



# Botball 2014

# Educators' Workshop

V1.0.8 2014.01.21



# Day 1: Getting Started

1. Sign in and collect your materials and electronics
2. Charge your Link Controllers (next slide)
3. Update the firmware on your KIPR Link controllers (slide 37)
4. Install current version of the KISS IDE and KIPR Link USB driver for your computer (slide 38)
5. Put battery in Create and charge the battery
6. Go through the parts list and materials and verify everything is present
7. Start building the DemoBot (see DemoBot guide)

Are you confused?  
Please ask the KIPR staff for help!



# Charging the KIPR Link Controller

- For charging the KIPR Link, **use only the power supply which came with your Link**
  - Damage to the Link from using the wrong charger is easily detected and will void your warranty!
- The KIPR Link power pack is a lithium polymer battery so the rules for charging a lithium battery for any electronic device apply
  - You should NOT leave the unit unattended while charging
  - Charge away from any flammable materials and in a cool, open area



# 2014 Botball National Sponsors



**iRobot®**

**igus®**

**D3S SolidWorks**

**COMMON SENSE RC**

**Botball®**



# 2014 Regional Botball Sponsors



To Oklahoma & You.<sup>SM</sup>



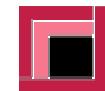
Oklahoma  
Aeronautics  
Commission



*Rockwell*  
*Collins*



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



KIRKPATRICK FOUNDATION



**robonation**  
ROBOTICS COMMUNITY

**Botball**<sup>®</sup>

# 2014 Regional Workshop & Tournament Hosts



GROSSMONT  
COLLEGE



*Preparing people to lead extraordinary lives*



WORCESTER  
STATE  
UNIVERSITY



جامعة كارنيجي ميلون في قطر  
**Carnegie Mellon Qatar**



**Botball®**



# Botball 2014

©1993-2014 KISS Institute for Practical Robotics  
prepared by:

The KISS Institute for Practical Robotics

with significant contributions from  
the staff of KIPR and the Botball Instructors Summit participants



# 2014 is KIPR's 20<sup>th</sup> Anniversary!

KIPR welcomes its new regional in China!

KISS  
INSTITUTE<sup>FOR</sup>  
PRACTICAL  
ROBOTICS



# Housekeeping

## Day 1

- Bathrooms and Food
- Introductions
  - First time teams are identified by a colored name tag
    - If you've done this before, then you know how they feel!
    - Please help them out
  - Workshop staff
- Daily schedule



# Workshop Schedule

- Day 1:
  - Overview of Botball
    - Botball season, related events
    - Game preview/video
    - Resources & teams
  - Topics and Activities
    - Activity 0: The KISS IDE
    - Activity 1: Programming basics
    - Activity 2: Driving straight
    - Activity 3: Build DemoBot (throughout)
  - Lunch
    - Activity 4: Conditions and functions
    - Activity 5: Starting / shutting down the robot using sensors
    - Activity 6: Motors and servos
    - Activity 7: Line following
  - Homework
- Day 2:
  - New Team Suggestions
  - BOPD
  - 30 minute game Q&A
  - T-shirts and Awards
  - Topics and Activities
    - Activity 8: Depth sensor
  - Lunch
    - Activity 9: Vision
  - Selected activities
    - Activity 10: Point servo at colored object
    - Activity 11: Bang-Bang control
    - Activity 12: Proportional control
    - Activity 13: Bang-Bang DemoBot arm
    - Activity 14: Proportional DemoBot arm
    - Activity 15: Accelerometer for bump detect
    - Activity 16: Graphics for custom user interfaces
    - Activity 17: Reducing accumulated errors
    - Activity 18: Reading QR code



# Overview of Botball

- Botball is brought to you by the KISS Institute for Practical Robotics (KIPR)
- Botball season
- Game preview/video
- Botball Related events
  - Global Conference on Educational Robotics (GCER)
  - KIPR Open
- Related curriculum topics



# 2014 Botball Season

## Jan - Mar 2014

- Botball Professional Development Workshops
- KIPR Open Game released

## February 2014

- GCER Call for Papers
- GCER Call for Autonomous Showcase submissions

## Mar - May 2014

- Botball Regional Tournaments

## April 2014

- ECER

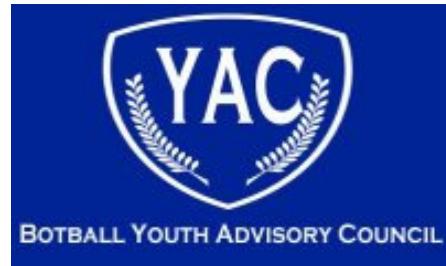
## Spring 2014

- GCER Registration Opens

## July-August 2014

Global Conference on Educational Robotics (GCER)

- Fun and networking
- International Botball Tournament
- KIPR Open Tournament
- Autonomous Showcase
- Presentations/Papers/Guest Speakers



# The Botball Youth Advisory Council

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!



## A place for Botballers

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

A Botball YAC and KIPR Project



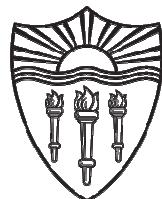
# ECER14 - Hard Facts

- [ European Conference on Educational Robotics
  - 3<sup>rd</sup> Botball competition in Europe
- [ Venue: Vienna, Austria
  - TGM (Vienna Institute of Technology)
- [ Two main parts:
  - European Botball Competition
  - Talks by Researchers and Students



April  
8th - 11th  
2014

# GCER 2014



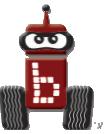
**USC** University of  
Southern California



The 2014 Global Conference on Educational Robotics will be held at the University of Southern California Galen Center from **July 30th - August 3rd, 2014** with preconference classes on July 29<sup>th</sup>

**Global Conference on Educational Robotics**

<http://www.kipr.org/gcer>



# Global Conference on Educational Robotics

## ALL TEAMS ARE INVITED!

### When

- July 30<sup>th</sup> – August 3<sup>rd</sup>
- Pre-conference activities and workshops July 29<sup>th</sup>



### Who

- Middle school and high school students, educators, robotics enthusiasts, and professionals from around the world

### Activities

- Meet and network with students from around the country and world
- Talks by internationally recognized robotics experts
- Teacher, student, and peer reviewed track sessions
- International Botball Tournament
- KIPR Open Tournament (Botball for grown-up kids!)
- Aerial Robotics Competition
- Autonomous Robotics Showcase
- Elementary Botball Challenge



# GCER Preconference

The 2014 Preconference classes haven't been scheduled yet. Keep an eye out for them on the GCER website!

## GCER 2013 Preconference Classes

- Learning to MAKE “Smart” Robots
- Expressive Robotics: Motion and Emotion
- Learning About Microcontrollers
- Programming and Controlling Micro-Helicopters with Vision
- Making Sense of Sensors





# Coming July 2015



## GCER 2015

Location and Date TBA

**Global Conference on Educational Robotics**  
<http://www.kipr.org/gcer>





# KIPR Open Tournament

- The KIPR Open is a tournament produced by KIPR to encourage ongoing robotics educational activity beyond high school and Botball
- KIPR Open team entry forms and conference registration can be found at [www.kipr.org/kipr\\_open](http://www.kipr.org/kipr_open)
- The 2014 International KIPR Open Tournament will be held in conjunction with GCER 2014 July 30-August 3, 2014
  - See the KIPR website for information on KIPR Open Tournaments
- Collegiate courses are encouraged to incorporate the KIPR Open Game into their curriculum



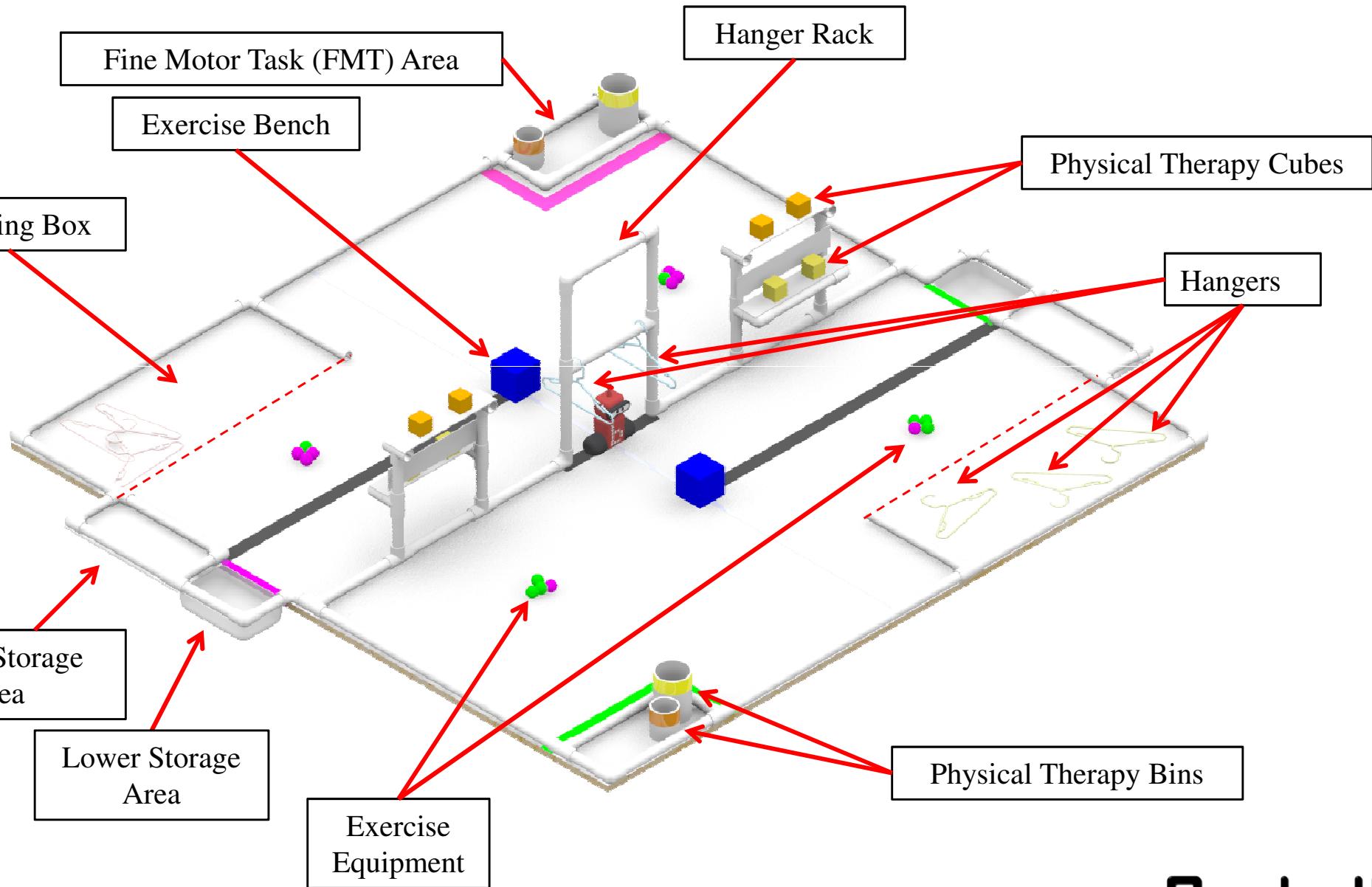
# Preview of This Year's Game

Hold Your Questions! Game Q&A is Tomorrow



# The Game Board

Hold Your Questions! Complete Review is Tomorrow





# Tonight

- **Review the game rules on your team home base.** We will have a 30 minute Q&A session tomorrow.
- Ask questions about game rules in the Game Rules Forum (the rules are at the Team Home Base)
  - You should regularly review this Forum
  - You will find answers to game questions there



# Team Home Base Resources

*At [homebase.kipr.org](http://homebase.kipr.org)*



The screenshot shows the homepage of the Botball Team Home Base. The top half features the Botball logo with a red robot head and the text "STANDARDS-BASED EDUCATIONAL ROBOTICS PROGRAM". Below the logo is a photograph of a robotics competition event with many spectators and robots on the floor. The bottom half contains a white box with the text "Welcome to the Botball Team Home Base" and "2014 Team Home Base". It also includes a list of resources: "The Team Home Base is your resource for:" followed by a bulleted list: "• Botball online project documentation", "• Botball game FAQs", and "• Other Botball game related resources".

Welcome to the Botball Team Home Base

---

[2014 Team Home Base](#)

The Team Home Base is your resource for:

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



# Resources!

- **On your Team Home Base**
  - 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 2014 Game Documents
- **Botball Community Site:** <http://community.botball.org/>
- **KISS Institute for Practical Robotics Tech Support:** +1-405-579-4609



# Why are we teaching this?

- There are enormous problems facing humanity today.
- There are also small, but important problems that no one has figured out how to solve.
- Today's students will need to approach these challenges in innovative ways.



Science, technology, engineering,  
math, and computer science

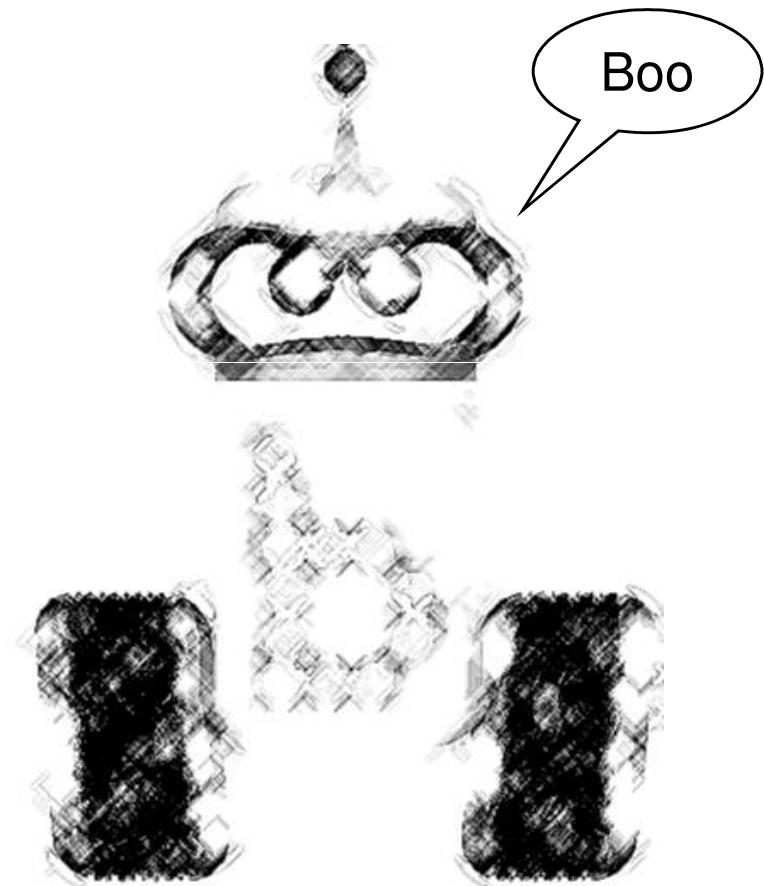
***Botball provides real life  
learning opportunities***

Project management, teamwork,  
communication, and leadership



# The Spirit of Botball

- Botball is an educational experience for students
- Parents, teachers and mentors are there to guide, not to *do*
- Adults who want to *do* should build practice boards and work on an entry for the KIPR Open
- Parents and mentors should set good examples of behaviour and sportsmanship -- especially at tournaments





# Successful Botball Team Members . . .

- Communicate and work well as individuals and as a team
- Understand how to look at a challenging situation and break it into solvable problems
- Think about how to apply what they've learned to other things outside of Botball
- Try a lot of approaches just to see what happens
- Have fun!



# For a Successful Botball Entry

- **Organize** your team: people, time, equipment
- **Think** about the problem before building
- Pick parts of the problem to **solve**
- Build a team of robots that **complement** each other
- Build software and hardware **together**
- **Document** everything and use it for team communications
- Spend as much or more time **testing** and tuning as building
- **Observe** what other teams do at regionals and then use what you **learn** to make an improved entry for GCER
- Use **checklists**



# Thank you for participating!



We couldn't do it  
without you!

## 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.





# Our Larger Mission





# Activity 0

## The KISS Platform and IDE

KIPR Instructional Software System Platform  
and Integrated Development Environment (IDE)



# Activity 0: Objectives

## The KISS Platform and IDE

- Update the Firmware on your KIPR Link
- Install the KISS Platform on your personal computer



# Charging the KIPR Link Controller

- For charging the KIPR Link, **use only the power supply which came with your Link**
  - Damage to the Link from using the wrong charger is easily detected and will void your warranty!
- The KIPR Link power pack is a lithium polymer battery so the rules for charging a lithium battery for any electronic device apply
  - You should NOT leave the unit unattended while charging
  - Charge away from any flammable materials and in a cool, open area



# Is your KIPR Link Firmware Up to Date?

Probably not...

- A quick visit to the KIPR web site is all that is needed to check if your KIPR Link has the most recent version of its embedded firmware.
- Procedure to check firmware version
  1. Slide the power switch to turn on the KIPR Link (the screen will turn off for several seconds when booting).
  2. The home screen has an *About* tab to identify the firmware version.
  3. The current firmware release is at  

<http://www.kipr.org/hardware-software>
  4. If your version number is lower than the current version, it's time to update!



# KIPR Link Firmware Update Procedure

This is only needed if firmware isn't up to date!

1. Download and copy the firmware update file (*kovan\_update.img.gz*) to a USB memory stick in the top level directory. Download site is  

<http://www.kipr.org/hardware-software>
2. Connect the KIPR Link Charger to your Link (important!!)
3. Turn off your KIPR Link and while holding down the side button (opposite side from the power switch) turn your KIPR Link back on
  - The normal splash screen will first come up, followed by a screen instructing you to insert the flash drive
4. Insert the flash drive in either USB port
  - A progress bar will appear and in a short time the update will complete
  - Your KIPR Link will then come up with its opening screen
  - **Click on *Settings >> Calibrate* to calibrate screen touch settings**



# The KISS Platform

- The KISS Platform programming environment is "donation ware"
  - It can be freely distributed and used for personal and educational purposes
  - If you like it, please make a donation to KIPR (tax deductible!)
  - If you would like to use the KISS Platform or IDE in a commercial product, contact KIPR about licensing
- The latest version of the KISS Platform for your type of computer is at the same web site as the KIPR Link firmware:

<http://www.kipr.org/hardware-software>

- It is also included in the electronic media distributed to participants of Botball workshops through the Team Home Base



# KISS Platform Installation

- Installation is the same as for most other software applications (the Team Home Base also includes instructions)
- The KISS Platform IDE supports Mac OS versions from 10.7 up, and Windows from Vista through 8 (for Windows 8 see KISS Platform website)
  - For Windows, the KISS Platform installer, in addition to the KISS Platform, (optionally) installs necessary support software, drivers, and documentation (which are necessary for a first install)
    - If you are running Windows 7 you will need to right click on the installer.exe file so it runs in administrator mode
    - If you are using a Mac, then you will need to install the Developer Command Line Tools on your system, unless you did so at some earlier time
      - For OS versions from 10.7 and up, the tools are a 171Mb download from either the Apple Developer or KIPR web site. There will be media at the workshop for the different OS versions.



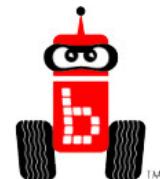
# Support for Windows XP??

- Since Microsoft is ending support for Windows XP this Spring, KIPR is following suit by no longer providing support for using the KISS Platform with Windows XP, although there is an XP-specific install that may work for XP SP3 machines
- More to the point, the KISS Platform uses features that may not be supported by Windows XP.
- Windows XP was originally released on August 24<sup>th</sup>, 2001. This is before some of you were born!





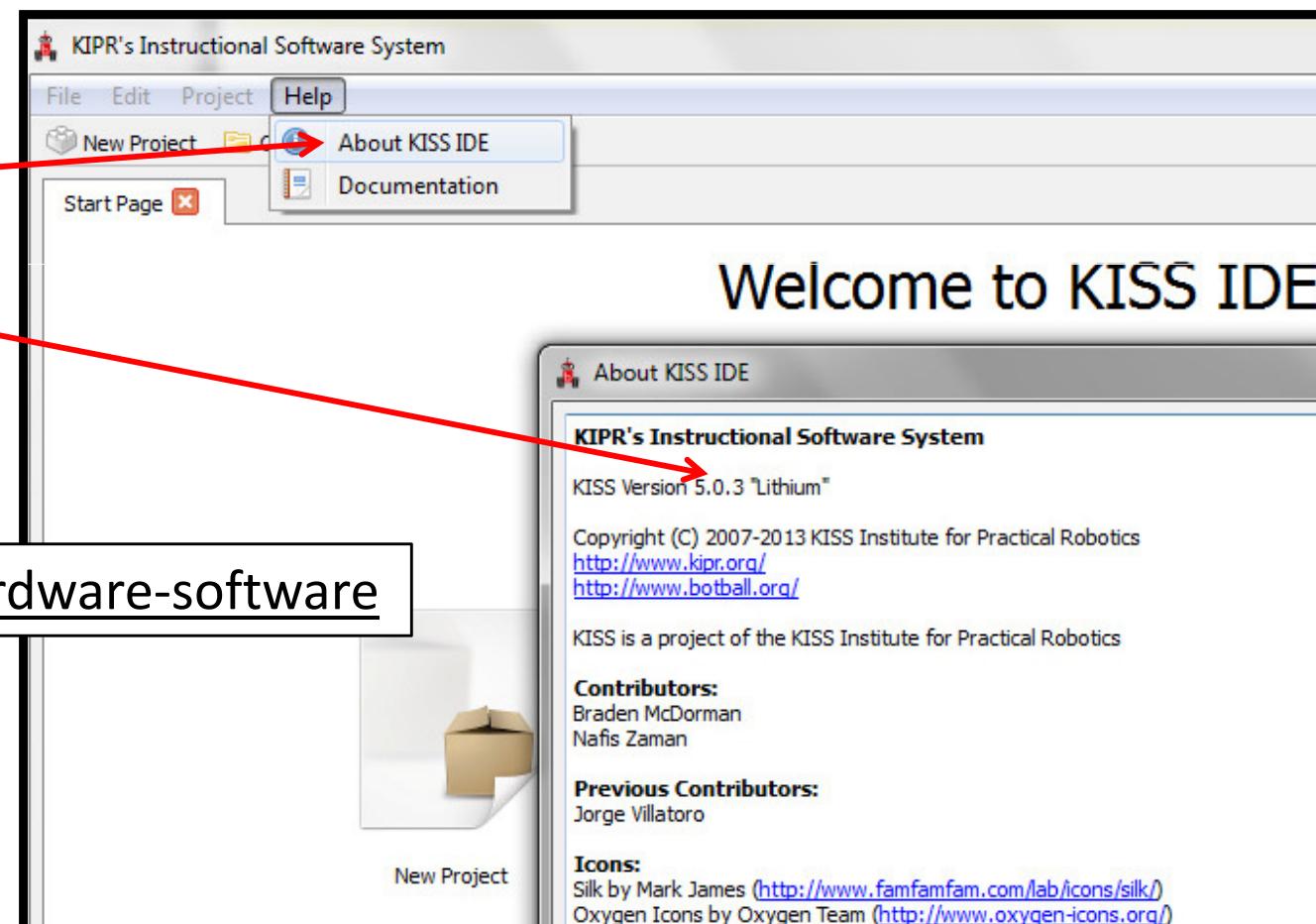
# KISS Platform Version



- If your installation was successful, a KISS IDE icon should now be on your computer's desktop (Win) or in your Applications folder (Mac)
- Launch the KISS Platform IDE by clicking on the icon; the Welcome screen will appear
- Click on Help and select "About KISS IDE"
- Verify version number is current
  - Version 5.0 or higher
  - check

<http://www.kipr.org/hardware-software>

to see if your version is current and to download latest version





# Using the KISS Platform

- The heart of the KISS Platform is an IDE with facilities for simplifying the production of programs for the KIPR Link robotics controller
- The integrated editor and robot simulator can be used whether or not a robot controller is connected
- Programs can be checked for syntax errors such as typos from within the IDE interface
- To check for logic errors you:
  - Can simulate execution of your program using the built in graphical simulator
  - Attach a KIPR Link controller and try running your program



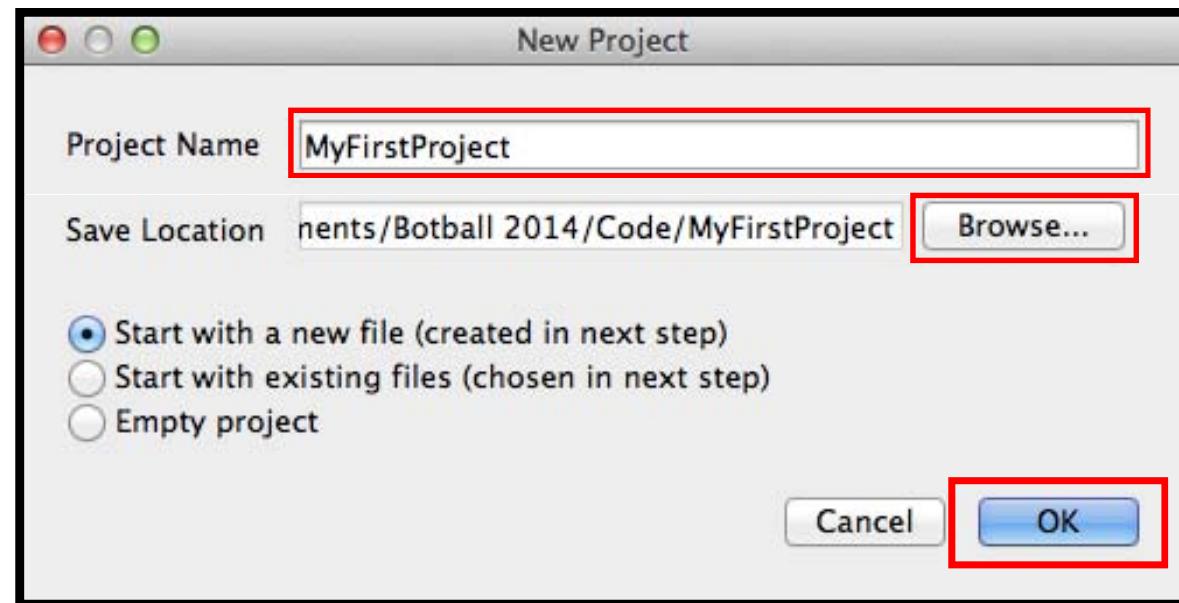
# Let's rock and roll!

