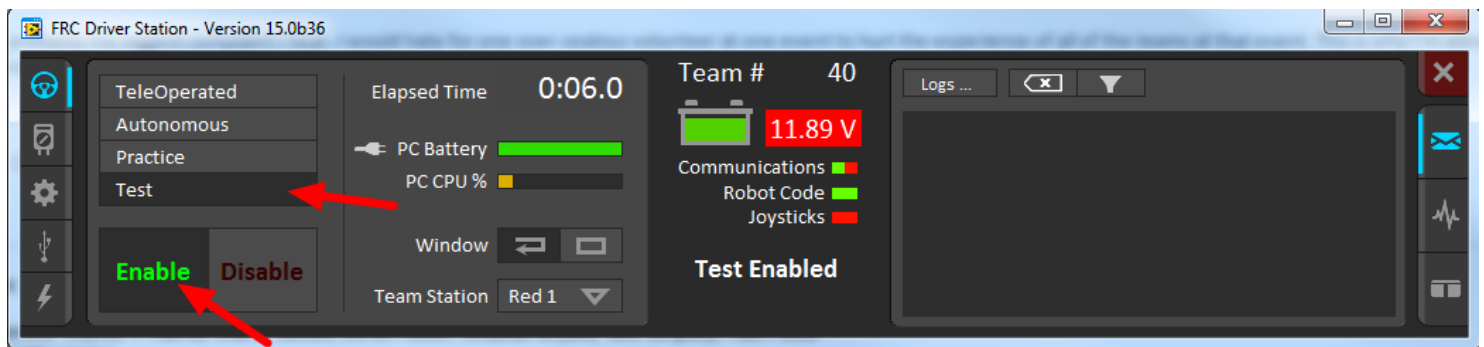
in case there are any errors. All plugins can be viewed under settings>plugins.

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



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

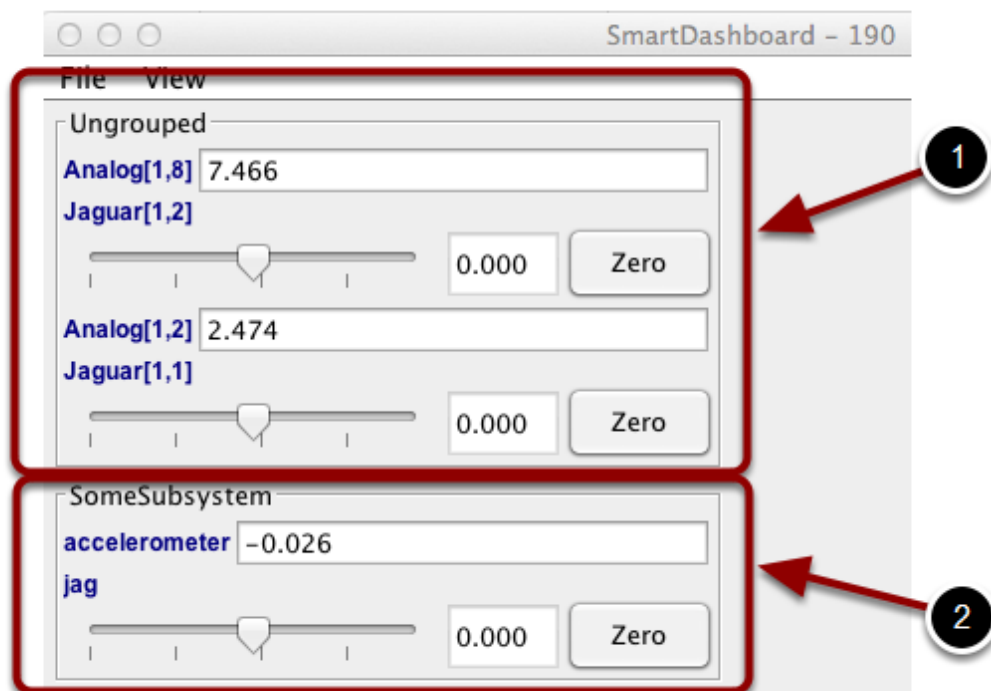
```
RobotDrive drive = new RobotDrive(1, 2);
Jaguar jag;
Accelerometer accel = new Accelerometer(1, 2);

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);
}
```

SmartDashboard

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 highlighted code.

