Welcome to Botball 2016!

Before we get started…

1. **Sign in**, and collect your materials and electronics.
2. KIPR staff will come around and **install software KIPR Robotics Controller - Wallaby** (Wallaby - the robot's "brain") as needed.
3. **Charge your KRC Wallaby batteries**





KIPR Robotics
Controller - Wallaby



4. Open the "**2016 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.**
5. Build the **DemoBot**.

Raise your hand if you need help or have questions.

© 1993 – 2016 KIPR

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 to the battery from using the wrong charger or from too deep a discharge!**



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

© 1993 – 2016 KIPR

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.

When your Wallaby is not in use please **be sure to do the following**:

- TURN YOUR WALLABY OFF
- UNPLUG THE BATTERY FROM THE WALLABY
  - Leaving your battery plugged in and your Wallaby turned on 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. If this happens you can call the KIPR office and purchase a replacement - 405-579-4609

© 1993 – 2016 KIPR

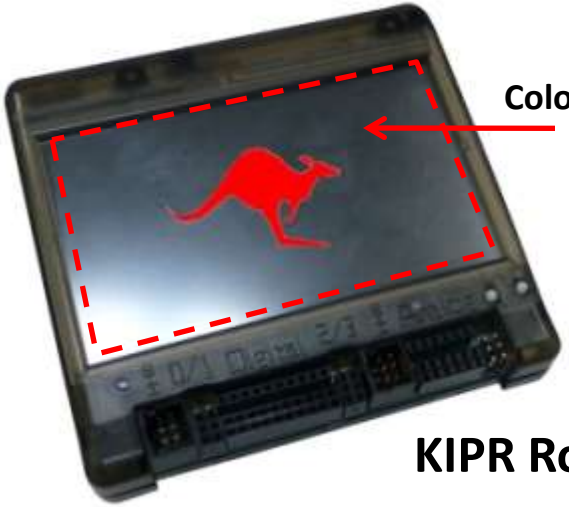
Wallaby Power Down

- 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”
- Use a stylus/toothpick to slide the power switch to off
- Unplug the battery being careful not to pull on the wires

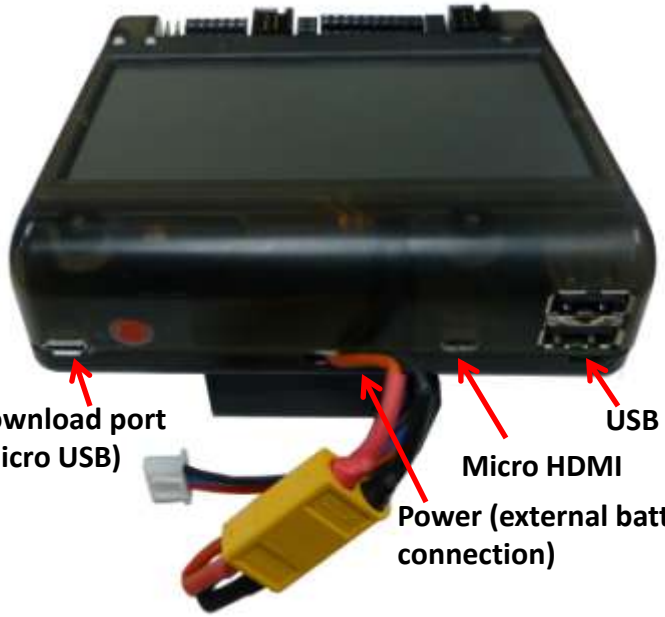
© 1993 – 2016 KIPR

KRC Wallaby Controller Guide

Resource



KIPR Robotics Controller Wallaby







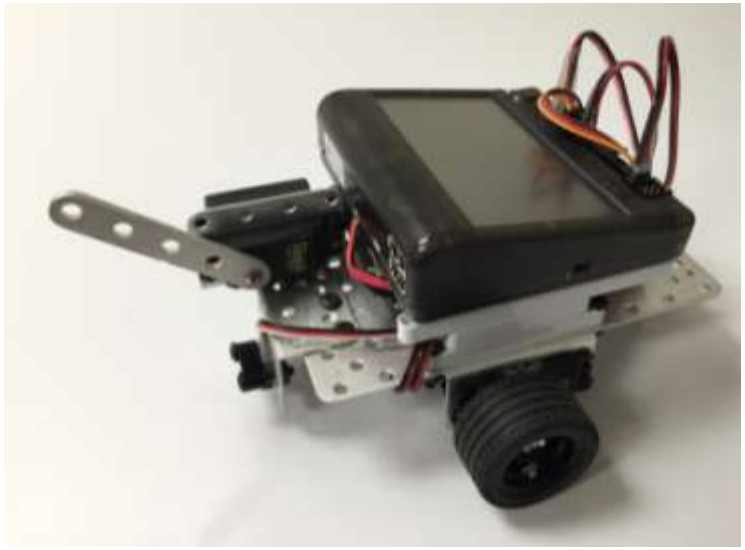
Professional Development Workshop
© 1993 – 2016 KIPR

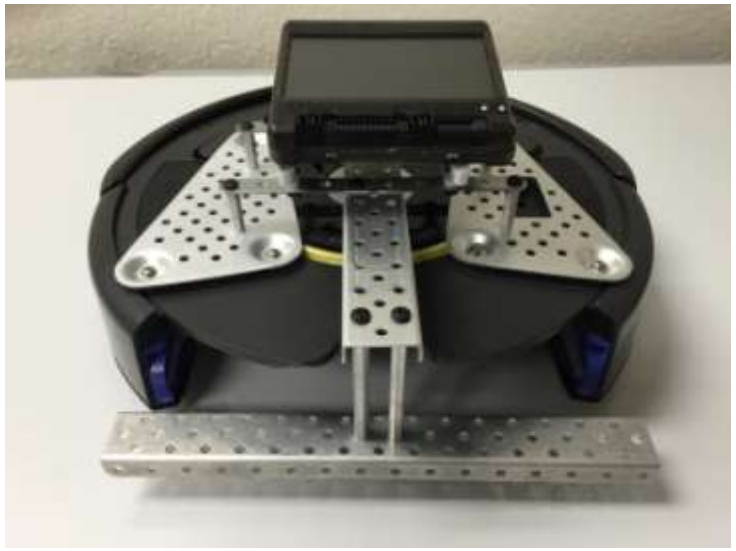
Botball®

Build a DemoBot

Build your robot using the DemoBot Building Guide

(Build the demobot on the left first. If you have time or spare hands, build the Create based robot on the right)





© 1993 – 2016 KIPR



Hi! I'm Botguy, the Botball mascot!

Botball 2016

Professional Development Workshop

Prepared by the KISS Institute for Practical Robotics (KIPR)

with significant contributions from KIPR staff

and the Botball Instructors Summit participants

v2016.01.16-0

© 1993 – 2016 KIPR

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.

© 1993 – 2016 KIPR

Botball®

Housekeeping

- **Introductions:** workshop staff
- **Bathrooms and Food:** lunch is on your own
- **Workshop schedule:** days 1 and 2

© 1993 – 2016 KIPR

Workshop Schedule

Day 1

- Botball Overview
- Getting started with the KIPR web-based software suite
- Explaining the “Hello, World!” C Program
- Designing Your Own Program
- Moving the DemoBot with Motors
- Fun with Functions
- Moving the DemoBot Servos
- Making Smarter Robots with Sensors
- Repetition, Repetition, Repetition
- 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

© 1993 – 2016 KIPR

Thanks to our national sponsors!



COMMON SENSE RC

igus®

iRobot®

SOLIDWORKS

© 1993 – 2016 KIPR

Thanks to our regional sponsors!



ARVEST

Practical Robotics Institute Austria

College of Engineering

ITCCC

青少年国际竞赛与交流中心

International Teenager Competition and Communication Center

ANALOG DEVICES

Oklahoma Aeronautics Commission

astellas

Leading Light for Life

NSF

QUALCOMM

BancFirst Loyal

To Oklahoma & You.

HITACHI

Inspire the Next

EDUCATION

robonation

ROBOTICS COMMUNITY

ROC

ROBOTICS ORGANIZING COMMITTEE

HAWAII

Rockwell Collins

KIRKPATRICK FOUNDATION

© 1993 – 2016 KIPR

Thanks to our regional hosts!

USC University of Southern California

HTL Hollabrunn, Austria

NASA

Southern Illinois University Edwardsville

Grossmont College

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

UNF University of North Florida

University of Maryland

NM State University

Loyola University Chicago

Carnegie Mellon Qatar

UMass Lowell

Preparing people to lead extraordinary lives

Worcester State University

tgm
Vienna Institute of Technology
Austria

APL STEM

© 1993 – 2016 KIPR

Botball Overview

What and when?

GCER and ECER

Preview of this year's game

Homework for tonight

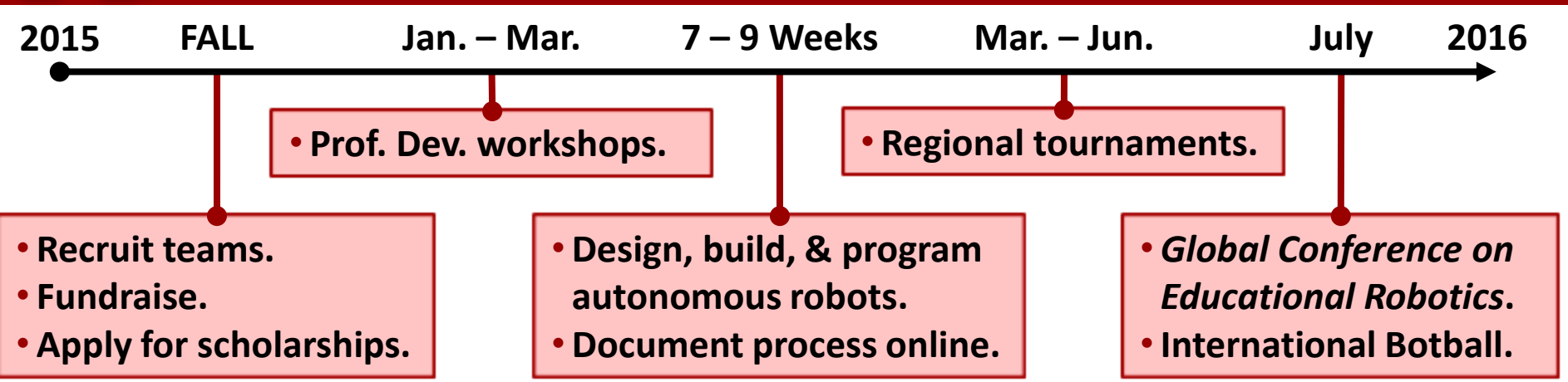
© 1993 – 2016 KIPR

What is Botball?

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

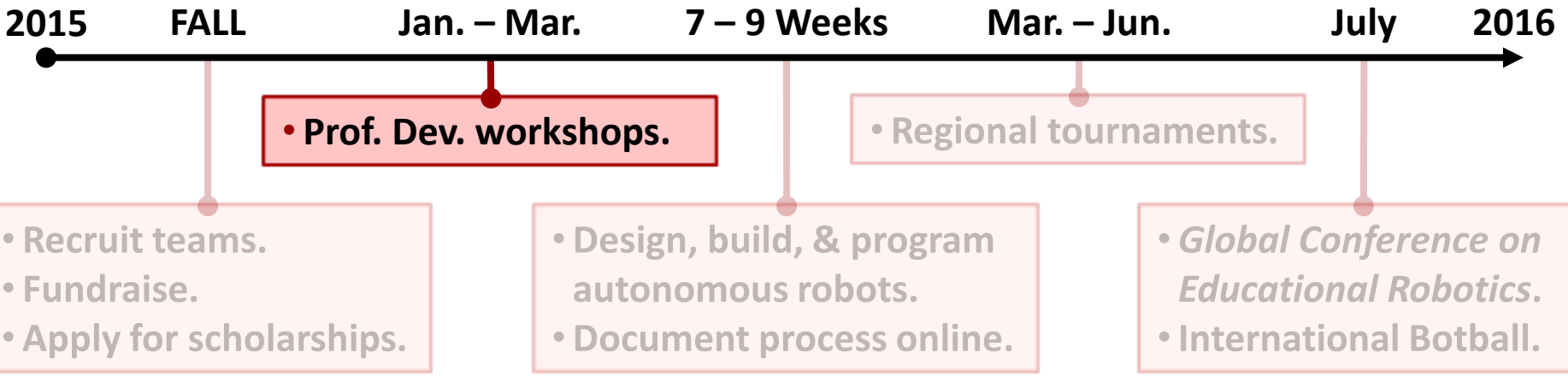
© 1993 – 2016 KIPR

When is Botball?



© 1993 – 2016 KIPR

When is Botball?



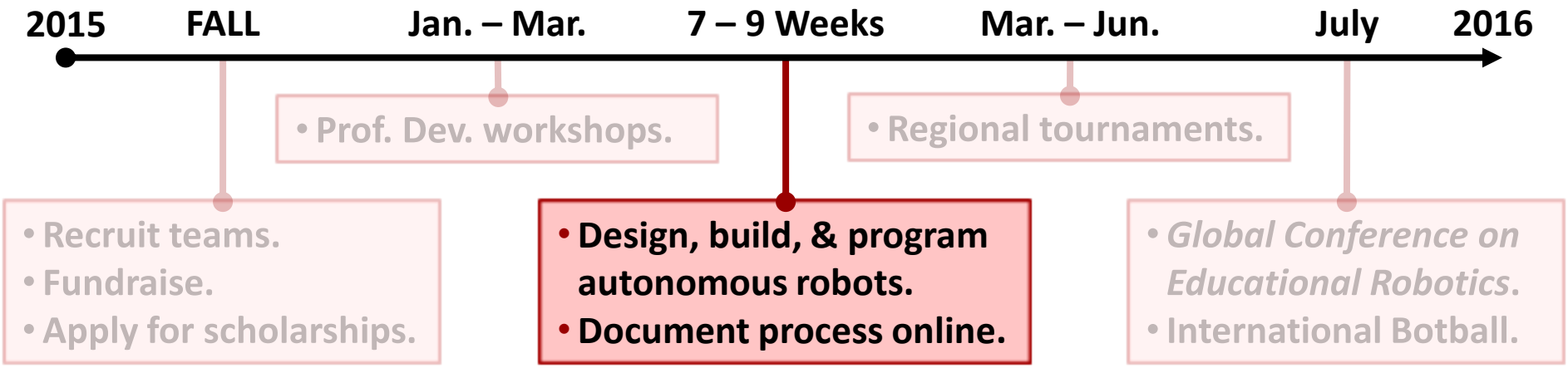
YOU ARE HERE!

- **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**.)

© 1993 – 2016 KIPR

When is 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.

© 1993 – 2016 KIPR

When is Botball?

2015 | FALL | Jan. – Mar. | 7 – 9 Weeks | Mar. – Jun. | July | 2016



- Recru
- Fund
- Appl

uild, & program
ous robots.
t process online.

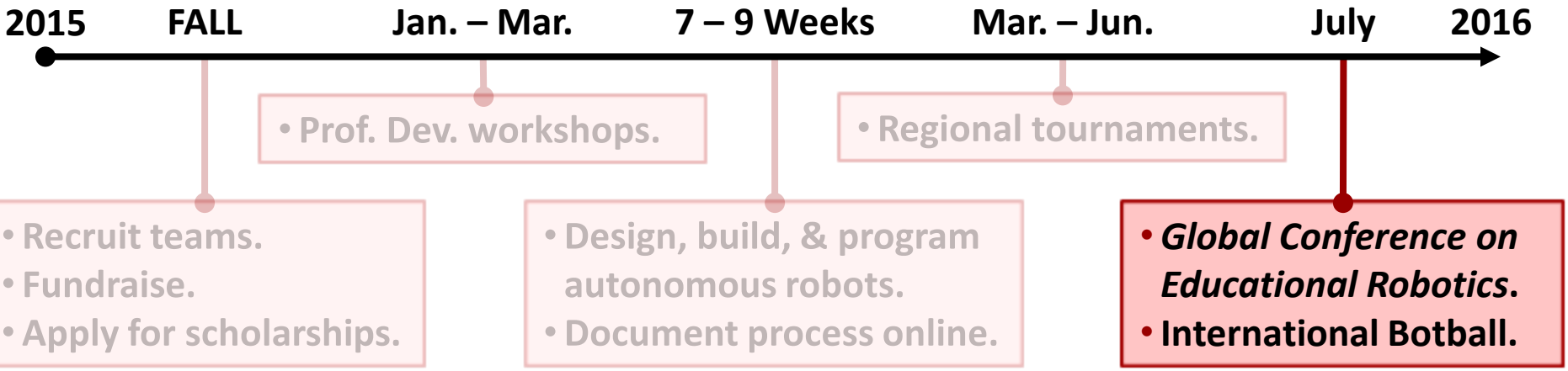
- **Regional tournaments.**

- *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

© 1993 – 2016 KIPR

When is 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!

© 1993 – 2016 KIPR

GCER-2016

Global Conference on Educational Robotics

- St. Augustine, Florida
- July 6-10, 2016
- 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://www.kipr.org/gcer

© 1993 – 2016 KIPR

GCER-2016

Global Conference on Educational Robotics

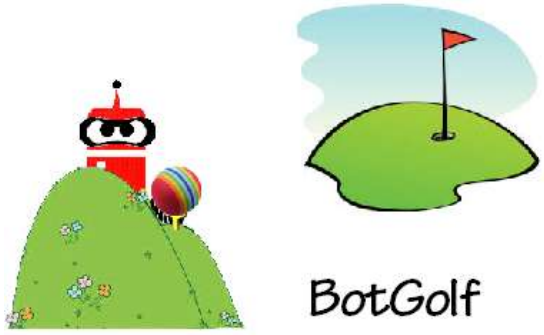


Preconference classes on July 6th

- Learning About Microcontrollers
- Expressive Robotics
- Making Sense of Sensors

KIPR Open Autonomous Robotics Game

- Botball for grown-up kids!



Autonomous Aerial Vehicle Competition



© 1993 – 2016 KIPR

ECER-2016

European **C**onference on **E**ducational **R**obotics

- Vienna, Austria
- April 11-15, 2016

- European Botball Competition
- Talks by Researchers and Students



© 1993 – 2016 KIPR

Preview of this year's Botball game

Hold your questions!

Game Q&A is tomorrow!

© 1993 – 2016 KIPR

Botball 2016 Game Video



© 1993 – 2016 KIPR

Botball game board



© 1993 – 2016 KIPR

Homework for tonight:
game review

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.

© 1993 – 2016 KIPR

Botball Team Home Base

Found at http://homebase.kipr.org



© 1993 – 2016 KIPR

Getting Started with the KIPR Web-based Software Suite

What is a programming language?

How can I create new projects and files?

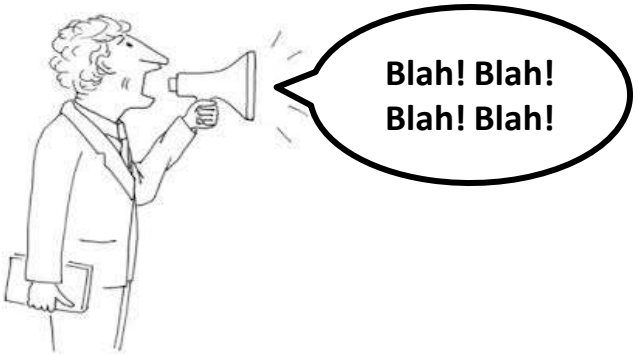
How can I write and compile source code?

How can I run programs on the KIPR Wallaby?

© 1993 – 2016 KIPR

What is a *programming language*?

Human



Computer

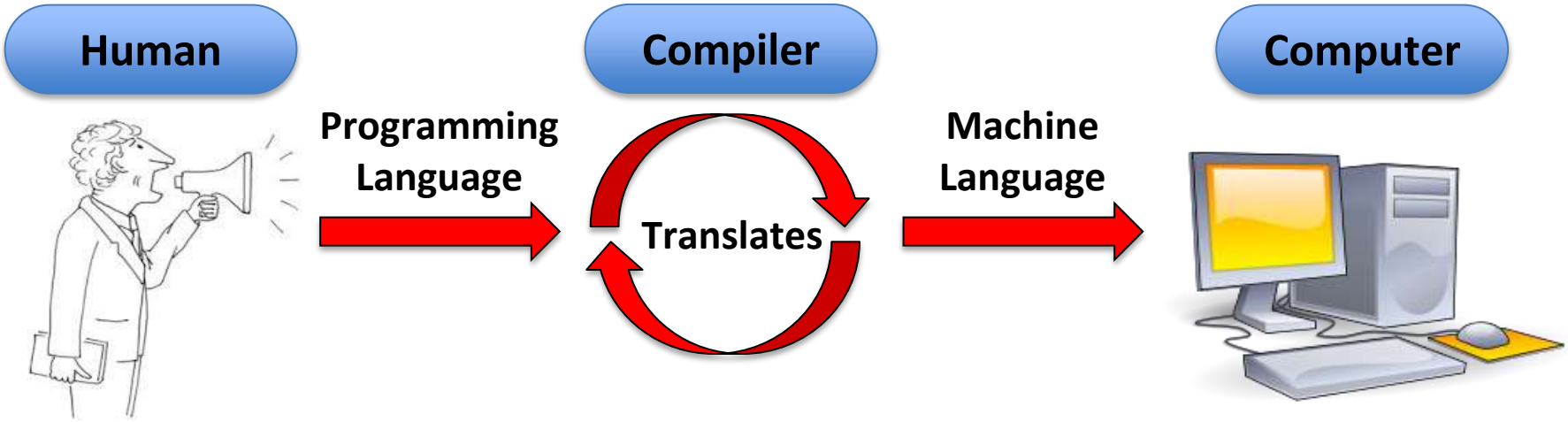


- **Computers** only understand **machine language** (stream of bytes), which computers can **read and execute** (run).

- Unfortunately, **humans** don't speak **machine language**…

© 1993 – 2016 KIPR

What is a *programming language*?



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

© 1993 – 2016 KIPR

Connect the Wallaby to your computer

Activity

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

1. Turn on the Wallaby with the **black switch on the side** – you may need to use a stylus.



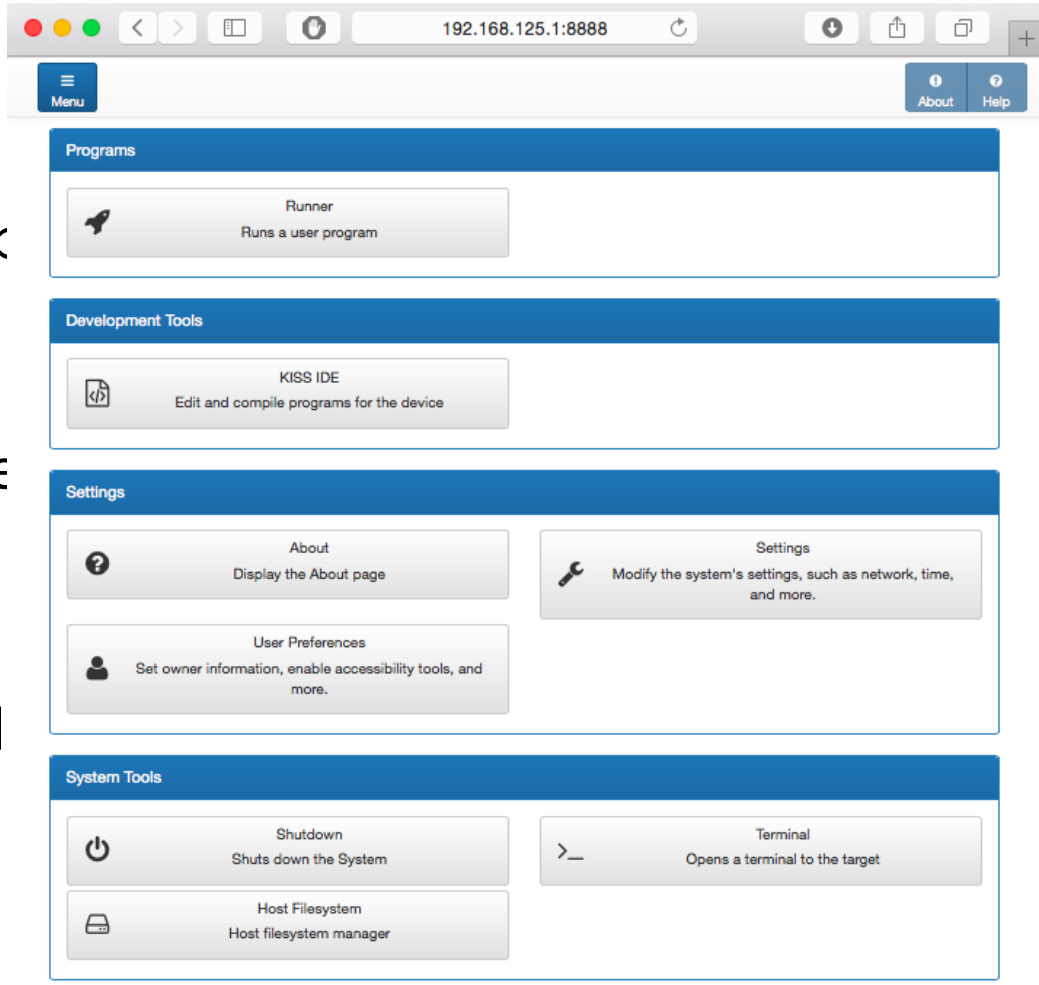
2. Once your Wallaby has booted, the Wallaby will appear in the list of available Ethernet connections for your computer.
3. If you get a message about the driver raise your hand for help.

© 1993 – 2016 KIPR

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**.



© 1993 – 2016 KIPR

Loading the Starting Web Page

Activity

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

192.168.124.1:8888

IP address Port #

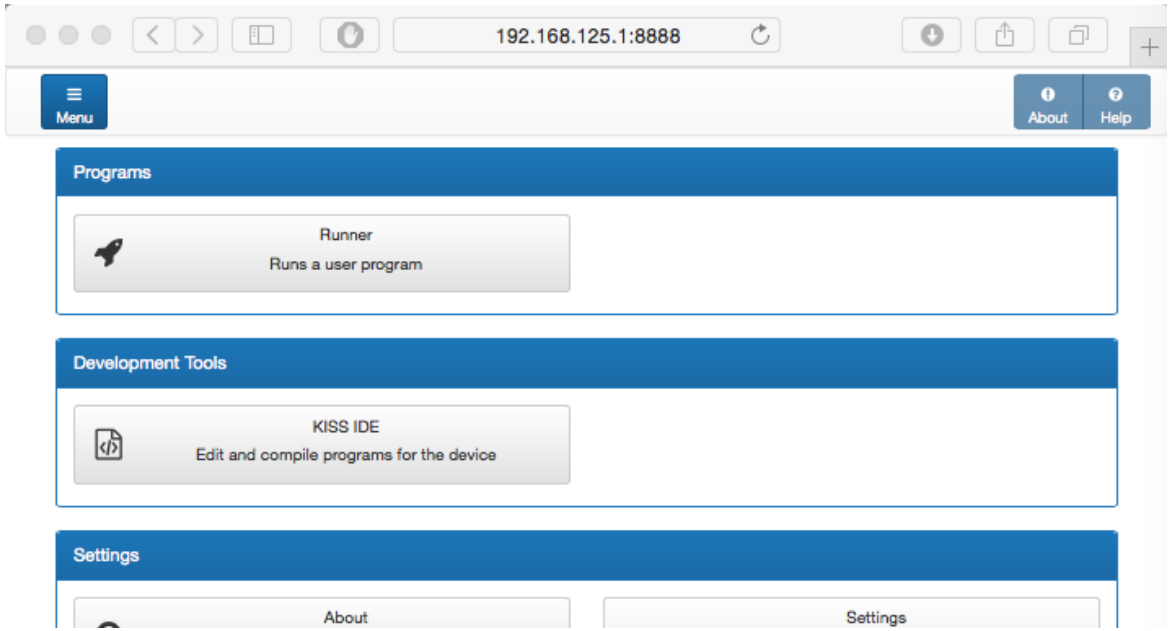
4. The user interface for the package will now come up in your browser.
5. You may use a computer, tablet or even a smart phone through WiFi

© 1993 – 2016 KIPR

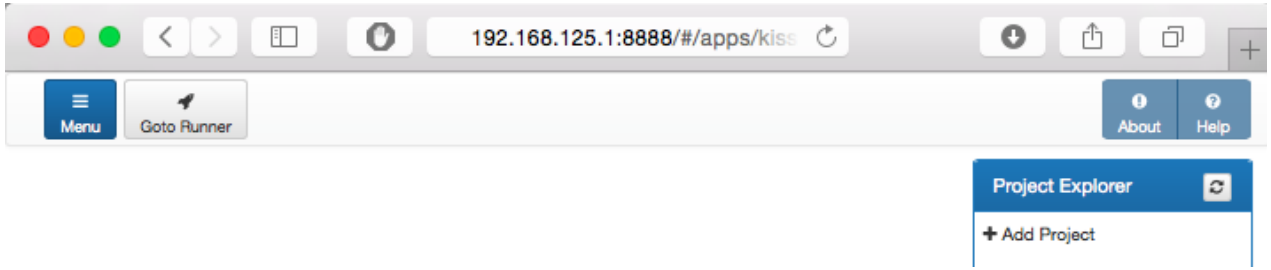
Creating your first project

Activity

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



2. Click on "+ Add Project"



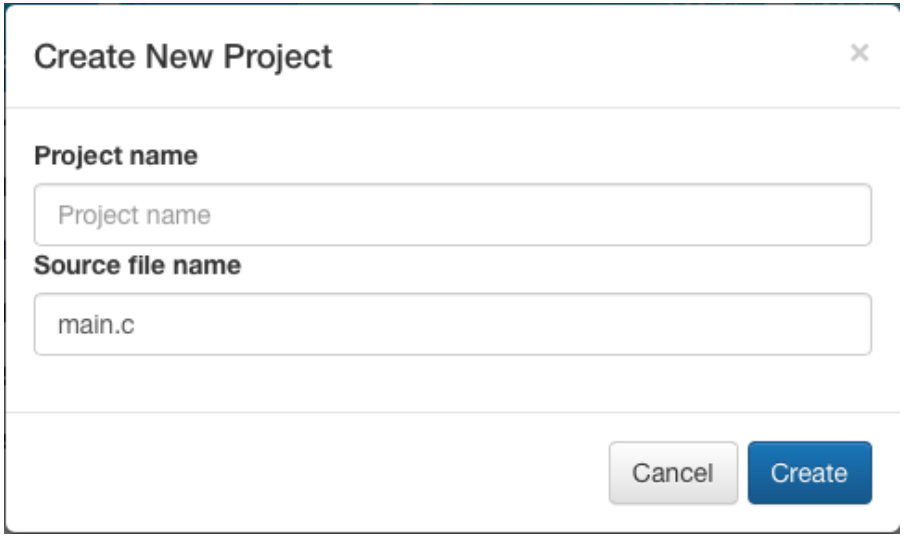
Professional Development Workshop
© 1993 – 2016 KIPR

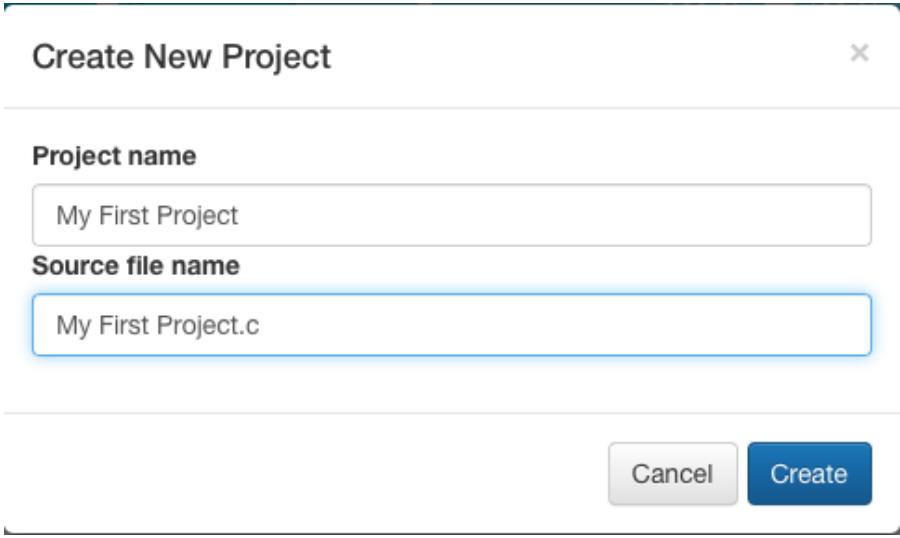
Botball®

Name your project

Activity

1. Give your *project* a **descriptive name**
   - **Note:** you will have a lot of student's projects, so maybe use 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.