Bring up the KISS IDE and select New Project



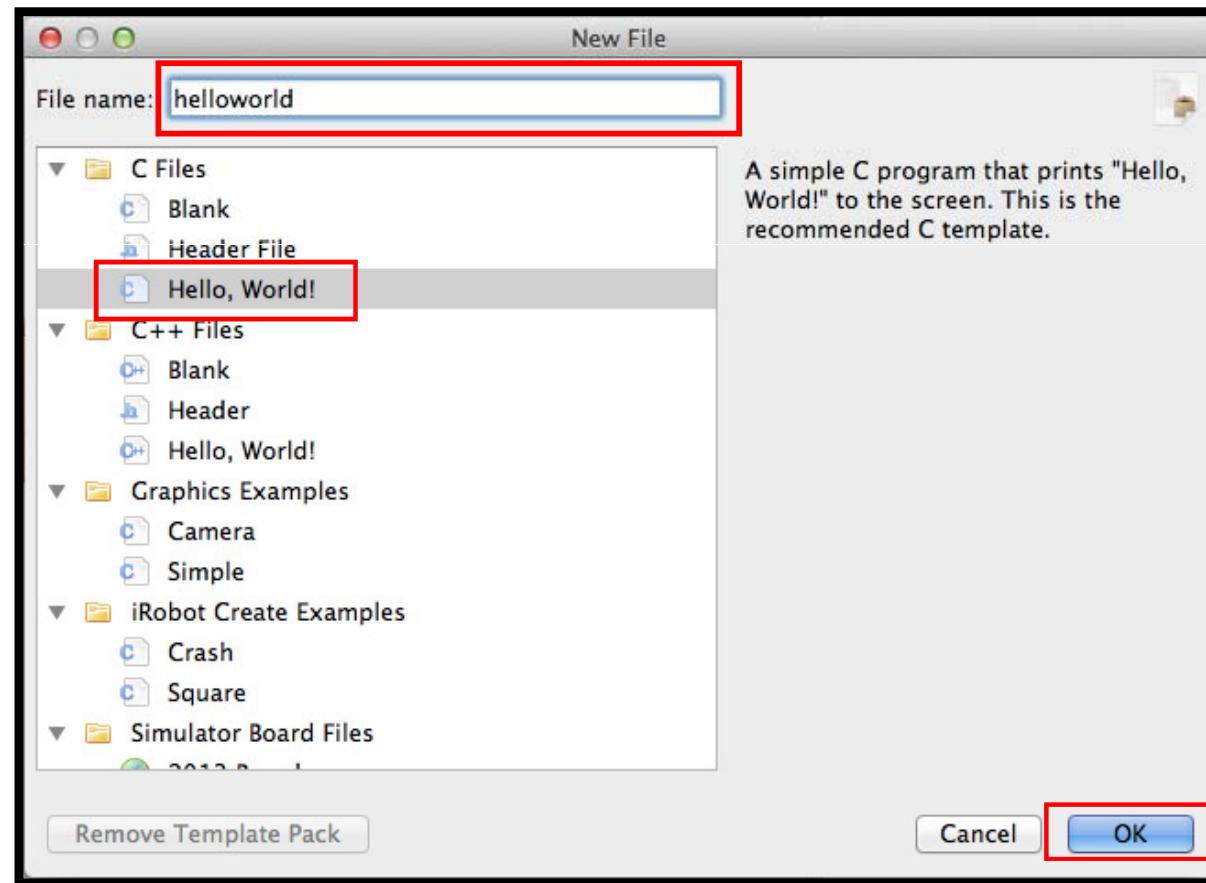


# Name your project, select where to save it, and click OK.





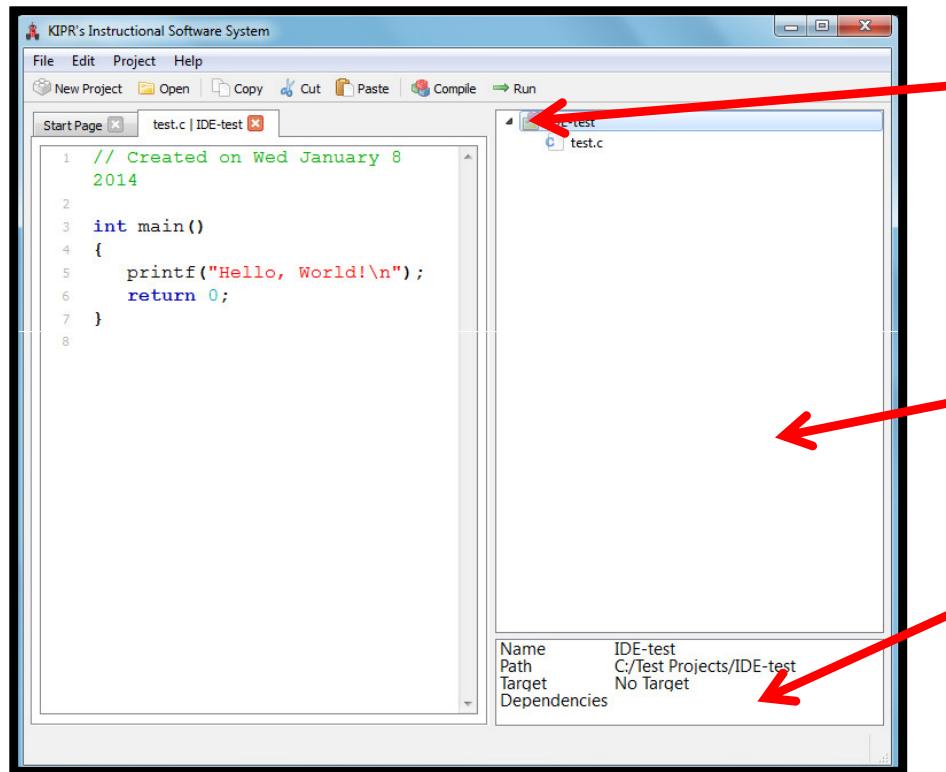
# Name your file, select “Hello, World!”, and then hit OK.





# Verifying the IDE Operation

A few of the IDE features

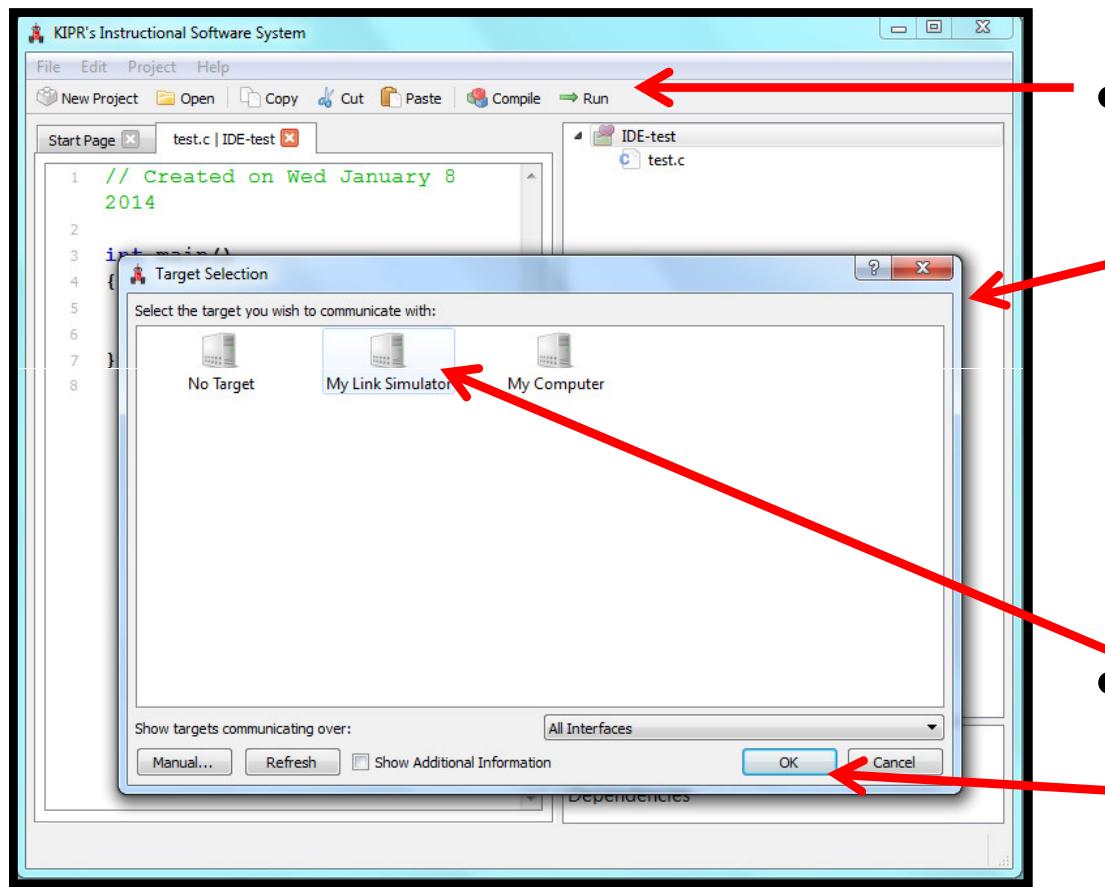


- Each project has a marker to click to show or hide its contents
- Panels can be resized via the usual click and drag
- Clicking on a panel entry shows its properties

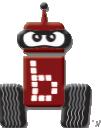


# Verifying the IDE Operation

## Running a Project



- Clicking *Run* brings up the *Target Selection* window
  - If a KIPR Link is plugged in, it will also appear in the listing
- Highlight the target and click *OK* to run it (in the simulator in this case)



# Click Run!

The screenshot shows the KIPR's Instructional Software System interface. The window title is "KIPR's Instructional Software System". The menu bar includes "New Project", "Open", "Copy", "Cut", "Paste", "Compile", and "Run". A red box highlights the "Run" button. The main workspace displays a C program:

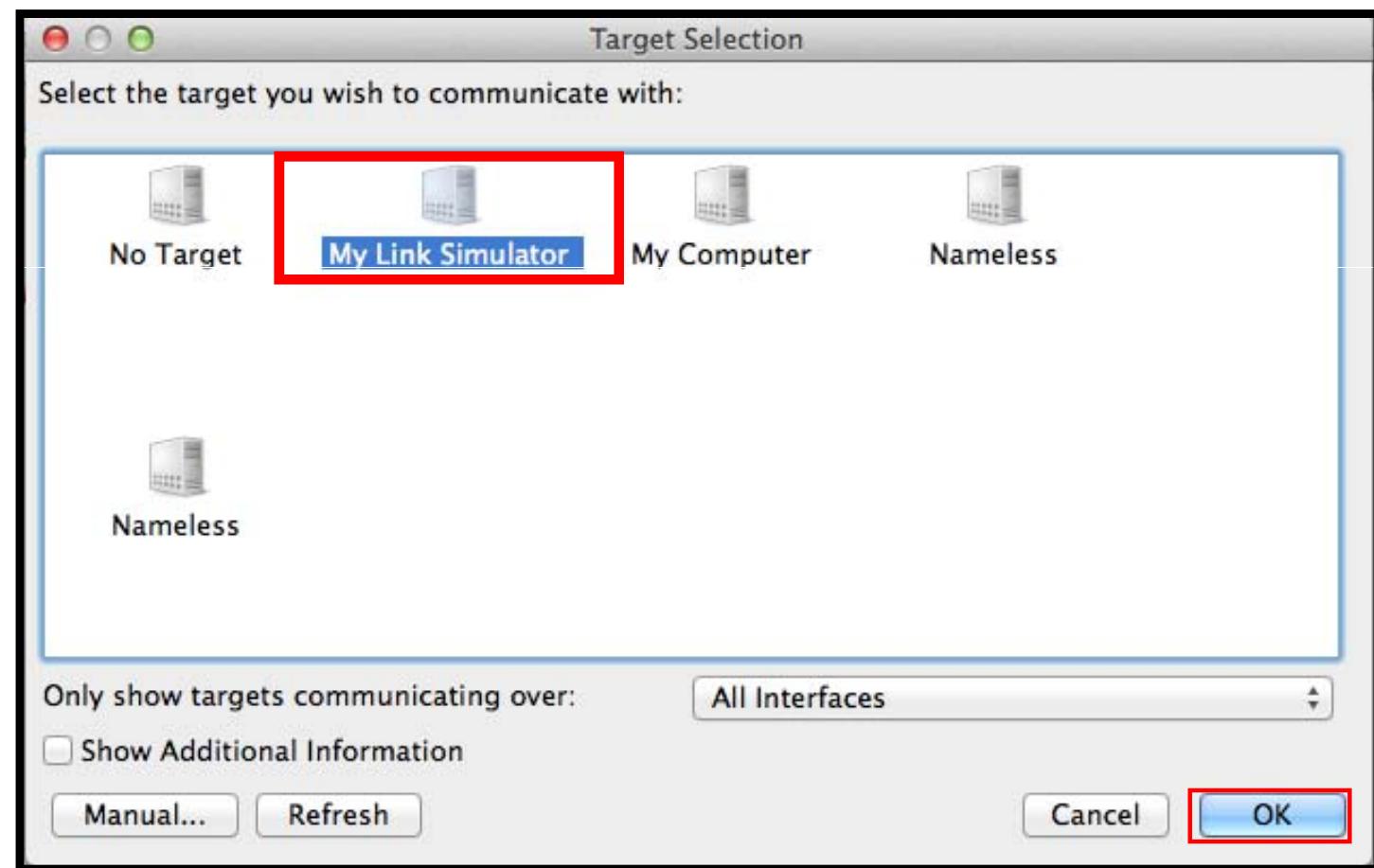
```
// Created on Fri January 10 2014
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

The project sidebar shows "MyFirstProject". The project details pane at the bottom right lists:

Name	MyFirstProject
Path	/Users/wymyers/Documents/B
Target	No Target
Dependencies	

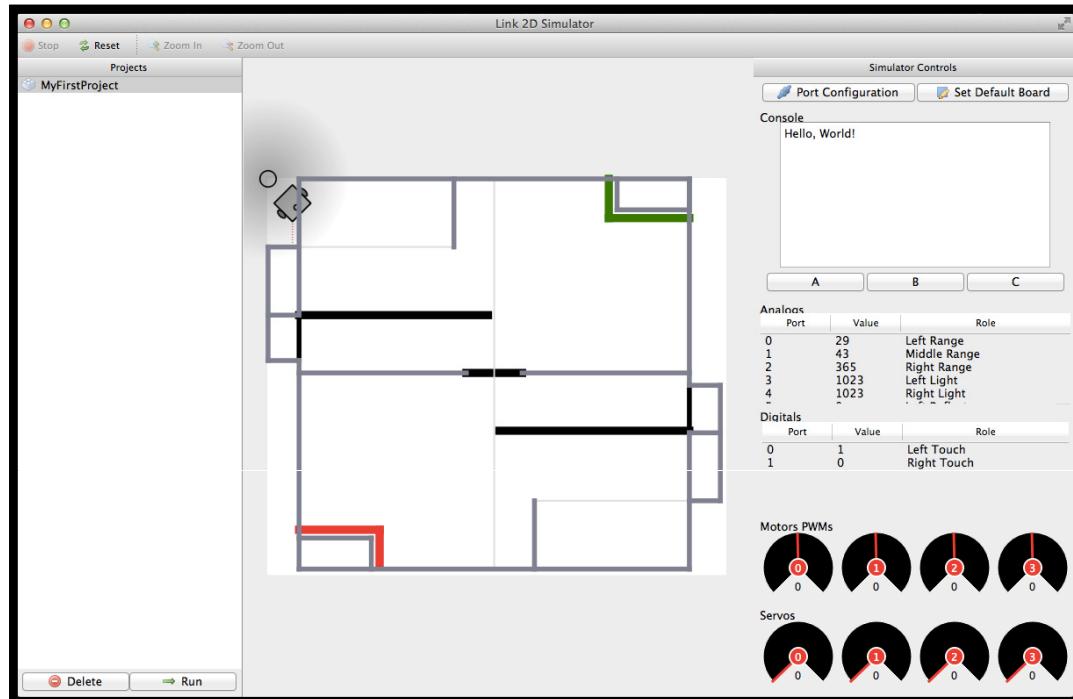


# Select My Link Simulator and click OK





# Output!



**It works!**

Now let's make something move!



# Let's Make Something Move!

- We are going to make a robot move
- We will gloss over a bunch of stuff
  - Don't worry if you don't understand why something works
  - Just worry if it doesn't – ask for help if needed



# Attach Computer to KIPR Link

USB Cable in Botball Kit



See Quick Start Guide in KIPR Link Manual on  
Team Home Base for more detail



# Power Up the KIPR Link

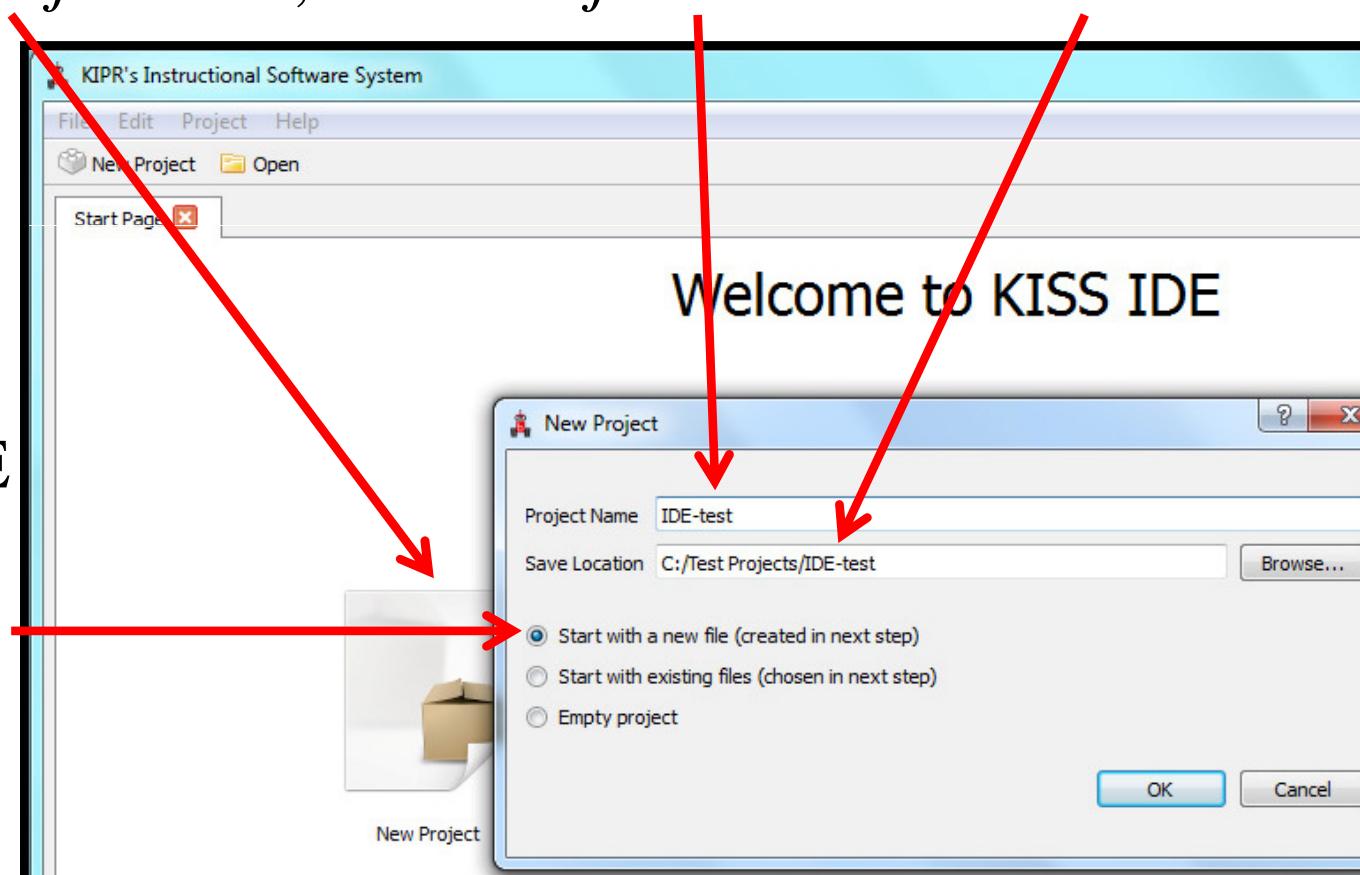
## Opening Screen





# Making a New Project

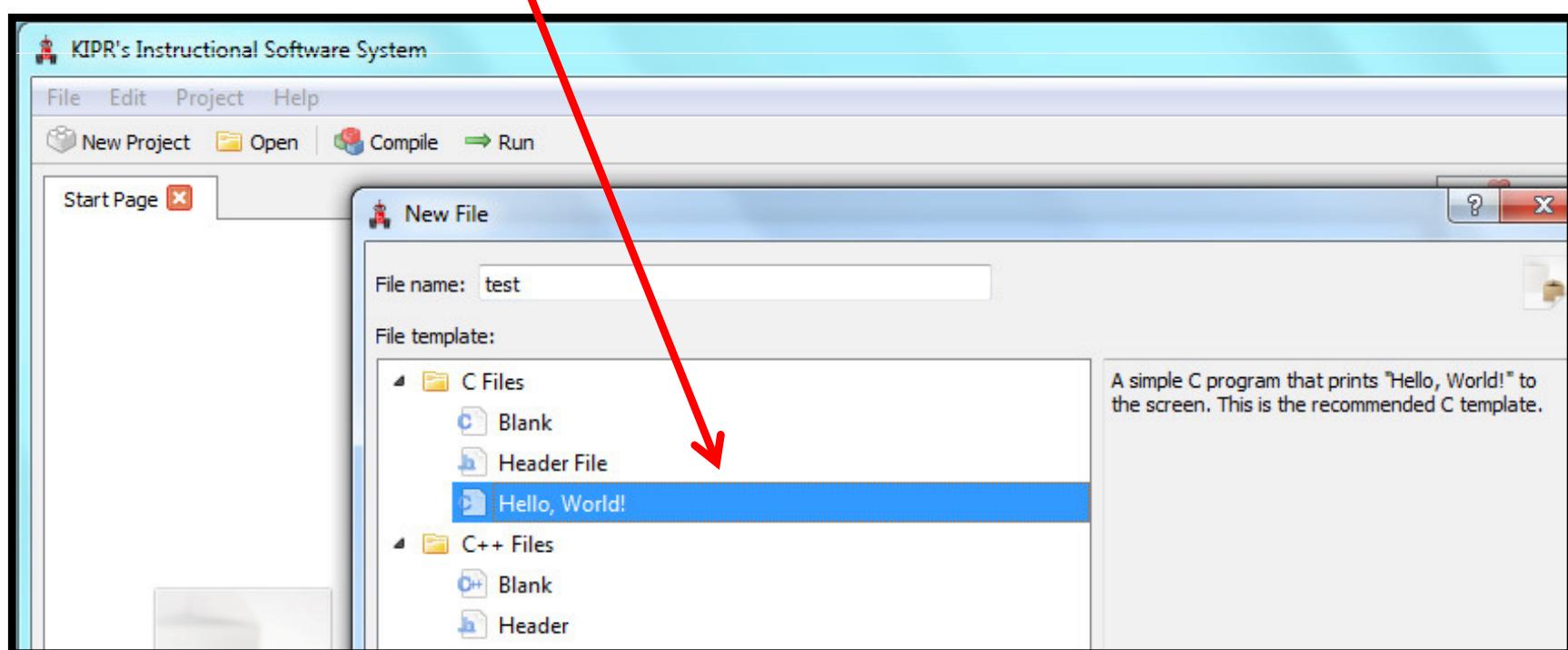
- If not already started, launch the KISS IDE by clicking on its icon to get the Welcome screen
- Click on the *New Project* icon, enter *Project Name* and *Save Location*
- The default start-up option will be the one you usually use (and directs the IDE to prompt you to select an initial file for your project)





# Add a Program File to Your Project

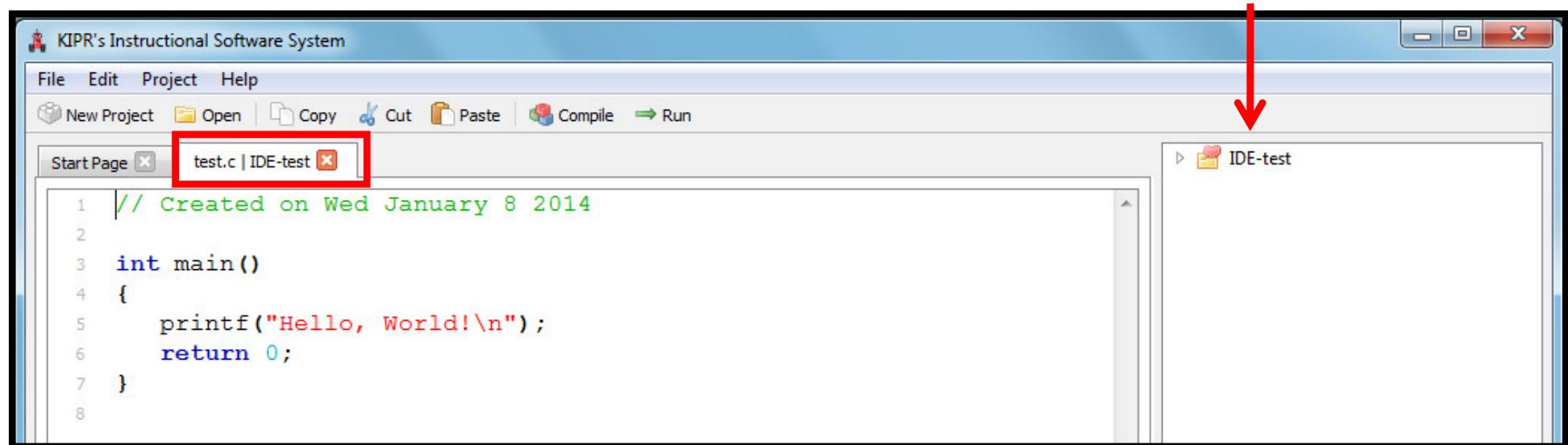
- A *New File* selection will appear when you continue your IDE dialogue
- Select the “*Hello World!*” **C** program file





# Now You Are Set to Enter the Code on the Next Slide

- The IDE
  - Opens a tab for editing your program
    - Adds .c to the program name if you didn't
  - Shows your new project in the open projects panel



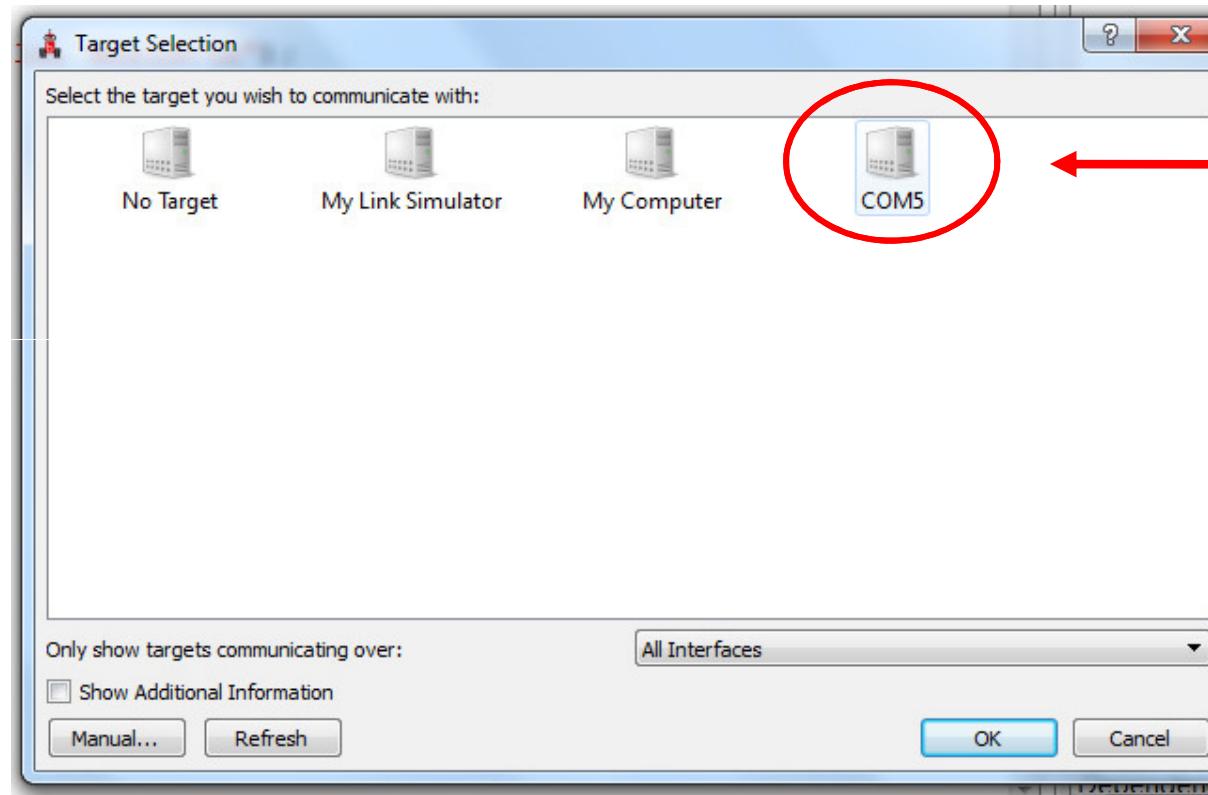


# Make a New Project and Add a Program File to It That Has a Copy of This Code

