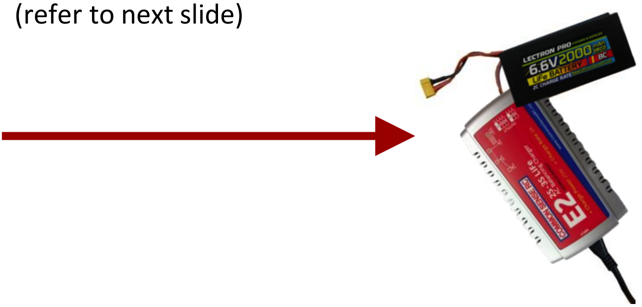
Welcome to Botball 2019!

Before we get started…

1. **Sign in**, and collect the workshop materials and electronics.
2. KIPR staff may come around and **install/copy files** as needed.
3. Charge the Wallaby batteries: WHITE to WHITE (refer to next slide)

KIPR Robotics Controller - Wallaby



4. Open the "**2019 Parts List**" folder, which contains files that list all of the Botball robot kit components. **Please go through the lists and verify that everything has been received.**
5. Go through the **team** packet.
6. **Review slides 2-31.**
7. Build the **non-Create DemoBot**.

Raise your hand if you need help or have questions.

Charging the Controller’s Battery

- For charging the controller’s battery, **use only the power supply which came with the controller**.
  - **It is possible to damage the battery by using the wrong charger or excessive discharge!**

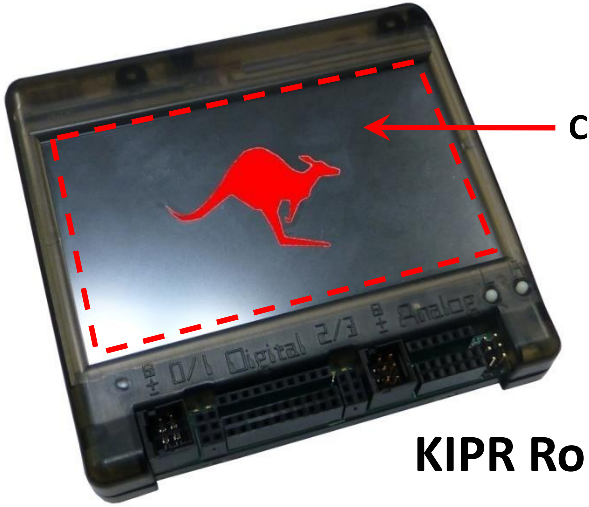
- The standard power pack is a **lithium iron (LiFe) battery**, a safer alternative to lithium polymer batteries. The safety rules applicable for recharging any battery still apply:
  - **Do <u>NOT</u> leave the battery unattended** while charging.
  - Charge in a cool, open area away from flammable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

Making the Connection

All connections are as follows:

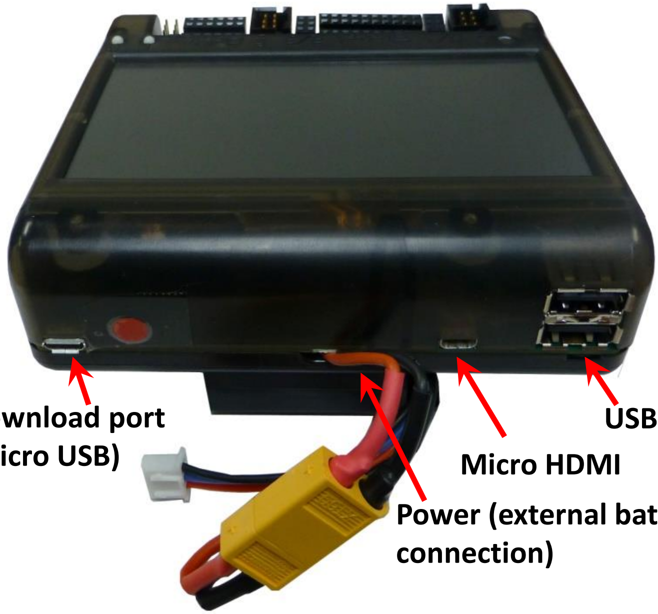
- **Yellow to Yellow** (battery to controller)
- **White small to White small** (charger to battery)
  - The charger may vary slightly, use caution unplugging
- **Black to Black** (motors, servos, sensors)

KRC Wallaby Controller Guide



Color Touch Screen

KIPR Robotics Controller Wallaby

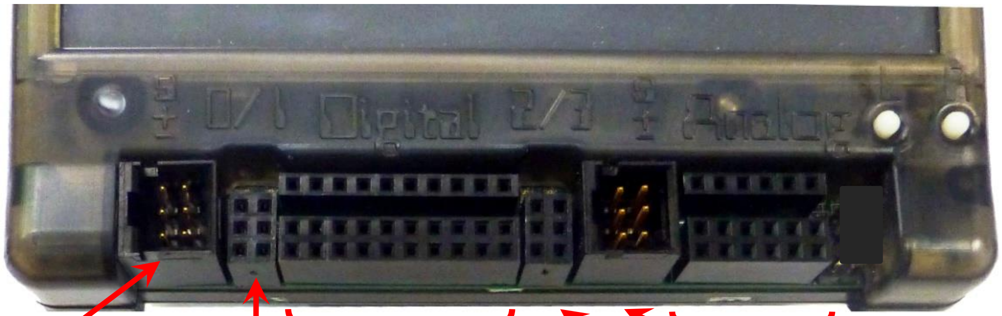


Download port (micro USB)

USB

Micro HDMI

Power (external battery connection)



6 Analog Sensor Ports (Port # 0 - 5)

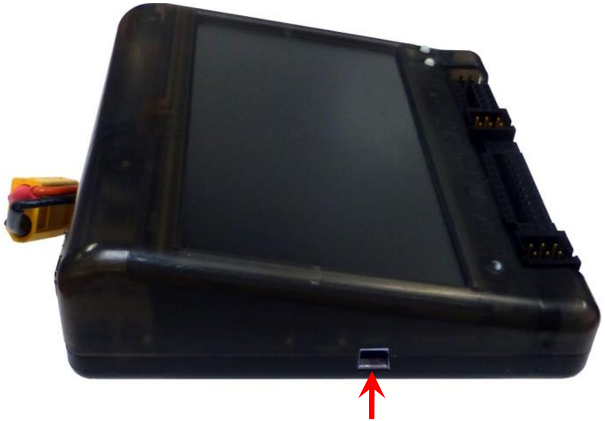
2 Servo Motor Ports (Port # 0 & 1)

2 Motor Ports (Port # 0 & 1)

10 Digital Sensor Ports (Port # 0 - 9)

2 Motor Ports (Port # 2 & 3)

2 Servo Motor Ports (Port # 2 & 3)



Power Switch

Wallaby Power

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

- Make sure to follow the shutdown instruction on the next slide. Failure to do so will drain the battery to the point where it can no longer be charged. If the blue lights continues to flash when the battery is plugged into the charger, then it was probably drained to the point where it cannot be charged again. A replacement battery can be purchased from www.KIPR.org

Wallaby Power Down

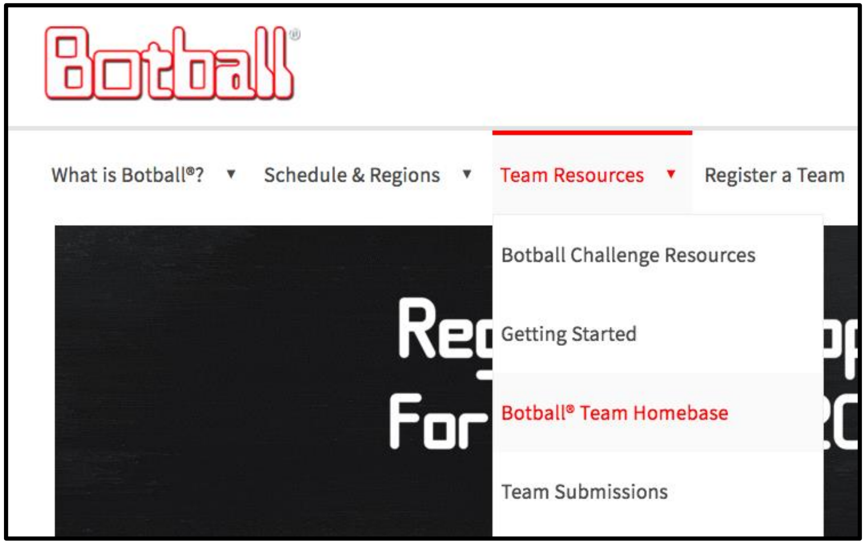
- From the Wallaby Home Screen press *Shutdown*
  - Select *Yes*
- Go to the Wallaby screen and check to see if it is halted
  - If the Wallaby shows shutdown failed, then rerun the last program either to completion or just start and stop it. This should allow the Wallaby to shut down.
- Slide the power switch to off AND unplug the battery, using the yellow connectors, being careful not to pull on the wires

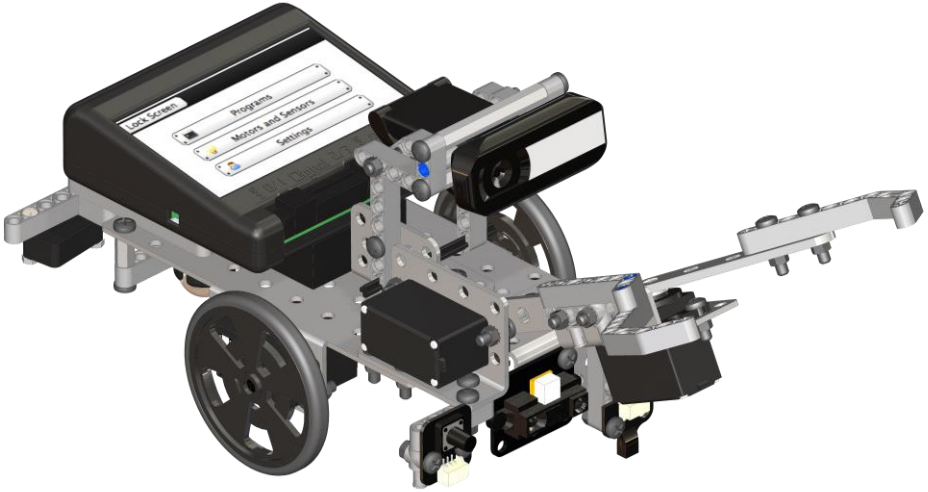
Build the DemoBots

Build the robot using the DemoBot Building Guide

This can be found from the provided materials. Also accessible via the Team Homebase:

www.KIPR.org/Botball -> Team Resources -> Team Homebase





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

Botball 2019

Professional Development Workshop

Prepared by the KISS Institute for Practical Robotics (KIPR)
with significant contributions from KIPR staff
and the Botball Instructors Summit participants

V2019

Thank You for Participating!

We couldn't do it without you!

KISS INSTITUTE FOR PRACTICAL ROBOTICS

KIPR's mission is to:

- Improve the public's understanding of science, technology, engineering, and math;
- Develop the skills, character, and aspirations of students; and
- Contribute to the enrichment of our school systems, communities, and the nation.

Housekeeping

- **Introductions:** workshop staff and volunteers
- **Bathrooms**
- **Food:** lunch is on your own
- **Workshop schedule:** 2 days

Workshop Schedule

Day 1

- Botball Overview
- Getting started with the KIPR Software Suite
- Explaining the “Hello, World!” C Program
- Designing A Program
- Moving the DemoBot with Motors
- Moving the DemoBot Servos
- Making Smarter Robots with Sensors
- Introduction to while Loops
- Measuring Distance
- Motor Position Counter
- Fun with Functions
- Making a Choice
- Line-following
- Homework

Day 2

- Botball Game Review
- Starting with a Light
- Tournament Code Template
- More Variables and Functions with Arguments
- Moving the iRobot *Create*: Part 1
- Moving the iRobot *Create*: Part 2
- iRobot *Create* Sensors
- Color Camera
- Logical Operators
- Resources and Support

Thanks to our National Sponsors!

NASA

COMMON SENSE RC

igus®

iRobot®

3DS SOLIDWORKS

Join Us Online!

Empowering Educator
Impacting Students

KIPR.org

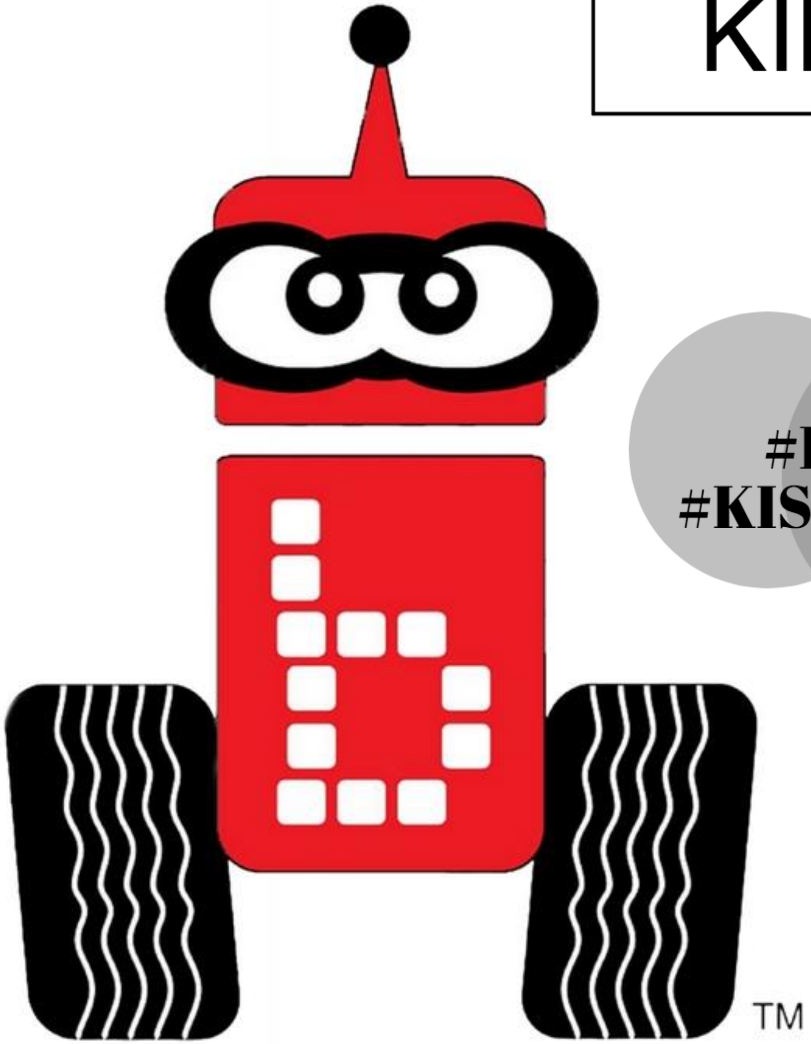
Like & Follow Us Today!

🐦 @botballrobotics

👍 /BotballRobotics

👻 @botballrobotics

📷 @botballrobotics



#Botball
#KISSInstitute

Thanks to our Regional Hosts!

NASA

OU

UMASS
LOWELL

UNIVERSITY OF
MARYLAND

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

SOUTHERN ILLINOIS UNIVERSITY
EDWARDSVILLE

tgm
Vienna Institute of Technology
Austria

Botball Overview

What and When?

GCER and ECER

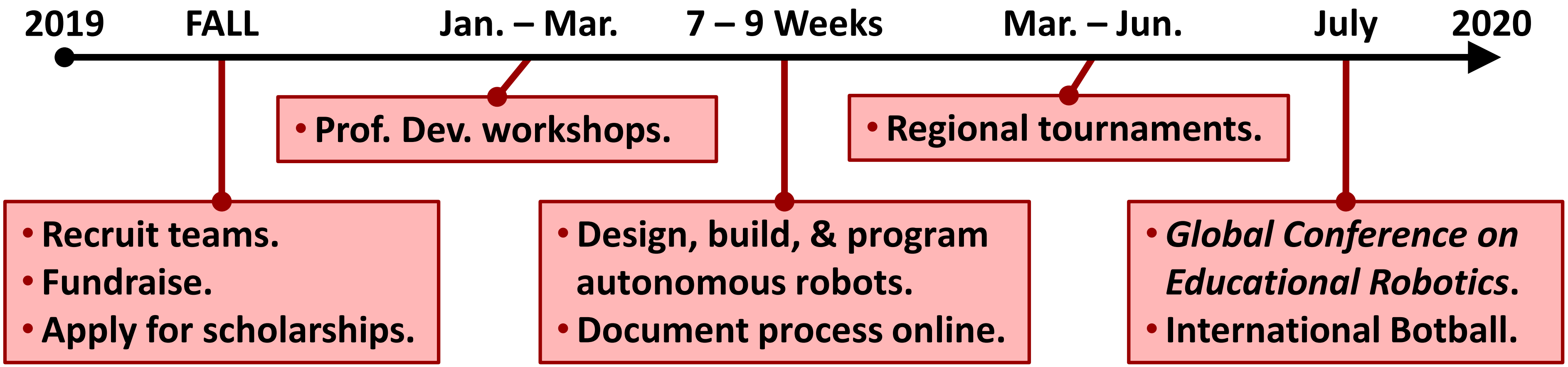
Preview of this year's game

Homework for tonight

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.

When is Botball?



2019 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2020

- Prof. Dev. workshops.

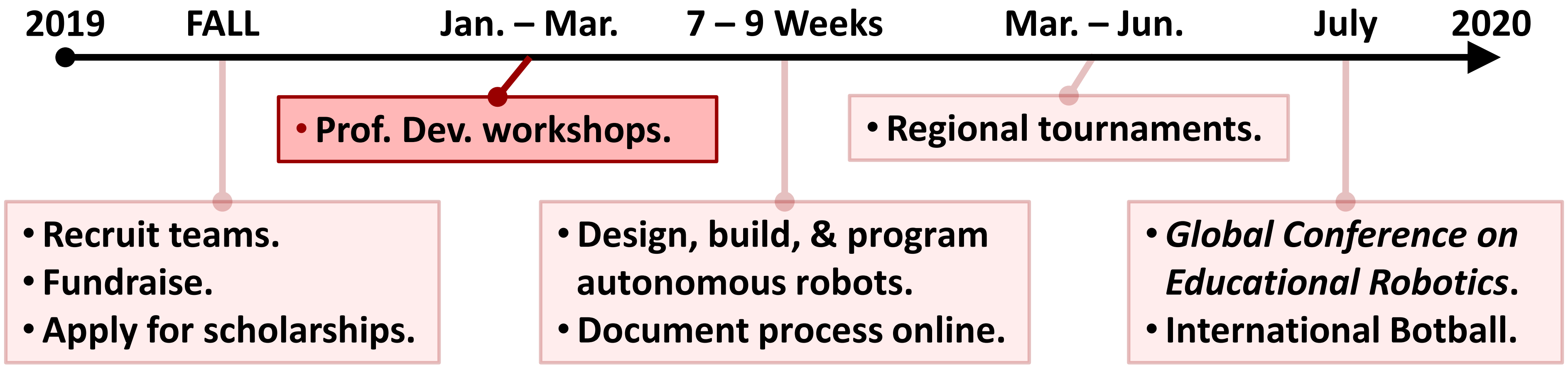
- Regional tournaments.

- Recruit teams.
- Fundraise.
- Apply for scholarships.

- Design, build, & program autonomous robots.
- Document process online.

- *Global Conference on Educational Robotics*.
- International Botball.

When is Botball?



2019 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2020

- Prof. Dev. workshops.

- Regional tournaments.

- Recruit teams.
- Fundraise.
- Apply for scholarships.

- Design, build, & program autonomous robots.
- Document process online.

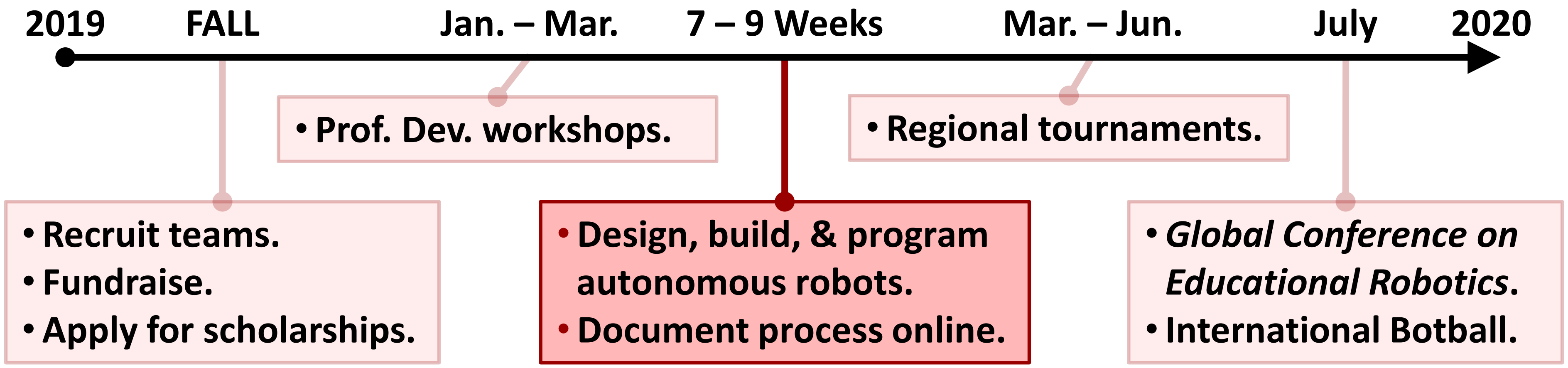
- *Global Conference on Educational Robotics*.
- International Botball.

WE ARE HERE!

- **Provides the skills and tools necessary** to compete in the tournament.
- Teams will learn to program robots, and **will leave with working systems**.
- **Skills and tools/equipment are kept** and are reusable outside of Botball.
- **Not a standalone curriculum!** Goal is to **support team success in Botball**!

(For building and programming resources, visit the **Team Home Base**.)

When is Botball?



2019 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2020

- Prof. Dev. workshops.

- Regional tournaments.

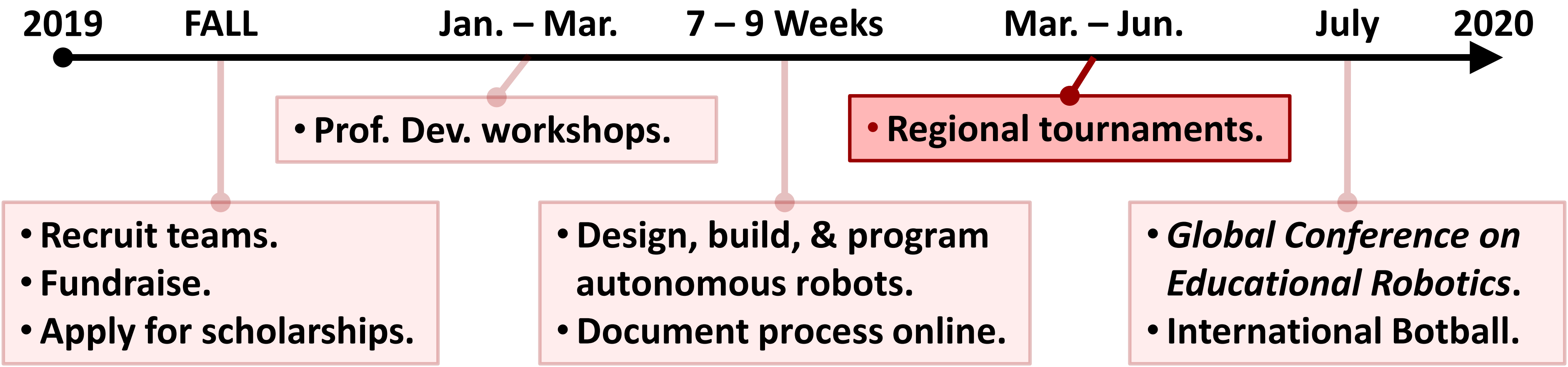
- Recruit teams.
- Fundraise.
- Apply for scholarships.

- Design, build, & program autonomous robots.
- Document process online.

- *Global Conference on Educational Robotics*.
- International Botball.

- Reinforces **computational thinking** and the **engineering design process**.
- Teams must submit three online project documents, **which count for points**.
- **Online support** throughout the season from KIPR and other Botball teams.

When is Botball?



2019

FALL

- Recruit teams.
- Fundraise.
- Apply for scholarships.

Jan. – Mar.

- Prof. Dev. workshops.

7 – 9 Weeks

- Design, build, & program autonomous robots.
- Document process online.

Mar. – Jun.

- Regional tournaments.

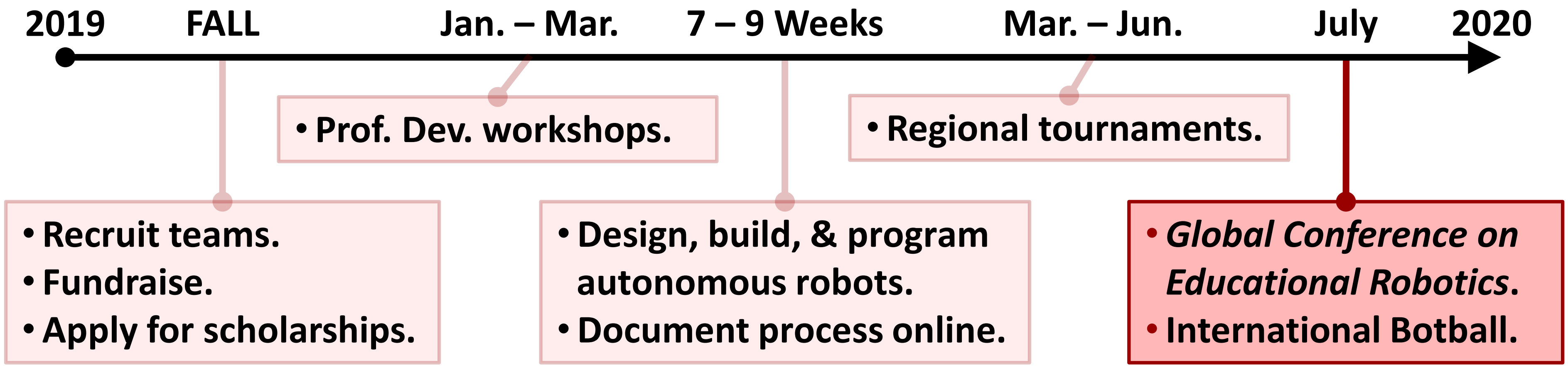
July

- *Global Conference on Educational Robotics*.
- International Botball.

2020

- **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

When is Botball?



2019

FALL

- Recruit teams.
- Fundraise.
- Apply for scholarships.

Jan. – Mar.

- Prof. Dev. workshops.

7 – 9 Weeks

- Design, build, & program autonomous robots.
- Document process online.

Mar. – Jun.

- Regional tournaments.

July

- *Global Conference on Educational Robotics.*
- International Botball.

2020

Global Conference on Educational Robotics (GCER)

- **International Botball Tournament:** all teams are invited to participate
- **Paper presentations:** students may submit and present papers at GCER
- **Guest speakers:** presentations from academic and industry leaders
- **Autonomous showcase:** students display projects in a science fair style

EVERYONE IS ELIGIBLE!

GCER 2019

Global Conference on Educational Robotics

- Norman, Oklahoma
- July 7-11, 2019
- International Botball Tournament
- Autonomous Robotics Showcase
- Autonomous Aerial Tournament
- International Junior Botball Challenge
- Meet and network with students from around the country and world
- Talks by internationally recognized robotics experts
- Teacher, student, and peer reviewed track sessions

www.KIPR.org

#Botball®

GCER 2019

Global Conference on Educational Robotics

Preconference classes on July 6th

International Junior Botball Challenge

Preconference Workshops

Autonomous Aerial Robot Competition & Challenges

ECER 2019

European Conference on Educational Robotics

- Vienna, Austria
- April 8-12, 2019

- European Botball Competition
- Talks by Researchers and Students

Practical
Robotics
Institute
Austria

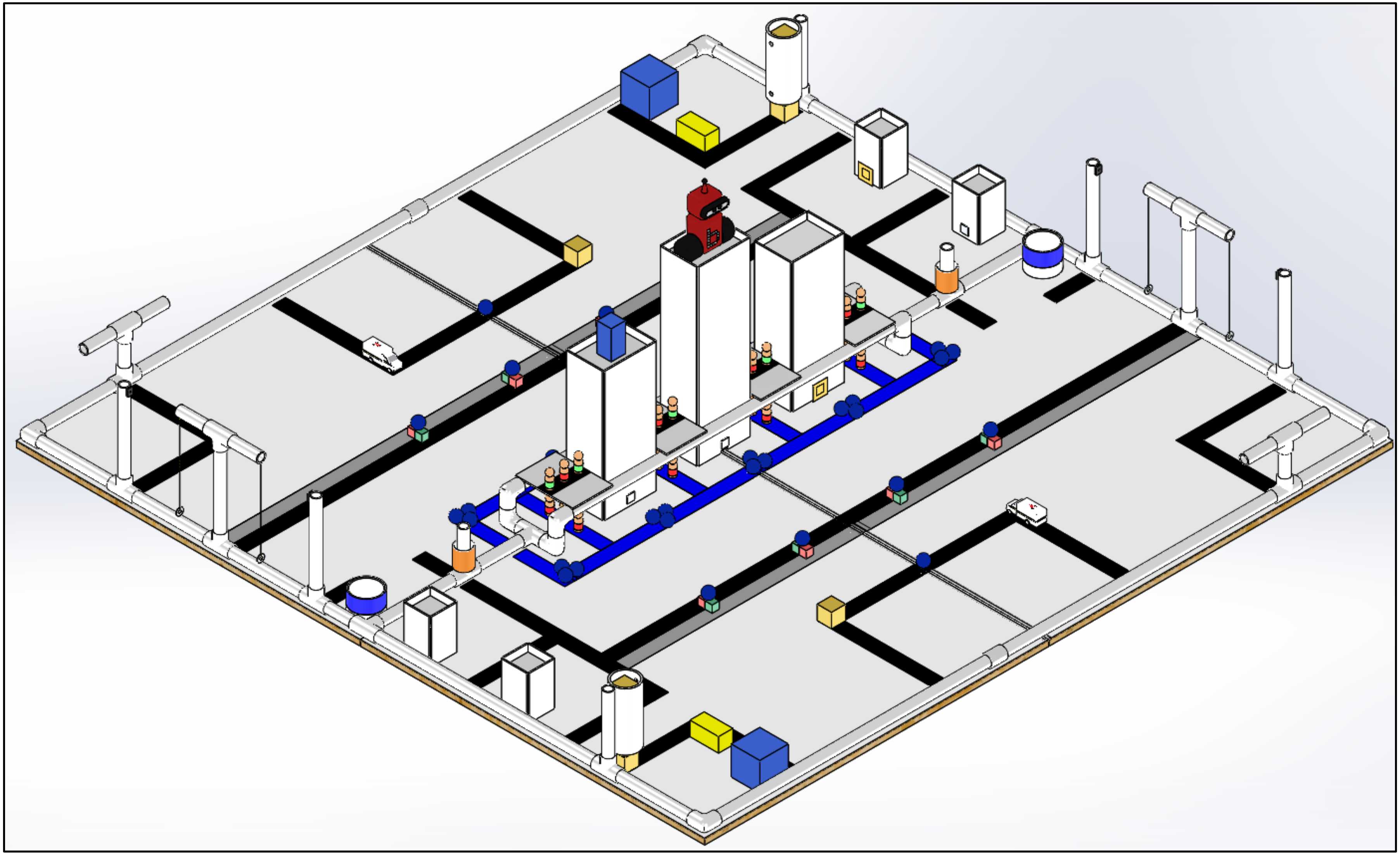
www.pria.at

2019 Botball Theme

Botguy Directs Disaster Relief Efforts

Disaster has struck Botguy's home in the form of a massive storm! Lightning has started numerous structure fires and flooding has left citizens of Botopia stranded. High winds and isolated tornadoes have caused damage to structures resulting in widespread power outages and serious natural gas leaks. Your team must work with *Botguy* and the *Mayor of Botopia* in the *Disaster Relief Zone* to ensure that emergency vehicles and personnel are dispatched to fight fires and rescue citizens. Transport the *Injured Citizens* to the hospital to receive much needed treatment and *Uninjured Citizens* should be taken to the *Disaster Relief Zone*. Get your firefighters to the buildings that are on fire and douse those buildings with water. Assist the utility crews by shutting off the natural gas and restore electrical service to the downtown area. *Water*, *Food*, and *Medical Supplies* need to be collected and taken to the *Disaster Relief Zone* and *Medical Complex*. Now hurry and save the city!

Botball Game Board



Professional Development Workshop
© 1993 – 2019 KIPR

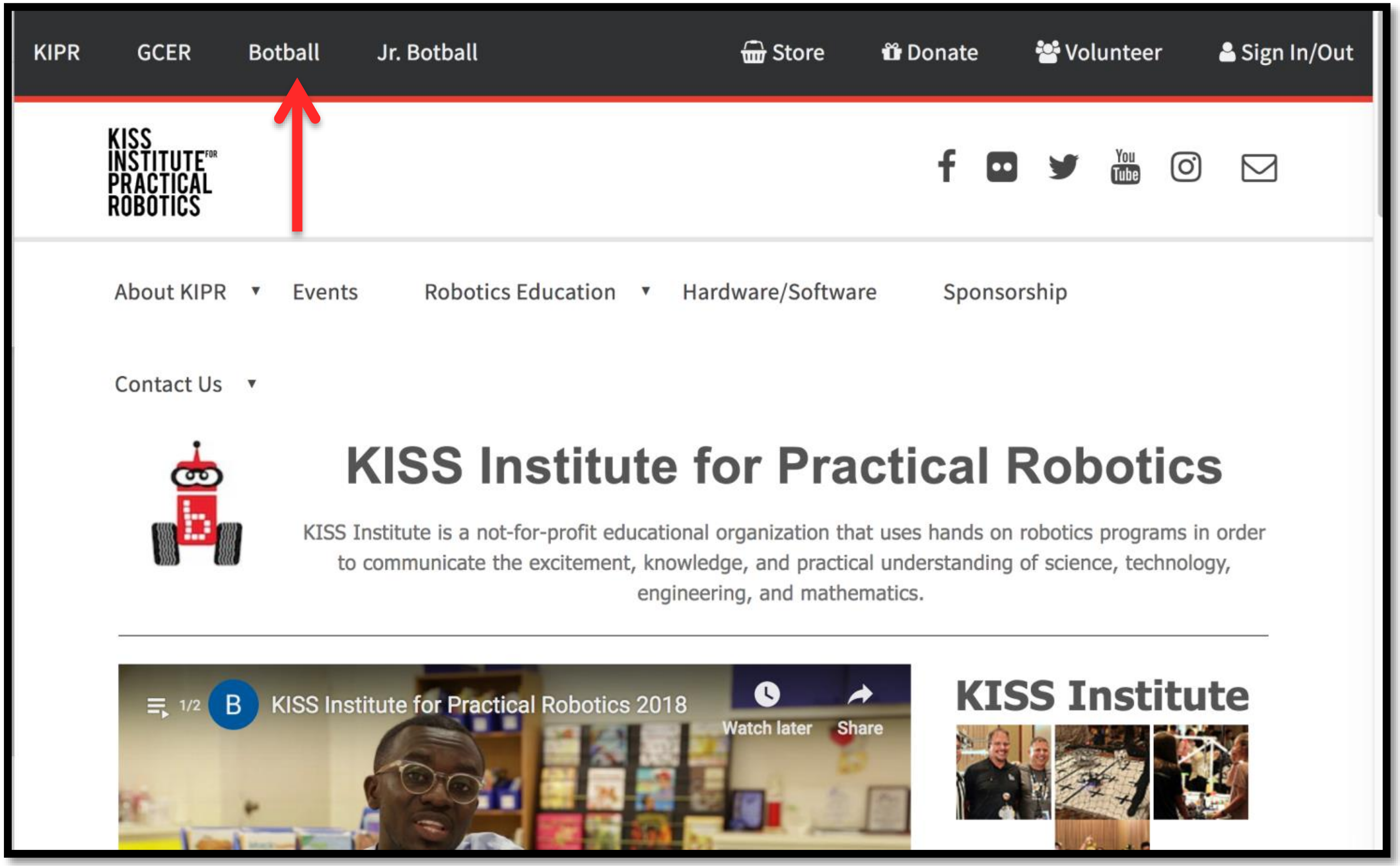
Homework for Tonight

Review the game rules on the Team Home Base

- We will have a **30-minute Q&A session** tomorrow.
- After the workshop, ask questions about game rules in the **Game Rules FAQ**.
  - Everyone should **regularly visit this forum**.
  - Everyone will **find answers to the game questions** there.

Botball Team Home Base

Found at www.KIPR.org



KISS Institute for Practical Robotics

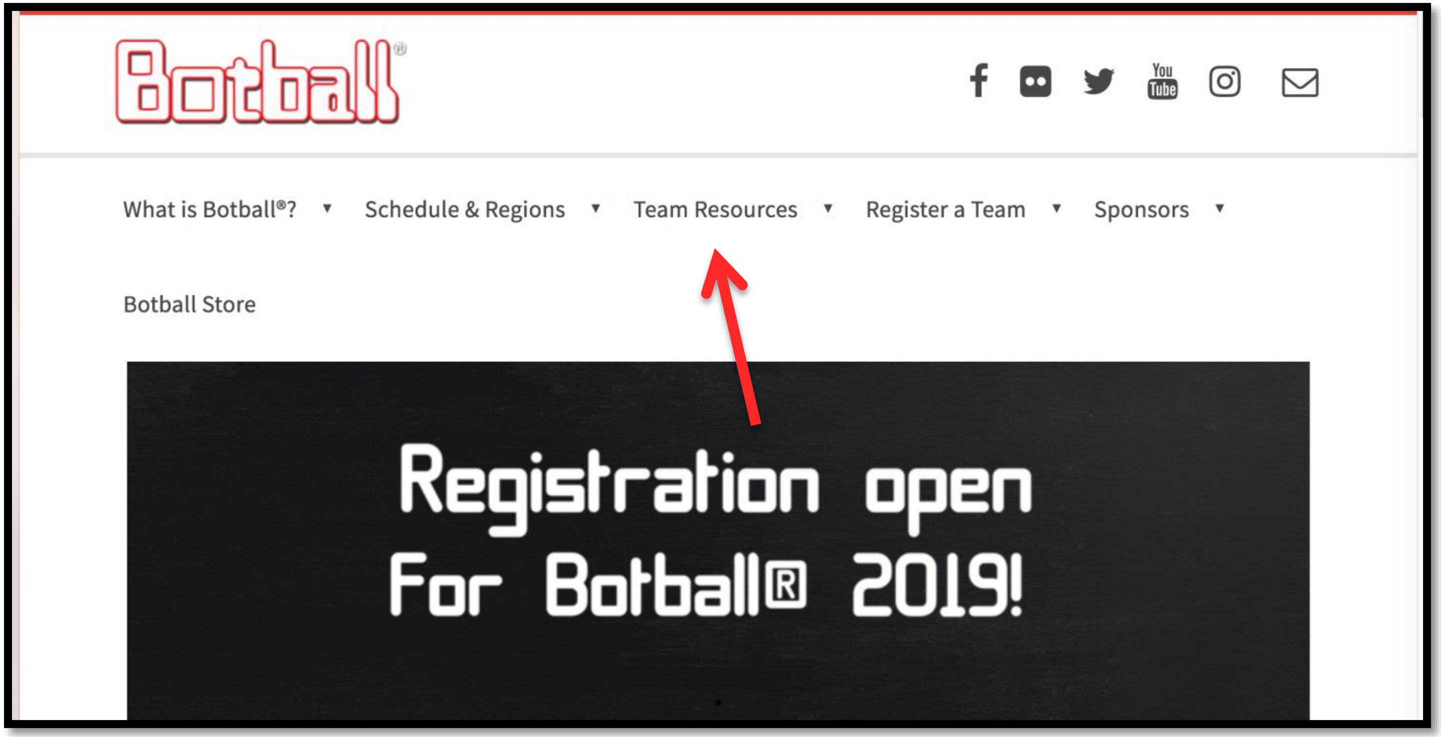
KISS Institute is a not-for-profit educational organization that uses hands on robotics programs in order to communicate the excitement, knowledge, and practical understanding of science, technology, engineering, and mathematics.

KISS Institute for Practical Robotics 2018

KISS Institute

Botball Team Home Base

Found at www.KIPR.org



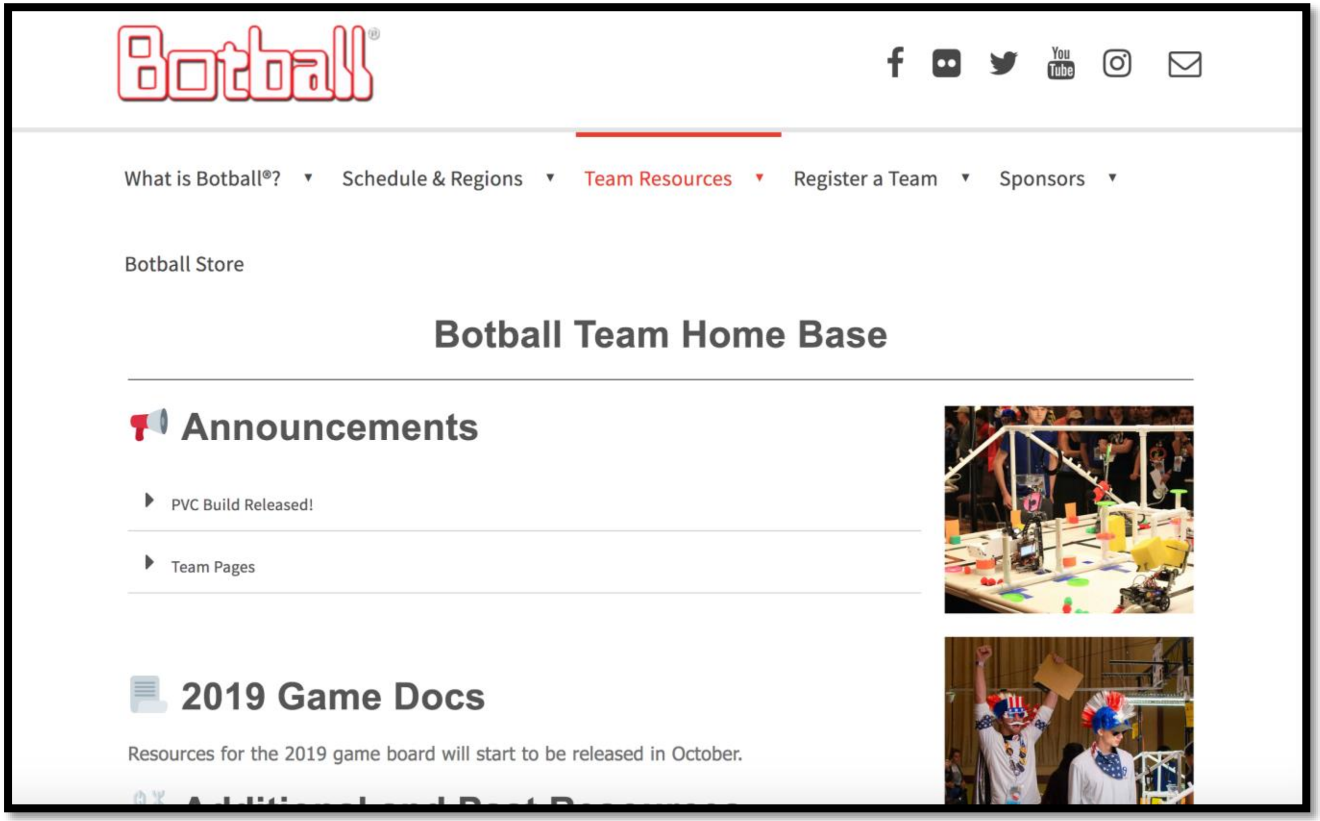
What is Botball®? ▾ Schedule & Regions ▾ Team Resources ▾ Register a Team ▾ Sponsors ▾

Botball Store

Registration open For Botball® 2019!

Botball Team Home Base

Found at www.KIPR.org



What is Botball®? ▾ Schedule & Regions ▾ Team Resources ▾ Register a Team ▾ Sponsors ▾

Botball Store

Botball Team Home Base

Announcements

- PVC Build Released!
- Team Pages

2019 Game Docs

Resources for the 2019 game board will start to be released in October.

Preview of this Year's Botball Game

Hold questions until tomorrow! Game Q&A is tomorrow!

Getting Started with the KIPR Software Suite

What is a programming language?

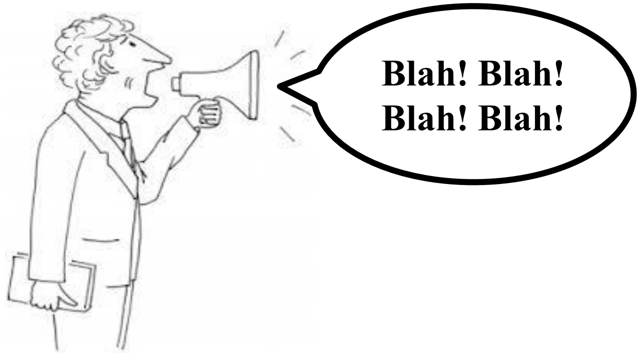
How can I create new projects and files?

How can I write and compile source code?

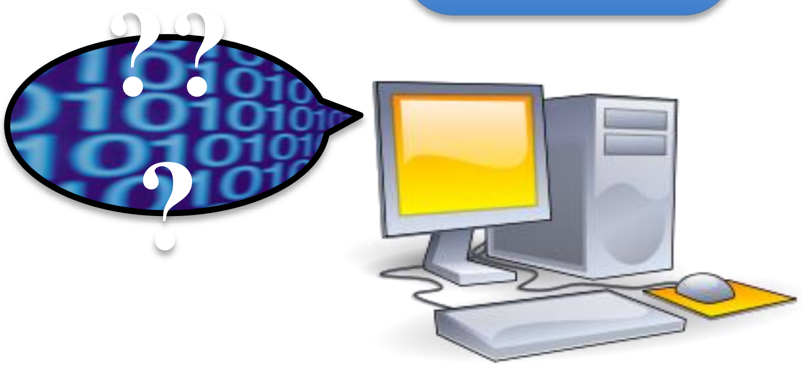
How can I run programs on the KIPR Wallaby?

What is a *Programming Language*?

Human



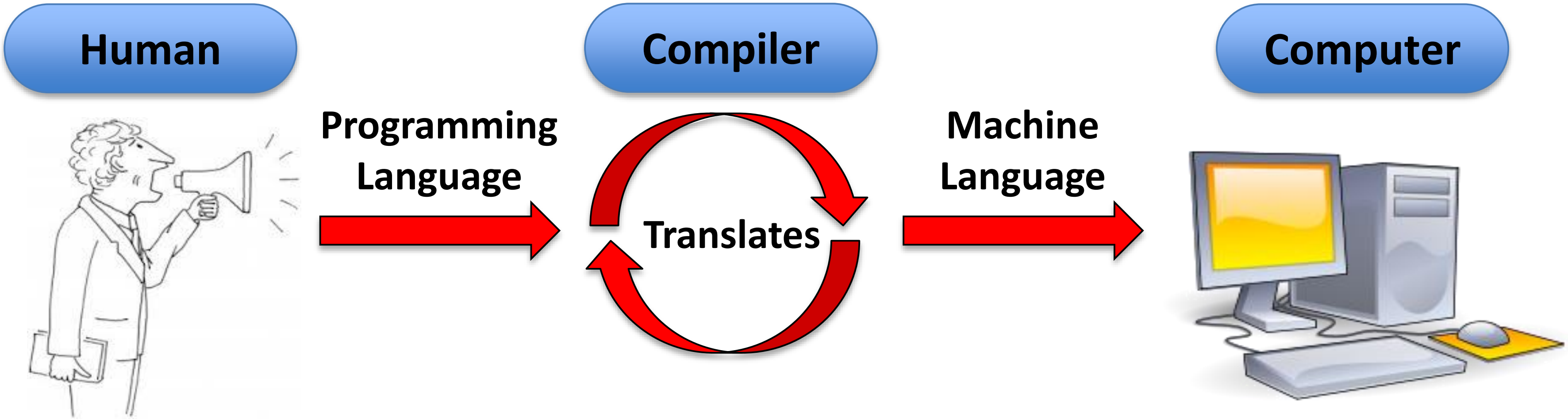
Computer



- **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*?



Human

Programming Language

Compiler

Translates

Machine Language

Computer

- **Humans** have created **programming languages** that allow them (humans) to write “**source code**” that is easier for them (humans) to understand.