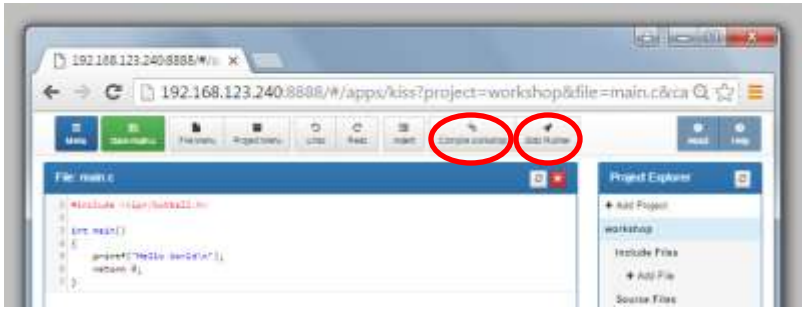


© 1993 – 2016 KIPR

Compile and Then Run Your Project

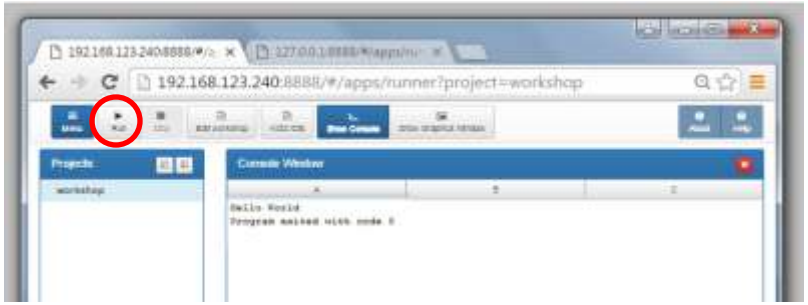
Activity

1. Click the “Compile” button for your project and (if successful) click the “Goto Runner” so you can run your project to see if it works.



- When you compile, your project is automatically saved.

2. In the Run page click the “Run” button and observe the results.



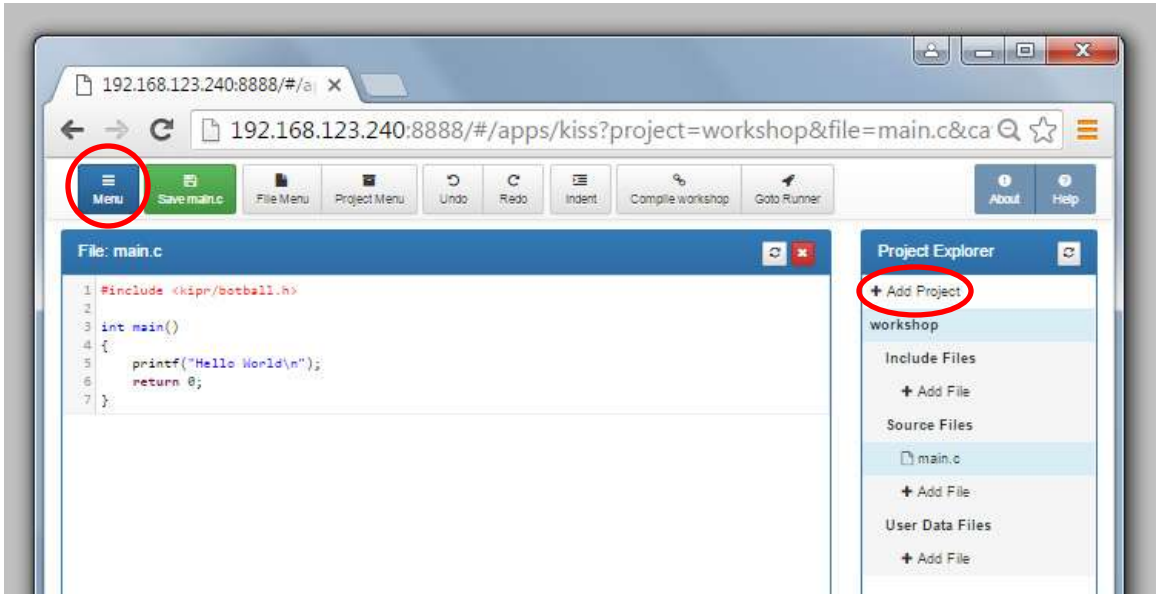
© 1993 – 2016 KIPR

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 all of your projects but only show the files for the one your are currently working on.



© 1993 – 2016 KIPR

Explaining the “Hello, World!” C Program

Program flow and the main function

Programming statements and functions

Comments

© 1993 – 2016 KIPR

“Hello, World!”

File: main.c

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

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.

© 1993 – 2016 KIPR

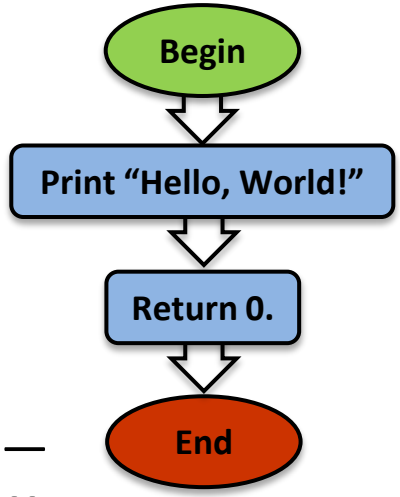
Program flow and line numbers

Top

Bottom

File: main.c

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



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!

© 1993 – 2016 KIPR

Source code

File: main.c

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



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

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

© 1993 – 2016 KIPR

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.

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

This is the "**`main` function**".

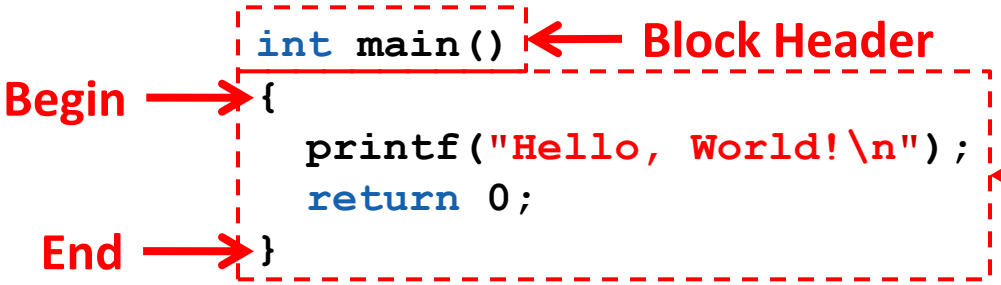
When you run your program,
the **`main` function** is executed.

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

© 1993 – 2016 KIPR

Block of code

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



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

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 {.

© 1993 – 2016 KIPR

Programming statements

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

Statement #1 → `printf("Hello, World!\n");`

Statement #2 → `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.

© 1993 – 2016 KIPR

Ending a programming statement

```
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.

© 1993 – 2016 KIPR



Ending the `main` function

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

The **`main` function** ends with a **`return`** statement, which is the 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**.

© 1993 – 2016 KIPR



Comments

Explanatory text (called “comments”) identified by a preceding “//” can be added to programs.

```
// Created on Sat January 9, 2016

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

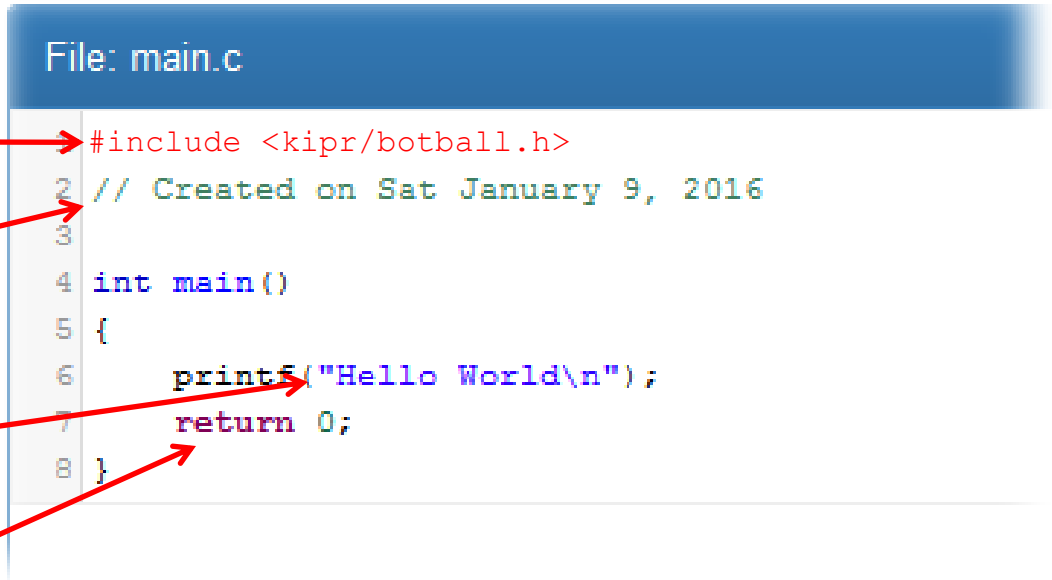
Comments are helpful notes that can be read by you or your team—**they are *ignored* (not read) by the computer!**

© 1993 – 2016 KIPR

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.)

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



© 1993 – 2016 KIPR



Print your name

Activity

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

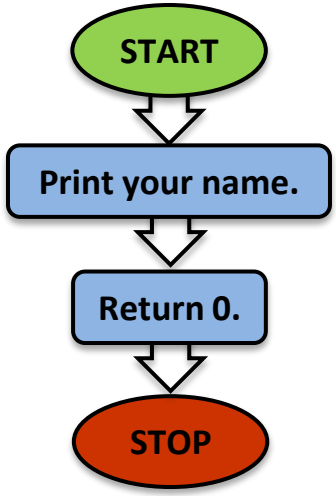
Solution:

Source Code

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

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

Flowchart



© 1993 – 2016 KIPR



Designing Your Own Program

Breaking down a task

Pseudocode, flowcharts, and comments

`wait_for_milliseconds` function

Debugging your program

© 1993 – 2016 KIPR

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.

© 1993 – 2016 KIPR

Practice printing

Activity

Description: Write out (in English) how to get the KIPR Robotics Controller to print "Hello, World!" on one line, and then print 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.

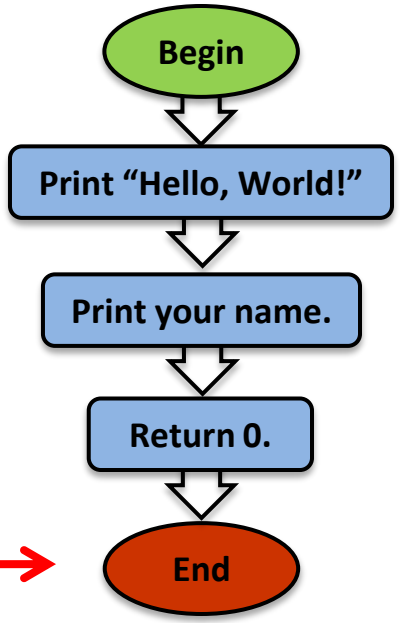
In English, write a list of actions to solve an activity.

Comments

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

Helps you *plan* the real code!

Flowchart



© 1993 – 2016 KIPR



Practice printing

Activity

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

- **Note:** remember to give your project a meaningful name!

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 Wallaby.

© 1993 – 2016 KIPR

Practice printing

Activity

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

- The KIPR Robotics Controller reads the code and goes to the next line faster than a blink of your eye.
- At over 1 GHz, 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 KISS built-in function called **`wait_for_milliseconds`** (or equivalently, **msleep**).

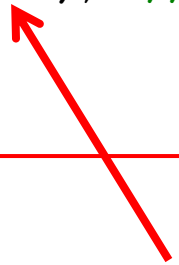
↑

Let's use this!

© 1993 – 2016 KIPR

Using `wait_for_milliseconds`

```
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)`.

© 1993 – 2016 KIPR

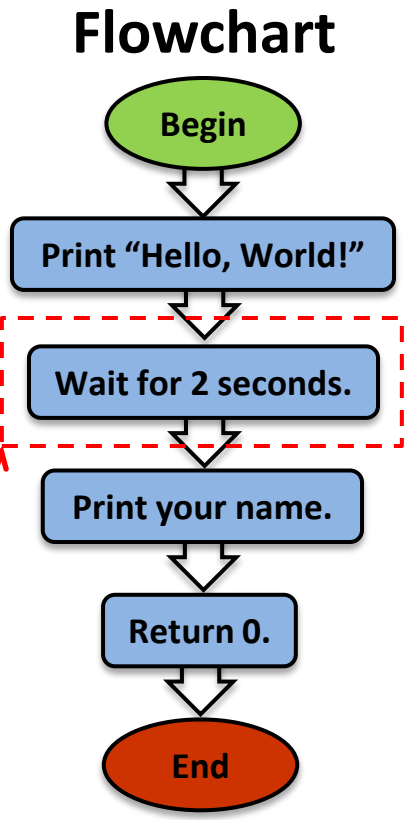
Waiting for some time

Activity

Description: Write a program for the KIPR Robotics Controller 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 2 seconds.`
3. Print Your name.	`// 3. Print your name.`
4. End the program.	`// 4. End program`



Professional Development Workshop
© 1993 – 2016 KIPR

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 Wallaby.

© 1993 – 2016 KIPR



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**"?

© 1993 – 2016 KIPR

Debugging Errors

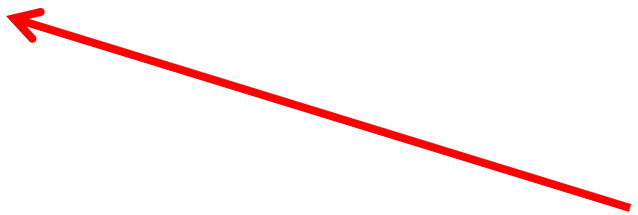
!!! ERROR !!!

- 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 "**Compilation Failed!**"
- The KISS IDE 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.

© 1993 – 2016 KIPR

Debugging Practice

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



Create a new file based on the Hello World template and delete this semicolon and compile.

© 1993 – 2016 KIPR

Debugging Errors

Activity

line # : col # (the error is on or before the line # 8)

```
s\KISS\src\workshop\main.c: In function 'main':
s\KISS\src\workshop\main.c:8:1: error: expected ';' before '}' token
```

" expected : " (semicolon)

File: main.c

```
1 #include <kipr/botball.h>
2 // Created on Sat January 9, 2016
3
4 int main()
5 {
6     printf("Hello World\n");
7     return 0
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 ← **Yikes! Our program did not compile!**

```
Compilation Failed

C:\Users\Charles Winton\Documents\KISS\src\workshop\main.c: In function 'main'
C:\Users\Charles Winton\Documents\KISS\src\workshop\main.c:8:1: error: expected ';' before '}' token
```


© 1993 – 2016 KIPR



Moving the DemoBot with Motors

Plugging in motors (ports and direction)

`motor` functions

© 1993 – 2016 KIPR

Check your robot's motor ports

Activity

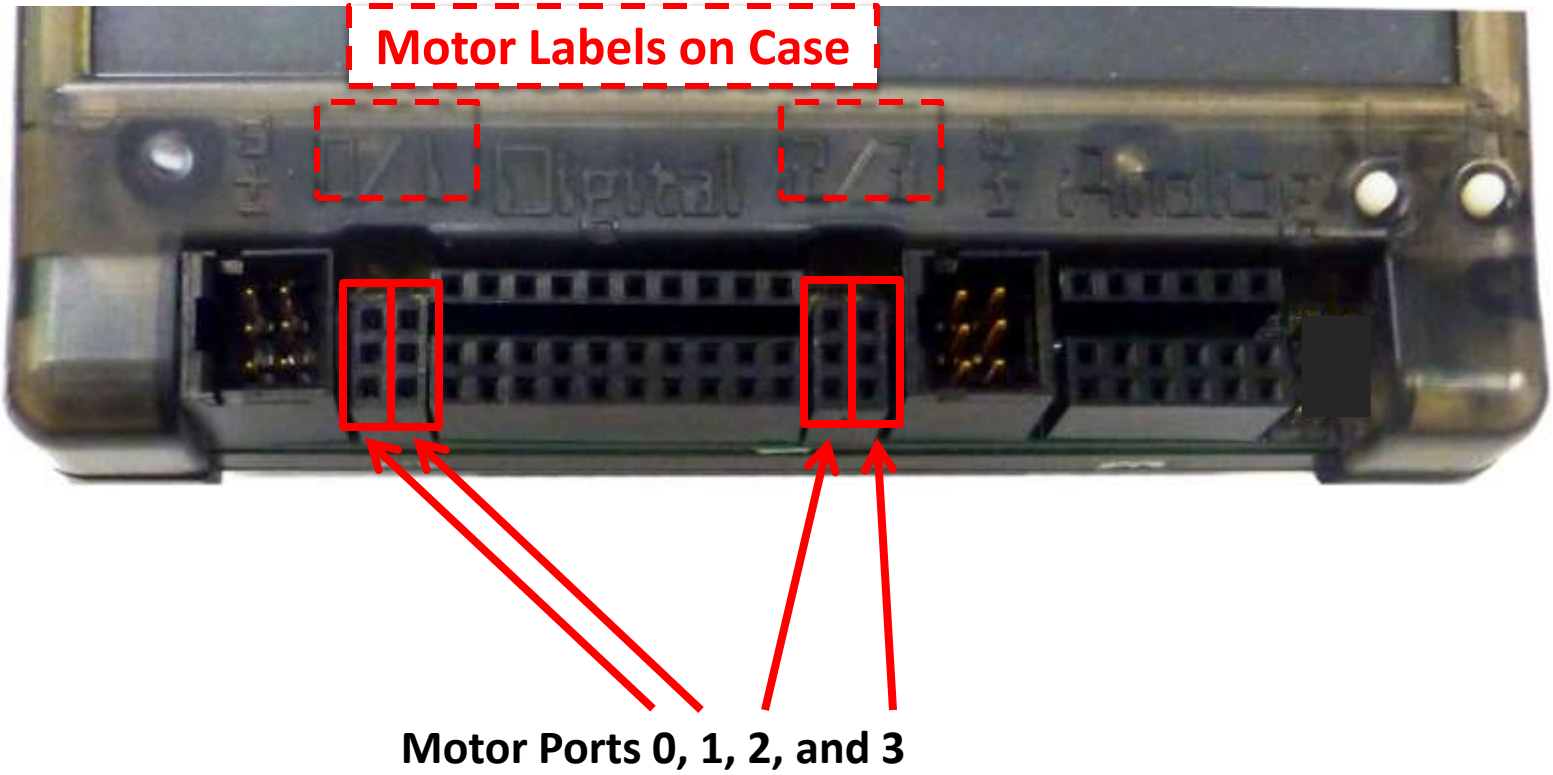
- To program your robot to move, you need to know which **motor ports** your motors are plugged into.

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

© 1993 – 2016 KIPR

Wallaby motor ports

Activity

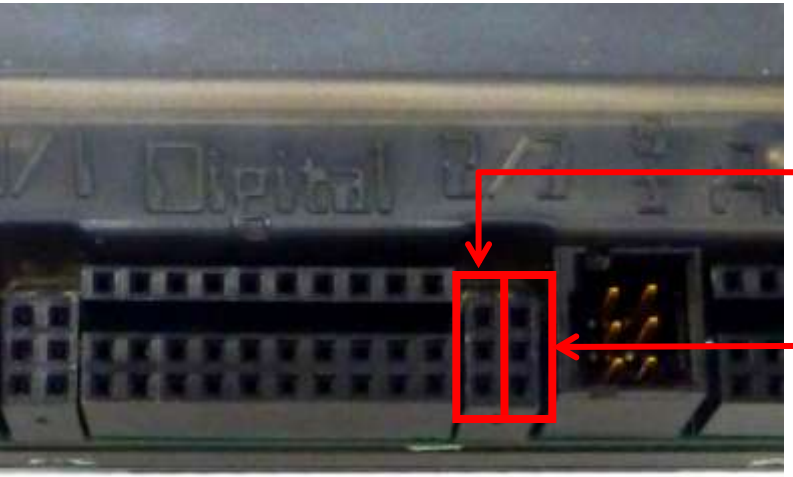


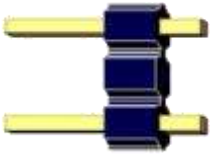
© 1993 – 2016 KIPR

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 the left, and **2** & **3** on the 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** (**–**).



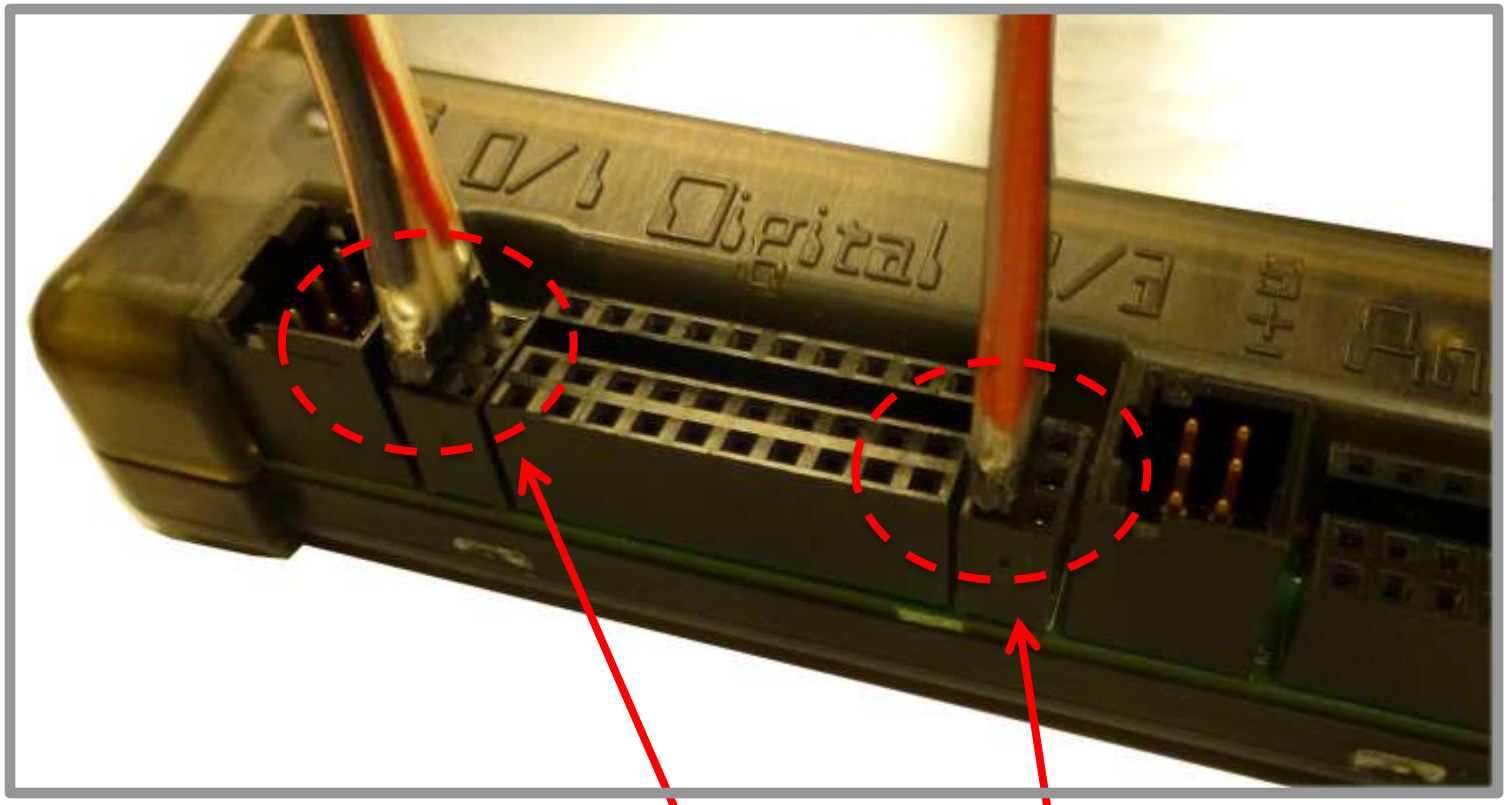




Drive motors have a two-prong plug.

Plugged in motors

Activity



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

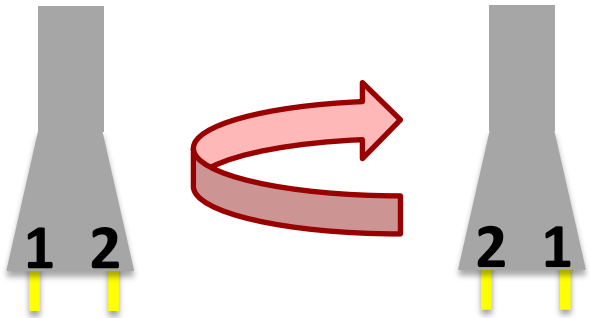
© 1993 – 2016 KIPR

Motor direction

Activity

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.



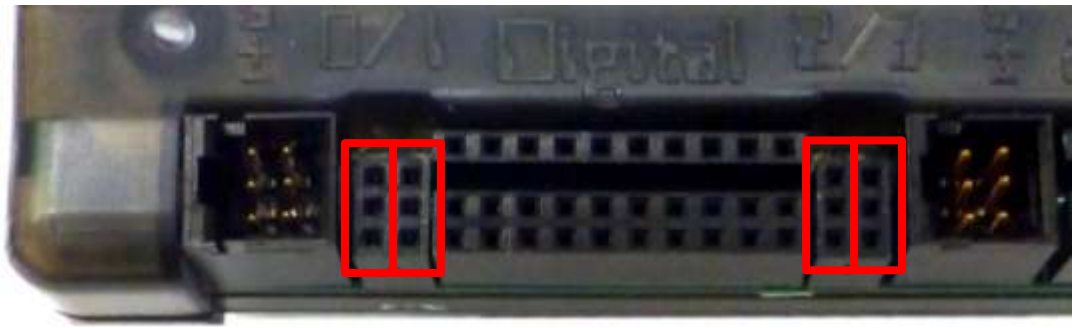
© 1993 – 2016 KIPR

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** (**–**).



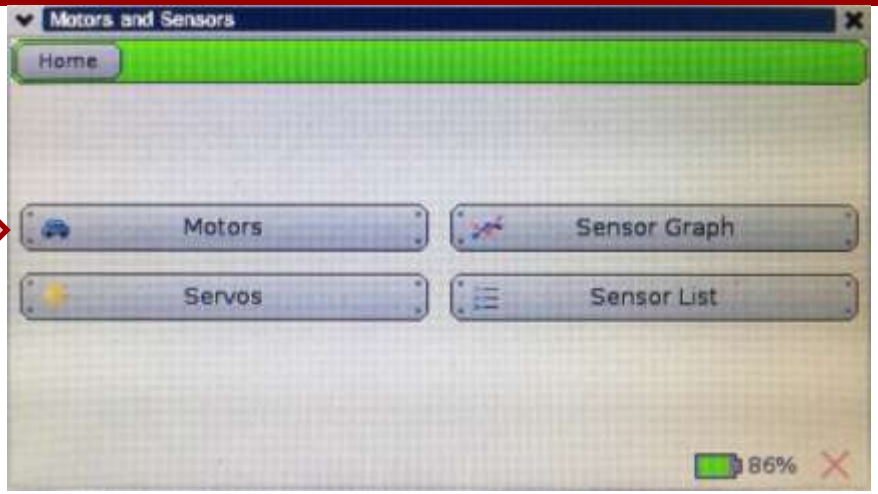


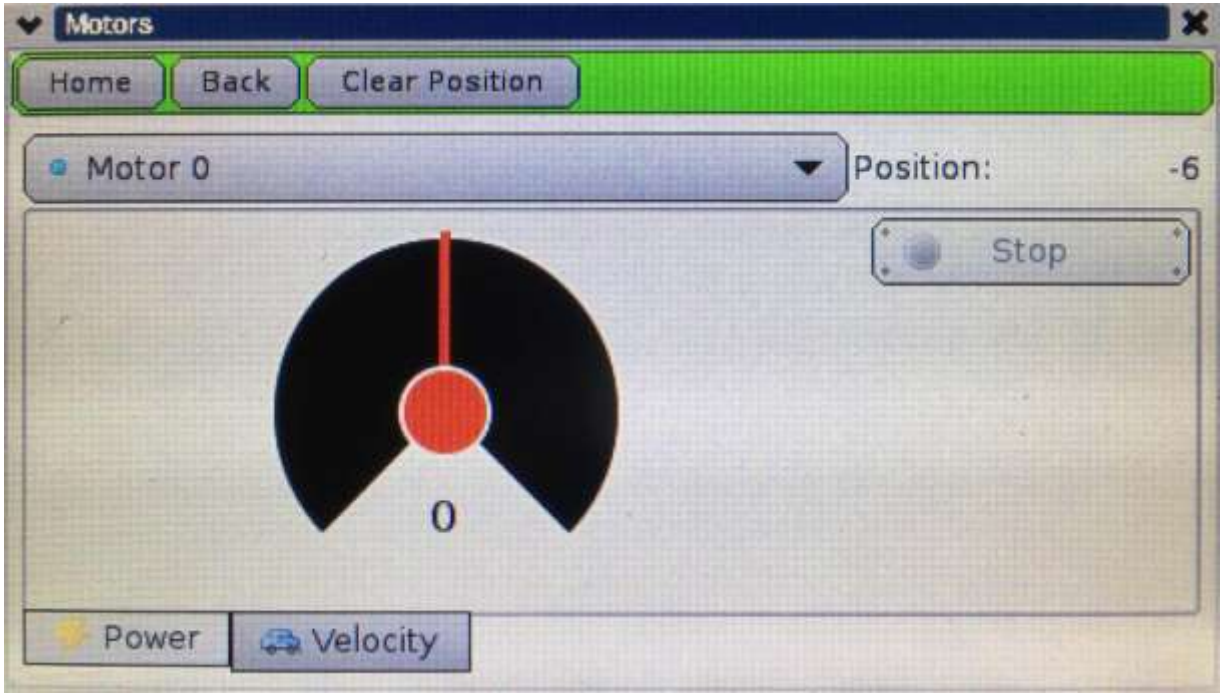
- Use this trick to check the **port #**'s and **direction** of your **motors**.
  - If one is **red** and the other is **green**,
    turn one motor plug 180° and plug it back in.
  - The lights should both be **green** if the robot is moving forward.

© 1993 – 2016 KIPR

Use the Motor Widget







Botball®

Common motor functions

There are several functions for motors.
We will begin with mav
(motor angular velocity)

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

```
mav(0, 500);
// Turns on motor port #0 at a motor
// angular velocity of 500 ticks per second
// (velocity between -1000 and +1000).

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

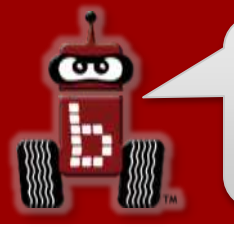
ao();
// Turn off all of the motors.
```

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.

© 1993 – 2016 KIPR



Using mav and ao

```
int main()
{
  printf("Drive forward!\n"); // State intent
  mav(0, 500); // turn on motor
  wait_for_milliseconds(2500);
  ao(); // turn off all motors
  return 0;
}
```


© 1993 – 2016 KIPR



KISS IDE functions cheat sheet

Resource

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 #, % 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 #
analog_et(port #)                          // Get an ET sensor reading from a specified analog port #
shut_down_in(time in seconds);             // Shuts down all motors after specified # of seconds
```


© 1993 – 2016 KIPR

Sensor and Motor Manual

Resource

For additional details about **motors**, consult the "**Sensor and Motor Manual**", available via the KISS IDE "**Help**" menu or on your **KIPR USB drive**.



© 1993 – 2016 KIPR

Moving the DemoBot

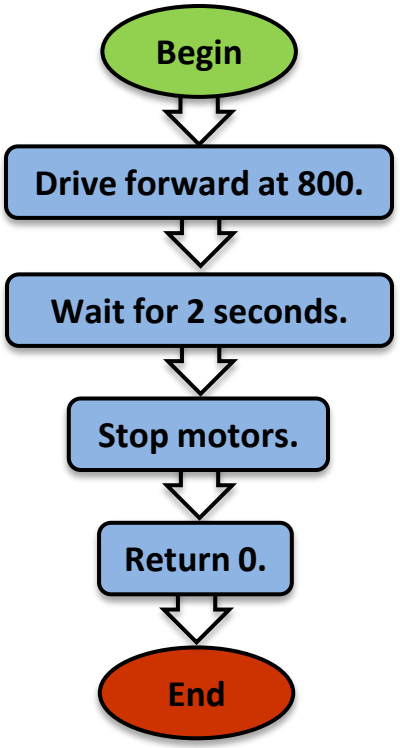
Activity

Description: Write a program for the KIPR Robotics Controller that drives the DemoBot forward at a motor velocity of 800 ticks per second for two seconds, and then stops.

Analysis: What is the program supposed to do?

Pseudocode	Comments
1. Drive forward at 800.	`// 1. Drive forward at 800 tps.`
2. Wait for 2 seconds.	`// 2. Wait for 2 seconds.`
3. Stop motors.	`// 3. Stop motors.`
4. End the program.	`// 4. End the program.`

Flowchart



© 1993 – 2016 KIPR

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!

Psuedocode (Comments)

```
int main()
{
  // 1. Drive forward at 800.
  // 2. Wait for 2 seconds.
  // 3. Stop motors.
  // 4. End the program.
}
```

Source Code

```
int main()
{
  // 1. Drive forward at 80%.
  mav(0, 800);
  mav(2, 800);

  // 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 Wallaby.