```
// Spin the create CW about 1 rev
int main()
{
    printf("connecting to the Create\n");
    create_connect();
    //Spin CW
    create_drive_direct(200,-200);
    msleep(3000); // for 3 secs
    create_stop();
    create_disconnect();
    printf("Program is done!\n");
    return 0;
}
```



# Compile Your Project and Select Your KIPR Link as Target



Icon for your Link should appear here within a few seconds after the target selection screen opens

If the compile was successful, your project is now on your KIPR Link!



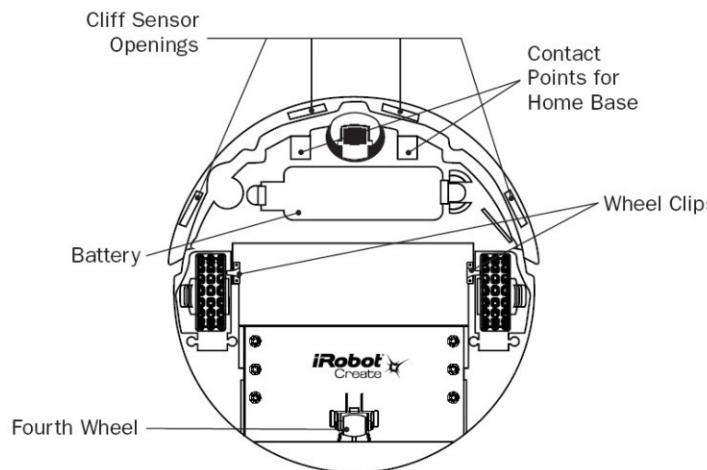
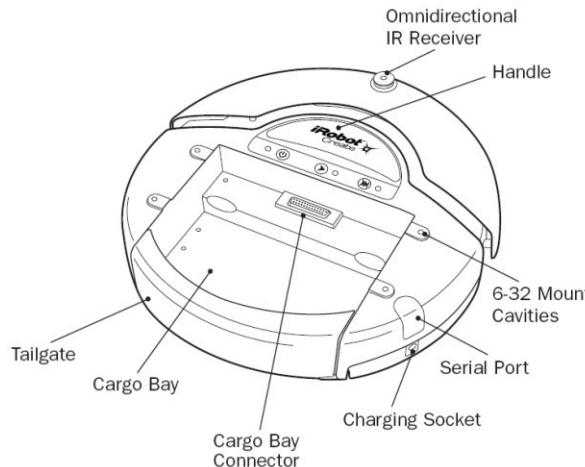
# Connect Your Link to Your Create Module using the Link-Create Cable



- The KIPR Link-Create Cable is keyed, so it will only insert in the correct orientation
- The KIPR Link-Create Cable is keyed, so you will need to release the lock before removing
  - **DO NOT JUST PULL**
- The Create will only TRICKLE charge the Link.



# Put KIPR Link in Create Cargo Bay



*From iRobot Create specs*



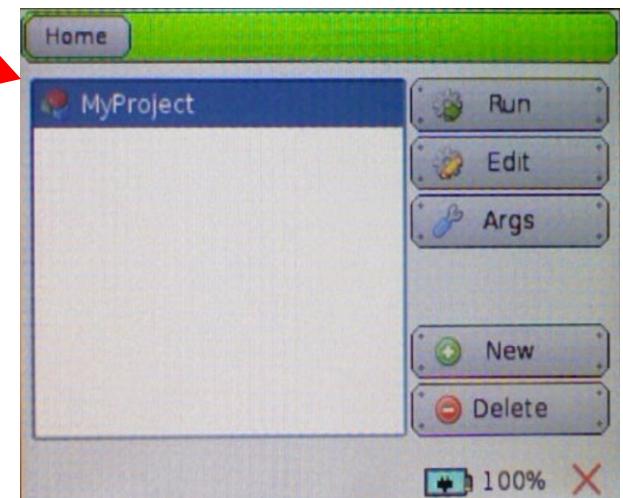
If not already done so, install the **yellow** battery into the bottom of the Create – make sure it snaps in completely. Connect the Create charger when not using the Create.

See Quick Start Guide for more details on connecting the Create to the KIPR Link



# Running Your Project's Program on the Link

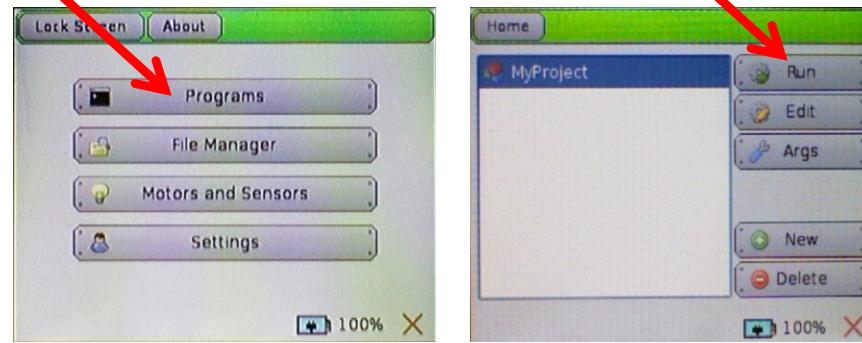
- When you compile a project to the KIPR Link as the IDE Target, it is automatically downloaded and made ready to run
- Under the *Programs* menu on your KIPR Link, select your project
- Pressing *Run* causes the specified project to run
  - Your project code is retained on the KIPR Link under the *Programs* menu until manually deleted
- If your Link is connected to a Create, **be sure that the Create is turned on** before you press *Run!*





# Running Your Project's Program: Make Your Create Move

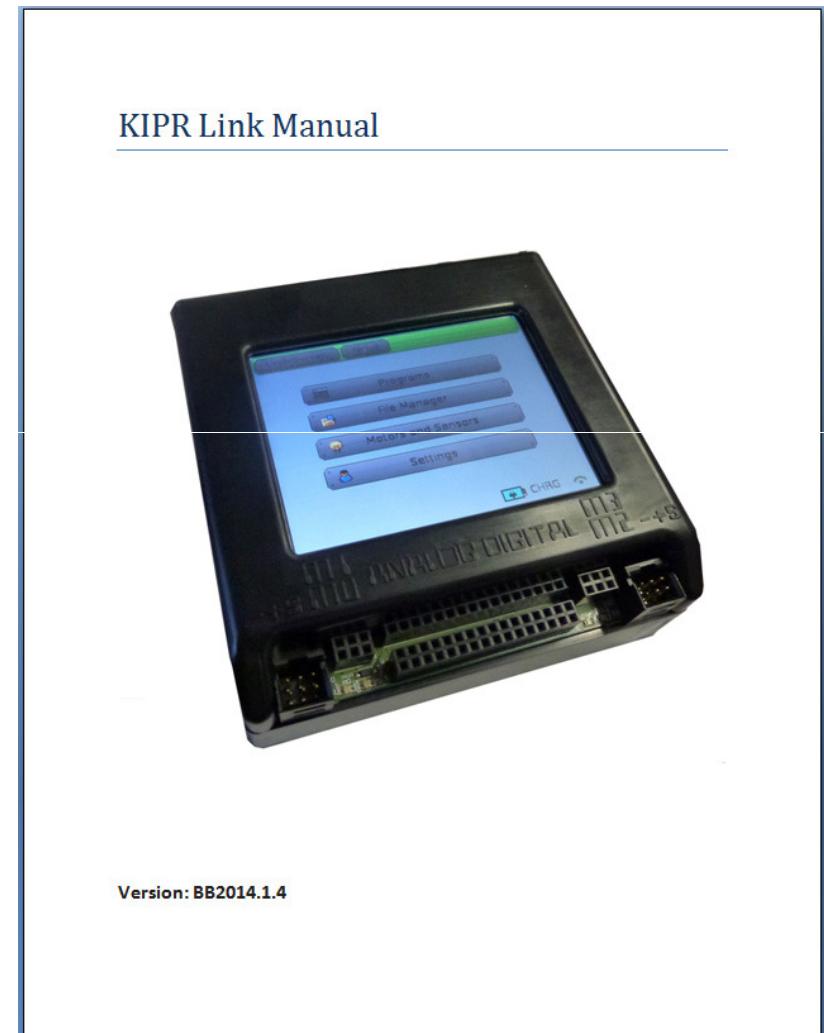
- Make sure your Create is turned on and is sitting on the floor
- Your KIPR Link should be connected to the Create with the Link-Create cable
- Your project should have been downloaded onto your KIPR Link
- Go to your project on the Link through the *Programs* menu and press ***Run***





# FYI: KIPR Link Manual

For further detail about the KIPR Link, programming the Link for Botball, the Vision System, the Graphics API, the Xtion Depth Sensor, and the function libraries (including those for vision, depth, and the Create module), consult the KIPR Link Manual on your Team Home Base.





# Our First Programs Were Written in **C** - Why Did We Start With **C**?

- **C** is a high level programming language developed to support the Unix operating system
  - The KIPR Link controller utilizes a version of Unix called Linux
- **C** is the most widely used language for systems programming
- Botball robots need to be programmed at the systems level to take advantage of the features of the KIPR Link
- For Botball, the KISS Platform provides a user friendly interface for developing Botball programs in **C, C++ (Java and Python – soon)**
- For this workshop we will focus on **C**



# Understanding Your First C Program



# Objectives

## First **C** program

- Examine the simple **C** program used to verify installation of the KISS Platform



# Prep

## First **C** program

- Recap the demo of the “*Hello World!*” program
  - Discussion of what is a **C** program
  - Where to find help with programming
- Detailed explanation of the simple program
- Recap of the step by step process



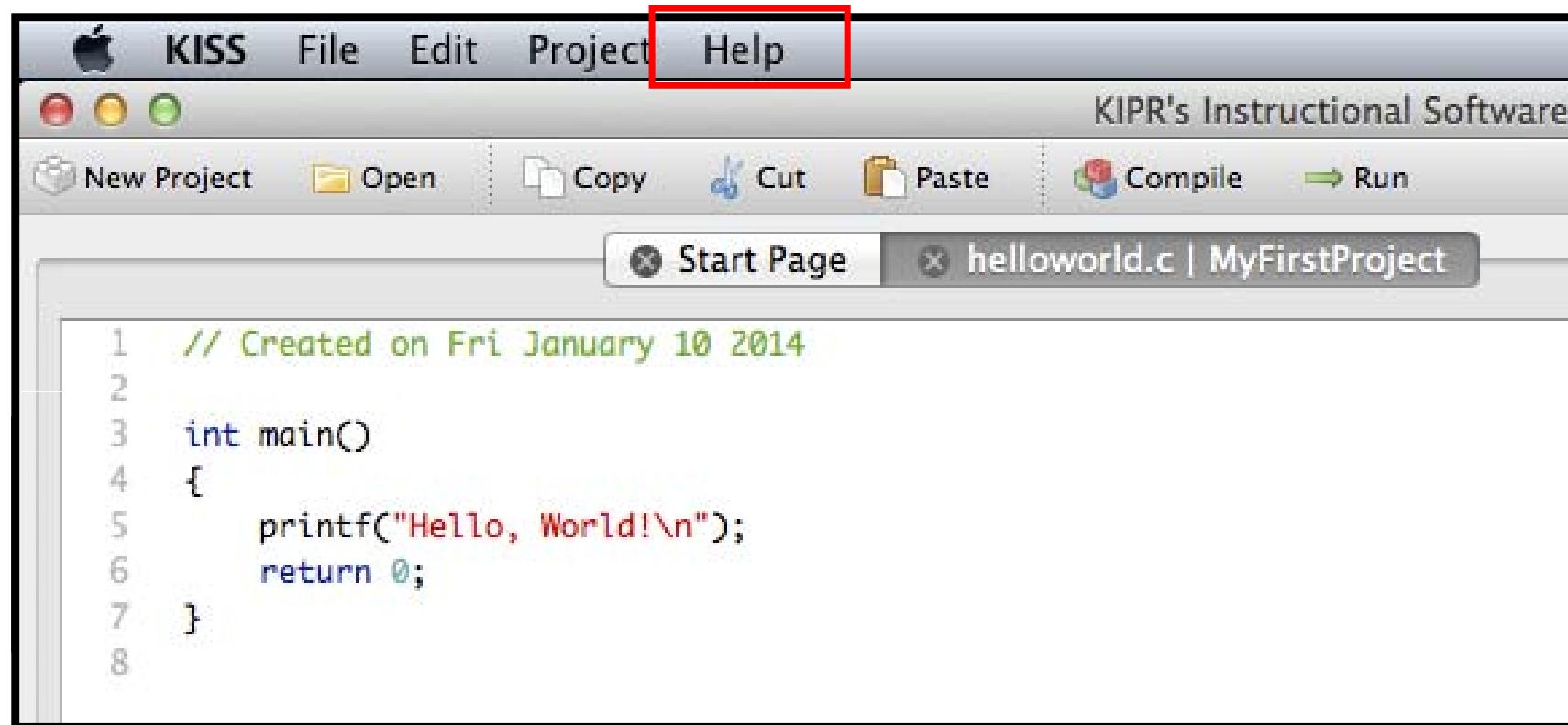
# Project Recap

- For the KISS Platform, programs reside within named projects, which combine in one place various elements necessary for the program to run
- When you click on *Run* for the active project within the KISS IDE, you first select an available *Target* for your project, then the IDE **saves**, **compiles**, and **executes** your program using the facilities provided by the target
  - *Compile* is the process of converting your program from ordinary text into a format the selected target understands
  - *Run* causes the target to compile and execute your program
  - Note that whenever you add a new file to a project, the KISS IDE will ask for a file name. If you ask for a C file, the IDE will extend the name you provide by adding “.c” to the end, unless you have already done so. The name should not have any other “.” in it.

*Good:* MyFile.c    *Bad:* My.File.c



# Where Can I Get Help?





# Select from list!

The screenshot shows the KIPR's Instructional Software System interface. The window title is "KIPR's Instructional Software System". The menu bar includes "New Project", "Open", "Compile", and "Run". The toolbar has buttons for "Start Page", "helloworld.c | MyFirstProject", and "Documentation". The main area displays a list of documentation items:

- KIPR/KISS Documentation**  
*A user manual for C programming and KISS IDE*
- KISS IDE Tutorial: Hello World!**  
*Creating and running a simple Hello World program*
- KISS IDE Tutorial: Existing Files**  
*Creating projects using existing files*
- KISS IDE Tutorial: Multiple Projects**  
*Working with multiple projects at a time*
- KISS IDE Tutorial: Targets and cs2**  
*Overview of targets and the cs2 target*
- KISS IDE Tutorial: Custom Boards**  
*Description and example of the .board file format*
- KISS IDE Tutorial: Dependencies**  
*Creating and using project dependencies*
- libkovan Documentation**  
*Explore the full libkovan standard library*

To the right, there is a sidebar titled "MyFirstProject" which lists "helloworld.c". A table at the bottom provides details about the selected file:

Name	helloworld.c
Path	/Users/wymyers/Documents/Botb



# KISS IDE User Manual for C

- Accessed from the KISS IDE *Help* tab
  - Under *KIPR/KISS Documentation*

**KISS IDE User Manual Index**

1. [Introduction](#)
2. [KISS IDE Interface](#)
3. [Programming in C](#)
4. [File I/O for USB Flash Drive](#)
5. [The KIPR Link Library](#)
6. [KIPR Link Images](#)
7. [Program Examples in this Manual](#)
8. [KIPR Link Simulator](#)

**KISS IDE User Manual for C**

## Introduction

The KISS IDE is an Integrated Development Environment (IDE) for KIPR's Instructional Software System (KISS), providing an editor and compilers for software development in multiple programming languages, with special purpose function libraries and simulation support for the KIPR Link Robot Controller. The KIPR Link is a complete Linux based system with color touch screen for user I/O. Its hardware provides analog and digital I/O ports, DC motor ports, servo ports, USB ports, TTL serial communication, and an HDMI port, plus an integrated accelerometer, IR send/receive, and other program accessible hardware.

This is the on-line manual for the development of C programs using the KISS IDE. The KISS IDE implements the full ANSI C specification and can target the host system, the simulator, or an attached KIPR Link. For information about the C programming language, including history and basic syntax, see



# Software Projects

- The KISS IDE organizes as a project the collection of elements produced to operate your robot
  - Even simple programs may need things like game board layouts in order for them to work effectively
- During the development of a working robot, project elements are added, subtracted, or modified
  - The project structure incorporated into the KISS IDE simplifies managing your robotics project as it evolves
- It is best to stick to one programming language for the program elements of a project, which requires learning how to write programs using that language (in our case **C**)



# Templates and Comments

- For starting a new **C** file for the KIPR Link you should use as a template one of the selections made available by the KISS IDE
- Comments are imbedded in programs to document information about the program
- Two ways to comment **C** programs

// is a comment for rest of line

or

/\* is a comment that goes from  
the initial slash-star till  
the first star-slash \*/



# The Simple C Program

By default, the KISS IDE colors your code and adds line numbers

- Comments appear in **green**
- Key words appear in **bold blue**
- Text strings appear in **red**
- Numerical constants appear in **aqua**
- Compiler directives appear in **blue**