Understanding what is displayed in Test mode

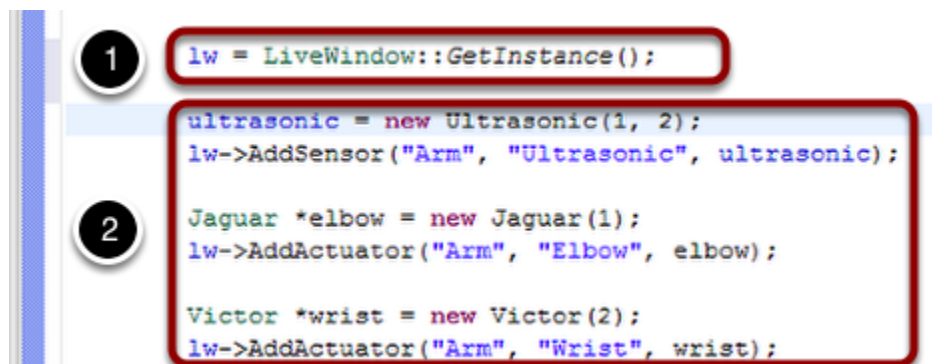


This is the output in the SmartDashboard display when the robot is placed into test mode. In the display shown above the objects listed as Ungrouped were implicitly created by WPILib when the corresponding objects were created. These objects are contained in a subsystem group called "Ungrouped" (1) and 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

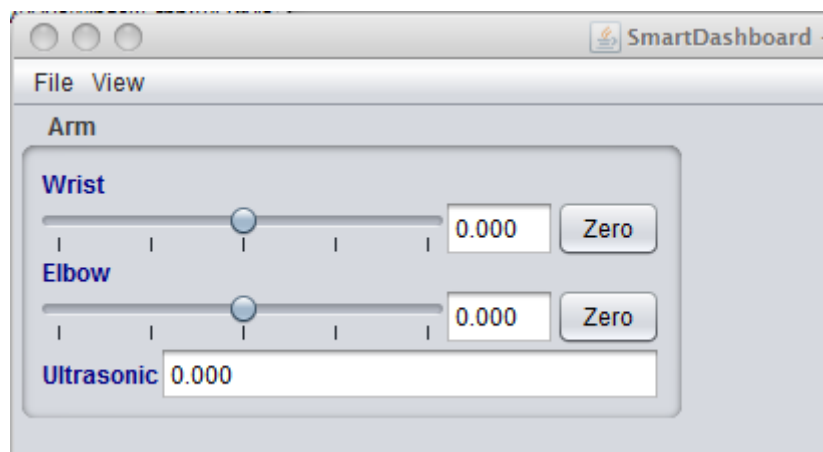
Typically LiveWindows are displayed as part of the automatically generated RobotBuilder code. You may also display LiveWindow values by writing the code yourself and adding it to your robot program. LiveWindow will display values grouped in subsystems. This is a convenient method of displaying whether 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



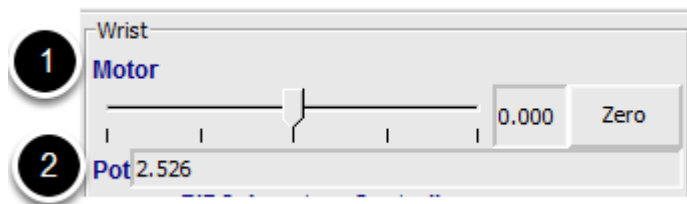
SmartDashboard

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

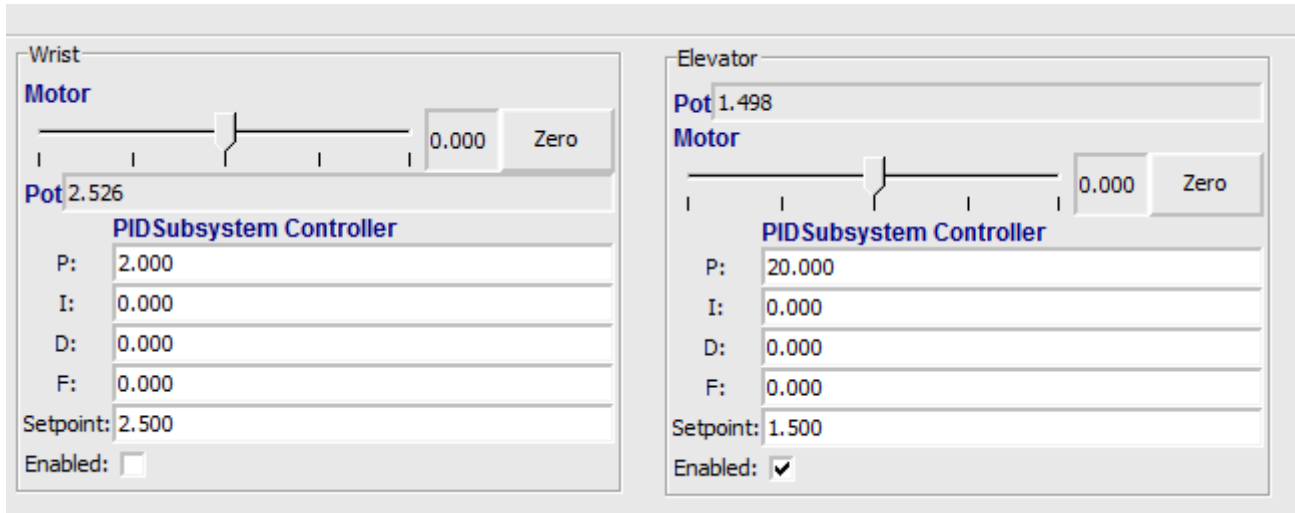
The PID (Proportional, Integral, Differential) is an algorithm for determining the motor speed based on sensor feedback to reach a setpoint as quickly as possible. For example, a robot with an elevator that moves to a predetermined position should move there as fast as possible then stop without excessive 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