© 1993 – 2016 KIPR

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 Link.
  - 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? (if you substitute motor commands for the mav commands, how straight do you think your DemoBot will go?).
- How can you make the robot turn left or right? (hints on an upcoming slide)

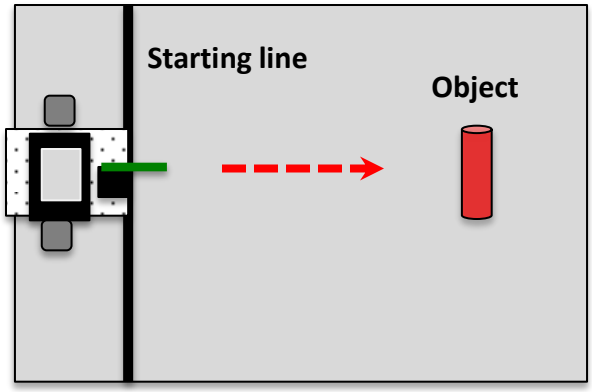
© 1993 – 2016 KIPR

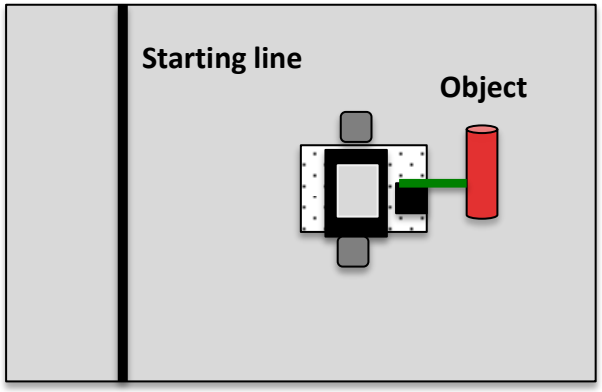
Touch an object and “go home”

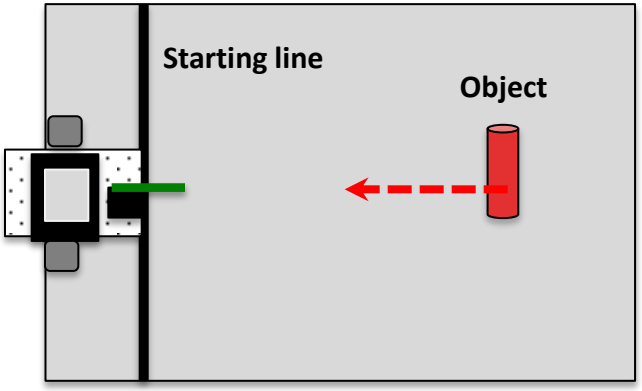
Activity

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

- Move the object twice as far. Move at twice the velocity or for twice as long.







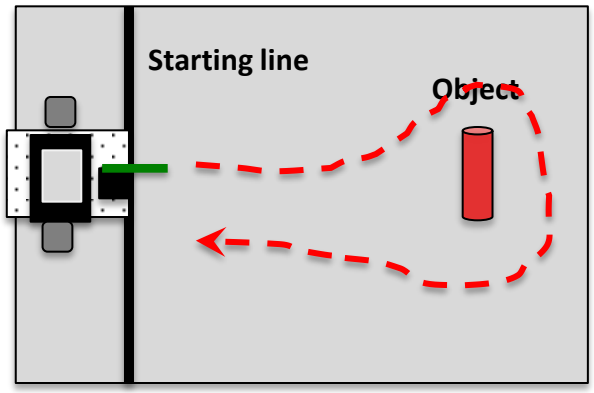
© 1993 – 2016 KIPR

Circle an object and “go home”

Activity

Description: Write a program for the KIPR Robotics Controller that drives the DemoBot *around* an object, and then returns to its starting location.

- How do you make the robot turn?
- Does the robot move a consistent amount for a given time and velocity?
- Find speed and time for doing a right turn.



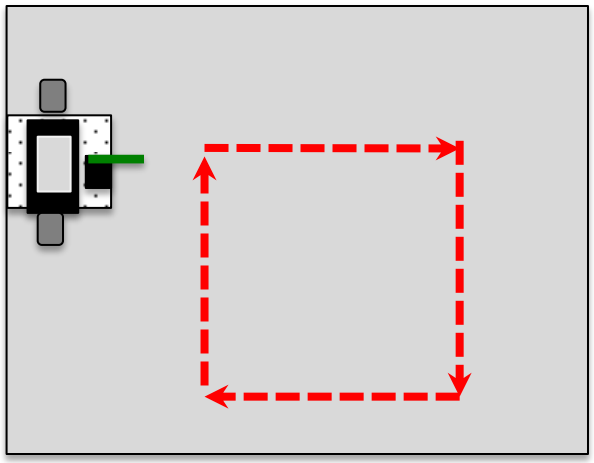
© 1993 – 2016 KIPR

Draw a square

Activity

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

- Start with having the robot make a 90° turn (both directions).
- Then add in forward movements to have the robot drive along a square path (remember the direction your robot is taking).
- Try drawing other regular polygons (pentagons, hexagons, octagons, etc.).



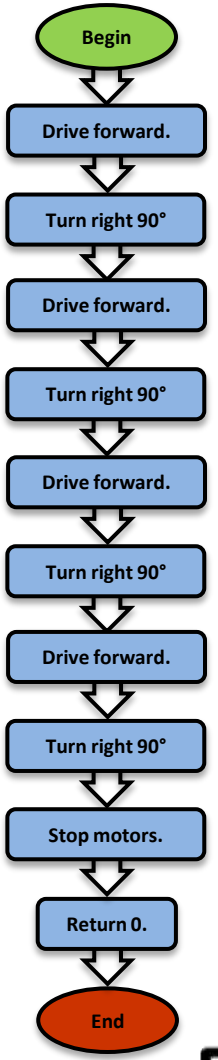
© 1993 – 2016 KIPR

Draw a square

Analysis: What is the program supposed to do?

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

Flowchart



Professional Development Workshop
© 1993 – 2016 KIPR

Botball®

Draw a square

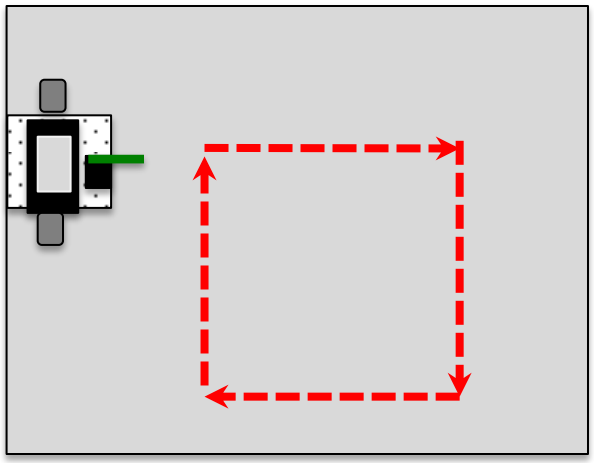
Activity

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.

The delays may not be correct!



```
int main()
{
  // 1. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 2. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 3. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 4. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 5. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 6. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 7. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

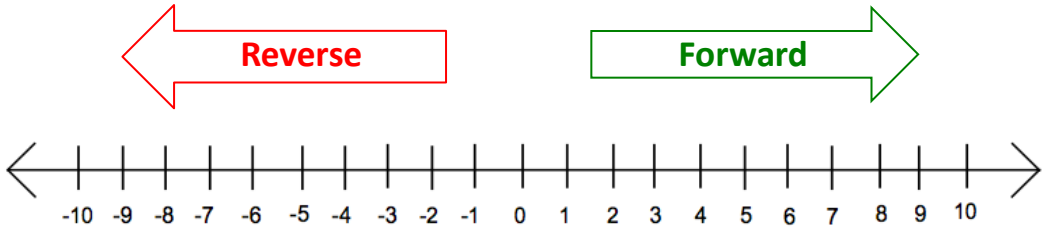
  // 8. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

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

Robot Driving Hints

Resource

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



Driving straight: it is surprisingly difficult to drive in a straight line using just the **motor** command

- **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, but the best solution is to use **move_at_velocity** (**mav**), which dynamically adjusts the power applied to a motor to keep a fixed motor velocity

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).

© 1993 – 2016 KIPR



Moving the DemoBot Servos

Plugging in servos (ports)

`enable_servos` and `disable_servos` functions

`set_servo_position` function

© 1993 – 2016 KIPR

Servos

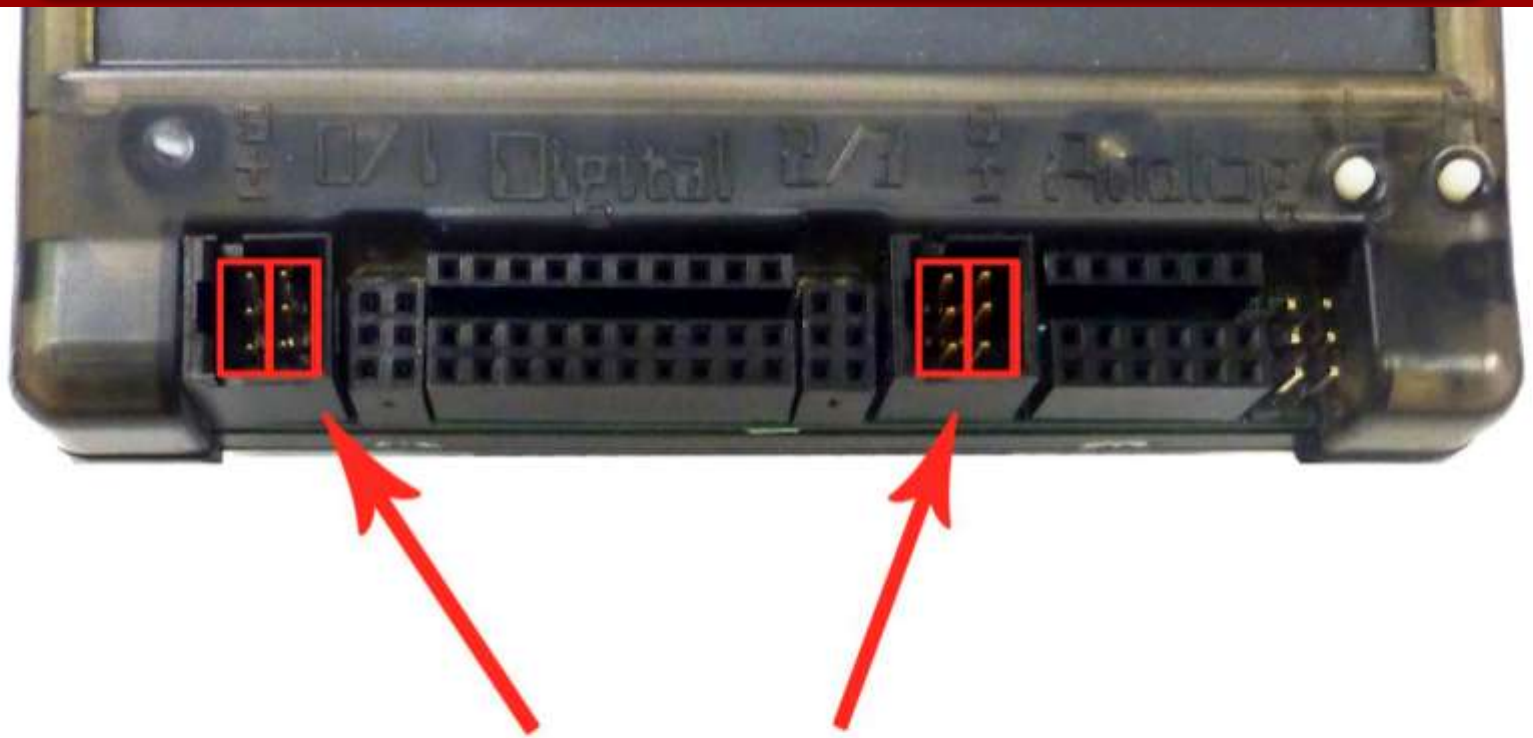
- A **servo motor** (or "**servo**" for short) is a motor that rotates to a specified **position between 0° and 180°**.
- Servos are great for raising an arm or closing a claw to grab something.
- Servo motors look very similar to non-servo motors, but there are differences…
  - A servo has **three wires**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



© 1993 – 2016 KIPR

KIPR Robotics Controller servo ports

Activity



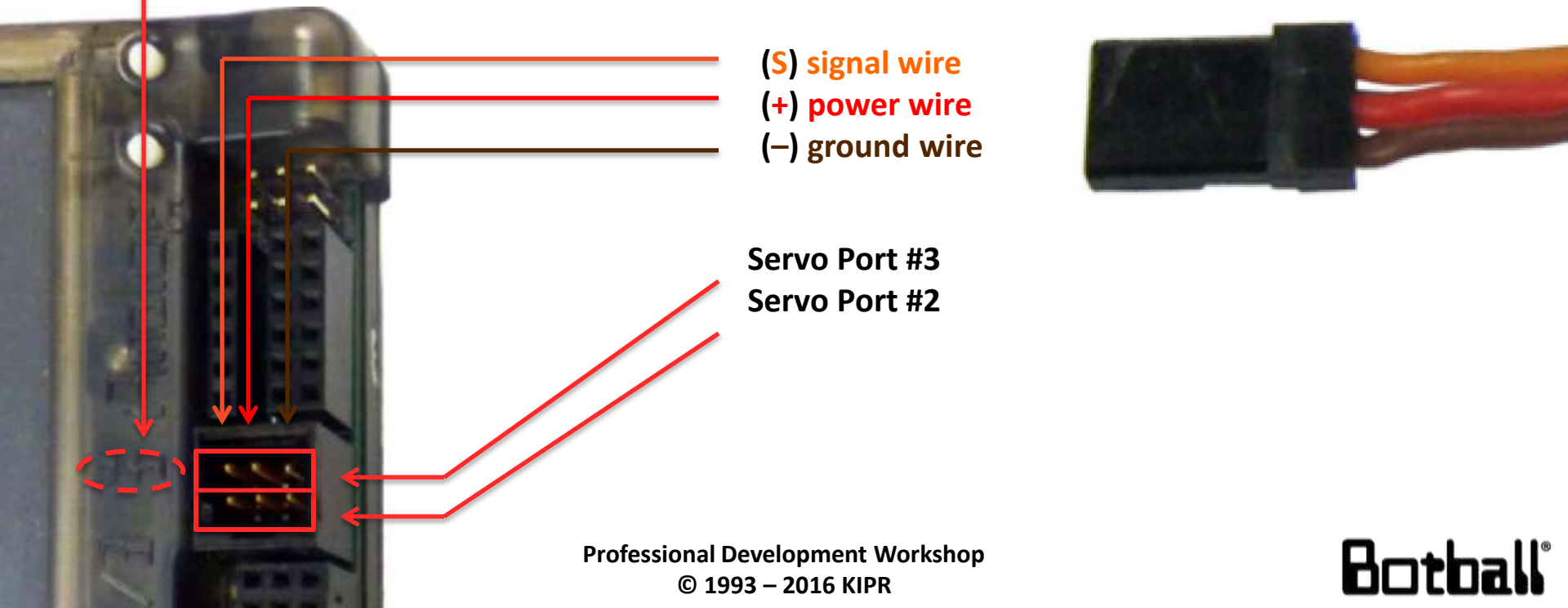
Servo Motor Ports 0, 1, 2, 3

© 1993 – 2016 KIPR

Plugging in servos

Activity

- The KIPR Robotics Controller has 4 servo ports numbered **0** (bottom) & **1** (top) on the left, and **2** (bottom) & **3** (top) 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")

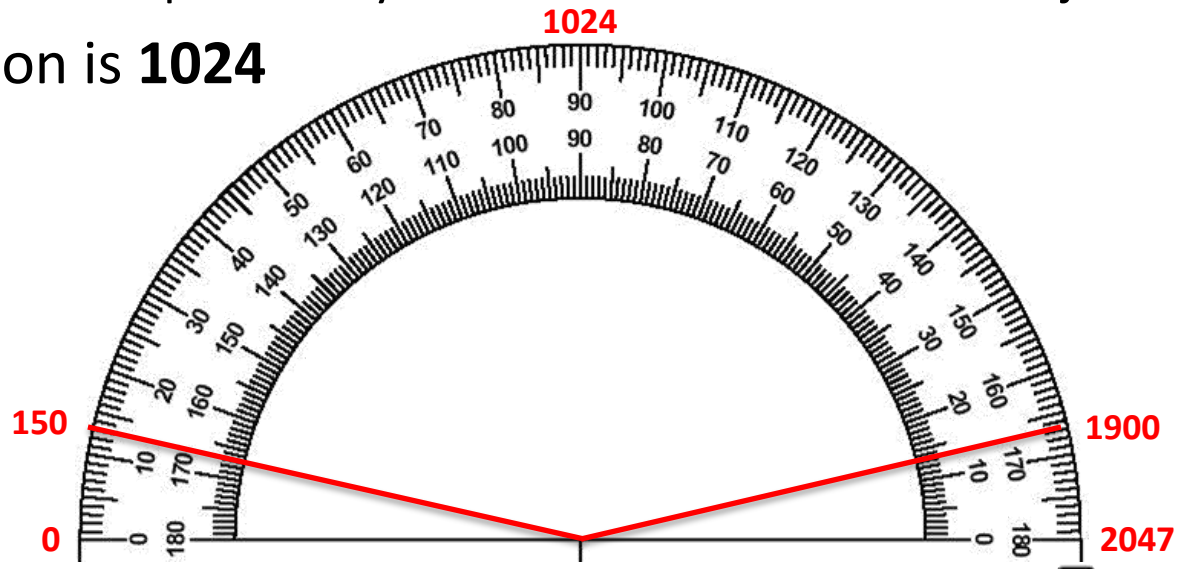


© 1993 – 2016 KIPR

Botball®

Servo positions

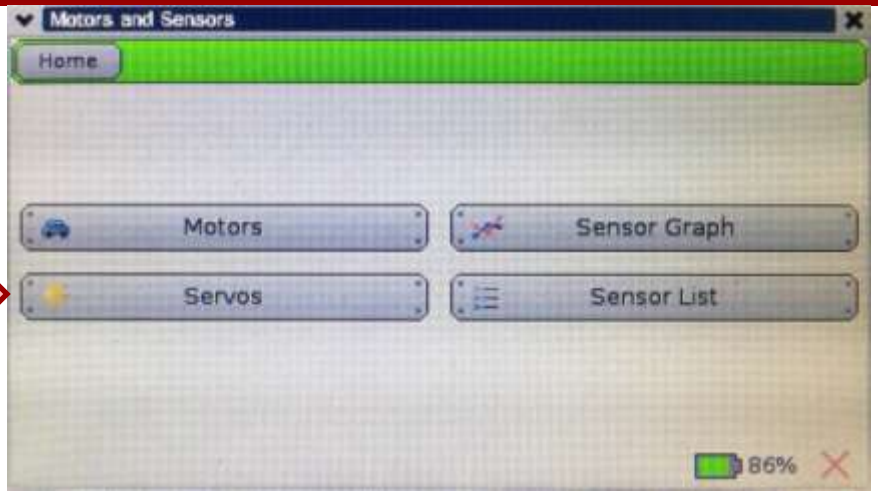
- Think of a servo like a protractor…
  - Angles in the **180° range of motion** (between 0° and 180°) are divided into **2048 servo positions**.
  - These **2048 positions** range from 0 to 2047, but due to internal mechanical hard stop variability you should use **150 to 1900**
    (**remember:** computer scientists start counting with 0, not 1).
  - This allows for greater precision when setting a position
    (you have 2048 different positions you can choose from instead of just 180).
- The default position is **1024** (centered).
- **No stops on new servos!**

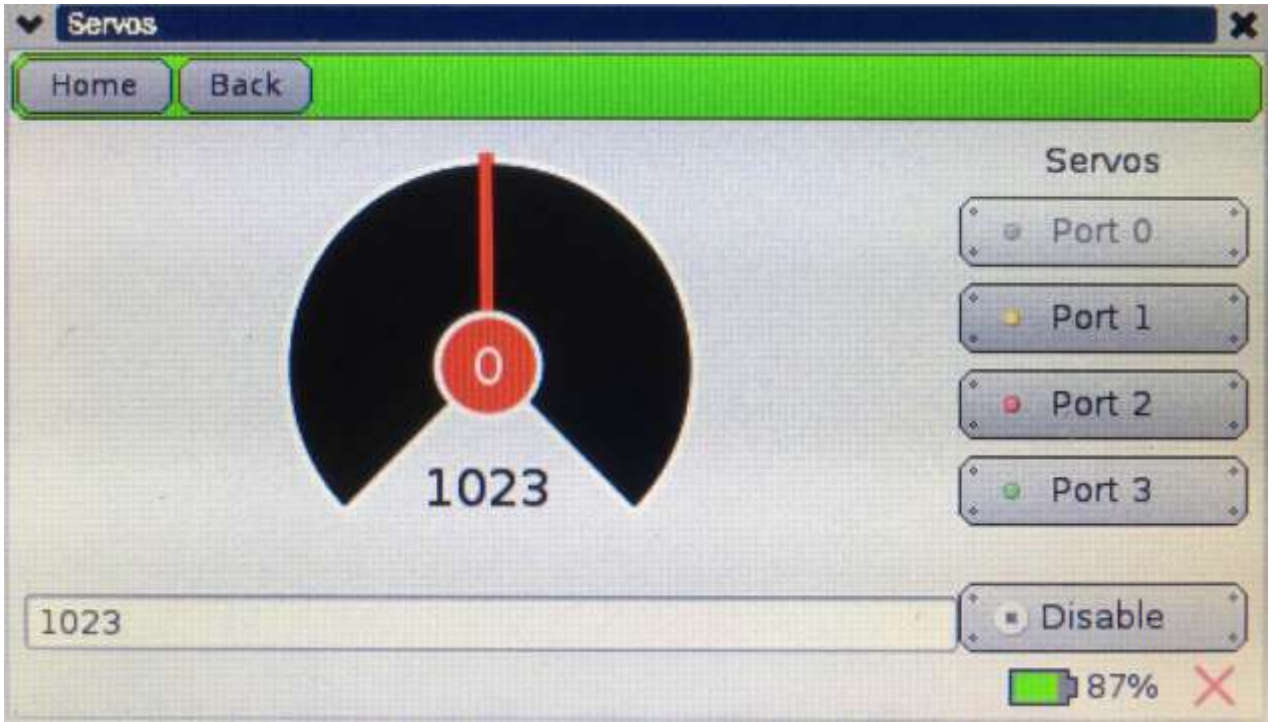


© 1993 – 2016 KIPR

Use the Servo Widget







Botball®



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.

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

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

- **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.

© 1993 – 2016 KIPR



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;
}
```


© 1993 – 2016 KIPR

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;
}
```


© 1993 – 2016 KIPR

Sensor and Motor Manual

Resource

For additional details about **servos**, consult the "**Sensor and Motor Manual**", available via the KISS IDE "**Help**" menu or from your team home base.



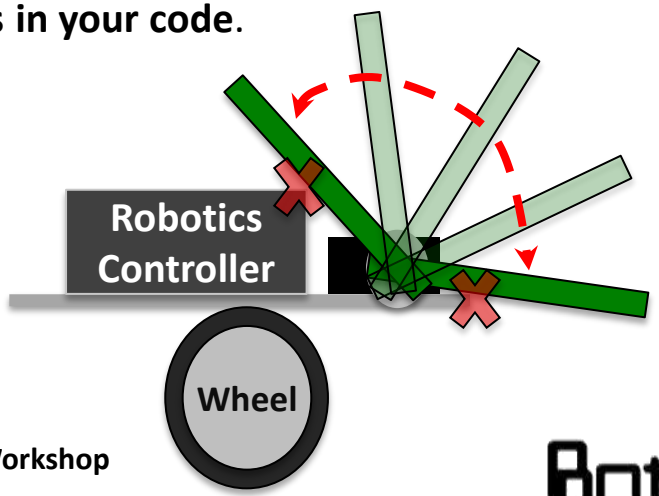
© 1993 – 2016 KIPR

Wave the servo arm

Activity

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

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



© 1993 – 2016 KIPR



Wave the servo arm

Activity

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

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.
```

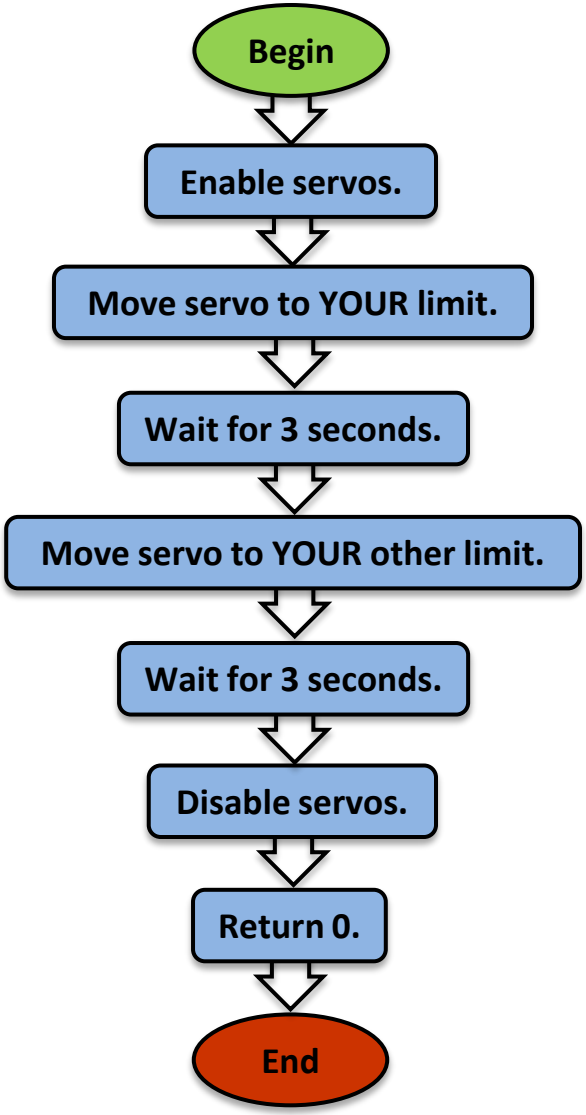

© 1993 – 2016 KIPR

Wave the servo arm

Activity

Analysis:

Flowchart





Wave the servo arm

Activity

Solution:

Pseudocode (Comments)

```
int main()
{
  // 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.
}
```