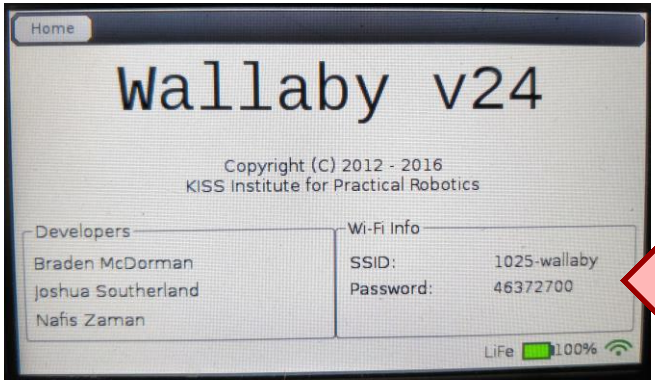
- **Source code** is **compiled** (translated) by a **compiler** (part of the **KIPR Software Suite**) into **machine language** so that the **computer** can **read and execute** (run) the code.

- Programming languages have funny names (C, C++, Java, Python, …)

Connect the Wallaby to the Computer, Smart Phone or Tablet at School

- Connect the **Wallaby** to a device via Wi-Fi
- This is great at home or School
- **Not recommended at Large Workshops or any Tournament**

1. Turn on the Wallaby with the **black switch on the side.** **After turning on, wait at least 3-7 minutes for Wallaby to completely boot. Skipping this step frequently results in connection issues.**



Home

Wallaby v24

Copyright (C) 2012 - 2016
KISS Institute for Practical Robotics

Developers
Braden McDorman
Joshua Southerland
Nafis Zaman

Wi-Fi Info
SSID: 1025-wallaby
Password: 46372700

LiFe 100%

2. Click the **About** button (top left of screen) and use the Wallaby SSID and Password to connect to it via Wi-Fi.

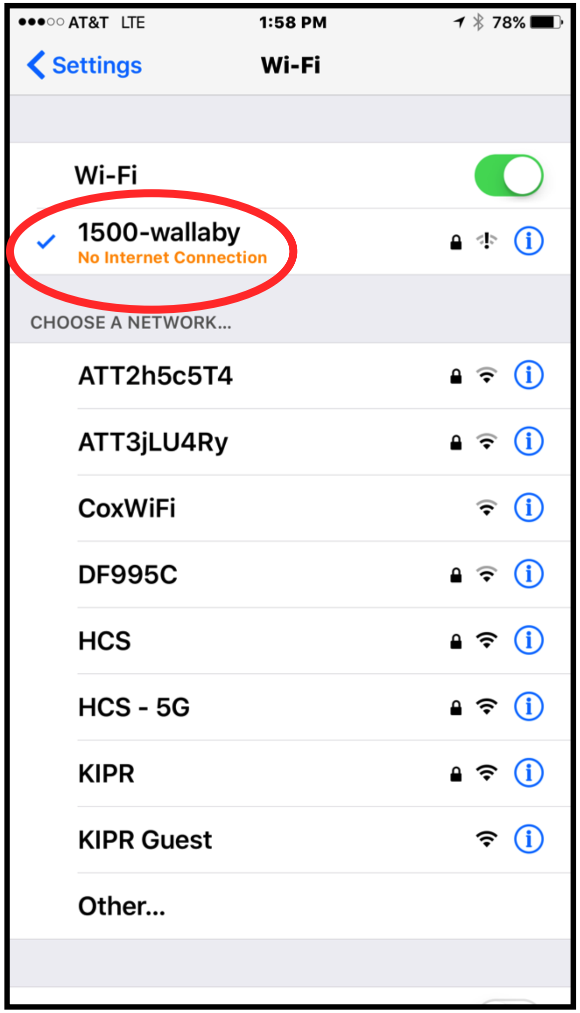
Loading the Starting Web Page (Wi-Fi)

1. Launch a web browser such as Chrome or Firefox (Internet Explorer **will not work**) and power up the Wallaby. Connect to the Wallaby via Wi-Fi.
2. Copy this IP address into the 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 load in the browser.
   a. Note: **during competitions** use the USB cable connection. Use IP address and port: 192.168.124.1:8888
4. A computer, tablet or even a smart phone can be used to interface with the Wallaby.

Connection



●●●○○ AT&T LTE 1:58 PM 78%

Settings Wi-Fi

Wi-Fi

✓ 1500-wallaby
No Internet Connection

CHOOSE A NETWORK...

- ATT2h5c5T4
- ATT3jLU4Ry
- CoxWiFi
- DF995C
- HCS
- HCS - 5G
- KIPR
- KIPR Guest
- Other...

When connected to the Wallaby, the device may give various errors; "***no internet connection***" or "***connected with limited***"

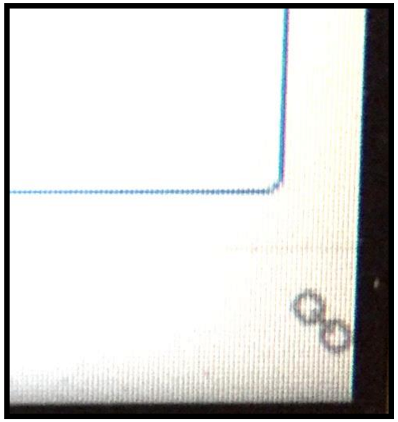
This is normal. Proceed with opening a browser and connecting to the KISS IDE.

Connection Issues

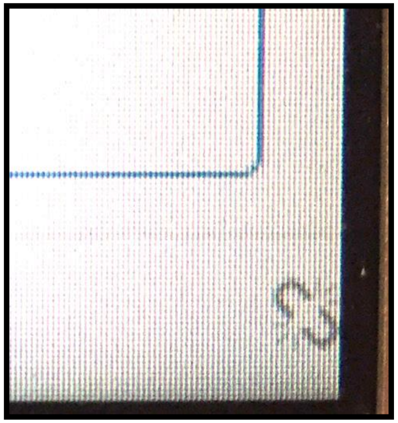
The device may disconnect from the Wallaby. This is evident when trying to compile and the button does not turn red (nothing happens).

In the **bottom right corner** of the KIPR IDE there is an icon that shows if the device is still connected to the Wallaby.

Connected



Not Connected

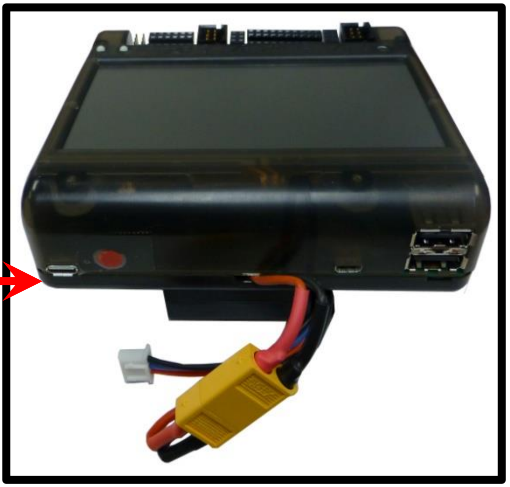


Connect the Wallaby to the device at Workshop and Tournament

- Connect the **Wallaby** to the device using **USB Cable**

1. Plug battery into Wallaby- YELLOW TO YELLOW.
2. Turn on the Wallaby with the **black switch on the side**

Insert the micro-USB end here

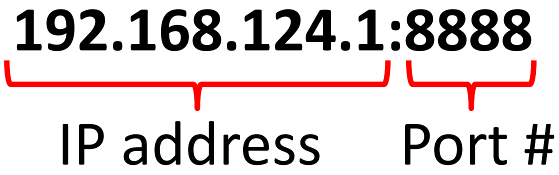


Attach the regular USB end to computer

3. Once the Wallaby has booted, the Wallaby will appear in the list of available Ethernet connections for the device.
4. If there is a message about the driver raise your hand for help or go to the team home base: ***Troubleshooting->USB driver*** for instructions

Loading the Starting Web Page (USB)

1. Launch the web browser (such as Chrome or Firefox, but not Internet Explorer) and power up the Wallaby.
2. Copy this IP address into the browser's address bar followed by ":" and port number 8888; e.g.,



192.168.124.1:8888

IP address

Port #

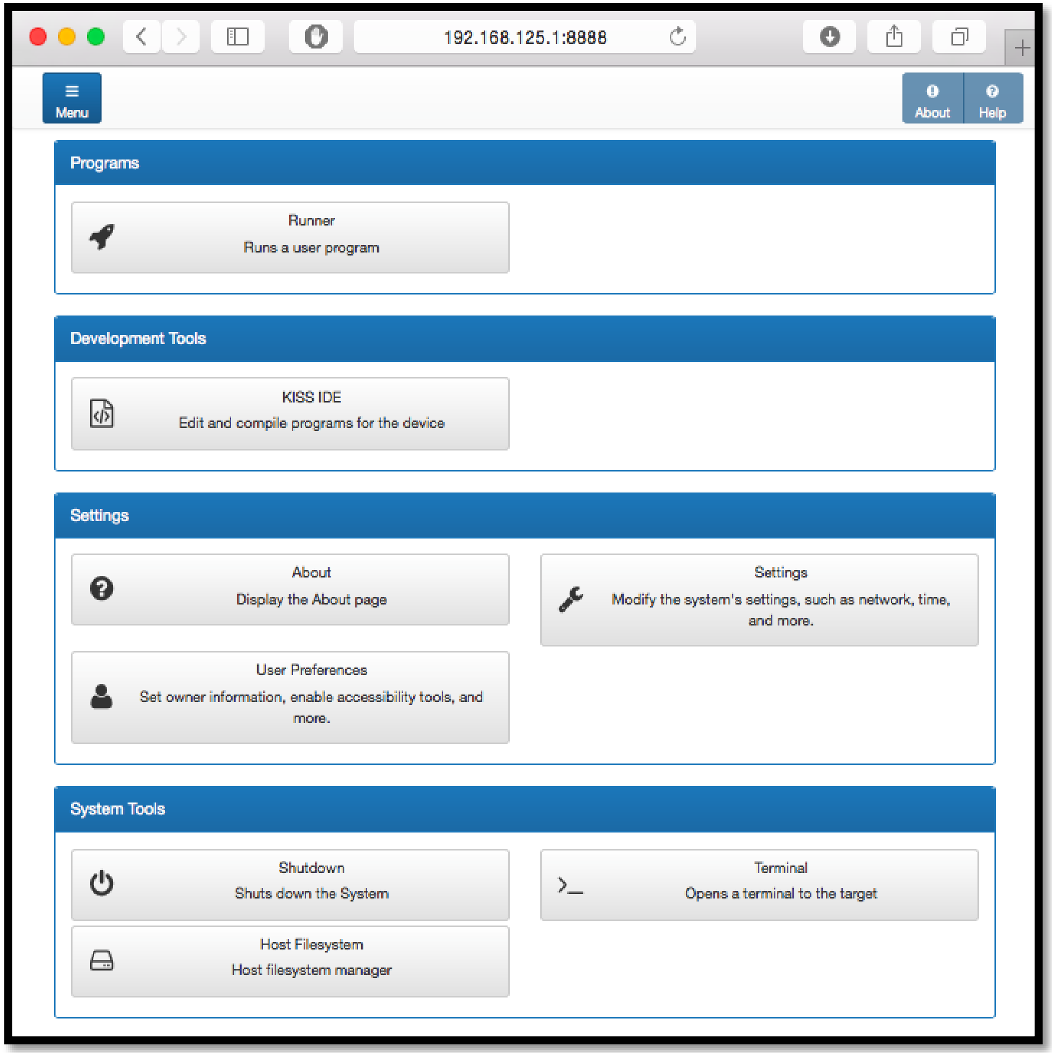
   a. Note that USB cable IP address is 192.168.**124**.1:8888

3. The user interface for the package will now come up in the browser.
4. Test this at the workshop at least once.
   a. See Team Resources -> Botball Team Homebase -> Connecting with USB Cable

Using the KIPR Integrated Development Environment (IDE)

To make it easier to learn and use a programming language, KIPR provides a web-based **Software Suite** which allows writing and compiling source code using the **C programming language**.

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



Programs

Runner
Runs a user program

Development Tools

KISS IDE
Edit and compile programs for the device

Settings

About
Display the About page

Settings
Modify the system's settings, such as network, time, and more.

User Preferences
Set owner information, enable accessibility tools, and more.

System Tools

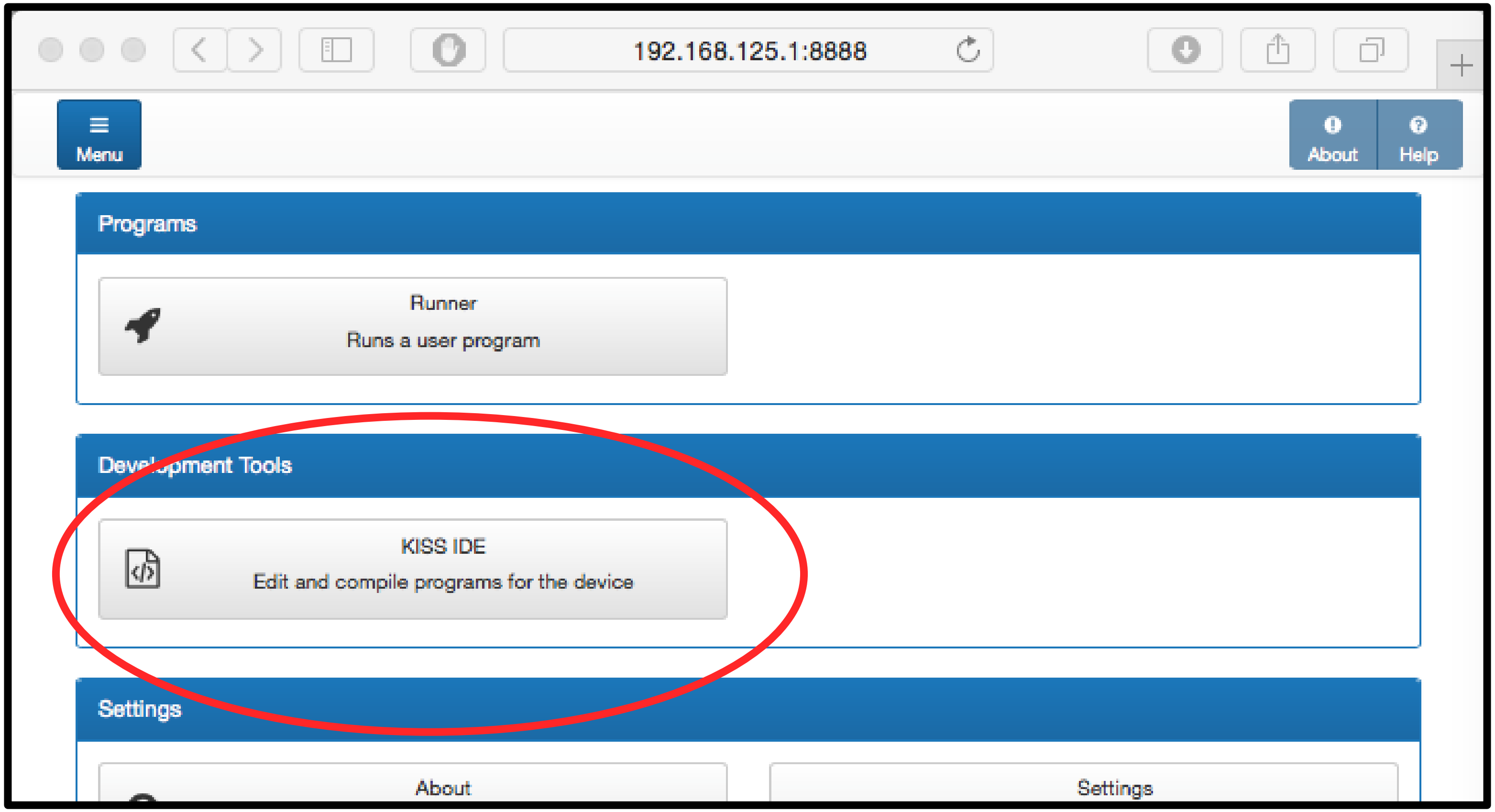
Shutdown
Shuts down the System

Terminal
Opens a terminal to the target

Host Filesystem
Host filesystem manager

Creating a Project

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



192.168.125.1:8888

Programs

Runner
Runs a user program

Development Tools

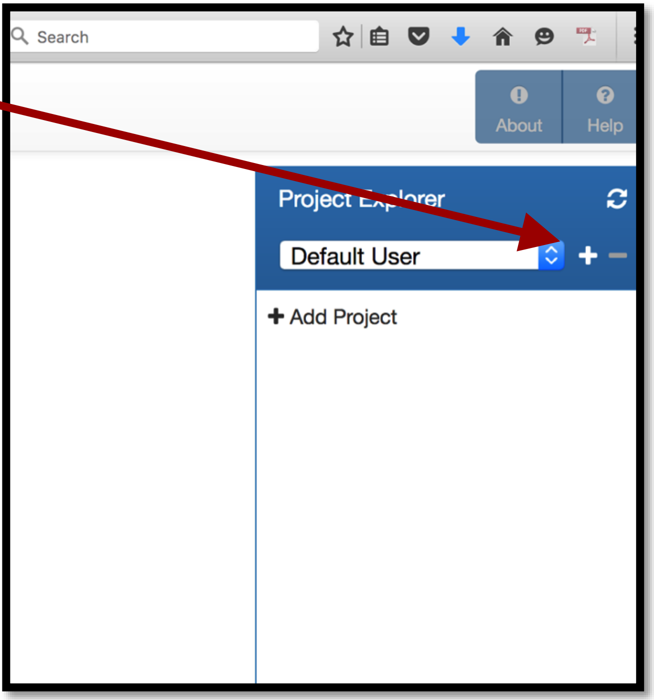
KISS IDE
Edit and compile programs for the device

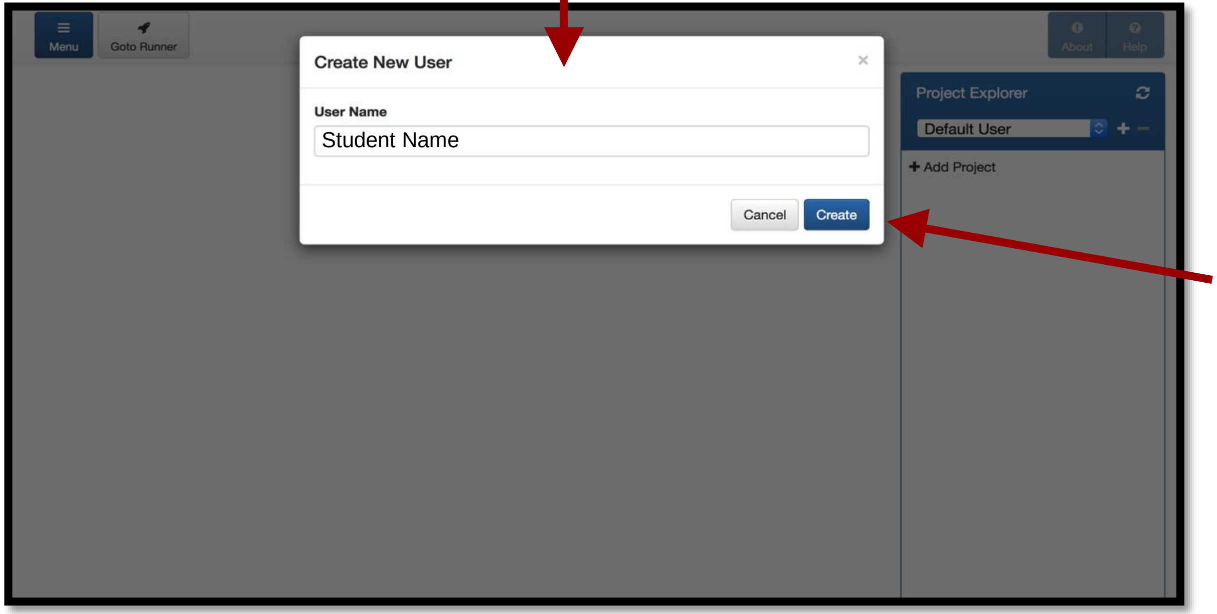
Settings

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

Creating a User Folder

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





Create New User

User Name: Student Name

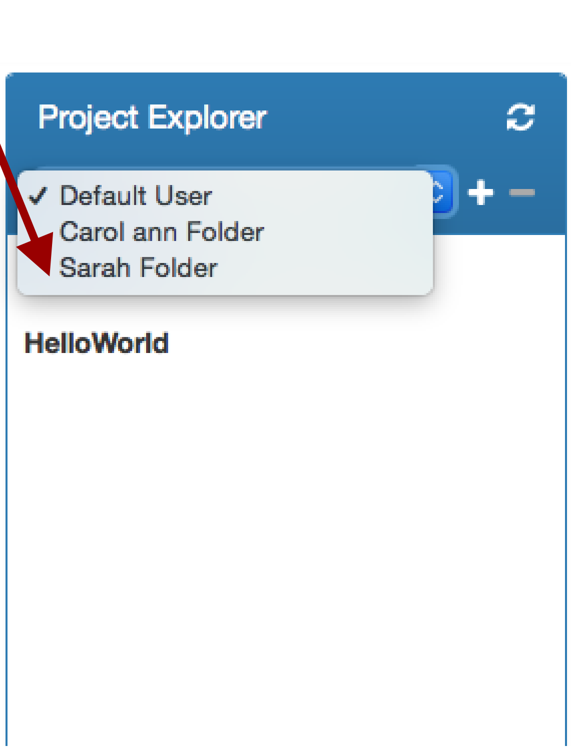
3. Click ***Create*** to complete.

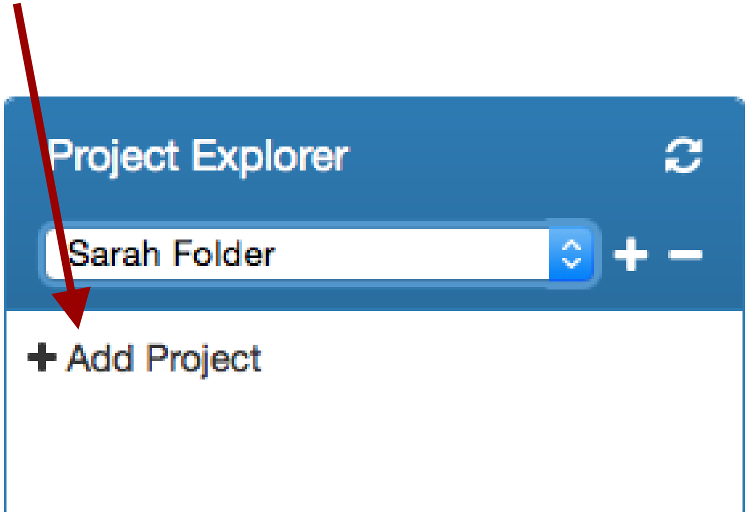
#Botball®

Creating a Project

1. Go back to **Project Explorer** and select the **User Name** created from the drop down.

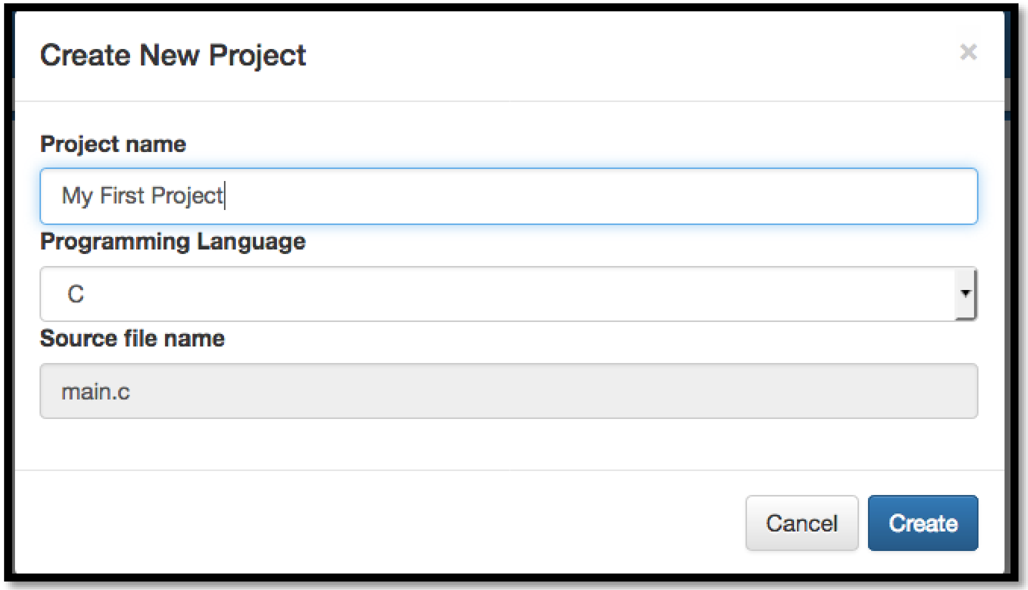
2. Click ***+Add Project***. This adds a project to the folder.





Creating a Project

1. Give the project a **descriptive name**
   - **Note:** there might be a lot of student's projects, so consider using student's first names followed by the name of the activity.
   - ***No special characters allowed in name. (For example: */#@%$.)***
2. Press the ***Create*** button



Create New Project

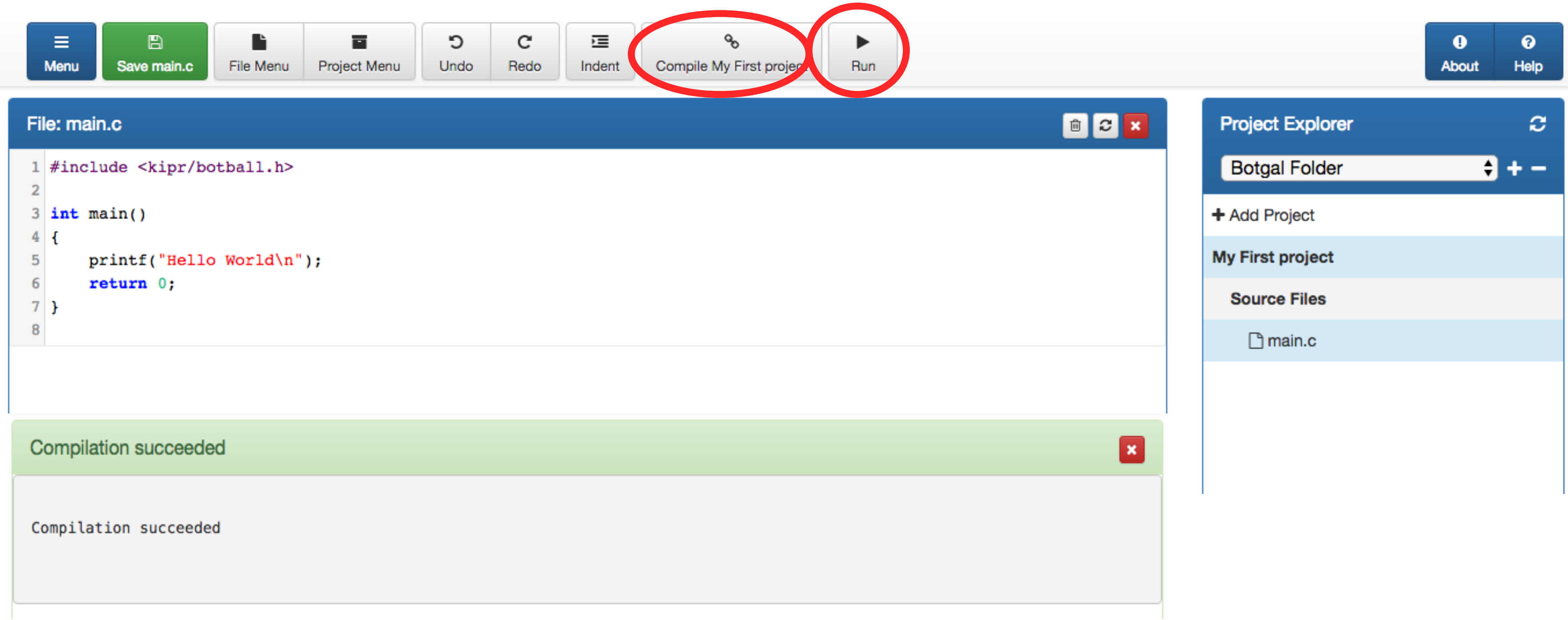
Project name: My First Project

Programming Language: C

Source file name: main.c

Compile and Run a Project

1. Click the ***Compile*** button for the project and, if successful (compilation succeeded), click ***Run*** to execute the project to see if it works.



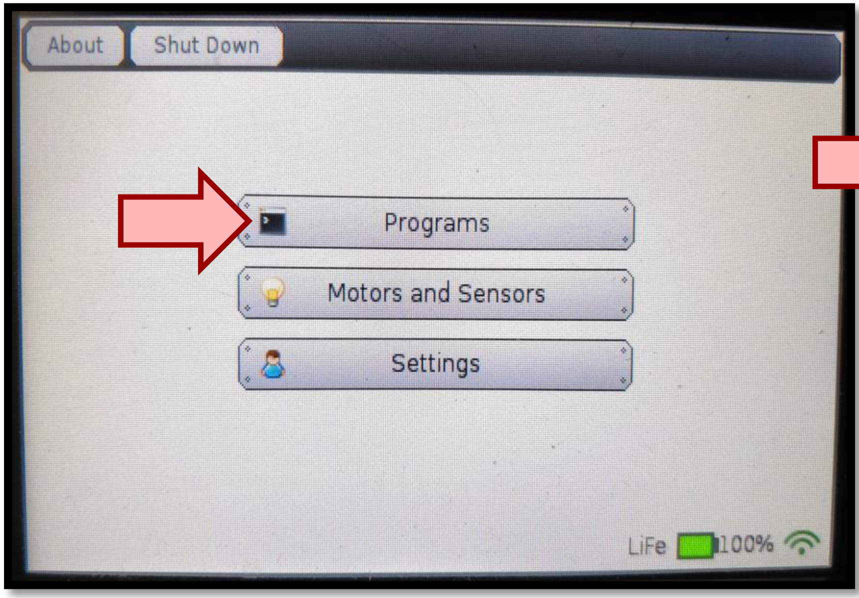
```
#include <kipr/botball.h>

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

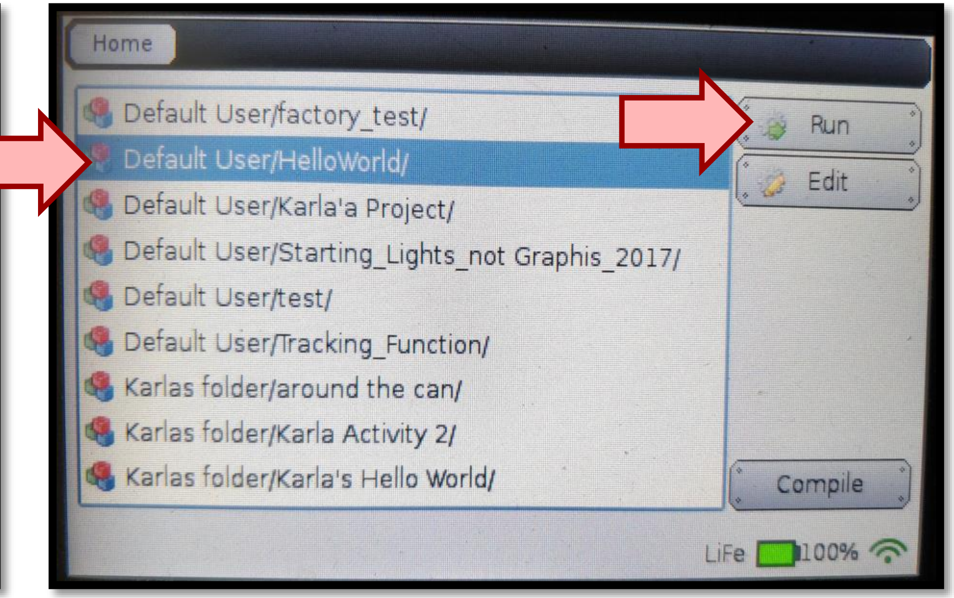
Compilation succeeded

```
Compilation succeeded
```

NOTE: When compiling the project it is automatically saved.

Running Program from Robot



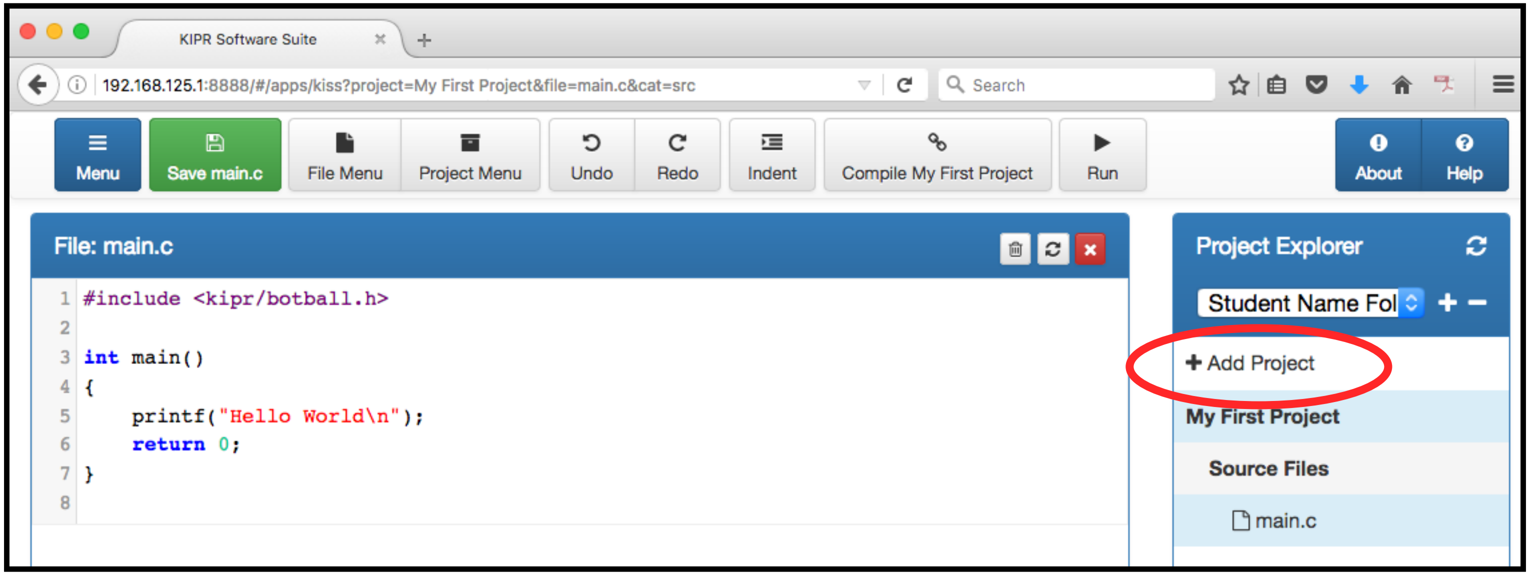

Home
Default User/factory_test/
Default User/HelloWorld/
Default User/Karla'a Project/
Default User/Starting_Lights_not Graphis_2017/
Default User/test/
Default User/Tracking_Function/
Karlas folder/around the can/
Karlas folder/Karla Activity 2/
Karlas folder/Karla's Hello World/
Run
Edit
Compile
LiFe 100%

Highlight program and then press run

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 all of the user folder projects and actively edited files.



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

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

```

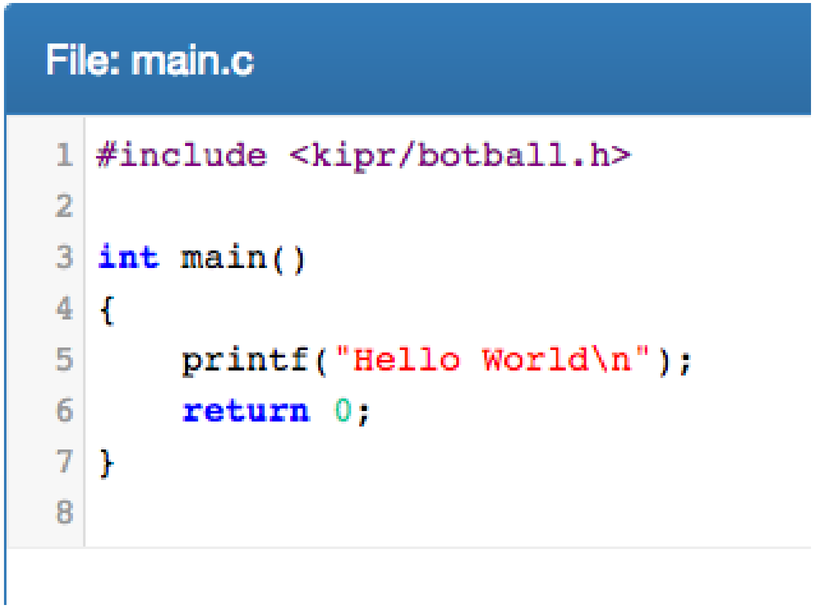
Explaining the “Hello, World!” C Program

Program flow and the main function

Programming statements and functions

Comments

“Hello, World!”



File: main.c

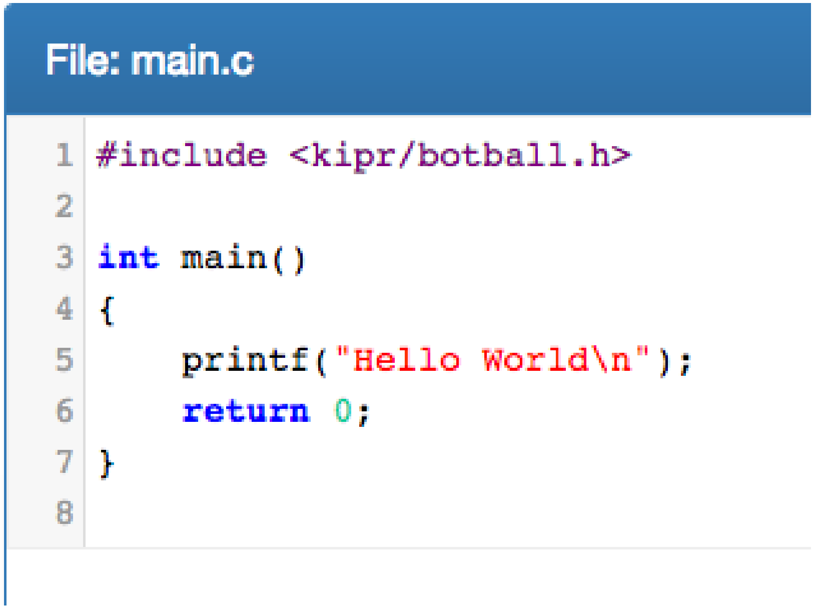
```c
#include <kipr/botball.h>

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

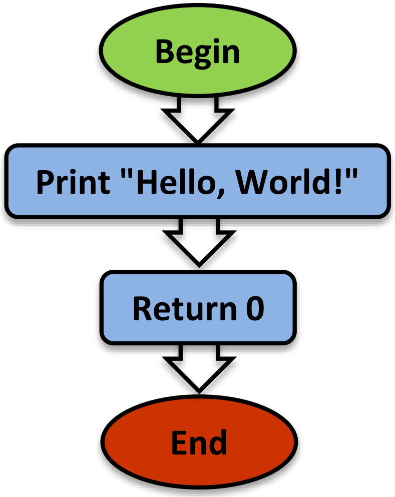
Top

Bottom



File: main.c

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

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

```



Begin

Print "Hello, World!"

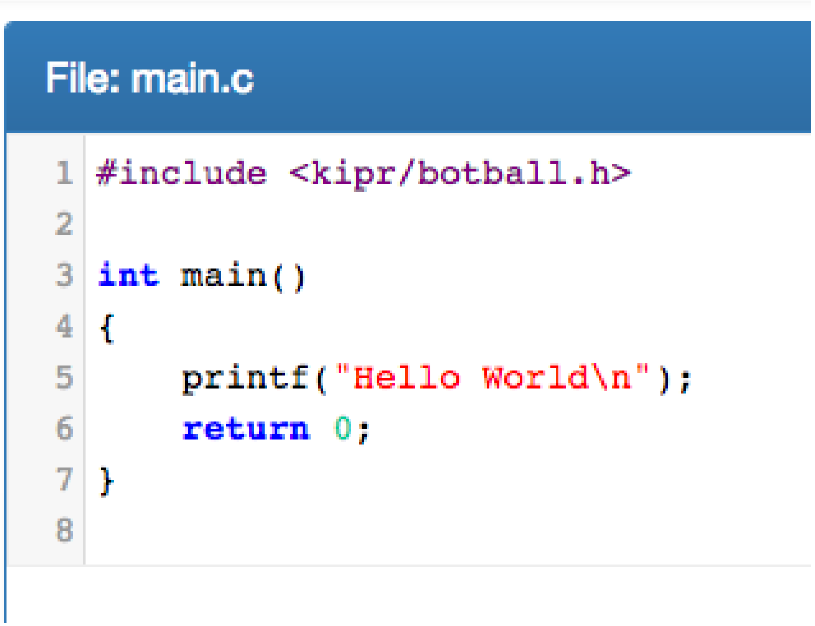
Return 0

End

Computers read a program just like humans 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 a function is **called** (used),
the program follows the instructions and bakes the cake.

```
// Created on Thu January 10 2019

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

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

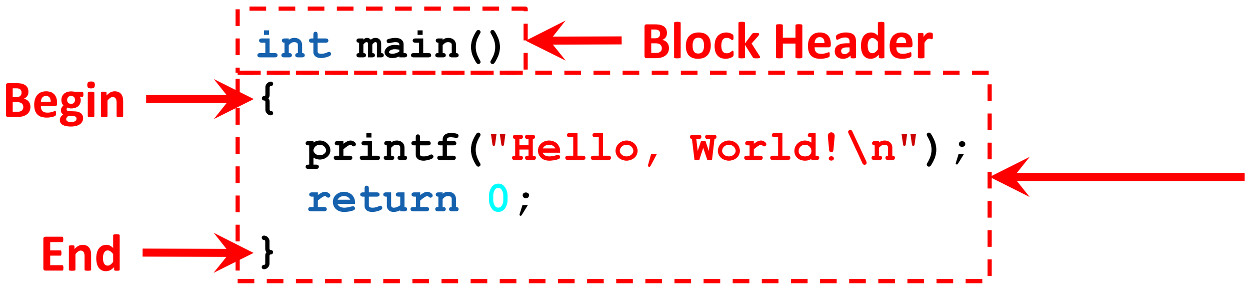
When running the program,
the **main function** is executed.

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

Block of Code

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

```
// Created on Thu January 10 2019
```



Block Header

Begin

End

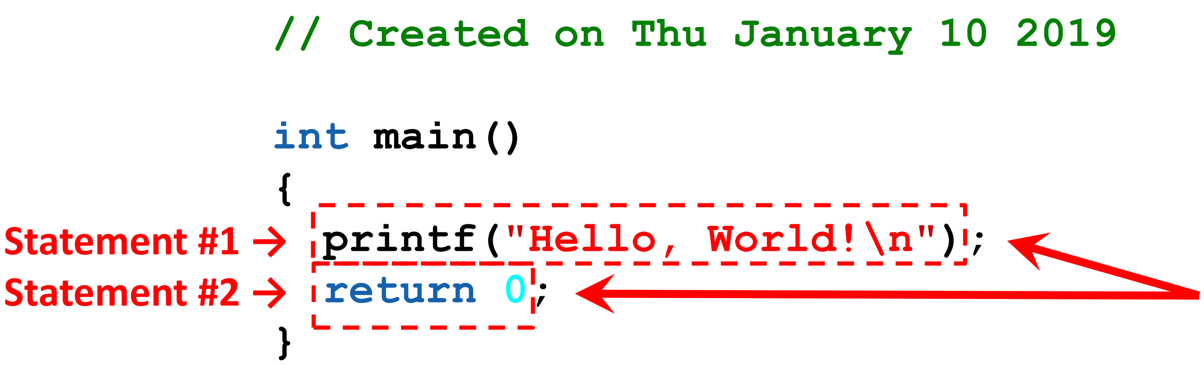
```
int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

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 }

Programming Statements



```
// Created on Thu January 10 2019

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

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**.

KIPR Wallaby functions hint sheet

Use this **cheat/hint sheet** as an easy reference.

Copying and pasting code is also very helpful.

```
printf("text\n");                           // Prints the specified text to the screen
msleep(# milliseconds);                     // Another name for wait_for_milliseconds (identical)
motor(port #, % velocity);                  // Turns on motor with port # at specified % velocity
motor_power(port #, % power);               // Turns on motor with specified port # at specified % power
mav(port #, velocity);                      // Move motor at specified velocity (# ticks per second)
mrp(port #, velocity, position);            // Move motor to specified relative position (in # ticks)
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
wait_for_touch(port #);                     // Waits for touch 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 all motors after specified # of seconds
```

Ending a Programming Statement

```
// Created on Thu January 10 2019

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.

Ending the `main` Function

```
// Created on Thu January 10 2019

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**”.

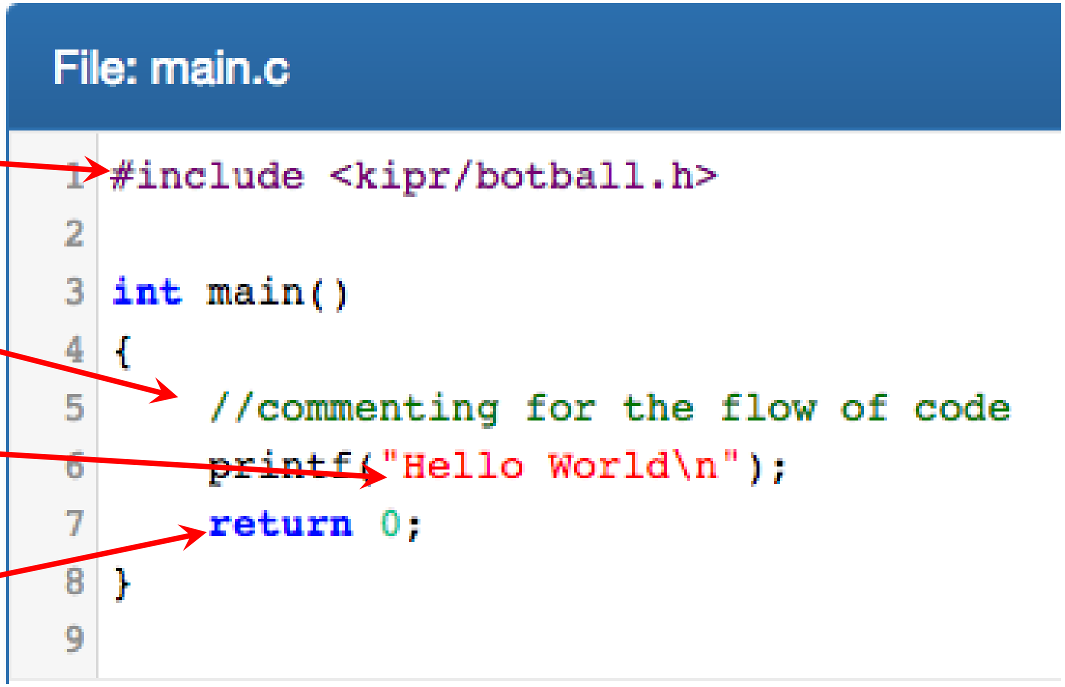
```
// Created on Thu January 10 2019

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

Comments are helpful notes that can be read by the programmer or other programmers. **They are *ignored* (<u>not</u> read) by the compiler!**

Text Color Highlighting

The KISS IDE highlights parts of a program to make it easier to read. (By default, the KISS IDE colors the 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

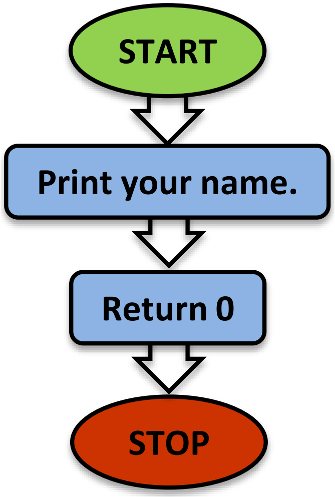
Description: Write a program for the KIPR Wallaby that prints your name.

Solution:

Source Code

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

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

Flowchart



START

Print your name.