```

1 // Created on Thu January 10 2013
2
3 int main()
4 {
5     printf("Hello, World!\n");
6     return 0;
7 }
```



# Running Your Project

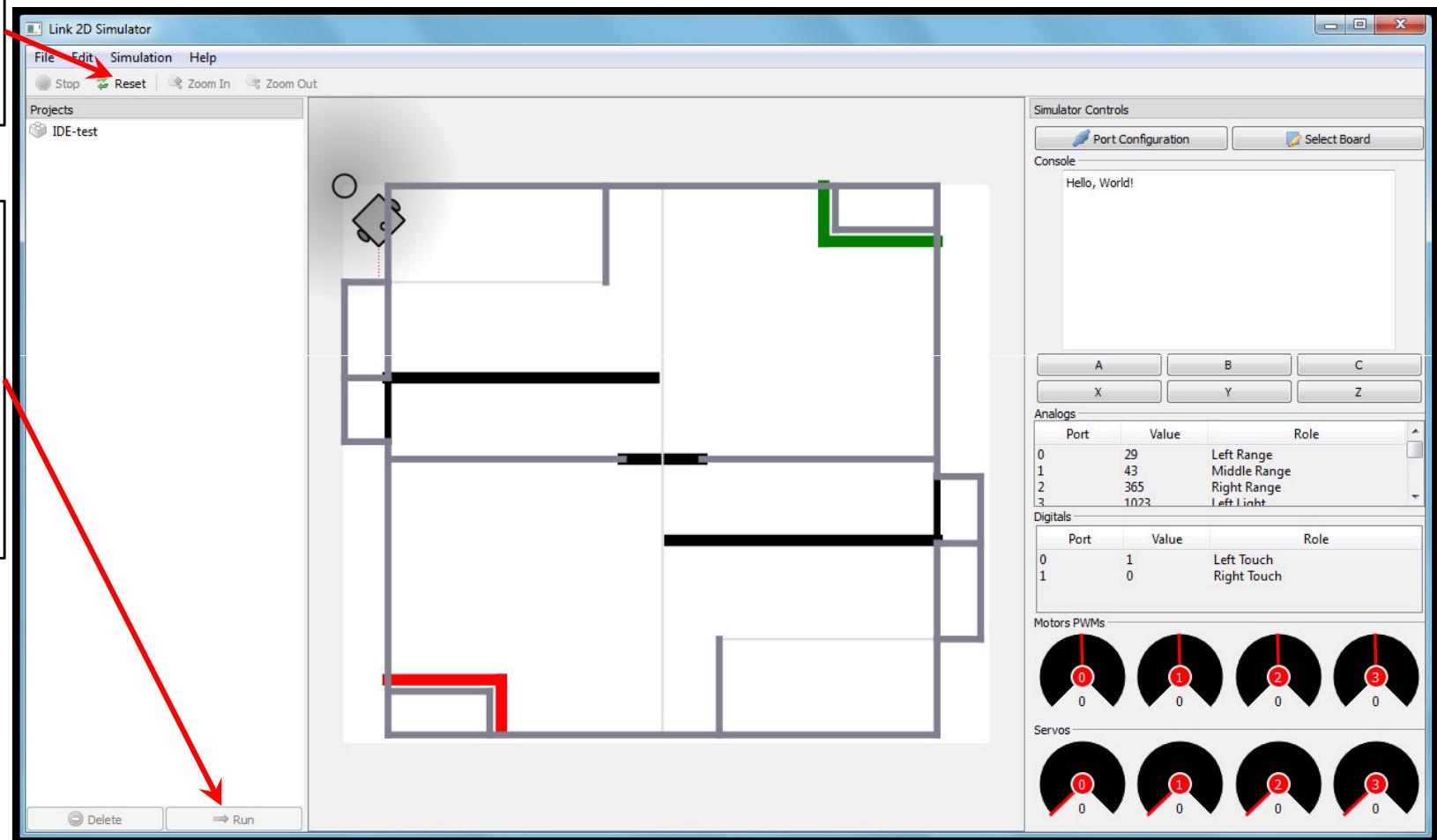
- To use the IDE to see what you program does, you need to run the program's project in the simulator
- If your project is the only project open in the IDE, it is the *active* project
- If there is more than one project open in the IDE you will have to click on which project you want to be the active project
- When you click the IDE's *Run* button, it will attempt to run the active project, launching the simulator if it is the selected target and not already running



# Executing a Project in the Simulator

*Reset* restores the board to its starting state

To run a project without resetting the board highlight its name and press the *Run* button

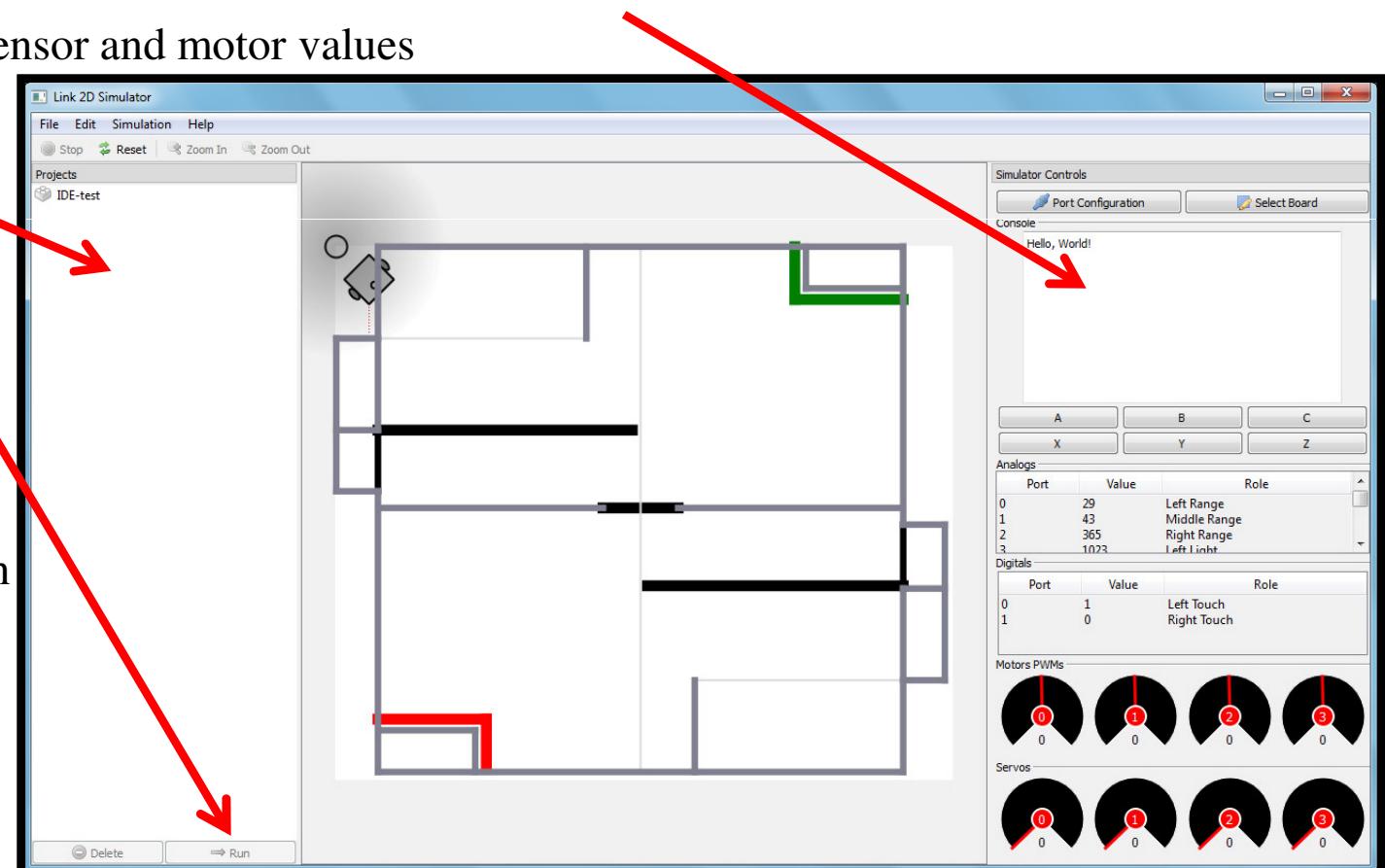




# Verifying the IDE Operation

It should now run in the Simulator

- If not already up, the IDE will bring up the Simulator and direct it to run your program (assuming your program is error free).
- Output will appear in the Simulator's console window.
  - Panels show sensor and motor values
  - Projects that have already been run are listed
  - Any listed project can be highlighted and run again (without resetting the board)





# Using the Simulator to Run a Project

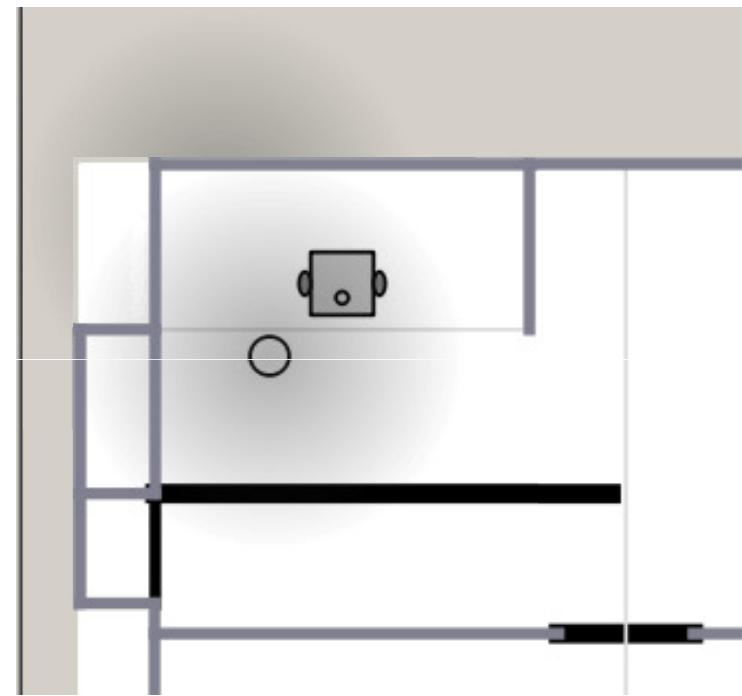
- For projects that have their target set to the simulator:
  - Clicking *Run* in the IDE saves, compiles, loads, and runs your project in the simulator
  - Clicking *Compile* in the IDE saves, compiles , and loads your project in the simulator
    - Allows you to place the robot and the light in the simulator before the program is run
    - Unless you are using a custom board, allows you to change the board the robot runs on to one of the alternatives
    - You can then run your program by highlighting it and clicking *Run*



# Configuring the Simulator

## Starting light and robot

- Positioning and using the light
  - The light is positioned by clicking on the light and dragging it
  - Double clicking on the light toggles it on/off
  
- Positioning the robot
  - The robot is positioned by clicking on the robot and dragging it
  - You can turn the robot by holding down the shift key, clicking on the robot, and dragging to turn it
  - You can then run your project by highlighting it and clicking *Run*

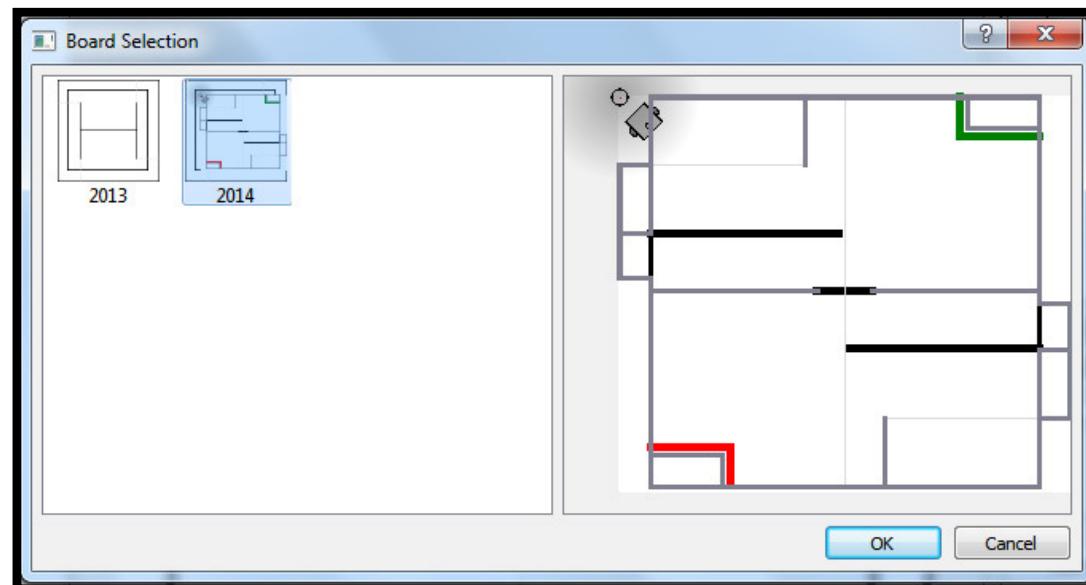




# Configuring the Simulator

## Game board selection

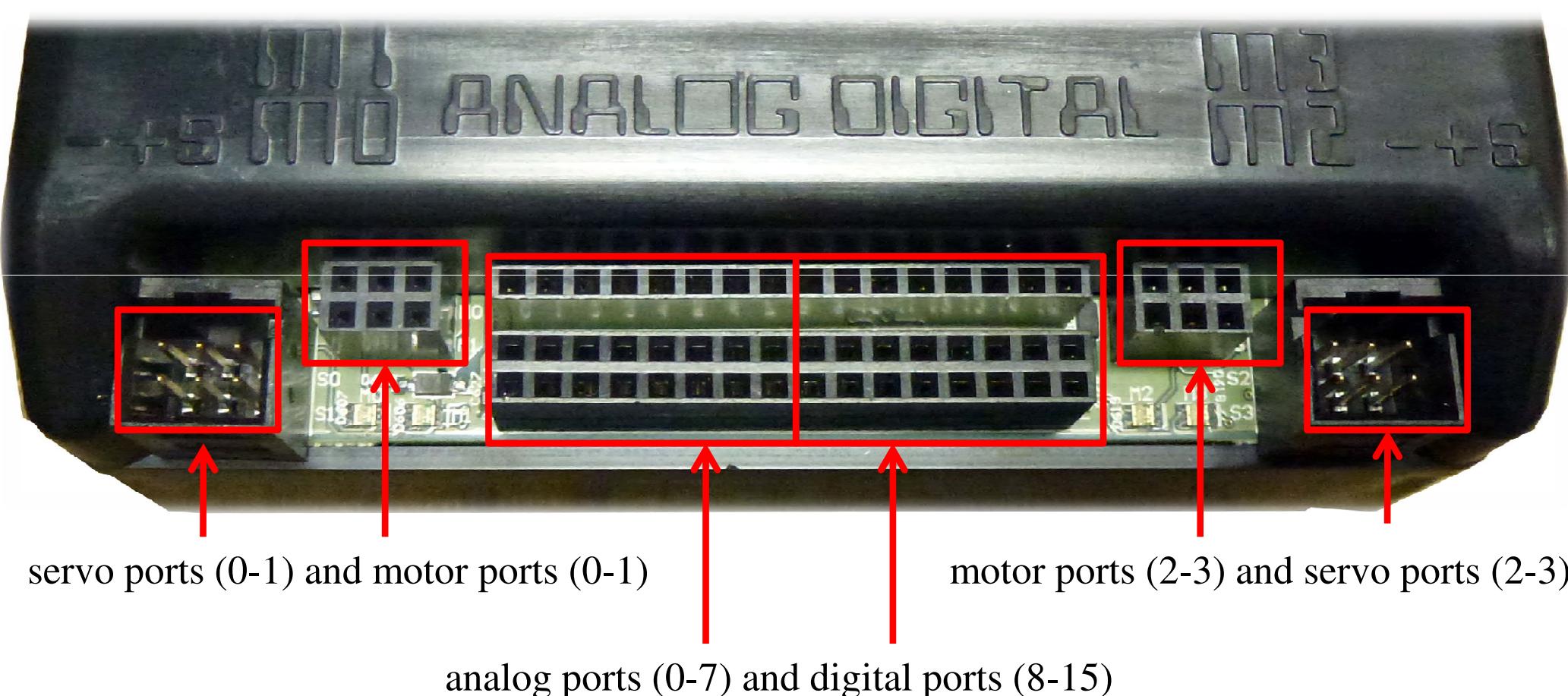
- Selecting a game board
  - You can either provide a custom board for your project or select one of the default boards under the Simulator’s *Select Board* tab
    - See *Help* for information on configuring a custom board
  - If you have a custom board file in your project, the Simulator will not let you change to one of its default boards
  - The *Select Board* tab brings up a window with the currently available default game boards you can select from among





# KIPR Link Sensor & Motor Ports

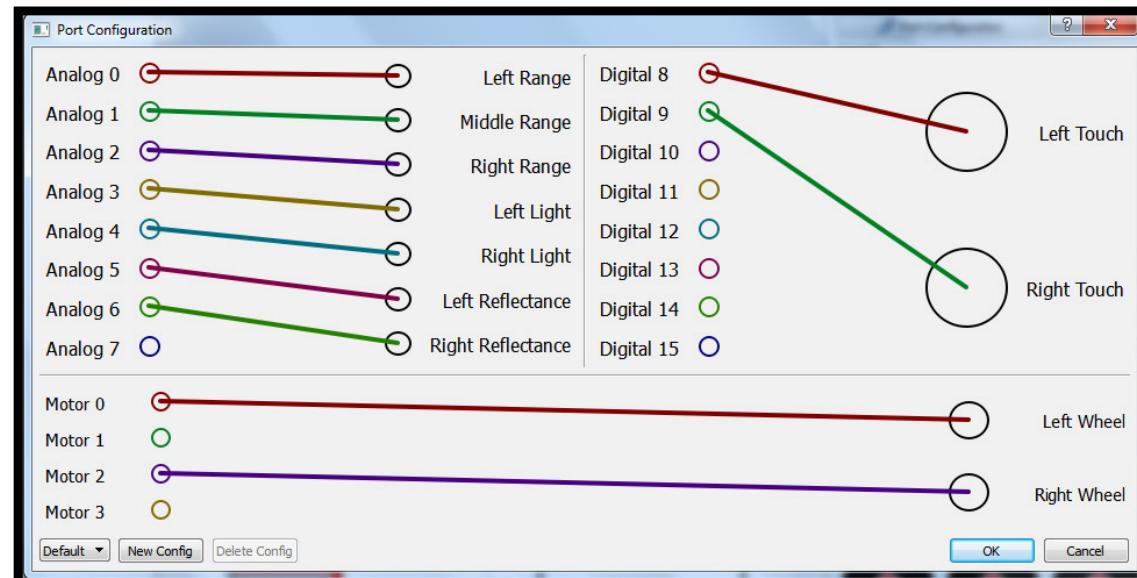
Assignable on the Simulator





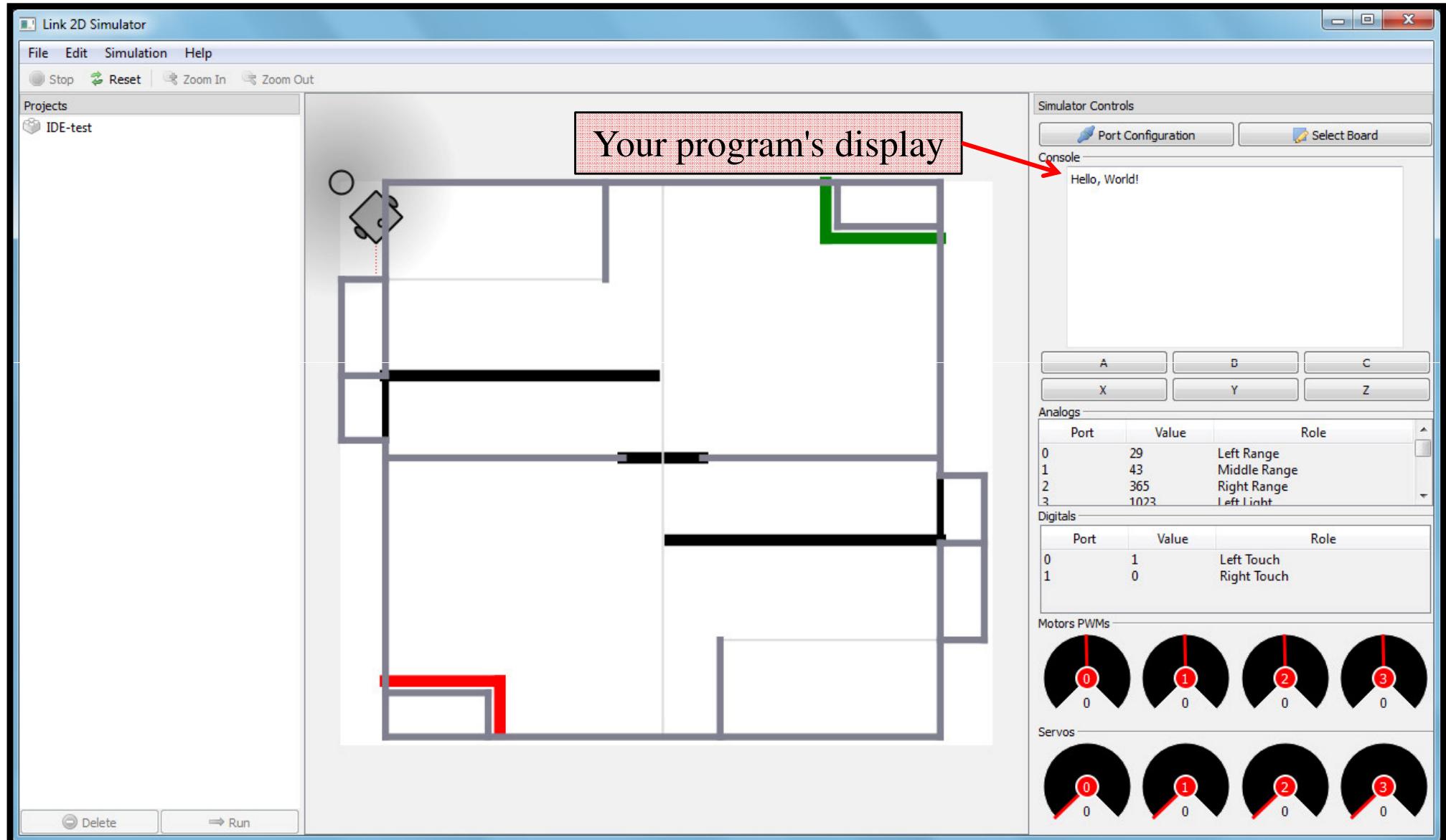
# Configuring the Simulator Ports

- Changing the port configuration
  - The Simulator's Port Configuration tab brings up the sensor and motor port assignments window showing the current assignments
  - To make or change an assignment for the robot, click on the port, then click on the motor or sensor assignment
  - To remove an assignment, click on the port, then click on the connecting line
  - Two ports can have the same sensor or motor assignment





# Observing Results





# The Create Program Explained



# Function Definition

*return type name argument list*

↓      ↓      ↓

**int main ()**

```

// Starts the Create Connection
// This will connect to the Create
// and return connected(0)
// Starts the Create
// Starts the Create at 200 - 200
// Middleape(200) -> false
// Starts the Create at 0
// Starts the Create at 0

```



# Blocks of Code

// Spin the create CW about 1 rev

{ ← The block of code **begins** here

```
private void connectingToTheCreate()
{
    create_connect(0);
}
```

```
//Spin CW
create_drive_direct(200, -200);
msleep(3000); // for 3 secs
```

```
create_stop();
```

```
create_disconnect();
```

```
printf("Program is done!\n");
```

```
main();
```

} ← The block of code **ends** here



Defined by braces



# Function Calls

```

// Spin CW and connect to Cat robot
int main()
{
    printf("connecting to the Create\n");
    create_connect();
    //Spin CW
    create_drive_direct(200,-200);
    msleep(3000); // for 3 secs
    create_stop();
    create_disconnect();
    printf("Program is done!\n");
}

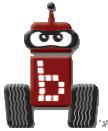
```

HyperProg4!



# Function Calls (2)

- **printf ("connecting to the Create\n");**
  - This prints a message out to the console screen of the Link
- **create\_connect ();**
  - This function starts communications between Create and Link (light goes to orange)
- **create\_drive\_direct (200, -200);**
  - Moves left wheel at 200mm/s and the right wheel at -200mm/s
- **msleep (3000);**
  - Pause program for 3 seconds – motors continue to run
- **create\_stop ();**
  - Stops the drive motors on the Create
- **create\_disconnect ();**
  - Communication connects stopped. Light goes back to green



# Terminating Statements

```

/*
 * Spins the Createbot CW about its center point
 */
int main()
{
    printf("connecting to the Create\n");
    create_connect();
    /* Spin CW
    create_drive_direct(200, -200);
    msleep(3000); /* for 3 seconds
    create_stop();
    create_disconnect();
    printf("Program is done!\n");
    return 0;
}

```

Defined by a semicolon



# Functions Return a Value

File Edit View Insert Tools Window Help

```
printf("connecting to the Create\n");
create connect();
//Spin CW
create drive direct(200, -200);
msleep(3000); // for 3 secs
create stop();
create disconnect();
printf("Program is done!\n");
return 0;
```

Untitled



# msleep()

- Like `printf()`, `msleep()` is a built-in function
- `msleep(3000)` causes the KIPR Link to pause for 3 seconds
  - Example :

```
printf("slow...");  
msleep(3000);  
printf("reader\n");
```



# Step by Step Process

- Start the KISS IDE and click on the *New Project* icon
- Name the project and confirm its save location
- Name and select the *Hello World!* C file
- Use the IDE editor to modify the **C** program to create your own program
- At any time you can save the file by using *File..Save* or *File..Save As*
  - File names can contain: letters, numbers, – and \_
  - Unless you override the file type specification, .c will automatically be appended to the file name
    - Required to compile and run the program
- Check your program by clicking on the *Run* button to run the project
  - This automatically saves your program, then compiles and runs it (if there is no error)
- If there is an error, the KISS IDE will give a message and a line number: column number (fix the first error, then recompile!)
  - **the error will be on OR before that line**



# Example Error Message

Filename      Line Number      Error Description

The screenshot shows a software interface for developing C programs. On the left, a code editor window titled 'test.c' displays the following code:

```
1 // Created on Wed January 8 2014
2
3 int main()
4 {
5     printf("Hello, World!\n")
6     return 0;
7 }
```

On the right, a project explorer window titled 'IDE-test' shows a single file named 'test.c'. Below it, a properties panel shows the name is 'test.c' and the path is 'C:/Test Projects/IDE-test/test.c'. At the bottom, a terminal window titled 'test.c' shows the error message:

```
test.c:
C:/Users/Charles Wintontest.c: In function 'main':
C:/Users/Charles Wintontest.c:6:2: error: expected ';' before 'return'
```

A large red rectangular box highlights the error message in the terminal. Three black arrows point from the labels 'Filename', 'Line Number', and 'Error Description' to the corresponding parts of the error message: 'test.c', '6:2', and 'error: expected ";" before "return"'.

**In English: There is a “;” missing on the previous line (Line 5).**



# Activity 1: Objectives

Programming Basics and Screen Output

Create a project with a program for the KIPR Link that displays "Hello World!" to the screen, delays two seconds, and then displays your name on the screen.

Execute the project on the simulator to test your program.



# Activity 1: Task Design

Programming Basics and Screen Output

Break the objectives down into separate tasks and think about how each might be accomplished; for example, the larger task might be developing a program to operate a robot's claw, which has tasks within for making the claw open or close. Since this is our first example, the tasks are pretty simple:

1. Display "Hello World!" on the screen.
2. Delay for 2 seconds.
3. Display your name on the screen.



# Activity 1: Program

## Programming Basics and Screen Output

Use our previous Task Design as *Pseudocode* to help write the real code...

1. Display "Hello World!" on the screen.
  - Use **printf()** function
2. Delay for 2 seconds.
  - Use **msleep()** function
3. Display your name on the screen.
  - Use **printf()** function

Comment your code (pseudocode makes great comments) - your comments show what you think you told your bot to do, but not necessarily what it will actually do!



# Activity 1: Experiments

## Programming Basics and Screen Output

- Try adding more **printf()** statements to your program (pay close attention to the syntax, particularly the terminating semi-colon needed by each statement)
- Have the program print out a haiku about robotics
- Execute your revised program (*Run* button)
- Experiment by leaving off or adding extra "**\n**" or "**\t**" to the start or end of the strings in your **printf()**
- Try adding the command **display\_clear();**
- Can you print out more lines than can show on the screen at one time?
- Have the program print out a poem or Haiku. What happens when the screen fills up?



# Activity 1: Solution

## Programming Basics and Screen Output

```
***** Activity 1 *****
int main()
{
    // 1. Display "Hello World!" to the screen
    printf("Hello World!\n");

    // 2. Delay for 2 seconds
    msleep(2000);      //2000ms = 2sec

    // 3. Display your name to the screen
    printf("My name is Botguy.\n");

    return 0;
}
```



# Activity 1: Reflections

## Programming Basics and Screen Output

- What have you learned from this activity?
  - e.g., what did you find out from the simulator
  - or what did you figure out in messing with the simulator?
- What does the "`\n`" and "`\t`" do in `printf()`?
- What does `display_clear()` do?
- How many lines can you see on the screen at once?  
What happens to the others?



# Operate a Simbot



# while statement

- Programs normally move from one statement to the next
- Sometimes we have a block of code we wish to repeat until some event takes place
- A **while** statement is used to accomplish this task by checking to see if something is true
  - Tests that check if something is true or false are called Boolean operations
  - More information on "Boolean operators" (such as **==**, **<=**, **>=**, **!=**, **<**, etc.) is in the KISS IDE help
    - **==** (two equals signs together, not one) is used to test if two values are the same



# Boolean Operators

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



# Logical Operators

Boolean	English Question	True Example	False Example
A <b>&amp;&amp;</b> B	Are <b>both</b> A <b>and</b> B true?	true <b>&amp;&amp;</b> true	true <b>&amp;&amp;</b> false false <b>&amp;&amp;</b> true false <b>&amp;&amp;</b> false
! (A <b>&amp;&amp;</b> B)	Is <b>at least one</b> of A <b>and</b> B false?	true <b>&amp;&amp;</b> false false <b>&amp;&amp;</b> true false <b>&amp;&amp;</b> false	true <b>&amp;&amp;</b> true
A <b>  </b> B	Is <b>at least one</b> of A <b>or</b> B true?	true <b>  </b> true false <b>  </b> true true <b>  </b> false	false <b>  </b> false

! negates the **true** or **false** boolean



# while statement example

```

int main()
{
    int i = 0; // counting variable

    while (i < 2) // check bumpers
        msleep(1000); // sleep for 1 second
        i = i + 1; // increment i

    return 0;
}

```

Variable **i** is equal to 2

**i** is not less than 2

Skip the block of code  
and go to the next  
program statement.



# Background

## Robot Simulator – Driving

1. The simulated robot has motors and wheels plugged into motor ports 0 and 2
  - the function **motor (<port>, <power>)** is used to control motors
  - there are 4 possible motor ports: 0, 1, 2, or 3
  - motor power levels can be anywhere from -100 to 100, with 0 being off. The function **ao ()** turns all 4 motors off.
2. The simulator has several sensors including a left bumper in digital port 8 and a right bumper in port 9
  - the function **digital (<port>)** returns 0 if that bumper is not pressed and 1 if it is pressed
3. There are light sensors on analog ports 3 and 4
  - The function **analog (<port>)** returns the value of that sensor 0-1023
  - Light sensors give small numbers for bright lights and large numbers for dark.



# Function Examples

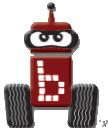
- `motor(0,100);`
  - Turns on motor 0 at 100% power
- `digital(8); // left bumper on port 8`
  - returns 0 if the left bumper is not pressed and 1 if it is pressed
- `analog(3); // light sensor on port 3`
  - returns the value of the left light sensor, Light sensors give low values for bright lights and high values for dark



# Pseudocode:

## Drive Till Bump

- Announce what the program does
- While the left AND right bumpers are not pressed
  - Turn on the left motor
  - Turn on the right motor
- When a bump occurs turn off the motors
- Announce that the program is done



# Program

## Simulator – Driving forward

```

 ****
Drive the simulated robot forward at half power till it bumps
****

int main()
{
    printf("Drive Straight till bump\n"); // announce the program
    msleep(1000); // wait 1 second

    while(digital(9)==0 && digital(8)==0) { // check bumpers
        motor(0,55); // Drive left wheel forward at 55% power
        motor(2,50); // Drive right motor forward at half power
    }

    ao(); // Stop motors
    printf("All Done\n"); // Tell user program has finished
    return 0;
}

```



# Activity 2: Objectives

## Programming Basics and Screen Output

Create a project with a program for the KIPR Link that waits for a light to turn on, then drives until the robot runs into a wall.  
You can go straight or in a curve

Compile the project to load it in the simulator.

- Drag the robot to the desired starting position, rotated if appropriate
- Drag the light near the robot, then *Run* the project
- Double click the light to turn it on



# Activity 2: Task Design

Programming Basics and Screen Output

1. Print the objectives
2. Start when the light comes on
3. Drive forward
4. Stop when bumper pressed
5. Print that the program is over



# Activity 2: Program

## Programming Basics and Screen Output

1. Print the objectives.
  - Use `printf()` function
2. Start when the light comes on .
  - Use `while` to check `analog(3) > 500`
3. Drive straight.
  - Use `motor()` functions like previous example
4. Stop when bumper pressed.
  - Use `while` to check `digital(8)` and `digital(9)`
  - Use the `ao()` functions like previous example
5. Print that the program is over.
  - Use `printf()` function

Comment your code (pseudocode makes great comments) - your comments show what you think you told your robot to do, but not necessarily what it will actually do!



# Activity 2: Experiments

## Programming Basics and Screen Output

- Try changing the motor speeds
- Try different values for the light sensor
- Try moving the light source farther away from your robot in the simulator
- What happens if you drive your robot backwards?
- What if you use:

`while(digital(9) == 0 || digital(8) == 0)`

instead of `while (digital(9) == 0 && digital(8) == 0)`



# Program

## Simulator – Start with Light and Driving Straight