Source Code

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

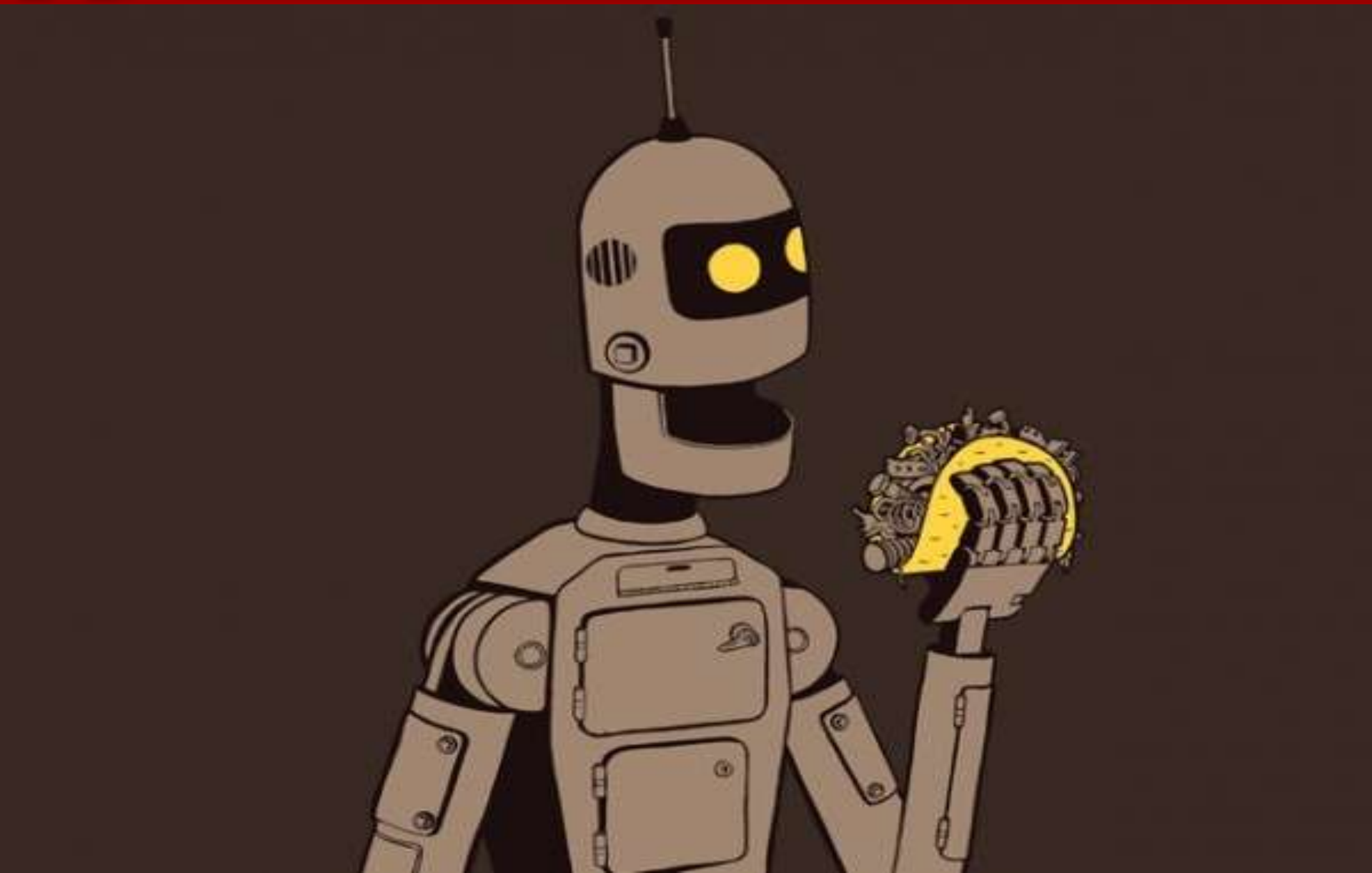
  // 7. End the program.
  return 0;
}
```

Use YOUR servo limits!

Execution: Compile and run your program on the KIPR Robotics Controller.

© 1993 – 2016 KIPR

Lunch!





Making Smarter Robots with Sensors

Analog and digital sensors

Light and touch sensors

`wait_for_touch` functions

© 1993 – 2016 KIPR



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?

© 1993 – 2016 KIPR

Analog and digital sensors

Analog Sensors

- **Range of values:**

  0 – 4095

- **Ports:** Analog 0-5
- **Function:** `analog(<port #>)`
- **Sensors:**
  - Light
  - Small reflectance
  - Large reflectance
  - "ET" Distance
  - Slide sensor







Digital Sensors

- **Range of values:**

  0 (not pressed) or 1 (pressed)

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





© 1993 – 2016 KIPR

Sensor and Motor Manual

Resource

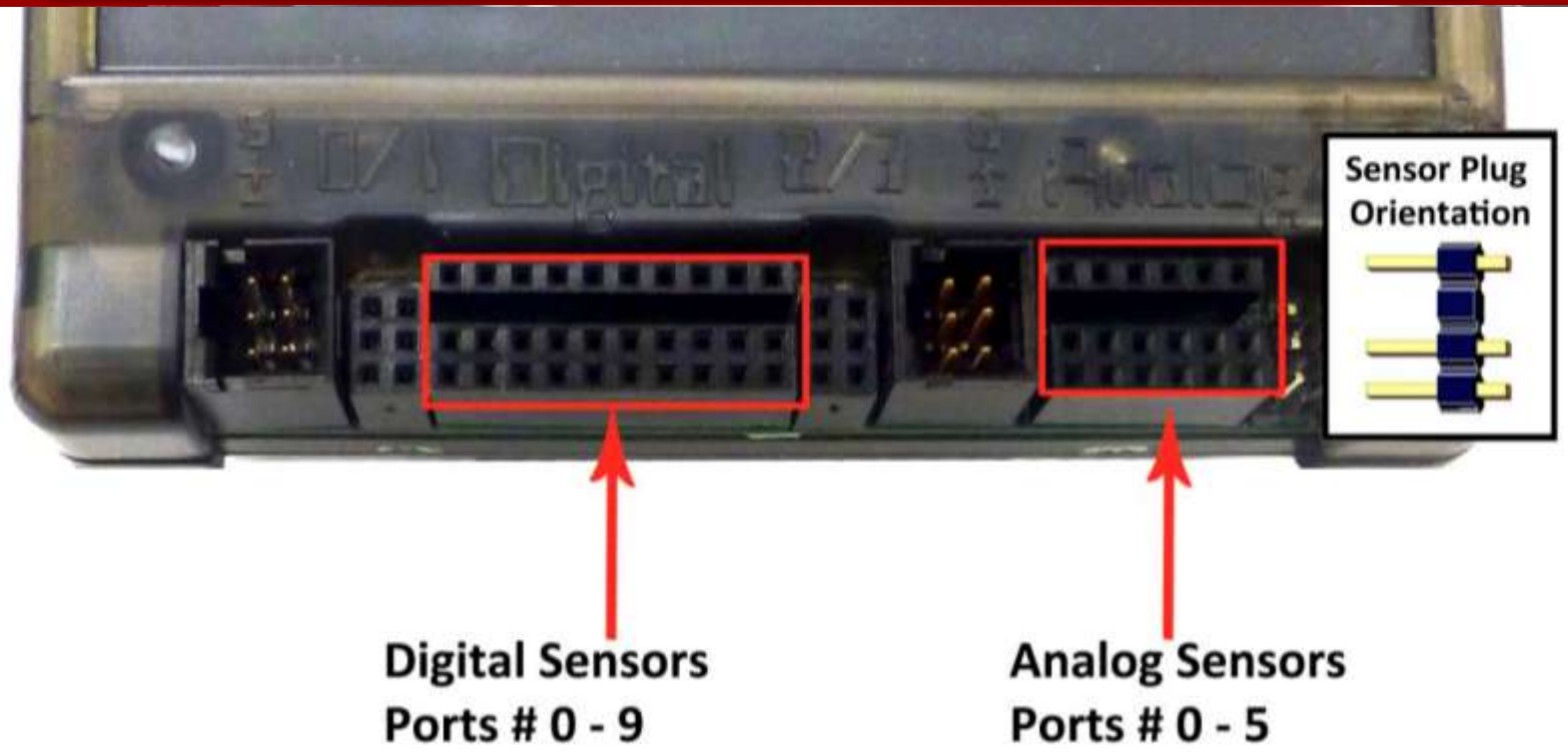
For additional details about **sensors**, consult the "**Sensor and Motor Manual**", available via the KISS IDE "**Help**" menu or from your **Team Home Base**.



© 1993 – 2016 KIPR

KIPR Robotics Controller sensor ports

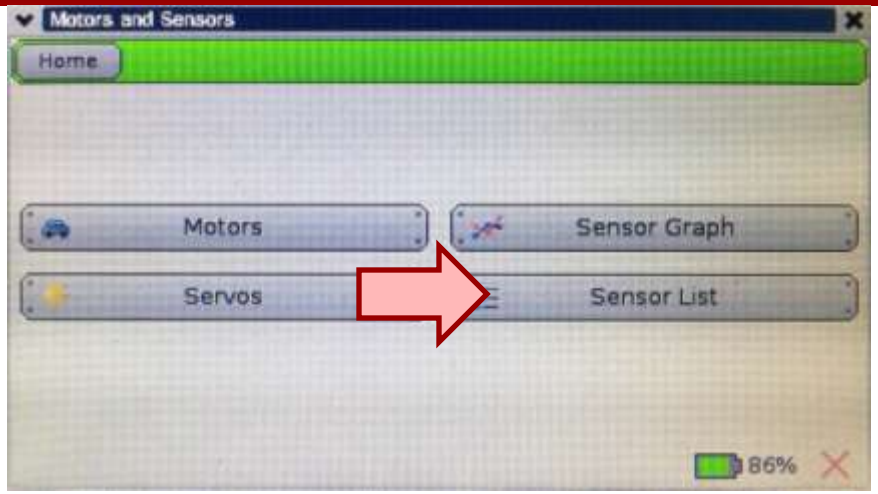
Activity

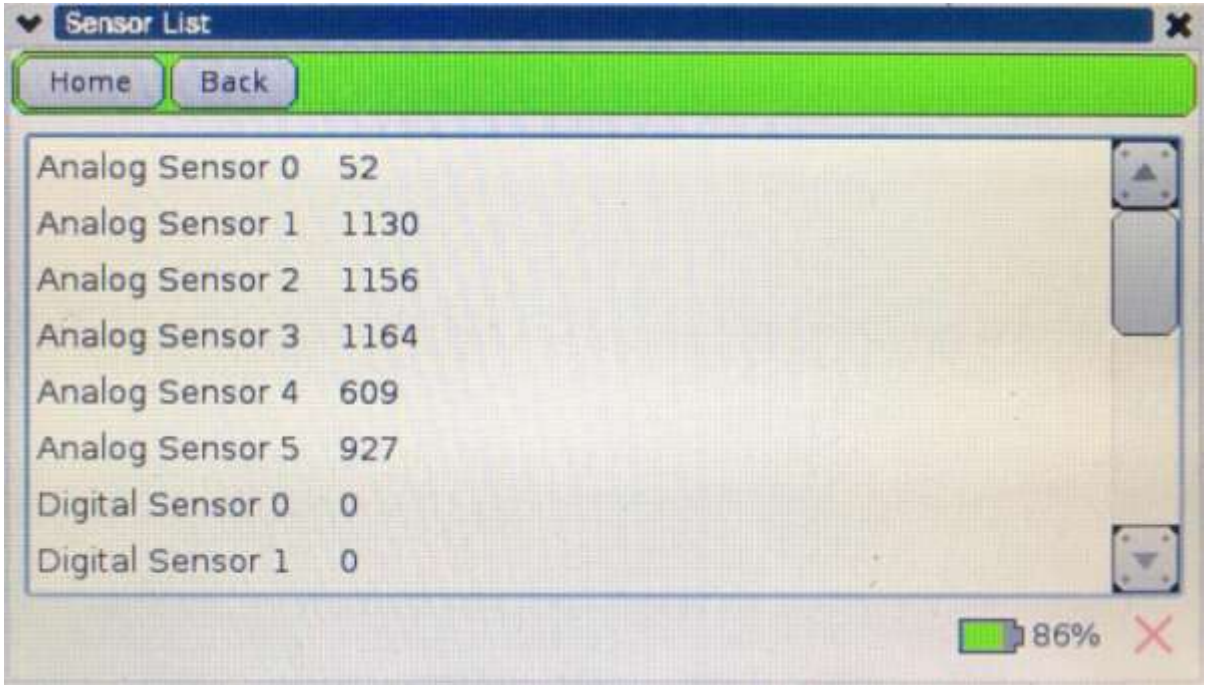


© 1993 – 2016 KIPR

Use the Sensor List



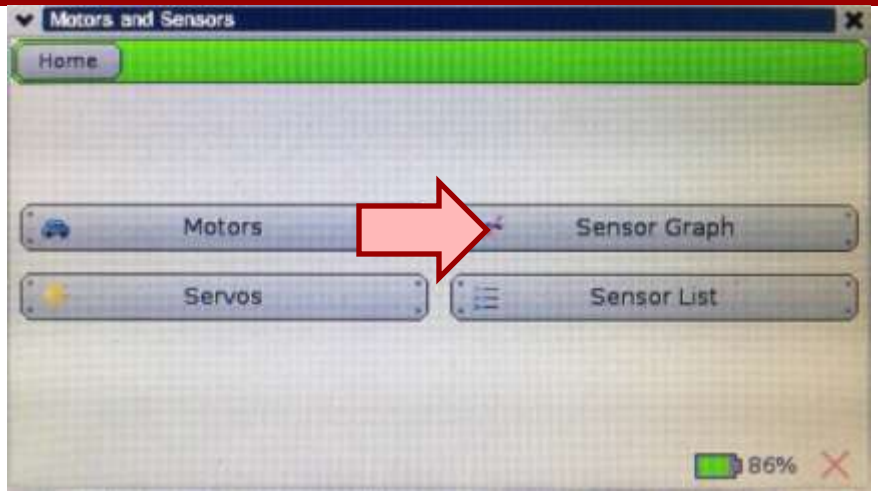


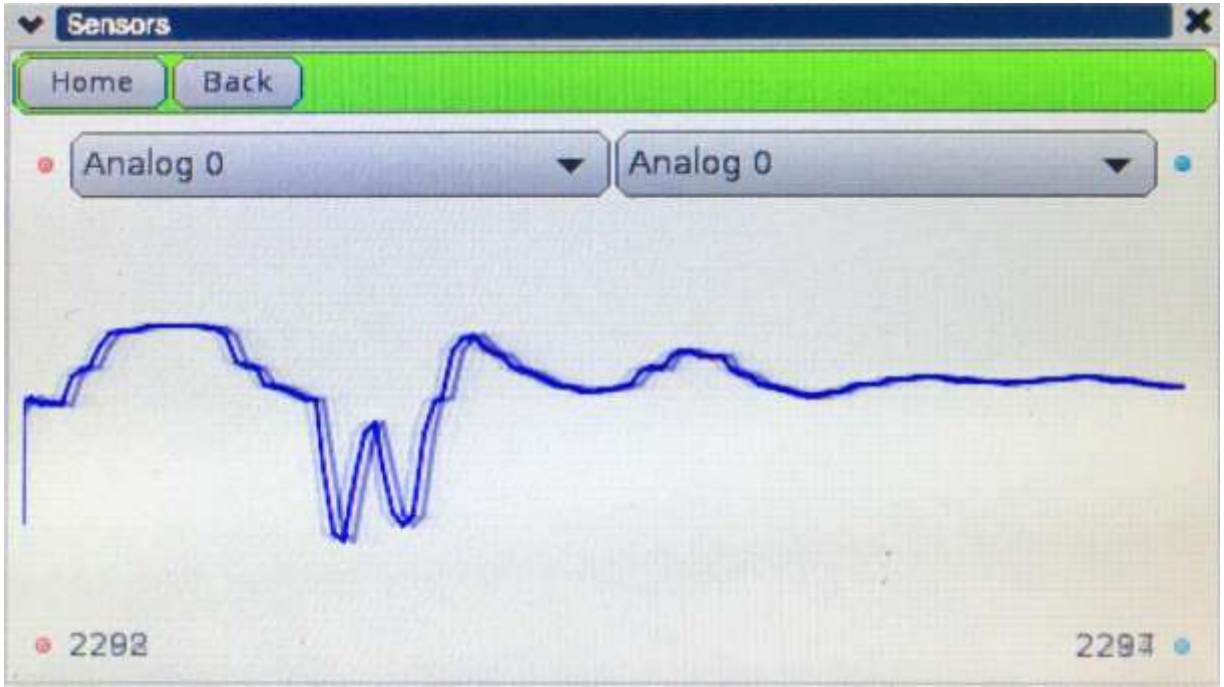


Botball®

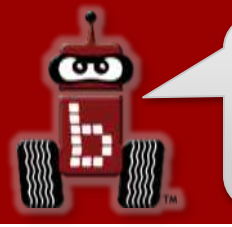
Use the Sensor Graph







Botball®



Sensor waiting functions

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

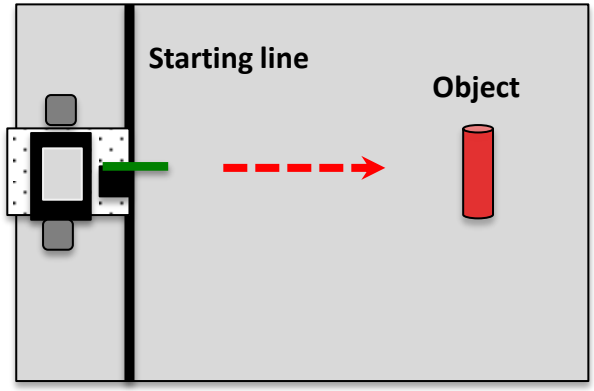

© 1993 – 2016 KIPR

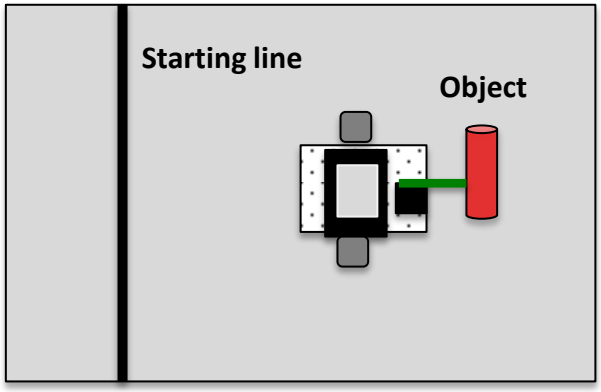
Touch an object

Activity

Description: Write a program for the KIPR Robotics Controller that drives the DemoBot forward until it touches an object

- Is there a way to detect when the object is touched? (see next slide)





© 1993 – 2016 KIPR

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**").

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



Large touch



Small touch



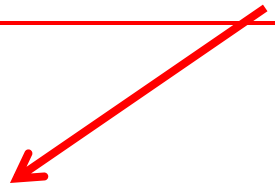
Lever touch

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

© 1993 – 2016 KIPR

Using wait_for_touch

What is this?



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


© 1993 – 2016 KIPR

Drive until touched

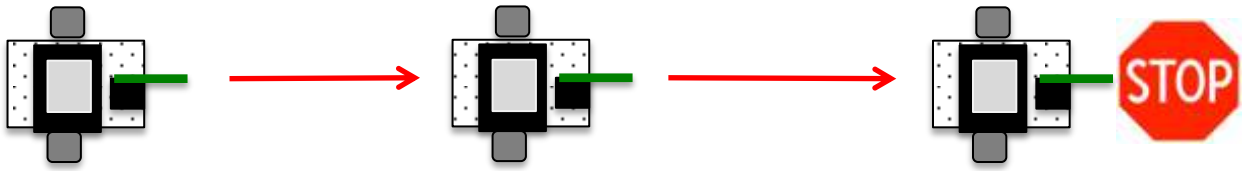
Activity

Description: Write a program for the KIPR Robotics Controller 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.





© 1993 – 2016 KIPR

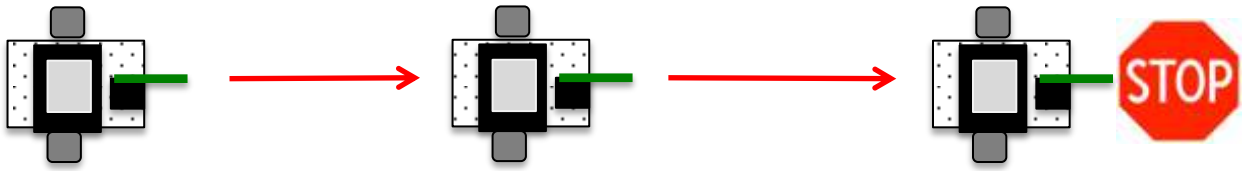
Drive until touched

Activity

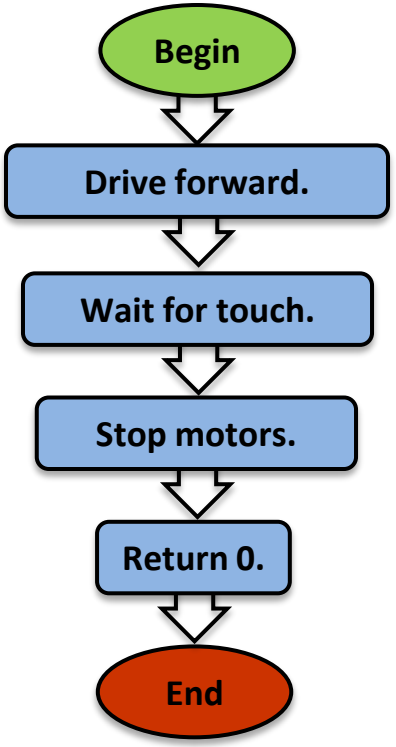
Description: Write a program for the KIPR Robotics Controller that drives the DemoBot forward until it detects a touch, then stops.

Analysis: What is the program supposed to do?

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



Flowchart



© 1993 – 2016 KIPR



Drive until touched

Activity

Solution:

Pseudocode (Comments)

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

Source Code

```
int main()
{
  // 1. Drive forward.
  mav(0, 1000);
  mav(2, 1000);

  // 2. Wait for touch.
  wait_for_touch(8);

  // 3. Stop motors.
  ao();

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

Execution: Compile and run your program on your robot.

© 1993 – 2016 KIPR



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 use a programming concept called a "**loop**" (see next section).

© 1993 – 2016 KIPR



Repetition, Repetition, Repetition

Program flow control with loops

`while` loops

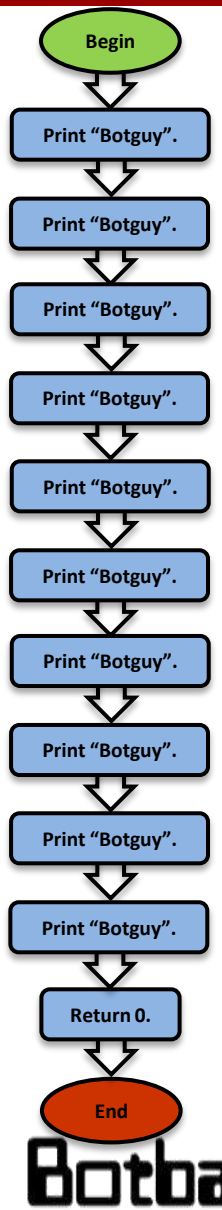
`while` and Boolean operators

© 1993 – 2016 KIPR

Program flow control with loops

Suppose your task is to print your name to the screen 10 times…

Pseudocode	Comments
1. Print “Botguy”.	`// 1. Print "Botguy".`
2. Print “Botguy”.	`// 2. Print "Botguy".`
3. Print “Botguy”.	`// 3. Print "Botguy".`
4. Print “Botguy”.	`// 4. Print "Botguy".`
5. Print “Botguy”.	`// 5. Print "Botguy".`
6. Print “Botguy”.	`// 6. Print "Botguy".`
7. Print “Botguy”.	`// 7. Print "Botguy".`
8. Print “Botguy”.	`// 8. Print "Botguy".`
9. Print “Botguy”.	`// 9. Print "Botguy".`
10. Print “Botguy”.	`// 10. Print "Botguy".`
11. End the program.	`// 11. End the program.`



© 1993 – 2016 KIPR

Botball®



Program flow control with loops

Now, suppose your objective is to print your name to the screen 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!)

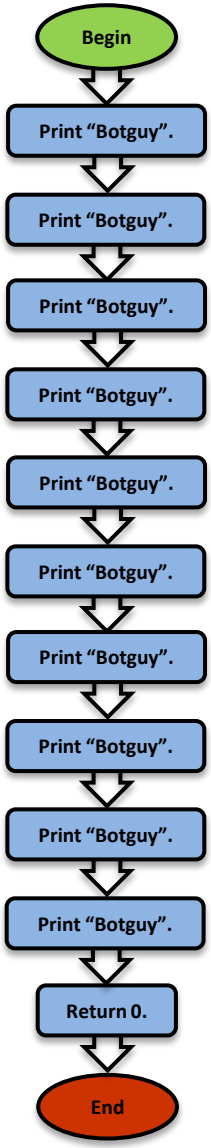
© 1993 – 2016 KIPR

Program flow control with loops

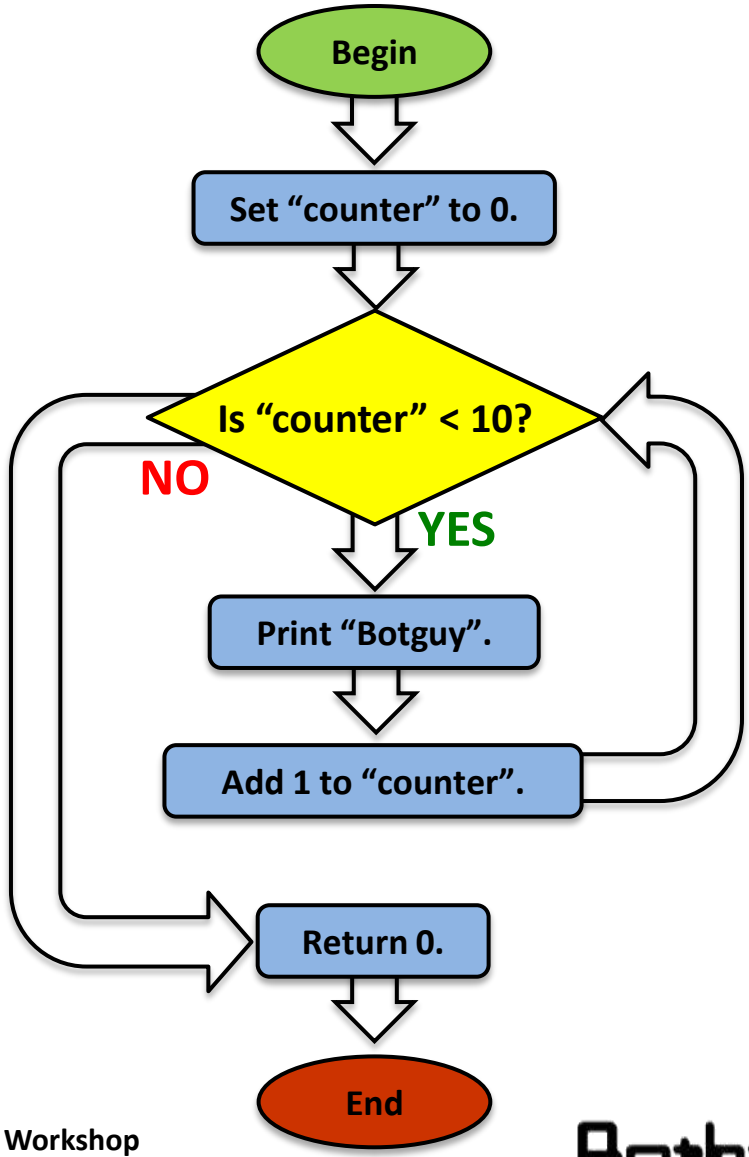
- What if we want to ***repeat*** the same **block of code** many times?
- We can do this using a "**loop**", which controls the "flow" of the program by repeating a **block of code**.

© 1993 – 2016 KIPR

Program flow control with loops



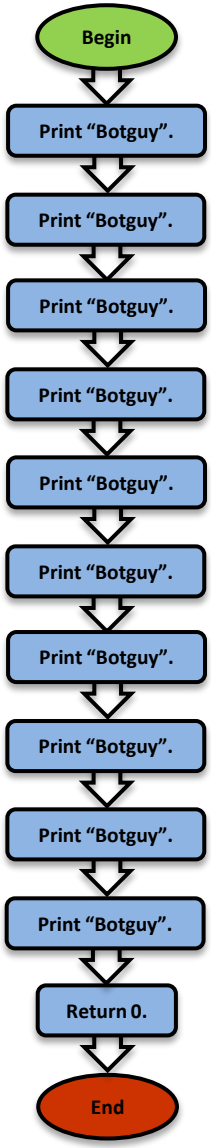
— VS —



Professional Development Workshop
© 1993 – 2016 KIPR

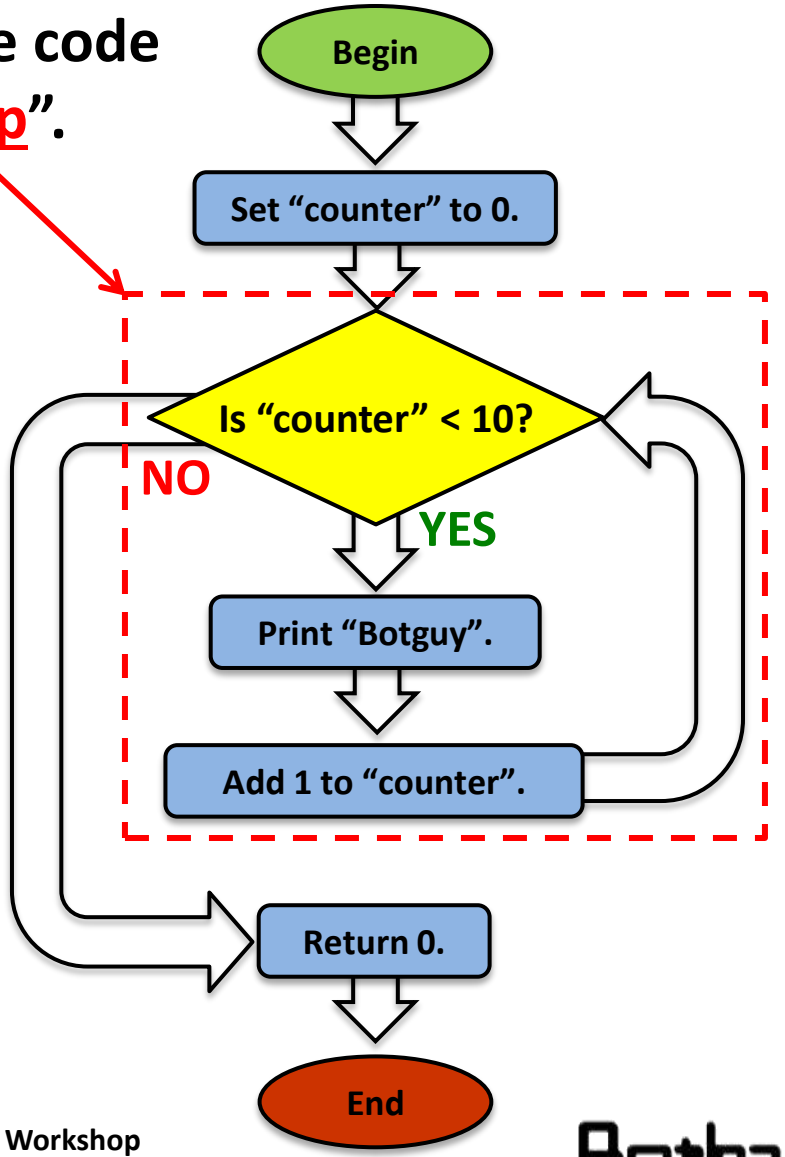
Botball®

Program flow control with loops



This part of the code is the “loop”.

— VS —



© 1993 – 2016 KIPR



Program flow control with loops

Old Code

Pseudocode	Comments
1. Print “Botguy”.	`// 1. Print "Botguy".`
2. Print “Botguy”.	`// 2. Print "Botguy".`
3. Print “Botguy”.	`// 3. Print "Botguy".`
4. Print “Botguy”.	`// 4. Print "Botguy".`
5. Print “Botguy”.	`// 5. Print "Botguy".`
6. Print “Botguy”.	`// 6. Print "Botguy".`
7. Print “Botguy”.	`// 7. Print "Botguy".`
8. Print “Botguy”.	`// 8. Print "Botguy".`
9. Print “Botguy”.	`// 9. Print "Botguy".`
10. Print “Botguy”.	`// 10. Print "Botguy".`
11. End the program.	`// 11. End the program.`

© 1993 – 2016 KIPR

Program flow control with loops

New Code

Pseudocode

1. Set “counter” to 0.
2. Loop: Is “counter” < 10?
    1. Print “Botguy”.
    2. Add 1 to “counter”.
3. End the program.

Comments

```
//  1. Set "counter" to 0.
//  2. Loop: Is "counter" < 10?
//      2.1. Print "Botguy".
//      2.2. Add 1 to "counter".
//  3. End the program.
```

In the **C** programming language, we accomplish this with a `while` **loop**.

In Botball, we are less concerned with *printing*.
Let’s try an example with **sensors** and **motors**…

© 1993 – 2016 KIPR

Using while loops

What is this?

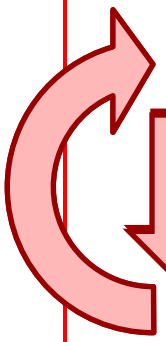
What does this say?

```
int main()
{
  while (digital(8) != 1)
  {
    mav(0, 1000);
    mav(2, 1000);
  }
  ao();
  return 0;
}
```


© 1993 – 2016 KIPR

Using while loops

While the touch sensor is not pressed (!= 1), move forward.



```
int main()
{
  while (digital(8) != 1)
  {
    mav(0, 1000);
    mav(2, 1000);
  }
  ao();
  return 0;
}
```

What does this say?

© 1993 – 2016 KIPR

Using while loops

```
int main()
{
  while (digital(8) != 1)
  {
    mav(0, 1000);
    mav(2, 1000);
  }
  ao();
  return 0;
}
```

Notice: no semicolon!
(Why not?)

© 1993 – 2016 KIPR



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 ...

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

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


© 1993 – 2016 KIPR



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 ...

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

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

Begin

End

Block Header
(no semicolon!)

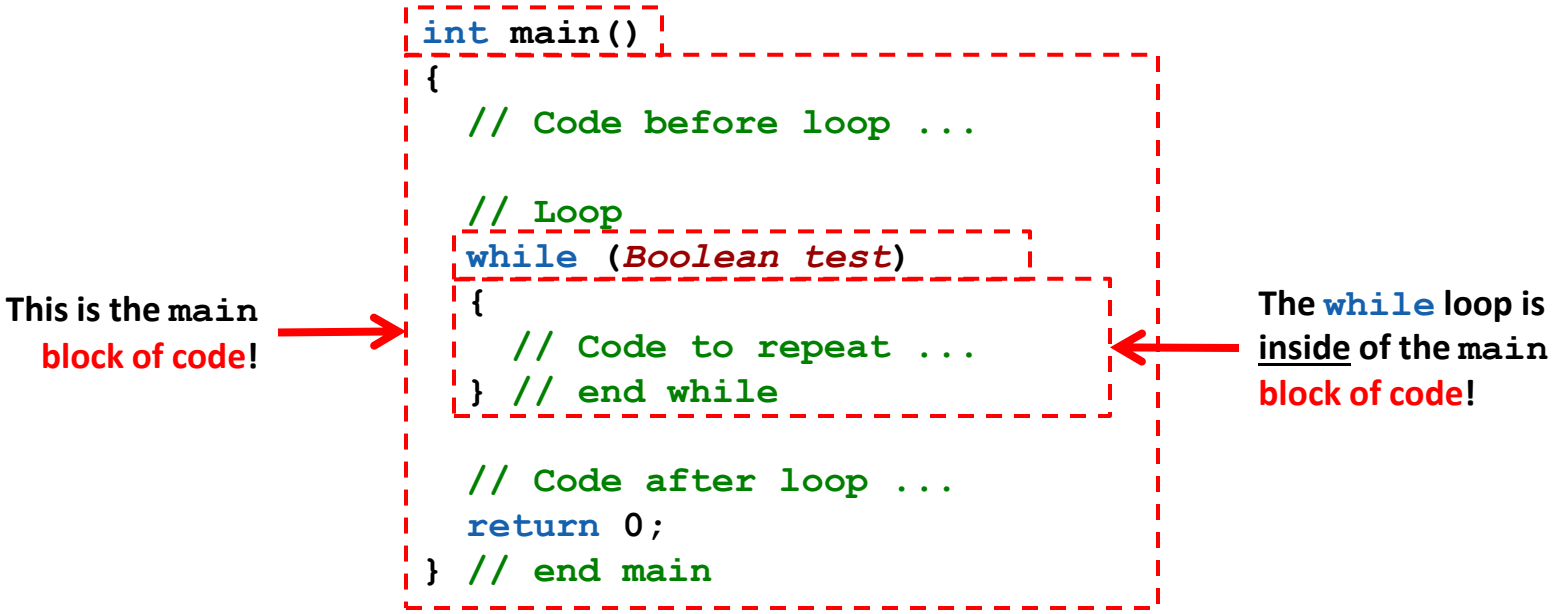
Block of Code

© 1993 – 2016 KIPR

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 ...

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

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


© 1993 – 2016 KIPR



`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

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

© 1993 – 2016 KIPR

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

© 1993 – 2016 KIPR



while and Boolean operators

```
// Repeat while the digital touch sensor on port #15 is equal to 0
// (i.e., while not pressed).
while (digital(9) == 0)
{
  // Code to repeat ...
}

// Repeat while the digital touch sensor on port #15 is not equal to 1
// (i.e., while not pressed).
// Note: this would have the same result as the loop above.
while (digital(9) != 1)
{
  // Code to repeat ...
}

// Repeat while analog sensor on port #3 reads less than 2024; = 512 if using analog10(3).
while (analog(3) < 2024)
{
  // Code to repeat ...
}

// Repeat while "countdown" (a variable) is greater than or equal to 1.
while (countdown >= 1)
{
  // Code to repeat ...
}
```


© 1993 – 2016 KIPR



Drive until sensor is pressed

Activity

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

Analysis: What is the program supposed to do?

