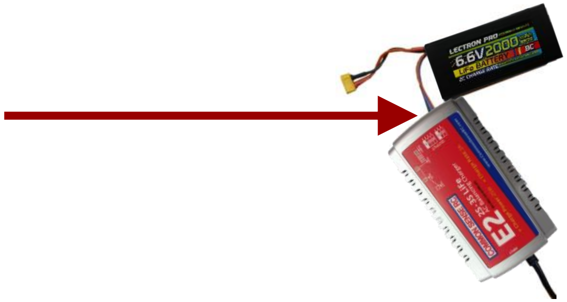
Welcome to Botball 2017!

Before we get started…

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

KIPR Robotics
Controller - Wallaby



1. Open the "**2017 Parts List**" folder, which contains files that list all of your Botball robot kit components. **Please go through the lists and verify that you have received everything.**
2. Build the **DemoBot**.

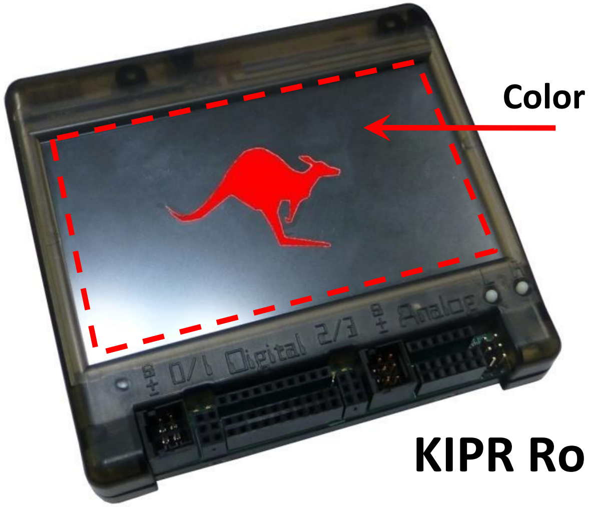
Raise your hand if you need help or have questions.

Charging the Controller's Battery

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

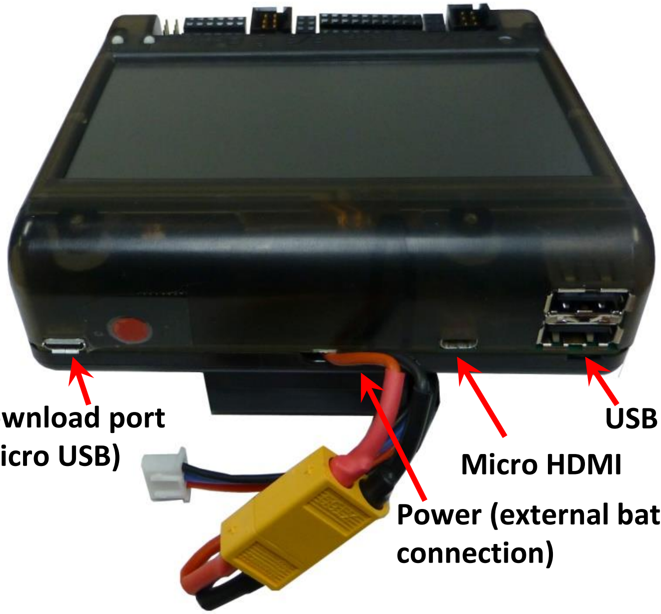
- The standard power pack is a **lithium iron (LiFe) battery**, a safer alternative to lithium polymer batteries. The safety rules applicable for recharging any battery still apply:
  - **Do NOT leave the battery unattended** while charging.
  - Charge in a cool, open area away from flammable materials.

KRC Wallaby Controller Guide



Color Touch Screen

KIPR Robotics Controller Wallaby

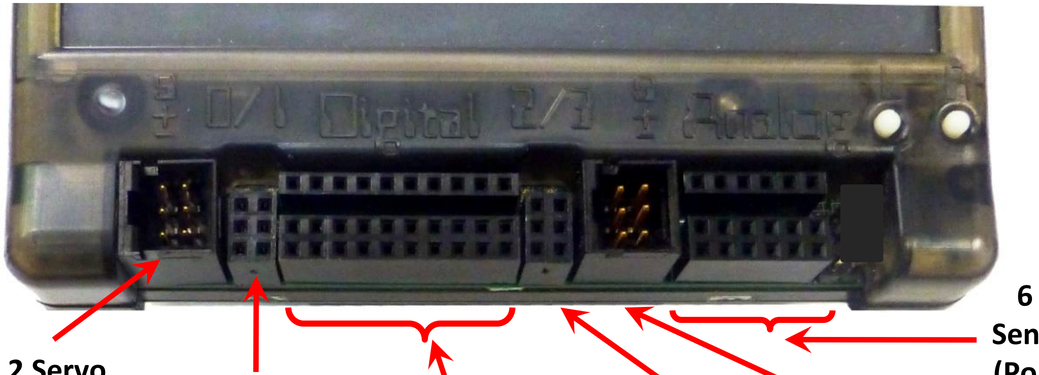


Download port (micro USB)

USB

Micro HDMI

Power (external battery connection)



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)

6 Analog Sensor Ports (Port # 0 - 5)

Power Switch

Making the Connection

All connections are as follows:

- **Yellow to Yellow** (battery to controller)
- **White small to White small** (charger to battery)
- **Black to Black** (motors, servos, sensors)

Wallaby Power

- The KIPR Robotics Controller – Wallaby, uses an external battery pack for power.
  - It will void your 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 your battery to the point where it can no longer be charged. If you plug your battery into the charger and the blue lights continue to flash then you have probably drained your battery to the point where it cannot be charged again. You can purchase a replacement battery from www.botballstore.org.

Wallaby Power Down

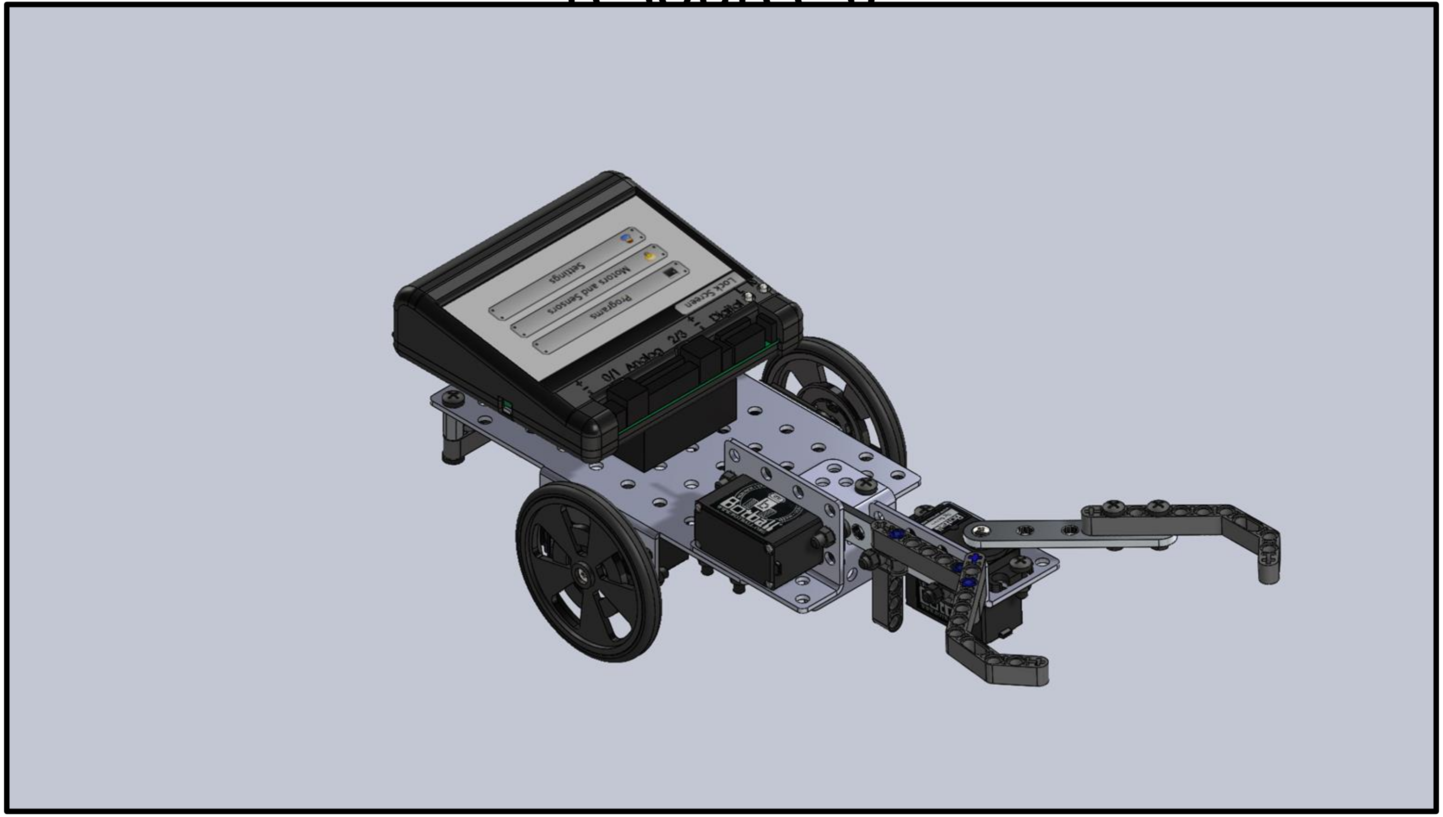
- From the Software Suite select *Shutdown*
  - Select *Yes*
- From the Wallaby Home Screen press *Shutdown*
  - Select *Yes*
- Go to your Wallaby screen and check to see if it is halted
- Slide the power switch to off AND unplug the battery, using the yellow connectors, being careful not to pull on the wires

Build the DemoBots

Activity

Build your robot using the DemoBot Building Guide

(Found on the team Homebase under 2017 team resources)



Hi! I’m Botguy, the Botball mascot!

Botball 2017

Professional Development Workshop

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

v2017.01.06-2

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
- **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 Your Own Program
- Moving the DemoBot with Motors
- Fun with Functions
- Moving the DemoBot Servos
- Repetition, Repetition: Counting
- Making Smarter Robots with Sensors
- Repetition, Repetition: Reacting
- Making a Choice
- Line-following
- Homework

Day 2

- Botball Game Review
- Motor Position Counter
- Measuring Distance
- Color Camera
- Moving the iRobot *Create*: Part 1
- Moving the iRobot *Create*: Part 2
- iRobot *Create* Sensors
- Logical Operators
- Resources and Support

Thanks to our national sponsors!

NASA

COMMON SENSE RC

igus®

THE SAMUEL ROBERTS NOBLE FOUNDATION

iRobot®

SOLIDWORKS

Thanks to our regional sponsors!

ARVEST®

PRIA Practical Robotics Institute Austria

OU College of Engineering

ITCCC 青少年国际竞赛与交流中心 International Teenager Competition and Communication Center

BancFirst. Loyal To Oklahoma & You.℠

Oklahoma Aeronautics Commission

astellas Leading Light for Life

HITACHI Inspire the Next

KIRKPATRICK FOUNDATION

eR4stem

Thanks to our regional hosts!

USC University of Southern California

NASA

OU

S·T·E·M Outreach
A MITRE Corporation Initiative

SOUTHERN ILLINOIS UNIVERSITY EDWARDSVILLE

UNIVERSITY OF MARYLAND

GROSSMONT COLLEGE

UNF UNIVERSITY of NORTH FLORIDA

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

UMASS LOWELL

NM STATE UNIVERSITY

LOYOLA UNIVERSITY CHICAGO
1870
AD · MAJOREM · DEI · GLORIAM
Preparing people to lead extraordinary lives

tgm
Vienna Institute of Technology
Austria

Botball Overview

What and when?

GCER and ECER

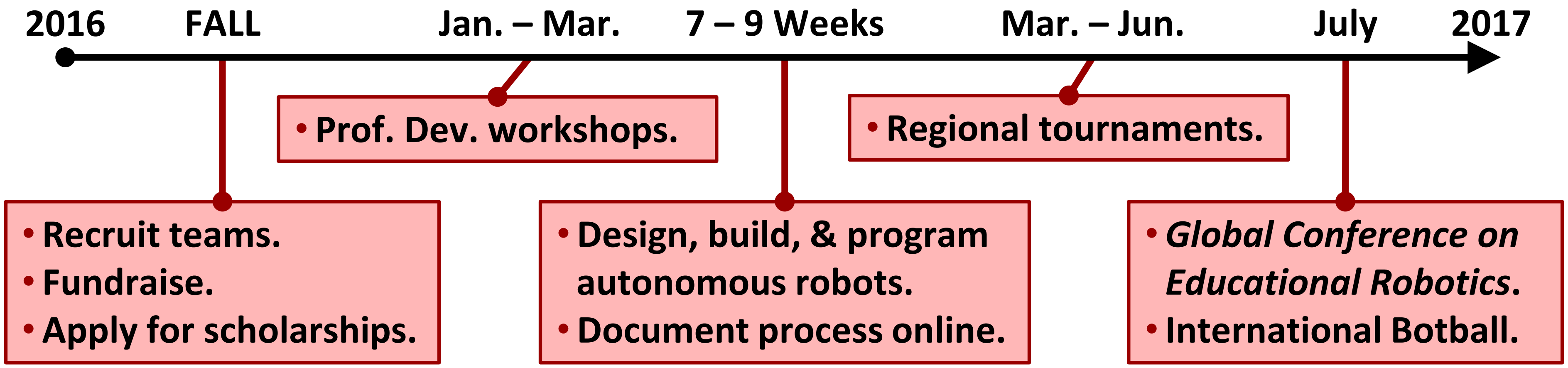
Preview of this year’s game

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?



2016

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.

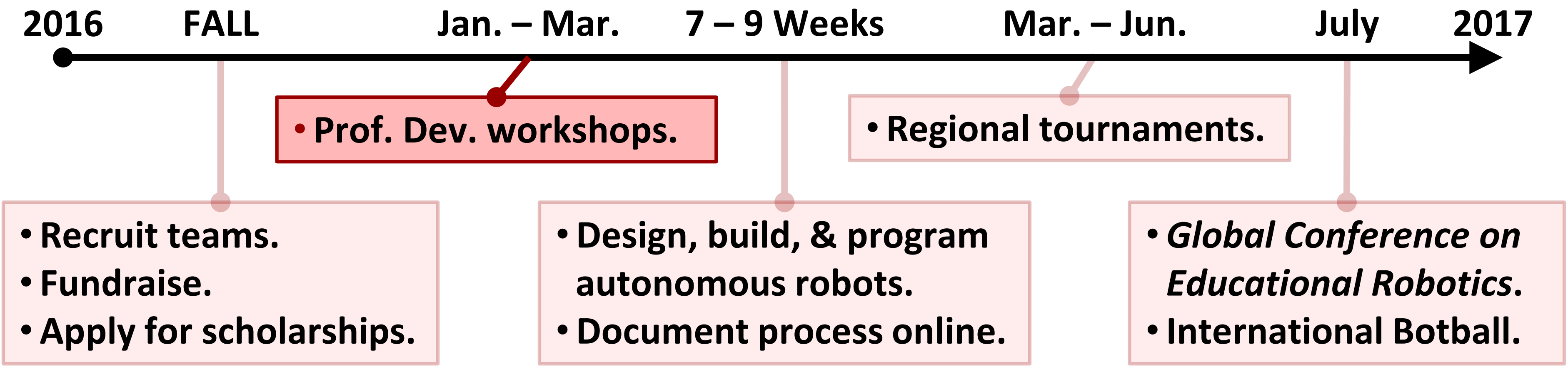
- Regional tournaments.

July

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

2017

When is Botball?



2016 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2017

- Prof. Dev. workshops.
- Regional tournaments.

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

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

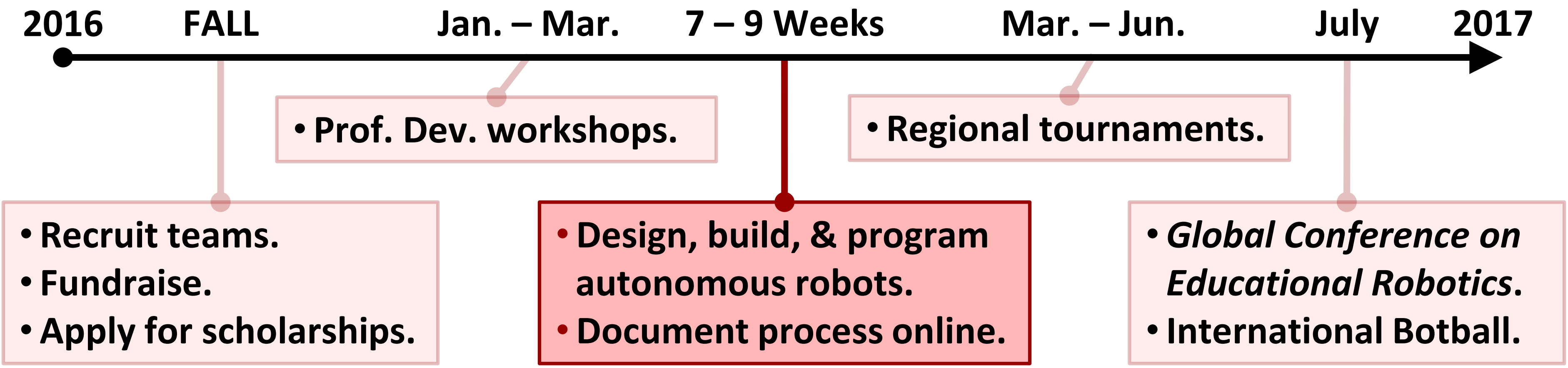
- *Global Conference on Educational Robotics.*
- International Botball.

YOU 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?



2016 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2017

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

- Prof. Dev. workshops.

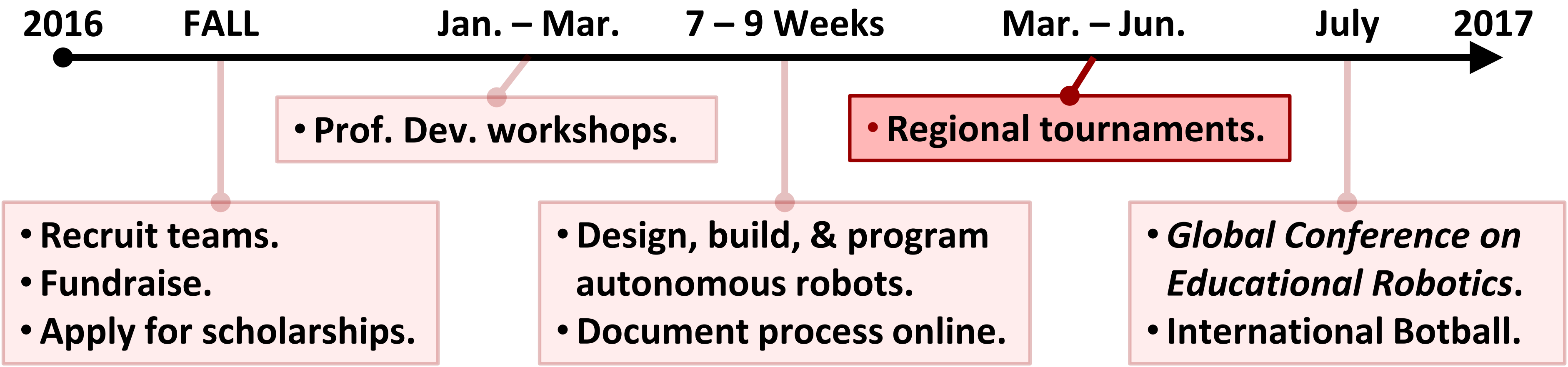
- Design, build, & program autonomous robots.
- Document process online.

- Regional tournaments.

- *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?



2016 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2017

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

- Prof. Dev. workshops.

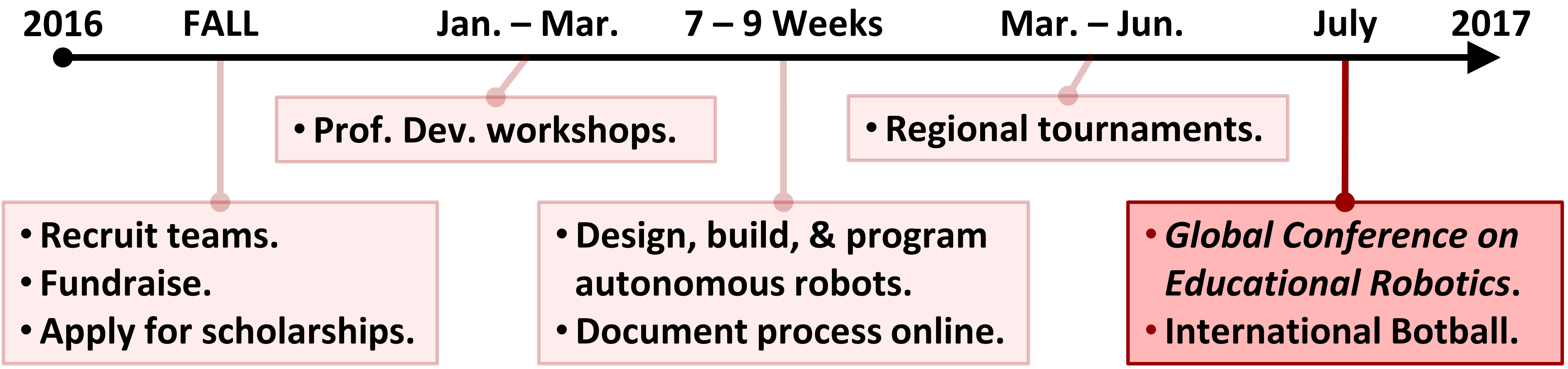
- Design, build, & program autonomous robots.
- Document process online.

- Regional tournaments.

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

- **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?



2016 FALL Jan. – Mar. 7 – 9 Weeks Mar. – Jun. July 2017

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

- Prof. Dev. workshops.

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

- Regional tournaments.

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

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

YOU ARE ALL ELIGIBLE!

GCER-2017

Global Conference on Educational Robotics

- Norman, Oklahoma
- July 8-12, 2017
- International Botball Tournament
- Autonomous Robotics Showcase

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

http://gcer.net

GCER-2017

Global Conference on Educational Robotics

Preconference classes on July 7th

International Junior Botball Challenge

KIPR Open Autonomous Robotics Game

- Botball for grown-up kids!

Autonomous
Aerial
Vehicle
Competition

ECER-2017

European Conference on Educational Robotics

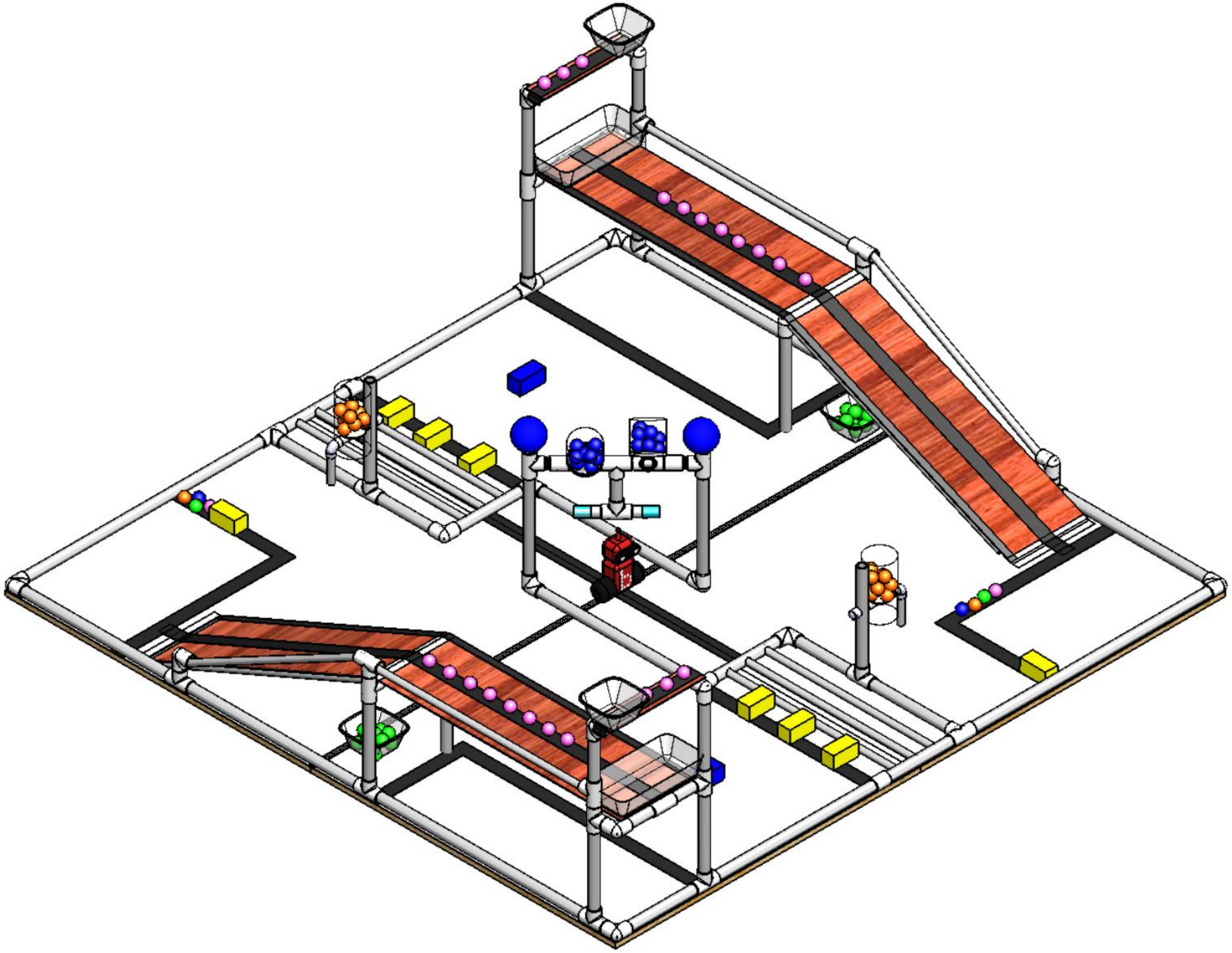
- Sofia Tech Park
  Sofia, Bulgaria
- April 24-28, 2017

- European Botball Competition
- Talks by Researchers and Students

Practical Robotics Institute Austria

www.pria.at

Botball game board



Professional Development Workshop
© 1993 – 2017 KIPR

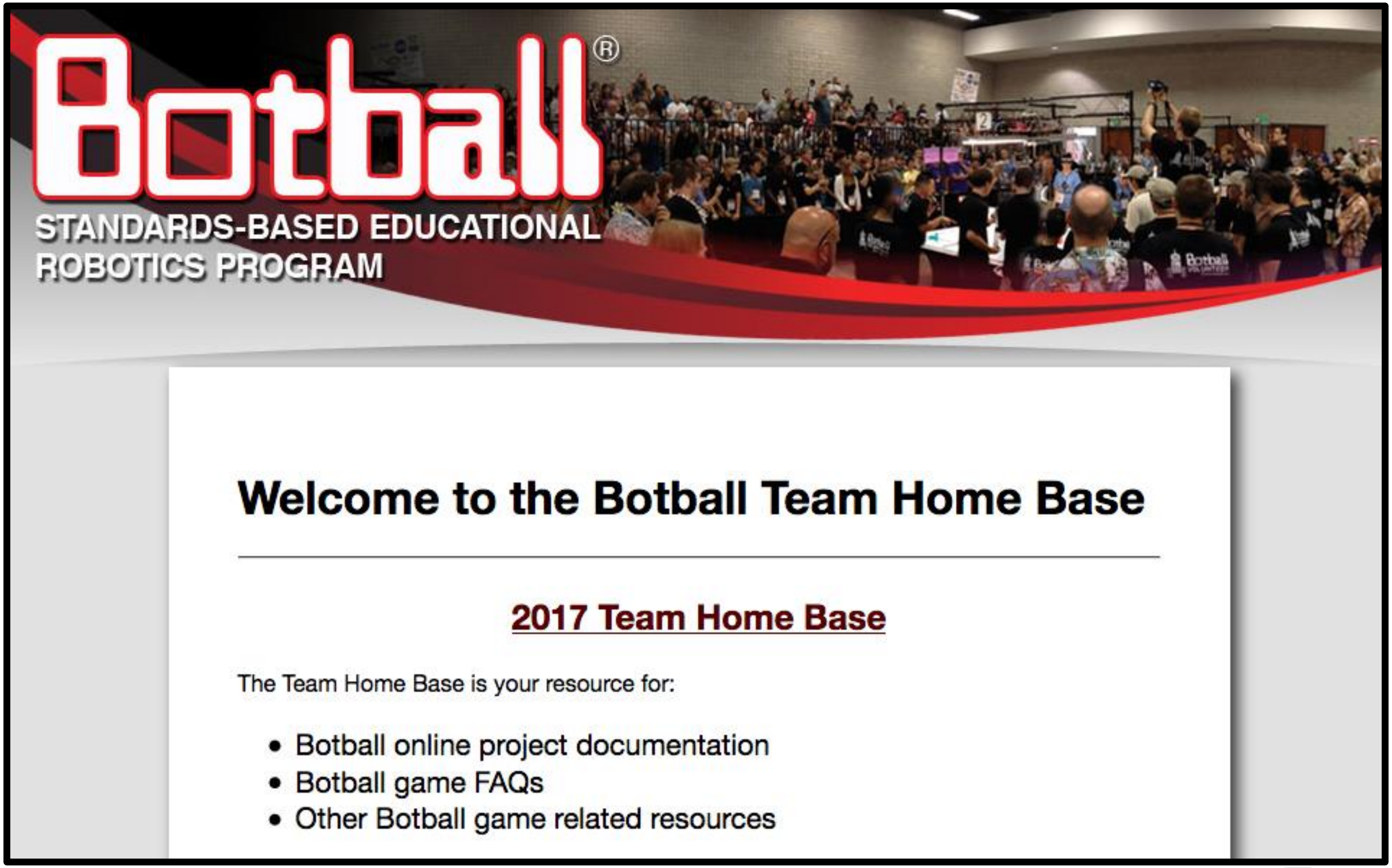
Homework for tonight

Review the game rules on your 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 Forum**.
  - You should **regularly visit this forum**.
  - You will **find answers to the game questions** there.

Botball Team Home Base

Found at http://homebase.kipr.org



Welcome to the Botball Team Home Base

2017 Team Home Base

The Team Home Base is your resource for:

- Botball online project documentation
- Botball game FAQs
- Other Botball game related resources

Preview of this year's Botball game

Robots Assisting a Modern Agricultural Operation

Managing a modern agricultural operation is hard work, but with the use of robotic technologies, the operations can become more efficient in their cultivation and use of resources, in particular water and fertilizer. It is planting time and Agrobot has just finished getting the family farm ready.

Hold your questions!

Game Q&A is tomorrow!

Getting Started with the KIPR Software Suite

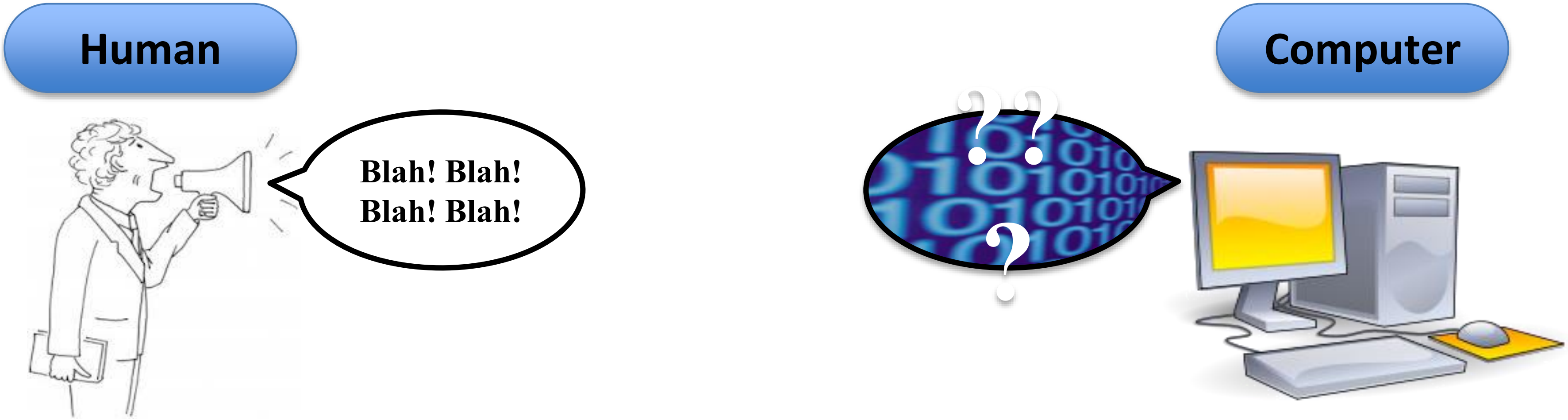
What is a programming language?

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?

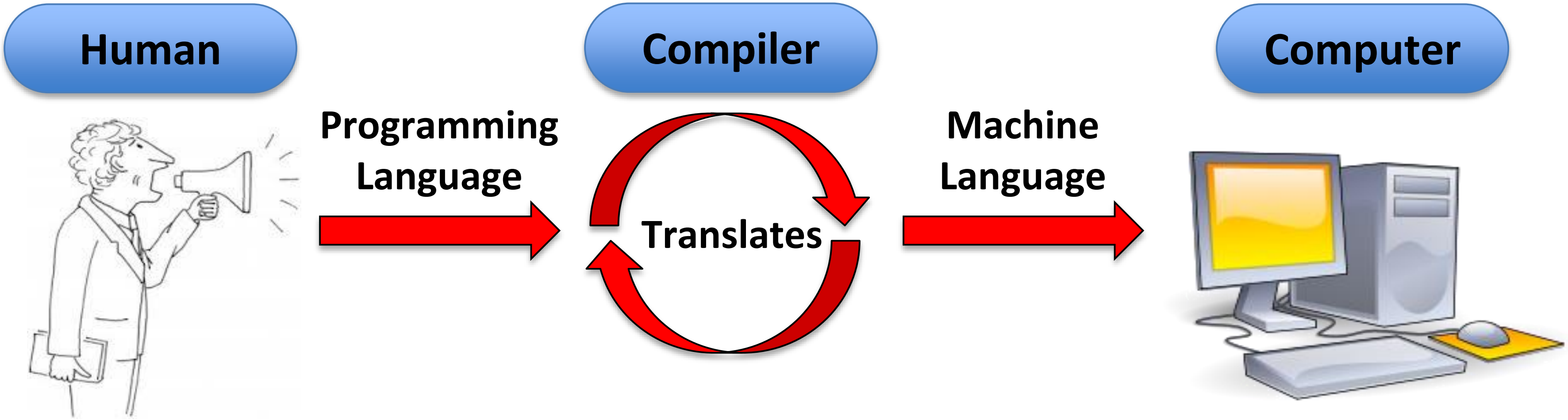
What is a *programming language*?



- **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

Compiler

Computer

Programming Language

Translates

Machine Language

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

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

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

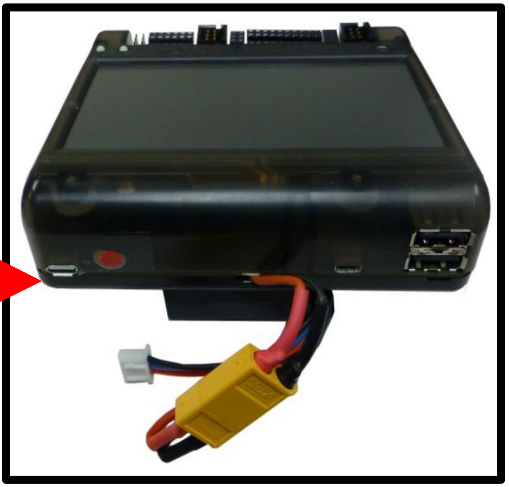
Connect the Wallaby to your computer at Workshop and Tournament

Activty

- Connect the **Wallaby** to your computer using **USB Cable**

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

1 Put micro USB end here



2 Attach USB to computer

1. Once your Wallaby has booted, the Wallaby will appear in the list of available Ethernet connections for your computer.
2. If you get 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)

Activity

1. Launch your web browser (such as Chrome or Firefox) and power up your Wallaby.
2. Copy this IP address into your browser's address bar followed by ":" and port number 8888; e.g.,

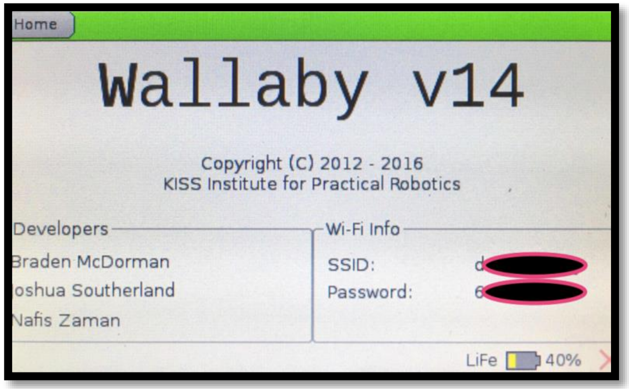
$$\underbrace{192.168.124.1}_{\text{IP address}}:\underbrace{8888}_{\text{Port \#}}$$

3. Note that USB cable IP address is 192.168.**124**.1:8888
4. The user interface for the package will now come up in your browser.

Connect the Wallaby to your computer, Smart Phone or Tablet At School

- Connect the **Wallaby** to your Browser 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**



Home

Wallaby v14

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

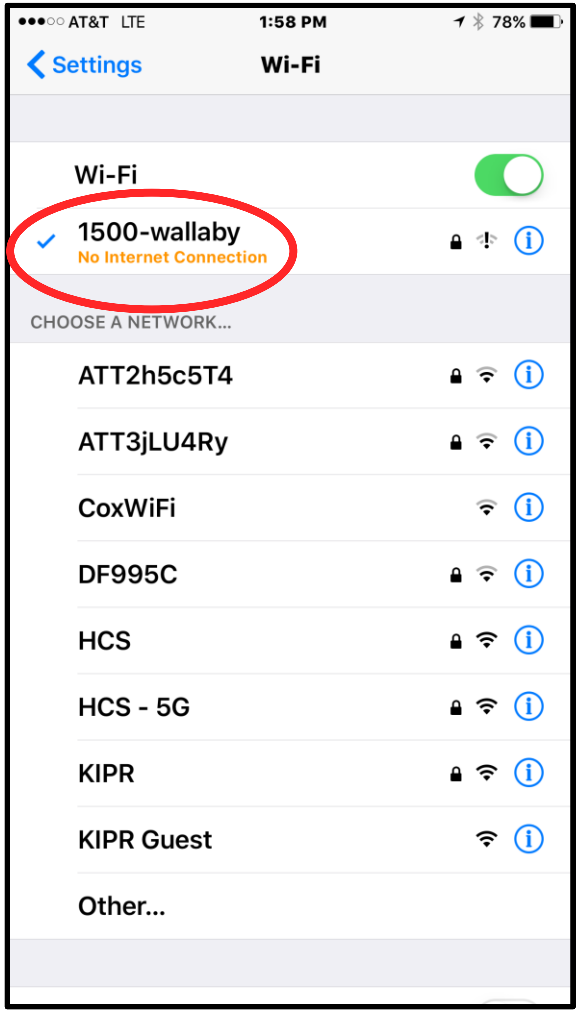
Developers
Braden McDorman
oshua Southerland
Nafis Zaman

Wi-Fi Info
SSID: [redacted]
Password: [redacted]

LiFe 40%

1. Use the info (Wallaby # and Pass Word) in the about page to connect via Wi-Fi

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

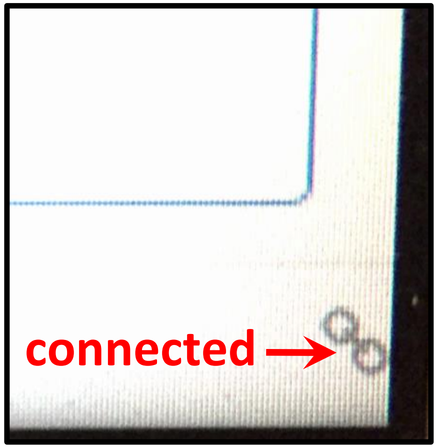
KIPR

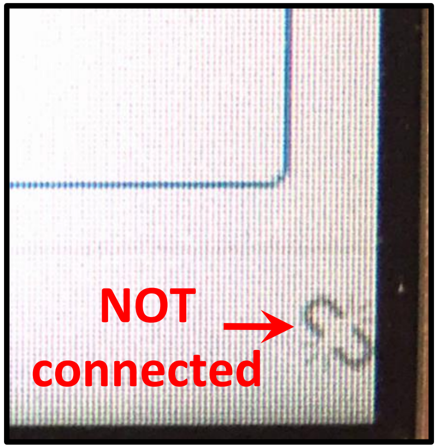
KIPR Guest

Other...

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

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





Loading the Starting Web Page (Wi-Fi)

Activity

1. Launch a web browser such as Chrome or Firefox and power up your Wallaby. Note that Internet Explorer **will not work**. Connect to the Wallaby via Wi-Fi.
2. Copy this IP address into your browser's address bar followed by ":" and port number 8888; e.g.,

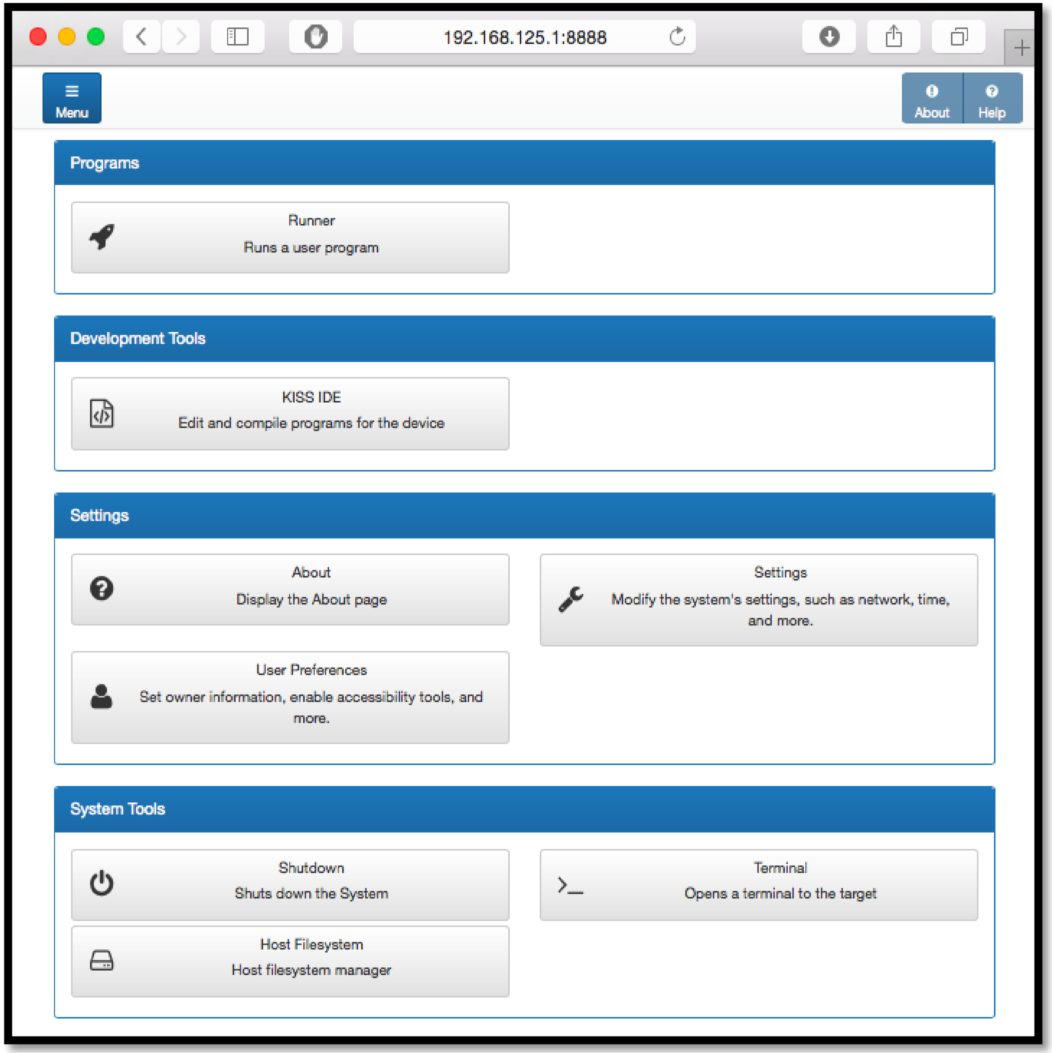
$$\underbrace{192.168.125.1}_{\text{IP address}}:\underbrace{8888}_{\text{Port \#}}$$

4. The user interface for the package will now come up in your browser.
5. You may use a computer, tablet or even a smart phone through Wi-Fi.

How can I write and compile my own source code?

