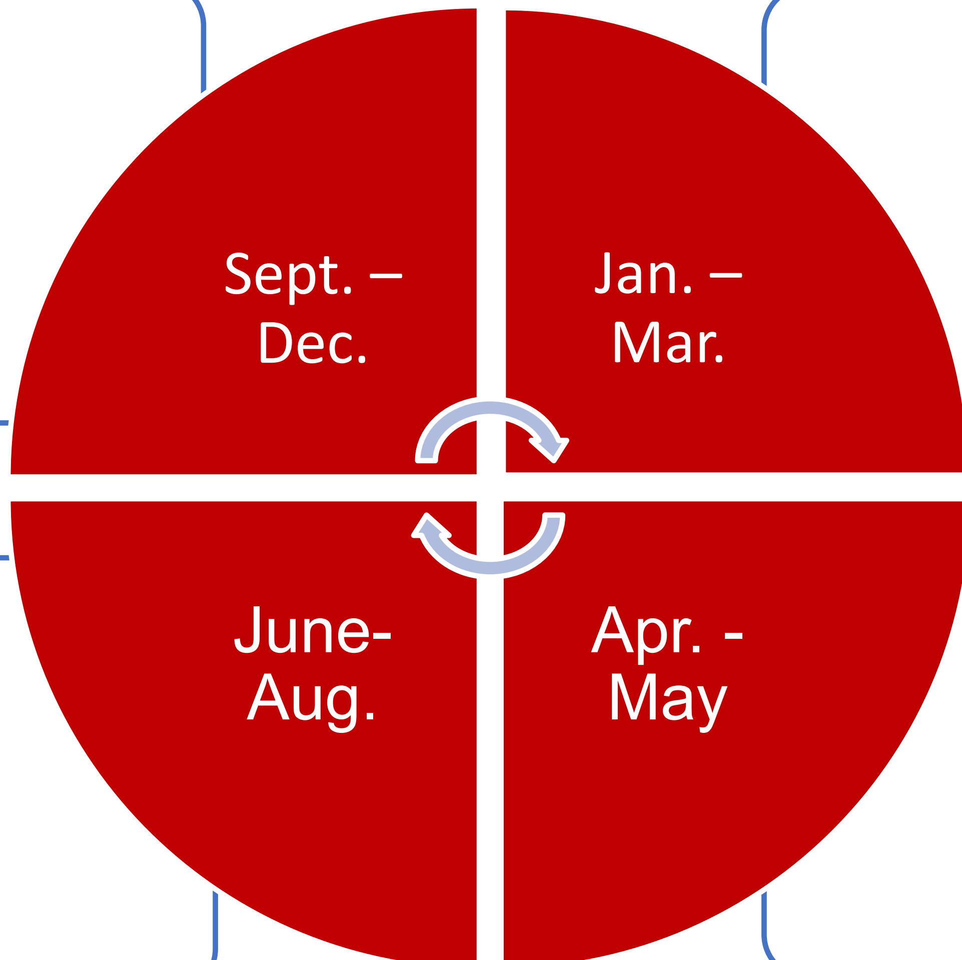
Welcome to the 2024 Botball Season

Botball Timeline

- Recruit
- Apply for Scholarship
- Kits ships to school
- Fall Game release
- Online workshops available
- Fall Game Dec.



- Game Release
- Workshops
- On-line workshops
- Documentation Starts
- Documentation 1 Due
- Social Media Posts

- Preparing for GCER
- GCER
- Game development

- Documentation 2 Due
- Tournaments
- Onsite Documentation

#Botball

What to Expect During the Season

Before

- Register for the workshop
- Build Demobot
- Build Create Demo-Bot
- Build Board if possible

During

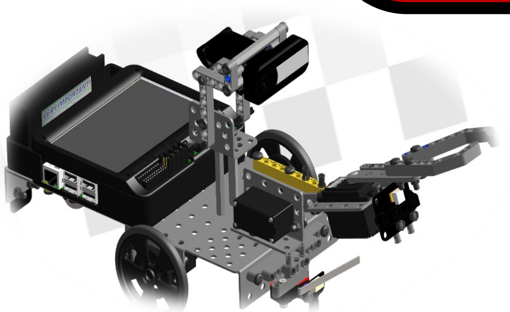
- Bring all materials
- Bring laptops for students (with chargers)
- Learn to Code
- Ask Questions
- Work on What You Need

After

- Break down tasks
- Continue to work on strategies
- Make timelines of Documentation Due Dates

Tournament

- On – site Presentation
- 3 Seeding Rounds
- Double Elimination
- Alliances



Professional Development Workshop

#Botball®

What to Expect at a Tournament

Practice
8:00 am - 9:00 am

→ **3 Seeding Rounds**

- Onsite Documentation

- After 3 seeding rounds go to lunch
- Judges will take 45 min. lunch after last seeding rounds

→ **Double Elimination**

- After 2 loses, sign up for Alliances
- Onsite Continues

→ **Finals**

- Alliances
- Awards will be immediately after finals are over
- Teams clean up pits
- Any help tearing down is greatly appreciated by KIPR staff

- **Practice:** teams test and calibrate robot entries on the official game boards
- **Seeding rounds:** teams compete against the task to score the most points
- **Double elimination (DE) rounds:** teams compete head-to-head
- **Alliance matches:** teams eliminated in DE pair up to score points *together*
- **Onsite documentation:** 8-minute technical presentation to judges

Navigating to Botball

Found at kipr.org -> Sign in -> Botball (top left corner) -> Team Resources (tab) -> Team Homebase

About Botball®? ▾ Schedule & Regions ▾ Team Resources ▾ Register for the Season! ▾ Sponsors

Botball Team Homebase

- About Botball®? → All about Botball
- Schedule & Regions → Regions, Event Calendar, Scores
- Team Resources → Curriculum, Team Homebase

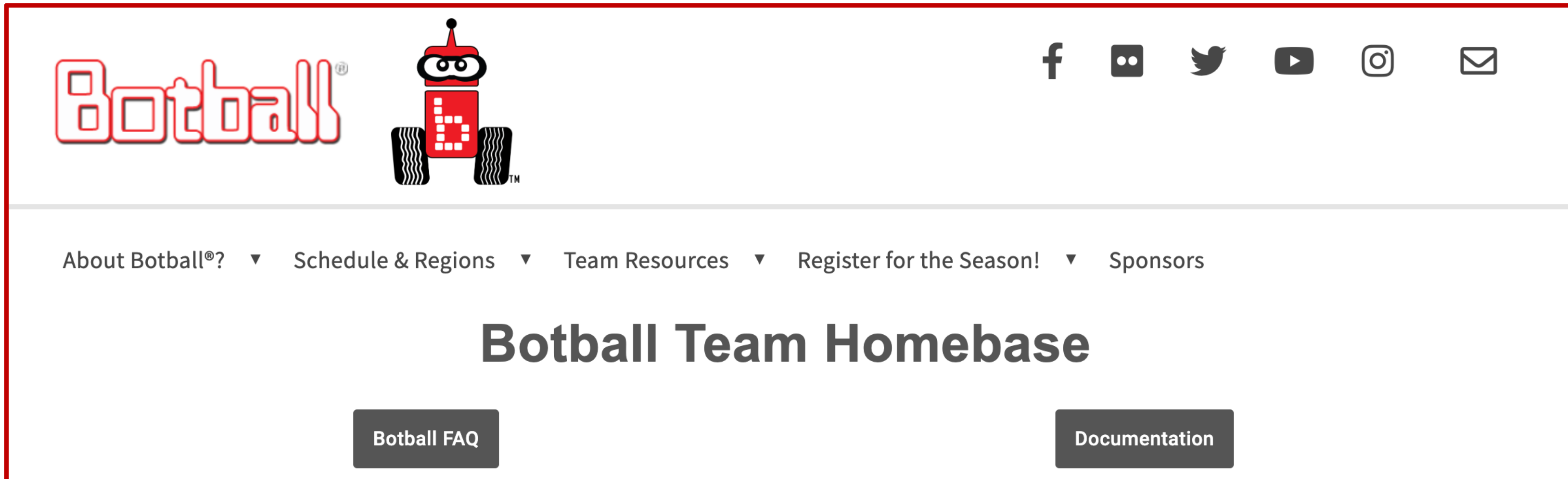
Team Homebase:

- Discord
- Documentation
- Tournament Schedule
- Announcements
- Game Documents
- Board Builds
- Registrations (Traveling, Worshops, Fall Game)
- Videos
- Resources and Curriculum

#Botball®

Navigating Team Home Base

Found at kipr.org -> Sign in -> Botball (top left corner) -> Team Resources (tab) -> Team Homebase



Find the Following:
- Discord
- Ask questions
- Get Answers
- Find help
- Discover ideas

Find the Following:
- Period 1, 2, 3, and onsite
- Grading Rubrics
- Exemplars
- Dates

Documentation

Due Dates are on the Documentation page

Period 1 Documentation:
- Part 1 – Organizing and Planning
- Part 2 - Quiz

Period 2 Documentation:
- TBD

Period 3 Documentation:
- Exact requirements are TBD

Onsite Presentation:
- (8 minutes) at the regional tournament

Why do we need to do documentation?
- To reinforce the Engineering Design Process
- Points earned in **Documentation** factor into the overall tournament scores!

GCER-2024

Global Conference on Educational Robotics

2024 - Concord, North Carolina

2025 - Norman, Oklahoma

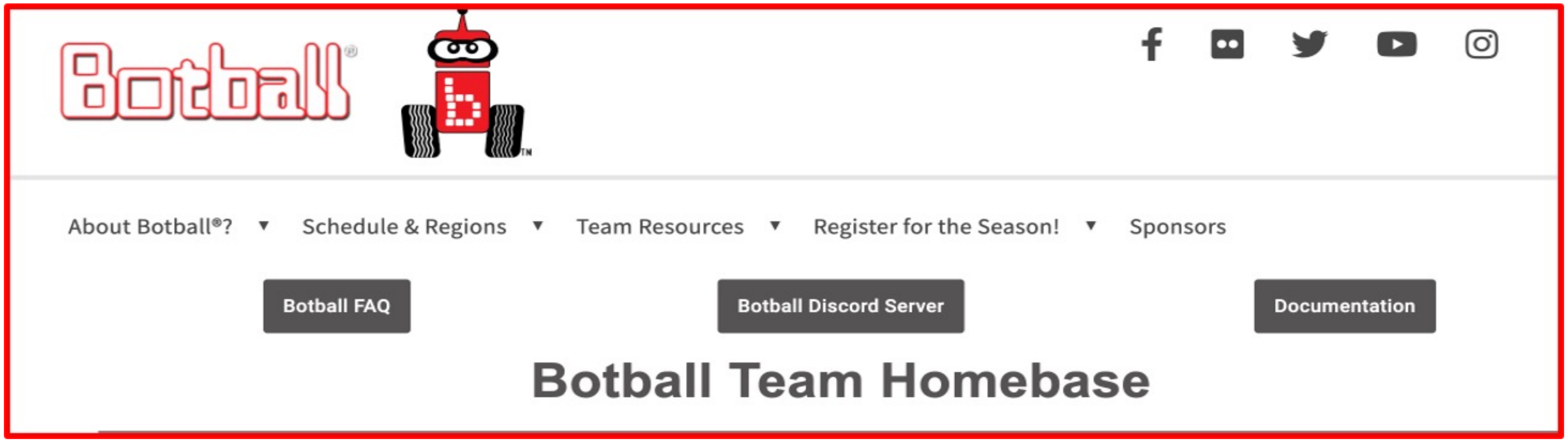
- Concord, North Carolina July 27-31, 2024- Embassy Suites
- International Botball Tournament
- Autonomous Robotics Showcase
- Aerial Botball Challenge
- International Junior Botball Challenge

kipr.org

- Meet and network with students from around the country and world
- Talks by internationally recognized robotics experts
- Teacher, student, and peer reviewed track sessions

Botball Team Home Base

Found at kipr.org -> Sign in -> Botball -> Team Resources -> Team Homebase



Index of Workshop Slides

- Botball Game Review
- Charging KIPR Robotics Controller
- Updating the Wombat
- Botball Overview
- Getting started with the KIPR Software Suite
- Explaining the "Hello, World!" C Program
- Designing Your Own Program
- Moving the DemoBot with Motors
- Moving the DemoBot Servos
- Making Smarter Robots with Sensors
- Motor Position Counters
- Making a Choice
- Line-following
- Tournament Code Template
- Fun with Functions
- Moving the iRobot *Create*: Part 1
- Moving the iRobot *Create*: Part 2
- Logical Operators
- Color Camera
- iRobot *Create* Sensors
- Resources and Support

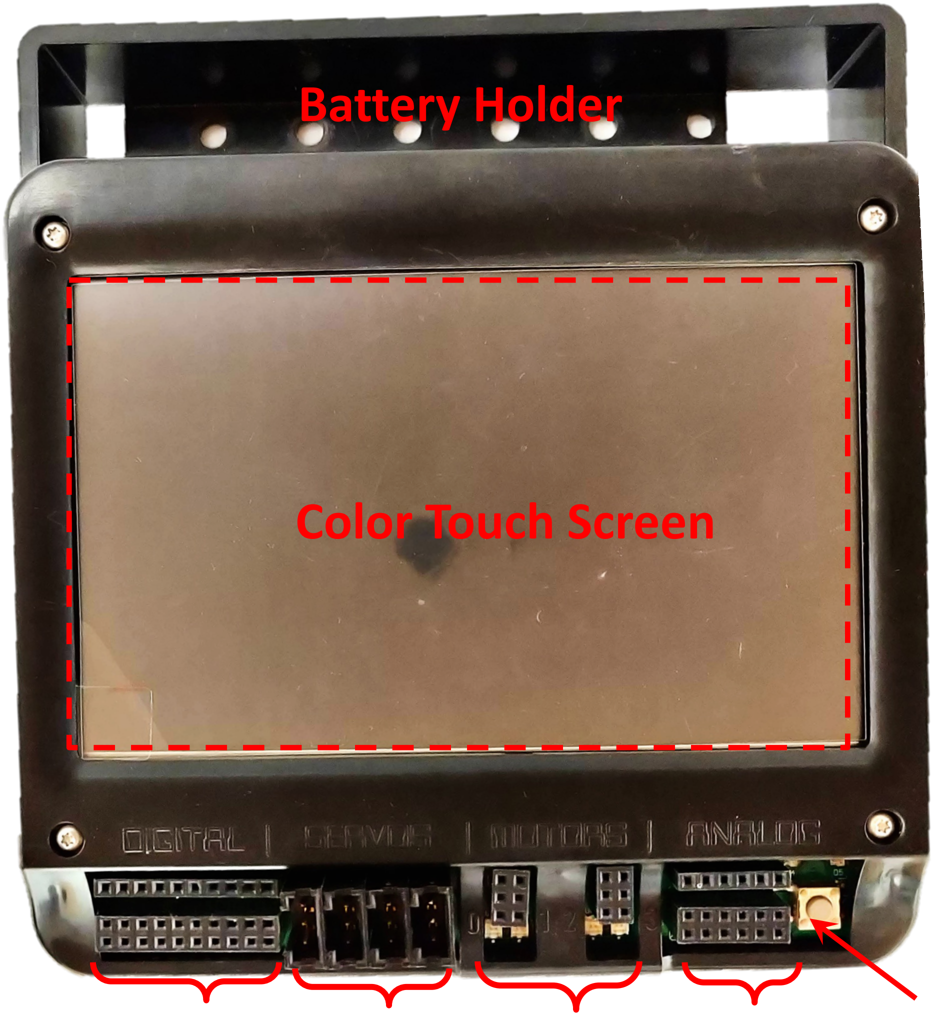
Quick Links

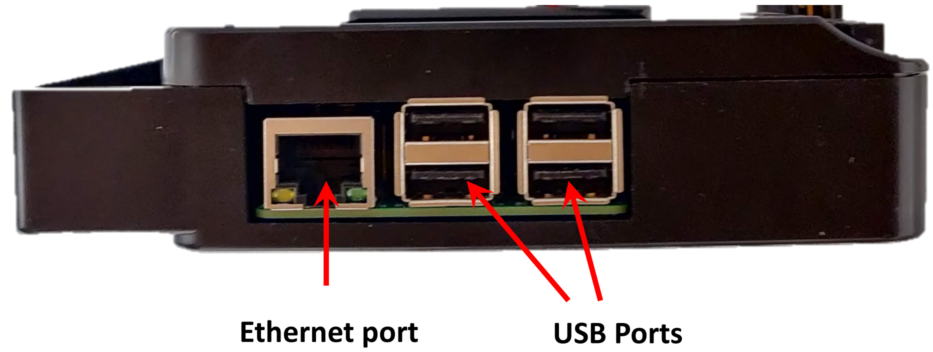
Can you do the following?

- Square up..... Slide 224
- Create ....Slide 290
- Slow down a servo.....Slide 254
- Following a line.....Slide 214
- Line follow with a Create.....Slide 287
- Start with light.....Slide 237
- Get motor position counter (drive to a certain distance).......Slide 177

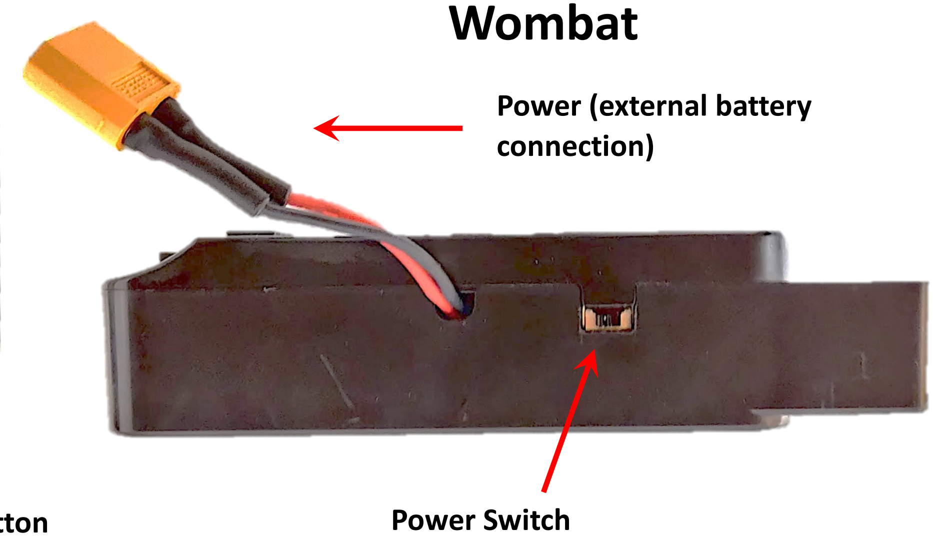
Wombat Controller Guide

Resource





KIPR Robotics Controller Wombat



Professional Development Workshop

#Botball®

Wombat Power

- The KIPR Robotics Controller – Wombat, uses an external battery pack for power.
  - It will void your warranty to use a battery pack with the Wombat that hasn't been approved by KIPR.

- Make sure to follow the shutdown instruction on the next slide. Failure to do so will drain your battery to the point where it can no longer be charged. If you plug your battery into the charger and the blue lights continue to flash then you have probably drained your battery to the point where it cannot be charged again. You can purchase a replacement battery from https://www.kipr.org/

Charging the Controller's Battery

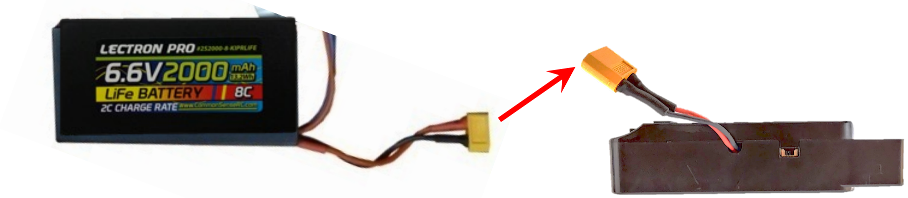
- For charging the controller's battery, **use only the power supply which came with your controller, see next slide for connections**.
  - **It is possible to damage the battery by using the wrong charger or excessive discharge!**

- The standard power pack is a **lithium iron (LiFe) battery**, a safer alternative to lithium polymer batteries. The safety rules applicable for recharging any battery still apply:
  - **Do <u>NOT</u> leave the battery unattended** while charging.
  - Charge in a cool, open area away from flammable materials.

Making the Connection

All connections are as follows:

- **Yellow to Yellow** (battery to controller)



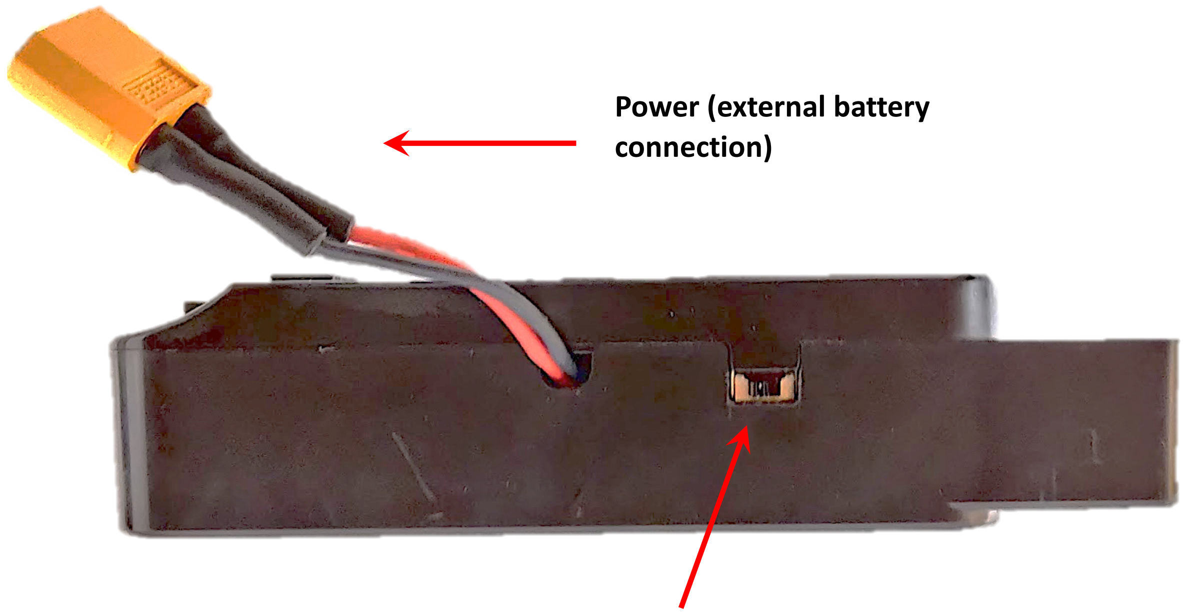
- **White small to White small** (charger to battery)
    - Yours may vary slightly, **use caution** unplugging

- **Black to Black** (motors, servos, sensors)

#Botball®

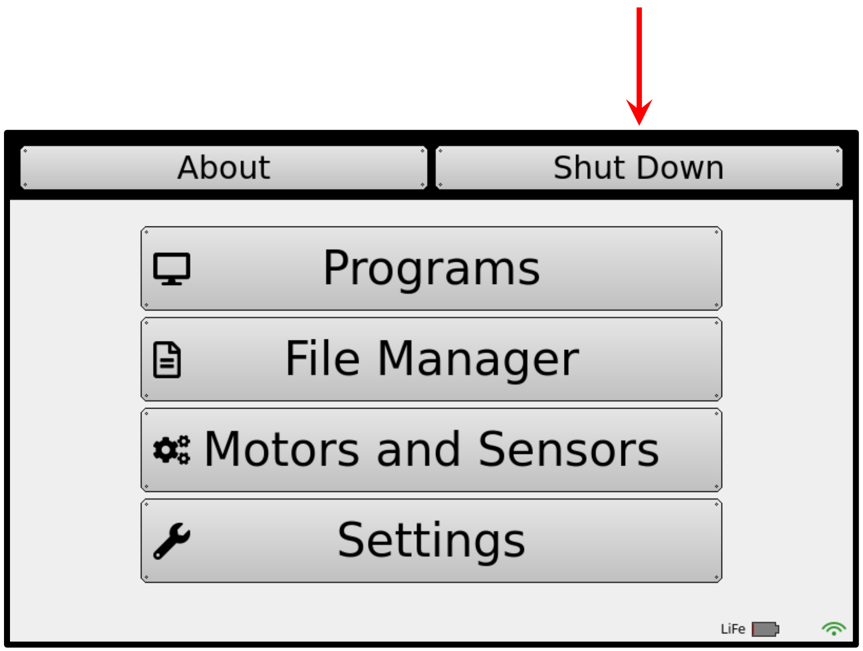
Wombat Powering On

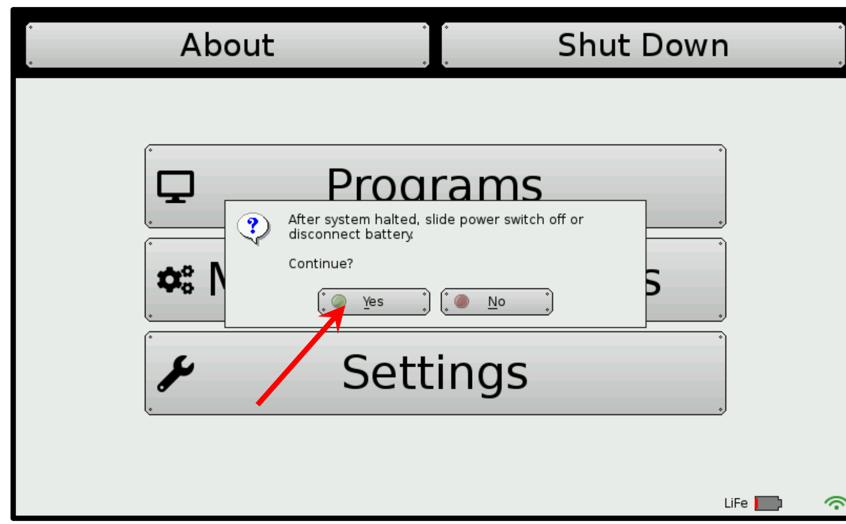
- ***The power switch is located on the side of the Wombat controller next to the external battery connection***



Wombat Power Down

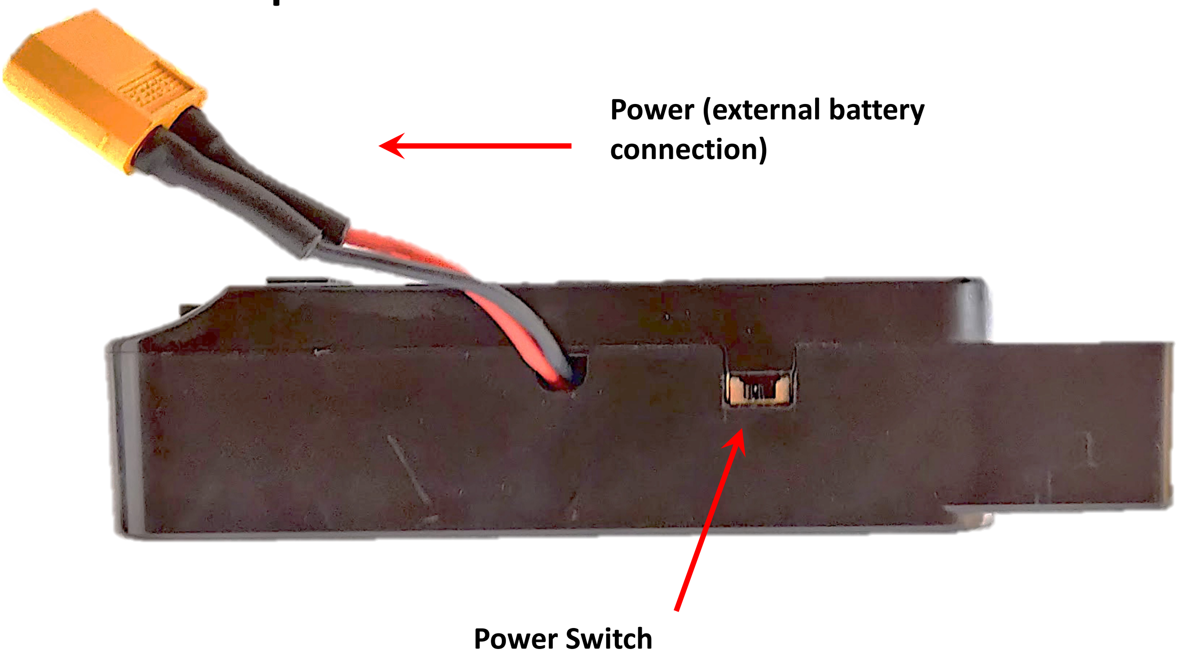
- From the Wombat Home Screen press *Shutdown*
  - Select *Yes*





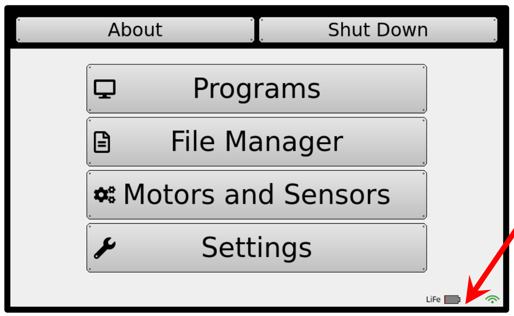
Wombat Power Down

- ***After shutting down from the main home screen***, slide the power switch to off AND unplug the battery; use/grab the yellow connectors, being careful not to pull on the wires



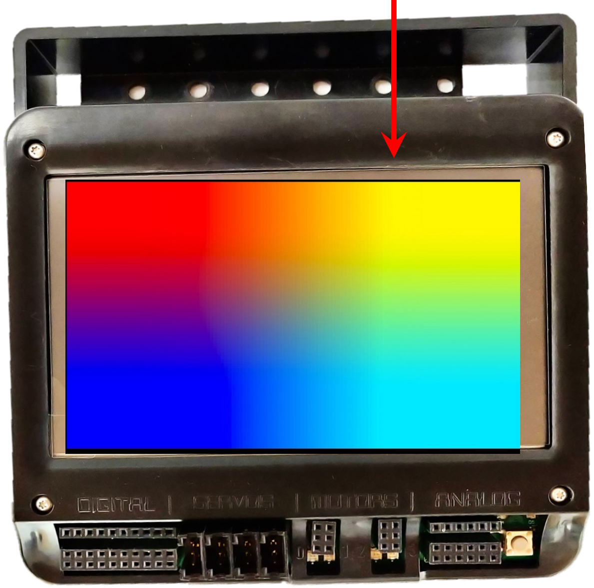
Battery Level

- The KIPR Robotics Controller – Wombat has a color keyed battery indicator (green-yellow is ok, red is getting low.
- A Yellow LED that will flash when the battery is critically low
- If it gets too low you may see this rainbow screen
  - If this happens, turn the Wombat off and replace the battery with one that is charged



Battery doesn't reflect actual life

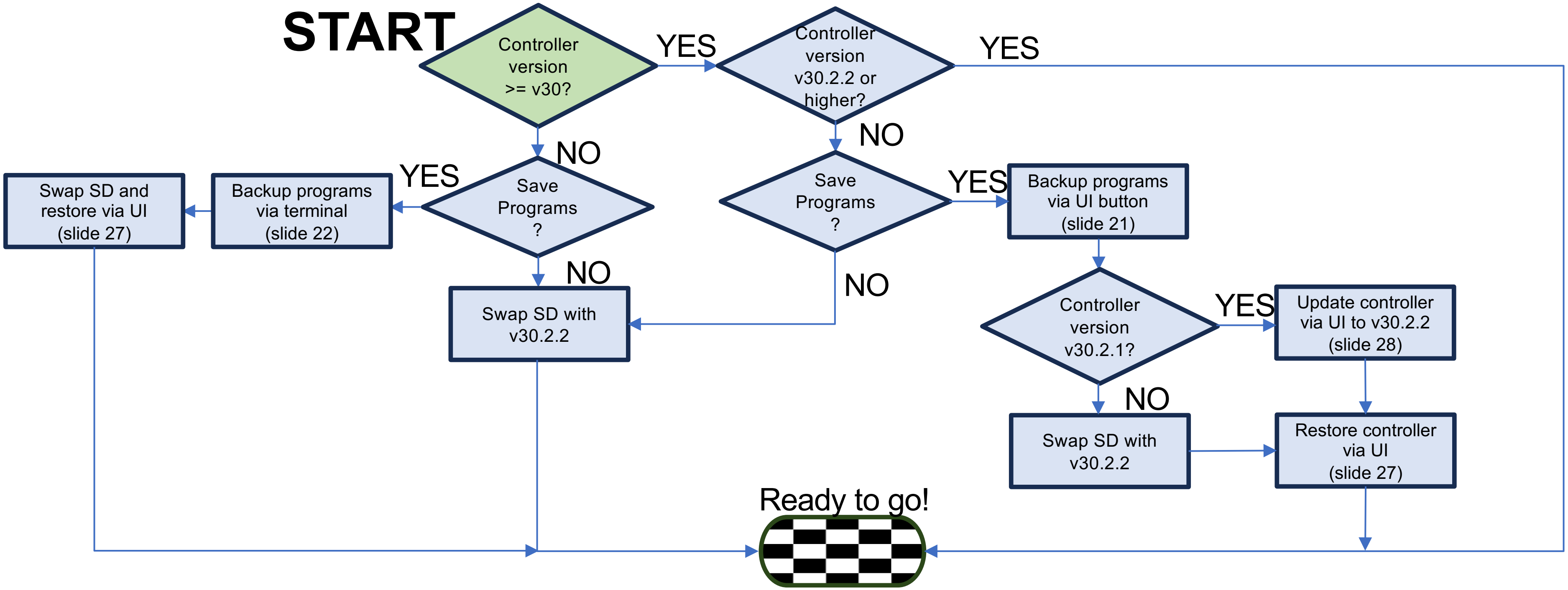
Yellow LED, visible only when battery is critically low



Professional Development Workshop

Update Wombats to 30.2.2

To update your Wombat Bring it to the KIPR Staff, who will follow this flowchart to determine the correct process:

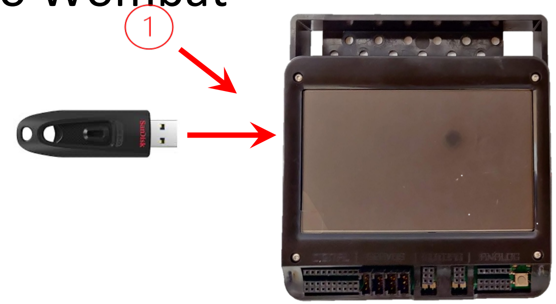


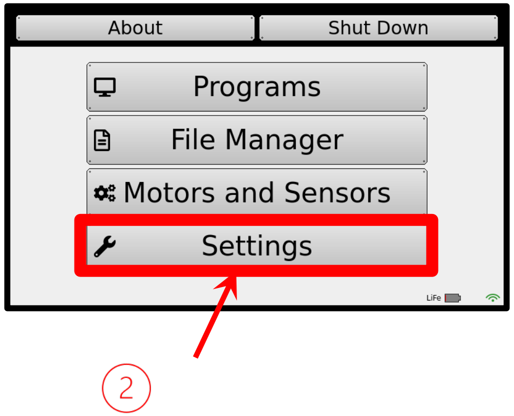
Professional Development Workshop

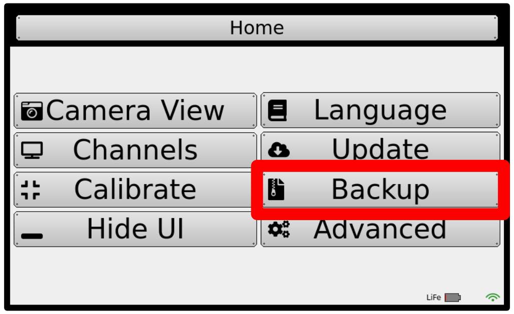
#Botball®

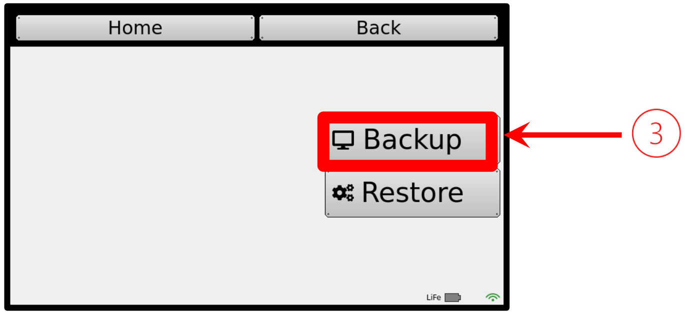
Backup Programs via Botui

1. Boot up and insert USB drive into Wombat
2. Select Settings
3. Select Backup
4. Click Backup







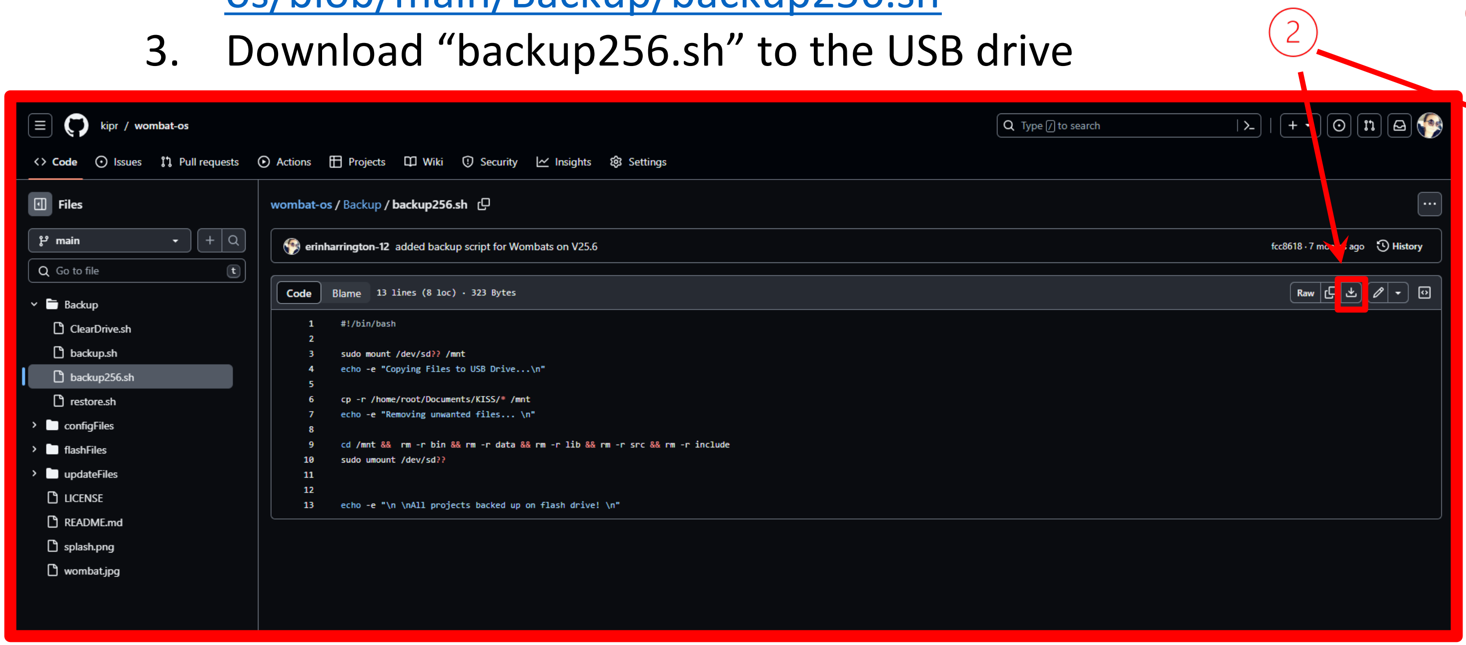


#Botball®

Backup Programs via terminal

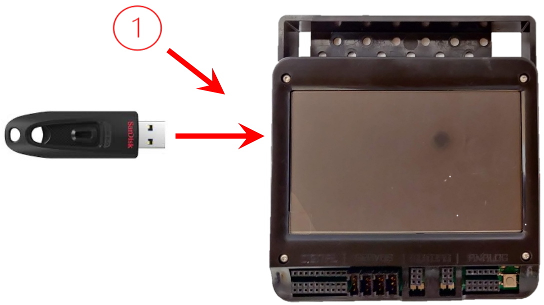
NOTE: If you are not comfortable with working in the terminal, ask a KIPR staff member

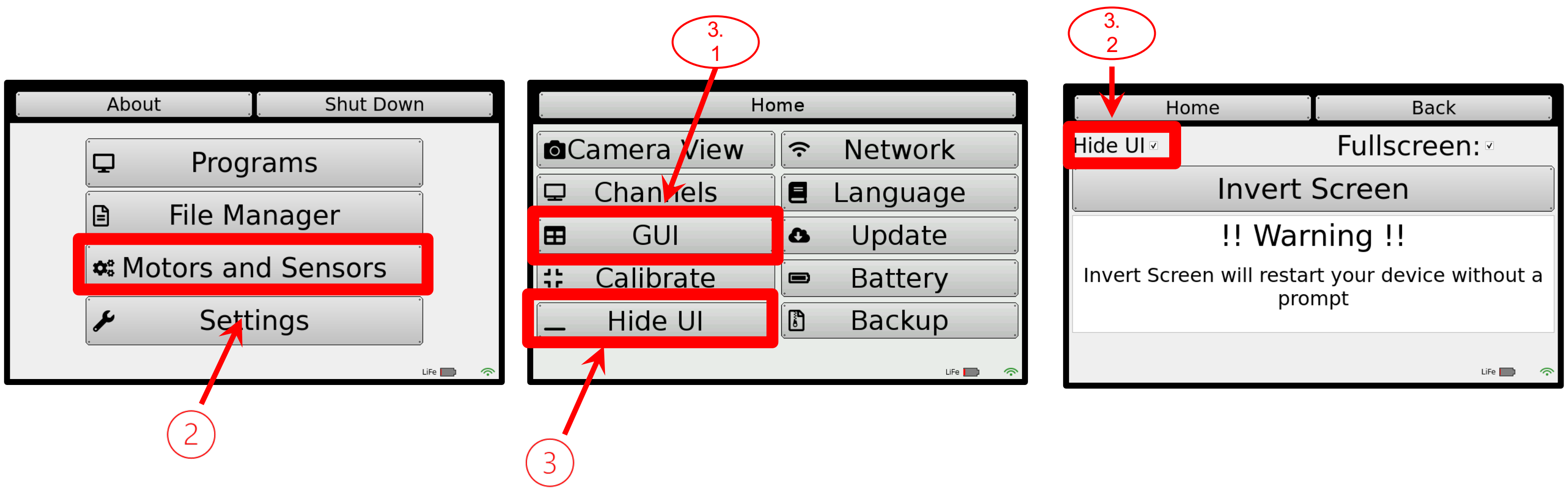
1. Insert USB drive into laptop/computer
2. Go to https://github.com/kipr/wombat-os/blob/main/Backup/backup256.sh
3. Download "backup256.sh" to the USB drive



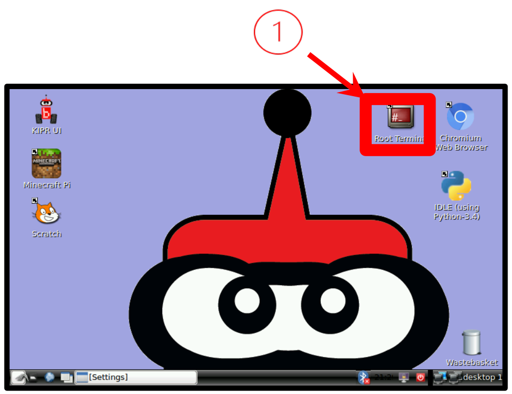
Backup Programs via terminal

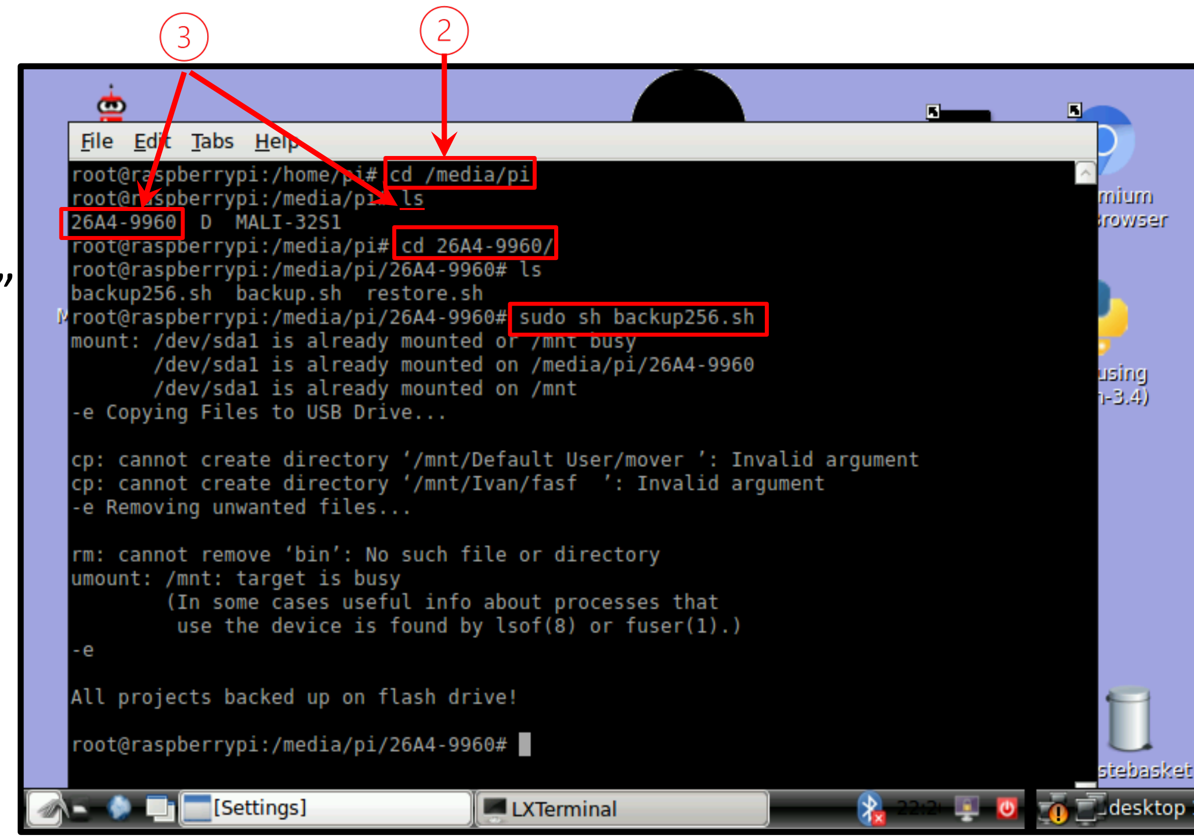
1. Boot up and insert USB drive into Wombat
2. Select Settings
3. Select Hide UI
    1. If you don’t see this option, select GUI
    2. Check the “Hide UI” box
4. Click Backup





1. Double click “Root Terminal”
2. Type “cd /media/pi”
3. Type “ls” to find USB name. Example shows “26A4-9960”
4. Type “cd *your_usb_name*”
5. Type “sudo sh backup256.sh”





Professional Development Workshop

#Botball

Backup Programs via terminal

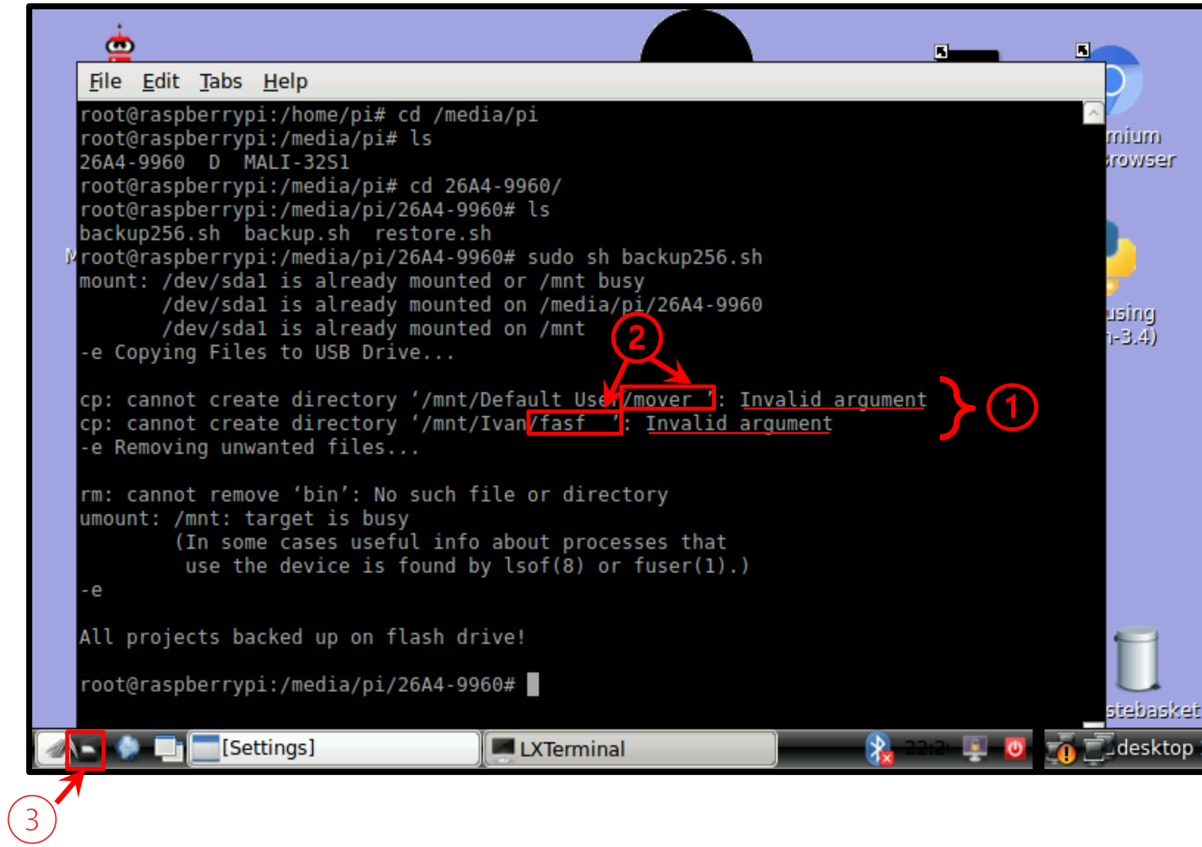
1. Notice there are two “Invalid argument” returns
2. This is because there are spaces after the program name

This is why you are not to Use special characters or spaces in program names

SOLUTION:

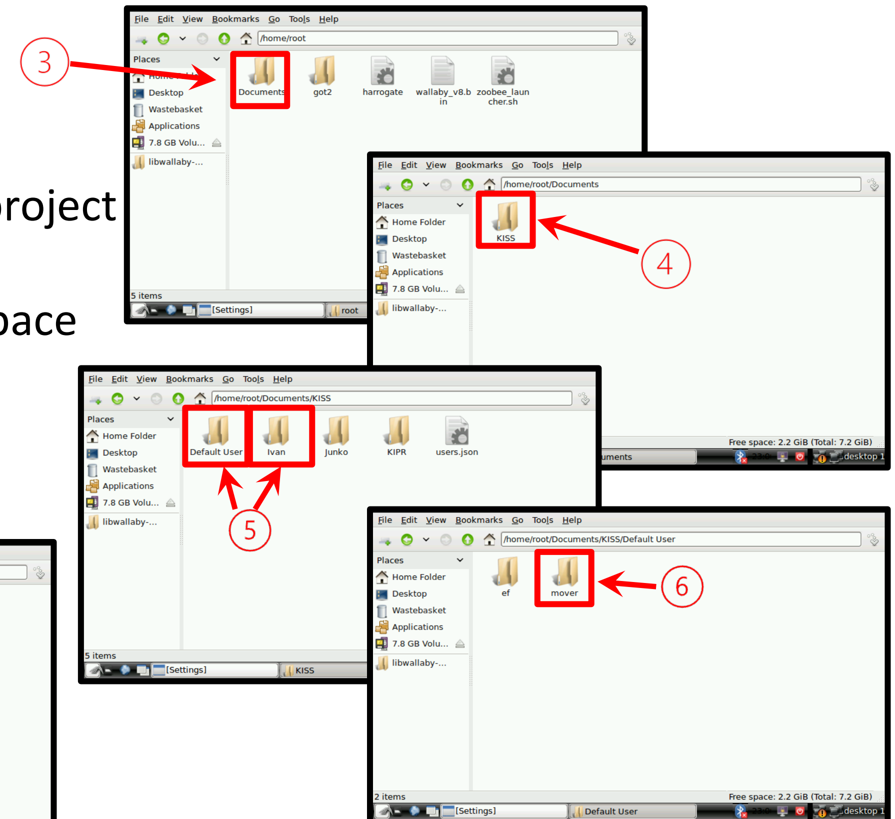
1. Click “File Manager”

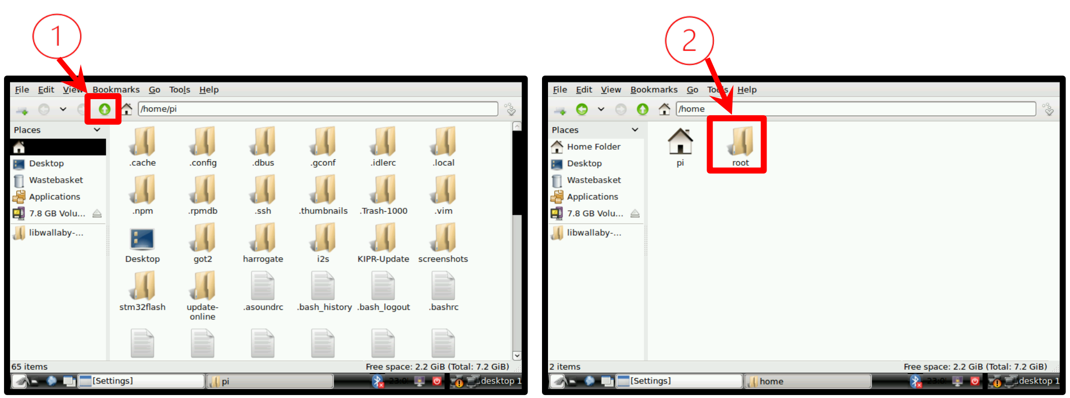
Continue on next slide



Backup Programs via terminal

1. Click the up arrow
2. Double click “root”
3. Double click “Documents”
4. Double click “KISS”
5. Double click the User that had the misnamed project
   Example shows Default User and Ivan
6. Right click on misnamed project and remove space
   See previous slide terminal for specific projects

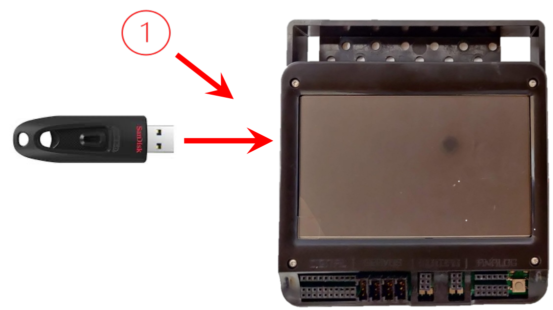


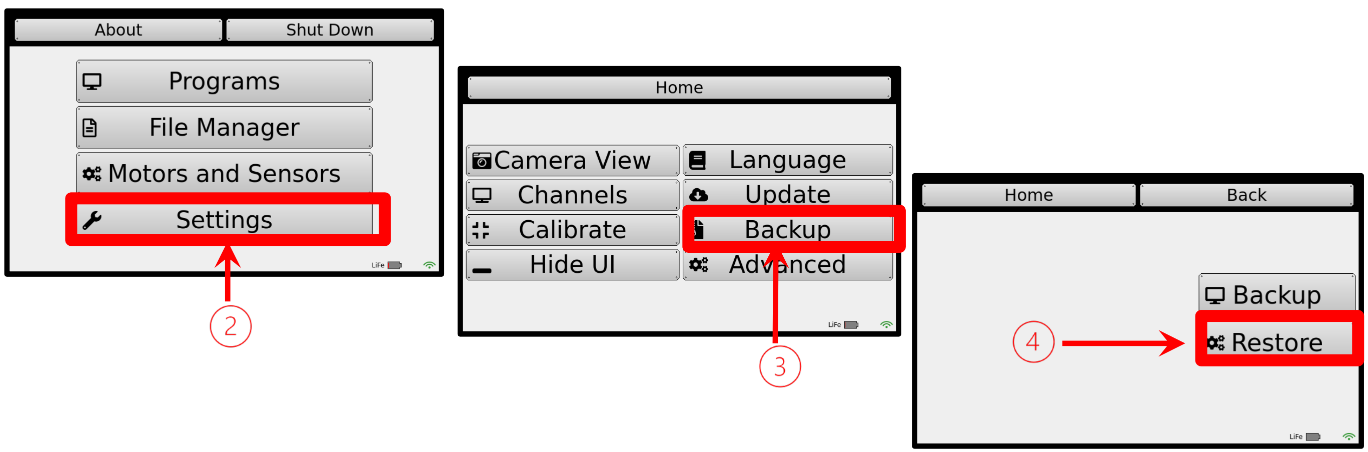


#Botball

Restore Programs via Botui

1. Insert a flash drive into one of the USB ports on the Wombat that has your programs on it (that you have previously backed up)
2. Select "*Settings*"
3. Select *"Backup"*
4. Select "*Restore*"
5. Wait for Restore Complete Message