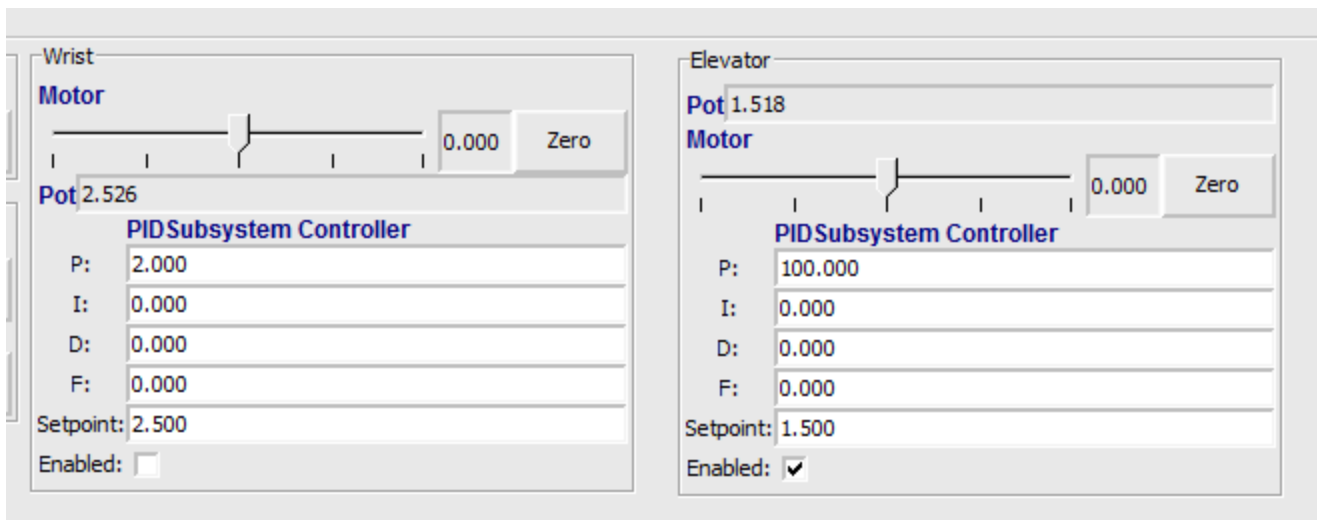
Create a PID Subsystem for each mechanism with feedback. The PID Subsystems contain the actuator (motor) and the feedback sensor (potentiometer in this case). You can use Test mode to display the subsystem sensors and actuators. Using the slider manually adjust the actuator to each desired position.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

SmartDashboard

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!

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.

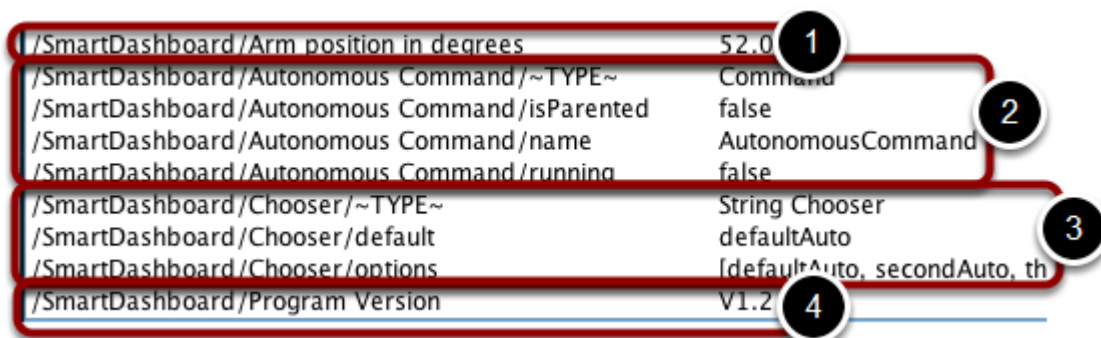
SmartDashboard namespace

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 TableViewer 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



The screenshot shows a TableViewer window with a list of SmartDashboard data entries. Four red boxes with numbered callouts (1, 2, 3, 4) highlight specific rows:

- 1: /SmartDashboard/Arm position in degrees 52.0
- 2: /SmartDashboard/Autonomous Command/~TYPE~ Command
- 3: /SmartDashboard/Autonomous Command/isParented false
- 4: /SmartDashboard/Autonomous Command/name AutonomousCommand

/SmartDashboard/Arm position in degrees	52.0
/SmartDashboard/Autonomous Command/~TYPE~	Command
/SmartDashboard/Autonomous Command/isParented	false
/SmartDashboard/Autonomous Command/name	AutonomousCommand
/SmartDashboard/Autonomous Command/running	false
/SmartDashboard/Chooser/~TYPE~	String Chooser
/SmartDashboard/Chooser/default	defaultAuto
/SmartDashboard/Chooser/options	[defaultAuto, secondAuto, th
/SmartDashboard/Program Version	V1.2

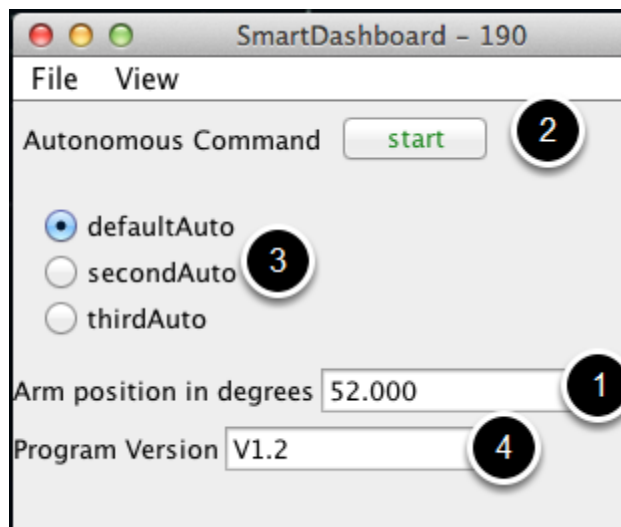
SmartDashboard values are created with key names that begin with "SmartDashboard/". The above values viewed with TableViewer correspond to data put to the SmartDashboard with the following statements:

```
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");
```

SmartDashboard

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

Key	Value
/LiveWindow/Drive train/Ultrasonic/~TYPE~	Analog Input
/LiveWindow/Drive train/Ultrasonic/Name	Ultrasonic
/LiveWindow/Drive train/Ultrasonic/Subsystem	Drive train
/LiveWindow/Drive train/Ultrasonic/Value	0.572972471
/LiveWindow/Elevator/Motor/~TYPE~	LW Subsystem
/LiveWindow/Elevator/Motor/Name	Speed Controller
/LiveWindow/Elevator/Motor/Subsystem	Motor
/LiveWindow/Elevator/Motor/Value	Elevator
/LiveWindow/Elevator/PIDSubsystem Controller/~TYPE~	PIDController
/LiveWindow/Elevator/PIDSubsystem Controller/d	0.0
/LiveWindow/Elevator/PIDSubsystem Controller/enabled	false
/LiveWindow/Elevator/PIDSubsystem Controller/f	0.0
/LiveWindow/Elevator/PIDSubsystem Controller/i	0.0
/LiveWindow/Elevator/PIDSubsystem Controller/Name	PIDSubsystem Controller
/LiveWindow/Elevator/PIDSubsystem Controller/p	1.0
/LiveWindow/Elevator/PIDSubsystem Controller/setpoint	0.0
/LiveWindow/Elevator/PIDSubsystem Controller/Subsystem	Elevator
/LiveWindow/Elevator/Pot/~TYPE~	Analog Input
/LiveWindow/Elevator/Pot/Name	Pot
/LiveWindow/Elevator/Pot/Subsystem	Elevator
/LiveWindow/Elevator/Pot/Value	3.9334140710000005
/LiveWindow/Wrist/~TYPE~	LW Subsystem
/LiveWindow/Wrist/Motor/~TYPE~	Speed Controller
/LiveWindow/Wrist/Motor/Name	Motor
/LiveWindow/Wrist/Motor/Subsystem	Wrist
/LiveWindow/Wrist/Motor/Value	0.0
/LiveWindow/Wrist/PIDSubsystem Controller/~TYPE~	PIDController
/LiveWindow/Wrist/PIDSubsystem Controller/d	0.0
/LiveWindow/Wrist/PIDSubsystem Controller/enabled	false
/LiveWindow/Wrist/PIDSubsystem Controller/f	0.0
/LiveWindow/Wrist/PIDSubsystem Controller/i	0.0
/LiveWindow/Wrist/PIDSubsystem Controller/Name	PIDSubsystem Controller
/LiveWindow/Wrist/PIDSubsystem Controller/p	1.0
/LiveWindow/Wrist/PIDSubsystem Controller/setpoint	0.0
/LiveWindow/Wrist/PIDSubsystem Controller/Subsystem	Wrist
/LiveWindow/Wrist/Pot/~TYPE~	Analog Input
/LiveWindow/Wrist/Pot/Name	Pot
/LiveWindow/Wrist/Pot/Subsystem	Wrist
/LiveWindow/Wrist/Pot/Value	3.4917683650000004