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

The development 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

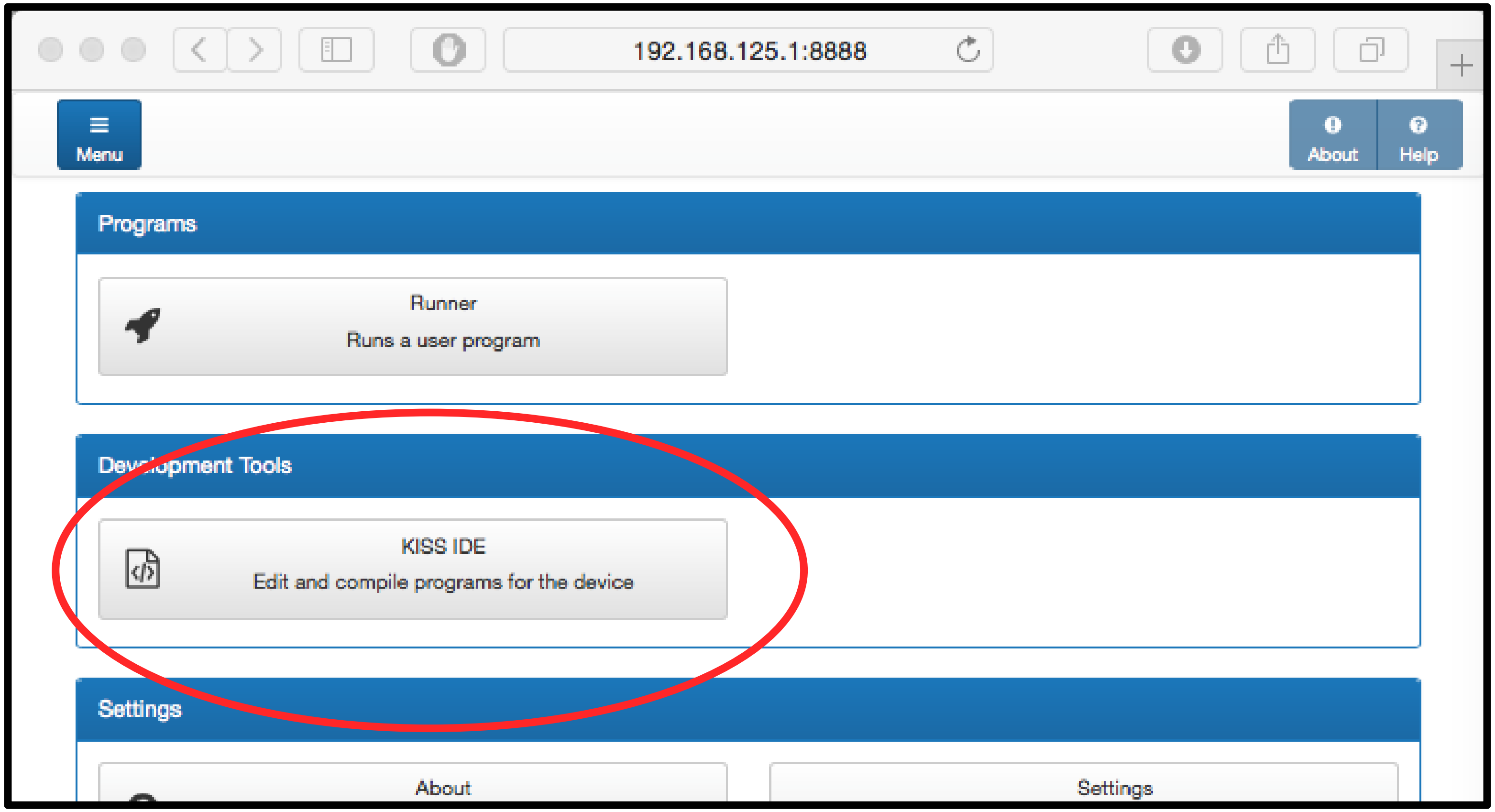
Shutdown
Shuts down the System

Terminal
Opens a terminal to the target

Host Filesystem
Host filesystem manager

Creating your first project

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



192.168.125.1:8888

Menu

About

Help

Programs

Runner
Runs a user program

Development Tools

KISS IDE
Edit and compile programs for the device

Settings

About

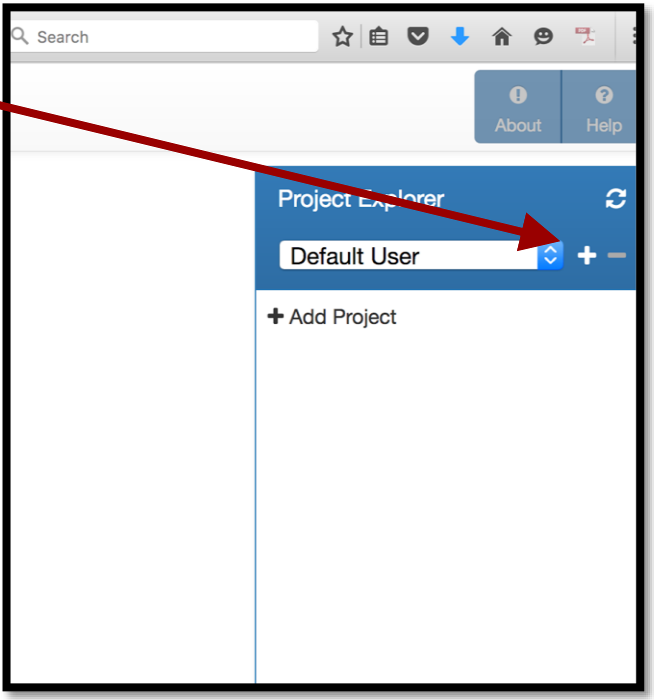
Settings

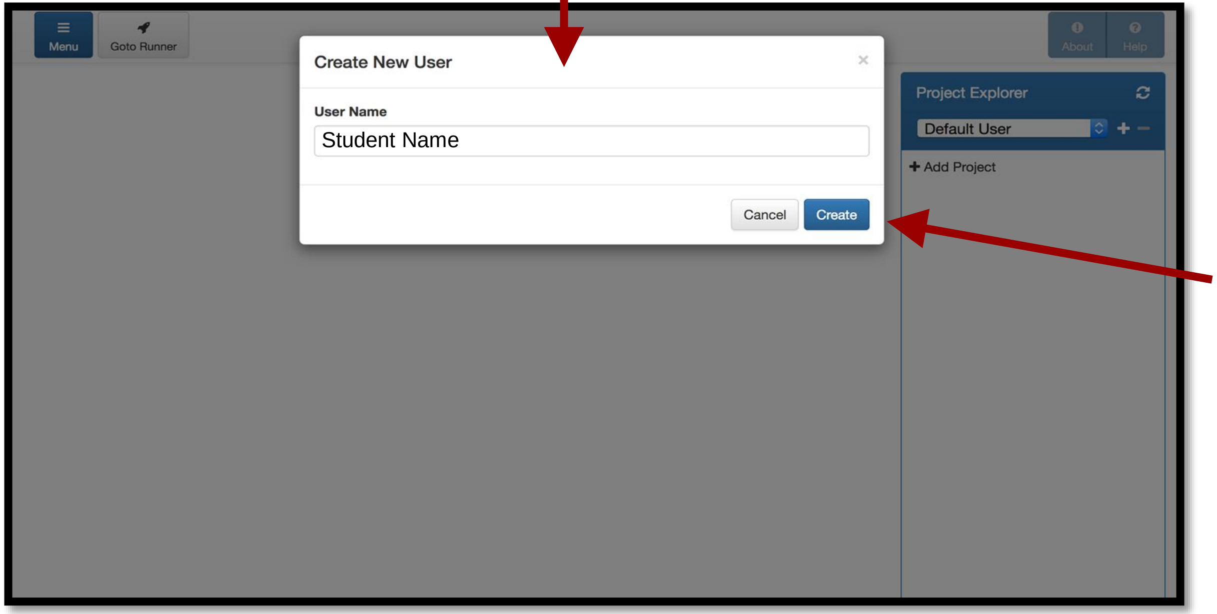
NOTE: The buttons might be in different locations depending on device type.

Creating your first user folder

Activity

1. Add a new user folder by clicking the **+** sign in the **Project Explorer**.
2. Name your new user folder by the student's name to help organization. All of your different projects will go into this user folder.



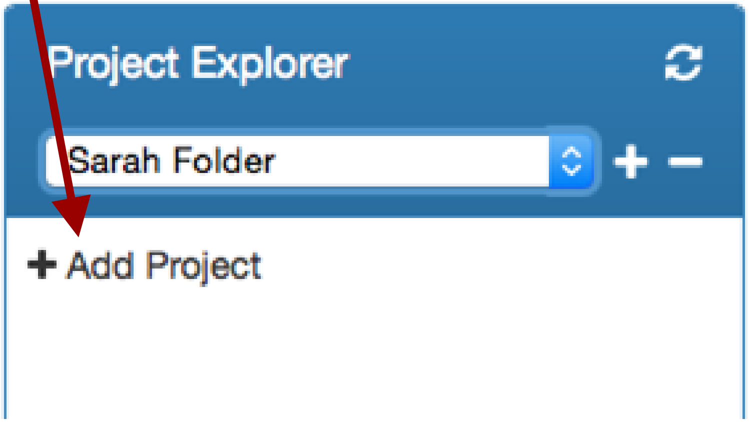


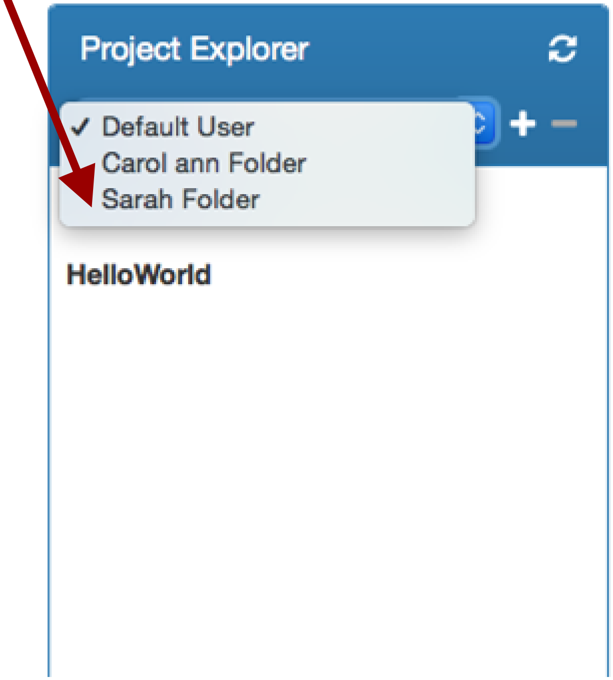
User Name: Student Name

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

Creating your first project

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

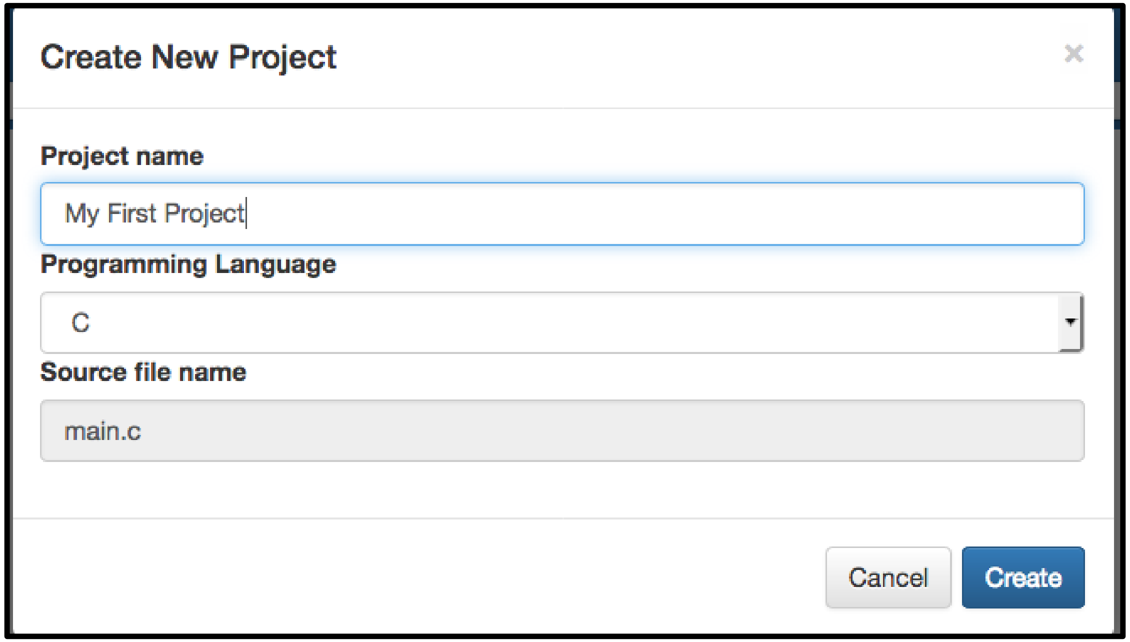




Name your project

Activity

1. Give your project a **descriptive name**
   - **Note:** you will have a lot of student's projects, so consider using their first name followed by the name of the activity.
2. Give a descriptive Source File Name as well. The Source File needs to end with a **.c**
   - Then press the ***Create*** button.



Create New Project

Project name: My First Project

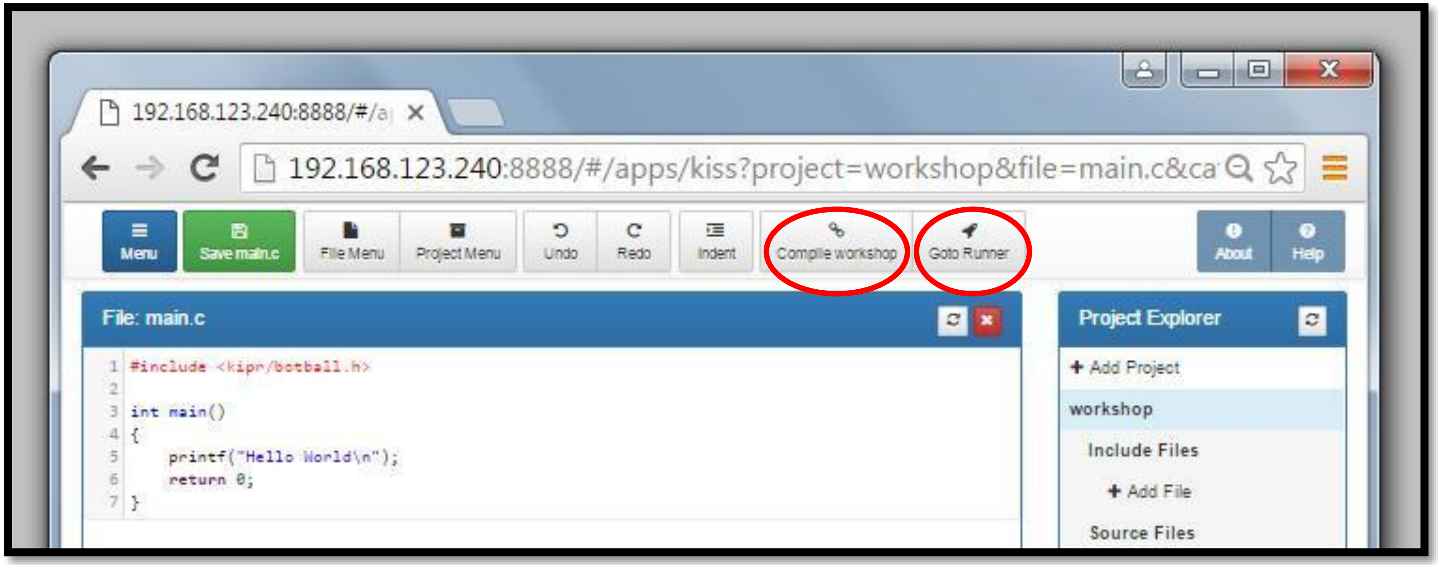
Programming Language: C

Source file name: main.c

Compile and Then Run Your Project

Activity

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



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

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

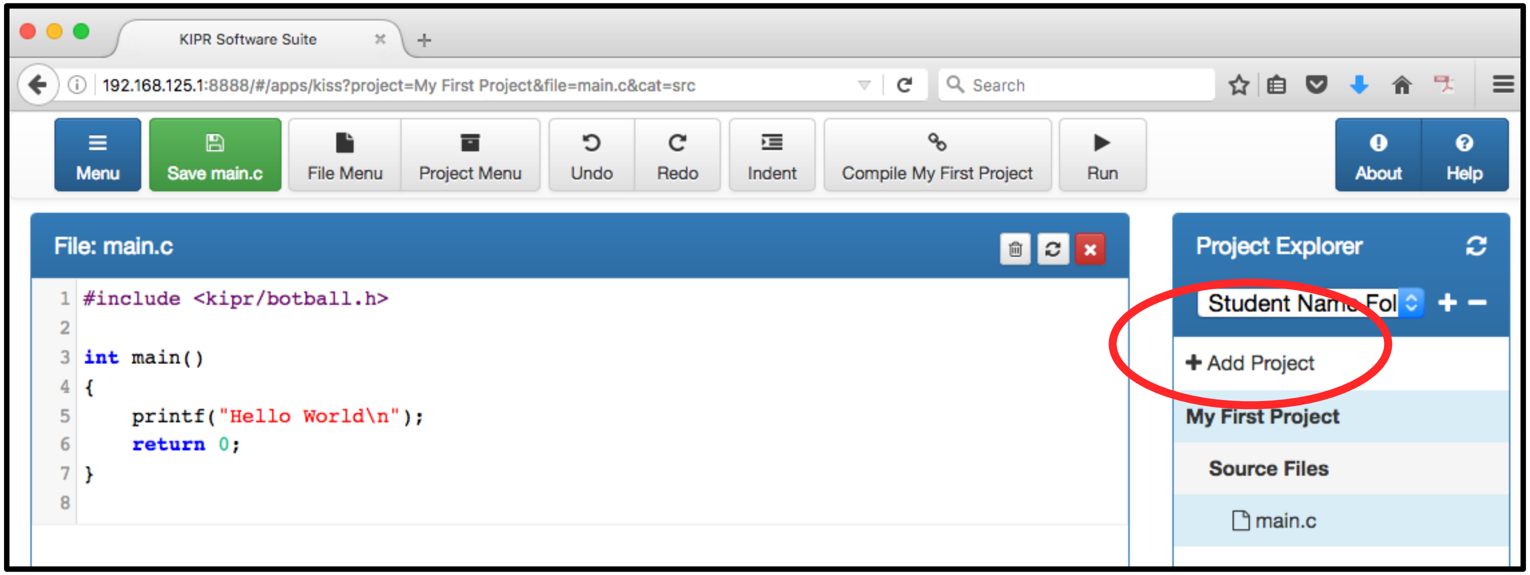
NOTE: When you compile, your project is automatically saved.

Starting another project

Activity

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

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



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

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

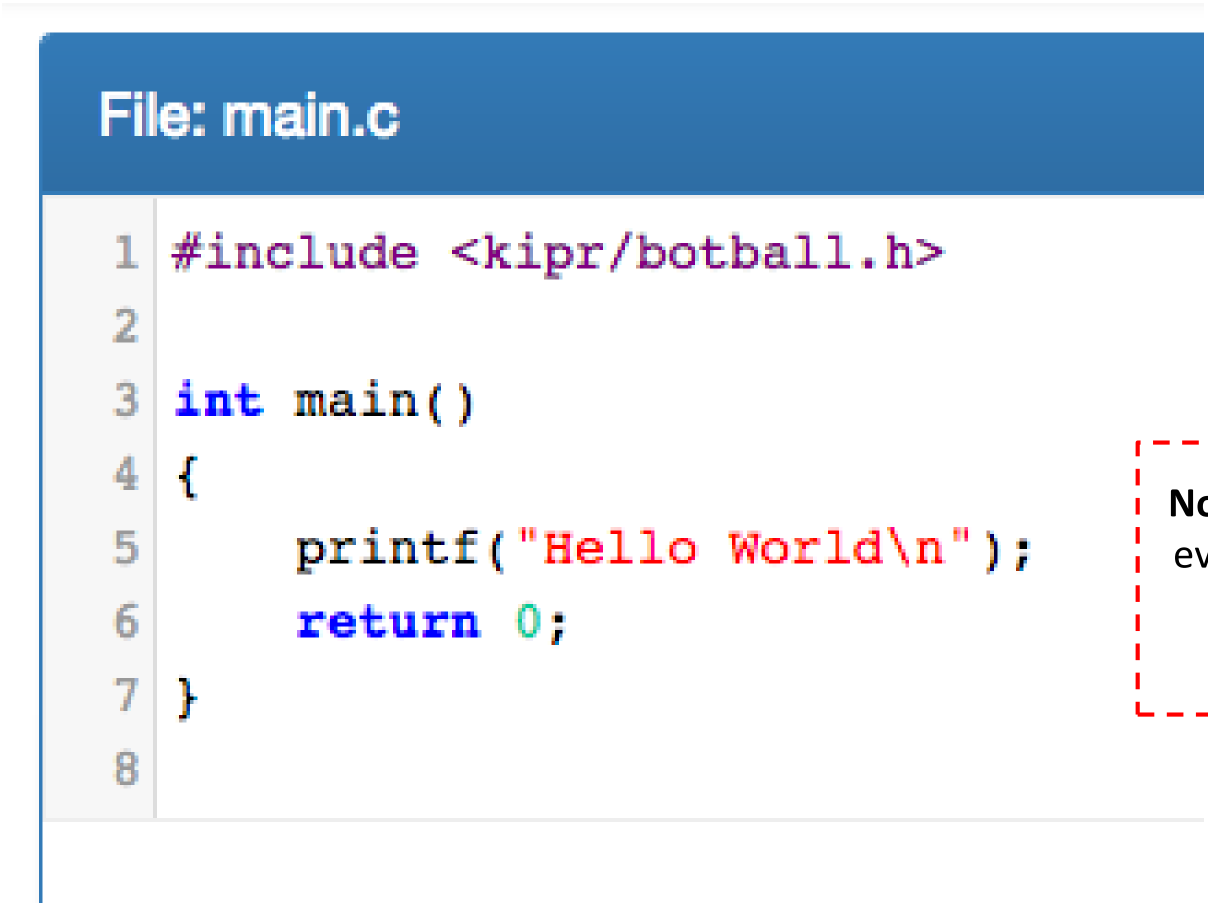
Explaining the “Hello, World!” C Program

Program flow and the main function

Programming statements and functions

Comments

“Hello, World!”



File: main.c

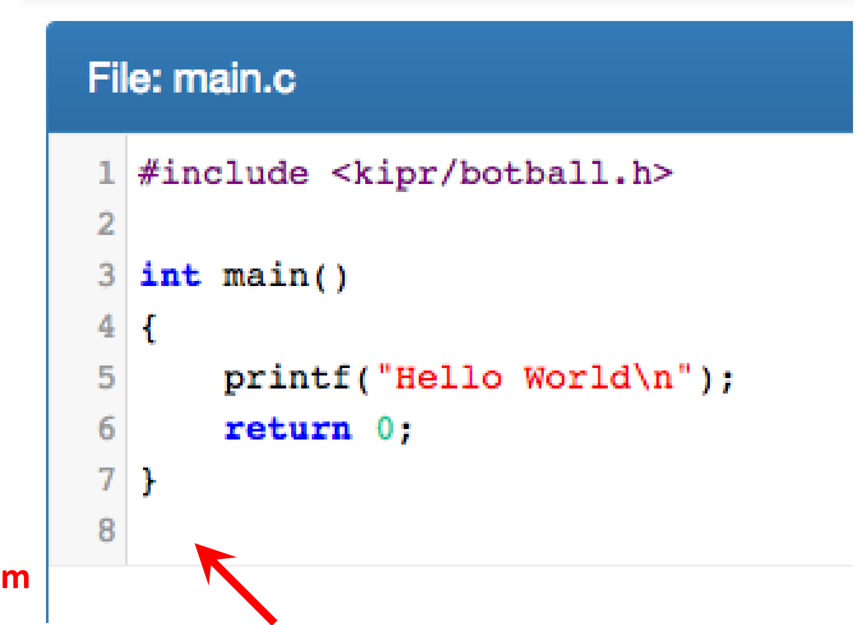
```c
#include <kipr/botball.h>

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

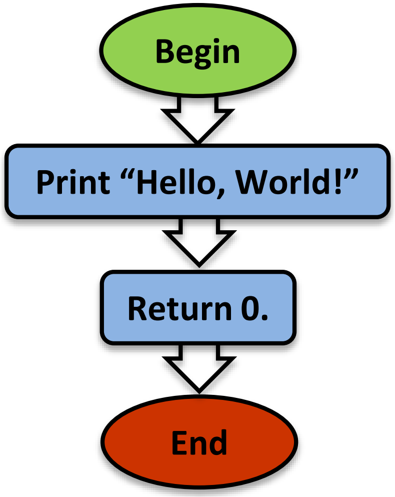
Top

Bottom



File: main.c

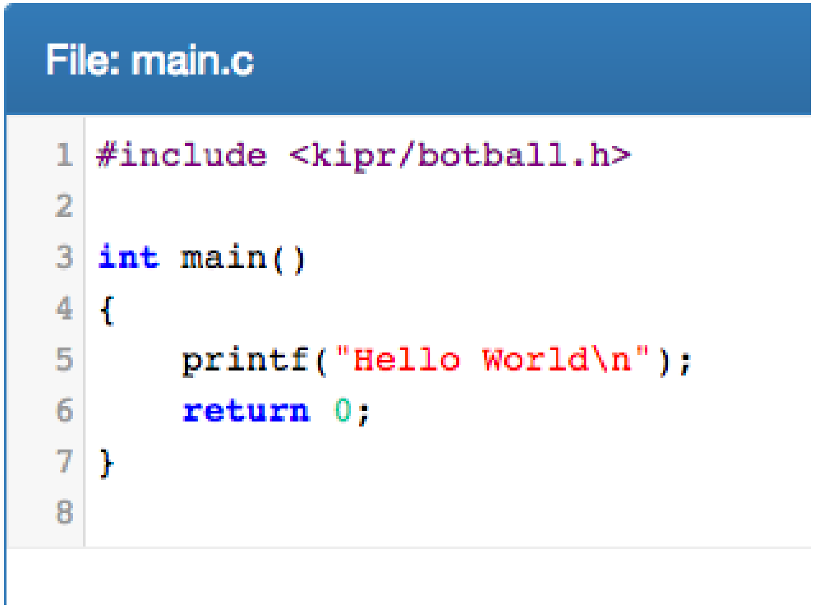
```
#include <kipr/botball.h>

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

```



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

Computers read incredibly quickly—
approximately 800-million lines per second!

Source code



File: main.c

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

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

```

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

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

The main function

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

```
// Created on Thu January 10 2017

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

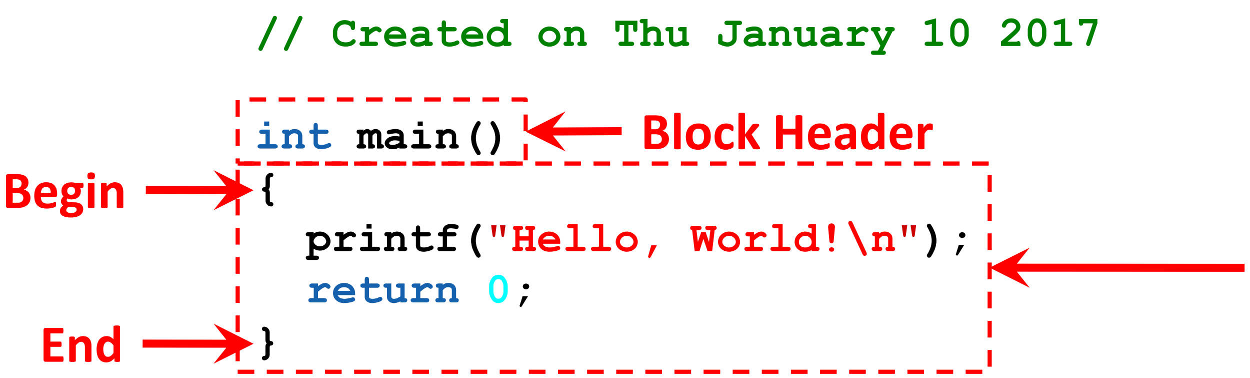
This is the **`main()` function**.

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

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

Block of code

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



```
// Created on Thu January 10 2017

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

Block Header

Begin

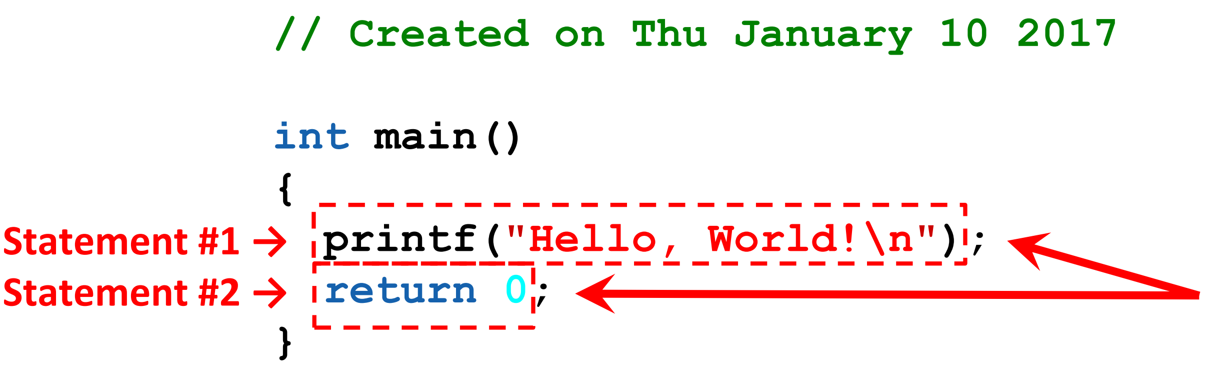
End

This is a **block of code**.

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

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

Programming statements



```
// Created on Thu January 10 2017

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

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**.

Ending a programming statement

```
// Created on Thu January 10 2017

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 statement** is missing a **period**, then it is a run-on sentence.

Ending the main function

```
// Created on Thu January 10 2017

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 the **last line before the } brace**.

Comments

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

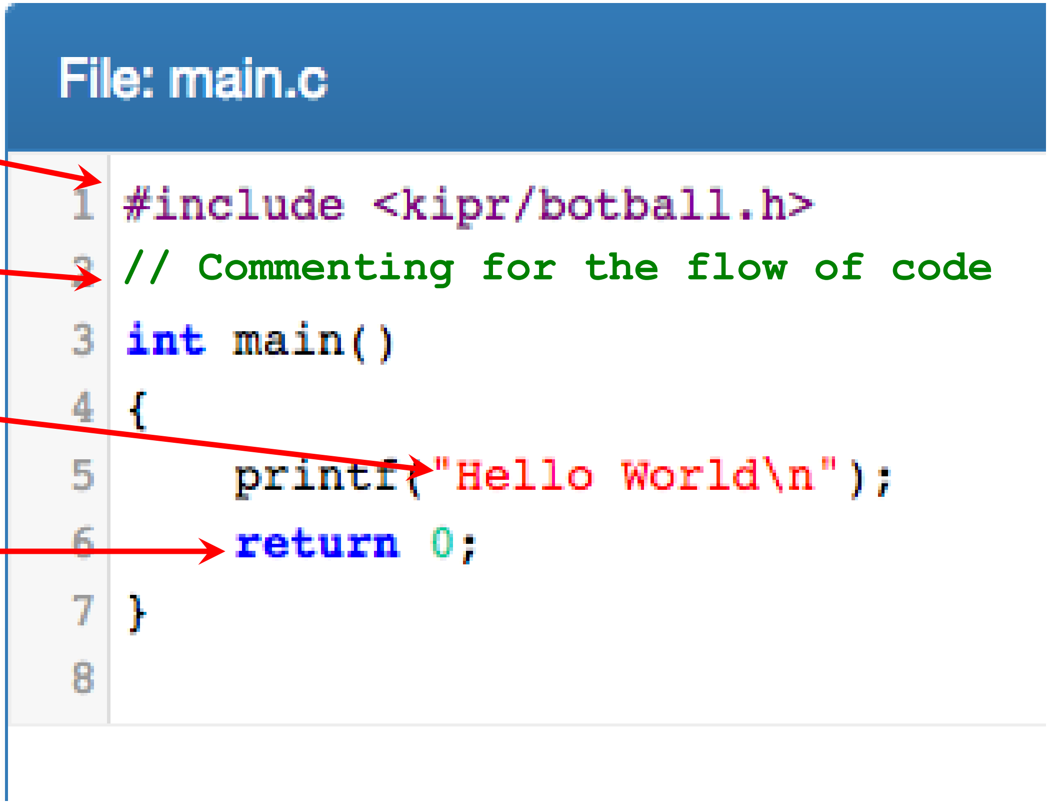
```
// Created on Thu January 10 2017

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

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

Text color highlighting

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

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



File: main.c

```
#include <kipr/botball.h>
// Commenting for the flow of code
int main()
{
    printf("Hello World\n");
    return 0;
}

```

Activity

Print your name

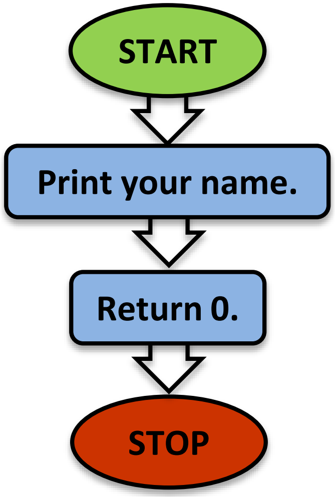
Description: Write a program for the KIPR Wallaby that prints your name.

Solution:

Source Code

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

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

Flowchart



Designing Your Own Program

Breaking down a task

Pseudocode, flowcharts, and comments

`wait_for_milliseconds` function

Debugging your program

Complex tasks → simple subtasks

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

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

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

Activity

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?

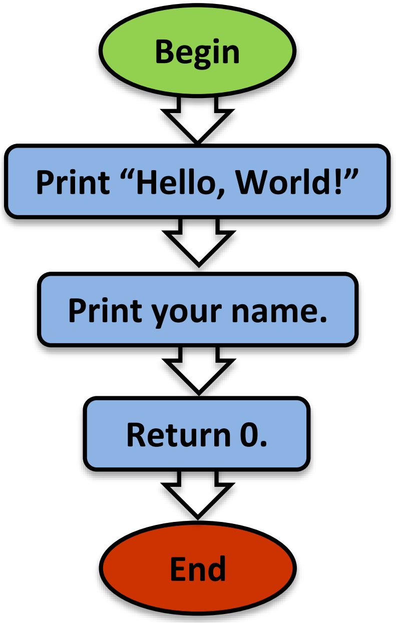
Pseudocode

```
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.
```

Flowchart



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

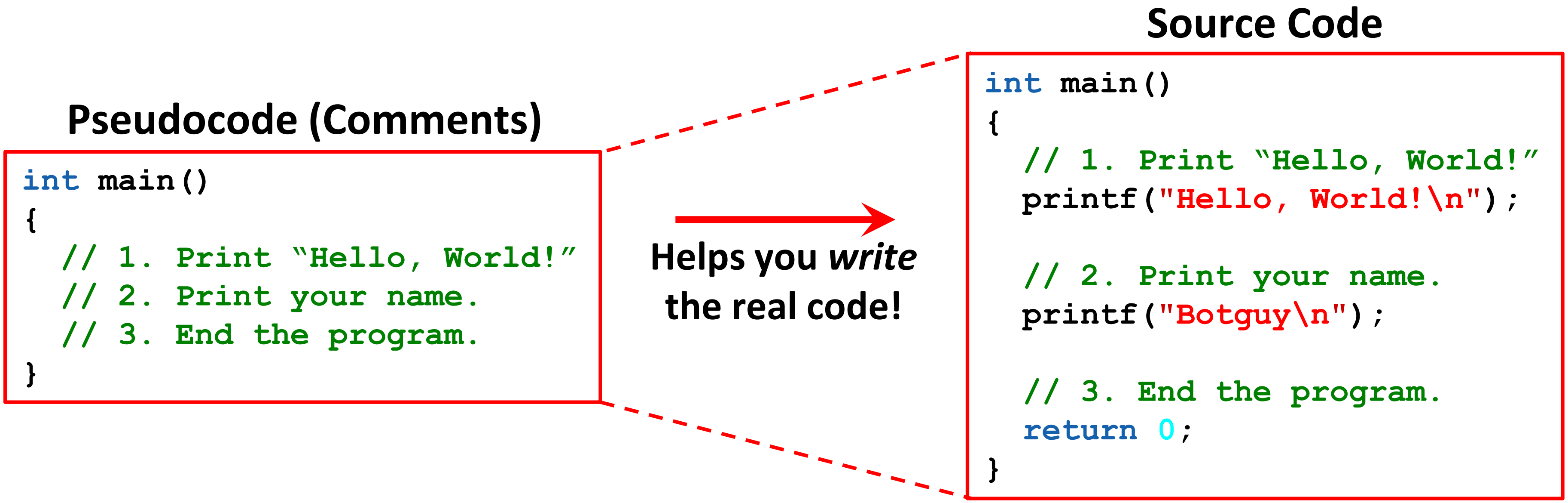
These are three different ways to do this.

Activity

Practice printing

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

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



Pseudocode (Comments)

```
int main()
{
  // 1. Print “Hello, World!”
  // 2. Print your name.
  // 3. End the program.
}
```

Helps you *write* the real code!

Source Code

```
int main()
{
  // 1. Print “Hello, World!”
  printf("Hello, World!\n");

  // 2. Print your name.
  printf("Botguy\n");

  // 3. End the program.
  return 0;
}
```

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

Practice printing

Activity

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

- The Wallaby reads code and goes to the next line faster than a blink of your 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 **`wait_for_milliseconds()`**, later this can be shortened to **`msleep()`**

Let's use this!

Using wait_for_milliseconds

```c
int main()
{
  printf("slow ");
  wait_for_milliseconds(2500);  // wait for 2500 ms
  printf("printer\n");
  return 0;
}
```

What is this?

Another name for **`wait_for_milliseconds()`** is **`msleep()`**.
It is identical and shorter to type, but more difficult to remember.

`msleep(2500)` is the same as **`wait_for_milliseconds(2500)`**.

Activity

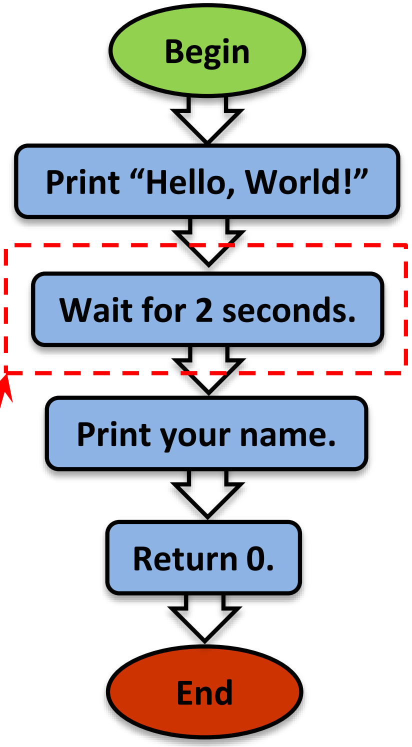
Waiting for some time

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

Activity

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

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

Pseudocode (Comments)

```
int main()
{
  // 1. Print “Hello, World!”
  // 2. Wait for 2 seconds.
  // 3. Print your name.
  // 4. End the program.
}
```

Source Code

```
int main()
{
  // 1. Print “Hello, World!”
  printf("Hello, World!\n");

  // 2. Wait for 2 seconds.
  wait_for_milliseconds(2000);

  // 3. Print your name.
  printf("Botguy\n");

  // 4. End the program.
  return 0;
}
```

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

Waiting for some time

Activity

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

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

Debugging Errors

!!! ERROR !!!

- If you do not follow the rules of the **programming language**, then the **compiler** will get confused and not be able to **translate** your **source code** into **machine code**—it will say “**Compile Failed!**”

- The Wallaby will try to tell you where it *thinks* the **error** is located.

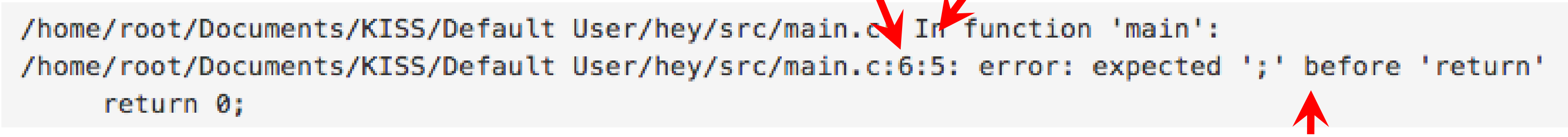
- The process of trying to resolve this **error** is called “**debugging**”.

- To test this, remove a *;* from one of your programs and compile it.

Activity

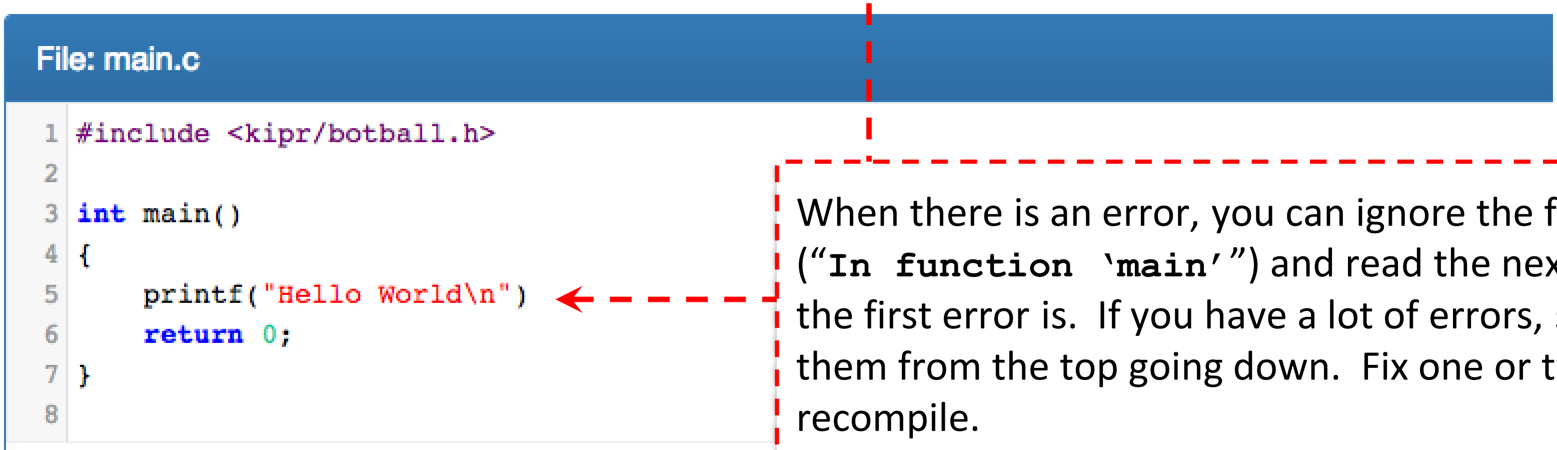
Debugging Errors

line # : col # (the error is <u>on or before</u> the 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;
```

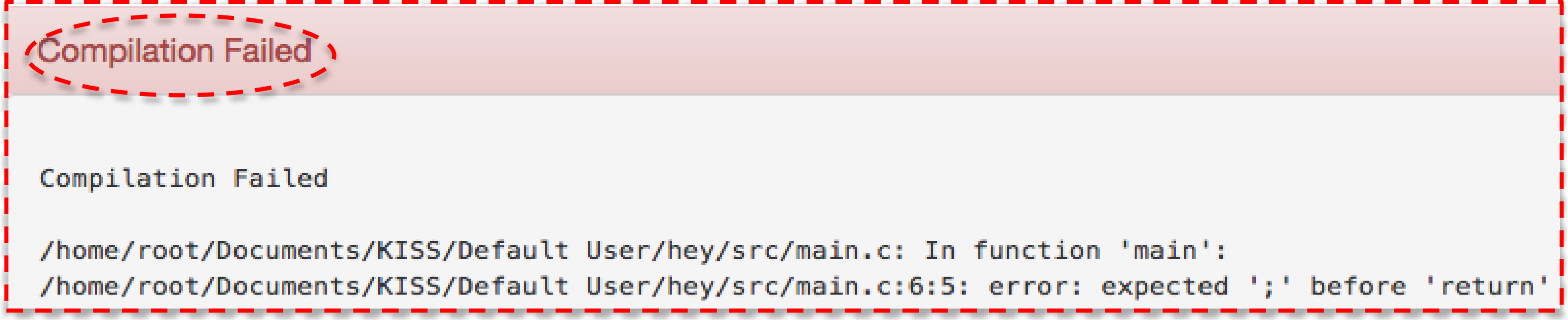
“ expected ; ” (semicolon)



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, you can ignore the first error line (“**In function ‘main’**”) and read the next to see what the first error is. If you have a lot of errors, start fixing them from the top going down. Fix one or two and recompile.



Compilation Failed

```
Compilation Failed

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

Moving the DemoBot with Motors

Plugging in motors (ports and direction)

`motor` functions

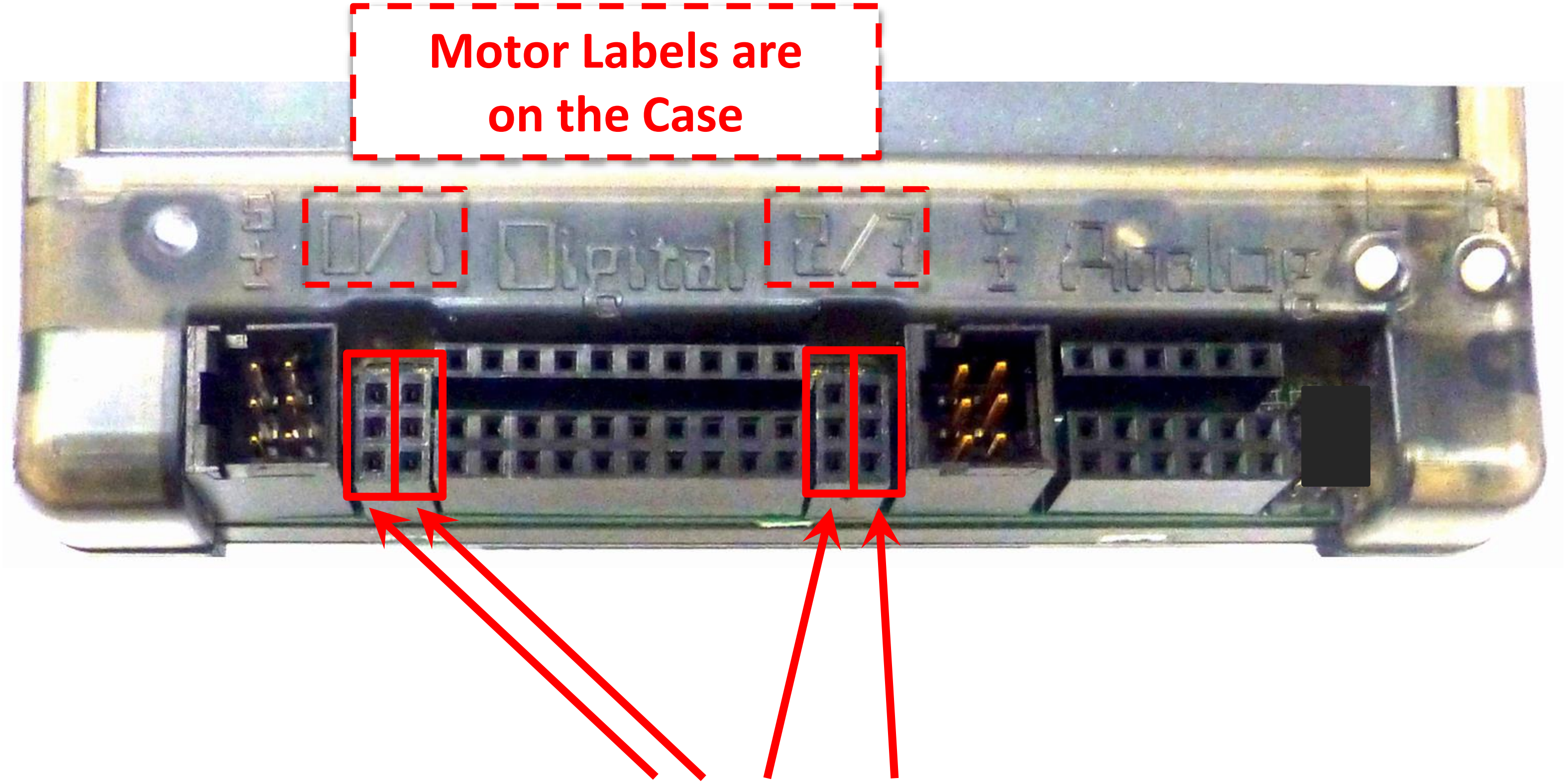
Check your robot's motor ports

Activity

- To program your robot to move, you need to know which **motor ports** your motors are plugged into.

- Computer scientists start counting at 0, so the **motor ports** are numbered **0**, **1**, **2**, and **3**.