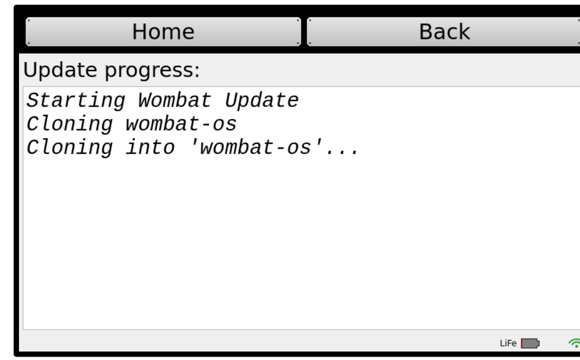


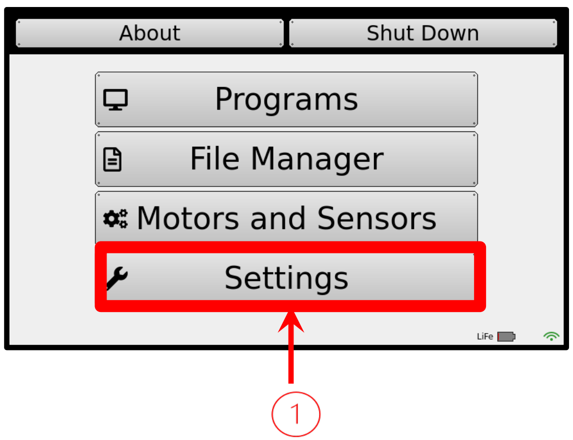
Professional Development Workshop

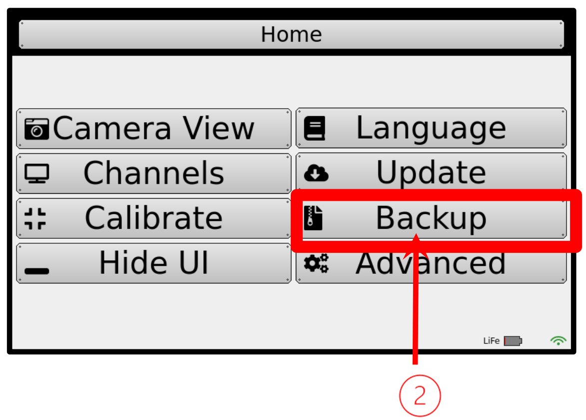
#Botball

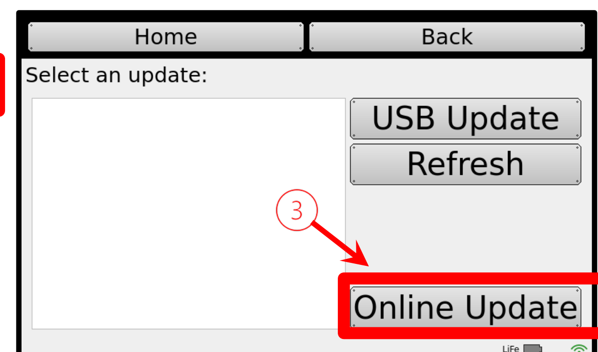
Update Wombat via Botui

1. Select *“Settings”*
2. Select *“Update”*
3. Be sure to be connected to Wi-Fi via Client Mode (slide 40) or connected by ethernet
4. Select *“Online Update”*





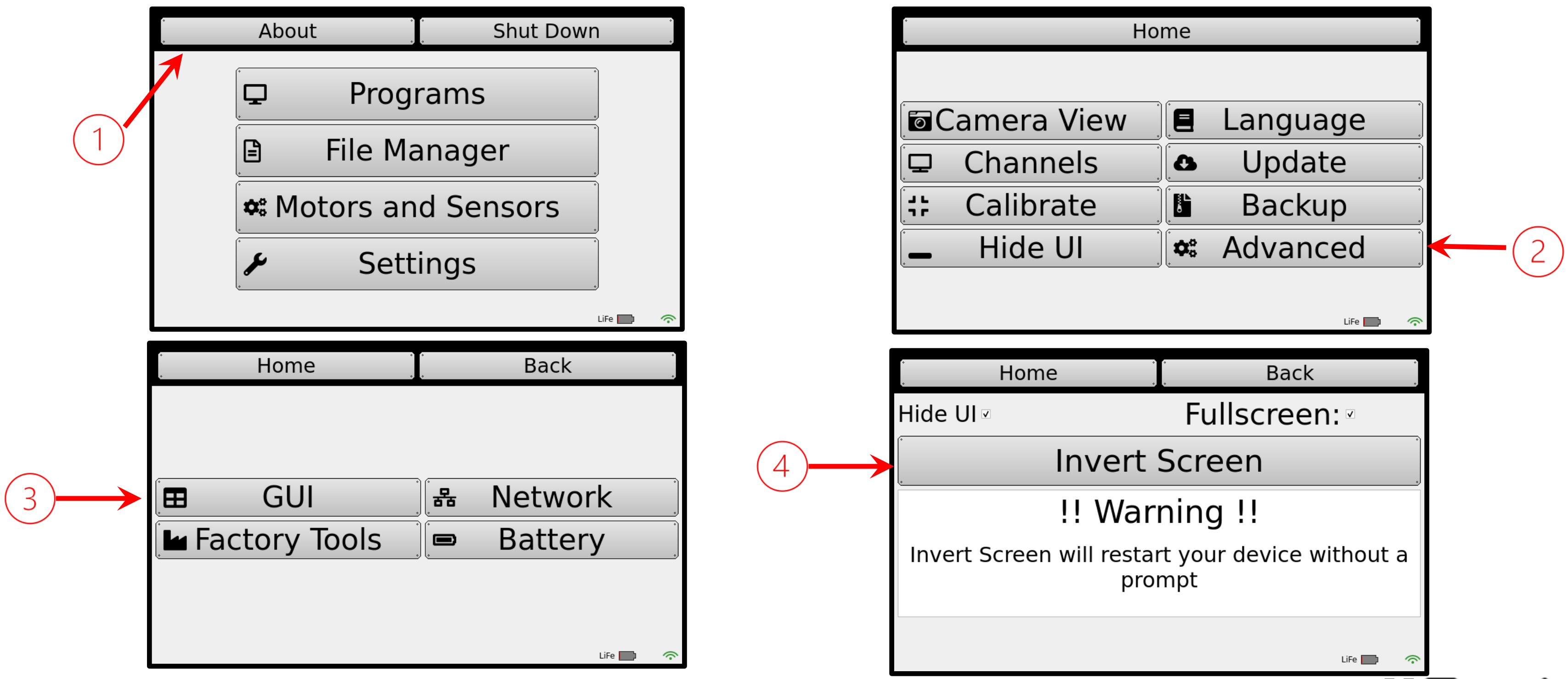




Professional Development Workshop

Inverting your screen

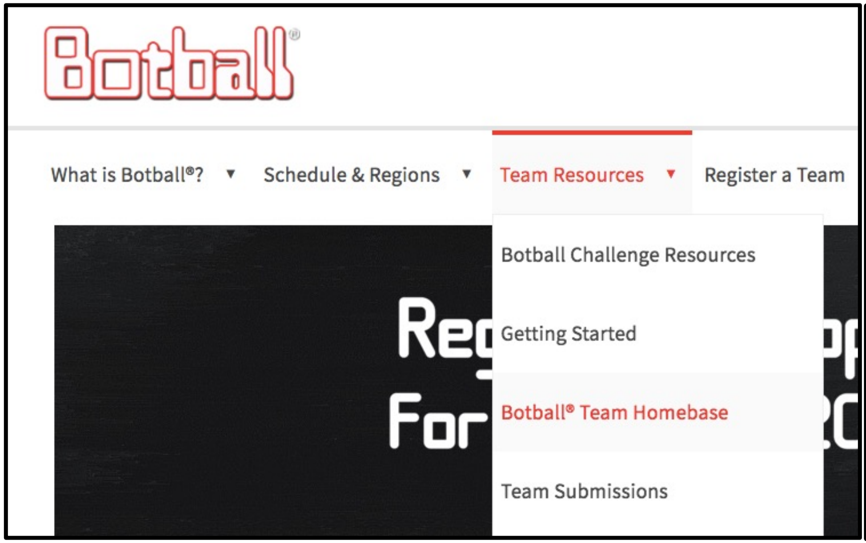
1. You can invert your screen
2. Select “*Settings*”
3. Select “*Advanced*”
4. Select “*GUI*”
5. Select “*Invert Screen*”

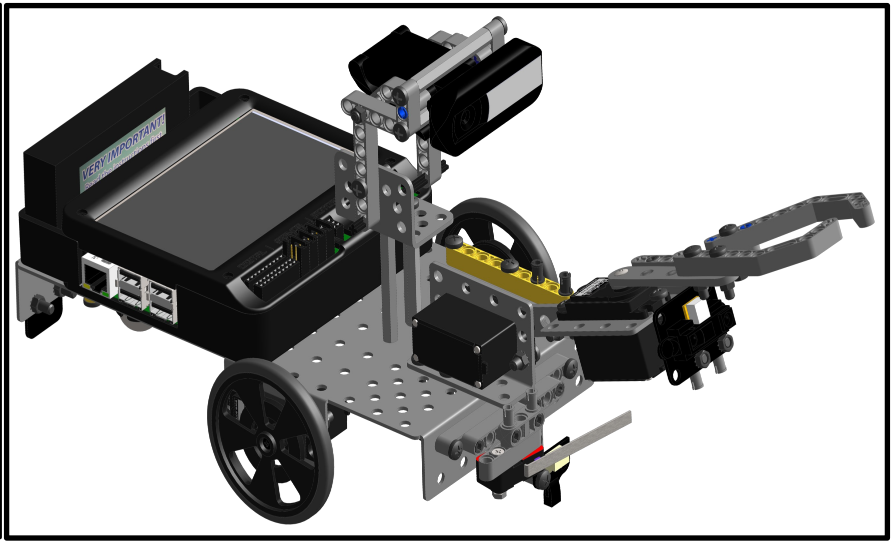


Build the DemoBots

Build your robot using the DemoBot Building Guide

(This can be found on your desktop. Also accessible via your Botball account: kipr.org/Botball -> **Sign in** -> Team Resources -> Team Homebase)





***Must be signed into your Botball team account to view the Team Homebase.**

#Botball®

Join Us Online!

Like & Follow Us Today!

@botballrobotics

/BotballRobotics

@botballrobotics

@botballrobotics

KIPR.org

#Botball

#KISSInstitute

TM

Botball Overview

What and When?

GCER

What is Botball?

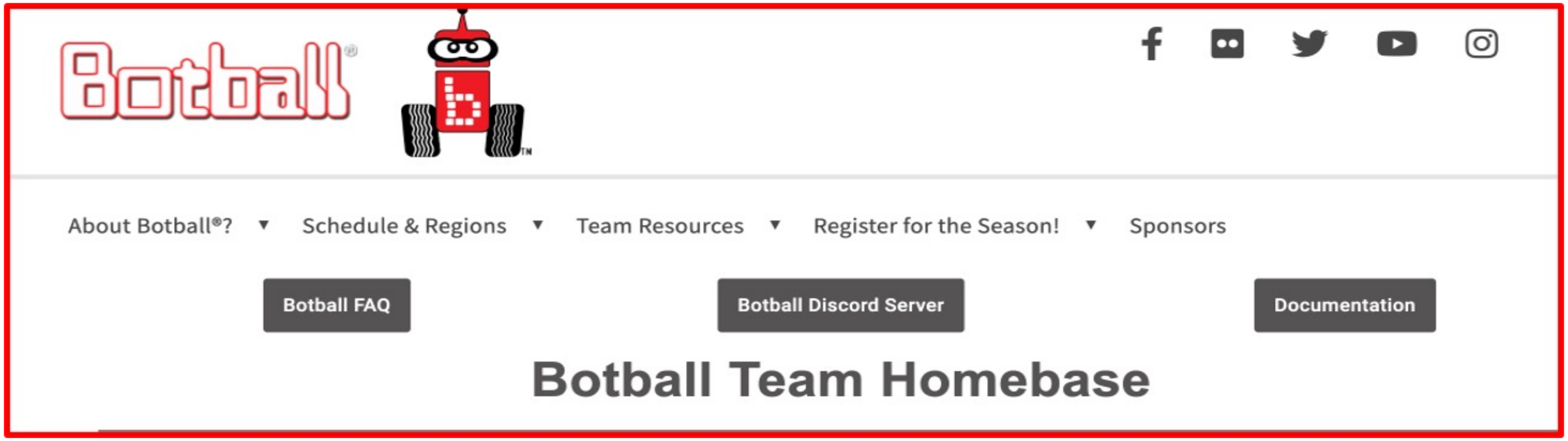
- Produced by the **KISS Institute for Practical Robotics (KIPR)**, a non-profit organization based in Norman, OK.

- Engages middle and high school aged students in a **team-oriented robotics competition** based on **national education standards**.

- By **designing**, **building**, **programming**, and **documenting** robots, students use **science**, **technology**, **engineering**, **math**, and **writing** skills in a **hands-on project** that reinforces **their learning**.

Botball Discord: How-To

Found at kipr.org -> Sign in -> Botball -> Team Resources -> Team Homebase



Professional Development Workshop

#Botball

Botball Student Discord

What is Discord?

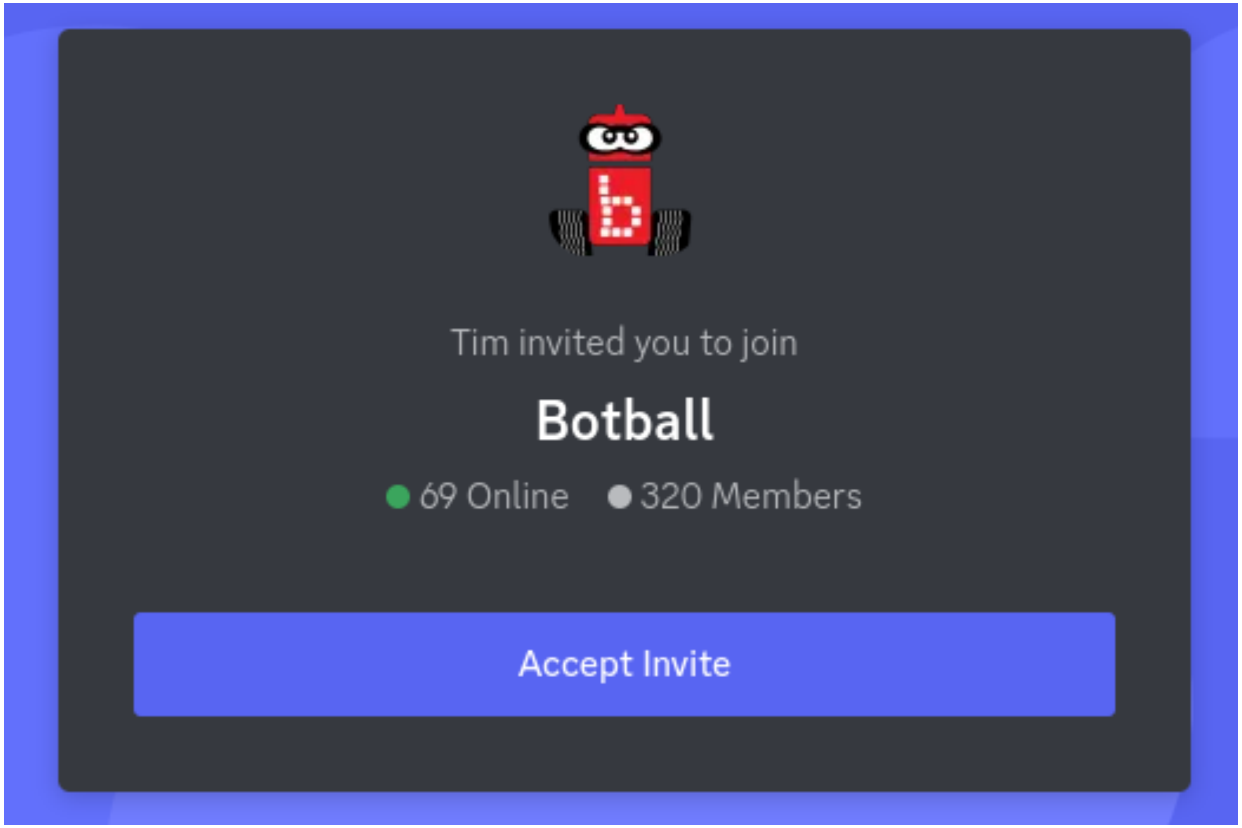
- Discord is a messaging social platform that gives users the ability to communicate with voice calls, video calls and messaging

What is the Botball Student Discord?

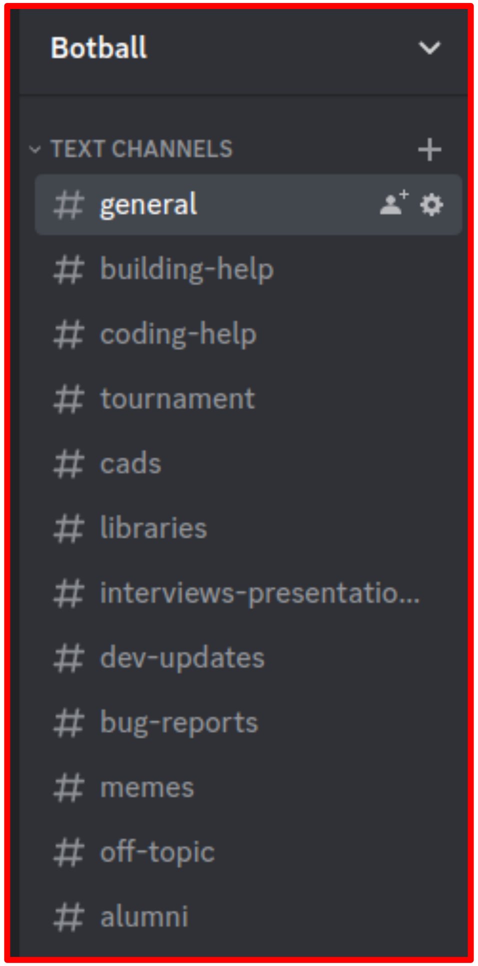
- The Botball Student Discord is for Botball students and coaches to connect with others in the activity
- **The official point of contact for KIPR GAME FAQ**

Click "Accept Invite"

Note: You may have to create a Discord account

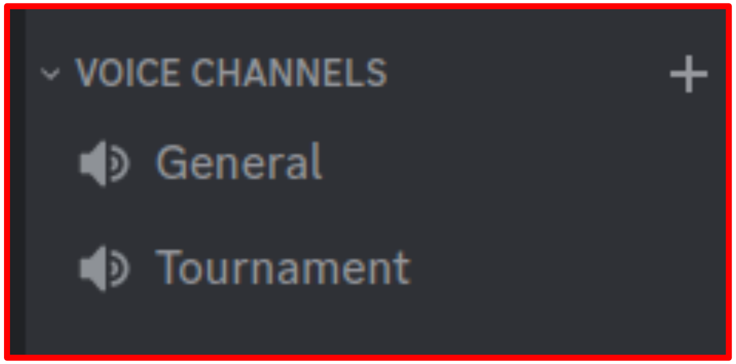


Botball Discord: How-To



- Each Text Channel has a separate messaging space for specific topics of conversation
- This directory is located on the left side of the page

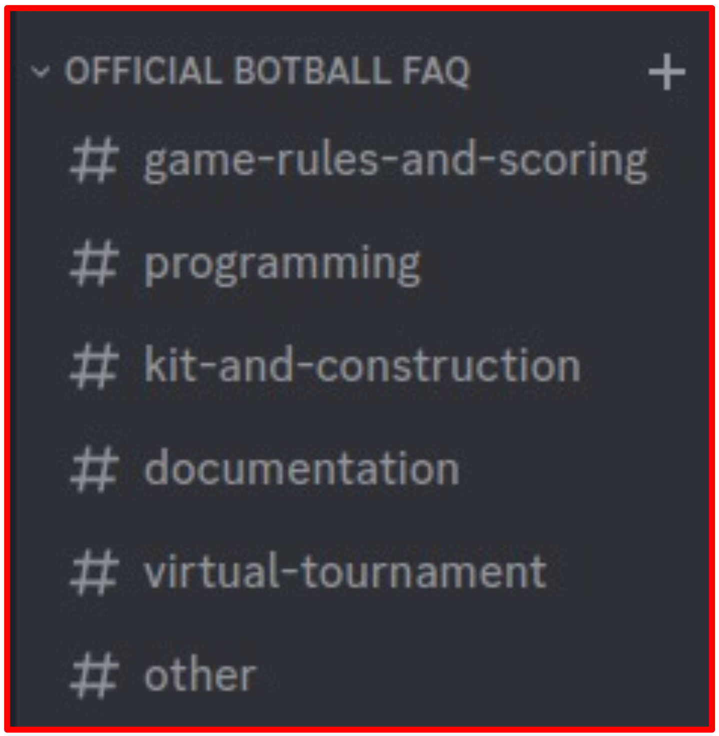
Botball Discord: How-To



- Each <u>Voice Channel</u> has a separate voice chatting space for specific topics of conversation
- This directory is located on the left side of the page

#Botball

Botball Discord: How-To



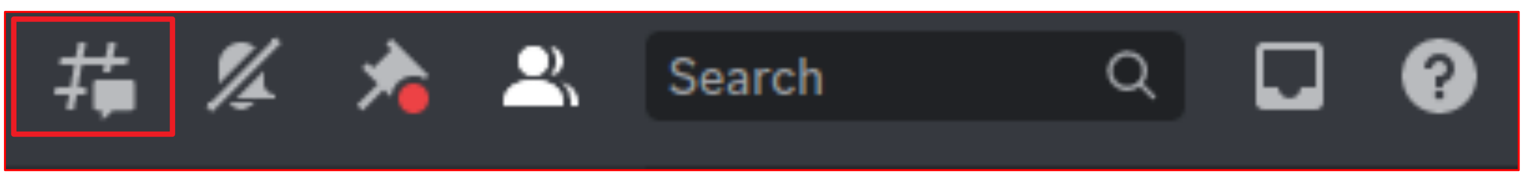
- The Official Botball FAQ section is for asking specific questions and viewing other questions previously asked
- Each channel is dedicated to that specific topic
- This directory is located on the left side of the page

Botball Discord: How-To

Before asking a question in each channel, search existing threads using keywords from your question

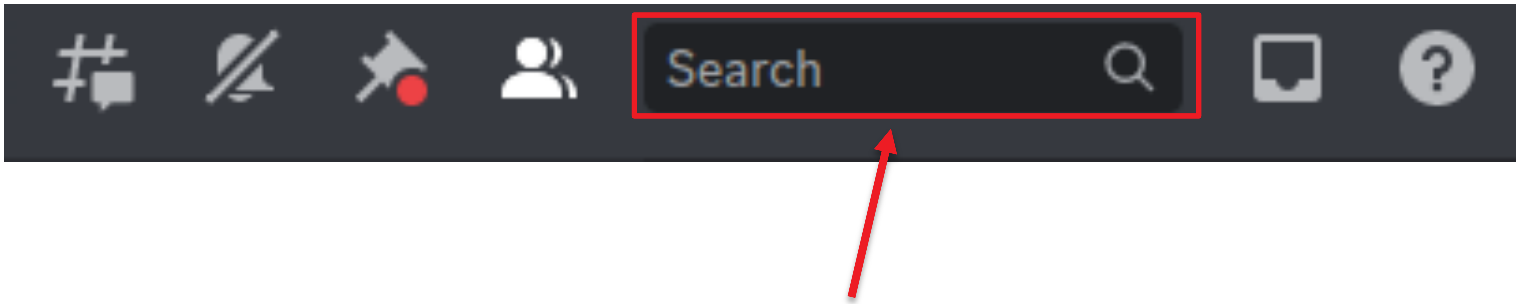
To ask a question, create a Thread

The Threads icon is located at the top right of the page



#Botball®

Botball Discord: How-To



- You can also search topics using keywords in the search bar
- The search bar is located at the top right of the page

Getting Started with the KIPR Software Suite

What is a programming language?

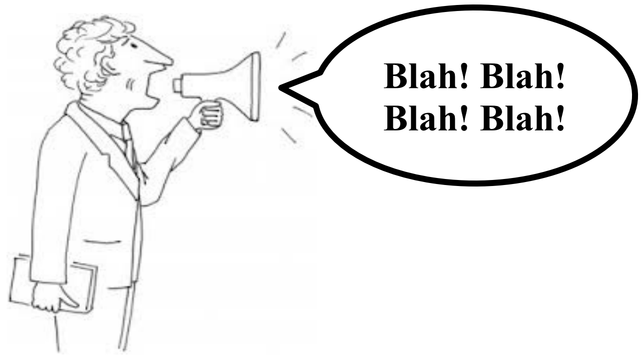
How can I create new projects and files?

How can I write and compile source code?

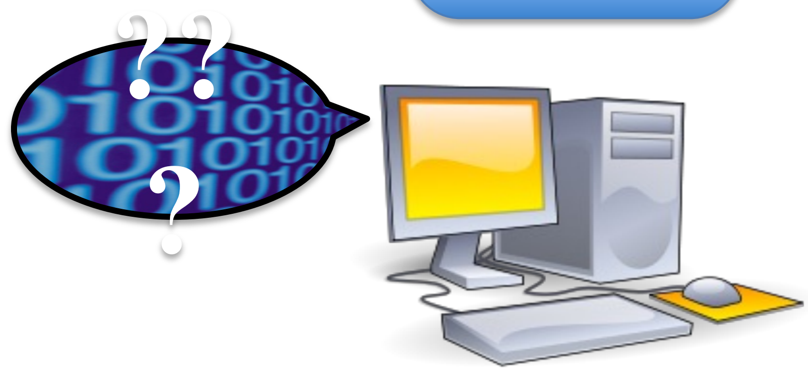
How can I run programs on the KIPR Wombat?

What is a *Programming Language*?

Human

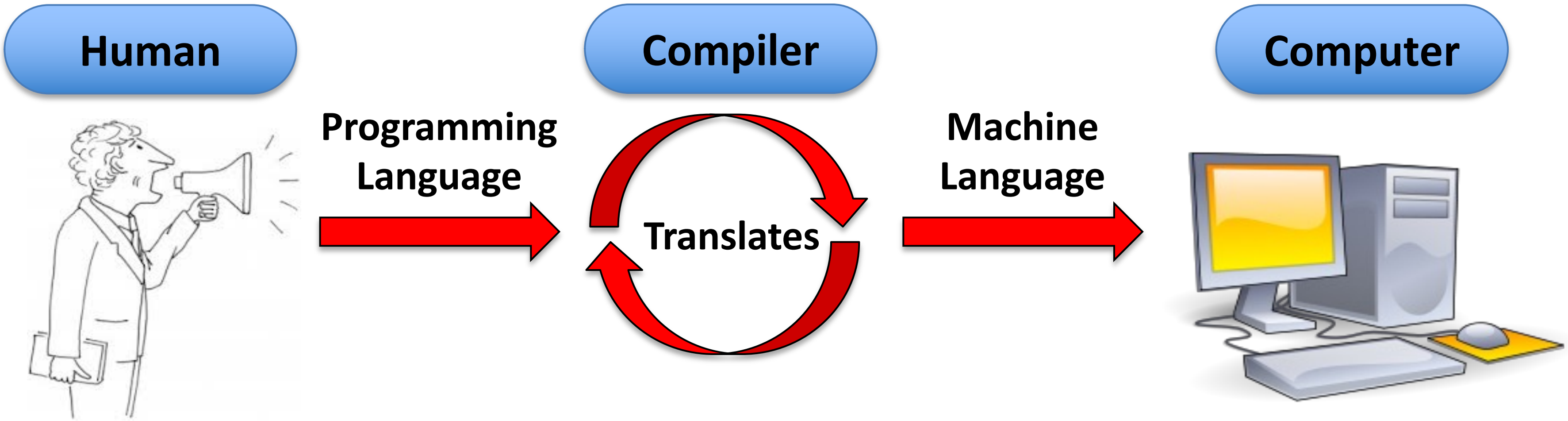


Computer



- **Computers** only understand **machine language** (stream of bytes), which computers can **read and execute** (run).
- Unfortunately, **humans** don't speak **machine language**...

What is a *Programming Language?*



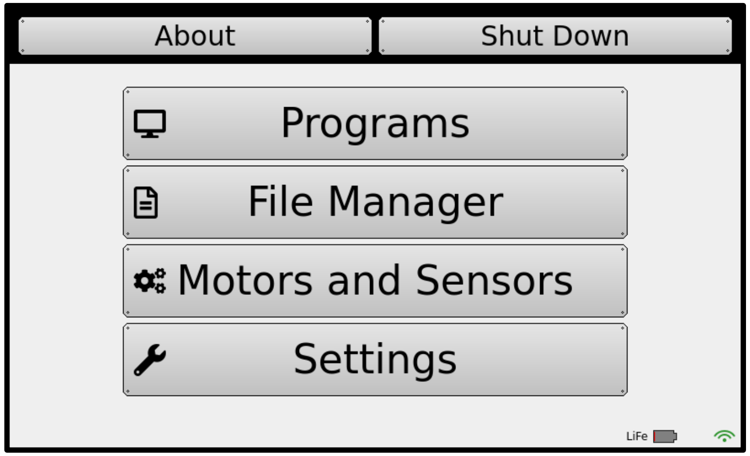
- **Humans** have created **programming languages** that allow them (humans) to write "**source code**" that is easier for them (humans) to understand.
- **Source code** is **compiled** (translated) by a **compiler** (part of the **KIPR Software Suite**) into **machine language** so that the **computer** can **read and execute** (run) the code.
- Programming languages have funny names (C, C++, Java, Python, …)

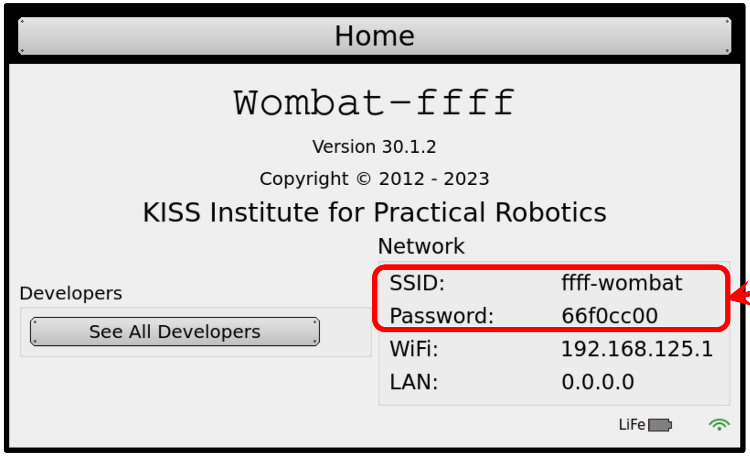
Connect the Wombat to your Computer, Smart Phone or Tablet at School

- Connect the **Wombat** to your Browser device via Wi-Fi

1. Turn on the Wombat with the **black switch on the side** *(after turning on, wait until you see your Wombat as available to connect to with your device Wi-Fi. This should take a minute or so)*

2



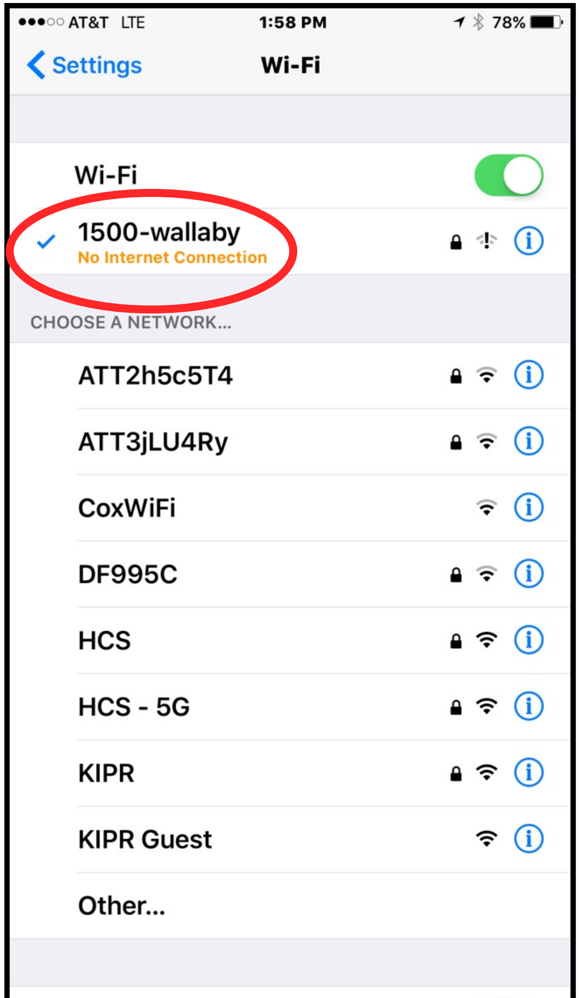


2

2. Use the info (Wombat SSID # and Password), from the **about** page, to connect via Wi-Fi.

#Botball

Connection



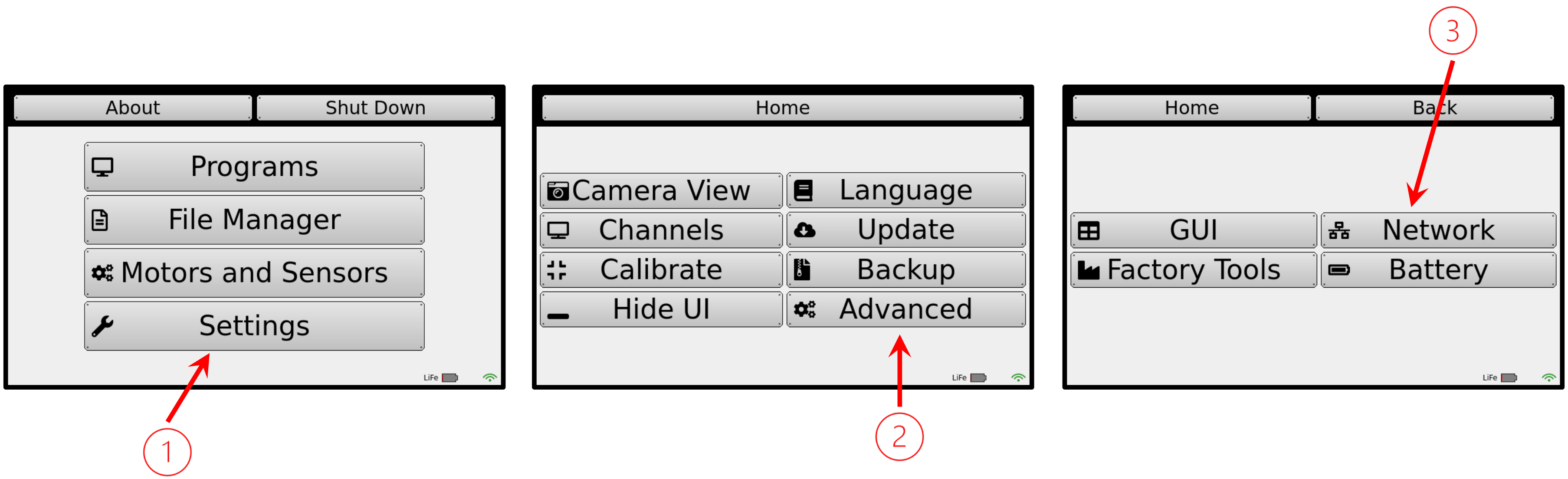
When you are connected to your Wombat, your device may give various errors; "***no internet connection***" or "***connected with limited***"

<u>This is normal.</u> Proceed with opening a browser and connecting to the KISS IDE.

Switching Wombat to Client Mode

For more advanced users, you can put your Wombat in client mode:

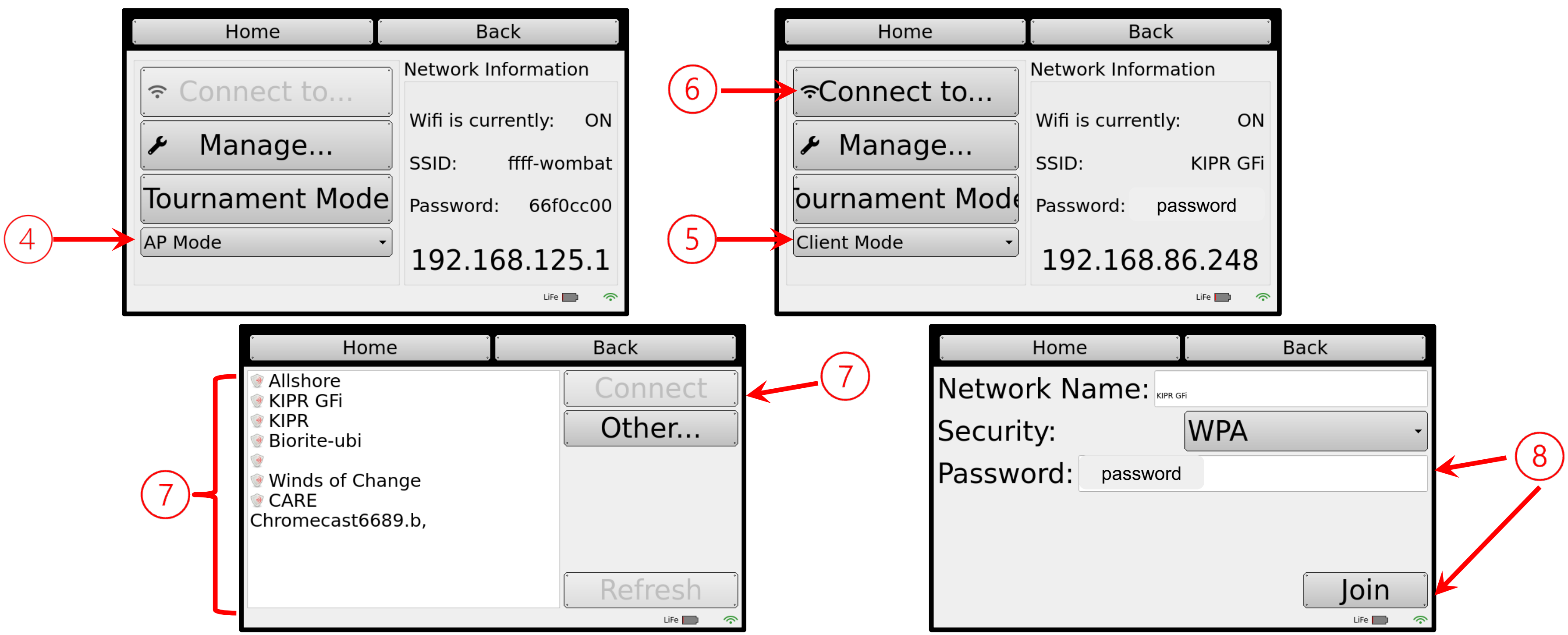
1. Select *"Settings"*
2. Select *"Advanced"*
3. Select *"Network"*



#Botball

Switching Wombat to Client Mode

4. Select drop down box that says *"AP Mode"*
5. Change selection to *"Client Mode"*
6. Select *"Connect to…"*
7. Select desired network, then select *"Connect"*
8. Enter password for given network, then select *"Join"*



Professional Development Workshop

#Botball

Loading the Starting Web Page (Wi-Fi)

1. Launch a web browser such as Chrome or Firefox (Internet Explorer **will not work**).
2. Copy this IP address into your browser's address bar followed by ":" and port number 8888; e.g.,

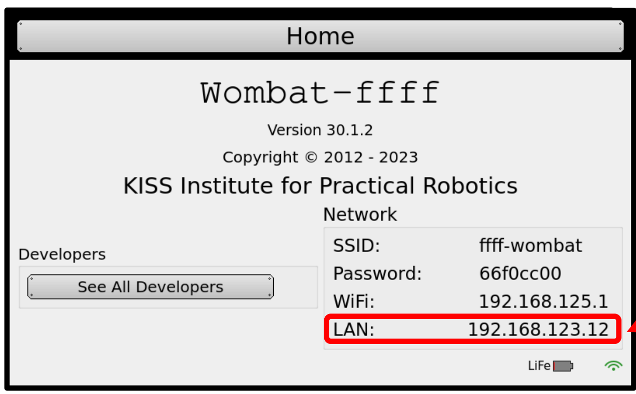
$$\underbrace{\mathbf{192.168.125.1}}_{\text{IP address}}\mathbf{:}\underbrace{\mathbf{8888}}_{\text{Port \#}}$$

3. The user interface for the package will now come up in your browser.
4. You may use a computer, tablet or even a smart phone through Wi-Fi.
    1. Optionally you may use an ethernet cable (instead of Wi-Fi).

Connection Troubleshooting

- If you have trouble connecting your Wombat via Wi-fi, try ethernet:

1. Plug ethernet cord from Wombat to ethernet switch, then an ethernet cord from ethernet switch to computer



$$\underbrace{192.168.123.12}_{\text{LAN address}}:\underbrace{8888}_{\text{Port \#}}$$

2. Copy this LAN address into your browser's address bar followed by ":" and port number 8888; e.g.,
3. *Note: this LAN address will appear when plugged into ethernet switch to computer*

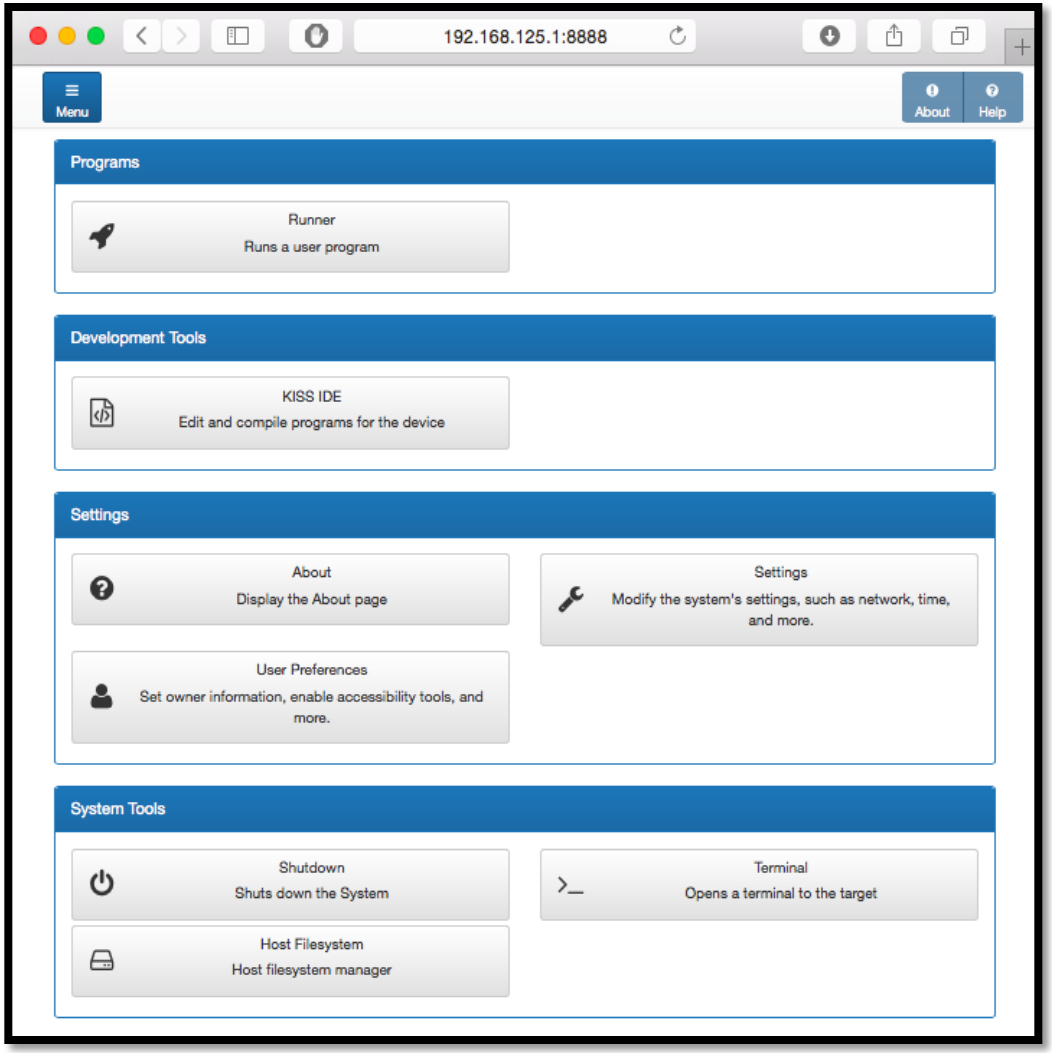
Professional Development Workshop

#Botball®

Using the KIPR Integrated Development Environment (IDE)

To make it easier for you to learn and use a programming language, KIPR provides a web-based **Software Suite** which will allow you to write and compile source code using the **C programming language**.

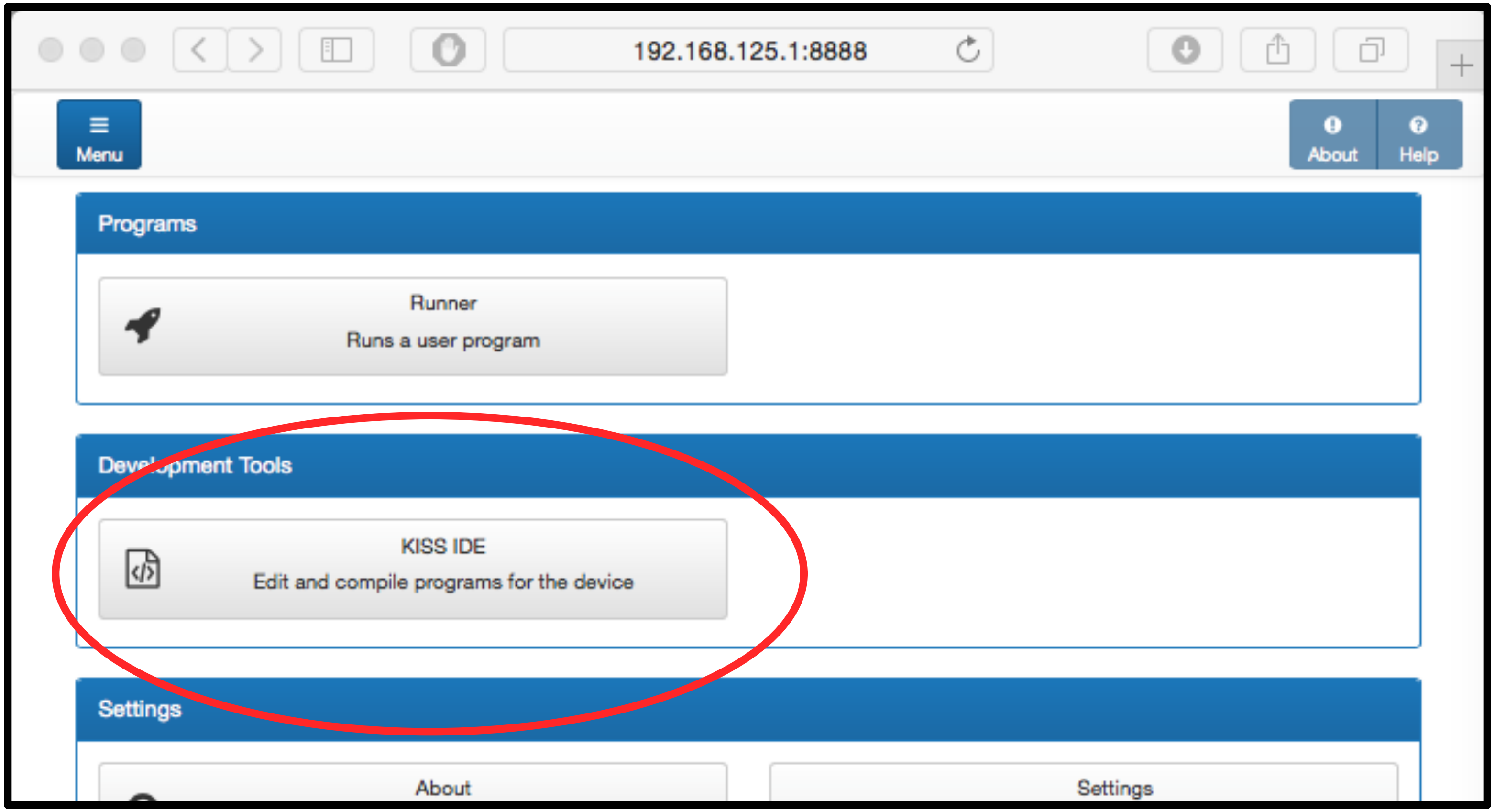
The development environment will work with almost any web browser **except Internet Explorer**.



#Botball®

Creating a Project

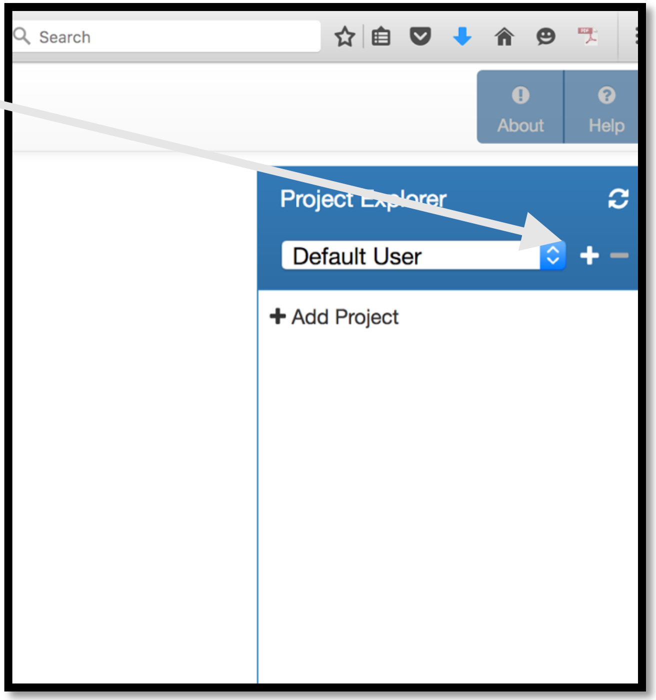
1. Click on the ***KISS IDE*** button.

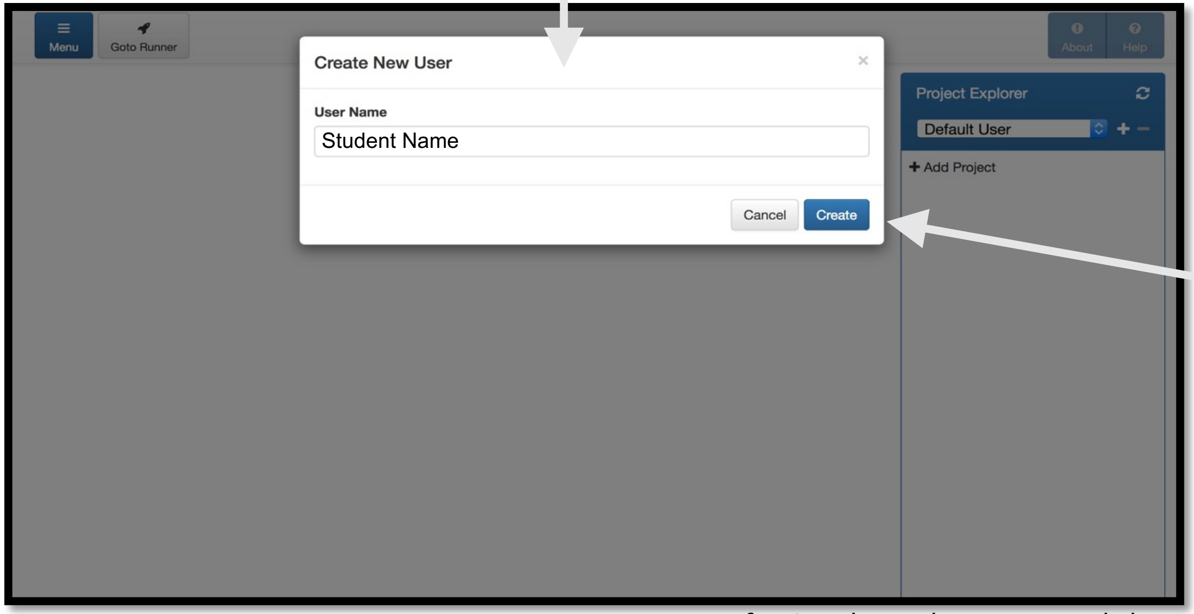


NOTE: The buttons might be in different locations depending on device type and screen size.

Creating a User Folder

1. Add a new user folder by clicking the **+** sign in the **Project Explorer**.
2. Name your new user folder by the student's name to help organization. All of your different projects will go into this user folder. ****No special characters allowed in name.***

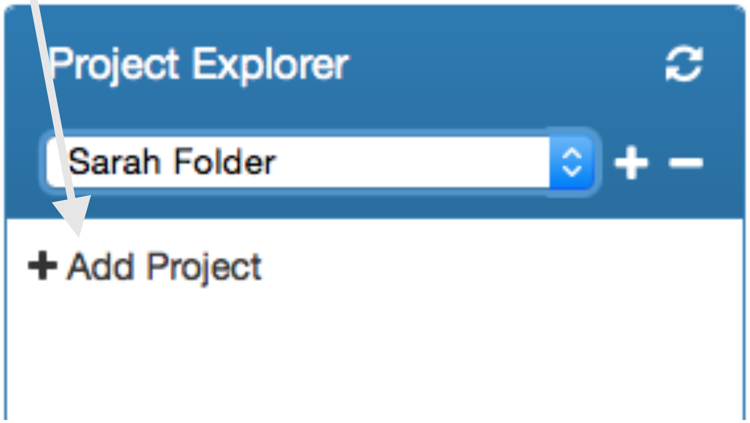


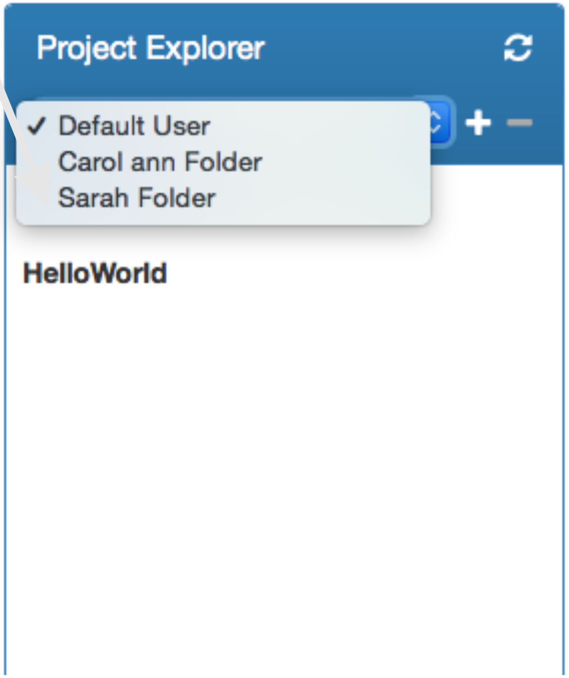


3. Click ***Create*** to complete.

Creating a Project

1. Go back to **Project Explorer** and select the **User Name** you created from the drop down. This is the folder you created.
2. Click ***+Add Project***. You are adding a project to your folder.