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);
```


SmartDashboard

```
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);
```

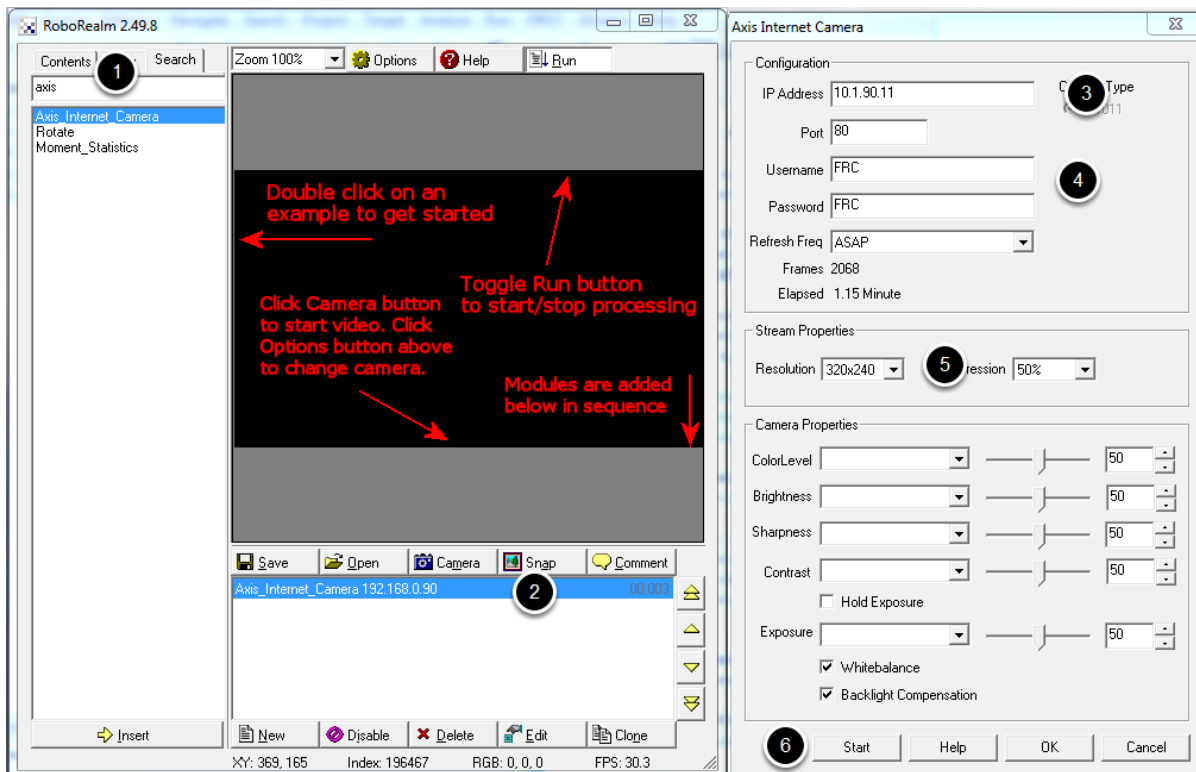
Values that correspond to actuators are not only displayed, but can be set using sliders created in the SmartDashboard in Test mode.

Using the SmartDashboard Vision installer

Viewing the RoboRealm output in SmartDashboard

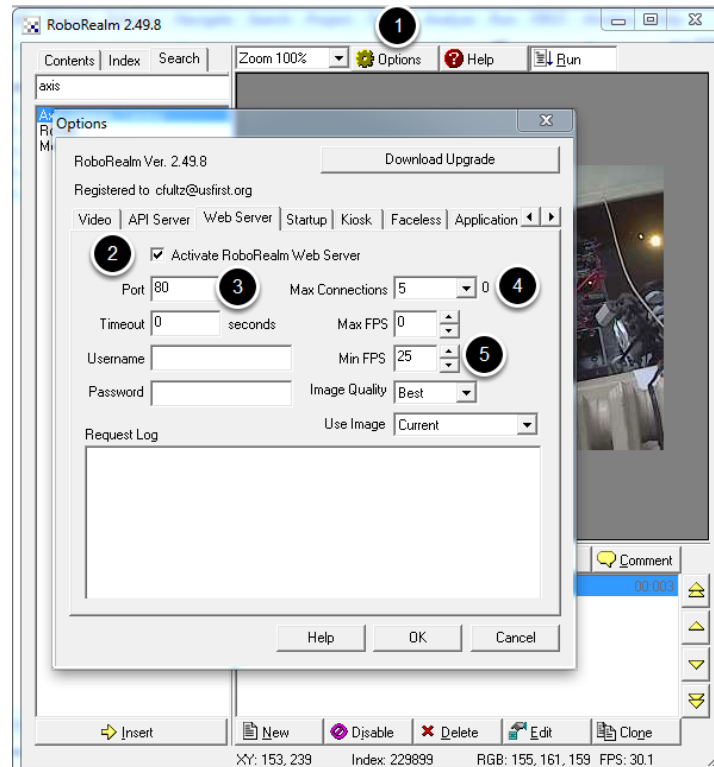
RoboRealm is a vision processing application that runs on a Windows PC connected to the robot via a network connection. It can read the camera stream, process images and send results back to the robot. It is often desirable to see the results of the image processing on your driver station laptop, but