Wallaby motor ports



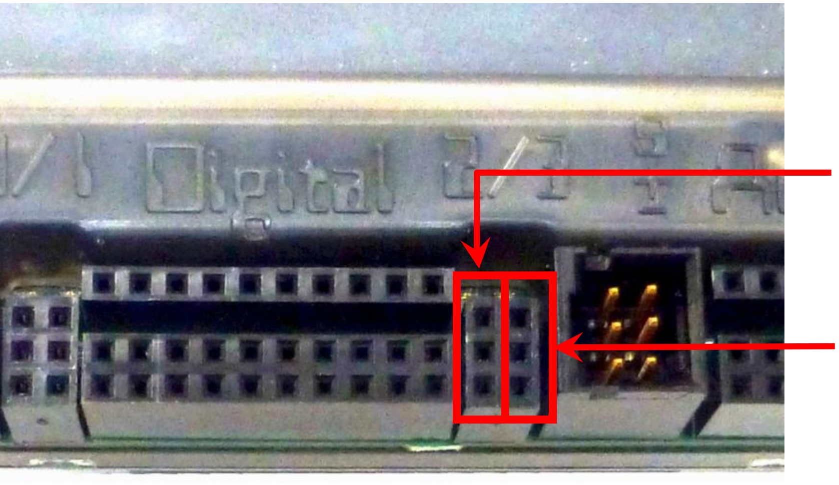
Motor Labels are on the Case

Motor Ports 0, 1, 2, and 3

Plugging in motors

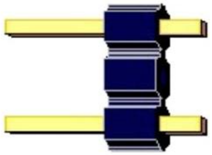
Activity

- **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** (**–**).



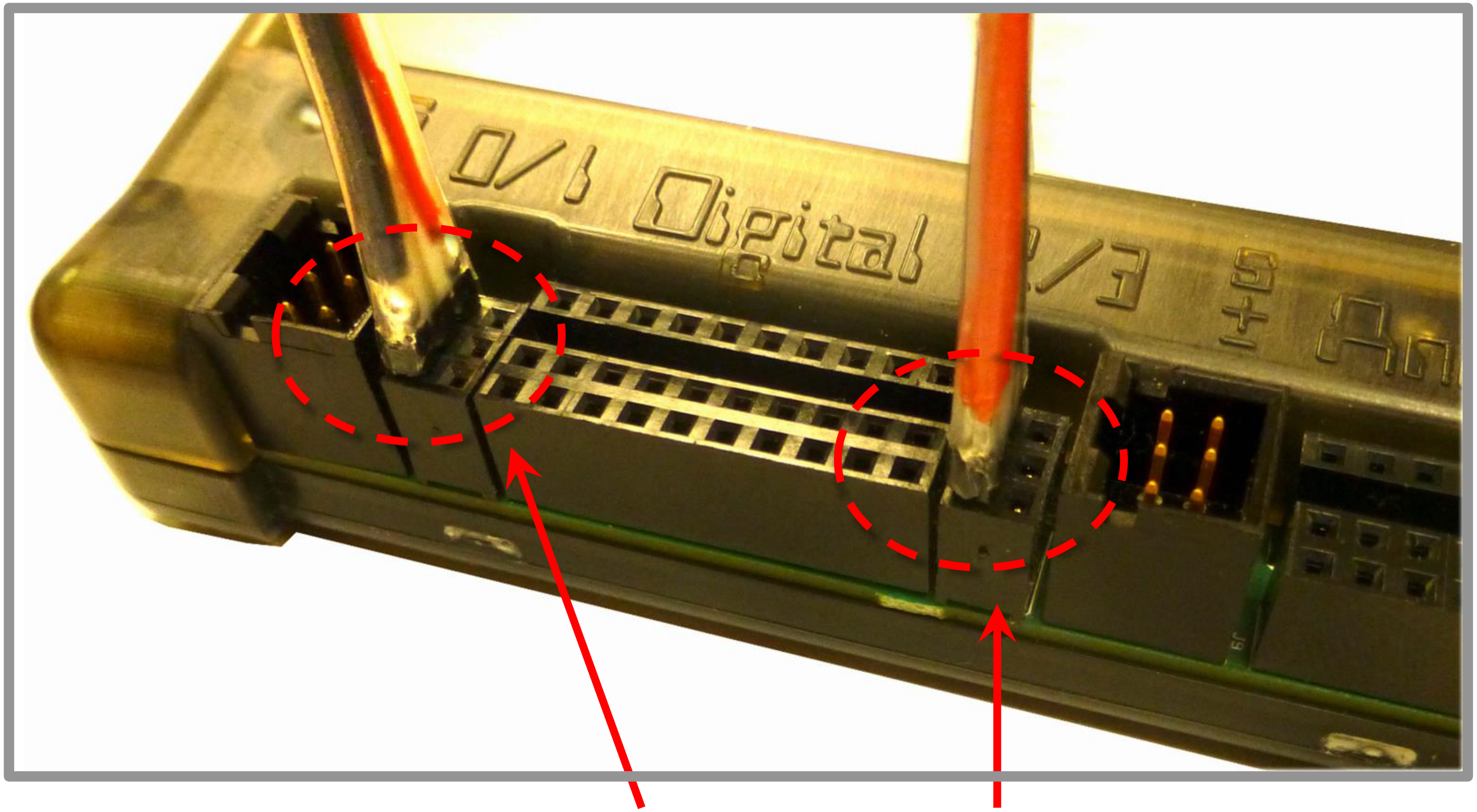
Motor Port #2

Motor Port #3



Drive motors have a two-prong plug.

Plugged in motors



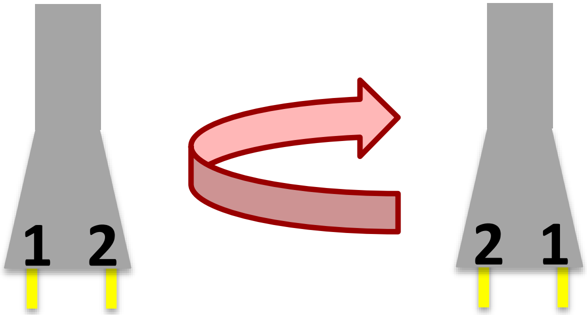
DemoBot Motor Ports 0 (left wheel) and 2 (right wheel)

Activity

Motor direction

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

- **Motors** have red wire and a black wire with a **two-prong plug.**
- There is no left side or right side.
- You can plug these in two different ways:
  - One direction is clockwise, and the other direction is counterclockwise.
  - The red and black wires help determine motor direction.

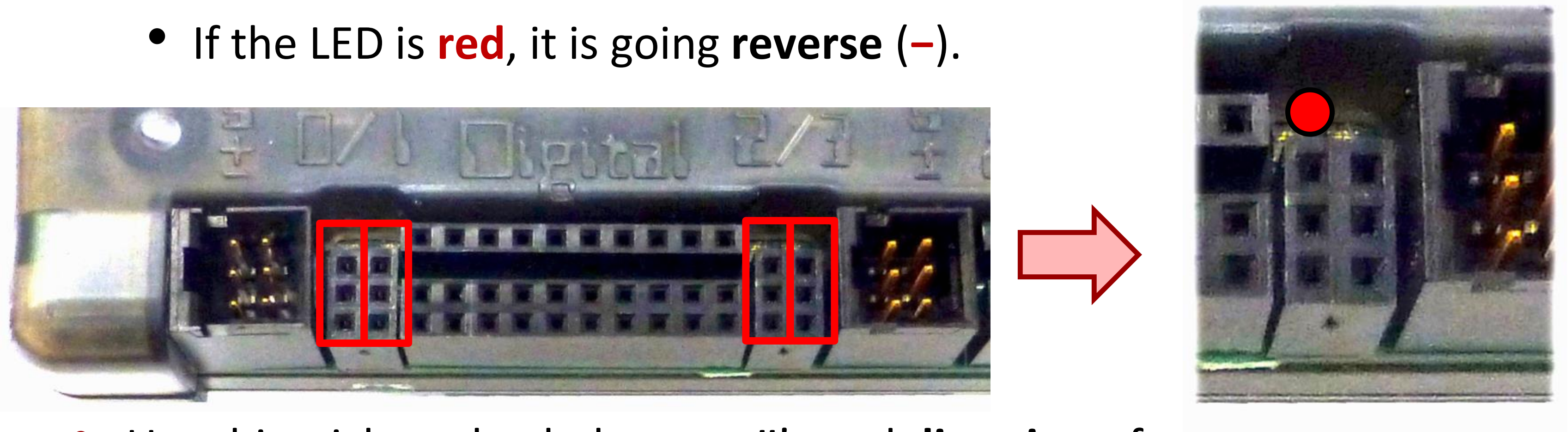


Motor port and direction check

Activity

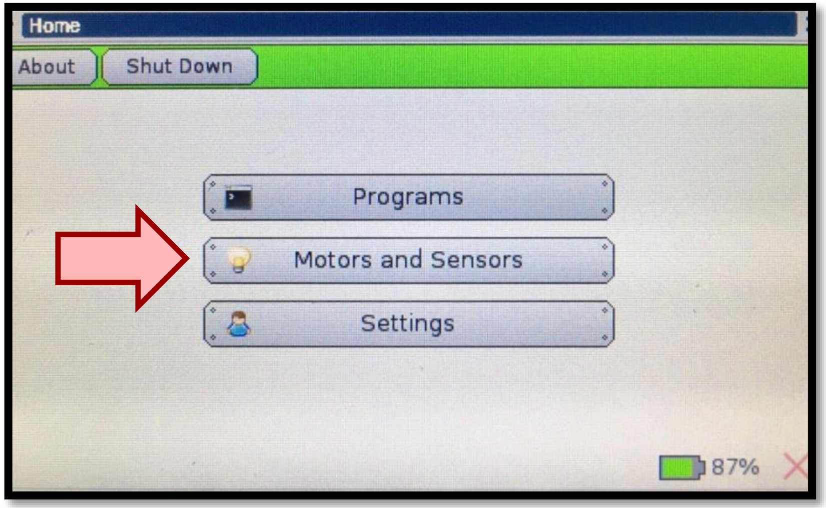
There is an easy way to check this!

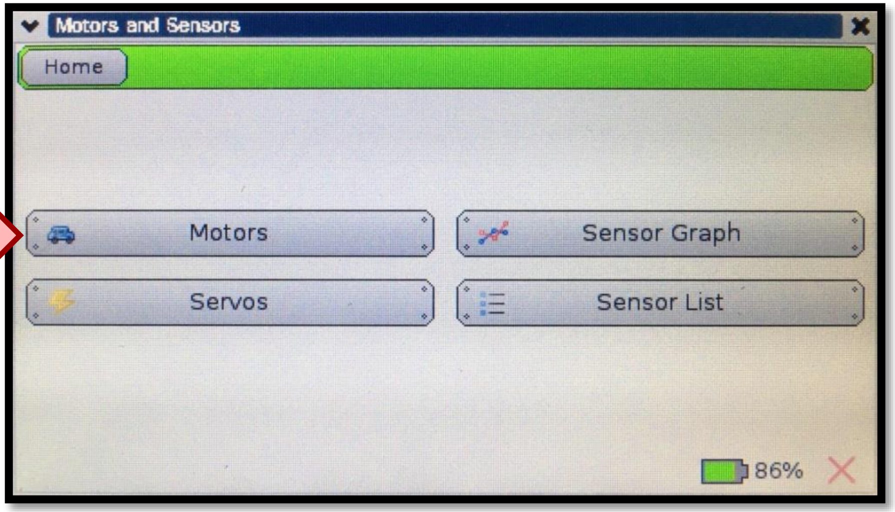
- Manually rotate the tire, and you will see an LED light up by the **motor port** (the **port #** is labeled on the board).
  - If the LED is **green**, it is going **forward** (**+**).
  - If the LED is **red**, it is going **reverse** (**–**).

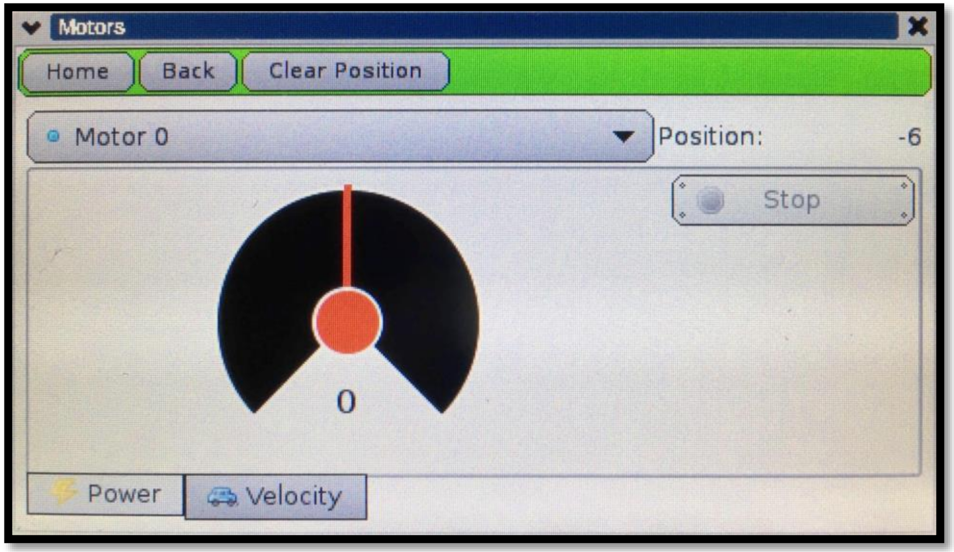


- Use this trick to check the **port #**'s and **direction** of your **motors**.
  - If one is **red** and the other is **green**,
    turn one motor plug 180° and plug it back in.
  - The lights should both be **green** if the robot is moving forward.

Use the Motor Widget







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.
// Select any power between -100% and 100%.

wait_for_milliseconds(# 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.

Using motor and ao

```
int main()
{
  motor(0, 100);
  wait_for_milliseconds(2500);
  ao();
  return 0;
}
```

Resource

KIPR Wallaby functions cheat sheet

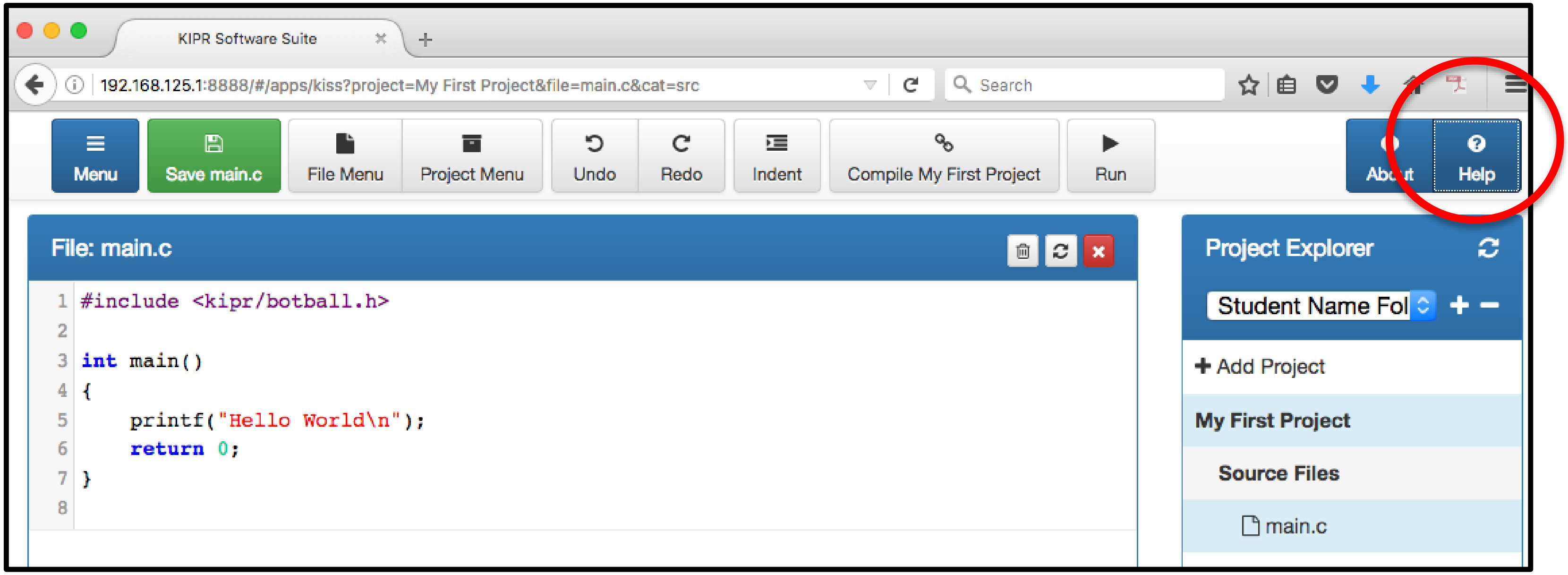
Until you are familiar with the functions that you will be using, use this **cheat sheet** as an easy reference.

Copying and pasting your own code is also very helpful.

```
printf("text\n");                           // Prints the specified text to the screen
wait_for_milliseconds(# milliseconds);      // Waits specified number of milliseconds before next line
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
```

Wallaby Library Documentation

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



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

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

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?

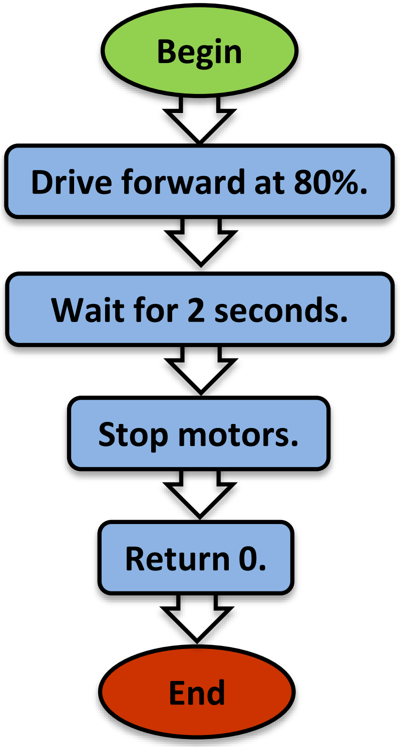
Pseudocode

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

Comments

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

Flowchart


Begin

Drive forward at 80%.

Wait for 2 seconds.

Stop motors.

Return 0.

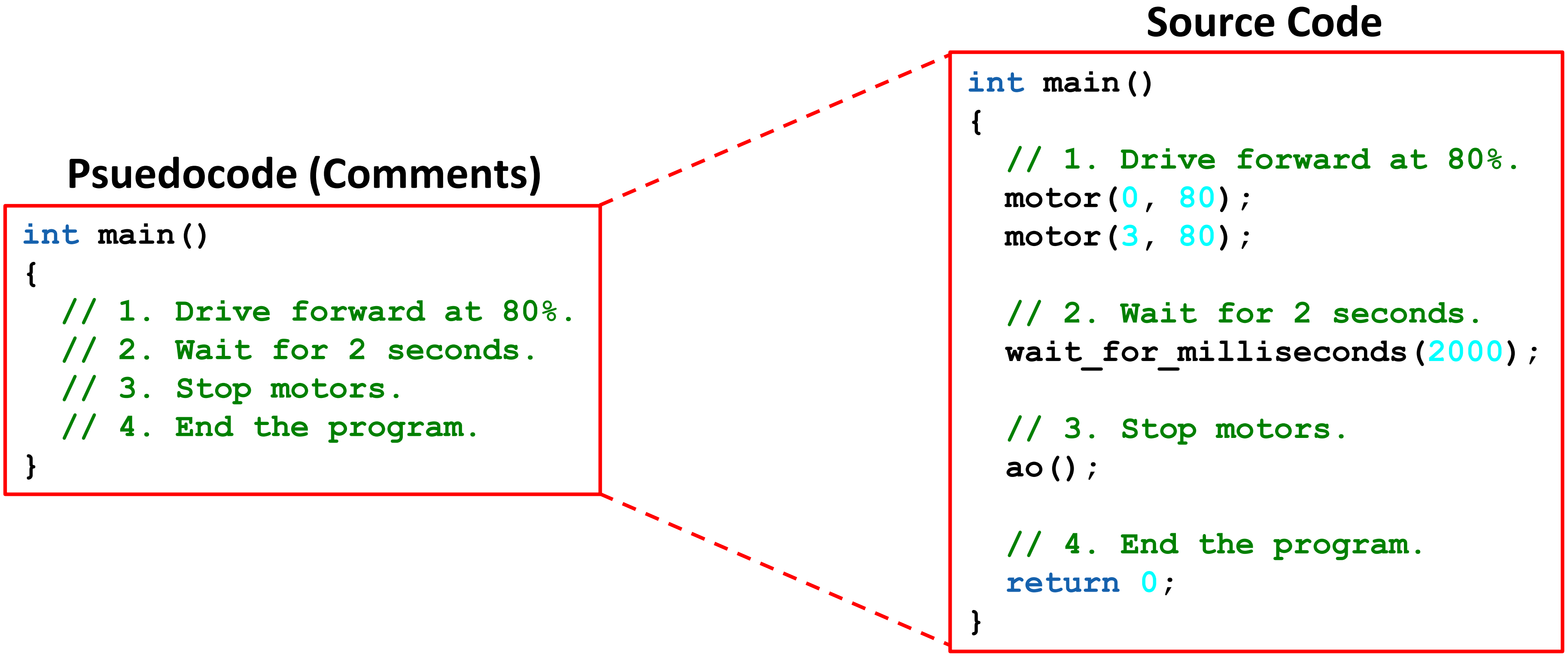
End

Moving the DemoBot

Activity

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

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



Source Code

Psuedocode (Comments)

```
int main()
{
  // 1. Drive forward at 80%.
  // 2. Wait for 2 seconds.
  // 3. Stop motors.
  // 4. End the program.
}
```

```
int main()
{
  // 1. Drive forward at 80%.
  motor(0, 80);
  motor(3, 80);

  // 2. Wait for 2 seconds.
  wait_for_milliseconds(2000);

  // 3. Stop motors.
  ao();

  // 4. End the program.
  return 0;
}
```

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

Moving the DemoBot

Activity

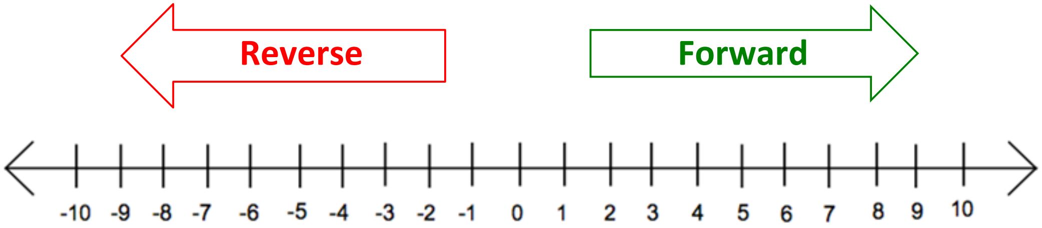
Reflection: What did you notice after you ran the program?

- Did the DemoBot move forward?
  - **Positive** (+) numbers should move the motors in a clockwise direction (**forward**); if not, rotate the motor plug 180° where it plugs into the Wallaby.
  - If your robot moves in a circle, one motor is either not moving (is it plugged in?) or they are moving in opposite directions (rotate the motor plug 180°).
- Did the DemoBot drive straight?
- How could you adjust the code to make the robot drive straight?
- How can you make the robot drive backwards?
- How can you make the robot turn left or right?

Robot driving hints

Resource

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



Reverse

Forward

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

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

And many, many other reasons…

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

Making turns:

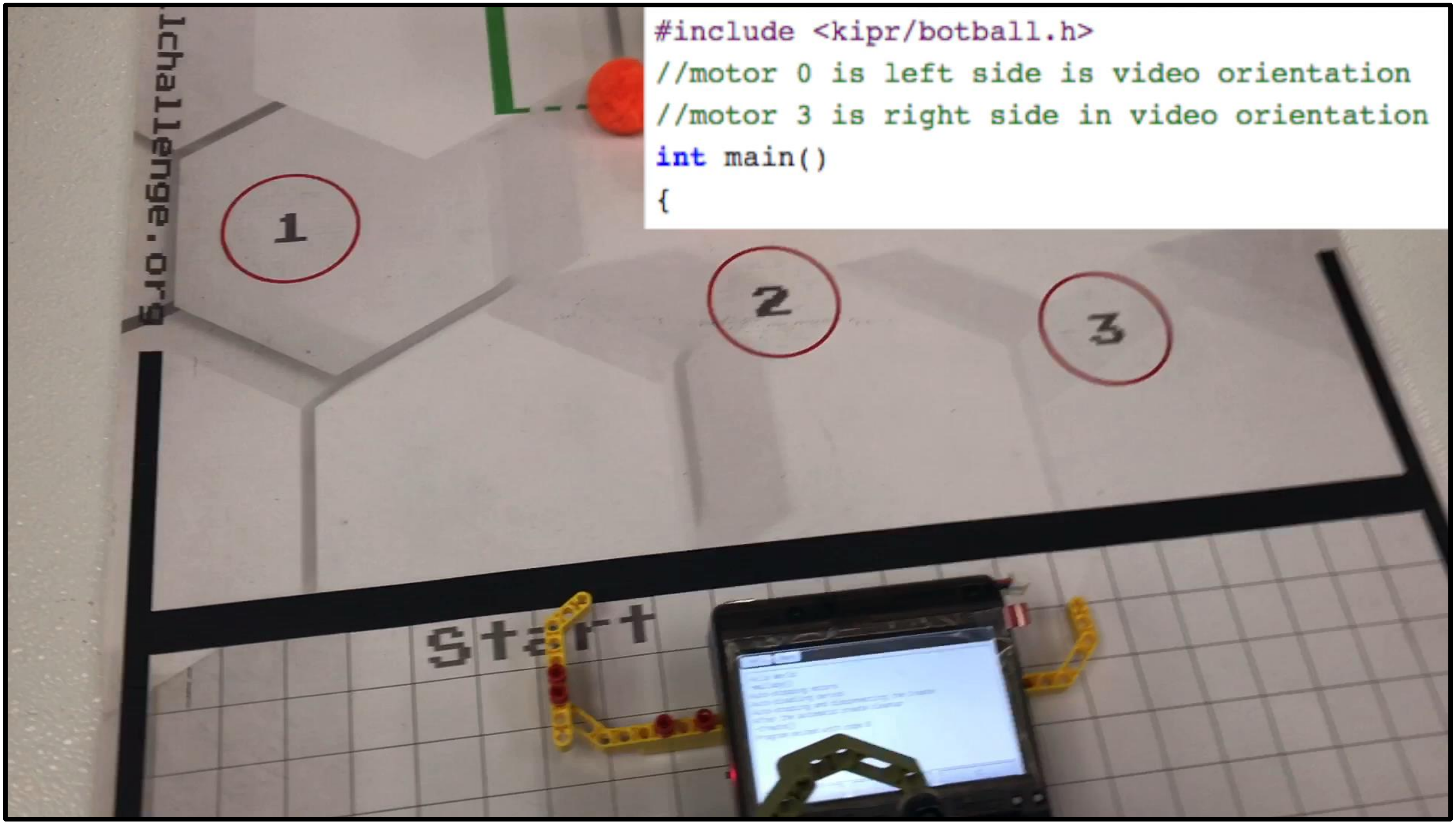
- **Solution:** Have one wheel go faster or slower than the other.
- **Solution:** Have one wheel move while the other one is stopped.
- **Solution:** Have one wheel move forward and the other wheel move in reverse (friction is less of a factor when both wheels are moving).

Activity 1 (connections to the game)

You have a paper copy of this activity in your registration packet.

1. Start with *DemoBot* completely within the starting box on mat A.
2. Recover 4 poms and a yellow foam brick that start out within the nearest garage or around circle 4 (12-14” away on game board or FRP).
3. The poms and yellow brick must come to rest completely within the starting box.

Activity 1 video (possible solution)



```
#include <kipr/botball.h>
//motor 0 is left side is video orientation
//motor 3 is right side in video orientation
int main()
{
```

Moving the DemoBot Servos

Plugging in servos (ports)

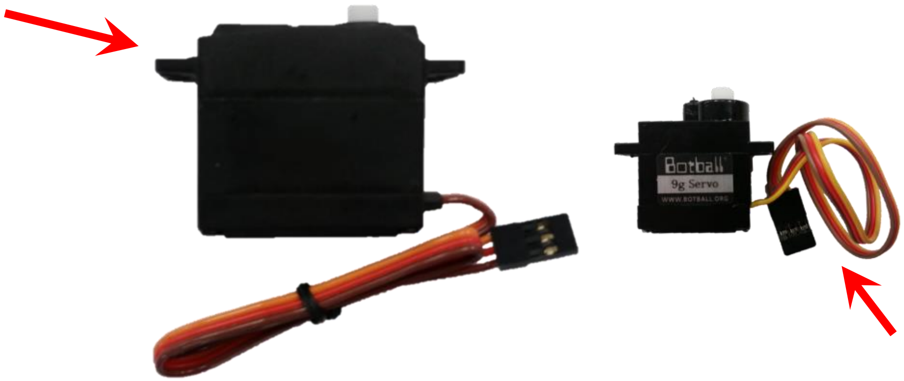
`enable_servos` and `disable_servos` functions

`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**.

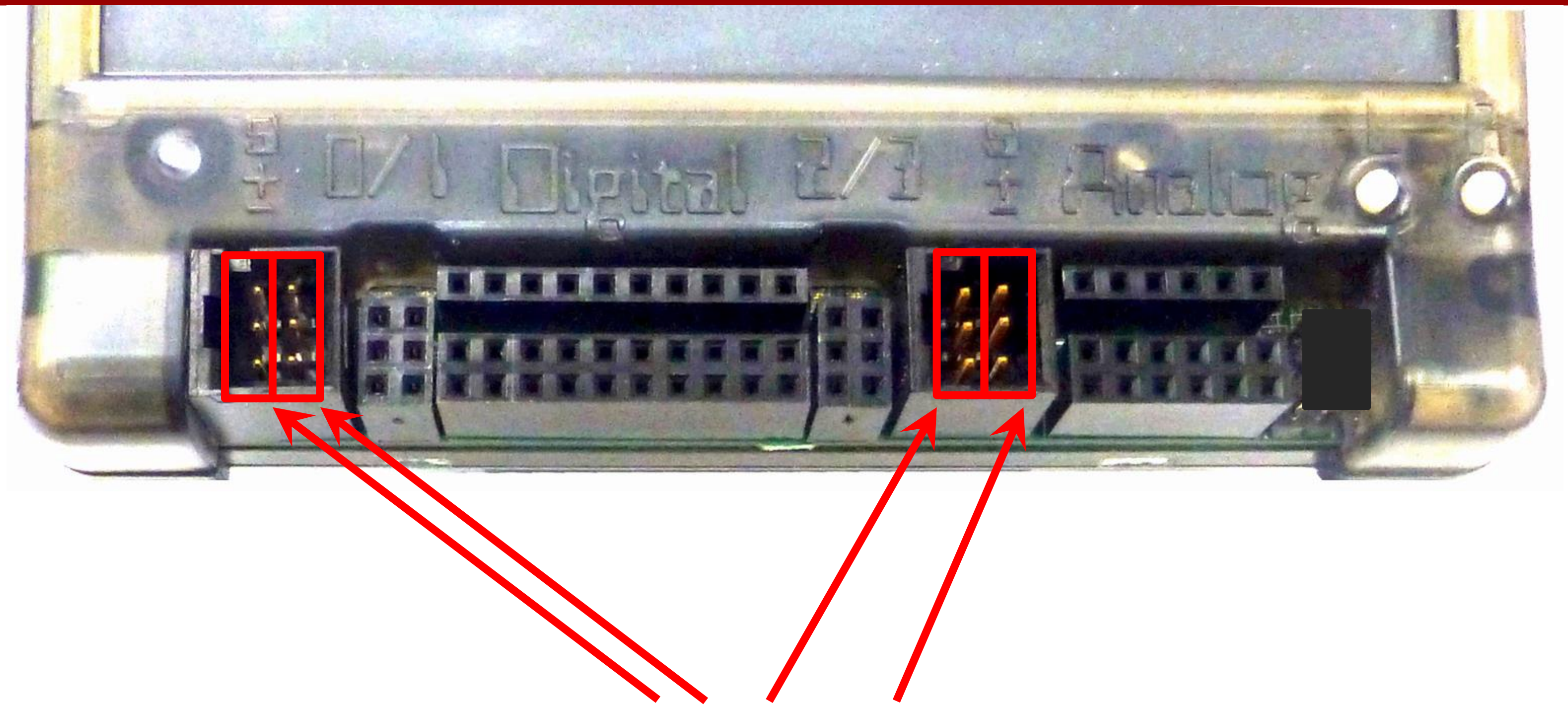
Large servo



Small servo

KIPR Robotics Controller servo ports

Activity

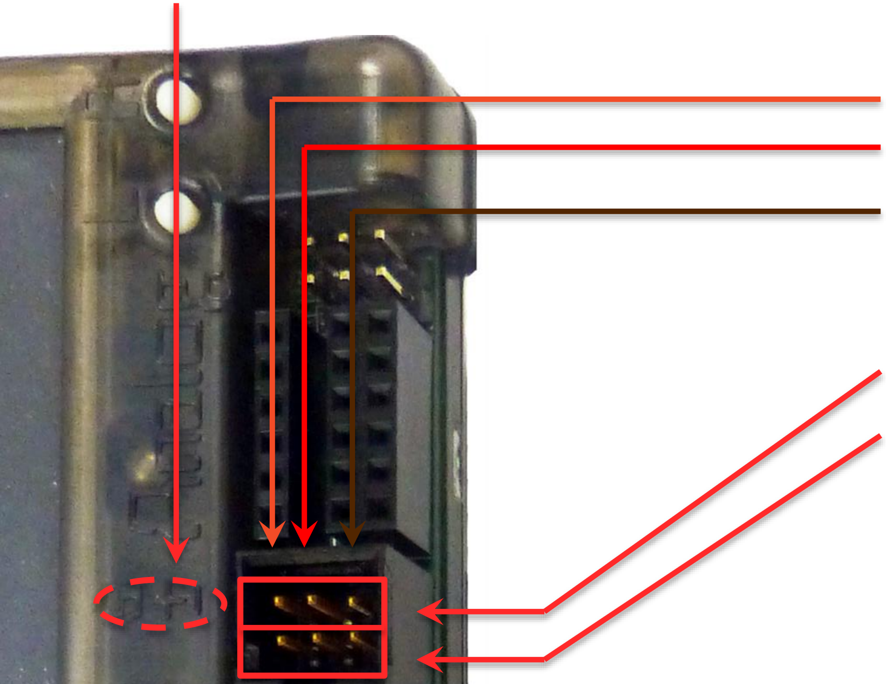


Servo Ports 0, 1, 2, and 3

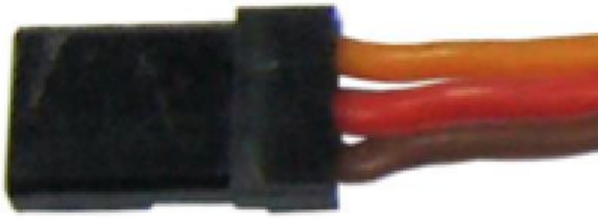
Plugging in Servos

Activity

- 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”)



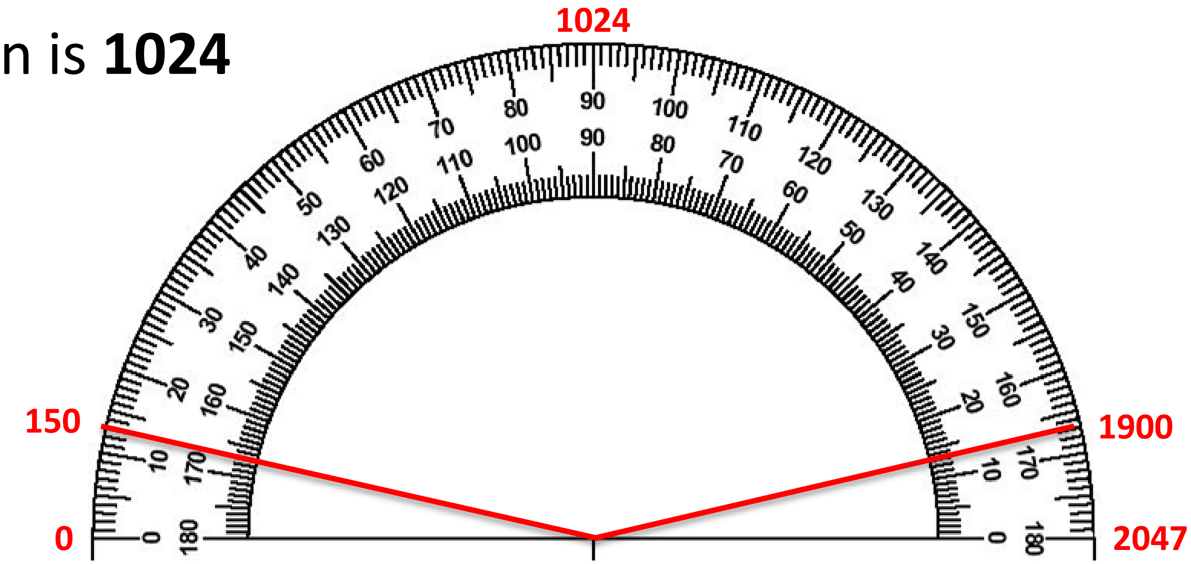
(S) signal wire
(+) power wire
(–) ground wire



Servo Port #3
Servo Port #2

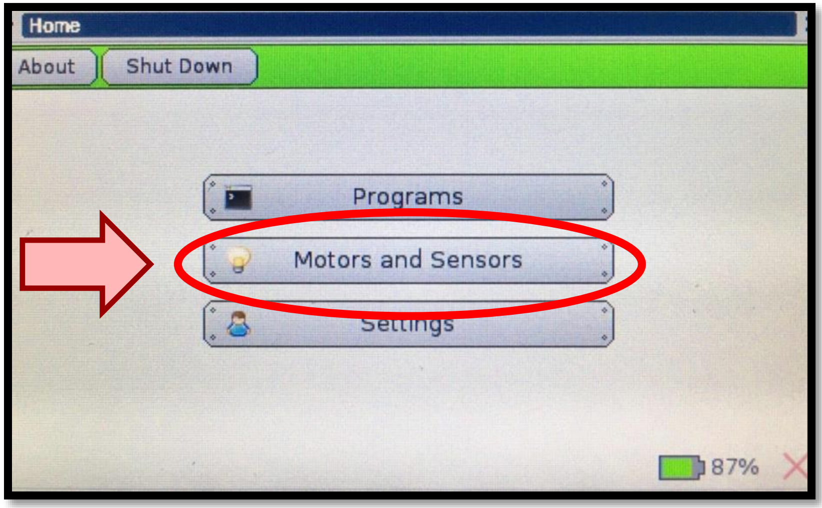
Servo positions

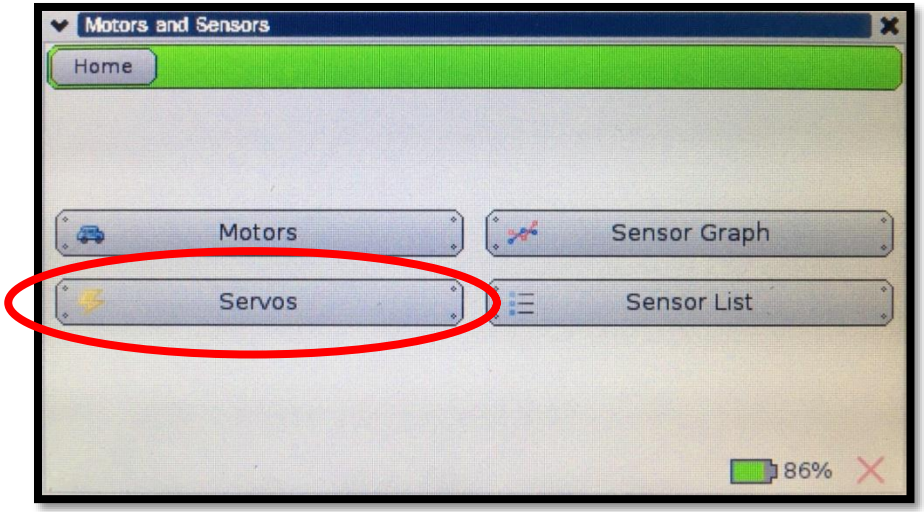
- Think of a servo like a protractor…
  - Angles in the **180° range of motion** (between 0° and 180°) are divided into **2048 servo positions**.
  - These **2048 positions** range from 0 to 2047, but due to internal mechanical hard stop variability you should use **150 to 1900** (**remember:** computer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

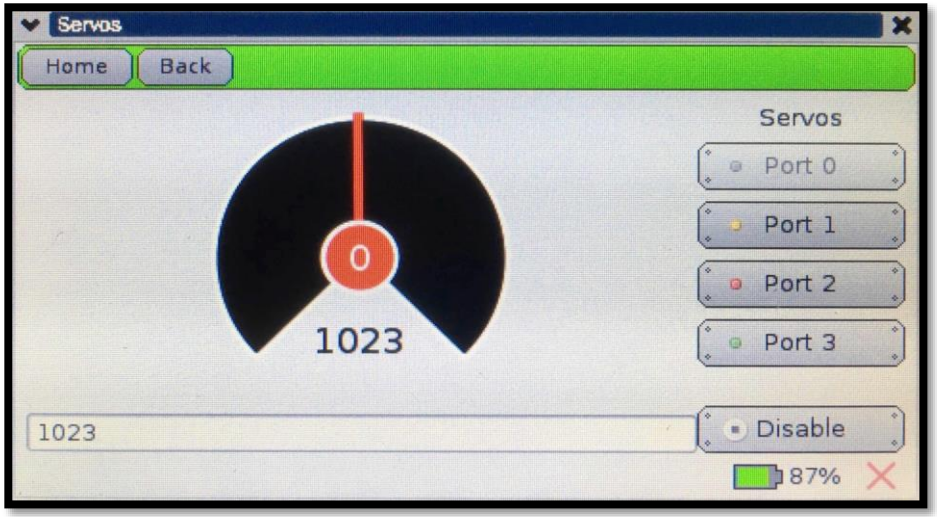


Professional Development Workshop
© 1993 – 2017 KIPR

Use the Servo widget

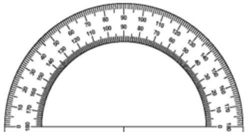


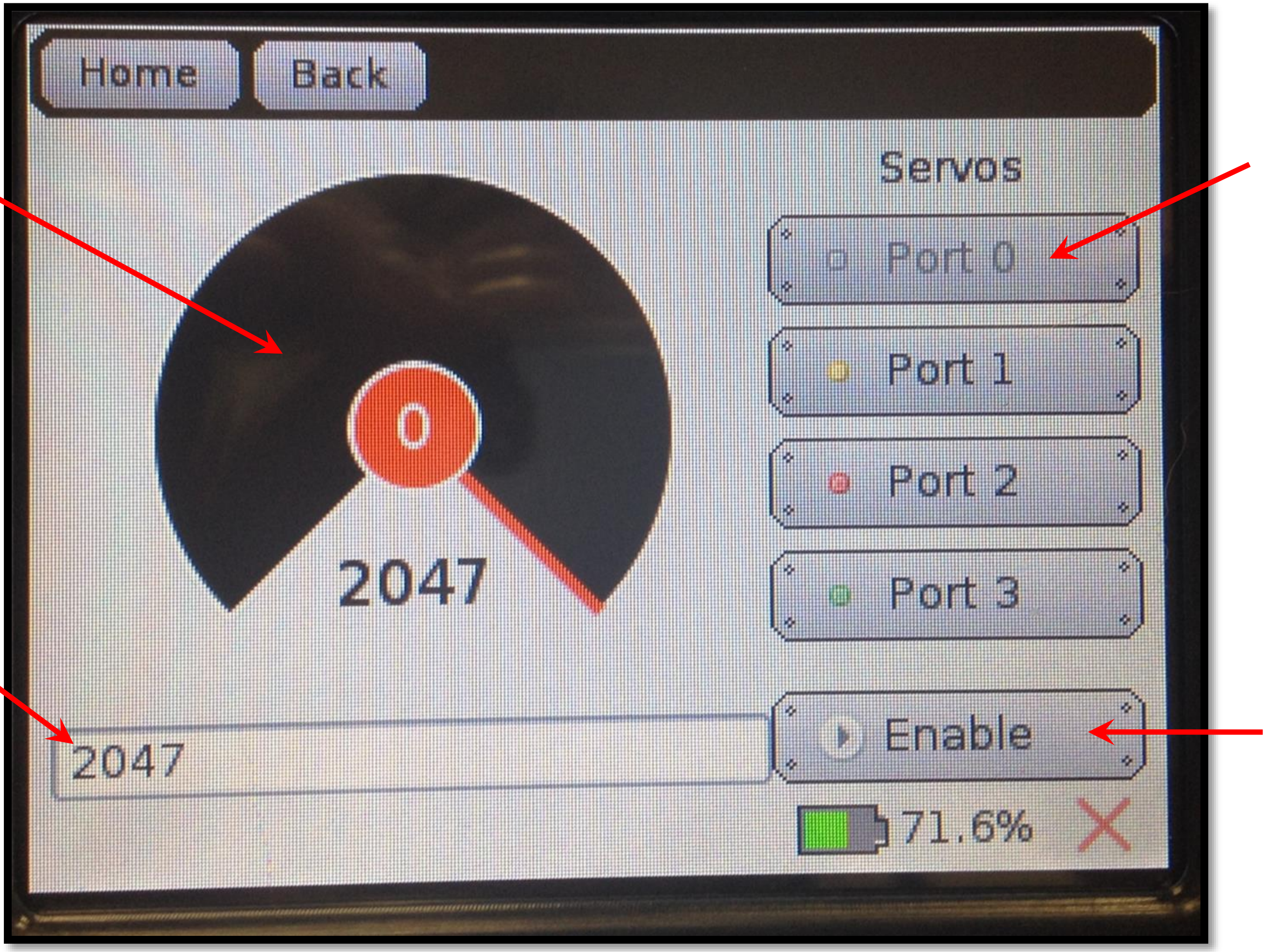




Testing Servos with the Servos screen

Activity





Select the servo port

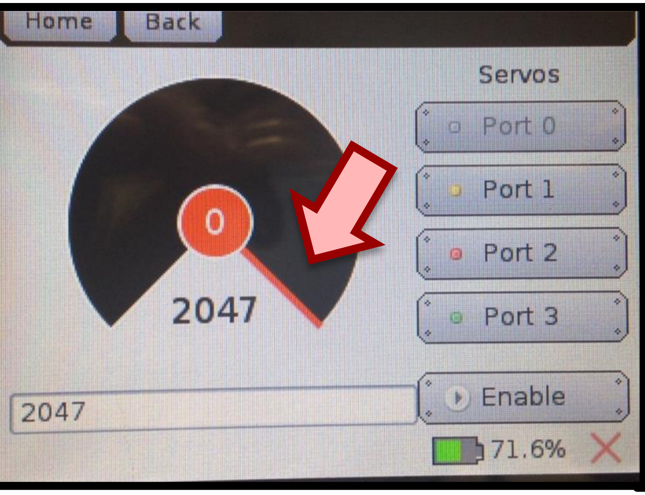
The current servo position

Enable servos

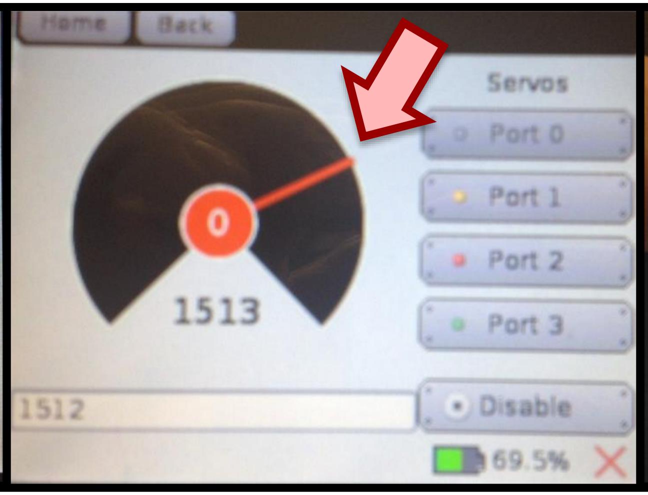
Testing Servos with the Servos screen

Activity

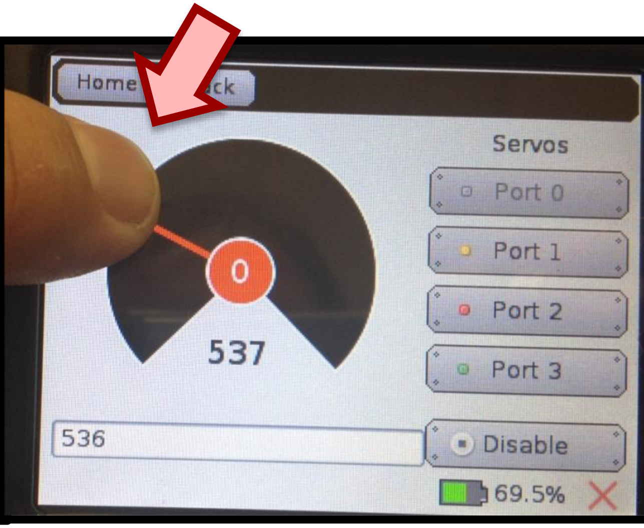
Use your finger
to move the dial.