Return 0

STOP

Designing a Program

Breaking Down a Task

Pseudocode, Flowcharts, and Comments

`msleep()` Function

Debugging a 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 Wallaby 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.
```

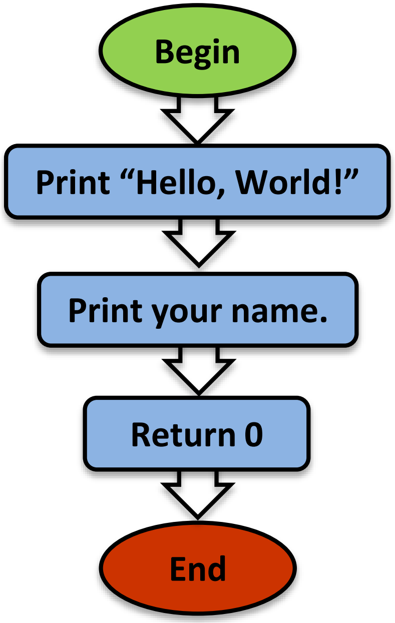
Comments

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

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

These are three different ways to do this.

Flowchart



Begin

Print “Hello, World!”

Print your name.

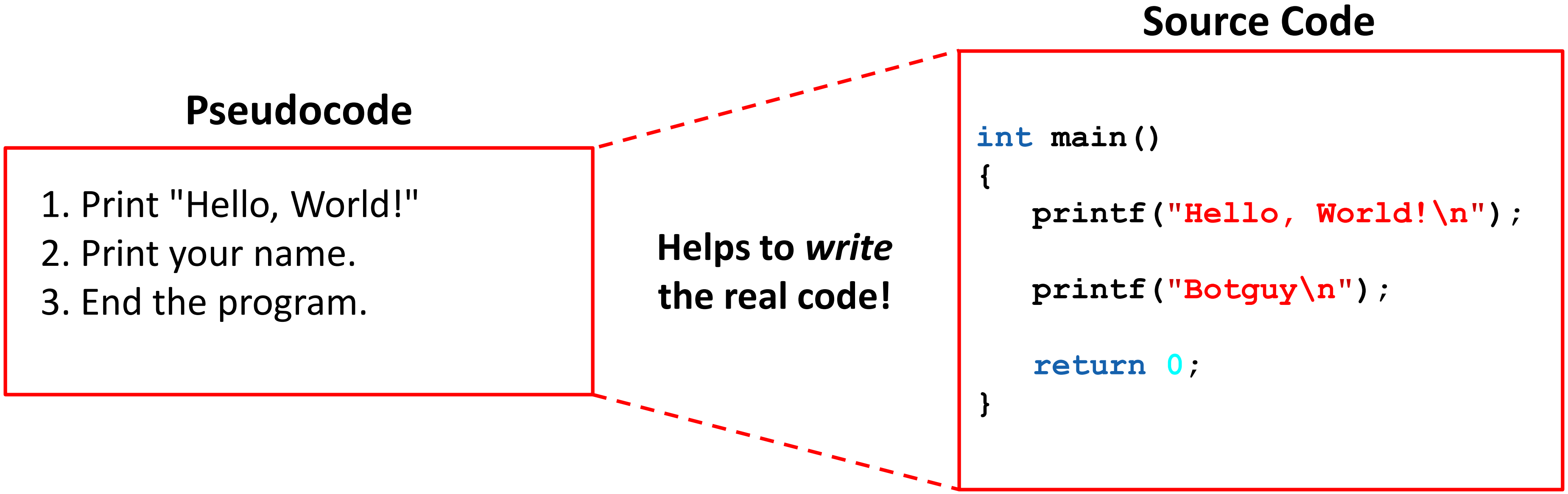
Return 0

End

Practice Printing

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

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



Pseudocode

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

Helps to *write* the real code!

Source Code

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

   printf("Botguy\n");

   return 0;
}
```

Execution: Compile and run the program on the KIPR Wallaby.

Practice Printing

Reflection: What was noticed after running the program?

- The Wallaby reads code and goes to the next line faster than a blink of an eye.
- At 800MHz, the Wallaby is executing millions 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 run on the robot.
- To do this, we use the built-in function called **`msleep()`**

↑

Let's use this!

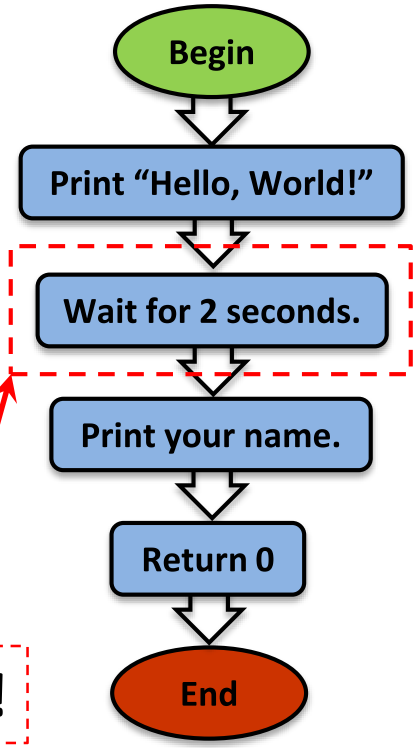
Waiting for Some Time

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

Analysis: What is the program supposed to do?

Pseudocode	Comments
1. Print “Hello, World!”	`// 1. Print "Hello, World!"`
2. Wait for 2 seconds.	`// 2. Wait for 2 seconds.`
3. Print your name.	`// 3. Print your name.`
4. End the program.	`// 4. End the program.`

Flowchart



New!

Waiting for some time

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

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

Pseudocode

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

Source Code

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

    msleep(2000);

    printf("I’m Botguy\n");

    return 0;
}
```

Execution: Compile and run the program on the KIPR Wallaby.

Waiting for Some Time

Reflection: What was noticed after running the program?

- Did the code work the first time?
- Were there any **errors**?

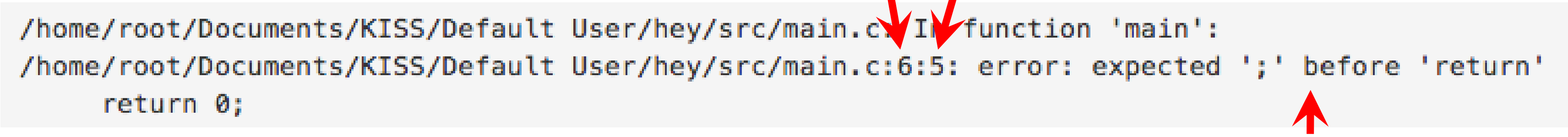
Debugging Errors

!!! ERROR !!!

- Not following the rules of the **programming language** will result in the **compiler** getting confused and not being able to **translate** the **source code** into **machine code**—it will say "**Compile Failed!**"
- The Wallaby will try to identify 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 the programs and compile it.
  - Try removing a `"` from one of the `printf()` statements.
  - What happens if `msleep()` is written as `Msleep()`?

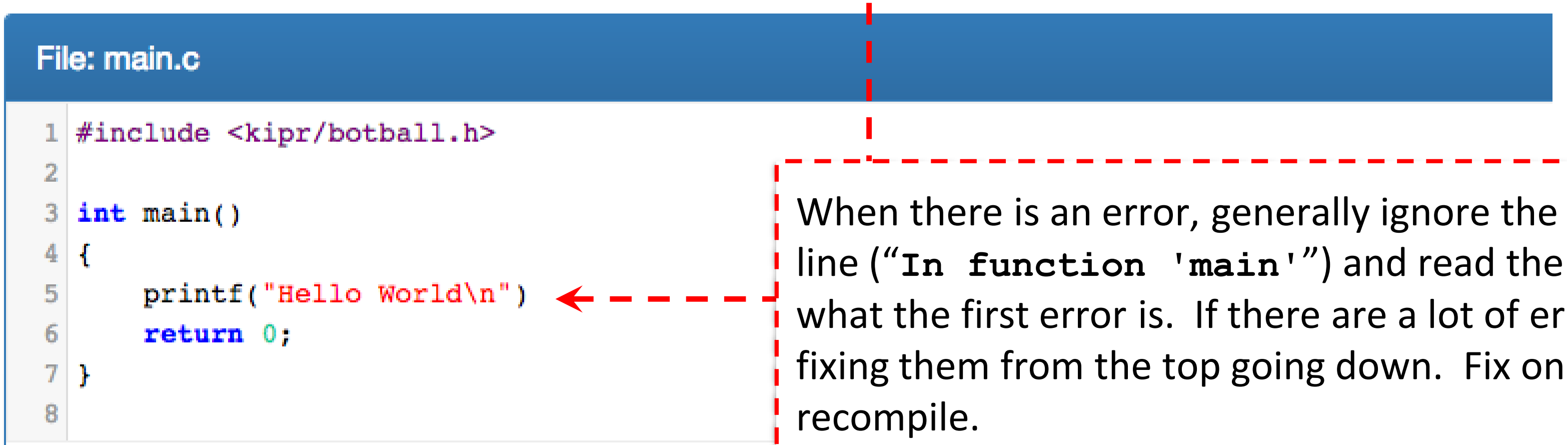
Debugging Errors

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;
```

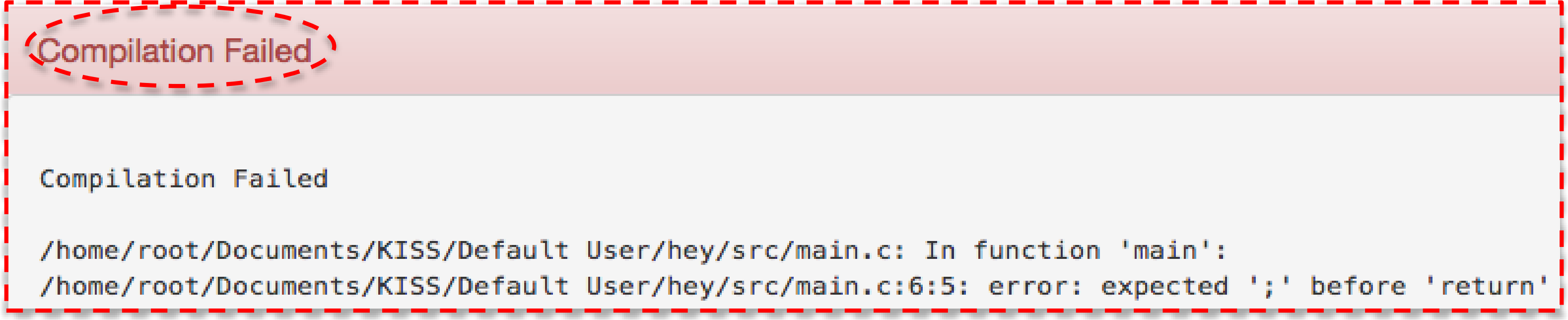
“ expected ; ” (semicolon)



File: main.c

```
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, generally ignore the first error line (“**`In function 'main'`**”) and read the next to see what the first error is. If there are 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

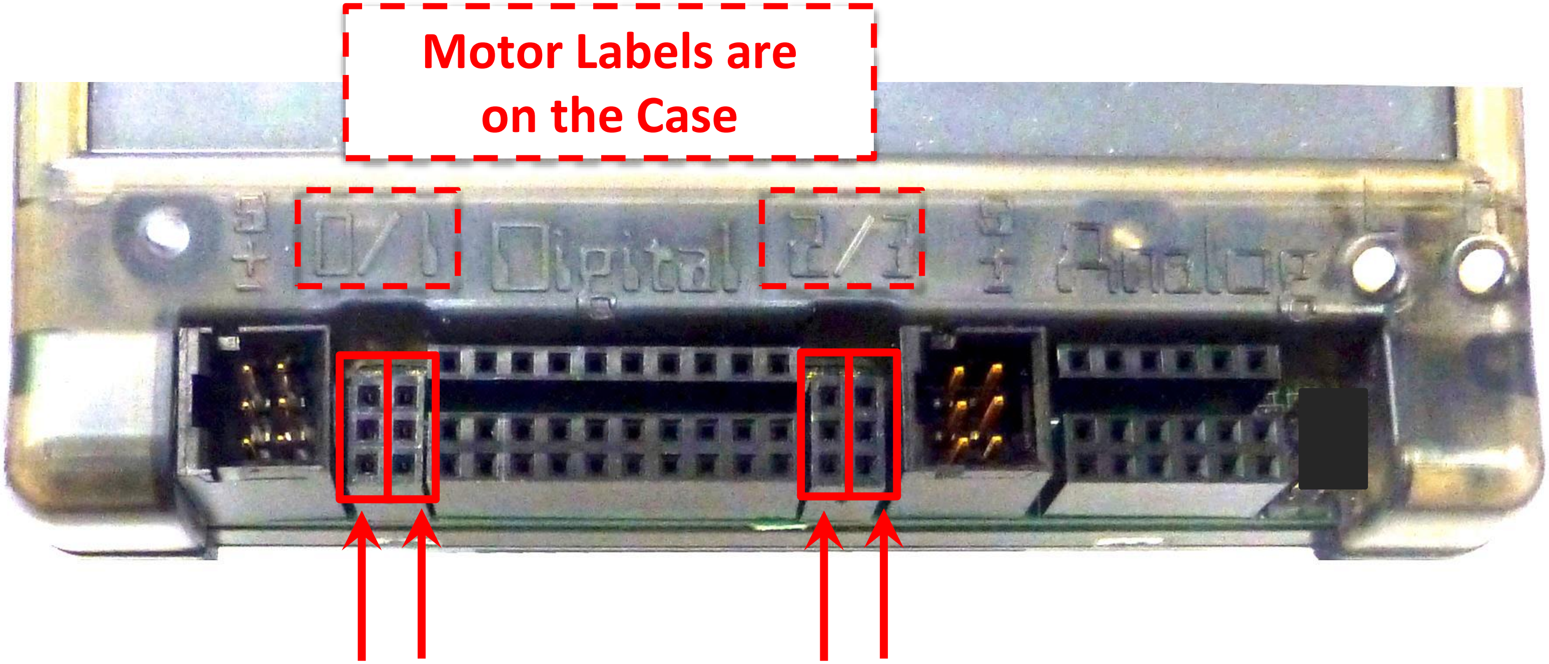
Plugging in motors (ports and direction)

`motor()` functions

Check the Robot's Motor Ports

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

Wallaby Motor Ports

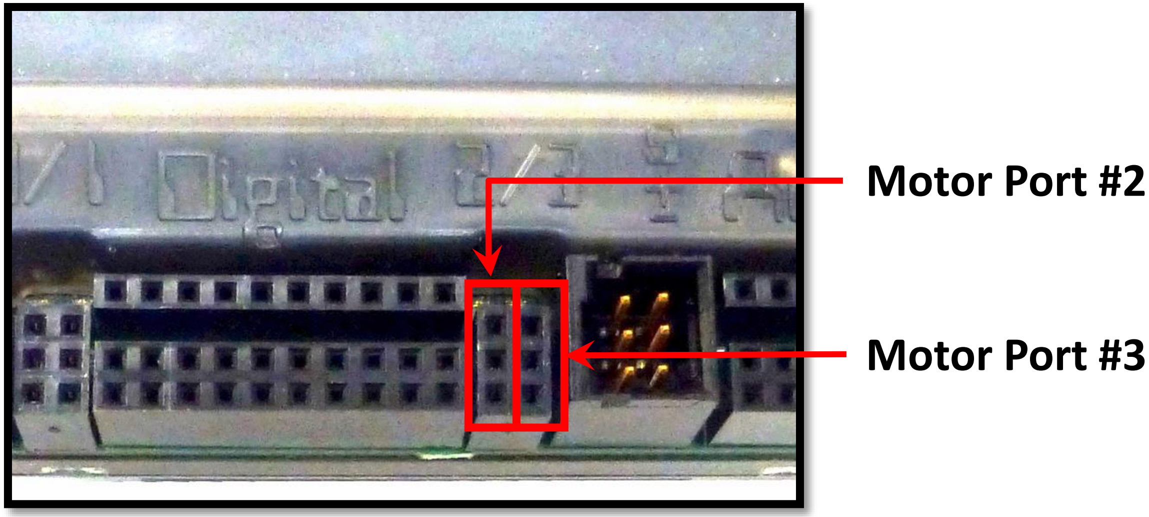


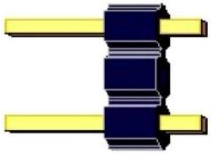
Motor Ports 0 & 1

Motor Ports 2 & 3

Plugging in Motors

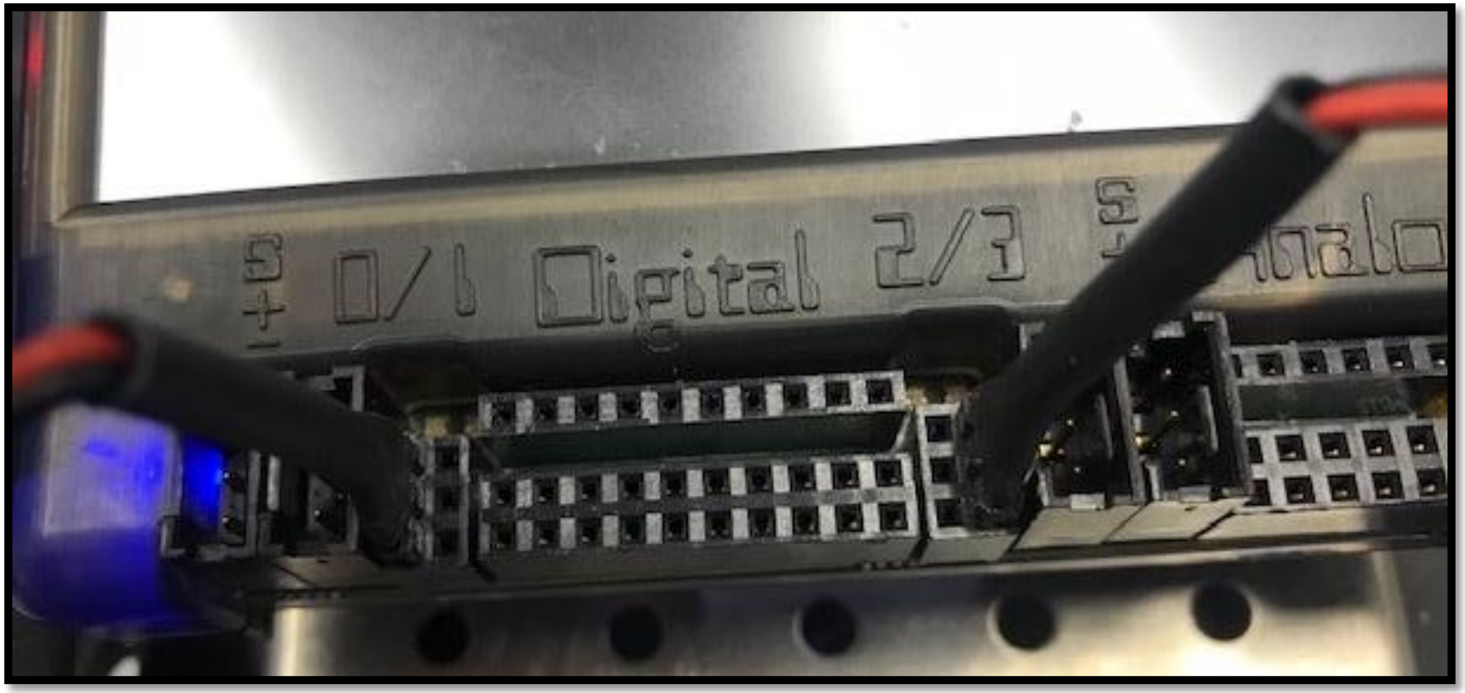
- **Motors** have red wire and a black wire with a **two-prong plug**.
- The Wallaby has 4 motor ports numbered **0** & **1** on left, and **2** & **3** on right.
- When a port is powered (receiving motor commands), it has a light that glows **green** for one direction and **red** for the other direction.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





Drive motors have a two-prong plug.

Plugged in Motors

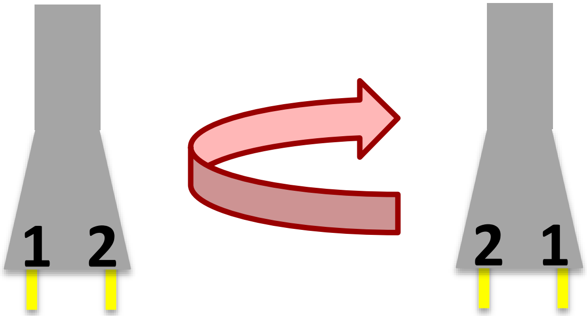


DemoBot Motor Ports 0 (right wheel) and 3 (left wheel)

Motor Direction

The motors should be going in the same direction; otherwise, the robot will go in circles!

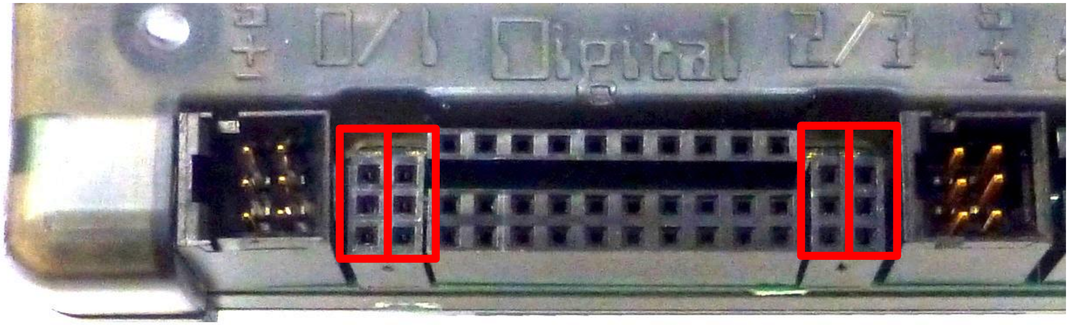
- **Motors** have a red wire and a black wire with a **two-prong plug.**
- These can be plugged in two different ways:
  - One direction is clockwise, and the other direction is counterclockwise.
  - The red and black wires help determine motor direction.

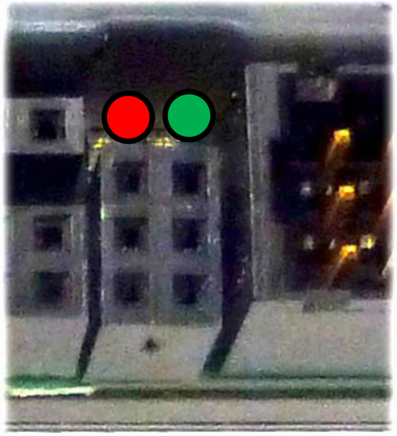


Motor Port and Direction Check

There is an easy way to check this!

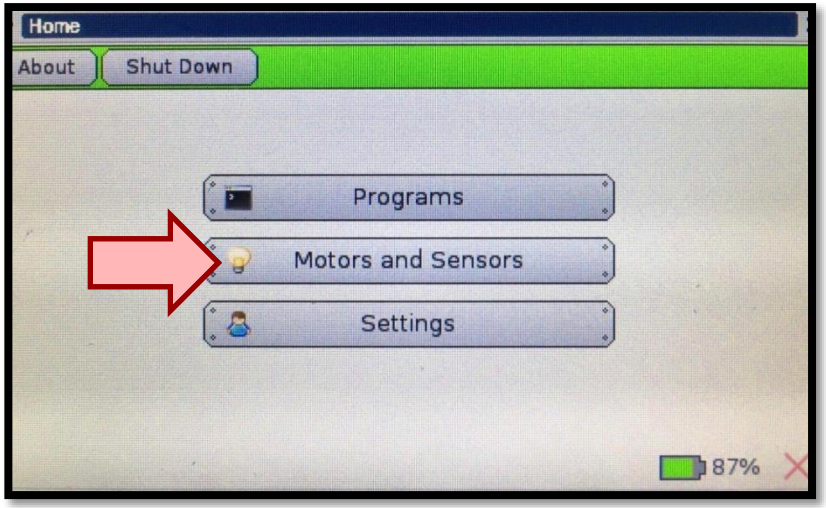
- Manually rotate the tire to see an LED light up by the **motor port** (the **port #** is labeled on the board).
  - If the LED is green, it is going **forward** (+).
  - If the LED is red, it is going **reverse** (–).

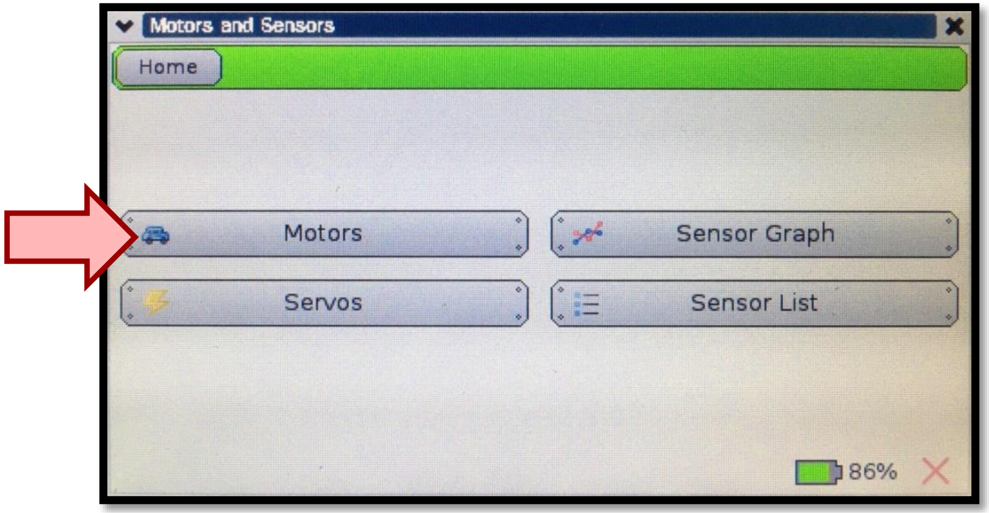


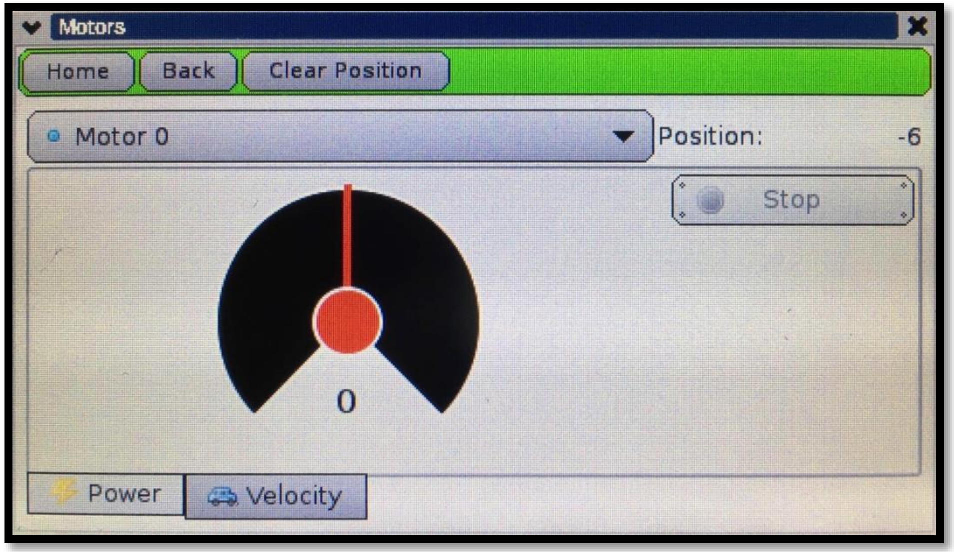


- Use this trick to check the **port #**'s and **direction** of the **motors**.
  - If one is red and the other is green,
    turn one motor plug 180° and plug it back in.
  - The lights should both be green if the robot is moving forward.

Use the Motor Widget







Position: -6

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.

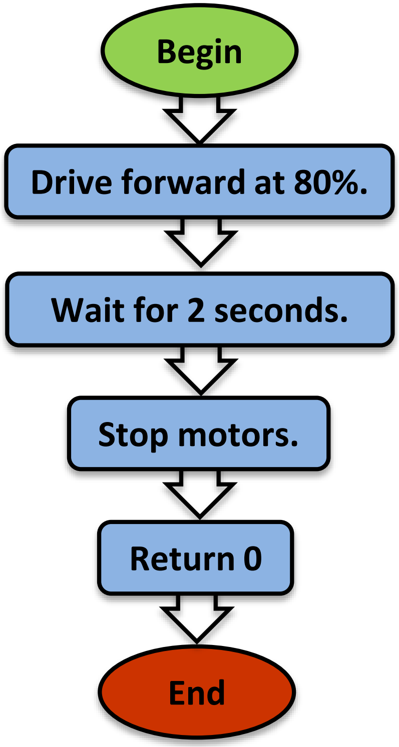
Moving the DemoBot

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

Analysis: What is the program supposed to do?

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

Flowchart



Moving the DemoBot

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

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

Pseudocode

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

Source Code

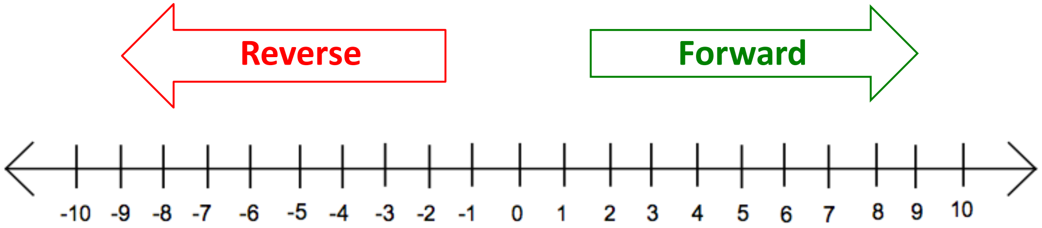
```c
#include <kipr/botball.h>
int main()
{
  //forward
  motor(0, 80);
  motor(3, 80);
  msleep(2000);

  ao();

  return 0;
}
```

Moves robot forward

Execution: Compile and run the program on the KIPR Wallaby.

Robot Driving Hints

Remember the # 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:** Adjust this by slowing down or speeding up the motors.

Making turns:

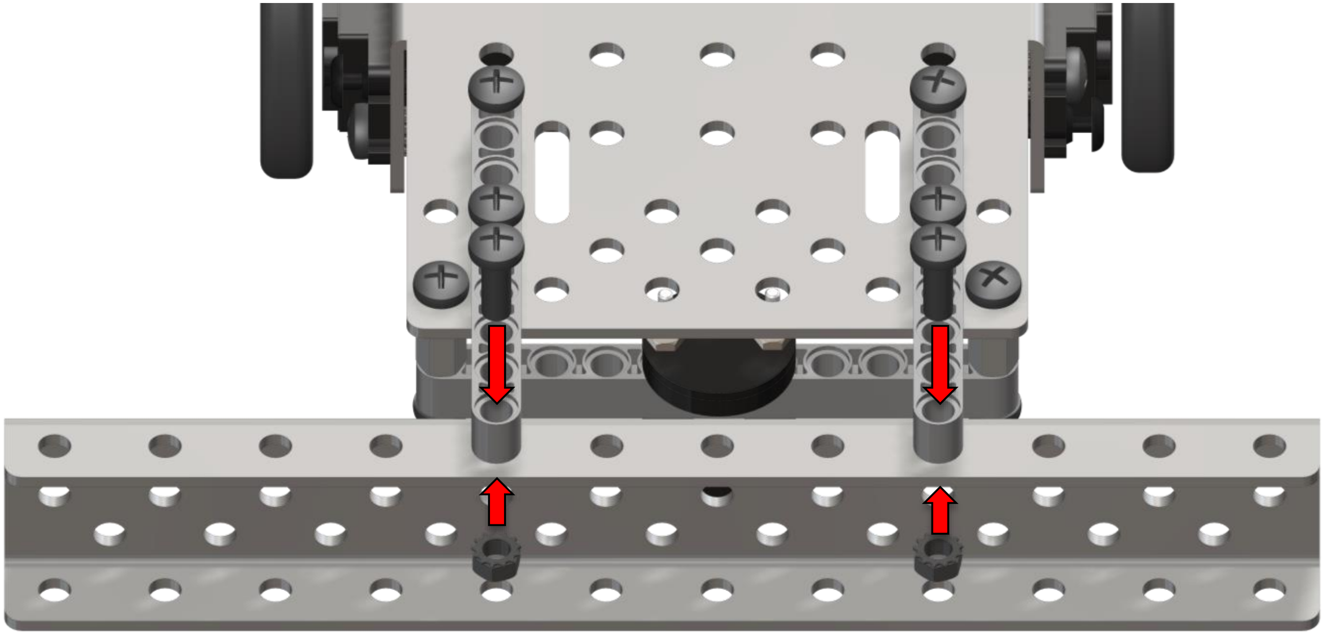
- **Solution:** Have one wheel go faster or slower than the other.
- **Solution:** Have one wheel move while the other one is stopped.
- **Solution:** Have one wheel move forward and the other wheel move in reverse (friction is less of a factor when both wheels are moving).

Moving the DemoBot

Task #1: Place a 2” foam block on circle 6 of Mat A. Write a program that will drive the DemoBot from the start box towards the can, touch it without knocking it over or pushing it outside of the circle. The drive back to the start box.

Task #2: Place a 2” foam block on circle 6 of Mat A. Write a program that will drive the DemoBot from the start box around the can without touching it, and then drive back to the start box.

Bulldozing



Attach the bulldozer blade to back of the robot as shown.
Use the 3x11 hole channel.

Connections to the Game Board

Description: Build and attach a custom piece to the DemoBot build that will allow the robot to successfully bulldoze game pieces to specified areas on the mats.

Goal #1: Mat A - Stack poms in piles of 4, 3 as a base and 1 on top. Starting in the start box, bulldoze the stacks of poms from circle 3 into the blue garage. Robot or game pieces may not cross solid lines of targeted garage. *Bonus: Starting in the start box, bulldoze the stacks of poms from circles 3 and 10 into the yellow garage.*

Goal #2: Mat B – Set four groups of two 1” cubes along the blue dotted line, evenly spread from the start line to the top curve of the black line. Starting behind the start line, bulldoze the blocks so that none are left touching the blue line. *Bonus: Starting behind the start line, bulldoze the blocks from the blue dotted line, so that they end completely outside the perimeter of the black line.*

Moving the DemoBot Servos

Plugging in servos (ports)

`enable_servos()` and `disable_servos()` functions

`set_servo_position()` function

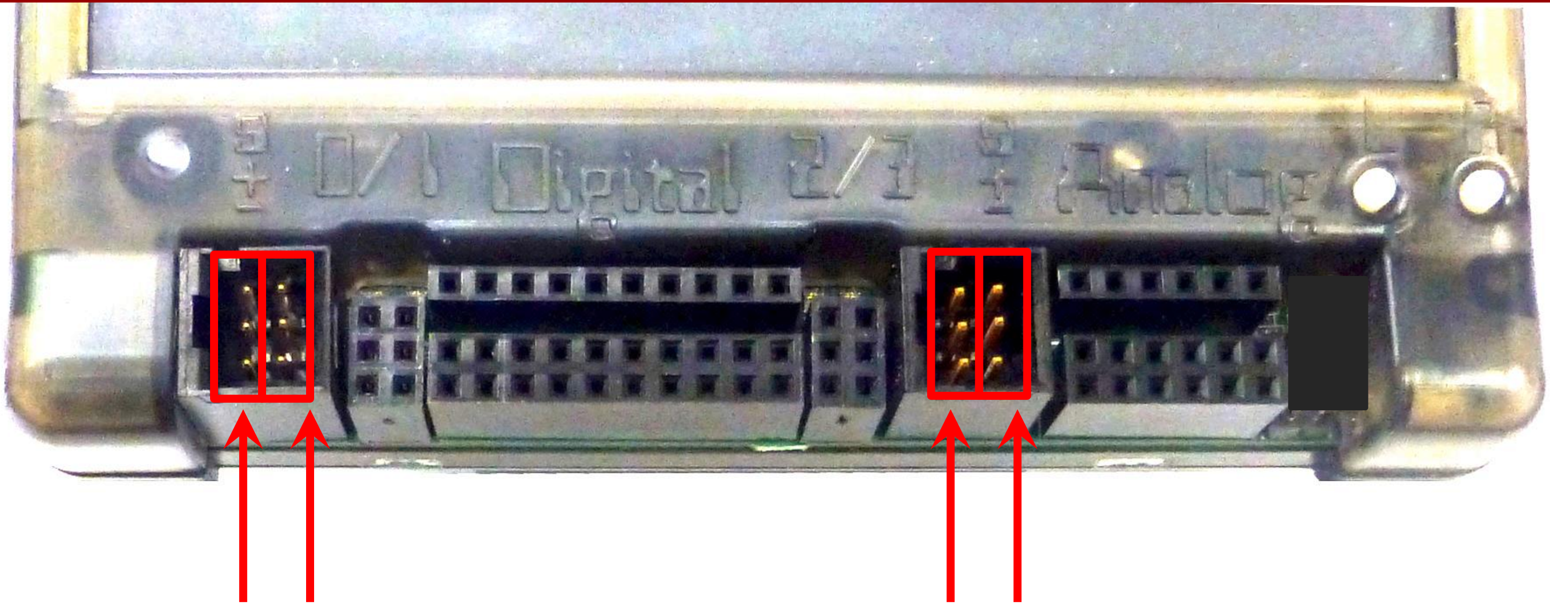
Servos

- 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**.

Large Servo

Micro Servo

KIPR Robotics Controller Servo Ports

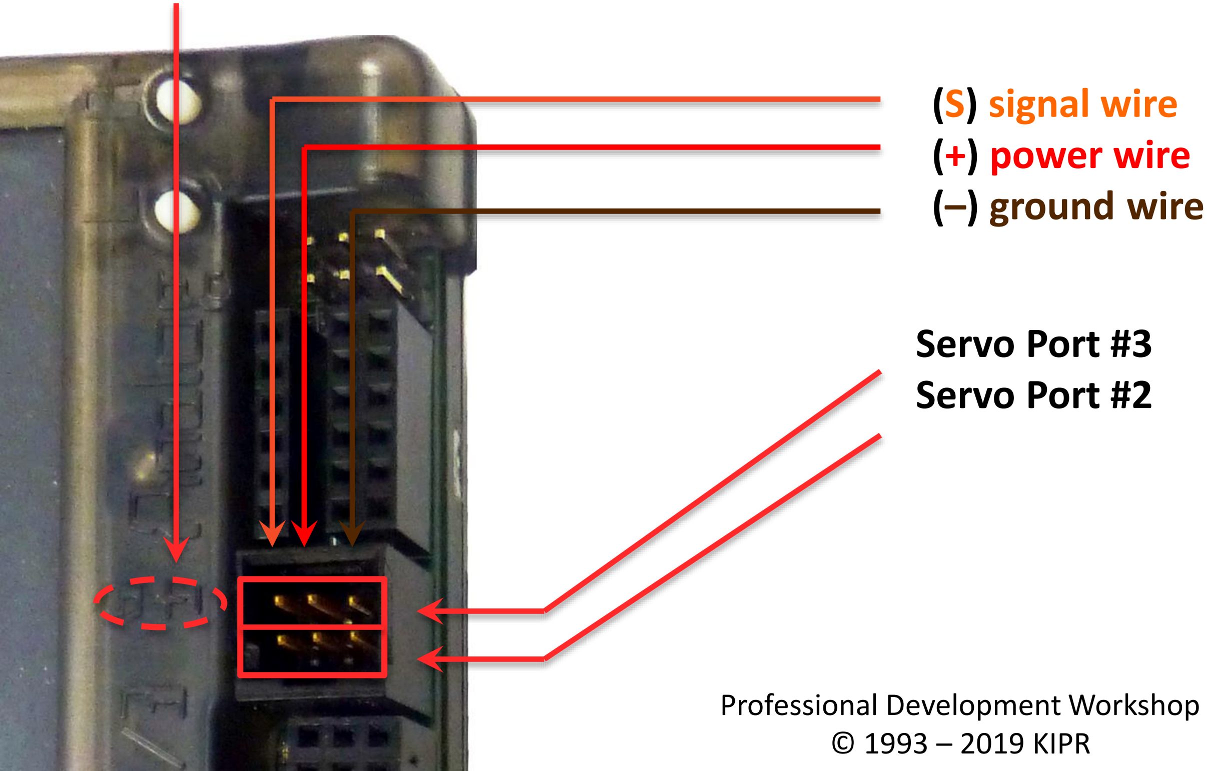


Servo Ports 0 & 1

Servo Ports 2 & 3

Plugging in Servos

- The KIPR Robotics Controller has 4 servo ports numbered **0** (left) & **1** (right) on the left, and **2** (left) & **3** (right) on the right.
- Notice that the case of the KIPR Robotics Controller is marked:
  - (S) for the orange (signal) wire, which regulates servo position
  - (+) for the red (**power**) wire
  - (–) for the **brown** (**ground**) wire (“the ground is down, down is negative”)



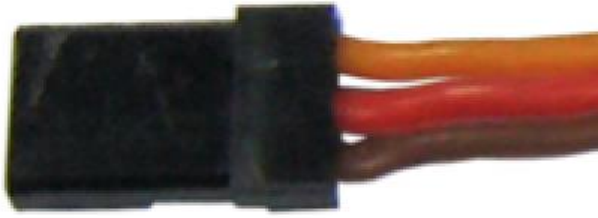
(S) signal wire

(+) power wire

(–) ground wire

Servo Port #3

Servo Port #2

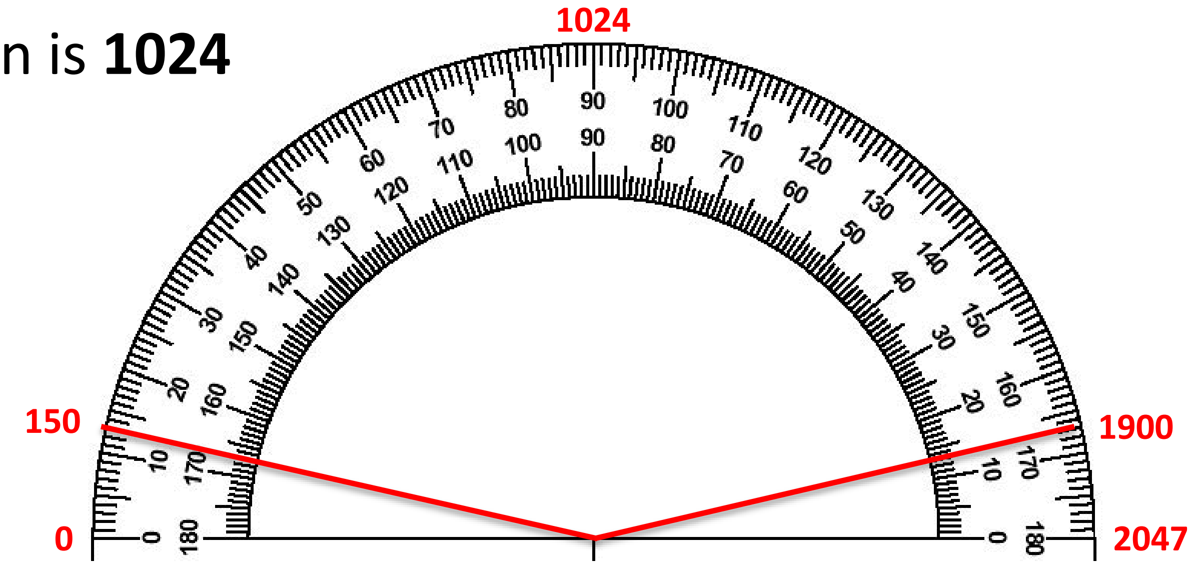


NOTICE: orientation plugging in the servos is very important

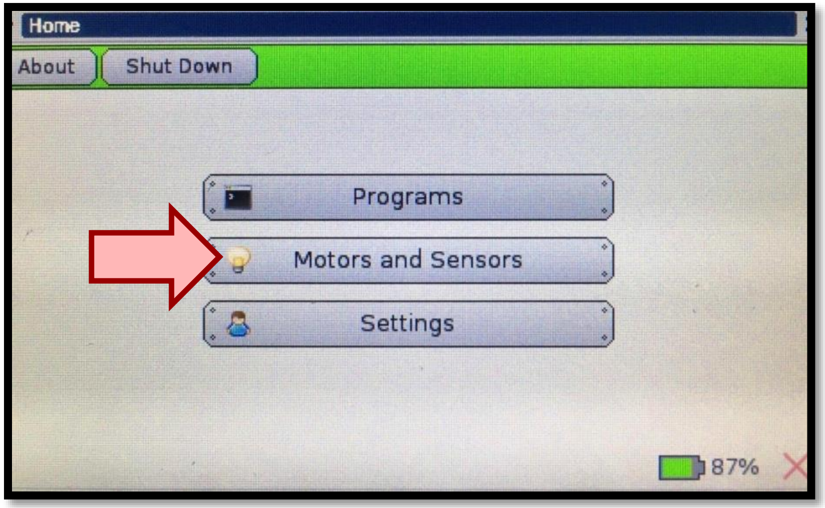
Professional Development Workshop
© 1993 – 2019 KIPR

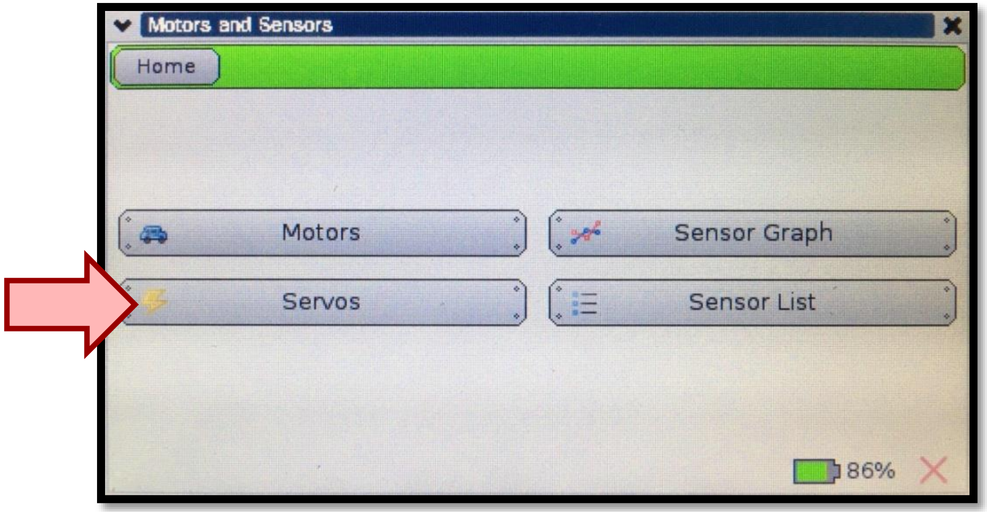
Servo Positions

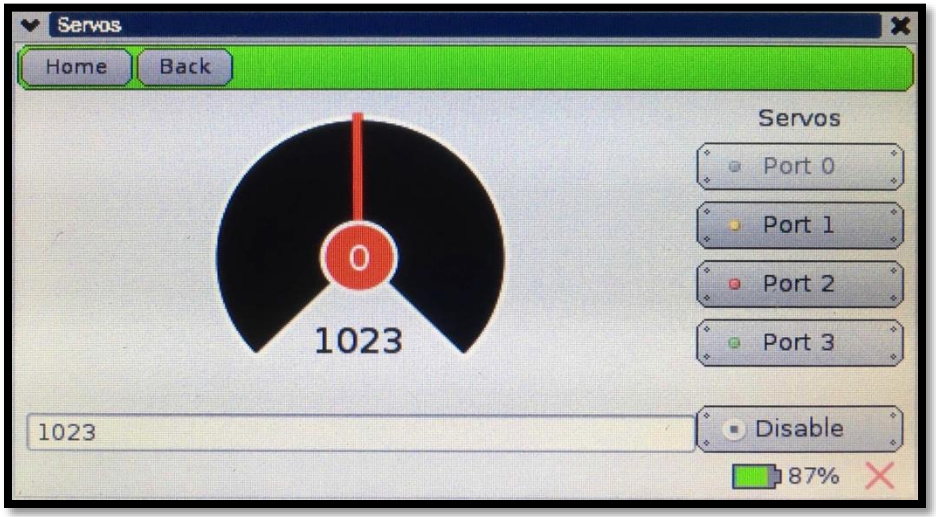
- Think of a servo like a protractor…
  - Angles in the **~180° range of motion** (between ~0° and ~180°) are divided into **2048 servo positions**.
  - These **2048 positions** range from 0 to 2047, but due to internal mechanical hard stop variability **~150 to ~1900** should be used
    (**remember:**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



Use the Servo Widget

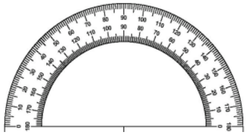


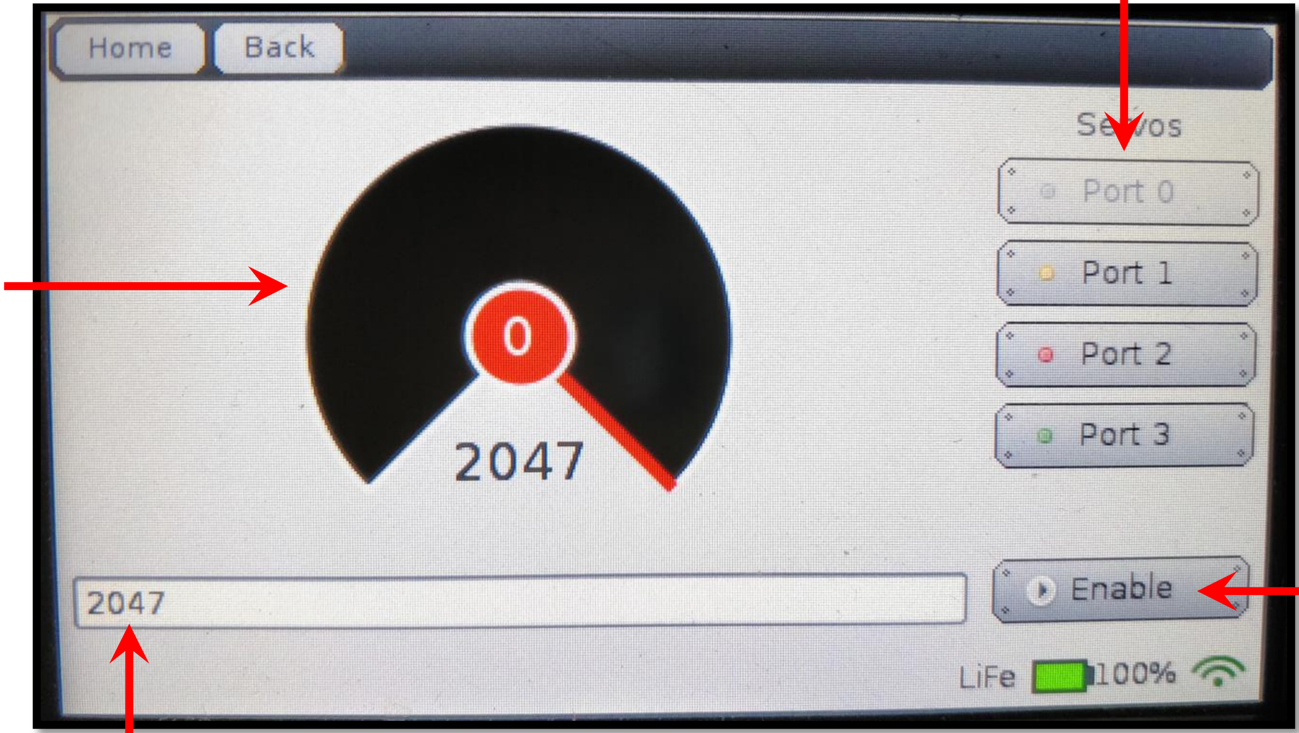




Testing Servos with the Widget

Select the servo port



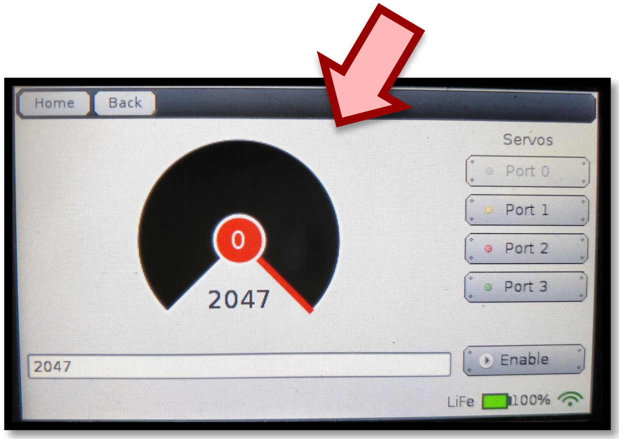


Enable servos

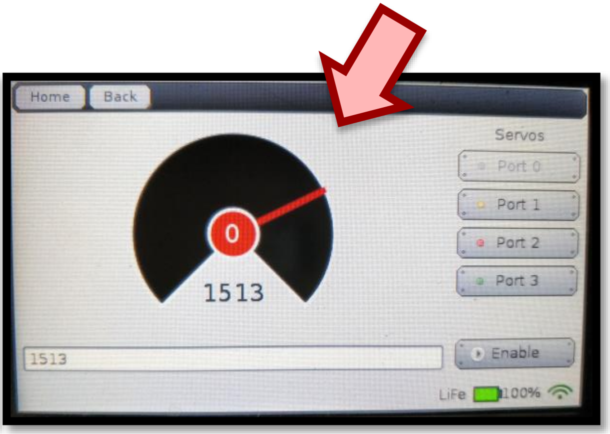
The current servo position

Testing Servos with the Widget

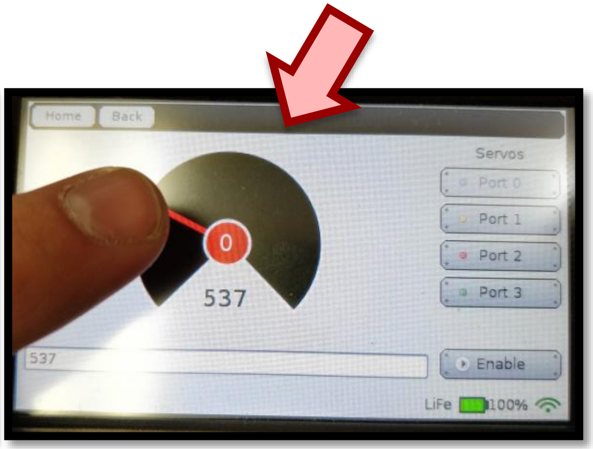
Use a finger
to move the dial.



Servo @ 2047
(maxed out)



Servo @ 1513

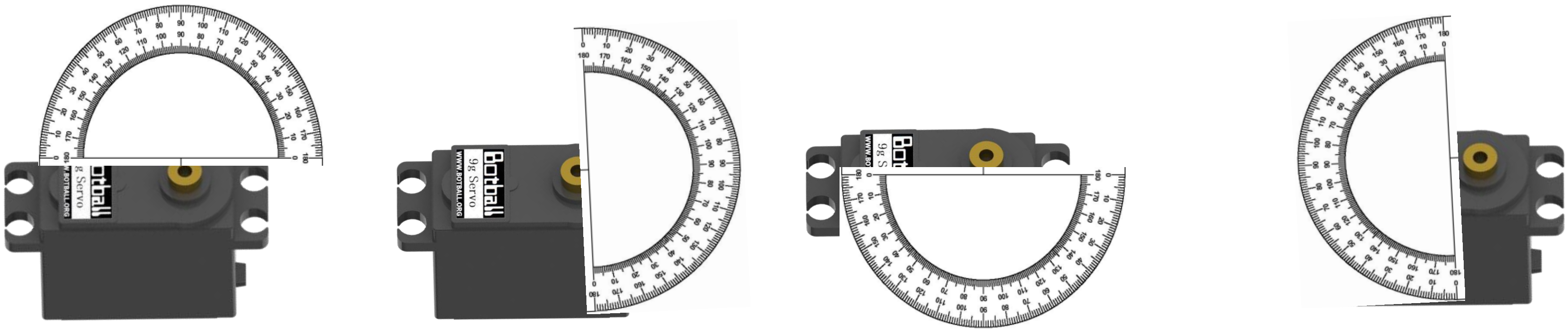


Servo @ 537

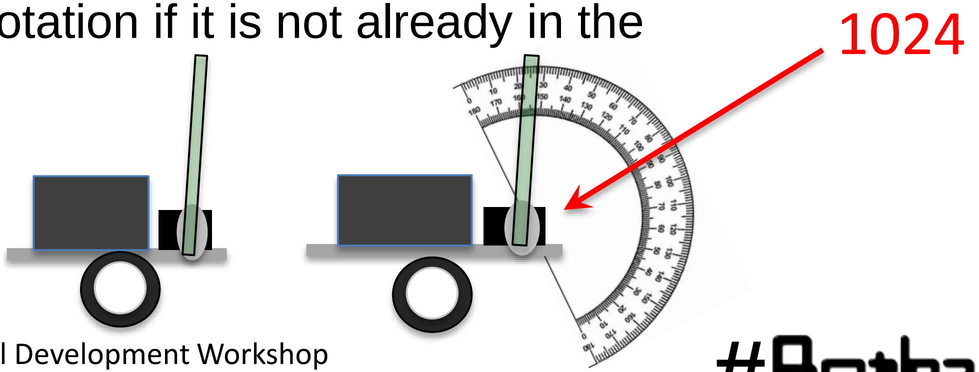
Do <u>not</u> push a servo beyond its limits
(less than ~150 or more than ~1900).
This can burn out the servo motor!

Centering the Servo Horn

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



Using the Servo Widget, enable the servo on the robot. When it is enabled, it will go to 1024. Unscrew the servo horn on the 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 a 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 it is set to another position without giving it **time** the **code** runs very fast and does not wait for the servo to move.
- A servo position can be "**preset**" by calling `set_servo_position()` ***before*** calling `enable_servos()`. This will make the servo move to this position immediately upon calling `enable_servos()`.

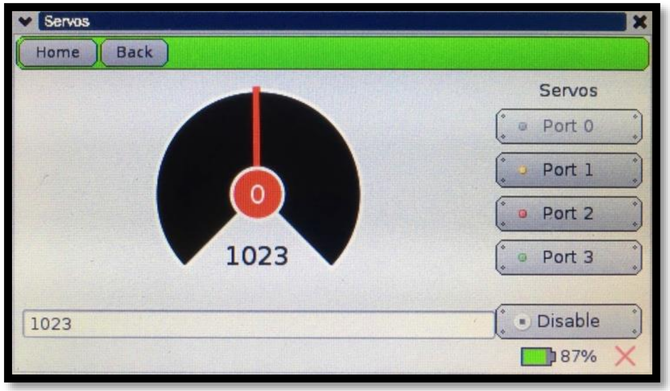
Using Servo Functions

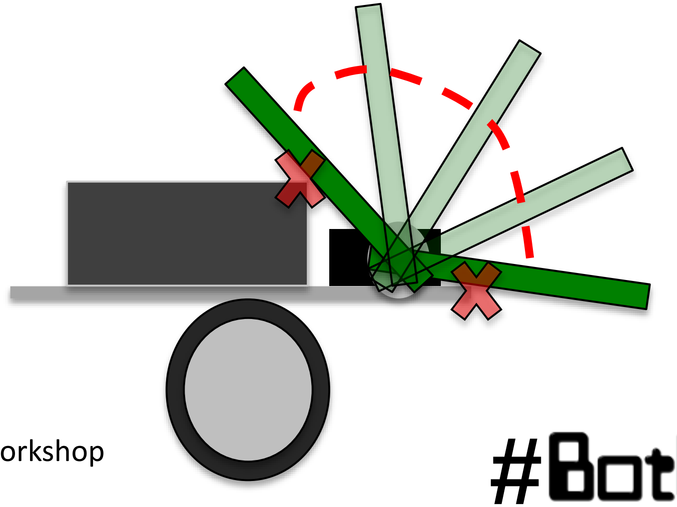
```
int main()
{
  enable_servos();