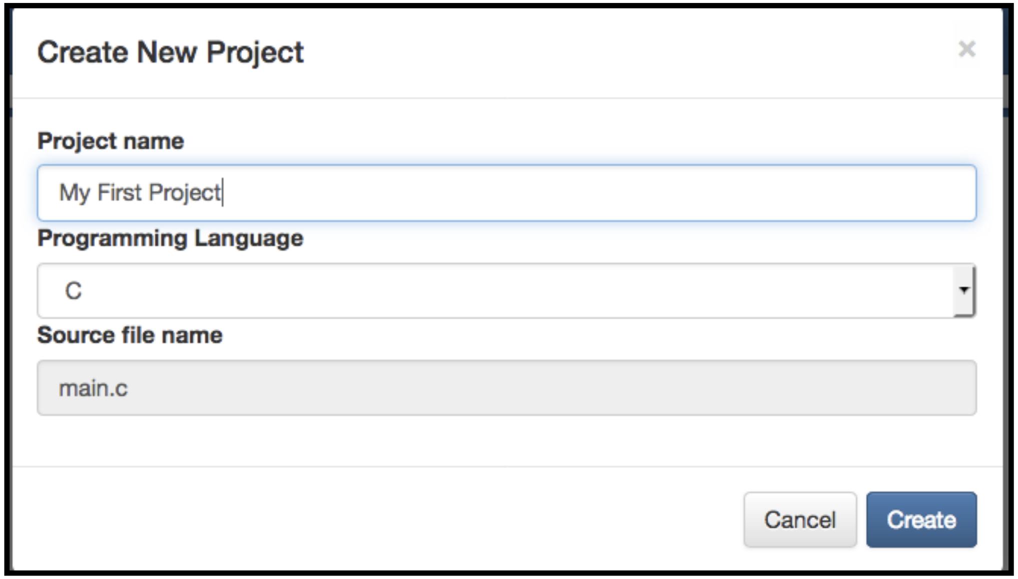




#Botball®

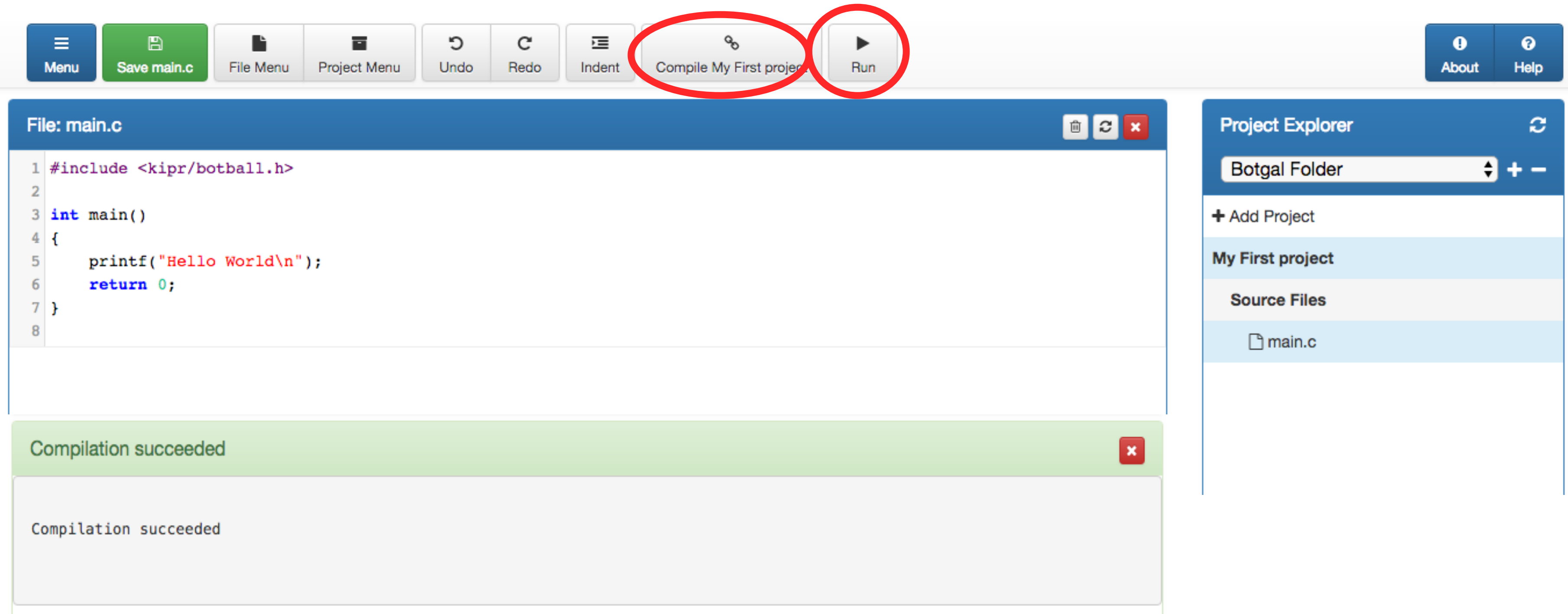
Creating a Project

1. Give your project a **descriptive name**
   - **Note:** you will have a lot of student's projects, so consider using their first name followed by the name of the activity.
   - <u>***No special characters allowed in name. . , / ? $ # % ~ _ -&* or emojis***</u>
2. Press the ***Create*** button



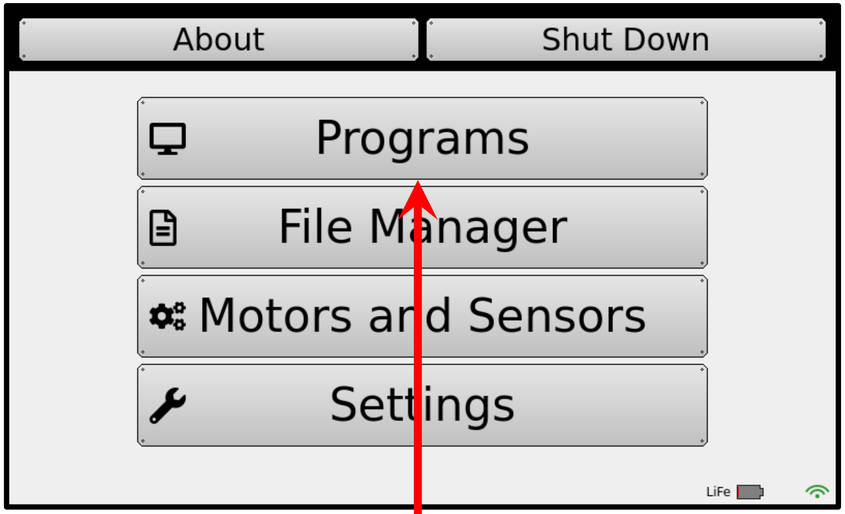
Compile and Run a Project

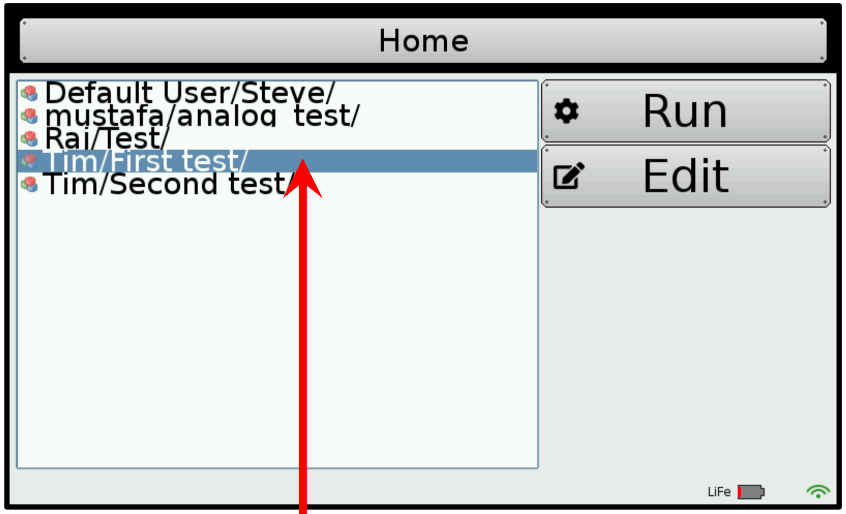
1. Click the ***Compile*** button for your project and, if successful (compilation succeeded), click ***Run*** so you can run your project to see if it works.



NOTE: When you compile, your project is automatically saved.

Running Program from Robot





Select Programs

Highlight program and press run

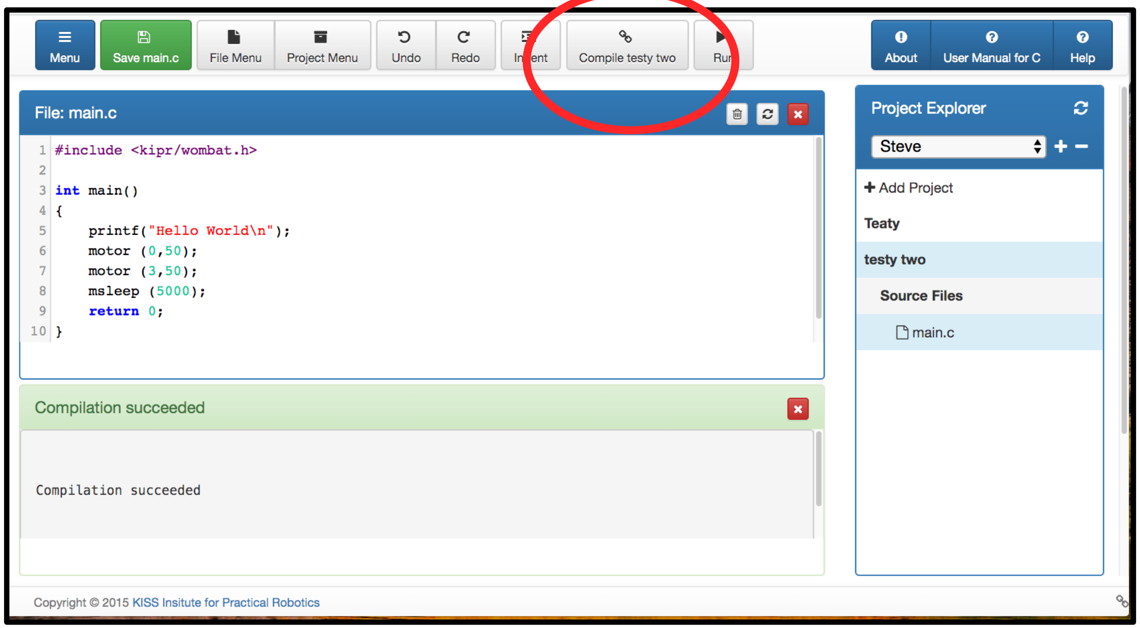
Professional Development Workshop

#Botball®

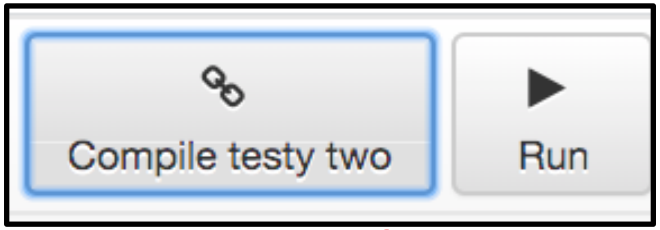
Connection Issues

Your computer may disconnect from the KIPR Robotics Controller. You will know this happens when:

You hit compile and the button **<u>Does Not</u>** turn red (nothing happens)



Connected



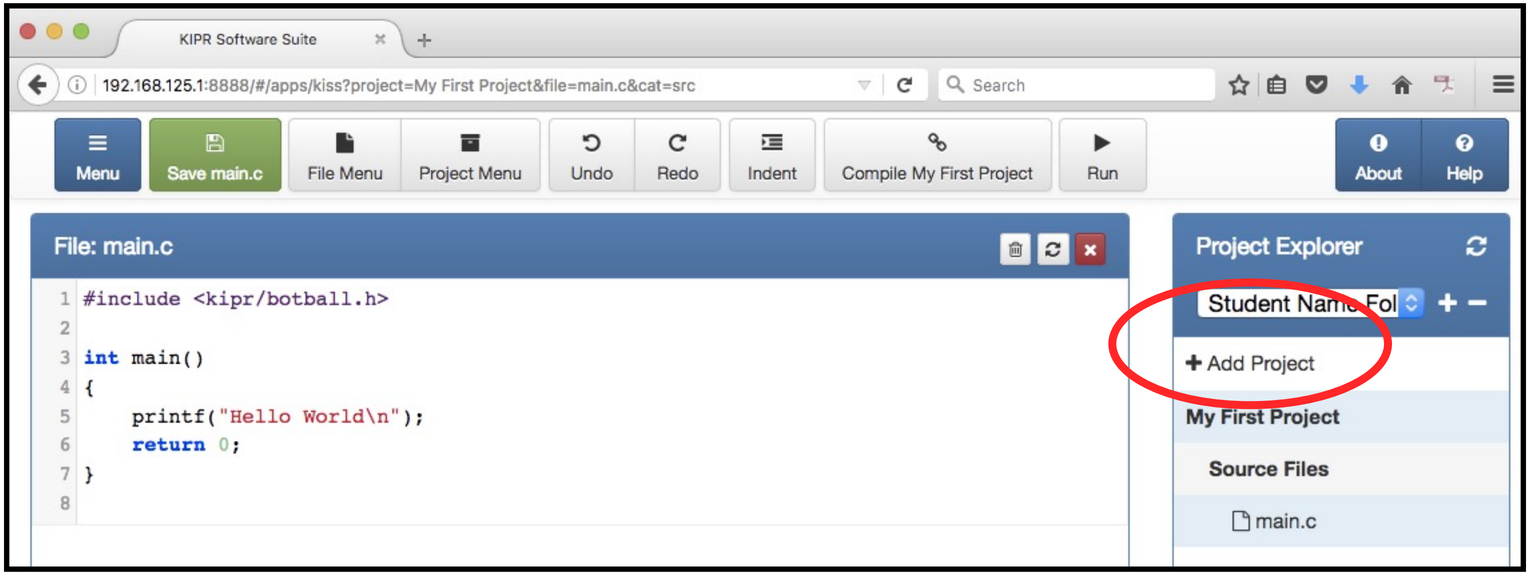
Not Connected

#Botball

Starting Another Project

Note: one *project* = one *program*.

- Click the ***+ Add Project*** button or click the ***Menu*** button to return to the starting menu.
- Proceed as before.
- The **Project Explorer** panel will show you all of the user folder projects and actively edited files.



Professional Development Workshop

Explaining the “Hello, World!” C Program

Program flow and the main function

Programming statements and functions

Comments

"Hello, World!"

File: main.c

```c
#include <kipr/botball.h>

int main()
{
    printf("Hello World\n");
    return 0;
}

```

Note: We will use this template every time; we will delete lines we don't want, and we will add lines that we do want.

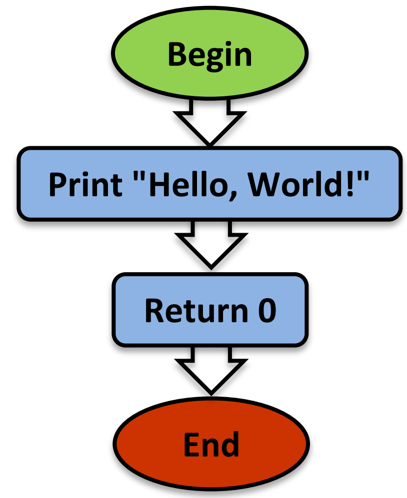
Program Flow and Line Numbers

File: main.c

Top

```c
#include <kipr/botball.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Bottom



Computers read a program just like you read a book—
they read each line starting at the top and go to the bottom.

Computers can read incredibly quickly—
Millions of lines per second!

Source Code

File: main.c

```c
#include <kipr/botball.h>

int main()
{
    printf("Hello World\n");
    return 0;
}

```

This is the **source code** for our first **C program**.

Let’s look at each part of the **source code**.

The `main` Function

A **function** defines a list of actions to take.
A function is like a **recipe** for baking a cake.
When you **call** (use) the function,
the program follows the instructions and bakes the cake.

```
// Created on Thu January 5 2018

int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

This is the **`main()` function**.

When you run your program,
the **`main` function** is executed.

A C program must have
exactly one **`main()` function**.

Professional Development Workshop

#Botball®

Block of Code

The list of actions that the function performs is defined inside a **block of code**.

```
// Created on Thu January 5 2018

int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

Block Header

Begin

End

This is a **block of code**.

A block of code should always be preceded by a **block header**, which is the line just before the {

A block is defined between a **beginning** curly brace { and an **ending** curly brace }

#Botball

Programming Statements

```
// Created on Thu January 5 2018

int main()
{
    printf("Hello, World!\n");   // Statement #1
    return 0;                    // Statement #2
}
```

Statement #1 →
Statement #2 →

Inside the **block of code** (between the { and } braces), we write lines of code called **programming statements**.

Each **programming statement** is an action to be executed by the computer (or robot) **in the order that it is listed**.

There can be any number of **programming statements** within a **block of code**.

#Botball

KIPR functions reference sheet

Until you are familiar with the functions that you will be using, use this function reference **sheet** as an easy reference.

Copying and pasting your own code is also very helpful.

Function Reference Guide 2020

Wombat	
printf("text\n");	// Prints the specified text to the screen
msleep(# milliseconds);	// Another name for wait_for_milliseconds
motor(port #, power);	// Turns on motor with specified port # at % velocity
mav(port #, velocity);	// Move motor at specified velocity (# ticks per second)
ao();	// All off; turns all motor ports off
enable_servos();	// Turns on servo ports
disable_servos();	// Turns off servo ports
set_servo_position(port #, position);	// Moves servo in specified port # to specified position
wait_for_light(port #);	// Waits for light in specified port # before next line
analog(port #);	// Get a sensor reading from a specified analog port #
digital(port #);	// Get a sensor reading from a specified digital port #
shut_down_in(time in seconds);	// Shuts down program after specified # of seconds
Camera	
camera_open();	// Opens the camera for use
camera_close();	// Closes the current camera instance
camera_update();	// Pulls a new image from the camera for processing
get_object_center_x(channel #, object #);	// The x-axis center of a specified object on a specified channel
get_object_area();	//Returns area of bounding box
get_object_count(channel);	// Counts the number of objects using the given channel
Create	
create_connect();	// Establishes a connection to the create
create_disconnect();	// Disconnects from the create
create_drive_direct(l_speed, r_speed);	// Moves left(l) and right(r) create motors at specified speeds
create_stop();	// Turns all create motors off
get_create_total_angle();	// Gets the creates current angle; negative is counterclockwise
set_create_total_angle(angle);	// sets the total angle of the create to the specified value
get_create_lbump():	// returns value of left bump sensor
get_create_rbump()	// returns value of right bump sensor
get_create_lfcliff_amt()	//returns the value from the left front cliff sensor
get_create_rfcliff_amt()	//returns the value from the right front cliff sensor
Printing Sensor Values	
printf("left cliff Value: %d\n",get_create_lfcliff_amt())	//prints the value of the left front cliff sensor
printf("Distance Value: %d\n",get_create_distance());	//prints the value form the create distance sensor
printf("Angle Value: %d\n",get_create_total_angle());	//prints the value from the total angle sensor

www.kipr.org
405-579-4609

Ending a Programming Statement

```
// Created on Thu January 5 2018

int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

Each **programming statement** ends with a **semicolon** ; (*unless* it is followed by a new **block of code**).

This is similar to an **English sentence**, which ends with a **period**.

If an **English sentence** is missing a **period**, then it is a run-on sentence.

#Botball®

Ending the `main` Function

```
// Created on Thu January 5 2018

int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

The **`main` function** ends with a **`return`** statement, which is a response or answer to the computer (or robot).

In this case, the "answer" back to the computer is `0`.

The **`return`** statement is generally the **last line before the } brace**.

Comments

The **green** text at the top of the program is called a "**comment**".

```
// Created on Thu January 5 2018

int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

Comments are helpful notes that can be read by you or your team—**they are *ignored* (<u>not</u> read) by the computer!**

Text Color Highlighting

The KISS IDE highlights parts of a program to make it easier to read. (By default, the KISS IDE colors your code and adds line numbers.)

- **Includes** in **purple**
- **Comments** in **green**
- **Text strings** appear in **red**
- **Keywords** appear in **blue**

File: main.c

```
#include <kipr/botball.h>

int main()
{
    //commenting for the flow of code
    printf("Hello World\n");
    return 0;
}

```

Print Your Name

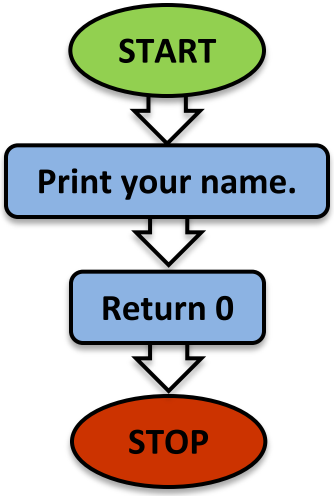
Description: Write a program for the KIPR Wombat that prints your name.

Solution:

Source Code

```
int main()
{
  // 1. Print your name.
  printf("Botguy\n");

  // 2. End the program.
  return 0;
}
```

Flowchart



Designing Your Own Program

Breaking Down a Task

Pseudocode, Flowcharts, and Comments

`msleep()` Function

Debugging Your Program

Complex Tasks → Simple Subtasks

- Break down the objectives (**complex tasks**) into smaller objectives (**simple subtasks**).

- Break down the smaller tasks into even smaller tasks.
  Continue this process until each subtask can be accomplished by a list of individual programming statements.

- For example, the larger task might be to make a PB&J Sandwich which has smaller tasks of getting the bread and PB&J ready and then combining them.

Practice Printing

Description: Write a program for the KIPR Wombat that prints "Hello, World!" on one line, and then prints your name on the next line.

Analysis: What is the program supposed to do?

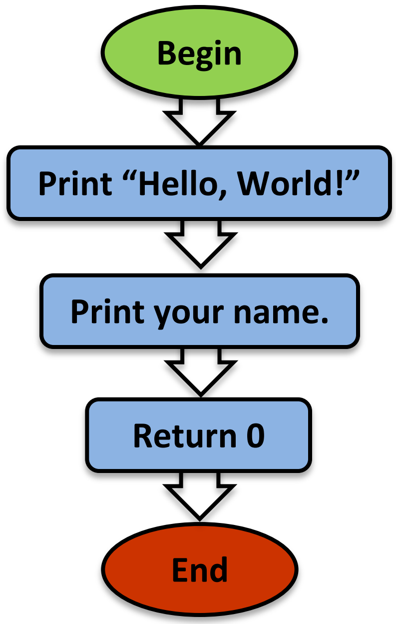
Pseudocode

1. Print "Hello, World!"
2. Print your name.
3. End the program.

Comments

```
// 1. Print "Hello, World!"
// 2. Print your name.
// 3. End the program.
```

Flowchart



In English, write a list of actions to solve an activity.

These are three different ways to do this.

Practice Printing

Solution: Create a **new project**, create a **new file**, and enter your **pseudocode** and **source code** in the **`main`** function.

- **Note:** remember to give your project and file descriptive (unique) names!

Pseudocode

```
1. Print "Hello, World!"
2. Print your name.
3. End the program.
```

Helps you *write* the real code!

Source Code

```c
int main()
{
    printf("Hello, World!\n");

    printf("Botguy\n");

    return 0;
}
```

Execution: Compile and run your program on the KIPR Wombat.

Practice Printing

Reflection: What did you notice after you ran the program?

- The KIPR Robotics Controller reads code and [generally] goes to the next line faster than a blink of your eye.
- The KIPR Robotics Controller is executing thousands of lines of code per second!
- To control a robot, sometimes it is helpful to **wait for some duration of time** after a function has been called so that it can actually perform the task.
- To do this, we use the built-in function called **`msleep()`**

Let's use this!

Waiting for Some Time

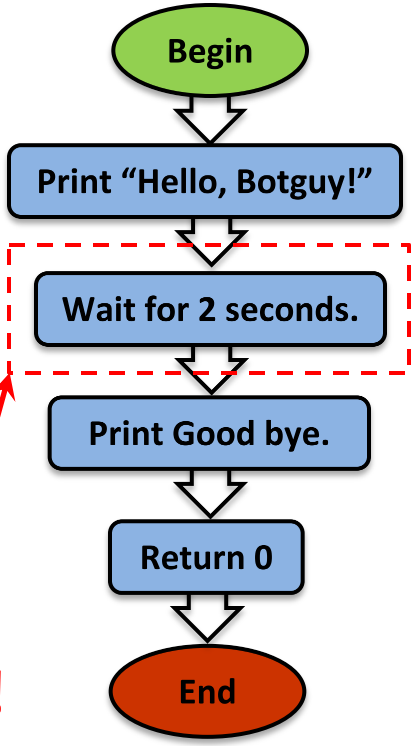
Description: Write a program that prints "Hello, [your name]!" on one line, waits two seconds, and then prints "Good bye." on the next line.

Analysis: What is the program supposed to do?

Pseudocode	Comments
1. Print “Hello, Botguy!”	`// 1. Print "Hello, Botguy!"`
2. Wait for 2 seconds.	`// 2. Wait for 2 seconds.`
3. Print “Good bye”.	`// 3. Print "Good bye."`
4. End the program.	`// 4. End the program.`

New!

Flowchart



#Botball

Waiting for some time

Solution: Create a **new project**, create a **new file**, and enter your **pseudocode** and **source code** in the `main` function.

- **Note:** remember to give your project and file descriptive (unique) names!

Pseudocode

```
1. Print "Hello, Botguy!"
2. Wait for 2 seconds.
3. Print "Good bye".
4. End the program.
```

Source Code

```c
int main()
{
    printf("Hello, Botguy!\n");

    msleep(2000);

    printf("Good bye.\n");

    return 0;
}
```

Execution: Compile and run your program.

#Botball®

Waiting for Some Time

Reflection: What did you notice after you ran the program?

- Did your code work the first time you typed it in?
- Did you have any **errors**?

Debugging Errors

!!! ERROR !!!

- If you do not follow the rules of the **programming language**, then the **compiler** will get confused and not be able to **translate** your **source code** into **machine code**—it will say "**Compile Failed!**"
- The Wombat will try to tell you where it *thinks* the **error** is located.
- The process of trying to resolve this **error** is called "**debugging**".
- **To test this**, remove a `;` from one of your programs and compile it.
  - How about if you remove a `"` from one of your printf statements?
  - What if you type `msleep()` as **`Msleep()`**?

Debugging Errors

line # : col # (the error is on or before line # 6)

```
/home/root/Documents/KISS/Default User/hey/src/main.c: In function 'main':
/home/root/Documents/KISS/Default User/hey/src/main.c:6:5: error: expected ';' before 'return'
     return 0;
```

" expected ; " (semicolon)

File: main.c

```c
1 #include <kipr/botball.h>
2
3 int main()
4 {
5     printf("Hello World\n")
6     return 0;
7 }
8
```

When there is an error, you can ignore the first error line ("`In function 'main'`") and read the next to see what the first error is. If you have a lot of errors, start fixing them from the top going down. Fix one or two and recompile.

Compilation Failed

```
Compilation Failed

/home/root/Documents/KISS/Default User/hey/src/main.c: In function 'main':
/home/root/Documents/KISS/Default User/hey/src/main.c:6:5: error: expected ';' before 'return'
```

Moving the DemoBot with Motors

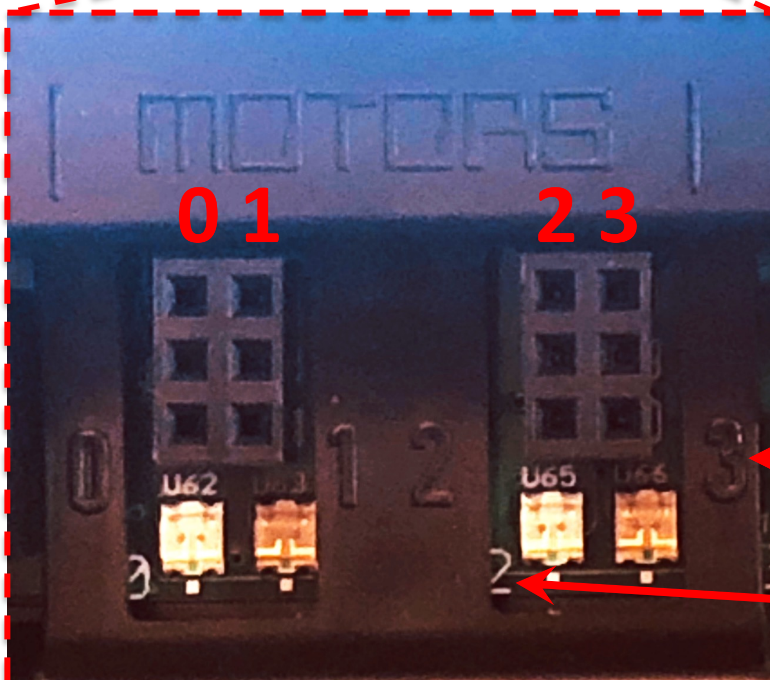
Plugging in motors (ports and direction)

`motor` functions

Check the Robot’s Motor Ports

- To program your robot to move, you need to know which **motor ports** your motors are plugged into.
- Computer scientists tend to start counting at 0, so the four **motor ports** are numbered **0**, **1**, **2**, and **3**.

Wombat Motor Ports



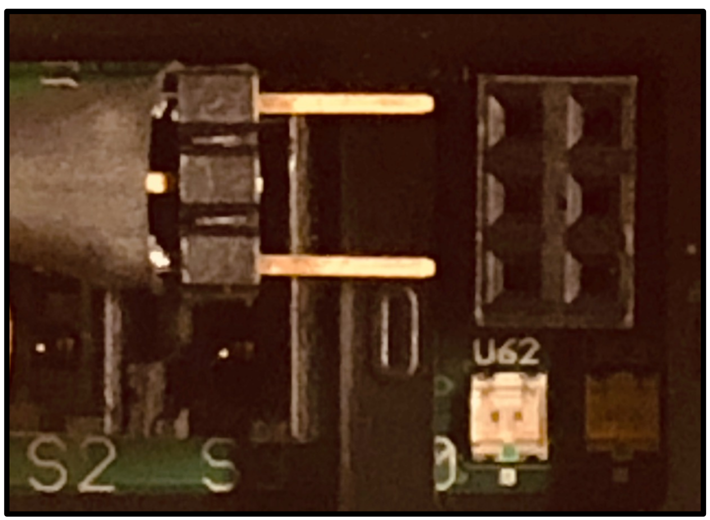
Motor Port Labels are on the Case

Motor Port Labels 0, 2 are also on the board

#Botball®

Plugging in Motors

- **Motors** have red wire and a black wire with a <u>**two-prong plug**</u>.
- The Wombat has 4 motor ports numbered **0** & **1** on left, and **2** & **3** on right.
- When a port is powered (receiving motor commands), it has a light that glows **green** for one direction and **red** for the other direction.
  - Plug orientation order determines motor direction.
  - By convention, **green** is **forward** (**+**) and **red** is **reverse** (**–**)
    - ***Unless you plug in the motors "backwards"***.



Drive motors have a two-prong plug.

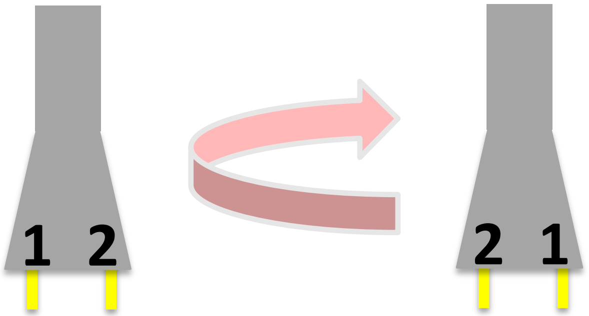
Plugged in Motor

Motor Plugged into Port 0 (right wheel on DemoBot)

Motor Direction

You want your motors going in the same direction; otherwise, your robot will go in circles!

- **Motors** have a red wire and a black wire with a **two-prong plug.**
- You can plug these in two different ways:
  - One direction is clockwise, and the other direction is counterclockwise.
  - The red and black wires help determine motor direction.



Professional Development Workshop

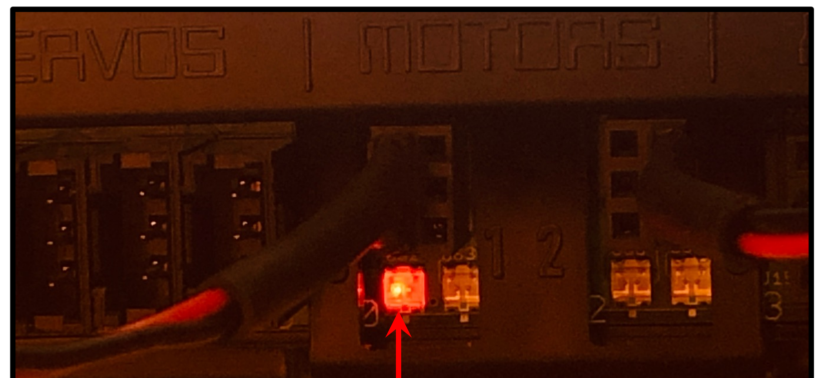
#Botball

Motor Port and Direction Check

There is an easy way to check this!

- Manually rotate the tire, and you will see an LED light up below the **motor port** (the **port #** is labeled on the board).
  - If the LED is **green**, it is going **forward** (**+**).
  - If the LED is **red**, it is going **reverse** (**–**).

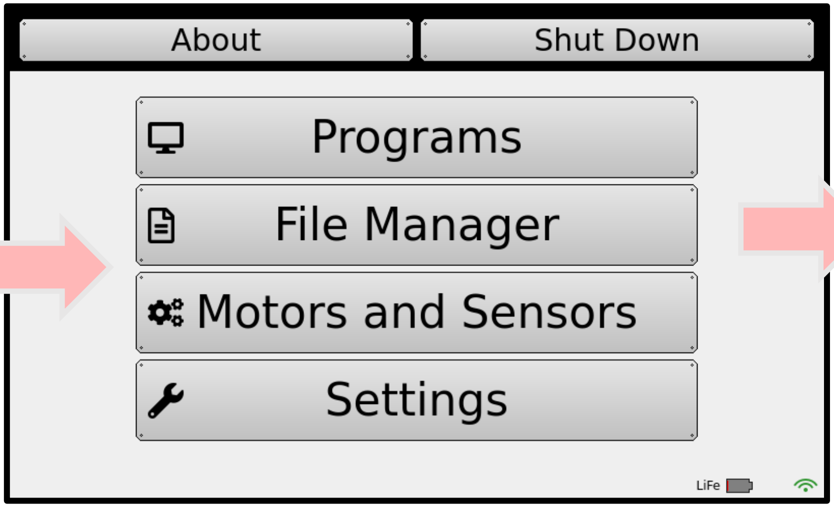
forward

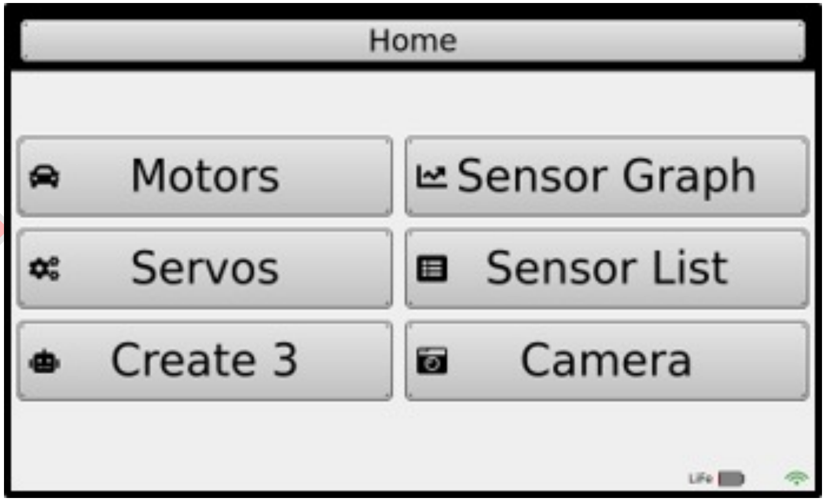


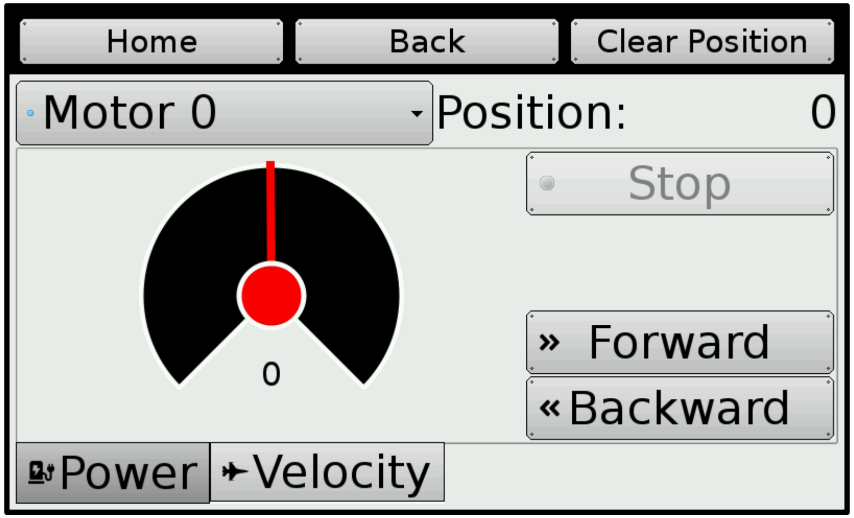
backward

- Use this trick to check the **port #**'s and **direction** of your **motors**.
  - If one is **red** and the other is **green**,
    turn one motor plug 180° and plug it back in.
  - The lights should both be **green** if the robot is moving forward.

Use the Motor Widget







#Botball®

Common Motor Functions

There are several functions for motors.
We will begin with `motor()`

Motor port #
(between **0** and **3**)

```
motor(0, 100);
// Turns on motor port #0 at 100% power.
// Power should be between -100% and 100%.

msleep(# milliseconds);
// Wait for the specified amount of time.

ao();
// Turn off all of the motors.
```

A **positive number** should drive the motor **forward**; if not, rotate the motor plug 180°.

A **negative number** should drive the motor **reverse**.

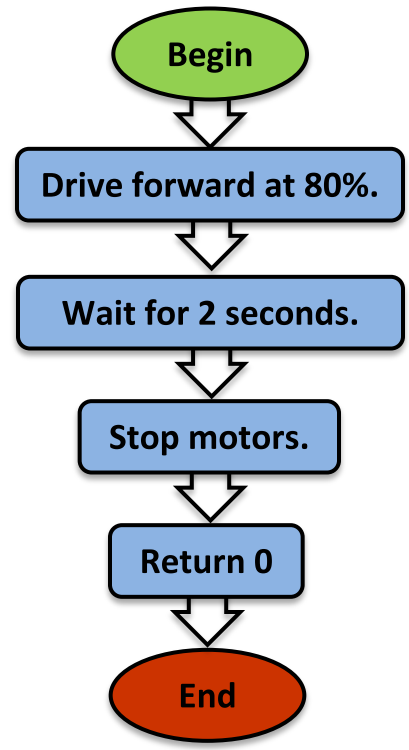
If two drive motors are plugged in in opposite directions from each other, then the robot will go in a circle.

Moving the DemoBot

Description: Write a program that drives the DemoBot forward at 80% power for two seconds, and then stops.

Analysis: What is the program supposed to do?

Flowchart



Pseudocode

1. Drive forward at 80%.
2. Wait for 2 seconds.
3. Stop motors.
4. End the program.

Comments

```
// 1. Drive forward at 80%.
// 2. Wait for 2 seconds.
// 3. Stop motors.
// 4. End the program.
```

Moving the DemoBot

Solution: Create a **new project**, create a **new file**, and enter your **pseudocode** (as **comments**) and **source code** in the **`main`** function.

- **Note:** remember to give your project and file descriptive, unique names!

Psuedocode (Comments)

```
1. Drive forward at 80%.
2. Wait for 2 seconds.
3. Stop motors.
4. End the program.
```

Source Code

```
#include <kipr/botball.h>
int main()
{
  motor(0, 80);
  motor(3, 80);    //forward
  msleep(2000);

  ao();

  return 0;
}
```

Execution: Compile and run your program.

#Botball®

Moving the DemoBot

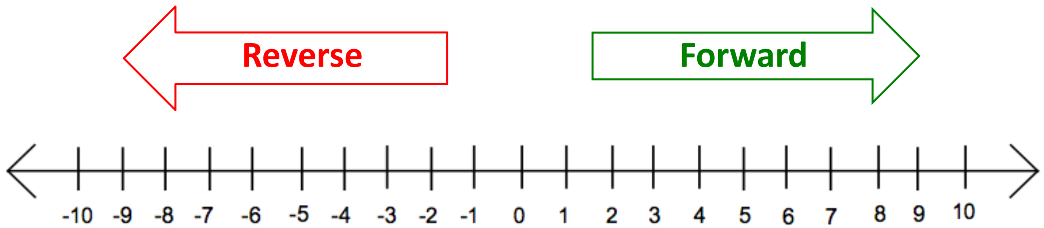
Reflection: What did you notice after you ran the program?

- Did the DemoBot move forward?
  - **Positive** (+) numbers should move the motors in a clockwise direction (**forward**); if not, rotate the motor plug 180° where it plugs into the Wombat.
  - If your robot moves in a circle, one motor is either not moving (is it plugged in?) or they are moving in opposite directions (rotate the motor plug 180°).

- Did the DemoBot drive straight?
- How could you adjust the code to make the robot drive straight?
- How can you make the robot drive backwards?
- How can you make the robot turn left or right?

Robot Driving Hints

Remember your # line:
positive numbers (+) go forward and negative numbers (–) go in reverse.



Driving straight: it is surprisingly difficult to drive in a straight line…

- **Problem:** Motors are not exactly the same.
- **Problem:** The tires might not be aligned perfectly.
- **Problem:** One tire has more resistance.

And many, many other reasons…

- **Solution:** You can adjust this by slowing down or speeding up the motors.

Making turns:

- **Solution:** Have one wheel go faster or slower than the other.
- **Solution:** Have one wheel move while the other one is stopped.
- **Solution:** Have one wheel move forward and the other wheel move in reverse (friction is less of a factor when both wheels are moving).

More Motor functions

Resource

```
motor(0, 100);
// Turns on motor port #0 at 100% power.
```

- Is great for turning gears or winding up string on a pulley.
- Is not so great for driving robots as it is dependent on battery charge.

```
mav(0, 800);
// Move motor on port #0 at 800 ticks/sec.
```

- Is great for driving robots and not as dependent on battery charge.
- Greater precision of motor control (think of this as being like cruise control in a car).
- Must use `wait_for_milliseconds` function correctly.

```
mrp(0, 800, 3000);
// Move motor on port #0 forward 3000 ticks at 800 ticks/sec.
```

- Provides the most precise level of motor control.
- Most complicated to use (must do a lot of calculations to move correctly).

Motor Position Counter

Motor position counter functions

Ticks and revolutions

Motor Position Counter

Each motor used by the DemoBot has a built-in motor position counter, which you can use to calculate the distance traveled by the robot!

Motor Port #
(0 to 3)

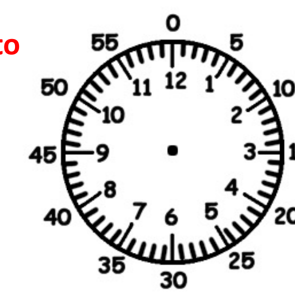
```
get_motor_position_counter(0)          — OR —          gmpc(0)
// Tells us the number of ticks the motor on port #0 has rotated.
```

Motor Port #
(0 to 3)

```
clear_motor_position_counter(0);       — OR —          cmpc(0);
// Resets the tick counter to 0 for the motor on port #0.
```

- The motor position is measured in "**ticks**".
- Botball motors have ***approximately* 1800 ticks per *revolution***.
- Use **wheel circumference divided by 1800** to calculate distance!

Similar to how a clock is divided into 60-second intervals (ticks).

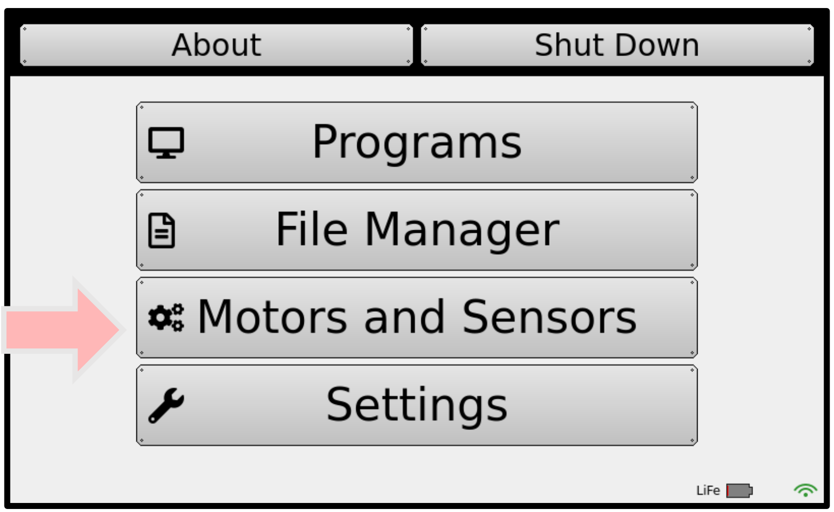


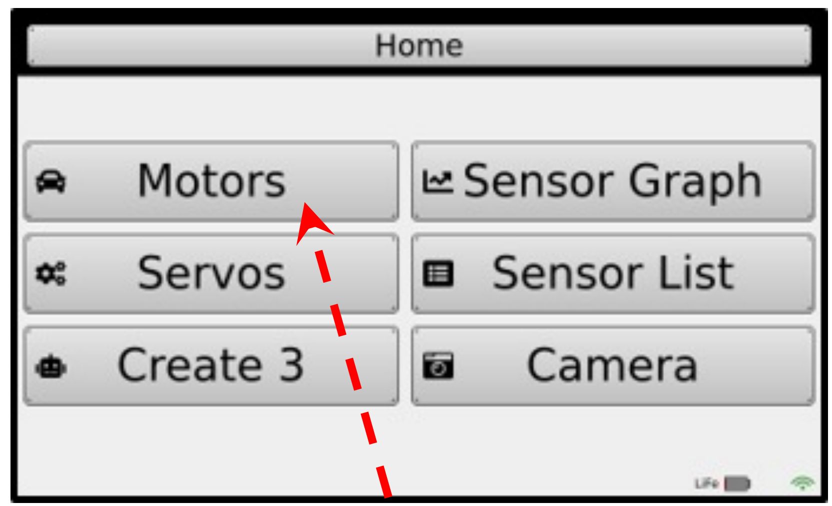
#Botball®

Seeing Counters on Screen

You can access the Motors from the Motors and Sensors section

- This is very helpful to test your motors and see the actual motor position counters “*in action*”





Select Motors

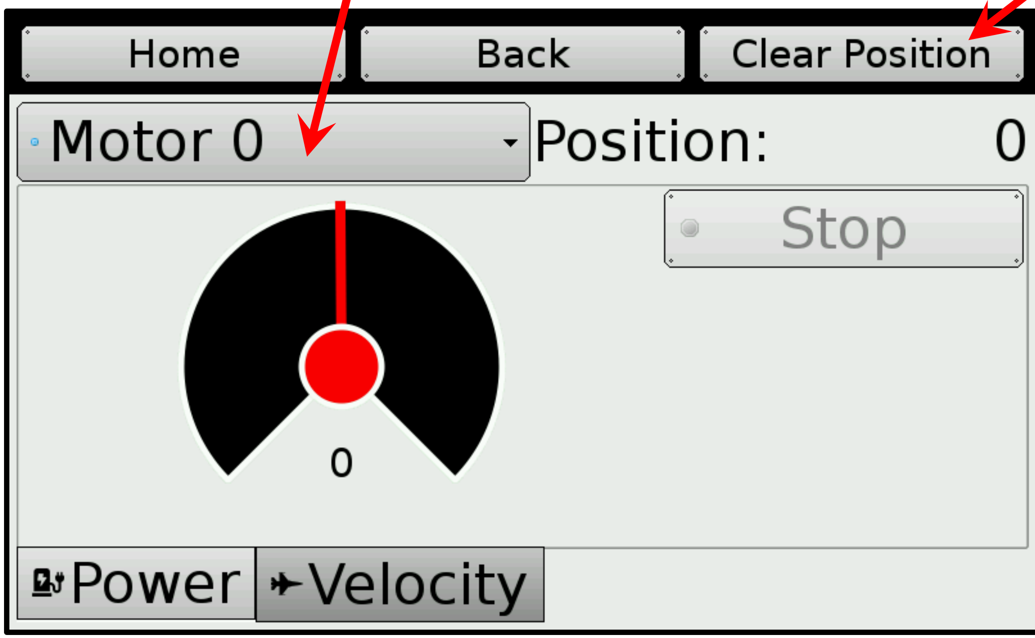
#Botball®

Seeing Counters on Screen

Select motor port (allows you to select the motor of your choice)

To clear (reset) the counter

Motor Position in "ticks"



Use your hand to rotate the robot's wheel (plugged into port 0) and watch the position counter.

What happens if you turn the wheel in the opposite direction?

You can also place your robot on a surface and roll it forward to measure the # ticks from a starting position to another location or object

Drive to a Specific Point

You can also place your robot on a surface and roll it forward to measure the # ticks from a starting position to another location or object.

Place the robot in the *start box* of **KIPR Mat A** and using the motors/widget screen:

1) reset the left motor counter
2) manually push the robot forward to *circle 9* on the mat
3) visually record/remember the tick count

Description: Write your program to drive the DemoBot forward that many "ticks" and then stop.

Pseudocode

Generate it!

Drive to a Specific Point

Solution:

Source Code

```c
int main()
{
  int distance = 4500;   // in ticks

  cmpc(0);

  while (gmpc(0) < distance)
  {
    motor(0, 50);
    motor(3, 50);
  }
  ao();

  return 0;
}
```

Pseudocode

```
1. Reset motor position counter.
2. Loop: Is counter < my
   distance?
     2.1. Drive forward.
3. Stop motors.
4. End the program.
```


#Botball®

Drive to a Specific Point

Reflection: What did you notice after you ran the program?

- How far did the robot travel? Was it always the same (you tested it more than once, right)?
  - Your robot most likely went FURTHER than you programmed it to (check the motors screen after it stops to see the actual final tick count). Why? Hint: inertia
  - Change your loop so that it actually goes to “distance - (actual - desired)”:

```
while(gmpc(0) < distance - (4832 – distance))
```

- How could you modify your program to travel a specific distance in millimeters? (**Hint:** Use **wheel circumference (in mm) divided by 1800** to calculate number of mm per tick!)

Drive to a Specific Point + Backup

Description: Write your program to drive the DemoBot forward to a specific point and then back up to where you started.

Pseudocode

1. Drive forward.
2. Stop at specific distance
3. Drive backwards.
4. Stop at starting point.

Comments

```
// 1. Drive forward.
// 2. Loop: Is motor position at specific count?
// 3. Drive Backwards to specific distance.
// 4. End the program.
```

Drive to a Specific Point + Backup

Solution:

Now back up to position (tick count 0).
Note: clear counter not needed this time

```
int main()
{
  int distance = 4500;  // in ticks

  cmpc(0);
  while (gmpc(0) < distance)
  {
    motor(0, 50);
    motor(3, 50);
  }
  ao();
  msleep(2000);   // see it stop?

  while (gmpc(0) > 0)
  {
    motor(0, -50);
    motor(3, -50);
  }
  ao();

  return 0;
}
```

[Bonus] Drive Straight!

Description: Write a program for the KIPR Robotics Controller that drives the DemoBot straight for 14000 ticks by adjusting the right motor power so that the position of the left motor is the same (or close) to the right.

Analysis: How can you adjust the left motor's position?

Pseudocode

```
1.  Clear both motor counters
2.  Loop: If total distance < 14000
      Move left motor 75% power
      If: Right is behind left
         speed up right
      Else:
         slow down right
3.  Stop motors
4.  End the program
```


#Botball®

Drive Straight!

Solution:

Pseudocode

1. Clear both motor counters.
   Loop: check left position
   power left motor at 75%.
   If: slower
   right motor at 100%
   Else: faster
   right motor at 50%
3. Stop motors.
4. End the program.

Source Code

```
int main()
{
    cmpc(0);
    cmpc(3);

    while(gmpc(3) < 14000)
    {

      motor(3, 75);

      if(gmpc(0) < gmpc(3))
      {
        motor(0, 100);
      }
      else
      {
        motor(0, 50);
      }
    }

    ao();

    return 0;
}
```

Drive Straight

Reflection: What did you notice after you ran the program?

- Did the robot go straighter than in the previous program?
- How could you use this technique whenever you wanted to drive straight? (**Hint:** Consider writing a function with an argument for the distance.)
- How could you modify your program to go straight at different speeds?

Other motor functions

Move At Velocity:

```
mav(0, 1000);
```

Motor Port #
(between 0 and 3)

Velocity (in ticks per second)
between -1000 (reverse)
and 1000 (forward)

Move Relative Position:

```
mrp(0, 1000, 3000);
```

Motor Position (in ticks);
~1000 ticks = 1 tire revolution

#Botball

Move At Velocity

STUDENT ACTIVITY

Description: Write a program for the KIPR Link that drives the DemoBot forward at 1000 ticks per second for 3 seconds, then in reverse at 1000 ticks per second for 3 second, then stops.

Analysis: What is the program supposed to do?

Pseudocode

1. Drive forward at 1000 ticks/sec.
2. Wait for 3 seconds.
3. Drive reverse at 1000 ticks/sec.
4. Wait for 3 seconds.
5. Stop motors.
6. End the program.

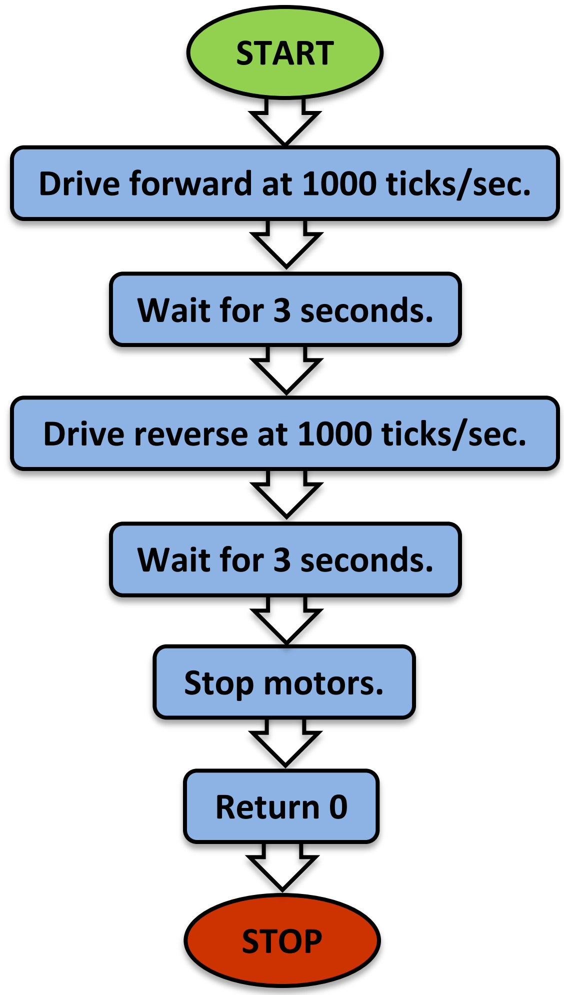
Comments

```
// 1. Drive forward at 1000 ticks/sec.
// 2. Wait for 3 seconds.
// 3. Drive reverse at 1000 ticks/sec.
// 4. Wait for 3 seconds.
// 5. Stop motors.
// 6. End the program.
```

Move At Velocity

Analysis:

Flowchart



Move At Velocity

STUDENT ACTIVITY

Solution:

Pseudocode (Comments)

```c
int main()
{
  // 1. Drive forward at 1000 ticks/sec.
  // 2. Wait for 3 seconds.
  // 3. Drive reverse at 1000 ticks/sec.
  // 4. Wait for 3 seconds.
  // 5. Stop motors.
  // 6. End the program.
}
```

Source Code

```c
int main()
{
  // 1. Drive forward at 1000 ticks/sec.
  mav(0, 1000);
  mav(3, 1000);

  // 2. Wait for 3 seconds.
  wait_for_duration(3);

  // 3. Drive reverse at 1000 ticks/sec.
  mav(0, -1000);
  mav(3, -1000);

  // 4. Wait for 3 seconds.
  wait_for_duration(3);

  // 5. Stop motors.
  ao();

  // 6. End the program.
  return 0;
}
```

Execution: Compile and run your program on the KIPR Link.

#Botball®

Precision Turning

Description: Write a program that turns left 90 degrees and then turns right 90 degrees using motor position counter.

Hint: Remember how we manually moved our robots to find the correct position, and that inertia needs to be accounted for…

Pseudocode

1. Turn left 90 degrees.
2. Stop
3. Turn right 90 degrees.
4. Stop at same orientation as start.

Start “small” (try to accomplish the first turn before adding in / working on the second one)

#Botball

Moving the DemoBot Servos

Plugging in servos (ports)

`enable_servos` and `disable_servos` functions

`set_servo_position` function

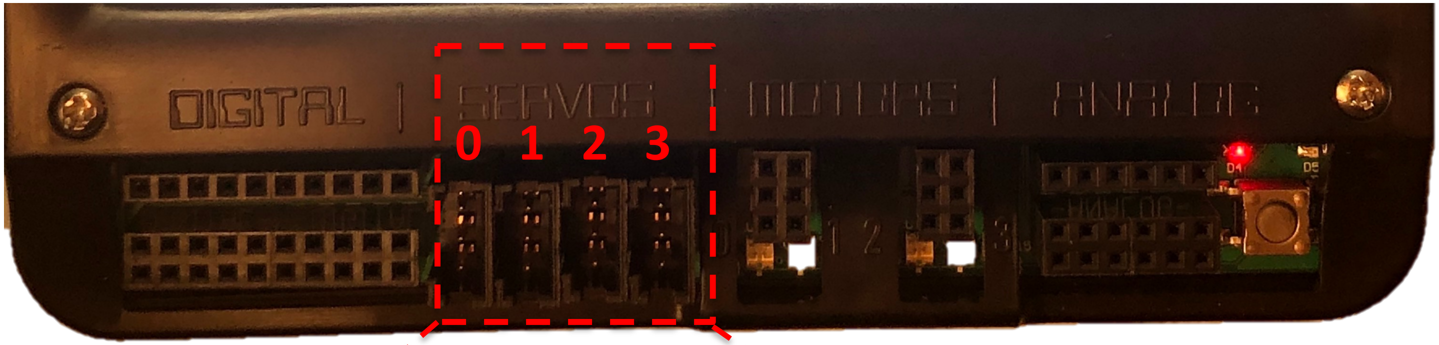
Servos

- A **servo motor** (or **servo** for short) is a motor that rotates to a specified **position between ~0° and ~180°**.
- Servos are great for raising an arm or closing a claw to grab something.
- Servo motors look very similar to non-servo motors, but there are differences…
  - A servo has **three wires** (orange, red, and brown) and a **black plastic plug**.
  - A non-servo motor has **two gray wires** and a **two-prong plug**.