```
*****
After the Light comes on drive the simulated robot forward at half power till
it bumps
*****
int main()
{
    printf("Turn on light to Drive Straight until bump\n"); // announce

    while (analog(3) > 500) { //while light value is above 500 do nothing
        //go on when value is below or equal to 500

        while (digital(9) == 0 && digital(8) == 0) { // check bumpers
            motor(0,55); // Drive left wheel forward at 55% power
            motor(2,50); // Drive right motor forward at half power
        }

        ao(); // Stop motors
        printf("All Done\n"); // Tell user program has finished
        return 0;
}
```

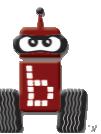
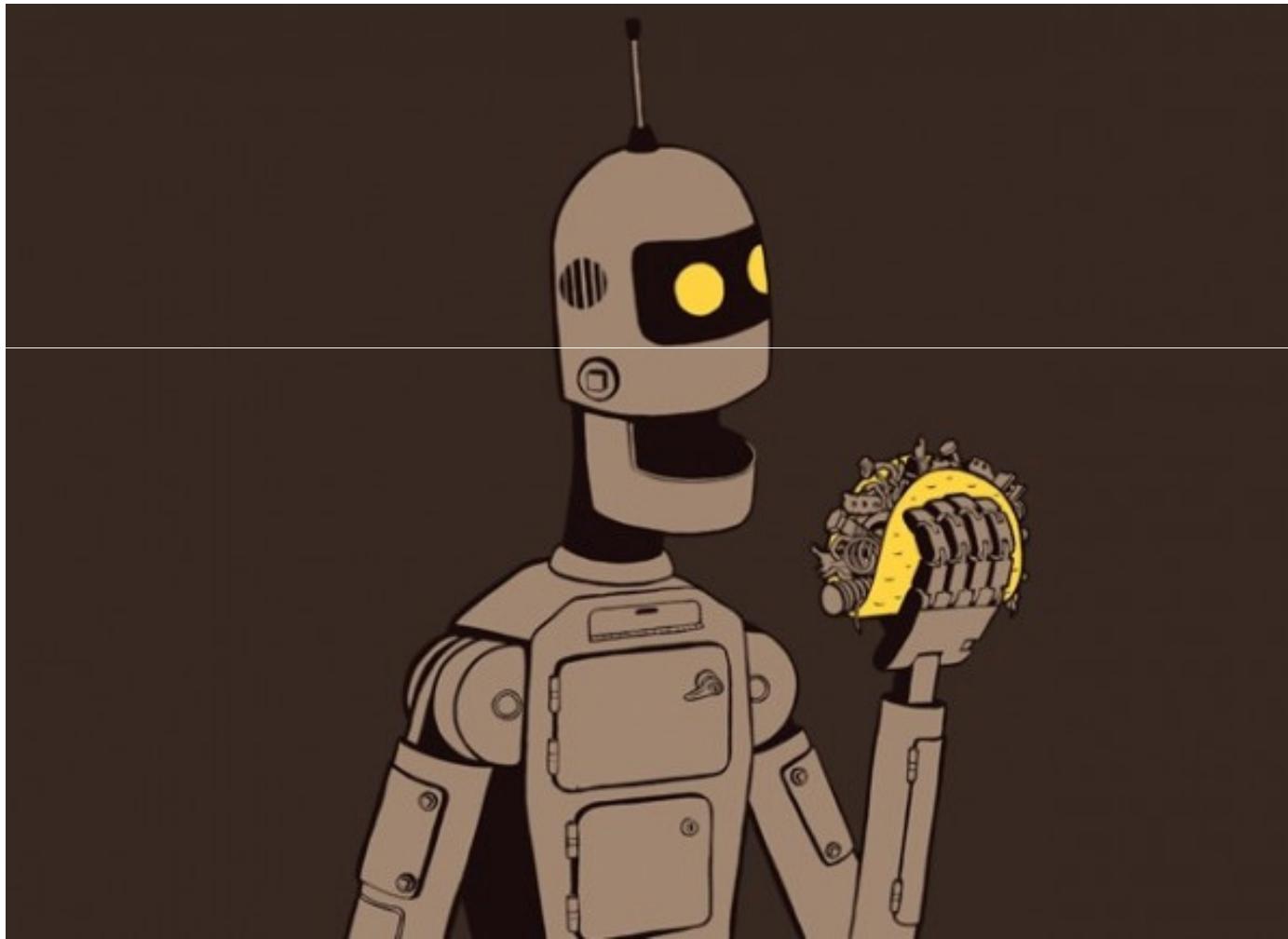


# Lunch Activity: Finish DemoBot

Use the DemoBot Building Guide



# Lunch!





# Moving the Create

- commands for moving
- commands for using Create sensors
- combining commands to do something interesting





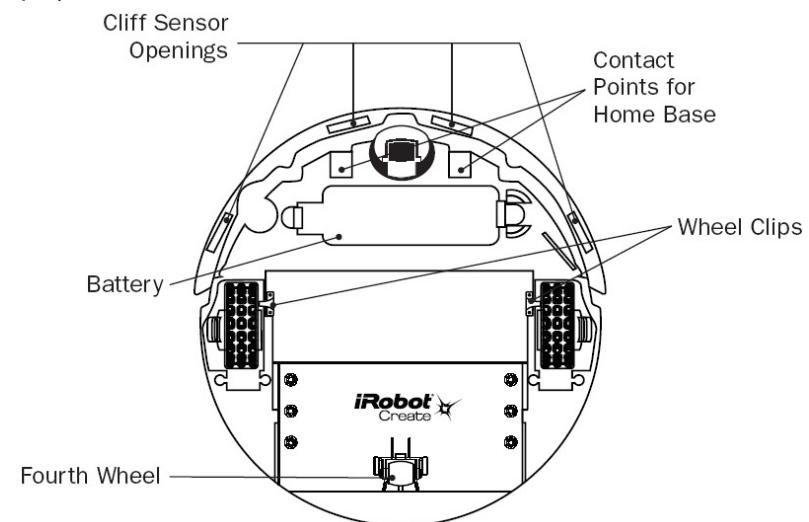
# **create\_connect () & create\_disconnect ()**

- The KIPR Link has a serial interface, which by default is set for downloading programs from the KISS IDE to your KIPR Link via a USB cable connection
  - Computer access to the Create module is also by serial interface
  - There is a cable in your kit to provide a connection between KIPR Link and Create
- The library function **create\_connect ()** ; sets the KIPR Link serial interface for the Create cable connection rather than the USB cable connection
  - The Create has to be turned on for this to work
  - Once connected, your KIPR Link can send commands to operate the Create
- The library function **create\_disconnect ()** ; sets the KIPR Link serial interface back to the USB cable connection and shuts off Create motors
  - Power cycling your KIPR Link also sets the serial interface to be for the USB cable connection (but it won't shut off Create motors!)



# Create Motor Functions

- **`create_drive_direct (<left_speed>, <right_speed>)`** ; specifies separate right and left speeds for the two drive motors – the command continues until a different motor command is received
  - Speed can range between -500 and 500 mm/sec
  - **`create_drive_direct (100, 100)`** ; moves the Create forward at a modest speed
  - **`create_drive_direct (100, 200)`** ; moves the Create counterclockwise at a modest speed
- **`create_stop ()`** ; stops the drive motors
- There are more drive commands; see the Help Manual





# Initializing Create Distance and Angle Calculations

- As the Create operates, the angle turned through and distance traveled are accumulated
- The functions `set_create_distance (<val>)` and `set_create_total_angle (<val>)` reset the distance accumulation to start from `<val>`
  - `set_create_distance (0);` initializes the distance accumulation to start at 0
  - `set_create_total_angle (0);` initializes the angle accumulation to start at 0

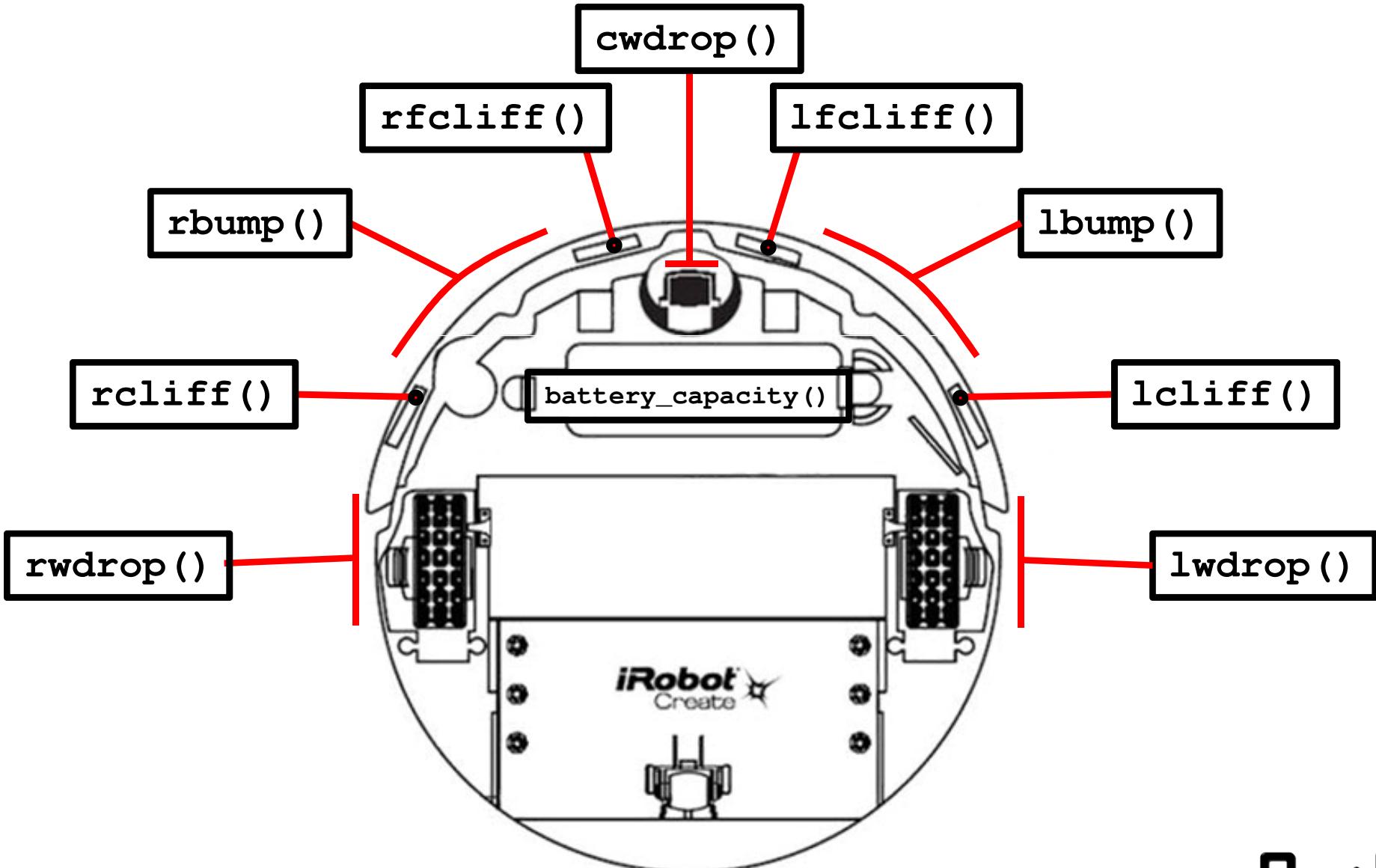


# Create Status Monitoring

- Internal Create sensor data can be accessed by the KIPR Link using "**get\_create**" library functions
  - **get\_create\_distance ()** returns the distance (in mm) that the center of the Create has traveled
  - **get\_create\_total\_angle ()** keeps track of the number of degrees the Create has turned through
  - **get\_create\_lbump ()** returns the value of the left bump sensor (pressed = 1, not pressed = 0)
- Other Create sensors have functions as well...



# get\_create\_...





# External vs. Internal Sensors

- Internal sensors measure things going on inside the robot and are used to infer how the robot is moving
  - e.g., `get_create_distance` does not measure distance but instead measures how far the wheels have turned and assumes that robot moves accordingly
  - This is called dead reckoning
- External sensing like bump and light sensors measure how the environment interacts with the robot
  - If there is nothing between your robot and the wall, then when the bumper is pressed, you have probably reached the wall
- Combining the two types increases reliability
  - e.g., there is a wall 3m in front of you, if your program moves your forward and stops when the bumper is pressed, and if your distance sensor says you have moved more than 2 and less than 4m, then you are probably at the wall you intended.



# Programming for Humans

- Programs should always announce their intentions:
  - Use `printf` and `msleep` to have the program print to the screen what the program does
- Whenever operator input is needed – make sure that the program prints out a prompt to the user
- Programs should announce when they are done



# Demo

## Driving Straight

Drive the Create forward until one of the  
 bumpers is pressed



# Demo Code

```
int main()
{
    printf("Trying to connect to Create...\n");
    create_connect(); // Program stops till it connects...
    printf("Connected. Drive straight until bumper hits\n");

    while(get_create_lbump() == 0 && get_create_rbump() == 0) {
        create_drive_direct(150, 150); // drive straight at 150mm/s
    }

    create_disconnect(); // stops communication

    printf("All Done\n"); // tell user program has finished
    return 0;
}
```



# Conditions and Functions



# Activity 4: Prep

## Conditions and Functions

- Buttons
- Functions
- **if-else** statements (and **else if**)



# Buttons

- On the KIPR Link there is 1 physical button (named *side*) and 6 soft buttons (named *a,b,c,x,y,z*)
  - All have `name_button()` functions which return 1 if the button is being pressed and 0 otherwise
  - All have `name_button_clicked()` functions which pause if the button is being pressed and then returns 1 when it is released or returns 0 otherwise
  - Soft buttons can have their display changed by using  
`set_name_button_text ("display text");`
  - By default only a, b and c are displayed. The 3 extra buttons can be shown using:  
`extra_buttons_show();`  
`extra_buttons_hide();`



# Demo

buttons and **if-else** statements (with **else if**)

```
int main()
{
    set_a_button_text("1 Beep");
    set_b_button_text("2 Beeps");
    printf("press a & b buttons for beeps, c button to stop\n");
    while(c_button() == 0) {
        if (a_button() == 1) { // hold button for continuous beeps
            printf("beep\n");
            beep(); // beep flashes the screen
            msleep(500);
        }
        else if (b_button_clicked() == 1) { // must release button
            // must press & release button before beeps happen
            printf("beep-beep\n");
            beep();
            msleep(300);
            beep();
            msleep(300);
        }
    }
    printf("All Done\n"); // Tell user program has finished
    return 0;
}
```



# What is a Function?

- Remember your math functions?
  - A typical function
    - Circumference of a circle is a function of the radius
      - The "circumference function" for a circle is the Greek circle constant  $\pi$  times twice the radius, or  $C(r) = 2\pi r$
    - In general we use the notation  $f(x)$  to represent a function where  $f$  is the name of the function and  $x$  is its argument
      - Functions can have more than one argument, e.g.,  $f(x,y)$
    - Functions are "deterministic", meaning that if you supply values for the arguments, the function produces a unique result
      - $C(50) = 100\pi$  which is approximately 314.159



# Functions in C

- A C program is comprised of 1 or more C functions, one and only one of which must be named **main**
- C functions follow the same rules as math functions, except a C function doesn't have to return a value and it doesn't have to have any arguments
- Since variables in C have differing types, you have to specify the data type for each of your function's arguments, and the type of data returned by the function (which can be **void** if nothing is being returned)



# Function Prototype

- C expects you to specify a prototype for any functions before they are ever used.
- A prototype for the **drive\_direct** function would appear as:

*function name*                   *argument 1 name*                   *argument 2 name*

The diagram shows a C function prototype: `int drive_direct(int left_speed, int right_speed);`. Three red arrows point from labels above to specific parts of the code: one arrow points down to the `int` keyword in `int drive_direct`, another points down to the first `int` in `left_speed`, and a third points down to the second `int` in `right_speed`. Below the code, three red arrows point up from labels below to the same positions: one arrow points up to the first `int` in `drive_direct`, another points up to the `int` in `left_speed`, and a third points up to the `int` in `right_speed`.

```
int drive_direct(int left_speed, int right_speed);
```

*data type returned*   *data type of argument 1*   *data type of argument 2*

- In the KIPR Link help Manual the documentation for each KIPR Link library function gives the function's prototype.



# Using the Drive Direct Function

```
//function prototype
int drive_direct(int left_speed, int right_speed);

int main()
{
    drive_direct(100, 50); //turn right
    sleep(1);
    drive_direct(0, 0); //stop robot

    return 0;
}

// set both motors to a speed
int drive_direct(int left_speed, int right_speed)
{
    motor(0, left_speed);
    motor(1, right_speed);
    return 0;
}
```



# Demo

## Previous Demo Converted to Use Functions

```

void beep_once();
void beep_twice();
int main()
{
    set_a_button_text("1 Beep");
    set_b_button_text("2 Beeps");
    printf("press a & b buttons for beeps, c button to stop\n");
    while (c_button() == 0) {
        if (a_button() == 1) {
            beep_once();
        }
        else if (b_button_clicked() == 1) {
            beep_twice();
        }
    }
    printf("All Done\n"); // Tell user program has finished
    return 0;
}
void beep_once()
{
    printf("beep\n");
    beep(); msleep(500);
}
void beep_twice()
{
    printf("beep-beep\n");
    beep(); msleep(300);
    beep(); msleep(300);
}

```

**Previous Demo Code**

```

int main()
set_a_button_text("1 Beep");
set_b_button_text("2 Beeps");
printf("press a & b buttons for beeps, c button to stop\n");
while (c_button() == 0) {
    if (a_button() == 1) { // can hold for continuous beeps
        printf("beep\n");
        beep(); msleep(500); // beep flashes the screen
    }
    else if (b_button_clicked() == 1) { // must release button
        // must press & release button before beeps happen
        printf("beep-beep\n");
        beep(); msleep(300);
        beep(); msleep(300);
    }
}
printf("All Done\n"); // Tell user program has finished
return 0;
}

```



# Variables

- Just like arguments for functions, symbolic names can be used to retain data such as the current value of the distance traveled

`int distance;` specifies a "variable" that can hold an integer

- For the variable `distance` a program might use

`distance = get_create_distance();`

to store the value returned by `get_create_distance`

- Variable names in **C** are made up of contiguous letters (upper and lower case), the "\_" character, and digits
  - Variable names cannot begin with a digit
  - Notice the practice of using "\_" as a substitute for a space
- Variable types include `int`, `double`, and many others



# Combining Time & Sensing

- If your robot uses **msleep()** to drive for a specified time, it is literally "sleep moving" and will not be monitoring bumpers, buttons or other sensors
- The function **seconds()** returns a value of type double that represents the Link's internal clock.
- By getting the difference between the current value of **seconds()** and one stored from an earlier time, you can get the elapsed time and use that in your loop conditional.



# Time & Sensing Example

Here is a program that moves the Create for 5 seconds or until a bumper is pressed, whichever happens first

```
int main()
{
    double start; // will be used to hold the starting time
    create_connect();
    start=seconds(); // save the start time
    // check the time & bumps
    while(((seconds()-start)<5.0) && (get_create_lbump()==0)
          && (get_create_rbump()==0)) {
        create_drive_direct(100, 100);
    } // exit loop when time is up or bumpers are bumped
    create_stop();
    create_disconnect();
    return 0;
}
```



# Activity 4 Objectives

## Conditions and Functions

Write a program that behaves differently, dependent upon which software button is pressed. The code should be written using functions that you create for each of the different behaviors.



# Activity 4 Pseudocode

## Conditions and Functions

### **main ()**

1. Rename the A and C buttons to say Left and Right
2. Connect to the Create
3. Loop while the side button is not pressed.
  - If the ‘A’ software button is pressed, then call the turn left function.
  - Otherwise, if the ‘C’ software button is pressed, then call the turn right function
4. Disconnect from the Create



# Activity 4 Pseudocode

## Conditions and Functions

**turn\_left (double secs)**

1. Move the Create left wheel backward and the right wheel forward.
2. Delay the program for an amount of time equal to the parameter value.
3. Stop the Create.

**turn\_right (double secs)**

*Pseudocode for this function is left as an exercise*

**Solution is on next two slides; try before you look!**



# Activity 4 Solution

## Conditions and Functions

```

***** If A turn left, if C turn right (mirror behavior) *****
int turn_left(double seconds); // prototype for turn_left
void turn_right(double seconds); // prototype for turn_right
int main()
{ // 1. Rename buttons
    set_a_button_text("Left"); set_c_button_text("Right");
    printf("Side button to stop\n"); // announce
    create_connect(); // 2. Connect to the Create
    // 3a. Loop until the side button is pressed.
    while (side_button() == 0) {
        // 3b. If the 'A' button is pressed, then turn left
        if (a_button() == 1) {
            printf("turned left %i degrees\n", turn_left(1.0));
        }
        // 3c. Else if the 'C' button is pressed, then turn right
        else if (c_button() == 1) {
            turn_right(1.0);
        }
    }
    create_disconnect(); // 4. Disconnect from the Create
    printf("All Done\n"); // Tell user program has finished
    return 0;
}

```

*(cont'd next slide)*



# Activity 4 Solution, Cont'd

## Conditions and Functions

```
/*Function definitions go below*/
int turn_left(double seconds)
{
    int initial_angle = get_create_total_angle(0);
    // Move left wheel backward and right wheel forward
    create_drive_direct(-300, 300);
    // Delay for time equal to the parameter value
    msleep(seconds*1000);
    // Stop the Create
    create_stop();
    // return the angle turned left
    return get_create_total_angle(0) - initial_angle;
}
void turn_right(double seconds)
{
    // Move the Create left wheel forward
    // and the right wheel backward
    create_drive_direct(300, -300);
    // Delay for time equal to the parameter value
    msleep(seconds*1000);
    // Stop the Create
    create_stop();
}
```



# Activity 4 Experiments

## Conditions and Functions

- Modify solution program so that the number of degrees during a right turn is printed
- Download your code and run your program on the KIPR Link connected to the Create



# Activity 4 Reflections

## Conditions and Functions

- Does every **if** statement have to be succeeded by an **else** statement?
- Can you have **if** statements within other **if** statements?
- How many **else if** statements can follow an **if** statement?
- What is the difference between having an **else** and an **else if** statement at the end?
- Why is it useful to use functions?

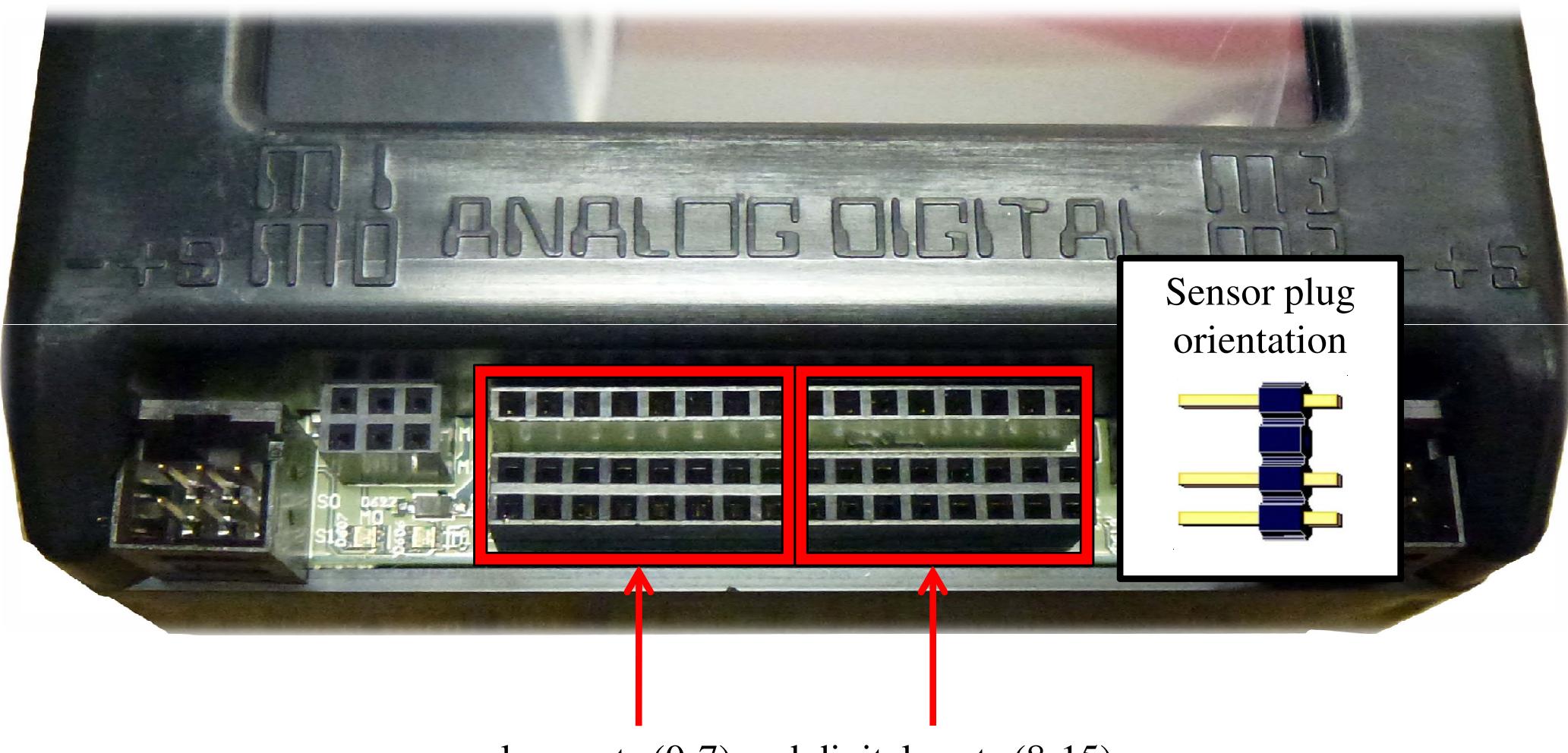


# Starting/Shutting Down the Robot Using Sensors



# KIPR Link Sensor Ports

## Analog and Digital

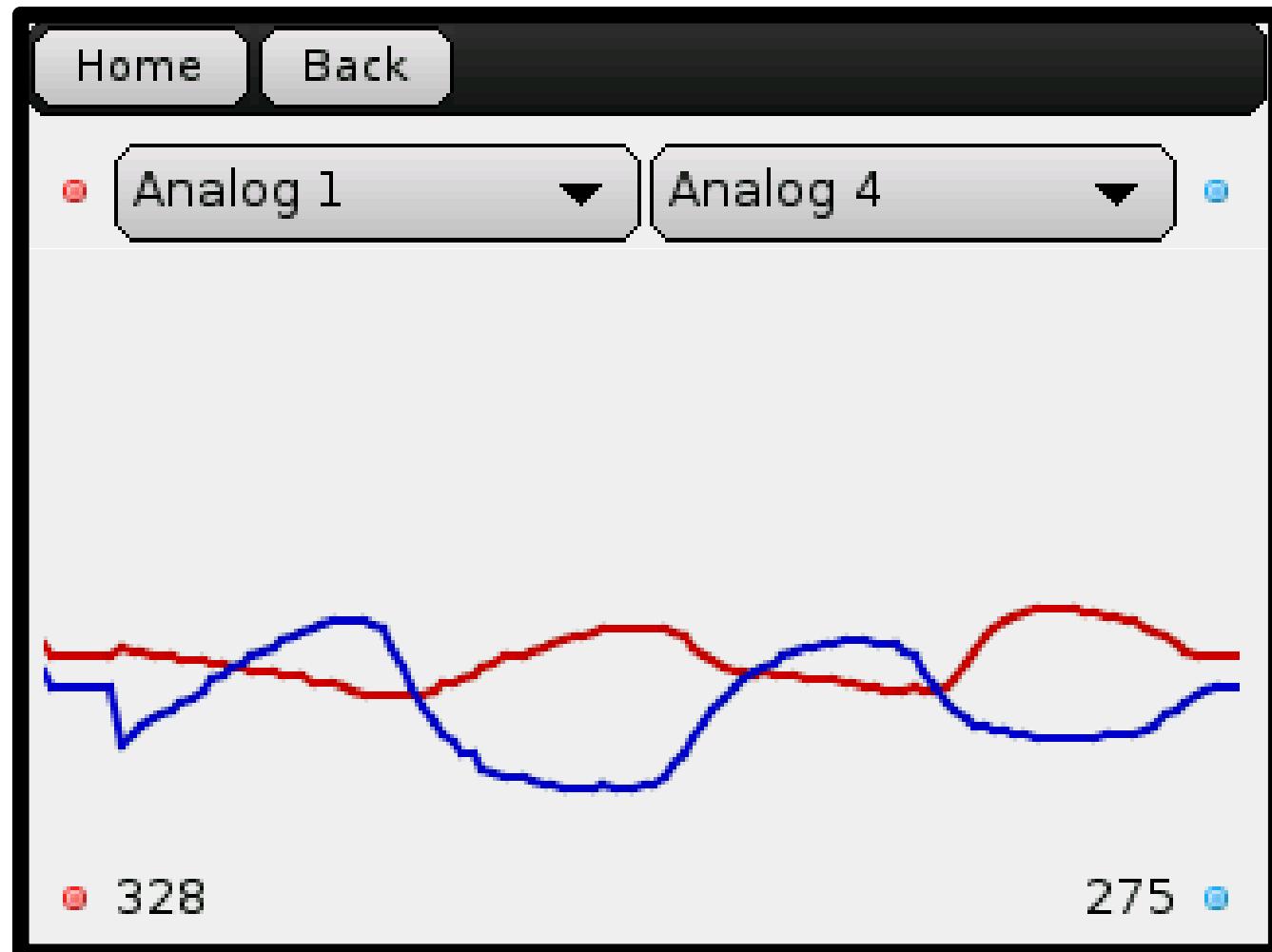


analog ports (0-7) and digital ports (8-15)



# KIPR Link *Sensor Scope* Screen

- Go to the *Sensor Scope* screen
  - Under the *Motors and Sensors* tab on the opening screen, then under *Sensors*





# KIPR Link *Sensor Scope* Screen

- Plug the two sensors to be used for this activity into analog ports on the KIPR Link.

- Plug the **light sensor** into any analog port (ports 0-7)



- Plug the **reflectance sensor** into any other available analog port



- When you point the reflectance sensor towards the IR light sensor you should see a low value for its port.
- If you aim the reflectance sensor at the table and move it across the table edge its value will change.



# Infrared Interlude

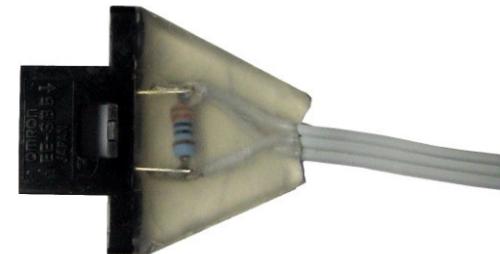
- Plug the USB camera into your KIPR Link and navigate to the Camera Page
- Point the camera at an infrared emitting sensor
- Most cameras are sensitive to infrared light
- You should see a lighted spot where the sensor's emitter is located
- Your light sensors can detect the emissions from the reflectance sensor emitter



# Sensors for Activity

Using the KIPR Link and sensors

- IR light sensor
  - Analog sensor
  - Plug into any port 0-7
- Reflectance sensors
  - Analog sensor
  - Plug into any port 0-7
  - Has an IR emitter and an IR detector
  - Light source for this activity





# analog()



- For an analog sensor such as a light or reflectance sensor plugged into analog port 2, **analog(2)** will provide the current value of the sensor
  - An analog sensor provides a range of values
  - The **analog** function gives values from 0-1023



# digital ()



- For a digital sensor such as a button sensor plugged into digital port 8, **digital(8)**; will provide the current value of the sensor
  - A digital sensor's value is 0 if off and  $\neq 0$  if on



# Sensor and Motor Manual

For further detail about sensors, consult the Sensor and Motor Manual on your workshop Team Home Base





# Shielding Light Sensors



# A Botball Robot Should Have a Shielded Starting Light Sensor

- The table will be brightly lit
- Overhead lights from the game table will flood an unshielded starting light sensor rendering it incapable of discriminating ambient light from the starting light
  - Generally, to be effective IR light sensors should be shielded from all extraneous sources by a light tube
- Opaque objects stop IR light (e.g., foil, black electrical tape)
- Soda straws are not opaque; printer paper is not opaque; two layers of printer paper are not opaque; a straw wrapped in printer paper is not opaque



# How to Shield a Light Sensor



**No!!**

Paper is NOT  
adequate shielding



**Yes!**

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

Don't forget to leave the end open!



# **wait\_for\_light ()**

- Botball tournament programs should start with the **wait\_for\_light ()** library function
- The **wait\_for\_light ()** function needs an argument (also called a parameter) which should be an analog port number
  - e.g., **wait\_for\_light (3)**;
  - The KIPR Link has 8 analog ports (0-7)
- The **wait\_for\_light** function checks the value of the IR light sensor plugged into the port
- A low value indicates more IR (light on) is being detected, a high value less IR (light off)



# Botball Robots Start ...

- Botball robots have to start by themselves when the game table starting lights go on
- This requires determining the level of light at the table when lights are off and the level when the lights are on
- The **`wait_for_light`** function steps you through this calibration and then pauses until lights are on
- You **must** use **`wait_for_light`**, or a version of it of your own creation, for Botball



# Timing for Botball

## **shut\_down\_in**

- When executed, the function