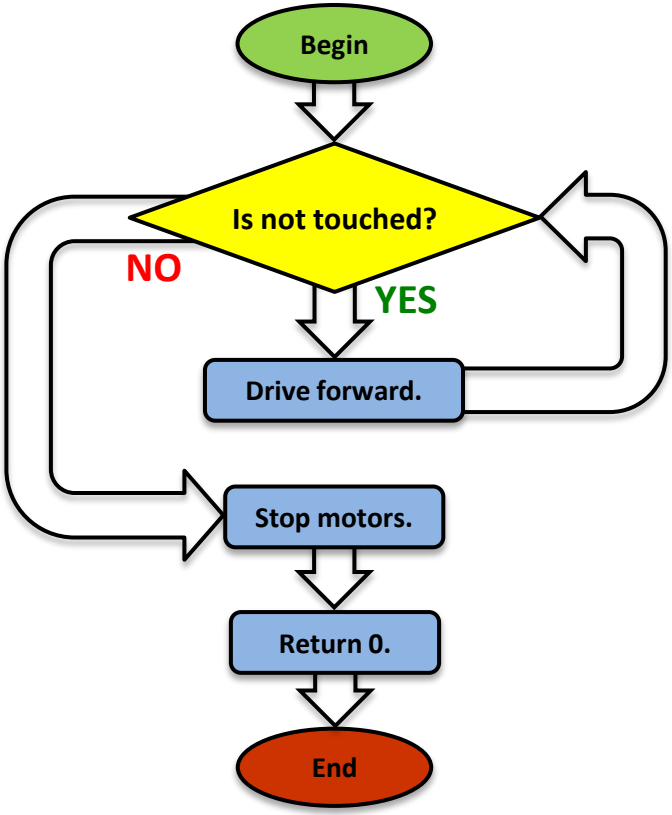
Pseudocode	Comments
1. Drive forward.	`// 1. Drive forward.`
2. Loop: Is not touched?	`// 2. Loop: Is not touched?`
1. Print message.	`//  2.1. Print message.`
3. Stop motors.	`// 3. Stop motors.`
4. End the program.	`// 4. End the program.`

© 1993 – 2016 KIPR

Drive until sensor is pressed

Activity

Analysis: Flowchart



© 1993 – 2016 KIPR



Drive until sensor is pressed

Activity

Solution:

Pseudocode (Comments)

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

Source Code

```
int main()
{
  // 1. Loop: Is not touched?
  while (digital(8) != 1)
  {
    // 1.1. Drive forward.
    mav(0, 1000);
    mav(2, 1000);
  } // end while

  // 2. Stop motors.
  ao();

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


© 1993 – 2016 KIPR



Draw a square using a loop

Activity

Description: Write a program for the KIPR Robotics Controller 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 “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

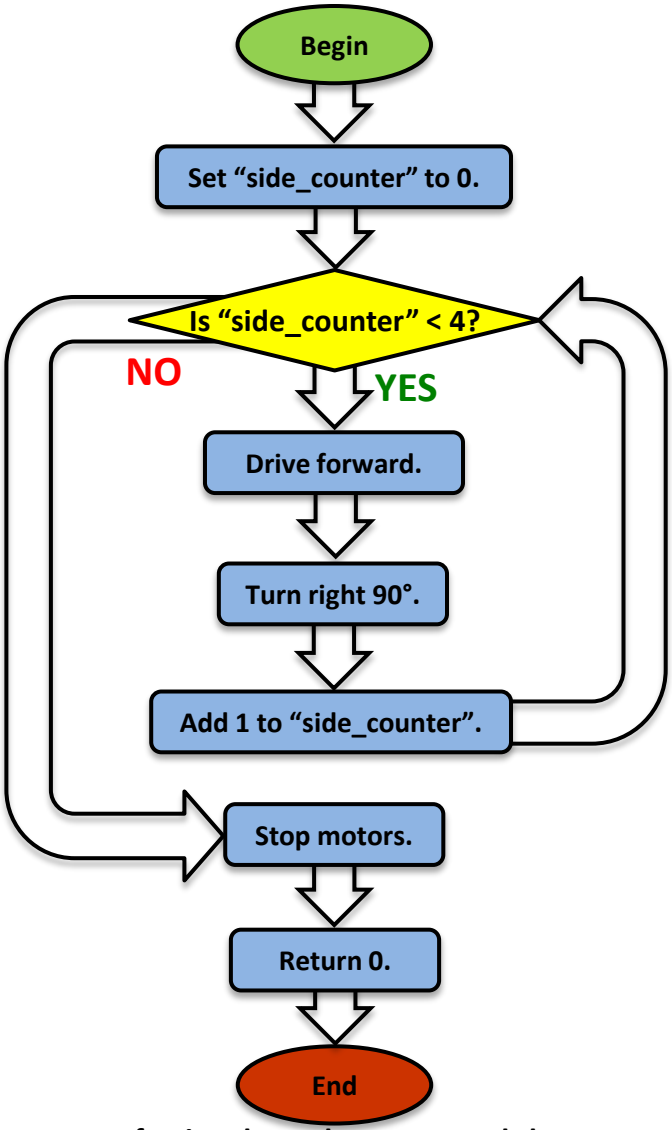
```
// 1. Set "side_counter" to 0.
// 2. Loop: Is "side_counter" < 4?
//    2.1. Drive forward.
//    2.2. Turn right 90-degrees.
//    2.3. Add 1 to "side_counter".
// 3. Stop motors.
// 4. End the program.
```


© 1993 – 2016 KIPR

Draw a square using a loop

Activity

Analysis: Flowchart



© 1993 – 2016 KIPR



Draw a square using a loop

Activity

Solution:

Pseudocode (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()
{
  // 1. Set "side_counter" to 0.
  int side_counter = 0;

  // 2. Loop: Is "side_counter" < 4?
  while (side_counter < 4)
  {
    // 2.1. Drive forward.
    mav(0, 1000);
    mav(2, 1000);
    wait_for_milliseconds(4000);

    // 2.2. Turn right 90-degrees.
    mav(0,  500);
    mav(2, -500);
    wait_for_milliseconds(1500);

    // 2.3. Add 1 to "side_counter".
    side_counter = side_counter + 1;
  } // end while

  // 3. Stop motors.
  ao();

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


© 1993 – 2016 KIPR



Move the servo arm using a loop

Activity

Description: Write a program for the KIPR Robotics Controller that moves the DemoBot servo arm from position 200 to 1800 in increments of 100.

- Remember to **enable the servos** at the beginning of your program, and **disable the servos** at the end of your program!

Analysis: What is the program supposed to do?

Pseudocode

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

Comments

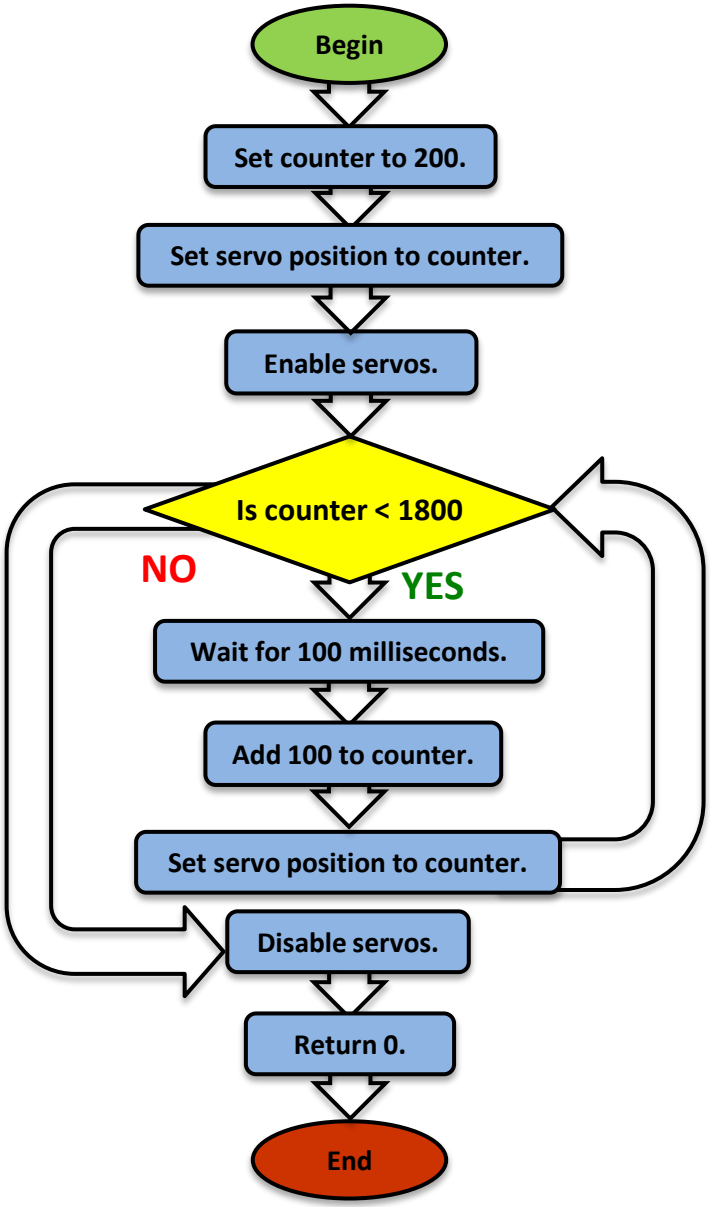
```
// 1. Set counter to 200
// 2. Set servo position to counter
// 3. Enable servos.
// 4. Loop: Is counter < 1800?
//    4.1. Wait for 100 milliseconds.
//    4.2. Add 100 to servo position.
//    4.3 Set servo position to counter.
// 5. Disable servos.
// 6. End the program.
```


© 1993 – 2016 KIPR

Move the servo arm using a loop

Activity

Analysis: Flowchart



Botball®



Move the servo arm using a loop

Activity

Solution:

Pseudocode (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()
{
  // 1. Set counter to 200.
  int counter;
  counter = 200;

  // 2. Set servo_position to counter.
  set_servo_position(0, counter);

  // 3. Enable servos.
  enable_servos();

  // 4. Loop: Is servo position < 1800?
  while (counter < 1800)
  {
    // 4.1. Wait for 0.1 seconds.
    wait_for_milliseconds(100);

    // 4.2. Add 100 to servo position.
    counter = counter + 100;

    // 4.3 Set servo position to counter.
    set_servo_position(0, counter);
  } // end while

  // 5. Disable servos.
  disable_servos();

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



Fun with Functions

Writing your own functions

Function prototypes, definitions, and calls

© 1993 – 2016 KIPR

Draw a square

Activity

Reflection:

Notice there are many repeated steps.
For example:

```
// Drive forward.
mav(0, 1000);
mav(0, 1000);
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()
{
  // 1. Drive forward.                    Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 2. Turn right 90-degrees.            Turn right.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 3. Drive forward.                    Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 4. Turn right 90-degrees.            Turn right.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 5. Drive forward.                    Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 6. Turn right 90-degrees.            Turn right.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 7. Drive forward.                    Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 8. Turn right 90-degrees.            Turn right.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  ao();       // 9. Stop motors.
  rurn 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!

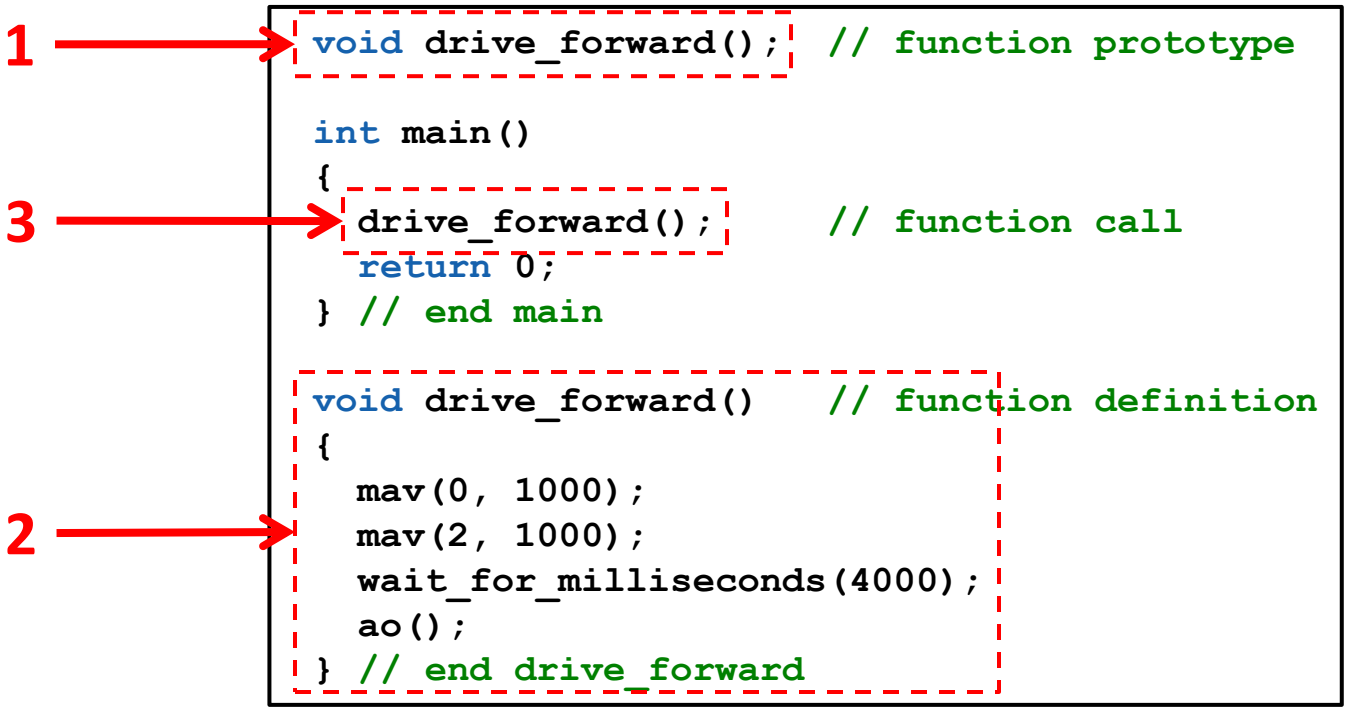
You can write your own functions to do whatever you want!

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

© 1993 – 2016 KIPR

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



© 1993 – 2016 KIPR

Writing your own functions

Function prototypes go above `main`.

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

Function definitions go below `main`.

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


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


void drive_forward()   // function definition
{
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```


© 1993 – 2016 KIPR



Writing your own functions

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

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



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



void drive_forward()   // function definition
{
  mav(0, 100);
  mav(2, 100);
  wait_for_milliseconds(4000);
  ao();
} // end drive_forward
```

Notice: no semicolon!
(Why not?)

© 1993 – 2016 KIPR



Writing your own functions

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


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


void drive_forward()    // function definition
{
  mav(0, 1000);
  mav(2, 1000);
  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…

© 1993 – 2016 KIPR

Writing your own functions

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


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


void drive_forward()   // function definition
{
  mav(0, 1000);
  mav(2, 1000);
  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.

© 1993 – 2016 KIPR

Writing your own functions

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



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


void drive_forward()    // function definition
{
  mav(0, 1000);
  mav(2, 1000);
  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.

© 1993 – 2016 KIPR



Writing your own functions

```
// function prototypes
void drive_forward_turn();

int main()
{
  drive_forward_turn();     // drive forward & turn right function call
  return 0;
} // end main

void drive_forward_turn()  // drive_forward_turn function definition
{
  mav(0, 1000);
  mav (2, 1000);
  wait_for_milliseconds(4000); // go straight
  mav (0,  500); // left motor forward
  mav (2, -500); // right motor backwards
  wait_for_milliseconds(1500); // turn right
  ao(); // turn off the motors
} // end drive_forward_turn
```

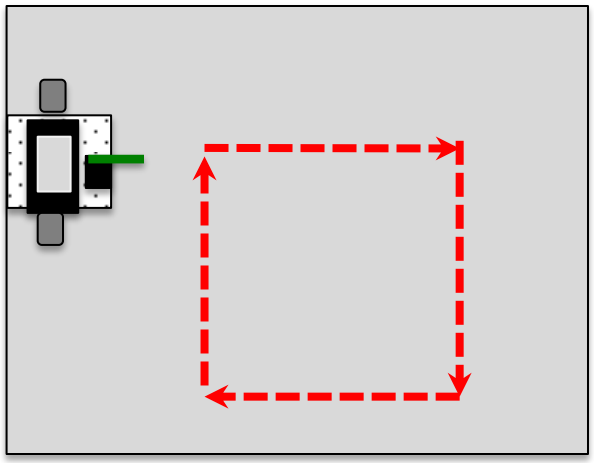

© 1993 – 2016 KIPR

Draw a square using functions

Activity

Description: Write a program for the KIPR Robotics Controller 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!



© 1993 – 2016 KIPR

Activity

Code without your functions

```
int main()
{
  // 1. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 2. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 3. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 4. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 5. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 6. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

  // 7. Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // 8. Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  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.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

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

Draw a square using a loop

Activity

Solution: (using a function & loop)

Pseudocode (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

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

// Function definition for main.
int main()
{
  // 1. Set "side_counter" to 0.
  int side_counter = 0;

  // 2. Loop: Is "side_counter" < 4?
  while (side_counter < 4)
  {
    // 2.1. Drive forward.
    // 2.2. Turn right 90-degrees.
    drive_forward_and_turn_right();

    // 2.3. Add 1 to "side_counter".
    side_counter = side_counter + 1;
  } // end while

  // 3. Stop motors.
  ao();

  // 4. End the program.
  return 0;
} // end main

// Function definition for drive_forward_and_turn_right.
void drive_forward_and_turn_right()
{
  // Drive forward.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // Turn right 90-degrees.
  mav(0, 500);
  mav(2, -500);
  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 to 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`**.

```
// 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.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(4000);

  // Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

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



Variables and Functions with Arguments

Data types

Creating and setting a variable

Variable arithmetic

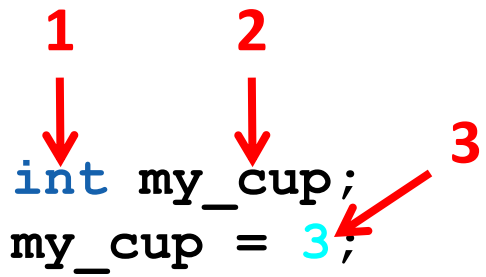
Functions with arguments and return values

© 1993 – 2016 KIPR

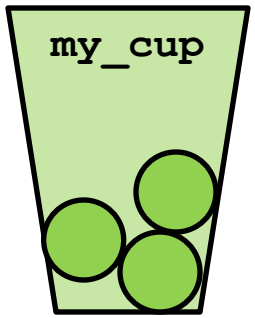
Variables

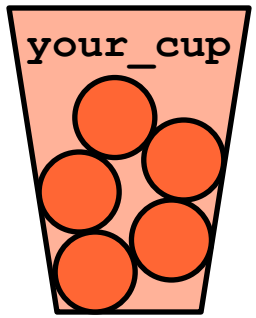
- A "**variable**" is a ***named*** container that stores a **type** of **value**.

- A **variable** has the following three components:
  1. the **type** of data it stores (holds),
  2. the **name**, and
  3. the **value**.



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





© 1993 – 2016 KIPR

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**.

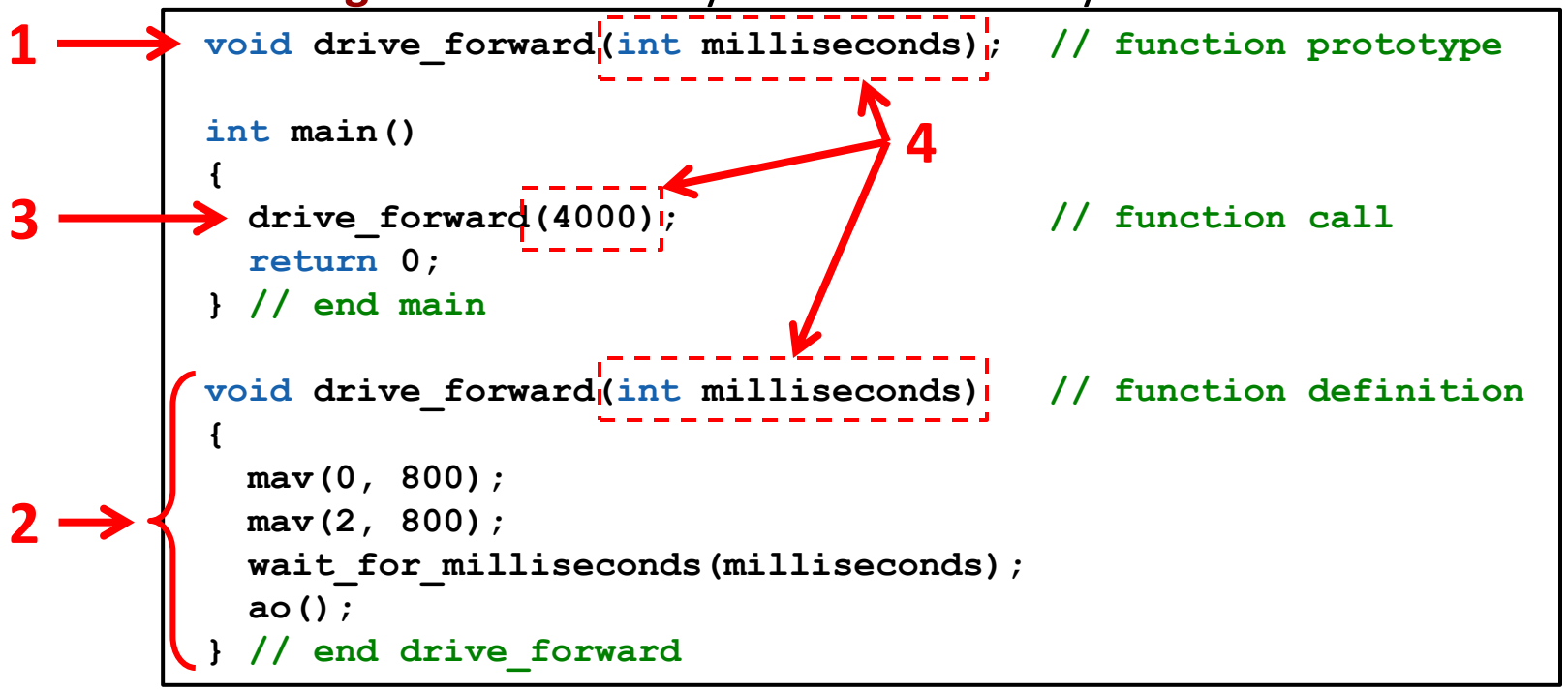
An Example:

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


© 1993 – 2016 KIPR

Functions with arguments

- There are **four components** to a function *with arguments*:
  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
  4. **Function arguments:** values you will set when you call the function





Writing your own functions with arguments

Function prototypes go above `main`.

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

Function definitions go below `main`.

```
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
{
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(milliseconds);
  ao();
} // end drive_forward
```


© 1993 – 2016 KIPR

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
{
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(milliseconds);
  ao();
} // end drive_forward
```

Notice: no semicolon!
(Why not?)

© 1993 – 2016 KIPR

Writing your own functions with arguments

```
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
{
  mav(0, 1000);
  mav(2, 1000);
  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.

© 1993 – 2016 KIPR



Writing your own functions with arguments

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



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


void drive_forward(int velocity, int milliseconds)  // function definition
{
  mav(0, velocity);
  mav(2, velocity);
  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.

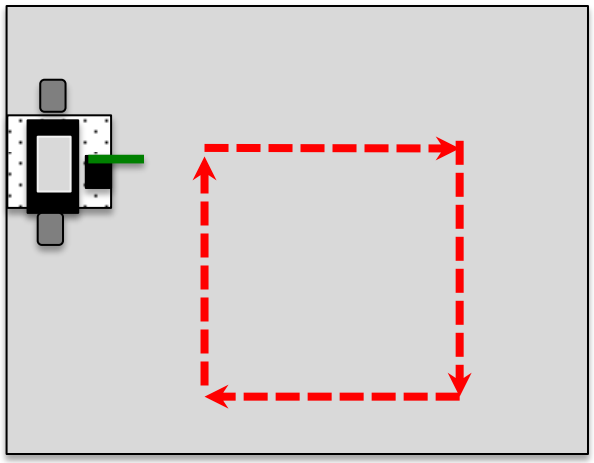
© 1993 – 2016 KIPR

Draw squares of different sizes

Activity

Description: Write a program for the KIPR Robotics Controller k that drives the DemoBot along a path in the shape of squares of different side lengths *using functions*.

- **Hint:** modify your old square-drawing program to use your own functions that include an argument that influences the length of the square side(s).
- **First:** try modifying your old function by adding one argument.
- **Next:** try modifying your function by adding a second argument.



© 1993 – 2016 KIPR



Draw squares of different sizes

Activity

Solution:

Function with one argument.

```
// Function prototype (with one argument)
// for drive_forward_and_turn_right.
void drive_forward_and_turn_right(int milliseconds);


// Function definition for main.
int main()
{
  // Four function calls (with one argument)
  // for drive_forward_and_turn_right.
  drive_forward_and_turn_right(5000);
  drive_forward_and_turn_right(5000);
  drive_forward_and_turn_right(5000);
  drive_forward_and_turn_right(5000);
  return 0;
} // end main


// Function definition (with one argument)
// for drive_forward_and_turn_right.
void drive_forward_and_turn_right(int milliseconds)
{
  // Drive forward
  // for the specified number of milliseconds.
  mav(0, 1000);
  mav(2, 1000);
  wait_for_milliseconds(milliseconds);

  // Turn right 90-degrees.
  mav(0,  500);
  mav(2, -500);
  wait_for_milliseconds(1500);

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



Draw rectangles of different sizes

Activity

Solution:

Function with two arguments.

```
// Function prototype (with two arguments)
// for drive_forward_and_turn_right.
void drive_forward_and_turn_right(int velocity, int milliseconds);


// Function definition for main.
int main()
{
  // Four function calls (with two arguments)
  // for drive_forward_and_turn_right. Draw a Rectangle
  drive_forward_and_turn_right(500, 5000); // long side
  drive_forward_and_turn_right(200, 5000); // short side
  drive_forward_and_turn_right(500, 5000); // long side
  drive_forward_and_turn_right(200, 5000); // short side
  return 0;
} // end main


// Function definition (with two arguments)
// for drive_forward_and_turn_right.
void drive_forward_and_turn_right(int velocity, int milliseconds)
{
  // Drive forward at the specified power
  // for the specified number of milliseconds.
  mav(0, velocity);
  mav(2, velocity);
  wait_for_milliseconds(milliseconds);

  // Turn right 90-degrees.
  mav(0, velocity);
  mav(2, -1*velocity);
  wait_for_milliseconds(1500);

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



Making a Choice

Program flow control with conditionals

`if-else` conditionals

`if-else` and Boolean operators

Using `while` and `if-else`

© 1993 – 2016 KIPR



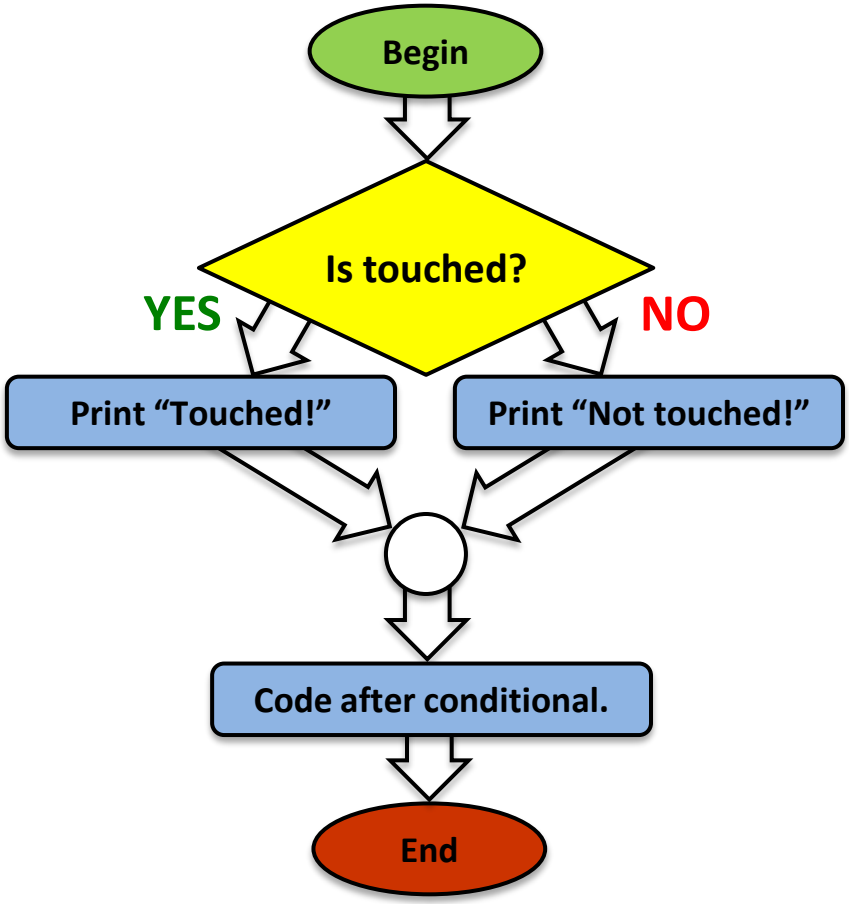
Program flow control with conditionals

- 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**.

© 1993 – 2016 KIPR

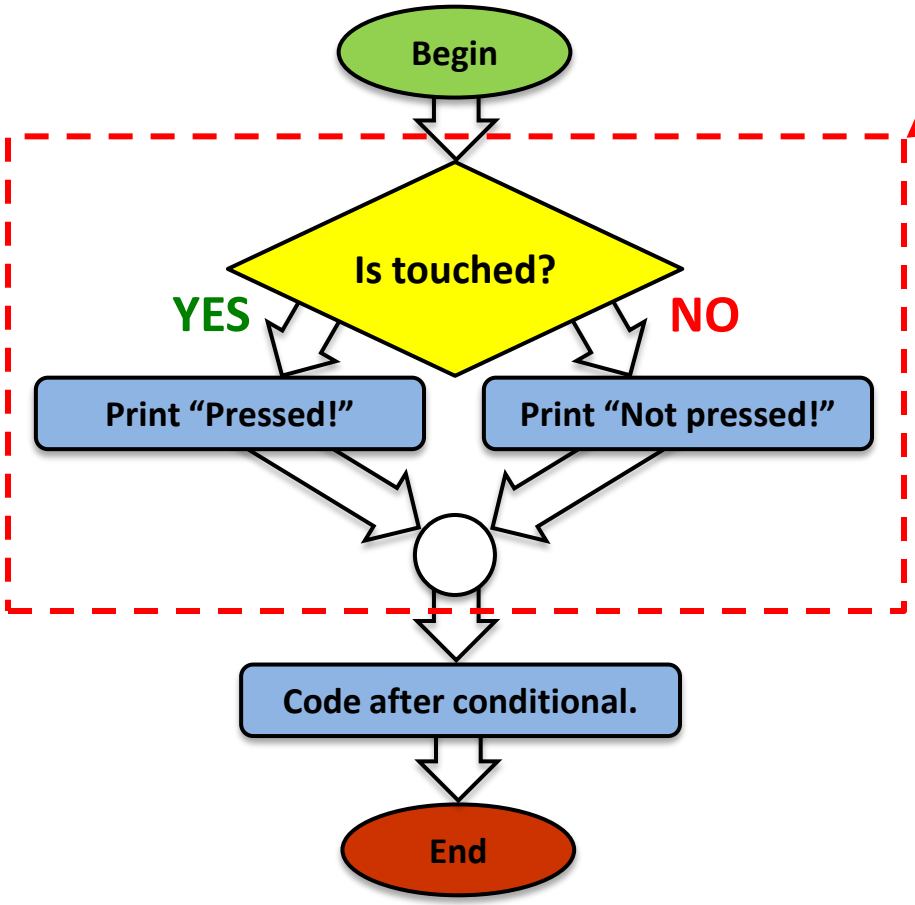
Program flow control with conditionals



© 1993 – 2016 KIPR

Program flow control with conditionals

This part of the code is the "conditional".



© 1993 – 2016 KIPR



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**.

© 1993 – 2016 KIPR

Using `if-else` conditionals

What is this?

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

What does this say?

© 1993 – 2016 KIPR



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?)

© 1993 – 2016 KIPR



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()
{
  // Code before conditional ...

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

  // Code after conditional ...
  return 0;
} // end main
```


© 1993 – 2016 KIPR

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()
{
  // Code before conditional ...

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

  // Code after conditional ...
  return 0;
} // end main
```

Begin

End

Block Header
(no semicolon!)

Block of Code

© 1993 – 2016 KIPR

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**.

The `else` is below the `}` brace of the `if` block of code!

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

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

  // Code after conditional ...
  return 0;
} // end main
```


© 1993 – 2016 KIPR

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()
{
  // Code before conditional ...

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

  // Code after conditional ...
  return 0;
} // end main
```

Block Header
(no semicolon!)

Begin

End

Block of Code

© 1993 – 2016 KIPR



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**.

If this Boolean test is *true*… → **… then the if block is executed!**

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

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

  // Code after conditional ...
  return 0;
} // end main
```


© 1993 – 2016 KIPR



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**.

If this Boolean test is *false*… →

… then the else block is executed!