  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;
}
```

Wave the Servo Arm

Description: Write a function for the KIPR Wallaby that waves the DemoBot servo arm up and down.

- Remember to **enable the servos** at the beginning of the program, and **disable the servos** at the end of the program!
- **Warning:** The arm mounted on the DemoBot prevents the servo from freely rotating to all possible positions. It will run into the KIPR Wallaby 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 the robot.
  - Use the Servo screen to **determine the limits** of the DemoBot arm, **write these numbers down**, and then **use these numbers in the code**.





Wave the Servo Arm

Description: Write a program for the KIPR Wallaby that waves the DemoBot servo arm up and down. Write a function that does one wave. Call it from the 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.
```

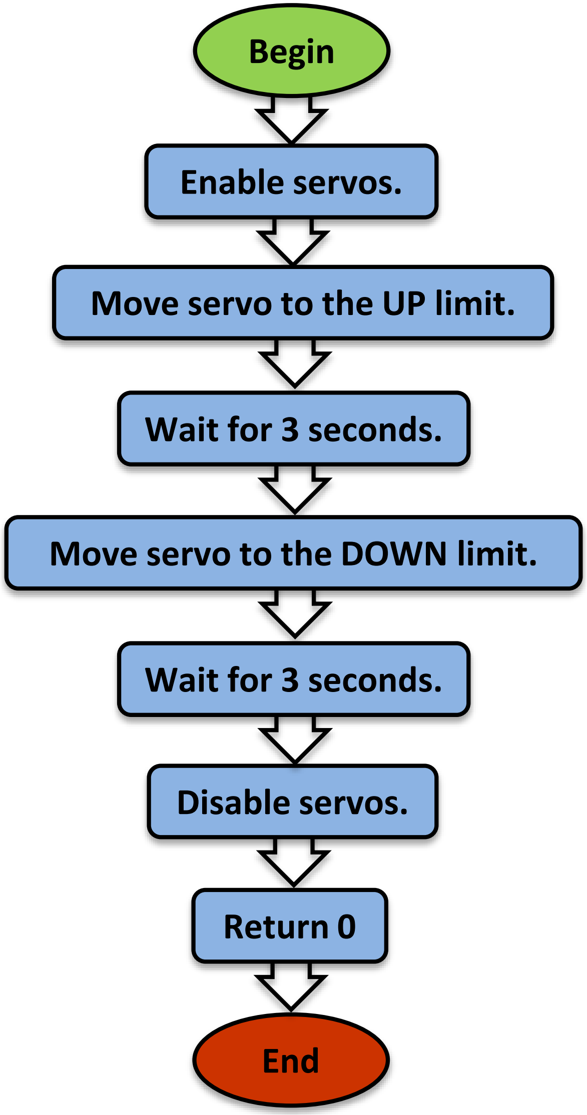
Comments

```
// 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



Begin

Enable servos.

Move servo to the UP limit.

Wait for 3 seconds.

Move servo to the DOWN limit.

Wait for 3 seconds.

Disable servos.

Return 0

End

Commenting Within a Program

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

  printf("Wave Servo Exercise\n");
  return 0;
}
```

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

Keeping track of the ports, positions, etc could also be done in a notebook, but what if that notebook is misplaced?

Variables

Some reasons to use a variable:

1. Do not have to *remember* which value is a certain servo position – the computer remembers instead

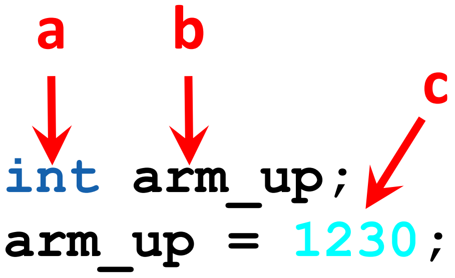
2. It makes the program easier to read and understand

3. Makes it easier to debug the program

4. Variables can be used to store the results of computations

Variables

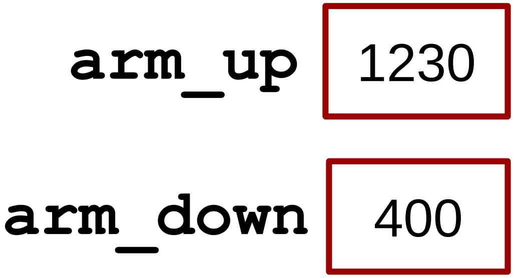
- A **variable** is a ***named*** container that stores a **type** of **value**
  A **variable** has the following three components:
  a. the **type** of data it stores (holds),
  b. the **name**, and
  c. the **value**.



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

Use `int` as the data type to store whole numbers, aka 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.



Variable Names

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

- Variable names can be *almost* anything.
- Variable names can contain **letters**, **numbers**, and **underscores** (“_”).
- Variable names **cannot** begin with a **number**.
- Variable names should be ***meaningful*** and not “x”

An Example:

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

The declaration and initialization can be done at the same time

```
int arm_up = 1230;
```

Working with Variables

1. *Declaring* a variable:

```
int arm_up;
```

2. *Initializing* 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

Using Variables for Servo Motors

```
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 the comments, add int for the data type, and since it is now code add the semicolon

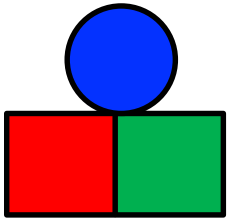
```
int main ()
{

  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;
}
```

Connections to the Game Board

Description: Navigate to and manipulate game pieces using the claw and servos.

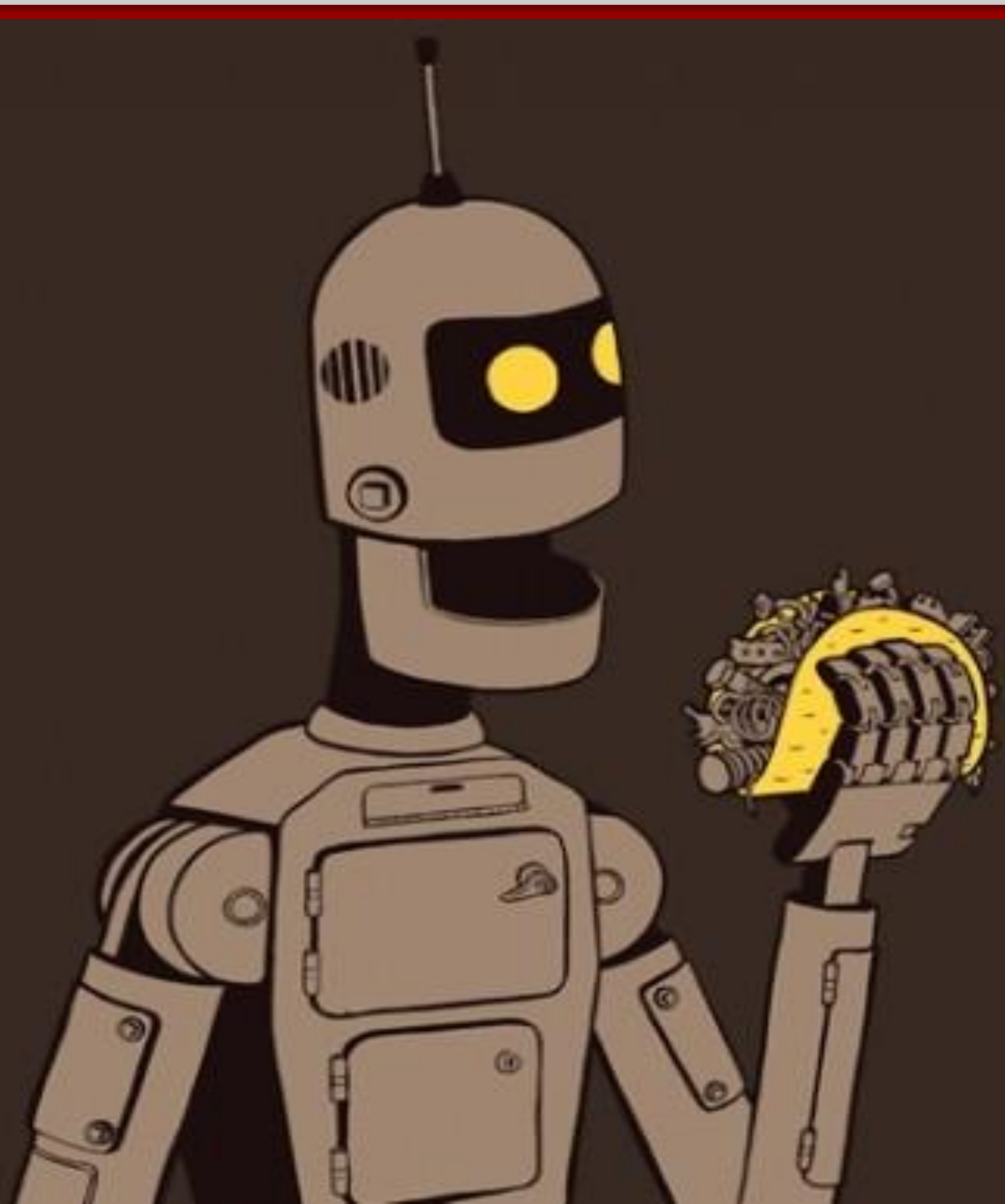
Goal #1: Mat A - Starting in the start box, move the firetruck from circle 7 to circle 10. *Bonus: Adding to the previous program, after setting firetruck down, pick up a 2" cube from circle 12 and stack on top of the firetruck.*

Goal #2: Mat A – Create the following stack in circle 5 using a blue pom and two 1" blocks:



Customize the claw so that it can pick up the whole stack. Starting in the start box, pick up the stack from circle 5 and move it to the yellow garage. Robot or game pieces may not cross solid lines of targeted garage. *Bonus*: *After setting the stack in the yellow garage, pick only the pom back up and move it to the blue garage.*

Lunch!



Making Smarter Robots with Sensors

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

`wait_for_light()` function

Sensors

- It is difficult 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 – 4095

- **Ports:** 0 – 5
- **Function:** `analog(port #)`
- **Sensors:**
  - Light

  - Small reflectance

  - Large reflectance

  - Slide sensor

  - Range Finder

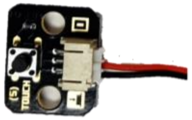
Digital Sensors

- **Range of values:**

  0 (not pressed) or 1 (pressed)

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

  - Small touch



  - Lever touch

Remember Sensor Functions

Retrieve the analog sensor value with a function

- There are 6 analog ports (0-5)

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

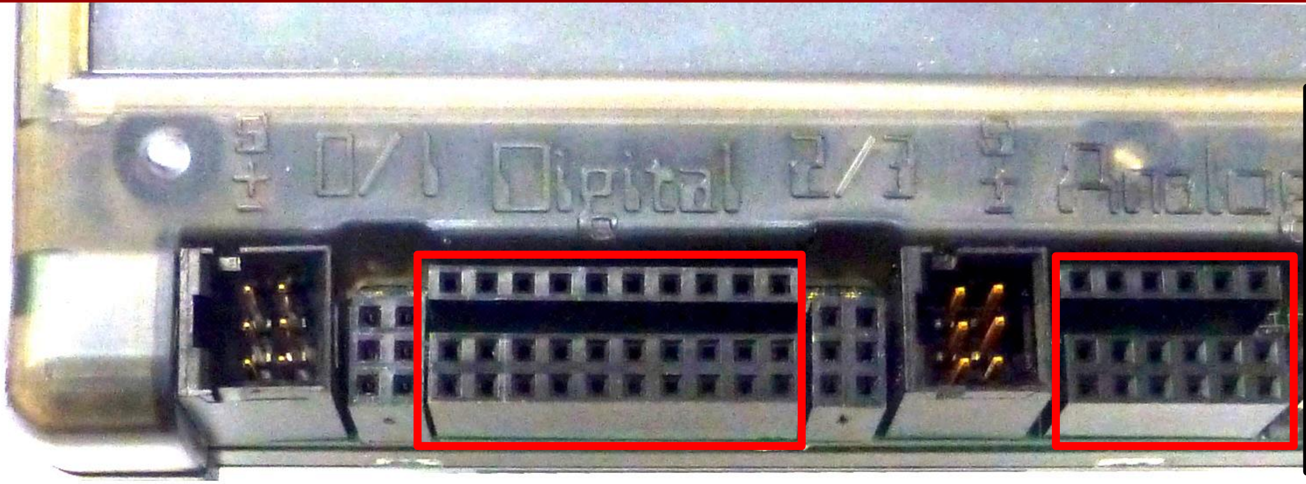
Retrieve the digital sensor value with a function

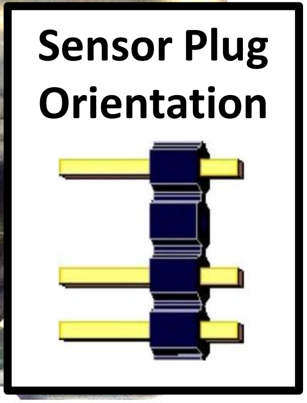
- There are 10 digital ports (0-9)

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

NOTE: when these functions are called, they return an integer value into the “code” where they were called at the time the code is run.

KIPR Robotics Controller Sensor Ports




Sensor Plug Orientation

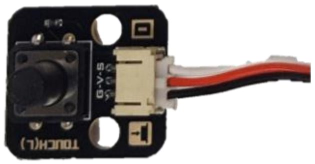
Digital Sensors
Ports # 0 – 9

Analog Sensors
Ports # 0-5

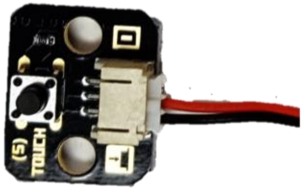
Detecting Touch

There are many digital sensors in the kit that can detect touch.

Large Touch



Small Touch



Lever Touch

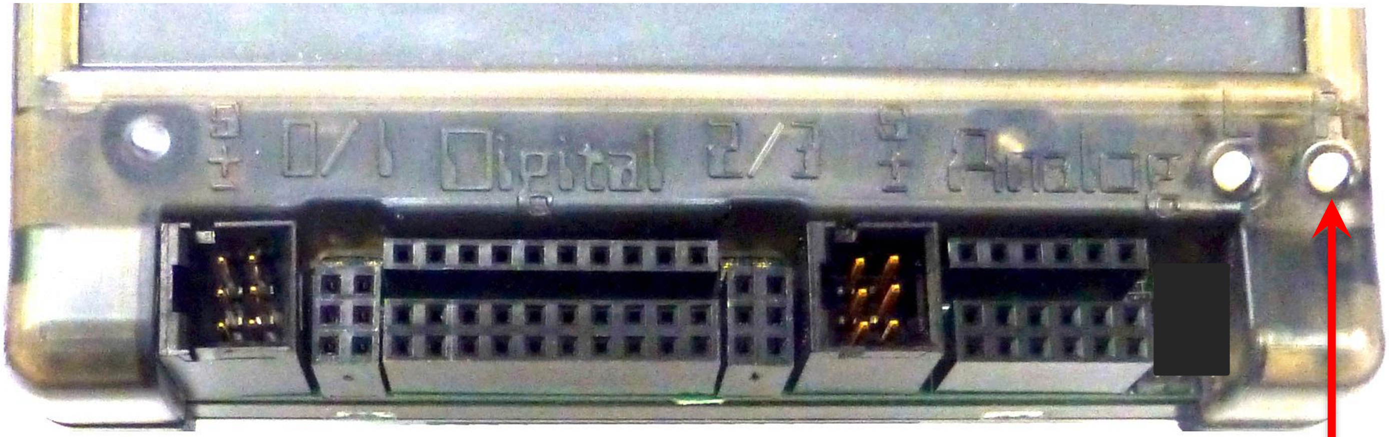
Built-In Digital Sensor

- The Wallaby has built-in buttons on the right side (opposite the power switch)

```
right_button()
left_button()
a_button()
b_button()
c_button()
```

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

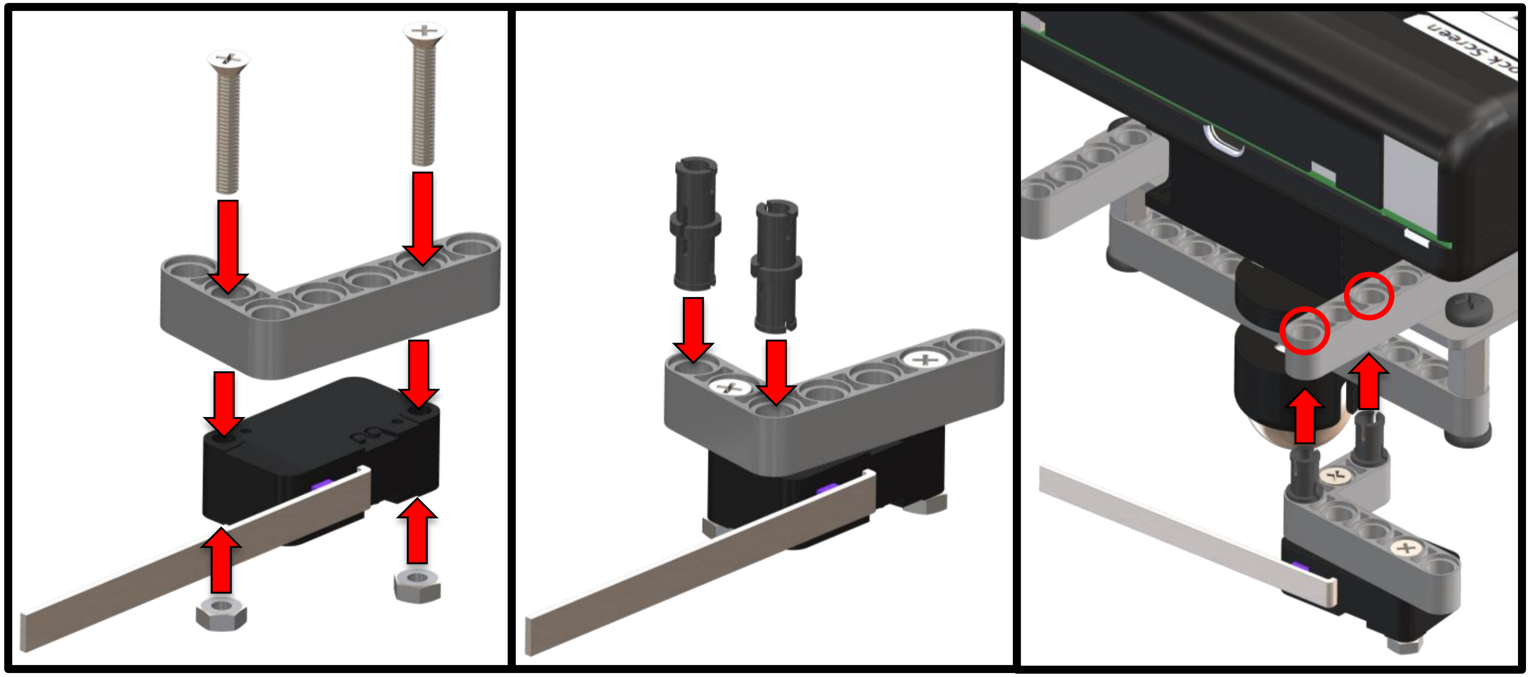
while Loop on R Button



```
int main()
{
  // Has R button been touched?
  while(right_button() != 1)
  {
    printf("Press the R Button!\n");
  }

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

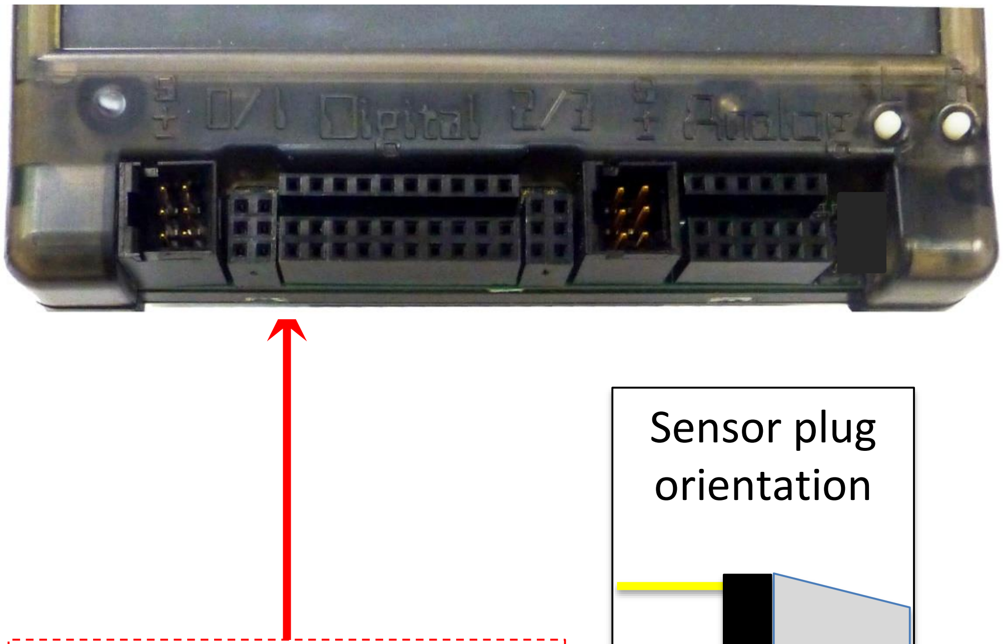
R button

Mounting Lever Touch Sensor



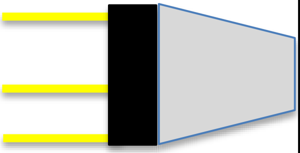
To add the lever sensor mount. Remove the channel you used for the bulldozer on the back of the Wallaby and then follow the steps above.

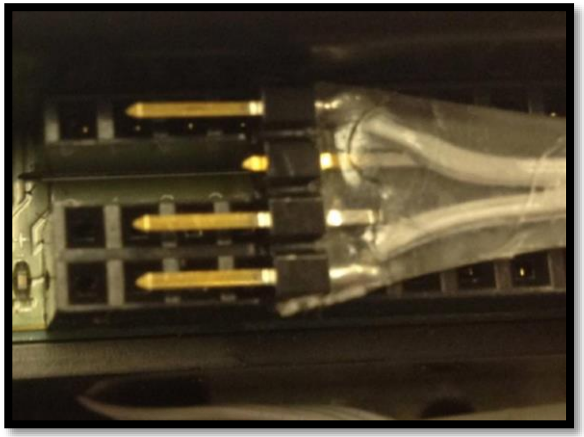
Plug in the Lever Touch Sensor



Plug the touch sensor into digital port 0

Sensor plug orientation



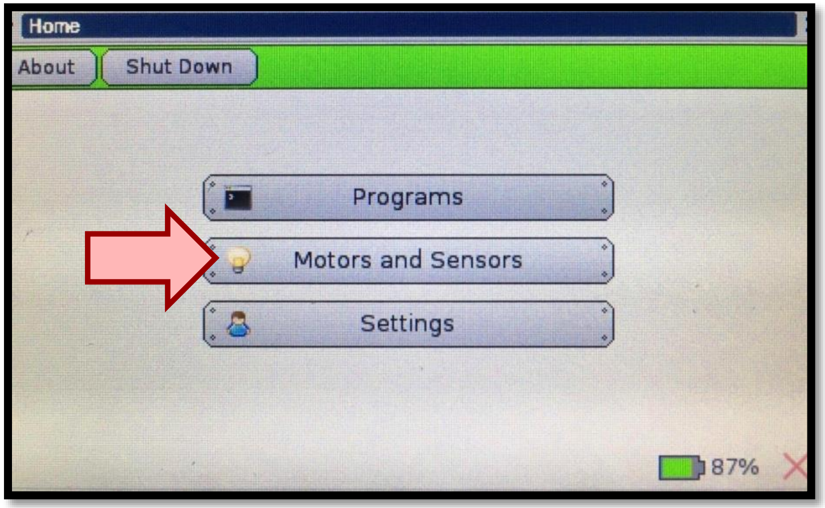


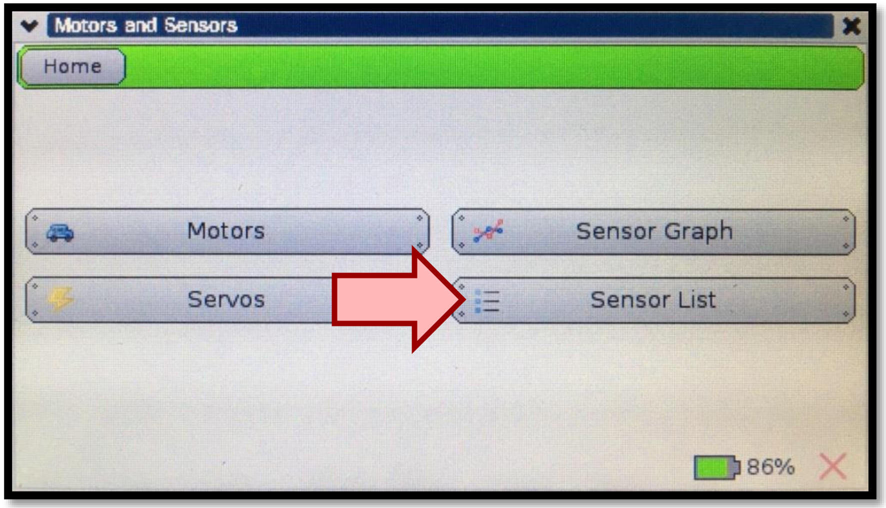
Close-up of sensor plug orientation

Reading Sensor Values from Robot

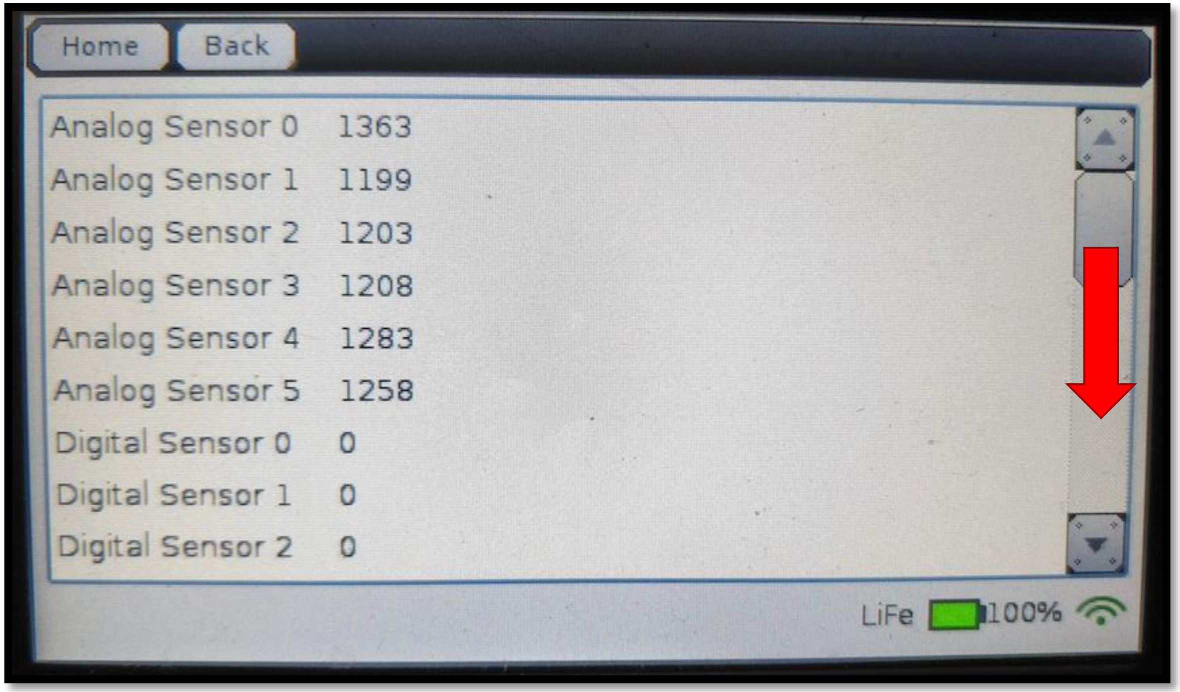
Sensor Values can be accessed from the Sensor List on the Wallaby

- This is very helpful to see readings from all of the sensors being used prior to utilizing the values in the code.





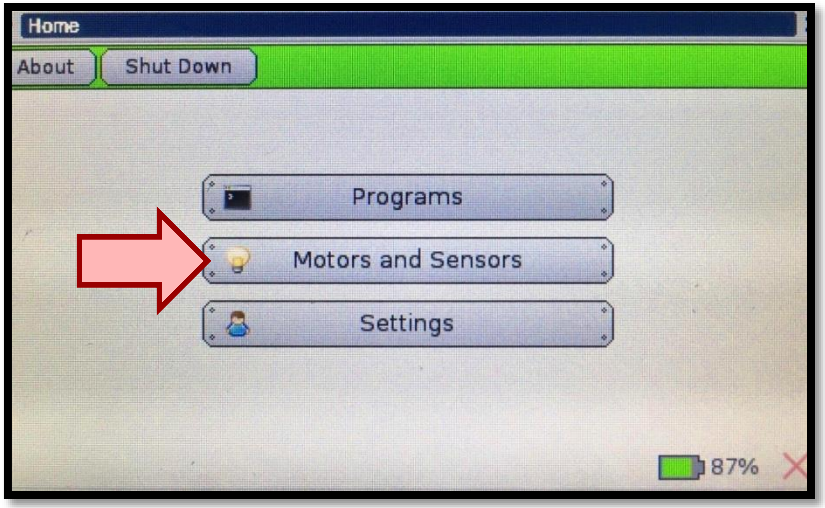
Reading Sensor Values from Robot

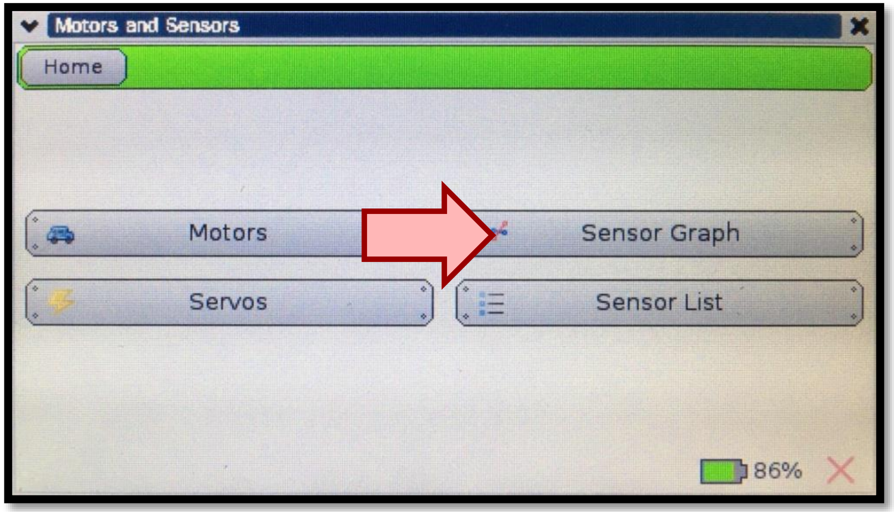


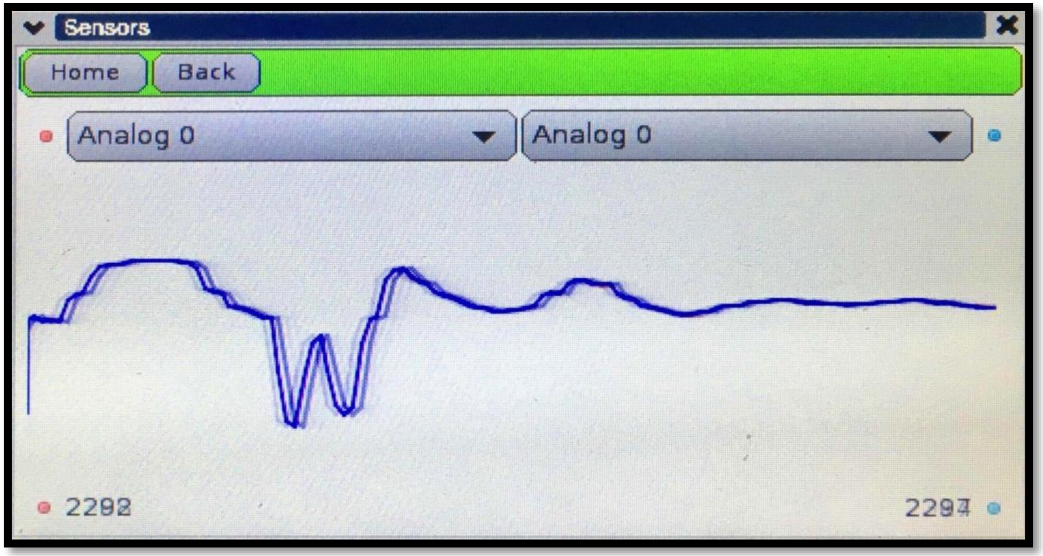
```
Analog Sensor 0    1363
Analog Sensor 1    1199
Analog Sensor 2    1203
Analog Sensor 3    1208
Analog Sensor 4    1283
Analog Sensor 5    1258
Digital Sensor 0   0
Digital Sensor 1   0
Digital Sensor 2   0
```

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

Use the Sensor Graph







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

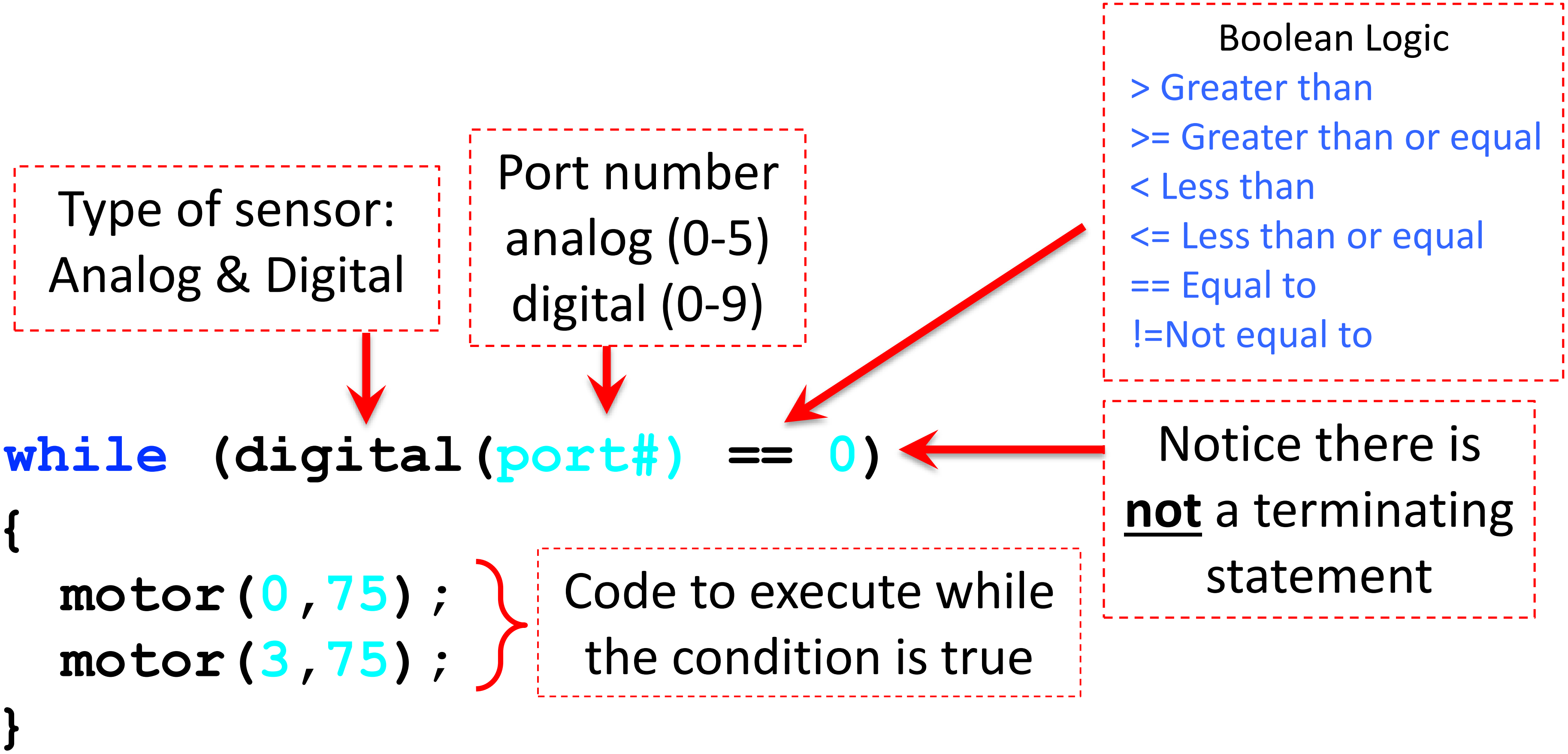
We accomplish this loop with a while statement.

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

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

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

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

Code to execute while the condition is true

Notice there is **not** a terminating statement

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

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 Wallaby that drives the DemoBot forward until a touch sensor is pressed, and then stops.

Analysis: What is the program supposed to do?