```
shut_down_in(<game_secs>);
```

starts a process that turns off all motors after *game\_secs* has elapsed and keeps any new commands from being processed

- The **shut\_down\_in** function issues a **create\_stop** command, but if your KIPR Link loses its serial connection to the Create (probably the result of a loose cable or an error in your program code), your Create won't receive the **create\_stop** (and so won't stop in time, in which case you will lose the round!)



# Activity 5 Objectives

Starting / Shutting Down the Robot Using Sensors

Write a program that monitors a light sensor and automatically moves the robot once light is detected.

- The robot should automatically turn off after a predetermined amount of time.

Run the program on the KIPR Link using the Create platform.



# Demo

## wait\_for\_light

```
*****  
wait_for_light demo  
*****  
  
int main()  
{  
    wait_for_light(0); // light sensor in analog port 0  
    printf("I have seen the light!\nAll done\n");  
    return 0;  
}
```



# Demo

## shut\_down\_in

```
*****  
shut_down_in demo  
*****  
int main()  
{  
    printf("Program stops in 3 sec\n");  
    shut_down_in(3.0);  
    while (side_button()==0) {  
        beep();  
        msleep(200);  
    }  
    printf("All done\n"); // shuts down before this!  
    return 0;  
}
```



# Activity 5 Pseudocode

Starting / Shutting Down the Robot Using Sensors

1. Monitor light sensor.
2. Move robot when light detected.
3. Have robot automatically shutdown after a certain amount of time.



# Activity 5

## Starting / Shutting Down the Robot Using Sensors

- Implement a program that follows the pseudocode
- The reflectance sensor contains an emitter and can be used as the light source for simulating the start light. Any time the start light should be on, shine the reflectance sensor at the light sensor (if there is a bright light available, the reflectance sensor is not needed)
- A solution is on the next slide if you need it



# Activity 5 Solution

## Starting / Shutting Down the Robot Using Sensors

```

 ****
Starting/shutting down the robot using the sensors
****

int main()
{
    printf("Activity 6\n");
    // 1. Connect to the Create
    while (create_connect()) {
    }
    // 2. Wait for the light sensor on analog port 0 to turn on
    wait_for_light(0);
    // 3. Shut down in 5 seconds
    shut_down_in(5.0);
    // 4. Drive each of the Create motors at 100 mm/sec
    while (side_button() == 0) {
        create_drive_direct(100, 100);
    }
    printf("All done\n"); // shut down before getting here!
    return 0;
}

```



# Activity 5: Experiments

Starting / Shutting Down the Robot Using Sensors

- Increase the amount of time that the robot is active before it shuts down, and make the robot go straight and then turn during this time.
- Add a reflectance sensor that points downward. Have the robot stop when this sensor detects a black line.



# Activity 5: Reflections

Starting / Shutting Down the Robot Using Sensors

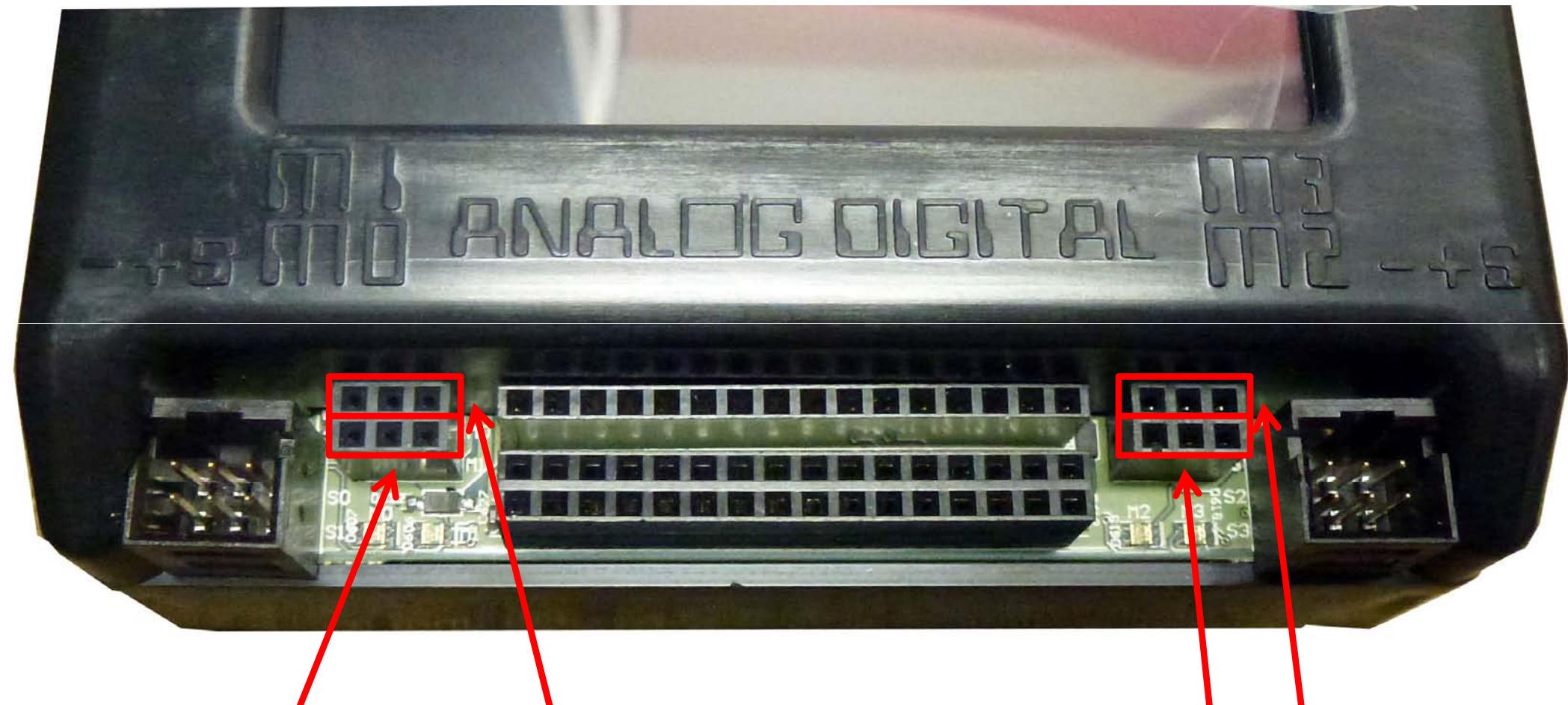
- In your own words, describe what the **wait\_for\_light** function does. How is this useful for the Botball competition?
- Describe what the **shut\_down\_in** function does? Why is it important that you use this function in your robot during the competition?
- What are the differences and similarities between analog and digital sensors?



# Motors & Servos



# KIPR Link Motor Ports



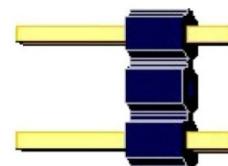
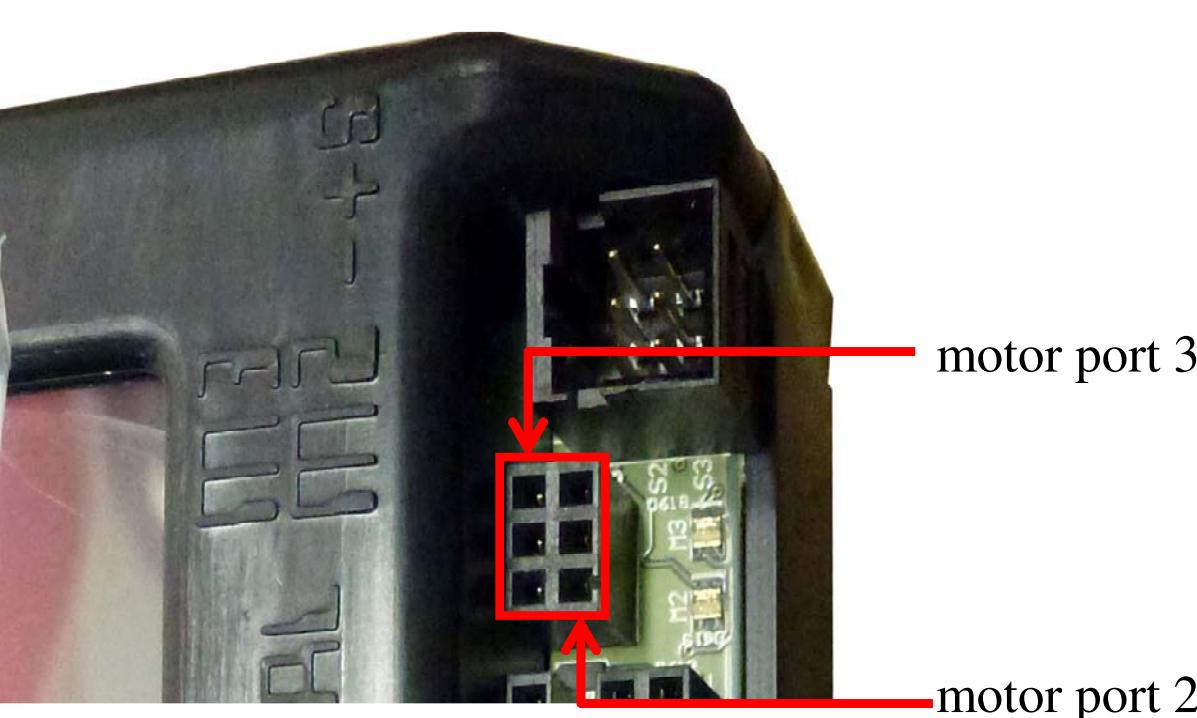
Motor ports 0 (Demobot) and 1

Motor ports 2 and 3 (Demobot)



# Plugging in DC Drive Motors

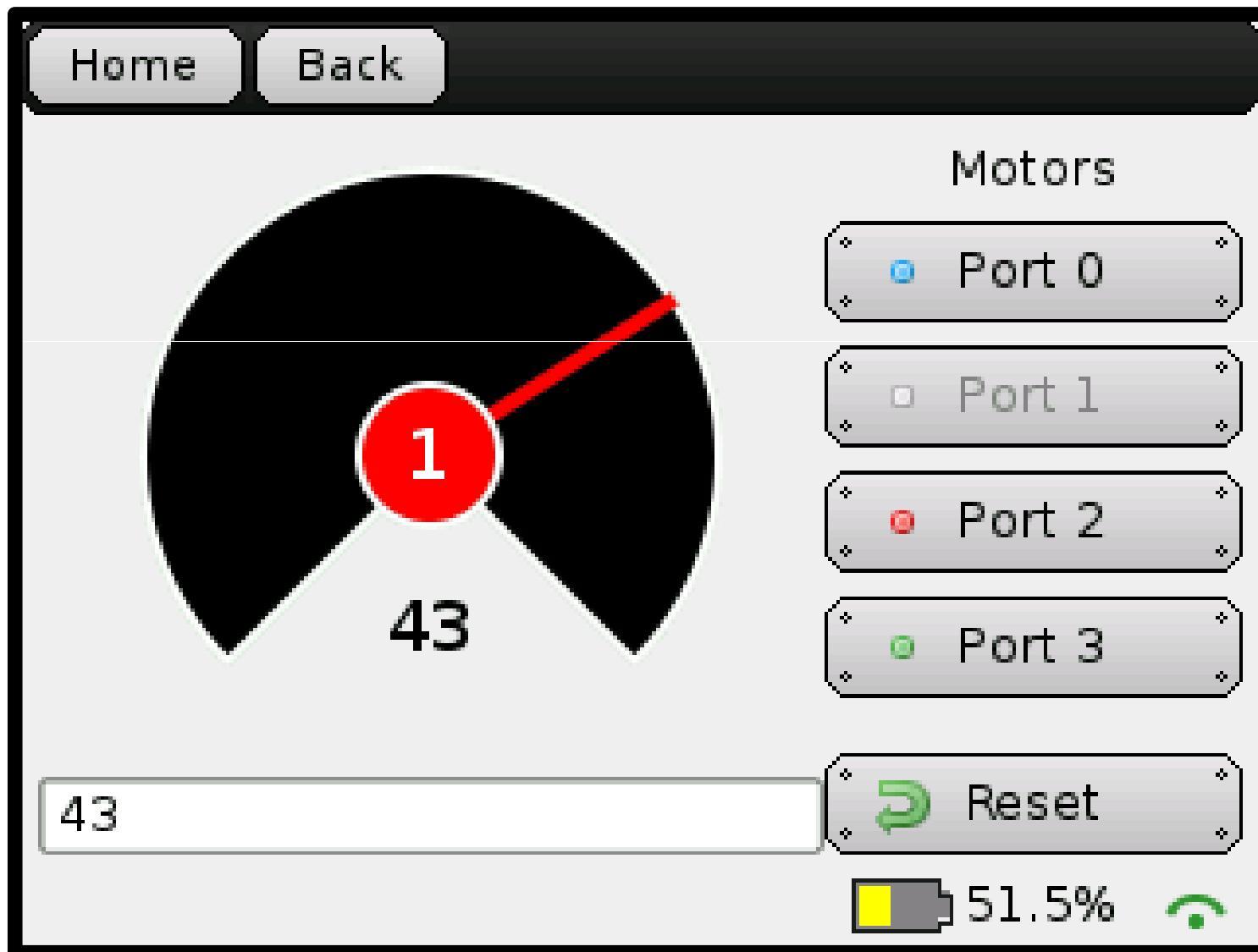
- DC drive motors are the ones with two-prong plugs and gray wires.
- The KIPR Link has 4 drive motor ports numbered 0 & 1 on the left and 2 & 3 on the right.
- When a port is powered it has a light that glows green for one direction and red for the other.
- Plug orientation order determines motor direction, but by convention, green is forward and red reverse.



Drive motors  
have a 2 prong  
plug



# KIPR Link Motor PWMs Screen





# KIPR Link Motor Commands

**mav()**, **ao()**, **off()**

- **mav (<motor#>, <vel>);** (mav means move at velocity)
  - *motor#* is the motor port (0-3) being used
  - *vel* is the rotational speed of the motor measured in ticks per second (-1000 to 1000)
  - the amount of rotation per tick depends on the kind of motor
  - the motor runs at the set speed until turned off or commanded otherwise
- **ao();** turns off all motor ports
- **off (<motor#>);** turns off the specified motor port



# Motor Position Counter

- As a DC motor runs, the KIPR Link keeps track of its current position in ticks
  - **get\_motor\_position\_counter(<motor#>);**  
is a library function that returns this value for *motor#*
  - **clear\_motor\_position\_counter(<motor#>);**  
is a library function that resets the *motor#* counter to 0
  - You can see the current value of the counter for a motor on the *motors..test* and *Sensor Ports* screens



# Motor Polarity

- Plug the drive motors into KIPR Link motor ports 0 and 3 (corresponding to simbot when running a program in the simulator)
  - Motor port numbers are labeled on the case below the screen
- Check motor polarity
  - Manually rotate each motor and observe its power light (it will glow red or green as you rotate the motor)
  - If a motor does not turn in the direction you want to correspond to forward (green), reverse its plug

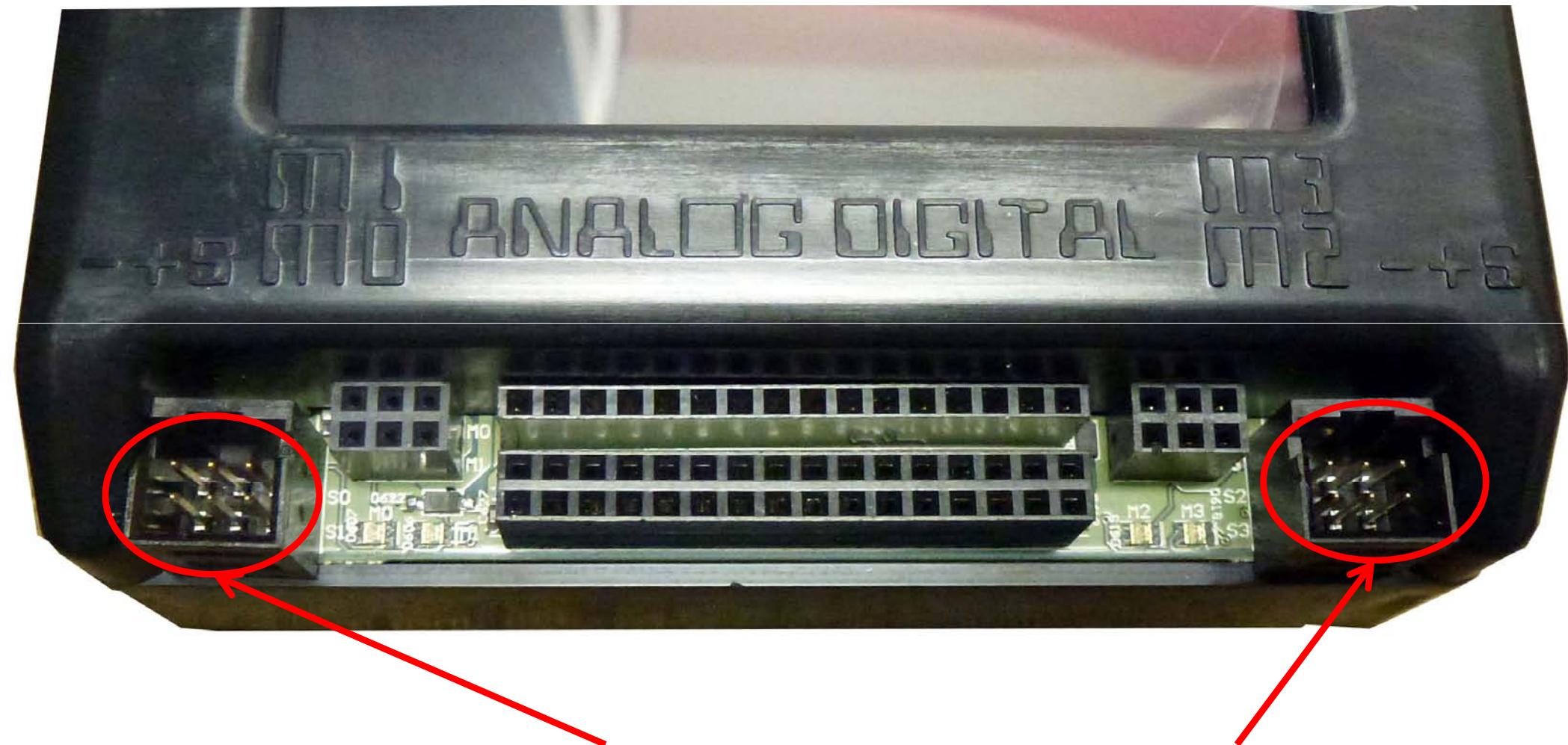


# Servo Motors

- A servo is a motor designed to rotate to a specified position and hold it
- To help save power, servo ports by default are not active until enabled
- A command is provided in the KIPR Link library for enabling (or disabling) all servo ports
  - `enable_servos();` activates all servo ports
  - `disable_servos();` de-activates all servo ports
- `set_servo_position(2, 925);` rotates servo 2 to position 925
  - Position range is 0-2047
  - You can preset a servo's position before enabling servos so it will immediately move to the position you want when you enable servos
  - Default position when servos are first enabled is 1024
- `get_servo_position(2);` provides the current position of servo 2
  - Works only when servos are enabled



# KIPR Link Servo Motor Ports



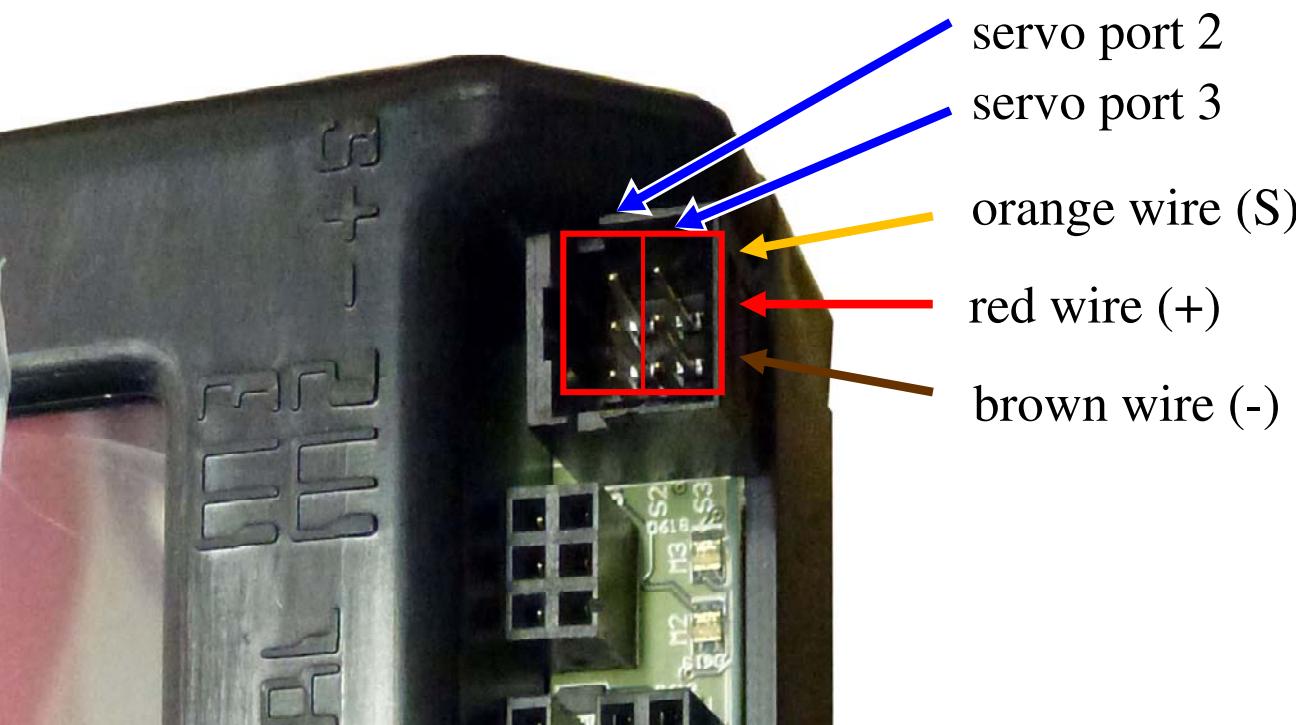
Servo Ports 0 and 1

Servo Ports 2 and 3



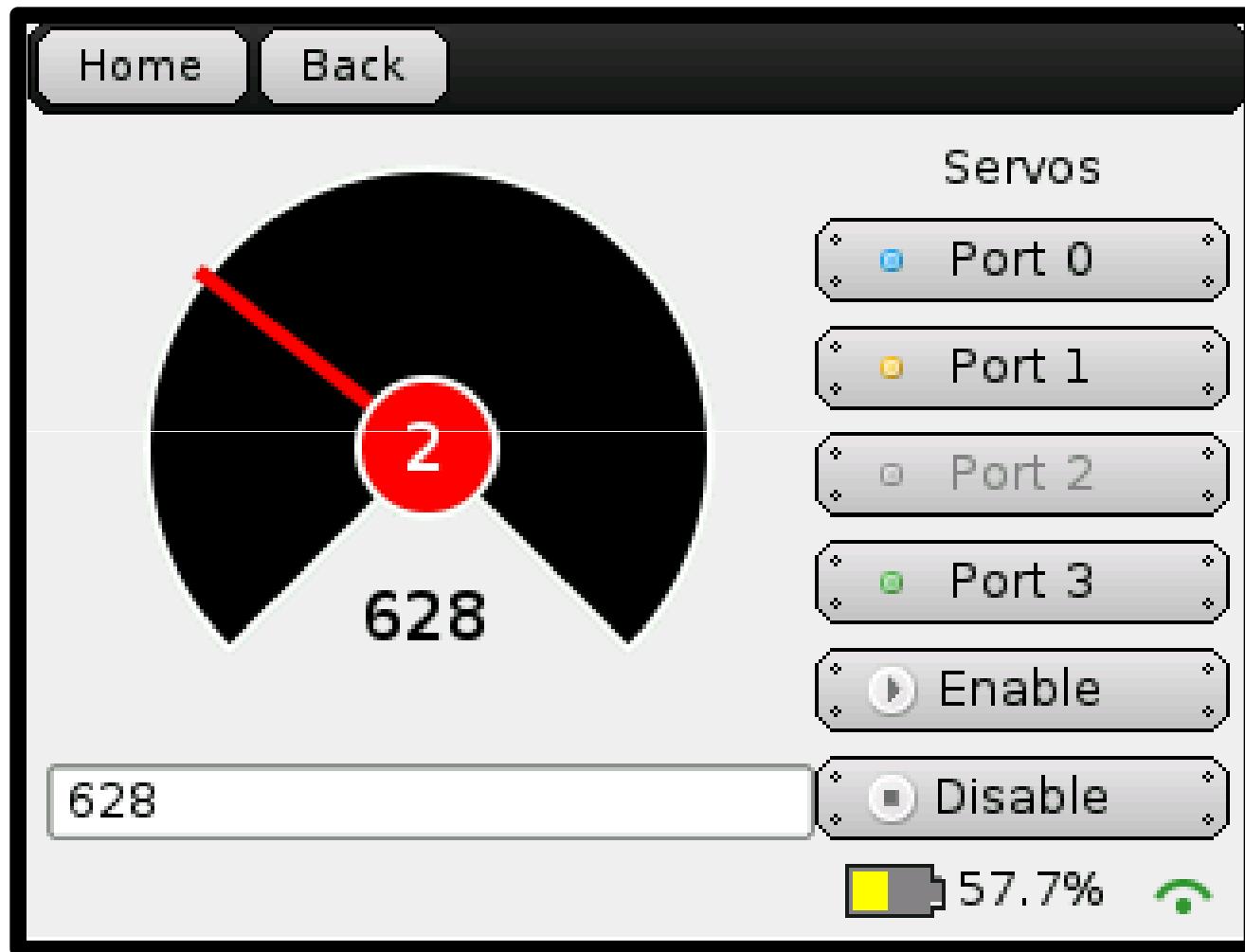
# Plugging in Servos

- Servo motors (brown/black-red-yellow cables with 3 prong receptacle) plug into the KIPR Link servo ports
- The KIPR Link has 4 servo ports numbered 0 & 1 on the left and 2 & 3 on the right
- Plug orientation order is, left to right, brown-red-orange when the KIPR Link is oriented so the screen can be read (or follow the labeling: - + S; the orange signal wire goes in S)





# KIPR Link *Servo* Screen



The KIPR Link *Servo* Test screen can be used to center a servo or determine what position values to use once the servo is installed on a bot



# Sensor and Motor Manual

For further detail about motors, consult the Sensor and Motor Manual available via KISS IDE help





# while Loop Operating a Servo

- A loop is a program construction used to repeat program steps until some condition is met
- Suppose we want to have a servo move from position 200 to position 1800 in steps of 100
  - we could do this by writing 16 separate `set_servo_position` commands after starting with `set_servo_position(1, 200);`
  - with less effort, this can be done by using a `while` loop