Servo @ 2047
(maxed out)



Servo @ 1513



Servo @ 537

Do not push a servo beyond its limits
(less than 0 or more than 2047).
This can burn out the servo motor!

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();  // Activate (turn on) all servo ports.

set_servo_position(2, 925);  // Rotate servo on port #2 to position 925.

disable_servos();  // De-activate (turn off) all servo ports.
```

- **Remember:** the useable range of positions is from 150 to 1900.
- The **default position** when servos are enabled is **1024** (**centered**), which means that all **servos will automatically move to this position** when `enable_servos` is called.
- You can "**preset**" a servo position by calling `set_servo_position` ***before*** calling `enable_servos`. This will make the servo move to this position rather than center.

Using servo functions

```
int main()
{
  enable_servos();
  wait_for_milliseconds(1000);
  set_servo_position(2, 925);
  wait_for_milliseconds(1000);
  set_servo_position(2, 675);
  disable_servos();
  return 0;
}
```

Using servo functions

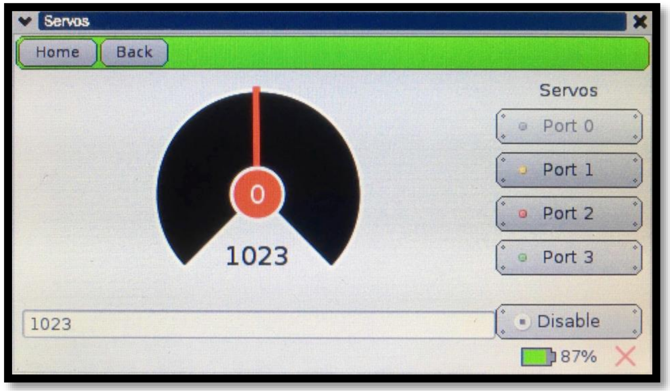
What happens when we set the servo position before `enable_servos`?

```
int main()
{
   set_servo_position(2, 1500);
   enable_servos();
   wait_for_milliseconds(1000);
   set_servo_position(2, 925);
   wait_for_milliseconds(1000);
   set_servo_position(2, 675);
   disable_servos();
   return 0;
}
```

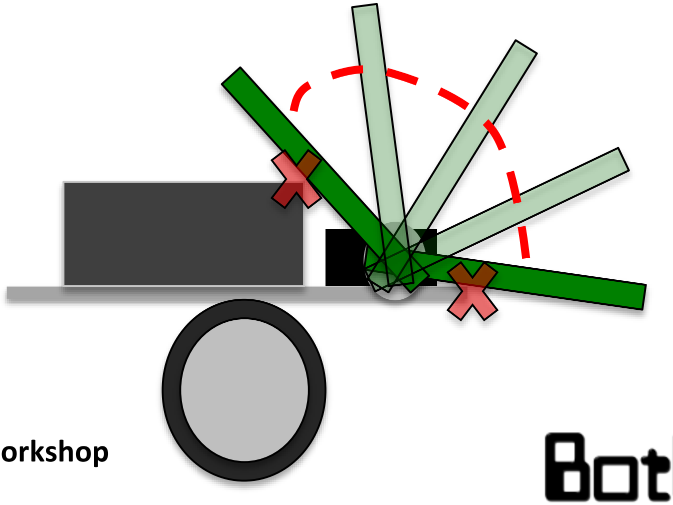
Wave the servo arm

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



1023



Activity

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 your main function

Analysis: What is the program supposed to do?

Pseudocode

```
1.	Enable servos.
2.	Move servo to YOUR limit.
3.	Wait for 3 seconds.
4.	Move servo to YOUR other limit.
5.	Wait for 3 seconds.
6.	Disable servos.
7.	End the program.
```

Comments

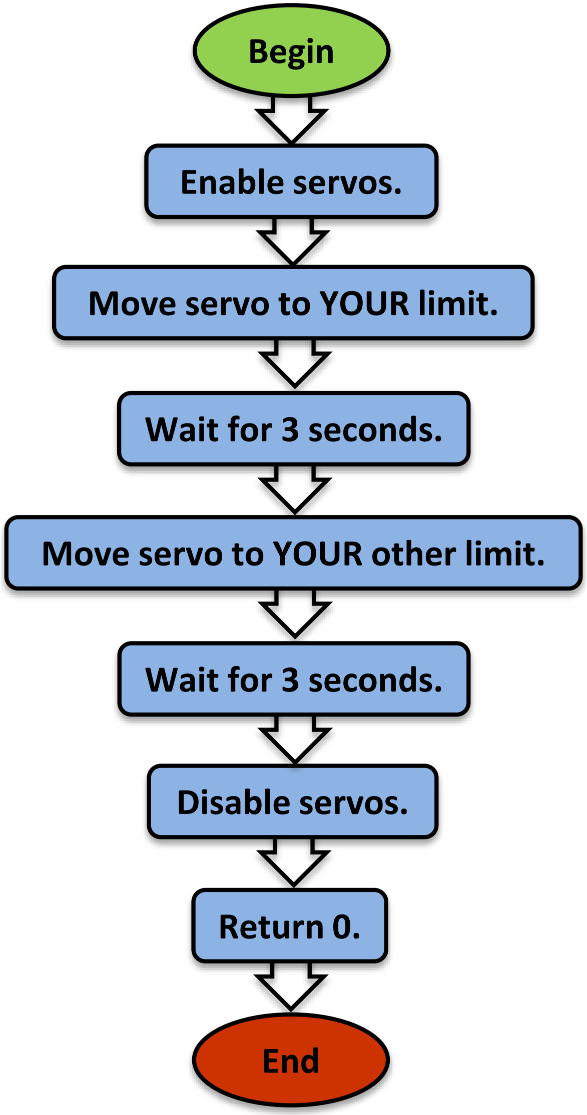
```
// 1. Enable servos.
// 2. Move servo to YOUR limit.
// 3. Wait for 3 seconds.
// 4. Move servo to YOUR other limit.
// 5. Wait for 3 seconds.
// 6. Disable servos.
// 6. End the program.
```

Wave the servo arm

Activity

Analysis:

Flowchart



Begin

Enable servos.

Move servo to YOUR limit.

Wait for 3 seconds.

Move servo to YOUR other limit.

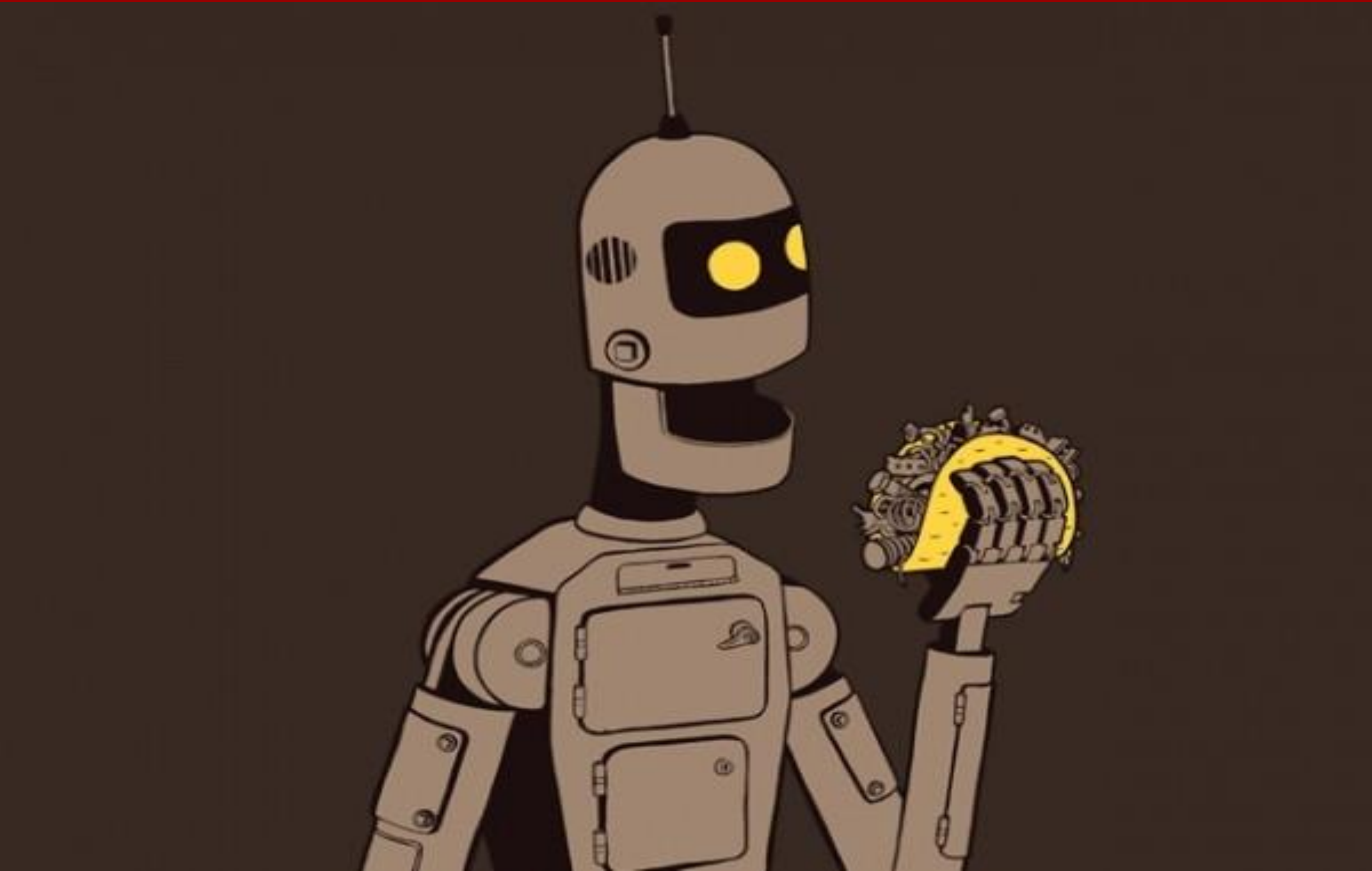
Wait for 3 seconds.

Disable servos.

Return 0.

End

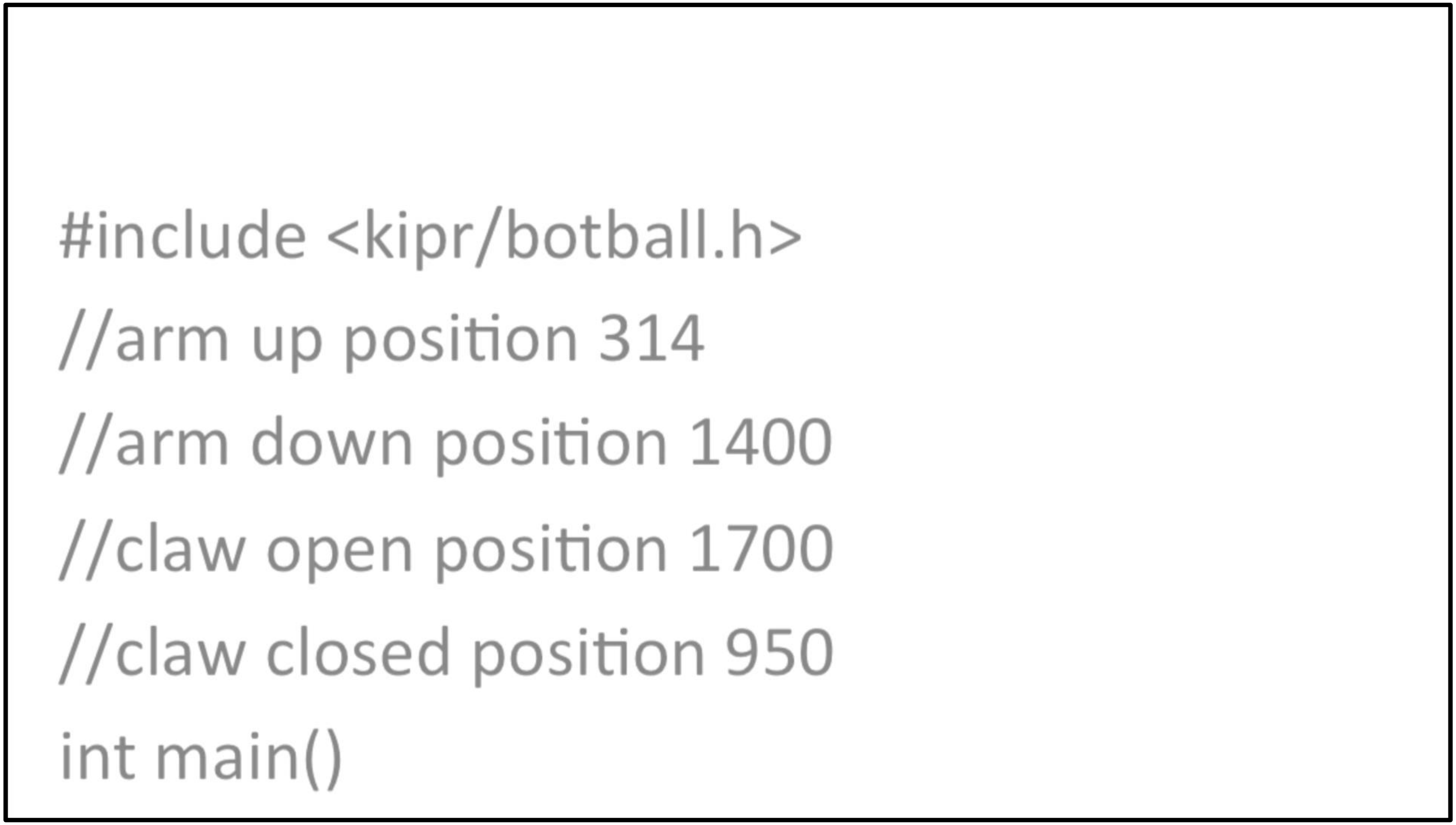
Lunch!



Activity 2 (connections to the game)

1. Start with your DemoBot at least partially within the starting box. See extension for more practical application.
2. Using a servo controlled claw, recover a blue foam block from circle 9 on mat A.
3. The blue foam block should be elevated off the surface while wheel movement occurs and should be placed on the playing surface inside of the starting box.

Activity 2 Video (possible solution)

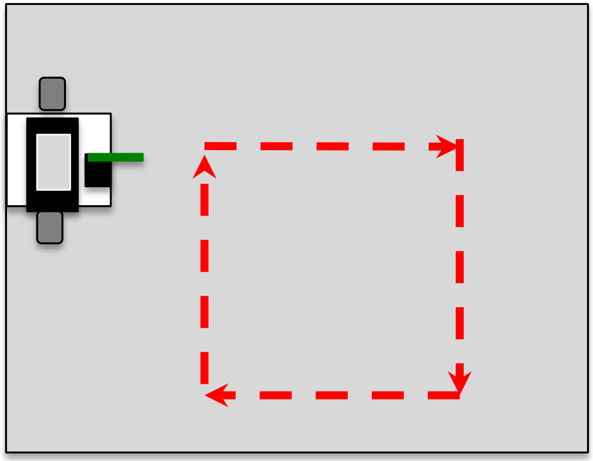


```
#include <kipr/botball.h>
//arm up position 314
//arm down position 1400
//claw open position 1700
//claw closed position 950
int main()
```

Draw a square

Description: Write a program for the KIPR Wallaby that drives the DemoBot along a path in the shape of a square.

- Start with having the robot make a 90° turn.
- Then add in forward movements to have the robot drive along a square path. Remember the direction your robot is taking.



Draw a square

Analysis: What is the program supposed to do?

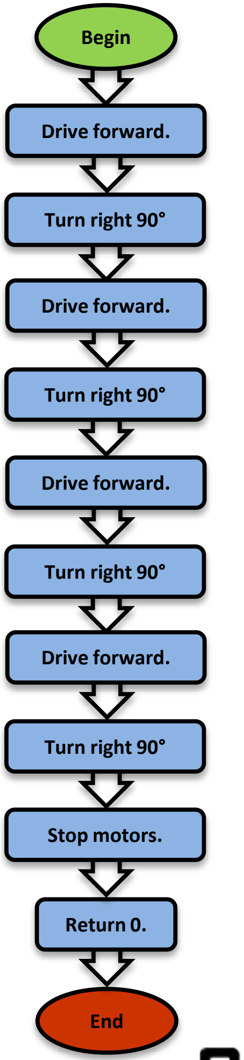
Pseudocode

```
1.  Drive forward.
2.  Turn right 90°.
3.  Drive forward.
4.  Turn right 90°.
5.  Drive forward.
6.  Turn right 90°.
7.  Drive forward.
8.  Turn right 90°.
9.  Stop motors.
10. End the program.
```

Comments

```
// 1. Drive forward.
// 2. Turn right 90-degrees.
// 3. Drive forward.
// 4. Turn right 90-degrees.
// 5. Drive forward.
// 6. Turn right 90-degrees.
// 7. Drive forward.
// 8. Turn right 90-degrees.
// 9. Stop motors.
// 10. End the program.
```

Flowchart



Begin

Drive forward.

Turn right 90°

Drive forward.

Turn right 90°

Drive forward.

Turn right 90°

Drive forward.

Turn right 90°

Stop motors.

Return 0.

End

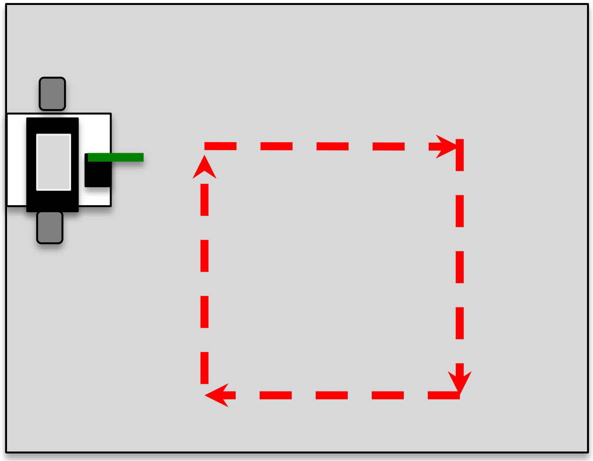
Activity

Draw a square

Solution:

Here is some code that uses the **motor()** and **wait_for_milliseconds()** functions to drive the robot in a square.

Note: this is just *one of many* solutions.



```
int main()
{
  // 1. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 2. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // 3. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 4. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // 5. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 6. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // 7. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 8. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  ao();       // 9. Stop motors.
  return 0;  // 10. End the program.
} // end main
```

Fun with Functions

Writing your own functions

Function prototypes, definitions, and calls

Activity

Draw a square

Reflection:

Notice there are many repeated steps. For example:

```
// Drive forward.
motor(0, 100);
motor(0, 100);
wait_for_milliseconds(4000);
```

… is **repeated 4 times** in this program!

- Also, Turn right 90-degrees.

You will quickly learn to use copy-and-paste over and over again, but there is a better and easier way…

Learning to write your own functions allows you to reuse code easily!

```
int main()
{
```

Drive forward.

```
    // 1. Drive forward.
    motor(0, 100);
    motor(3, 100);
    wait_for_milliseconds(4000);
```

Turn right.

```
    // 2. Turn right 90-degrees.
    motor(0,  100);
    motor(3, -100);
    wait_for_milliseconds(1500);
```

Drive forward.

```
    // 3. Drive forward.
    motor(0, 100);
    motor(3, 100);
    wait_for_milliseconds(4000);
```

Turn right.

```
    // 4. Turn right 90-degrees.
    motor(0,  100);
    motor(3, -100);
    wait_for_milliseconds(1500);
```

Drive forward.

```
    // 5. Drive forward.
    motor(0, 100);
    motor(3, 100);
    wait_for_milliseconds(4000);
```

Turn right.

```
    // 6. Turn right 90-degrees.
    motor(0,  100);
    motor(3, -100);
    wait_for_milliseconds(1500);
```

Drive forward.

```
    // 7. Drive forward.
    motor(0, 100);
    motor(3, 100);
    wait_for_milliseconds(4000);
```

Turn right.

```
    // 8. Turn right 90-degrees.
    motor(0,  100);
    motor(3, -100);
    wait_for_milliseconds(1500);

    ao();      // 9. Stop motors.
    return 0;  // 10. End the program.
} // end main
```

Writing your own functions

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

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

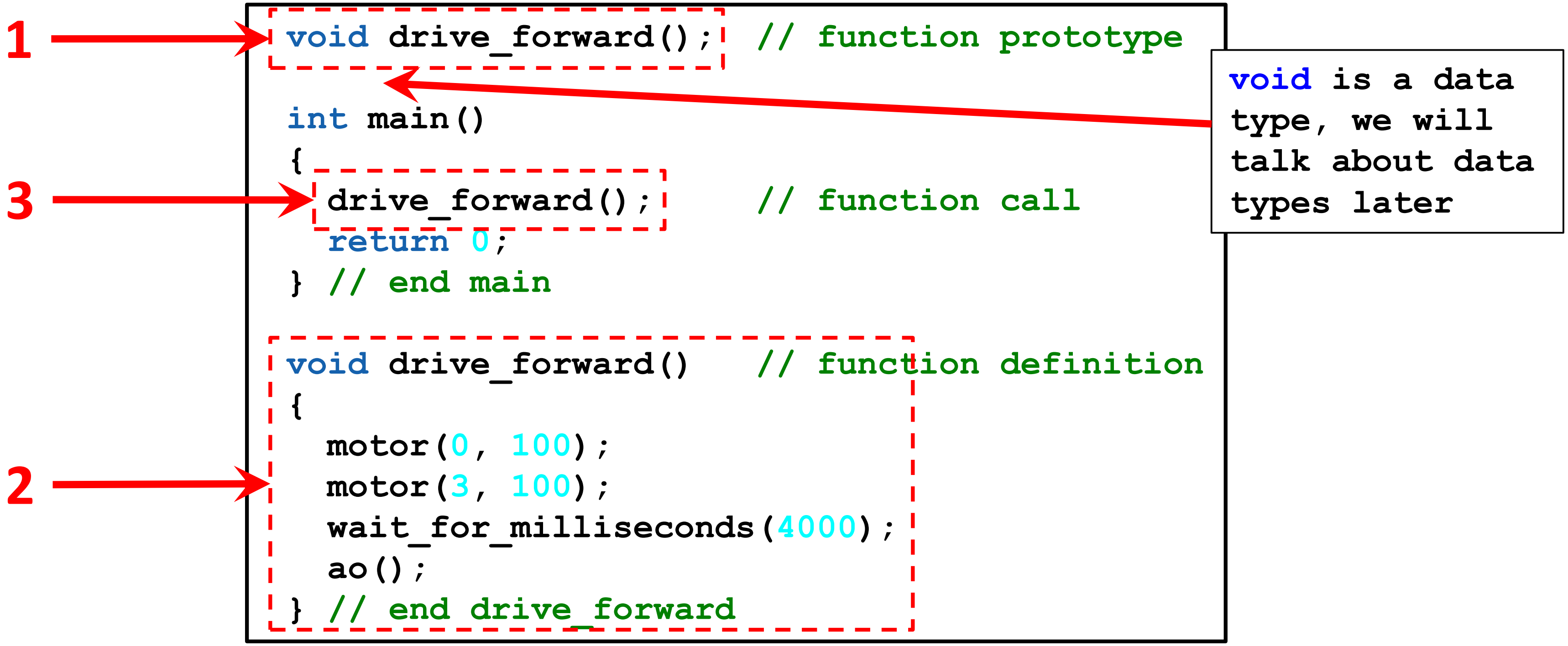
We made these up… and that's the point!

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

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

Writing your own 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 your program



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

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

void drive_forward()   // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

void is a data type, we will talk about data types later

Writing your own functions

Function prototypes go above `main`.

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

Function calls go inside `main` (or inside other functions).

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

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

Function definitions go below `main`.

```
void drive_forward()    // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

Writing your own functions

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

prototype

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



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

definition

```
void drive_forward()    // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

Notice: no semicolon!
(Why not?)

Writing your own functions

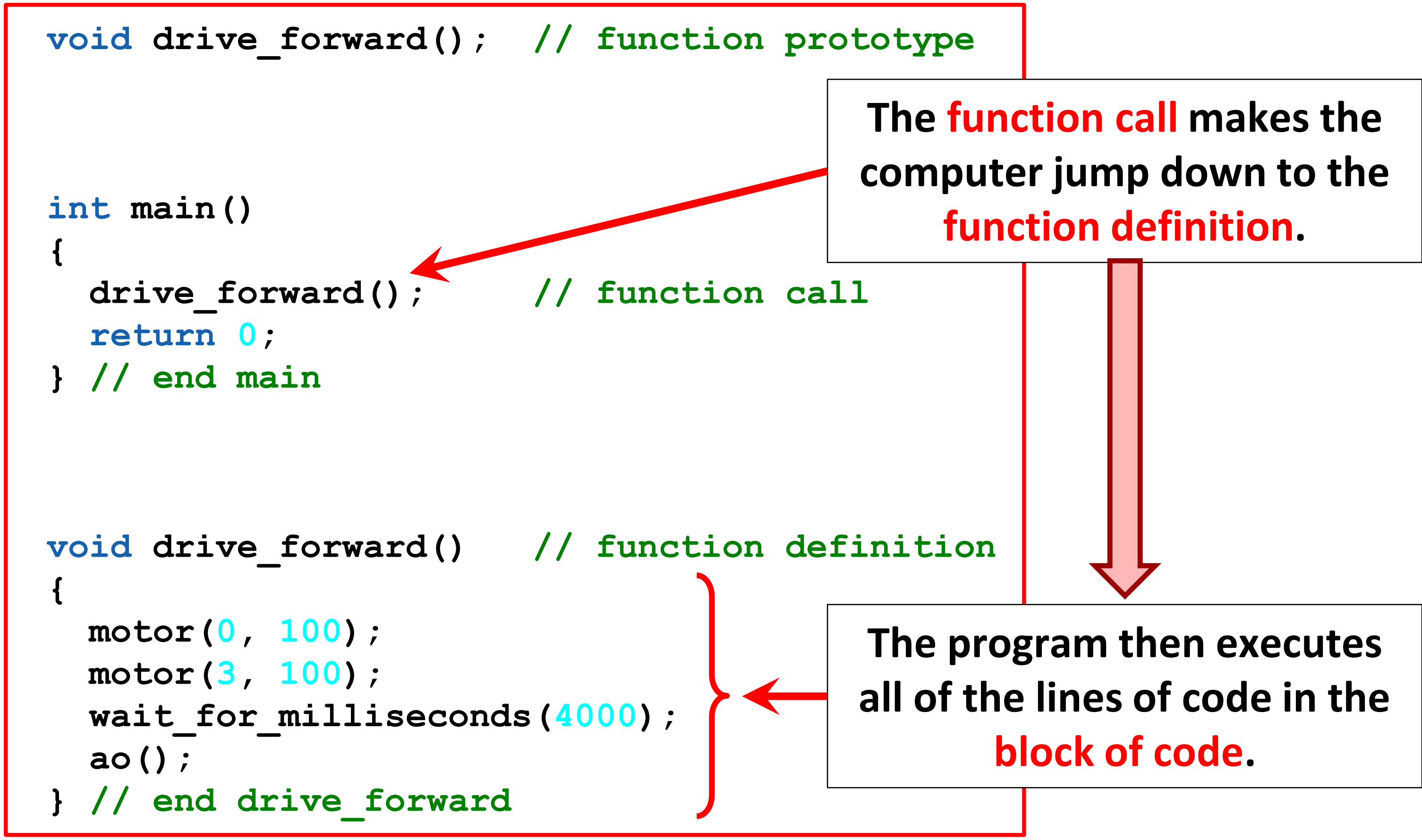
```
void drive_forward();   // function prototype


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


void drive_forward()    // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

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

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

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

Writing your own functions



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



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



void drive_forward()    // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

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**.

Writing your own functions

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



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


void drive_forward()    // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

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.

This is the *end* } of the function definition.

Writing your own functions

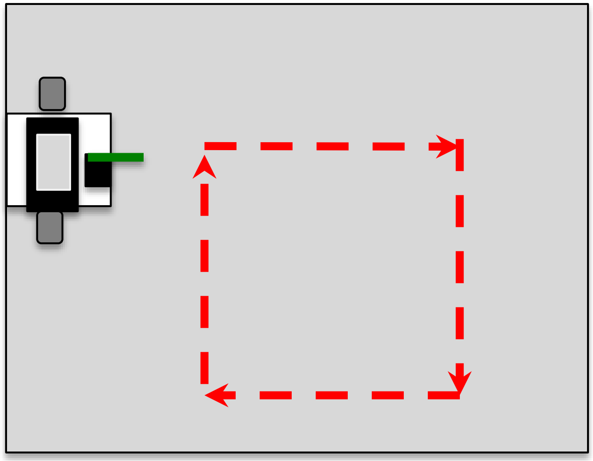
```
// function prototypes
void drive_forward();
void turn_right();

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

void drive_forward()  // drive_forward function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward

void turn_right()     // turn_right function definition
{
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);
  ao();
} // end turn_right
```

Draw a square using functions

Activity

Description: Write a program for the KIPR Wallaby that drives the DemoBot along a path in the shape of a square *using functions*.

- **Hint:** modify your old square-drawing program to use your own functions.
- Break the task down into common subtasks → these become your functions!



Activity

Code without your functions

```
int main()
{
  // 1. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 2. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // 3. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 4. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // 5. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 6. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // 7. Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // 8. Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  ao();       // 9. Stop motors.
  return 0;  // 10. End the program.
} // end main
```

main is shorter and easier to read.

Code with your functions

```
// Function prototype for
// drive_forward_and_turn_right.
void drive_forward_and_turn_right();


// Function definition for main.
int main()
{
  // Four function calls for
  // drive_forward_and_turn_right.
  drive_forward_and_turn_right();
  drive_forward_and_turn_right();
  drive_forward_and_turn_right();
  drive_forward_and_turn_right();
  return 0;
} // end main


// Function definition for
// drive_forward_and_turn_right.
void drive_forward_and_turn_right()
{
  // Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // Stop motors.
  ao();
} // end drive_forward_and_turn_right
```

Draw a square

Activity

Reflection:

1. It makes the main function easier to read and understand, and spotting mistakes is much easier.
2. You only have to change a value **one time** in the **function definition** for it to affect the entire program.
   - For example, to draw a smaller square, simply change the **`wait_for_milliseconds()`** value in your **`drive_forward_and_turn()`** function definition from `4000` to `2000`.

```c
// Function prototype for
// drive_forward_and_turn_right.
void drive_forward_and_turn_right();

// Function definition for main.
int main()
{
  // Four function calls for
  // drive_forward_and_turn_right.
  drive_forward_and_turn_right();
  drive_forward_and_turn_right();
  drive_forward_and_turn_right();
  drive_forward_and_turn_right();
  return 0;
} // end main

// Function definition for
// drive_forward_and_turn_right.
void drive_forward_and_turn_right()
{
  // Drive forward.
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  // Turn right 90-degrees.
  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);

  // Stop motors.
  ao();
} // end drive_forward_and_turn_right
```

Advanced waving the servo arm

Activity

Create a function to wave your servo arm.

Comments

```
void wave()
{
  // 1. Enable servos.
  // 2. Move servo to YOUR limit.
  // 3. Wait for 3 seconds.
  // 4. Move servo to YOUR other limit.
  // 5. Wait for 3 seconds.
  // 6. Disable servos.
}
```

Move the Servo using functions

Activity

Solution:

Comments

```
void wave()
{
  // 1. Enable servos.
  // 2. Move servo to YOUR limit.
  // 3. Wait for 3 seconds.
  // 4. Move servo to YOUR other limit.
  // 5. Wait for 3 seconds.
  // 6. Disable servos.
}
```

Source Code

```
void wave();

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

void wave()
{
  // 1. Enable servos.
  enable_servos();
  // 2. Move servo to YOUR limit.
  set_servo_position(0, 1400);

  // 3. Wait for 3 seconds.
  wait_for_milliseconds(3000);

  // 4. Move servo to YOUR other limit.
  set_servo_position(0, 1024);
  // 5. Wait for 3 seconds.
  wait_for_milliseconds(3000);
  // 6. Disable servos.
  disable_servos();
}
```

Use YOUR servo limits!

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

Variables and Functions with Arguments

Data types

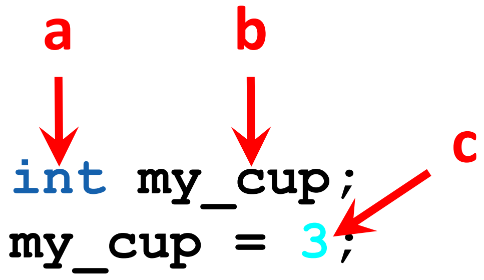
Creating and setting a variable

Variable arithmetic

Functions with arguments and return values

Variables

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



a

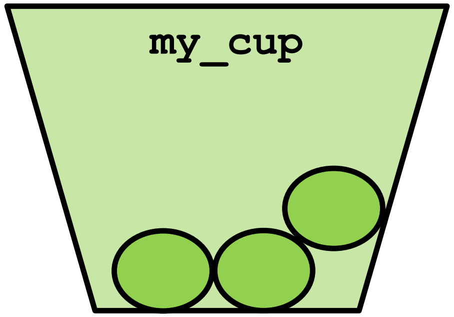
b

c

```
int my_cup;
my_cup = 3;
```

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

- Think of a **variable** like a cup with your name on it…



Variable names

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

- Variable names can be *almost* anything you would like.
- Variable names can contain **letters**, **numbers**, and **underscores** (“_”).
- Variable names **cannot** begin with a **number**.

An Example:

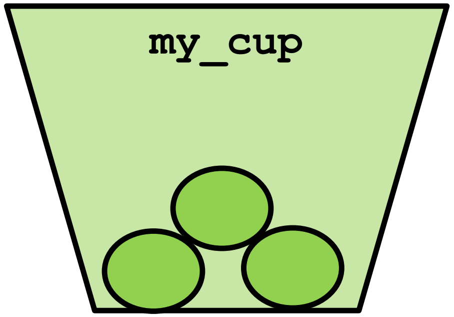
```
// Variable to keep a count of events.
int my_variable;        // variable declaration
my_variable = 0;        // variable “initialization”
```

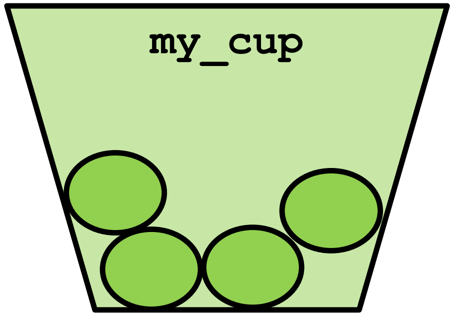
Variables

You can set the value to any integer you choose.

```
int my_cup;
my_cup = 3;
```

```
int my_cup;
my_cup = 4;
```



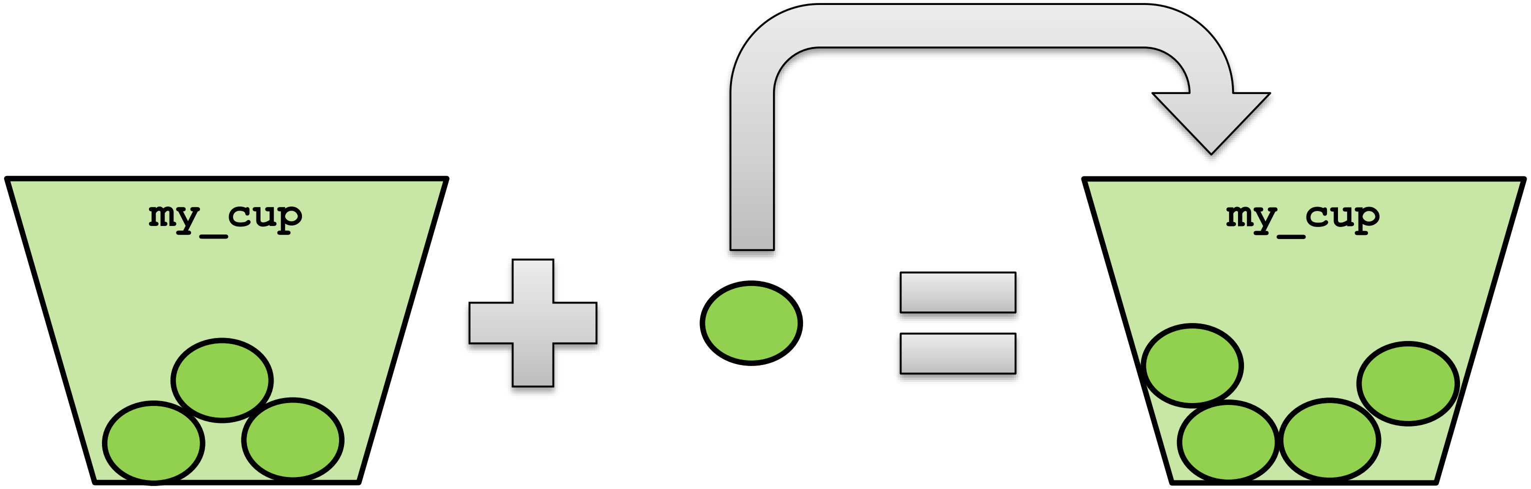


Variables

- So how could this be useful?

- What if we wanted to add balls to the cup

```
int my_cup;
my_cup = 3;
my_cup = my_cup + 1; // now my_cup is equal to 4
```



Remember This?

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



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


void drive_forward()    // function definition
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);
  ao();
}
```

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

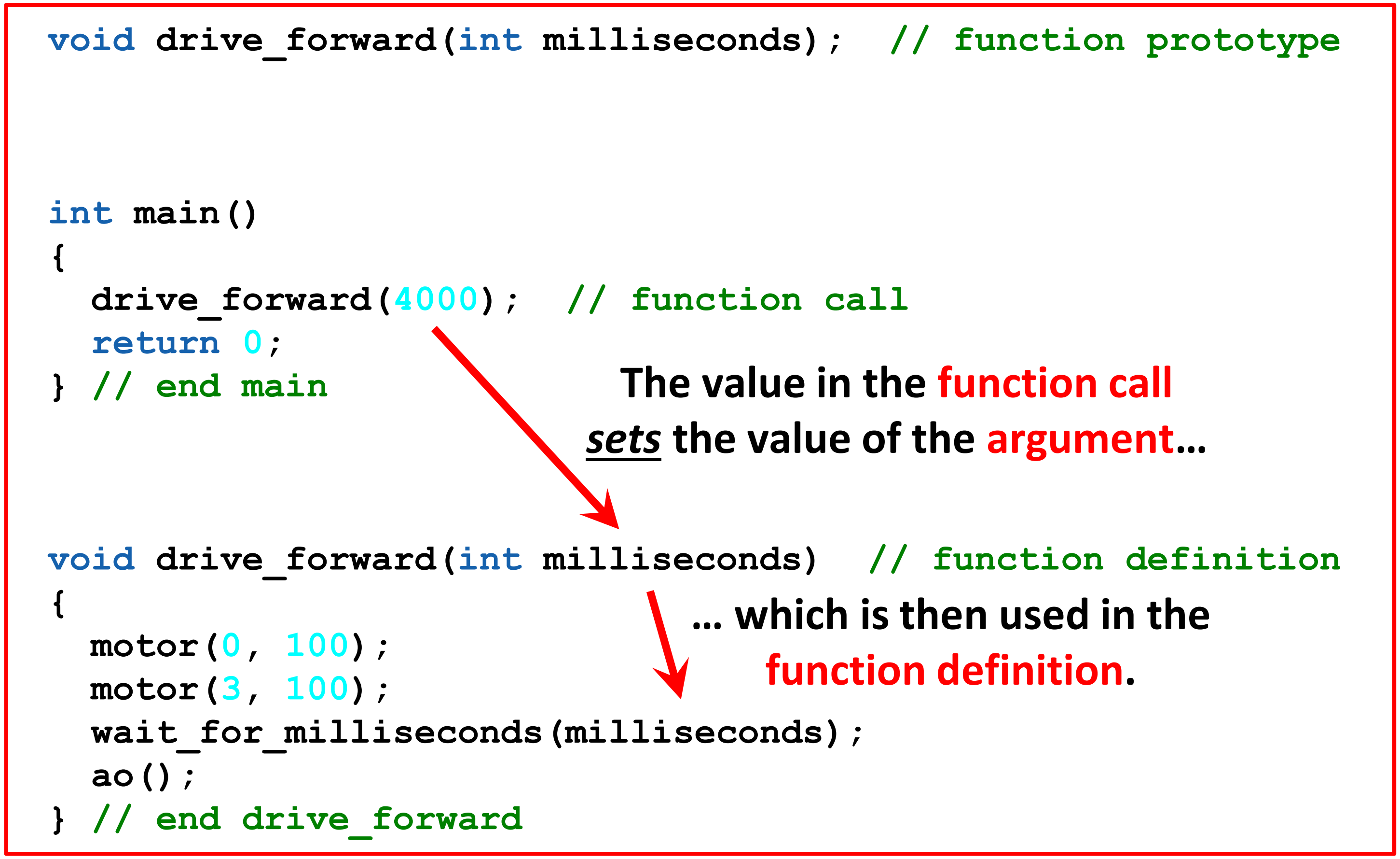
What if you don’t want it to run for this long each time?

Functions with arguments

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

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

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

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

Writing your own functions with arguments



```c
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, 100);
  motor(3, 100);
  wait_for_milliseconds(milliseconds);
  ao();
} // end drive_forward
```

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

… which is then used in the function definition.

Writing your own functions with arguments

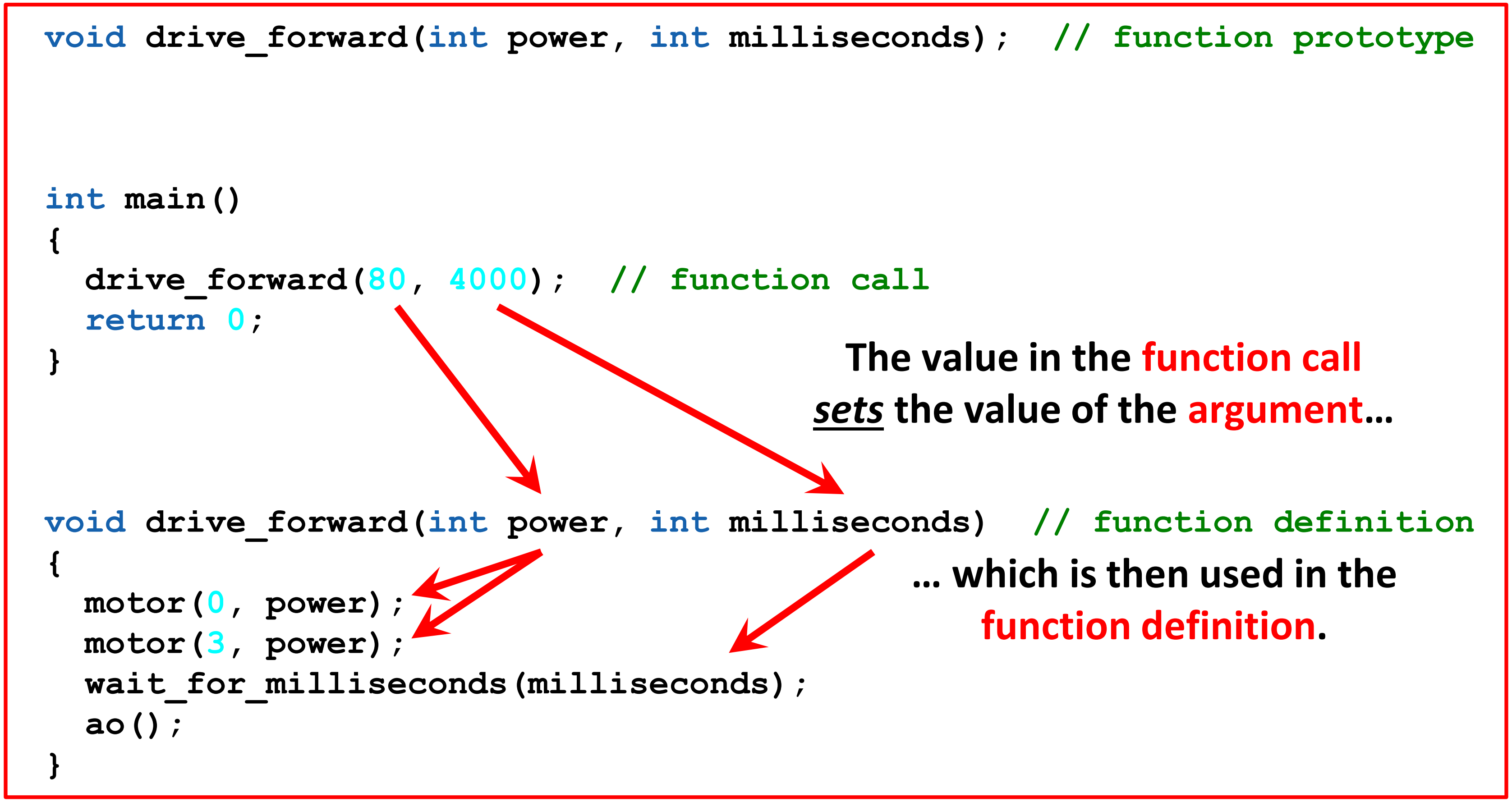
The function prototype and the function definition look the same except for one thing…