Pseudocode

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

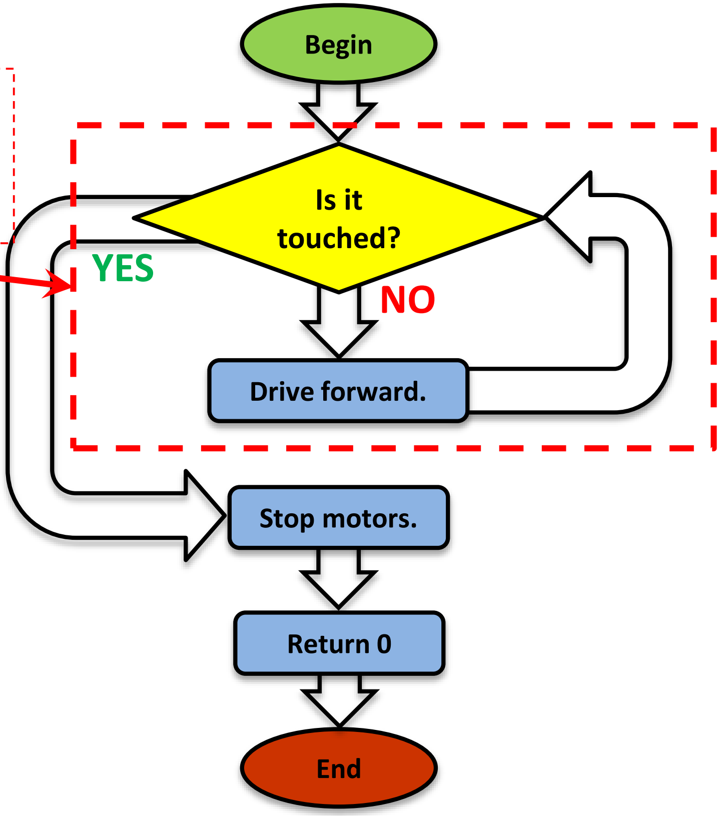
Comments

```
// 1. Drive forward.
// 2. Loop: Is not touched?
// 3. Stop motors.
// 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;
}
```

Changing the Condition

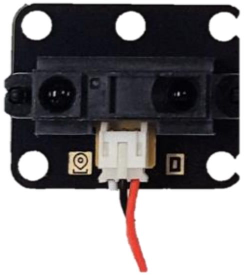
1. **Change the <u>expected</u> (test condition) value from 0 to 1**
2. Objective: Predict/describe what the robot will do
3. Run the program

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

   ao();
   return 0;
}
```

Learning about Analog Sensors

- Returns the analog value of the port (a value in the range 0-4095). Analog ports are numbered 0-5.
- Light sensors, slide, range finders and reflectance are examples of sensors that are used in analog ports.

Slide Sensor



Range Finder

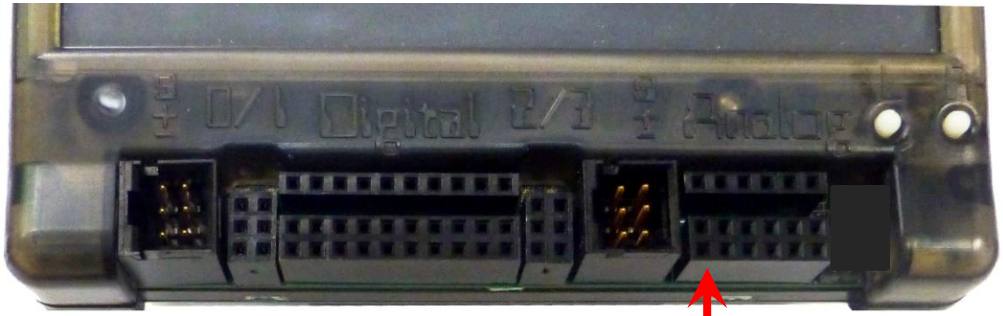
Small IR Reflectance Sensor

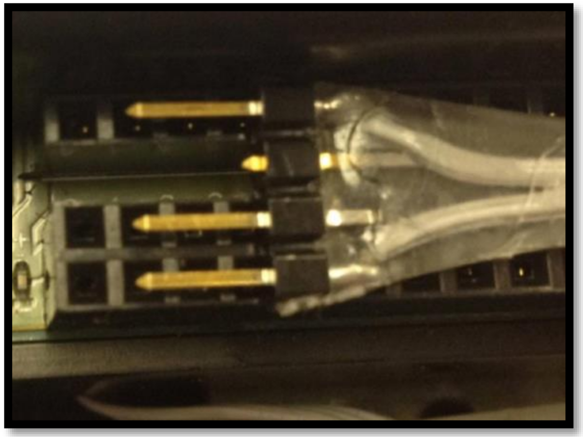
Light Sensor

Measuring Distance

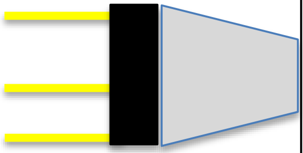
Infrared range finder sensor

Plug in the Range Finder Sensor

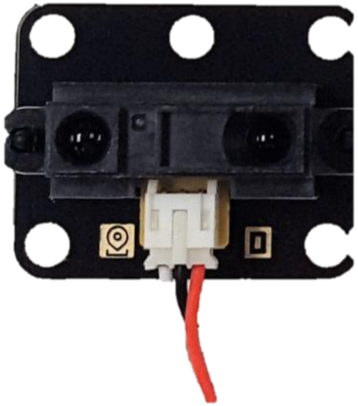




Sensor plug orientation

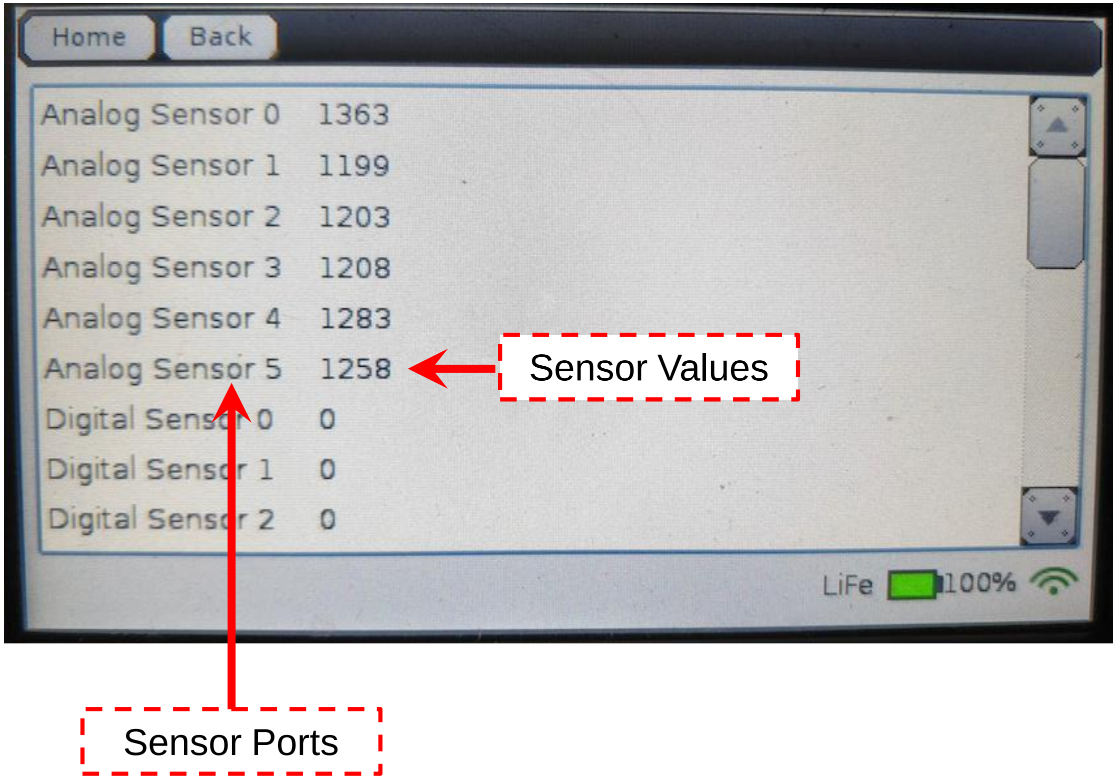


Plug analog sensor into analog port 0

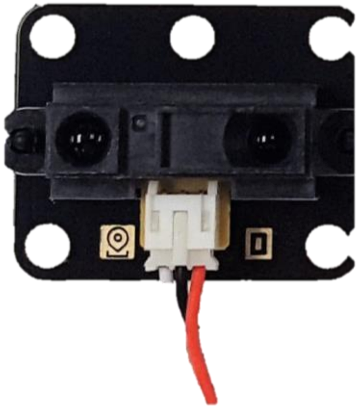


Range Finder

Check Range Finder Sensor on Wallaby Screen



```
Analog Sensor 0    1363
Analog Sensor 1    1199
Analog Sensor 2    1203
Analog Sensor 3    1208
Analog Sensor 4    1283
Analog Sensor 5    1258
Digital Sensor 0   0
Digital Sensor 1   0
Digital Sensor 2   0
```



Range Finder

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

Range Finder Sensor Information

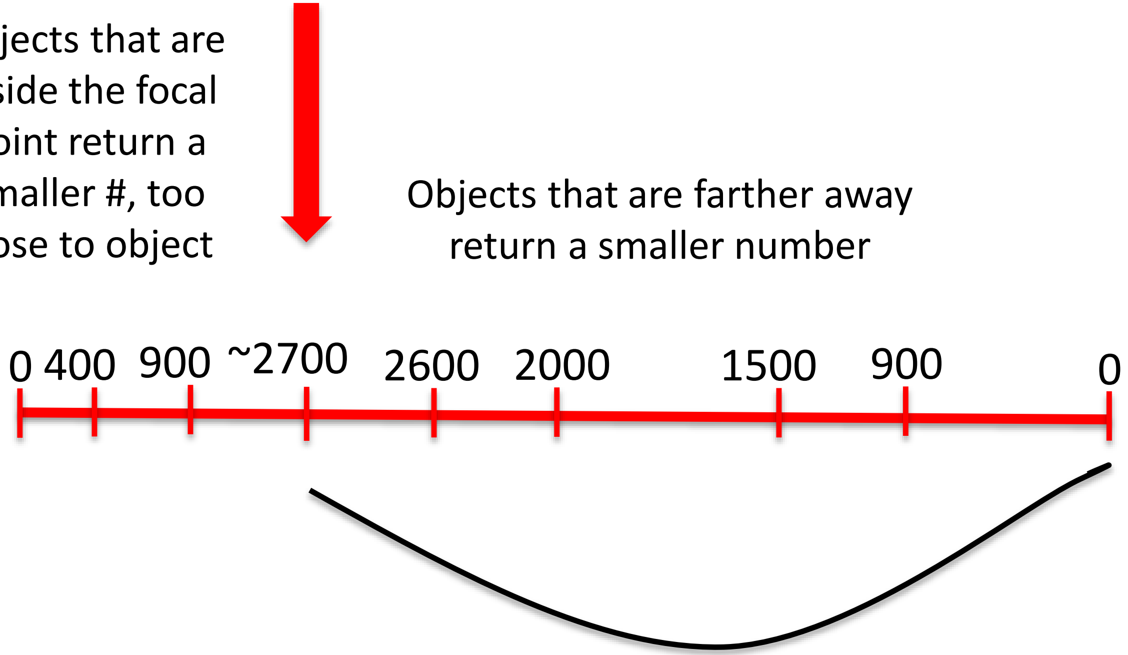
- **Low values:** indicate greater distance (farther from robot)
- **High values:** indicate shorter distance (closer to robot)
- Optimal range is ~4” and up
- 0” to 3.5” values are not optimal.
- Objects closer than the focal point (~4”) will have the same readings as those far away.

Range Finder Sensor Values

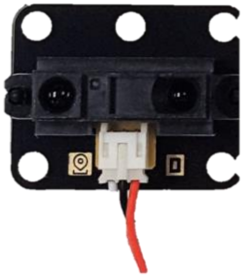
Focal Point

Objects that are inside the focal point return a smaller #, too close to object

Objects that are farther away return a smaller number



Useful range of the sensor



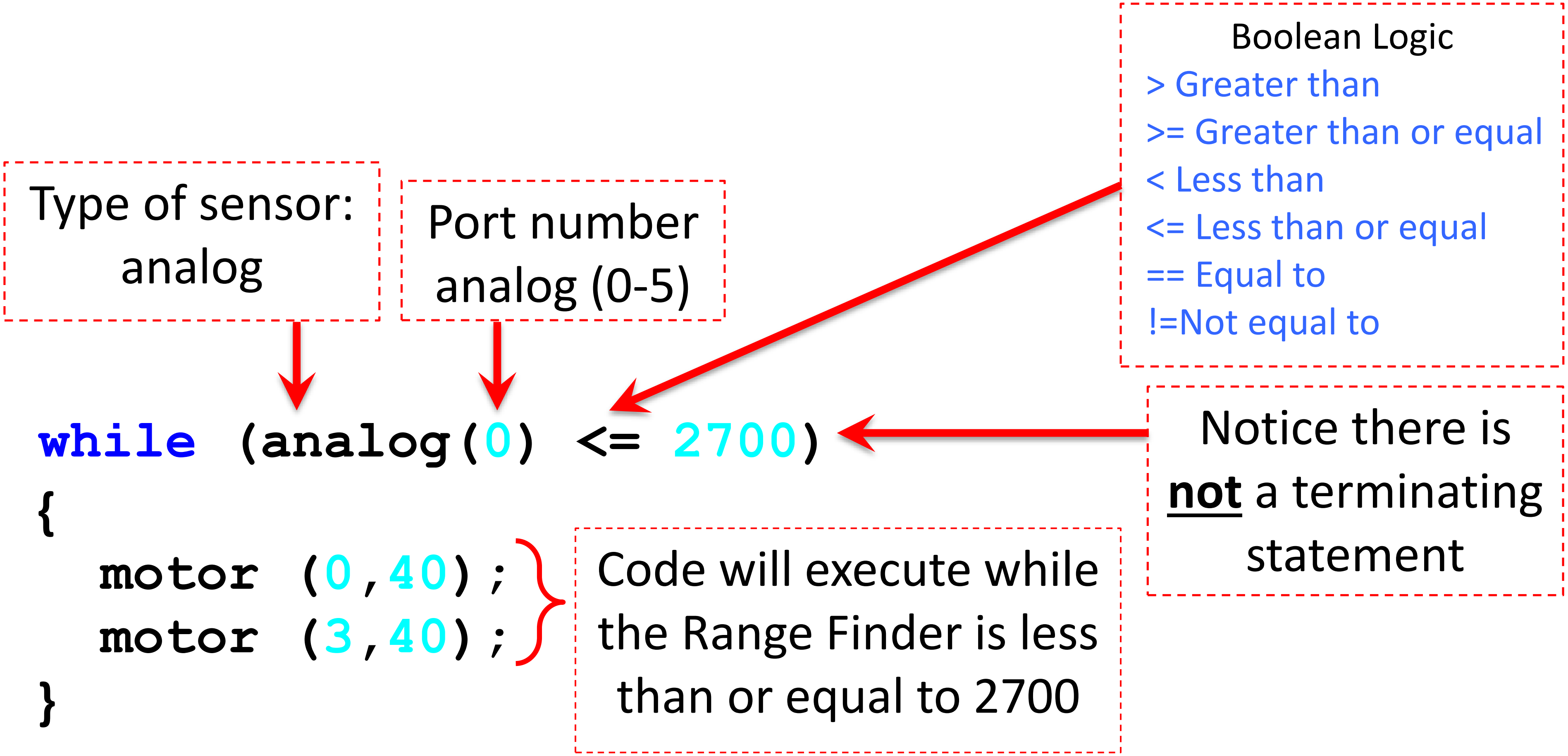
The value chosen may need to be adjusted up or down a little to get the desired distance from an object. Optimal distance is about 4.5” away from the sensor.

Range Finder Sensor Focal Point Problem

Use the sensor value to 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 Range Finder 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 the Range Finder sensor to the robot consider the ~4” distance from the sensor to its focal point

Learning to Use the Range Finder Sensor



Type of sensor: analog

Port number analog (0-5)

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

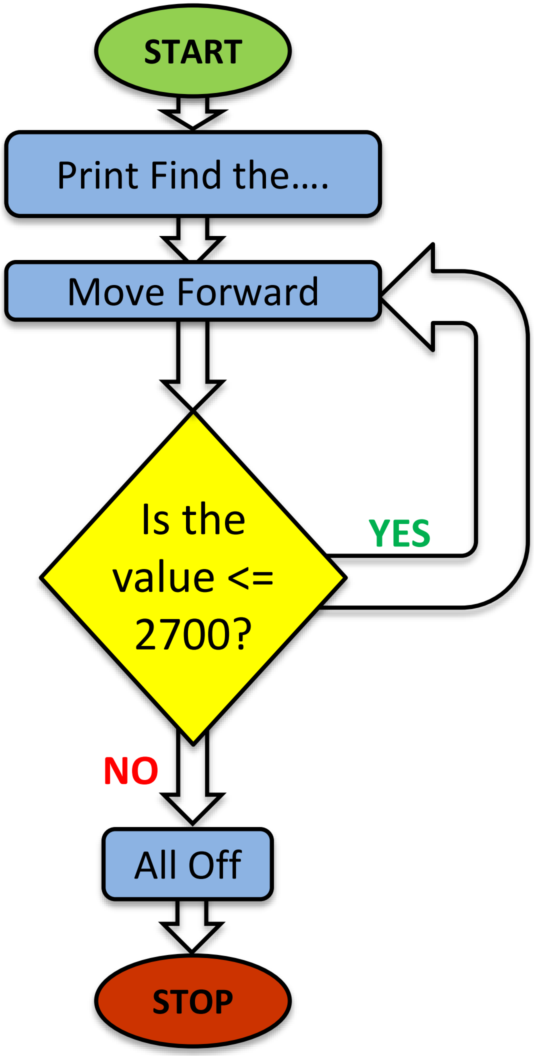
```
while (analog(0) <= 2700)
{
   motor (0,40);
   motor (3,40);
}
```

Notice there is **not** a terminating statement

Code will execute while the Range Finder is less than or equal to 2700

Find the Wall

1. Create a new project named “Find the Wall”.
2. Write and compile a program that will find the wall and stop.



Pseudocode

```
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 a value determined earlier)
4. Exit loop when value is greater than 2700
5. Shut everything off
```

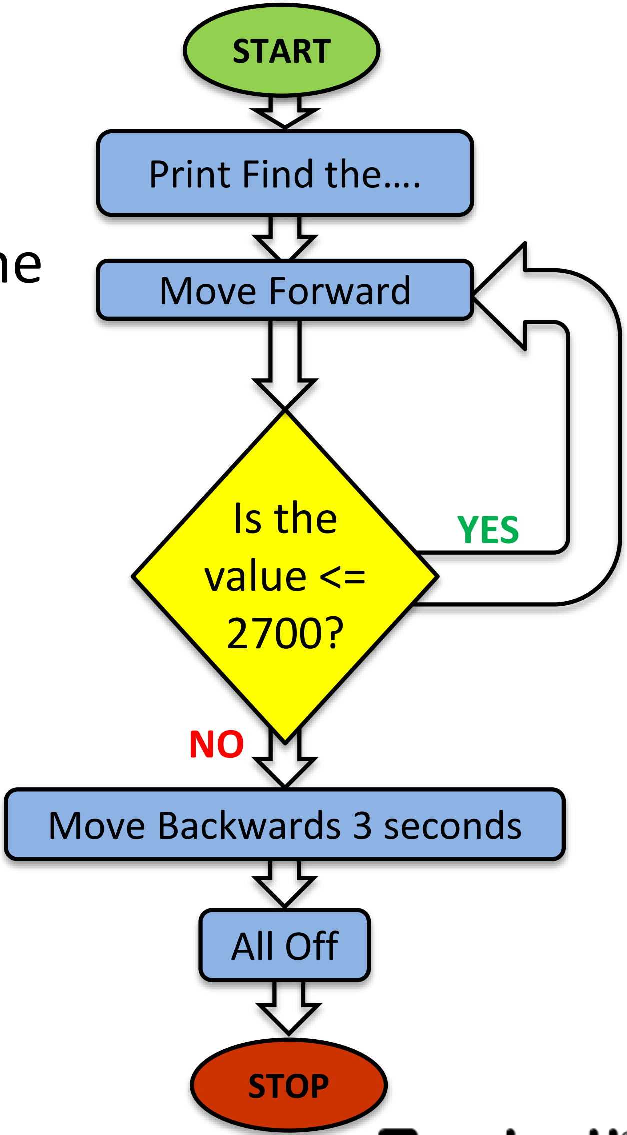
while “Find the Wall” Solution

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

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

  ao();
  return 0;
}
```

Find the Wall and Back Up

Pseudocode

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 a value determined earlier)
4. Exit loop when value is greater than 2700
5. Back up for 3 seconds
6. Shut everything off



Analog Sensor: Small Top Hat Sensors

This sensor is really a short range reflectance sensor. 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.

This sensor is excellent for line following

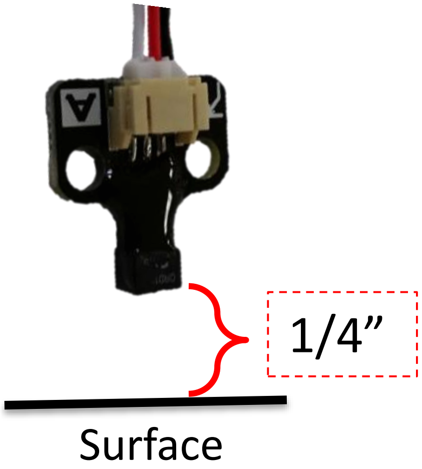
Black materials typically absorb IR and reflect very little IR and white materials typically absorb little IR and reflect most of it back

- ***<u>If this sensor is mounted at a fixed height above a surface</u>***, it is easy to distinguish a black surface from a white surface
- Connect to analog port 0 through 5

Reflectance Sensor Ports

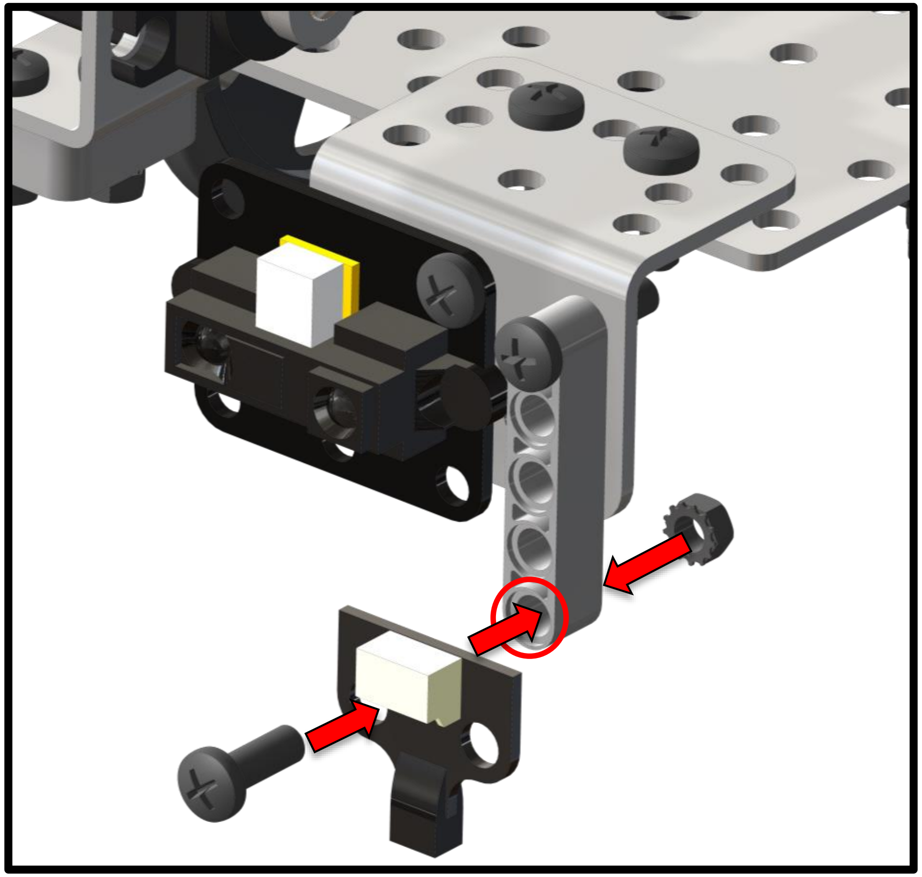
This is an **`analog()`** sensor, so plug it into any of the analog ports (0-5)

- Values can be between 0 and 4095
- Mount the sensor on the front of the robot so that it is pointing to the ground and ~1/4” from the surface



Mounting Sensor on DemoBot

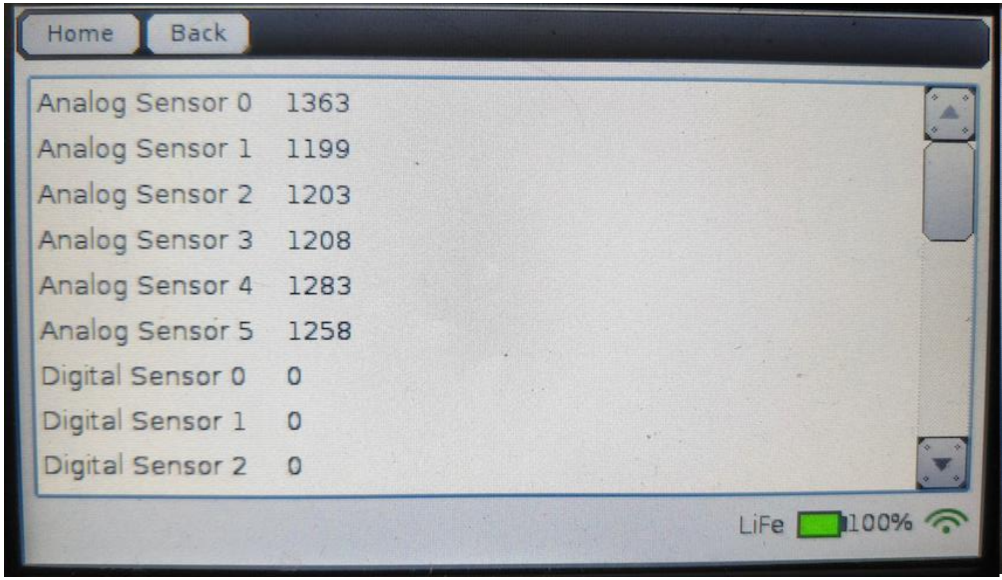
The small top hat (reflectance) sensor works best if mounted ~1/8 to ~1/4 inch off the surface such that the distance to the ground does not vary much/at all while the robot moves.



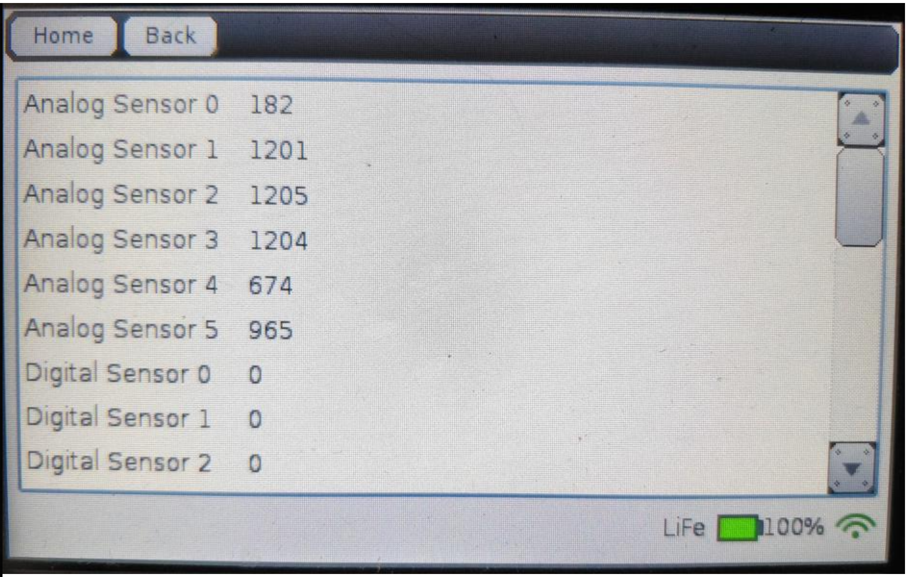
Reading Sensor Values From the Sensor List

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)



Analog Sensor 0	1363
Analog Sensor 1	1199
Analog Sensor 2	1203
Analog Sensor 3	1208
Analog Sensor 4	1283
Analog Sensor 5	1258
Digital Sensor 0	0
Digital Sensor 1	0
Digital Sensor 2	0



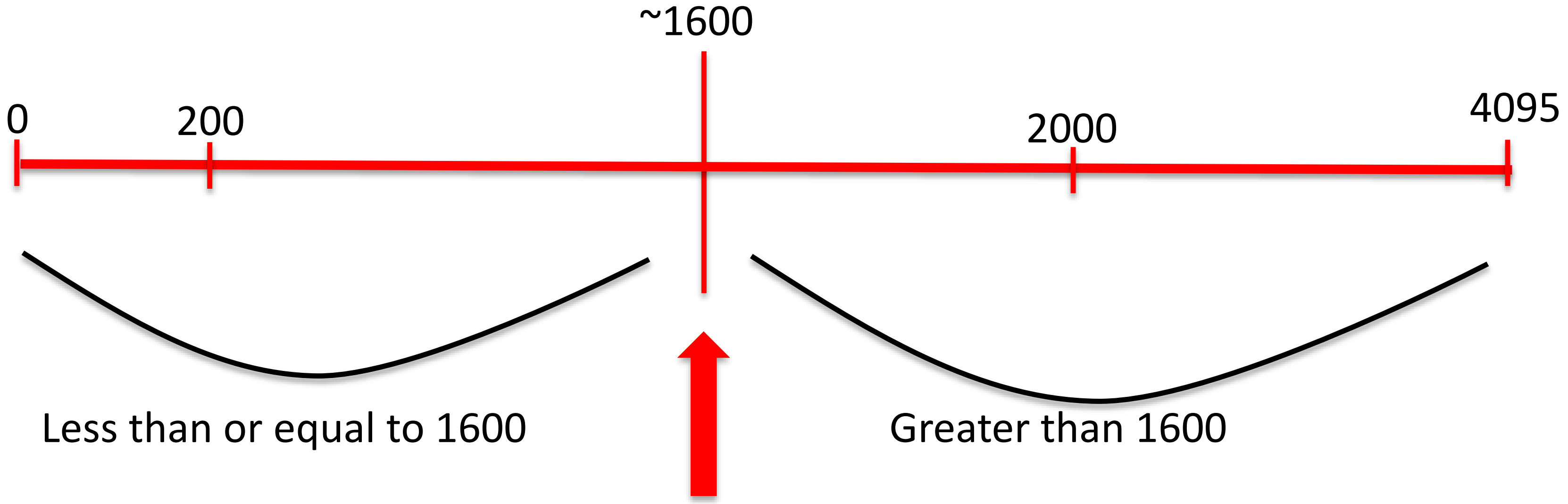
Analog Sensor 0	182
Analog Sensor 1	1201
Analog Sensor 2	1205
Analog Sensor 3	1204
Analog Sensor 4	674
Analog Sensor 5	965
Digital Sensor 0	0
Digital Sensor 1	0
Digital Sensor 2	0

The IR sensor is correctly mounted when the values are between 2900 and 3100 on a black surface

The IR sensor is correctly mounted when the values are between 175 and 225 on a white surface

Understanding the IR Values

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

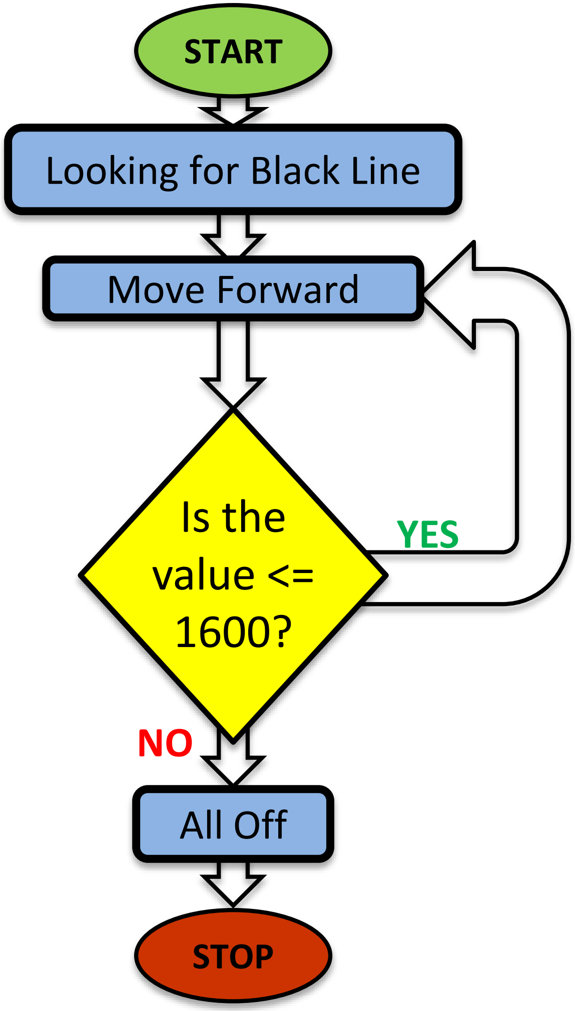


The black ***threshold*** value is ~1600

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
```



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);
   }

  ao();
  return 0;
}
```

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 can be used to calculate the distance traveled by the robot!

Motor Port #
(#0 – 3)

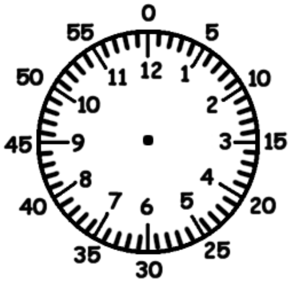
```
get_motor_position_counter(0)          — OR —          gmpc(0)
// Tells us the number of ticks the motor on port #0 has rotated.
```

Motor Port #
(#0 – 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!

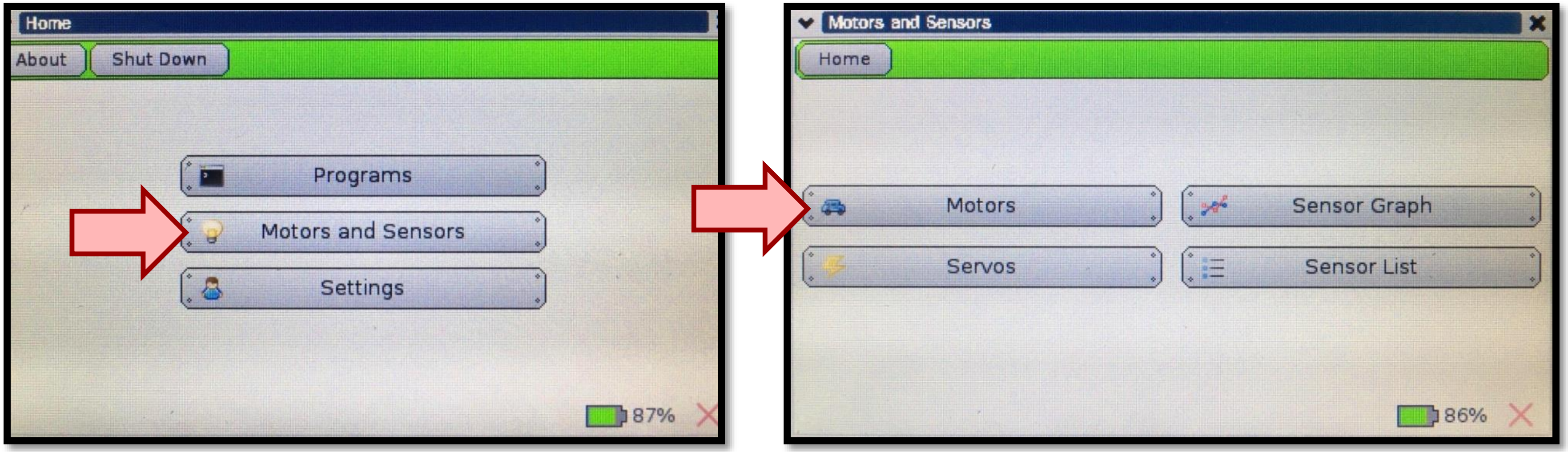
Similar to how a clock is divided into 60-second intervals (ticks).



Seeing Counters on Wallaby

Access the Motors from the Motors and Sensors screen

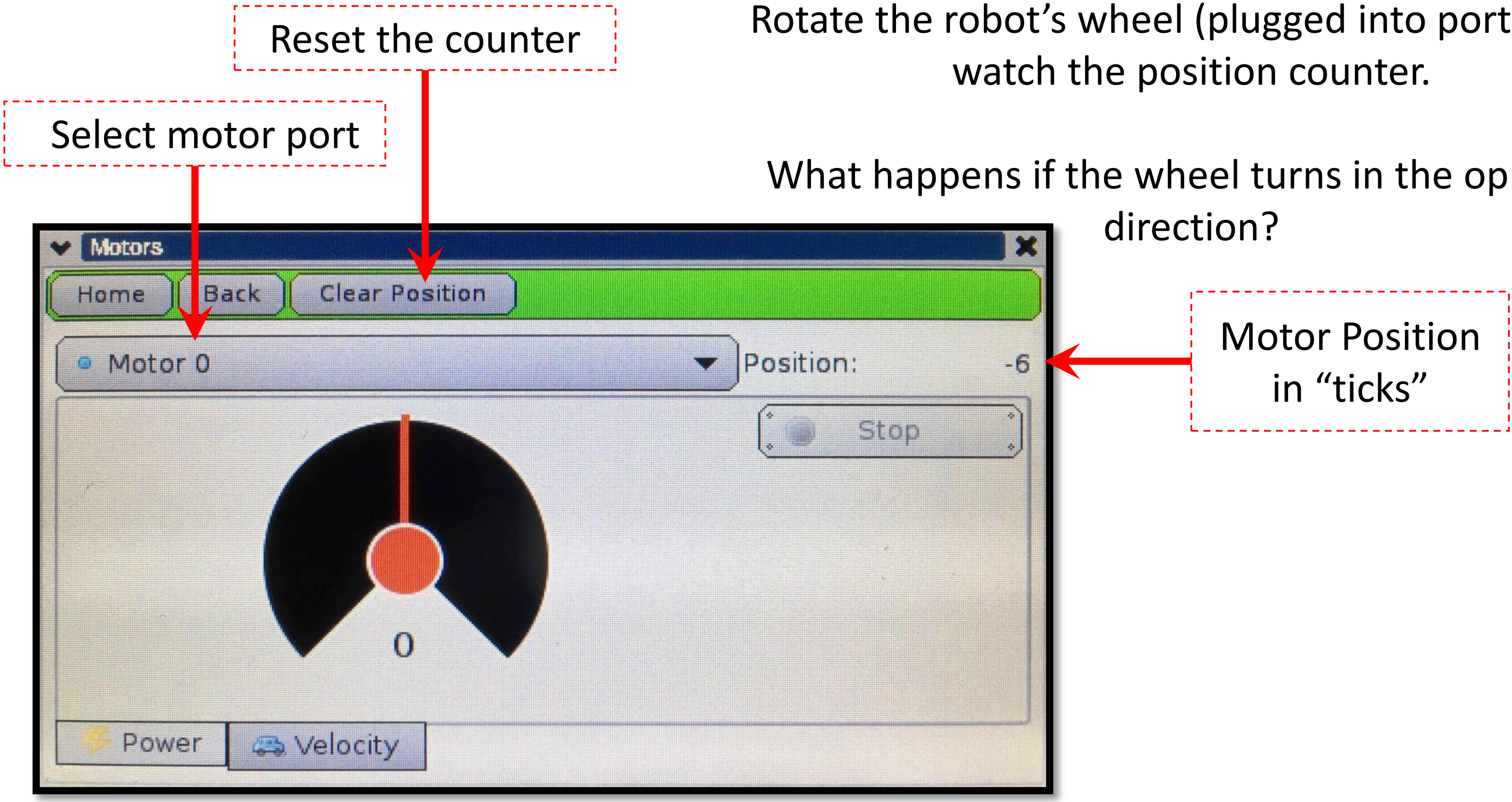
- This is very helpful to test the motors and see the actual motor position counters “*in action*”



Seeing Counters on Wallaby

Rotate the robot's wheel (plugged into port 0) and watch the position counter.

What happens if the wheel turns in the opposite direction?



An alternative is to place the robot on a surface and roll it forward to measure the number of ticks from a starting position to another location or object.

Drive to a Specific Point

Place the robot on a surface and roll it forward to measure the number of ticks from a starting position to another location or object.

Place the robot in the *start box* of **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 a program to drive the DemoBot forward that many “ticks” and then stop.

Pseudocode

Try making it!

Drive to a Specific Point

Solution:

Pseudocode

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

Source Code

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

  cmpc(0);

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

  return 0;
}
```

Drive to a Specific Point

Reflection: What can be noticed after the program was ran?

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

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

- How could the program be modified 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 a program to drive the DemoBot forward to a specific point and then back up to where it 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();

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

  return 0;
}
```

Connections to the Game Board

Description: Navigate to and manipulate game pieces utilizing sensors and motor position counter.

Goal #1: Mat A – 2” block will be set on circle 4, 6, or 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 1” block on coordinates A12. Driving using motor position counter, pick up the 1” block and set it in the yellow garage. Robot or game pieces may not cross solid lines of targeted garage. *Bonus*: *Set 1” blocks on A6, A12, and A18. One by one pick them up, and deposit all of them in the yellow garage.*

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.
```

Comments

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

Fun with Functions

Writing new functions

Function prototypes, definitions, and calls

Writing Custom Functions

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

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

- **Functions** are very helpful if an action is repeated 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();`

These are made up… and that's the point!

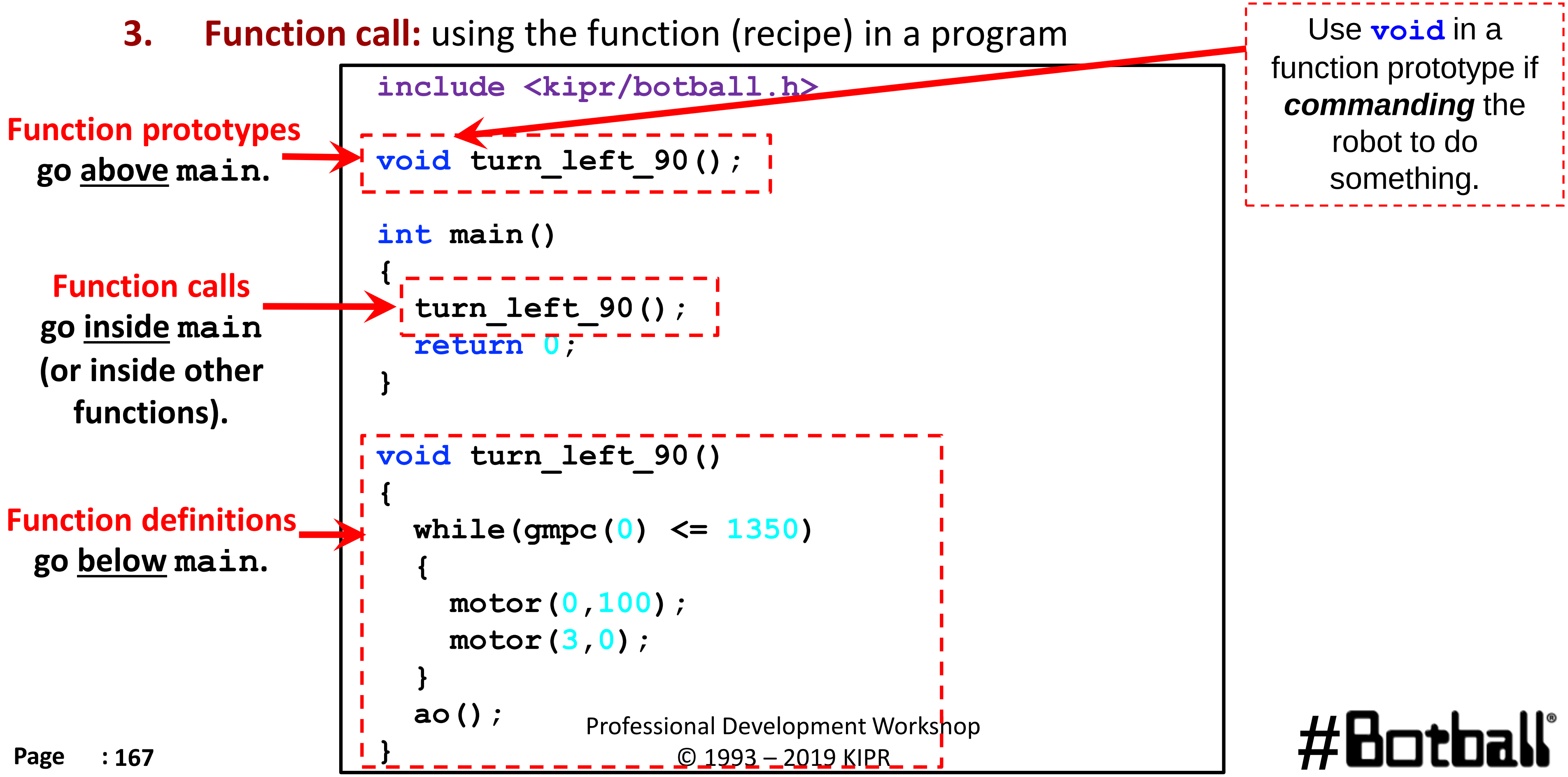
Write functions to do whatever is needed!

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

Writing Custom Functions

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

1. **Function prototype:** a *promise* to the computer that the function is defined somewhere (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 a program



Use `void` in a function prototype if ***commanding*** the robot to do something.

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

Function calls go inside `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();
}
```

Writing Custom Functions

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

prototype →

```
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();
}
```

definition →

Notice: no semicolon!
(Why not?)

Writing Custom Functions

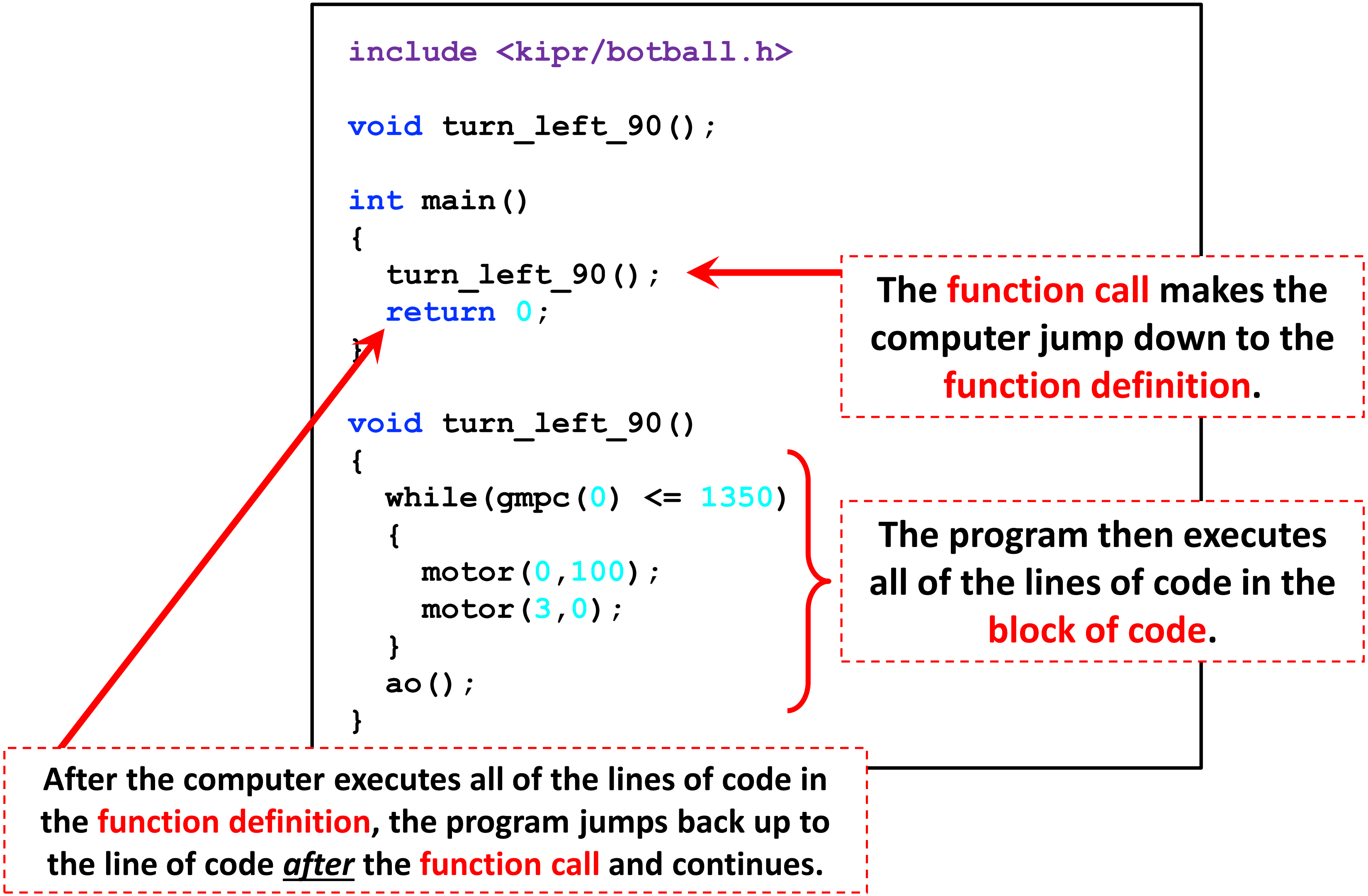
```
include <kipr/botball.h>

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 it 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…

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();
}
```

Connections to the Game Board

Description: Write some custom function to navigate the robot using motor position counter. All movement must be completed using custom functions.

Goal #1: Mat A– Drive around the green garage and return to the start box. *Bonus: Place a 2" block on circle 7. Adding to the previous program, once the cube is sensed, pick it up and navigate back to the start box using the same parameters except deposit the block in the start box.*

Goal #2: Mat A– Start in the start box and navigate to, and park, in the orange garage. No part of the robot may cross the solid boundaries of the orange garage.