Micro servo

Large servo

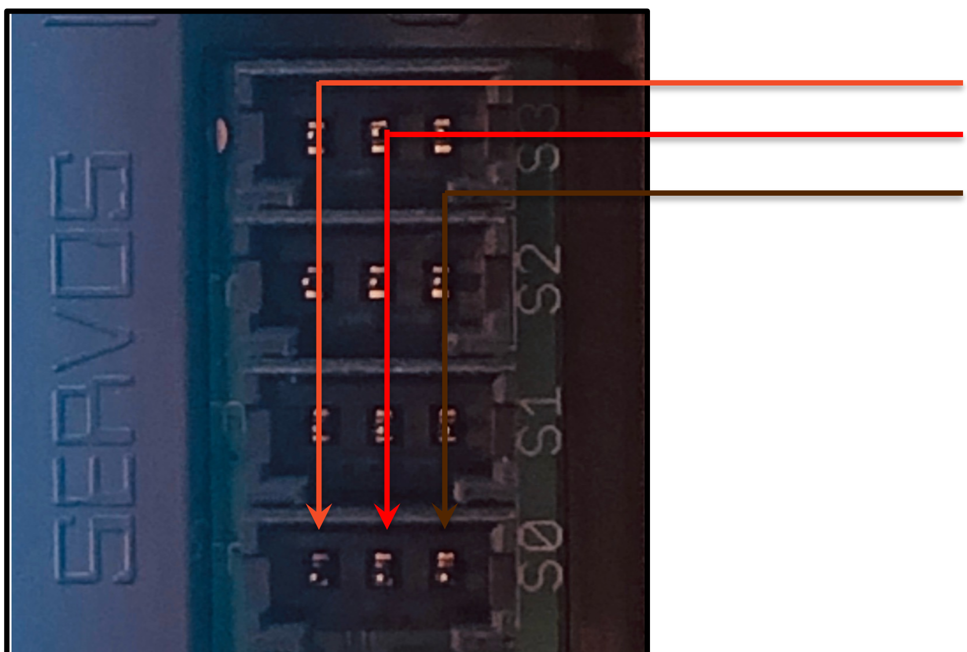
KIPR Robotics Controller Servo Ports



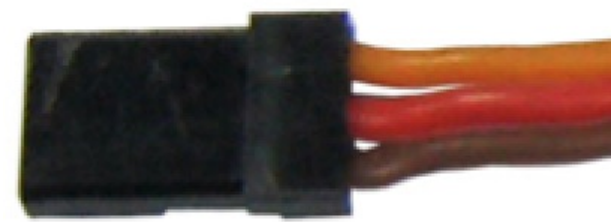
Servo Port Labels are on the board

Plugging in Servos

- The KIPR Robotics Controller has 4 servo ports numbered **0** through **3**.
- Note that the orientation of the wires is very important:
  - (S) for the orange (signal) wire, which regulates servo position
    - Closest to the screen (orange "up", brown "down")
  - (+) for the **red** (**power**) wire
  - (–) for the **brown** (**ground**) wire ("the ground is down, down is negative")



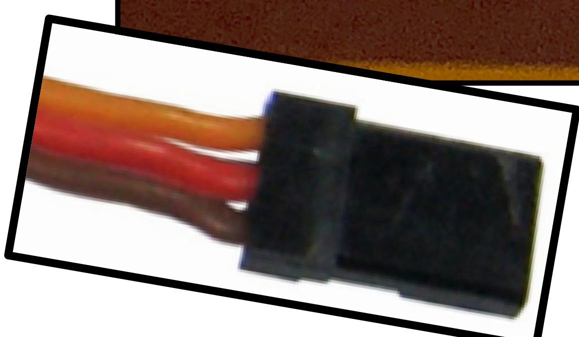
(S) signal wire
(+) power wire
(–) ground wire



NOTICE: **orientation when plugging in the servos is very important**

Plugged in Servos

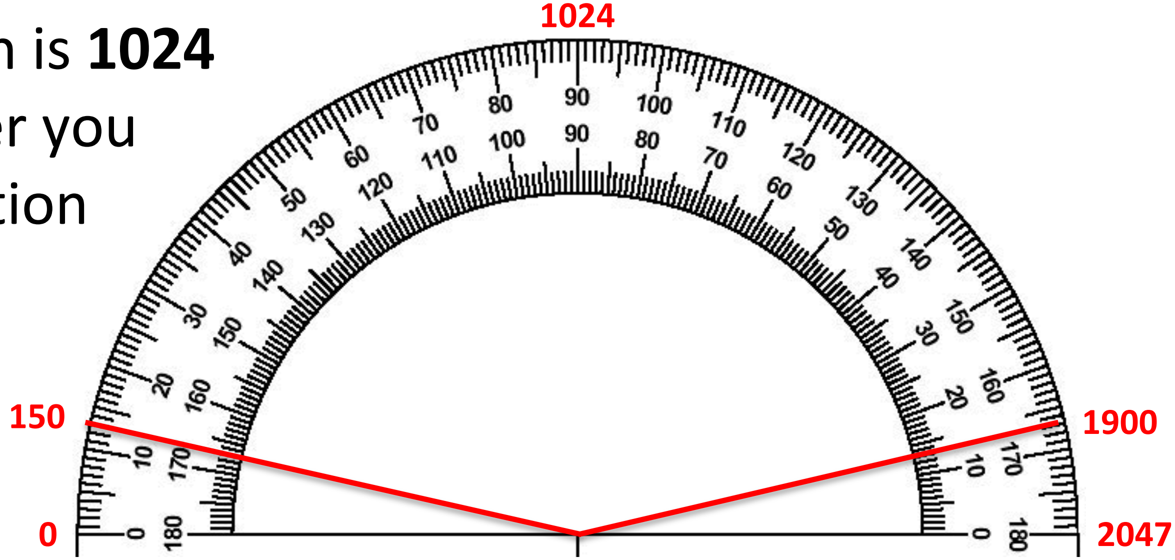
- One servo motor is plugged into Port **0**



#Botball

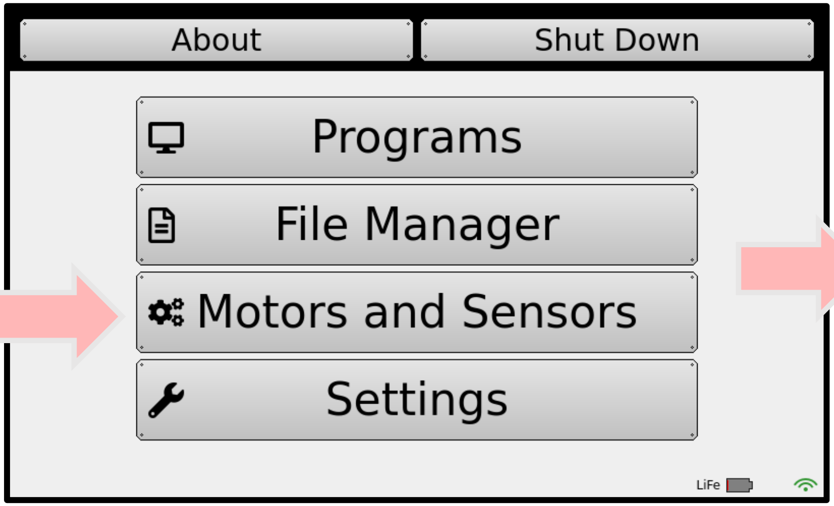
Servo Positions

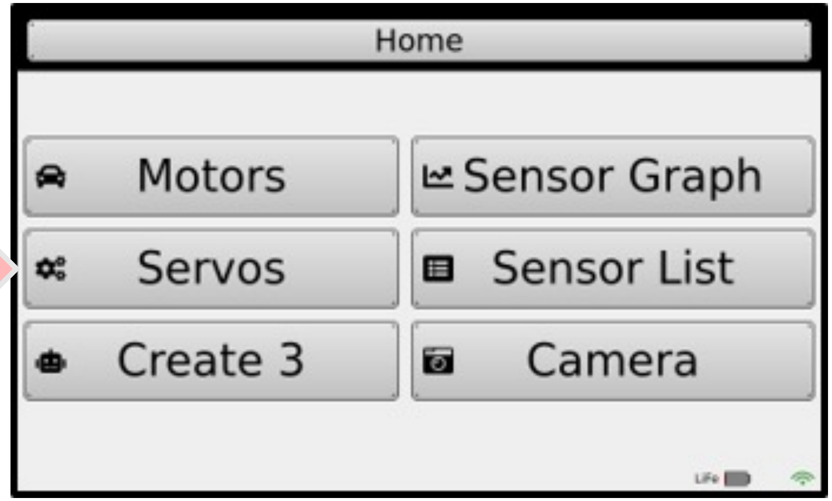
- Think of a servo like a protractor…
  - Angles in the **~180° range of motion** (between ~0° and ~180°) are divided into **2048 servo positions**.
  - These **2048 positions** range from 0 to 2047, but due to internal mechanical hard stop variability you should use **~150 to ~1900**
    (**remember:** computer scientists start counting with 0, not 1).
  - This allows for greater precision when setting a position
    (you have ~2048 different positions to choose from instead of just 180).
- The default position is **1024** (centered), however you should still use caution when setting up initial position.

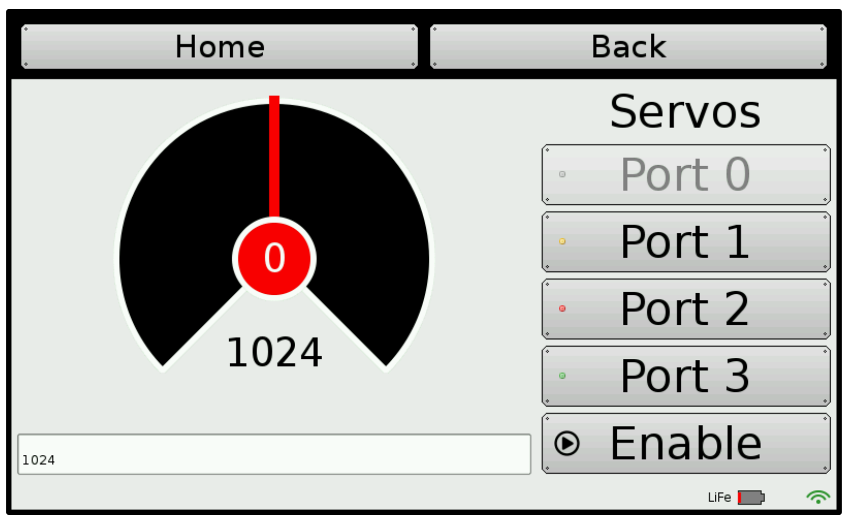


Professional Development Workshop

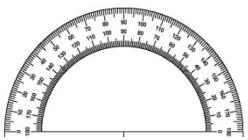
Use the Servo Widget

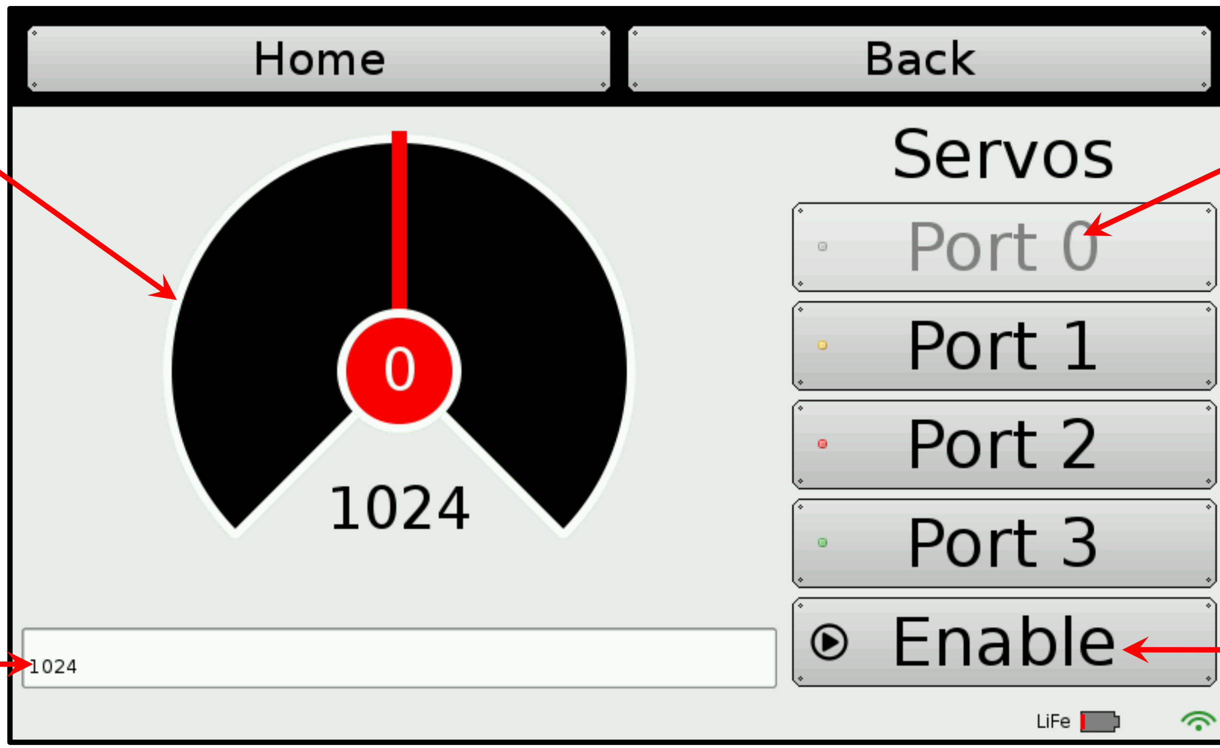






Testing Servos with the Widget





Select the servo port

The current servo position

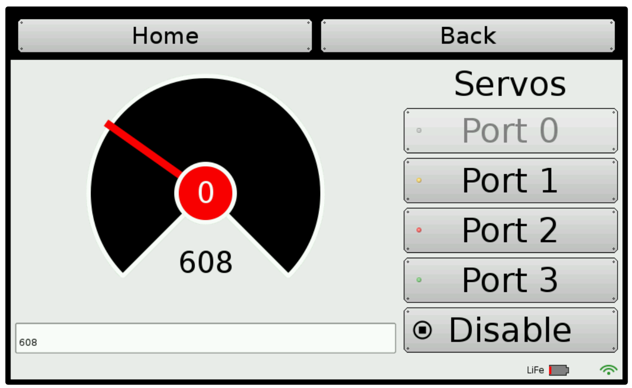
Enable servos

Professional Development Workshop

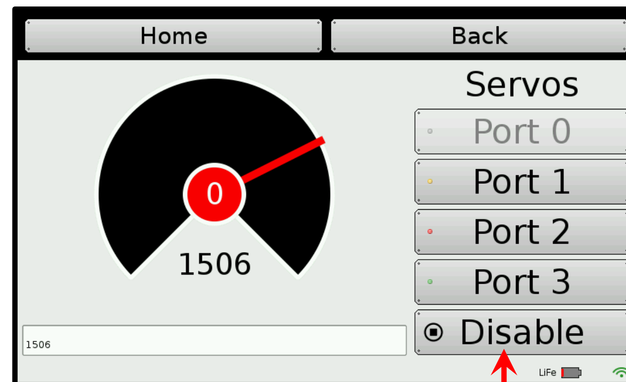
Testing Servos with the Widget

Use your finger
to move the dial.

Servo @ 537



Servo @ 608



Servo @ 1506

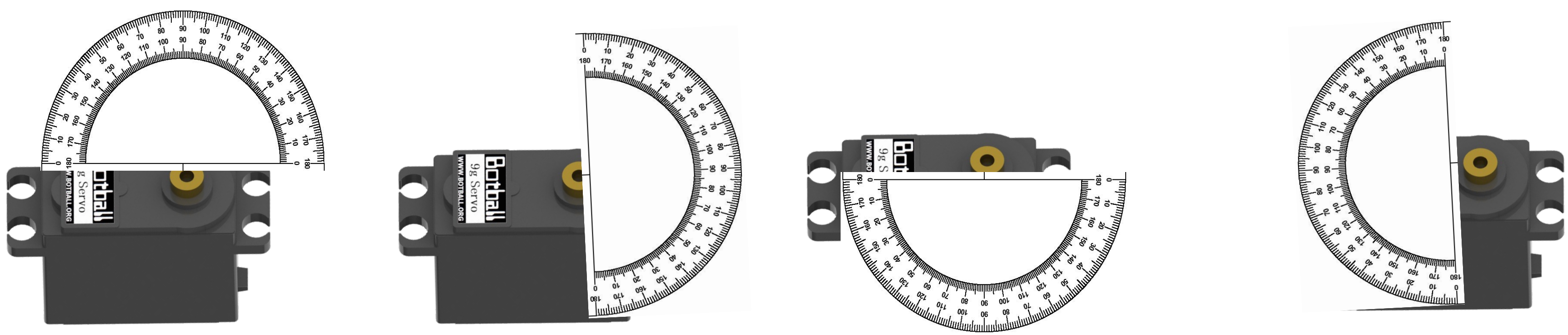
When you are
finished disable
(turn off) the Servo

Do <u>not</u> push a servo beyond its limits
(less than ~150 or more than ~1900).
This can burn out the servo motor!

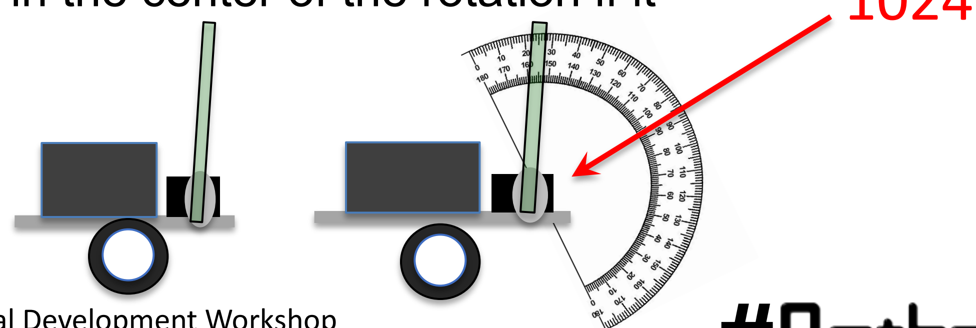
#Botball®

Centering the Servo Horn

- The Servo motor only has a range of motion of (rotates) ~180 degrees, but ***you cannot see by looking at the motor where this range of motion is located in relation to your robot***



- Using the Servo Widget screen, enable the servo on your robot. When you enable it, it will go to 1024. You can unscrew the servo horn on your arm or claw and place it in the center of the rotation if it is not already in the correct position



Servo Functions

- To help save power, servo ports by default are **not** active until they are **enabled**.
- Functions are provided for **enabling** or **disabling** all servo ports.
- A function is also provided for **setting the position** of an individual servo.

```
enable_servos();   // Enable (turn on) all servo ports.

set_servo_position(0, 925);   // set servo on port #0 to position 925.

disable_servos();   // Disable (turn off) all servo ports.
```

- **Note:** it takes the servo TIME to move to a position so if you set it to another position without giving it TIME the CODE runs very fast and does not wait for the servo to move
- You can “**preset**” a servo position by calling **`set_servo_position()`** ***before*** calling **`enable_servos()`**. This will make the servo move towards this position immediately upon calling **`enable_servos()`**.

#Botball

Using Servo Functions

Example:

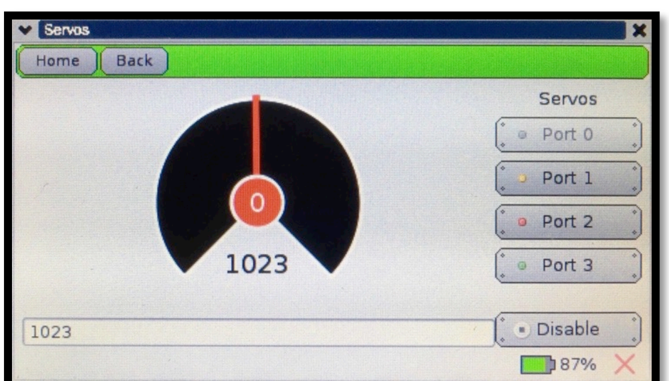
```
int main()
{
  enable_servos();

  set_servo_position(0, 1500);
  msleep(500);

  set_servo_position(0, 925);
  msleep(500);

  set_servo_position(0, 675);
  msleep(500);

  disable_servos();
  return 0;
}
```

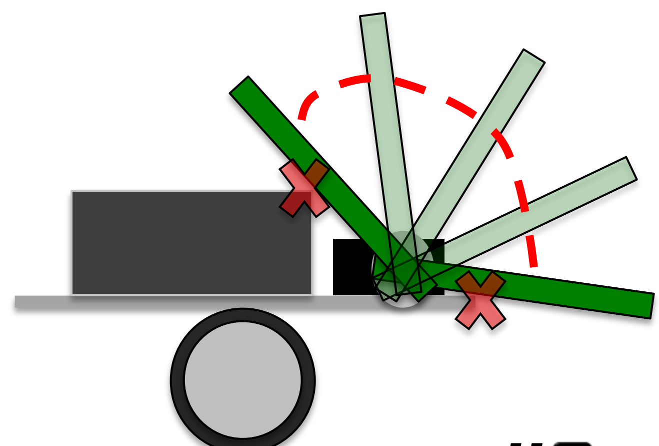
(Note the use, and placement, of msleep to give the servo time to move to each new position)

Wave the Servo Arm

Description: Write a program for the KIPR Robotics Controller that waves the DemoBot servo arm up and down.

- Remember to **enable the servos** at the beginning of your program, and **disable the servos** at the end of your program!
- **Warning:** The arm mounted on your DemoBot prevents the servo from freely rotating to all possible positions (it will run into the KIPR Wombat controller or the chassis of the robot)!
  - Do **not** keep trying to move a servo to a position it cannot reach, as this can burn out the servo and also consume a lot of power from your robot.
  - Use the Servo screen to **determine the limits** of the DemoBot arm, **write these numbers down**, and then **use these numbers in your code**.





Professional Development Workshop

#Botball

Wave the Servo Arm

Description: Write a program for the KIPR Robotics Controller that waves the DemoBot servo arm up and down. Advanced: Write a function that does one wave. Call it from your main function

Analysis: What is the program supposed to do?

Pseudocode

1. Enable servos.
2. Move servo to up.
3. Wait for 3 seconds.
4. Move servo to down.
5. Wait for 3 seconds.
6. Disable servos.
7. End the program.

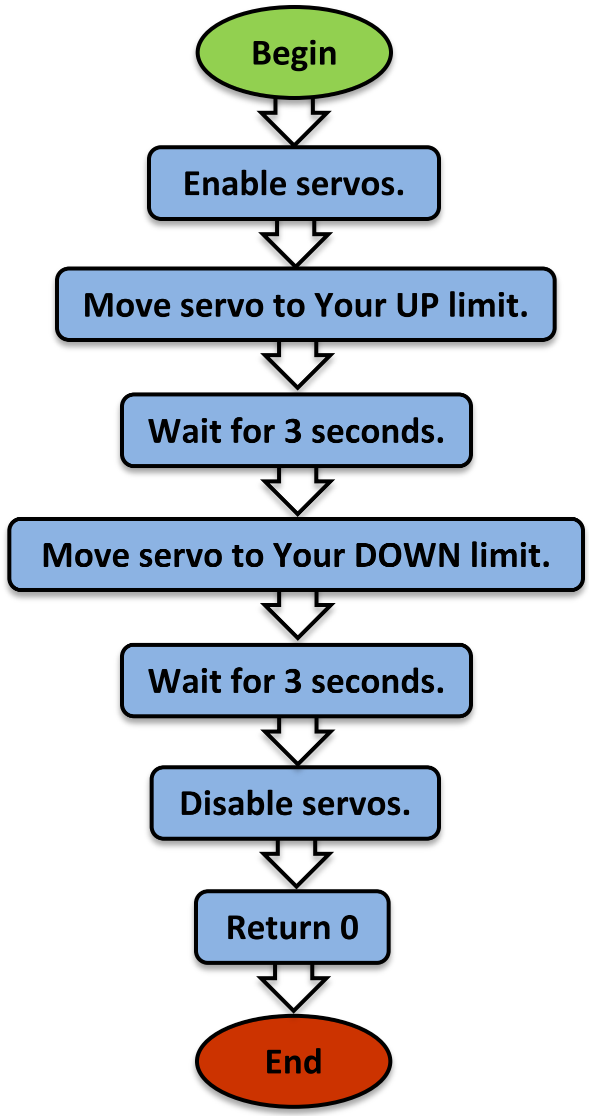
Comments

```
// 1. Enable servos.
// 2. Move servo to UP.
// 3. Wait for 3 seconds.
// 4. Move servo to DOWN.
// 5. Wait for 3 seconds.
// 6. Disable servos.
// 7. End the program.
```

Wave the Servo Arm

Analysis:

Flowchart



Commenting Within your Program

```
int main ()
{
  // arm = 0
  // down = 400
  // up = 1230
  printf("Wave Servo Exercise\n");
  return 0;
}
```

Make your comments after the first curly bracket and before the printf

Arm is plugged into servo port 0

Arm down position is 400

Arm up position is 1230

This (keeping track of <u>your</u> ports, positions, etc) could also be done in a notebook, but what if you misplace that notebook?

Variables

Some reasons to use a variable:

1. You don’t have to *remember* which value is a certain servo position – the computer remembers for you

2. It makes your program easier to read and understand

3. Makes it easier to debug your program

4. You can do computation and store results in variables

Variables

- A **variable** is a ***named*** container that stores a **type** of **value**

  A **variable** has the following three components:

  a. the **type** of data it stores (holds),

  b. the **name**, and

  c. the **value** it is currently storing.

a b c

```
int arm_up;
arm_up = 1230;
```

Use `int` as your data type if you want to store whole numbers (integers)

- Visualize/think of a **variable** like a *storage space* that holds a value with a name on it…
  - Servo "up" position
  - Servo "down" position
  - Etc.

`arm_up`	1230
`arm_down`	400

Variable Names

Each **variable** is given a unique name so we can identify it…

- Variable names can be *almost* anything you would like.
- Variable names can contain **letters**, **numbers**, and **underscores** ("_").
- Variable names **cannot** begin with a **number**.
- Variable names should be ***meaningful*** and not generic like "x"

An Example:

```
int arm_up;             // variable "declaration"
arm_up = 1230;          // variable "initialization"
```

You can do the declaration and initialization at the same time

```
int arm_up = 1230;
```

Working with Variables

1. *Declaring* a variable:

```
int arm_up;
```

2. *Initializing/setting* a variable:

```
arm_up = 1230;
```

What is `int`?

`int` stands for "integer". This means that the variable `arm_up` will have an integer (whole number) value.

2. *Calling* a variable:

```
arm_up
```

See the Team Homebase resources for more information on data types

#Botball®

Using Variable for Drive Motors

1. Variable declarations should go inside a block of code (i.e., inside the **{ }**) immediately after the starting curly brace (i.e., **{**) and before any other code.

```
int main ()
{
  // left = 3
  // right = 0

  printf("Drive and turn\n");

  motor(3, 100);
  motor(0, 100);
  msleep(1000);

  motor(3, -50);
  motor(0, 50);
  msleep(500);

  return 0;
}
```

Remove the forward slashes from your comments, add int for the data type and since it is now code add the semicolon

```
int main ()
{
  int left = 3;
  int right = 0;

  printf("Drive and turn\n");

  motor(left, 100);
  motor(right, 100);
  msleep(1000);

  motor(left, -50);
  motor(right, 50);
  msleep(500);

  return 0;
}
```


#Botball®

Using Variables for Servo Motors

1. Variable declarations generally go inside a block of code (i.e., inside the **{ }**), after the starting curly brace (i.e., **{**) and before any other code.

```
int main ()
{
  // arm port = 0
  // arm up = 1230
  // arm down = 400

  printf("Wave servo\n");
  enable_servos();
  set_servo_position(0,1230);
  msleep(500);
  set_servo_position(0,400);
  msleep(500);

  return 0;
}
```

Remove the forward slashes from your comments, add int for the data type and since it is now code add the semicolon

```
int main ()
{
  int arm_port = 0;
  int arm_up = 1230;
  int arm_down = 400;

  printf("Wave servo\n");
  enable_servos();
  set_servo_position(arm_port, arm_up);
  msleep(500);
  set_servo_position(arm_port, arm_down);
  msleep(500);

  return 0;
}
```

How many *potential* lines of code have to change if the arm servo is switched to port 3?

Slowing Down A Servo

Move the Servo Arm Using a Loop

1. Set counter to 200.
2. Set servo position to counter.
3. Enable servos.
4. *Loop:*Is counter < 1800?
Wait for 0.1 seconds.
Add 100 to counter.
Set servo position to counter.
5. Disable servos.
6. End the program.

```
int main()
{
intcounter = 200;
set_servo_position(0, counter);
enable_servos();
while(counter < 1800)
{
msleep(100);
counter = counter + 100;
set_servo_position(0, counter);
}
msleep(100);
disable_servos();
return0;
}
```

More Variables and Functions with Arguments

Data types

Creating and setting a variable

Variable arithmetic

Functions with arguments and return values

Variables (Quick Recap)

You can set the value of an int variable to any integer you choose and change it when you need in the code.

Note that a single equal sign (=) means *is assigned* (sometimes it is called the “assignment operator”).

```
int counter;
int ticks;
```

`counter`	3
`ticks`	??

“visualize” the variable storage spaces

So `counter = 3;` means “counter is assigned 3”.

And `ticks = 2000 * (1400.0 / circumferenceMM);` means “ticks is assigned 2000 times 1400.0 divided by circumference (in mm)” (used to calculate how many ticks needed to travel ~2meters).

Move the Servo Arm Using a Loop

Description: Write a program for the KIPR Robotics Controller that moves the DemoBot servo arm from position 200 to 1800 in increments of 100. Remember to **enable the servos** at the beginning of your program, and **disable the servos** at the end of your program!

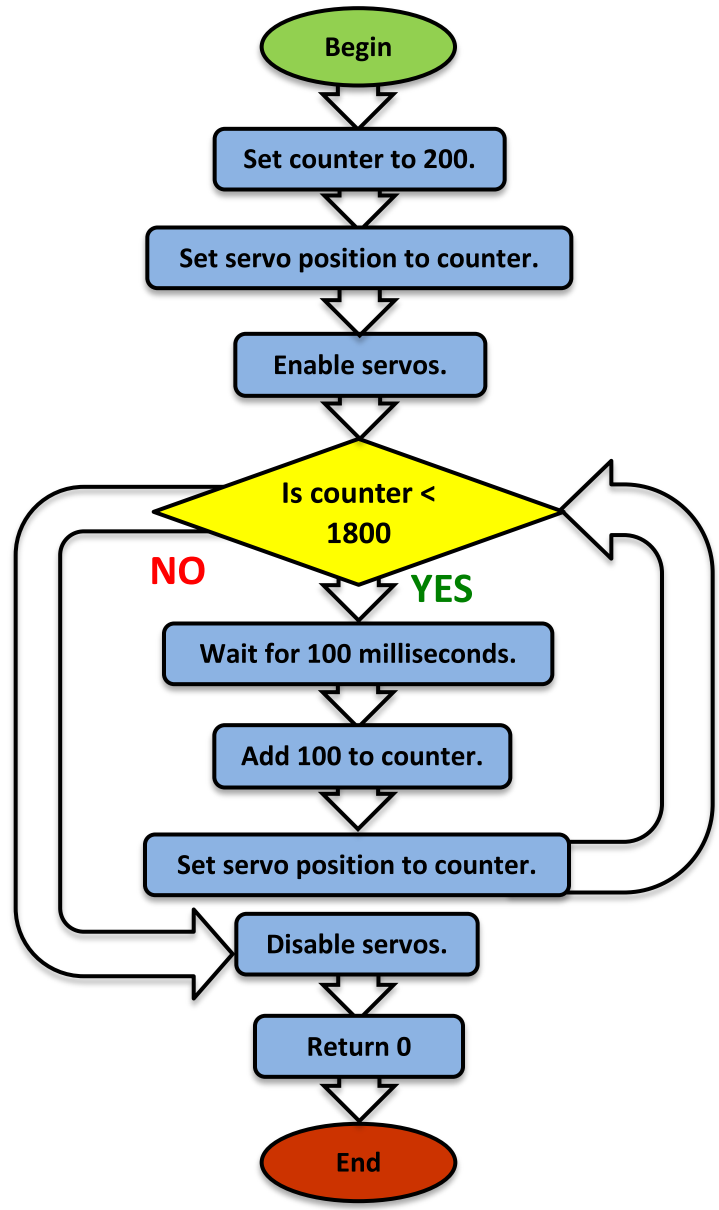
Analysis: What is the program supposed to do?

Pseudocode:

1. Set counter to 200.
2. Set servo position to counter.
3. Enable servos.
4. *Loop:* Is counter < 1800?
   - Wait for 0.1 seconds.
   - Add 100 to counter.
   - Set servo position to counter.
5. Disable servos.
6. End the program.

Move the Servo Arm Using a Loop

Analysis: Flowchart



#Botball

Move the Servo Arm Using a Loop

Solution:

Pseudocode

1. Set counter to 200.
2. Set servo position to counter.
3. Enable servos.
4. *Loop:* Is counter < 1800?
   Wait for 0.1 seconds.
   Add 100 to counter.
   Set servo position to counter.
5. Disable servos.
6. End the program.

Source Code

```
int main()
{
  int counter = 200;

  set_servo_position(0, counter);

  enable_servos();

  while (counter < 1800)
   {
    msleep(100);
    counter = counter + 100;

    set_servo_position(0, counter);
   }
  msleep(100);

  disable_servos();

  return 0;
}
```

Making Smarter Robots with Sensors

`analog()` and `digital()` sensors

`wait_for_light()` function

Sensors

- You might have realized how difficult it is to be consistent with *just* "**driving blind**".

- By adding **sensors** to our robots, we can allow them to **detect things** in their environment and **make decisions** about them!

- Robot **sensors** are like human **senses**!
  - What **senses** does a **human** have?
  - What **sensors** should a **robot** have?

Analog and Digital Sensors

Analog Sensors

- **Range of values:**

  0 to 4095

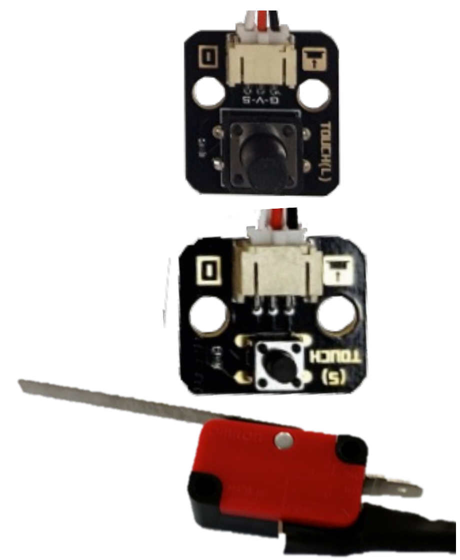
- **Ports:** 0 to 5
- **Function:** `analog(port #)`
- **Sensors:**
  - Light
  - Rangefinder
  - Small reflectance
  - Large reflectance
  - Slide sensor

Digital Sensors

- **Range of values:**

  0 (not pressed) or 1 (pressed)

- **Ports:** 0 to 9
- **Function:** `digital(port #)`
- **Sensors:**
  - Large touch
  - Small touch
  - Lever touch



Professional Development Workshop

#Botball®

Remember Your Sensor Functions

You call for the analog sensor value with a function

- You have 6 analog ports (0 through 5)

`analog(Port#)` `analog(1)`

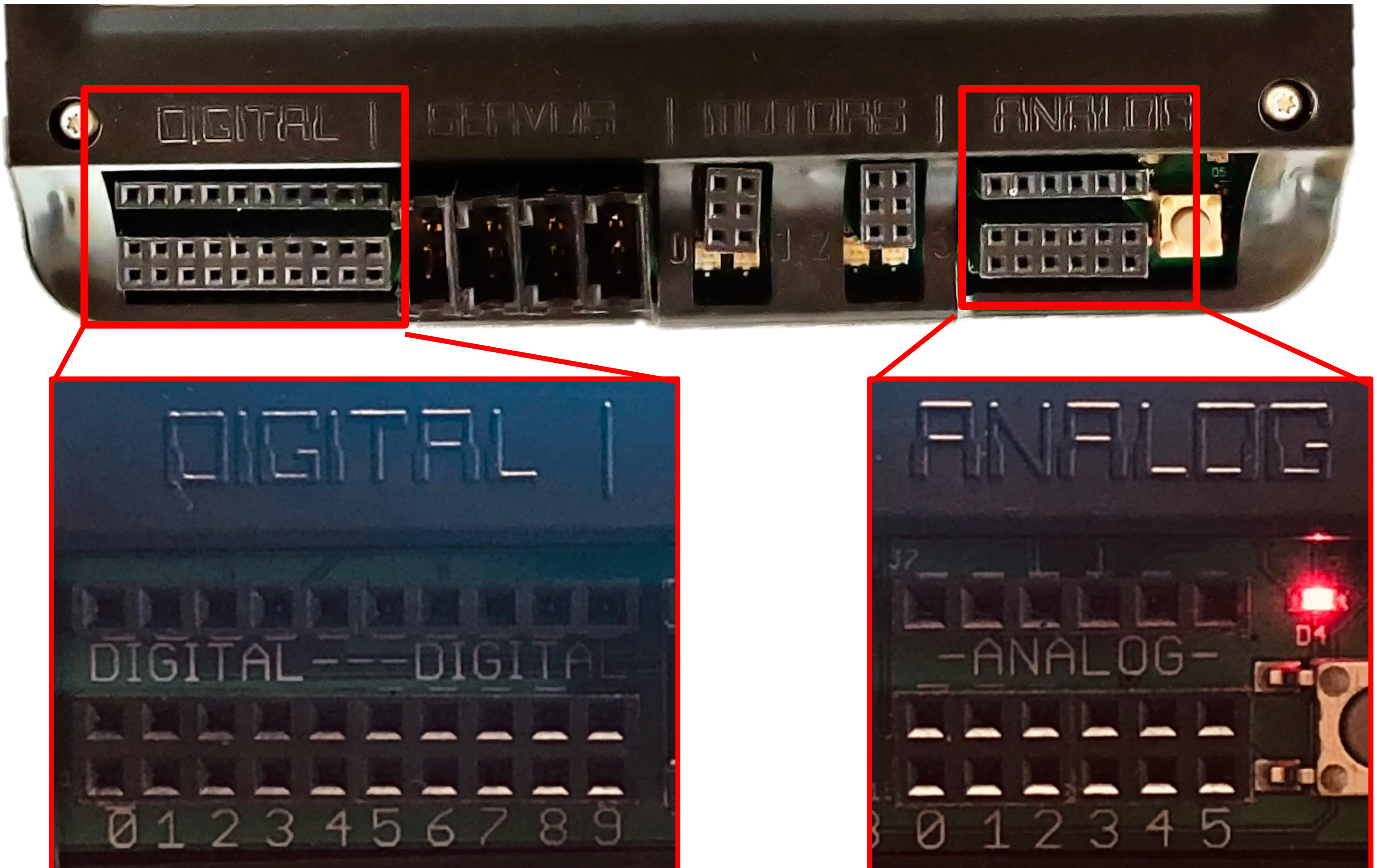
You call for the digital sensor value with a function

- You have 10 digital ports (0 through 9)

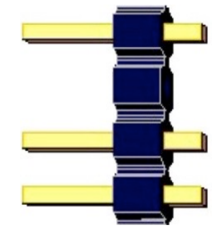
`digital(Port#)` `digital(8)`

NOTE: when you call these functions they ***"read the sensor" at that instant in time*** and return a single INTEGER value into the "code" where they were called.

KIPR Robotics Controller Sensor Ports



Sensor Plug Orientation



Digital Sensors
Ports # 0 to 9

Analog Sensors
Ports # 0 to 5

#Botball

Detecting Touch

There are many digital sensors in your kit that can detect touch…

Large Touch

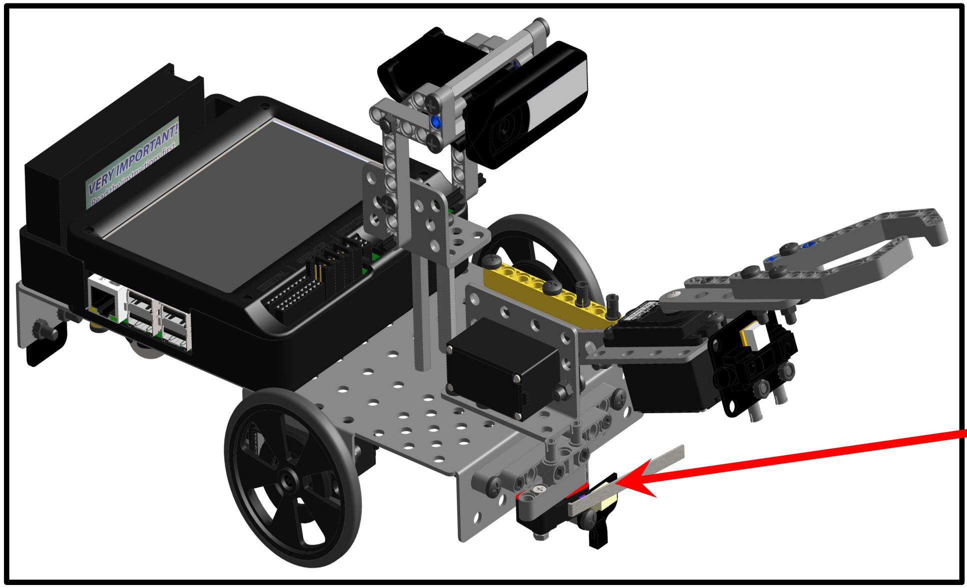
Small Touch

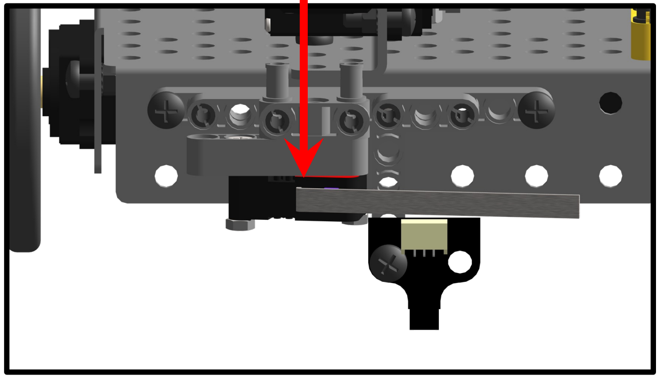
Lever Touch

Professional Development Workshop

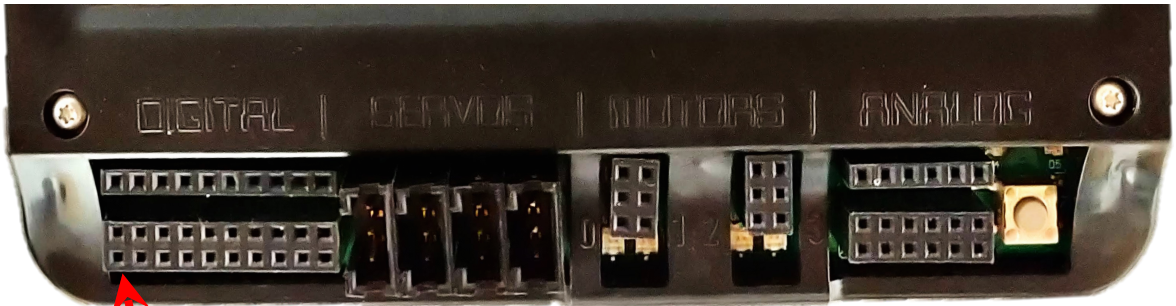
#Botball®

Mounted Lever Touch Sensor

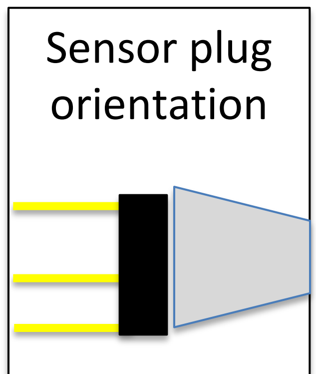


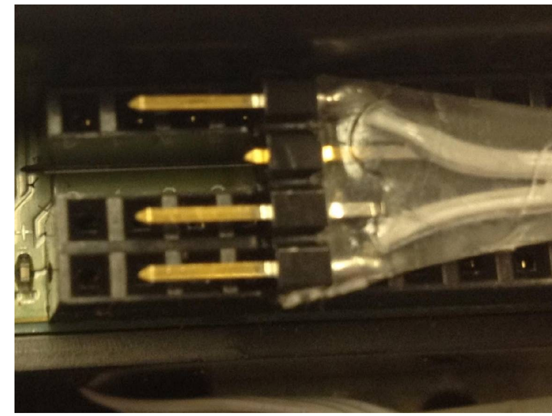


Plug in the Lever Touch Sensor



Plug your touch sensor into digital port 0





Close-up of sensor plug orientation

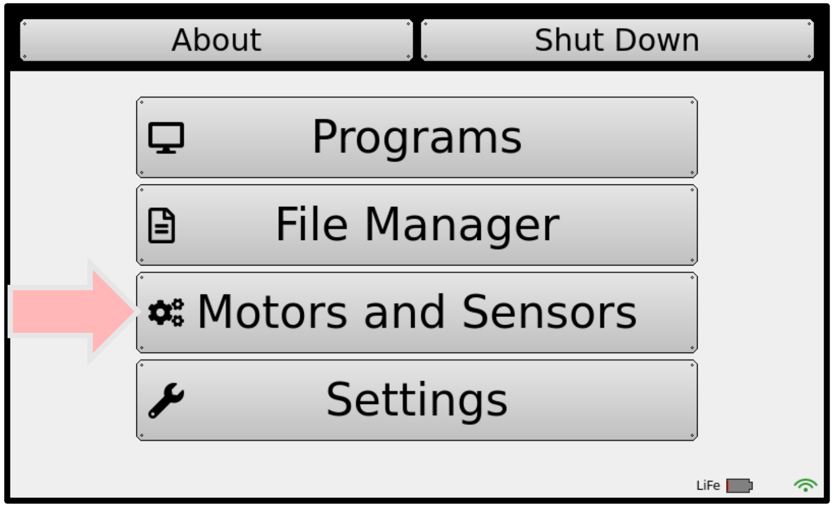
Professional Development Workshop

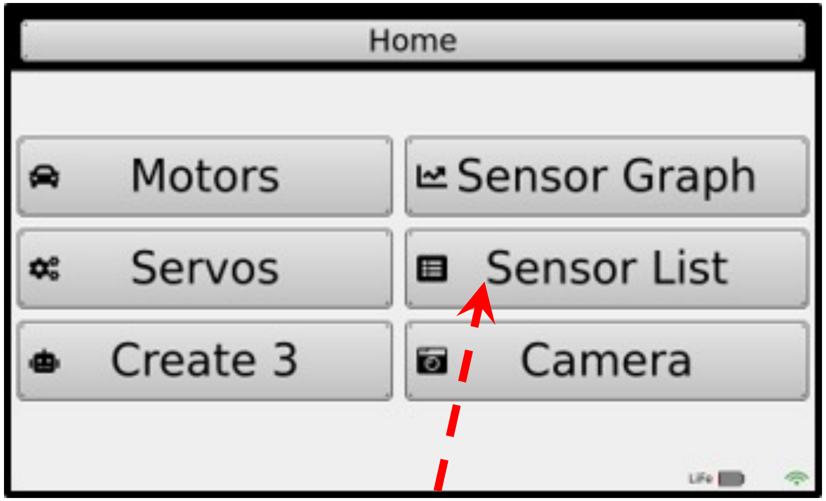
#Botball®

Reading Sensor Values from the Robot

You can access the Sensor Values from the Sensor List on your KIPR Robotics Controller

- This is very helpful to get readings from all of the sensors you are using, and then you can then use the values in your code

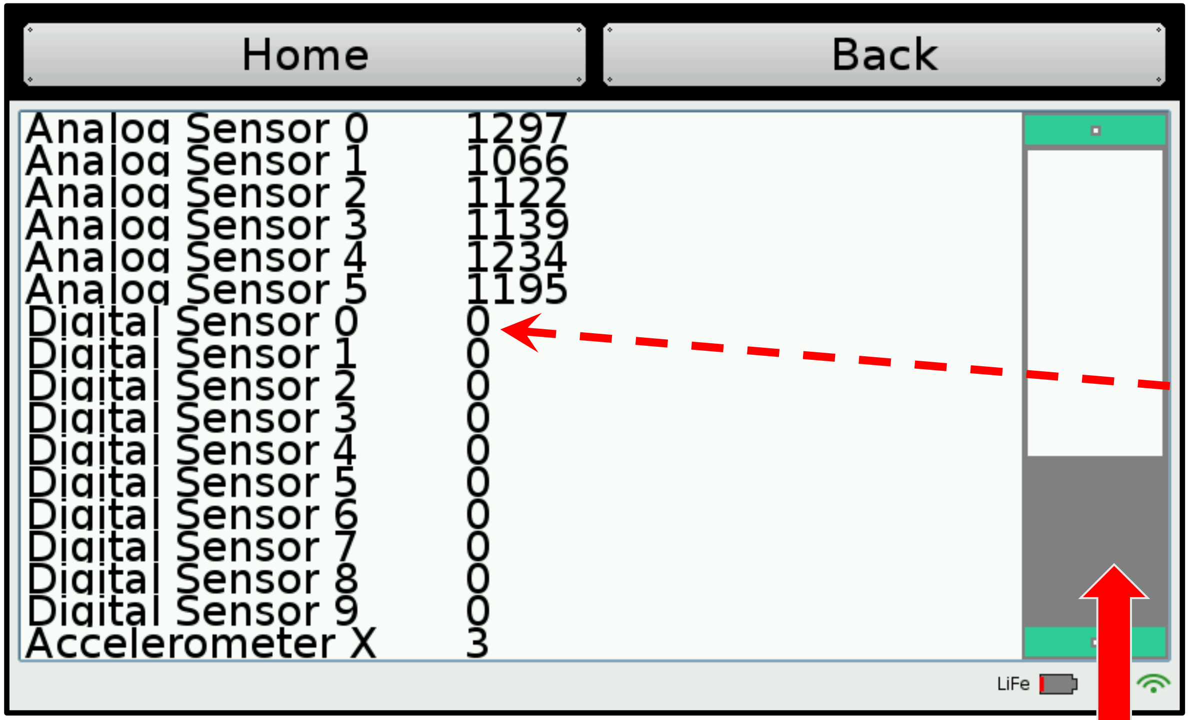


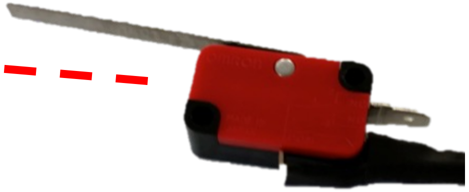


Select Sensor List

#Botball

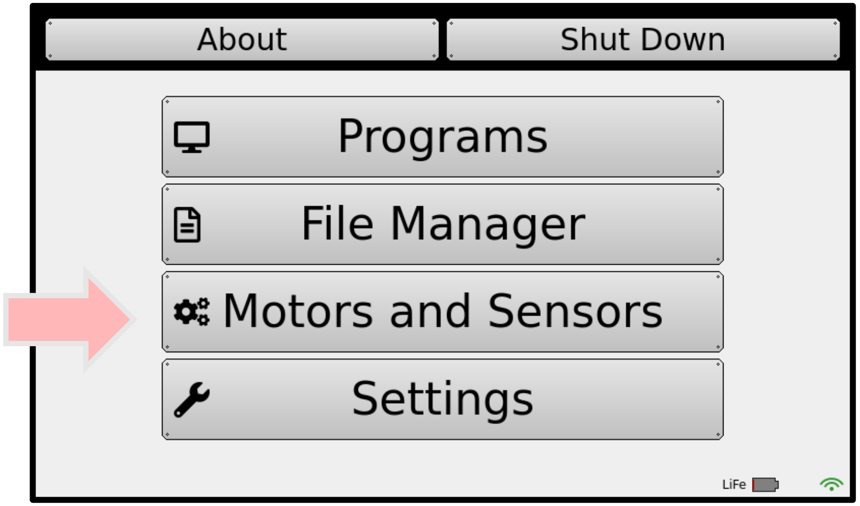
Reading Sensor Values from the Robot

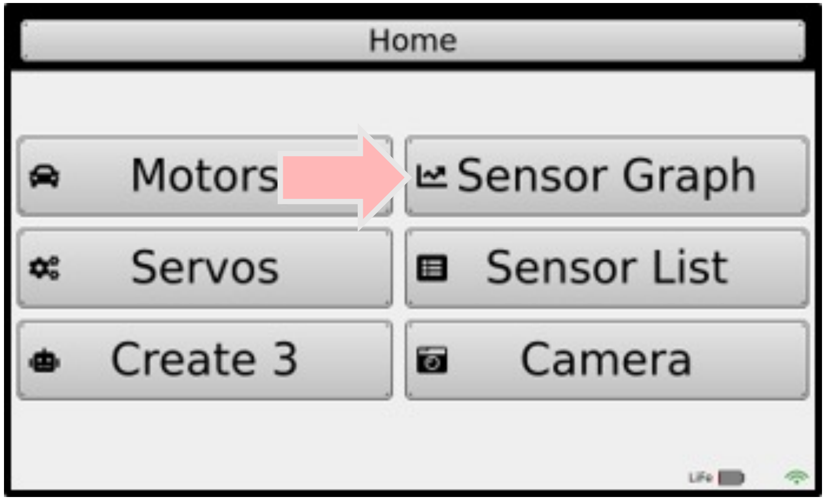


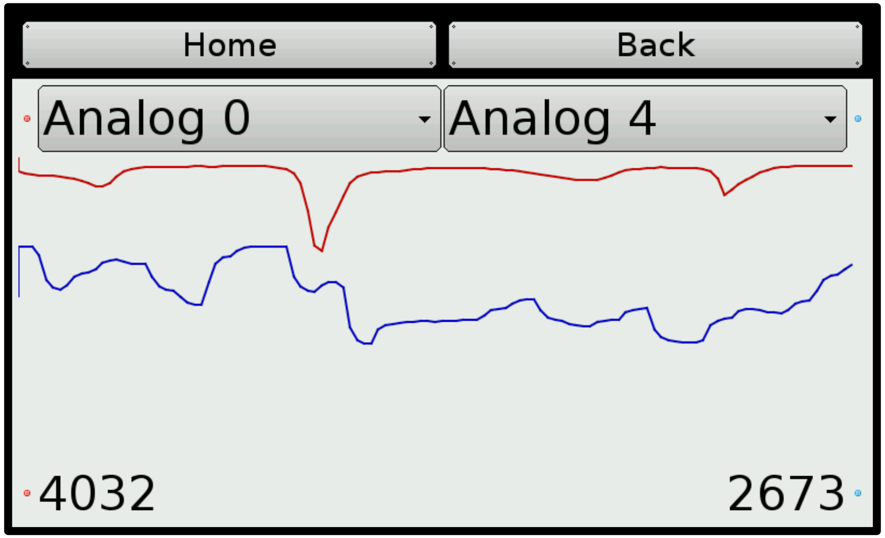


Scroll down to the digital sensor and read the value when your touch sensor is pressed and when it is not pressed

Use the Sensor Graph







#Botball®

Introduction to `while` loops

Program flow control with *sensor driven* loops

`while` and Boolean operators

Program Flow Control with Loops

- What if we want to ***repeat*** the same "item/action" over and over (and over and over)?
  - For example, checking to see if a touch sensor has been pressed.
- We can do this using a **loop**, which controls the **flow** of the program by repeating a **block of code**.

while Loops

We accomplish this loop with a `while` statement.

`while` statements keep a block of code running (repeating/looping) so that sensor values can be continually checked and a decision made.

The while statement checks to see if something is true or false (via Boolean operators).

```
while ( condition )
{
   Code to execute while
   the condition is true
}
```

Notice there is no terminating semicolon after the while statement

while Statement

Type of sensor; analog, digital, analog

Port number; analog (0-5) digital (0-9)

Notice **no** terminating statement

```
while (digital(port#) == 0)
{
  motor(0,75);
  motor(3,75);
}
```

Code to execute while the condition is true

Boolean logic;
- > Greater than
- >= Greater than or equal
- < Less than
- <= Less than or equal
- == Equal to
- !=Not equal to

#Botball®

`while` Loops

The `while` loop checks to see if a **Boolean test** is **true** or **false**...

- If the **test** is **true**, then the `while` loop **continues** to execute the **block of code** that *immediately* follows it.
- If the **test** is **false**, then the `while` loop **finishes**, and the line of code *after* the **block of code** is executed.

```
int main()
{
  // Code before loop

  while (Boolean test)
  {
    // Code to repeat ...
  }

  // Code after loop

  return 0;
}
```

Block Header
(no semicolon!)

Begin

End

Built-In Digital Sensor

- The Wombat has a built-in physical button on the right side of the controller

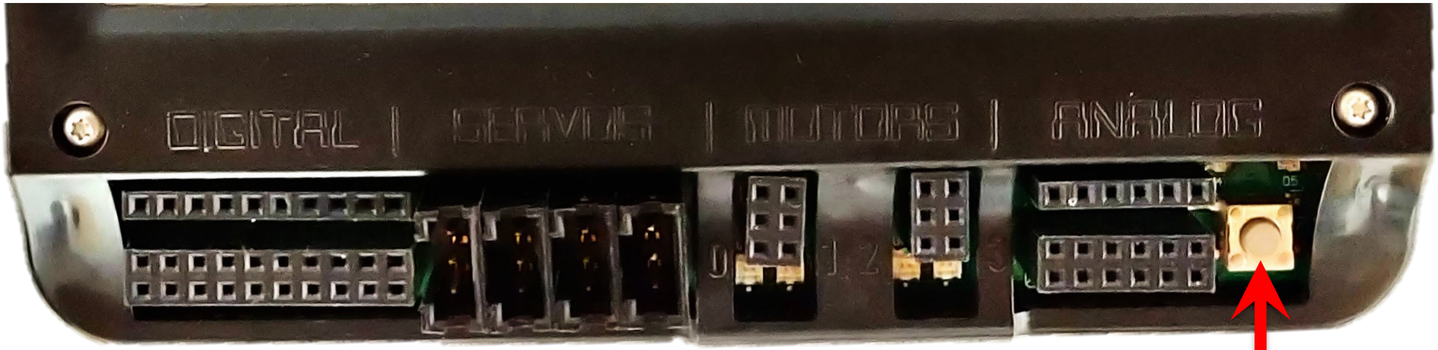
  `push_button()`

The Wombat also has built-in touch screen buttons on the bottom of the robot screen (a, b, c and more if needed)

`a_button()` **`b_button()`** **`c_button()`**

- returns a value of 1 if the button is currently pressed
- returns a value of 0 if the button is not being pressed at that time

while Loop on Push Button



push_button

Example:

```
int main()
{
  // Has push button been touched?
  while(push_button() == 0)
  {
    printf("Press the Push Button!\n");
  }

  printf("Ahh! Something touched my Push Button!\n");
  return 0;
}
```

`while` and Boolean Operators

The **Boolean test** in a `while` loop is asking a question:

Is this statement true or false?