```
set_servo_position(1,200); // move servo 1 to position 200
msleep(100); // give servo time to move
while (get_servo_position(1) < 1800) {
    // move servo 1 in steps of 100
    set_servo_position(1,get_servo_position(1)+100);
    msleep(100); // give it time to move
}
```



# while Loop in a Program

Operate the Demobot arm using the buttons

```

int main()
{
    int s1Pos=1024;
    set_a_button_text("Down"); set_b_button_text("Quit"); set_c_button_text("Up");
    set_servo_position(1, s1Pos); // preset servo 1 position
    enable_servos(); // turn on servo
    printf("Move servo arm up and down with buttons\n");
    while(b_button()==0) { // move servo 1 in steps of 100 until quit
        if (a_button() == 1) { // direction is down
            set_servo_position(1, s1Pos-100);
        }
        if (c_button() == 1) { // direction is up
            set_servo_position(1, s1Pos+100);
        }
        s1Pos = get_servo_position(1);
        if (s1Pos > 1950) { // upper position limit is 1950
            set_servo_position(1,1950);
        }
        if (s1Pos < 150) { // lower position limit is 150
            set_servo_position(1,150);
        }
        s1Pos = get_servo_position(1);
        if (a_button() != c_button()) { // if a button is pressed report position change
            printf("servo at %i\n", s1Pos);
        }
        msleep(200); // pause before next move
    }
    disable_servos();
    printf("done\n");
    return 0;
}

```



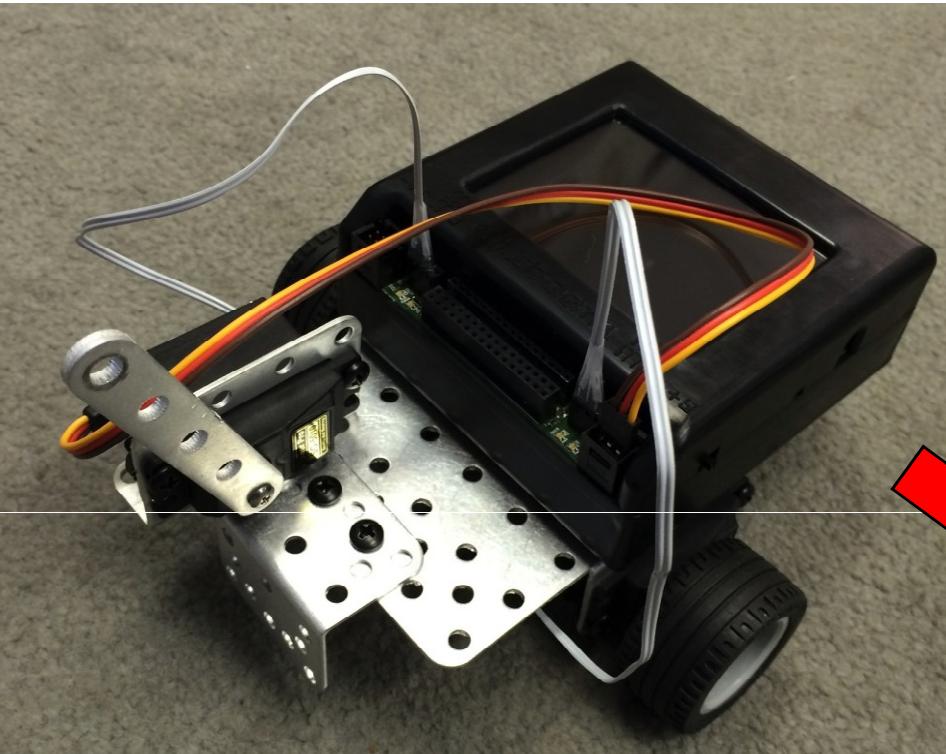
# Activity 6 (Objectives)

Motors and Servos

Lift the Demobot to a desired position  
using the servo and the accelerometer

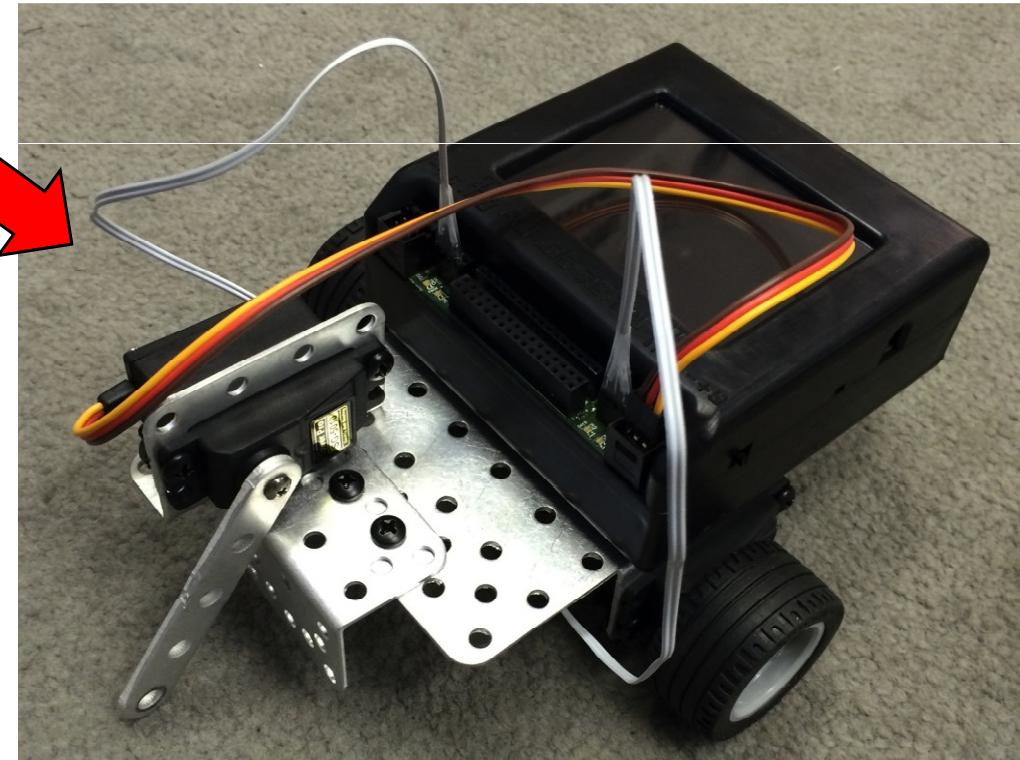


# Expected Behavior



Before

After





# Activity 6

## Motors and Servos

- Have the robot detect when it is tilted, then stops the servo motion
- You should rely on the accelerometer values, not the servo position



# Activity 6: Solution

## Motors and Servos

```

***** Stop when accelerometer shows robot has tilted
*****
int main() {
    // preset servo 1 position
    printf("advance using A button\n\nB to quit\n");
    set_servo_position(1,200);
    enable_servos(); // turn on servos
    msleep(2000); // pause while it moves and user reads screen
    while((accel_y() > -150) && (b_button() == 0)) {
        // move servo 1 in steps of 100
        set_servo_position(1, get_servo_position(1)+100);
        printf("servo at %d\n", get_servo_position(1));
        msleep(200); // pause before next move
        while((!a_button()) && (!b_button())) { // wait for button
        }
    }
    disable_servos();
    printf("Tilt! Robot is done\n");
    return 0;
}

```



# Activity 6: Reflections

## Motors and Servos

- Why is the value of the B button being checked in each while statement?
- Why is the **msleep** statement before the second while?
- What does the robot do when **disable\_servos** is executed?

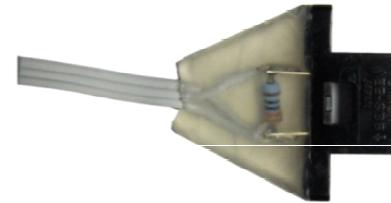


# Analog and Floating Analog Sensors



# IR Reflectance Sensors

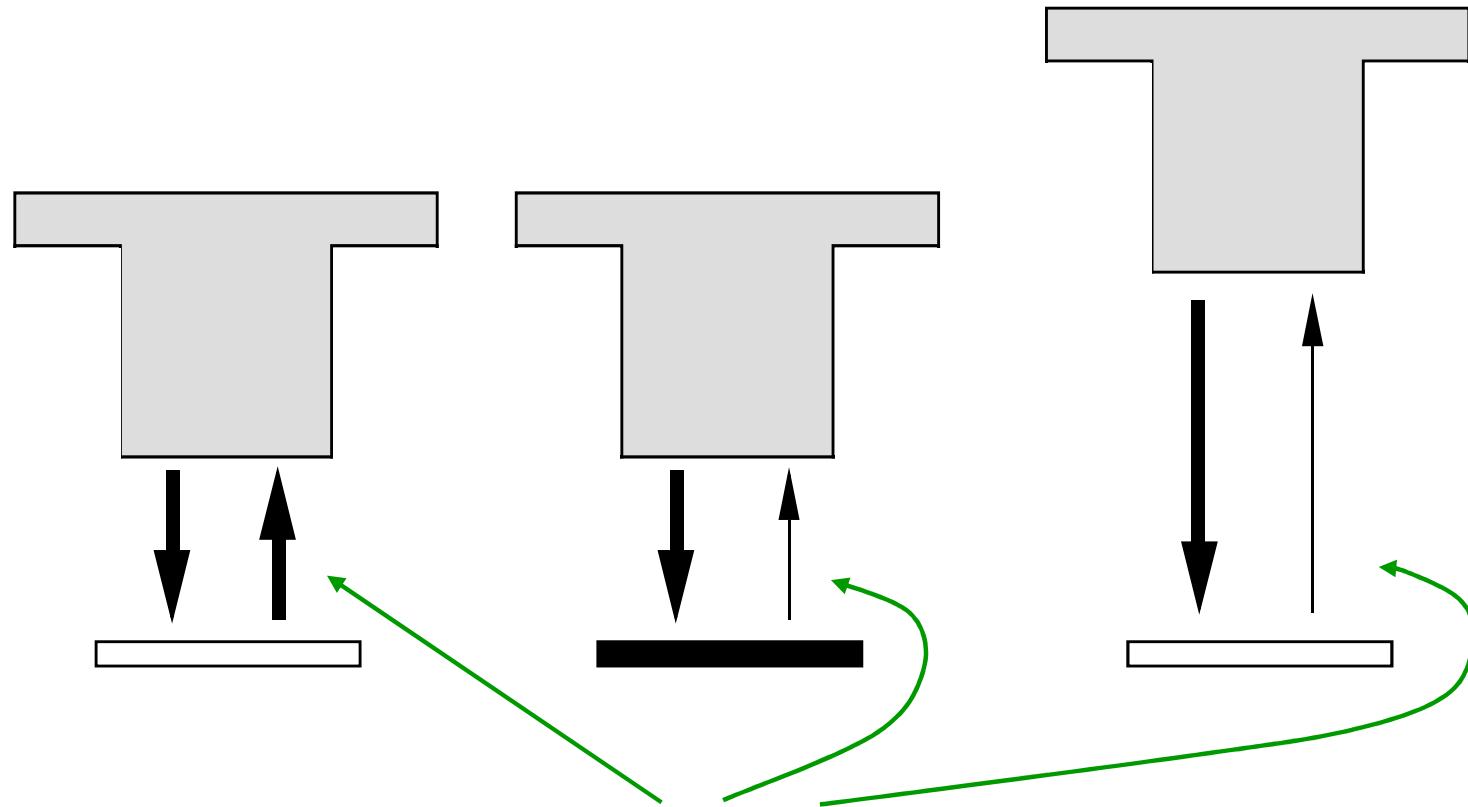
- An IR reflectance sensor has an emitter producing an IR beam and an IR light sensor that measures the amount of IR reflected when the beam is directed at a surface.
- There are two reflectance sensors in the Botball kit



- The KIPR Link library function **analog** returns an amount that measures the amount of light reflected as a value between 0 and 1023.
- A dark spot reflects less IR, producing a higher reading.



# IR Reflectance Sensor Behavior



Amount of reflected IR depends on surface texture, color, and distance to surface (higher values mean less IR indicating a dark surface or a drop off)



# Demo

- Plug a reflectance sensor in port 0 and a light sensor in port 2

```
int main()
{
    while(b_button() == 0) {
        printf("reflectance: %i ", analog(0));
        printf("light: %i\n", analog(2));
        printf("B button exits\n");
        msleep(1000);
    }
    return 0;
}
```



# Optical Rangefinder "ET"

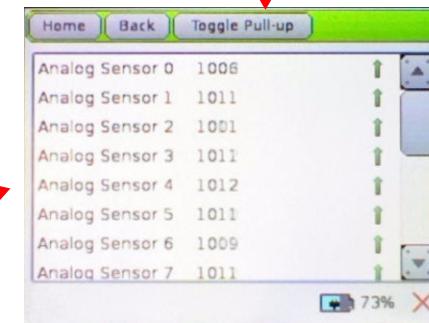
- Analog sensor
- Connect to analog ports 0-7
- Access with library function **analog\_et (port#)**
  - This is a special analog function for the ET sensor
- Low values (0) indicate large distance
- High values indicate distance approaching ~4 inches
- Range is 4-30 inches. Result is approximately  $1/d^2$ . Objects closer than 4 inches will produce values indistinguishable from objects farther away





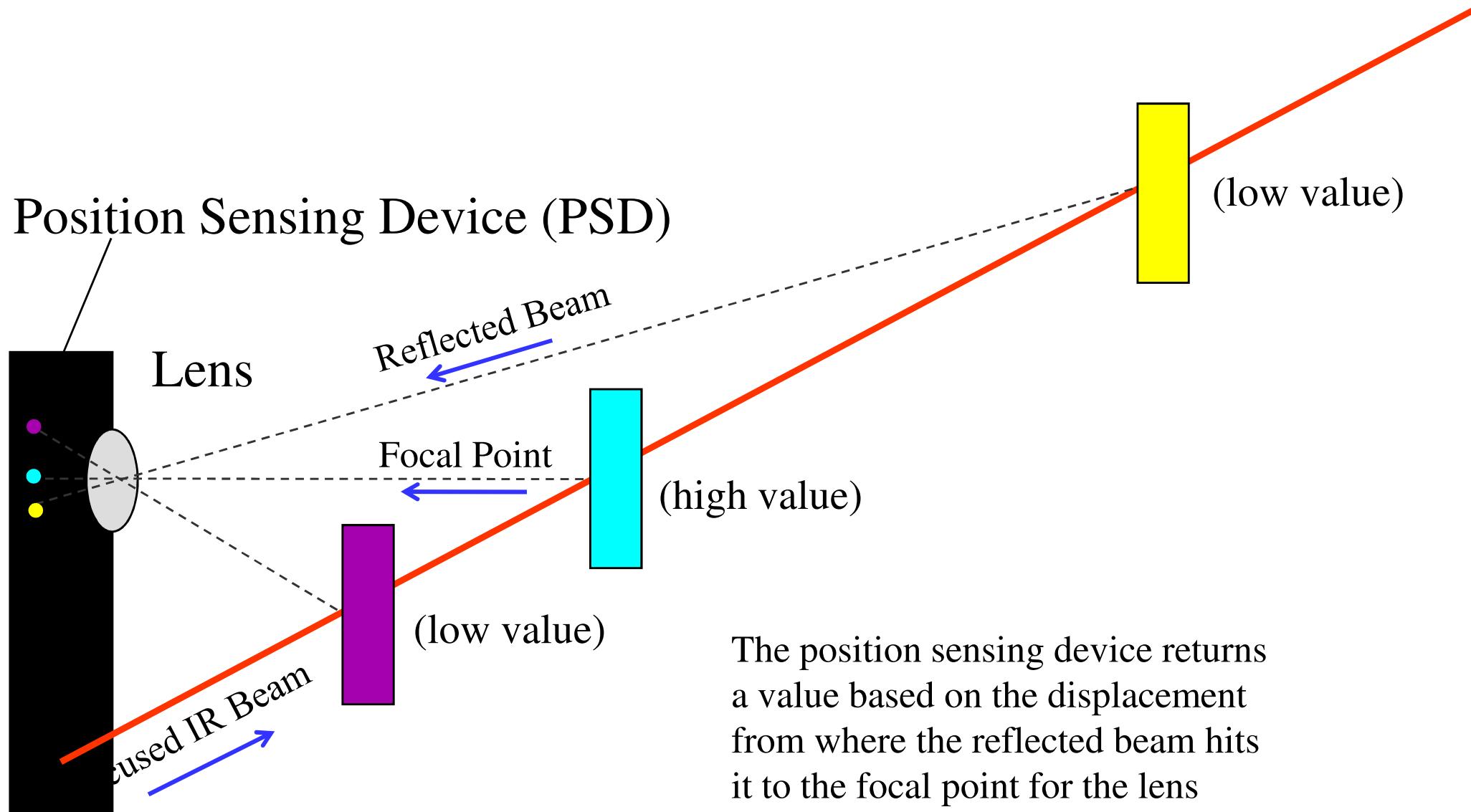
# Pull-Up Resistors

- Most sensors need a pull-up resistor to register accurate values
  - Pull-up resistors are engaged (↑) by default
- Some sensors, e.g., the ET range sensor requires the port to be **floating**, i.e., have no pull-up resistor
- The KIPR Link can change an analog port to be either analog (pull-up resistor) or floating analog (no pull-up)
  - **`set_analog_pullup(3, 0);`** sets port 3 to floating and leaves the other analog ports as they were
- For testing, highlighting a port on the *Sensor List* screen and pressing *Toggle Pull-up* will switch the port between pull-up resistor enabled (↑) or not.





# Optical Rangefinder





# Demo