Making a Choice

Program flow control with conditionals

`if-else` conditionals

`if-else` and Boolean operators

Using `while` and `if-else`

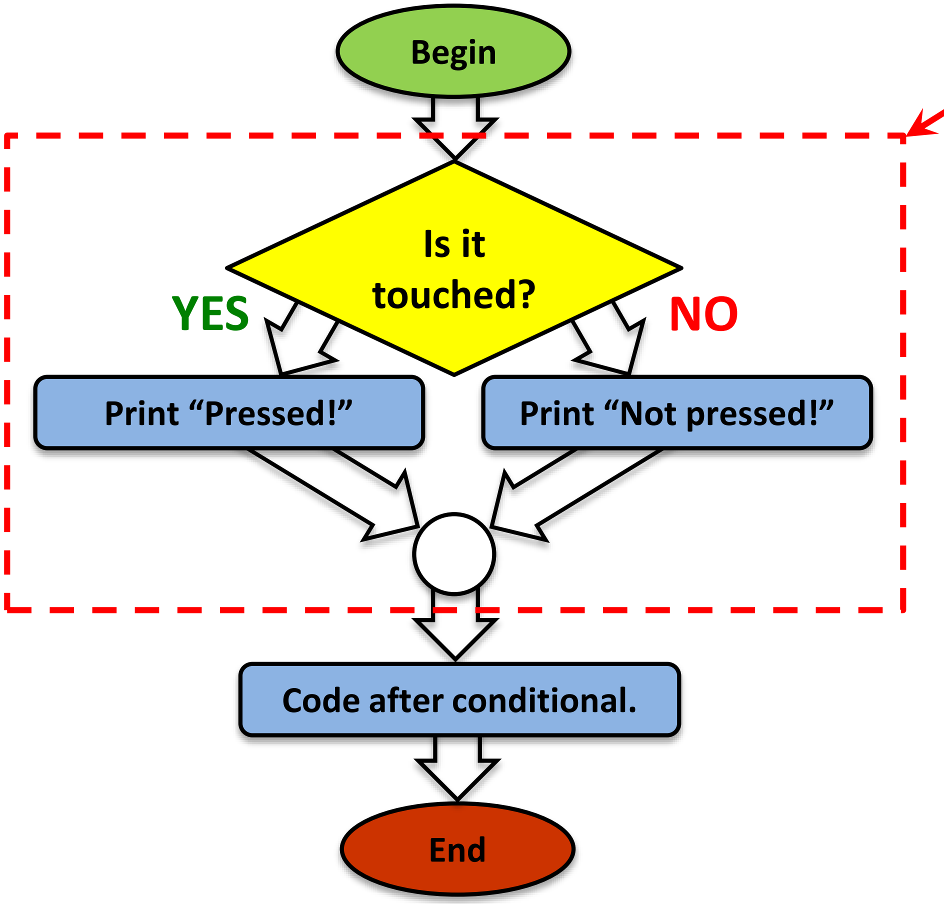
Program Flow Control with Conditionals

- 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

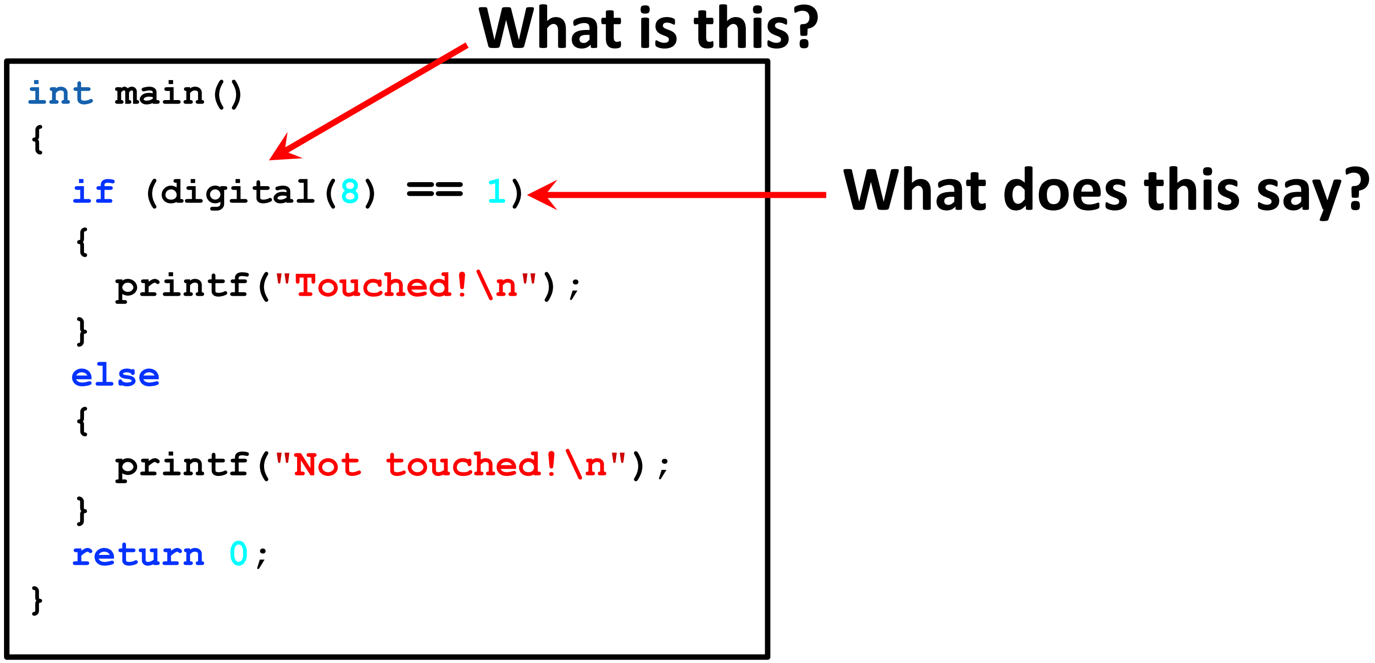
This part of the code is the **conditional**.



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");
  }
  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

Conditionals can be placed inside of loops. This is beneficial when wanting to keep checking a set of conditions over and over, instead of just a single time.

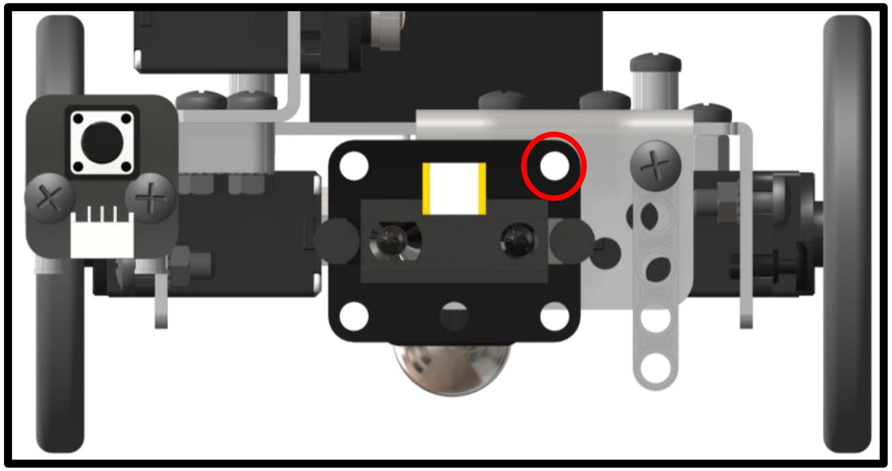
What should go inside the condition for the while loop?

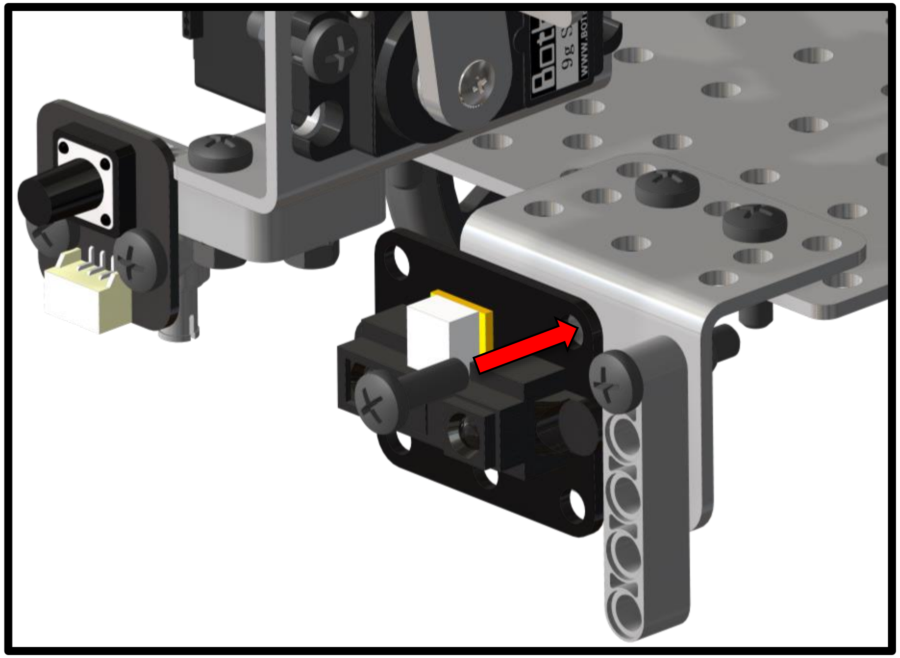
```
int main()
{
  while (digital(0) == 0)
  {
    if (analog(0) > 1600)
    {
      printf("It's dark in here!\n");
    }
    else
    {
      printf("I see the light!\n");
    }
  } // loop ends when button is pressed
  return 0;
}
```

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

Mounting the Range Finding sensor

Generally this sensor should be mounted to the front. Ideal use of this sensor is for when the robot is 4-inches or less away from the target object. Note that this mechanical example shown below is a quick solution, not a game winning solution.

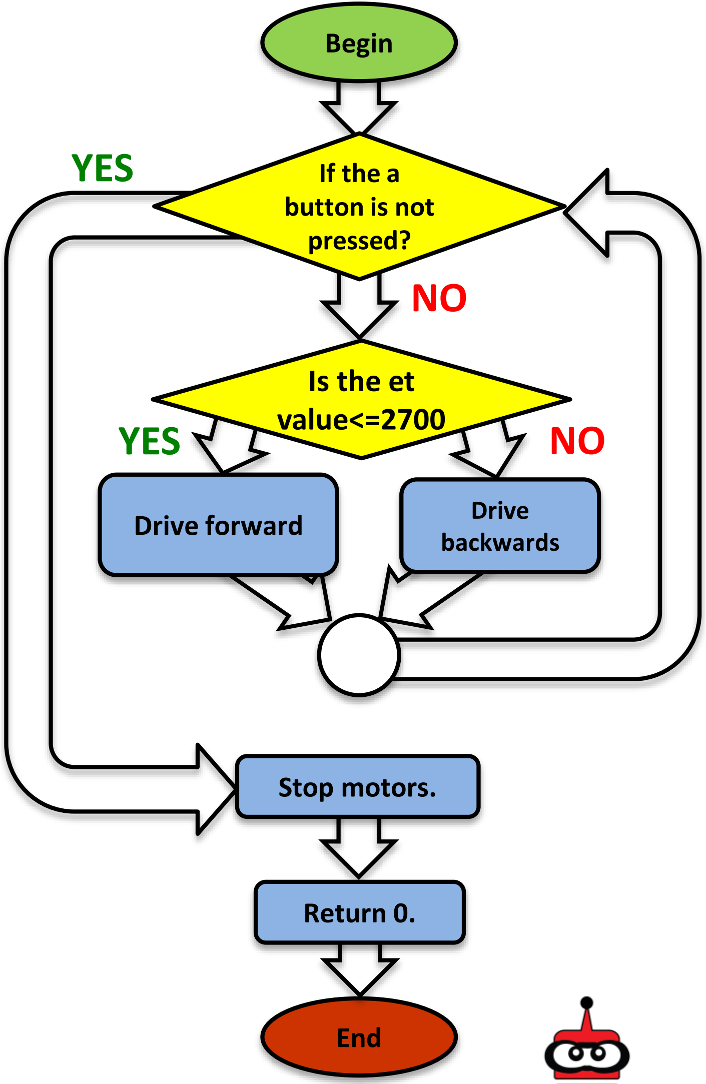




Find the Wall and Back Up then Drive forward

Pseudocode

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



Range Finder Find the Wall and Back Up then Drive forward

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

  while (digital(0) == 0)  // Touch sensor not touched
  {
     if (analog(0) <= 2700) // Far away drive forward
     {
       motor(0,80);
       motor(3,80);
     }
     if (analog(0) > 2701) // Too close back up
     {
       motor(0,-80);
       motor(3,-80);
     }
  }
  ao();
  return 0;
}
```

Maintain Distance

Description: Write a program for the KIPR Wallaby 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

This activity requires a **reflectance sensor**.

- This sensor is really 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!

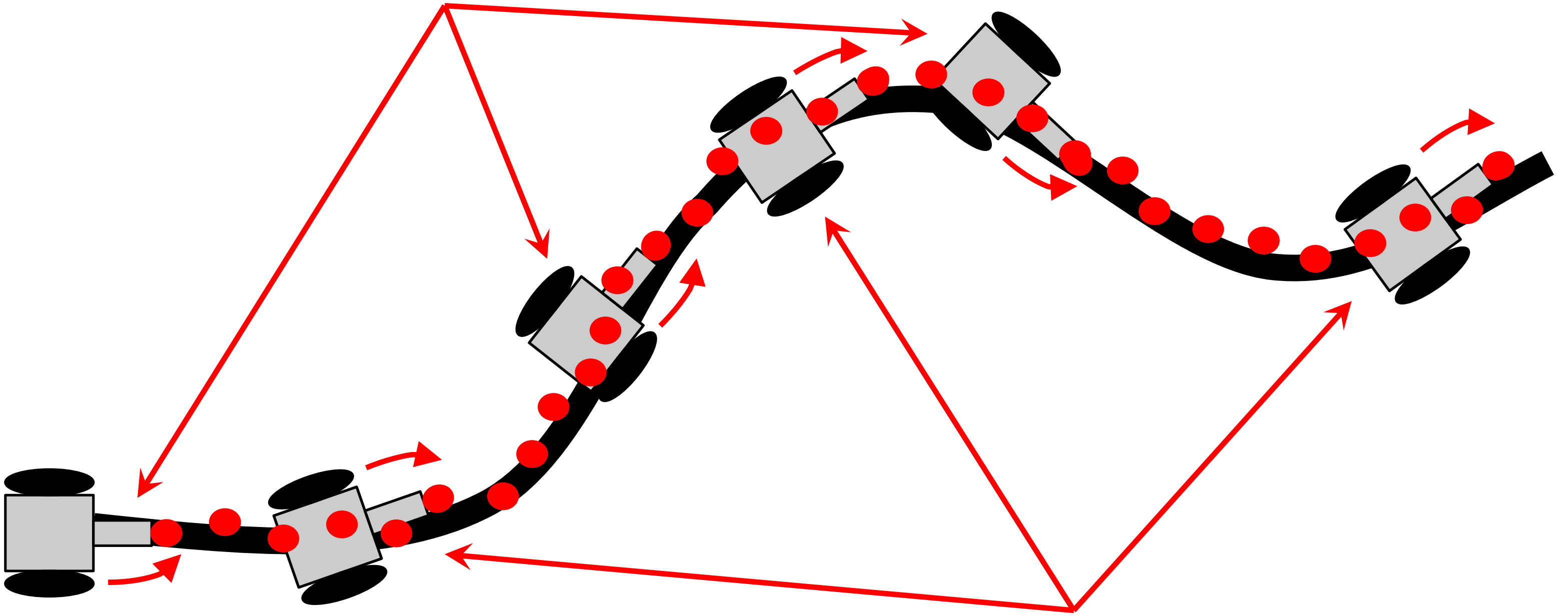
- **Black materials** typically **absorb most IR** → they **reflect little IR back**!
- **White materials** typically **absorb little IR** → they **reflect most IR back**!
- If this sensor is mounted at a *fixed height* above a surface, it is easy to distinguish a black line from a white surface.

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:

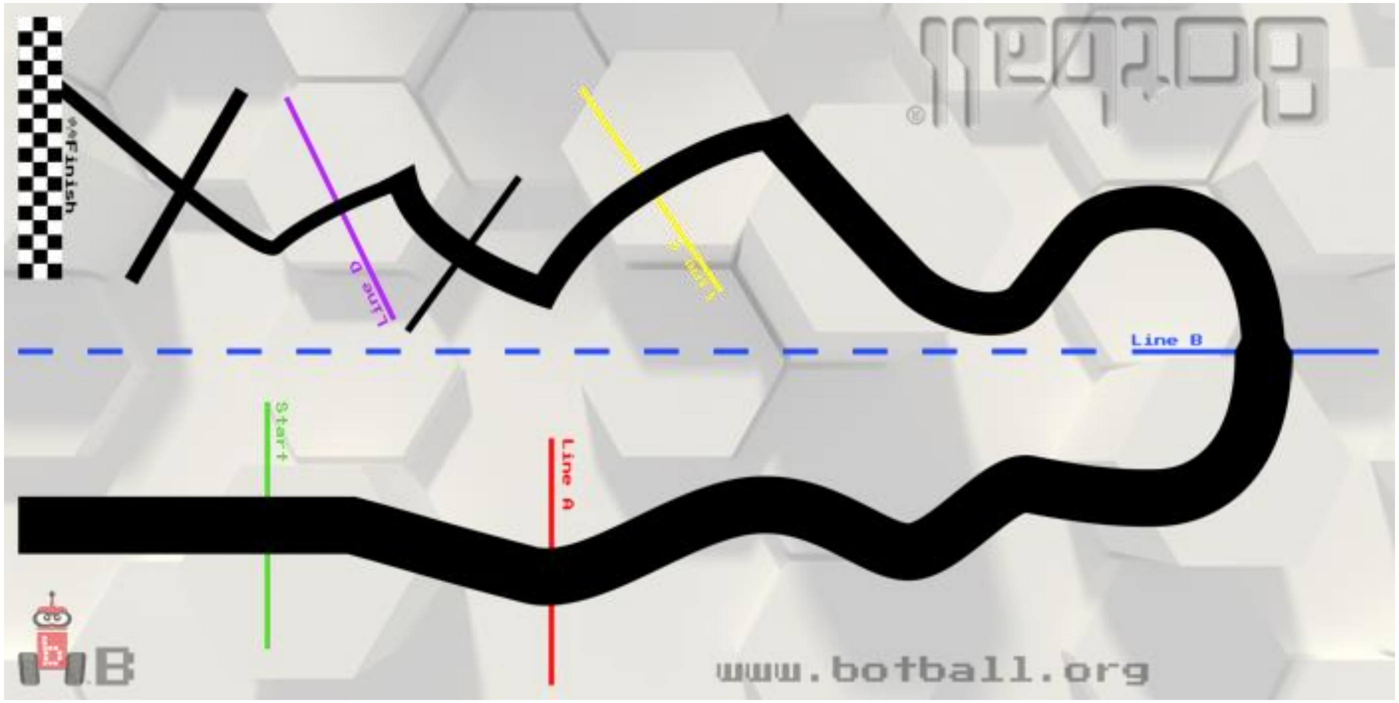
1. **if** detecting dark, arc/turn right



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

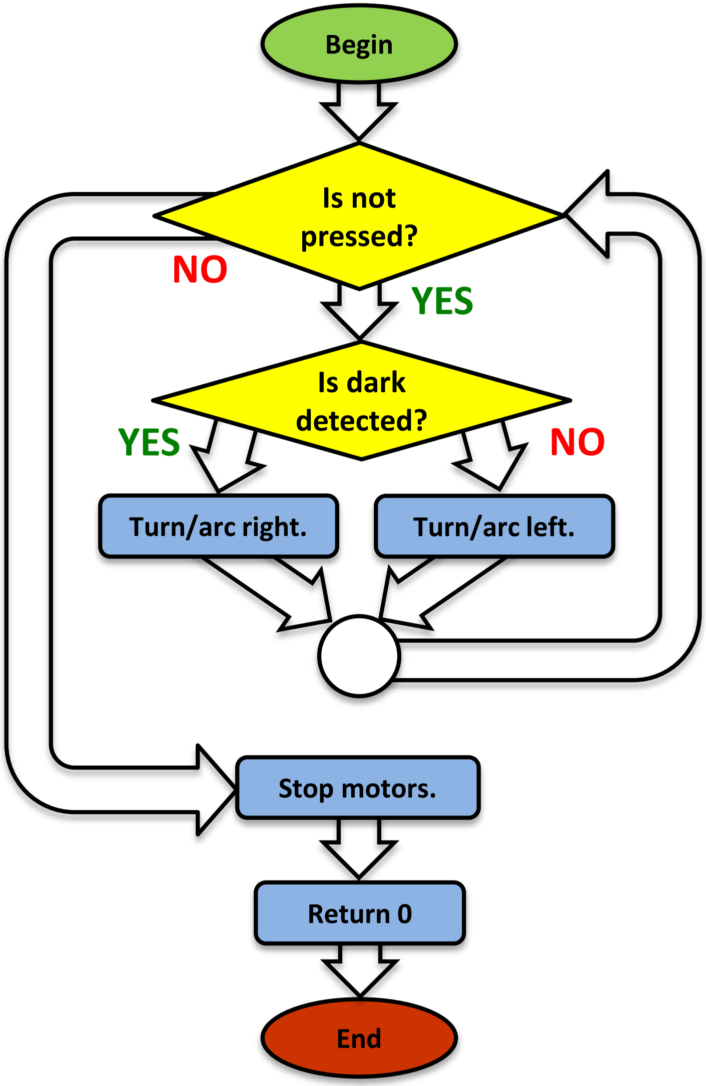
Line-Following

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

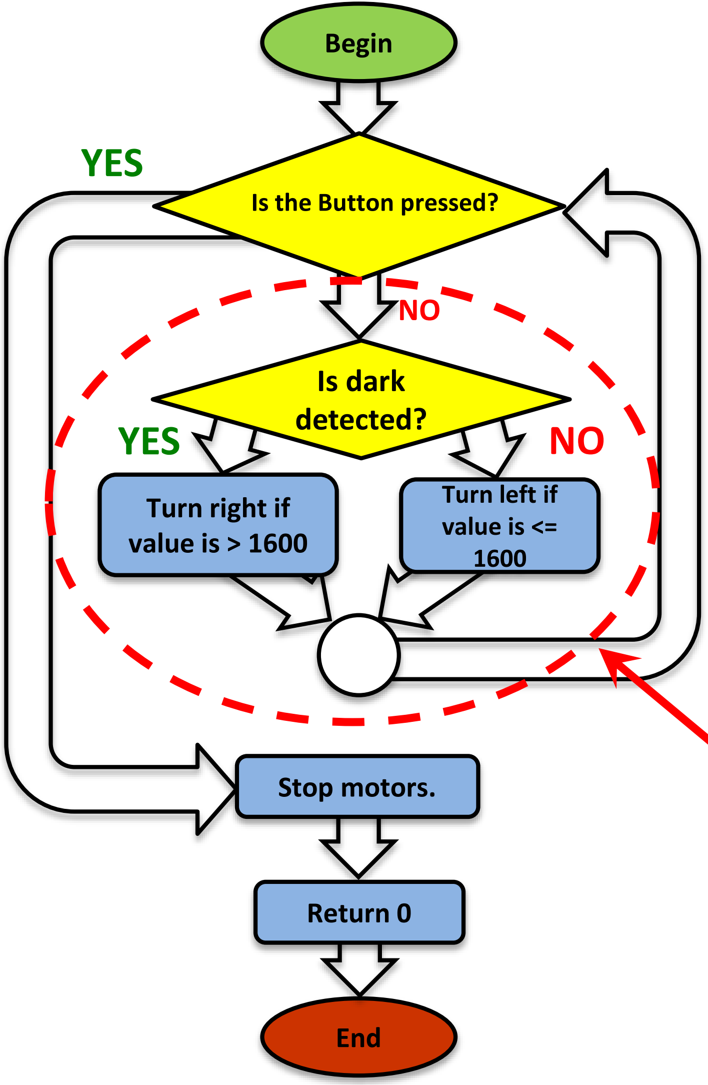


Line-Following

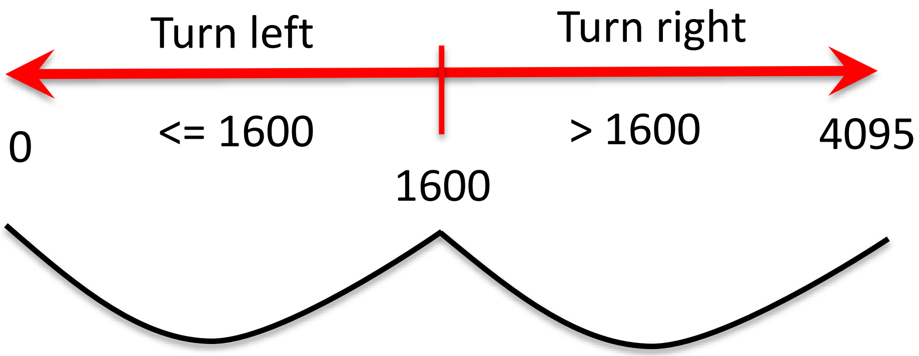
Analysis: Flowchart



Understanding while and if



It 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 Wallaby what to do when it sees black or white.

Line-Following

Solution:

Pseudocode

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

Source Code

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

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

  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, 5);
            motor(3, 90);
        }
        else
        {
            motor(0, 90);
            motor(3, 5);
        }
    }
    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 so the value is lower.

Notice the Boolean operators > 1600 or <= 1600

The value may be much lower due to lighting, placement of sensor and arc speed.

Also increasing the “arc speed” (by making the ***difference*** between the two motor power values greater) may have a significant impact.

Line-following with Functions

Solution:

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

```c
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, 10);
  motor(3, 80); // Turn/arc left.
}

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

Homework

Game Review

Game Strategy

Workshop Survey

Homework for Tonight:
Game Review

Visit www.KIPR.org/Botball

Review the game rules under the Team Home Base tab.

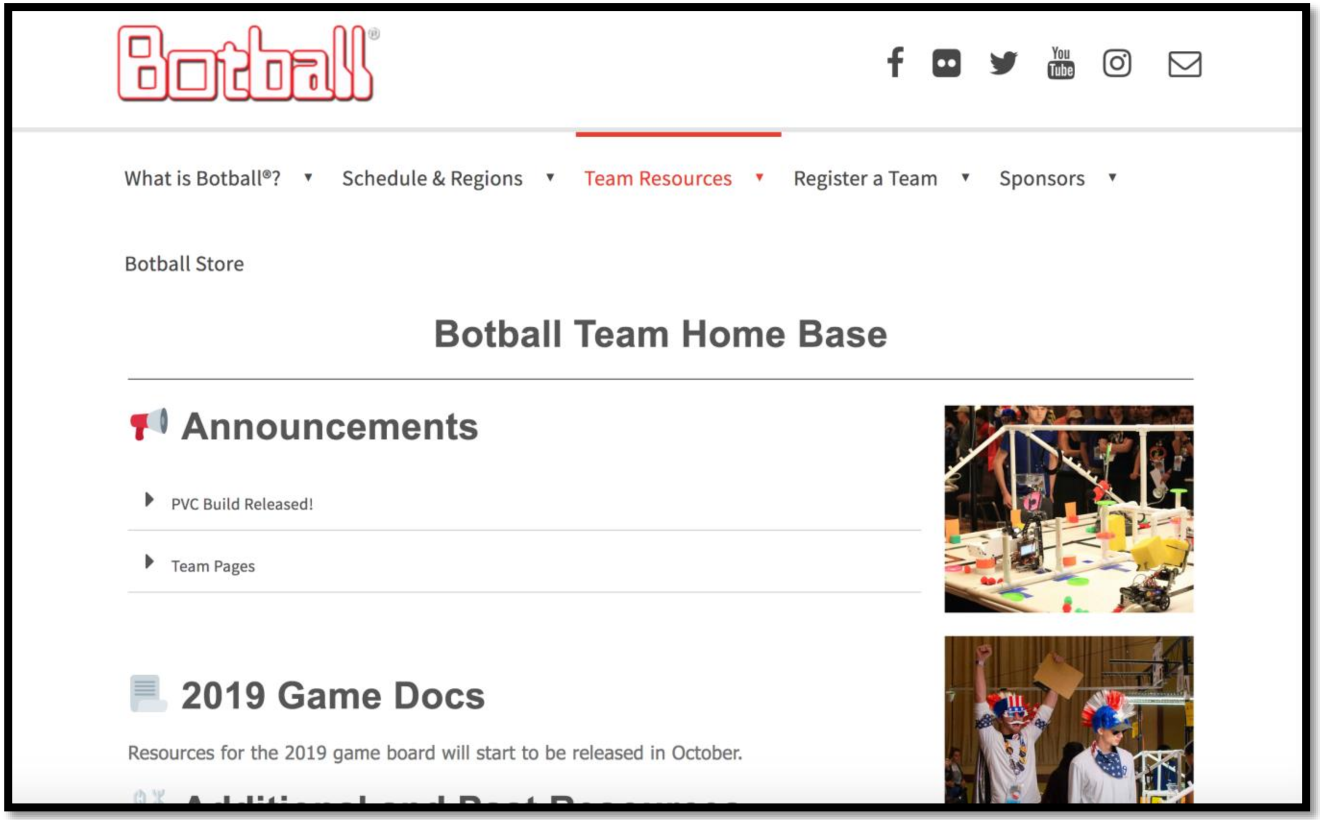
- We will have a **30-minute Q&A session** tomorrow.
- After the workshop, ask questions about game rules in the **Game Rules FAQ**.
  - **Regularly visit this forum.**
  - **Answers to questions will be found there.**

Homework for Tonight:
Game Strategy

- Break down the game into subtasks!
- Write **pseudocode** and/or create **flowcharts**!
- Start with **easy points**—score early and score often!
- Keep it simple and make sure it works.
- Discuss the strategy with the coach tomorrow.
- Think about the Engineering Design Process.

Have a Good Night!

Don’t forget to visit www.KIPR.org



What is Botball®? ▾ Schedule & Regions ▾ Team Resources ▾ Register a Team ▾ Sponsors ▾

Botball Store

Botball Team Home Base

Announcements

- PVC Build Released!
- Team Pages

2019 Game Docs

Resources for the 2019 game board will start to be released in October.

Additional and Past Resources

Welcome Back!

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/WQ8CQ65

While you wait:
Build the Create DemoBot

The build slides should be saved on your desktop.

Welcome Back!

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/WQ8CQ65

Botball 2019

Professional Development Workshop

Prepared by the KISS Institute for Practical Robotics (KIPR)

with significant contributions from KIPR staff

and the Botball Instructors Summit participants

While waiting, work on yesterday's exercises or build the Create DemoBot!

Workshop Schedule – Day 2

Day 1

- Botball Overview
- Getting started with the KIPR Software Suite
- Explaining the “Hello, World!” C Program
- Designing A Program
- Moving the DemoBot with Motors
- Moving the DemoBot Servos
- Making Smarter Robots with Sensors
- Introduction to while Loops
- Measuring Distance
- Motor Position Counter
- Fun with Functions
- Making a Choice
- Line-following
- Homework

Day 2

- **Botball Game Review**
- **Starting with a Light**
- **Tournament Code Template**
- **More Variables and Functions with Arguments**
- **Moving the iRobot *Create*: Part 1**
- **Moving the iRobot *Create*: Part 2**
- **iRobot *Create* Sensors**
- **Color Camera**
- **Logical Operators**
- **Resources and Support**

Botball Game Review

Game Q&A

Construction, documentation, and changes

`shut_down_in()` function

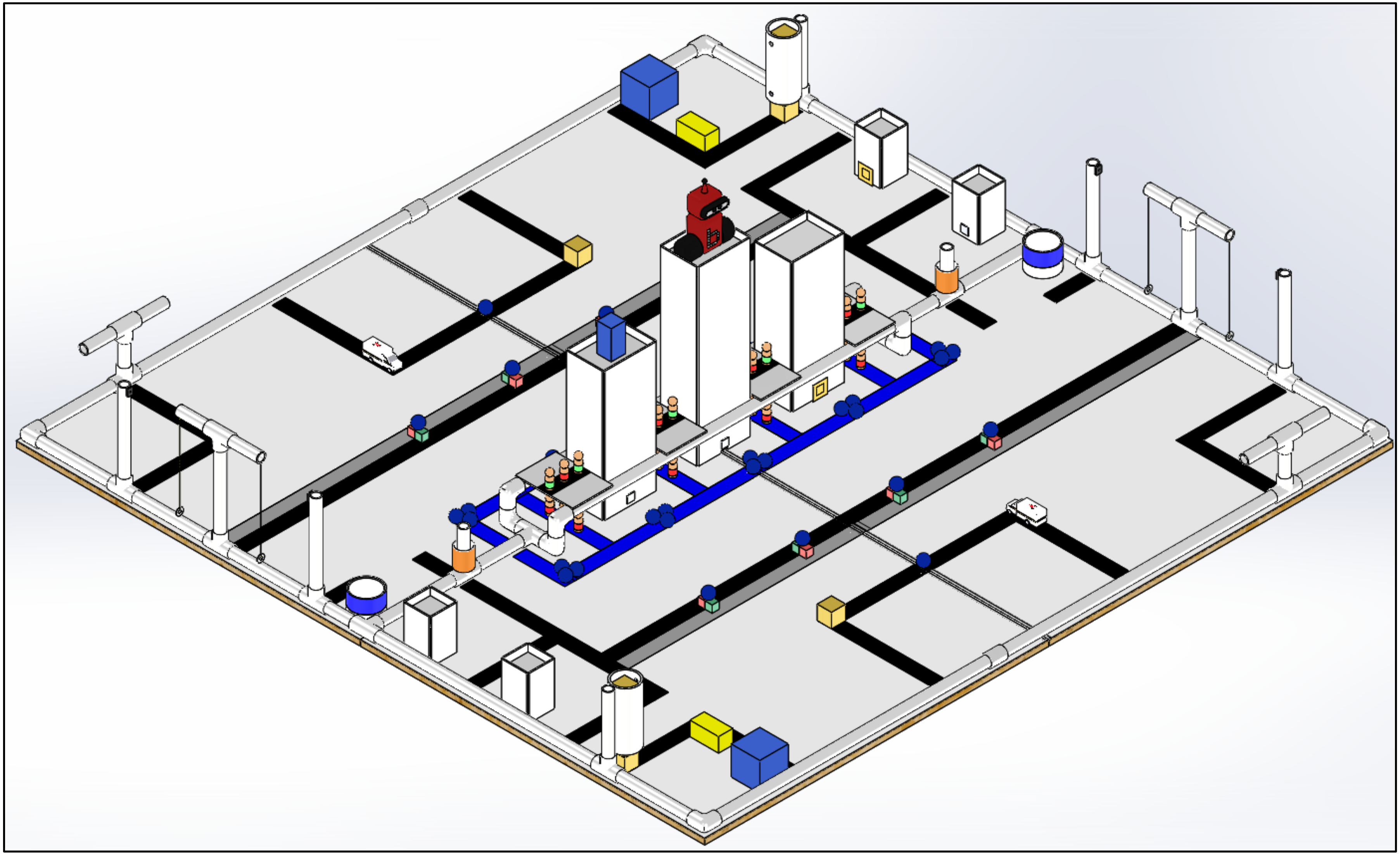
`wait_for_light()` function

Botball Game Q&A starts…

NOW!

You have 30 minutes…

Botball Game Board



Ideas on Construction

Note: Our competition tables are built to specifications with allowable variance.

- Do **NOT** engineer robots that are so precise that a 1/4" difference in a measurement means they are not successful.
  - For example: the specified height of the tram assembly is set to be 13" above the game surface, if the actual height was 13 ¼" off the surface, an effector with too low of a tolerance may fail to do it's job.
- Review construction documents (like the ones on the **Home Base**!) to get building ideas.
- Search the internet for robots and structures to get building ideas.
- Test structure robustness ***before*** the tournament!

Documentation

What?

- **Botball Online Project Documentation (BOPD)**
- Rubrics and examples are on the **Team Home Base**
- **NO NAMES OR SCHOOL NAMES ALLOWED ON SUBMISSIONS**

When?

- 3 document submissions during design and build portion
- 1 Onsite Presentation (8 minute) at regional tournament

Why?

- To reinforce the Engineering Design Process
- Points earned in **Documentation** factor into the overall tournament scores!

See BOPD Examples on the Team Homebase via Team Resources -> Team Homebase -> Team Submissions

Changes this Season

- See the Team Homebase for a document covering all changes made in regards to Hardware, Rules, the Wallaby, Software, and Documentation.
- Kit Parts – #2 New micro servo brackets
- Game Rules – Coins up to 250 grams (be prepared to have them weighed-make sure they can be easily removed)
- Resources – other updates can be found online: www.KIPR.org/Botball

Starting Programs with a Light

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

- The `wait_for_light()` function allows the program to run when the 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 the activities, use a flashlight.

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

wait_for_light() Function

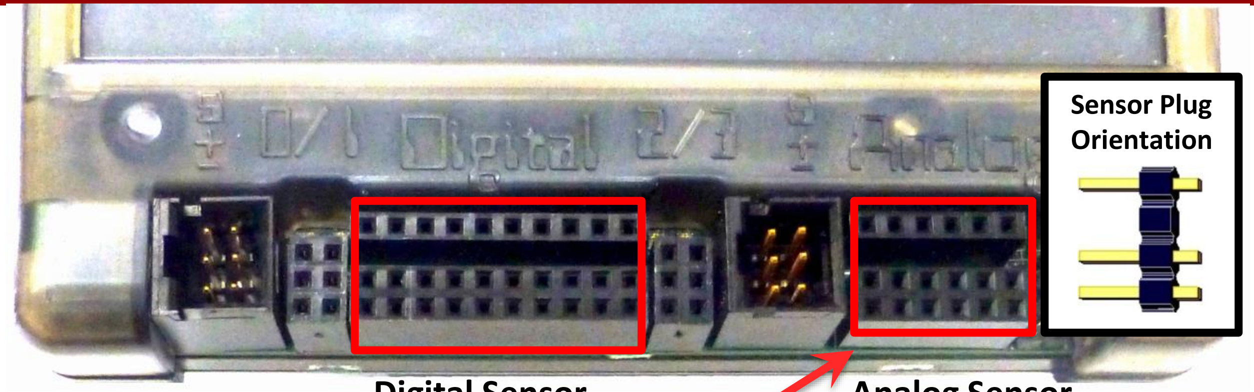
```
wait_for_light(0);
// Waits for the light on port #0 before going to the next line.
```

Review: What is this?

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

Plug in the Light Sensor

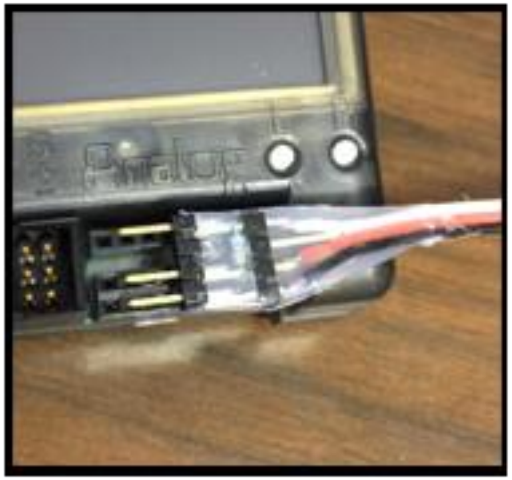
(Light source needed, cell phone works)

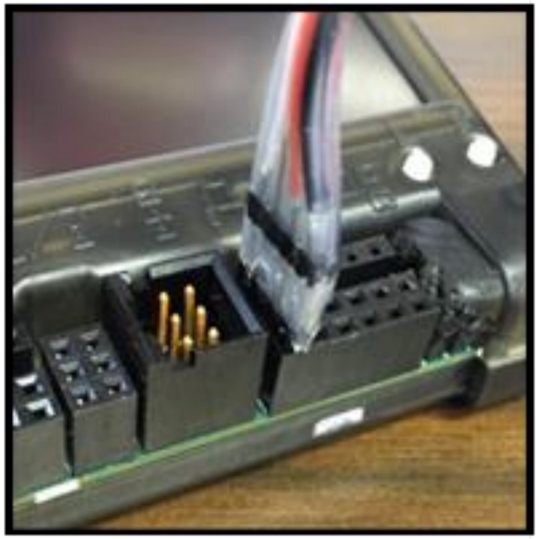


Digital Sensor Ports # 0 – 9

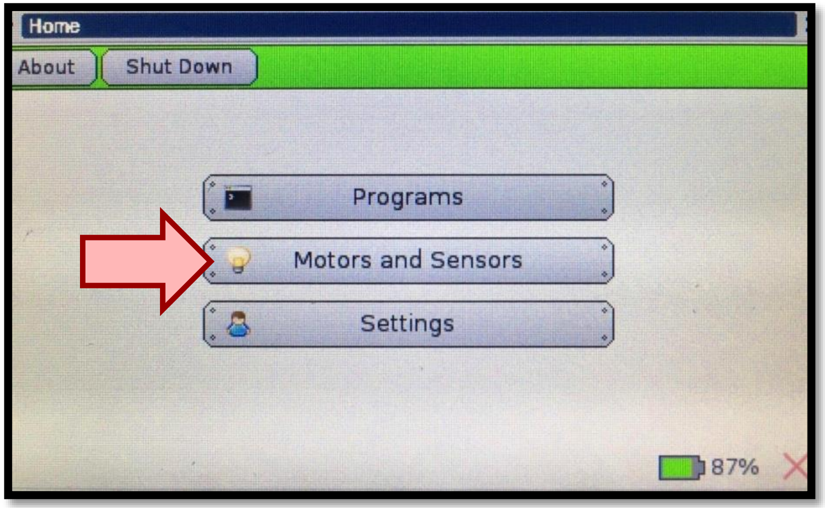
Analog Sensor Ports # 0-5

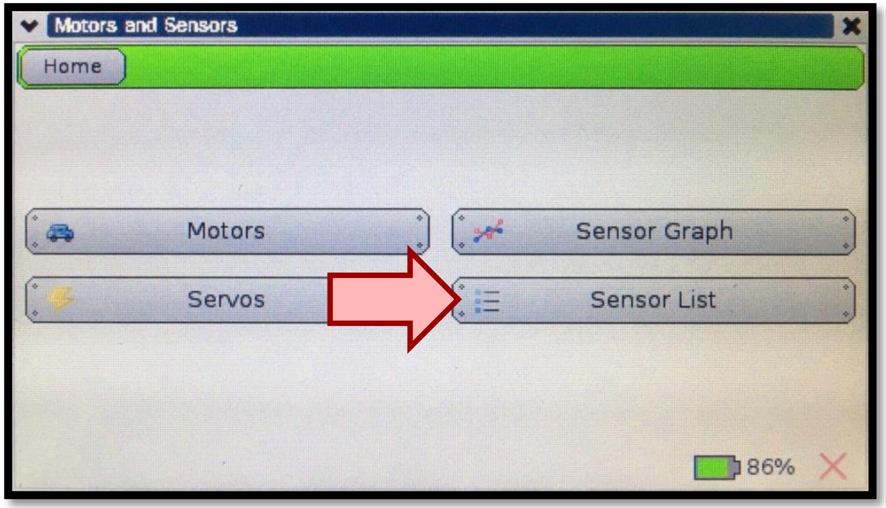
Plug the Light Sensor into Analog Port #0

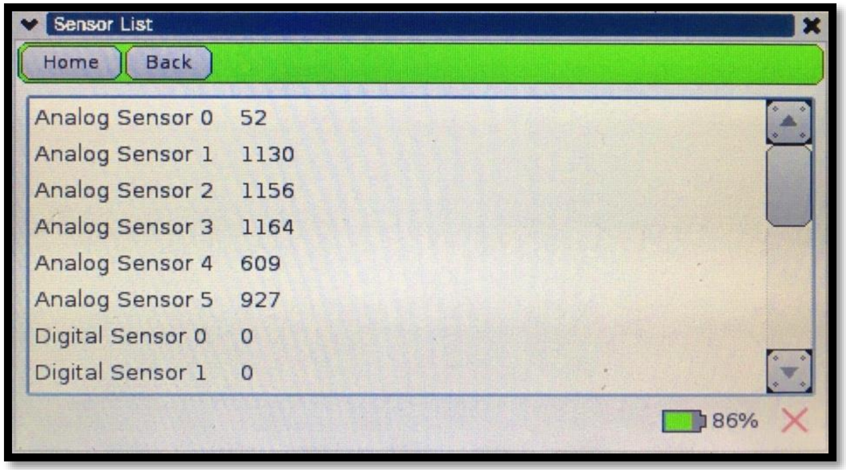




Use the Sensor List

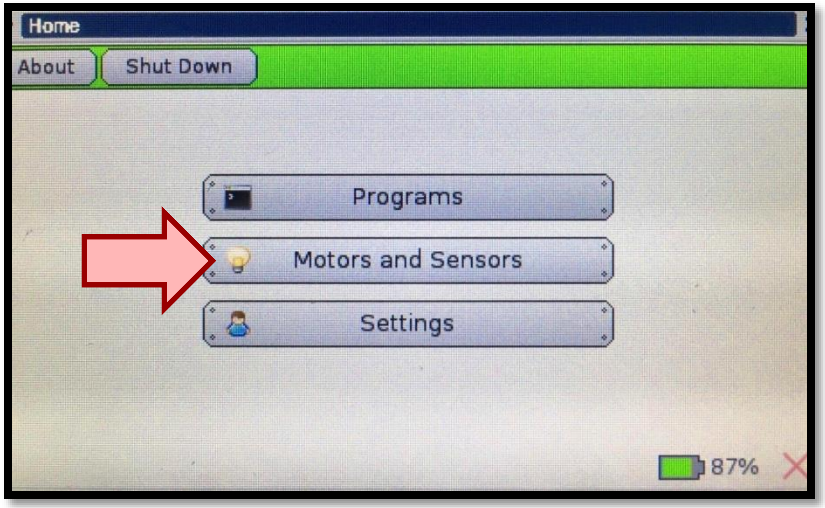


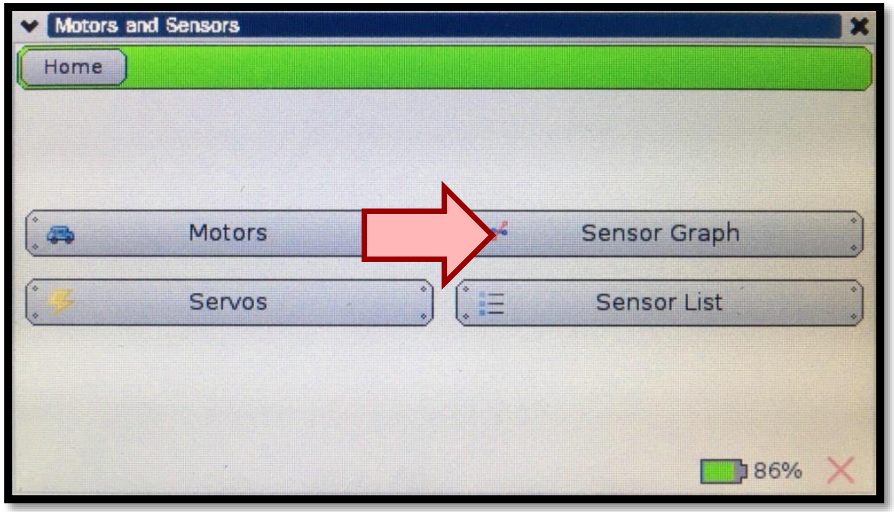


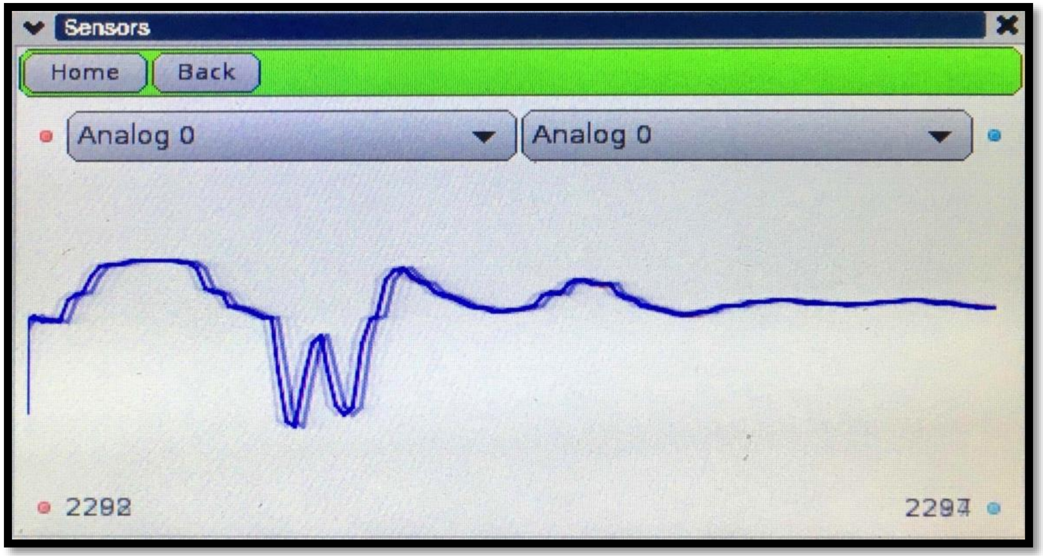


```
Analog Sensor 0   52
Analog Sensor 1   1130
Analog Sensor 2   1156
Analog Sensor 3   1164
Analog Sensor 4   609
Analog Sensor 5   927
Digital Sensor 0  0
Digital Sensor 1  0
```

Use the Sensor Graph







Starting with a Light

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

Analysis: What is the program supposed to do?