- The **Boolean test** (question) often compares two values to one another using a **Boolean operator**, such as:

`==`	Equal to (NOTE: two equal signs, not one which is an assignment!)
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

Boolean Operators Cheat Sheet

Boolean	English Question	True Example	False Example
A == B	Is A **equal to** B?	5 == 5	5 == 4
A != B	Is A **not equal to** B?	5 != 4	5 != 5
A < B	Is A **less than** B?	4 < 5	5 < 4
A > B	Is A **greater than** B?	5 > 4	4 > 5
A <= B	Is A **less than or equal to** B?	4 <= 5 5 <= 5	6 <= 5
A >= B	Is A **greater than or equal to** B?	5 >= 4 5 >= 5	5 >= 6

Drive Until Sensor is Pressed

Description: Write a program for the KIPR Robotics Controller that drives the DemoBot forward until a touch sensor is pressed, and then stops.

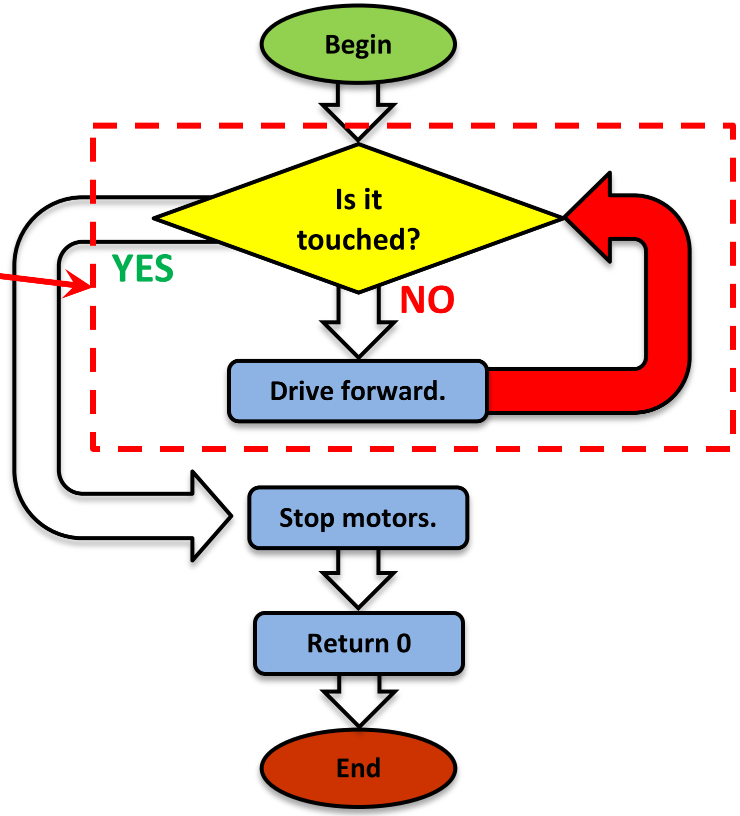
Analysis: What is the program supposed to do?

Pseudocode	Comments
1. Drive forward.	`// 1. Drive forward.`
2. Loop: Is not touched?	`// 2. Loop: Is not touched?`
3. Stop motors.	`// 3. Stop motors.`
4. End the program.	`// 4. End the program.`

Drive Until Sensor is Pressed

Analysis: Flowchart

This part of the code is the **loop**.



Drive Until Sensor is Pressed

Solution:

Pseudocode

```
1. Loop: Is it Touched?
   1.1 Drive Forward
2. Stop Motors
3. End the Program
```

Source Code

```
int main()
{
  printf("Drive until bump\n");
  while (digital(0) == 0)
  {
    motor(0, 75);
    motor(3, 75);
  }

  ao();

  return 0;
}
```


#Botball®

Changing the Condition

1. **Change the <u>expected</u> (test condition) value from 0 to 1**
2. Objective: Predict/describe what you think the robot will do
3. Run the program

```
int main()
{
    printf("Drive until bump\n");
    while (digital(0) == 1)
    {
        motor(0,50);
        motor(3,50);
    }

    ao();
    return 0;
}
```

Square Up using Bump Sensors

- Sometimes it is useful to have a robot "square up" to then drive straight 90 degrees from a "wall".
- This can be done in a number of ways and one common one is to use two bump (digital) sensors mounted at two "corners" of the back of the robot.
- What follows are **two** possible "algorithms/methods"

Square Up using Bump Sensors

Description: Use the pair of touch sensors on the back of the DemoBot to square up on a wall or PVC structure.

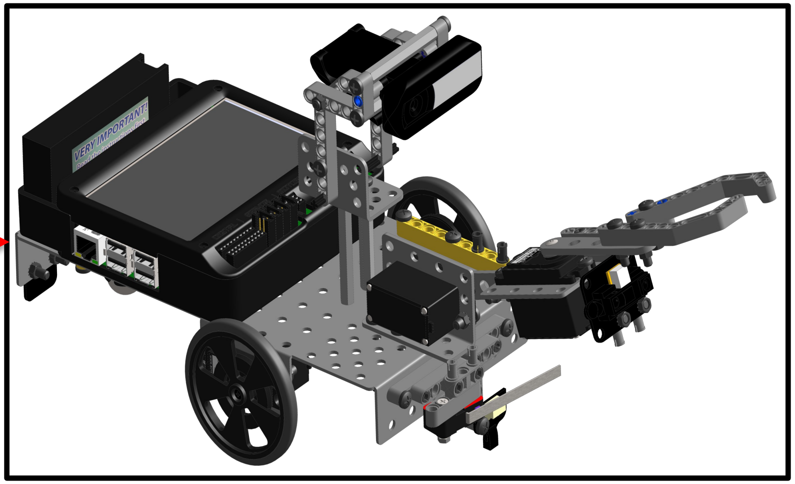
Background diagnostic work: You will need to plug your digital button sensors into digital ports. A good strategy might be to use the same port number as your motor port. E.g. right motor plugged into port 0, right button sensor plugged into digital 0.

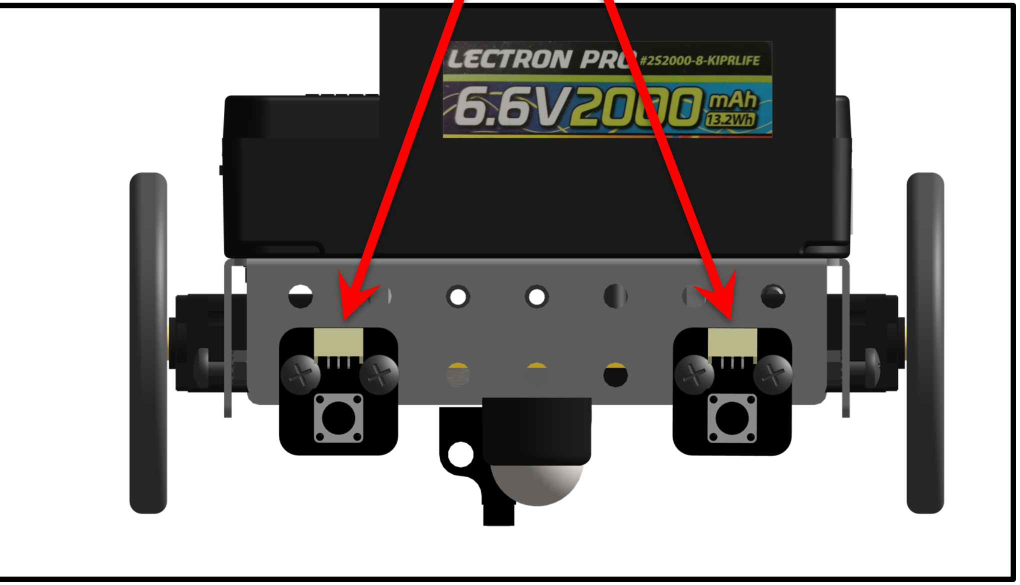
Key Coding Concepts: Each of the digital sensors will need to be “married” to a wheel in code. One way to handle this is to nest two (if-else pairs) inside of a while loop. Essentially, one of these pairs will control the left wheel and one will control the right wheel.

Method #1: The robot will move backward until it senses either back bump sensor is pushed. Upon a sensor being pushed, its’ corresponding wheel will freeze, the other wheel will continue to move backward until its’ sensor is pushed. At the point, the robot will exit the loop

Bonus: Upon completing a square up, your robot will move forward 1000 ticks.

Large Touch Sensors Mounted on Back of Robot





Square Up Method #2

Description: Write a program for the KIPR Robotics Controller that drives backward to orient your robot perpendicular to a "PVC wall".

Analysis: What is the program supposed to do?

Pseudocode	**Comments**
1. Loop: Both sensors touched?	`// 2. Loop: Are both sensors pressed?`
2. *If* **only** right sensor touched?	`// 3. If right sensor is touched turn CCW`
3. *Else If* **only** left sensor touched?	`// 4. Else-If left sensor is touched turn CW`
4. *Else* drive backward	`// 5. Else drive backward`
5. End the program when both touched	`// 4. End the program.`
	`//  note: CCW means counter clockwise (CW)`

#Botball®

Square Up Method #2 Solution

```
int main()
{
   printf("Back Up to Square Up :-)\n");
   while ((digital(3) == 0) || (digital(0) == 0)) // Left or Right is not pressed
   {
     if ((digital(3) == 0) && (digital(0) == 1))   // Right is pressed (not Left)
     {
       motor(3, -90);
       motor(0, 10);    // turn CCW backwards with right motor at zero
     }
     else if ((digital(3) == 1) && (digital(0) == 0))
     {
       motor(3, 10);    // turn CW backwards with left motor at zero
       motor(0, -90);
     }
     else // just keep going backwards
     {
       motor(3, -75); motor(0, -75);
     }
  }

  ao();
  return 0;
}
```

Assumes that motor 0 and digital 0 are on the right side and motor 3 and digital 3 are on the left side.

#Botball

Tophat Sensors to Square Up

Description: Use a pair of tophat sensors to assist you in squaring up on a black line.

Physical Build: Each large tophat sensor will need to be mounted, pointed down and approximately 1/4” off of the surface. To make it an accurate square up, the two top hats will need to be the mounted on opposite sides but equidistance from the wheels. (Ask for assistance if needed) This sensor-based system is easier with large tophat sensors compared to small top hats, but can be done with either.

Background diagnostic work: You will need to plug your tophat sensors into analog ports. A good strategy might be to use the same port number as your motor port. E.g. right motor plugged into port 0, right tophat plugged into analog 0. You will want to determine the black and white value for each sensor and determine a midpoint that will allow you denotate

Key Coding Concepts: Each of the tophat sensors will need to be “married” to a wheel in code. One way to handle this is to nest two (if-else pairs) inside of a while loop. Essentially, one of these pairs will control the left wheel and one will control the right wheel.

Goal #1: The robot will move forward until it senses a black line. Upon a sensor reading black, its’ corresponding wheel will freeze, the other wheel will continue to move forward until its sensor reads black. At the point, the robot will exit the loop

Bonus: Upon sensing black, your robot will slowly move backwards until the exact black/white line is reached

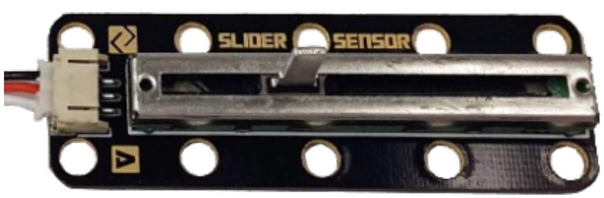
Measuring Distance

Infrared “ET” Range (distance) Sensor

Learning about Analog Sensors

- Returns the analog value of the port (a value in the range 0 to 4095). Analog ports are numbered 0 through 5.
- Light, slide, range and reflectance sensors are examples of sensors you would use in analog ports.


Light Sensor

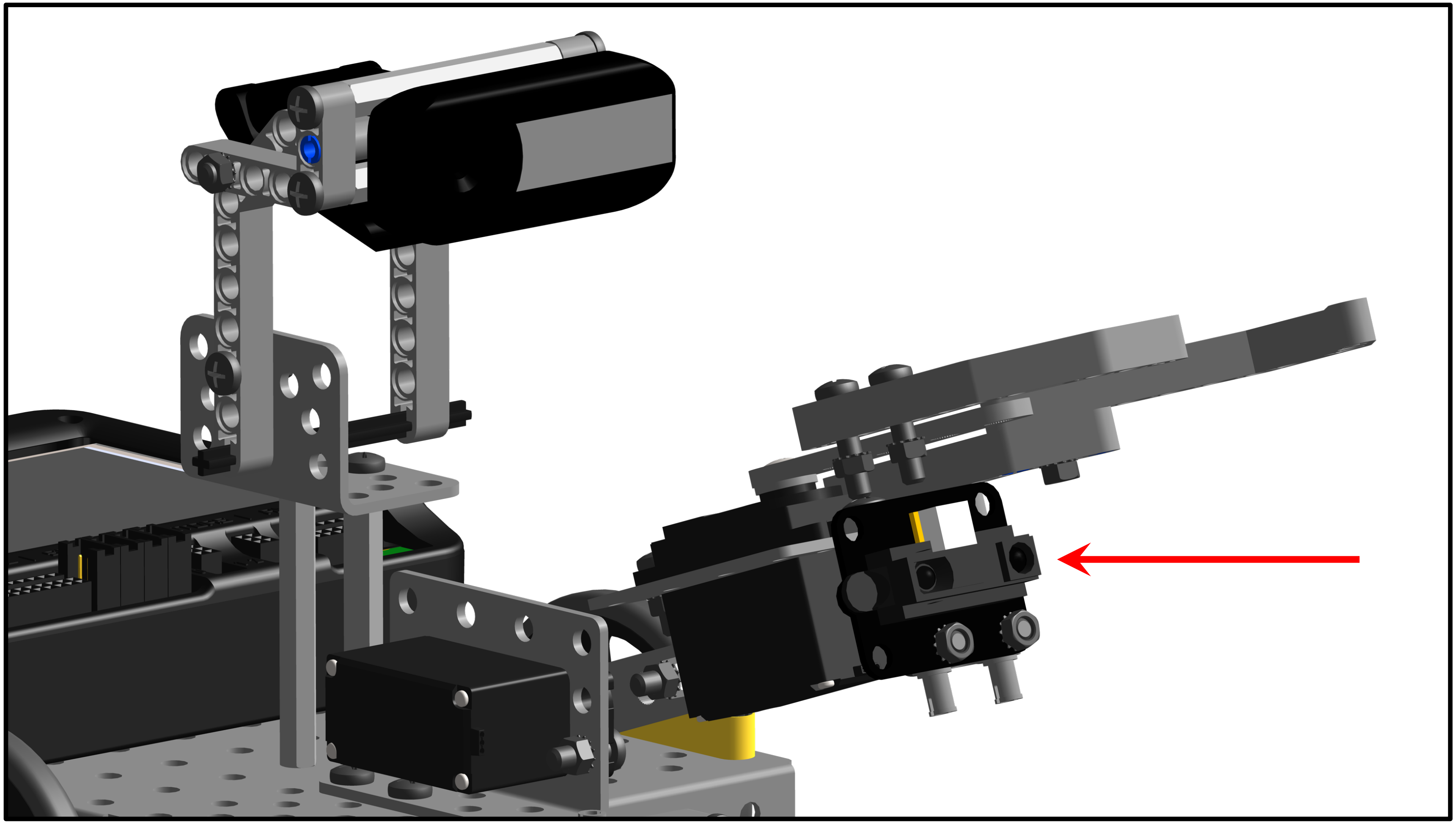


Slide Sensor

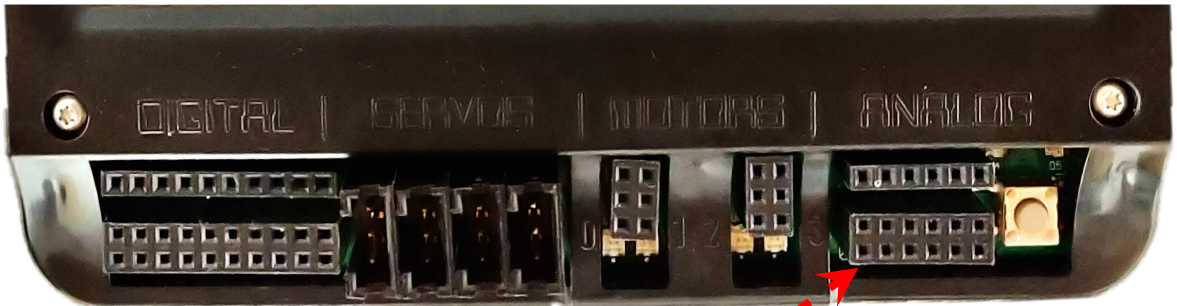

“ET” Range Sensor

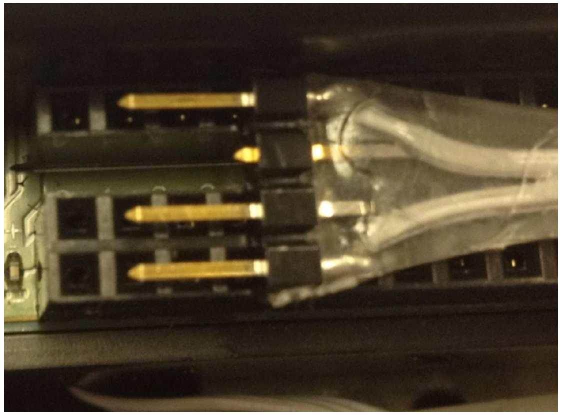

Small Reflectance Sensor

Range Sensor Mounted on Robot



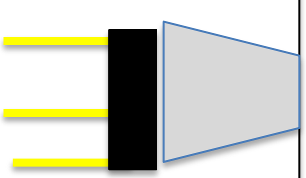
Plug in your Range Sensor





Plug your analog sensor into analog port 0

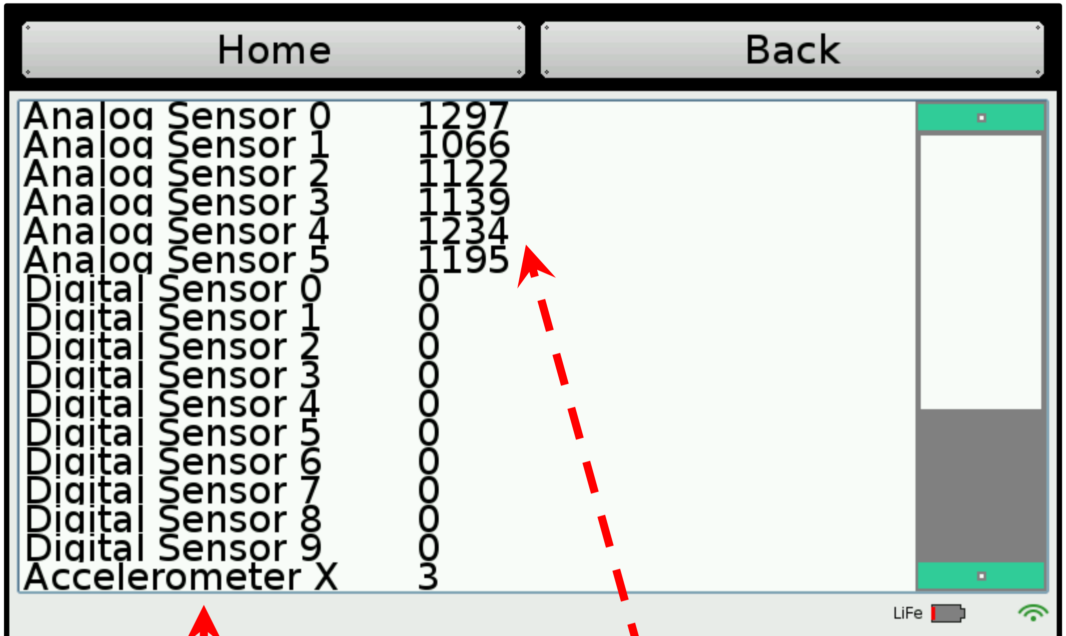
Sensor plug orientation



Range Sensor

#Botball®

Check ET Sensor on Wombat Screen



Sensor Ports

Sensor Values

“ET” Range Sensor
(or Wall·E?)

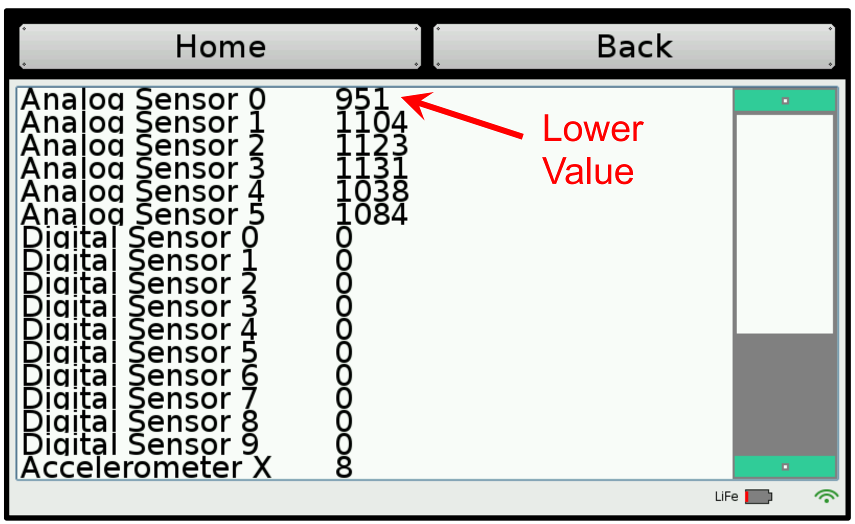
Read the values when your ET sensor is pointed at an object and slowly move it toward/away from the object

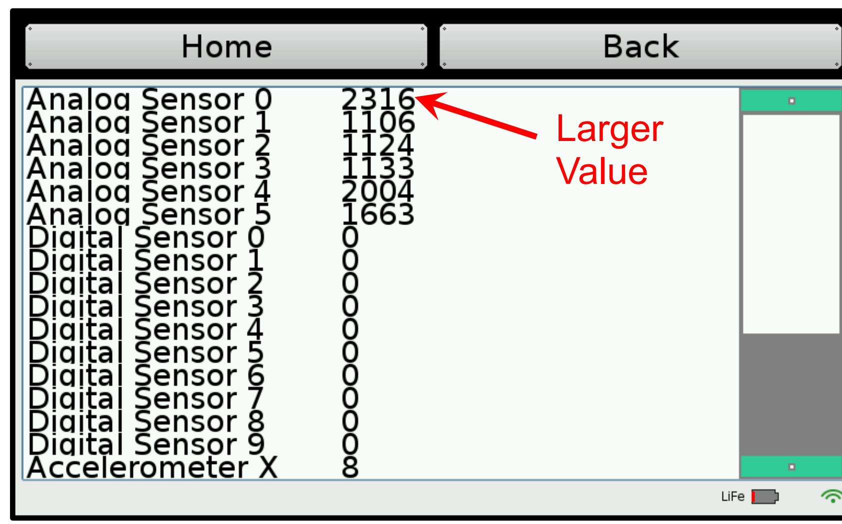
(this is a distance sensor)

#Botball®

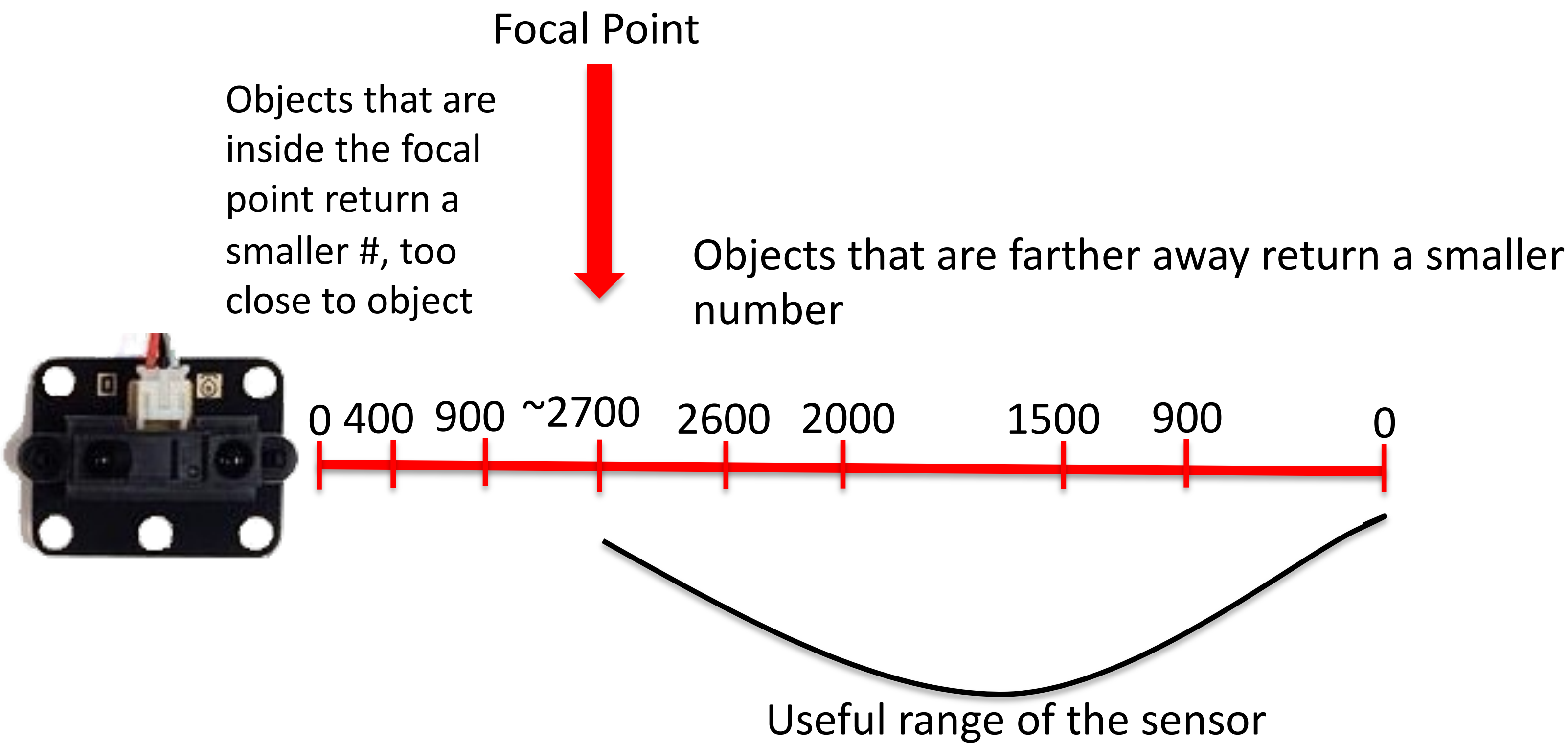
ET (Wall·E) Sensor Information

- **Low values:** indicate greater distance (farther from robot)
- **High values:** indicate shorter distance (closer to robot)
- Optimal range is ~4” and further away
- 0” to 3.5” values are not optimal
- Objects closer than the focal point (~4”) will have the same readings as those further away.





ET Sensor Values



You may need to adjust the value chosen, up or down a little, for your desired distance from an object. Optimal distance is about 4” away from the sensor.

ET Sensor Focal Point Problem

Using the sensor values you should see that the farther away an object is the lower the value returned. The closer an object is the higher the value until you get within ~4" of the sensor.

1. Extend your arm in front of you with your thumb pointed up.
2. Focus on your thumb and then slowly bring your thumb toward your face.
3. What happens when your thumb gets close to your face?
   – Did it get blurry? Yes! It got within the focal point of your eyes (where you could focus on it and make it clear)
4. The ET sensor also has a focal point and if the object is too close the sensor cannot tell if it is close or far away.
5. When attaching your ET sensor to your robot consider the ~4" distance from your sensor to its focal point

Learning to Use an ET Analog Sensor

Type of sensor: analog, digital,

Port number: analog 0-5 digital 0-9

Notice **no** terminating semi-colon

```
while (analog(port#) <= ?)
{
   motor (0,40);
   motor (3,40);
}
```

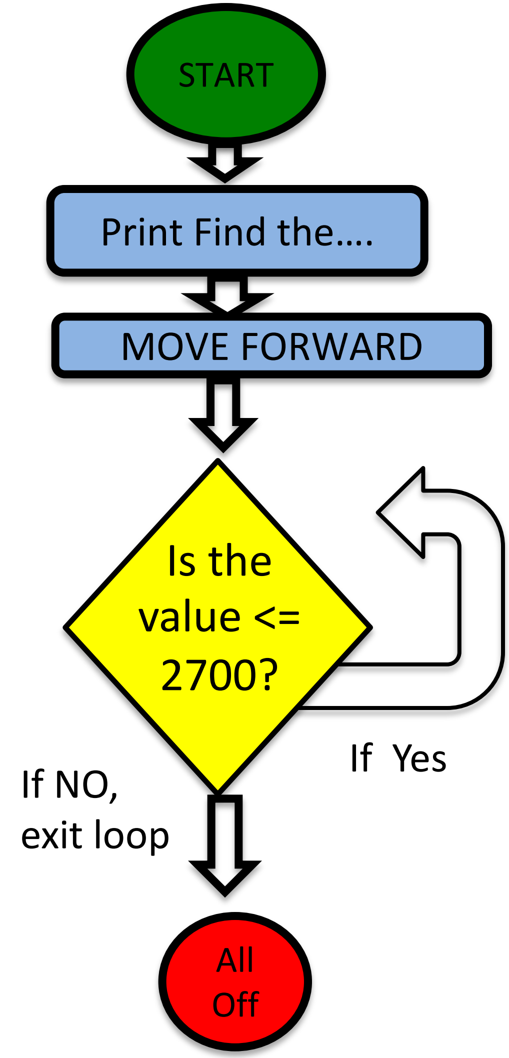
Boolean logic

- > Greater than
- >= Greater than or equal
- < Less than
- <= Less than or equal
- == Equal to
- !=Not equal to

What you want it to repeat while checking to see if the while statement is true

Find the Wall

1. Open a new project, name it “Find the Wall”.
2. Write and compile a program that will find the wall and stop.



Pseudocode (Task Analysis)

```
1. Print Find the Wall and Stop
2. Check the sensor value in analog port 1,
Is the value <= 2700?
3. Drive forward as long as the value is <=
2700 (or your determined value)
4. Exit loop when value is 2700(or your
determined value) or greater
5. Shut everything off
```

while "find the wall" Solution

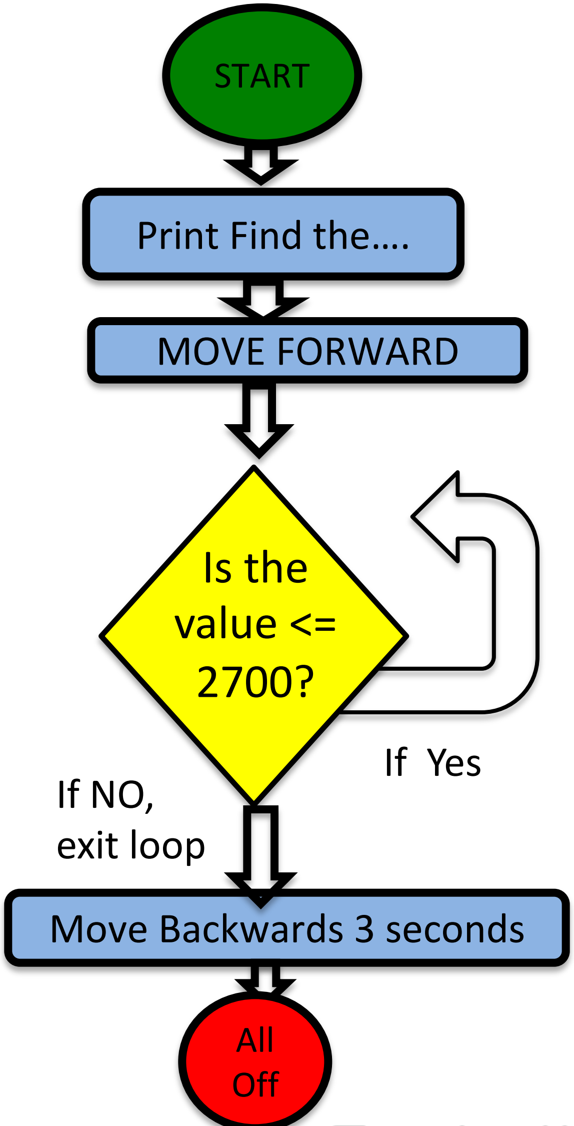
```
#include <kipr/botball.h>

int main()
{
   printf("Find the wall\n");
   while (analog(0) <= 2700)
   {
      motor(0,40);
      motor(3,40);
   }

  ao();
  return 0;
}
```


#Botball

ET - Find the Wall and Back Up

Pseudocode (Task Analysis)

1. Print Find the Wall and Back Up
2. Check the sensor value in analog port 1, Is the value <= 2700?
3. Drive forward as long as the value is <= 2700 (or your determined value)
4. Exit loop when value is 2700(or your determined value) or greater
5. Back up for 3 seconds
6. Shut everything off



This is an example of taking a shorter program and building/expanding upon it to accomplish more.

#Botball

Analog Sensor: Small Top Hat Sensors

This is a reflectance sensor that works at short distances. There is an infrared (IR) emitter and an IR collector in this sensor. The IR emitter sends out IR light and the IR collector measures how much is reflected back.

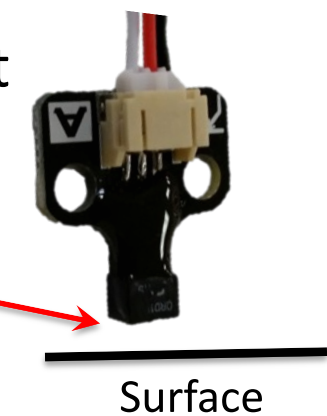
Amount of IR reflected back depends on surface texture, color and distance to surface among other factors.

This sensor is excellent for line following

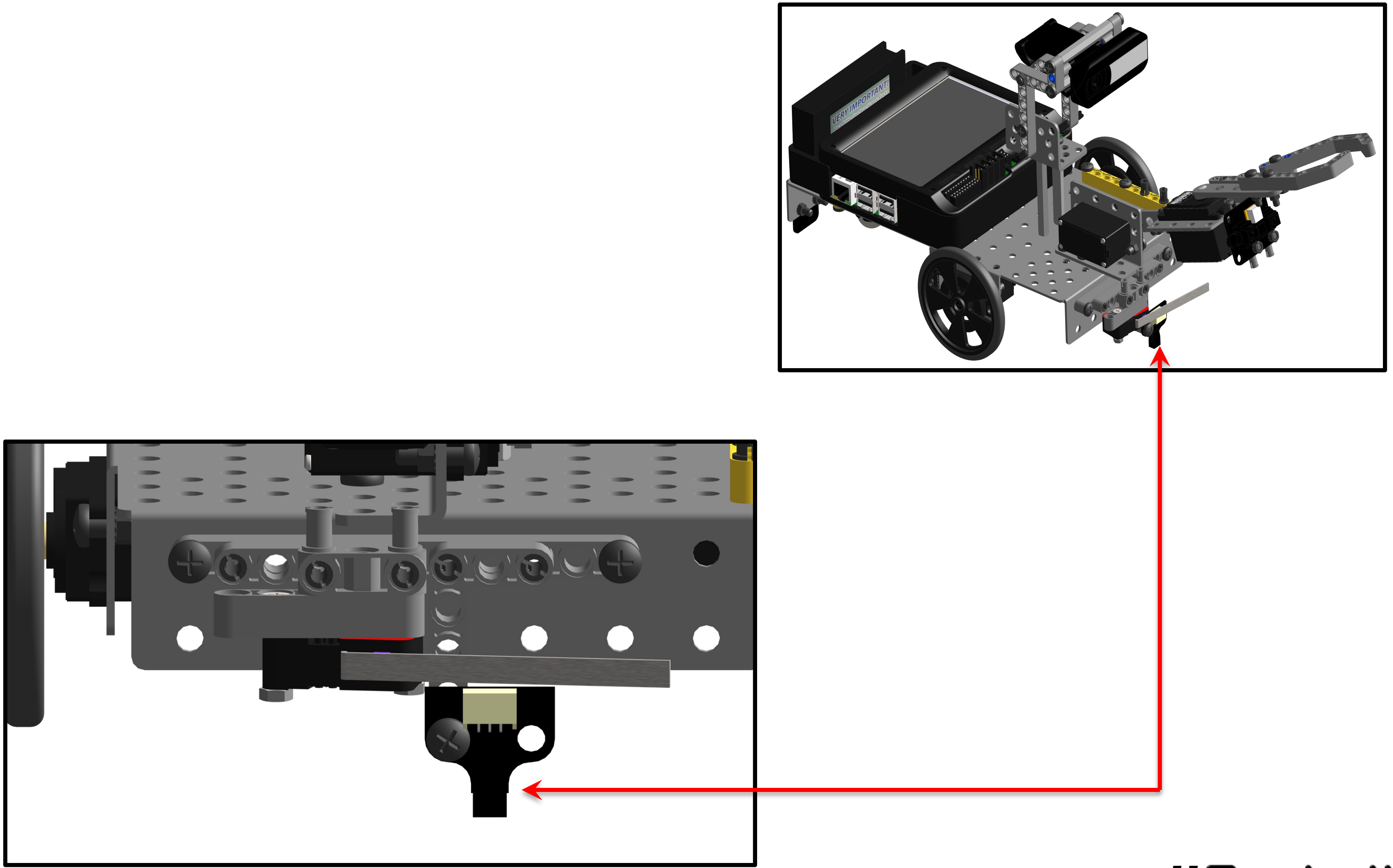
Black materials typically absorb IR and reflect very little IR while white materials typically absorb little IR and reflect most of it back

- ***<u>If this sensor is mounted at a fixed height above a surface</u>***, it is easy to distinguish a dark color from a light color
- Connect to an analog port (0 to 5)

1. This is an **`analog()`** sensor so plug it into any of your analog ports 0 through 5
    - Values returned can be between 0 and 4095
    - Mount the sensor on the front of your robot so that it is pointing to the ground and ~1/4” from the surface



Mounted Sensor on DemoBot

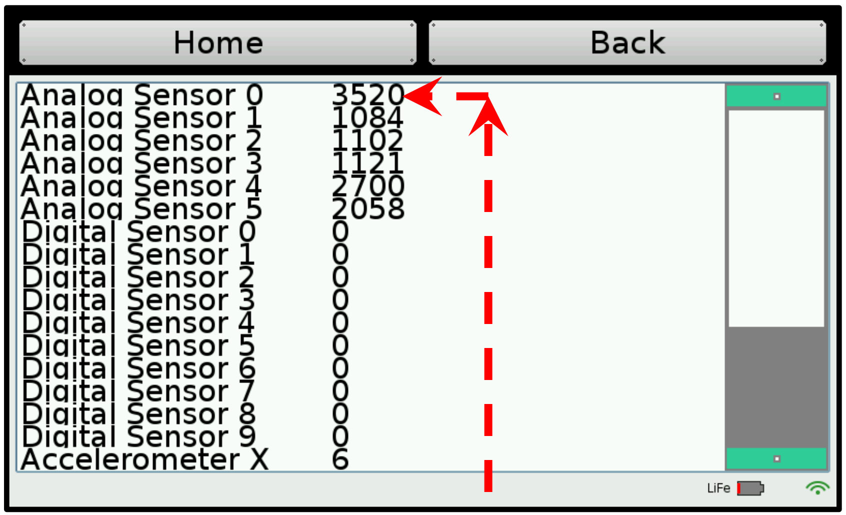


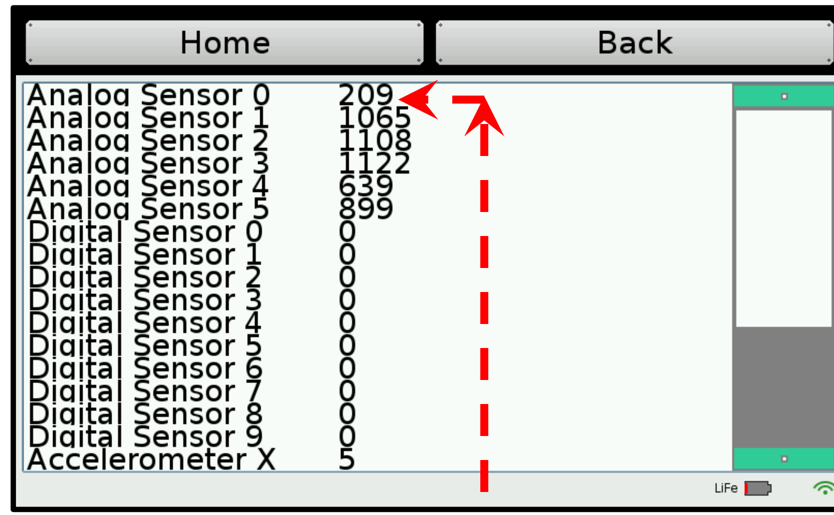
#Botball®

Reading Sensor Values From the Sensor List

With the IR sensor plugged into analog port #0

- Over a white surface the value is (~200)
- Over a black surface the value is (~3200)



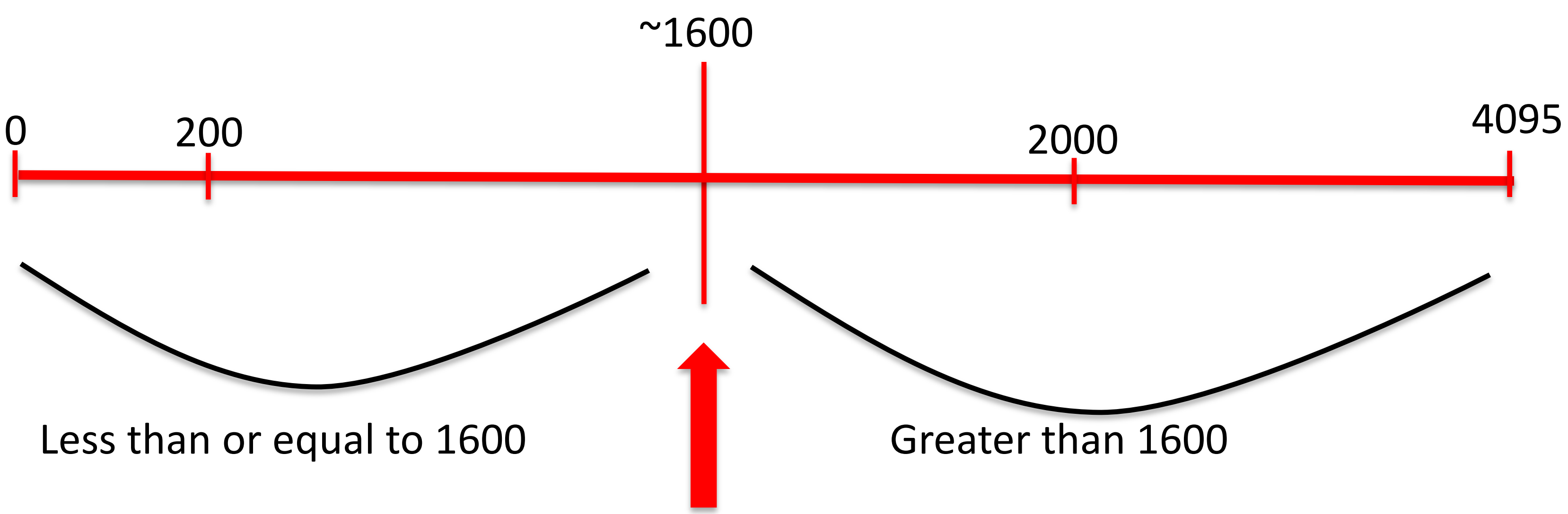


Your IR sensor is correctly mounted when you have values between ~2900 and ~3800 on the Black Surface

Your IR sensor is correctly mounted when you have values between ~175 and ~300 on the White Surface.

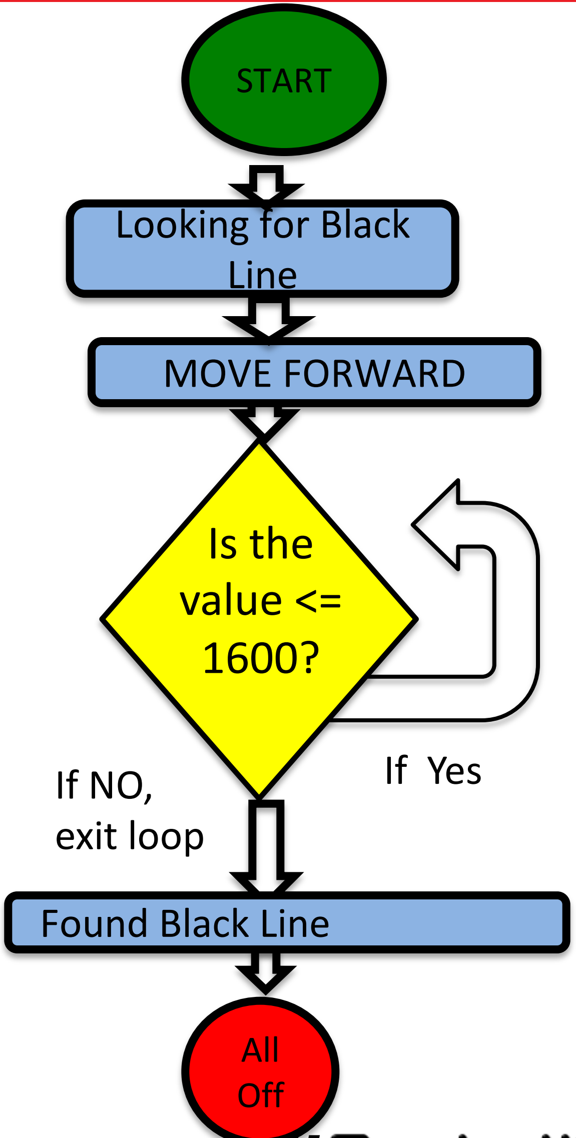
Understanding the IR Values

1. Place your IR analog sensor in one of the analog ports (0 to 5).
2. After mounting your IR sensor, check value when sensor is over black on Mat A, B or black tape



My black ***threshold*** value is ~1600

Find the Black Line



Pseudocode (Task Analysis)

```
1. Prints looking for black line
2. Check the sensor value in analog port 0,
   <= 1600
3. Drive forward as long as the value is <=
   1600
4. Exit loop when value is 1600 or greater
5. Shut everything off
```


#Botball

while "find black line" Solution

```
#include <kipr/botball.h>

int main ()
{
   printf("Find the black line\n");
   while (analog(0) < 1600)
   {
      motor(0,78);
      motor(3,74);    // why slightly less?
   }

  ao();
  return 0;
}
```


Professional Development Workshop

Connections to the Game Board

<u>Description</u>: Navigate to and manipulate game pieces utilizing sensors and motor position counter.

Goal #1: Mat A – Place a stack of two, 3” blocks on circle 4, 6, ***<u>or</u>*** 9. Starting in the start box, drive forward until the cube is sensed and then stop within 3” without touching it. *Bonus: Adding to the previous program, once the cube is sensed, pick it up and navigate back to the start box.*

Goal #2: Mat A – Set a 3” block on coordinates A12. Driving using motor position counter, pick up the 3” block and set it in the yellow garage. Robot or game pieces may not cross solid lines of targeted garage. *Bonus*: *Set 3” blocks on A6, A12, and A18. One by one pick them up, and then stack them in the yellow garage.*

Starting Programs with a Light

- The **light sensor** is a cool way to *automatically* start your robot and *critical* for Botball robots at the beginning of the game.

- The **`wait_for_light()`** function allows your program to run after your robot senses a light.
  - **Note:** It has a built-in calibration routine that will come up on the screen (a step-by-step guide for this calibration routine is on a following slide).

- The light sensor senses *infrared light*, so light must be emitted from an *incandescent light*, not an *LED light*.
  - For our activities, you can use a flashlight.

- The ***more*** light (infrared) detected, the ***lower*** the reported value.

wait_for_light Function

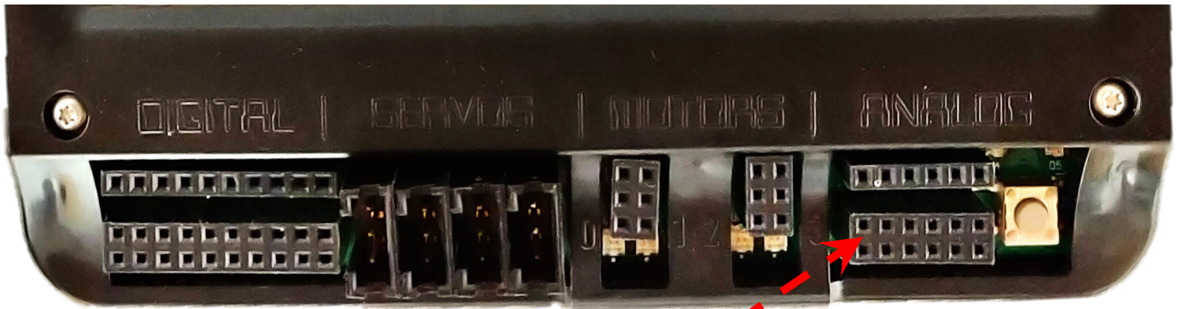
```
wait_for_light(0);
// Waits for the light on port #0 before going to the next line.
```

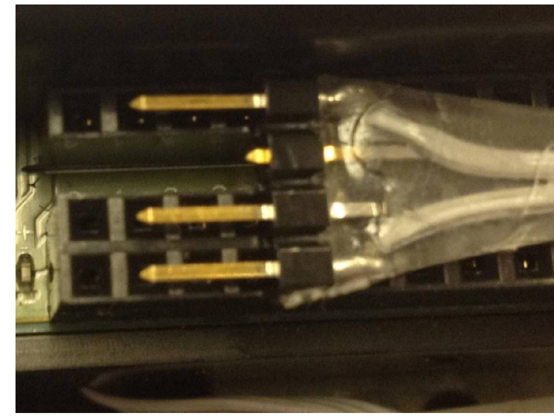
What is this?

```
int main()
{
  wait_for_light(0);
  printf("I see the light!\n");
  return 0;
}
```

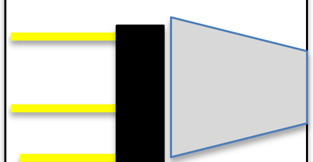
Plug in the Light Sensor

(Light source needed, cell phone works)





Sensor plug orientation

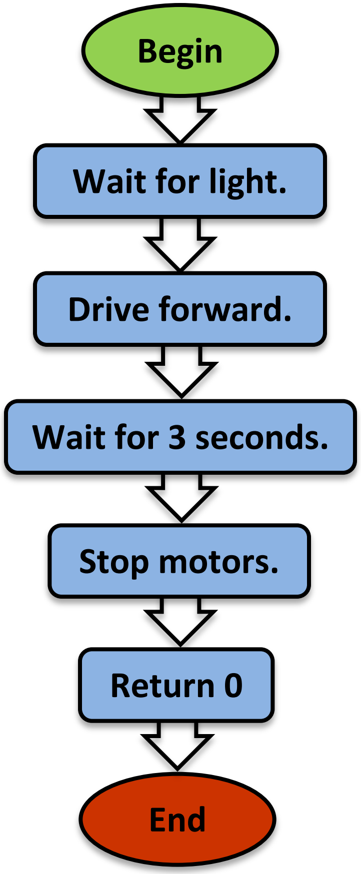


Plug your light sensor into analog port 0

Starting with a Light

Description: Write a program for the KIPR Wombat that waits for a light to come on, drives the DemoBot forward for 3 seconds, and then stops.

Flowchart

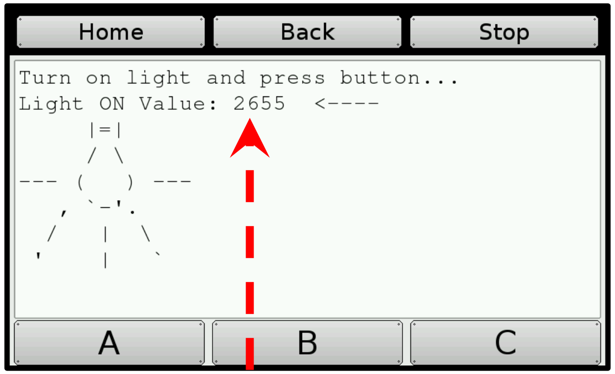


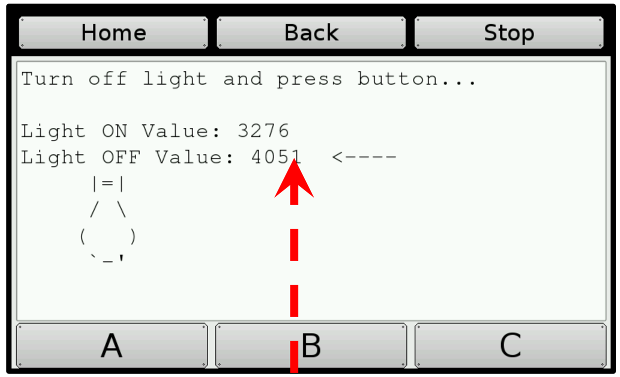
Analysis: What is the program supposed to do?

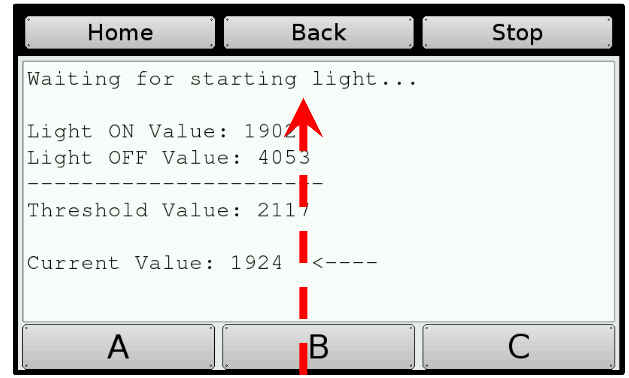
Pseudocode	Comments
1. Wait for light.	`// 1. Wait for light.`
2. Drive forward.	`// 2. Drive forward.`
3. Wait for 3 seconds.	`// 3. Wait for 3 seconds.`
4. Stop motors.	`// 4. Stop motors.`
5. End the program.	`// 5. End the program.`

`wait_for_light` Calibration Routine

When you use the **`wait_for_light()`** function in your program, the following calibration routine will run automatically.







When the light is *on* (low value), press the "push" button.

When the light is *off* (high value), press the "push" button.

You will get a "Waiting for starting light" when done *correctly*. You will get a "BAD CALIBRATION" message when not done correctly, and you will need to push the "push" button to run through the routine again.

"push" button

Note: For Botball, **`wait_for_light()`** should be one of the first functions called in your program.

#Botball®

Starting with a Light

Solution:

Pseudocode

```
1. Wait for light.
2. Drive forward.
3. Wait for 3 seconds.
4. Stop motors.
5. End the program.
```

Source Code

```
int main()
{
  wait_for_light(0);

  motor(0, 100); //forward
  motor(3, 100);
  msleep(3000);
  ao();

  return 0;
}
```

Execution: Compile and run your program (test it with a light sensor).

Starting with a light

Solution: Use a function!

Source Code

```
void drive_forward();
int main()
{
  wait_for_light(0);

  drive_forward();
  msleep(3000);

  ao();

  return 0;
}

void drive_forward()
{
  motor(0, 100);
  motor(3, 100);
}
```

Pseudocode

```
1. Wait for light.
2. Drive forward.
3. Wait for 3 seconds.
4. Stop motors.
5. End the program.
```

Execution: Compile and run your program.

Starting Programs with a Light

- The **light sensor** is a cool way to *automatically* start your robot and *critical* for Botball robots at the beginning of the game.
- The `wait_for_light()` function allows your program to run after your robot senses a light.
  - **Note:** It has a built-in calibration routine that will come up on the screen (a step-by-step guide for this calibration routine is on a following slide).
- The light sensor senses *infrared light*, so light must be emitted from an *incandescent light*, not an *LED light*.
  - For our activities, you can use a flashlight.