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

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

  // Code after conditional ...
  return 0;
} // end main
```


© 1993 – 2016 KIPR



`if-else` and Boolean operators

The "**Boolean test**" in an `if-else` conditional is asking a question:

Is this statement true or false?

- The "**Boolean test**" (condition) often compares two values to one another using a "**Boolean operator**", such as:
  - `==` Equal to
  - `!=` Not equal to
  - `<` Less than
  - `>` Greater than
  - `<=` Less than or equal to
  - `>=` Greater than or equal to

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

© 1993 – 2016 KIPR



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

© 1993 – 2016 KIPR



if-else and Boolean operators

```
// Execute if the touch sensor on port #12 is equal to 0
// (i.e., if not pressed).
if (digital(4) == 0)
{
  // Code to execute ...
}
else  // (i.e., if pressed)
{
  // Code to execute ...
}


// Execute if the analog sensor on port #3 reads less than 2048.
if (analog(3) < 2048)
{
  // Code to execute ...
}
else  // (i.e., analog(3) >= 2048)
{
  // Code to execute ...
}
```


© 1993 – 2016 KIPR

Using while and if-else

```
int main()
{
  while (digital(8) != 1)
  {
    if (analog(0) > 2048)
    {
      printf("It's dark in here!\n");
    }
    else
    {
      printf("I see the light!\n");
    }
  }
  return 0;
}
```

What do these lines of code say?

© 1993 – 2016 KIPR



Using while and if-else

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

```
int main()
{
  while (digital(8) != 1)
  {
    if (analog(0) > 512)
    {
      printf("It's dark in here!\n");
    }
    else
    {
      printf("I see the light!\n");
    }
  }
  return 0;
}
```


© 1993 – 2016 KIPR



Using while and if to print values

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

```
int main()
{
  while (digital(13) != 1)// R button, Press to exit
  {
    if (digital(0) == 1) // Button is pressed print value
    {
      // an int placeholder for printf is %d
      printf("Light Sensor value = %d\n", analog(0));
    }
    wait_for_milliseconds(50); // don't print too fast
  } // end while
  printf("R Button is pressed. Program Ends.\n" );
  return 0;
} // end main
```


© 1993 – 2016 KIPR



Line-following

Reflectance sensor

Using `while` loops and `if-else` conditionals

© 1993 – 2016 KIPR

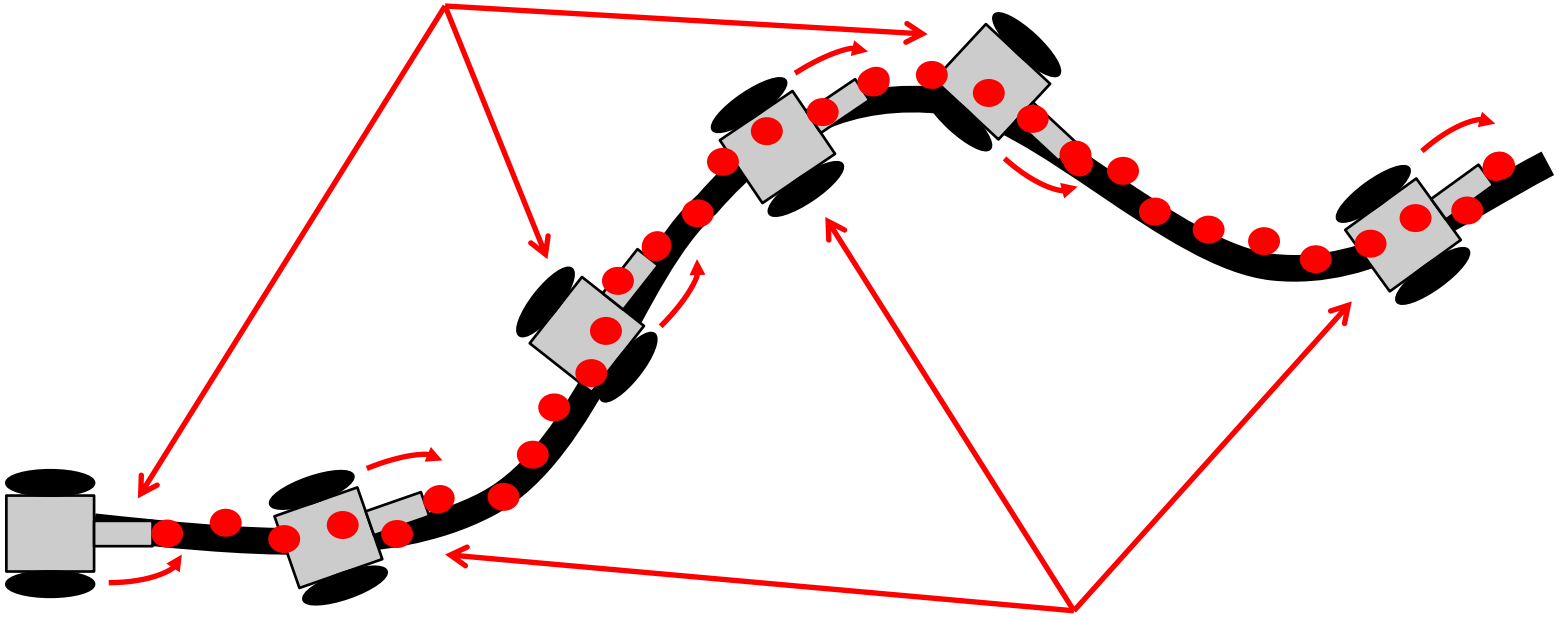
Line-following

Activity

Description: Write a program for the KIPR Robotics Controller that makes the DemoBot follow a line, and stops when the touch sensor is touched.

- Follow the line's **left edge** by *alternating* the following 2 actions.

1. If detecting *dark*, then turn/arc *left*.



2. Else it is *not dark*, then turn/arc *right*.

© 1993 – 2016 KIPR

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**.



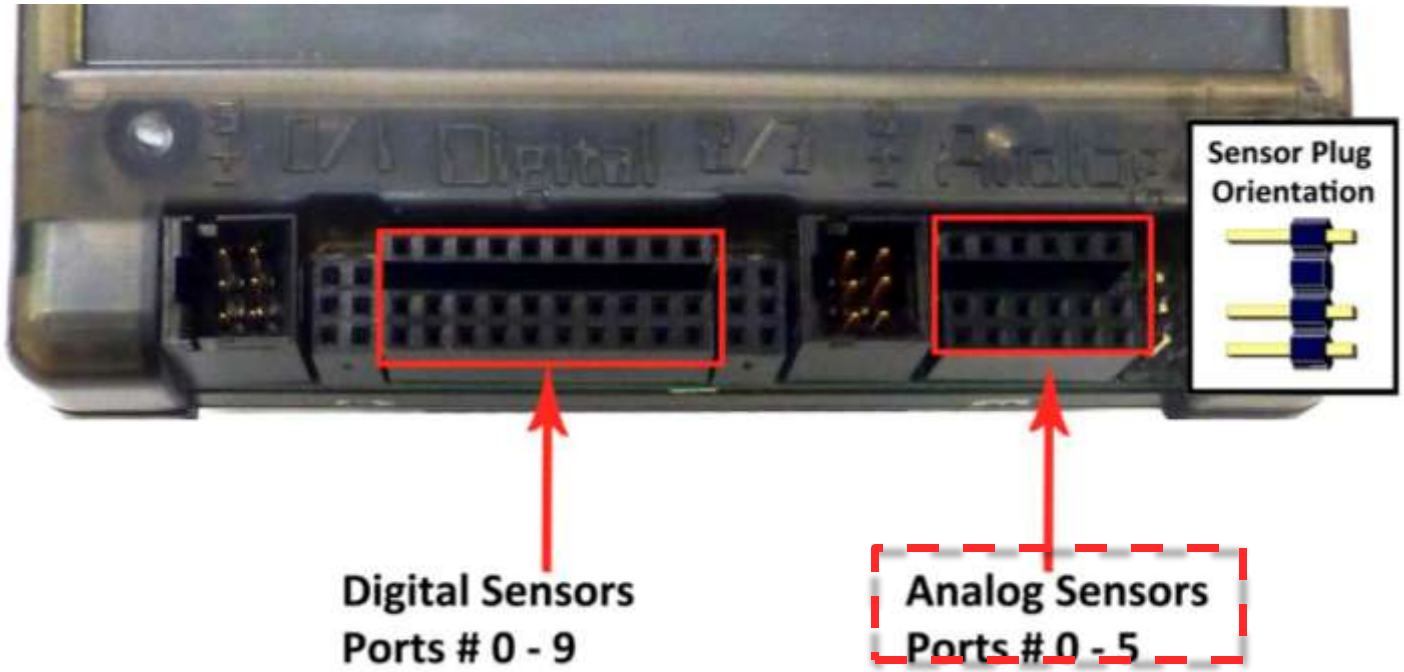
This sensor is excellent for line-following!

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

© 1993 – 2016 KIPR

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**.
  - Recall that analog sensor values range **from 0 to 1023**.



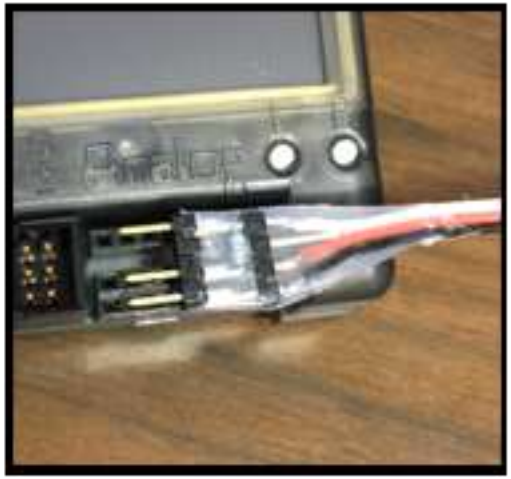


© 1993 – 2016 KIPR

Plug in your reflectance sensor

Activity



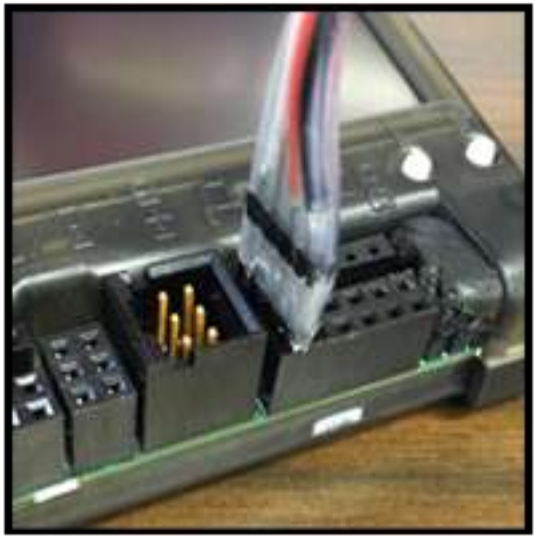


Digital Sensors
Ports # 0 - 9

Analog Sensors
Ports # 0 - 5

Plug your Reflectance Sensor into Analog Port #0.





© 1993 – 2016 KIPR

Botball®

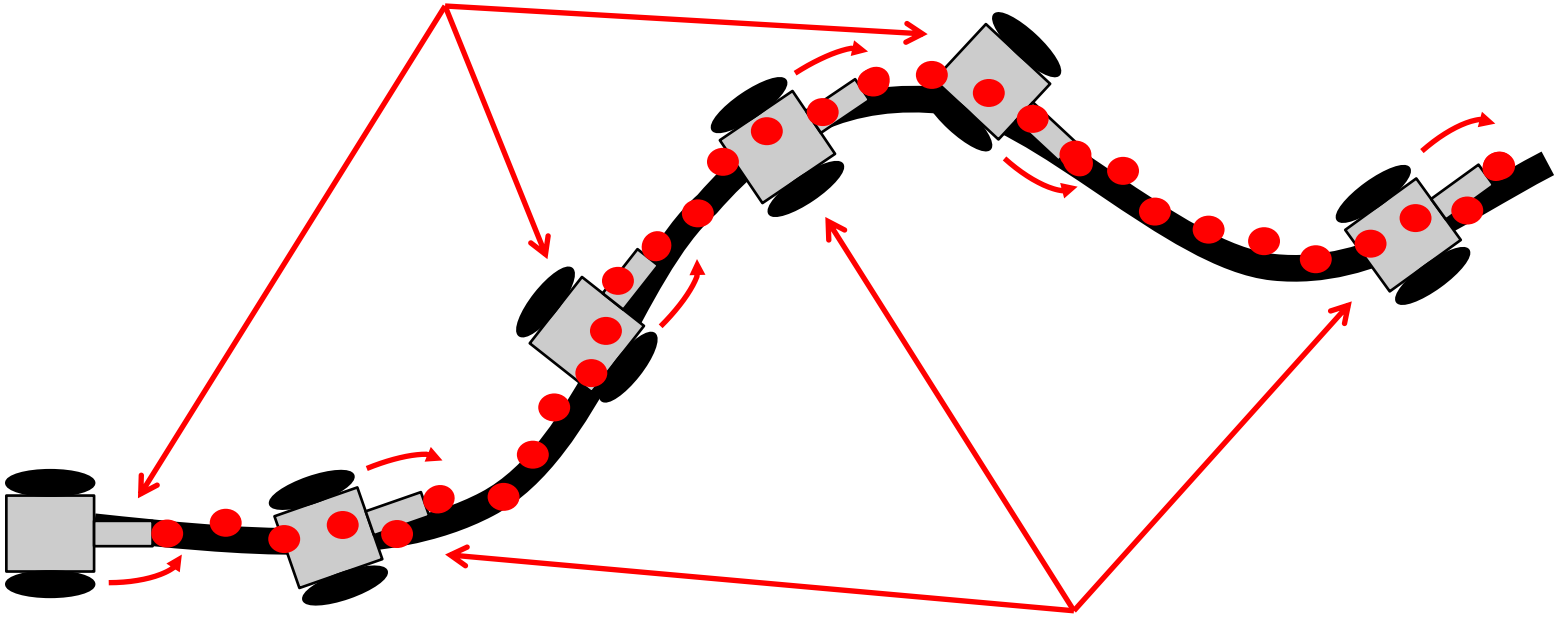
Line-following

Activity

Description: Write a program for the KIPR Link that makes the DemoBot follow a line, and stops when the touch sensor is touched.

- Follow the line's **left edge** by *alternating* the following 2 actions:

1. If detecting *dark*, then turn/arc *left*.



2. Else it is *not dark*, then turn/arc *right*.

© 1993 – 2016 KIPR



Line-following

Activity

Description: Write a program for the KIPR Robotics Controller that makes the DemoBot follow a line, 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 dark detected?
        1. Turn/arc left.
    2. Else:
        1. Turn/arc right.
2. Stop motors.
3. End the program.

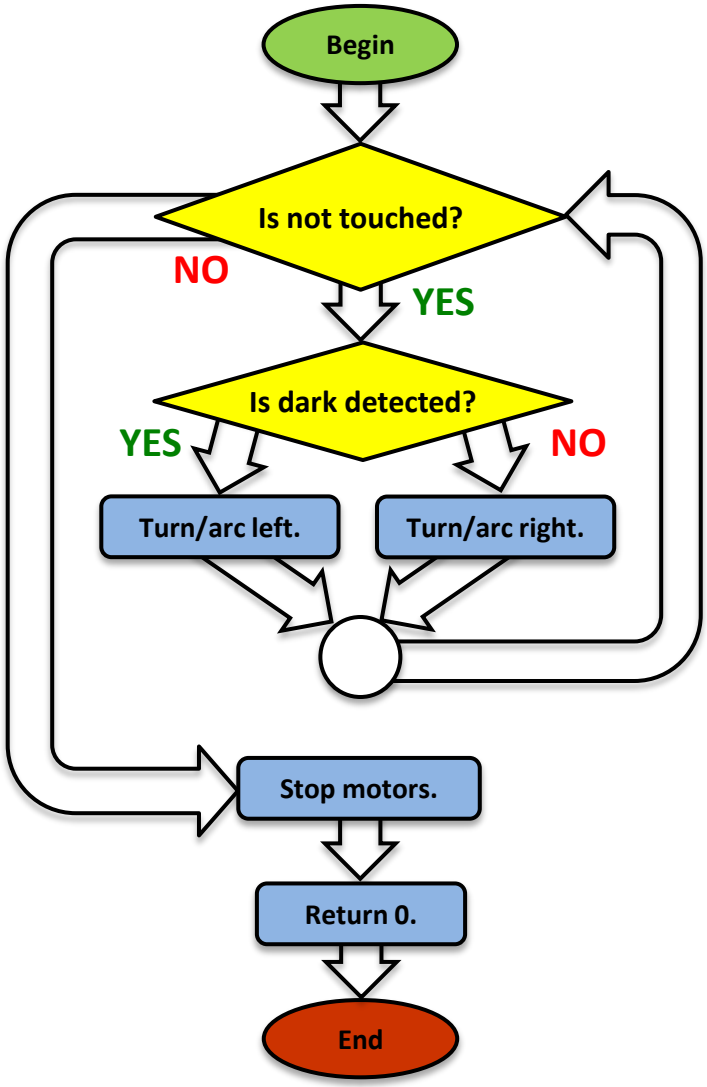
Comments

```
// 1. Loop: Is not touched?
//    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.
```


© 1993 – 2016 KIPR

Line-following

Analysis: Flowchart



Professional Development Workshop
© 1993 – 2016 KIPR



Line-following

Activity

Solution:

Pseudocode (Comments)

```
int main()
{
  // 1. Loop: Is not touched?
  //    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

```
int main()
{
  // 1. Loop: Is not touched?
  while (digital(8) != 1)
  {
    // 1.1. If: Is dark detected?
    if (analog(0) > 2048)
    {
      // 1.1.1. Turn/arc left.
      mav(0, 200);
      mav(2, 1000);
    } // end if
    // 1.2. Else:
    else
    {
      // 1.2.1. Turn/arc right.
      mav(0, 1000);
      mav(2, 200);
    } // end else
  } // end while

  // 2. Stop motors.
  ao();

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


© 1993 – 2016 KIPR



Line-following

Activity

Solution: (using two functions)

Pseudocode (Comments)

```
int main()
{
  // 1. Loop: Is not touched?
  //    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

```
// Function prototypes for turn_left and turn_right.
void turn_left();
void turn_right();

// Function definition for main.
int main()
{
  // 1. Loop: Is not touched?
  while (digital(8) != 1)
  {
    // 1.1. If: Is dark detected?
    if (analog(0) > 2048)
    {
      // 1.1.1. Turn/arc left.
      turn_left();
    } // end if
    // 1.2. Else:
    else
    {
      // 1.2.1. Turn/arc right.
      turn_right()
    } // end else
  } // end while

  // 2. Stop motors.
  ao();

  // 3. End the program.
  return 0;
} // end main

// Function definition for turn_left.
void turn_left()
{
  // Turn/arc left.
  mav(0, 200);
  mav(2, 1000);
} // end turn_left

// Function definition for turn_right.
void turn_right()
{
  // Turn/arc right.
  mav(0, 1000);
  mav(2, 200);
} // end turn_right
```



Homework

Game review

Game strategy

Workshop survey

© 1993 – 2016 KIPR

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.

© 1993 – 2016 KIPR



Homework for tonight: game strategy

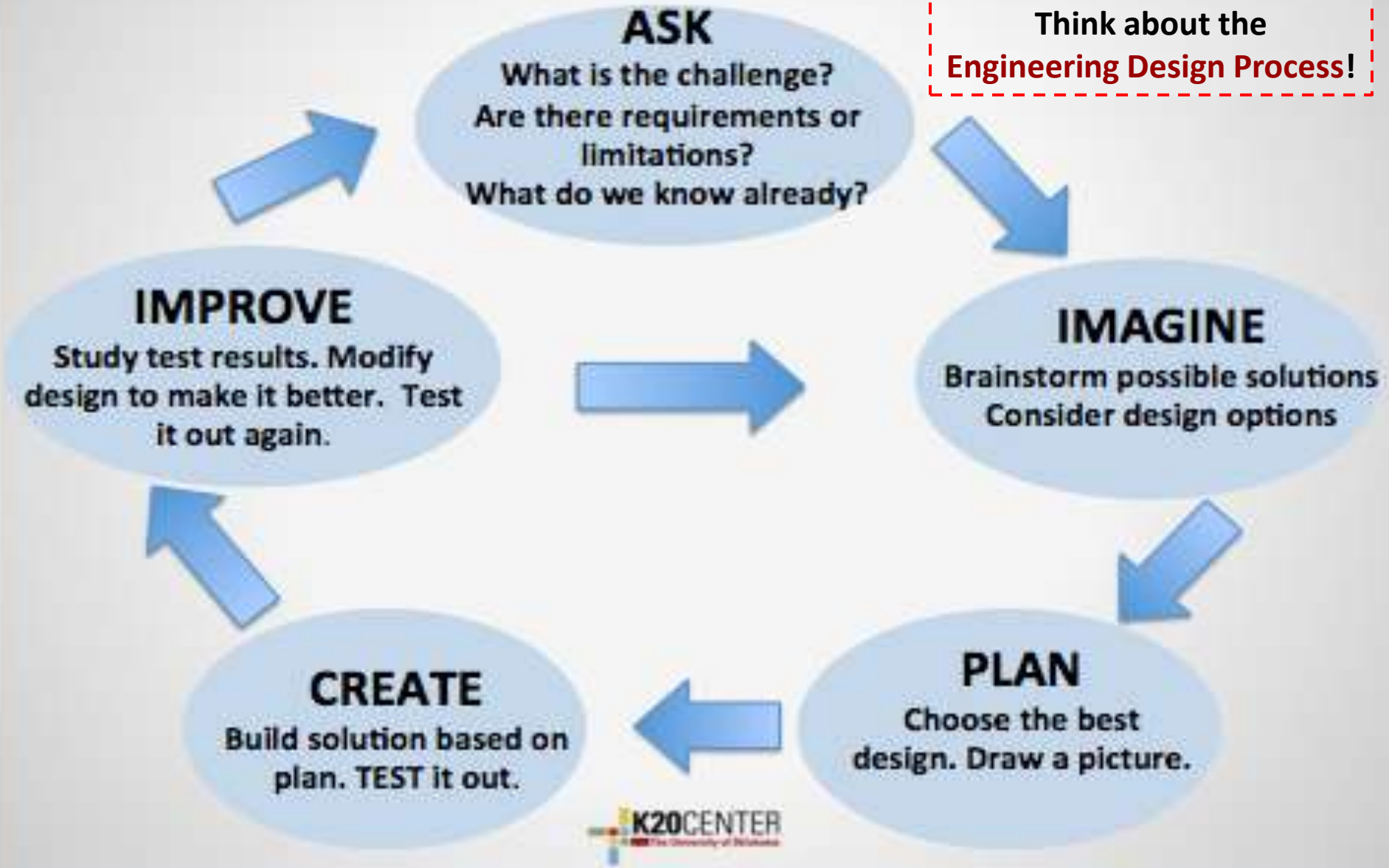
- Break down the game into subtasks!
- Write **pseudocode** and/or create **flowcharts**!
- Start with "**easy points**"—score early and score often!
- Keep it simple and make sure it works.
- Discuss your strategy with your instructor tomorrow.

© 1993 – 2016 KIPR

Homework for tonight:
game strategy

Resource

Think about the **Engineering Design Process**!





Homework for tonight:
workshop survey

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/s/Botball_workshop8RHYLG6

© 1993 – 2016 KIPR

Botball Team Home Base

Found at http://homebase.kipr.org



© 1993 – 2016 KIPR



Welcome back!

Please take our survey to give feedback about the workshop:
https://www.surveymonkey.com/s/Botball_workshop8RHYLG6

Botball 2016

Professional Development Workshop

Prepared by the KISS Institute for Practical Robotics (KIPR)

with significant contributions from KIPR staff

and the Botball Instructors Summit participants

v2016.01.30-0

© 1993 – 2016 KIPR

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
- Making Smarter Robots with Sensors
- Repetition, Repetition, Repetition
- 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**

© 1993 – 2016 KIPR

Botball Game Review

Game Q&A

Construction, documentation, and changes

Tournament template

`shut_down_in` function

© 1993 – 2016 KIPR

Botball Game Q&A starts…

NOW!

You have 30 minutes…

© 1993 – 2016 KIPR

Botball game board



© 1993 – 2016 KIPR

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!

© 1993 – 2016 KIPR

Documentation

What?

- **Botball Online Project Documentation (BOPD)**
- Rubrics and examples are on the **Team Home Base**

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).

© 1993 – 2016 KIPR

Changes this season

- New controller (Wallaby)
- Different Create module and brackets
- Omni wheels and platform
- Different camera
- New Servos (no stop), motors, and sensors

© 1993 – 2016 KIPR

Resources and Support

Team Home Base

Remind, YAC, Community, PYR, and social media

T-shirts and awards

What to do after the workshop

© 1993 – 2016 KIPR

Botball Team Home Base

Found at http://homebase.kipr.org

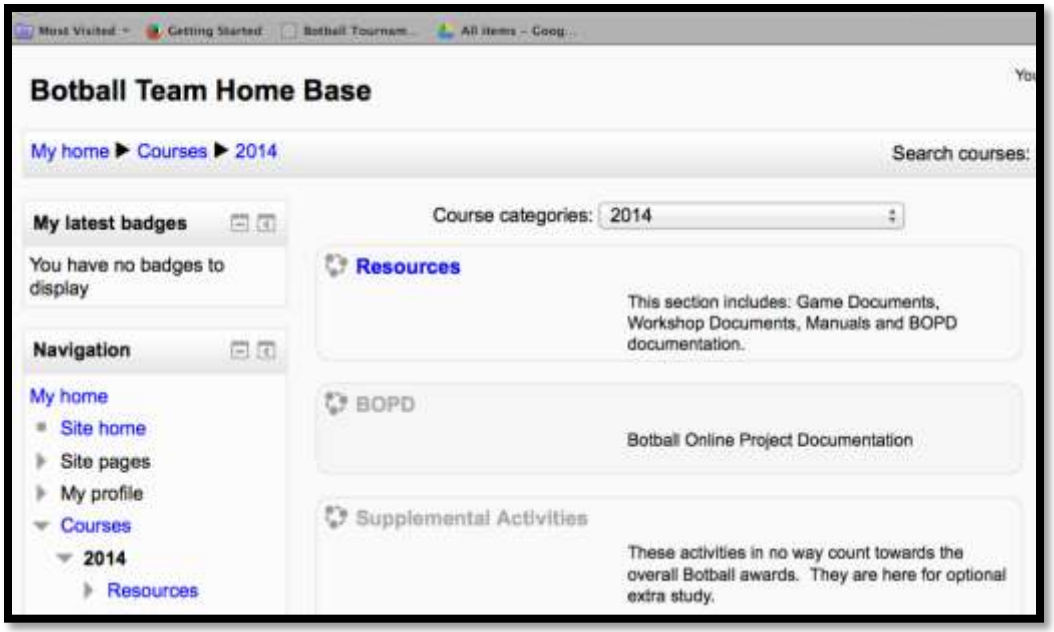


© 1993 – 2016 KIPR

Botball Team Home Base

KIPR Support

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



Forum and FAQ

- **Site:** 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 2016 Game Documents

© 1993 – 2016 KIPR

Botball Remind



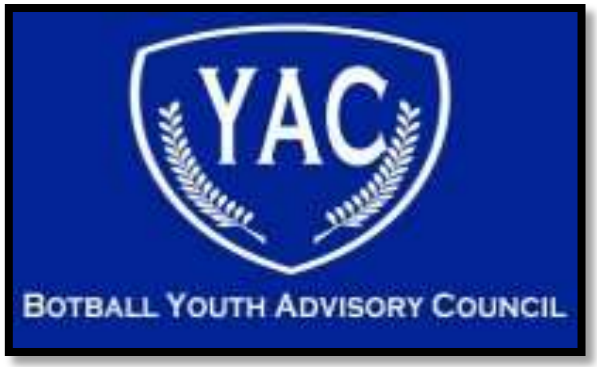
https://www.remind.com/join

Botball General: @bot16

Texas: @bbtx16

© 1993 – 2016 KIPR

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!

© 1993 – 2016 KIPR

Botball Community



http://community.botball.org

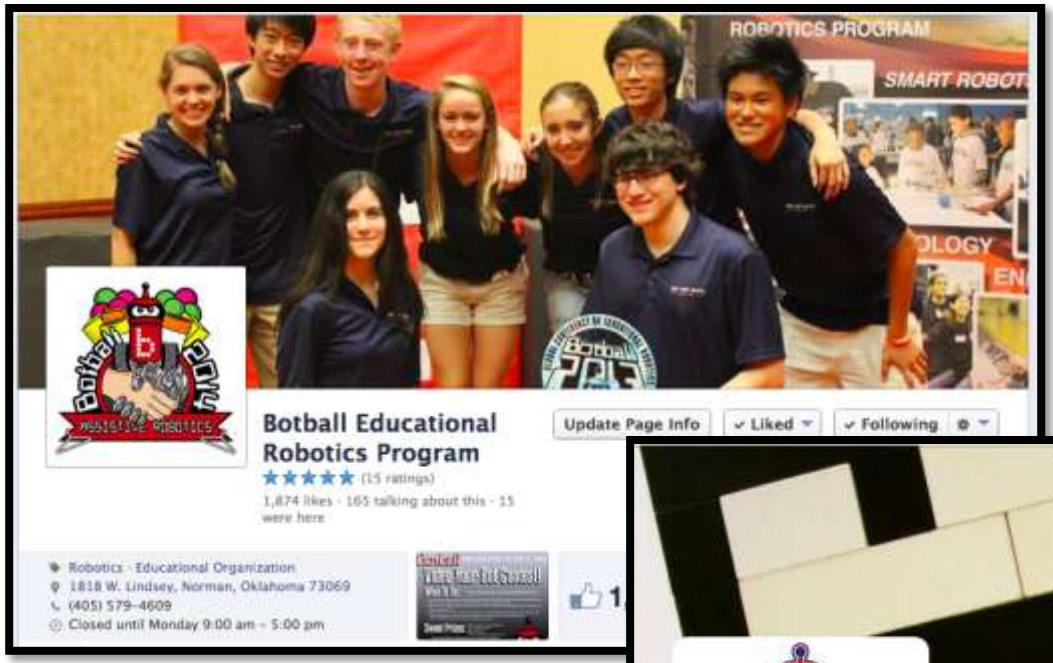
- The Community is a social network for current and former Botball participants to meet and hang out, discuss Botball and robotics technology in general, or just have a good time.

- Contains a safe and user-friendly chat-room, the Botballer's Chat, for getting immediate help to technical problems, or just hanging out with fellow Botballers around the world.

- Discuss Botball, technology, and everything else during and after the Botball season.

© 1993 – 2016 KIPR

Social media

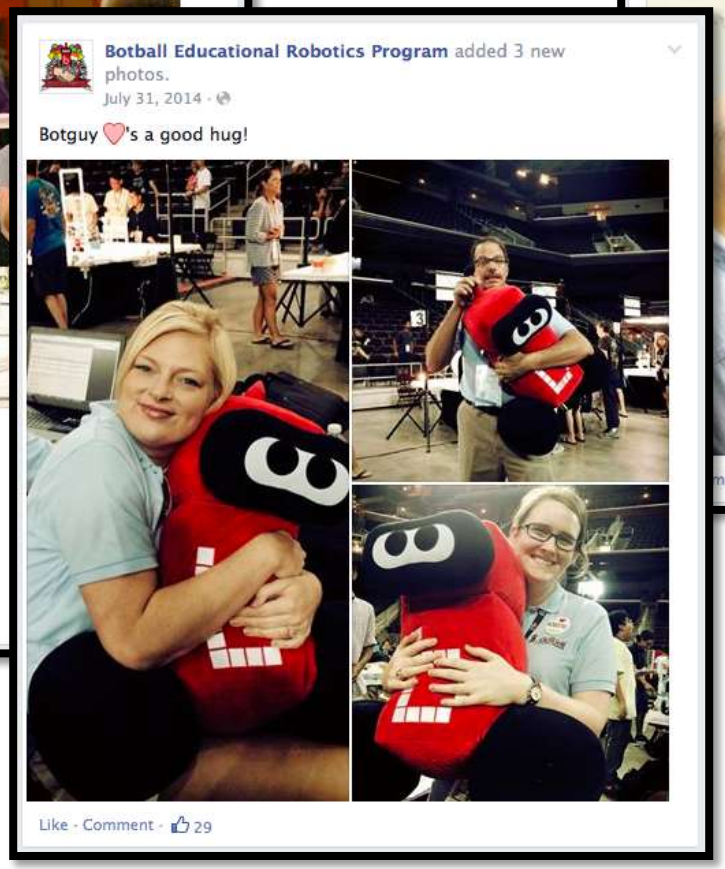




© 1993 – 2016 KIPR

Social media







© 1993 – 2016 KIPR

Botball T-shirts

Team pre-order: $?

—VS —

At tournament: $?++



FRONT



BACK

Notes:

- T-shirts are not provided.
- One pre-order per team.
- Pre-orders can be modified up to 1 week prior to tournament.

http://www.botball.org/shirts

© 1993 – 2016 KIPR

Tournament awards







© 1993 – 2016 KIPR

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

© 1993 – 2016 KIPR

What to do after the workshop

1. Recruit team members.

If you haven't already recruited team members you can use the game video 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.