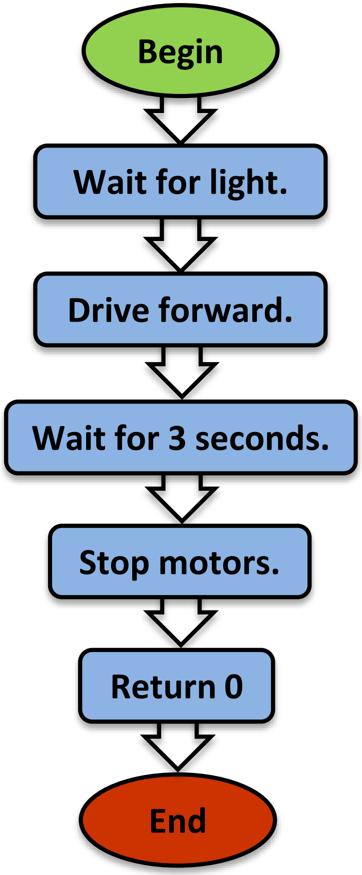
Pseudocode

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

Comments

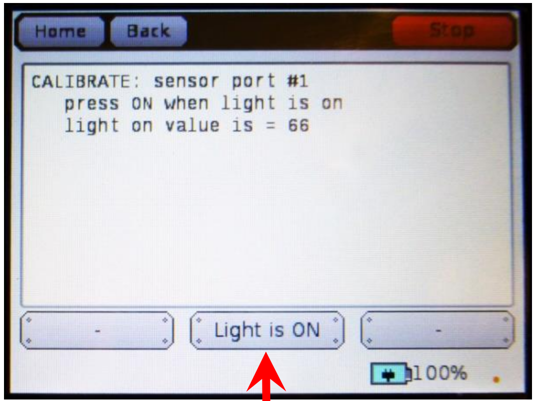
```
// 1. Wait for light.
// 2. Drive forward.
// 3. Wait for 3 seconds.
// 4. Stop motors.
// 5. End the program.
```

Flowchart

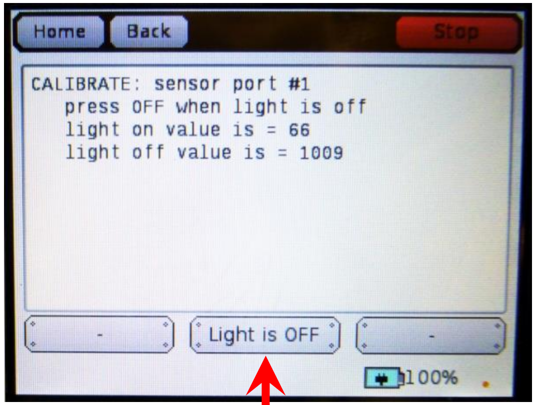


wait_for_light() Calibration Routine

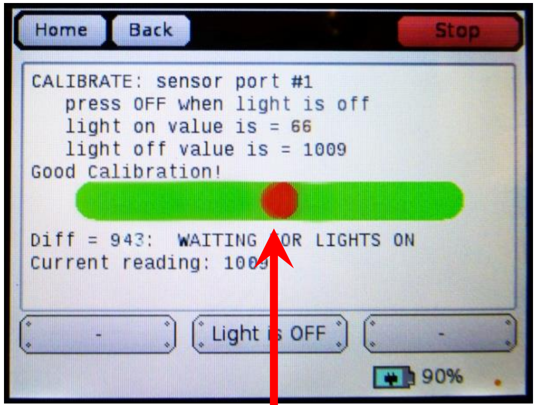
When using the **wait_for_light()** function in a program, the following calibration routine will run automatically.



```
CALIBRATE: sensor port #1
   press ON when light is on
   light on value is = 66
```



```
CALIBRATE: sensor port #1
   press OFF when light is off
   light on value is = 66
   light off value is = 1009
```



```
CALIBRATE: sensor port #1
   press OFF when light is off
   light on value is = 66
   light off value is = 1009
Good Calibration!

Diff = 943:  WAITING FOR LIGHTS ON
Current reading: 1009
```

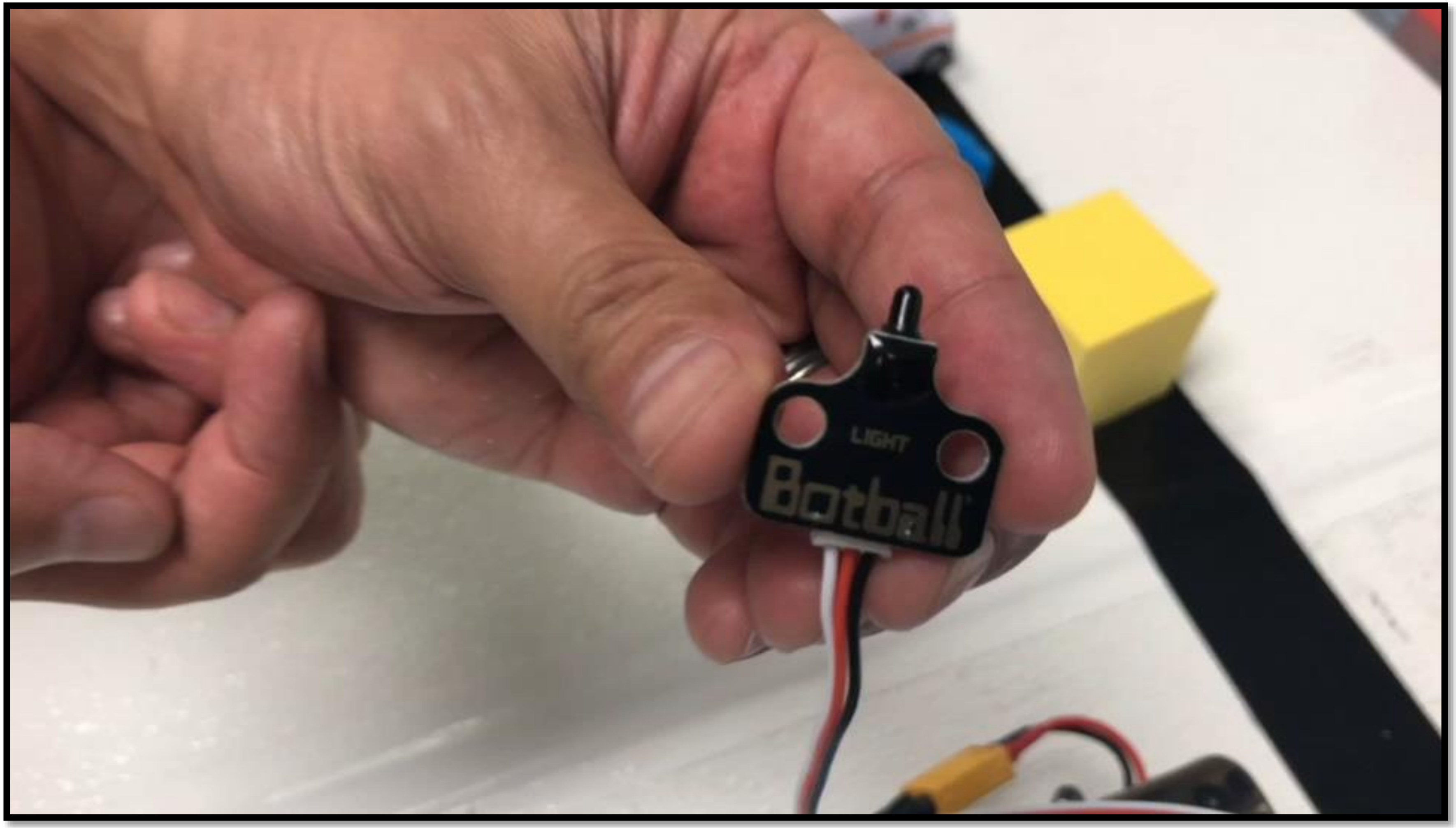
When the light is *on* (low value), press the "Light is On" button.

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

A "Good Calibration!" message is a moving red dot on green bar when done *correctly*.
A "BAD CALIBRATION" message will appear when not done correctly. The program will need to be run again.

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

`wait_for_light()` Calibration Routine



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

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

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

  return 0;
}
```

Execution: Compile and run the program on the KIPR Wallaby.

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 the program on the KIPR Wallaby.

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 the 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 the code.
- If this function is not in the code, the robot may not automatically turn off its motors at the end of the Botball round and **will be disqualified**!

Tournament Templates

```
int main()
{
  wait_for_light(0); // change the port number to match the port the robot uses
  shut_down_in(119); // shut off the motors and stop the robot after 119 seconds

  // The code

  return 0;
}
```

Running a Botball Tournament Program

Description: Write a program for the KIPR Wallaby 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

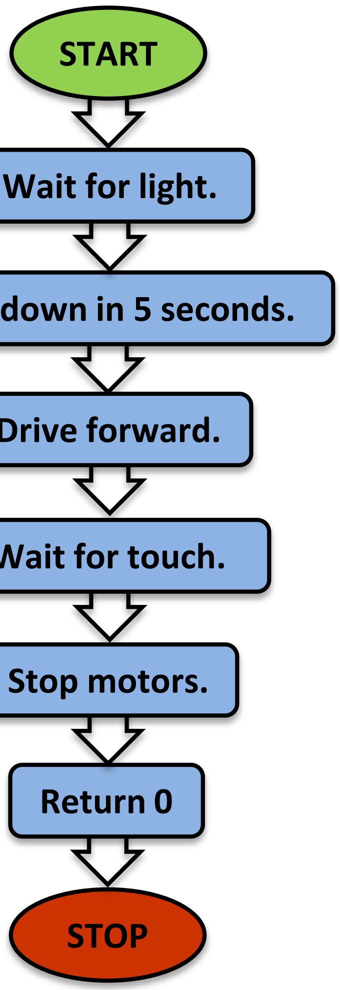
```
1. Wait for light.
2. Shut down in 5 seconds.
3. Drive forward.
4. Wait for touch.
5. Stop motors.
6. End the program.
```

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



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

```c
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 the program on the KIPR Wallaby.

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 the program will ***start with the light*** in a Botball tournament round? (**Answer: `wait_for_light(0)`**)
- What is the best way to guarantee that the program will ***stop within 120 seconds*** in a Botball tournament round? (**Answer: `shut_down_in(119)`**)

Use these functions in the Botball tournament code!

More Variables and Functions with Arguments

Data types

Creating and setting a variable

Variable arithmetic

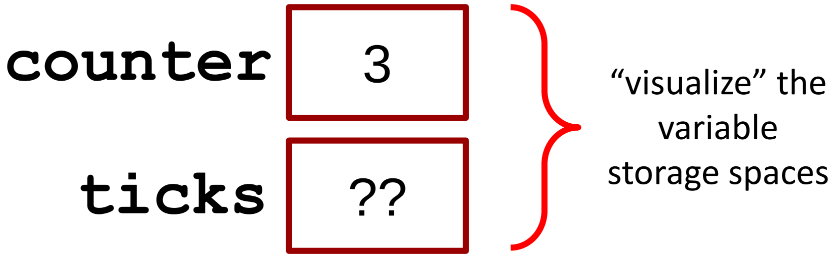
Functions with arguments and return values

Variables (Quick Recap)

Set the value of an int variable to any integer chosen and change it when needed 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 Read this as `ticks` is assigned `2000` times `1400.0` divided by circumference all in millimeters. This is used to calculate how many `ticks` needed to travel ~2 meters.

Move the Servo Arm Using a Loop

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

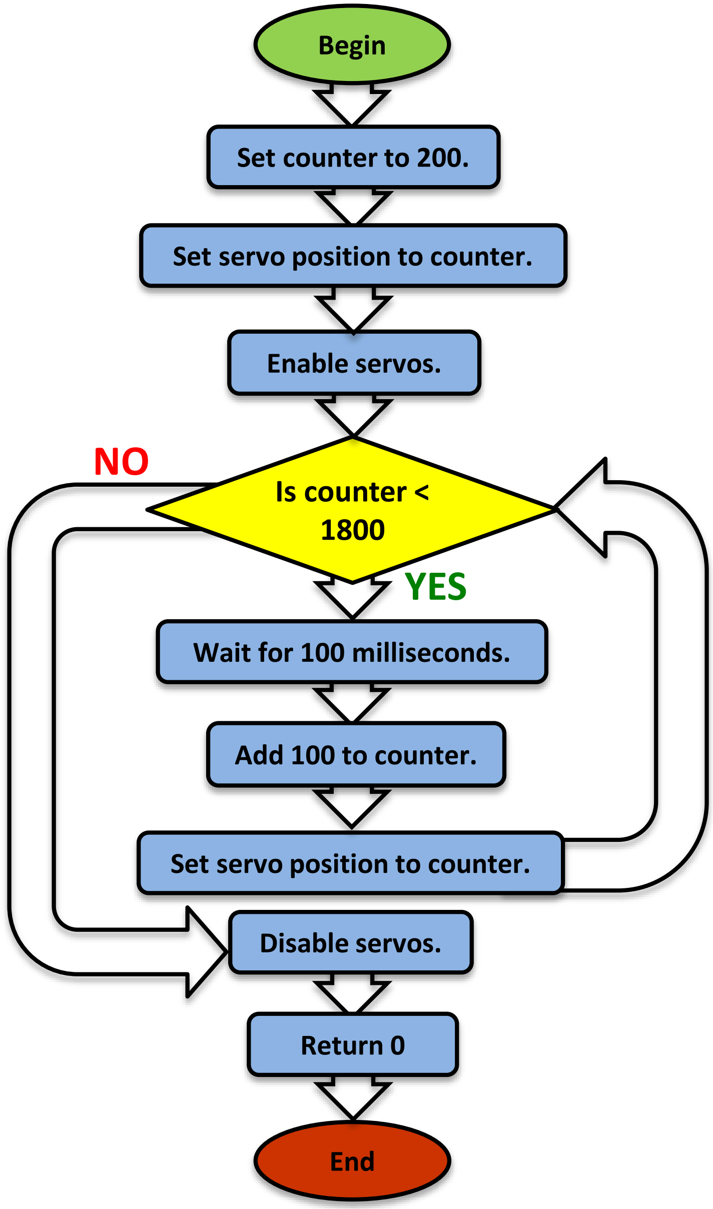
Analysis: What is the program supposed to do?

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

```c
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;
}
```

Custom Functions (Quick Recap)

When this function is called, 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();
}
```

Functions with Arguments

Function arguments: values that are set when the function is called

```
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();
}
```

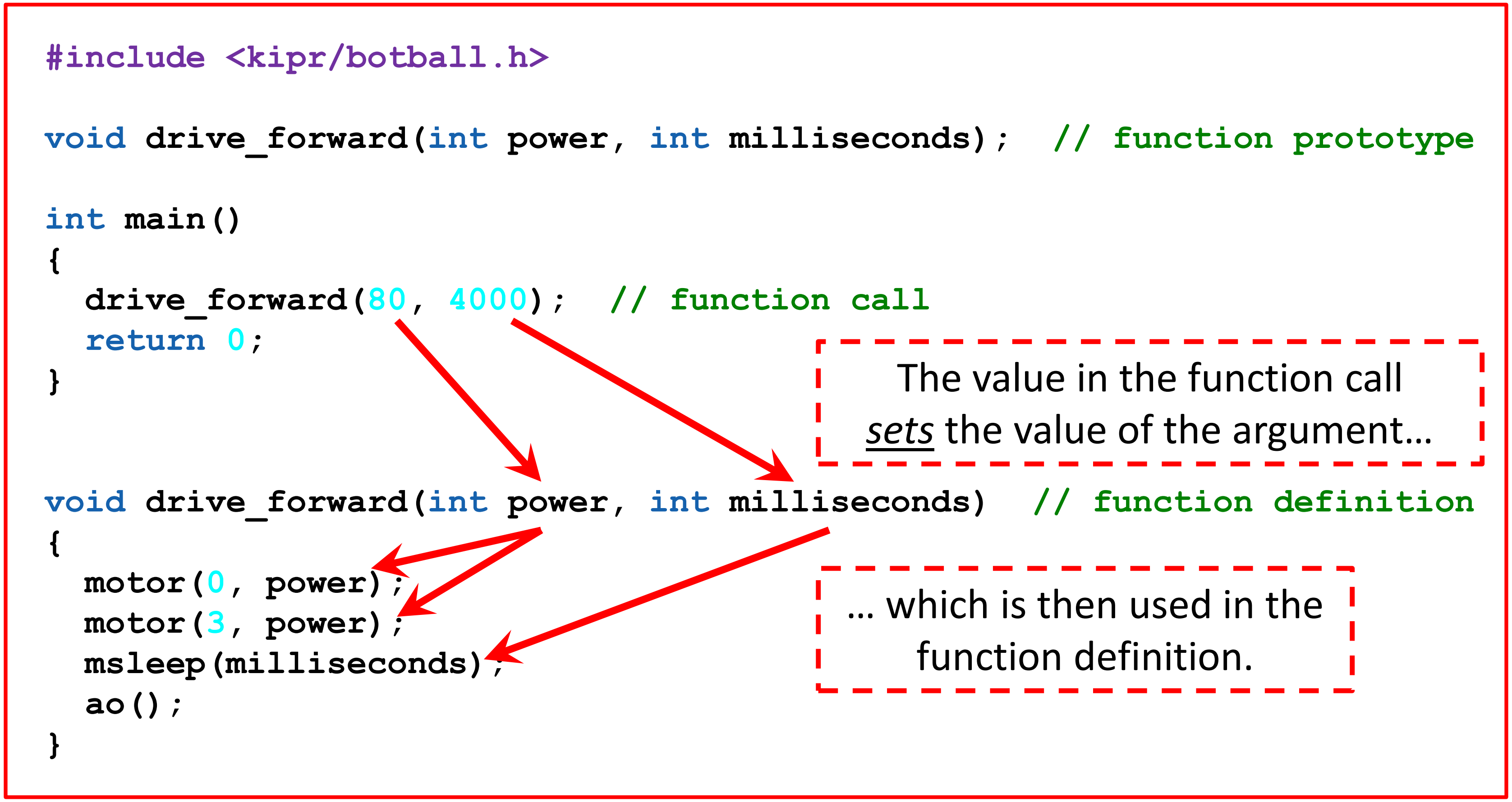
Writing Custom Functions with Arguments

```
#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 Functions with Multiple Arguments

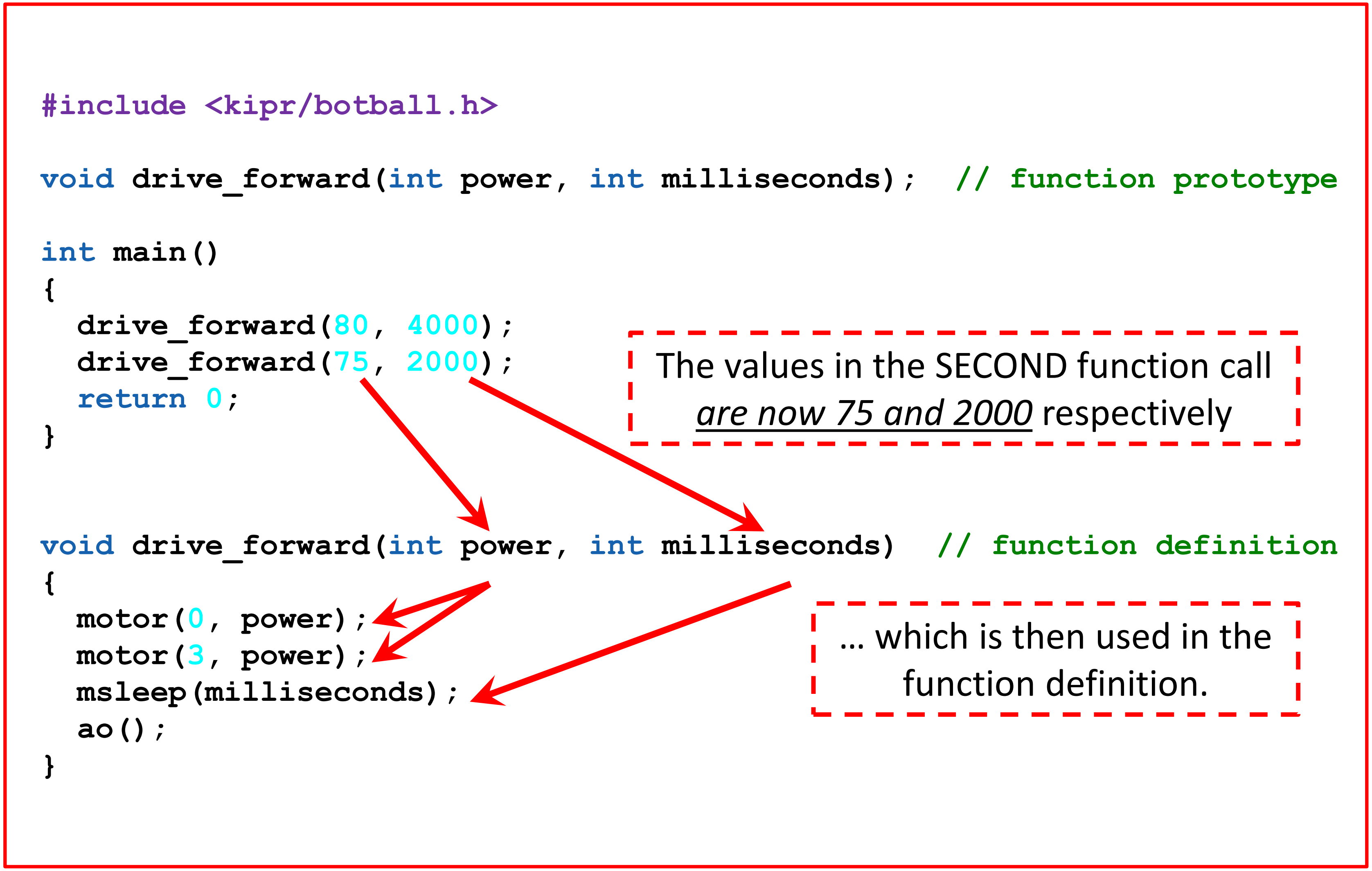


```
#include <kipr/botball.h>

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.

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.

Moving the iRobot *Create*: Part 1

Setting up the *Create*

The *Create* and the KIPR Wallaby

Create functions

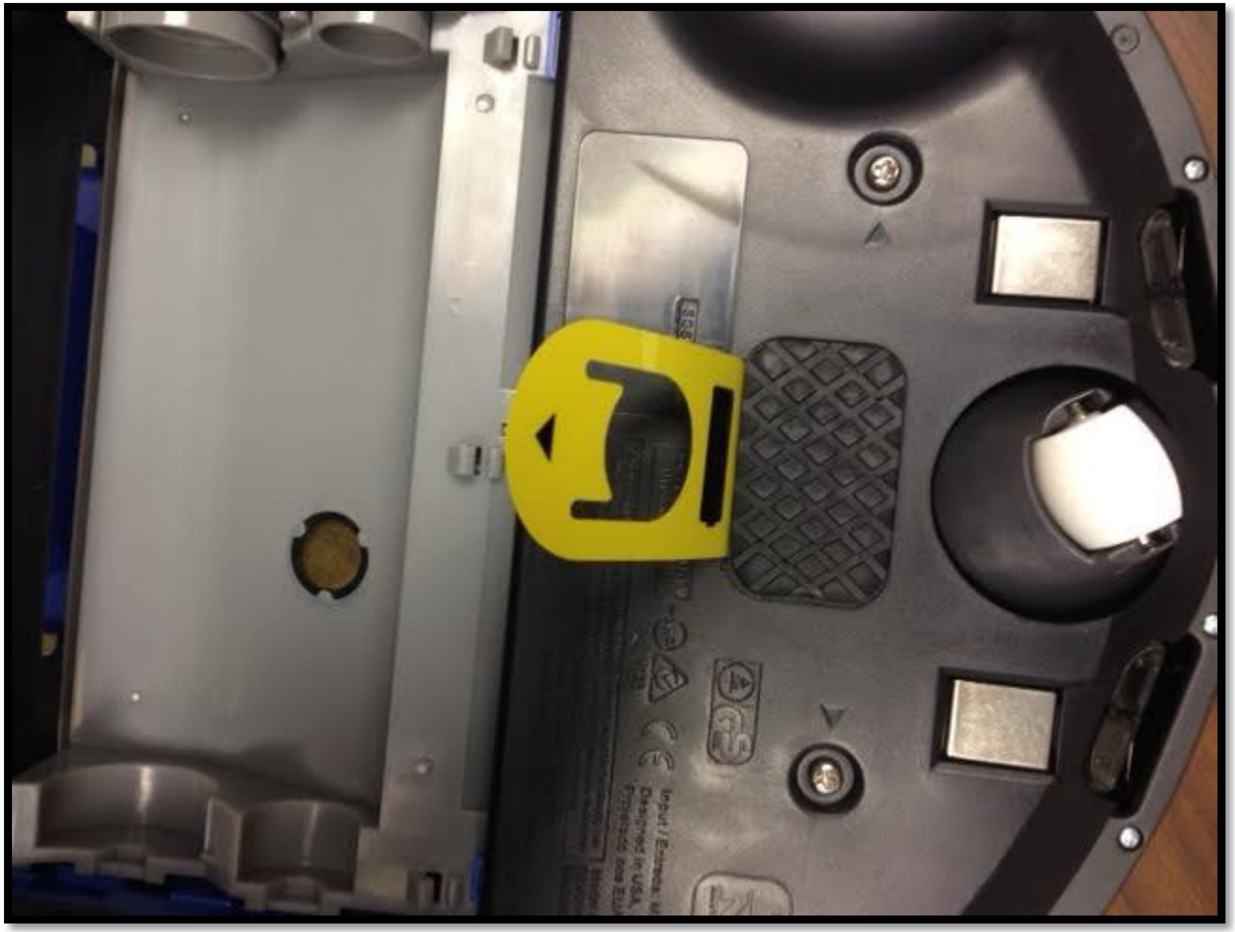
Charging the *Create*

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

- The **Create** 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 NOT leave the unit unattended** while charging.
  - Charge in a cool, open area away from flammable materials.

Enabling the Battery of the *Create*

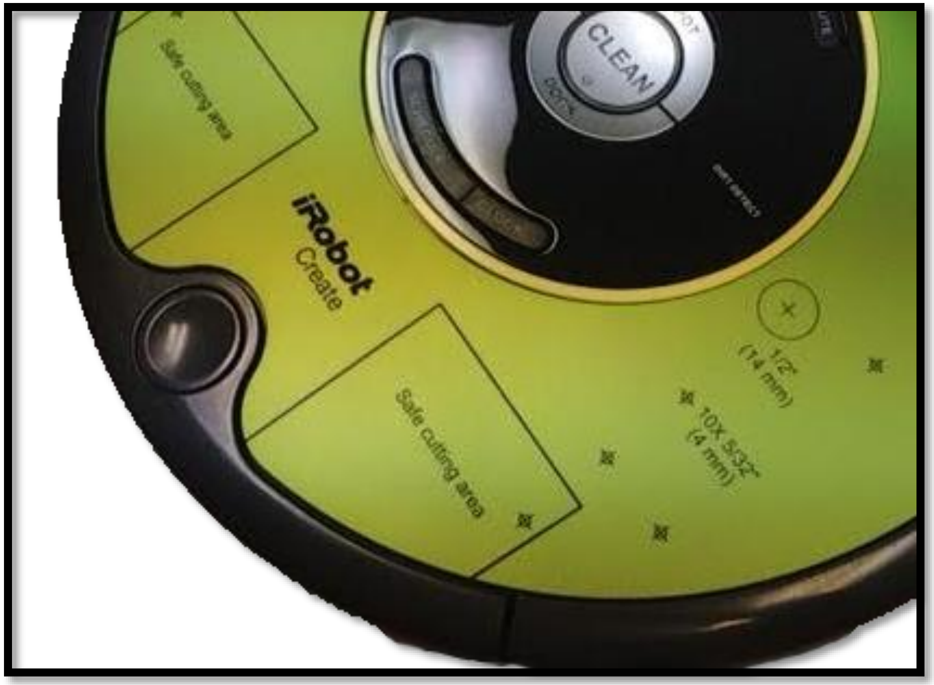
- The **yellow battery** tab pulls out of place on the bottom of the *Create*.
- The battery will be enabled as soon as the tab is removed.

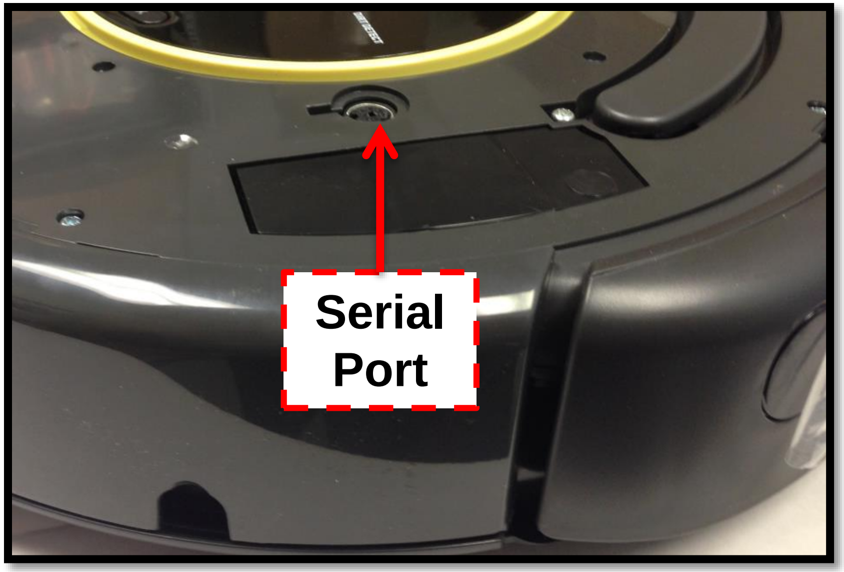


Create Underside

Uncovering and Charging the *Create*

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





Mounting the Robotics Controller onto the *Create*

Build the Create DemoBot

The build slides should be saved on your desktop.

Create Connect/Disconnect Functions

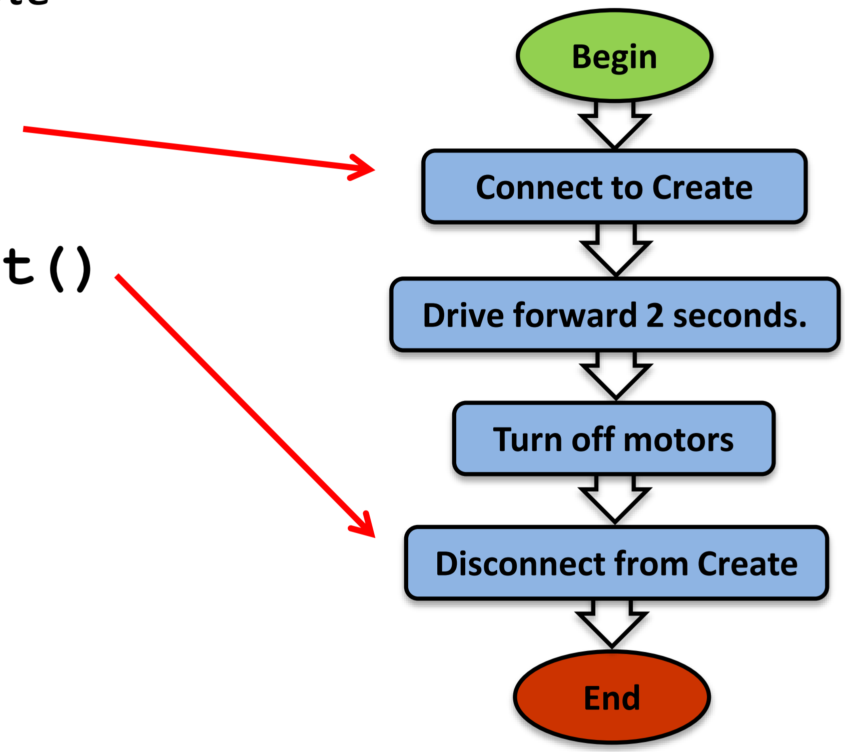
All programs used with the *Create* **MUST** *start* with

```
create_connect()
```

and *end* with

```
create_disconnect()
```

Flowchart



Tournament Templates

```
int main() // for the Create robot
{
  create_connect();
  wait_for_light(0); // change the port number to match the port used
  shut_down_in(119); // shut off the motors and stop the robot after 119 seconds

  // The code

  create_disconnect();
  return 0;
}
```

Create 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*

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

Analysis: What is the program supposed to do?

Pseudocode

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

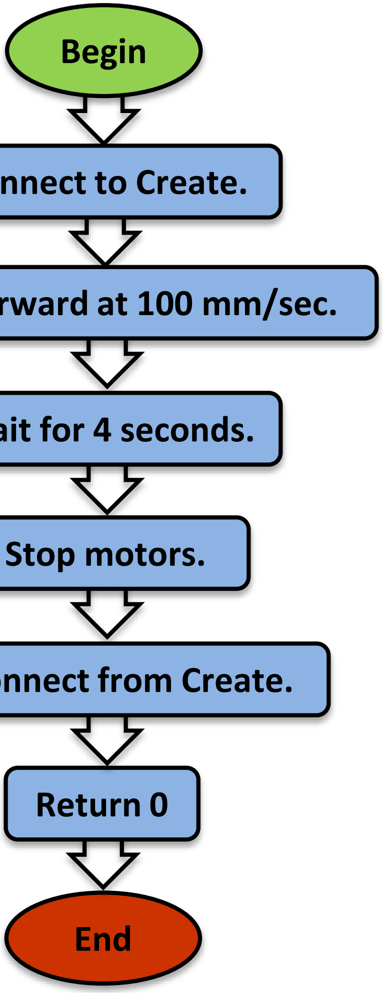
Comments

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

Moving the *Create*

Analysis:

Flowchart



Moving the *Create*

Solution:

Pseudocode

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

Source Code

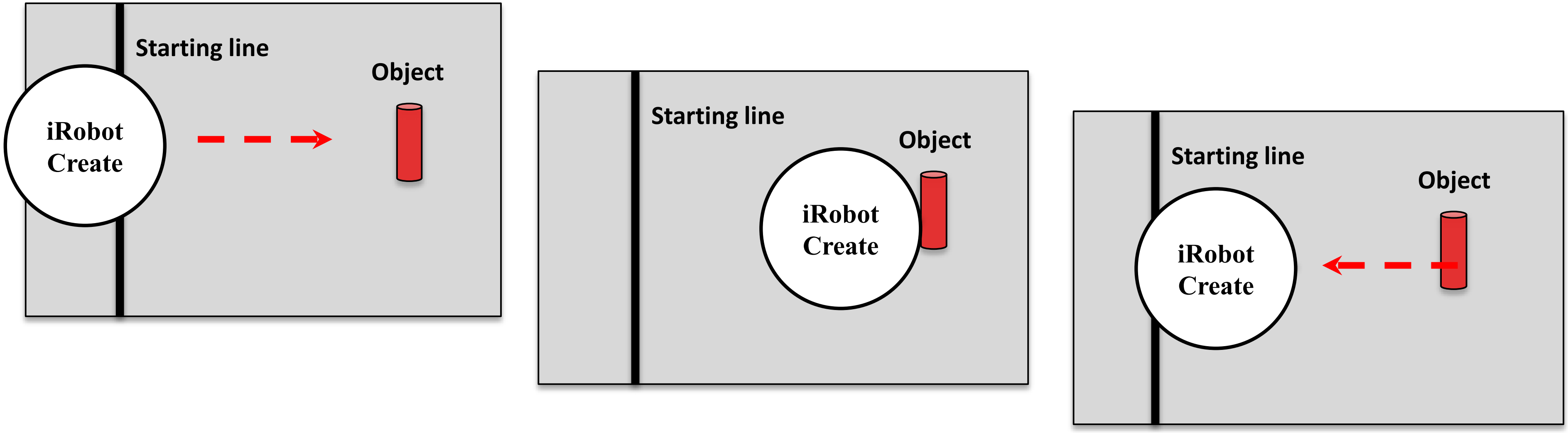
```
int main()
{

  create_connect();

  create_drive_direct(100, 100);

  msleep(4000);

  create_stop();

  create_disconnect();

  return 0;
}
```

Execution: Compile and run the program on the KIPR Wallaby.

Touch an Object and “Go Home”

Description: Write a program for the KIPR Wallaby 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.



Create Safety Feature

Create has some built in safety features that disable the motors if the robot is picked up, or if the front part goes over an edge. The **`create_full()`** command will disable all built in safety features. Use it with caution.

Why would this be done?

- During calibration the program is already started and the robot is lifted off the ground. Lights turn on and the Create doesn't move.
- The Create is driving on the game board and the front edge of it gets on top of the PVC tube, and then the robot stops.

```
create_full(); //disable safety features

create_safe(); //enable safety features
```

Add this after **`create_connect()`**

Moving the iRobot *Create*: Part 2

Create Distance and Angle Functions

Create 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* Functions

Examples

```c
int main()
{
  create_connect();

  set_create_distance(0);

  while (get_create_distance() < 1000)
  {
    create_drive_direct(200, 200);
  }

  create_stop();
  create_disconnect();

  return 0;
}
```

```c
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;
}
```

Printing Create Sensor Values

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

To print a changing integer value, use a `%d` *placeholder* in the print statement.

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 provide the value to print. It can be a function call (as here) or a variable name.

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;
}
```