- The ***more*** light (infrared) detected, the ***lower*** the reported value.

wait_for_light Function

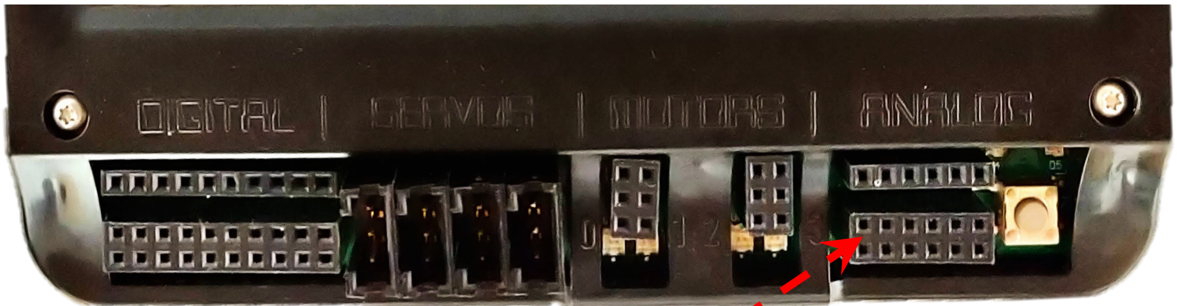
```
wait_for_light(0);
// Waits for the light on port #0 before going to the next line.
```

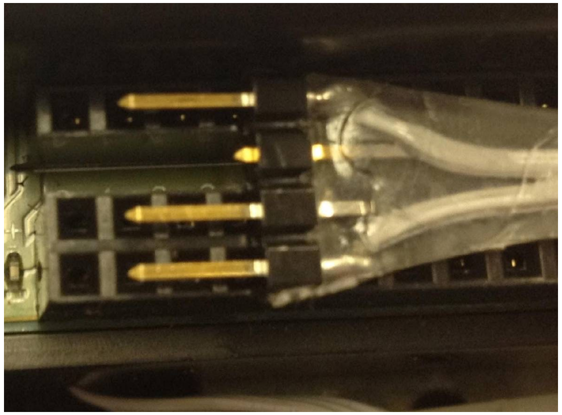
What is this?

```
int main()
{
  wait_for_light(0);
  printf("I see the light!\n");
  return 0;
}
```

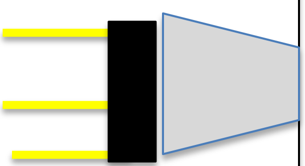
Plug in the Light Sensor

(Light source needed, cell phone works)





Sensor plug orientation

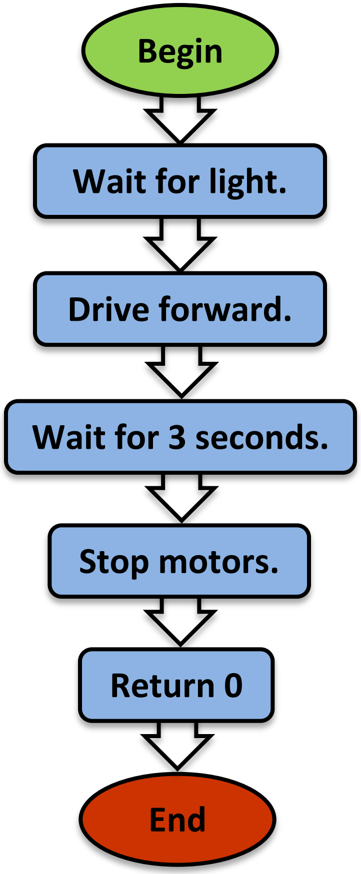


Plug your light sensor into analog port 0

Starting with a Light

Description: Write a program for the KIPR Wombat that waits for a light to come on, drives the DemoBot forward for 3 seconds, and then stops.

Flowchart

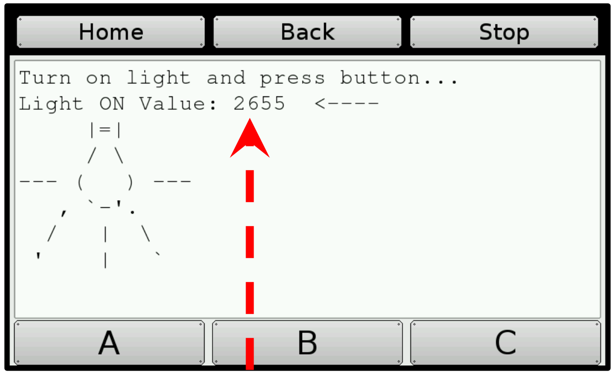


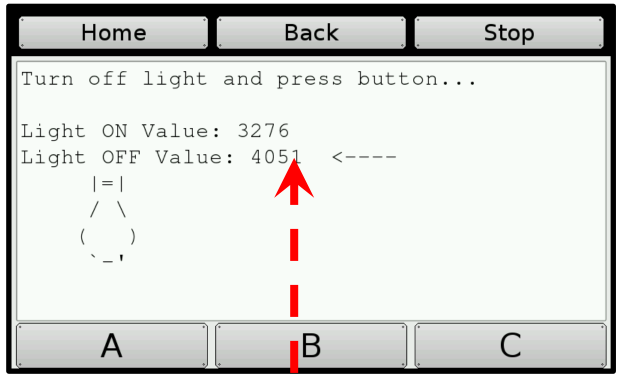
Analysis: What is the program supposed to do?

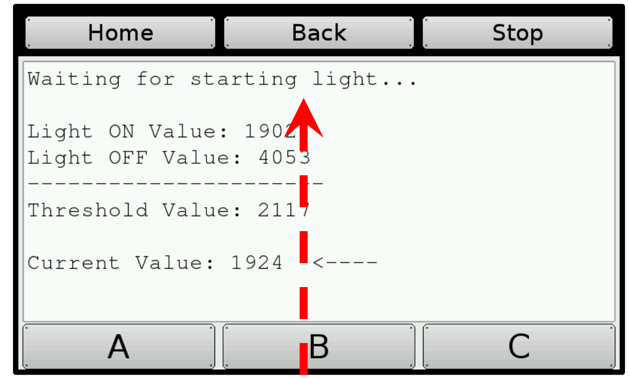
Pseudocode	Comments
1. Wait for light.	`// 1. Wait for light.`
2. Drive forward.	`// 2. Drive forward.`
3. Wait for 3 seconds.	`// 3. Wait for 3 seconds.`
4. Stop motors.	`// 4. Stop motors.`
5. End the program.	`// 5. End the program.`

wait_for_light Calibration Routine

When you use the **`wait_for_light()`** function in your program, the following calibration routine will run automatically.







When the light is *on* (low value), press the "push" button.

When the light is *off* (high value), press the "push" button.

You will get a "Waiting for starting light" when done *correctly*. You will get a "BAD CALIBRATION" message when <u>not</u> done correctly, and you will need to push the "push" button to run through the routine again.

"push" button

Note: For Botball, **`wait_for_light()`** should be one of the first functions called in your program.

#Botball®

Starting with a Light

Solution:

Pseudocode

```
1. Wait for light.
2. Drive forward.
3. Wait for 3 seconds.
4. Stop motors.
5. End the program.
```

Source Code

```
int main()
{
  wait_for_light(0);

  motor(0, 100); //forward
  motor(3, 100);
  msleep(3000);
  ao();

  return 0;
}
```

Execution: Compile and run your program (test it with a light sensor).

Starting with a light

Solution: Use a function!

Pseudocode

```
1. Wait for light.
2. Drive forward.
3. Wait for 3 seconds.
4. Stop motors.
5. End the program.
```

Source Code

```
void drive_forward();
int main()
{
  wait_for_light(0);

  drive_forward();
  msleep(3000);

  ao();

  return 0;
}

void drive_forward()
{
  motor(0, 100);
  motor(3, 100);
}
```

Execution: Compile and run your program.

Making a Choice

Program flow control with conditionals

`if-else` conditionals

`if-else` and Boolean operators

Using `while` and `if-else`

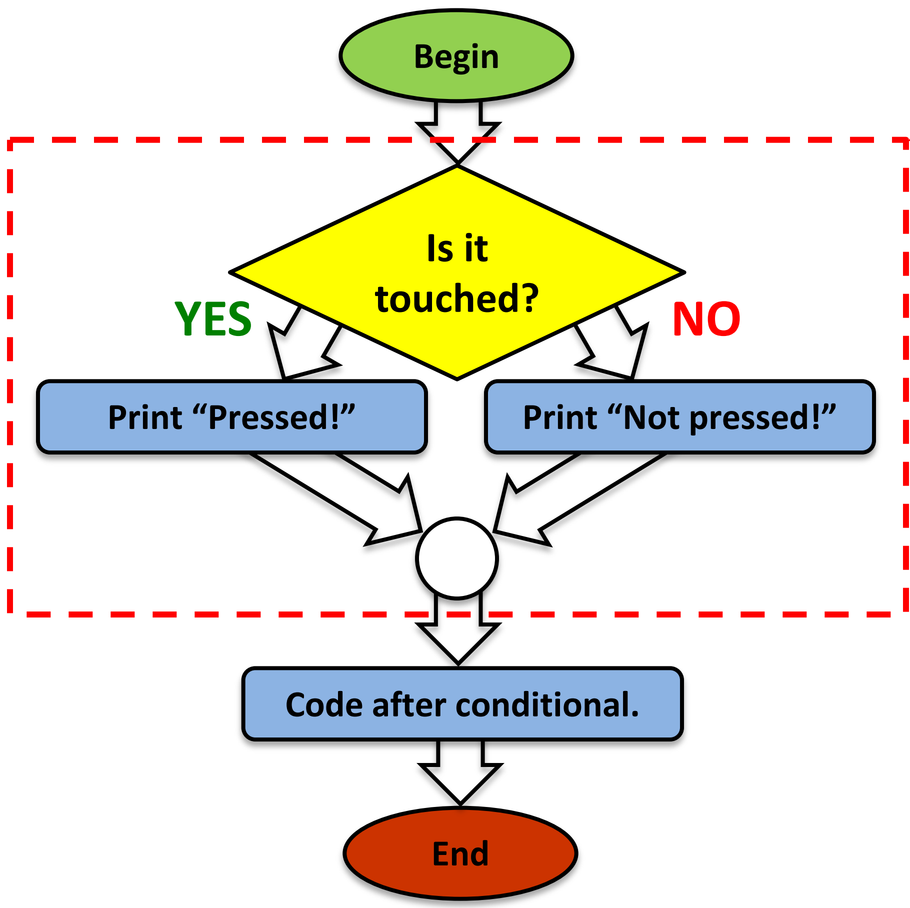
Program Flow Control with Conditionals

- What if we want to execute a **block of code** *only if certain conditions are met*?

- We can do this using a **conditional**, which controls the **flow** of the program by executing ***a certain*** **block of code** if its conditions are met or a ***different*** **block of code** if its conditions are not met.
  - This is similar to a **loop**, but differs in that it **only executes once**.

Program Flow Control with Conditionals

This part of the code is the conditional.



if-else Conditionals

The **if-else** conditional checks to see if a **Boolean test** is **true** or **false**…

- If the **test** is **true**, then the if conditional **executes** the **block of code** that *immediately* follows it.
- If the **test** is **false**, then the if conditional **does not execute** the **block of code**, and the else **block of code** is **executed instead**.

What is this?

What does this say?

```
int main()
{
  if (digital(8) == 1)
  {
    printf("Touched!\n");
  }
  else
  {
    printf("Not touched!\n");
  }

  printf("Good-Bye.\n");
  return 0;
}
```

Notice: *In the same way that a* while *loop doesn't have a semicolon after the condition, neither does an if-else conditional.*

Using while and if-else

You can also put conditionals inside of (nested in) loops. This is beneficial when we want to keep checking a set of conditions over and over, instead of just a single time.

What should go inside the condition for the while loop?

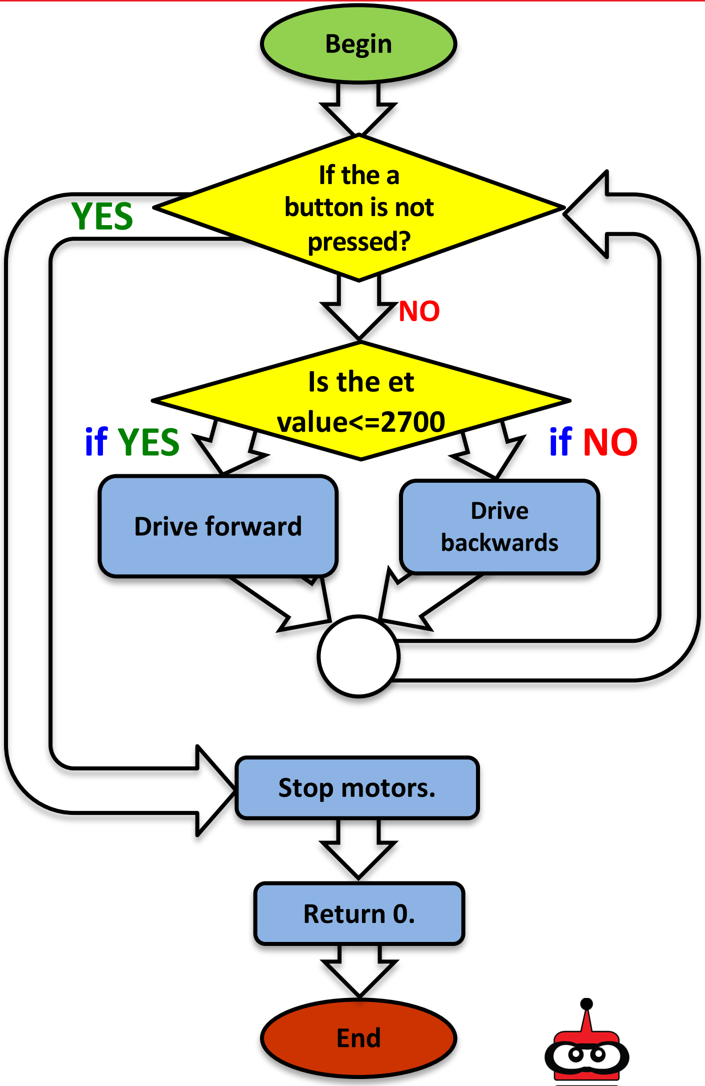
```
int main()
{
  while (digital(0) == 0)
  {
    if (analog(0) > 1600)
    {
      printf("It's dark in here!\n");
    }
    else
    {
      printf("I see the light!\n");
    }
  } // loop ends when button is pressed
  return 0;
}
```

Notice how the { and } braces line up for each block of code!

ET Drive forward to object

Pseudocode (Task Analysis)

1. Check the a button, if it is not pressed
2. Drive forward as long as the value is <=2700 (or your determined value)
3. Drive backwards as long as the value is >2700 (or determined value)
4. Exit loop when a button is pressed
5. Shut everything off



ET Drive forward to object

```
#include <kipr/botball.h>
int main()
{
  printf ("Drive to the object\n");

  while (a_button() == 0)     // A button not pressed
  {
      if (analog(0) <= 2700)   // Far away drive forward
      {
         motor(0,80);
         motor(3,80);
      }
      if (analog(0) > 2700)    // Too close back up
      {
         motor(0,-80);
         motor(3,-80);
      }
  }
  ao();
  return 0;
}
```

Maintain Distance

Description: Write a program for the KIPR Robotics Controller that makes the DemoBot maintain a specified distance away from an object, and stops when the touch sensor is touched.

Pseudocode

1. *Loop:* Is not touched?

    If: Is distance too far?
        Drive forward.
    Else:
        If: Is distance too close?
            Drive reverse.
        Else:
            Stop motors.
2. Stop motors.
3. End the program.

Maintain Distance

Solution:

Pseudocode

1. *Loop:* Is not touched?
    If: Is distance too far?
        Drive forward.
    Else:
        If: Is distance too close?
            Drive reverse.
        Else:
            Stop motors.
2. Stop motors.
3. End the program.

Source Code

```
int main()
{
   while (digital(0) == 0)
  {

    if (analog(5) < 1800)
    {
      motor(0, 80);
      motor(3, 80);
    }
    else
    {
      if (analog(5) > 2600)
      {
        motor(0, -75);
        motor(3, -75);
      }
      else // sensor value is 1800-2600
      {
        ao();
      }
    }
  } // end of loop

  ao();
  return 0;
}
```

Reflectance Sensor for Line Following

For this activity, you will need a **reflectance sensor**.

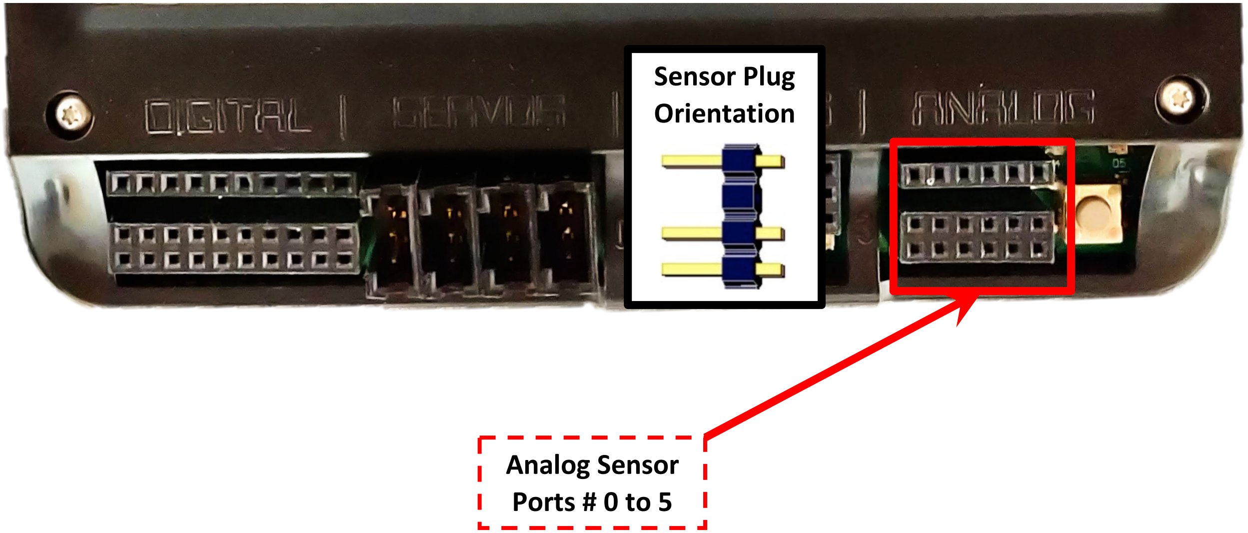
- This sensor is a short-range reflectance sensor.
- There is both an infrared (IR) *emitter* and an IR *detector* inside of this sensor.
- IR *emitter* sends out IR light → IR *detector* measures how much reflects back.
- The amount of IR reflected back depends on many factors, including **surface texture**, **color**, and **distance to surface**.

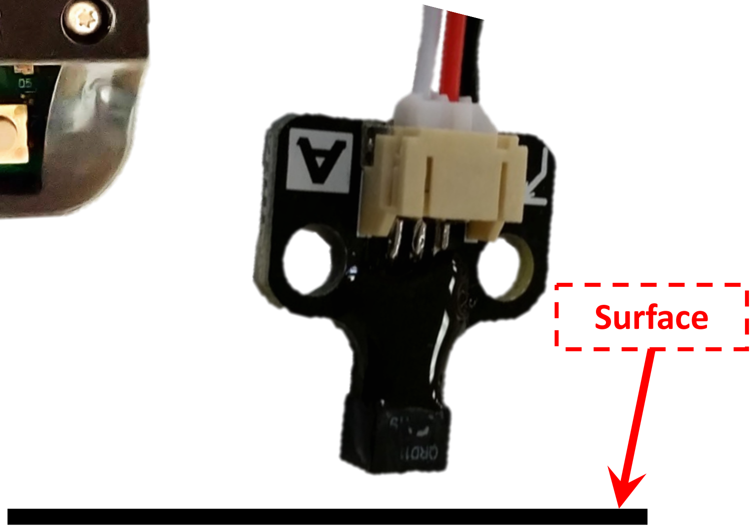
This sensor is **excellent** for line-following!

- **Black materials** typically **absorb <u>most</u> IR** → they **reflect <u>little</u> IR back**!
- **White materials** typically **absorb <u>little</u> IR** → they **reflect <u>most</u> IR back**!
- If this sensor is mounted at a *fixed height* above a surface, it is easy to distinguish a black line from a white surface.

Attach the Reflectance Sensor

- Attach the sensor on the front of your robot so that it is **pointing down at the ground** and is **approximately 1/8" from the surface**.
- A **reflectance sensor** is an **analog sensor**, so plug it into any of **analog sensor port #0 through 5**. Port 0 for this example.
  - Recall that analog sensor values range **from 0 to 4095**.



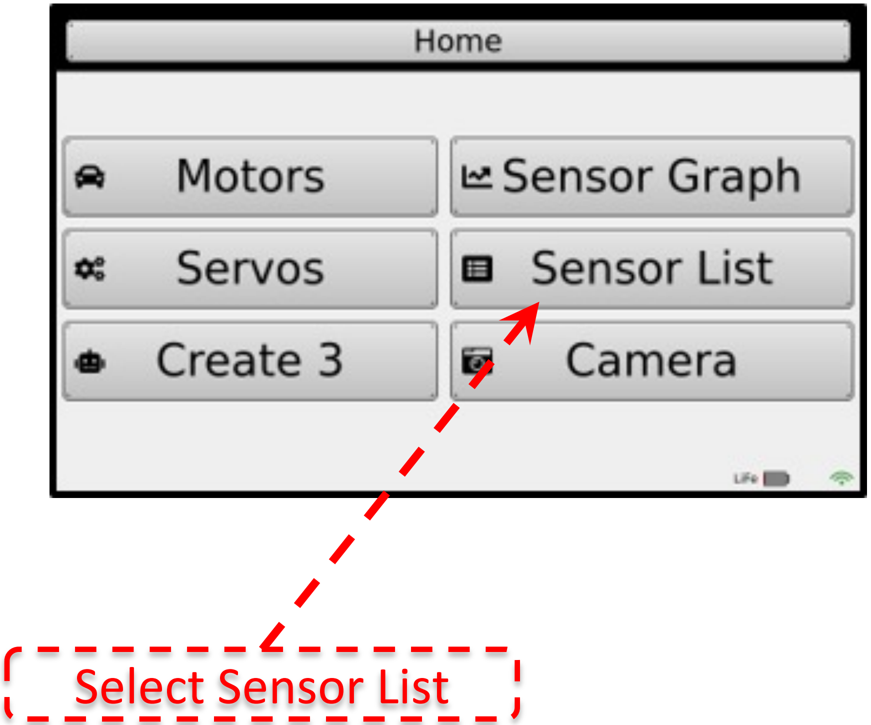


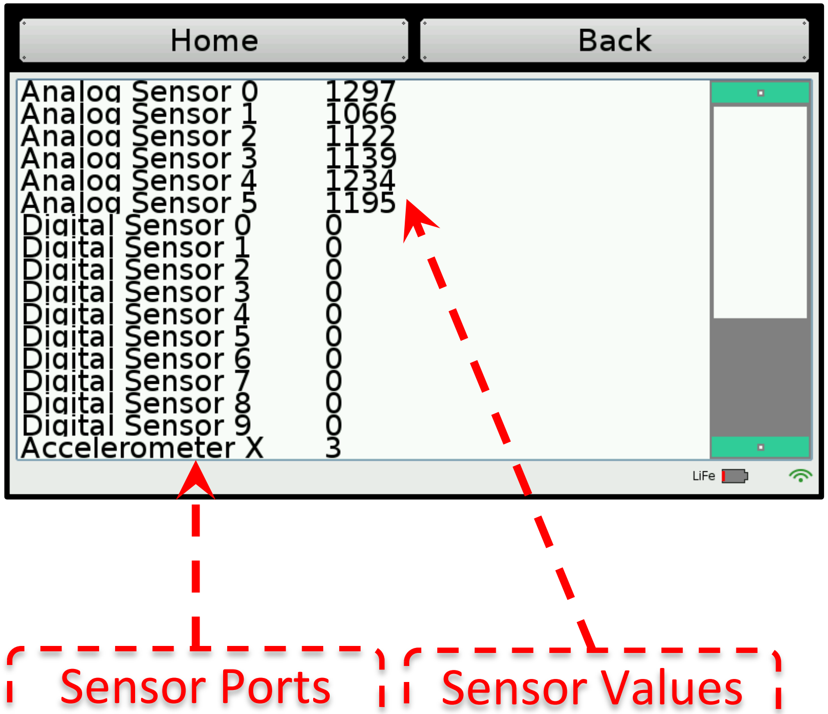
#Botball

Reading Sensor Values from the Sensor List

You can access the Sensor Values from the Sensor List on your Wombat

- This is very helpful to get readings from all of the sensors you are using, and then know which values/ranges to use in your code



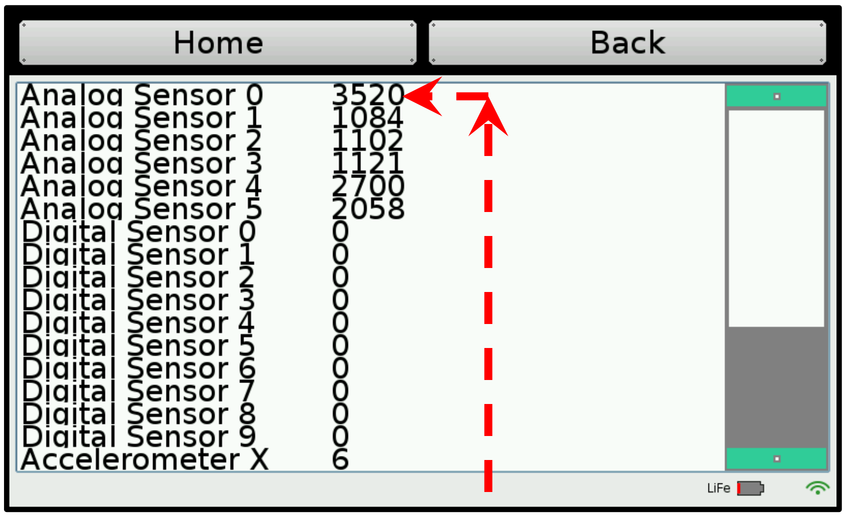


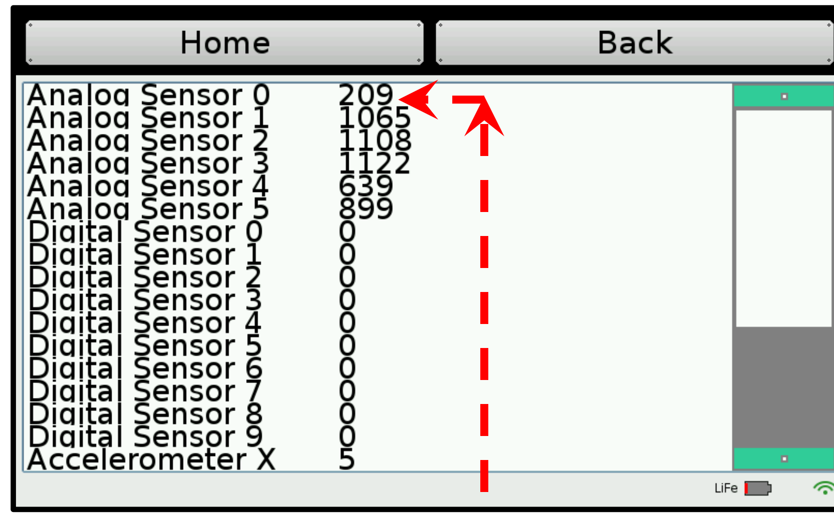
Reading Sensor Values from the Sensor List

With the IR sensor plugged into analog port #0

- Over a white surface the value is (~200)
- Over a black surface the value is (~3000)

Your *values* will be different, but the *process* will be the same!





Values between ~2900-~3800 over the Black Surface

Values between ~175-~300 over the White Surface.

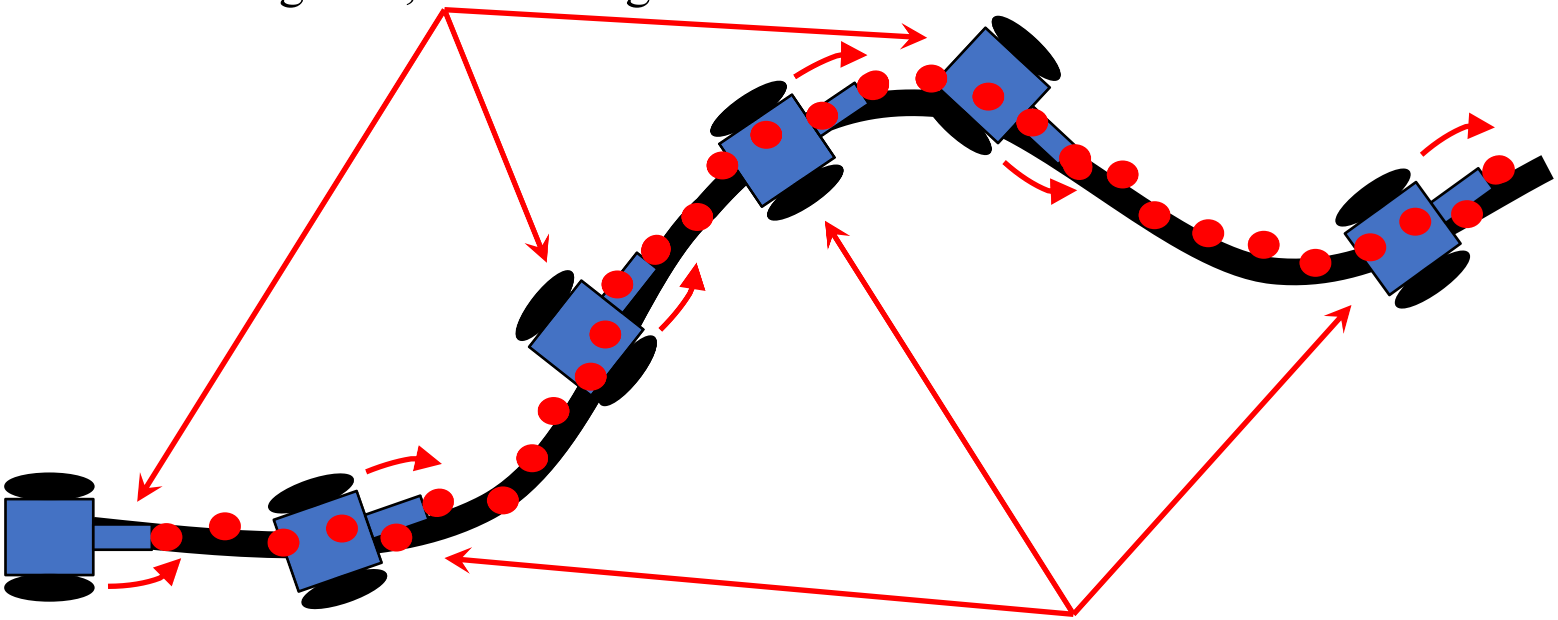
#Botball®

Line Following Strategy Using the Reflectance Sensor

Line Following Strategy: **while** - Is the button pushed?

Follow the line's right edge by alternating the following 2 actions:

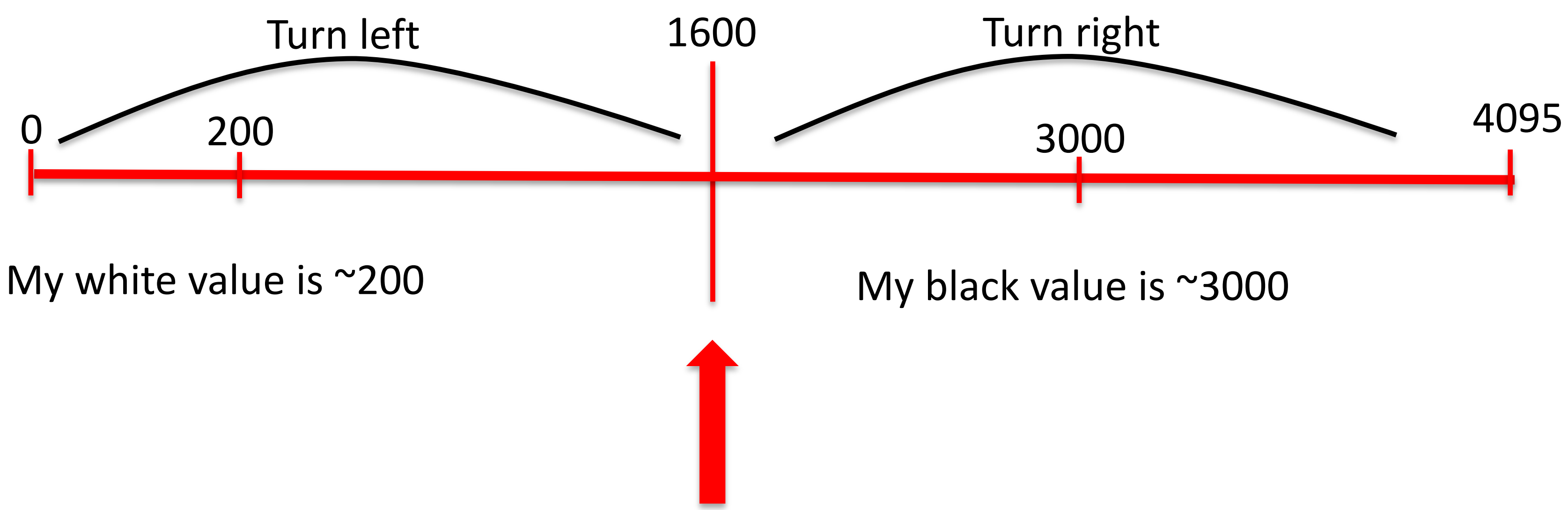
1. **if** detecting dark, arc/turn right



2. **if** detecting light, arc left.
3. Think about a sharp turn. What will your motor function look like? Remember the bigger the difference between the two motor powers the sharper the turn.

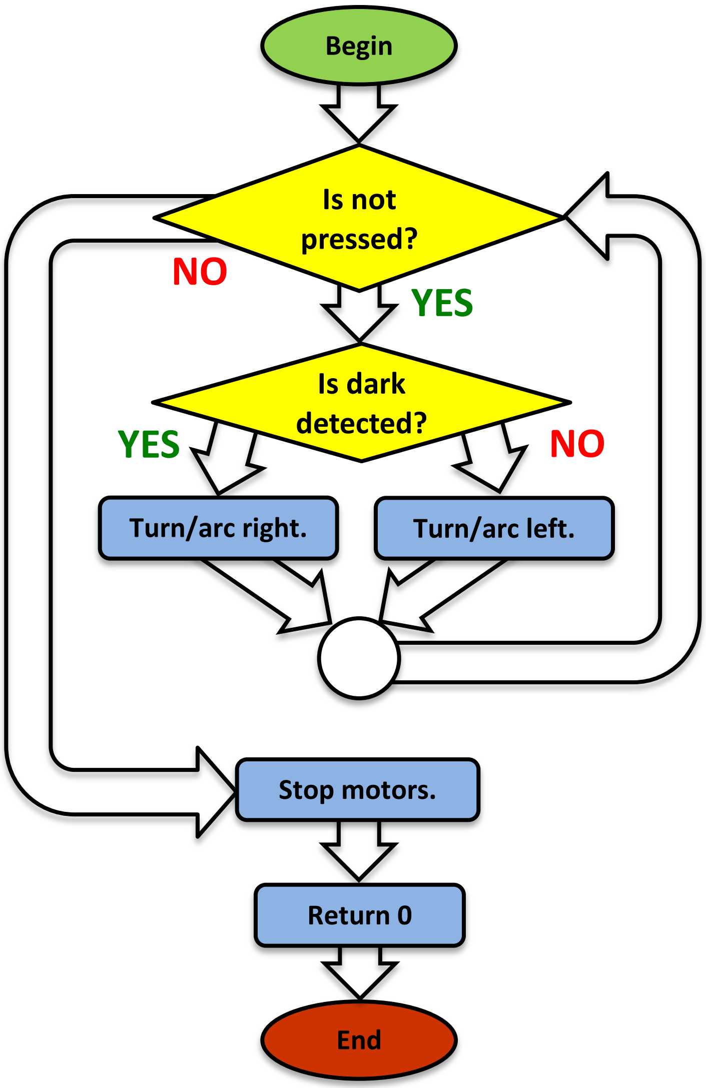
Understanding the IR Values

1. Place your IR analog sensor in one of the analog ports (0 to 5).
2. After mounting your IR sensor, check that the values are: white between 175-225 and black between 2900-3100; write down your values.
3. Find your threshold or ***middle*** value (approximately)
4. This number will be the value you need for the find the black line activity.

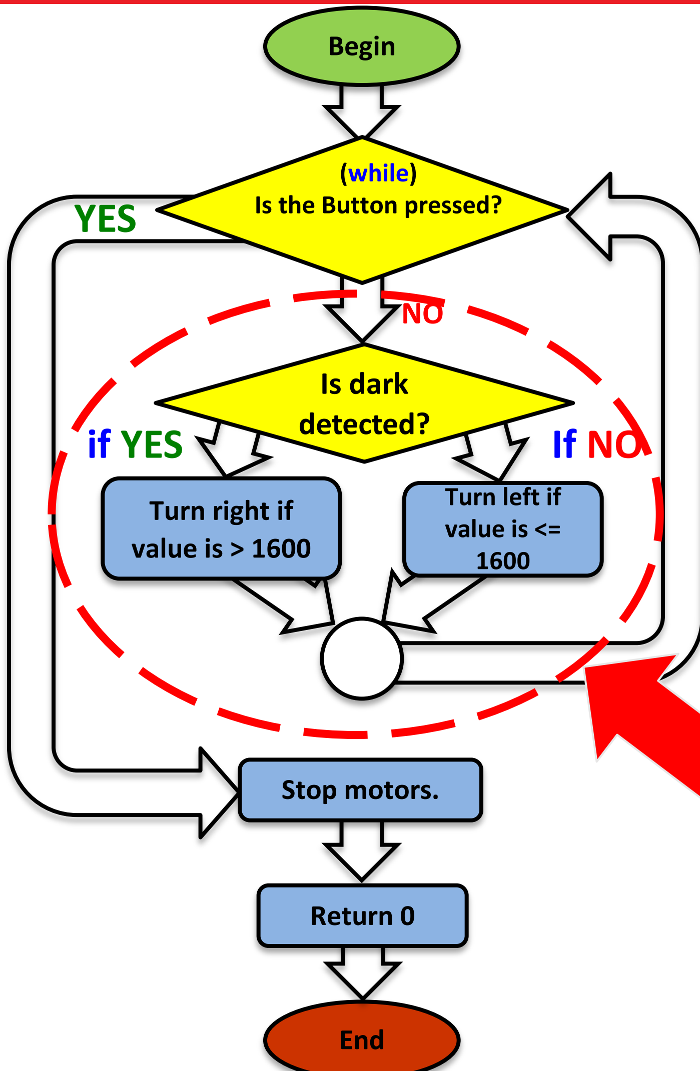


Determine what your threshold or “half way” point will be.
This example is ~1600.

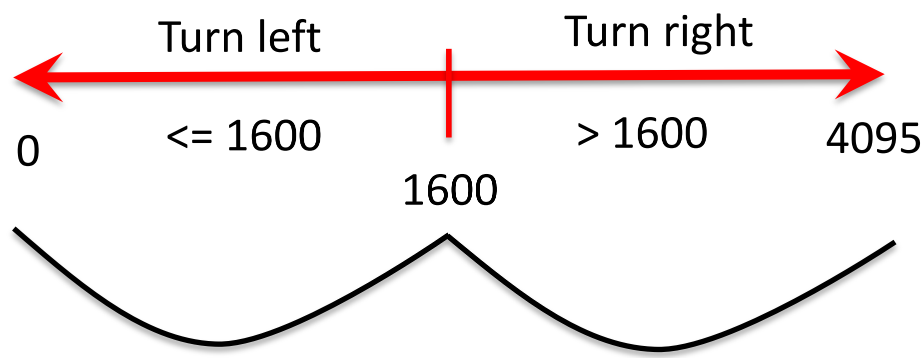
Analysis: Flowchart



Understanding while and if



You must cover all values



Assume all these values are WHITE

Assume all these values are BLACK

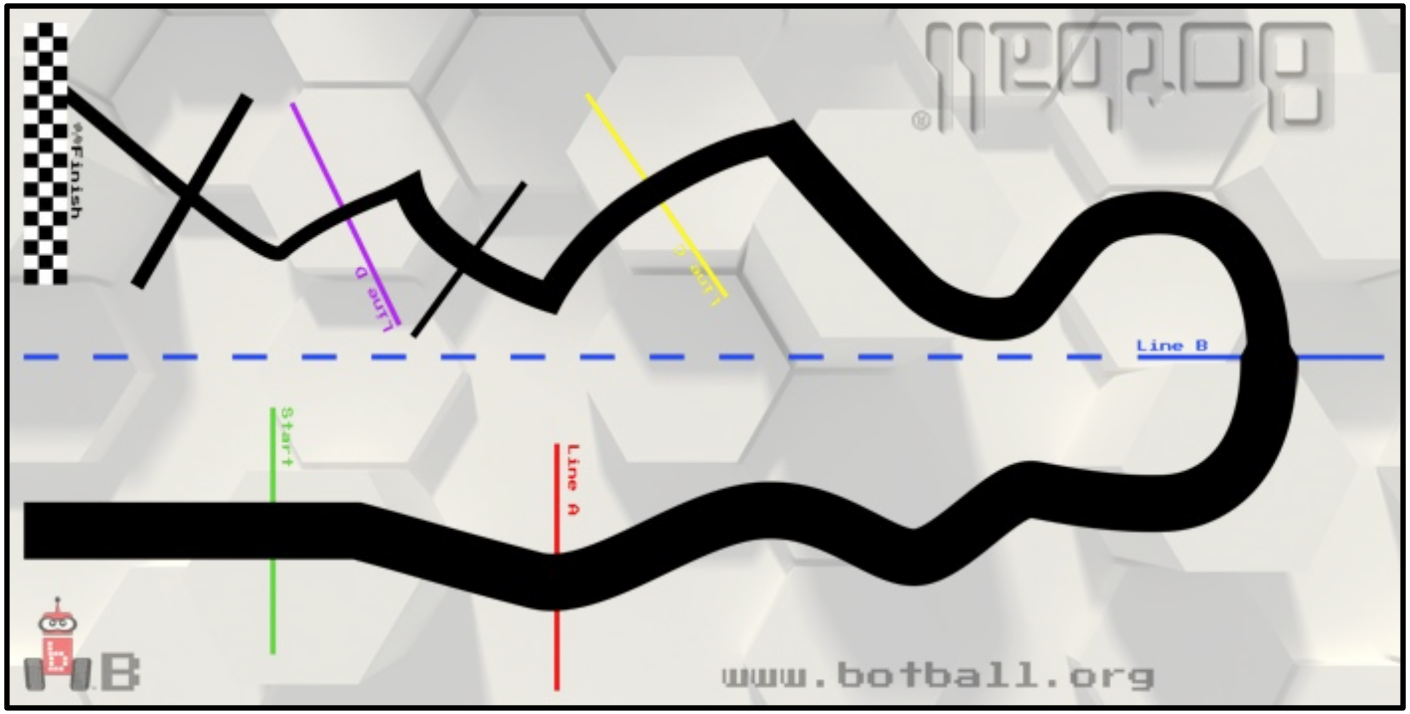
This is the part of the code that tells the Robot what to do when it sees black or white.

Professional Development Workshop

#Botball

Line Following

Description: Starting with your DemoBot at the starting line of the KIPR Mat B. Write a program to have the robot travel along the path using the Top Hat sensor (line follow).



Line Following Solution

Solution:

Pseudocode (Comments)

1. *Loop:* Is not pressed?
   If: Is dark detected?
      Turn/arc right.
   Else:
      Turn/arc left.
2. Stop motors.
3. End the program.

Source Code

```
int main()
{
  while (digital(0) == 0)
  {
    if (analog(0) > 1600)
    {
      motor(0, 90);
      motor(3, 5);
    }

    else
    {
      motor(0, 5);
      motor(3, 90);
    }
  }

  ao();

  return 0;
}
```

Tips

Change the threshold. Increase the "arc speed".

```
int main()
{
    printf("Follow the line\n");
    while (digital(0) == 0)
    {
        if (analog(0) > 1600)
        {
            motor(0, 90);
            motor(3, 5);
        }
        else
        {
            motor(0, 5);
            motor(3, 90);
        }
    }

    ao();
    return 0;
}
```

The value of 1600 or the "threshold" value is ½ way between the observed values. Remember black reflects less IR than white but the value is higher.

Notice the Boolean operators > 1600 or <= 1600

Your value may be much lower due to lighting, placement of sensor and other factors.

Also increasing the "arc speed" (by making the ***difference*** between the two motor power values <u>greater)</u> may have a significant impact.

#Botball

Line following with Functions

Solution: (using two functions)

Pseudocode

```
1. Loop: Is not pressed?
     If: Is dark detected?
        Turn/arc right.
     Else:
        Turn/arc left.
  2. Stop motors.
  3. End the program.
```

Source Code

```
void turn_left();
void turn_right();

int main()
{
  while (digital(0) == 0)
  {
    if (analog(0) > 1600)
    {
      turn_right();
    }
    else
    {
      turn_left();
    }
  }
  ao();
  return 0;
}

void turn_right()
{
  motor(0, 80);
  motor(3, 10); // Turn/arc right.
}

void turn_left()
{
  motor(0, 10);
  motor(3, 80); // Turn/arc left.
}
```

Logical Operators

Multiple Boolean Tests

`while`, `if`, and Logical Operators

Logical Operators

Recall the **Boolean test** for `while` loops and `if-else` conditionals…

```
while (Boolean test)                    if (Boolean test)
```

- The **Boolean test** (conditional) can contain ***multiple*** **Boolean tests** combined using a "**Logical operator**", such as:
  - `&&` And
  - `||` Or
  - `!` Not

We put parentheses (and) around *each* Boolean test…

```
while ((Boolean test 1) && (Boolean test 2))

   if ((Boolean test 1) || (!Boolean test 2))
```

- The next slide provides a cheat sheet for **Logical operators**.

#Botball®

Logical Operators Cheat Sheet

Boolean	English Question	True Example	False Example

`A && B`	Are **both** A **and** B true?	`true && true`	`true && false`
			`false && true`
			`false && false`

`A || B`	Is **at least one** of A **or** B true?	`true || true`
		`false || true`
		`true || false`
			`false || false`

`!(A && B)`	Is **at least one** of A **or** B false?	`true && false`
		`false && true`
		`false && false`
			`true && true`

`!(A || B)`	Are **both** of A **and** B false?	`false || false`	`true || true`
			`false || true`
			`true || false`

! negates the **`true`** or **`false`** Boolean test.

#Botball®

while, if, and Logical Operators Examples

```
while ((get_create_lbump() == 0) && (get_create_rbump() == 0))
{
  // Code to execute ...
}
```

```
while ((digital(9) == 0) && (digital(8) == 0))
{
  // Code to repeat ...
}
```

```
if ((digital(9) == 1) || (digital(8) != 0))
{
  // Code to execute ...
}
```

```
if ((analog(3) < 512) || (digital(9) == 1))
{
  // Code to repeat ...
}
```

Using Logical Operators

What does this say?

```
int main()
{
  create_connect();
  while ((get_create_lbump() == 0) && (get_create_rbump() == 0))
  {
    create_drive_direct(100, 100);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```

Botball Game Review

Game Q&A

Construction, documentation, and changes

`shut_down_in()` function

`wait_for_light()` function

Remember Loops?

- How does the `wait_for_light()` function work?
- We can use a **loop**, which controls the **flow** of the program by repeating a **block of code** until a sensor reaches a particular value.
  - The number of repetitions is unknown
  - The number of repetitions depends on the conditions sensed by the robot

Botball Tournament Functions

These two functions should be two of the first lines of code in your Botball tournament program!

```
wait_for_light(0);
// Waits for the light on port #0 before going to the next line.

shut_down_in(119);
// Shuts down all motors after 119 seconds (just less than 2 minutes).
```

- **This function call should come immediately after the `wait_for_light()`** in your code.
- If you do not have this function in your code, your robot may not automatically turn off its motors at the end of the Botball round and **you will be disqualified**!

#Botball®

Tournament Templates

```
int main()
{
  // initial variable declarations, camera and servo setup may go here

  wait_for_light(0); // change the port number to match the port you use
  shut_down_in(119); // shut off the motors and stop the robot after 119 seconds

  // This is where most of your code will go

  // Specifically the code to play the game
  // after the light comes on (after hands off)

  return 0;
}
```

Running a Botball Tournament Program

Description: Write a program for the KIPR Robotics Controller that waits for a light to come on, shuts down the program in 5 seconds, drives the DemoBot forward until it detects a touch, and then stops.

Analysis: What is the program supposed to do?

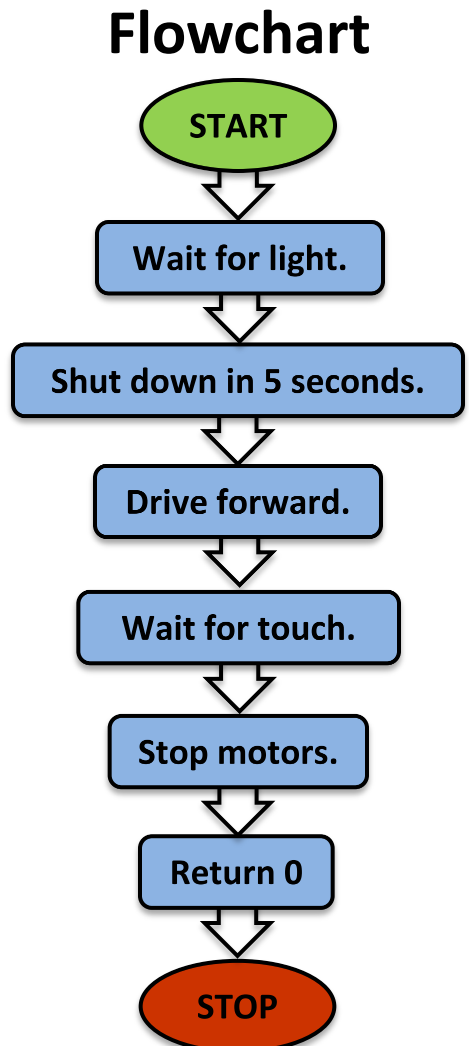
Pseudocode

1. Wait for light.
2. Shut down in 5 seconds.
3. Drive forward.
4. Wait for touch.
5. Stop motors.
6. End the program.

Comments

```
// 1. Wait for light.
// 2. Shut down in 5 seconds.
// 3. Drive forward.
// 4. Wait for touch.
// 5. Stop motors.
// 6. End the program.
```

Analysis:


Flowchart
START
Wait for light.
Shut down in 5 seconds.
Drive forward.
Wait for touch.
Stop motors.
Return 0
STOP

Running a Botball Tournament Program

Solution:

Pseudocode

```
1. Wait for light.
2. Shut down in 5 seconds.
3. Drive forward.
4. Wait for touch.
5. Stop motors.
6. End the program.
```

Source Code

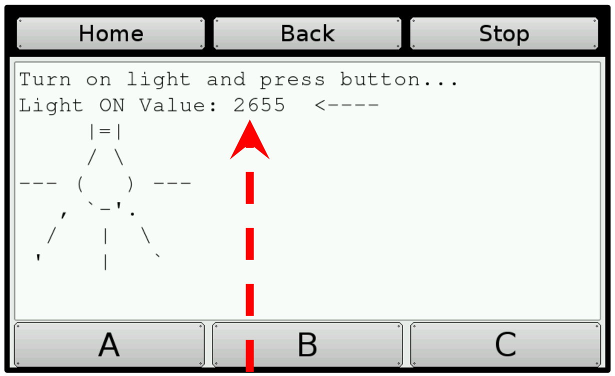
```
int main()
{
  wait_for_light(0);

  shut_down_in(5);

  while (digital(0) == 0)
  {
    motor(0, 100);
    motor(3, 100);
  }
  ao();

  return 0;
}
```

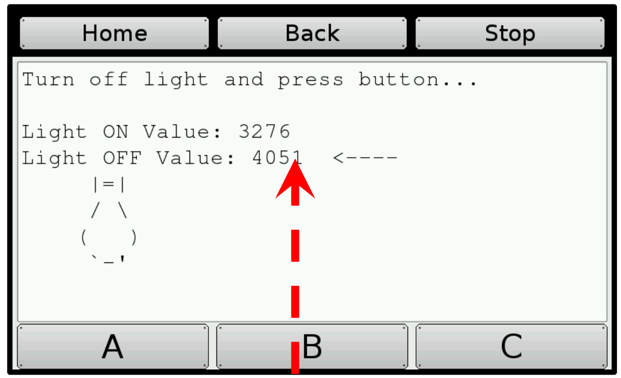
Execution: Compile and run your program (test it at different distances).

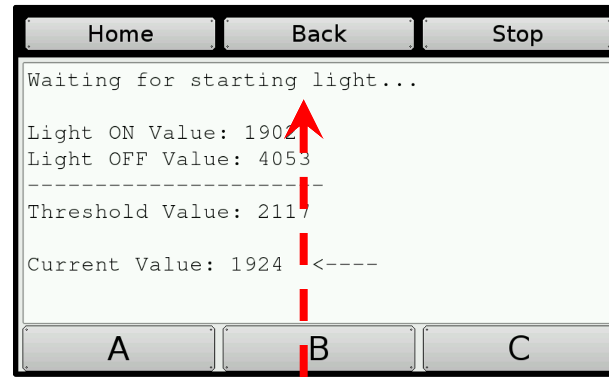
#Botball®

`wait_for_light` Calibration Routine

When you use the `wait_for_light()` function in your program, the following calibration routine will run automatically.







When the light is *on* (low value), press the "push" button.

When the light is *off* (high value), press the "push" button.

You will get a "Waiting for starting light" when done *correctly*. You will get a "BAD CALIBRATION" message when done <u>not</u> correctly, and you will need to push the "push" button to run through the routine again.

"push" button

Note: For Botball, `wait_for_light()` should be one of the first functions called in your program.

#Botball®

Running a Botball Tournament Program

Reflection:

- What happens if the touch sensor is pressed in ***less than 5 seconds*** after starting the program?
- What happens if the touch sensor is **not** pressed in ***less than 5 seconds*** after starting the program?
- What is the best way to guarantee that your program will ***start with the light*** in a Botball tournament round? (**Answer:** `wait_for_light(0)`)
- What is the best way to guarantee that your program will ***stop within 120 seconds*** in a Botball tournament round? (**Answer:** `shut_down_in(119)`)

Use these functions in your Botball tournament code!

#Botball®

Custom Functions (Quick Recap)

When you call this function, how long will it run for?