screen real estate is at a premium. You can display images from RoboRealm on the SmartDashboard by using its internal web server as shown in this article.

Set up the Axis camera in RoboRealm



Start RoboRealm and (1) search for Axis_internet_Camera and (2) add the Axis camera to the program. You'll see a popup of the camera properties. Set (3) the camera IP address (10.10.10.11), the (4) Username and Password to FRC and the (5) Resolution to 320x240, and hit (6) Start. You should see the image appear in the RoboRealm window. Hit OK to dismiss the Axis Internet Camera properties.

Enable the RoboRealm web server



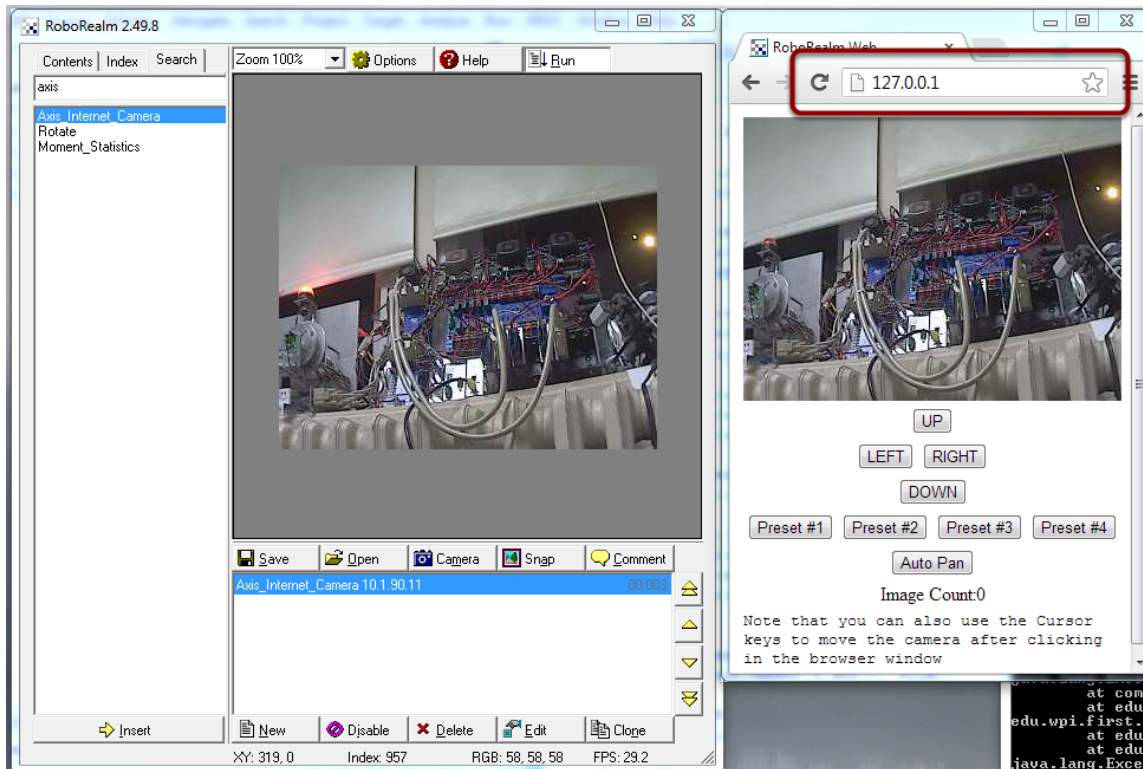
Do the following steps to set up the RoboRealm web server which will serve the processed image:

1. Click Options
2. Activate the RoboRealm web server
3. Set the port number to 80
4. Set the Max Connections to 5 for testing
5. Set the Min FPS to 25

And click OK to save the settings and start the web server.

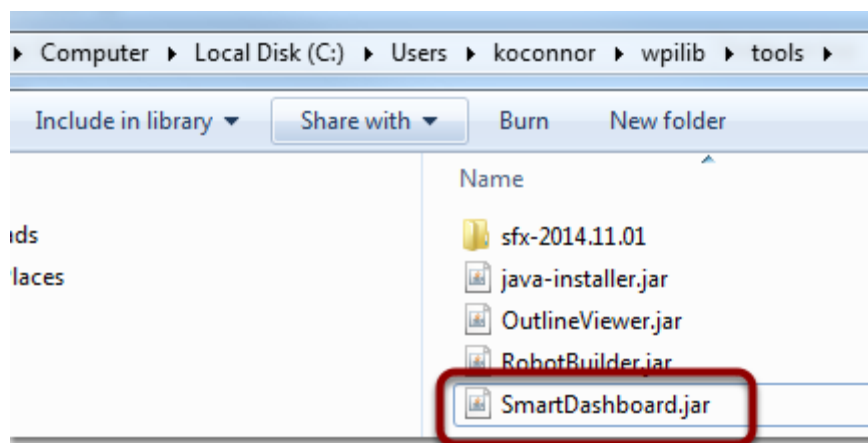
SmartDashboard

Verify that the web server is running



Verify the web server is running by opening up a browser on the system and entering the URL: 127.0.0.1. This is the address of this system (localhost) and should show the same image that RoboRealm is displaying.

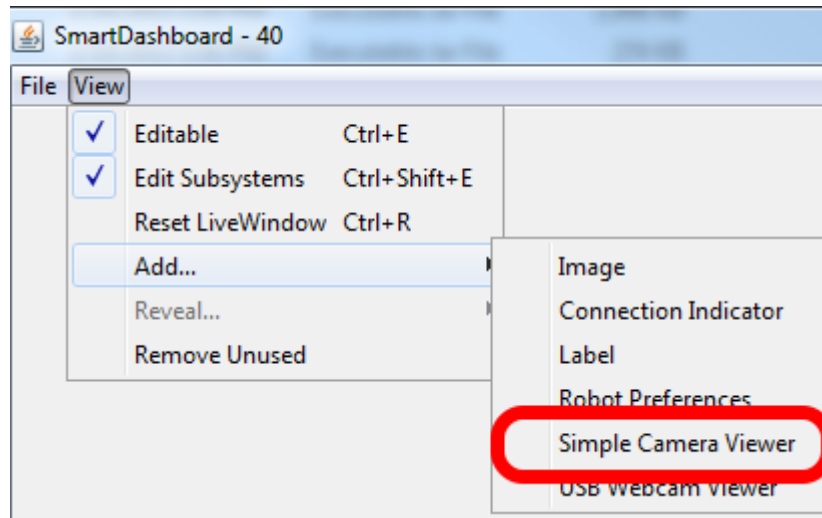
Run SmartDashboard



SmartDashboard

Run SmartDashboard located in HOME\wpilib\tools (on Windows HOME is usually C:\Users\USERNAME)

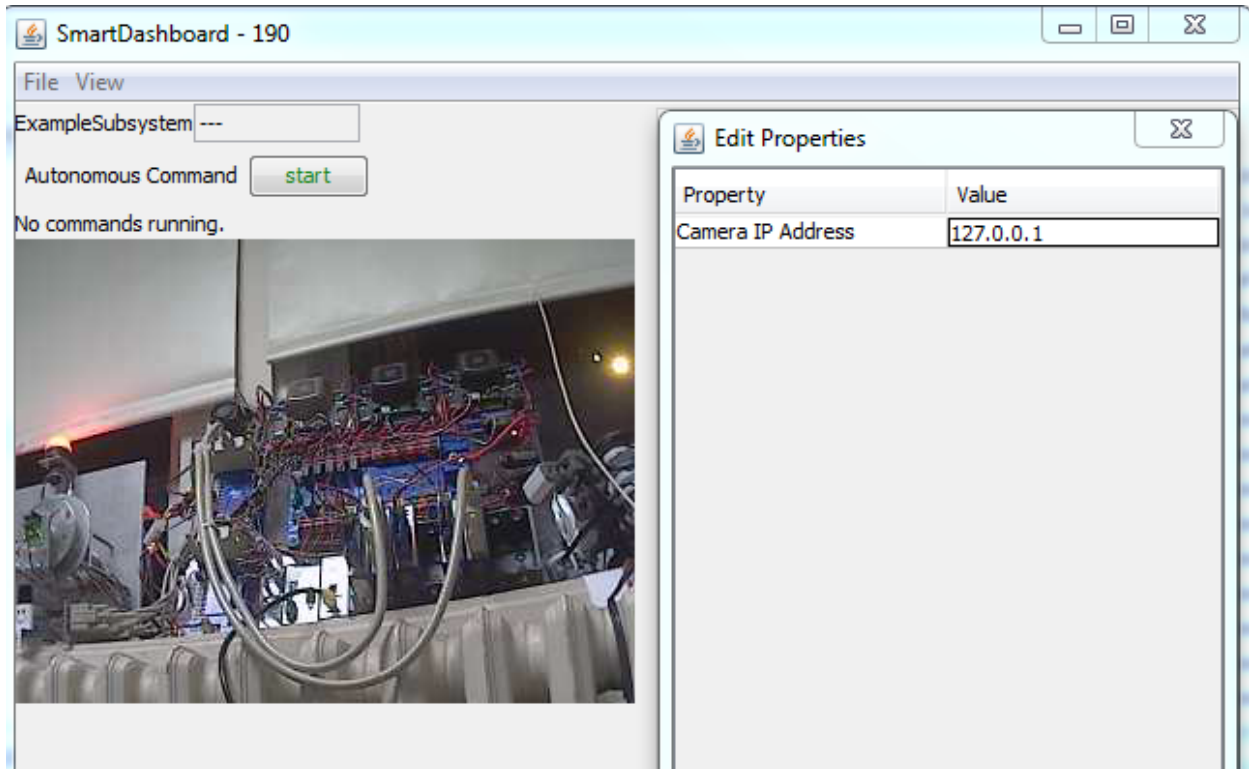