© 1993 – 2016 KIPR

What to do after the workshop

3. Plan out the season.

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

© 1993 – 2016 KIPR

What to do after the workshop

4. Build the game board.

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

5. Organize your Botball kit.

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

6. Understand the game.

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

© 1993 – 2016 KIPR



Botball Tournament Functions

wait_for_light

shut_down_in

© 1993 – 2016 KIPR

Tournament Template

```
int main()
{
  // initialize servo, camera, etc here?.
  // if you are using a Create you should uncomment the next line
  // create_connect();

  wait_for_light(0); // change the port # to where your sensor is plugged in

  shut_down_in(119); // shut off the motors and stop the Create after 119 seconds
  // note that shut_down_in should immediately follow wait_for_light!!

  printf("My program should score more points than this template!\n");
  // replace printf above with your code

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


© 1993 – 2016 KIPR

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!**

© 1993 – 2016 KIPR

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 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.

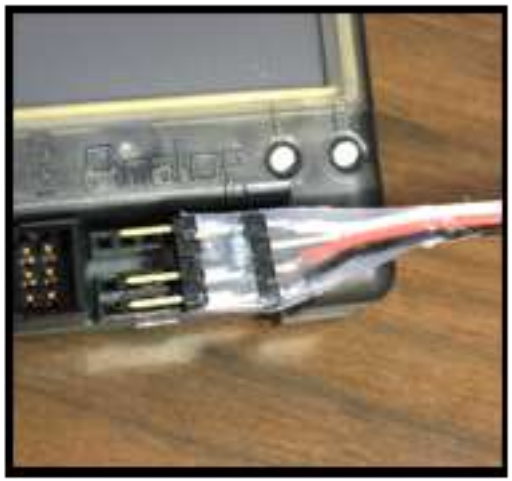
© 1993 – 2016 KIPR

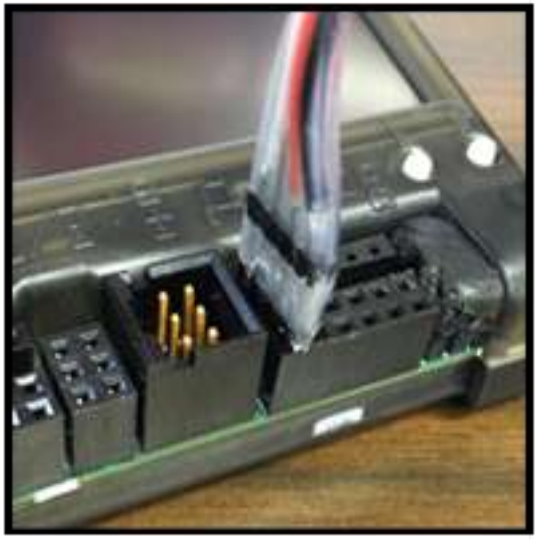
Plug in your light sensor

(and get your flashlight!)

Activity







Digital Sensors
Ports # 0 - 9

Analog Sensors
Ports # 0 - 5

Plug your Light Sensor into Analog Port #0.



© 1993 – 2016 KIPR

Botball®

Starting with a light

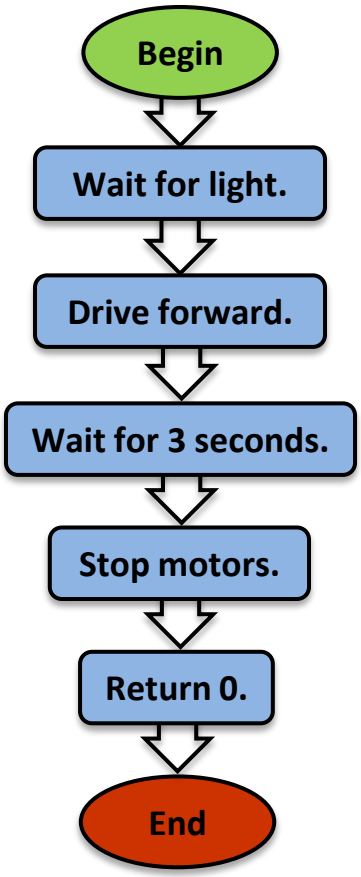
Activity

Description: Write a program for the KIPR Robotics Controller 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	Comments
1. Wait for light.	`// 1. Wait for light.`
2. Drive forward.	`// 2. Drive forward.`
3. Wait for 3 seconds.	`// 3. Wait for 3 seconds.`
4. Stop motors.	`// 4. Stop motors.`
5. End the program.	`// 5. End the program.`

Flowchart

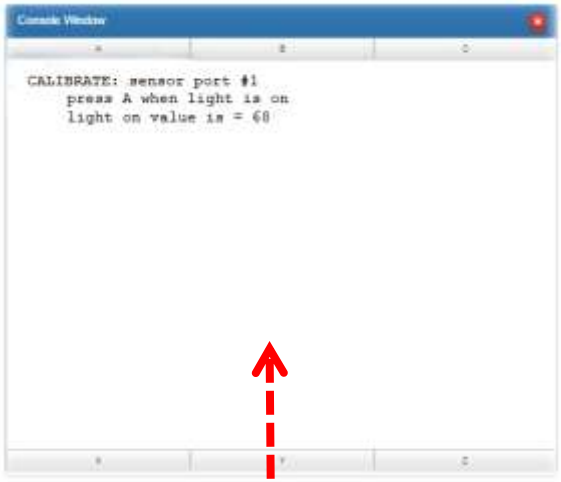


© 1993 – 2016 KIPR

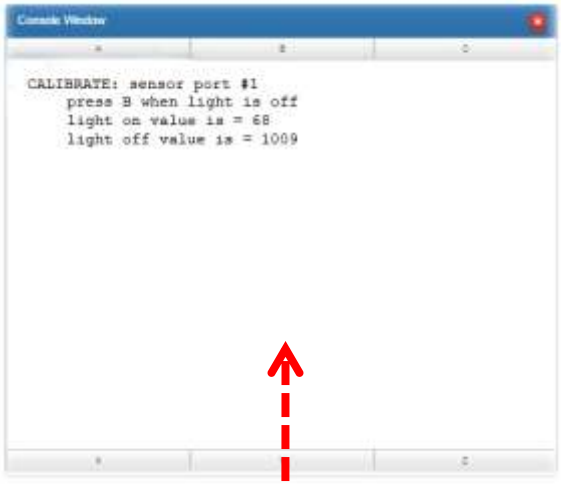
`wait_for_light` calibration routine

Activity

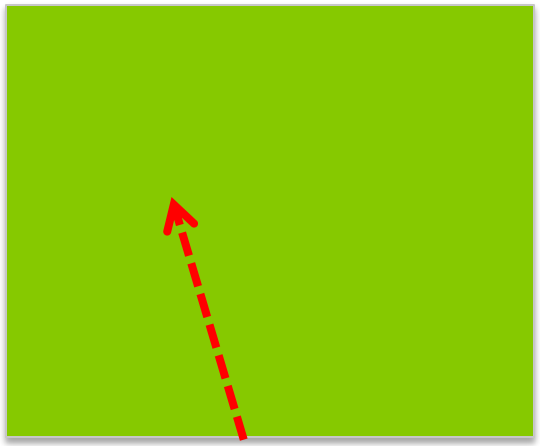
When you use the **`wait_for_light`** function in your program, the following calibration routine will run automatically.



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



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



You will get a "Good Calibration!" message and the screen will turn green 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.

© 1993 – 2016 KIPR



`shut_down_in` stopping the program

Activity

When you use the `shut_down_in(88.7)` function in your program, your program will exit 88.7 seconds later.

Note: For Botball,

`shut_down_in`

should come after

`wait_for_light` !!!

Why?

© 1993 – 2016 KIPR



Starting with a light

Activity

Solution:

Pseudocode (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()
{
  // 1. Wait for light.
  wait_for_light(0);

  // 2. Drive forward.
  mav(0, 1000);
  mav(2, 1000);

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

  // 4. Stop motors.
  ao();

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

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

© 1993 – 2016 KIPR



Running a Botball tournament program

Activity

Description: Write a program for the KIPR Link 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.

- **Hint:** use the **"Tournament" template**.

Analysis: What is the program supposed to do?

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

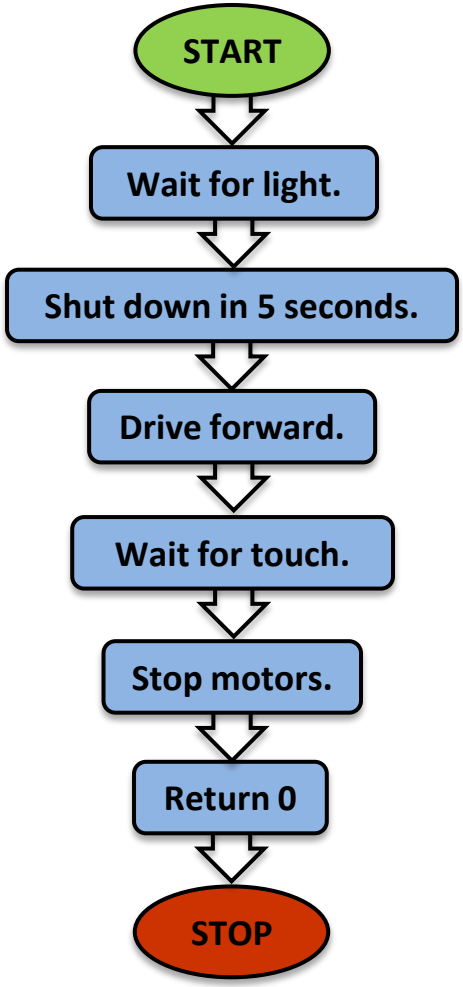
© 1993 – 2016 KIPR

Running a Botball tournament program

Activity

Analysis:

Flowchart



© 1993 – 2016 KIPR



Running a Botball tournament program

Activity

Solution:

Pseudocode (Comments)

```
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

```
int main()
{
  // 1. Wait for light.
  wait_for_light(0);

  // 2. Shut down in 5 seconds.
  shut_down_in(5);

  // 3. Drive forward.
  mav(0, 1000);
  mav(2, 1000);

  // 4. Wait for touch.
  wait_for_touch(8);

  // 5. Stop motors.
  ao();

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

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

© 1993 – 2016 KIPR

Botball®



Running a Botball tournament program

Activity

Reflection:

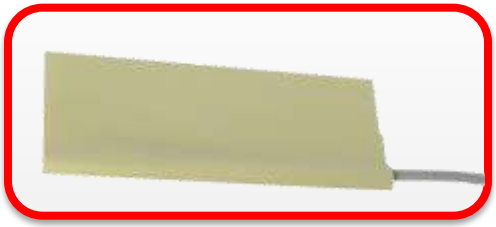
- What happens if the touch sensor is pressed in ***less than 5 seconds*** after starting the program?
- What happens if the touch sensor is **not** pressed in ***less than 5 seconds*** after starting the program?
- What is the best way to guarantee that your program will ***start with the light*** in 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!

© 1993 – 2016 KIPR

Shielding a light sensor

Activity



NO!!!

Paper is NOT adequate shielding.

Step #1



Step #2



Step #3



YES!!!

Slide a straw over the light sensor (leave a 1” gap in the front) and tape it in place covering the straw with electrical tape.

Professional Development Workshop
© 1993 – 2016 KIPR

Botball®

Motor Position Counter

Motor position counter functions

Ticks and revolutions

© 1993 – 2016 KIPR

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".
```

- The motor position is measured in "**ticks**".
- Botball motors have ***approximately*** **??? ticks per** ***revolution***.
- Experiment to find out the correct value!

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



© 1993 – 2016 KIPR



Using motor position counter functions

How long
will the motor rotate?

```
int main()
{
  int start_pos = get_motor_position_counter(2);
  msleep(100);
  while (get_motor_position_counter(2) < start_pos + 3000)
  {
    mav(2, 500);
    msleep(20);
  }
  ao();
  return 0;
}
```


© 1993 – 2016 KIPR



Drive a specific number of revolutions

Activity

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

Analysis: What is the program supposed to do? First find out how many ticks in a wheel revolution

Pseudocode	Comments
1. Get current position	`// 1. Get current position.`
2. Loop: Is counter < ???	`// 2. Loop: Is counter < start + 10*ticksPerRev?`
1. Drive forward.	`//  2.1. Drive forward.`
3. Stop motors.	`// 3. Stop motors.`
4. End the program.	`// 4. End the program.`

What is the value for ticks per revolution?

© 1993 – 2016 KIPR



Drive a specific number of revolutions

Activity

Solution:

Pseudocode (Comments)

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

Source Code

```
int main()
{
  // 1. Reset motor position counters.
  int start_pos = get_motor_position_counter(0);

  // 2. Loop: Is counter < ??
  // If ticsPerRev is 850 than ...
  if (get_motor_position_counter(0) < start_pos+8500)
  {
    // 2.1. Drive forward.
    mav(0, 1000);
    mav(2, 1000);
  }

  // 3. Stop motors.
  ao();

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



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 ticsPerRev** 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?

© 1993 – 2016 KIPR



Measuring Distance

Infrared “ET” distance sensor

`analog` function

© 1993 – 2016 KIPR

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.

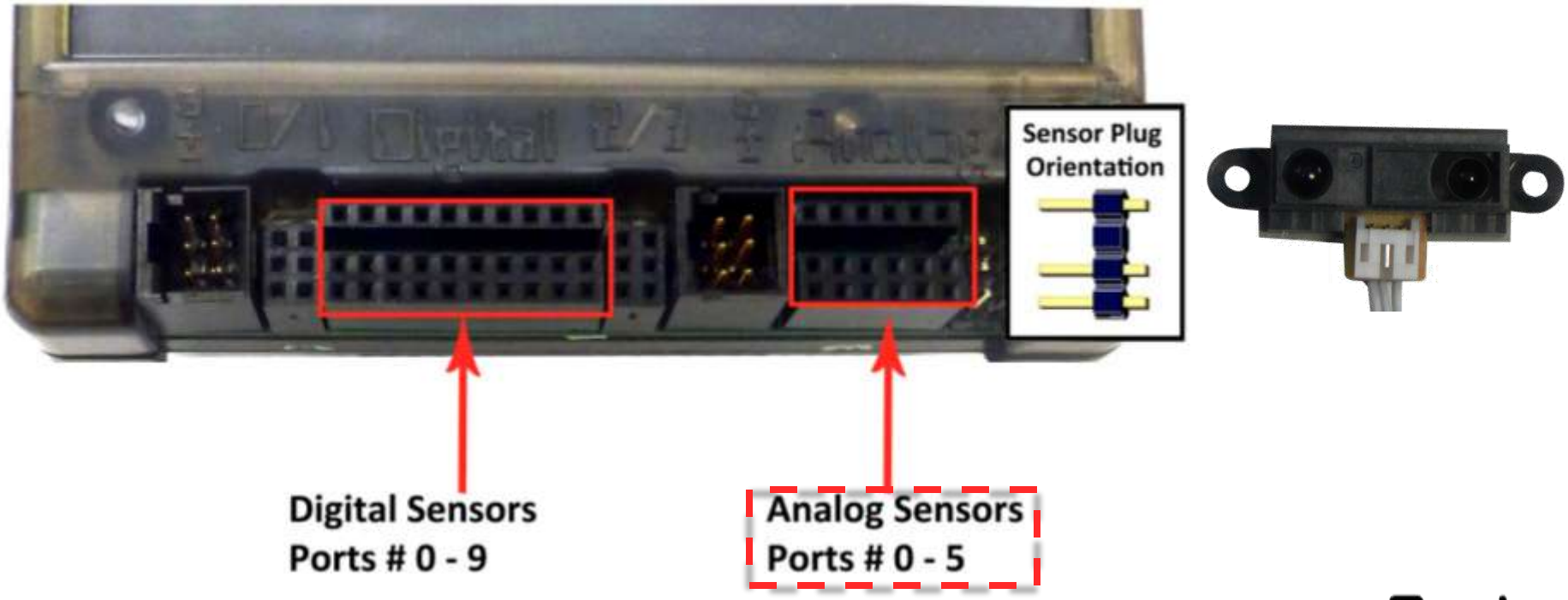
This sensor makes a great medium-range distance sensor.

- Values are ***reliable*** between 5cm and 80cm.
- 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.

© 1993 – 2016 KIPR

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**.
  - Recall that analog sensor values range **from 0 to 4095**.

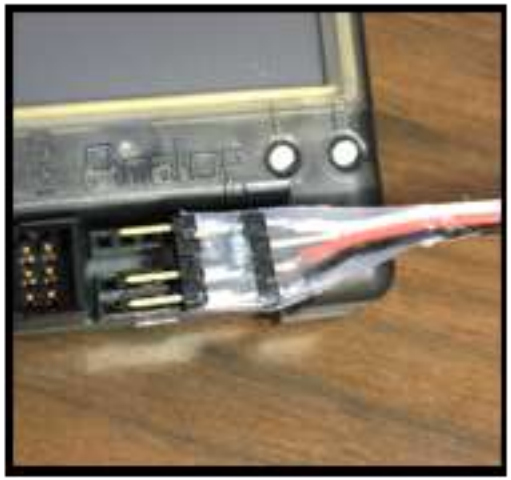


© 1993 – 2016 KIPR

Plug in your ET distance sensor

Activity



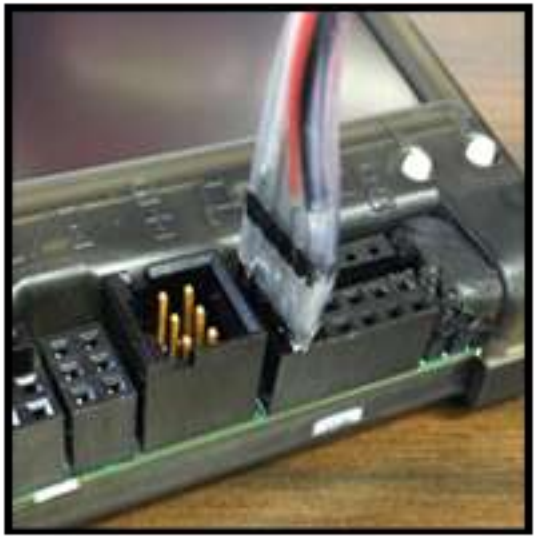


Digital Sensors
Ports # 0 - 9

Analog Sensors
Ports # 0 - 5

Plug your ET Distance Sensor into Analog Port #5.





© 1993 – 2016 KIPR

ET distance sensor function

To get sensor readings in a program,
use the **analog** function:

```
analog(3)
// Get an ET sensor reading from analog port #3.
```



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

© 1993 – 2016 KIPR

Using analog

What does this say?

```
int main()
{
  while (analog(3) < 2000)
  {
    mav(0, 1000);
    mav(2, 1000);
  }
  ao();
  return 0;
}
```

Notice: no semicolon!
(Why not?)

Remember

Lower values → *farther* distances.

Higher values → *closer* distances.

© 1993 – 2016 KIPR

Maintain distance

Activity

Description: Write a program for the KIPR Link 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.
```


© 1993 – 2016 KIPR



Maintain distance

Activity

Solution:

Pseudocode (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()
{
  // 1. Loop: Is not touched?
  while (digital(13) != 1)
  {
    // 1.1. Is distance too far?
    if (analog(7) < 2000)
    {
      // 1.1.1. Drive forward.
      mav(0, 300);
      mav(2, 300);
    } // end if
    // 1.2. Else.
    else
    {
      // 1.2.1. If: Is distance too close?
      if (analog(7) > 2000)
      {
        // 1.2.1.1. Drive reverse.
        mav(0, -300);
        mav(2, -300);
      } // end if
      // 1.2.2. Else.
      else
      {
        // 1.2.2.1. Stop motors.
        ao();
      } // end else
    } // end else
  } // end while

  // 2. Stop motors.
  ao();

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

Reflection on Distance Sensor

- Sensor noise is when the sensor values change even though what being sensed has not
- If there is a little noise in the distance reading the robot will jerk back and forth
- Roboticists often add a “dead band” to their programs to avoid the effects of sensor noise
  - The previous program only stops robot if distance reads exactly 2000
  - How could you change the program so the robot was stopped if the distance was anywhere from 1800-2200?

© 1993 – 2016 KIPR

Moving the iRobot *Create*: Part 1

Setting up the *Create*

The *Create* and the KIPR Robot Controller

Create functions

© 1993 – 2016 KIPR

Charging the *Create*

Activity

- 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 <u>NOT</u> leave the unit unattended** while charging.
  - Charge in a cool, open area away from flammable materials.

© 1993 – 2016 KIPR

Enabling the battery of the *Create*

Activity

- 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

© 1993 – 2016 KIPR

Uncovering and Charging the *Create*

Activity

- Remove the green protective tray from the top of the *Create.*
- Use only the *Create* charger provided with your kit.
- The charger plugs into the **Charging Socket** of the *Create*.



Remove this



Charging Socket **Serial Port**

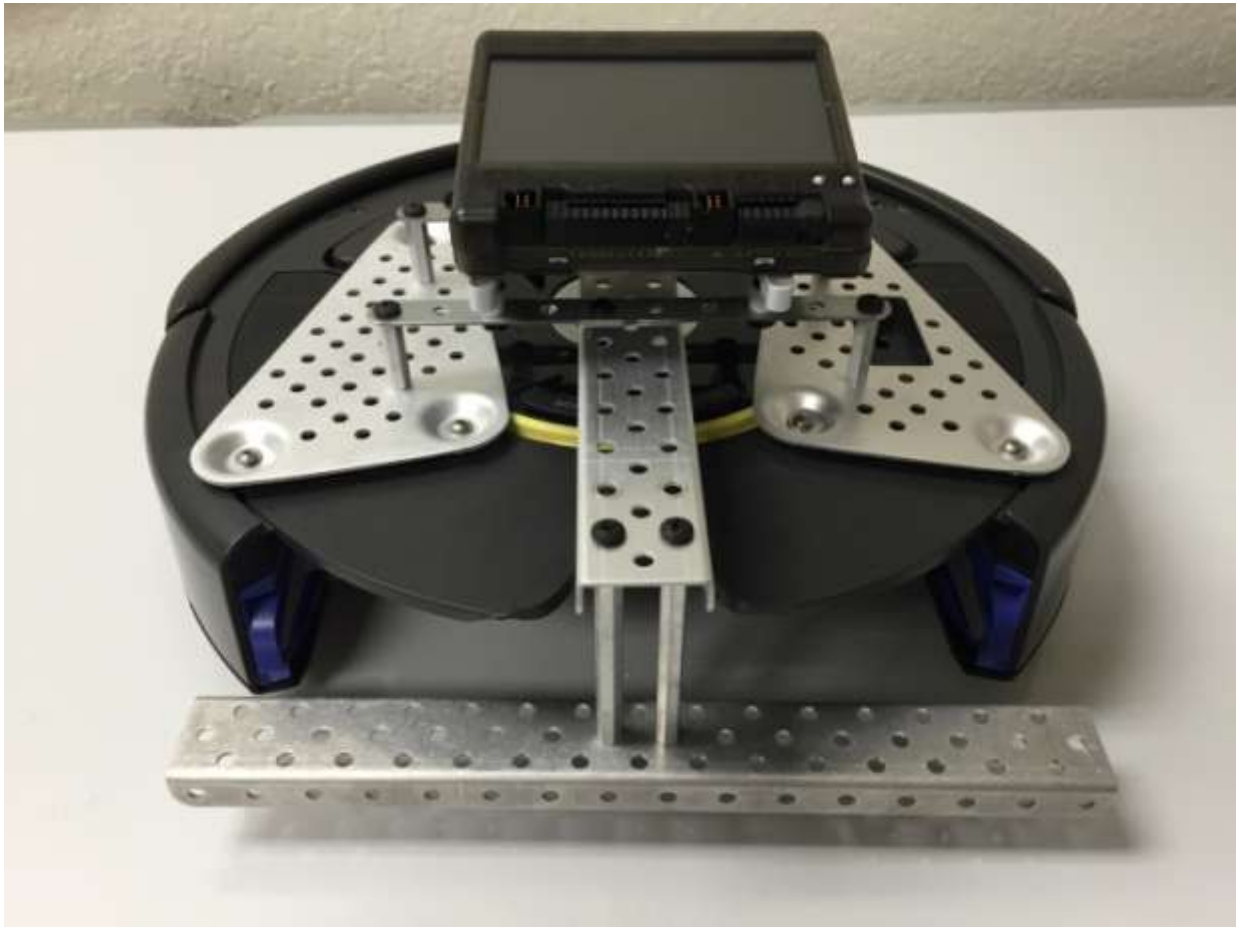
Professional Development Workshop
© 1993 – 2016 KIPR

Botball®

Mounting the Robotics Controller onto the *Create*

Activity

See the Demobot build which uses the Create



© 1993 – 2016 KIPR



Create connect/disconnect functions

- The KIPR Robotics Controller sends commands to the Create using the USB-Create cable.
- The *Create* must be **powered on** for this to work.

```
create_connect();
// Tells the KIPR Robotics Controller to connect to
// the serial connection input of the Create.

create_disconnect();
// Causes the Create to enter the mode where its power
// button, when pressed and held will turn it off
```

Once **`create_connect`** has been executed, the Create cannot be turned off until **`create_disconnect`** has been executed (or some other program is run that has **`create_connect`** followed subsequently by **`create_disconnect`**.

Unless put back on charge, failure to shut off the Create will cause a continuing battery drain which can ruin the battery.

© 1993 – 2016 KIPR

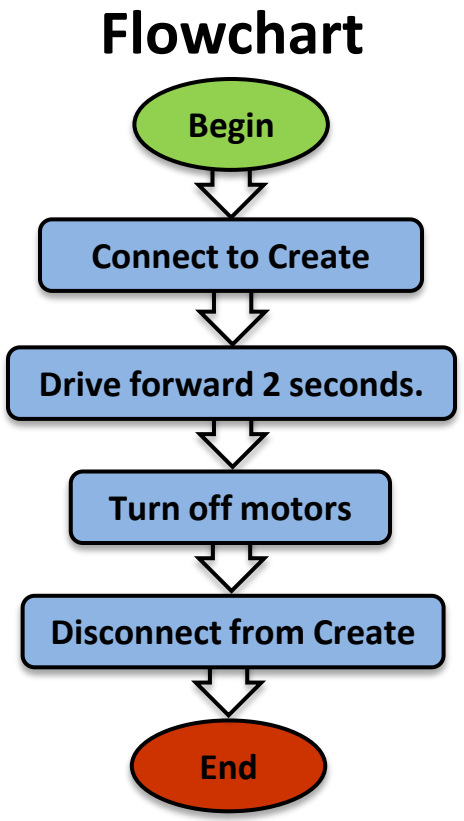
Create connect/disconnect functions

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

```
create_connect()
```

and *end* with

```
create_disconnect().
```



© 1993 – 2016 KIPR

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.

© 1993 – 2016 KIPR

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 th
Create drive?

© 1993 – 2016 KIPR



Moving the *Create*

Activity

Description: Write a program for the KIPR Link 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.
```

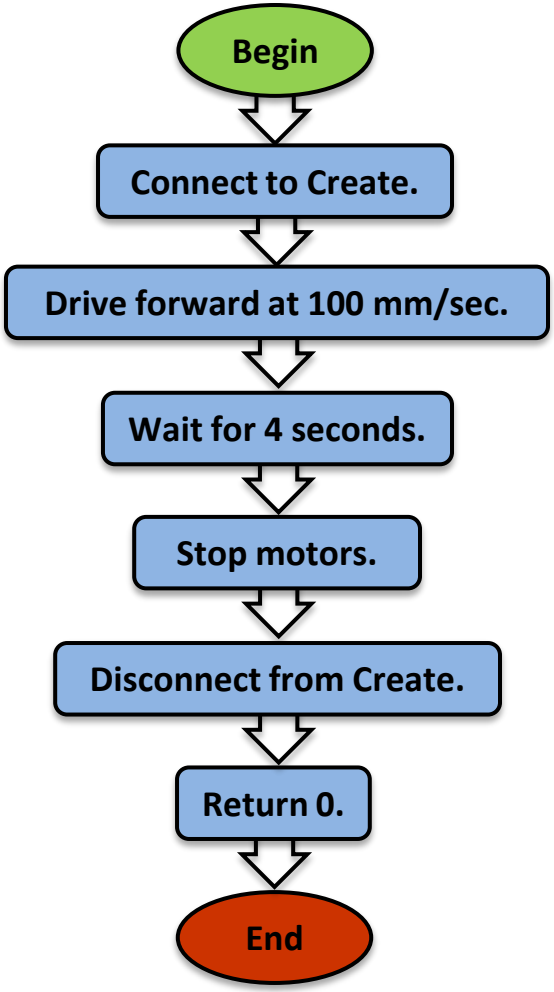

© 1993 – 2016 KIPR

Moving the *Create*

Activity

Analysis:

Flowchart





Moving the *Create*

Activity

Solution:

Psuedocode (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.
  // 6. End the program.
}
```

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();

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

Execution: Compile and run your program on the KIPR Robotics Controller.

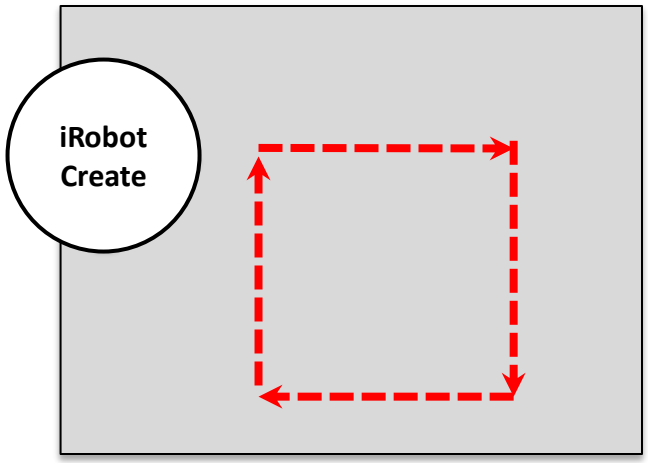
© 1993 – 2016 KIPR

Draw a square

Activity

Description: Write a program for the KIPR Robotics Controller that drives the *Create* along a path in the shape of a square.

- Start with having the robot make a 90° turn (both directions).
- Then add in forward movements to have the robot drive along a square path (remember the direction your robot is taking).
- Try drawing other regular polygons (pentagons, hexagons, octagons, etc.).
- Try writing your own **functions** and **functions with arguments**.



© 1993 – 2016 KIPR



Moving the iRobot *Create*: Part 2

Create distance and angle functions

© 1993 – 2016 KIPR

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.
```


© 1993 – 2016 KIPR

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);
    msleep(20);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```


© 1993 – 2016 KIPR

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);
    msleep(20);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```


© 1993 – 2016 KIPR

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);
    msleep(20);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```


© 1993 – 2016 KIPR

Using *Create* angle functions

```
int main()
{
  create_connect();
  set_create_total_angle(0);
  while (get_create_total_angle() > -90)
  {
    create_drive_direct(200, -200);
    msleep(20);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```

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

Notice: the signs changed!

© 1993 – 2016 KIPR



Drive a specific distance and angle

Activity

Description: Write a program for the KIPR Link that drives the *Create* forward for 1 meter, then turns right 90-degrees, and then stops.

Analysis: What is the program supposed to do?

Pseudocode

1. Connect to Create.
2. Reset *Create* distance.
3. Loop: Is distance < 1000?
    1. Drive forward.
4. Stop motors.
5. Reset *Create* total angle.
6. Loop: Is total angle > -90?
    1. Turn right.
7. Stop motors.
8. Disconnect from Create.
9. End the program.

Comments

```
// 1. Connect to Create.
// 2. Reset Create distance.
// 3. Loop: Is distance < 1000?
//    3.1. Drive forward.
// 4. Stop motors.
// 5. Reset Create total angle
// 6. Loop: Is total angle > -90?
//    6.1. Turn right.
// 7. Stop motors.
// 8. Disconnect from Create.
// 9. End the program.
```


© 1993 – 2016 KIPR



Drive a specific distance and angle

Activity

Solution:

Pseudocode (Comments)

```
int main()
{
   // 1. Connect to Create.
   // 2. Reset Create distance.
   // 3. Loop: Is distance < 1000?
   //    3.1. Drive forward.
   // 4. Stop motors.
   // 5. Reset Create total angle
   // 6. Loop: Is total angle > -90?
   //    6.1. Turn right.
   // 7. Stop motors.
   // 8. Disconnect from Create.
   // 9. End the program.
}
```

Source Code

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

  // 2. Reset Create distance.
  set_create_distance(0);

  // 3. Loop: Is distance < 1000?
  while (get_create_distance() < 1000)
  {
    //  3.1. Drive forward.
    create_drive_direct(200, 200);
    msleep(20); // slow the loop
  }

  // 4. Stop motors.
  create_stop();

  // 5. Reset Create total angle
  set_create_total_angle(0);

  // 6. Loop: Is total angle > -90?
  while (get_create_total_angle() > -90)
  {
    // 6.1. Turn right.
    create_drive_direct(200, -200);
    msleep(20); // slow the loop
  }

  // 7. Stop motors.
  create_stop();

  // 8. Disconnect from Create.
  create_disconnect();

  // 9. End the program.
  return 0;
} // end main
```

Drive a specific distance and angle

Solution: (using two functions)

Pseudocode (Comments)

```
int main()
{
  // 1. Connect to Create.
  // 2. Reset Create distance.
  // 3. Loop: Is distance < 1000?
  //      3.1. Drive forward.
  // 4. Stop motors.
  // 5. Reset Create total angle
  // 6. Loop: Is total angle > -90?
  //      6.1. Turn right.
  // 7. Stop motors.
  // 8. Disconnect from Create.
  // 9. End the program.
}
```

Source Code

Activity

```
// Function prototypes for
// create_drive_forward and create_turn_right.
void create_drive_forward();
void create_turn_right();

// Function definition for main.
int main()
{
  // 1. Connect to Create.
  create_connect();

  // 2-4. Function call for create_drive_forward.
  create_drive_forward();

  // 5-7. Function call for create_turn_right.
  create_turn_right();

  // 8. Disconnect from Create.
  create_disconnect();

  // 9. End the program.
  return 0;
} // end main

// Function definition for create_drive_forward.
void create_drive_forward()
{
  // Reset Create distance.
  set_create_distance(0);

  // Loop: Is distance < 1000?
  while (get_create_distance() < 1000)
  {
    create_drive_direct(200, 200); //  Drive forward.
    msleep(20); // slow the loop
  }

  // Stop motors.
  create_stop();
} // end create_drive_forward

// Function definition for create_turn_right.
void createe_turn_right()
{
  // Reset Create total angle
  set_create_total_angle(0);

  // Loop: Is total angle > -90?
  while (get_create_total_angle() > -90)
  {
    create_drive_direct(200, -200); // Turn right
    msleep(20); // slow the loop.
  }

  // Stop motors.
  create_stop();
} // end create_turn_right
```

Drive a specific distance and angle

Activity

Solution: (using functions w/ arguments)

Pseudocode (Comments)

```
int main()
{
  // 1. Connect to Create.
  // 2. Reset Create distance.
  // 3. Loop: Is distance < 1000?
  //      3.1. Drive forward.
  // 4. Stop motors.
  // 5. Reset Create total angle
  // 6. Loop: Is total angle > -90?
  //      6.1. Turn right.
  // 7. Stop motors.
  // 8. Disconnect from Create.
  // 9. End the program.
}
```

Source Code

```
// Function prototypes for
// create_drive_forward and create_turn_right.
void create_drive_forward(int distance_mm);
void create_turn_right(int angle_deg);

// Function definition for main.
int main()
{
  // 1. Connect to Create.
  create_connect();

  // 2-4. Function call for create_drive_forward.
  create_drive_forward(1000);

  // 5-7. Function call for create_turn_right.
  create_turn_right(-90);

  // 8. Disconnect from Create.
  create_disconnect();

  // 9. End the program.
  return 0;
} // end main

// Function definition for create_drive_forward.
void create_drive_forward(int distance_mm)
{
  // Reset Create distance.
  set_create_distance(0);

  // Loop: Is distance < distance_mm?
  while (get_create_distance() < distance_mm)
  {
    //  Drive forward.
    create_drive_direct(200, 200);
  }

  // Stop motors.
  create_stop();
} // end create_drive_forward

// Function definition for create_turn_right.
void createe_turn_right(int angle_deg)
{
  // Reset Create total angle
  set_create_total_angle(0);

  // Loop: Is total angle > angle_deg?
  while (get_create_total_angle() > angle_deg)
  {
    // Turn right.
    create_drive_direct(200, -200);
  }