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

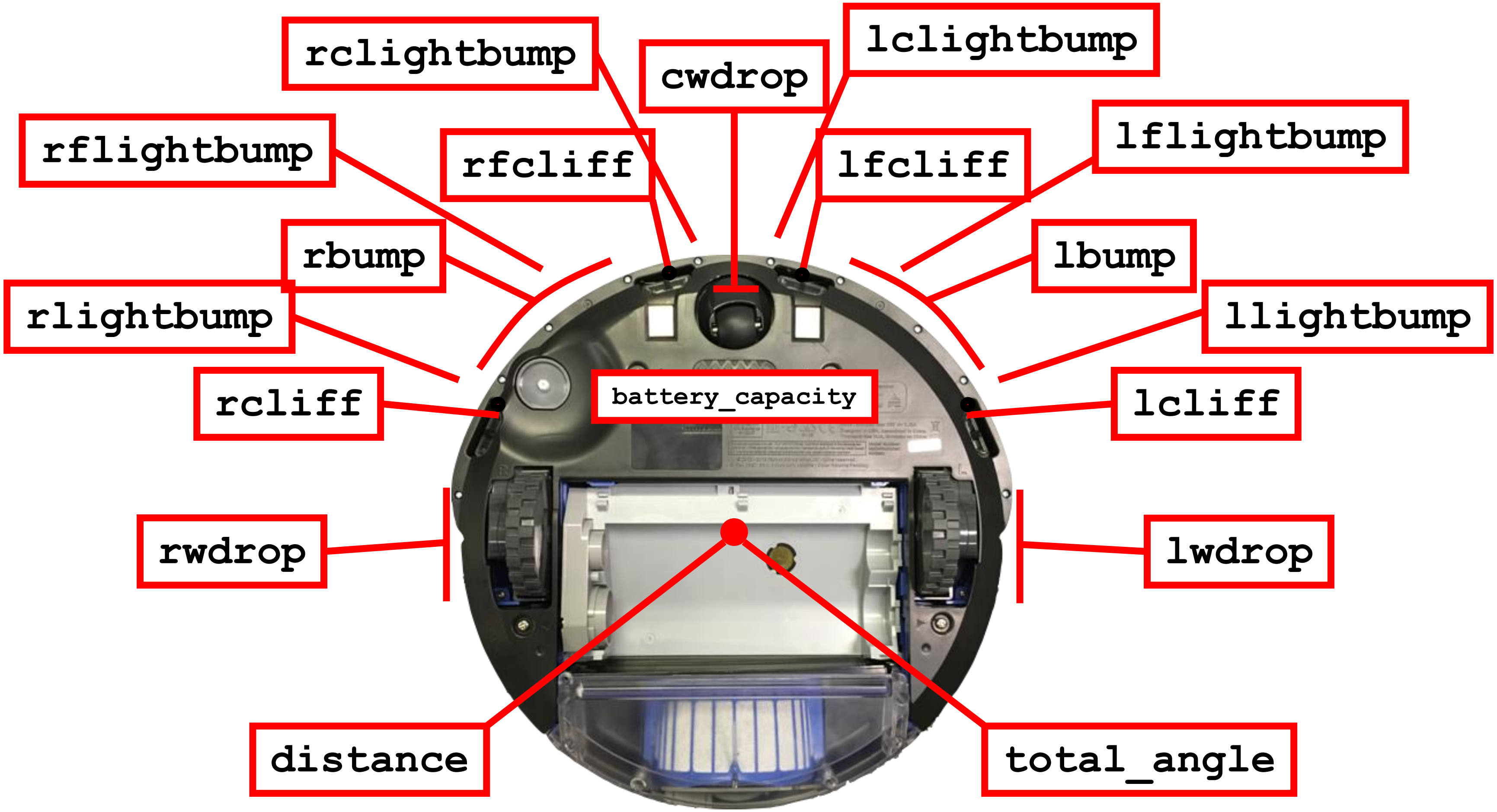
iRobot *Create* Sensors

Create Sensor Functions

Logical Operators

Create Sensor Functions

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



Create 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()
get_create_lfcliff()
get_create_rcliff()
get_create_rfcliff()
// 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* 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;
}
```

Drive Until Bumped

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

Analysis: What is the program supposed to do?

Pseudocode

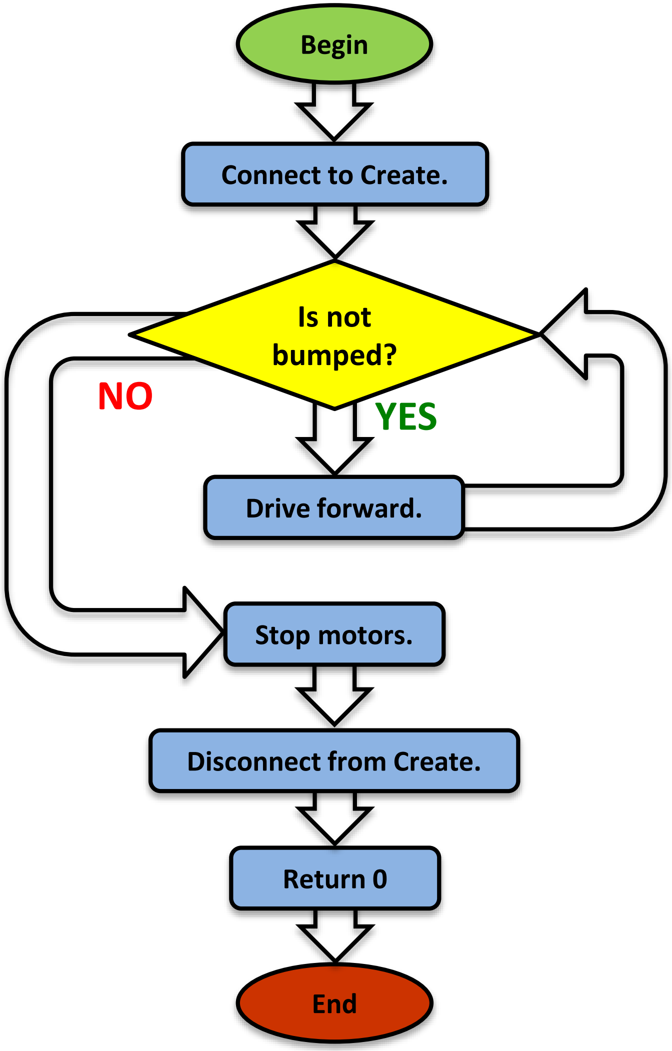
```
1.  Connect to Create.
2.  Loop: Is not bumped?
    1.  Drive forward.
3.  Stop motors.
4.  Disconnect from Create.
5.  End the program.
```

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. Loop: Is not bumped?
       Drive forward.
3. Stop motors.
4. Disconnect from Create.
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 the program ends, print to the screen the values for the total angle the create has turned and total distance it has driven.

LUNCH

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/WQ8CQ65

Color Camera

Using the Color Camera

Setting the Color Tracking Channels

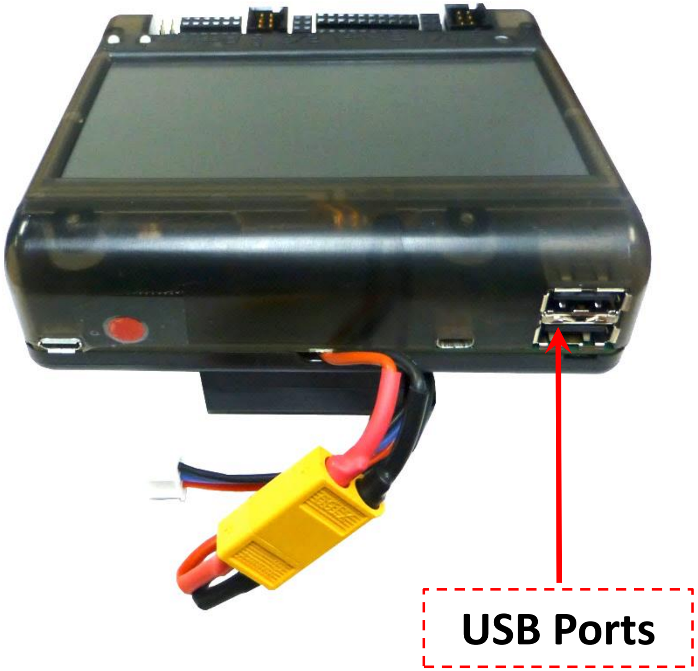
About Color Tracking

Camera Functions

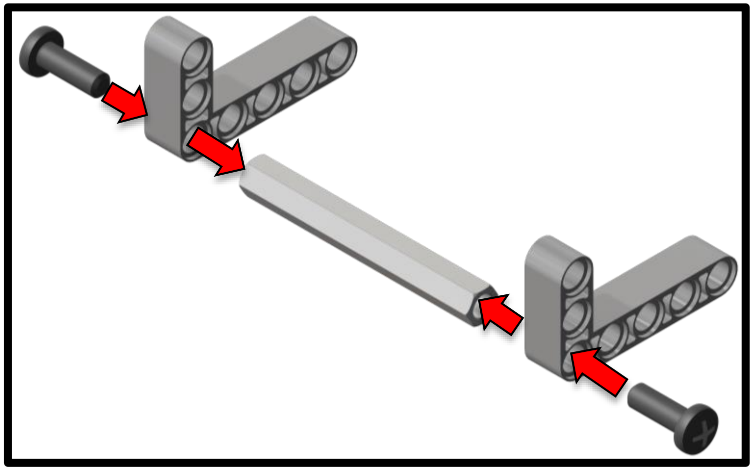
Color Camera

Use the **camera** for this activity

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



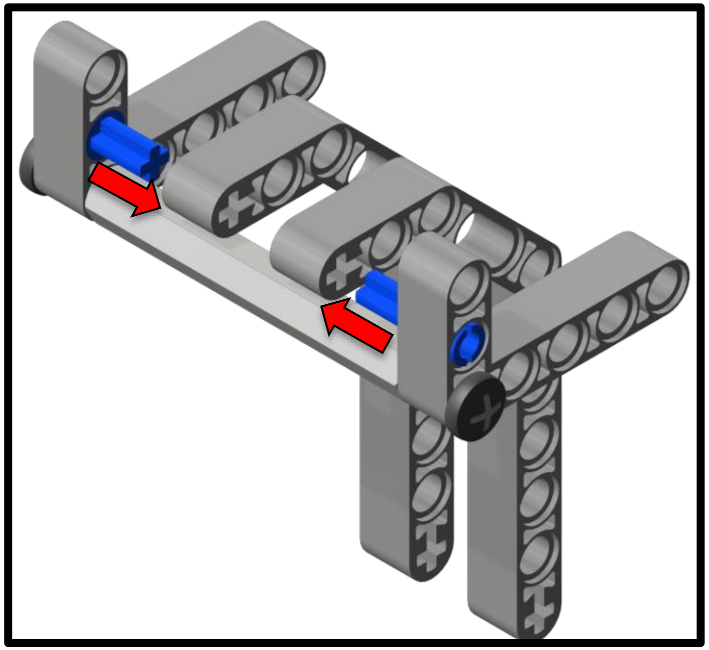
Camera Build

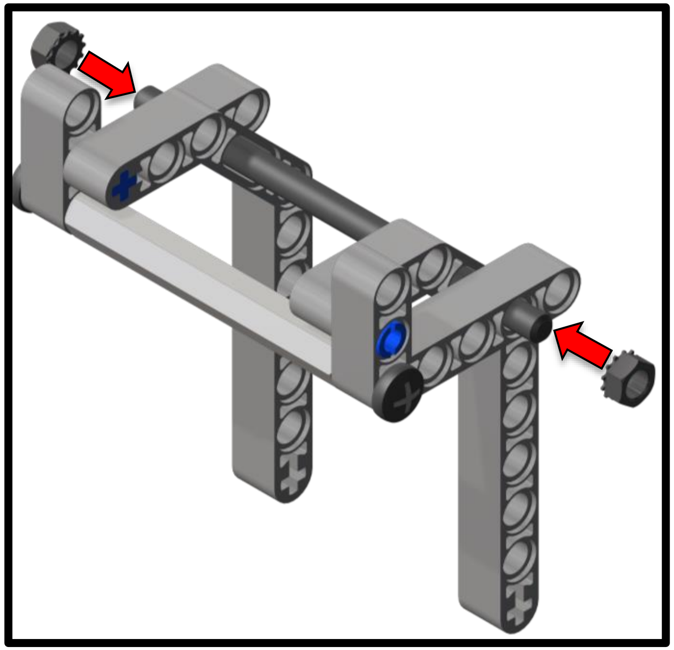


Start with a 2" Standoff attach a 3x5 liftarm to each end at the hole in the bend with regular screws.

Attach the large bent liftarms using the blue axel-pins as shown.

Run a 3" headless screw through the holes as shown and secure it with two nuts.

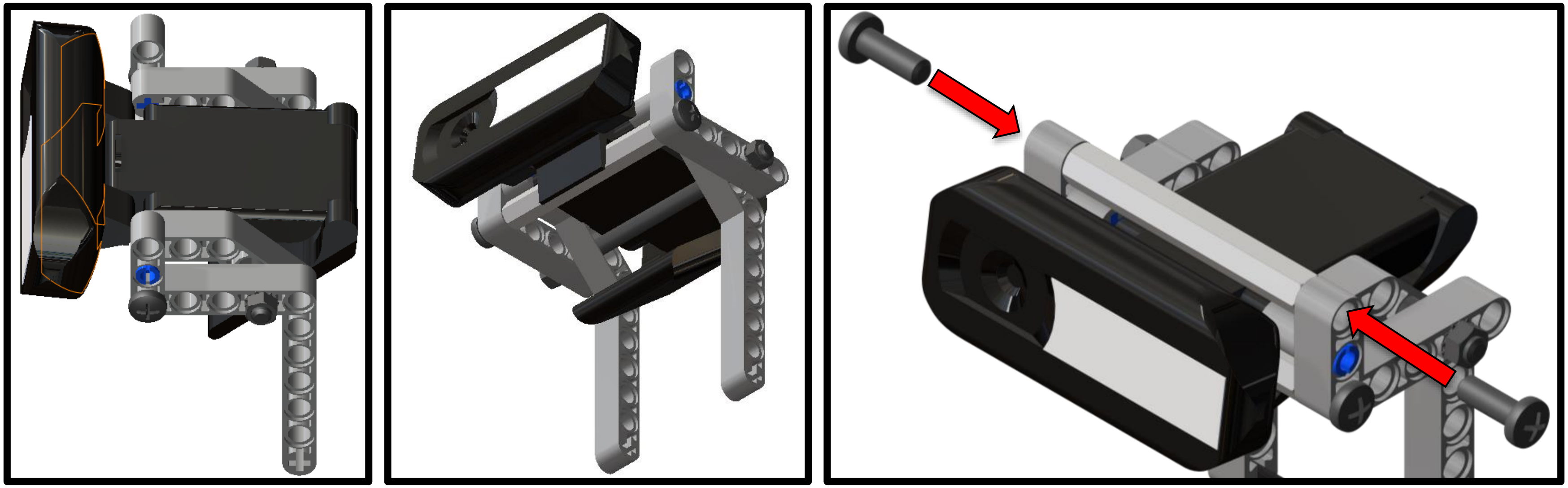




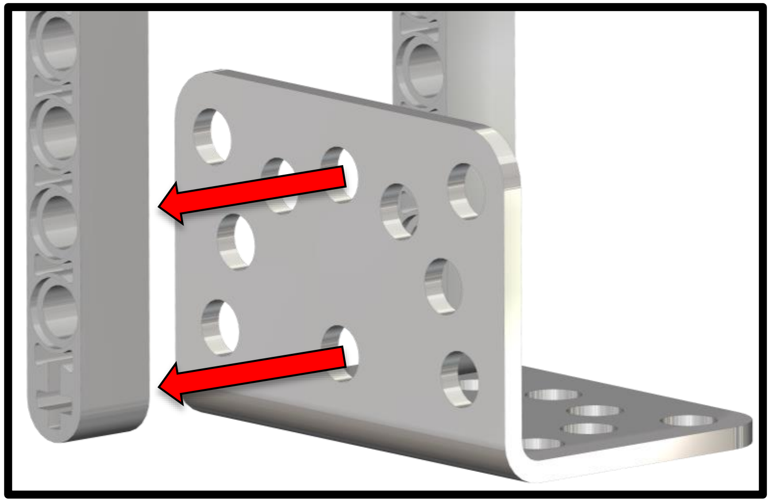
Camera Build

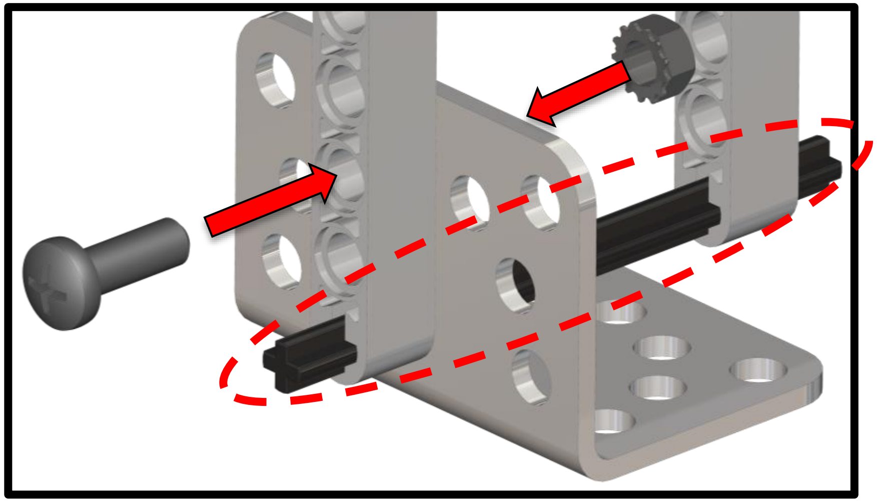
Wrap the camera around the standoff and headless screw as shown.

Then secure it with another standoff on top of it screwed in with two regular screws as shown.



Mounting the Camera



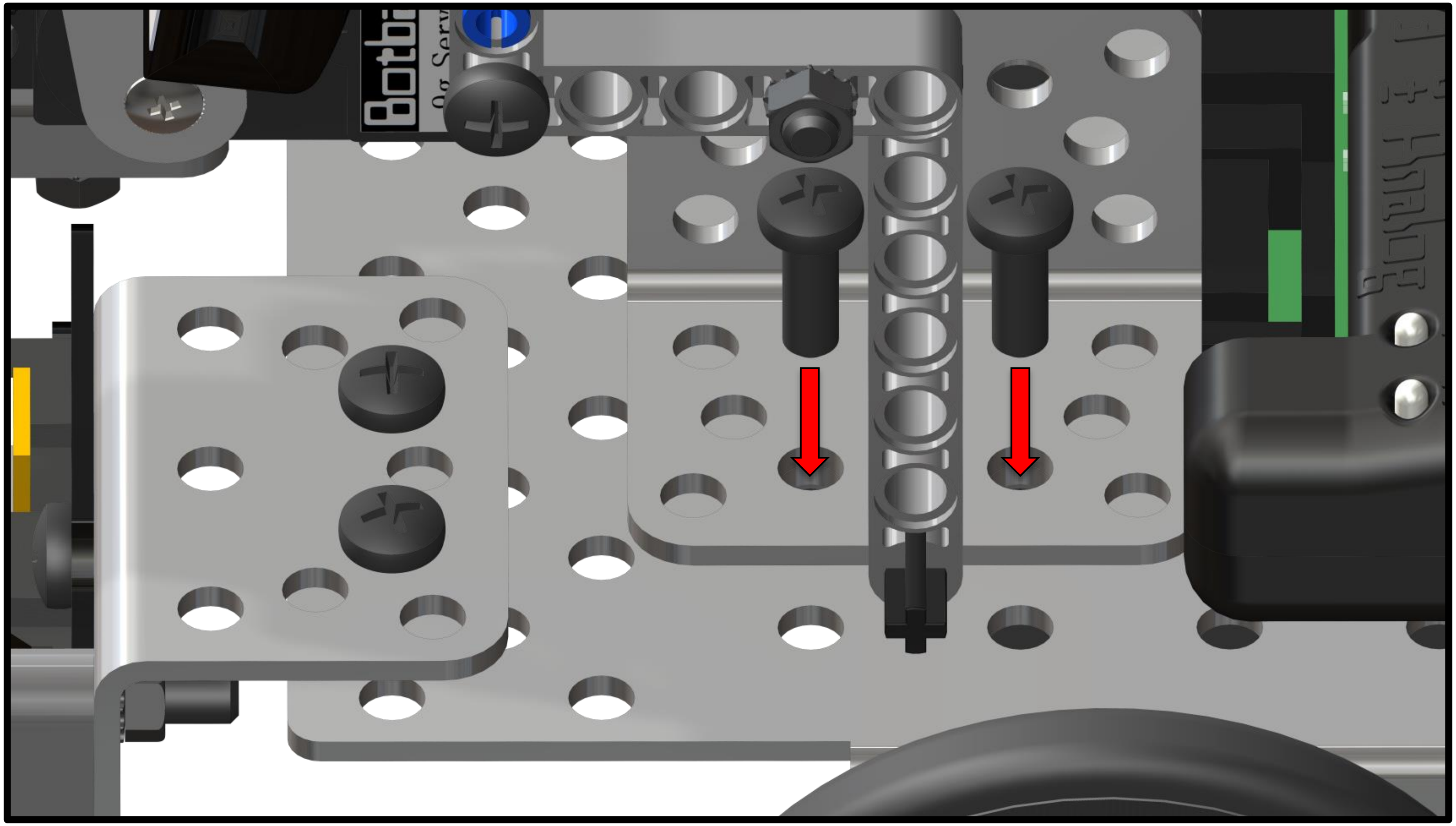


Line up the L-bracket holes with the bottom holes of the mount on the inside as shown above.

Then attach it by running an axle through both pieces of lego and the bottom middle hole of the bracket and securing the top of the bracket with a medium screw.

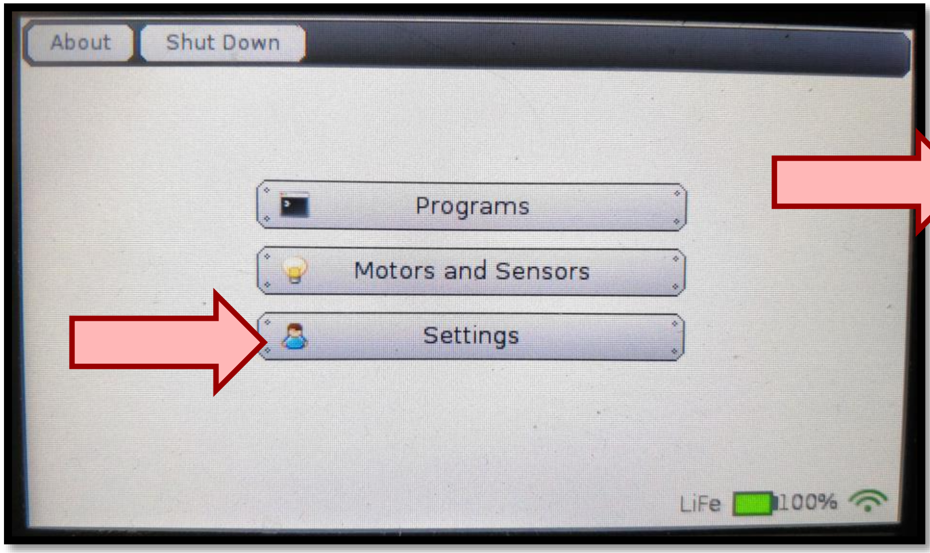
Mounting the Camea

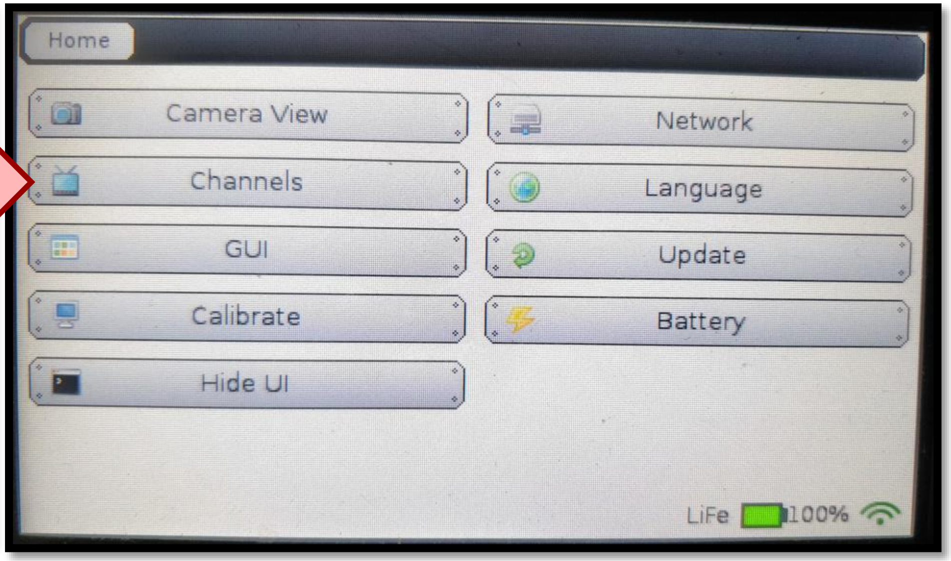
Secure the camera mount to the rest of the robot as shown using two regular screws.



Setting the Color Tracking Channels

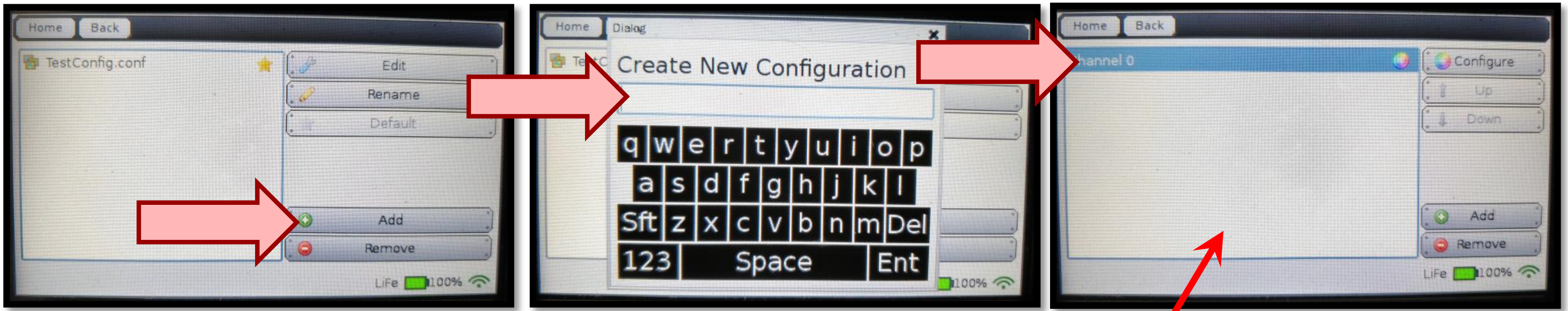
1. Select ***Settings***
2. Select ***Channels***





Setting the Color Tracking Channels

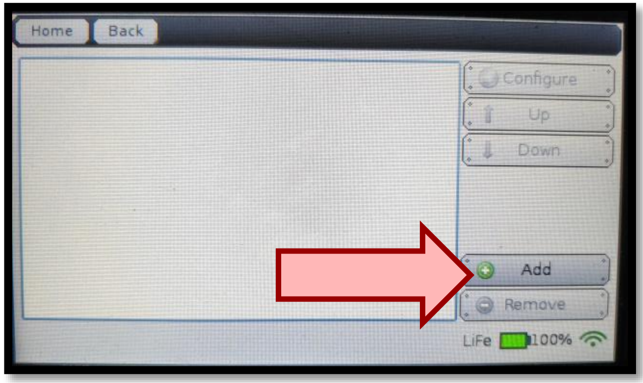
3. To specify a **camera configuration**, press the *Add* button.
4. Enter a configuration name, such as **find_green**, then press the *Ent* button.
5. Highlight the new configuration and press the *Edit* button.

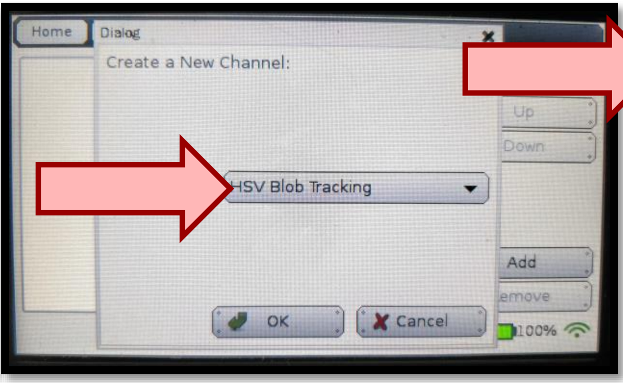


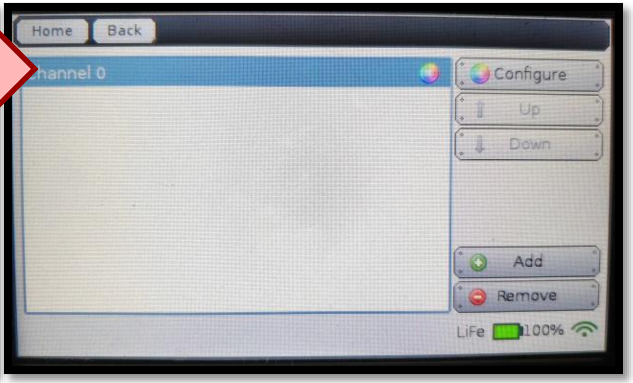
Note: if there is more than one configuration, select one, and press the "Default" button to make it be the one in use!

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 make this track color.
8. Highlight the channel, then press *Configure* to edit settings.
    - The first channel is 0 by default. There can be up to four: **0**, **1**, **2**, and **3**.



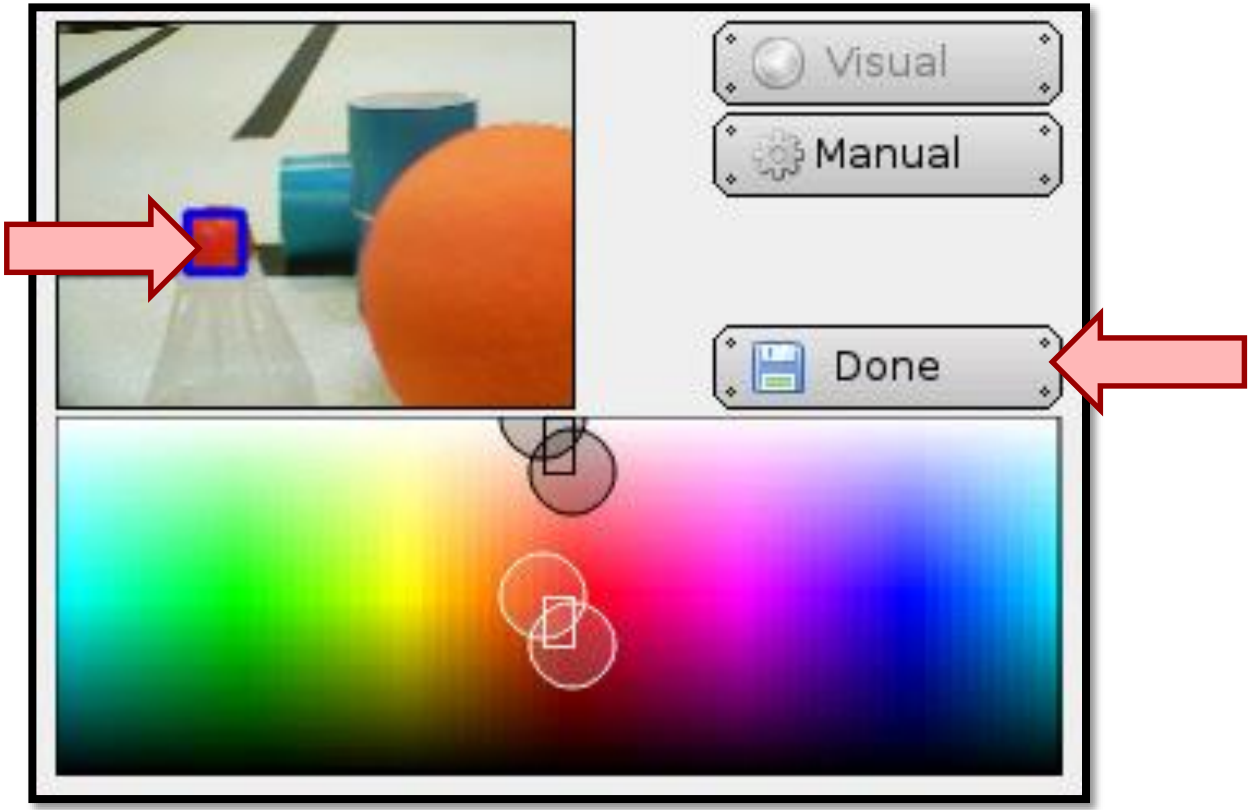




Setting the Color Tracking Channels

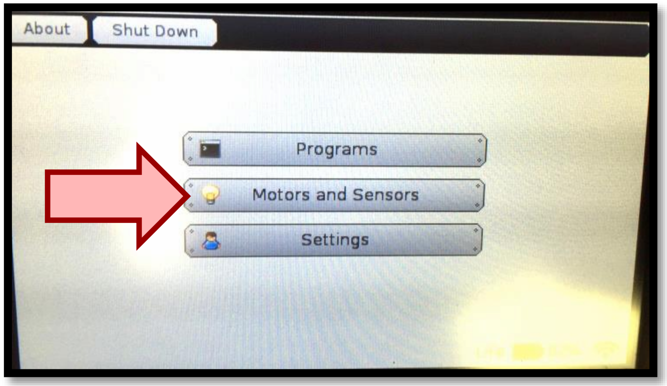
9. Place the colored object to track in front of the camera and **touch the object on the screen**.
   - A **bounding box** (dark blue) will appear around the selected object.

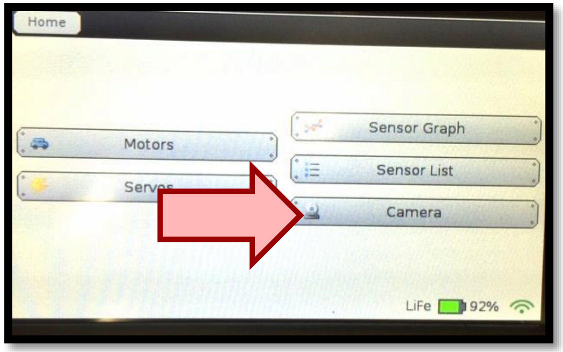
10. Press the *Done* button.

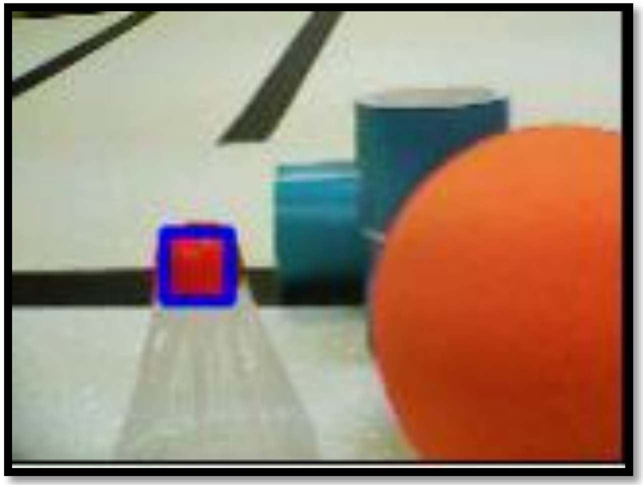


Verify the Color Channel is Working

11. From the **Home** screen, press *Motors and Sensors* button.
12. Press the *Camera* button.
13. Objects specified by the configuration should have a **bounding box (shown in blue)**.

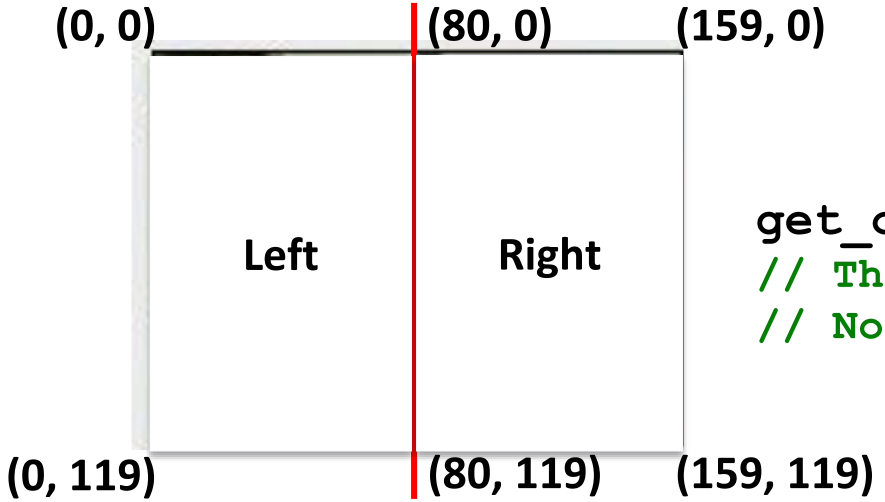






Tracking the Location of an Object

- 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***.
- 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_black();
// Opens the connection to the black 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.
```

Using Camera Functions

```
int main()
{
  int iteration_count = 0;
  int update_errors = 0;
  camera_open_black();

  while (digital(8) == 0)
  {
    if(!camera_update())
    {
      update_errors++;
      continue;
    }
    if (iteration_count > 1000)
    {
      iteration_count = 0;
      camera_close();
      camera_open_black();
    }

    //Code to be executed
  }
  camera_close();
  return 0;
}
```

Connections to the Game Board

Description: Calibrate and program the robot and camera combination so that it will turn on its axis in response to Botguy moving to the left or right in front of it.

Logical Operators

Multiple Boolean Tests

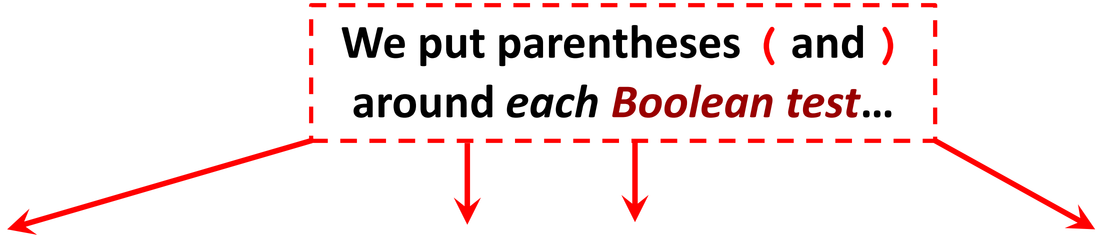
`while`, `if`, and Logical Operators

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**.

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.

while, if, and Logical Operators Examples

```
while ((get_create_lbump() == 0) && (get_create_rbump() == 0))
{
   // Run code if both bumpers are not pressed...
}
```

```
while ((digital(8) == 0) && (digital(9) == 0))
{
   // Run code if both touch sensors are not pressed...
}
```

```
if ((digital(4) == 1) || (digital(5) != 0))
{
   // Run code if one or both of the touch sensors is pressed...
}
```

```
if ((analog(3) < 512) || (digital(3) == 1))
{
   // Run code if IR sensor reads white and/or touch sensor pressed...
}
```

Using Logical Operators

What does this conditional 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;
}
```

Connections to the Board Game

Description: Write a program for the KIPR Wallaby that drives the *Create* forward for 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))`

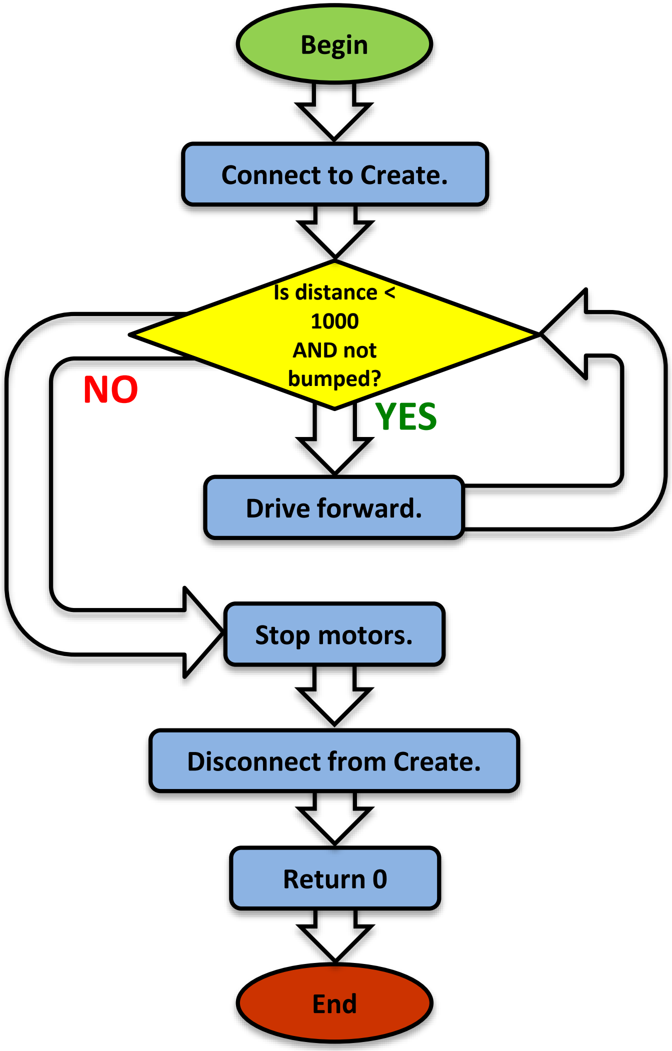
Analysis: What is the program supposed to do?

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.
```

Drive for Distance or Until Bumped

Analysis: Flowchart



Drive for Distance or Until bumped

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

```c
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 can be noticed after the program is run?

- 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 the program check to see if the *Create* ***left*** *bumper* is pressed?
  Answer:

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

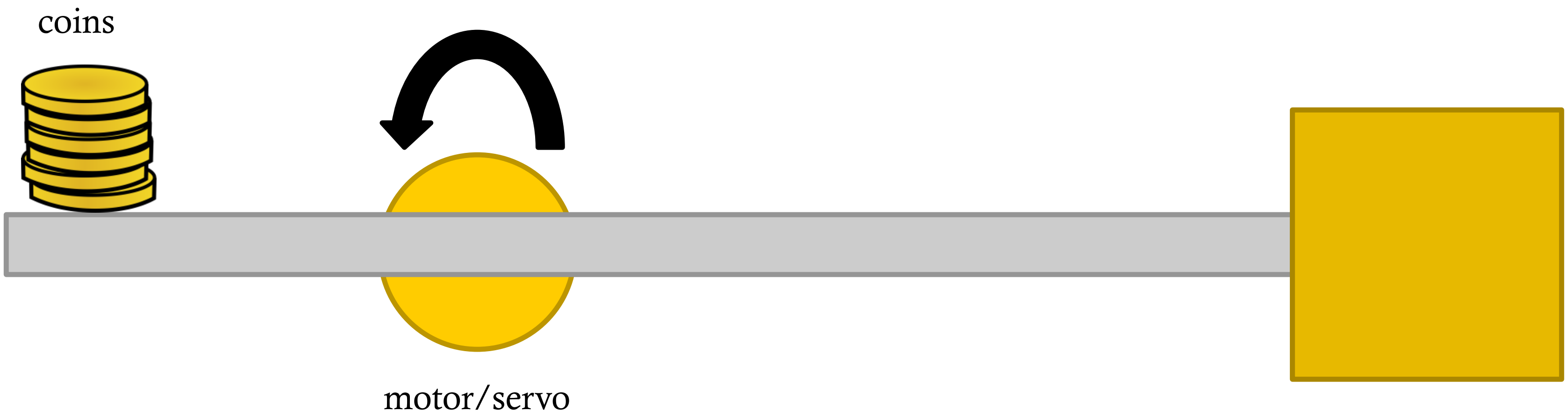
Mechanical Design

- Sometimes problems can be solved not through modifying the code, but rather by making changes to the mechanical design of the 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

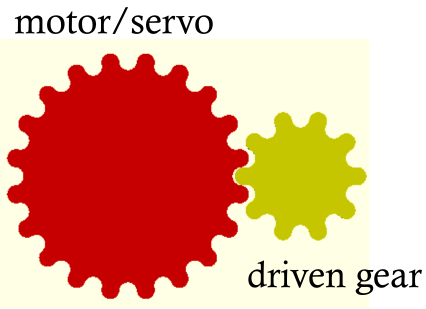
- Motors and servos have limited power
- Struggling to lift a structure?
  - Use coins as a counterbalance



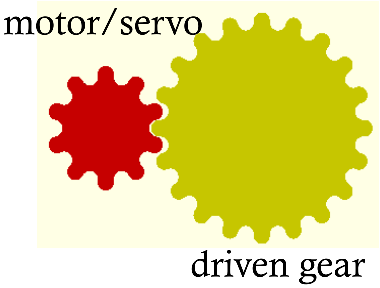
Gearing and Gear Trains

By “combining” gears into a “gear train”, using gears of varying sizes can INCREASE or DECREASE the speed and power (torque) of the motors!

- If the motor gear is **larger** than the next gear in the “gear train” the “driven gear” spins FASTER but at the expense of LESS torque (power).

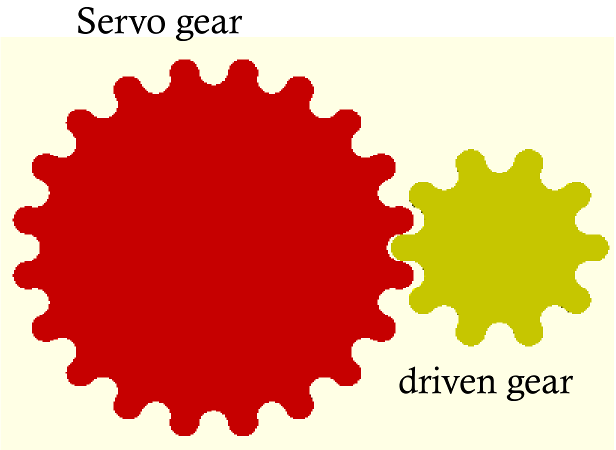


- If the motor gear is **smaller** than the next gear in the “gear train” the “driven gear” spins SLOWER but with MORE torque (power).



Gears to Increase Servo Range

- If a larger gear is attached to the 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, then it doubles (x2) the range of the servo (now 360 degrees instead of 180 degrees).



Resources and Support

Team Home Base

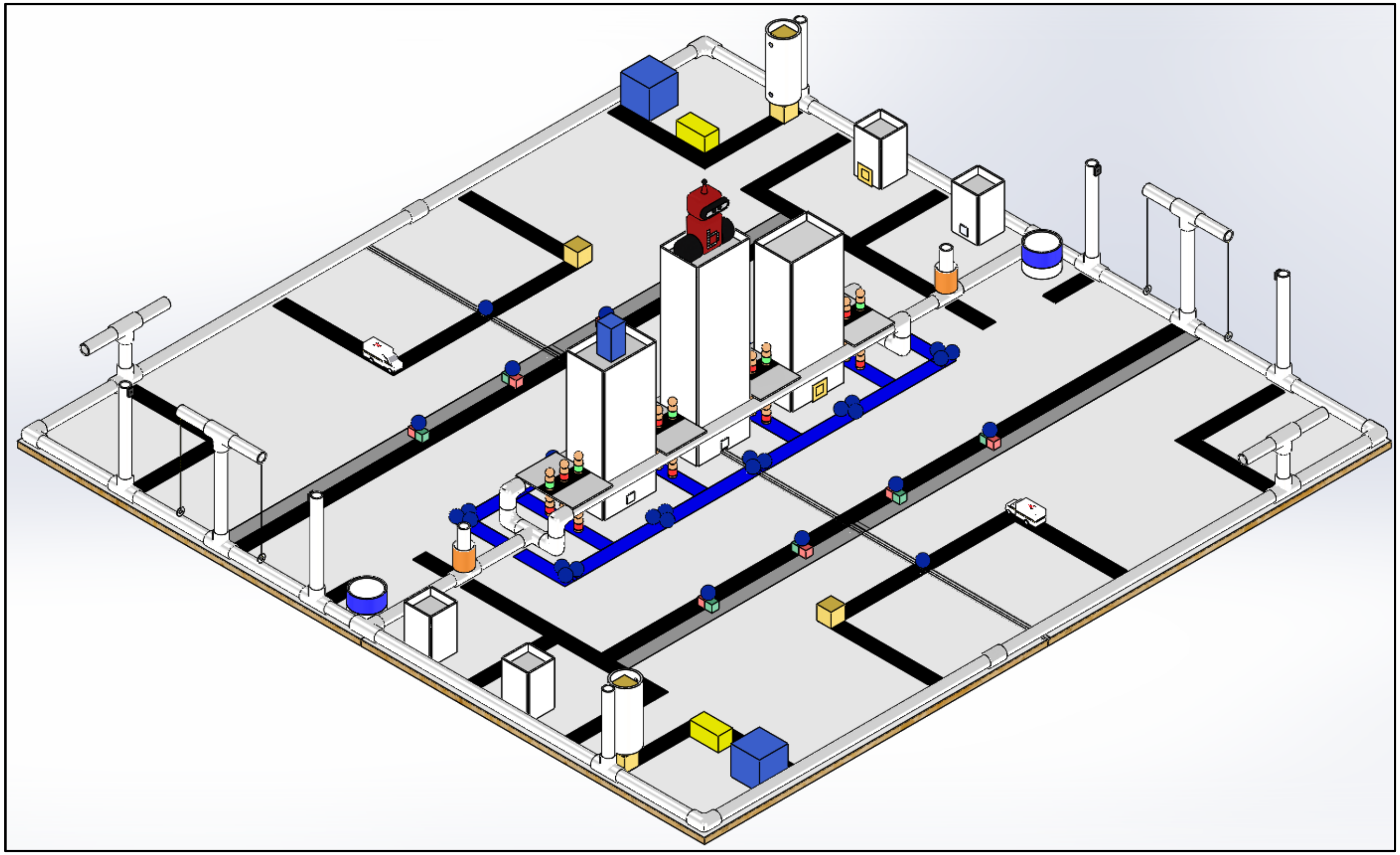
Social Media

T-shirts and Awards

What to do After the Workshop

Botball Team Home Base

Found at www.KIPR.org



Botball Team Home Base

KIPR Support

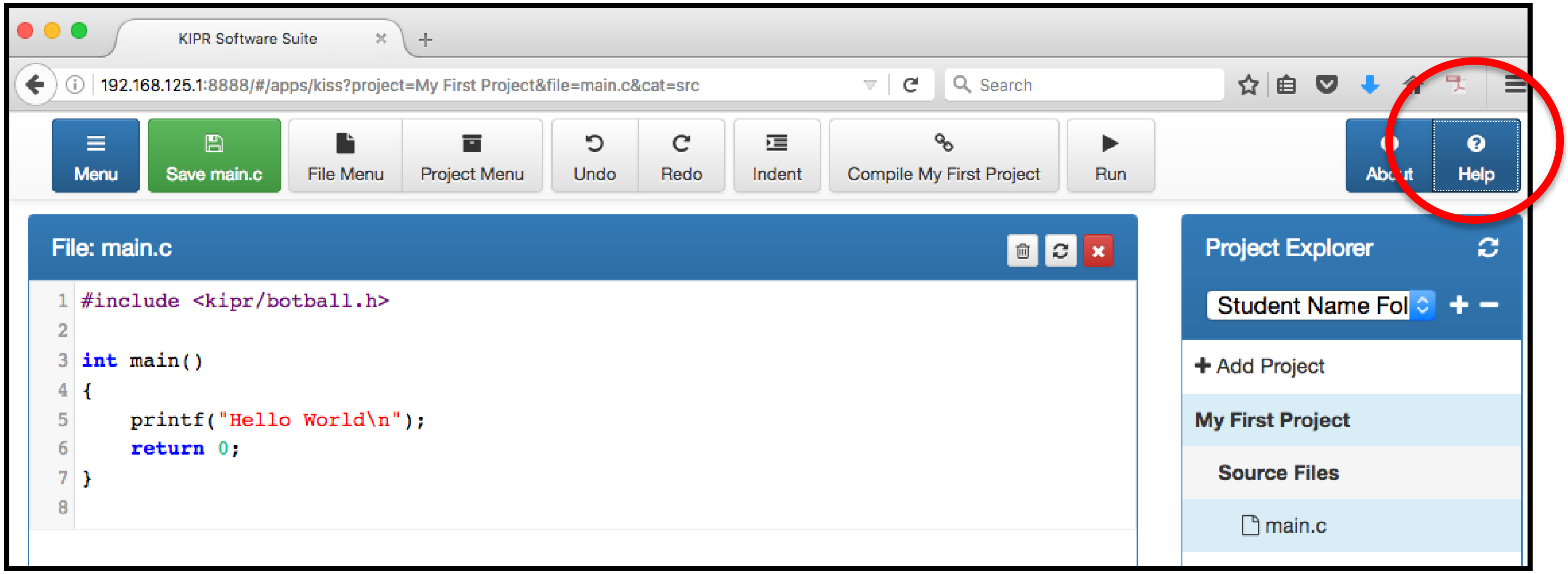
- **E-mail: support@kipr.org**
- **Phone: 405-579-4609**
- **Hours:** M-F, 8:30am-5:00pm CT

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 2019 Game Documents

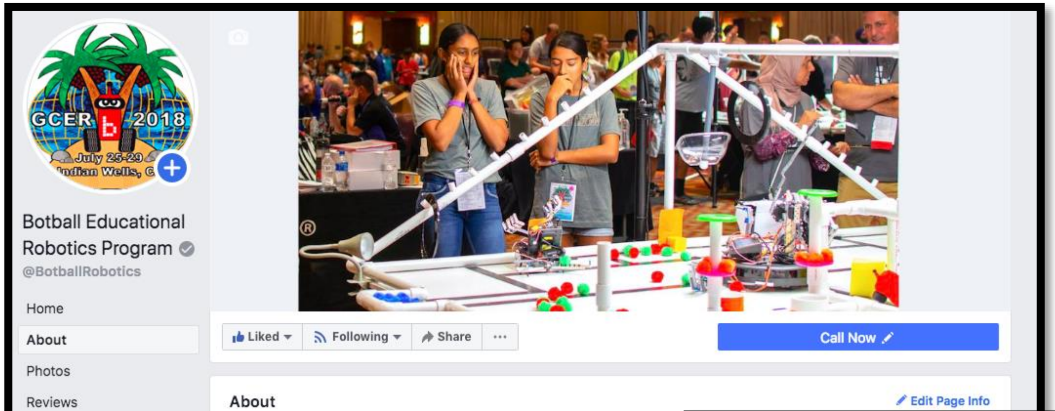
Wallaby Library Documentation

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



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

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

```

Social Media



Botball Educational Robotics Program

@BotballRobotics

- Home
- About
- Photos
- Reviews
- Events
- Videos
- Posts
- Services
- Shop
- Groups

Liked ▾ | Following ▾ | Share | ···

Call Now

About

Edit Page Info

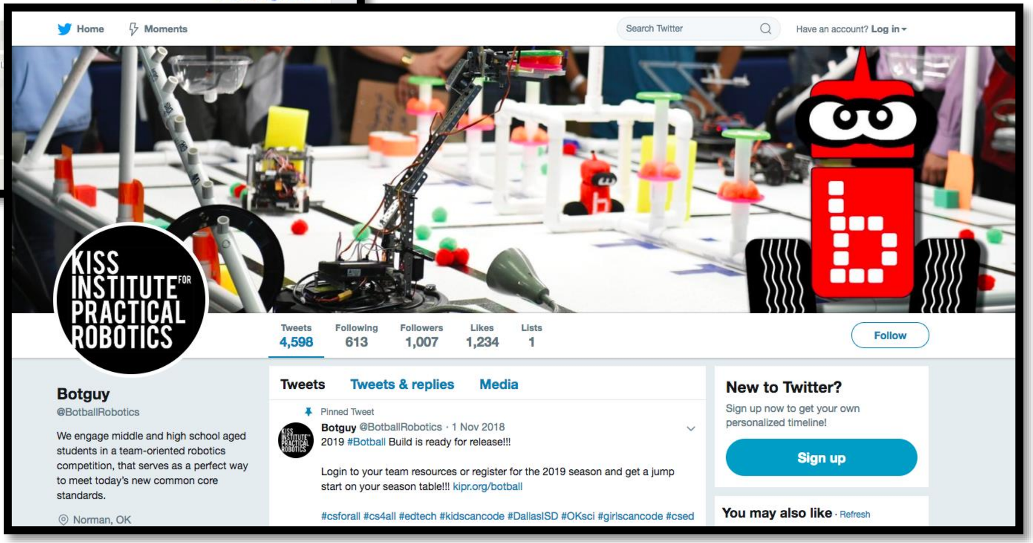
FIND US

1818 W Lindsey St
Norman, Oklahoma

Get Directions

m.me/BotballRobotics

Call (405) 579-4609



Home | Moments | Search Twitter | Have an account? Log in ▾

Tweets	Following	Followers	Likes	Lists
4,598	613	1,007	1,234	1

Follow

Botguy

@BotballRobotics

We engage middle and high school aged students in a team-oriented robotics competition, that serves as a perfect way to meet today's new common core standards.

Norman, OK

Tweets | Tweets & replies | Media

Pinned Tweet

Botguy @BotballRobotics · 1 Nov 2018

2019 #Botball Build is ready for release!!!

Login to your team resources or register for the 2019 season and get a jump start on your season table!!! kipr.org/botball

#csforall #cs4all #edtech #kidscancode #DallasISD #OKsci #girlscancode #csed

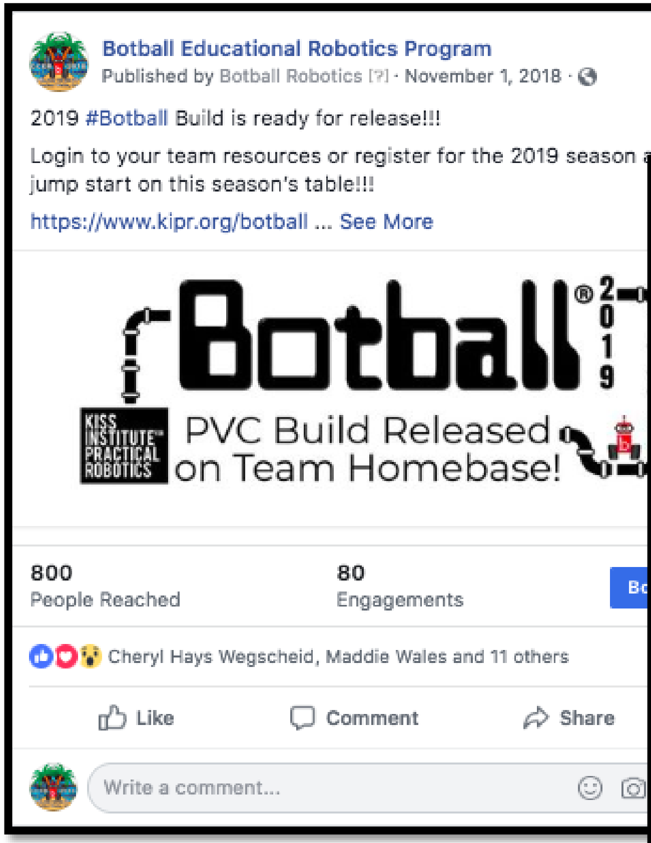
New to Twitter?

Sign up now to get your own personalized timeline!

Sign up

You may also like · Refresh

Social Media



Botball Educational Robotics Program
Published by Botball Robotics [?] · November 1, 2018 ·

2019 #Botball Build is ready for release!!!

Login to your team resources or register for the 2019 season and get a jump start on this season's table!!!

https://www.kipr.org/botball ... See More

Botball® 2019
KISS INSTITUTE PRACTICAL ROBOTICS
PVC Build Released on Team Homebase!

800 People Reached

80 Engagements

Cheryl Hays Wegscheid, Maddie Wales and 11 others

Like
Comment
Share
Write a comment...

Botball Educational Robotics Program
Published by Ashley Borgerding [?] · October 6, 2018 ·

Minefaire Botguy Kipr #KIPR #Botball #JuniorBotballChallenge

KISS INSTITUTE FOR PRACTICAL ROBOTICS
OFFICIAL MINECRAFT COMMUNITY EVENT
MINEFAIRE
BOTBALL ROBOTICS
kipr.org

Botguy @BotballRobotics · 21 Dec 2018
Wishing everyone Happy Holidays from the KIPR team!

Tournament Awards

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 team members haven't been recruited, then 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!
- There is a limited build time before the tournament.
- The workshop will still be fresh in the mind, so 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 Gannt 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 building a *full* game board is not an option, then try building ½ of the board.
- Tape the outline of the board onto a floor if the right type of flooring is available.

5. Organize the Botball Kit

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

6. Understand the Game

- Go over this with the students on the **first meeting** after the workshop.

Thanks, Have a Great Season!

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/WQ8CQ65