```
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, 100);
  motor(3, 100);
  wait_for_milliseconds(milliseconds);
  ao();
} // end drive_forward
```

Notice: no semicolon!

(Why not?)

Writing your own functions with multiple arguments



```
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);
  wait_for_milliseconds(milliseconds);
  ao();
}
```

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

… which is then used in the function definition.

Repetition, Repetition, Repetition

Program flow control with loops

`while` loops for counting

`while` and Boolean operators

Program flow control with loops

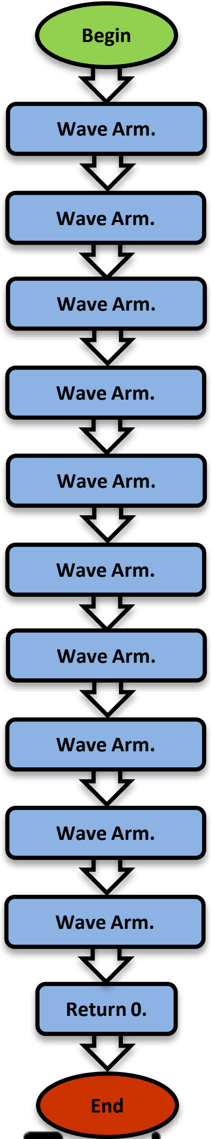
Suppose your task is to wave the robot arm 10 times…

Pseudocode

```
1.   Wave Arm.
2.   Wave Arm.
3.   Wave Arm.
4.   Wave Arm.
5.   Wave Arm.
6.   Wave Arm.
7.   Wave Arm.
8.   Wave Arm.
9.   Wave Arm.
10.  Wave Arm.
11.  End the program.
```

Comments

```
//  1. Wave Arm.
//  2. Wave Arm.
//  3. Wave Arm.
//  4. Wave Arm.
//  5. Wave Arm.
//  6. Wave Arm.
//  7. Wave Arm.
//  8. Wave Arm.
//  9. Wave Arm.
// 10. Wave Arm.
// 11. End the program.
```



Begin

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Wave Arm.

Return 0.

End

Program flow control with loops

Now, suppose your objective is to wave the arm 50 times…

… or 100 times…

… or 1,000 times…

… or 12,345 times…

You could copy-and-paste lines of code, but it would take a very long time…

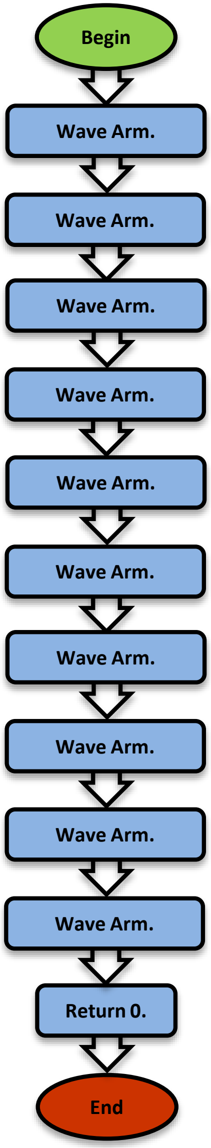
There has got to be a better way!

(And there is!)

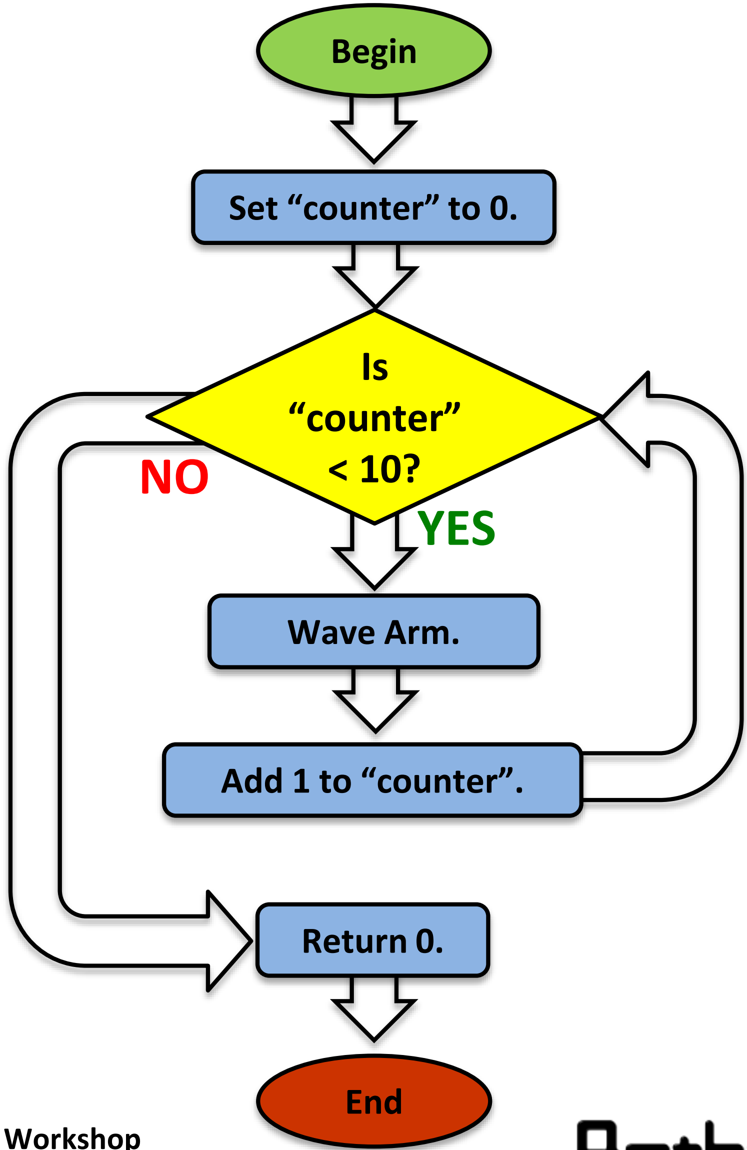
Program flow control with loops

- What if we want to ***repeat*** the same **block of code** many times?
- We can do this using a **loop**, which controls the **flow** of the program by repeating a **block of code**.

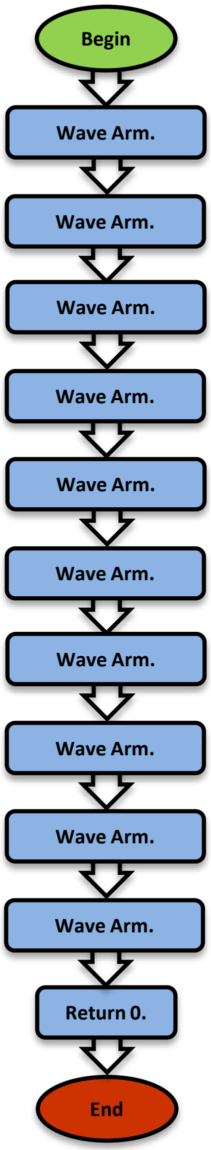
Program flow control with loops



— VS —

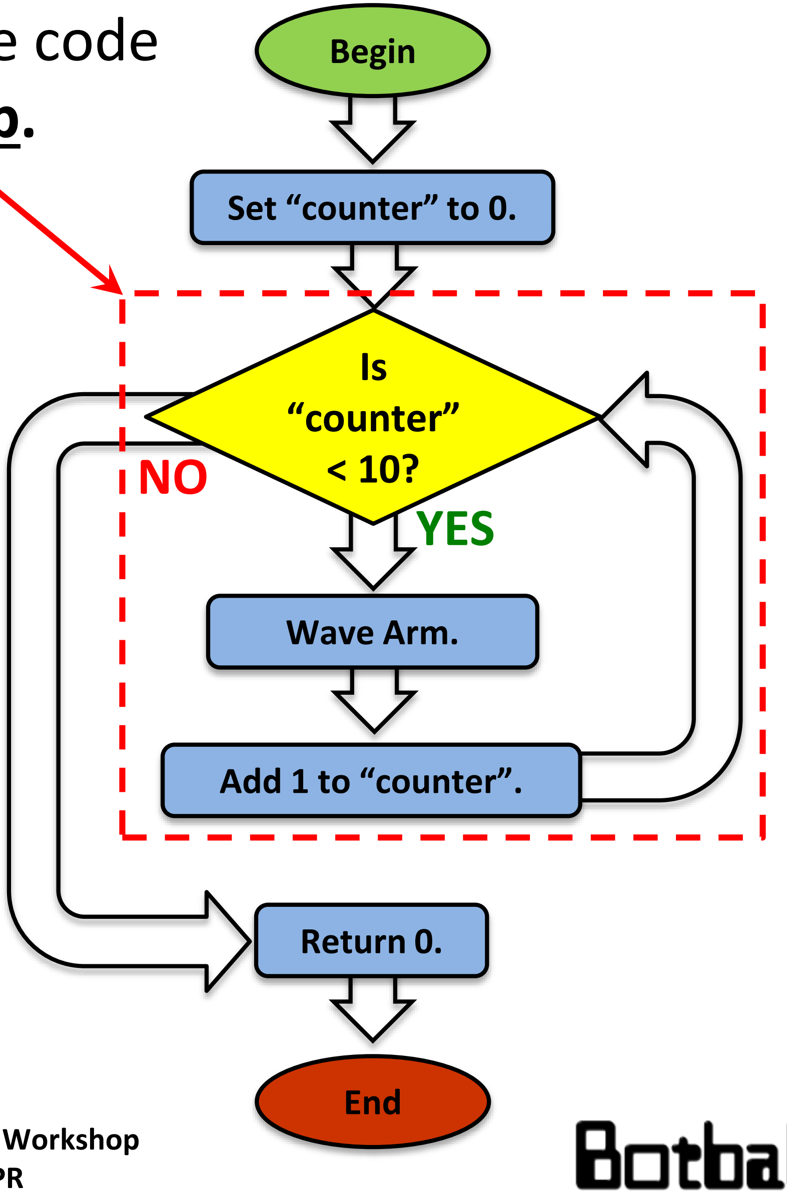


Program flow control with loops



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

— VS —



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) // Loop
  {
    // Code to repeat ...
  }

  // Code after loop ...
  return 0;
}
```

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;
}
```

Begin

End

Block Header
(no semicolon!)

`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

Resource

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`

Draw a square using a loop

Activity

Description: Write a program for the KIPR Wallaby that drives the DemoBot along a path in the shape of a square *using loops*.

- **Hint:** modify your old square-drawing program to use a `while` loop.
- **Bonus:** use a `while` loop *and* functions!

Analysis: What is the program supposed to do?

Pseudocode

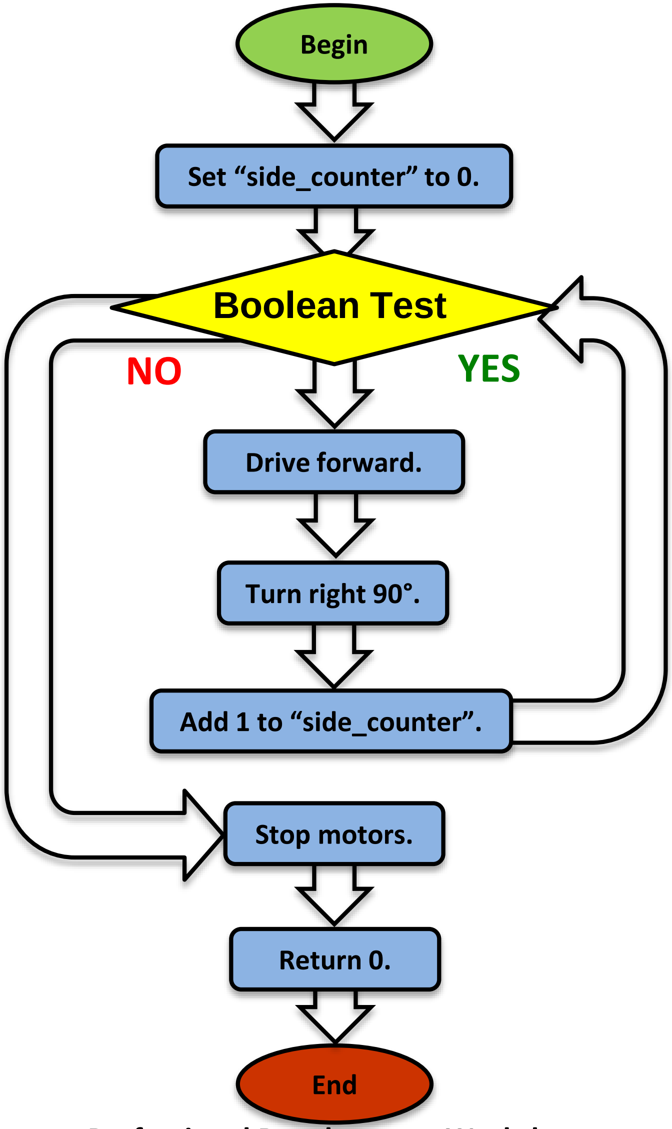
```
1.  Set  Variable “side_counter” to 0.
2.  Loop: Is “side_counter” < 4?
    1.  Drive forward.
    2.  Turn right 90°.
    3.  Add 1 to “side_counter”.
3.  Stop motors.
4.  End the program.
```

Comments

```
// 1. Set Variable “side_counter” to 0.
// 2. Loop: Is “side_counter” < 4?
//    2.1. Drive forward.
//    2.2. Turn right 90-degrees.
//    2.3. Add 1 to “side_counter”.
// 3. Stop motors.
// 4. End the program.
```

Draw a square using a loop

Analysis: Flowchart



Boolean Test
"side_counter" is < 4?

Draw a square using a loop

Activity

Solution:

Comments

```
int main()
{
   // 1. Set "side_counter" to 4.
   // 2. Loop: Is "side_counter" < 4?
   //     2.1. Drive forward.
   //     2.2. Turn right 90-degrees.
   //     2.3. Add 1 to "side_counter".
   // 3. Stop motors.
   // 4. End the program.
}
```

Source Code

```
int main()
{
  int side_counter = 0; // declare and
  //initialize variable all in one line

  while (side_counter < 4)
  {
    motor(0, 100);
    motor(3, 100);
    wait_for_milliseconds(4000); // forward

    motor(0,  100);
    motor(3, -100);
    wait_for_milliseconds(1500); //right turn

    side_counter = side_counter + 1;
  }

   ao();

   return 0;
}
```

Draw a square using a loop

Activity

Solution: Use a function!

Comments

```
int main()
{
  // 1. Set "side_counter" to 4.
  // 2. Loop: Is "side_counter" < 4?
  //    2.1. Drive forward.
  //    2.2. Turn right 90-degrees.
  //    2.3. Add 1 to "side_counter".
  // 3. Stop motors.
  // 4. End the program.
}
```

Source Code

```
void drive_forward_and_turn_right();

int main()
{
  int side_counter = 0 ;

  // loop exits when side counter = 4
  while (side_counter < 4)
  {
    drive_forward_and_turn_right();

    side_counter = side_counter + 1;
  }

  ao();
  return 0;
}


void drive_forward_and_turn_right()
{
  motor(0, 100);
  motor(3, 100);
  wait_for_milliseconds(4000);

  motor(0,  100);
  motor(3, -100);
  wait_for_milliseconds(1500);


  ao();
}
```

Move the servo arm using a loop

Activity

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 your program, and **disable the servos** at the end of your program!

Analysis: What is the program supposed to do?

Pseudocode

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

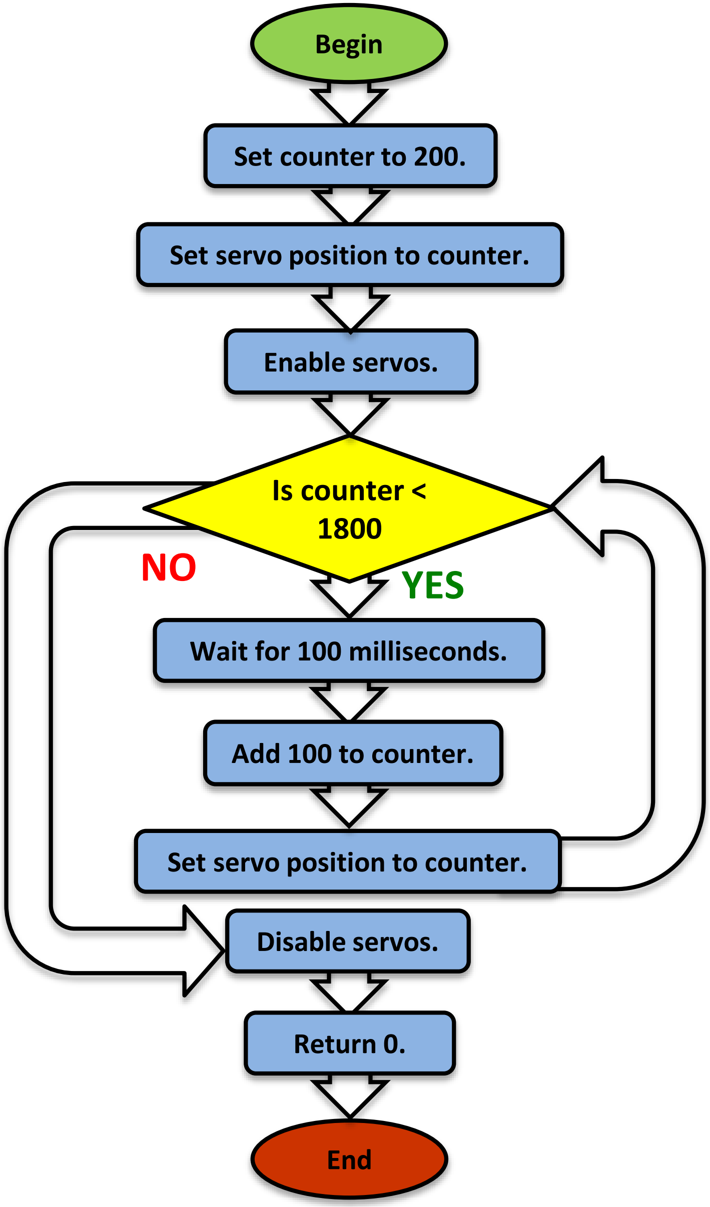
Comments

```
// 1. Set counter to 200
// 2. Set servo position to counter
         // 3. Enable servos.
// 4. Loop: Is counter < 1800?
//    4.1. Wait for 100 milliseconds.
//    4.2. Add 100 to servo position.
//    4.3 Set servo position to counter.
// 5. Disable servos.
// 6. End the program.
```

Move the servo arm using a loop

Activity

Analysis: Flowchart



Move the servo arm using a loop

Activity

Solution:

Comments

```
int main()
{
  // 1. Set counter to 200.
  // 2. Set servo position to counter.
  // 3. Enable servos.
  // 4. Loop: Is counter < 1800?
  //      4.1. Wait for 0.1 seconds.
  //      4.2. Add 100 to counter.
  //      4.3. Set servo position to counter.
  // 5. Disable servos.
  // 6. End the program.
}
```

Source Code

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

  set_servo_position(0, counter);

  enable_servos();

  // Is counter < 1800?
  while (counter < 1800)
  {
    wait_for_milliseconds(100);

    // Add 100 to counter
    counter = counter + 100;

    // Set servo position to counter
    set_servo_position(0, counter);
  }

  // Disable servos.
  disable_servos();

  return 0;
}
```

Making Smarter Robots with Sensors

Analog and digital sensors

Light and touch sensors

`wait_for_light()` and `wait_for_touch()` functions

Sensors

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

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

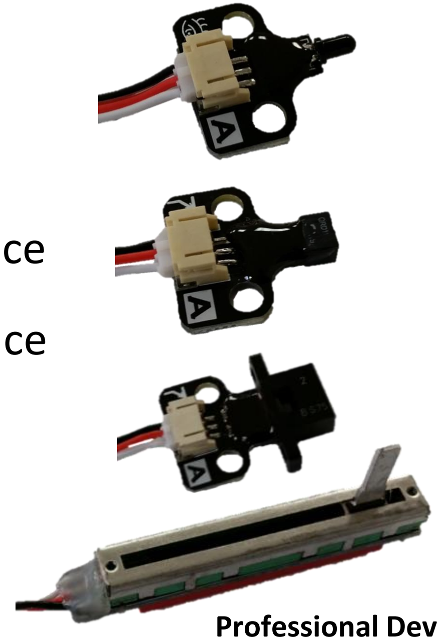
- Robot **sensors** are like human **senses**!
  - What **senses** does a **human** have?
  - What **sensors** should a **robot** have?

Analog and digital sensors

Analog Sensors

- **Range of values:**

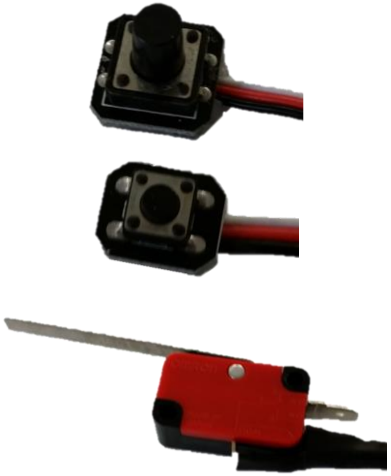
  0 – 4095
- **Ports:** 0 – 5
- **Function:** `analog(port #)`
- **Sensors:**
  - Light
  - Small reflectance
  - Large reflectance
  - Slide sensor



Digital Sensors

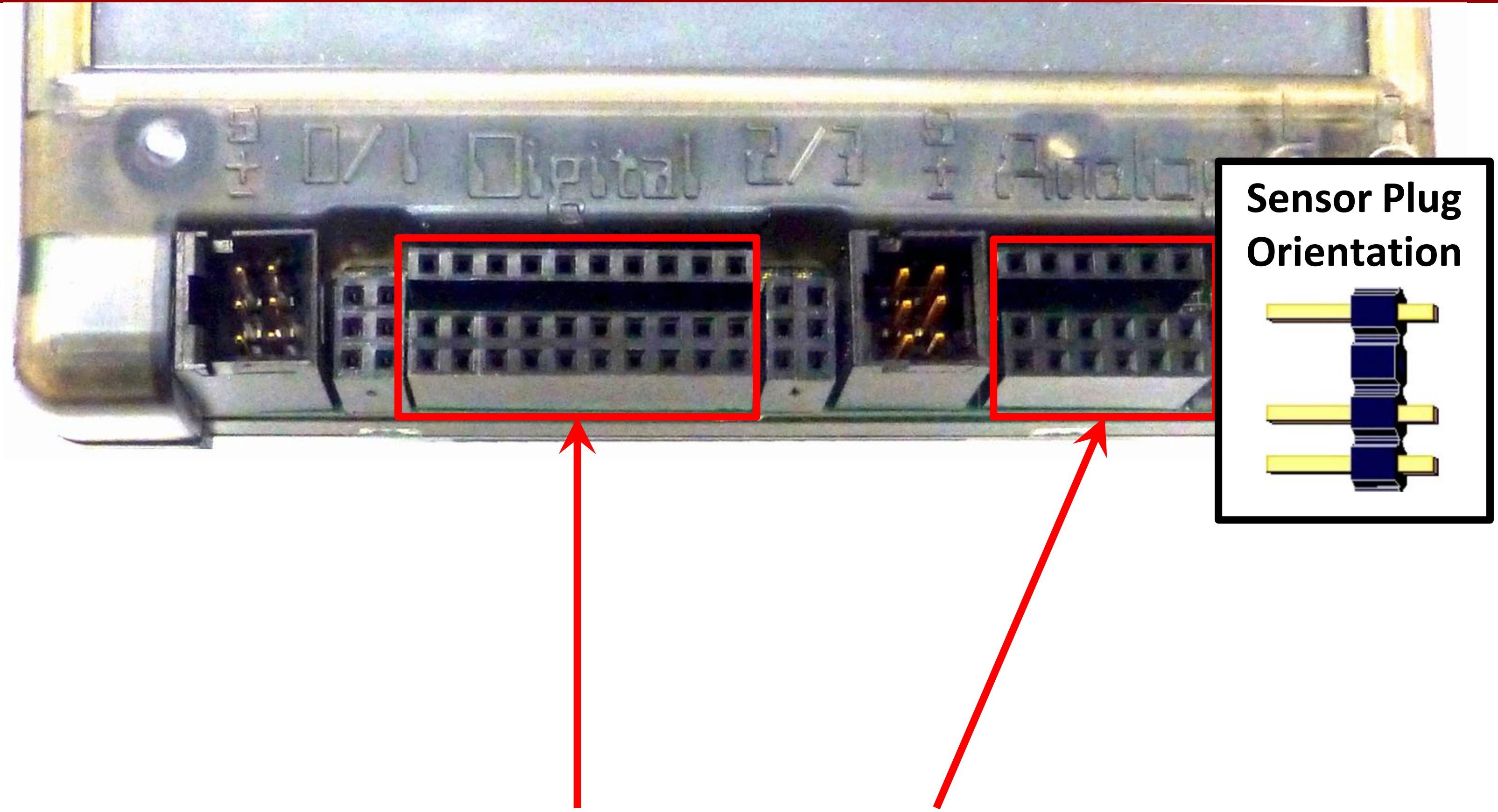
- **Range of values:**

  0 (not pressed) or 1 (pressed)
- **Ports:** 0 – 9
- **Function:** `digital(port #)`
- **Sensors:**
  - Large touch
  - Small touch
  - Lever touch



KIPR Robotics Controller sensor ports

Activity



Sensor Plug Orientation

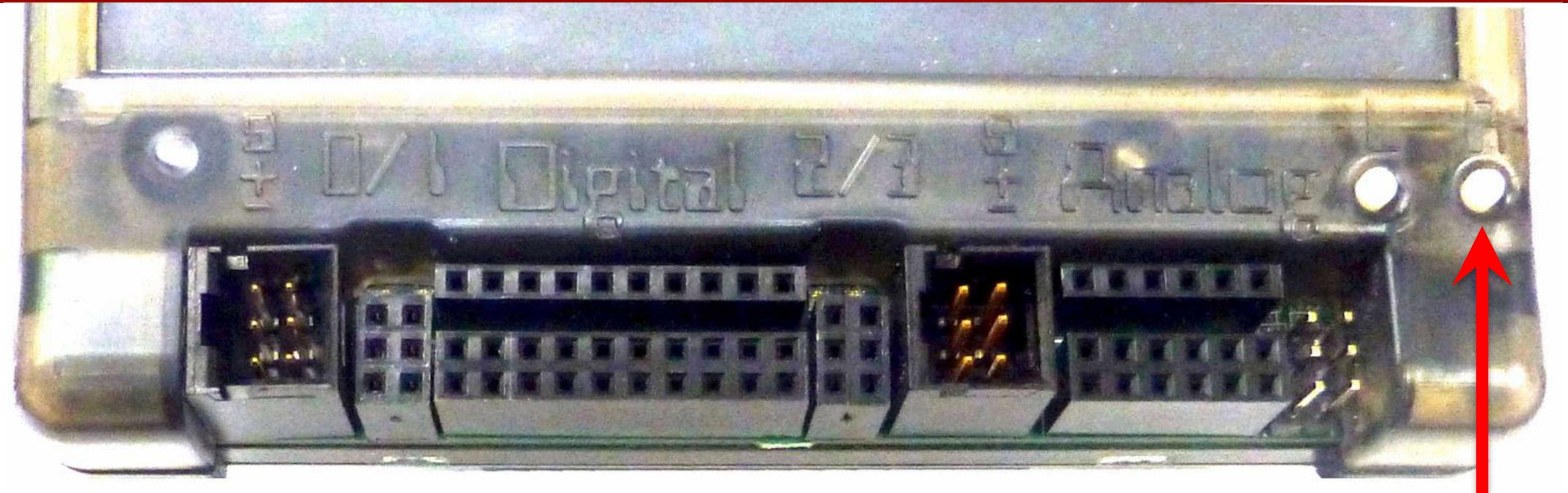
Digital Sensor Ports # 0 – 9

Analog Sensor Ports # 0-5

Built-In Digital Sensor: Buttons

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

Sensor waiting functions

```
wait_for_light(3);
// Waits for the light on port #3 before going to the next line.

wait_for_touch(8);
// Waits for the touch on port #8 before going to the next line.
```

Using wait_for_light

What is this?

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

Starting your programs with a light

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

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

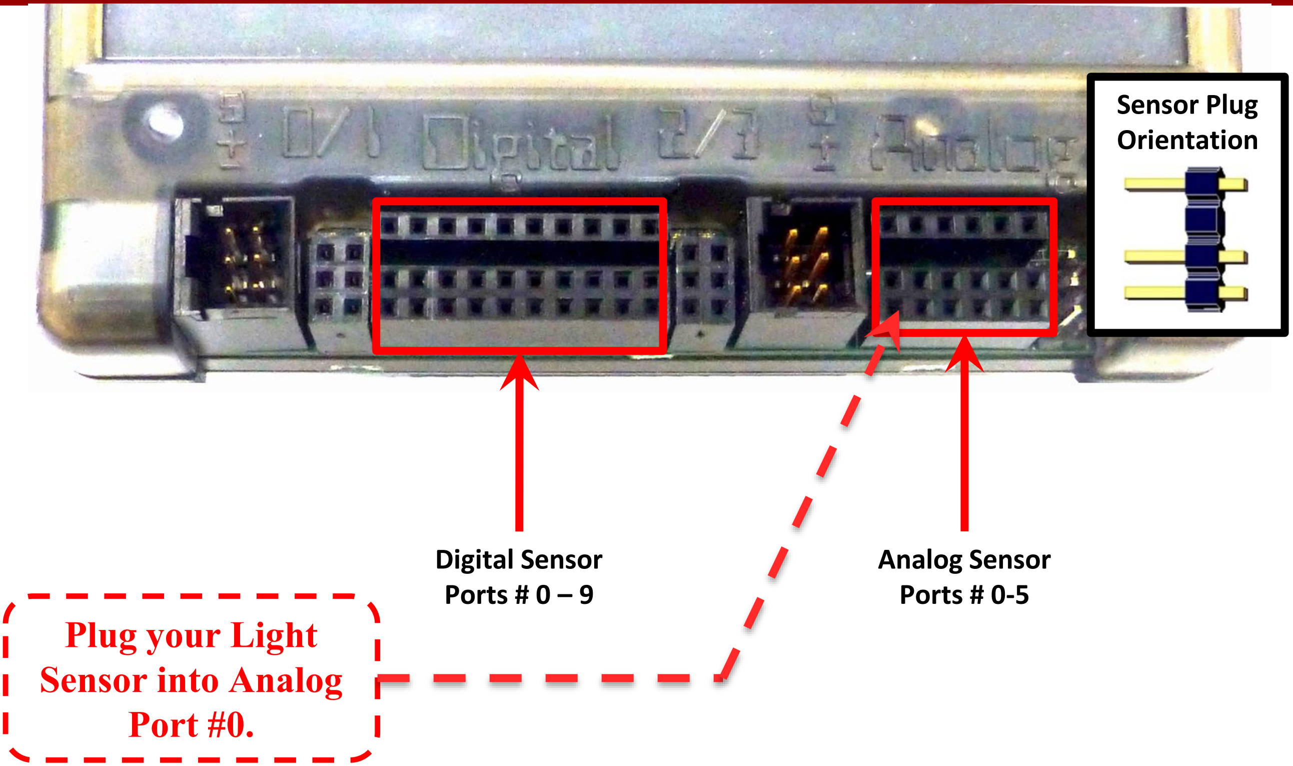
- The light sensor senses *infrared light*, so light must be emitted from an *incandescent light*, not an *LED light*.
  - For our activities, you can use a flashlight.

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

Activity

Plug in your light sensor

(and get your flashlight!)

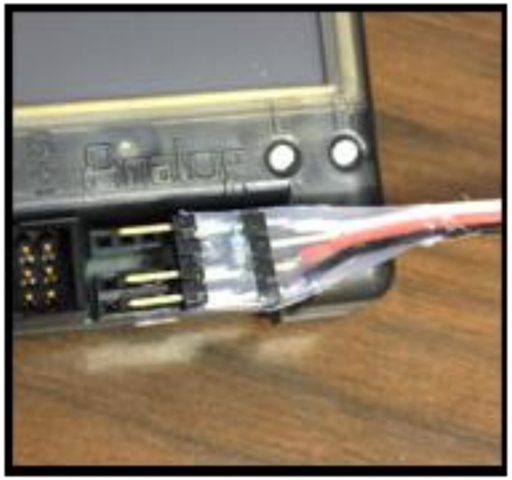


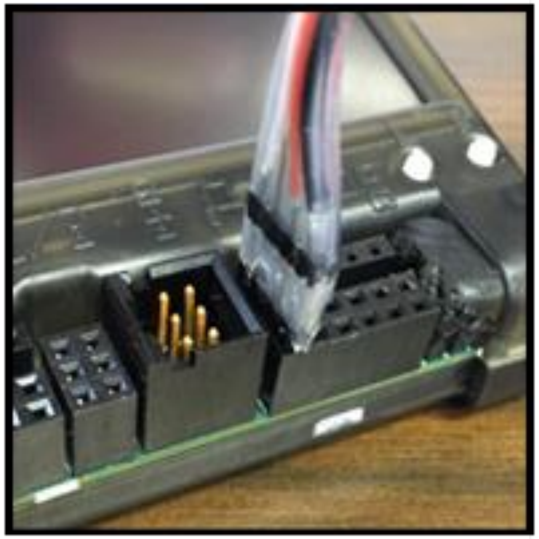
Sensor Plug Orientation

Digital Sensor Ports # 0 – 9

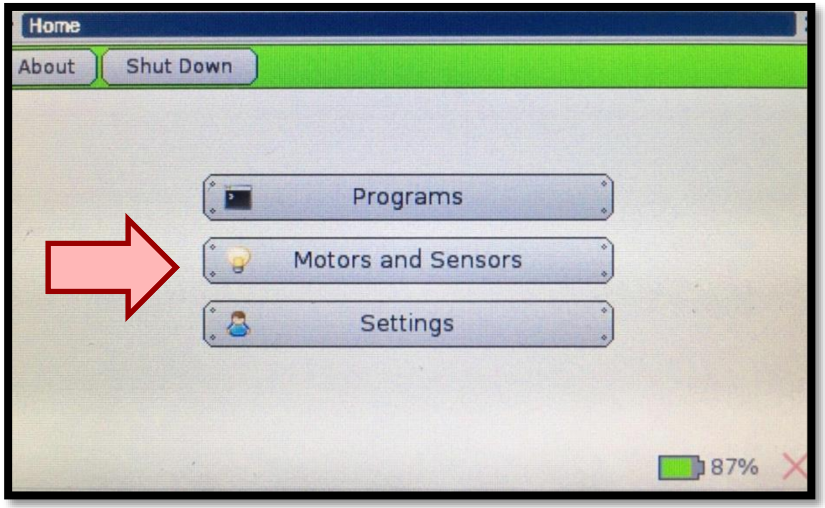
Analog Sensor Ports # 0-5

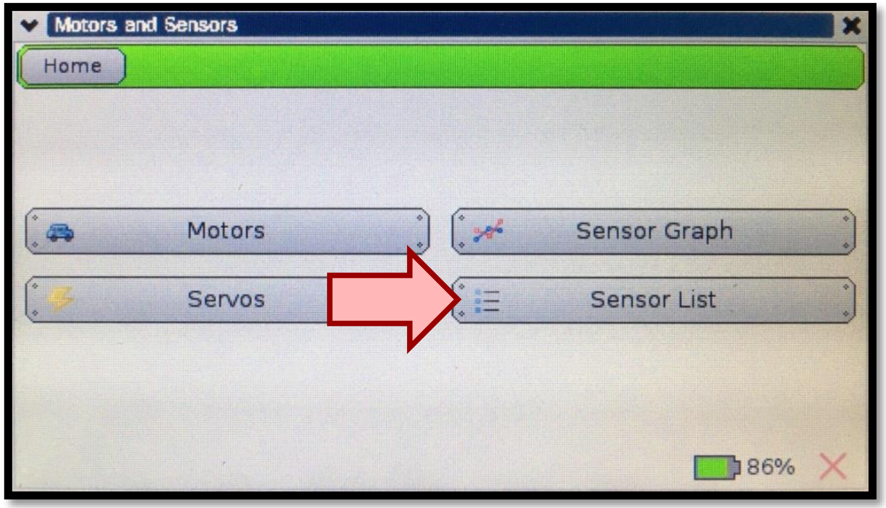
Plug your Light Sensor into Analog Port #0.

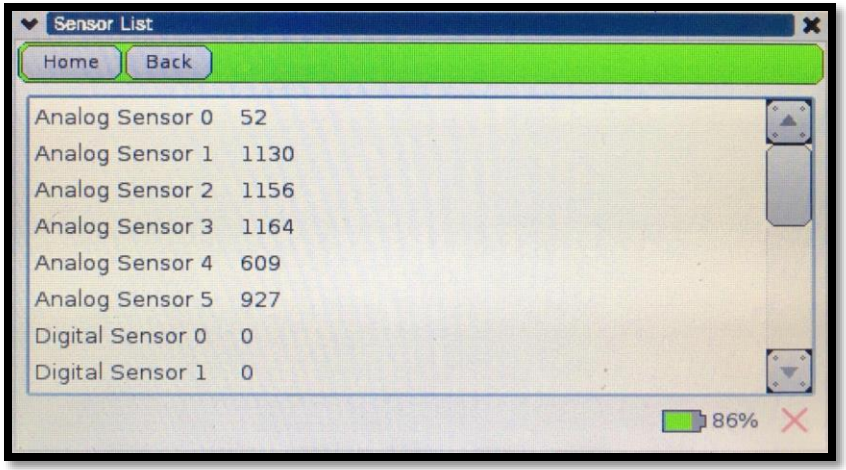




Use the sensor list

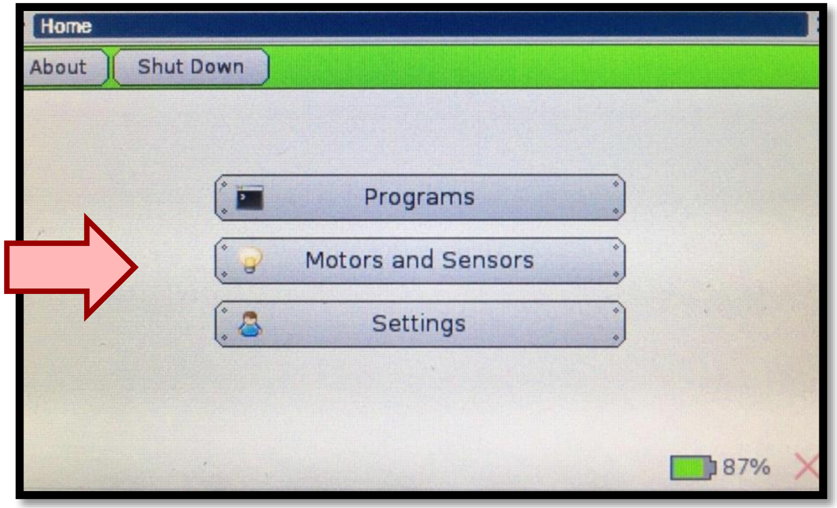


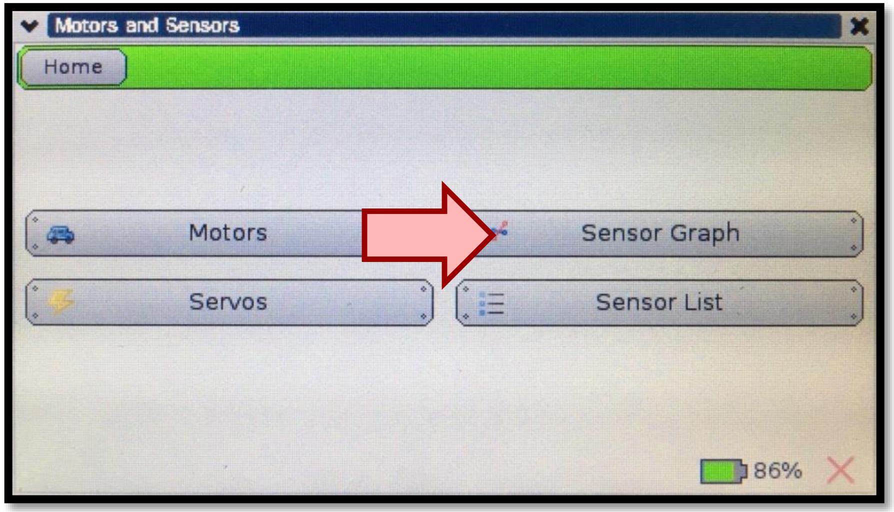


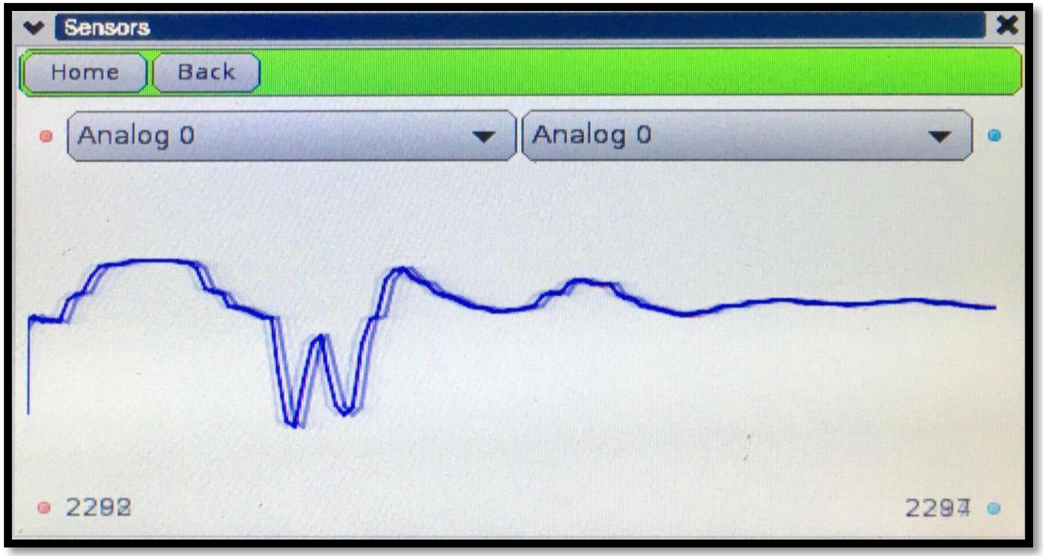


```
Analog Sensor 0   52
Analog Sensor 1   1130
Analog Sensor 2   1156
Analog Sensor 3   1164
Analog Sensor 4   609
Analog Sensor 5   927
Digital Sensor 0  0
Digital Sensor 1  0
```

Use the sensor graph







Starting with a light

Activity

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?

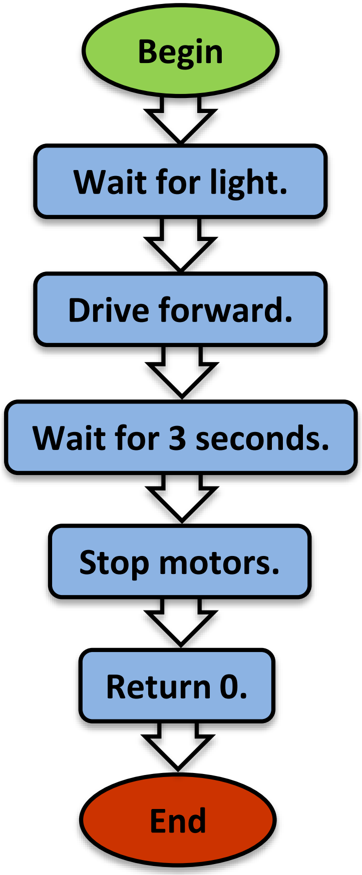
Pseudocode

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



Begin

Wait for light.

Drive forward.

Wait for 3 seconds.

Stop motors.

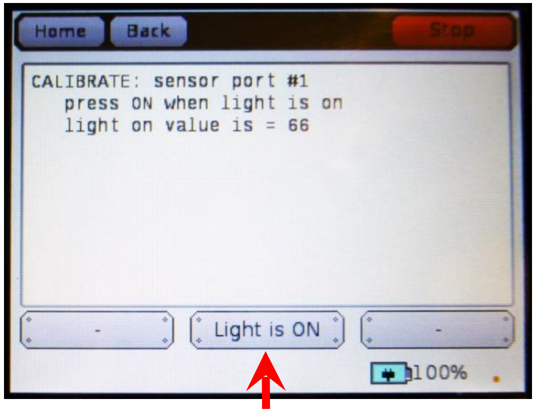
Return 0.

End

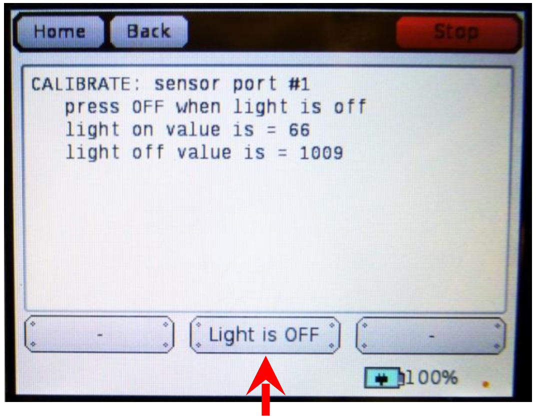
wait_for_light calibration routine

Activity

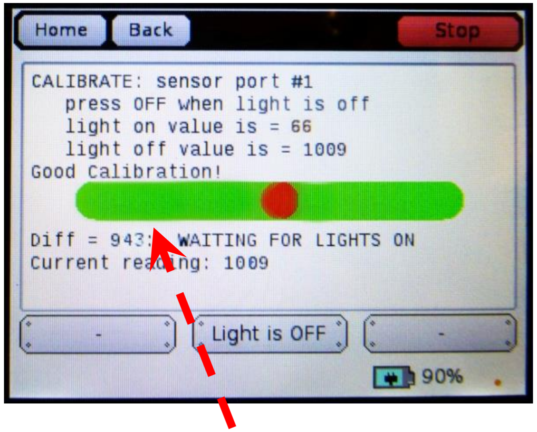
When you use the `wait_for_light()` function in your program, the following calibration routine will run automatically.



```
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.

You will get a "Good Calibration!" message and moving red dot on green bar when done *correctly*. You will get a "BAD CALIBRATION" message when not done correctly, and you will need to run through the routine again.

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

Activity

Starting with a light

Solution:

Comments

```
int main()
{
  // 1. Wait for light.
  // 2. Drive forward.
  // 3. Wait for 3 seconds.
  // 4. Stop motors.
  // 5. End the program.
}
```

Source Code

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

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

  return 0;
}
```

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

Starting with a light

Activity

Solution: Use a function!