  // Stop motors.
  create_stop();
} // end create_turn_right
```



Drive a specific distance and angle

Activity

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

- Did the robot go exactly 1 meter?
- Did the robot turn exactly 90-degrees to the right?
- How could you modify your program to drive to the left? Or in reverse?

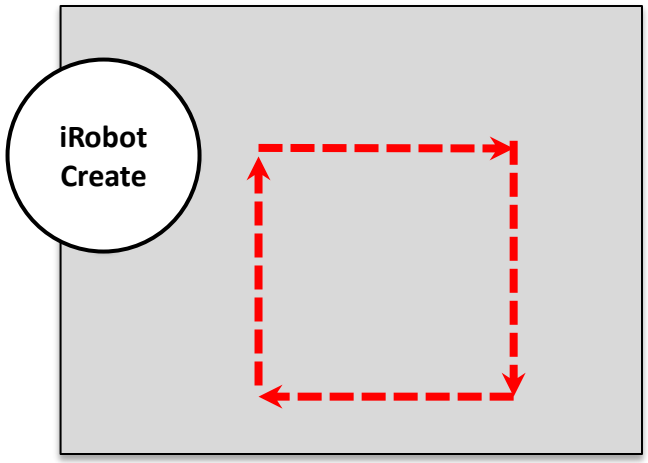
© 1993 – 2016 KIPR

Draw a square using distance/angle

Activity

Description: Write a program for the KIPR Robotics Controller that drives the *Create* along a path in the shape of a square *using distance/angle functions*.

- **Hint:** Repeat the steps in the previous activity four times.
- Try using a `while` loop to repeat the steps in the previous activity four times.
- Try writing your own **functions** to make the program easier to read and modify.
- Try writing your own **functions with arguments** to make the *Create* drive any distance and angle you want it to. **This would be very useful!**



© 1993 – 2016 KIPR

iRobot *Create* Sensors

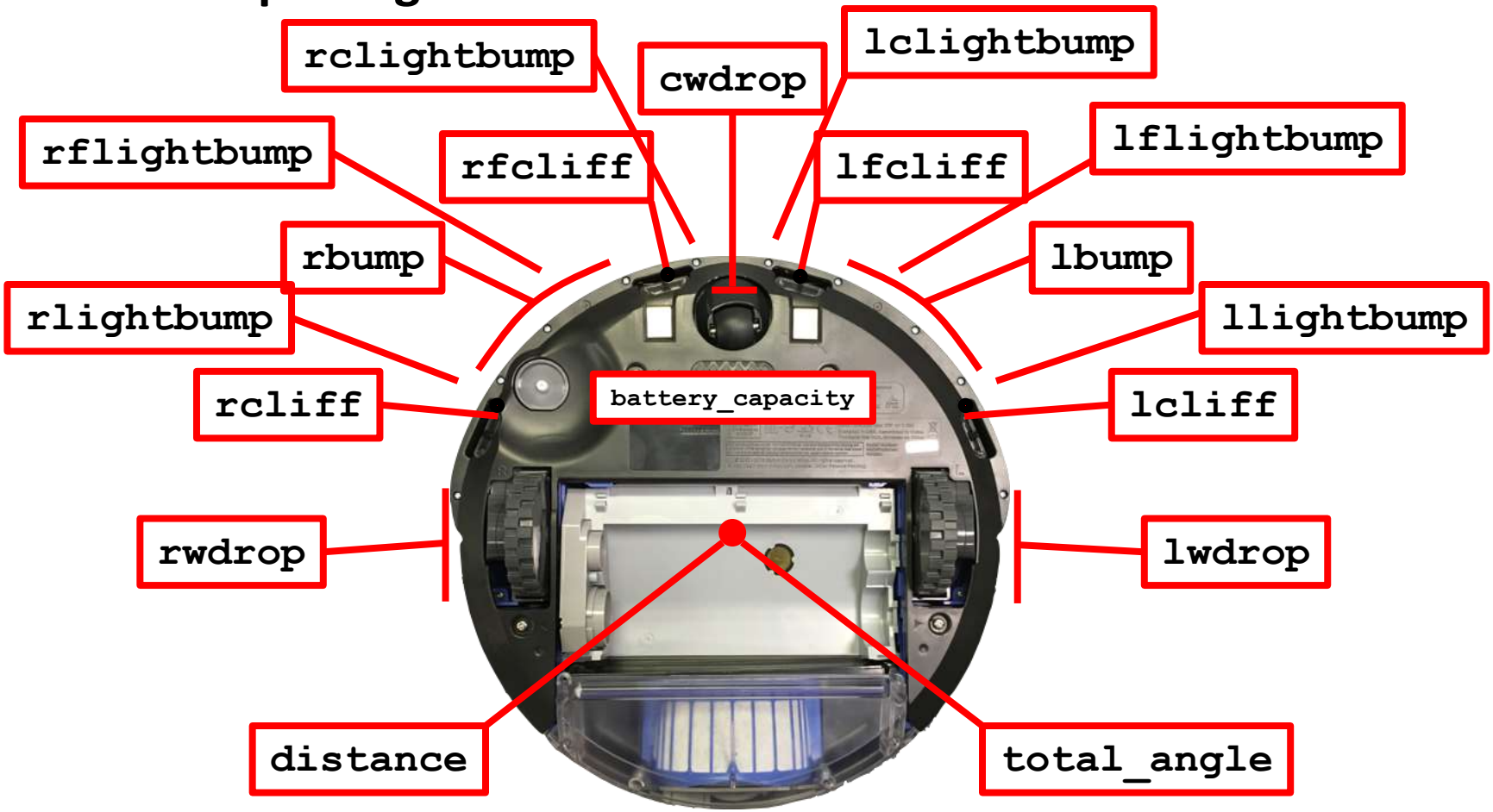
Create sensor functions

Logical operators

© 1993 – 2016 KIPR

Create sensor functions

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



© 1993 – 2016 KIPR



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).
```


© 1993 – 2016 KIPR

Using *Create* sensor functions

What does this say?

```
int main()
{
  create_connect();
  while (get_create_rbump() != 1)
  {
    create_drive_direct(100, 100);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```


© 1993 – 2016 KIPR



Drive until bumped

Activity

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

Analysis: What is the program supposed to do?

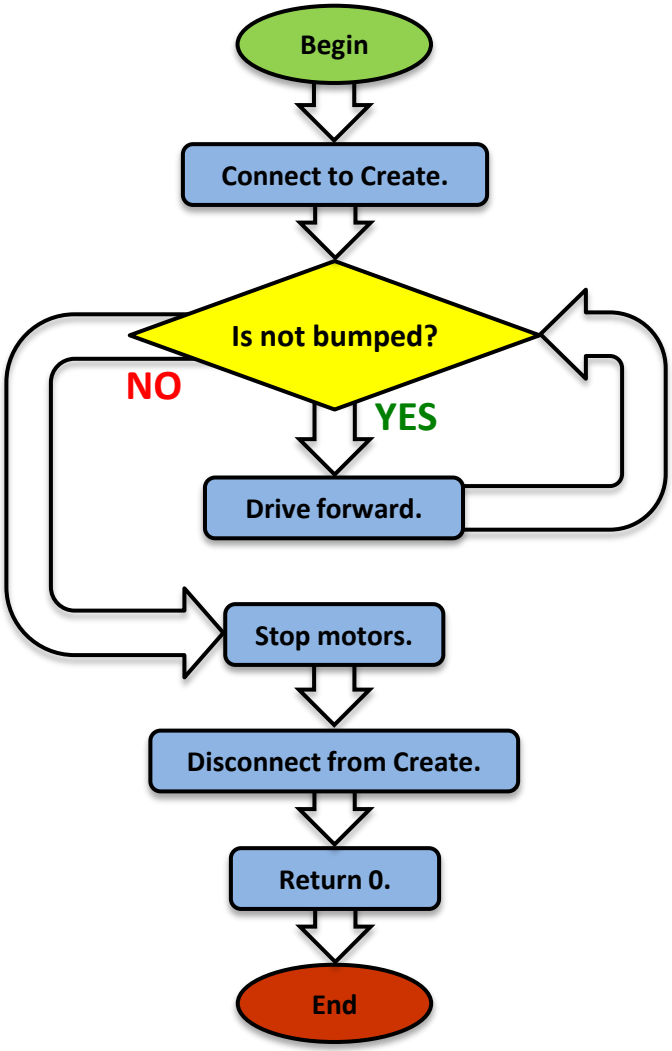
Pseudocode	Comments
1. Connect to Create.	`// 1. Connect to Create.`
2. Loop: Is not bumped?	`// 2. Loop: Is not bumped?`
1. Drive forward.	`//  2.1. Drive forward.`
3. Stop motors.	`// 3. Stop motors.`
4. Disconnect from Create.	`// 4. Disconnect from Create.`
5. End the program.	`// 5. End the program.`

© 1993 – 2016 KIPR

Drive until bumped

Activity

Analysis: Flowchart



© 1993 – 2016 KIPR



Drive until bumped

Activity

Solution:

Pseudocode (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() != 1)
  {
    // 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
```


© 1993 – 2016 KIPR



Drive until bumped

Activity

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

- What happens if the robot goes too fast?
- How might you use the other *Create* sensor functions?
- Could you use `get_create_lbump` instead of `get_create_rbump`?
- How can you check to see if *both* the left bumper *and* the right bumper are not pressed? To do this, we use **Logical operators**…

© 1993 – 2016 KIPR

LUNCH

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/s/Botball_workshop8RHYLG6

© 1993 – 2016 KIPR



Logical Operators

Multiple **Boolean tests**

`while`, `if`, and Logical operators

© 1993 – 2016 KIPR



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**.

© 1993 – 2016 KIPR



Logical operators cheat sheet

Resource

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.

© 1993 – 2016 KIPR



while, if, and Logical operators

```
// Repeat while both the Create left bumper AND Create right bumpers sensors
// are not equal to 1 (i.e., not pressed).
while ((get_create_lbump() != 1) && (get_create_rbump() != 1))
{
  // Code to execute ...
}

// Repeat while BOTH the touch sensors are not equal to 1 (i.e., not pressed).
while ((digital(8) != 1) && (digital(9) != 1))
{
  // Code to repeat ...
}

// Execute if the touch sensor on port #6 is equal to 1 (i.e., pressed)
// OR the touch sensor on port #7 is not equal to 0 (i.e., pressed).
if ((digital(6) == 1) || (digital(7) != 0))
{
  // Code to execute ...
}

// Execute if the analog sensor on port #3 reads less than 512
// OR the touch sensor is equal to 1 (i.e., pressed).
if ((analog(3) < 512) || (digital(6) == 1))
{
  // Code to repeat ...
}
```


© 1993 – 2016 KIPR

Using Logical operators

What does this say?



```
int main()
{
  create_connect();
  while ((get_create_rbump() != 1) && (get_create_rbump() != 1))
  {
    create_drive_direct(100, 100);
  }
  create_stop();
  create_disconnect();
  return 0;
}
```


© 1993 – 2016 KIPR



Drive for distance or until bumped

Activity

Description: Write a program for the KIPR Link 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() != 1`
- How do we check for that *both* are **true**?
  Answer: `((get_create_distance()) < 1000) && (get_create_rbump() != 1))`

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

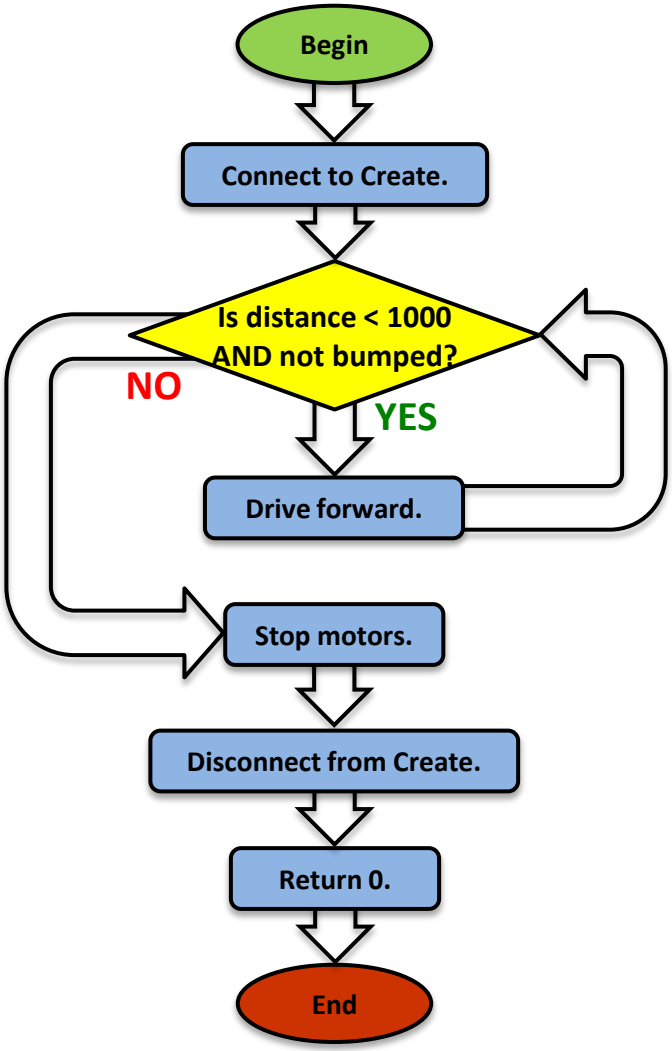
```
// 1. Connect to Create.
// 2. Loop: Is distance < 1000 AND not bumped?
//    2.1. Drive forward.
// 3. Stop motors.
// 4. Disconnect from Create.
// 5. End the program.
```


© 1993 – 2016 KIPR

Drive for distance or until bumped

Activity

Analysis: Flowchart



© 1993 – 2016 KIPR

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() != 1)
  {
    // 2.1. Drive forward.
    create_drive_direct(200, 200);
    msleep(20); // slow the loop
  } // end while

  // 3. Stop motors.
  create_stop();

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

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


© 1993 – 2016 KIPR



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_rbump() != 1) && (get_create_lbump() != 1))
```


© 1993 – 2016 KIPR



Color Camera

Using the color camera

Setting the color tracking channels

About color tracking

Camera functions

© 1993 – 2016 KIPR

Color camera

Activity

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

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



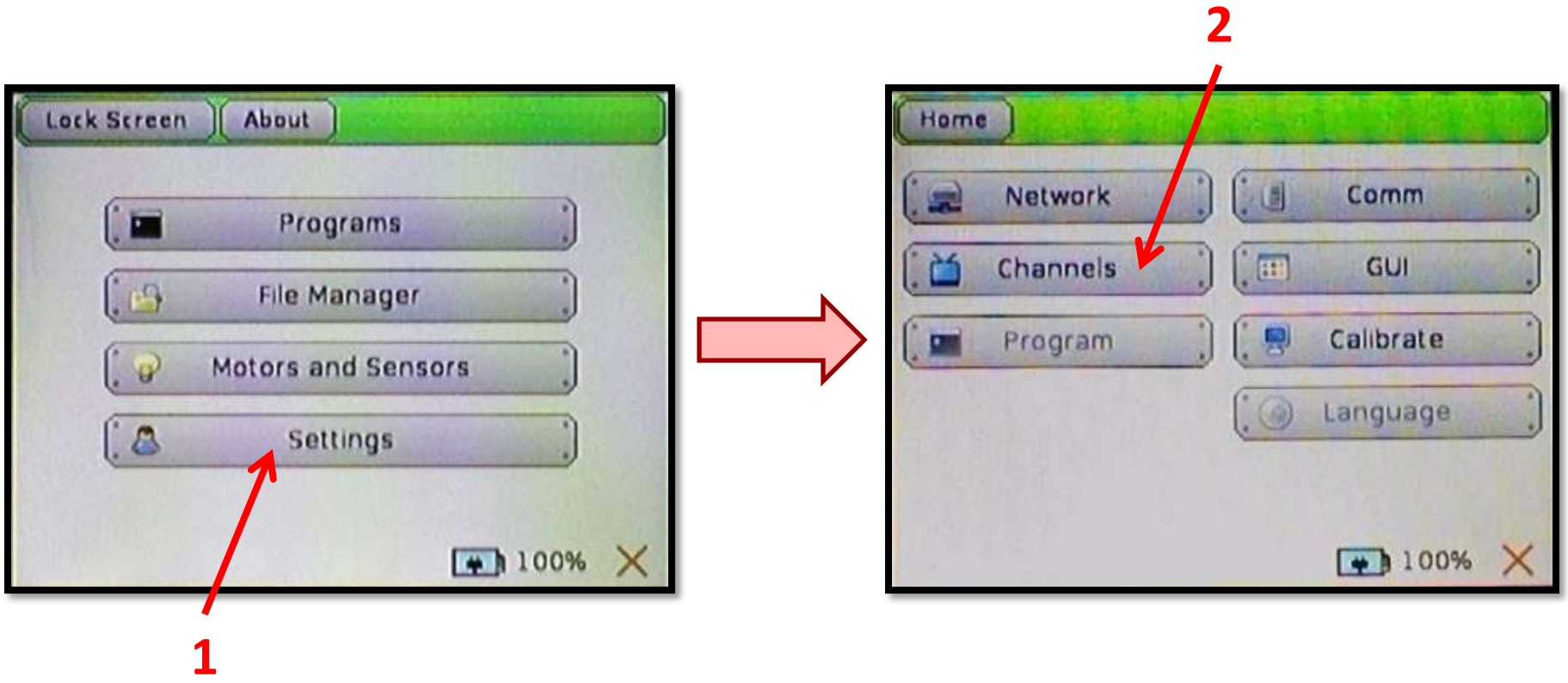


© 1993 – 2016 KIPR

Setting the color tracking channels: TBD

Activity

1. Select **“Settings”**.
2. Select **“Channels”**.

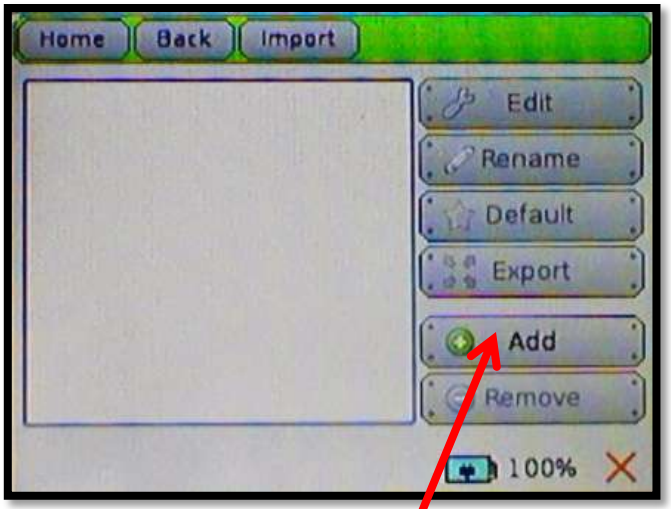


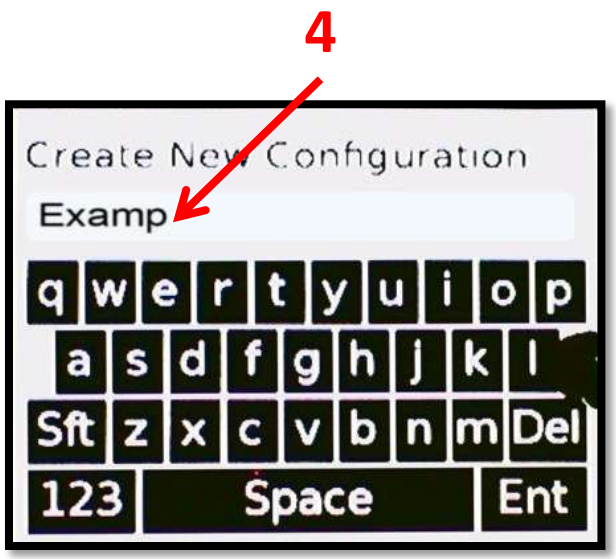
© 1993 – 2016 KIPR

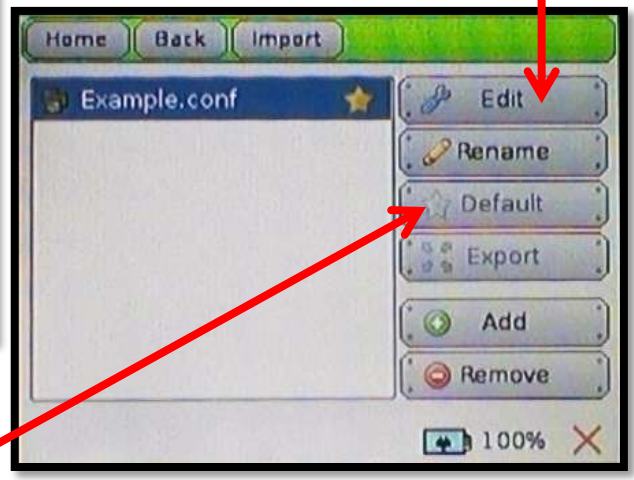
Setting the color tracking channels:TBD

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**.







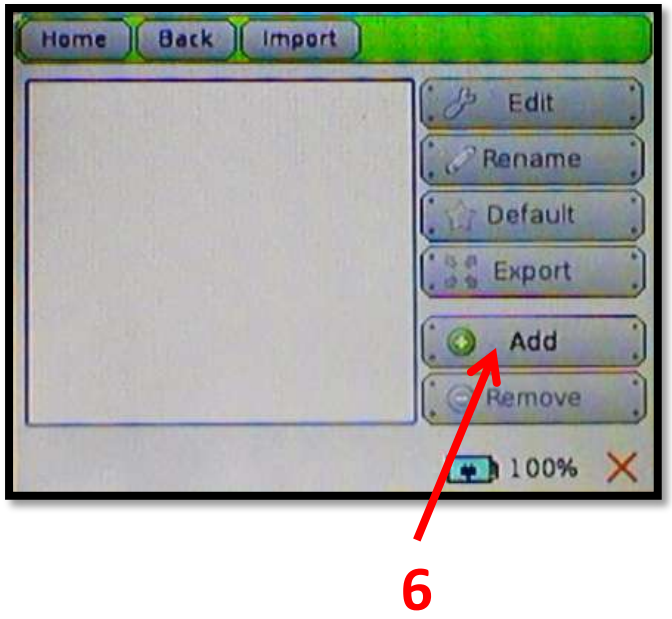
Note: if there is more than one configuration, select one, and press the "Default" button to make it be the one in use!

© 1993 – 2016 KIPR

Setting the color tracking channels:TBD

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**.







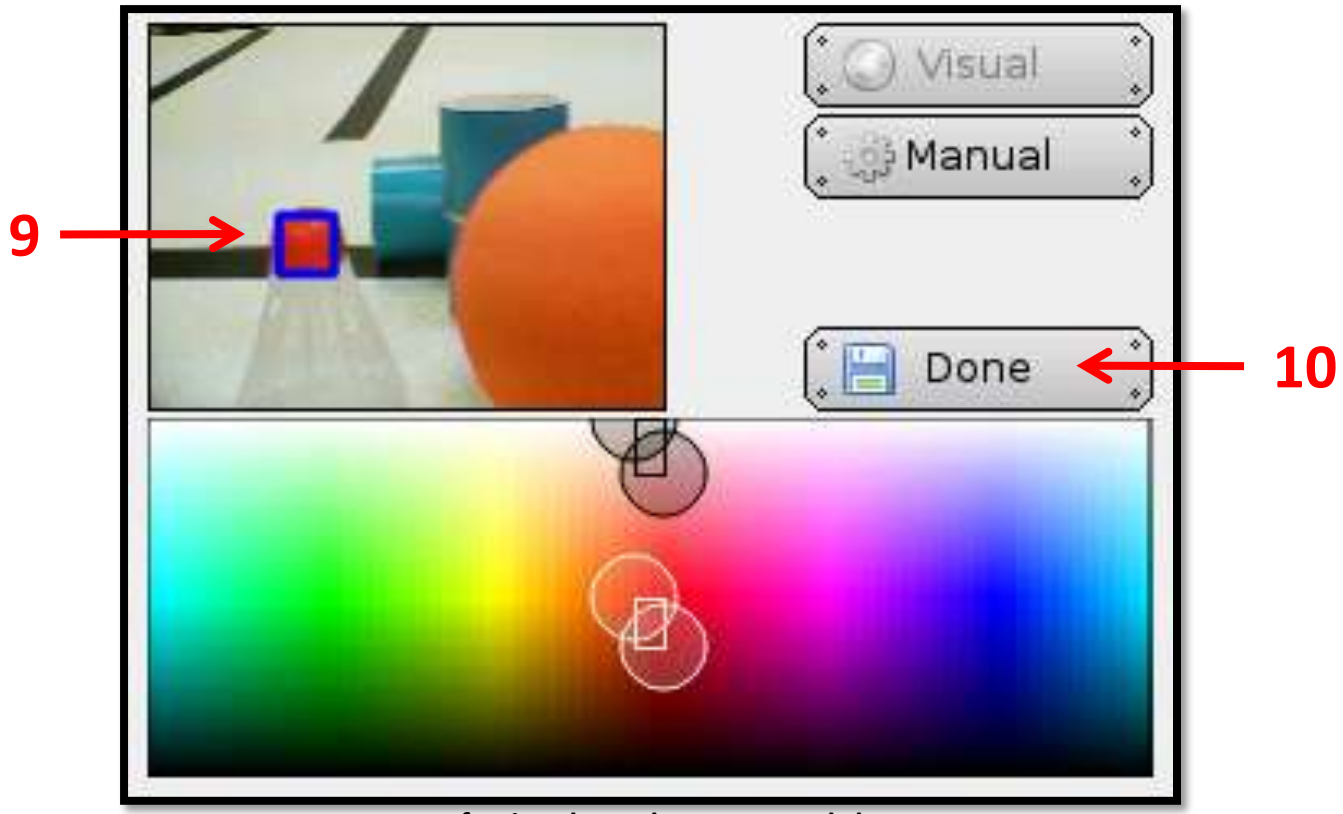
© 1993 – 2016 KIPR

Botball®

Setting the color tracking channels:TBD

Activity

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**.



© 1993 – 2016 KIPR

Tuning the color tracking channels:TBD

For additional details about
setting and *tuning* the color tracking channels,
please see the following instructional video:

https://www.youtube.com/watch?v=nSszFa7opMA

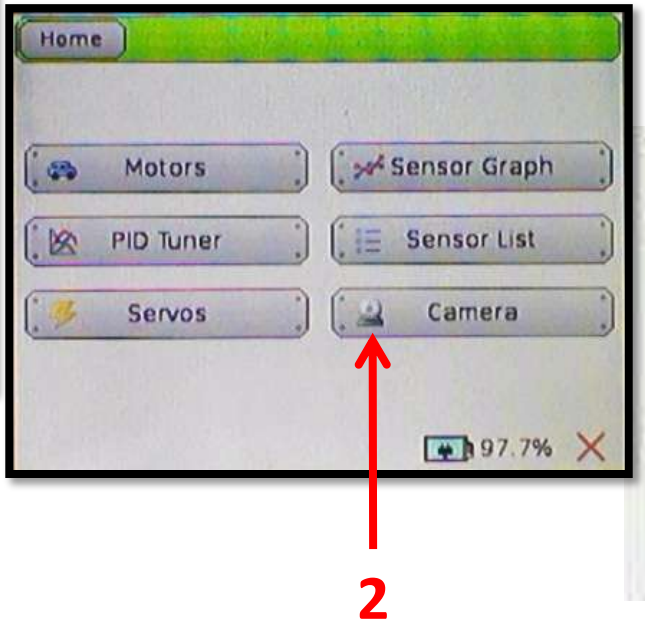
© 1993 – 2016 KIPR

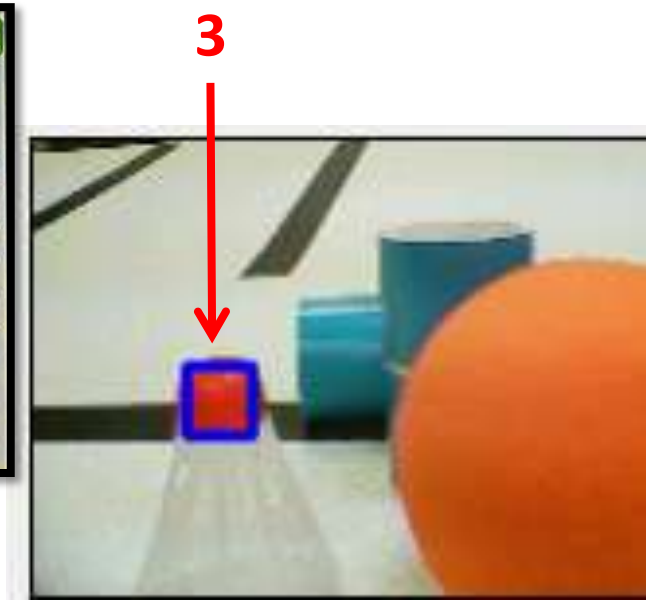
Verify the color channel is working:TBD

Activity

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**.



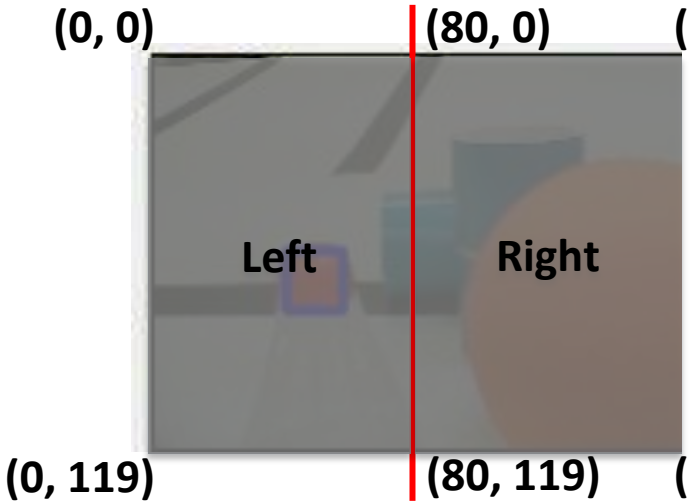




© 1993 – 2016 KIPR

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.
```


© 1993 – 2016 KIPR

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.
```


© 1993 – 2016 KIPR

Using camera functions

```
int main()
{
  camera_open();
  while (digital(8) != 1)
  {
    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;
} // end main
```

What do these say?

© 1993 – 2016 KIPR

Tracking an object

Activity

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

- This is the same type program as the line follow activity, but instead of the reflectance sensor it is using the camera—**consider copying and pasting from your line-following program!**
- Anything *less than* 80 is to the left, so turn left.
- Anything *greater than* 80 is to the right, so turn right
- If it does not see anything, then it stops.

© 1993 – 2016 KIPR



Tracking an object

Activity

Analysis: What is the program supposed to do?

Pseudocode

1. Open camera connection.
2. Loop: Is not touched?
    1. Update camera.
    2. If: Is object not detected?
        1. Stop motors.
    3. Else.
        1. If: Is object on left?
            1. Turn left.
        2. Else.
            1. Turn right.
3. Stop motors.
4. Close camera connection.
5. End the program.

Comments

```
// 1. Open camera connection.
// 2. Loop: Is not touched?
//    2.1. Update camera.
//    2.2. If: Is object not detected?
//         2.2.1. Stop motors.
//    2.3. Else.
//         2.3.1. Is object on left?
//                2.3.1.1. Turn left.
//         2.3.2. Else.
//                2.3.2.1. Turn right.
// 3. Stop motors.
// 4. Close camera connection.
// 5. End the program.
```


© 1993 – 2016 KIPR

Tracking an object

Solution:

Pseudocode (Comments)

```
int main()
{
  // 1. Open camera connection.
  // 2. Loop: Is not touched?
  //    2.1. Update camera.
  //    2.2. If: Is object not detected?
  //         2.2.1. Stop motors.
  //    2.3. Else.
  //         2.3.1. If: Is object on left?
  //                2.3.1.1. Turn left.
  //         2.3.2. Else.
  //                2.3.2.1. Turn right.
  // 3. Stop motors.
  // 4. Close camera connection.
  // 5. End the program.
}
```

Source Code

Activity

```
int main()
{
  // 1. Open camera connection.
  camera_open();

  // 2. Loop: Is not touched?
  while (digital(8) != 1)
  {
    // 2.1. Update camera.
    camera_update();

    // 2.2. If: Is object not detected?
    if (get_object_count(0) == 0)
    {
      // 2.2.1. Stop motors.
      ao();
    } // end if
    // 2.3. Else
    else
    {
      // 2.3.1. If: Is object on left?
      if (get_object_center_x(0, 0) < 80)
      {
        // 2.3.1.1. Turn left.
        mav(0, -500);
        mav(2,  500);
      } // end if
      // 2.3.2. Else.
      else
      {
        // 2.3.2.1. Turn right.
        mav(0,  500);
        mav(2, -500);
      } // end else
    } // end if
  } // end while

  // 3. Stop motors.
  ao();

  // 4. Close camera connection.
  camera_close();

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

Tracking an object

Solution: (using two functions)

Pseudocode (Comments)

```
int main()
{
  // 1. Open camera connection.
  // 2. Loop: Is not touched?
  //    2.1. Update camera.
  //    2.2. If: Is object not detected?
  //         2.2.1. Stop motors.
  //    2.3. Else.
  //         2.3.1. If: Is object on left?
  //                2.3.1.1. Turn left.
  //         2.3.2. Else.
  //                2.3.2.1. Turn right.
  // 3. Stop motors.
  // 4. Close camera connection.
  // 5. End the program.
}
```

Source Code

Activity

```
// Function prototypes for turn_left and turn_right.
void turn_left();
void turn_right();

// Function definition for main.
int main()
{
  // 1. Open camera connection.
  camera_open();

  // 2. Loop: Is not touched?
  while (digital(8) != 1)
  {
    // 2.1. Update camera.
    camera_update();

    // 2.2. If: Is object not detected?
    if (get_object_count(0) == 0)
    {
      // 2.2.1. Stop motors.
      ao();
    } // end if
    // 2.3. Else
    else
    {
      // 2.3.1. If: Is object on left?
      if (get_object_center_x(0, 0) < 80)
      {
        // 2.3.1.1. Turn left.
        turn_left();
      } // end if
      // 2.3.2. Else.
      else
      {
        // 2.3.2.1. Turn right.
        turn_right()
      } // end else
    } // end if
  } // end while

  // 3. Stop motors.
  ao();

  // 4. Close camera connection.
  camera_close();

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

// Function definition for turn_left.
void turn_left()
{
  // Turn/arc left.
  mav(0, -500);
  mav(2, 500);
} // end turn_left

// Function definition for turn_right.
void turn_right()
{
  // Turn/arc right.
  mav(0, 500);
  mav(2, -500);
} // end turn_right
```



Thanks and have a great season!

```
}   // end workshop
```

Please take our survey to give feedback about the workshop:

https://www.surveymonkey.com/s/Botball_workshop8RHYLG6

© 1993 – 2016 KIPR