Add the camera to the SmartDashboard display



Add the Simple Camera Viewer to the SmartDashboard display by selecting View -> Add -> Simple Camera Viewer.

SmartDashboard

Set the camera plugin IP address

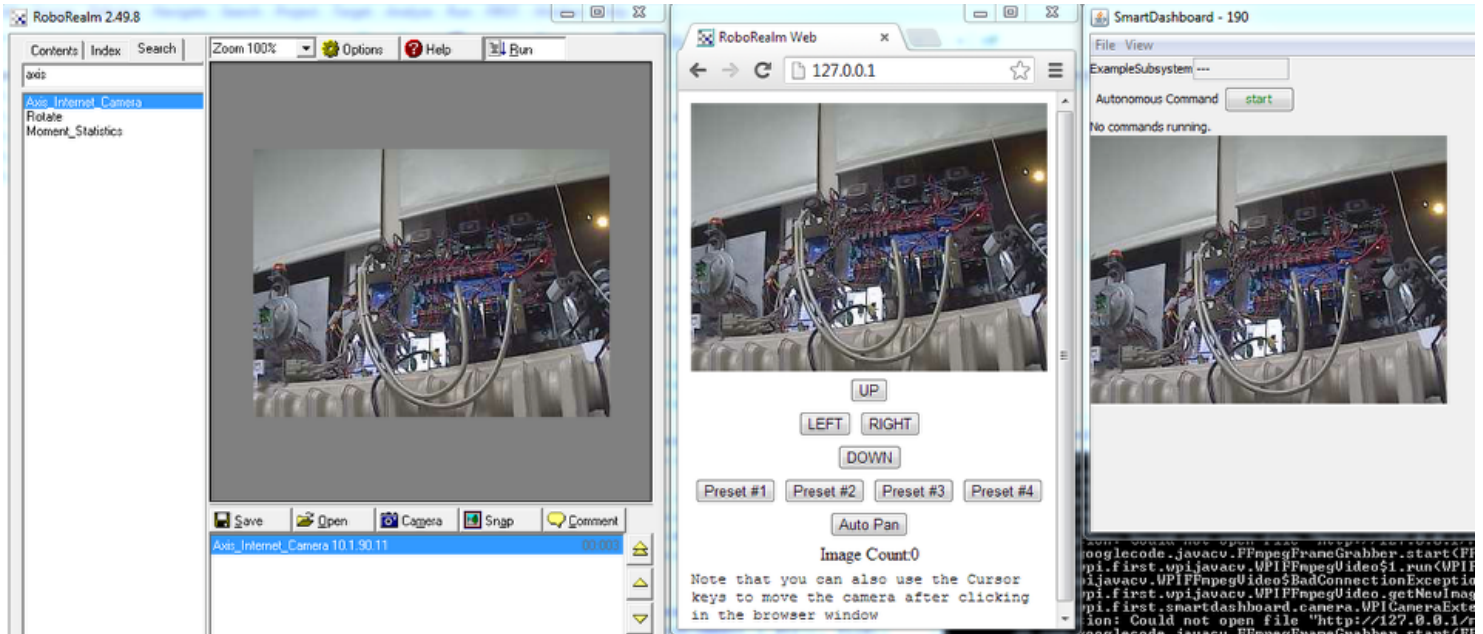


Verify that the camera IP address is set correctly by:

1. Selecting "View" then "Editable" to make the camera image properties accessible. Then right-click on the image and select Properties. Set the system IP address in the Camera IP Address field. This will be viewing the RoboRealm web server just as in the previous steps with the browser.

SmartDashboard

View the RoboRealm image



You should now see the RoboRealm image in both the Web Browser and in SmartDashboard at the same time.