Comments

```
int main()
{
  // 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();
  wait_for_milliseconds(3000);

  ao();

  return 0;
}

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

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

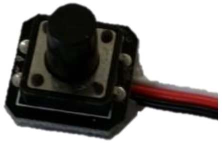
Detecting touch

```
wait_for_touch(port #);
```

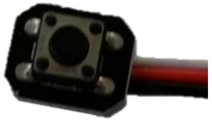
Digital Port #0 – 9

- The `wait_for_touch()` function pauses your program until a **digital sensor** on the specified **port #** reads a value of 1 (pressed or **touched**).
  - **Note:** Unlike `wait_for_light()`, it does **not** have a built-in calibration routine because it is not necessary—touched is touched!

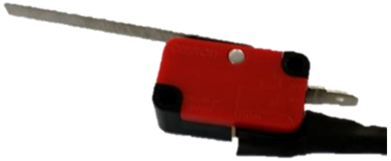
- There are many digital sensors in your kit that can detect touch…



Large touch



Small touch



Lever touch

Select the one that can be easily attached *and* can easily detect the objects.

Using wait_for_touch

What is this?

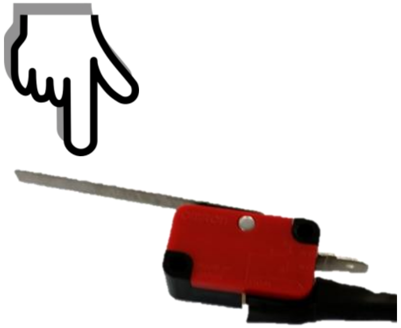
```
int main()
{
  wait_for_touch(8);
  printf("I'm touched!\n");
  return 0;
}
```

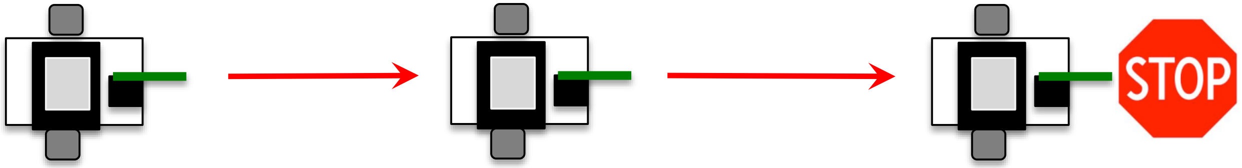
Drive until touched

Activity

Description: Write a program for the KIPR Wallaby that drives the DemoBot forward until it detects a touch, then stops.

- Use a **lever touch sensor**.
- Plug the lever touch sensor into ***any*** of the **digital sensor ports** (#0– 9).
- You can either **attach the sensor to your robot**, or **hold it in your hand** and **manually press it** whenever you would like the robot to stop.





Drive until touched

Activity

Description: Write a program for the KIPR Wallaby that drives the DemoBot forward until it detects a touch, then stops.

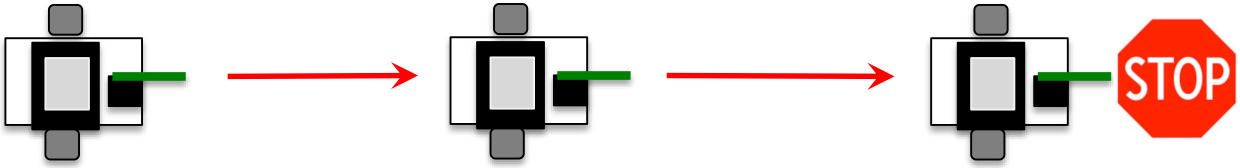
Analysis: What is the program supposed to do?

Pseudocode

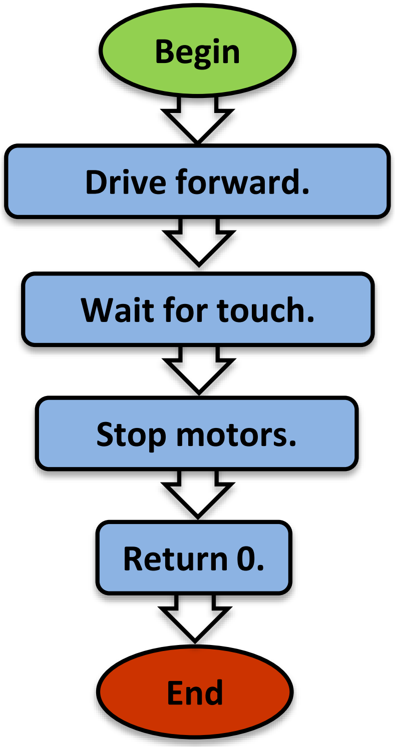
```
1.  Drive forward.
2.  Wait for touch.
3.  Stop motors.
4.  End the program.
```

Comments

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



Flowchart



Drive until touched

Activity

Solution:

Comments

```
int main()
{
  // 1. Drive forward.
  // 2. Wait for touch.
  // 3. Stop motors.
  // 4. End the program.
}
```

Source Code

```
int main()
{
  motor(0, 100);
  motor(3, 100);

  wait_for_touch(8);

  ao();

  return 0;
}
```

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

Drive until touched

Activity

Solution: Use a function!

Comments

```
int main()
{
  // 1. Drive forward.
  // 2. Wait for touch.
  // 3. Stop motors.
  // 4. End the program.
}
```

Source Code

```
void drive_forward();
int main()
{
  drive_forward();
  wait_for_touch(8);

  ao();

  return 0;
}

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

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

Drive until touched

Activity

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

- What happens if the robot goes too fast?
- How does **`wait_for_touch()`** work?
- How can I write my own version of something like it?
- To do this, we go back to our concept of using a **loop** (see next section).

More Repetition, Repetition, Repetition

Program flow control with sensor driven loops

`while` and Boolean operators

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

Using while loops

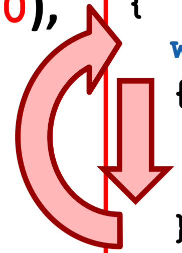
What is this?

What does this say?

```
int main()
{
  while (digital(8) == 0)
  {
    motor(0, 100);
    motor(3, 100);
  }
  ao();
  return 0;
}
```

Using while loops

While the touch sensor is not pressed (== 0), move forward.



```
int main()
{
  while (digital(8) == 0)
  {
    motor(0, 100);
    motor(3, 100);
  }
  ao();
  return 0;
}
```

What does this say?

while and Boolean operators examples

```
while (digital(15) == 0)
{
  // Code to repeat ...
}
```

```
while (digital(15) == 0)
{
  // Code to repeat ...
}
```

```
while (analog(3) < 512)
{
  // Code to repeat ...
}
```

```
while (countdown >= 1)
{
  // Code to repeat ...
}
```

Activity

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.
```

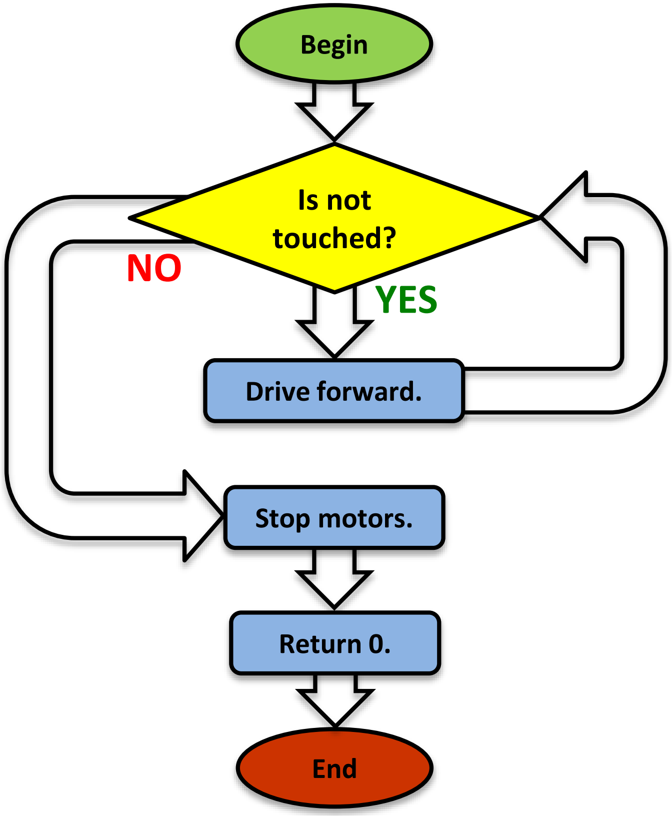
Comments

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

Drive until sensor is pressed

Activity

Analysis: Flowchart



Drive until sensor is pressed

Activity

Solution:

Comments

```
int main()
{
  // 1. Loop: Is not touched?
  //    1.1. Drive forward.
  // 2. Stop motors.
  // 3. End the program.
}
```

Source Code

```
int main()
{
  while (digital(8) == 0)
  {
    motor(0, 100);
    motor(3, 100);
  }

  ao();

  return 0;
}
```

Making a Choice

Program flow control with conditionals

`if-else` conditionals

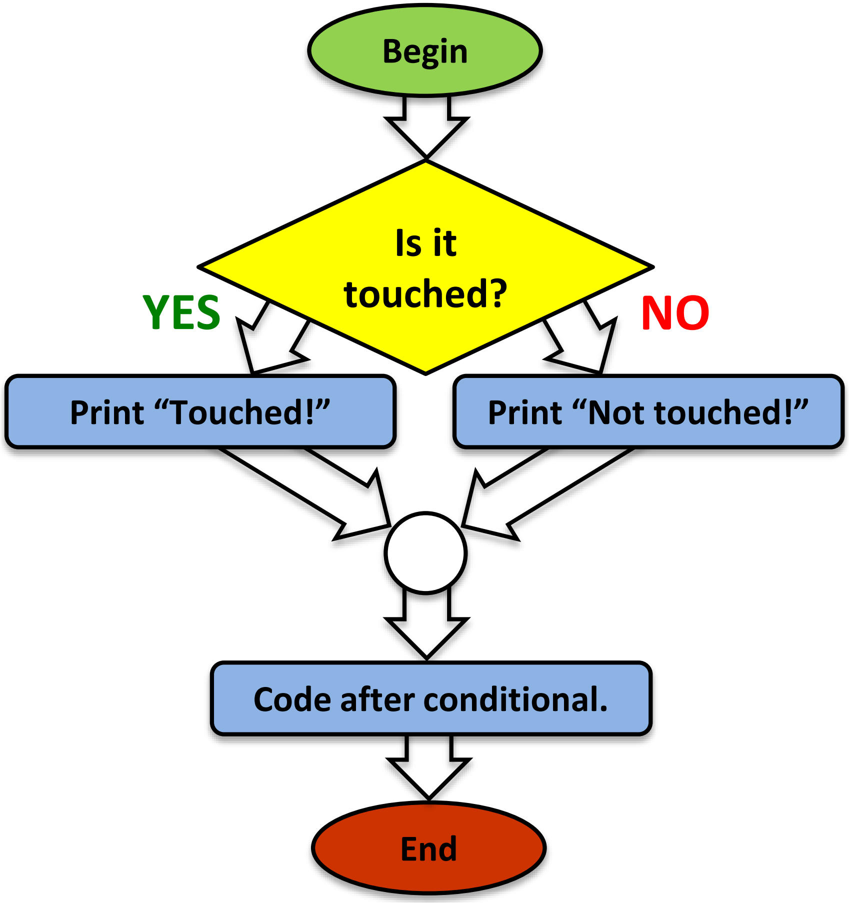
`if-else` and Boolean operators

Using `while` and `if-else`

Program flow control with conditionals

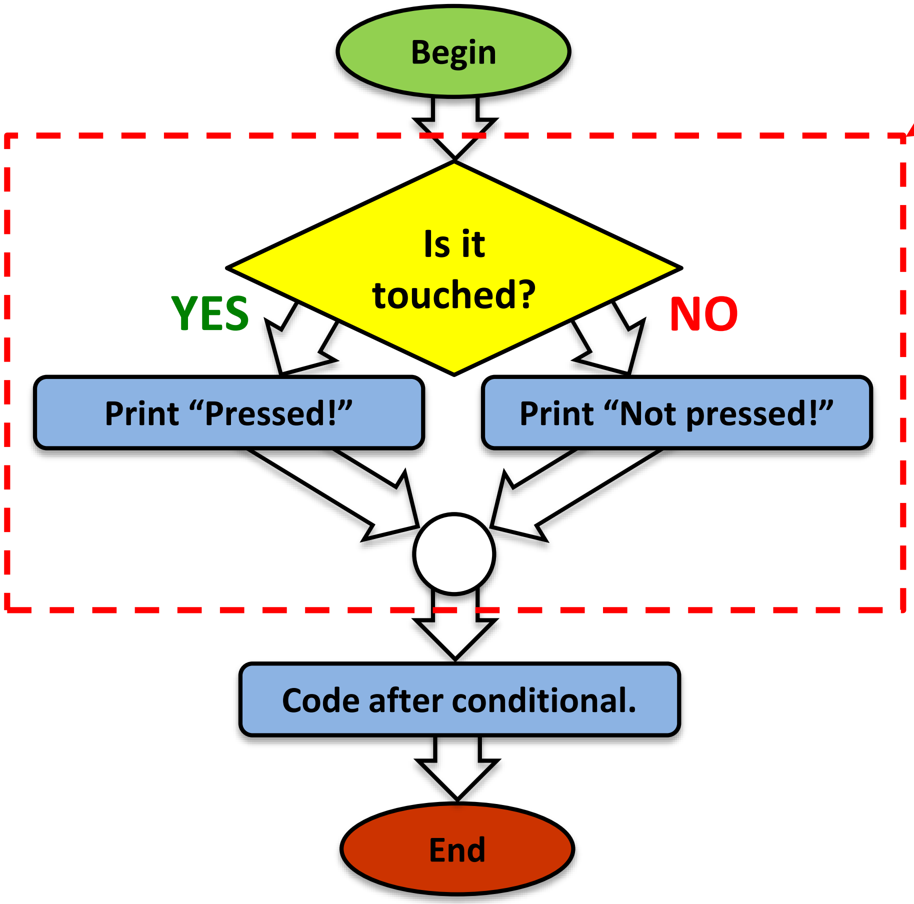
- What if we want to execute a **block of code *only if certain conditions are met***?
- We can do this using a **conditional**, which controls the **flow** of the program by executing ***one* block of code** if its conditions are met or a ***different* block of code** if its conditions are not met.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

Program flow control with conditionals



Program flow control with conditionals

This part of the code is the conditional.



Program flow control with conditionals

Pseudocode

```
1.  If: Is touched?
    1.  Print "Touched!".
2.  Else.
    1.  Print "Not touched!".
3.  End the program.
```

Comments

```
//   1. If: Is touched?
//        1.1. Print "Touched!".
//   2. Else.
//        2.1. Print "Not touched!".
//   3. End the program.
```

In the **C** programming language,
we accomplish this with an `if-else` **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**.

```
int main()
{

  if (Boolean test)
  {
    // Code to execute ...
  }
  else
  {
    // Code to execute ...
  }

  return 0;
}
```

Using if-else conditionals

What is this?

What does this say?

```
int main()
{
  if (digital(8) == 1)
  {
    printf(“Touched!\n");
  }
  else
  {
    printf(“Not touched!\n");
  }
  return 0;
}
```

Using if-else conditionals

```
int main()
{
  if (digital(8) == 1)
  {
    printf(“Touched!\n");
  }
  else
  {
    printf(“Not touched!\n");
  }
  return 0;
}
```

Notice: no semicolon!
(Why not?)

if-else conditionals

```
int main()
{
  // Code before conditional ...

  if (Boolean test)
  {
    // Code to execute if test is true
  }
  else
  {
    // Code to execute if test is false
  }

  return 0;
}
```

The else is <u>below</u> the } brace of the if block of code!

if-else examples

```
if (right_button() == 0)
{
  // Code to execute ...
}
else
{
  // Code to execute ...
}
```

```
if (analog(3) < 512)
{
  // Code to execute ...
}
else
{
  // Code to execute ...
}
```

Example using while and if-else

```
int main()
{
  while (right_button() == 0)
  {
    if (analog(0) > 512)
    {
      printf("It's dark in here!\n");
    }
    else
    {
      printf("I see the light!\n");
    }
  } // loop ends when button is pressed
  return 0;
}
```

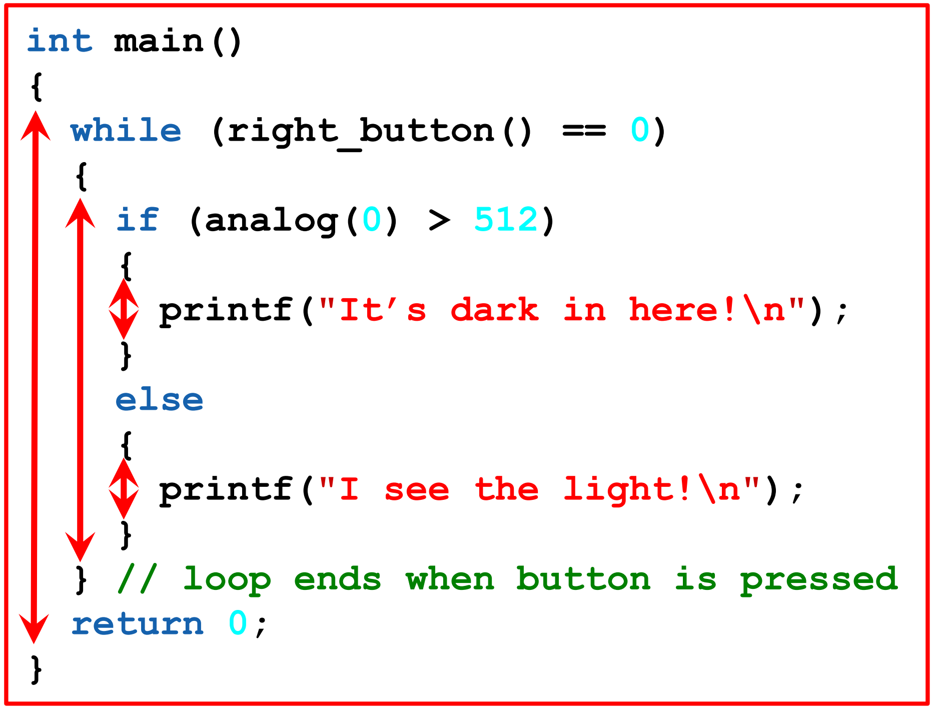
What do these lines of code say?

Note the ==

(2 equal signs)

Using while and if-else

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



```
int main()
{
  while (right_button() == 0)
  {
    if (analog(0) > 512)
    {
      printf("It's dark in here!\n");
    }
    else
    {
      printf("I see the light!\n");
    }
  } // loop ends when button is pressed
  return 0;
}
```

Reflectance sensor for line-following

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

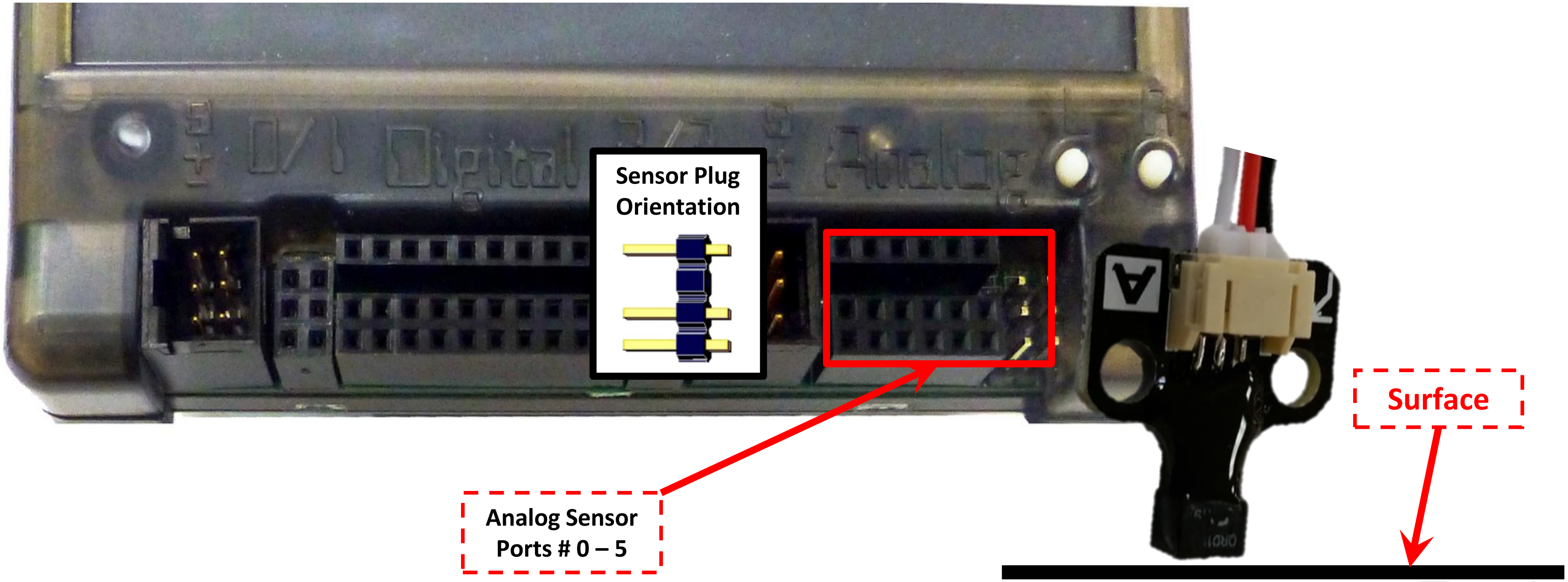
- This sensor is really a short-range reflectance sensor.
- There is both an infrared (IR) *emitter* and an IR *detector* inside of this sensor.
- IR *emitter* sends out IR light → IR *detector* measures how much reflects back.
- The amount of IR reflected back depends on many factors, including **surface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

Attach your reflectance sensor

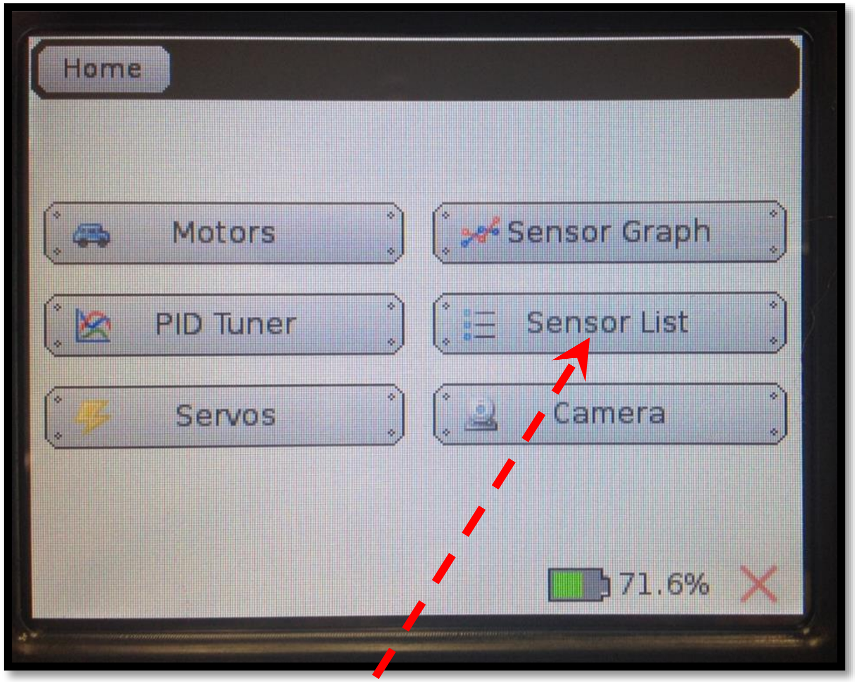
- Attach the sensor on the front of your robot so that it is **pointing down at the ground** and is **approximately 1/8" from the surface**.
- A **reflectance sensor** is an **analog sensor**, so plug it into any of **analog sensor port #0 – 5**. Port 0 for this example.
  - Recall that analog sensor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



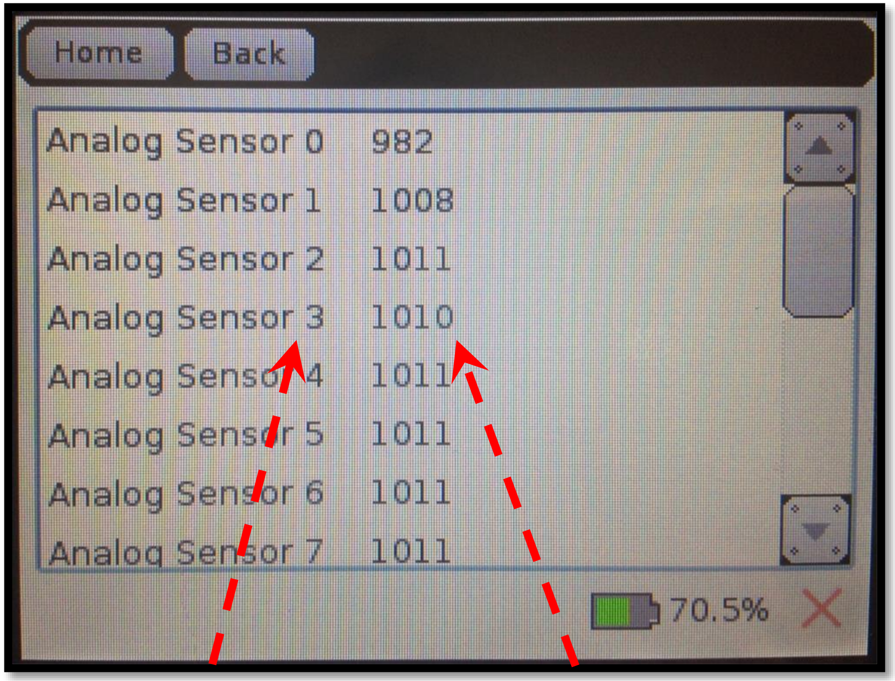
Activity

Reading sensor values from the Sensor List screen

- View sensor values from the **Sensor List** on your KIPR Wallaby.
  - This is very helpful to view readings from all of the sensors you are using.
  - You can view the values, then use them in your code.



Select "Sensor List".



Analog Sensor 0	982
Analog Sensor 1	1008
Analog Sensor 2	1011
Analog Sensor 3	1010
Analog Sensor 4	1011
Analog Sensor 5	1011
Analog Sensor 6	1011
Analog Sensor 7	1011

Sensor Ports

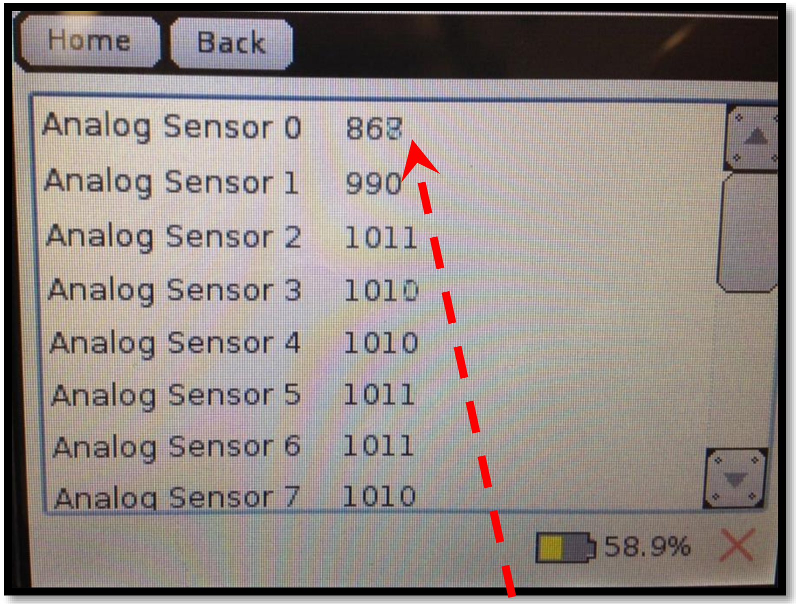
Sensor Values

Reading sensor values from the Sensor List screen

Activity

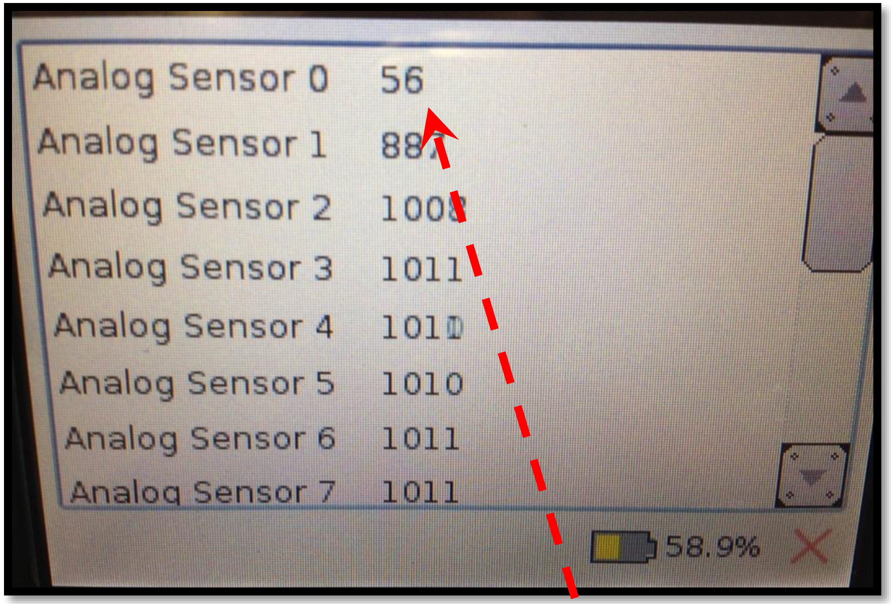
- With the reflectance sensor plugging into analog port #0…
  - Over a **black surface**, the sensor reading is **3863**.
  - Over a **white surface**, the sensor reading is **156**.

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



Analog Sensor 0 868
Analog Sensor 1 990
Analog Sensor 2 1011
Analog Sensor 3 1010
Analog Sensor 4 1010
Analog Sensor 5 1011
Analog Sensor 6 1011
Analog Sensor 7 1010

58.9%



Analog Sensor 0 56
Analog Sensor 1 887
Analog Sensor 2 1008
Analog Sensor 3 1011
Analog Sensor 4 1010
Analog Sensor 5 1010
Analog Sensor 6 1011
Analog Sensor 7 1011

58.9%

Value of 3863
(Black Surface)

Value of 156
(White Surface)

Activity 3 (connections to the game)

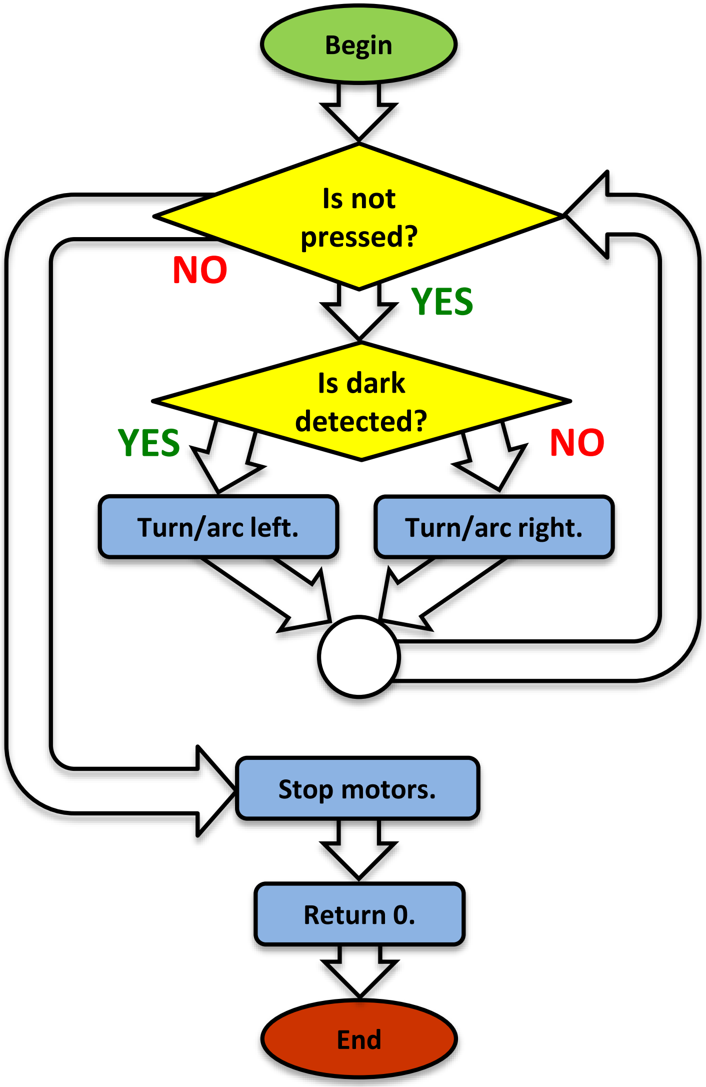
Starting with your DemoBot on one end of “JBC Mat 2” or using a piece of dark tape, have the robot travel along the path of the tape using the Top Hat sensor to determine the robot path (line following).

Activity 3 Video (possible solution)

Line-following

Activity

Analysis: Flowchart



Line-following with functions

Activity

Solution: (using two functions)

Pseudocode (Comments)

```
int main()
{
  // 1. Loop: Is not pressed?
  //    1.1. If: Is dark detected?
  //         1.1.1. Turn/arc left.
  //    1.2. Else:
  //         1.2.1. Turn/arc right.
  // 2. Stop motors.
  // 3. End the program.
}
```

Source Code

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

int main()
{
  while (right_button() == 0)
  {
    if (analog(0) > 512)
    {
      turn_left();
    }
    else
    {
      turn_right()
    }
  }

  ao();

  return 0;
}

void turn_left()
{
  motor(0, 20);
  motor(3, 100); // Turn/arc left.
}

void turn_right()
{
  motor(0, 100);
  motor(3, 20); // Turn/arc right.
}
```

Homework

Game review

Game strategy

Workshop survey

Homework for tonight:
game review

Visit http://homebase.kipr.org

Review the game rules on your 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 Forum**.
  - You should **regularly visit this forum**.
  - You will **find answers to the game questions** there.

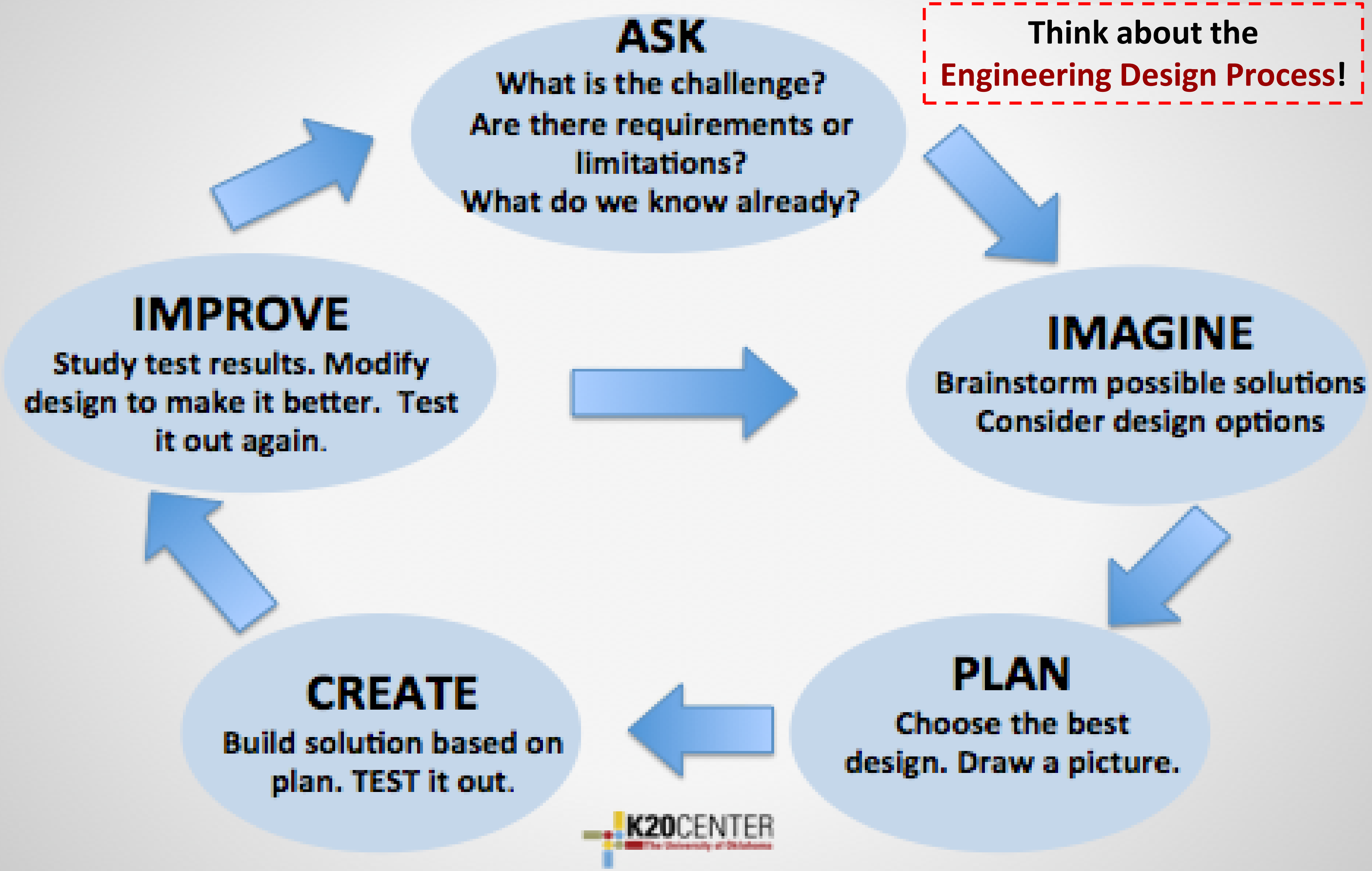
Homework for tonight:
game strategy

- 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 your strategy with your instructor tomorrow.

Resource

Homework for tonight: game strategy

Think about the Engineering Design Process!



ASK
What is the challenge?
Are there requirements or limitations?
What do we know already?

IMAGINE
Brainstorm possible solutions
Consider design options

PLAN
Choose the best design. Draw a picture.

CREATE
Build solution based on plan. TEST it out.

IMPROVE
Study test results. Modify design to make it better. Test it out again.

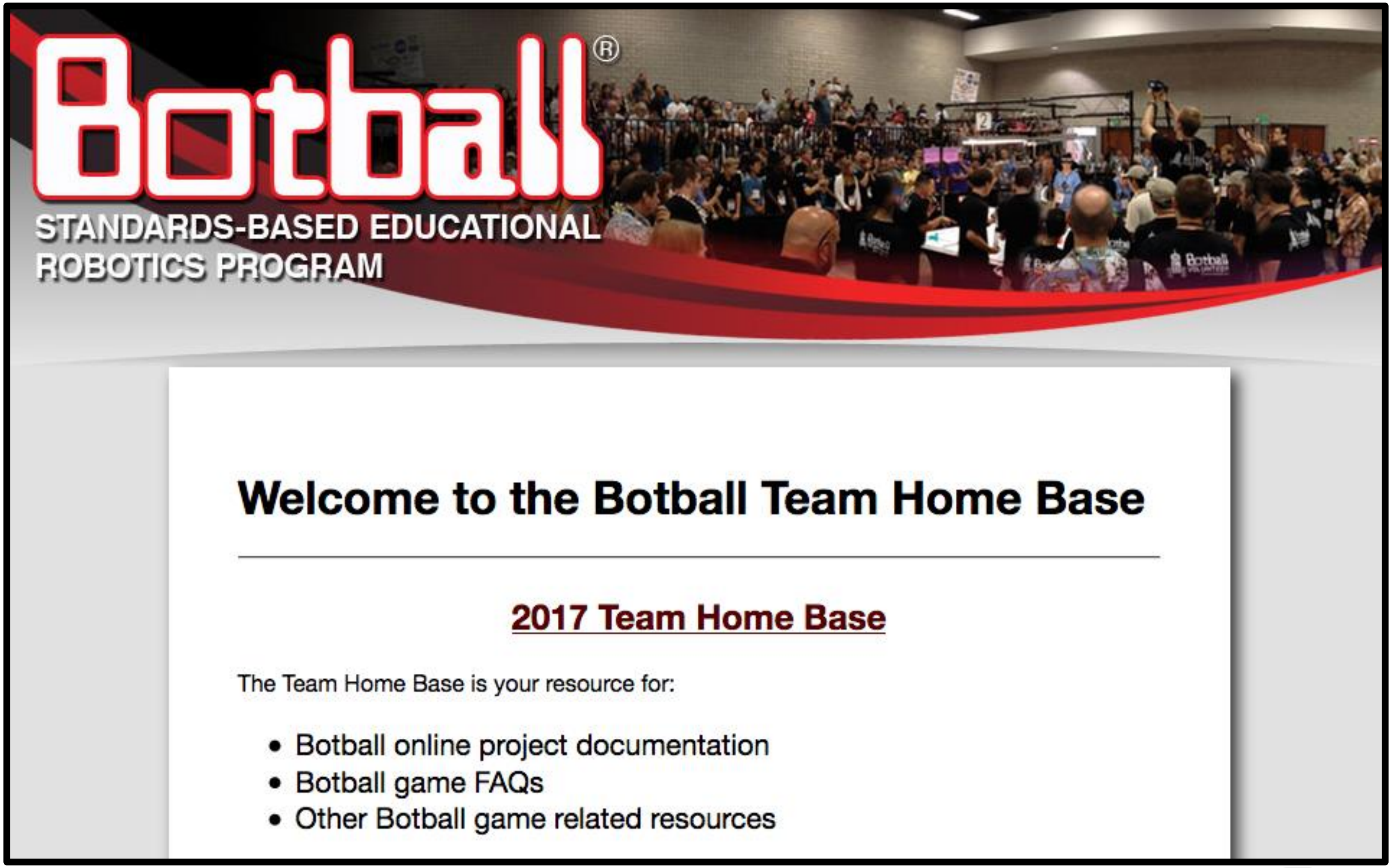
Homework for tonight:
workshop survey

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/LCYB7RY

Have a good night!

Visit http://homebase.kipr.org



Welcome to the Botball Team Home Base

2017 Team Home Base

The Team Home Base is your resource for:

- Botball online project documentation
- Botball game FAQs
- Other Botball game related resources

Welcome back!

Please take our survey to give feedback about the workshop:
https://www.surveymonkey.com/r/LCYB7RY

Botball 2017

Professional Development Workshop

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

v2017.01.06-2

Workshop Schedule

Day 1

- Botball Overview
- Getting started with the KISS IDE
- Explaining the “Hello, World!” C Program
- Designing Your Own Program
- Moving the DemoBot with Motors
- Fun with Functions
- Moving the DemoBot Servos
- Repetition, Repetition: Counting
- Making Smarter Robots with Sensors
- Repetition, Repetition: Reacting
- Making a Choice
- Line-following
- Homework

Day 2

- **Botball Game Review**
- **Motor Position Counter**
- **Measuring Distance**
- **Color Camera**
- **Moving the iRobot *Create*: Part 1**
- **Moving the iRobot *Create*: Part 2**
- **iRobot *Create* Sensors**
- **Logical Operators**
- **Resources and Support**

Botball Game Review

Game Q&A

Construction, documentation, and changes

Tournament template

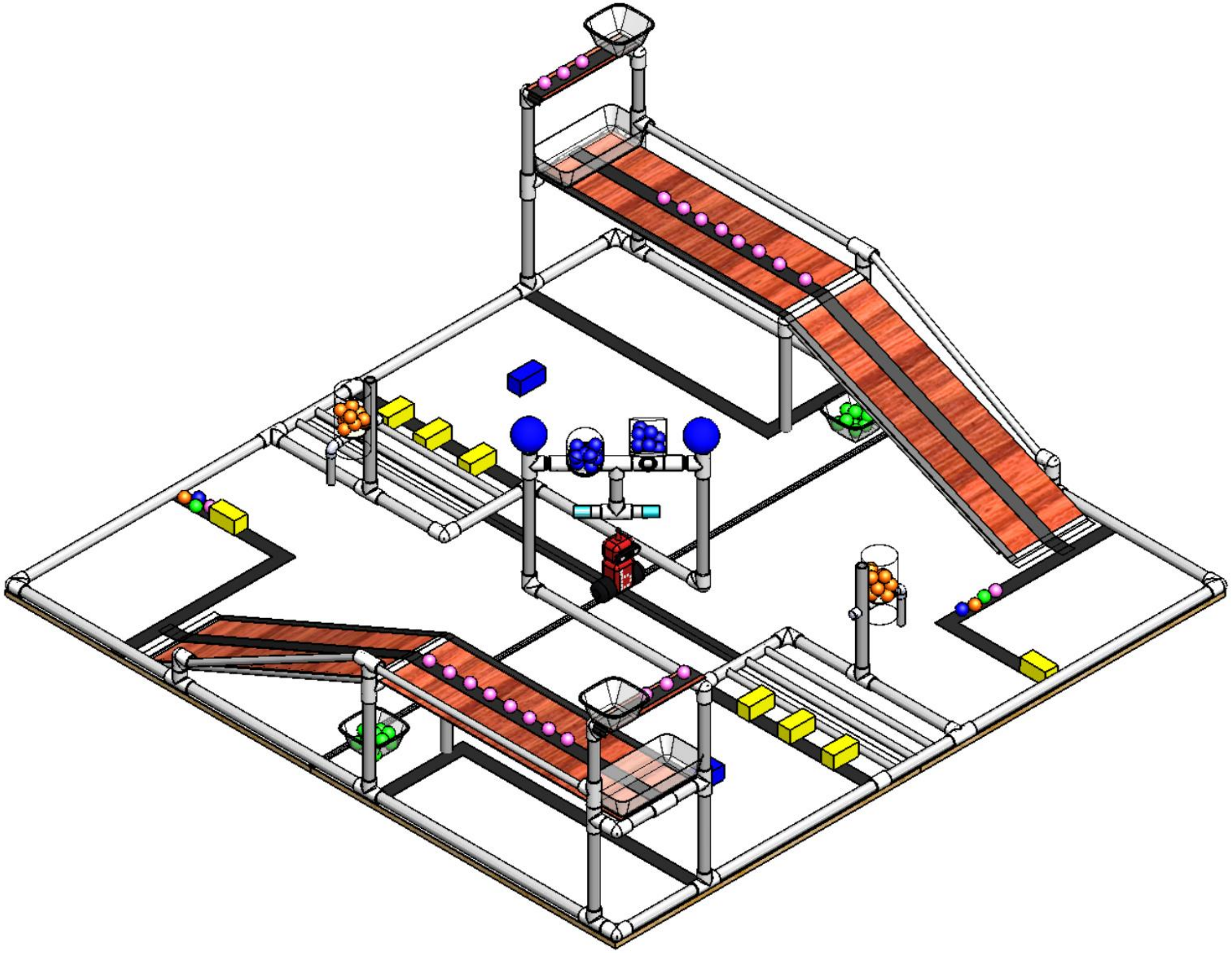
`shut_down_in()` function

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 elevated platform is 15-3/4", but at the tournament the platform could actually measure 15-7/8". If your arm is set for exactly 15-3/4", it would not work.
- 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 Handbook on the Team Home Base for more information (rubrics and exemplars).

Changes this season

- See the Team Homebase for a document covering all changes made in regards to Hardware, Rules, the Wallaby, Software, and Documentation.