```
void drive_forward();  // function prototype


int main()
{
  drive_forward();      // function call
  return 0;
}


void drive_forward()   // function definition
{
  motor(0, 80);
  motor(3, 80);
  msleep(4000);
  ao();
}
```

Now, what if you don't want it to run for this long each time?

#Botball®

Functions with Arguments

- **Function arguments:** values you will set when you call the function

```
void drive_forward(int milliseconds);   // function prototype

int main()
{
  drive_forward(4000);                   // function call
  return 0;
} // end main

void drive_forward(int milliseconds)    // function definition
{
  motor(0, 80);
  motor(2, 80);
  msleep(milliseconds);
  ao();
}
```

Writing Custom Functions with Arguments

```
#include <kipr/botball.h>

void drive_forward(int milliseconds);   // function prototype


int main()
{
  drive_forward(4000);   // function call
  return 0;
}


void drive_forward(int milliseconds)   // function definition
{
  motor(0, 80);
  motor(3, 80);
  msleep(milliseconds);
  ao();
}
```

The value in the function call *__sets__* the value of the argument…

… which is then used in the function definition.

Writing your Own Functions with Multiple Arguments

```
#include <kipr/botball.h>

void drive_forward(int power, int milliseconds);  // function prototype

int main()
{
  drive_forward(80, 4000);  // function call
  return 0;
}


void drive_forward(int power, int milliseconds)  // function definition
{
  motor(0, power);
  motor(3, power);
  msleep(milliseconds);
  ao();
}
```

The value in the function call *sets* the value of the argument…

… which is then used in the function definition.

#Botball®

Arguments that Change Over Time

```
#include <kipr/botball.h>

void drive_forward(int power, int milliseconds);   // function prototype

int main()
{
  drive_forward(80, 4000);
  drive_forward(75, 2000);
  return 0;
}


void drive_forward(int power, int milliseconds)  // function definition
{
  motor(0, power);
  motor(3, power);
  msleep(milliseconds);
  ao();
}
```

The values in the **SECOND function call** ***are now 75 and 2000*** **respectively**

… which is then used in the **function definition.**

Writing Your Own Driving with Gyro Function

- The built in Wombat Gyrometer can be used to help your robot drive straight
- To do this you need to account for the bias (how far off the gyro is from predicted-similar to magnetic declination)
- Then you can use the bias in a bang-bang code to drive your robot

```
#include <kipr/wombat.h>

double calibrate_gyro();
void drive_with_gyro(int speed, double time, double bias);

int main()
{
    printf("Hello World\n");
    double bias = calibrate_gyro();
    drive_with_gyro (1500,15,bias);
    return 0;
}
```

The main has two custom functions:
calibrate_gyro();
drive_with_gyro(speed,time, bias)

mav ticks/sec

seconds

The calibrate function returns a bias that is then used as the 3rd parameter in the drive with gyro function

Calibrating Your Gyro Function

```
double calibrate_gyro()
{
    int i = 0;
    double bias = 0.2;
    int avg = 0;
    while( i < 50){
        avg += gyro_z();
        msleep(1);
        i++;
        printf("Gyro Z: %d\n",gyro_z());
    }
    bias = avg / 50.0;
    printf("New Bias: %lf\n", bias);
    return bias;
}
```

The calibrate function returns a bias that is then used as the 3rd parameter in the drive with gyro function (next slide)

#Botball®

Calibrating Your Gyro Function

```
void drive_with_gyro(int speed, double time, double bias)
{
    double startTime = seconds();
    double theta = 0;
    int right_motor = 3;
    int left_motor = 0;
    while(seconds() - startTime < time)
    {
        if (theta < 1000 && theta > -1000)
        {
            mav(right_motor, speed);
            mav(left_motor, speed);
        }
        else if (theta < 1000)
        {
            mav(right_motor, speed + 100);
            mav(left_motor, speed - 100);
        }
        else
        {
            mav(right_motor, speed - 100);
            mav(left_motor, speed + 100);
        }
        msleep(10);
        theta += (gyro_z() - bias) ;
        printf("%f\n",theta);
    }
    ao();
}
```

The bias came from your other custom function

Calibrating Your Gyro Function

```c
#include <kipr/wombat.h>

double calibrate_gyro();
void drive_with_gyro(int speed, double time, double bias);

int main()
{
    printf("Hello World\n");
    double bias = calibrate_gyro();
    drive_with_gyro (1500,15,bias);
    return 0;
}

double calibrate_gyro()
{
    int i = 0;
    double bias = 0.2;
    int avg = 0;
    while( i < 50){
        avg += gyro_z();
        msleep(1);
        i++;
        printf("Gyro Z: %d\n",gyro_z());
    }
    bias = avg / 50.0;
    printf("New Bias: %lf\n", bias);
    return bias;
}

void drive_with_gyro(int speed, double time, double bias)
{
    double startTime = seconds();
    double theta = 0;
    int right_motor = 3;
    int left_motor = 0;
    while(seconds() - startTime < time)
    {
        if (theta < 1000 && theta > -1000)
        {
            mav(right_motor, speed);
            mav(left_motor, speed);
        }
        else if (theta < 1000)
        {
            mav(right_motor, speed + 100);
            mav(left_motor, speed - 100);
        }
        else
        {
            mav(right_motor, speed - 100);
            mav(left_motor, speed + 100);
        }
        msleep(10);
        theta += (gyro_z() - bias) ;
        printf("%f\n",theta);
    }
    ao();
}
```

Moving the iRobot *Create 2*: Part 1

Setting up the *Create 2*

The *Create 2* and the KIPR Robotics Controller

***Create* functions**

Charging the *Create 2*

- For charging the **Create 2**, **use only the power supply which came with your *Create***.
  - **Damage to the *Create* from using the wrong charger is easily detected and will void your warranty!**

- The **Create 2** power pack is a **nickel metal hydride battery**, so the rules for charging a battery for any electronic device apply.
  - Only an adult should charge the unit.
  - **Do <u>NOT</u> leave the unit unattended** while charging.
  - Charge in a cool, open area away from flammable materials.

#Botball®

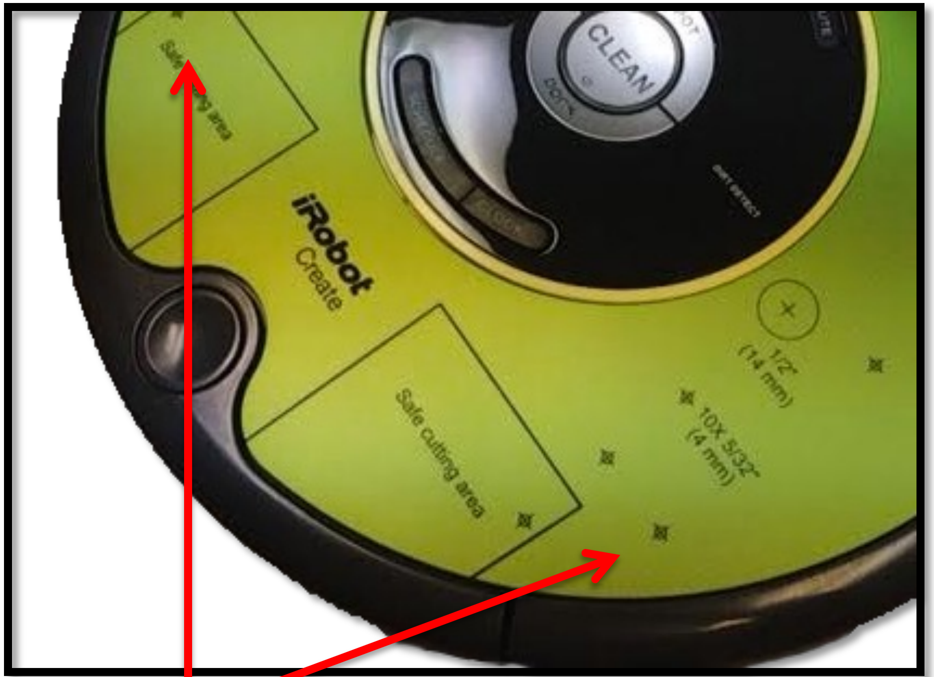
Enabling the Battery of the *Create 2*

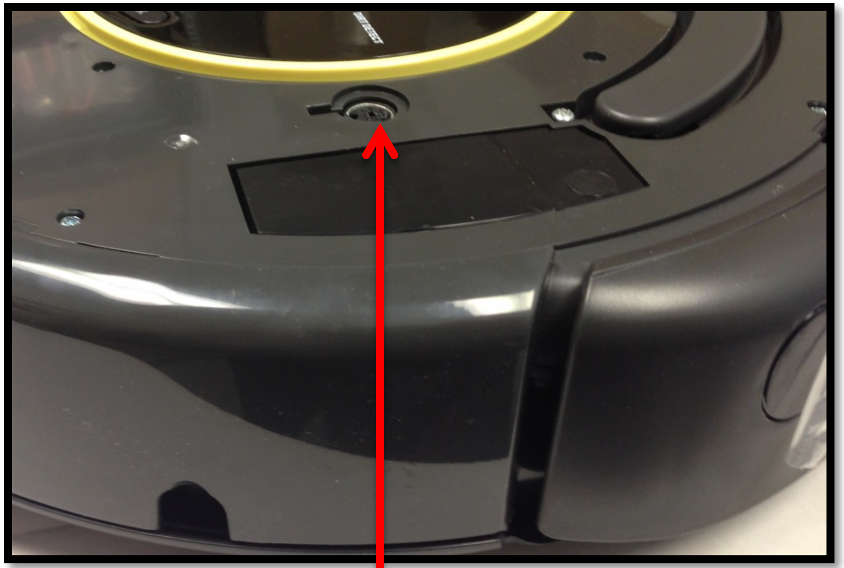
- The **yellow battery** tab pulls out of place on the bottom of the *Create*.
- The battery will be enabled as soon as the tab is removed.

Create Underside

Uncovering and Charging the *Create 2*

- Remove the green protective tray from the top of the **Create**.
- Use only the **Create** charger provided with your kit.
- The **Create** docks onto the charging station.





Remove this

Serial Port

Build the Create 2 DemoBot

Create 2 Connect/Disconnect Functions

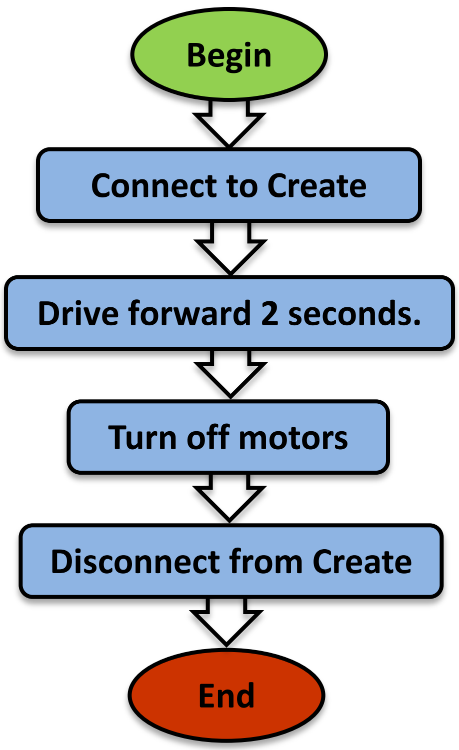
All programs used with the *Create*
MUST *start* with

```
create_connect()
```

and *end* with

```
create_disconnect()
```

Flowchart



Tournament Templates

```
int main() // for your Create robot
{
  create_connect();
  // other initial items as needed (servo and camera calibration for example)

  wait_for_light(0); // change the port number to match the port you use
  shut_down_in(119); // shut off the motors and stop the robot after 119 seconds

  // Your code
  create_disconnect();
  return 0;
}
```

Create 2 Motor Functions

Note: **Create** commands run until a different motor command is received.

```
create_drive_direct(left speed, right speed);
```

Left Motor Speed (in mm/second)

Right Motor Speed (in mm/second)

Examples:

```
create_drive_direct(100, 100);    // Moves forward at 100 mm/sec.

create_drive_direct(-200, 200);  // Create will turn left.

create_drive_direct(150, -150);  // Create will turn right.

create_stop();  // Turns off the Create motors.
```

WARNING: the maximum speed for the *Create* motors is 500 mm/second = 0.5 m/second.
It can jump off a table in *less than one second*!
Use something like 200 for the speed (moderate speed) until teams get the hang of this.

Moving the *Create 2*

Description: Write a program for the KIPR Robotics controller that drives the **Create** forward at 100 mm/second for four seconds, and then stops.

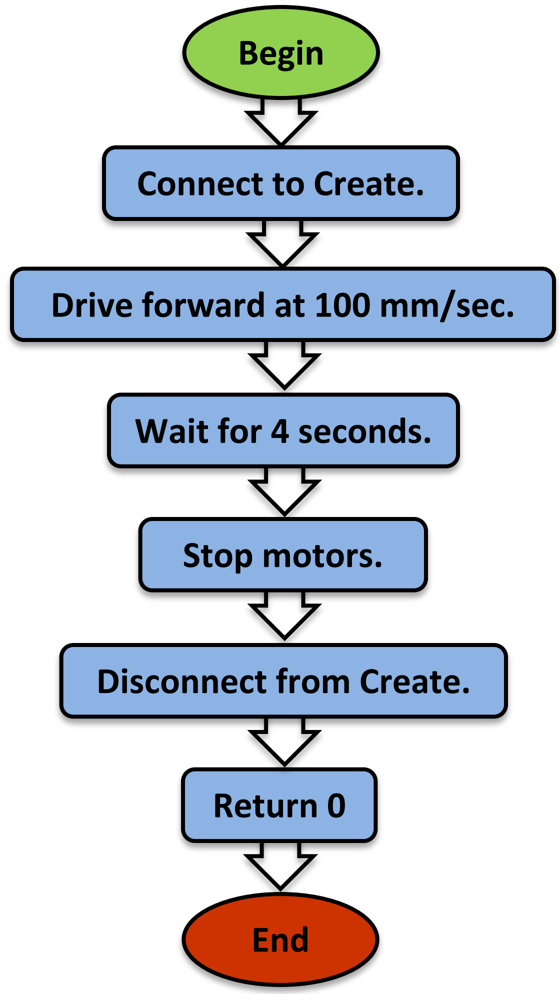
Analysis: What is the program supposed to do?

Pseudocode	**Comments**
1. Connect to Create.	`// 1. Connect to Create.`
2. Drive forward at 100 mm/sec.	`// 2. Drive forward at 100 mm/sec.`
3. Wait for 4 seconds.	`// 3. Wait for 4 seconds.`
4. Stop motors.	`// 4. Stop motors.`
5. Disconnect from Create.	`// 5. Disconnect from Create.`
6. End the program.	`// 6. End the program.`

Moving the *Create 2*

Analysis:

Flowchart



Moving the *Create 2*

Solution:

Source Code

Pseudocode

1. Connect to Create 2.
2. Drive forward at 100 mm/sec.
3. Wait for 4 seconds.
4. Stop motors.
5. Disconnect from Create 2.

```
int main()
{

  create_connect();

  create_drive_direct(100, 100);

  msleep(4000);

  create_stop();

  create_disconnect();

  return 0;
}
```

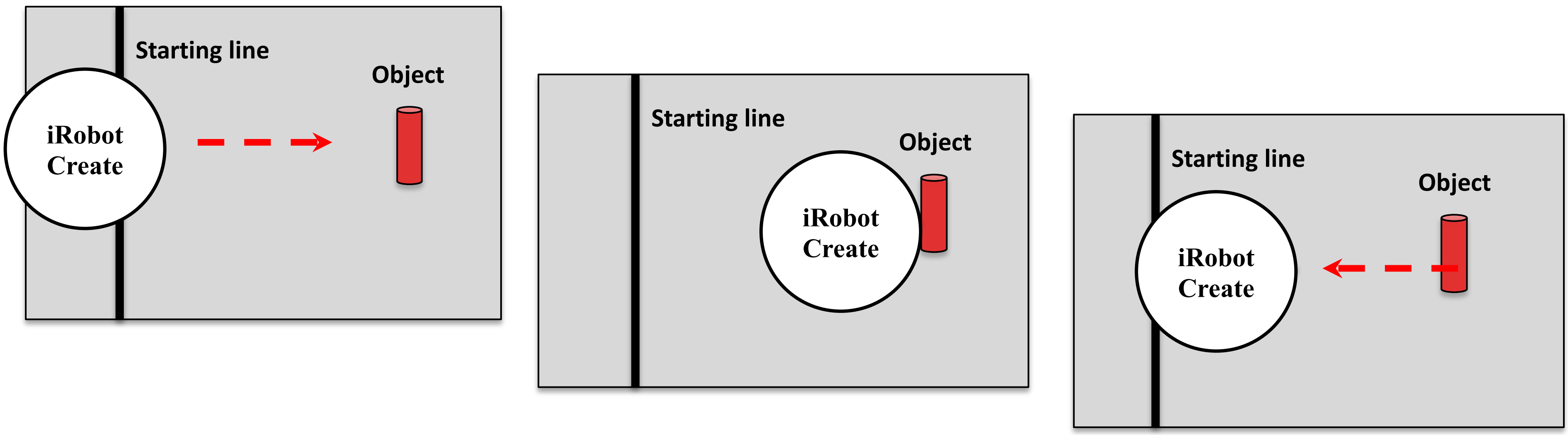
Execution: Compile and run your program.

#Botball®

Touch an Object and "Go Home"

Description: Write a program for the KIPR Robotics Controller that drives the **Create** forward until it touches an object (or gets as close as it can), and then returns to its starting location (home).

- Move the object to various distances.



#Botball®

Moving the iRobot *Create 2*: Part 2

Create Distance and Angle Functions

Create 2 Distance/Angle Functions

The *Create* has a built-in sensor that measures the distance traveled (in millimeters) and the angle turned (in degrees).

This is similar to the motor position counter... but *better!*

```
get_create_distance();
// Tells us the distance the Create has traveled in mm.

set_create_distance(0);
// Resets the Create distance traveled to 0 mm.

get_create_total_angle();
// Tells us the total angle the Create has turned in degrees.
// Positive angles are to the left. Negative angles are to the right.

set_create_total_angle(0);
// Resets the Create angle turned to 0 degrees.
```

Using *Create 2* Functions

Examples:

```
int main()
{
  create_connect();
  set_create_distance(0);
  while (get_create_distance() < 1000)
  {
    create_drive_direct(200, 200);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```

```
int main()
{
  create_connect();
  set_create_total_angle(0);
  while (get_create_total_angle() < 90)
  {
    create_drive_direct(-200, 200);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```

Distance is in millimeter. Have students try different distances. 92 mm + 36”

Printing *Create* 2 Sensor Values

Sometimes it is helpful to see the actual values from the create sensors. To do this, you can use the same print function we used before to print text.

To print a changing int value:

This is where it will print the provided value. Must be %d for integers.

```
printf("Angle Value: %d\n", get_create_total_angle());
printf("Value: %d\n", get_create_distance());
```

This is just regular text and can change.

After the comma is where you provide what value you want to print. It can be a function call (as here) or a variable name.

Turn function create

Try different angles
Notice that the bigger the angle the less accurate it is.
So you can find out how many degrees it's off.
Distance-degrees off

```
void create_spin_right (int degrees);
int main
{
create_spin_right (360);//full circle
  return 0;
}
void create_spin_right (degrees)
{
while (get_create_total_angle() > (degrees)
  {
    create_drive_direct(200, -200);
  }
  create_stop();

}
```

```
Void create_spin_right (int degrees);
int main
{
Create_spin_right (360);//full circle
Return o
}
Void create_spin_right (degrees)
{
while (get_create_total_angle() > (degrees +33)
  {
    create_drive_direct(200, -200);
  }
  create_stop();

}
```

Ran left side program and noticed it was about 33 degrees off, so now add 33 in this program.

Printing Create Sensor Values

```
int main()
{
  create_connect();
  set_create_total_angle(0);
  while (get_create_total_angle() > -90)
  {
    create_drive_direct(200, -200);
  }
  create_stop();

  printf("Angle Value: %d\n", get_create_total_angle());
  printf("Distance Value: %d\n", get_create_distance());

  create_disconnect();
  return 0;
}
```

Creighton uses <90. D

Printing the create sensor values can be a good way to debug an issue!

You can print before, inside and after the loop as well.

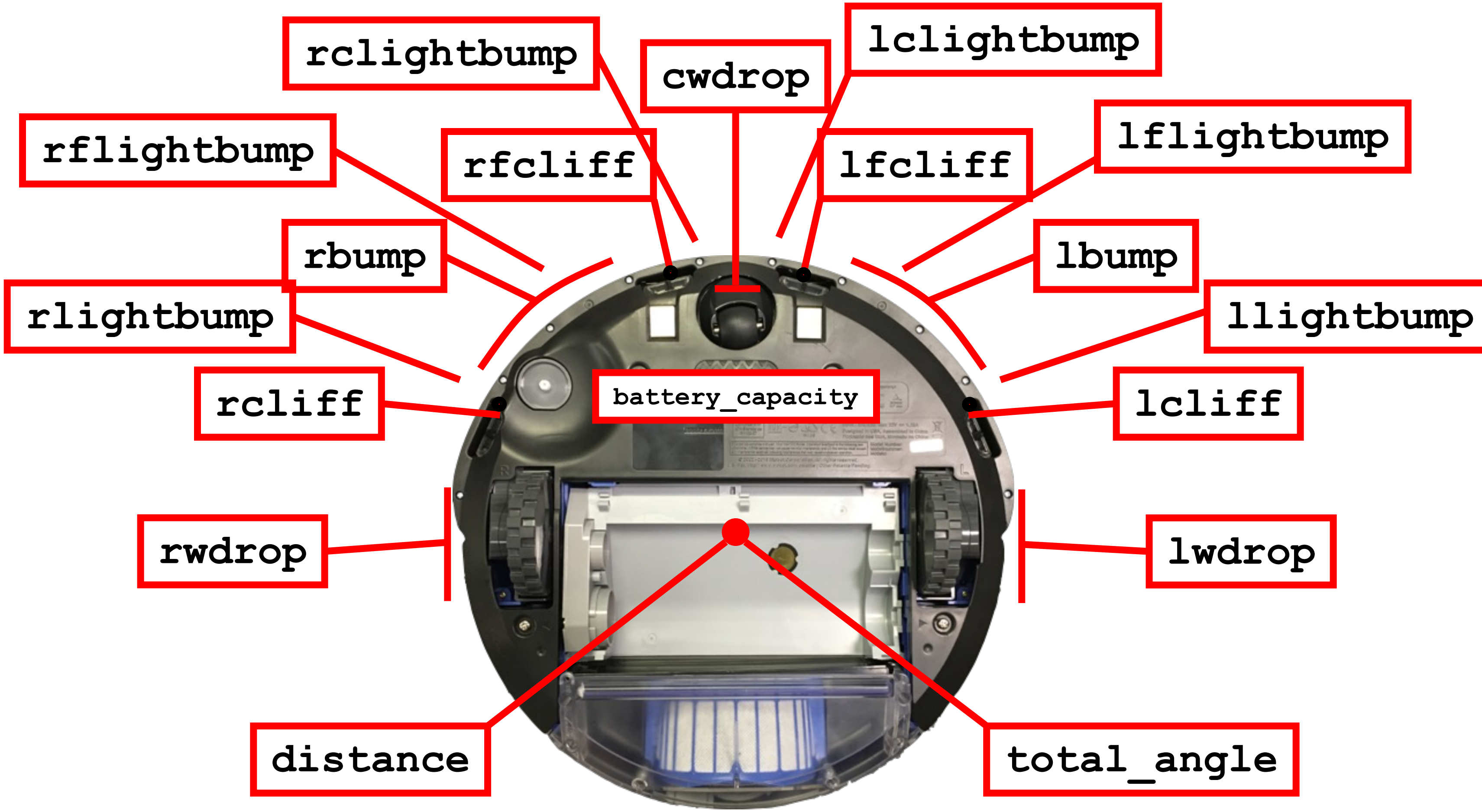
iRobot *Create 2* Sensors

Create 2 Sensor Functions

Logical Operators

Create 2 Sensor Functions

To get *Create 2* sensor values, type `get_create_sensor()`, replacing *`sensor`* with the name of the sensor



Create 2 Sensor Functions

```
get_create_lbump()
get_create_rbump()
// Tells us if the Create left/right bumper is pressed.
// Like a digital touch sensor.

get_create_lwdrop()
get_create_rwdrop()
get_create_cwdrop()
// Tells us if the Create left/right/center wheel is dropped.
// Like a digital touch sensor.

get_create_lcliff_amt()
get_create_lfcliff_amt()
get_create_rcliff_amt()
get_create_rfcliff_amt()
// Tells us the Create left/left-front/right/right-front cliff sensor value.
// Like an analog reflectance sensor.

get_create_battery_capacity()
// Tells us the Create battery level (0-100).
```

Using *Create 2* Sensor Functions

What does this say?

```
int main()
{
  create_connect();
  while (get_create_rbump() == 0)
  {
    create_drive_direct(100, 100);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```

Power is -500 to 500

Drive Until Bumped

Description: Write a program for the KIPR Wombat that drives the *Create 2* forward until a bumper is pressed, and then stops.

Analysis: What is the program supposed to do?

Pseudocode

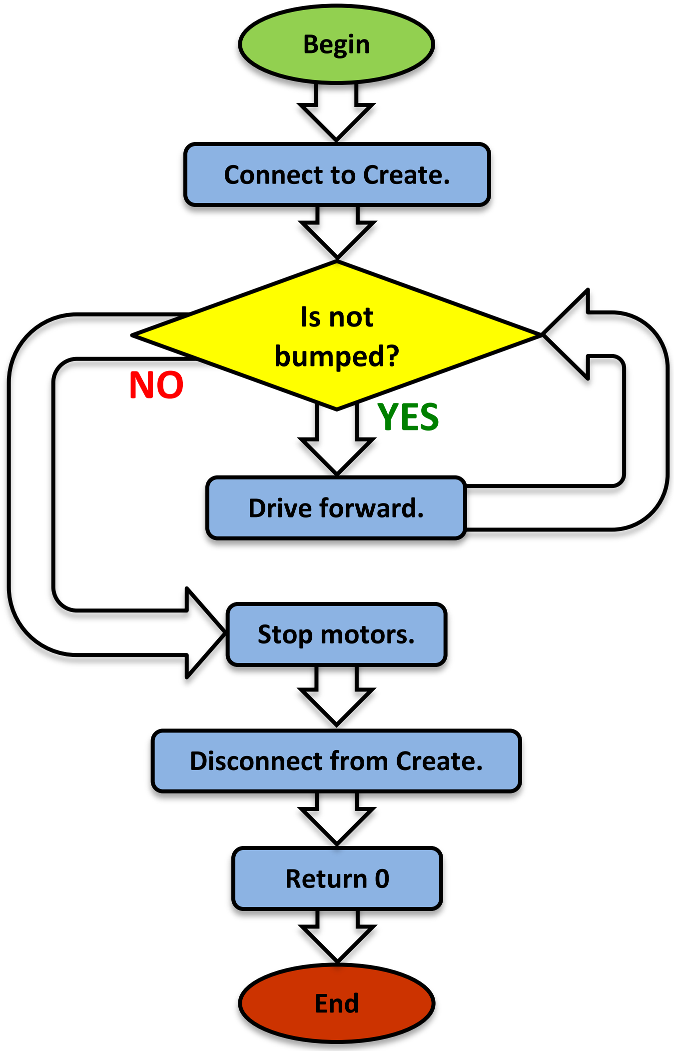
1. Connect to Create.
2. Loop: Is not bumped?
    1. Drive forward.
3. Stop motors.
4. Disconnect from Create.
5. End the program.

Comments

```
// 1. Connect to Create.
// 2. Loop: Is not bumped?
//    2.1. Drive forward.
// 3. Stop motors.
// 4. Disconnect from Create.
// 5. End the program.
```

Drive Until Bumped

Analysis: Flowchart



Drive Until Bumped

Solution:

Pseudocode

```
1. Connect to Create 2.
2. Loop: Is not bumped?
     Drive forward.
3. Stop motors.
4. Disconnect from Create 2.
5. End the program.
```

Source Code

```c
int main()
{

  create_connect();


  while (get_create_rbump() == 0)
  {

    create_drive_direct(200, 200);
  }

  create_stop();

  create_disconnect();

  return 0;
}
```

Connections to the Game Board

Description: Make the iRobot Create move forward in a straight line until it comes into contact with another object. Then have it make a 90º turn and again travel in a straight line for exactly 0.9 meters. Before your program ends, print to the screen the values for the total angle the create has turned and total distance it has driven. Solution to this one is on your own.

Line Follow With the Create

Description: Make the iRobot Create follow a line. The Create will follow a line or a piece of tape for a distance 1 yard.

What you need to know: You can use the Create front cliff sensor and the create_drive_direct commands to accomplish the same task that you accomplished in an earlier demobot line follow activity.

Point to note: The scale and value for the `get_create_lfcliff_amt()` may differ from the values of the analog sensor ports on the Wombat. You may consider finding the black and white values for this sensor by printing values of each to your screen.

Finding Sensor Values

```
int main()
{
    printf("Print values to screen to get threshold.\n");
    create_connect();
    while (a_button() == 0) // Push grey button to stop loop
    {
        printf("LF cliff value is %d\n", get_create_lfcliff_amt() );
        msleep(250);   // Give humans time to read it
    }

    create_disconnect();
    return 0;
}
```

Run this program and place the Create DemoBot over the white surface (***higher value with Create***) and black lines (***lower value with Create***) and record down the values and determine a “threshold” (middle) value.

Line Follow with Create Solution

```
int main()
{
    int threshold = INSERT_YOUR_VALUE;
    int speed = 250;

    printf("Follow the non-yellow brick road!\n");
    while ( (get_create_lbump() == 0) && (get_create_rbump() == 0) )
    {
      if (get_create_lfcliff_amt() < threshold)
      {
        create_drive_direct(0.5*speed, speed);
      }
      else
      {
        create_drive_direct(speed, 0.5*speed);
      }
    }

    ao();
    return 0;
}
```

You will need to find the threshold value (see prior slide) and adjust the speed and multiplier for how fast the turn is (for sharper turns the difference must be greater)

#Botball®

Square Up With the Create

Description: Make the iRobot Create square up on a black line while moving in the forward direction.

What you need to know: You can use the Create cliff sensors(these are further back than what you might use for line follow) and the create_drive_direct commands to accomplish the same task that you accomplished in the square up with demobot.

Point to note: The scale and value for the `get_create_lcliff_amt()` and `get_create_rcliff_amt()` may differ from the values of the analog sensor ports on the Wombat. You may consider finding the black and white values for this sensor by printing values of each to your screen.

Square Up With the Create sample code

```
while(1){
            if(get_create_lcliff_amt()>gray &&
get_create_rcliff_amt()>gray)//gray is the midpoint between black and
white
{
        create_drive_direct(speed,speed);//you will need to set a
variable speed that corresponds to some value
            }
            if(get_create_rcliff_amt()<gray){
                create_drive_direct(speed,stop);
            }
            if(get_create_lcliff_amt()<gray){
                create_drive_direct(stop,speed);
            }
            if(get_create_lcliff_amt()<gray &&
get_create_rcliff_amt()<gray) {
                create_drive_direct(stop,stop);
                break;
            }
        }
```

Square Up with Create Solution

```
int main()
{
    int threshold = INSERT_YOUR_VALUE;   int speed = 250;
    printf("Drive forward to black line, square up then stop.\n");
    while ( (get_create_lfcliff_amt() < threshold) ||
            (get_create_rfcliff_amt() < threshold) )
    {
        if ( (get_create_lfcliff_amt() < threshold) &&
             (get_create_rfcliff_amt() >= threshold) )
        {
            create_drive_direct(0.75*speed, -0.1*speed);
        }
        else if ( (get_create_lfcliff_amt() >= threshold) &&
                  (get_create_rfcliff_amt() < threshold) )
        {
            create_drive_direct(-0.1*speed, 0.75*speed);
        }
        else
        {
            create_drive_direct(speed,speed);
        }
    }
    create_stop();
    create_disconnect();
    return 0;
}
```

Logical Operators

Multiple Boolean Tests

`while`, `if`, and Logical Operators

Logical Operators

Recall the **Boolean test** for `while` loops and `if-else` conditionals…

`while (Boolean test)`   `if (Boolean test)`

- The **Boolean test** (conditional) can contain ***multiple*** **Boolean tests** combined using a "**Logical operator**", such as:
  - `&&`   And
  - `||`   Or
  - `!`   Not

We put parentheses (and) around *each* Boolean test…

```
while ((Boolean test 1) && (Boolean test 2))

if ((Boolean test 1) || (!Boolean test 2))
```

- The next slide provides a cheat sheet for **Logical operators**.

#Botball®

Logical Operators Cheat Sheet

Boolean	English Question	True Example	False Example

`A && B`	Are **both** A **and** B true?	`true && true`	`true && false`
			`false && true`
			`false && false`

`A || B`	Is **at least one** of A **or** B true?	`true || true`
		`false || true`
		`true || false`
			`false || false`

`!(A && B)`	Is **at least one** of A **or** B false?	`true && false`
		`false && true`
		`false && false`
			`true && true`

`!(A || B)`	Are **both** of A **and** B false?	`false || false`	`true || true`
			`false || true`
			`true || false`

! negates the **`true`** or **`false`** Boolean test.

Professional Development Workshop

#Botball®

while, if, and Logical Operators Examples

```
while ((get_create_lbump() == 0) && (get_create_rbump() == 0))
{
  // Code to execute ...
}
```

```
while ((digital(1) == 0) && (digital(2) == 0))
{
  // Code to repeat ...
}
```

```
if ((digital(1) == 1) || (digital(2) != 0))
{
  // Code to execute ...
}
```

```
if ((analog(3) < 512) || (digital(1) == 1))
{
  // Code to repeat ...
}
```

Using Logical Operators

What does this say?

```
int main()
{
  create_connect();
  while ((get_create_lbump() == 0) && (get_create_rbump() == 0))
  {
    create_drive_direct(100, 100);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```


#Botball

Connections to the Board Game

Description: Write a program for the KIPR Robotics Controller that drives the *Create* forward 1 meter or until a bumper is pressed, and then stops.

- How do we check for *distance traveled*? **Answer:** `get_create_distance() < 1000`
- How do we check for *bumper pressed*? **Answer:** `get_create_rbump() == 0`
- How do we check for that *both* are **true**?
  Answer: `((get_create_distance()) < 1000) && (get_create_rbump() == 0))`

Analysis: What is the program supposed to do?

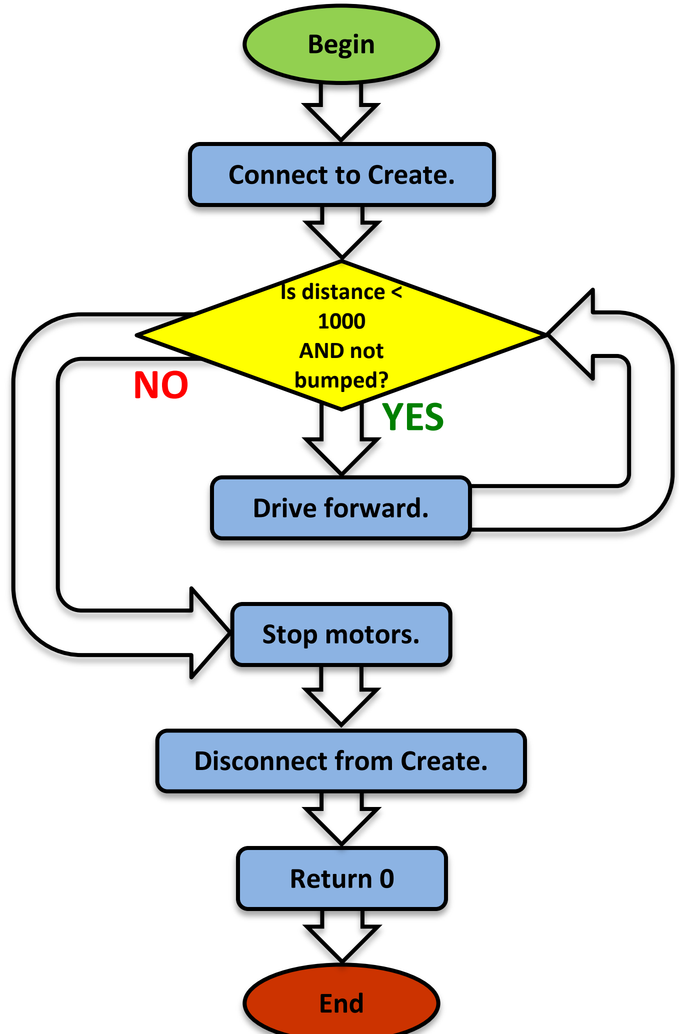
Pseudocode

1. Connect to Create.
2. Loop: Is distance < 1000
   AND not bumped?
   2.1. Drive forward.
3. Stop motors.
4. Disconnect from Create.
5. End the program.

#Botball®

Drive for Distance or Until Bumped

Analysis: Flowchart



#Botball

Solution:

Pseudocode

1. Connect to Create.
2. Loop: Is distance < 1000
   AND not bumped?
   2.1. Drive forward.
3. Stop motors.
4. Disconnect from Create.
5. End the program.

Source Code

```
int main()
{
  // 1. Connect to Create.
  create_connect();

  // 2. Loop: Is distance < 1000 AND not bumped?
  while ((get_create_distance() < 1000) && (get_create_rbump() == 0))
  {
    // 2.1. Drive forward.
    create_drive_direct(200, 200);
  } // end while

  // 3. Stop motors.
  create_stop();

  // 4. Disconnect from Create.
  create_disconnect();

  // 5. End the program.
  return 0;
} // end main
```

Drive for Distance or Until Bumped

Reflection: What did you notice after you ran the program?

- What happens if the *Create right bumper* is pressed ***before the Create travels a distance of 1 meter***?
- What happens if the *Create right bumper* is **not** pressed ***before the Create travels a distance of 1 meter***?
- What happens if the *Create **left** bumper* is pressed instead?
- How could you ***also*** check to see if the *Create **left** bumper* is pressed? **Answer:**

```
while ((get_create_distance() < 1000) && (get_create_lbump() == 0) && (get_create_rbump() == 0))
```

Color Camera

Using the Color Camera

Setting the Color Tracking Channels

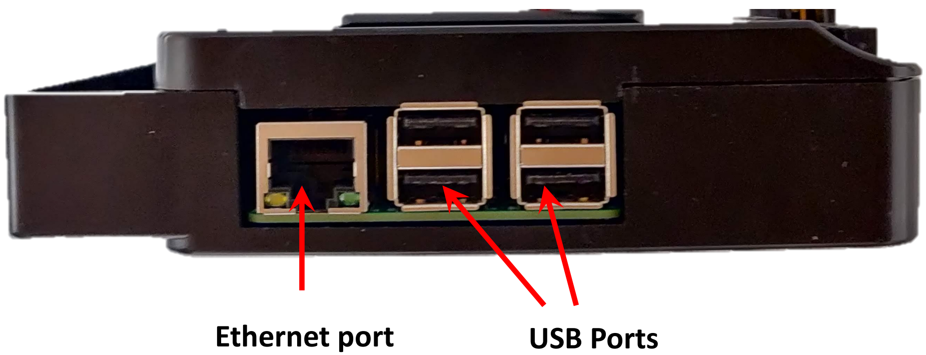
About Color Tracking

Camera Functions

Color Camera

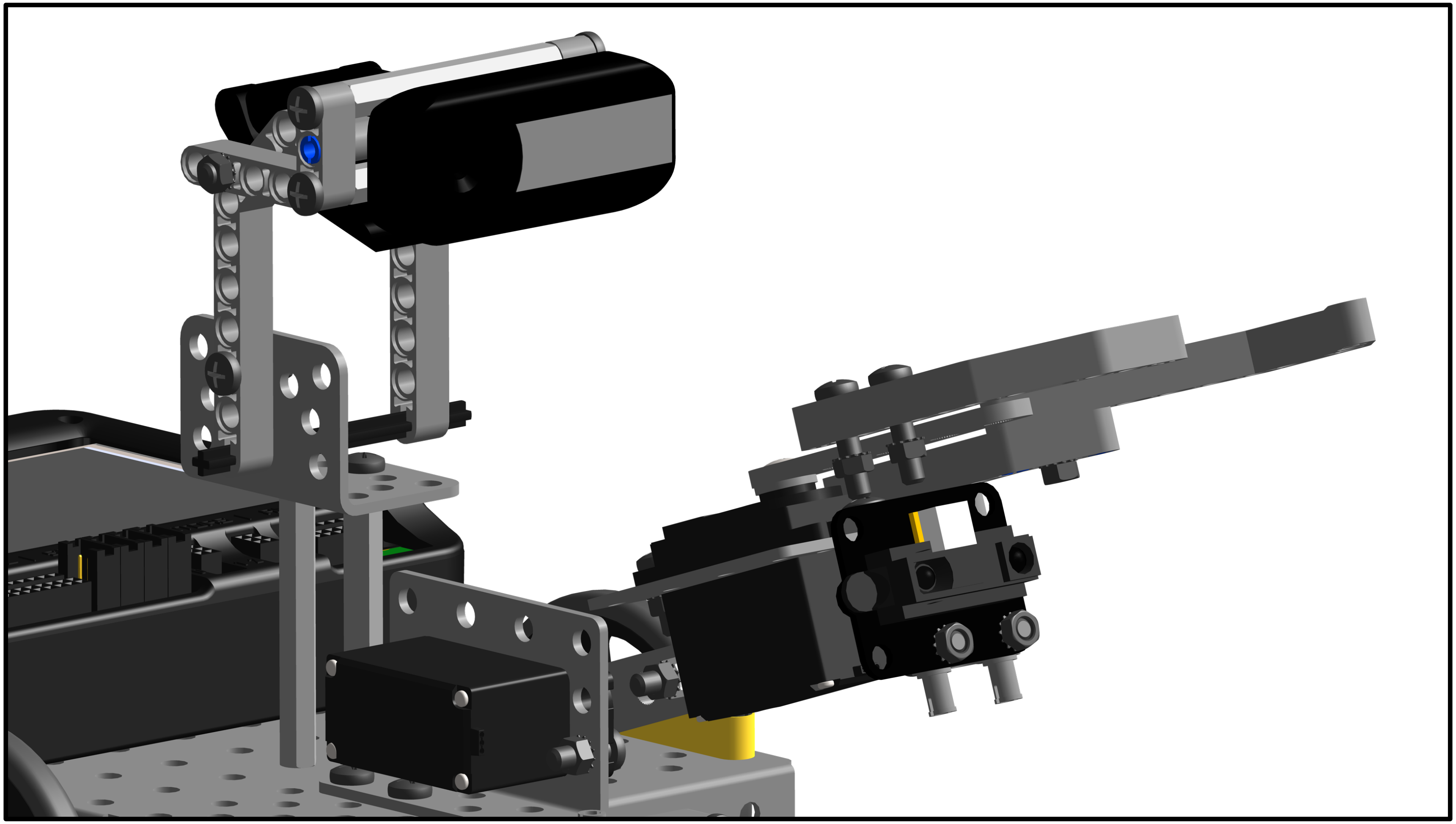
For this activity, you will need the **camera**.

- The camera plugs into one of the USB (type A) ports on the back of the Wombat.
- **Warning:** Unplugging the camera while it is being accessed can freeze the Wombat, requiring it to be rebooted.



Camera Build

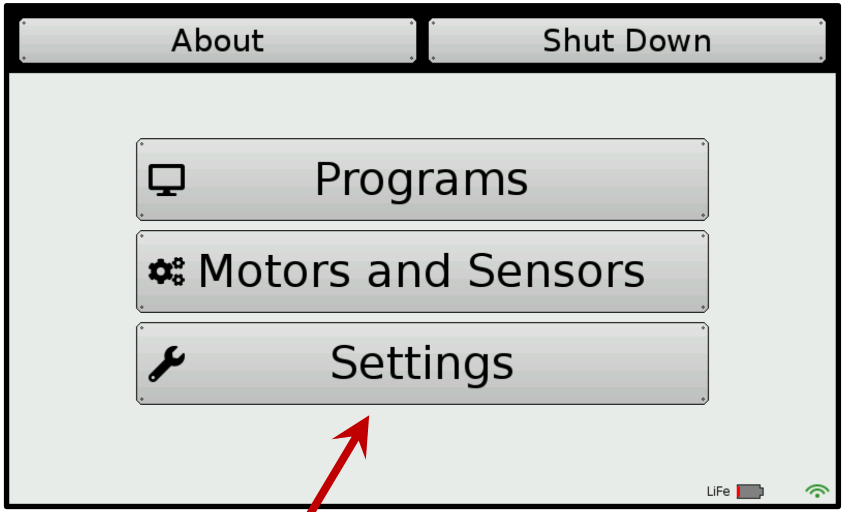
See the DemoBot build instructions for a way to mount the camera.

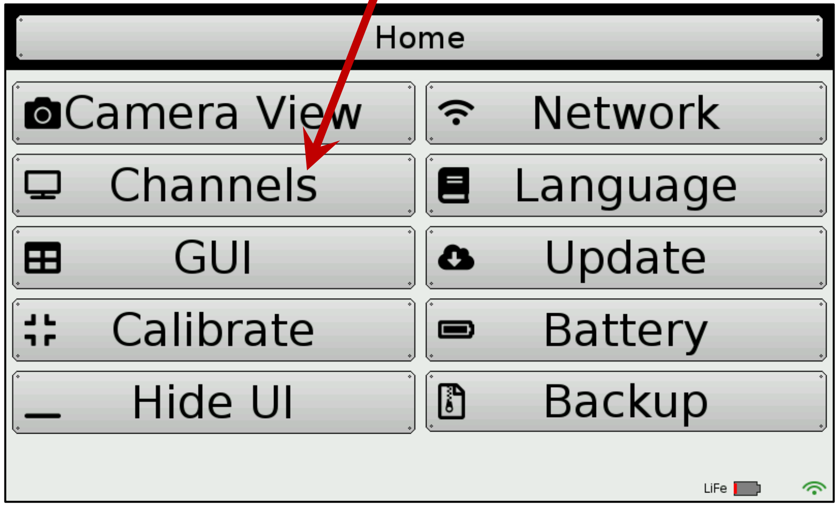


Setting the Color Tracking Channels

1. Select ***Settings***
2. Select ***Channels***

2

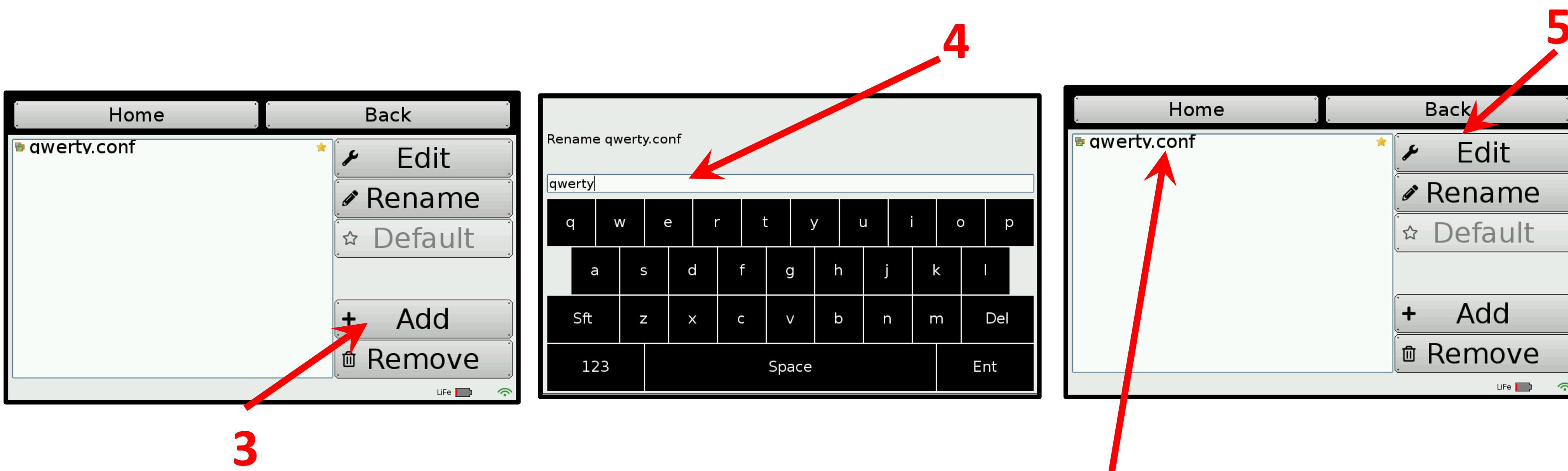




1

Setting the Color Tracking Channels

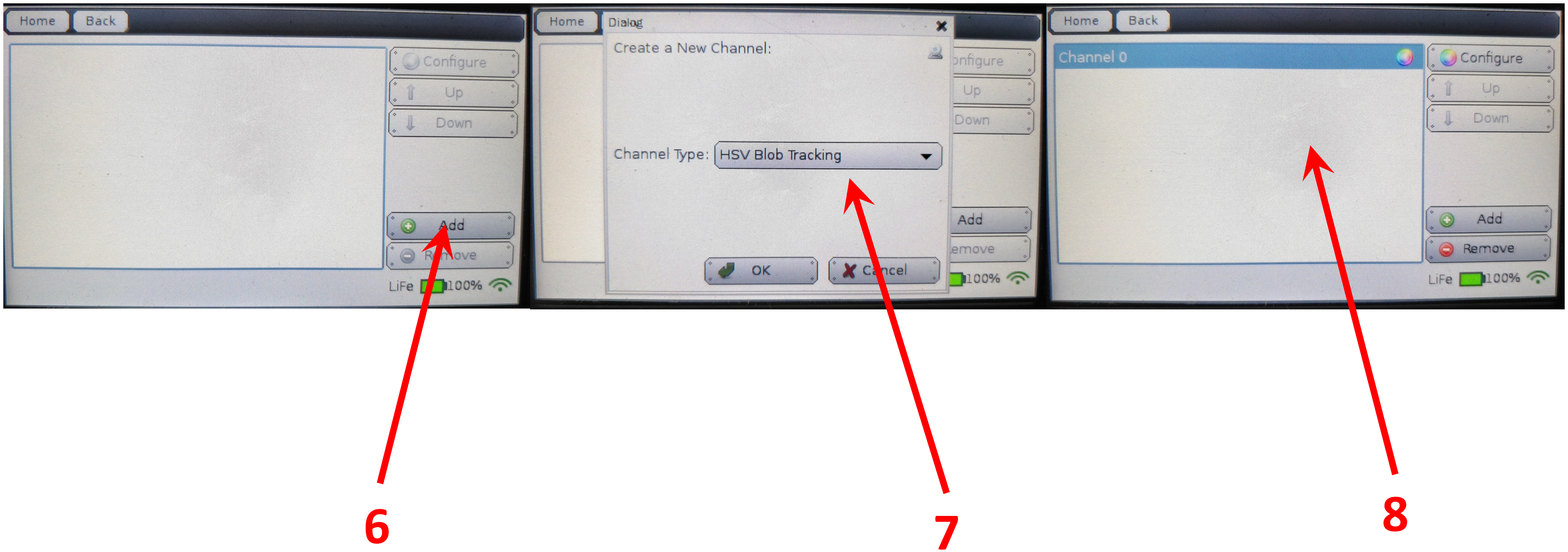
3. To specify a **camera configuration**, press the *Add* button.
4. Enter a configuration name, such as **find_green**, then press the *Ent* button.
5. Highlight the new configuration and press the *edit* button.



Note: if there is more than one configuration, select one, and press the "Default" button to make it be the one in use!

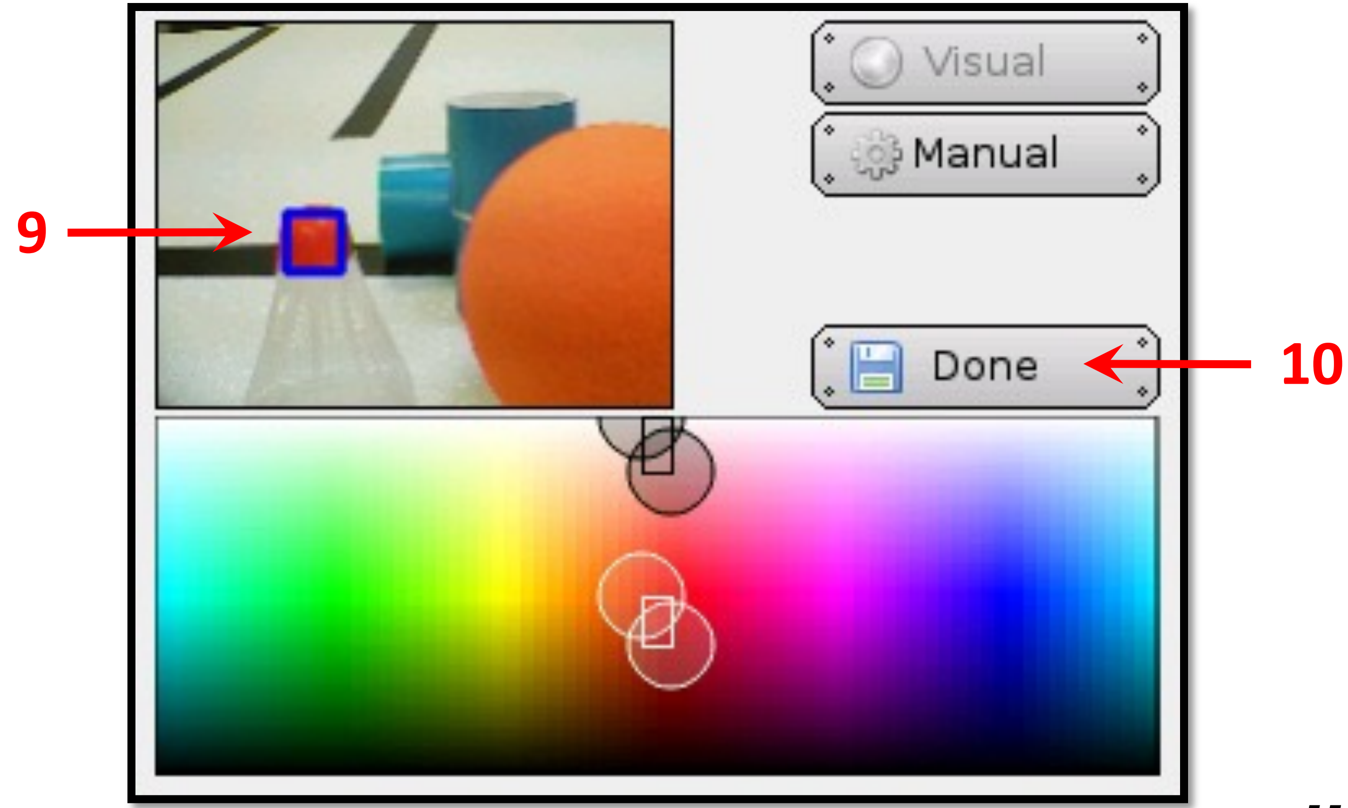
Setting the Color Tracking Channels

6. Press the *Add* button to add a channel to the configuration.
7. Select ***HSV Blob Tracking***, then *OK* to set this up to track a color.
8. Highlight the channel, then press *Configure* to edit settings.
   - The first channel is 0 by default. You can have up to four: **0**, **1**, **2**, and **3**.



Setting the Color Tracking Channels

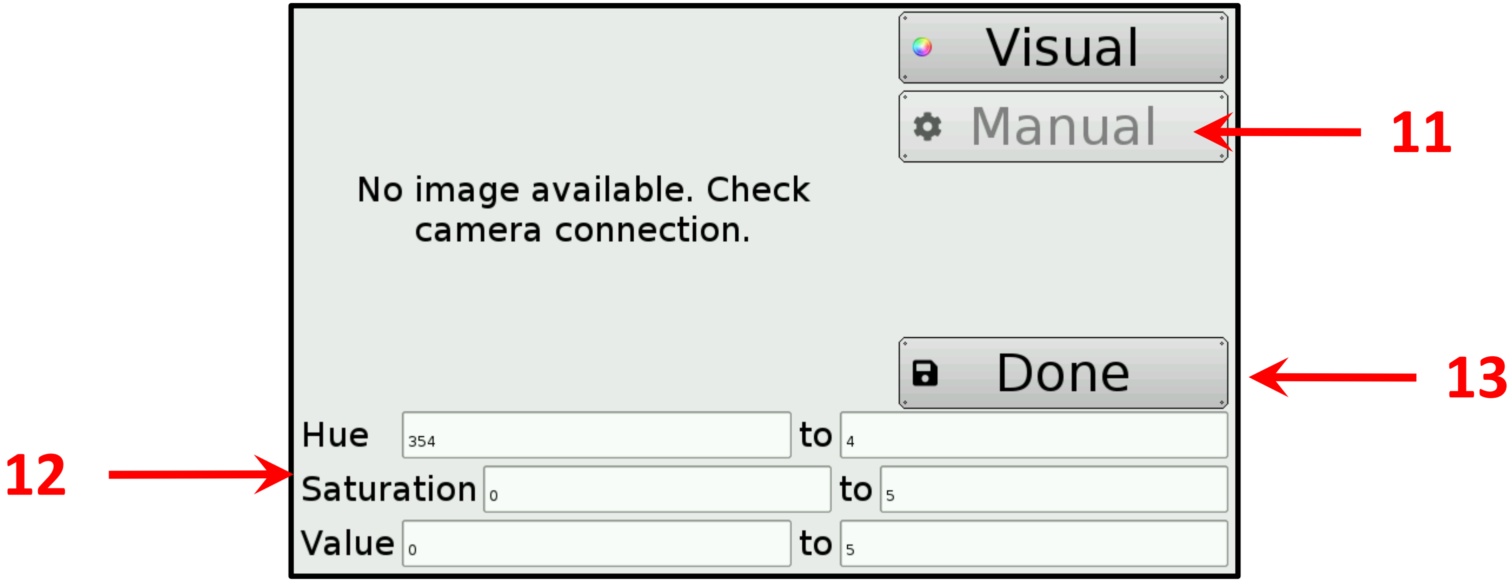
9. Place the colored object you want to track in front of the camera and **touch the object on the screen**.
   - A **bounding box** (**dark blue**) will appear around the selected object.
10. Press the *Done* button.



#Botball

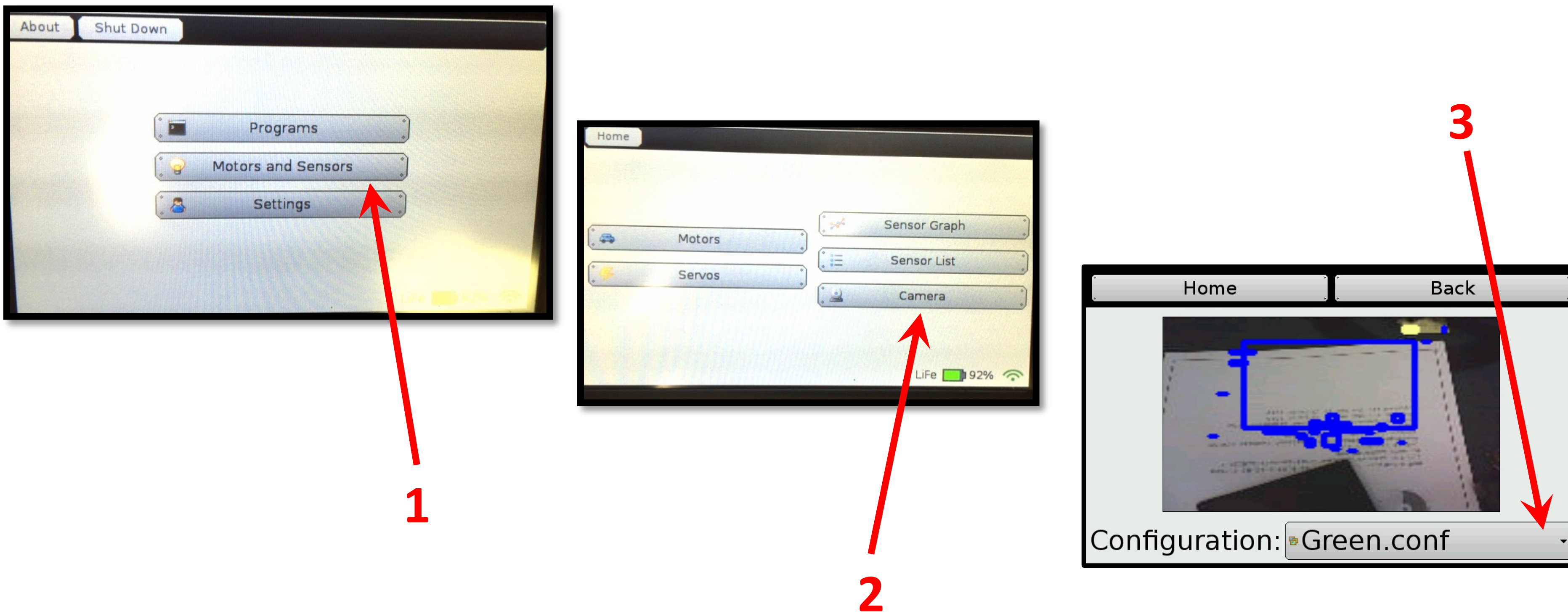
Setting the Color Tracking Channels

11. If you want to MANUALLY adjust the seetings, select *Manual*
12. Adjust individual values
13. Press the *Done* button.



Verify the Color Channel is Working

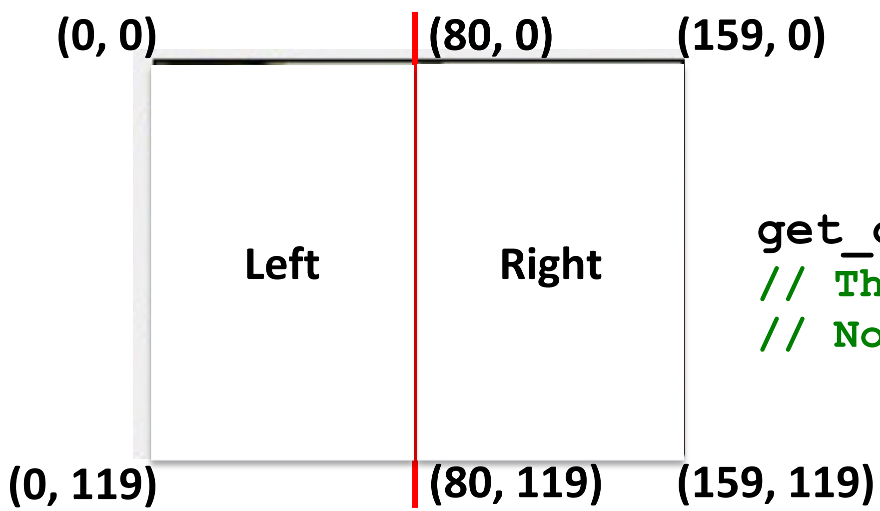
1. From the **Home** screen, press *Motors and Sensors* button.
2. Press the *Camera* button.
3. Make sure you select the configuration
4. Objects specified by the configuration should have a **bounding box**.



Tracking the Location of an Object

- You can use the **position** of the object in relation to the **center *x* (column)** of the image to tell if it is to the **left** or **right**.
  - The image is **160 columns wide**, so the **center column (*x*-value)** is 80.
  - An ***x*-value** of 80 is straight ahead.
  - An ***x*-value** between 0 and 79 is to the ***left***.
  - An ***x*-value** between 81 and 159 is to the ***right***.

- You can also use the **position** of the object in relation to the **center *y* (row)** of the image to tell **how far away** it is.



Object
0, 1, 2, …
(largest to smallest)

Channel #