Tournament templates

```
int main() // for your Create robot
{
  create_connect();
  wait_for_light(0); // change the port number to match the port you use
  shut_down_in(119); // shut off the motors and stop the robot after 119 seconds
  // Your code
  create_disconnect();
  return 0;
}
```

```
int main() // for not your Create robot
{
  wait_for_light(0); // change the port number to match the port you use
  shut_down_in(119); // shut off the motors and stop the robot after 119 seconds
  // Your code
  return 0;
}
```

Botball tournament functions

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

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

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

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

Running a Botball tournament program

Activity

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

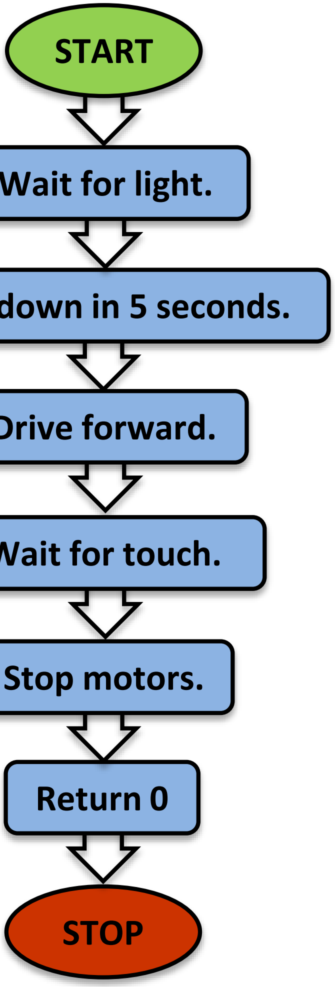
```
// 1. Wait for light.
// 2. Shut down in 5 seconds.
// 3. Drive forward.
// 4. Wait for touch.
// 5. Stop motors.
// 6. End the program.
```

Running a Botball tournament program

Activity

Analysis:

Flowchart



START

Wait for light.

Shut down in 5 seconds.

Drive forward.

Wait for touch.

Stop motors.

Return 0

STOP

Activity

Running a Botball tournament program

Solution:

Pseudocode (Comments)

```c
int main()
{
  // 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

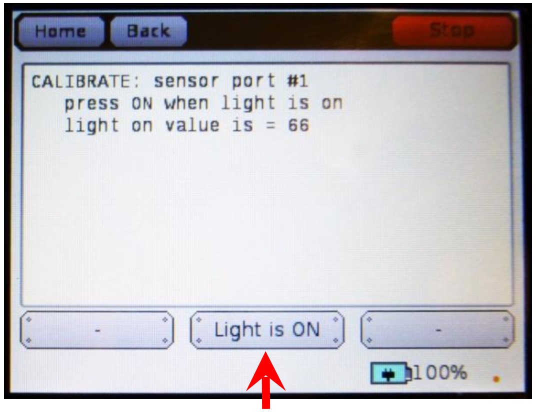
```c
int main()
{
  wait_for_light(0);

  shut_down_in(5);

  motor(0, 100);
  motor(3, 100);
  wait_for_touch(8);

  ao();

  return 0;
}
```

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

wait_for_light() calibration routine

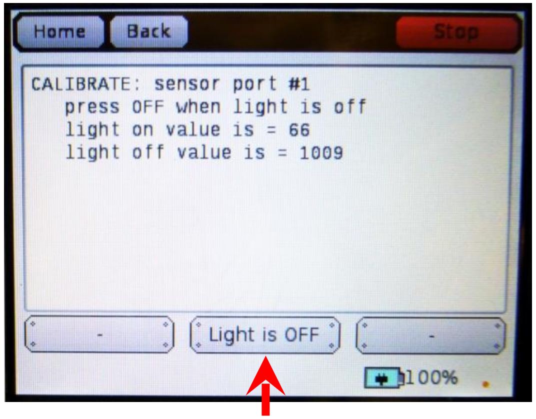
Activity

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



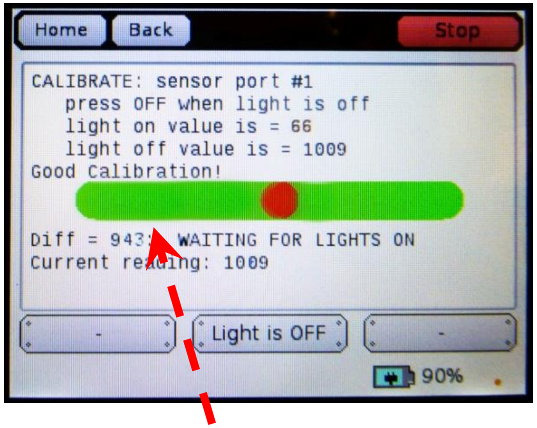
```
CALIBRATE: sensor port #1
   press ON when light is on
   light on value is = 66
```

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



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

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



```
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
```

You will get a "Good Calibration!" message and moving red dot on green bar when done *correctly*. You will get a "BAD CALIBRATION" message when <u>not</u> done correctly, and you will need to run through the routine again.

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

Running a Botball tournament program

Reflection:

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

Use these functions in your Botball tournament code!

Motor Position Counter

Motor position counter functions

Ticks and revolutions

Motor position counter

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

Motor Port #
(#0 – 3)

```
get_motor_position_counter(3)          — OR —          gmpc(3)
// Tells us the number of ticks the motor on port #3 has rotated.
// Note: “gmpc” is shorthand for “get_motor_position_counter”.
```

Motor Port #
(#0 – 3)

```
clear_motor_position_counter(3);       — OR —          cmpc(3);
// Resets the tick counter to 0 for the motor on port #3.
// Note: “cmpc” is shorthand for “clear_motor_position_counter”.
```

- The motor position is measured in “**ticks**”.
- Botball motors have ***approximately* 1400 ticks per *revolution***.
- Use **wheel circumference divided by 1400** to calculate distance!

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

Using motor position counter functions

How many revolutions will the motor rotate?

```
int main()
{
  clear_motor_position_counter(3);
  while (get_motor_position_counter(3) < 1400)
  {
    motor(3, 50);
  }
  ao();
  return 0;
}
```

Activity

Drive a specific number of revolutions

Description: Write a program for the KIPR Wallaby that drives the DemoBot forward for 10 *motor revolutions*, and then stops.

Analysis: What is the program supposed to do?

Pseudocode

```
1.  Reset motor position counters.
2.  Loop: Is counter < 14000?
    1.  Drive forward.
3.  Stop motors.
4.  End the program.
```

Comments

```
// 1. Reset motor position counters.
// 2. Loop: Is counter < 14000?
//     2.1. Drive forward.
// 3. Stop motors.
// 4. End the program.
```

Why 14000?

Drive a specific number of revolutions

Activity

Solution:

Comments

```
int main()
{
  // 1. Reset motor position counters.
  // 2. Loop: Is counter < 14000?
  //    2.1. Drive forward.
  // 3. Stop motors.
  // 4. End the program.
}
```

Source Code

```
int main()
{
  // 1. Reset motor position counters.
  clear_motor_position_counter(0);
  clear_motor_position_counter(3);

  // 2. Loop: Is counter < 14000?
  if (get_motor_position_counter(0) < 14000)
  {
    // 2.1 Drive forward.
    motor(0, 100);
    motor(3, 100);
  }

  ao(); // 3. Stop motors.

  return 0; //4. End the program.
}
```

Drive a specific number of revolutions

Activity

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

- How far did the robot travel? Was it always the same?
- How could you calculate an exact distance in millimeters to travel? (**Hint:** Use **wheel circumference divided by 1400** to calculate distance!)
- How could you modify your program to travel a specific distance in millimeters? (**Hint:** Consider writing a function with an argument for the distance.)
- How could you modify your program to accurately turn left or right?

Activity

Drive Straight!

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

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

Pseudocode

```
1.  Reset motor position counters.
2.  Loop: Is counter < 14000?
    1.  Move right motor at 75% power
    2.  Is left wheel behind right?
        1.  True: speed up left
            motor at 100%
        2.  False: slow down left
            motor at 50%
3.  Stop motors.
4.  End the program.
```

Comments

```
// 1. Reset motor position counts.
// 2. Loop: check right position.
// 2.1 power right motor at 75%
// 2.2 compare left to right counters
        // 2.2.1 slower: power left

        // 2.2.2 faster: power left

// 3. Stop motors.
// 4. End the program.
```

Drive Straight!

Activity

Solution:

Pseudocode (Comments)

```
int main()
{
  // 1. clear both motor counters.
  // 2. Loop: check right position
  //    2.1. power right motor at 75%.
  //    2.2. compare left to right counters.
  //      2.2.1. slower: left motor at 100%.
  //      2.1.2. faster: left motor at 50%.
  // 3. Stop motors.
  // 4. End the program.
}
```

Source Code

```
int main()
{
   // 1. clear both motor counters
   clear_motor_position_counter(0); // left motor
   cmpc(3); // right motor (shorter name)

   // 2. Loop: check right position.
   while(get_motor_position_counter(3) < 14000)
   {
     // 2.1 power right motor at 75%
     motor(3, 75);

     // 2.2 compare left to right counters
     if(gmpc(0) < gmpc(3))
     { // 2.2.1 slower: power left motor at 100%
       motor(0, 100);
     }
     else
     { // 2.2.2 faster: power left motor at 50%
       motor(0, 50);
     }
   }
   ao(); // 3. Stop motors.

   return 0; // 4. End the program.
}
```

Drive Straight

Activity

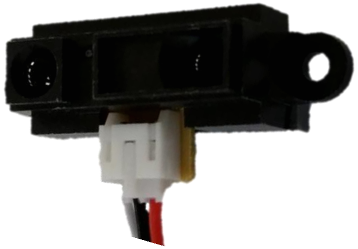
Reflection: What did you notice after you ran the program?

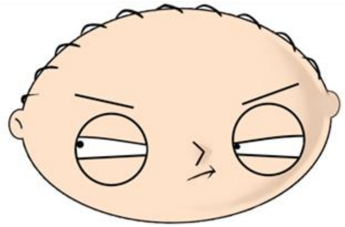
- Did the robot go straighter than in the previous program?
- How could you use this technique whenever you wanted to drive straight? (**Hint:** Consider writing a function with an argument for the distance.)
- How could you modify your program to go straight at different speeds?

Measuring Distance

Infrared “ET” distance sensor

Infrared “ET” distance sensor





For this activity, you will need the **infrared “ET” distance sensor**.

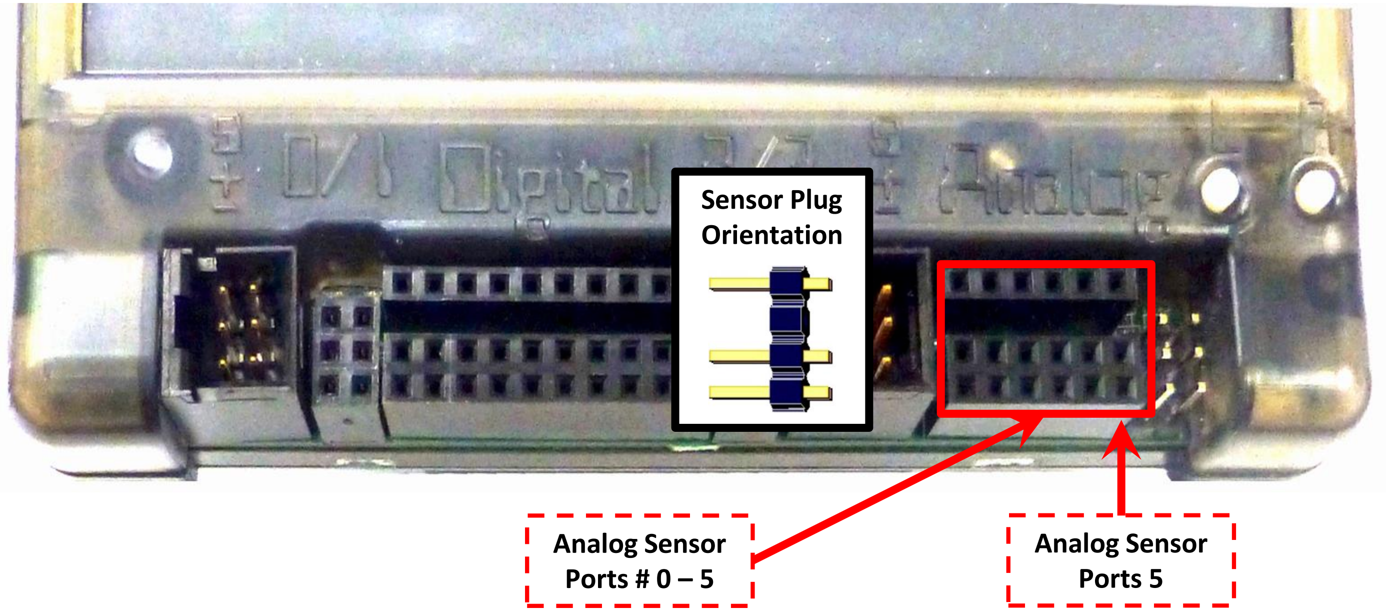
- There is both an infrared (IR) *emitter* and an IR *detector* inside of this sensor.
- The sensor works by sending out an IR beam and then measures the angle the reflected IR light returns at and triangulates the distance to an object.

This sensor makes a great medium-range distance sensor.

- Values are ***reliable*** between 5 cm and 80 cm.
- Values are ***not reliable*** beyond these distances, **though they <u>appear</u> to be**!
- Values are in **raw sensor units**, ***not*** in **centimeters**—**<u>but you can convert</u>**!
- Values ***decrease*** as an object gets ***farther away*** from the sensor.

Attach your ET distance sensor

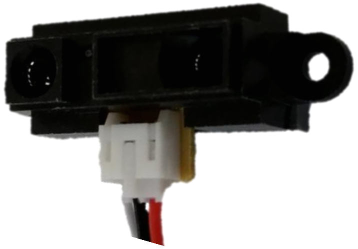
- Attach the sensor on the front of your robot so that it is **pointing forward**.
- The **ET distance sensor** is an **analog sensor**, so plug it into any of **analog sensor port #0 – 5**. For the purpose of this example, we will use analog port 5.
  - Recall that analog sensor values range **from 0 to 4095**.



Sensor Plug Orientation

Analog Sensor Ports # 0 – 5

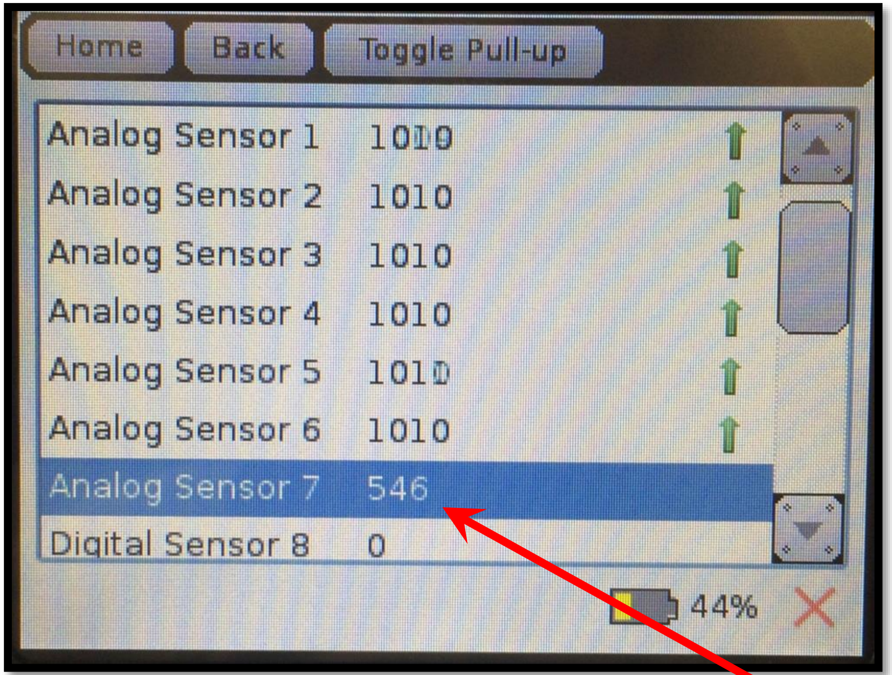
Analog Sensor Ports 5



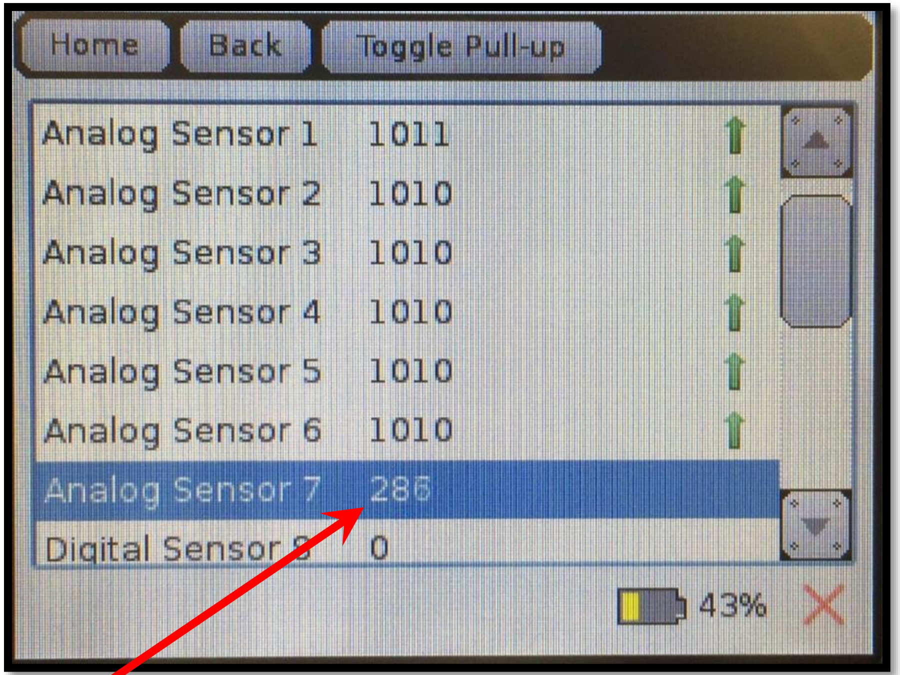
Reading ET distance sensor values from the Sensor List screen

Activity

Hold an object in front of the ET distance sensor at different distances and **read the value** on the **Sensor List** screen.



```
Analog Sensor 1    1010
Analog Sensor 2    1010
Analog Sensor 3    1010
Analog Sensor 4    1010
Analog Sensor 5    1010
Analog Sensor 6    1010
Analog Sensor 7    546
Digital Sensor 8   0
44%
```



```
Analog Sensor 1    1011
Analog Sensor 2    1010
Analog Sensor 3    1010
Analog Sensor 4    1010
Analog Sensor 5    1010
Analog Sensor 6    1010
Analog Sensor 7    286
Digital Sensor 8   0
43%
```

Higher **values →** ***closer*** **distances.**

3

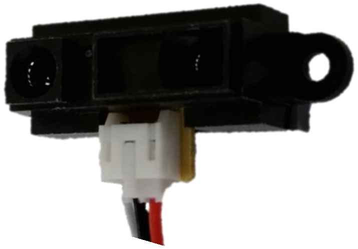
Lower **values →** ***farther*** **distances.**

ET distance grabbing

To get sensor readings from the **ET distance sensor**, you must call the analog function:

```
analog(5) // Get a reading from analog port #5.
```

- ***Lower* values → *farther* distances.**
- ***Higher* values → *closer* distances.**
- **Range:** 5 cm – 80 cm
  - Values are **not reliable** beyond these distances, though they **appear** to be!



Using analog for ET

What does this say?

```
int main()
{
  while (analog(5) < 475)
  {
    motor(0, 100);
    motor(3, 100);
  }
  ao();
  return 0;
}
```

Notice: no semicolon!
(Why not?)

Remember

Lower values → *farther* distances.

Higher values → *closer* distances.

Activity

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.

Analysis: What is the program supposed to do?

Pseudocode

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

Comments

```
// 1. Loop: Is not touched?
//    1.1. If: Is distance too far?
//      1.1.1. Drive forward.
//    1.2. Else.
//       1.2.1. If: Is distance too close?
//          1.2.1.1. Drive reverse.
                  //       1.2.2. Else.
//          1.2.2.1. Stop motors.
// 2. Stop motors.
// 3. End the program.
```

Activity

Maintain distance

Solution:

Comments

```
int main()
{
  // 1. Loop: Is not touched?
  //    1.1. If: Is distance to far?
  //         1.1.1. Drive forward.
  //    1.2. Else.
  //         1.2.1. If: Is distance too close?
  //                1.2.1.1. Drive reverse.
  //         1.2.2. Else.
  //                1.2.2.1. Stop motors.
  // 2. Stop motors.
  // 3. End the program.
}
```

Source Code

```
int main()
{
   while (digital(0) == 0)
  {
    // 1.1. Is distance too far?
    if (analog(5) < 475)
    {
      motor(0, 100);
      motor(3, 100);
    }
    else // sensor value is 475 or greater
    {
      // 1.2.1. If: Is distance too close?
      if (analog(5) > 525)
      {
        motor(0, -100);
        motor(3, -100);
      }
      else // sensor value is 475-525
      {
      ao();
      }
    }
  }

  ao();
  return 0;
}
```

Moving the iRobot *Create*: Part 1

Setting up the *Create*

The *Create* and the KIPR Wallaby

Create functions

Activity

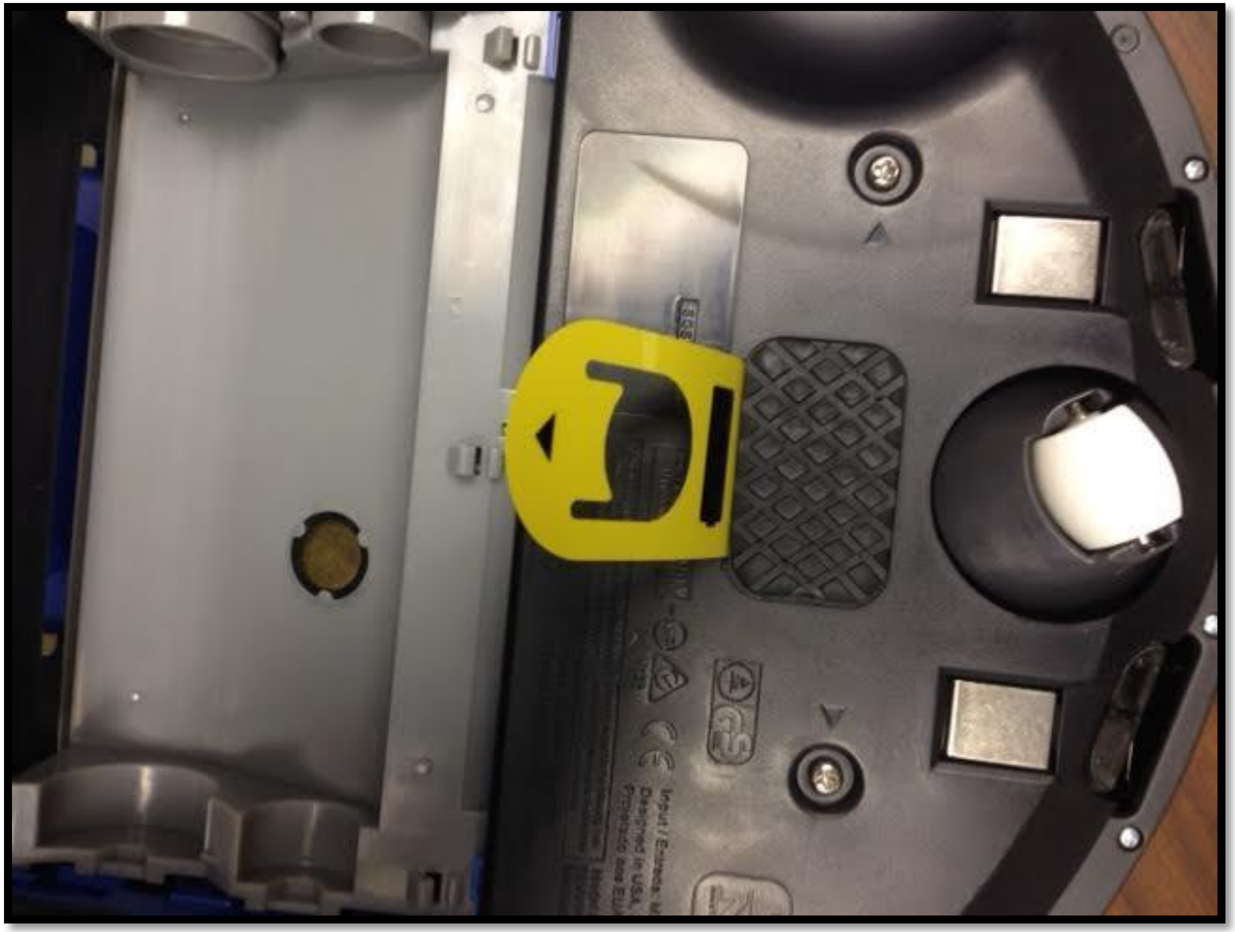
Charging the *Create*

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

- The **Create** 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.

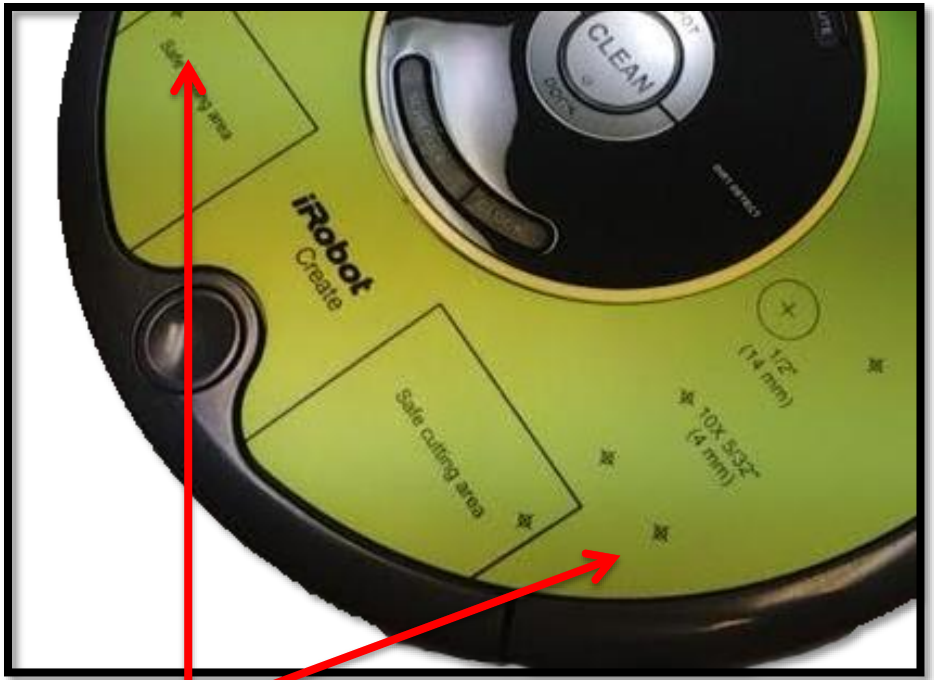


Create Underside

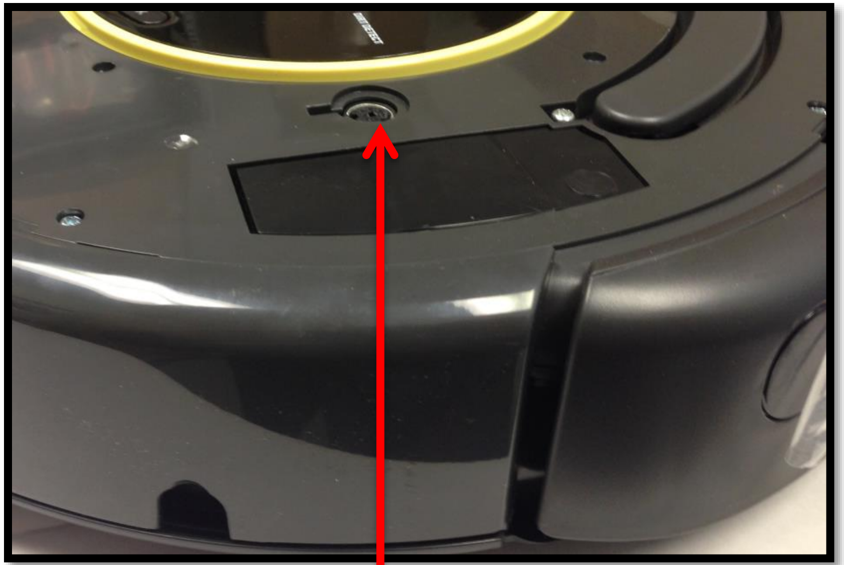
Activity

Uncovering and Charging the *Create*

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



Remove this



Serial Port

Mounting the Robotics Controller onto the *Create*

Activity

Build the Create DemoBot

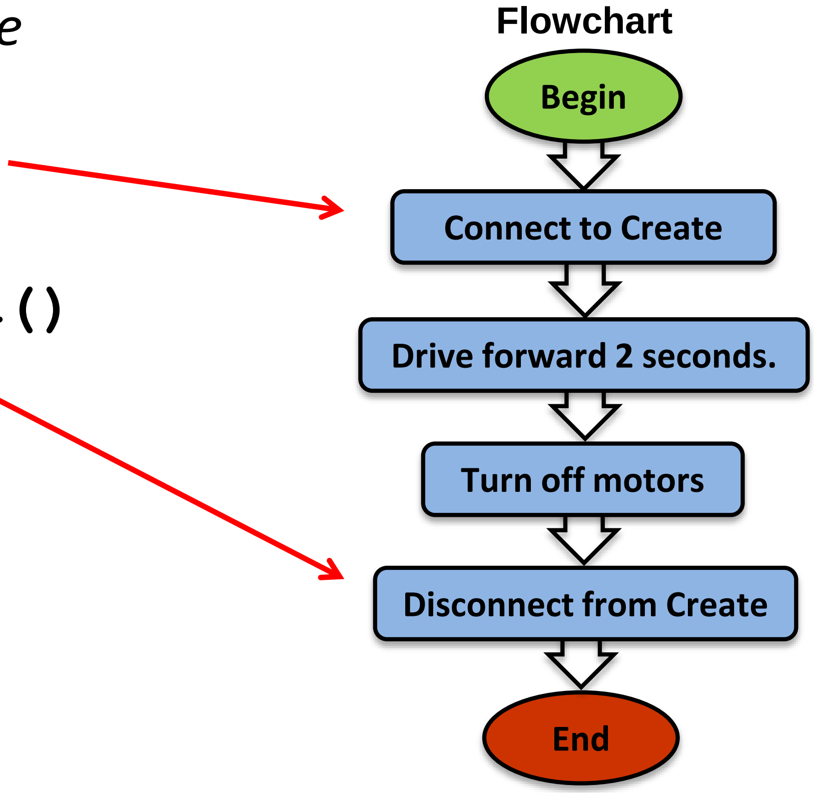
Create connect/disconnect functions

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

```
create_connect()
```

and *end* with

```
create_disconnect()
```



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.

Using *Create* functions

```
int main()
{
  create_connect();
  create_drive_direct(200, 200);
  wait_for_milliseconds(5000);
  create_stop();
  create_disconnect();
  return 0;
}
```

How far will the *Create* drive?

Activity

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

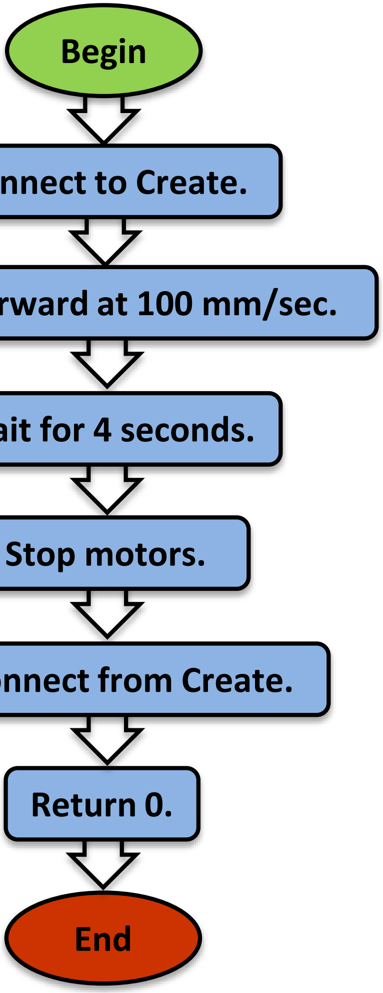
```
// 1. Connect to Create.
// 2. Drive forward at 100 mm/sec.
// 3. Wait for 4 seconds.
// 4. Stop motors.
// 5. Disconnect from Create.
// 6. End the program.
```

Moving the *Create*

Activity

Analysis:

Flowchart



Moving the *Create*

Activity

Solution:

Comments

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

  // 2. Drive forward at 100 mm/sec.
  create_drive_direct(100, 100);

  // 3. Wait for 4 seconds.
  wait_for_milliseconds(4000);

  // 4. Stop motors.
  create_stop();

  // 5. Disconnect from Create.
  create_disconnect();

  return 0;
}
```

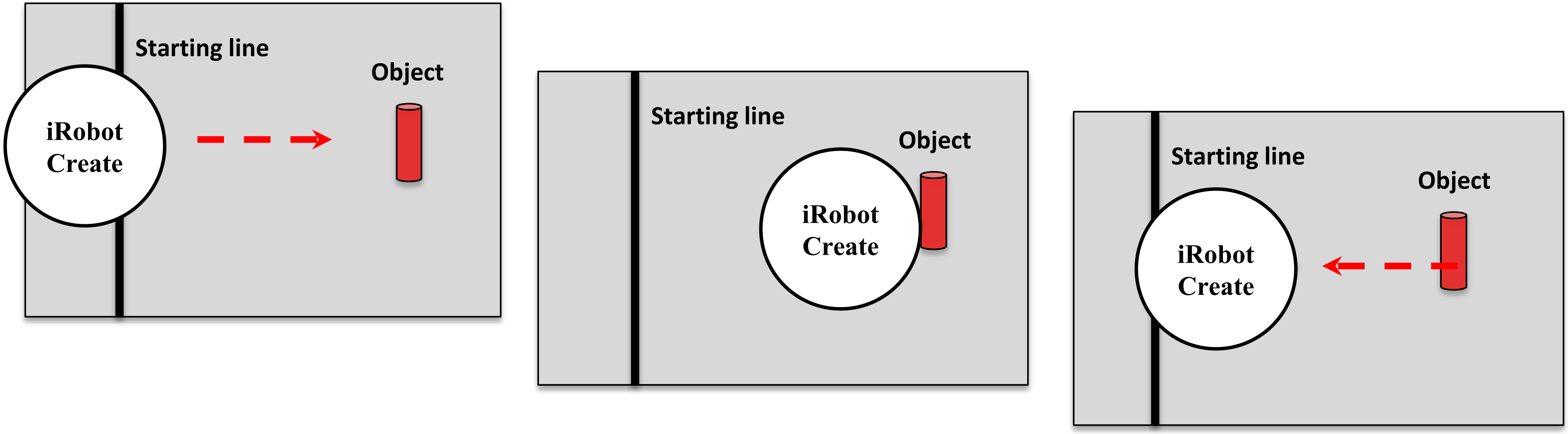
Execution: Compile and run your program on the KIPR Wallaby.

Touch an object and “go home”

Activity

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.



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* distance functions

What does this say?

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

Using *Create* angle functions

What does this say?

```
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;
}
```

Using *Create* angle functions

Positive angles are to the *left* (counter-clockwise).

```
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;
}
```

Using *Create* angle functions

Negative angles are to the *right* (clockwise).

```
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;
}
```

Notice:
the signs changed!

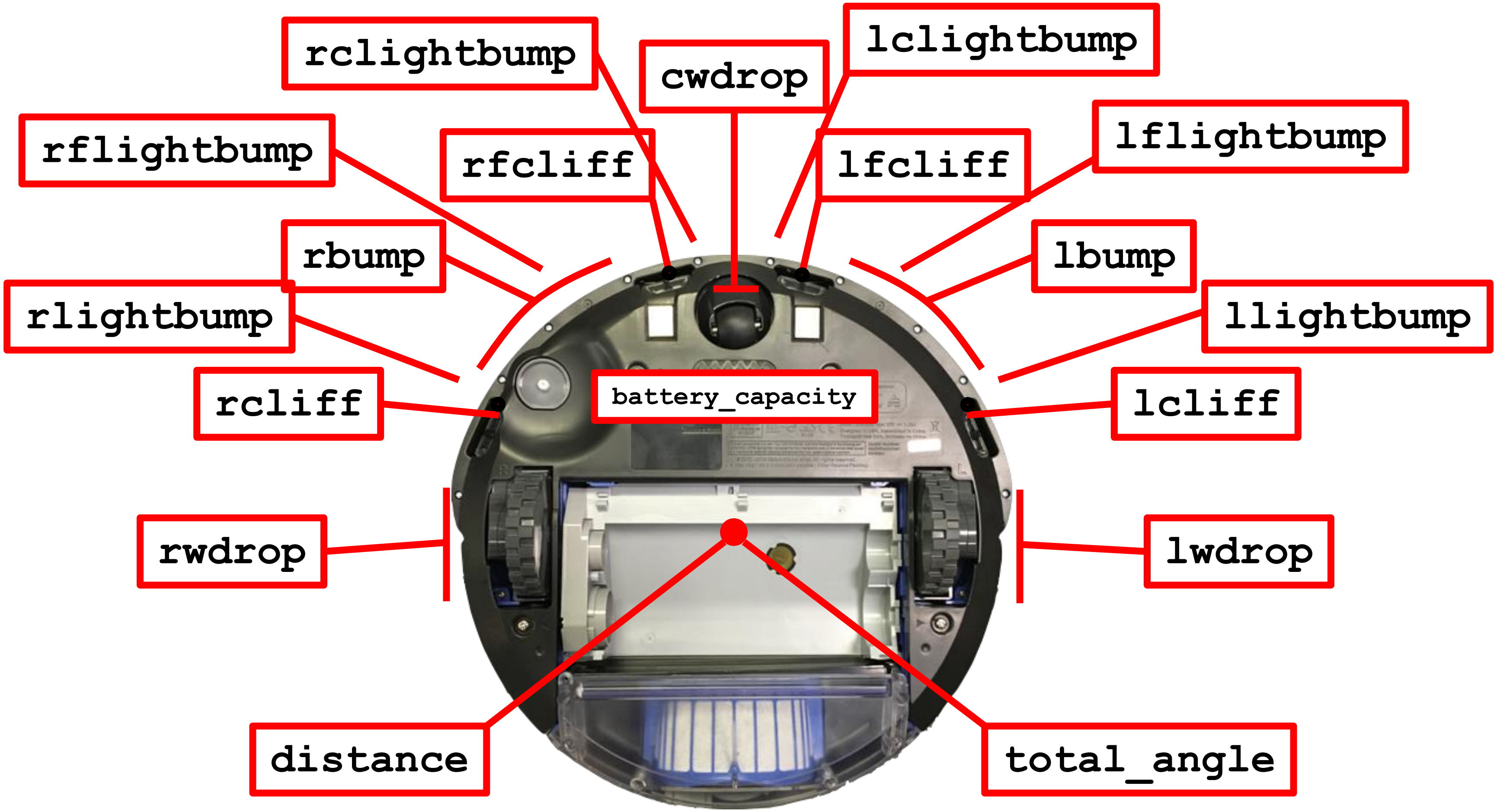
iRobot *Create* Sensors

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

Activity

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.
```

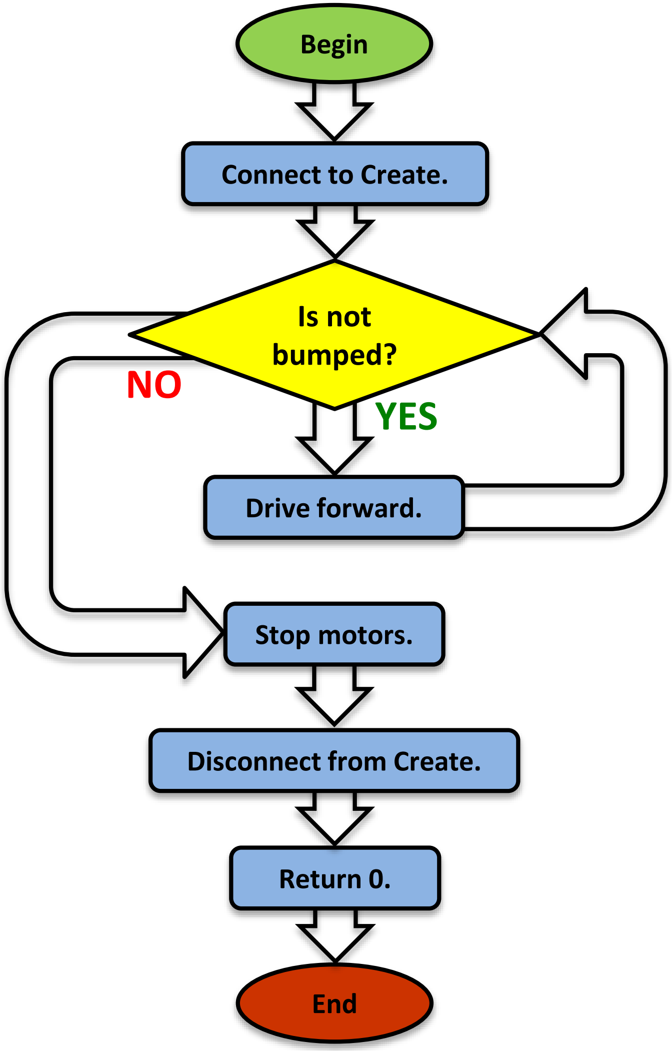
Comments

```
// 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

Activity

Analysis: Flowchart



Drive until bumped

Activity

Solution:

Comments

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

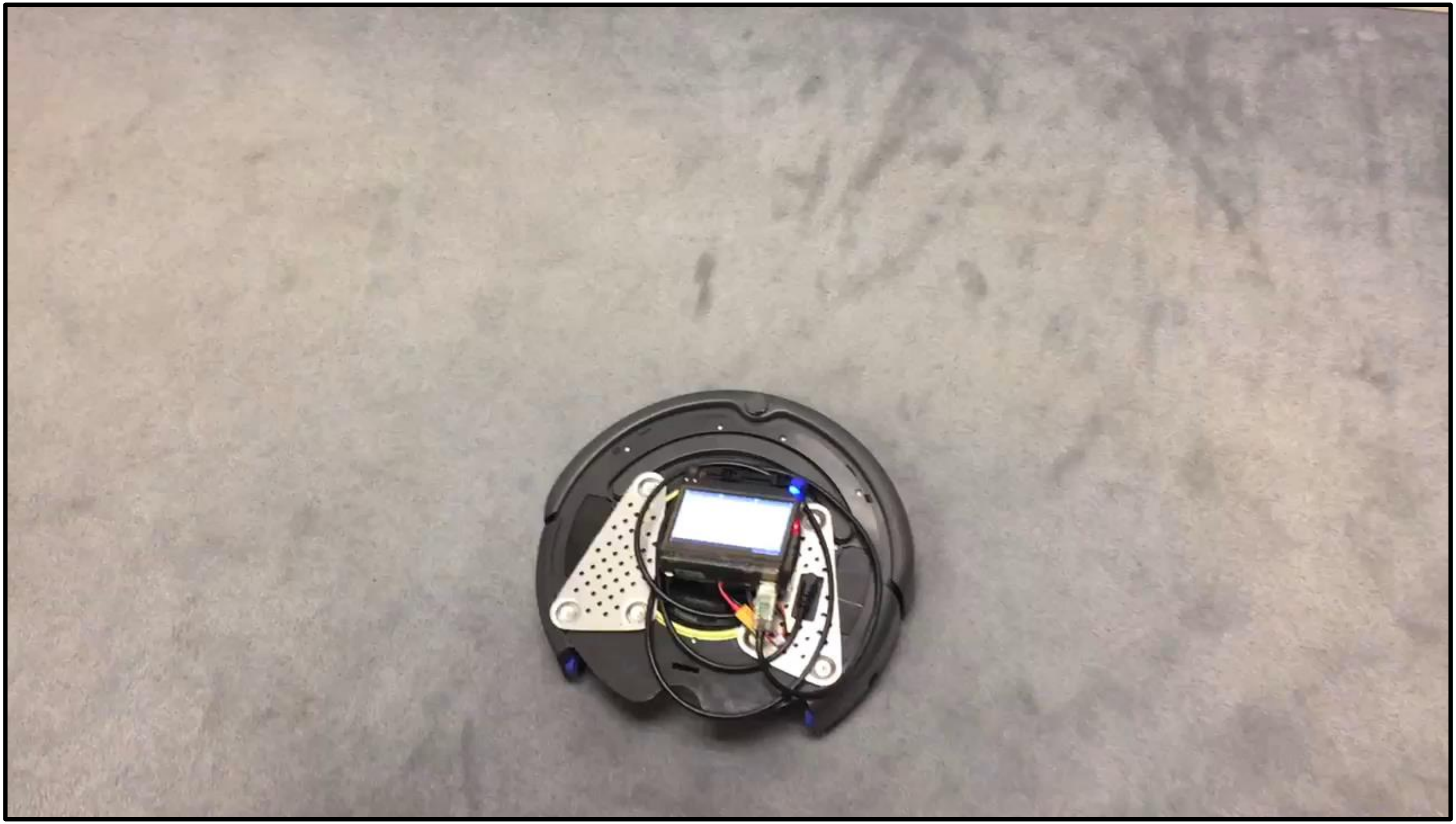
Source Code

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

  // 2. Loop: Is not bumped?
  while (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
```

Activity 4 (connections to the game)

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.

Activity 4 Video (possible solution)



LUNCH

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/LCYB7RY

Color Camera

Using the color camera

Setting the color tracking channels

About color tracking

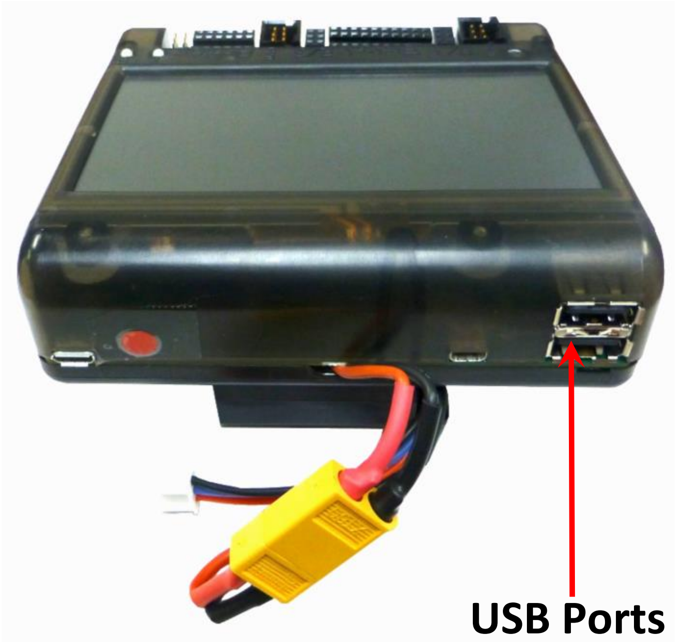
Camera functions

Activity

Color camera

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

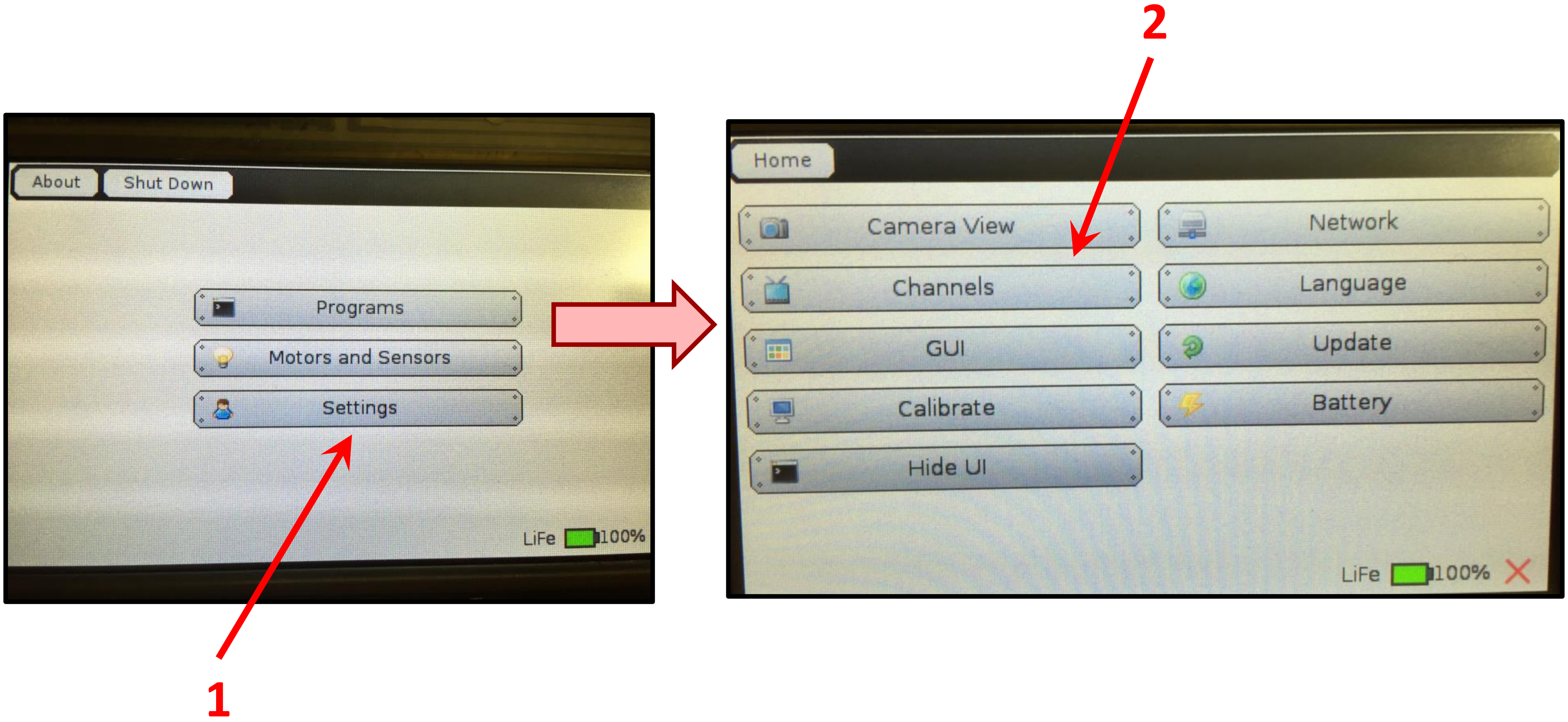
- The camera plugs into one of the USB (type A) ports on the back of the Wallaby.
- **Warning:** Unplugging the camera while it is being accessed can freeze the Wallaby, requiring it to be rebooted.



Setting the color tracking channels

Activity

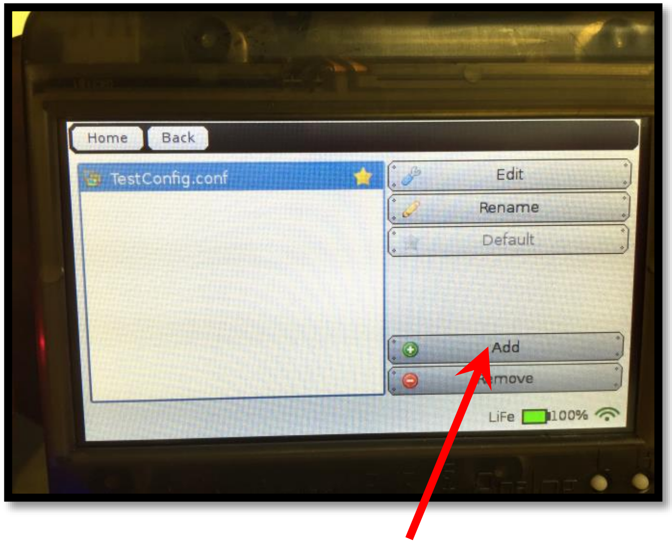
1. Select ***Settings***
2. Select ***Channels***

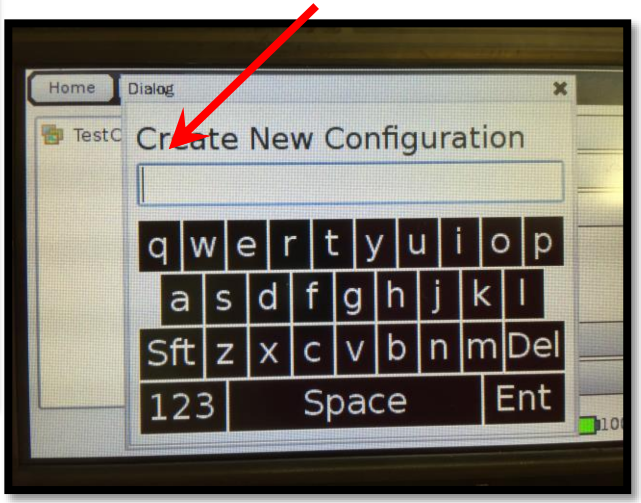


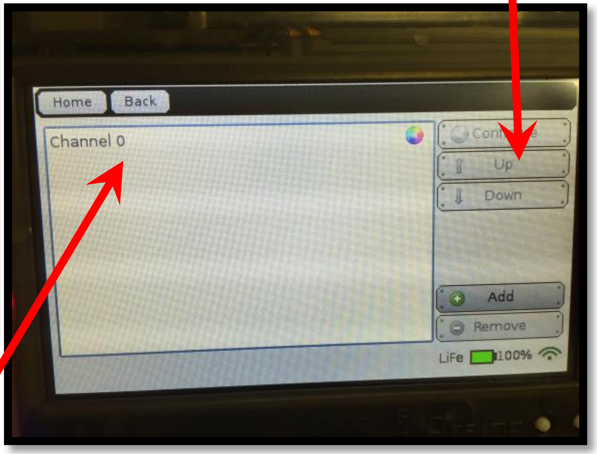
Setting the color tracking channels

Activity

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.





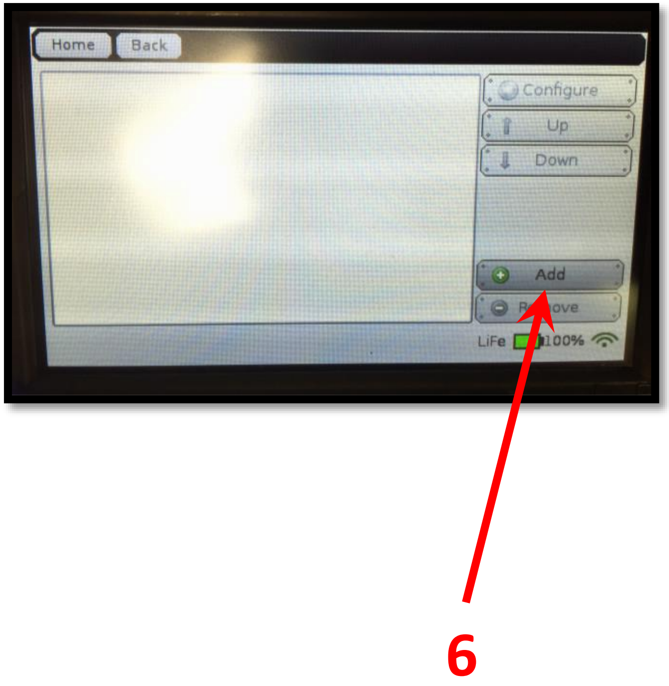


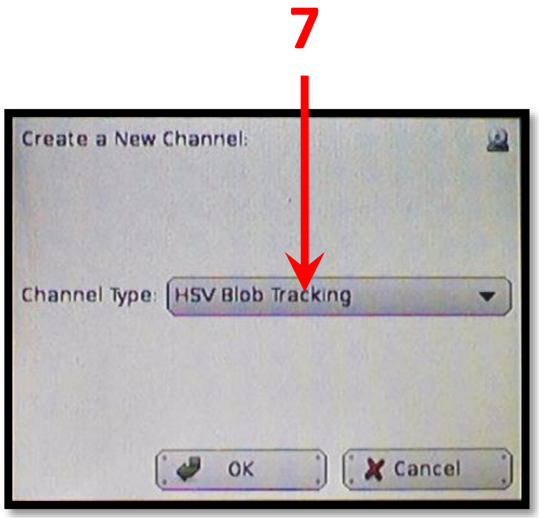
Note: if there is more than one configuration, select one, and press the "Default" button to make it be the one in use!

Setting the color tracking channels

Activity

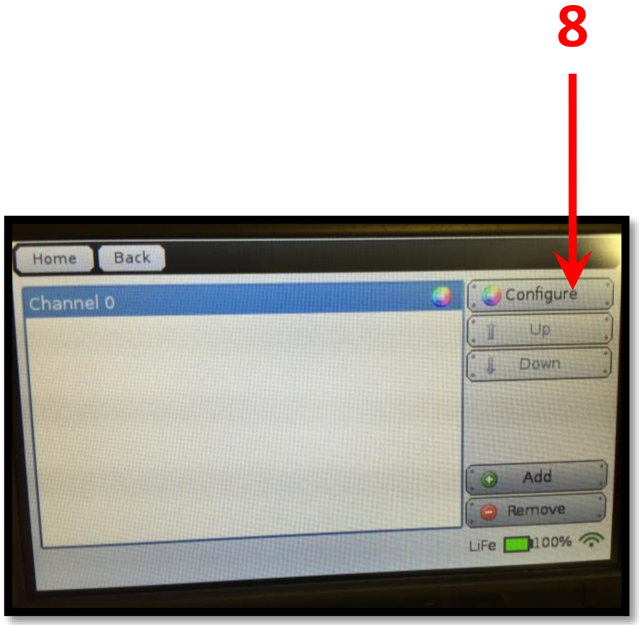
6. Press the *Add* button to add a channel to the configuration.
7. Select ***HSV Blob Tracking***, then *OK* to make this track color.
8. Highlight the channel, then press *Configure* to edit settings.
   - The first channel is 0 by default. You can have up to four: **0**, **1**, **2**, and **3**.





Create a New Channel:

Channel Type: HSV Blob Tracking

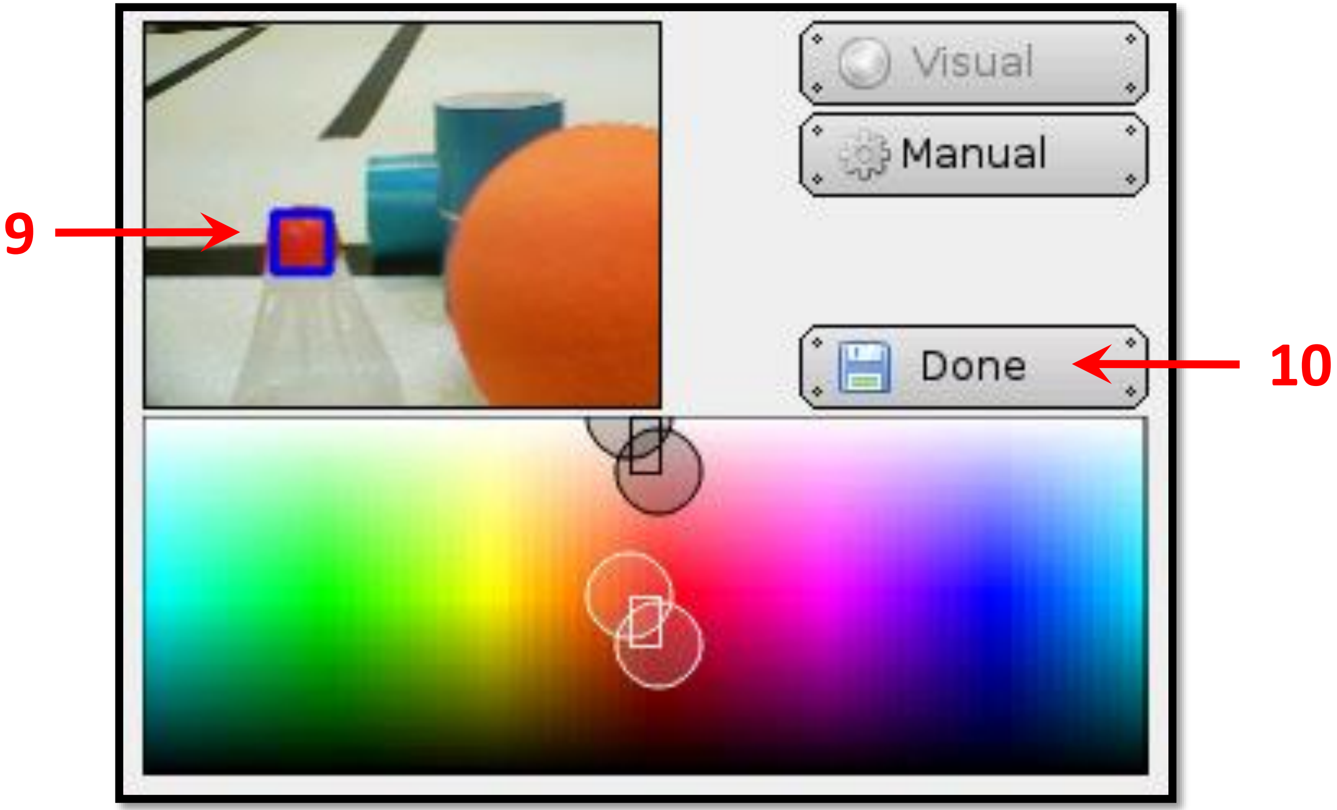


Channel 0

Setting the color tracking channels

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

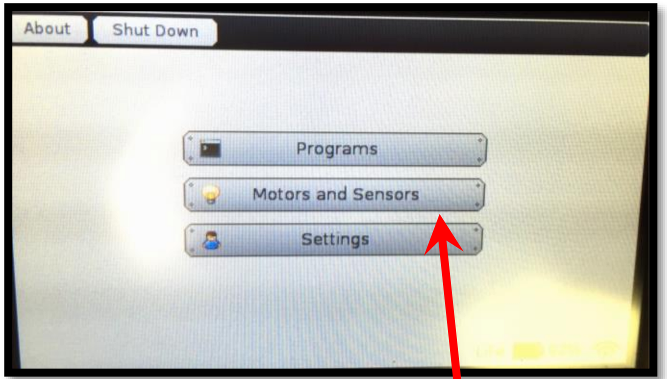
10. Press the *Done* button.

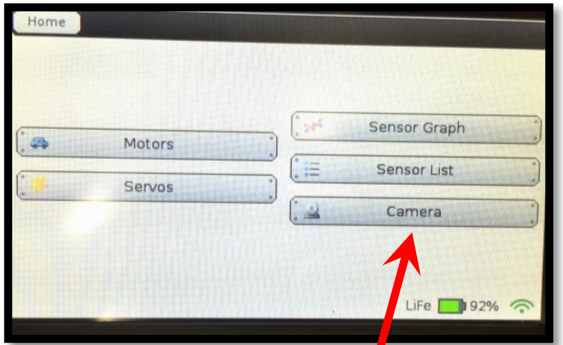


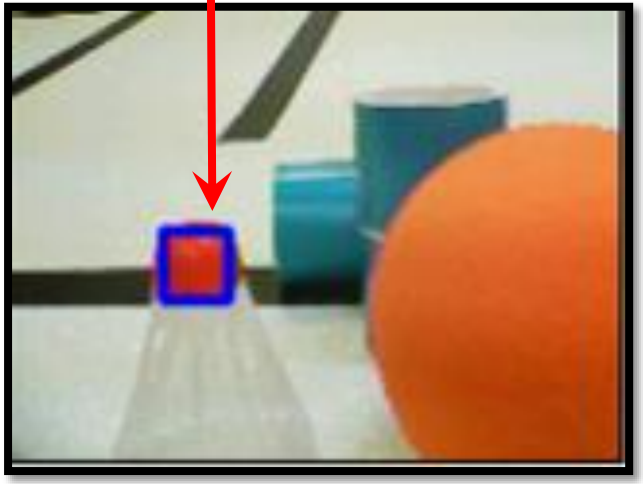
Activity

Verify the color channel is working

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

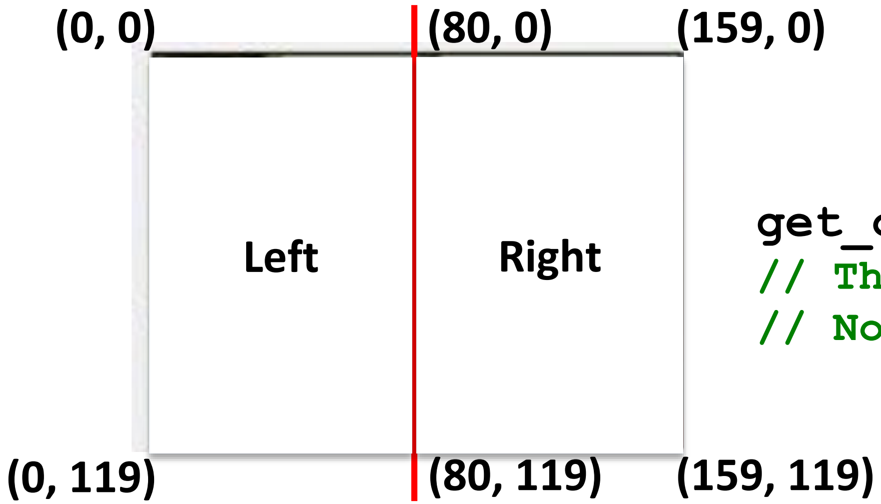






Tracking the location of an object

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



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

Channel #

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

Camera functions

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

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

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

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

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

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

Using camera functions

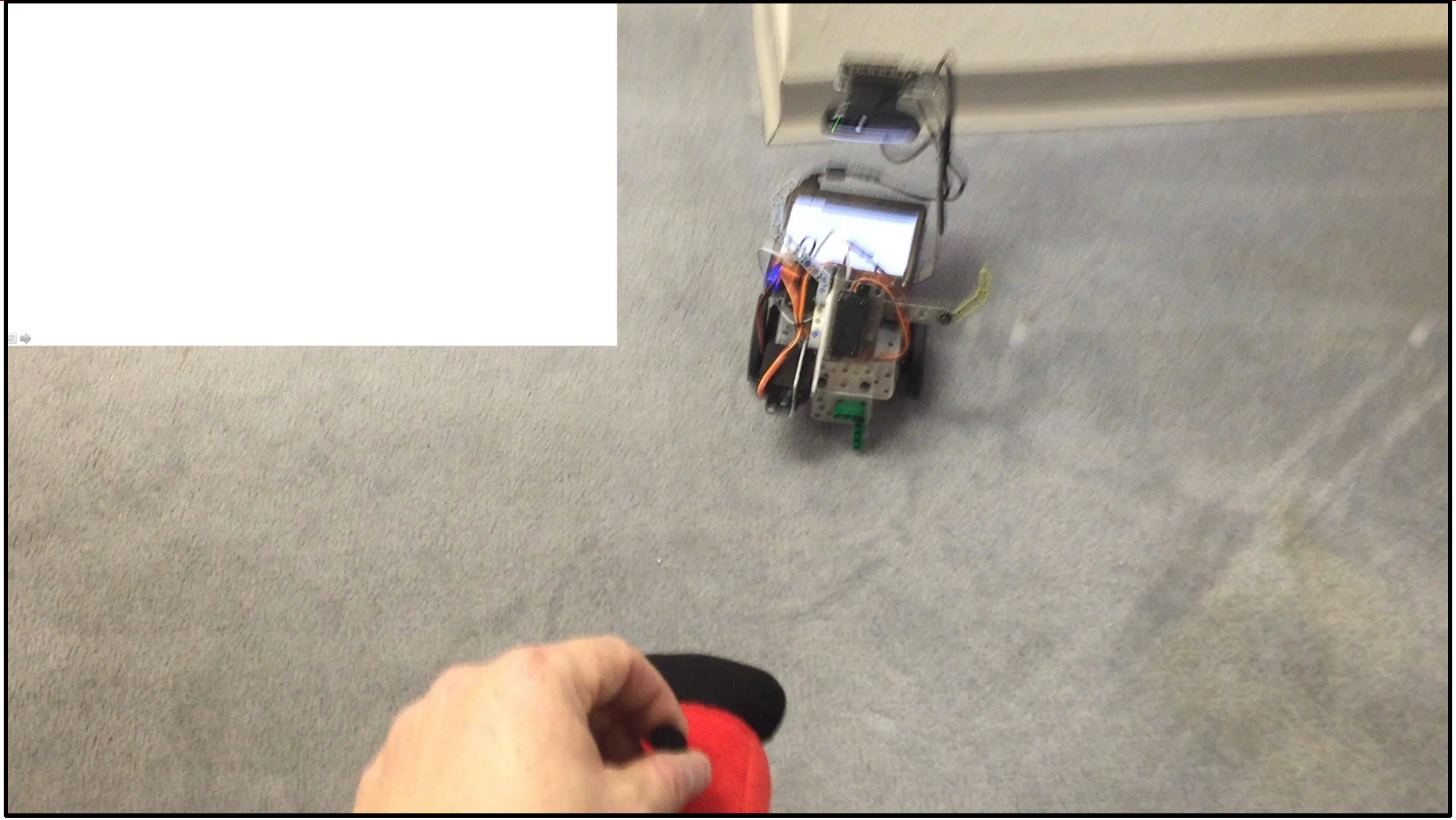
```c
int main()
{
  camera_open();
  while (digital(8) == 0)
  {
    camera_update();
    if (get_object_count(0) == 0)
    {
      printf("No objects detected.\n");
    }
    else
    {
      if (get_object_center_x(0, 0) < 80)
      {
        printf("Object is on the left!\n");
      }
      else
      {
        printf("Object is on the right!\n");
      }
    }
  }
  camera_close();
  return 0;
}
```

What do these say?

Activity 5 (connections to the game)

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.

Activity 5 Video (possible solution)



Professional Development Workshop
© 1993 – 2017 KIPR

Logical Operators

Multiple Boolean tests

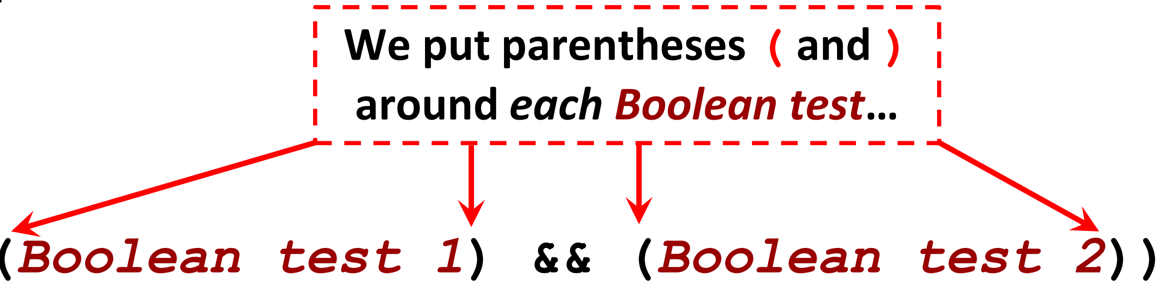
`while`, `if`, and Logical operators

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))
{
  // Code to execute ...
}
```

```
while ((digital(14) == 0) && (digital(15) == 0))
{
  // Code to repeat ...
}
```

```
if ((digital(12) == 1) || (digital(13) != 0))
{
  // Code to execute ...
}
```

```
if ((analog(3) < 512) || (digital(12) == 1))
{
  // Code to repeat ...
}
```

Using Logical operators

What does this say?

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

Drive for distance or until bumped

Activity

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

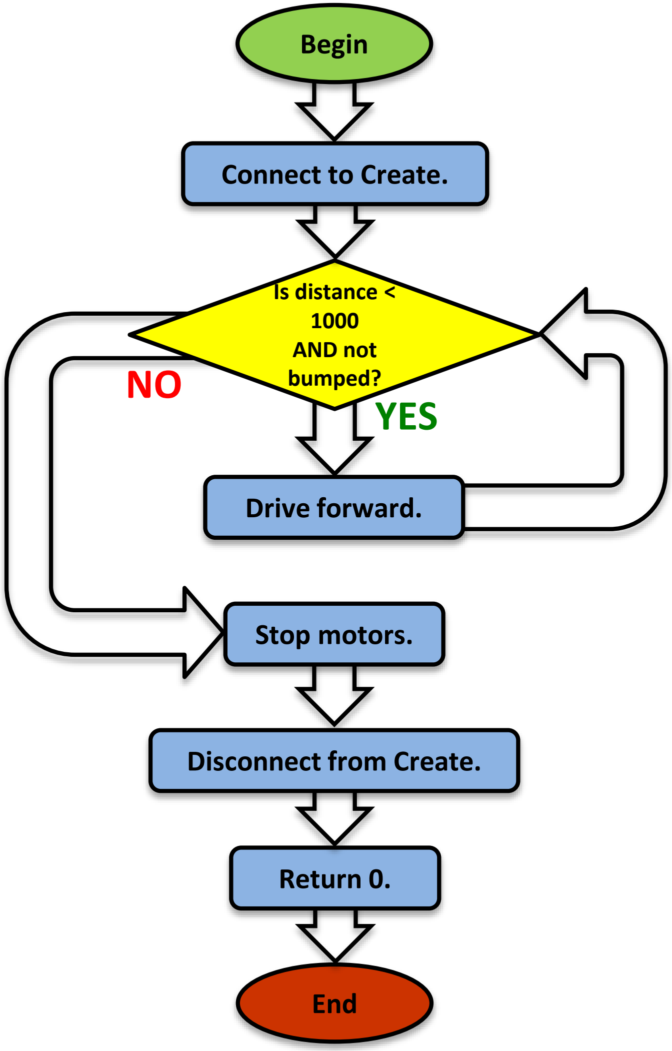
```
1. Connect to Create.
2. Loop: Is distance < 1000 AND not bumped?
    1. Drive forward.
3. Stop motors.
4. Disconnect from Create.
5. End the program.
```

Comments

```
// 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

Activity

Solution:

Pseudocode (Comments)

```
int main()
{
  // 1. Connect to Create.
  // 2. Loop: Is distance < 1000
  //          AND not bumped?
  //     2.1. Drive forward.
  // 3. Stop motors.
  // 4. Disconnect from Create.
  // 5. End the program.
}
```

Source Code

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

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

  // 3. Stop motors.
  create_stop();

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

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

Drive for distance or until bumped

Activity

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

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

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

Resources and Support

Team Home Base

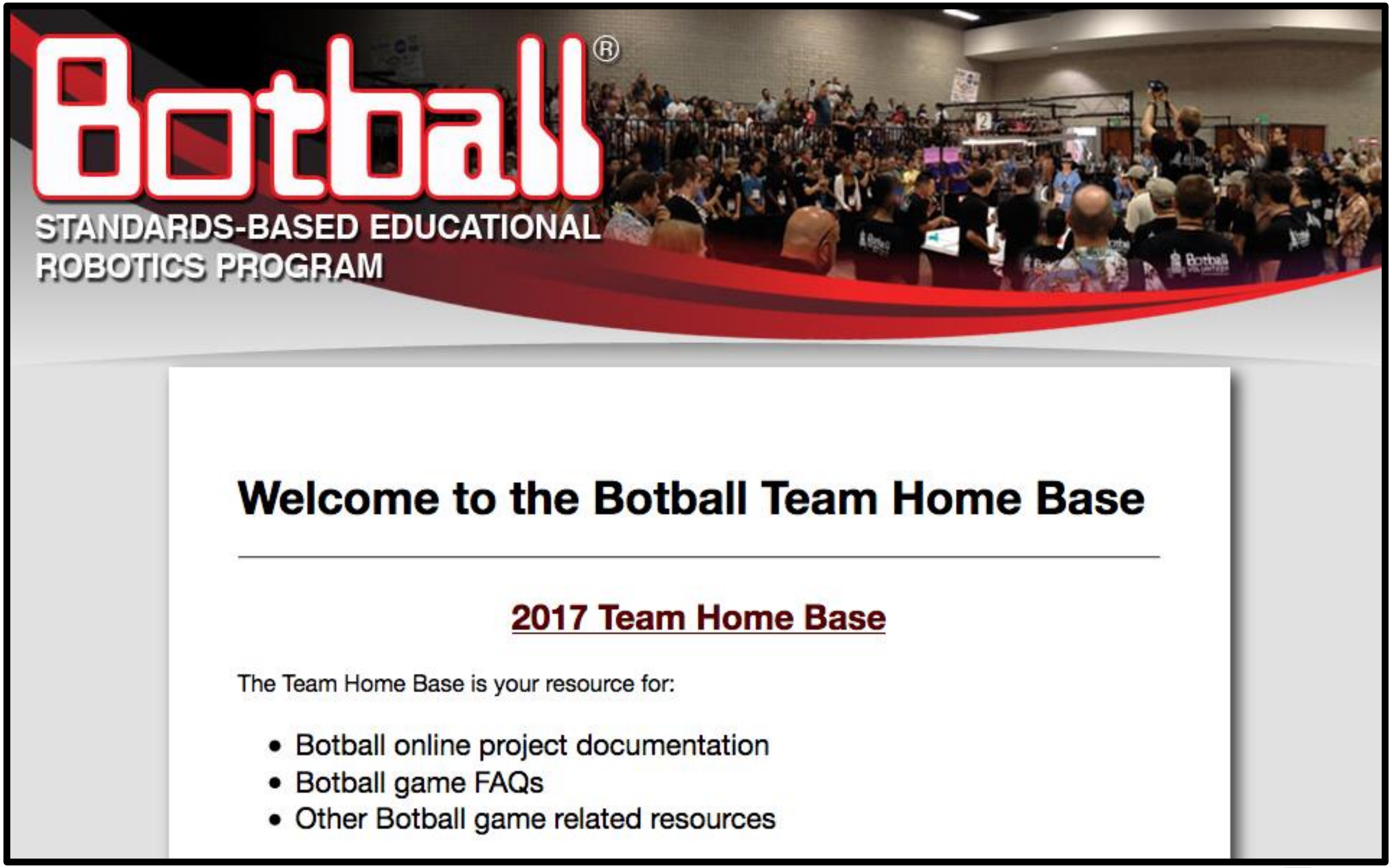
Remind, YAC, Community, PYR, and social media

T-shirts and awards

What to do after the workshop

Botball Team Home Base

Found at http://homebase.kipr.org



Welcome to the Botball Team Home Base

2017 Team Home Base

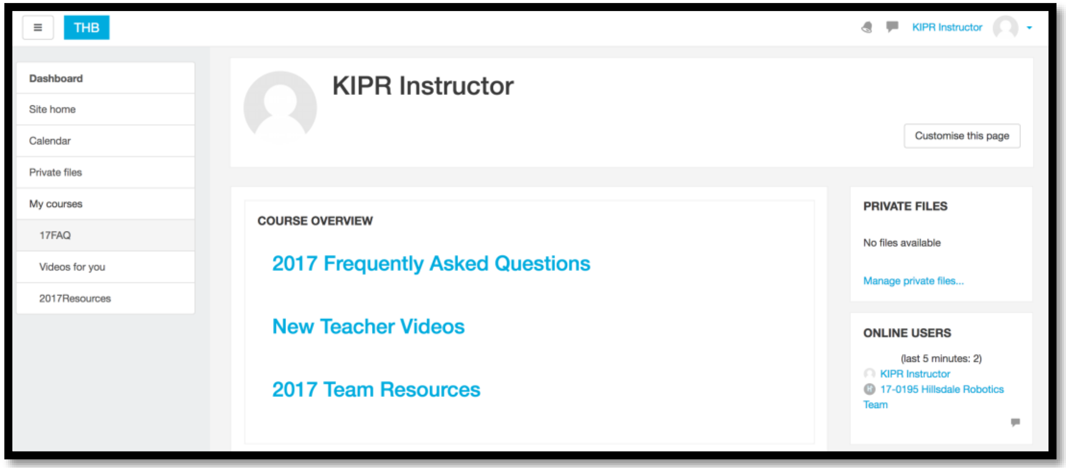
The Team Home Base is your resource for:

- Botball online project documentation
- Botball game FAQs
- Other Botball game related resources

Botball Team Home Base

KIPR Support

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



≡ THB

KIPR Instructor

Dashboard
Site home
Calendar
Private files
My courses
17FAQ
Videos for you
2017Resources

KIPR Instructor

Customise this page

COURSE OVERVIEW

2017 Frequently Asked Questions

New Teacher Videos

2017 Team Resources

PRIVATE FILES

No files available

Manage private files...

ONLINE USERS

(last 5 minutes: 2)
KIPR Instructor
17-0195 Hillsdale Robotics Team

Forum and FAQ

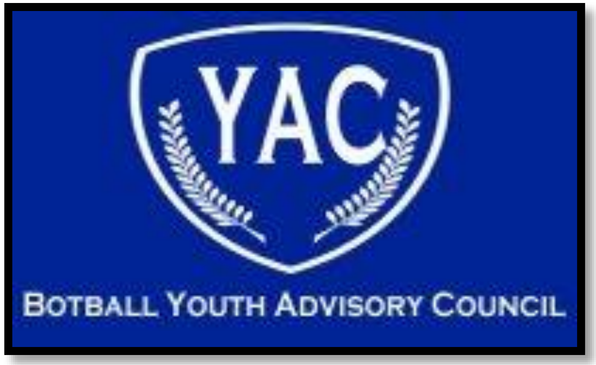
- **Site: http://homebase.kipr.org**
- **Content:**
  - Documentation Manual and Examples
  - Presentation Rubric & Example Presentation
  - DemoBot Build Instructions & Parts List
  - Controller Getting Started Manual
  - Construction Examples
  - Hints for New Teams
  - Sensor & Motor Manual
  - Game Table Construction Documents
  - All 2017 Game Documents

Botball Remind

https://www.remind.com/join

OK Regional: @393gf7

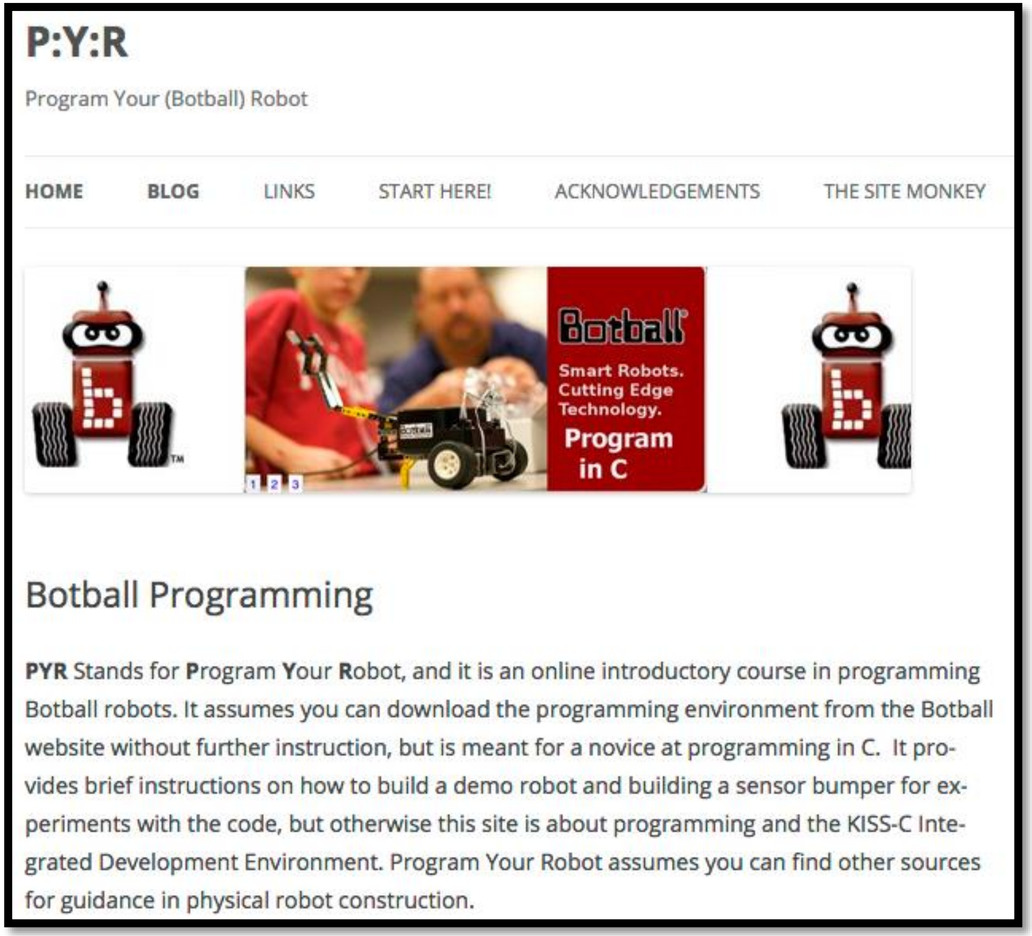
Botball Youth Advisory Council (YAC)



- We are a group of current and former Botballers who form Botball's student government.

- We work on many projects (e.g. blogs, forums, live-streaming), with one simple mission: keep making Botball better!

Program Your Robot (PYR)



P:Y:R

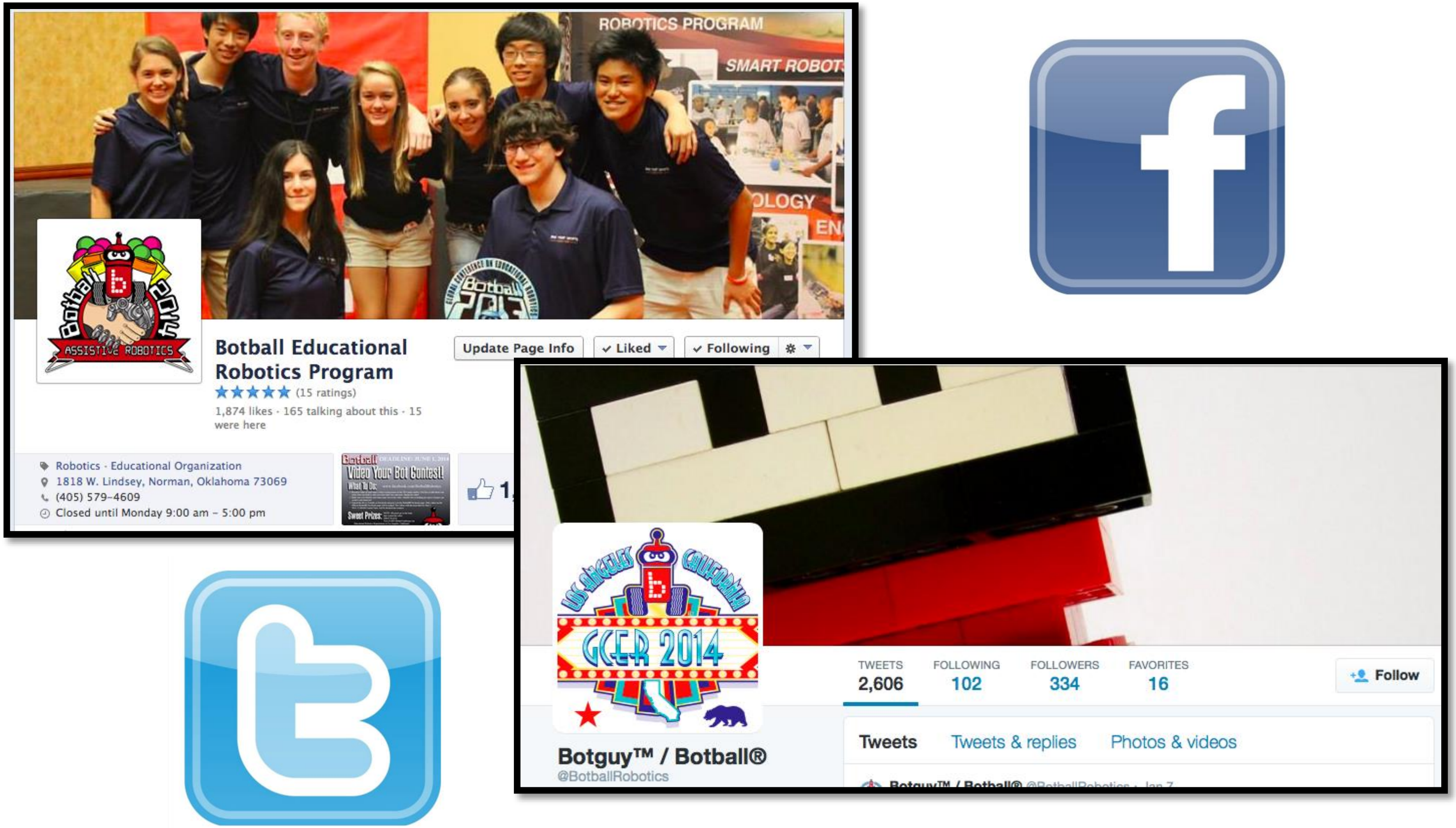
Program Your (Botball) Robot

Botball Programming

PYR Stands for **P**rogram **Y**our **R**obot, and it is an online introductory course in programming Botball robots. It assumes you can download the programming environment from the Botball website without further instruction, but is meant for a novice at programming in C. It provides brief instructions on how to build a demo robot and building a sensor bumper for experiments with the code, but otherwise this site is about programming and the KISS-C Integrated Development Environment. Program Your Robot assumes you can find other sources for guidance in physical robot construction.

http://botballprogramming.org

Social media



Botball Educational Robotics Program

Update Page Info · ✓ Liked · ✓ Following

★★★★★ (15 ratings)

1,874 likes · 165 talking about this · 15 were here

Robotics · Educational Organization
1818 W. Lindsey, Norman, Oklahoma 73069
(405) 579-4609
Closed until Monday 9:00 am – 5:00 pm

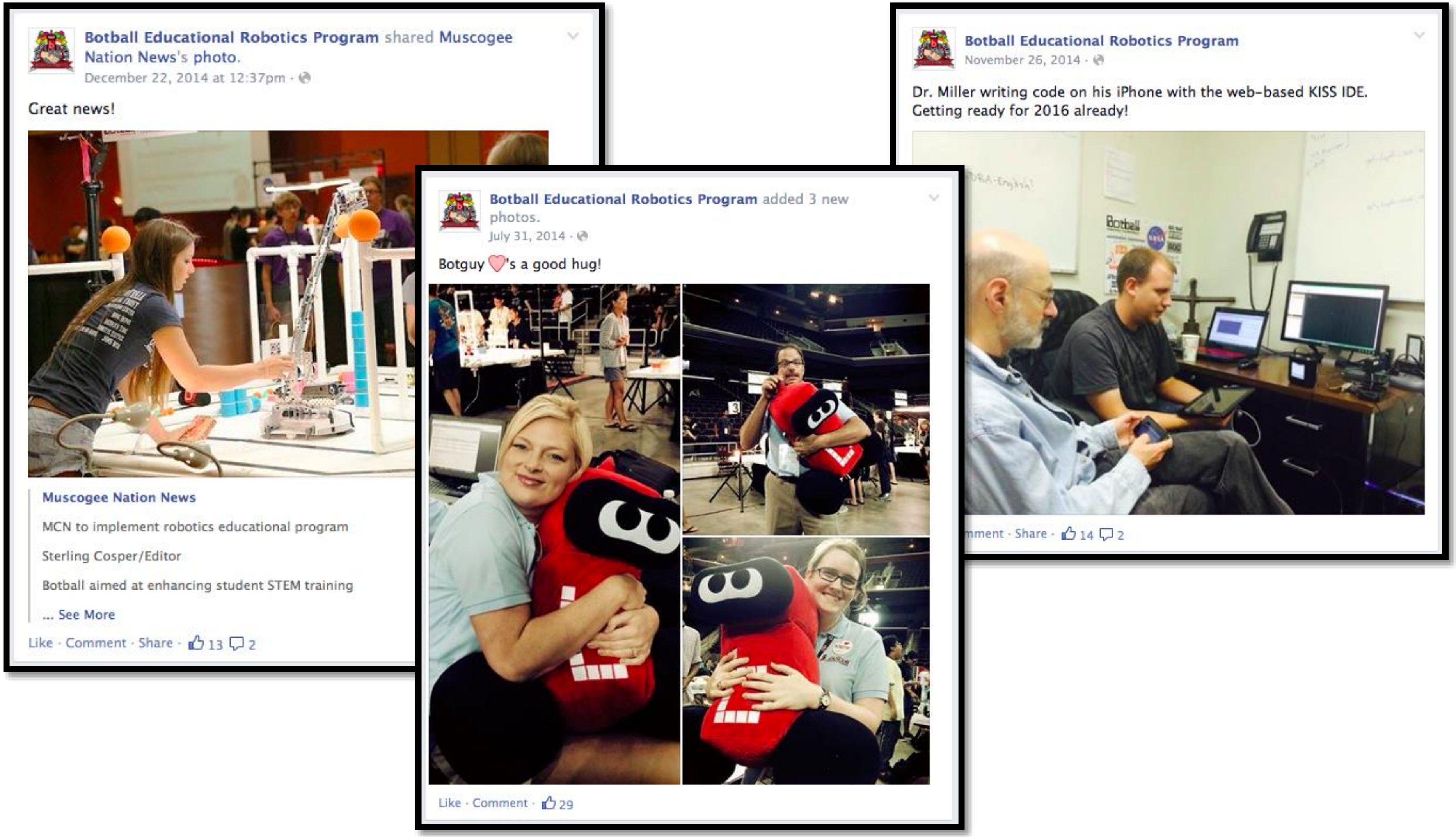
Botguy™ / Botball®
@BotballRobotics

TWEETS	FOLLOWING	FOLLOWERS	FAVORITES
2,606	102	334	16

Follow

Tweets · Tweets & replies · Photos & videos

Social media



Botball Educational Robotics Program shared Muscogee Nation News's photo.
December 22, 2014 at 12:37pm ·

Great news!

Muscogee Nation News

MCN to implement robotics educational program

Sterling Cosper/Editor

Botball aimed at enhancing student STEM training

... See More

Like · Comment · Share · 👍 13 💬 2

Botball Educational Robotics Program added 3 new photos.
July 31, 2014 ·

Botguy ♡'s a good hug!

Like · Comment · 👍 29

Botball Educational Robotics Program
November 26, 2014 ·

Dr. Miller writing code on his iPhone with the web-based KISS IDE. Getting ready for 2016 already!

mment · Share · 👍 14 💬 2

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

2. Hit the ground running.

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

What to do after the workshop

3. Plan out the season.

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

What to do after the workshop

4. Build the game board.

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

4. Organize your Botball kit.

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

4. Understand the game.

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

Thanks and have a great season!

```
}   // end workshop
```

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/r/LCYB7RY