```
get_object_center_x(0, 0);
// The x-value of the tracked object.
// Note: number between 0 and 159.
```

Camera Functions

```
camera_open();
// Opens the connection to the camera.

camera_close();
// Closes the connection to the camera.

camera_update();
// Gets a new picture (image) from the camera and performs color tracking.

get_object_count(channel #)
// The number of objects being tracked on the specified color channel.

get_object_center_x(channel #, object #)
// The center x (column) coordinate value of the object # on the color channel.

get_object_center_y(channel #, object #)
// The center y (row) coordinate value of the object # on the color channel.
```

Initial Camera Functions

Resource

Programming statements always used with the camera:

```
camera_open();    // opens camera

camera_update(); // retrieves current image
```

If either of these two functions execute successfully they return 1, otherwise they return a value of 0

```
camera_close(); // closes camera
```

On older controllers, after opening the camera you should wait (msleep) three seconds before doing anything else; this gives the camera time to boot.

Camera Functions Continued

A commonly used camera function, almost always after **camera_update()** but often forgotten about. This function returns the number of objects "seen/found" in the **last** camera update (which could have been a while ago)

```
if(get_object_count(0) > 0)
{
    // code if object seen on channel (color) 0
}
```

Channel #: 0,1,2,3

- We chose 0 as our default
- This could be red or blue or green, etc. If you use a variable you could have and integer named **red_channel** and that would be easier to understand here

Number of objects should be greater than zero otherwise nothing was seen for the color represented by this channel

#Botball®

Assessment: Camera Functions

Write the answers to the following questions:

1. Which function updates the camera image?
2. Which function turns the camera on?
3. When would you need to update the camera image? Before or after finding the object?
4. Which function is looking for the colored object?
5. What is the function that prints something to the screen?

Assessment: Camera Functions Answers

```
1. camera_update(); // retrieves current image

2. camera_open();

3. Before

4. get_object_count(channel#)> 0;

5. printf("Hi");
```


#Botball®

I See Green

Camera Activity 1

Goal: Write a program that will allow you to check to see if the camera is tracking the color that you want it to see.

1. Setup one of the channels for **green** objects

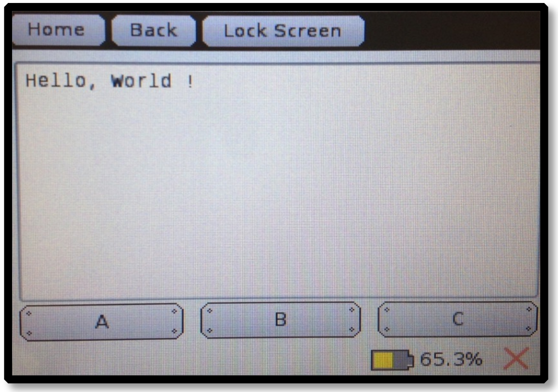
2. Write a program to look for **green** objects until the A button is pressed
   a) The program should print the words “I see green” when green objects come into view
   b) The program should print “Where is the green?” when it doesn’t seen green.

I See Green Continued

Example of code planning sheet:

1. Open the camera (starts communication between Controller & Camera)
2. Checks the status of the a_button
   a) We will use this step to create the **loop** that will keep your camera checking for images
3. Update the camera image (takes a snapshot of the current camera view)
4. Get an object count (the number of objects in the image)
5. Print "I see green." (if green object seen, otherwise "Where is the green?")
6. Remember if you want to stop the program you must press the A button: because you had a while loop that exits when a_button is pressed

Buttons

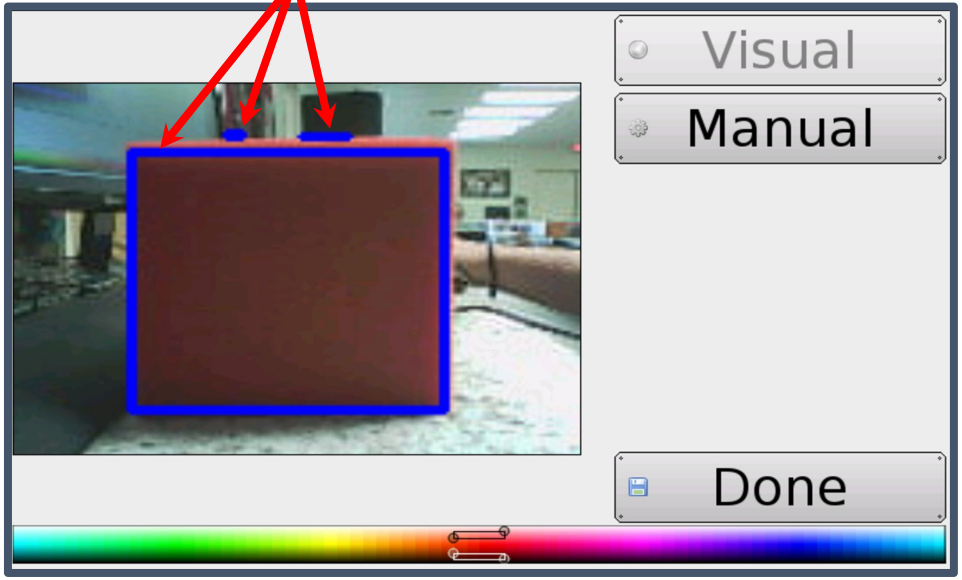


Getting the Object Count

Resource

- Each object is numbered with the one with the largest area being object 0, the next largest being 1, and so on.
- The function below can be used to get the number of objects visible. This should only be done after a **`camera_update()`**

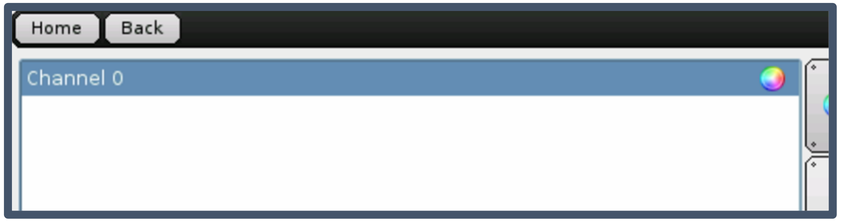
Each object is bounded by a blue box on the sensor screen



`get_object_count(channel#)`

Channel #: 0,1,2, or 3

- We setup 0 for green



Professional Development Workshop

#Botball

I See Green Example

```
#include <kipr/botball.h>

int main()
{
  camera_open(); // opens and establishes communication with the camera
  while (a_button() == 0) // loops until the a button is pressed
  {
    camera_update(); // retrieves current camera image

    if (get_object_count(0) > 0) //does the camera see at least 1 green object
    {
      printf("I see green.\n");
    }
    else
    {
      printf("Where is the green?\n");
    }
  }

  camera_close();    //disconnects from the camera
  return 0;
}
```

`(get_object_count(0) > 0)`

channel # (0 was the one we set for green)

number of objects

#Botball®

Camera Activity 2

Goal: Print the number of objects the camera can see.
Activity:

1. Make sure you have configured your camera for this activity. Open a new project in your folder and write a program that does the following:
   a. Opens the camera
   b. Update the camera image
   c. Print the number of objects on the screen
   d. Close camera at the end
2. Proceed to the next slide for a sample solution.

Variations -
Run your program multiple times (or add a loop!) with different amount of objects (in the desired color, and other colors) in front of the camera and watch the number change (or not change).

Camera Activity 2: One possible solution

```
int main()
{
  int count;          // Create an variable to represent the # of objects
  camera_open();     // Opens camera

  camera_update(); // Updates camera until it succeeds

  count = get_object_count(0); // Capture number of objects seen
  printf("There are %d objects on the screen.\n", count);
  camera_close(); // Camera closed

  return 0;
}
```

```
printf("There are %d objects on the screen.\n", count);
```

%d is a placeholder for an integer value

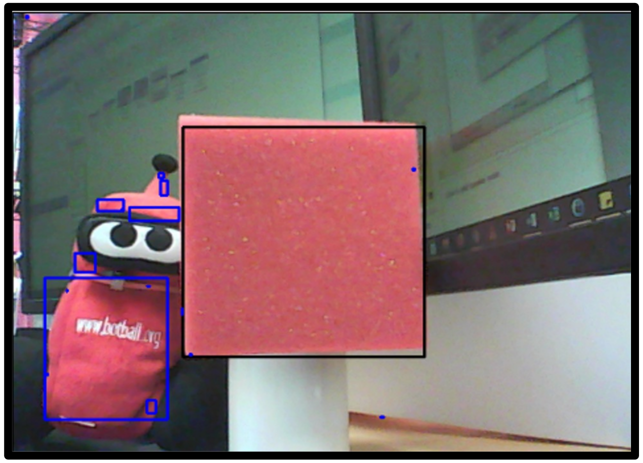
count is the integer value being placed into %d (note the use of a comma after the closed quote)

#Botball®

Output Examples

1 Object

15 Objects



Do you see 15 objects in the second image?

Each is highlighted by a blue bounding box. Some are very, very small. The computer counts each group, no matter how small, as a separate object. What your eye sees as blue may or may not be the same as what the camera sees as blue. As an example, a bright white reflected spot off of a table may look white to you but the camera sees it as having a high concentration of blue light.

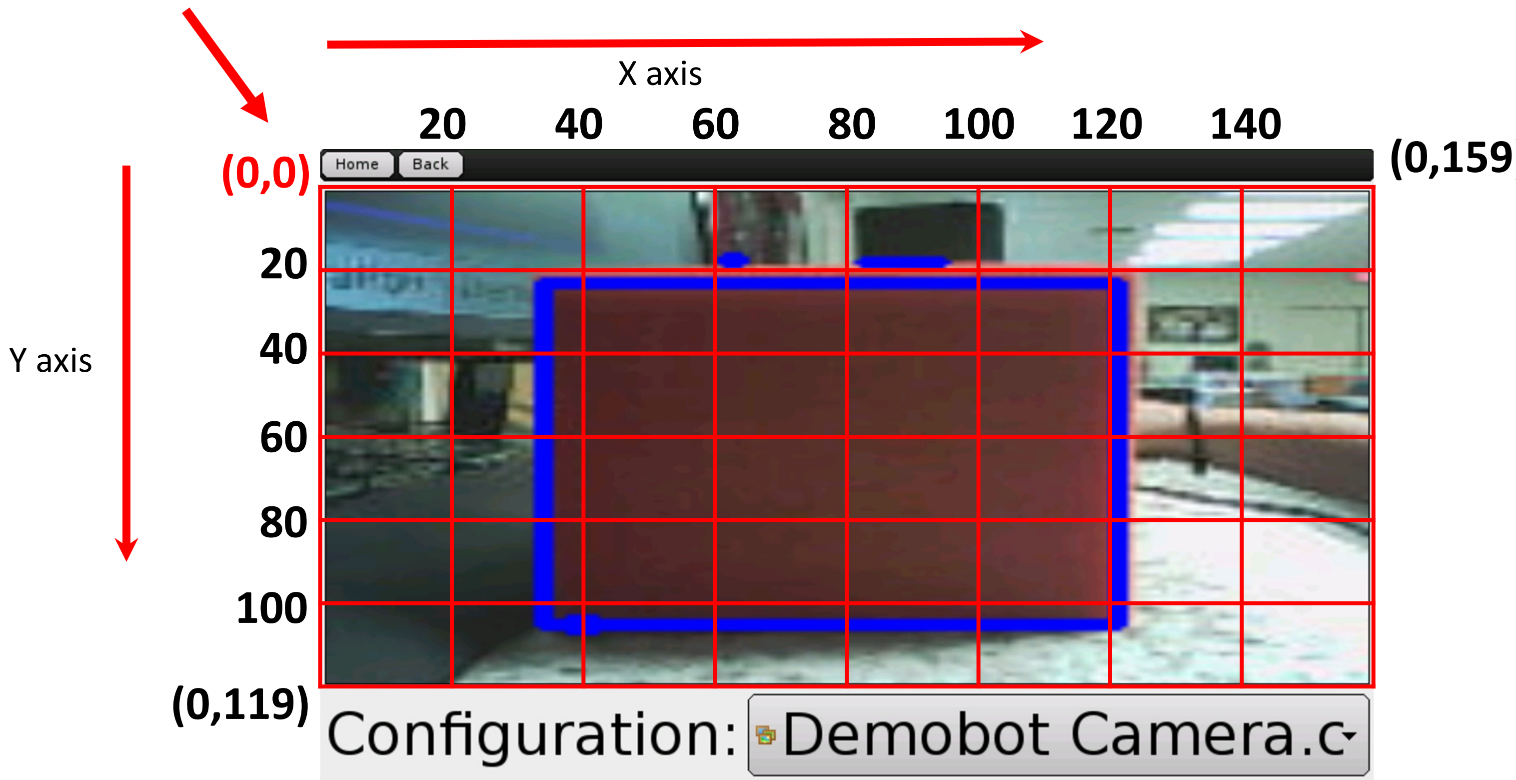
Objects versus Visual Noise

So, how do we figure out what objects are things we want the robot to interact with and which are just environmental noise?

There are other camera functions that we can use to get information about each object.

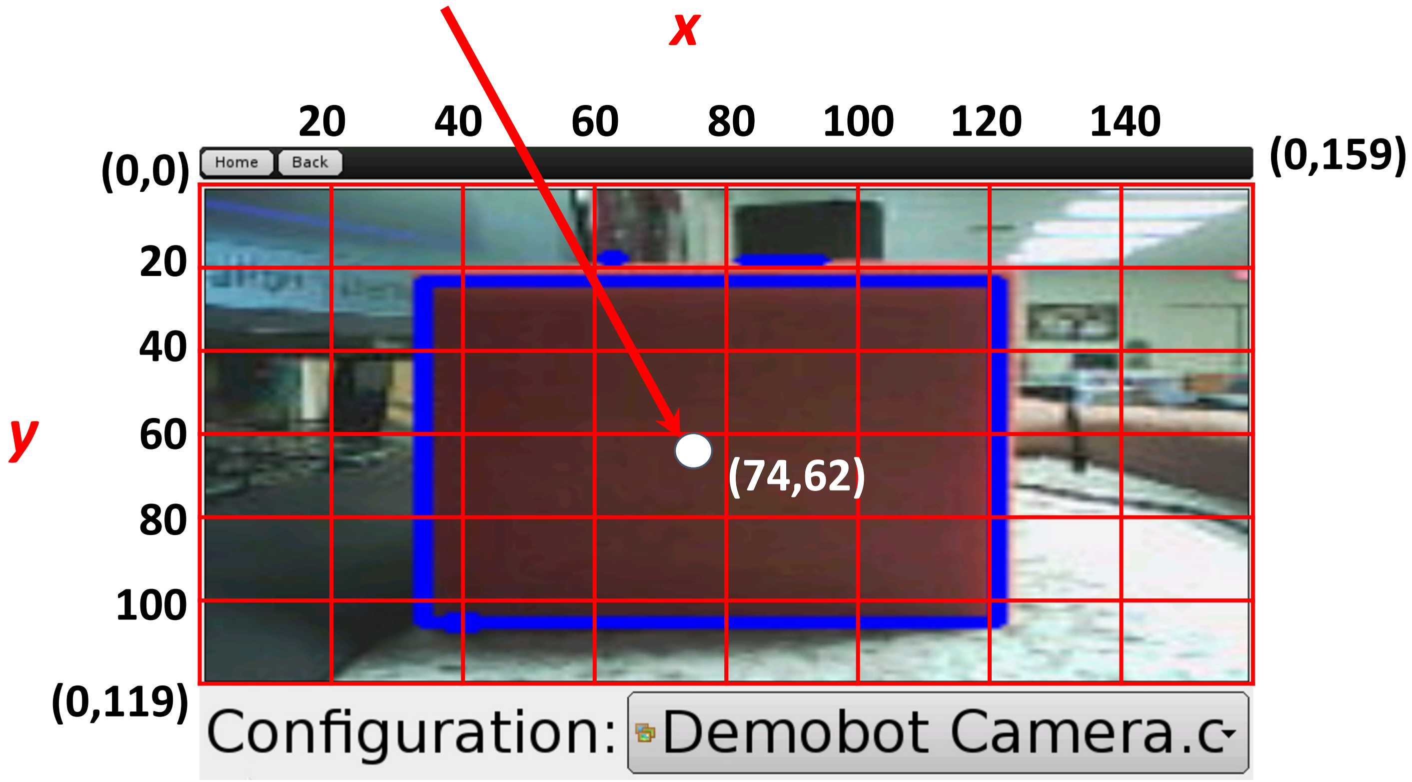
Resource

The camera view is like a graph except the coordinate (0, 0) is in the ***top left corner***. The max width is **159** and the max height is **119**.



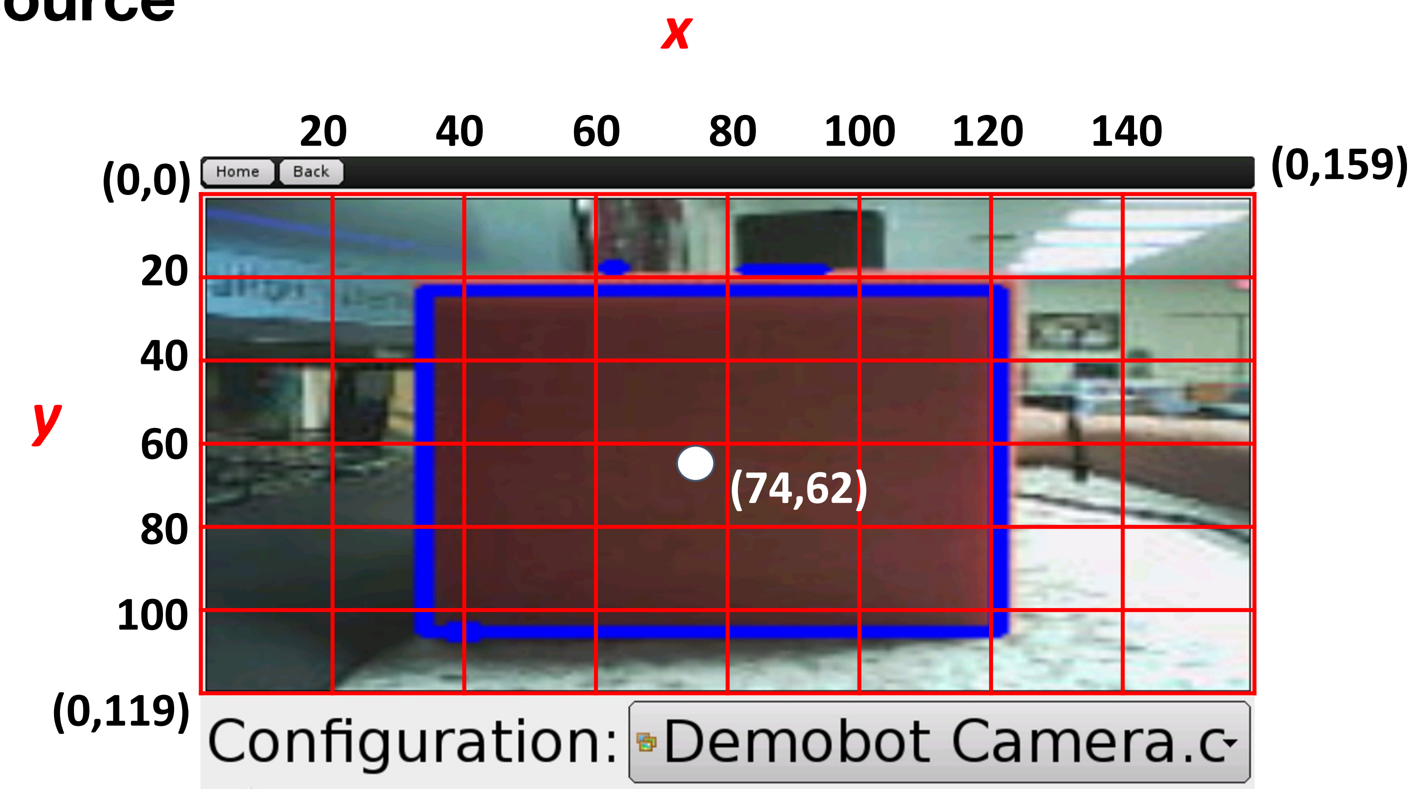
Resource

Each object has a center. In this case the center would have the coordinates (x = 74, y = 62).



Getting the Object Center

Resource



These functions can be used to get the center x and center y values of an object:

`get_object_center_x(channel#, object#)`

`get_object_center_y(channel#, object#)`

Note that the "first" **object#** (0) is the largest one of the color represented by **channel#**

Camera Activity 3

Goal: Find and print the center coordinates of an object with the camera

1. Make sure you have configured your camera for this activity. Open a new project in your folder and write a program that does the following:
   a. Opens the camera
   b. Update the camera image
   c. *Check to see if there is at least one object on the screen*
      i. *get_object_center functions order the objects by size. The largest object has ID number 0.*
   d. *If there is at least one object, print the object center x and y coordinates*
   e. Close camera connection

Variations -

Run your program multiple times with the object in different positions.

Activity 3 Template

```
int main()
{
```

(A) Variables go here

```
  camera_open(); //Opens camera

  camera_update(); //Updates camera until it succeeds
```

(B) New camera code goes here

```
  camera_close(); // Camera closed

  return 0;
}
```


Professional Development Workshop

Activity 3: Possible Solution

(A) Variables to be inserted in Camera Template (previous slide)

```
int x;
int y;
```

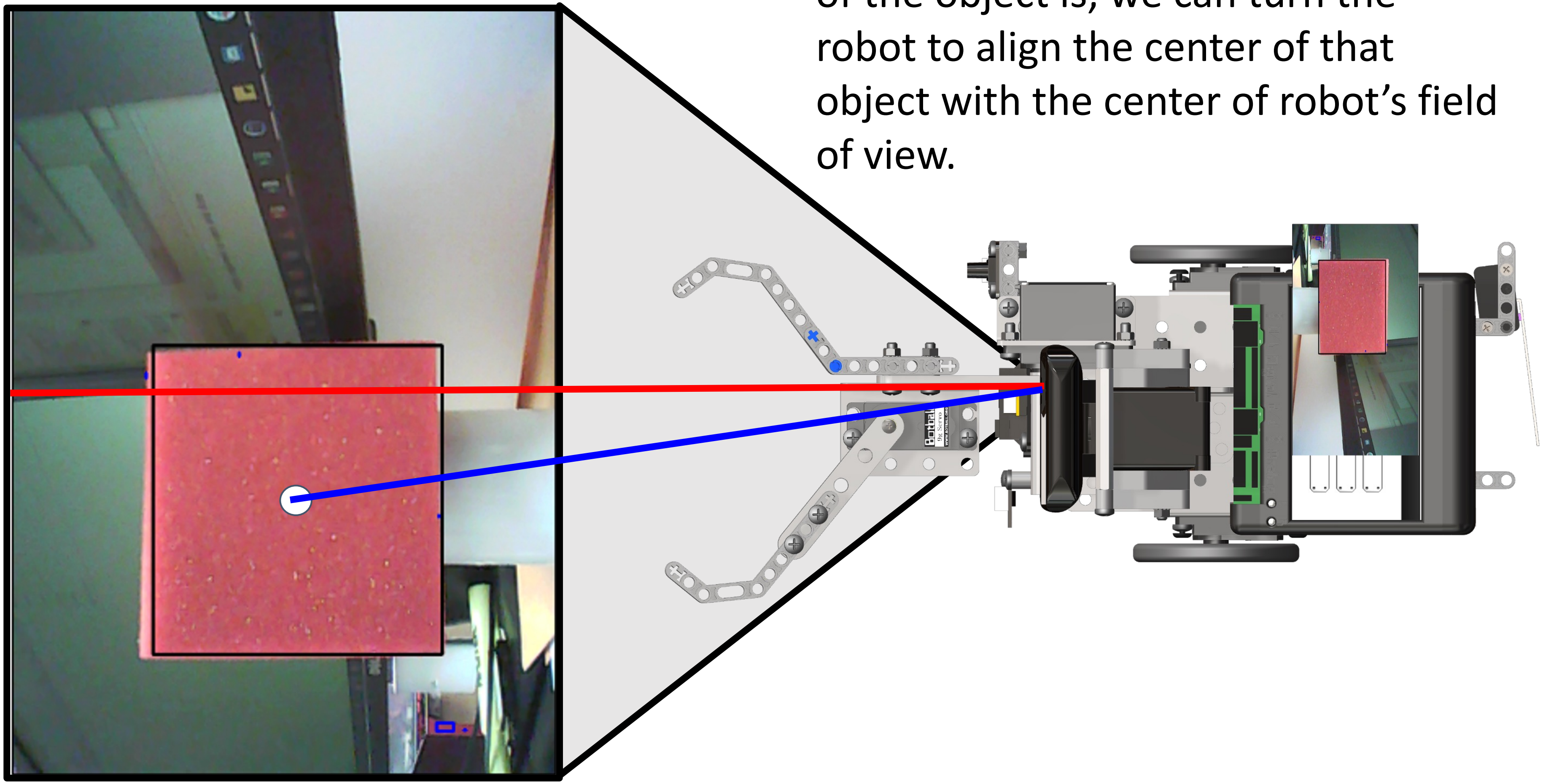
(B) New code to be inserted in Camera Template (previous slide)

```
if (get_object_count(0) > 0)
{
  x = get_object_center_x(0, 0);
  y = get_object_center_y(0, 0);
  printf("The center of the object is (%d,%d).\n",x,y);
}
```

To print out the x and y values, you could have made two separate printf statements as done previously. The solution above demonstrates how to format and use multiple integer values in one printf. Note that the two %d are separated by a comma; as is the two value variables: x, y.

Turning to an Object

Resource



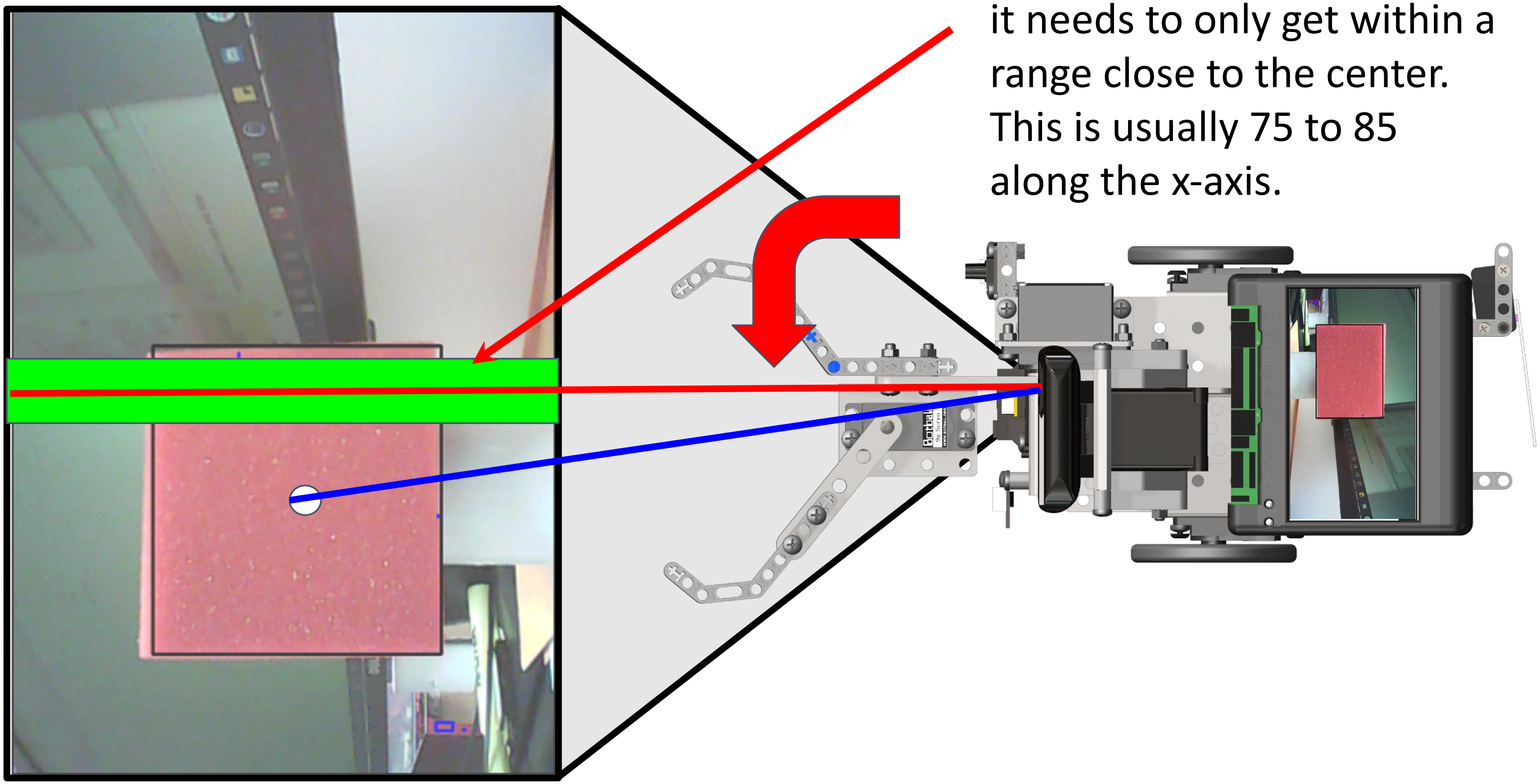
Now that we know where the center of the object is, we can turn the robot to align the center of that object with the center of robot's field of view.

Professional Development Workshop

#Botball

Resource

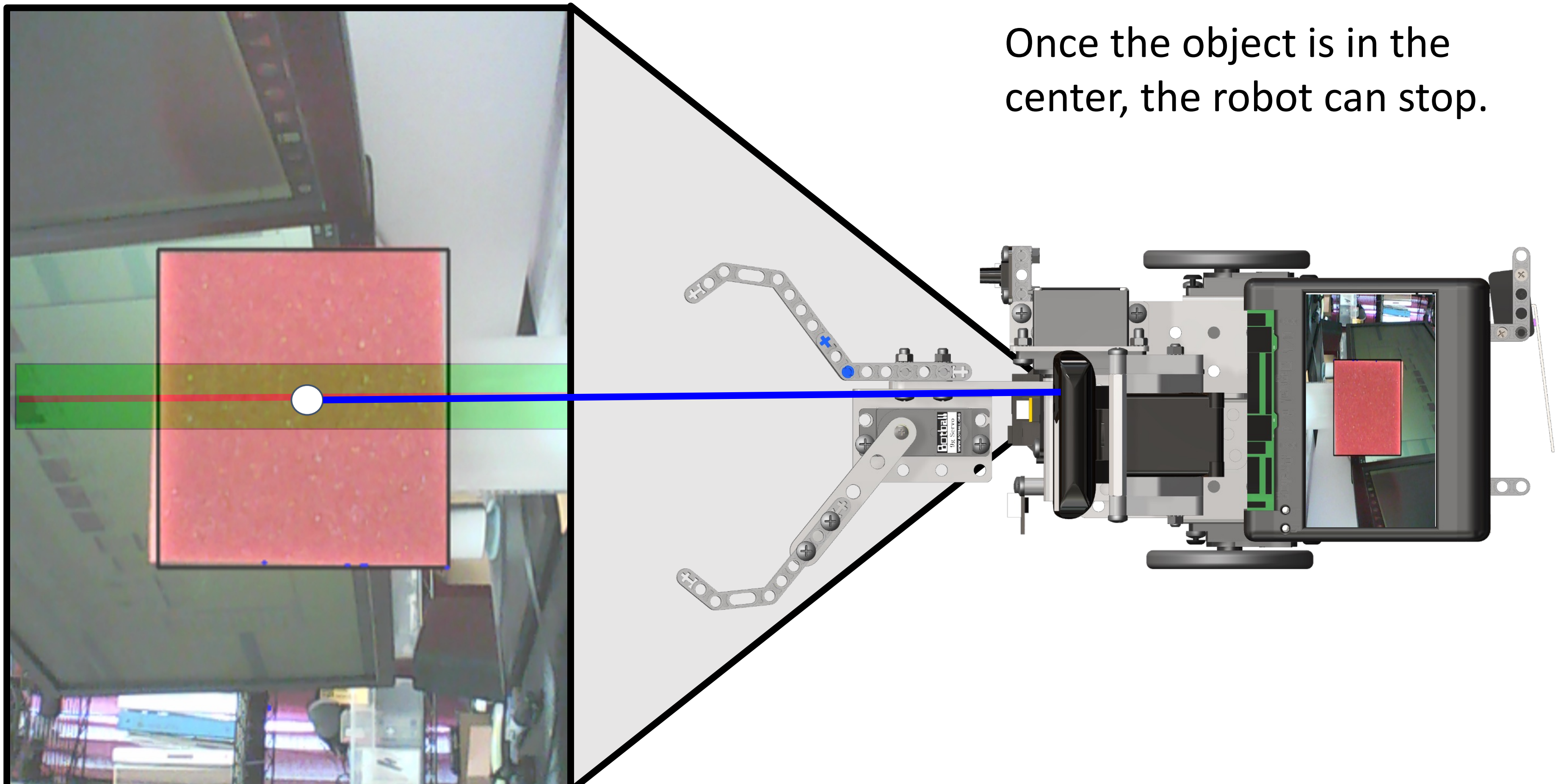
When the object is turning it needs to only get within a range close to the center. This is usually 75 to 85 along the x-axis.



#Botball

Turning to an Object

Resource



Once the object is in the center, the robot can stop.

#Botball

Camera Activity 4

Goal: Have a robot center itself on an object and print out the coordinates.
Activity:

1. Make sure you have configured your camera for this activity. Open a new project in your folder and write a program that does the following:
   a. Opens the camera
   b. If there is an object on the screen print the coordinates of the center
   c. Start turning until the object is in the center of the robot
   d. Print the new center coordinates of the object
   e. Close camera at the end
2. Proceed to the next slide for a sample solution.

Variations -
Have the object start off screen and have the robot turn until it sees it and it is centered.

Activity 4 Template

```
int main()
{
   int stop = 0;

   camera_open();

   while (stop == 0) // Updates camera image until stop pressed
   {
       camera_update();


   }

   camera_close(); // Camera closed


   return 0;
}
```

(A) Variables go here

(B) Code to find object center goes here

(C) Code goes here to turn the robot

Activity 4: Possible Solution

(A) Variables to be inserted in Camera Template (previous slide)

```
int x;
int y;
```

(B) New code to be inserted in Camera Template (previous slide)

```
camera_update();
if (get_object_count(0) > 0)
{
  x = get_object_center_x(0, 0);
  y = get_object_center_y(0, 0);
  printf("The center of the object is (%d,%d).\n",x,y);
}
```

Activity 4: Possible Solution

(C) New code to be inserted in Camera Template (previous slide)

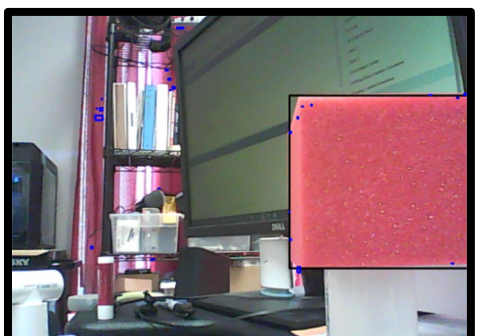
```
if (get_object_count(0) > 0)
{
  if (get_object_center_x(0,0) < 75)
  {
    motor(0,-25);    motor(3,25);
  }
  else if(get_object_center_x(0,0) > 85)
  {
    motor(0,25);    motor(3,-25);
  }
  else
  {
    stop = 1;
    ao();
    if (get_object_count(0) > 0)
    {
      x = get_object_center_x(0, 0);
      y = get_object_center_y(0, 0);
      printf("The center of the object is (%d,%d).\n",x,y);
    }
  }
}
```

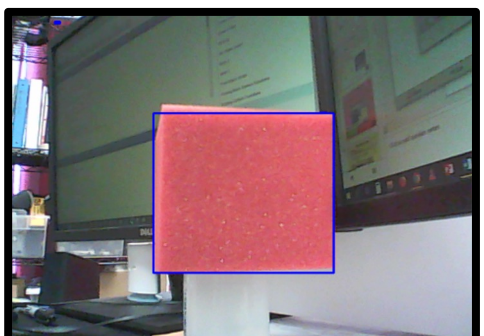
Output Examples

Success

Runner

```
Camera open successful.
The center of the object is (139,62).
The center of the object is (85,67).
~Wallaby()
Auto-stopping motors
Auto-disabling servos
Auto-stopping and disconnecting the Create
After the automatic create cleanup
~Create()
Program exited with code 0
```

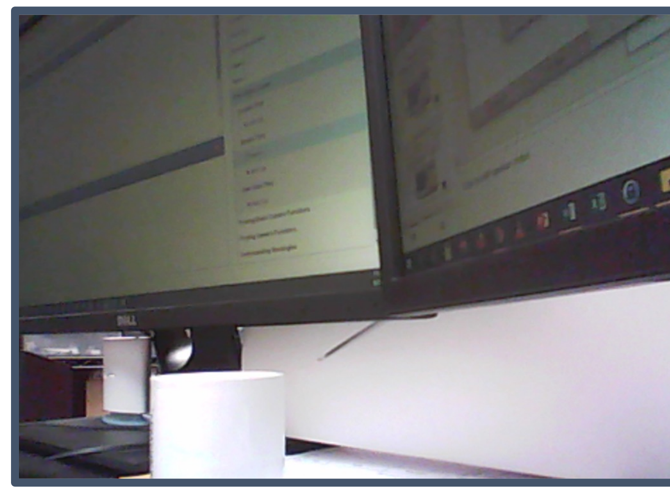




Fail

Runner

```
Camera open successful.
Did not find an object. Ending program.
Program exited with code 0
```



#Botball

Fun with Functions

Writing your own functions

Function prototypes, definitions, and calls

Writing Custom Functions

Remember: a **function** is like a recipe.

- When you **call** (use) a **function**, the computer (or robot) does all of the actions listed in the "recipe" **in the order they are listed**.

- **Functions** are very helpful if you take some actions multiple times:
  - driving straight forward → `drive_forward();`
  - making a 90° left turn → `turn_left_90();`
  - making a 180° turn → `turn_around();`
  - lifting an arm up → `lift_arm();`
  - closing a claw → `close_claw();`

We made these up… and that's the point!

You can write your own functions to do whatever you want!

- **Functions** often make it easier to **(1)** read the **`main`** function, and **(2)** change distance, turning, timing, or other values as necessary.

#Botball

Writing Custom Functions

There are **three components** to a function:

1. **Function prototype:** a *promise* to the computer that the function is defined somewhere (like an entry in the table of contents of a recipe book)
2. **Function definition:** the list of actions to be executed (the recipe)
3. **Function call:** using the function (recipe) in your program

Function prototypes go <u>above</u> `main`.

Function calls go <u>inside</u> `main` (or inside other functions).

Function definitions go <u>below</u> `main`.

```
include <kipr/botball.h>

void turn_left_90();

int main()
{
  turn_left_90();
  return 0;
}

void turn_left_90()
{
  while(gmpc(0) <= 1350)
  {
    motor(0,100);
    motor(3,0);
  }
  ao();
}
```

Use `void` in your function prototype if you are ***commanding*** the robot to do something.

#Botball®

Writing Custom Functions

The **function prototype** and the **function definition** first line *look* the same *except for one thing…*

prototype

definition

```
include <kipr/botball.h>

void turn_left_90();

int main()
{
  turn_left_90();
  return 0;
}

void turn_left_90()
{
  while(gmpc(0) <= 1350)
  {
    motor(0,100);
    motor(3,0);
  }
  ao();
}
```

Notice: no semicolon! (Why not?)

Writing Custom Functions

```
include <kipr/botball.h>

void turn_left_90();

int main()
{
  turn_left_90();
  return 0;
}

void turn_left_90()
{
  while(gmpc(0) <= 1350)
  {
    motor(0,100);
    motor(3,0);
  }
  ao();
}
```

The **function prototype** is a *promise* to the computer…

… that you will tell the computer *what* to do in the **function definition**.

Neither the function prototype nor the function definition tell the computer *when* to use the function. That is the job of the function call…

#Botball

Writing Custom Functions

```
include <kipr/botball.h>

void turn_left_90();

int main()
{
  turn_left_90();
  return 0;
}

void turn_left_90()
{
  while(gmpc(0) <= 1350)
  {
    motor(0,100);
    motor(3,0);
  }
  ao();
}
```

The function call makes the computer jump down to the function definition.

The program then executes all of the lines of code in the block of code.

After the computer executes all of the lines of code in the function definition, the program jumps back up to the line of code *after* the function call and continues.

Writing Custom Functions

```
// function prototypes
void turn_left();
void turn_right();
```

```
int main()
{
  turn_left(); // turn_left function call
  turn_right(); // turn_right function call
  return 0;
}
```

```
void turn_left() // turn_left function definition
{
 while(gmpc(0) <= 1350)
  {
    motor(0,100);
    motor(3,0);
  }
  ao();
}
```

```
void turn_right() // turn_right function definition
{
 while(gmpc(3) <= 1350)
  {
    motor(3,100);
    motor(0,0);
  }
  ao();
}
```

Mechanical Design

- At times you may have noticed that you solved problems not through modifying your code but rather by making changes to the mechanical design of your robot(s).
- The next couple slides provide some examples
- Additional resources may be found on the team home base and online
  - For example a great intro to Lego® technic design patterns can be found at:

http://handyboard.com/oldhb/techdocs/artoflego.pdf

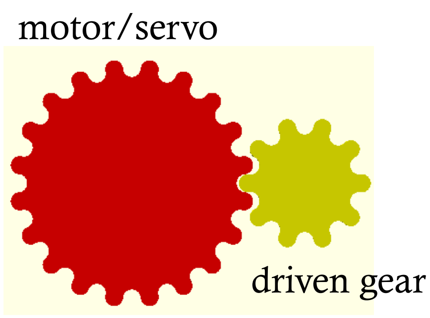
Counterbalance

- Motors and servos have limited power
- Struggling to lift a structure?
  - Use coins as a counterbalance

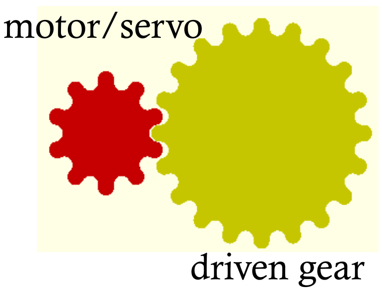
Gearing and Gear Trains

By "combining" gears into a "gear train", using gears of varying sizes you can INCREASE or DECREASE the speed and power (torque) of the end effectors connected to your motors!

- If your motor gear is **<u>larger</u>** than the next gear in the "gear train" the "driven gear" spins FASTER but at the expense of LESS torque (power).



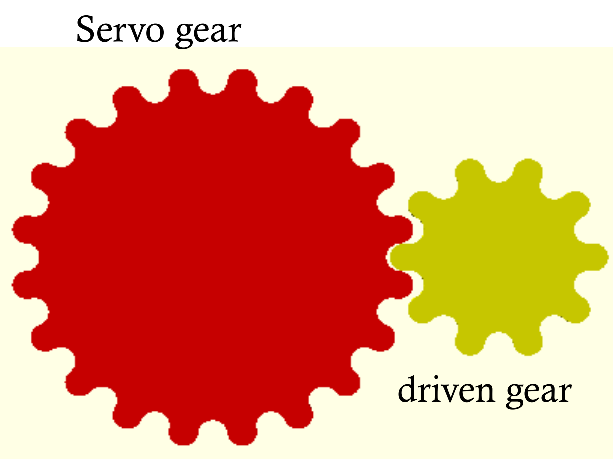
- If your motor gear is **<u>smaller</u>** than your next gear in the "gear train" the "driven gear" spins SLOWER but with MORE torque (power).



#Botball

Gears to Increase Servo Range

- If you attach a larger gear to your servo spline and the next gear in the "gear train" is smaller the range of the servo is increased
  - If the driven gear has ½ # of teeth as the servo gear you double (x2) the range of the servo (now 360 degrees instead of 180 degrees) but with less torque.

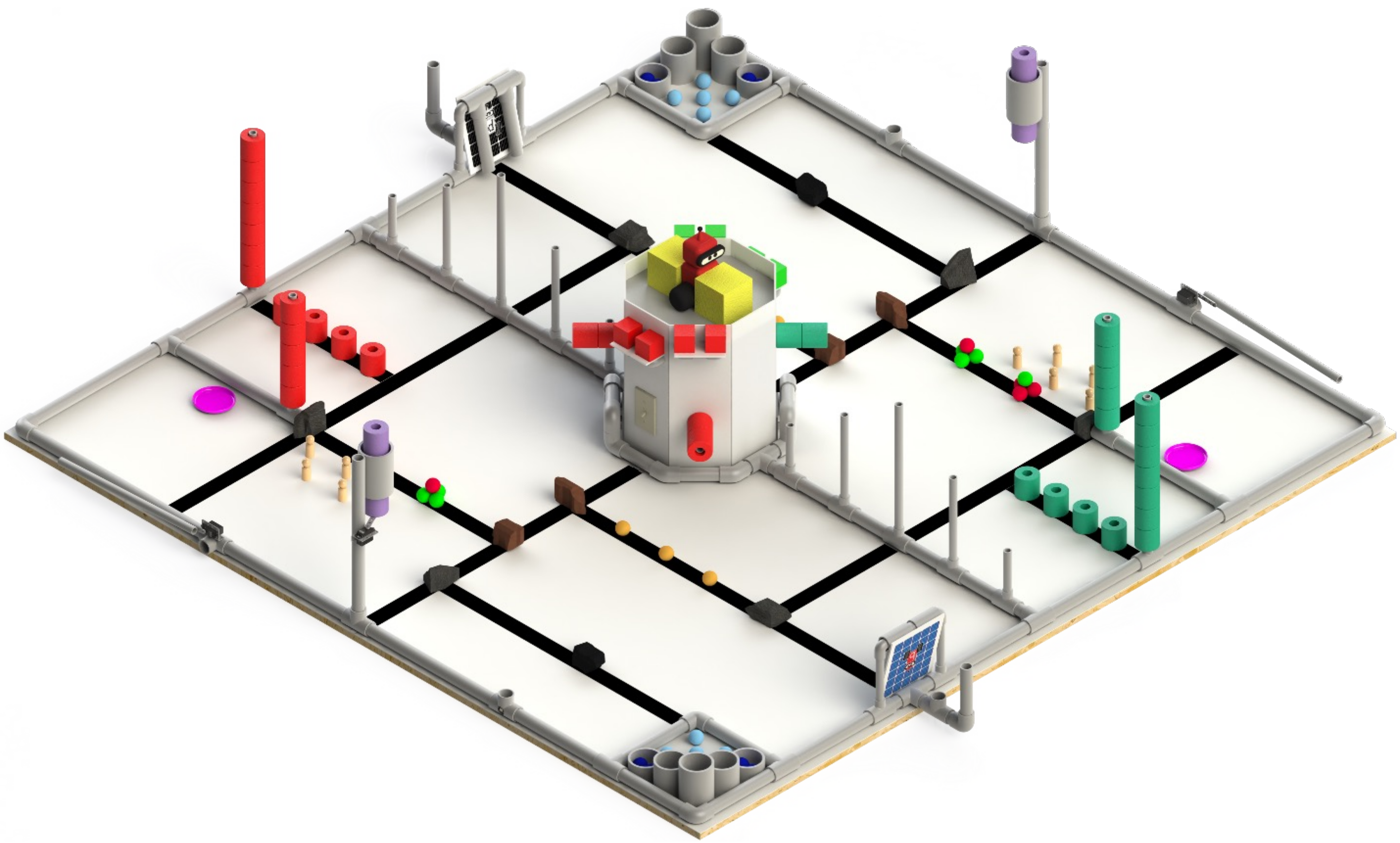


Resources and Support

Team Home Base
Social Media
T-shirts and Awards
What to do After the Workshop

Botball Team Home Base

Found at www.kipr.org



#Botball®

Botball Team Home Base

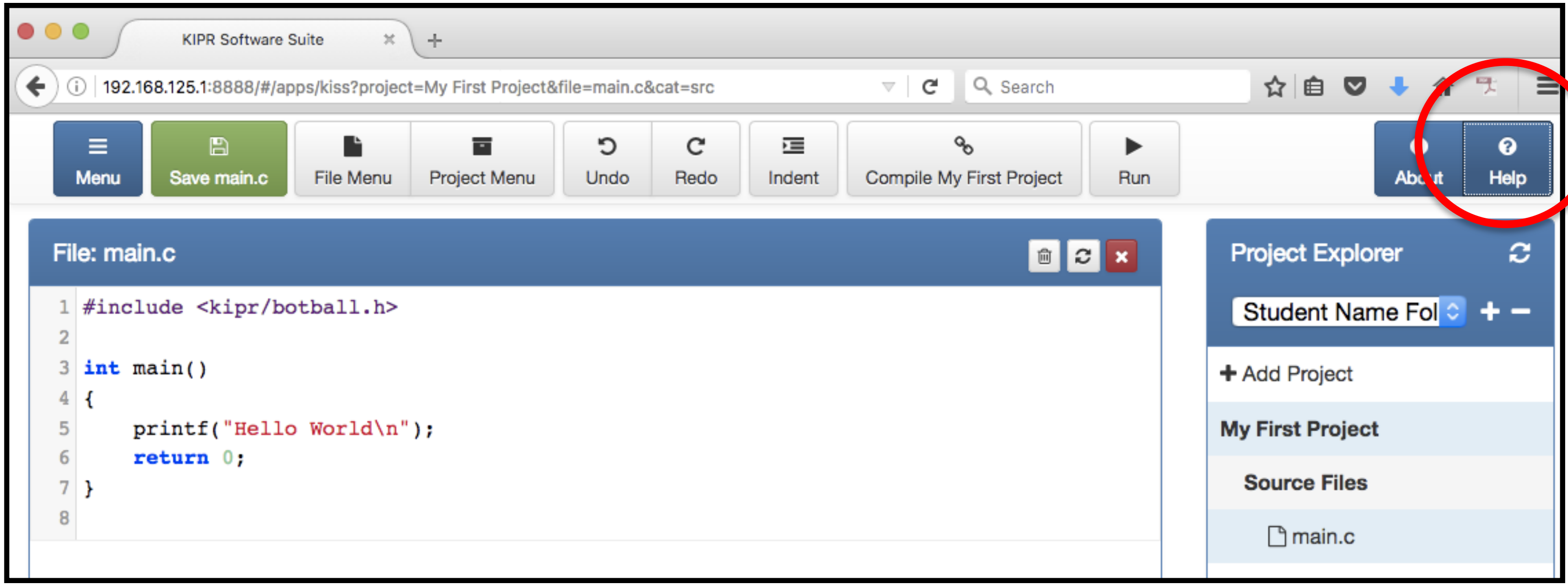
KIPR Support

- **E-mail:** support@kipr.org
- **Phone:** **405-579-4609**
- **Hours:** M-F, 8:30am-5:00pm CT

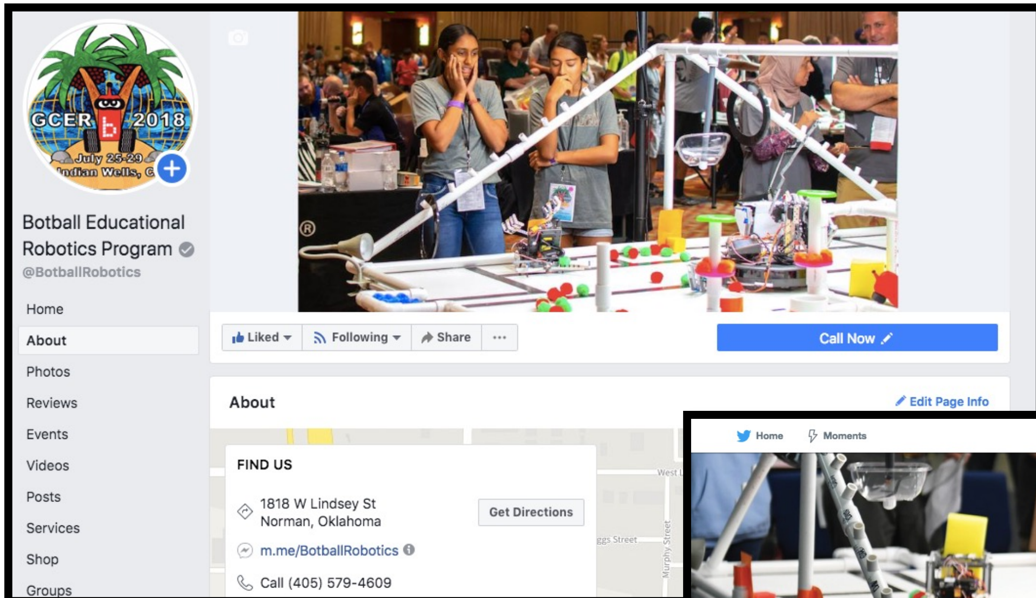
Forum and FAQ

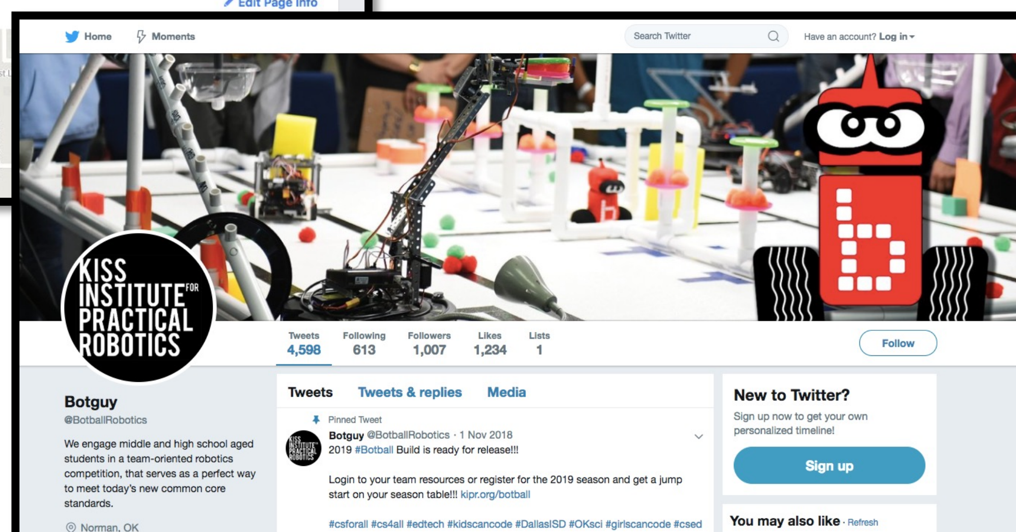
- **Site:** www.kipr.org/Botball
- **Content:**
  - Botball Curriculum
  - Botball Challenge Activities
  - Documentation Manual and Examples
  - Presentation Rubric & Example Presentation
  - DemoBot Build Instructions & Parts List
  - Controller Getting Started Manual
  - Construction Examples
  - Hints for New Teams
  - Game Table Construction Documents
  - All 2024 Game Documents

Access the Wombat documentation by selecting the *Help* button in the KISS IDE



Social Media





Social Media

#Botball®

Tournament Awards

Professional Development Workshop

#Botball®

Tournament Awards

There are a lot of opportunities for teams to win awards!

- Tournament Awards
  - Outstanding Documentation
  - Seeding Rounds
  - Double Elimination
  - Overall (includes Documentation, Seeding, and Double Elimination)

- Judges' Choice Awards (# of awards depends on # of teams)
  - KISS Award
  - Spirit of Botball
  - Outstanding Engineering
  - Outstanding Software
  - Spirit
  - Outstanding Design/Strategy/Teamwork

What to Do After the Workshop

1. Recruit Team Members

If you haven't already recruited team members you can use the materials from the workshop to show to interested students.

2. Hit the Ground Running

- Do not wait to get started—time is of the essence!
- You only have a limited build time before the tournament.
- The workshop will still be fresh in your mind if you start now.
- Plan on meeting sometime during the **first week** after the workshop.

What to Do After the Workshop

3. Plan Out the Season

- Students will not inherently know how to manage their time. Let's face it—it is difficult for many adults!
- Mark a calendar or make a Gantt chart with important dates:
  - 1st online documentation submission due
  - 2nd online documentation submission due
  - 3rd online documentation submission due
  - Tournament date
- Set dates and schedules for team meetings.
- Plan on meeting a **minimum** of 4 hours per week.

What to Do After the Workshop

4. Build the Game Board

- If you can't build the *full* game board, you can build ½ of the board.
- You could tape the outline of the board onto a floor if you have the right type of flooring.

5. Organize your Botball Kit

- Organized parts can lead to faster and easier construction of robots.

6. Understand the Game

- Go over this with your students on the **<u>first meeting</u>** after the workshop.

#Botball®