```
int main()
{
    while(a_button() == 0) {
        printf("ET: %i (A Button Exits)\n", analog_et(1));
        msleep(1000);
    }
    printf("All done\n");
    return 0;
}
```



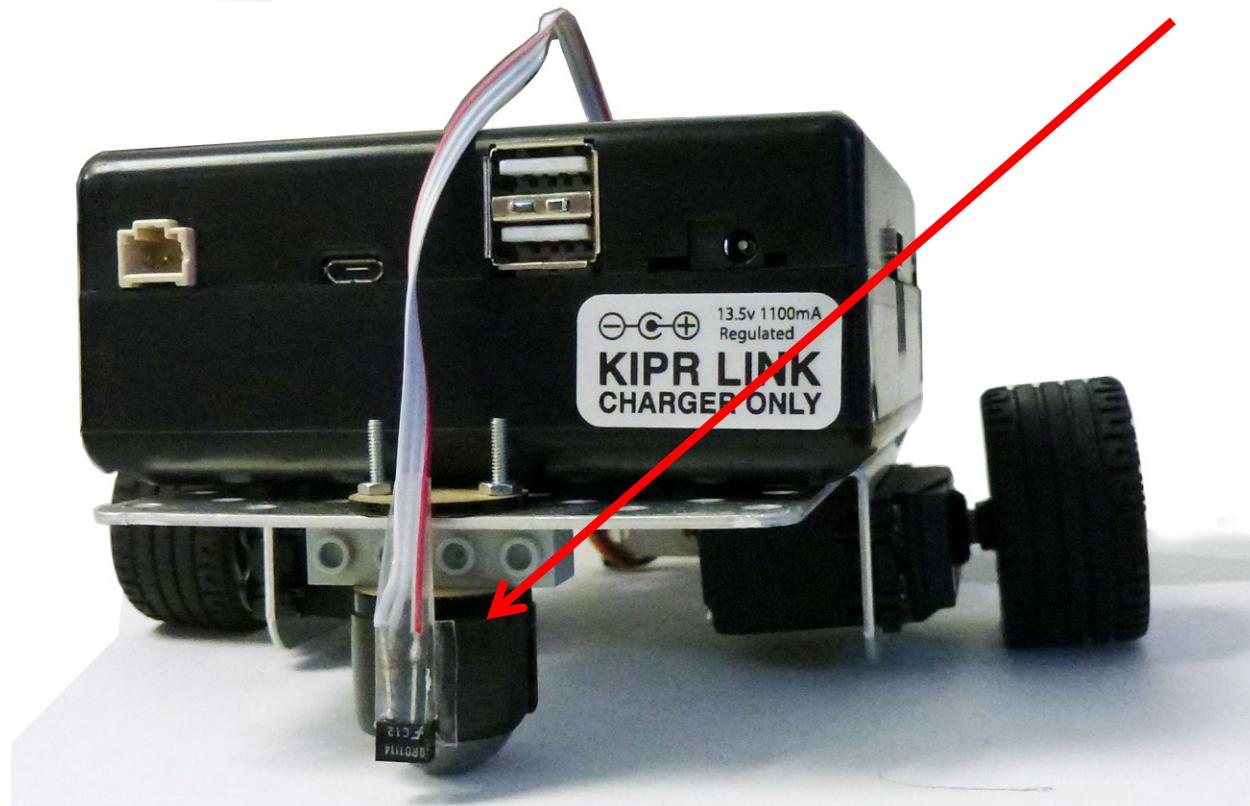
# Line Following



# Objectives

Line following

Have a robot follow a line it can detect using a reflectance sensor attached with UGlu





# Prep

## Line following

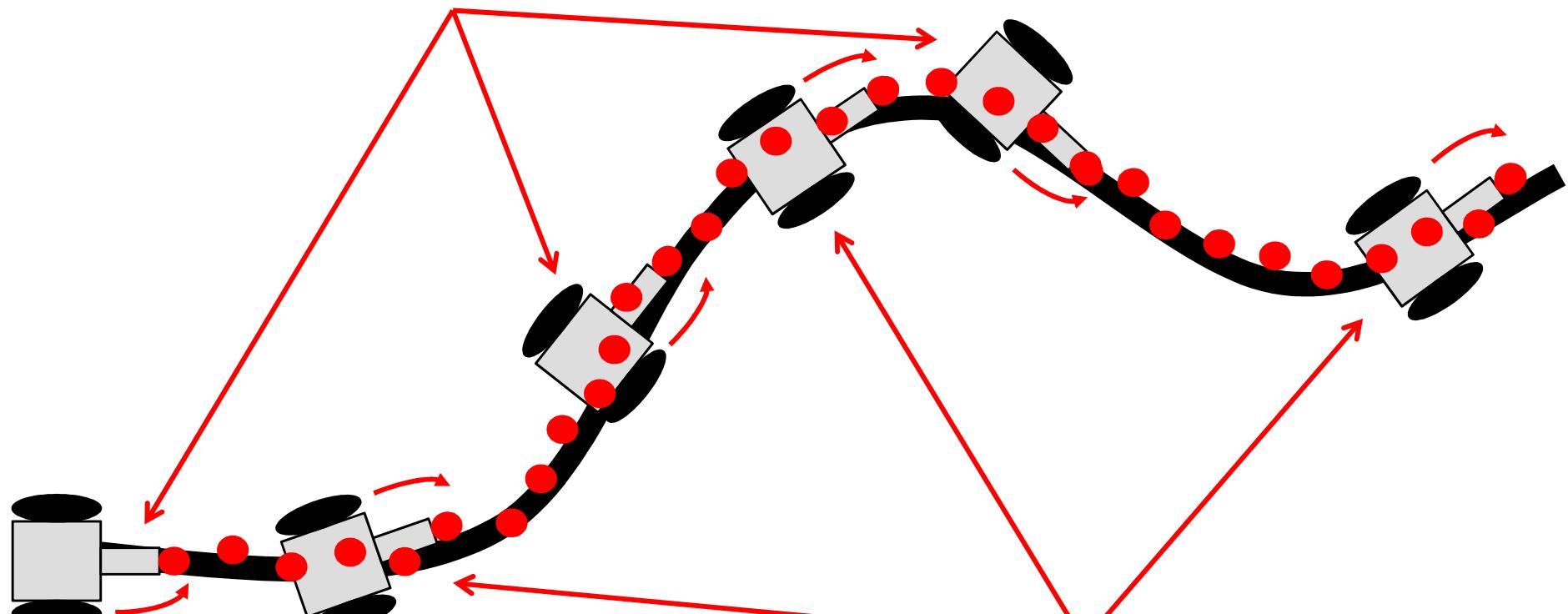
- Reflectance sensors
- Line following strategies
- Turning through an arc
- Robot setup
- Program steps for activity



# Line Following Strategy

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

1. If detecting dark, arc left



2. If detecting light, arc right



# Robot Setup

Line following

Position your robot so the sensor is over the line and observe values on the *Sensor Screen* as you move the sensor left or right over the line.



# Program Steps

## Line following

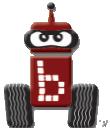
- Activity (DemoBot)
  - Starting with the sensor over the line have your program repeat the following two steps
    1. Until the sensor detects dark, turn in an arc left
    2. Until the sensor doesn't detect dark, turn in an arc right
      - the robot follows the left edge of the line
      - Each step requires a loop (indicated by the word until)



# Activity 7

## Line following

- Write a program to follow black tape line on a surface enough lighter than the tape that a reflectance sensor can tell whether or not it is over the tape.
  - Have your program wait for the side button to be pressed to start following the line.
- If you can't figure it out, then there is a solution in a couple of slides.



# Activity 7: Reflections

## Line following

- What happens if a turn in the line is too tight?
- What happens if there is a gap in the line?
- What happens when the robot reaches the end of the line?
- Only 1 reflectance sensor is being used for line following in this activity. What strategy using additional reflectance sensors might improve accuracy and/or allow you to go faster?



# Activity 7: Solution

## Line following

```

int main()
{
    // identify port and motors
    int sensor_port = 5, left_port    = 0, int right_port  = 2;

    // set wheel powers for moving in an arc (curve)
    int high_speed  = 100, low_speed   = -10;

    // set threshold for light (reflectance) conditions
    int threshold   = 512;
    printf("Line following: position robot on tape\n");
    printf("Press B button when ready\n\nPress side button to stop\n");
    while(b_button() == 0) { // wait for button press
    }
    while(side_button() == 0) { // stop if button is pressed
        if (analog(sensor_port) > threshold) { // if dark, arc left
            motor(left_port, low_speed);
            motor(right_port, high_speed);
        }
        else { // if not dark, arc right
            motor(left_port, high_speed);
            motor(right_port, low_speed);
        }
    }
    ao(); // stop because button pressed
    printf("Done!\n");
    return 0;
}

```



# Day 1 Homework

- Reuse Teams: Bring LEGO bricks and Create!
- Work on activities you didn't get to or come in early to work on them
  - Challenge: implement line following with the Create (the cliff sensors are reflectance sensors – see the KISS IDE help file for the function syntax)
- Thoroughly review the game slides on your Team Home Base
  - There will be no game review tomorrow, only a 30 minute Q&A
- Test sensors, motors, and KIPR Link ports
- Review the KISS IDE *Help*
- Review the manuals on your Team Home Base
  - KIPR Link Manual
  - Sensors and Motors Manual
- Review the BOPD Manual (Botball Online Project Documentation)
- Review "New items for 2014" on your Team Home Base
- Read the "Hints for New Teams Manual" on your Team Home Base
- Send your instructor any questions for the Day 2 recap (email or paper)



# Botball 2014

## Educators' Workshop

### Day 2



# Botball 2014

## Educator's Workshop

### Day 2

1. Sign in
2. Put the Links back on charge
3. Get the new slides, game video, and firmware
4. Review Game rules from Team Home Base
5. Early arrivers verify your DemoBot is ready
6. Work on line following!



# Please take the survey!

This survey provides KIPR valuable information  
on the workshops!

<https://www.surveymonkey.com/s/FPHLQ98>

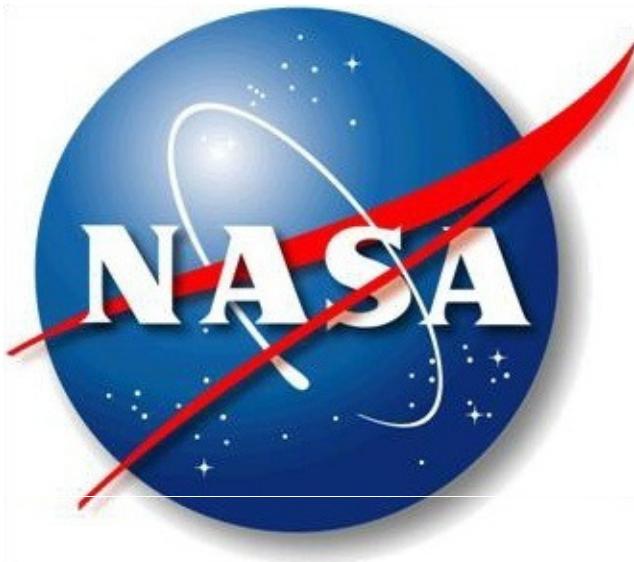


# Workshop Schedule

- Day 1:
  - Overview of Botball
    - Botball season, related events
    - Game preview/video
    - Resources & teams
  - Topics and Activities
    - Activity 0: The KISS IDE
    - Activity 1: Programming basics
    - Activity 2: Driving straight
    - Activity 3: Build DemoBot (throughout)
  - Lunch
    - Activity 4: Conditions and functions
    - Activity 5: Starting / shutting down the robot using sensors
    - Activity 6: Motors and servos
    - Activity 7: Line following
  - Homework
- Day 2:
  - New Team Suggestions
  - BOPD
  - 30 minute game Q&A
  - T-shirts and Awards
  - Topics and Activities
    - Activity 8: Depth sensor
  - Lunch
    - Activity 9: Vision
  - Selected activities
    - Activity 10: Point Servo at colored object
    - Activity 11: Bang-Bang control
    - Activity 12: Proportional control
    - Activity 13: Bang-Bang DemoBot arm
    - Activity 14: Proportional DemoBot arm
    - Activity 15: Accelerometer for bump detect
    - Activity 16: Graphics for custom user interfaces
    - Activity 17: Reduce accumulated errors
    - Activity 18: Reading QR codes



# 2014 Botball National Sponsors



**iRobot®**

**COMMON SENSE RC**

**igus®**

**D3S SolidWorks**

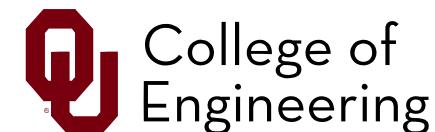
**Botball®**



# 2014 Regional Botball Sponsors



To Oklahoma & You.<sup>SM</sup>



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



KIRKPATRICK FOUNDATION



**robonation**  
ROBOTICS COMMUNITY

**Botball**<sup>®</sup>

# 2014 Regional Workshop & Tournament Hosts



SOUTHERN ILLINOIS UNIVERSITY  
**EDWARDSVILLE**

GROSSMONT  
COLLEGE



*Preparing people to lead extraordinary lives*



WORCESTER  
STATE  
UNIVERSITY



جامعة كارنيجي ميلون في قطر  
**Carnegie Mellon Qatar**



**Botball®**

# GCER 2014



**USC** University of  
Southern California



The 2014 Global Conference on Educational Robotics will be held at the University of Southern California Galen Center from **July 30th - August 3rd, 2014** with preconference classes on July 29<sup>th</sup>

**Global Conference on Educational Robotics**  
<http://www.kipr.org/gcer>



# Global Conference on Educational Robotics

## ALL TEAMS ARE INVITED!

### When

- July 30<sup>th</sup> – August 3<sup>rd</sup>
- Pre-conference activities and workshops July 29<sup>th</sup>



### Who

- Middle school and high school students, educators, robotics enthusiasts, and professionals from around the world

### Activities

- Meet and network with students from around the country and world
- Talks by internationally recognized robotics experts
- Teacher, student, and peer reviewed track sessions
- International Botball Tournament
- KIPR Open Tournament (Botball for grown-up kids!)
- Aerial Robotics Competition
- Autonomous Robotics Showcase
- Elementary Botball Challenge



# Suggestions and Tips

The following are helpful hints and suggestions, but by no means the only way to implement and manage your program

There is a lot of information on the next set of slides.

This information will be reference material.



# Right 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.
- You only have a limited build time before the tournament and time is of the essence!
- The workshop will still be fresh in your mind if you start now.
- Plan on meeting sometime during the first week after the workshop.



# Right After the Workshop!

Students will not inherently know how to mange their time, let's face it, it is hard for many adults!

## 3. Plan Out The Season

- Mark a calendar or make a Gantt chart with important dates:
  - 1st submission documentation due
  - 2nd submission documentation due
  - 3rd submission documentation due
  - Tournament date
- Set dates and schedules for team meetings
- Plan on meeting a **minimum** of 4 hours per week. (Botball teams average 8 hrs/week nationwide)
- Team meetings can be held with the first order of business being going over the calendar and any upcoming due dates



# Right After the Workshop!

### 3. Plan Out The Season (continued)

- A **large** calendar or project plan displayed where everyone can see it is a good way to go.
- You can draw one on your whiteboard (If the janitor doesn't erase it) or put it on butcher or poster paper.
- The local lumber supply store (Lowes or Home Depot) will carry 4' X 8' sheets of melamine backed 1/8" Masonite, that is relatively inexpensive (~\$12). You can write on it just like a whiteboard, using a permanent marker for the grid and whiteboard (erasable) markers for everything else. It can easily be cut into smaller sizes and mounted on the wall.



# Right After the Workshop!

## 4. Build the Game Board

- The material list and construction instructions for the KIPR tournament setup are on the **team home base**.
- This can be a great parent, mentor and student activity.
- The cost is ~ \$100, but you can reuse the expensive FRP for next year's game board.
- The board is designed so that you can take it down and put it back up in your classroom.
- Many teams have a classroom or another room in the school where they can leave it set up. Your school may or may not have another room you can use, but it doesn't hurt to ask.



# Right After the Workshop!

## 5. 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.
- You might be able to talk with another team who does have a board that they would let your team use on a practice day. If you are in this position and don't know whom to contact, call us and we can make introductions and see if something can be set up in your area.





# Right After the Workshop!

## 6. Kit Organization

- Organized parts can lead to faster & easier construction and redesign of robots.
- Tupperware® containers, tackle boxes, anything that keeps the parts organized.
- This makes it easier to lock or move the components when you have another class or are not working on the robots, including transporting everything to the tournament.
- If a part breaks, it is easier to find a replacement.
- This is a good job for team members and will help them learn what is in the kit by sorting and counting.



# Right After the Workshop!

## 6. Kit Organization (continued)

- Tupperware® containers or cardboard boxes are great for holding the robots in progress.
- Allows for easier transport to the tournament.
- You can keep the robots from distracting other classes.
- You can keep the robots safe.
- REMEMBER- There are no requirements to use all of the parts included in the kit.



# Right After the Workshop!

## 6. Understand the Game

- This is what you should go over with your students on the first meeting after the workshop.
- Go over the game by using the game table you have built or by drawing the game field on the board or by projecting the game field (on the team home base) onto a screen.
- The goal is to have students identify game pieces and areas on the board where points are scored.
- The game board has markings to help team's robots navigate or locate their position.
- If it is on the board there is a reason for it. For example: A black line leading from the starting box to the scoring area could be used by a line following program.



# Right After the Workshop!

## 6. Understand the Game (continued)

- The game always includes multiple tasks that score points.
- **There is always a relatively easy way to score points. DO NOT overlook this.**
- Many teams are very successful because they get the simple, “easy points” consistently every time their robot(s) run.
- Focus on one behavior/task at a time
- Start with easy tasks and work your way to the more difficult ones
- **Remember, scoring a few points consistently and having success is better than being unsuccessful and scoring no points.**
- **Score early and score often!**



# Don't forget about the resources!

The Link, sensor and motor manuals contain a lot of useful information.

- They are electronic, but some teachers choose to print them out and put them in 3 ring binders for easy access by the students.
- This is also true of the game rules & specifications and the documentation requirements.
- This can be a great student activity and it gives you an easy answer when students ask a question pertaining to those topics; “Did you look in the binder?”
- Use the construction hints (pictures of previous robots and robot subsystems) on the team home base.



# Ideas on Construction

It is important to note that our competition tables are built to specifications with allowable variance.

- **DO NOT** engineer robots that are so precise a 1/4" difference in a measurement means they are not successful. For example: the specified height of the elevated platform is 15", but at the tournament the platform could actually measure 15 1/8". If your arm is set for exactly 15" it will not work.
- Review construction documents (like the ones on the Home Base) to get building ideas
- Search the internet for other robots and structures to get building ideas
- Test structure robustness before the tournament



# Communication with KIPR

- **Newsletters:** These have important information-check your spam filter.
- **Emails:** These include important information so read them and forward them to your team-check your spam filter.
- **Team Home Base:** Check this often, especially the FAQ for rule questions and technical solutions.
- You can call KIPR staff/**technical support** during office hours 8:30-5:00 pm CT 405-579-4609.
- You can also **email** [support@kipr.org](mailto:support@kipr.org) any time.
- Use the **community site** <http://community.botball.org/>
- **Programming tutorial** <http://nasarobotproject.wordpress.com>



# A word about Documentation

- What?
  - Botball Online Project Documentation (BOPD)
- When?
  - 3 periods during design and build portion
  - 1 onsite presentation (8 minute) at regional tournament
- Why?
  - To reinforce the engineering process
  - **POINTS EARNED IN DOCUMENTATION FACTOR INTO OVERALL TOURNAMENT SCORES!**

See BOPD Handbook on the Team Home Base for more information (rubrics and exemplars)



# When you come to the tournament...

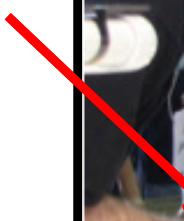
- Encourage your students to understand the rules and not be afraid to challenge a judge's scoring.
- Adults are NOT allowed in the pits. The adults can help the students get settled in and then they must leave.
- Bring ALL of your equipment; Especially your charging cables, extra LEGO, etc.
- Plan on staying for the awards even if you lose early! There are a lot of Judge's Choice Awards!



# About the starting lights....

The competition game board will have two moveable lights on each side.

Adjustable “gooseneck” lamp





# About the starting lights....

- The competition game board will have two moveable lights on each side which can be positioned on the table edge along the perimeter of the starting box and adjusted as necessary, but not so as to project over the game surface.
- All robots must use a light sensor and be programmed to **wait for the light** and then start autonomously.
- Robots must shut down automatically at the end of the match.
- If students do not understand and accomplish this, the robots will never start or they will not shut down and they will be disqualified.
- After calibration, do **not** move the robot or light sensor
- **MAKE SURE** your students understand how to shield and mount their light sensors, adjust the starting lights and calibrate them prior to the tournament.



# Management Ideas

## Recruit Some Help

- If possible, recruit another teacher or parent to help out.
- Parents do not have to be engineers or programmers to help.
- Someone to help organize, bring snacks, sit in the classroom, oversee students and keep them on task can be a big help.

## Divide and Conquer

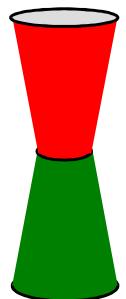
- You have two robots to design, build and program divide the team between the two robots.
- Don't forget about the documentation.
- Divide robots performance tasks into subtasks and use them as assignments.
- Facilitate- Keep them in check on goals, time lines and expectations. Team meetings are great for this.



# Management Ideas

## Herding Cats

- Many students will need a lot of help and practice working independently (not running to you for every question or problem)
- Set a policy/procedure of – “before coming to me did you ... check the resources, ask your team mates, look online, etc?”
- Use the green cup/red cup
  - Tape a green and red cup together (base to base) for each group.
  - The group leaves the green cup UP if they have no questions or issues.
  - They turn the red cup UP if they have gone through the policy/procedure and still need help.



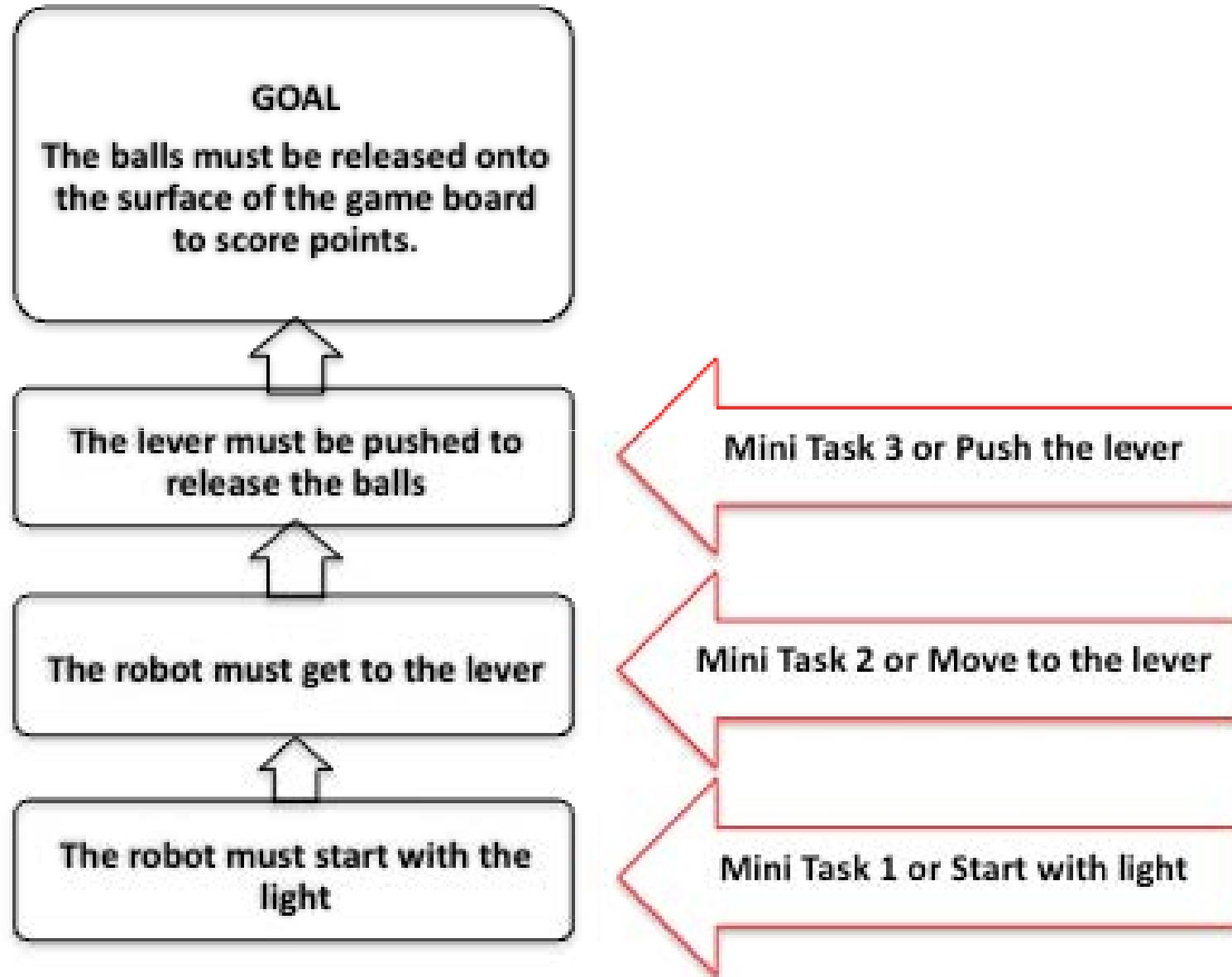


# How do I teach this?

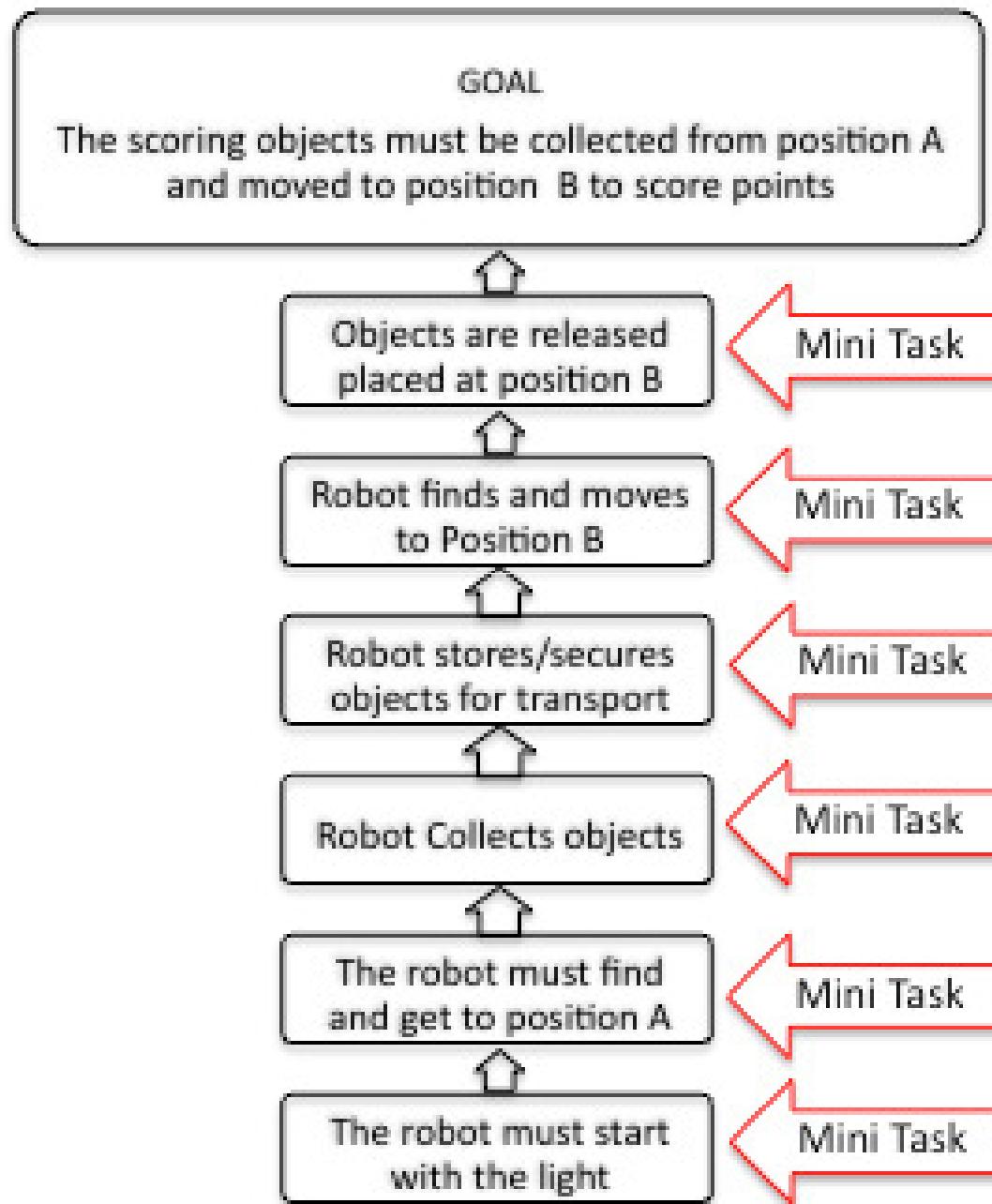
- Start with the “**easy points**” by having the students discuss and document **what has to happen** to score these points (goal) by working backwards (Task Analysis) from the desired goal.
- Have a planning strategy meeting to set common goals
- Working backwards helps the students focus on the goal and the step-by-step, sub task or “**Mini Tasks**” they have to accomplish to complete the final task.
- Keep calendar up to date with tasks and assignments



# How do I teach this?



# How do I teach this?



- Move on to the more complicated “What has to happen” for the Harder Task B (for harder points)



# How do I teach this?

- When students want the robot to do (task A) and then (task B) and then (task C) the charts add together making the mini tasks needed to accomplish both summative and dependent upon one another.
- Finish task A before moving on to task B.
- After completing task B recheck the functionality of task A.
- **REMEMBER THE LAST TASK** is always to shut the robot down.
- Students will have no basis to predict how long it will take them to complete mini tasks (often underestimating the time required).
- In the end, if they are successful in completing only one task (goal) A, they have been successful and most likely will be competitive at the tournament



# How do I teach this?

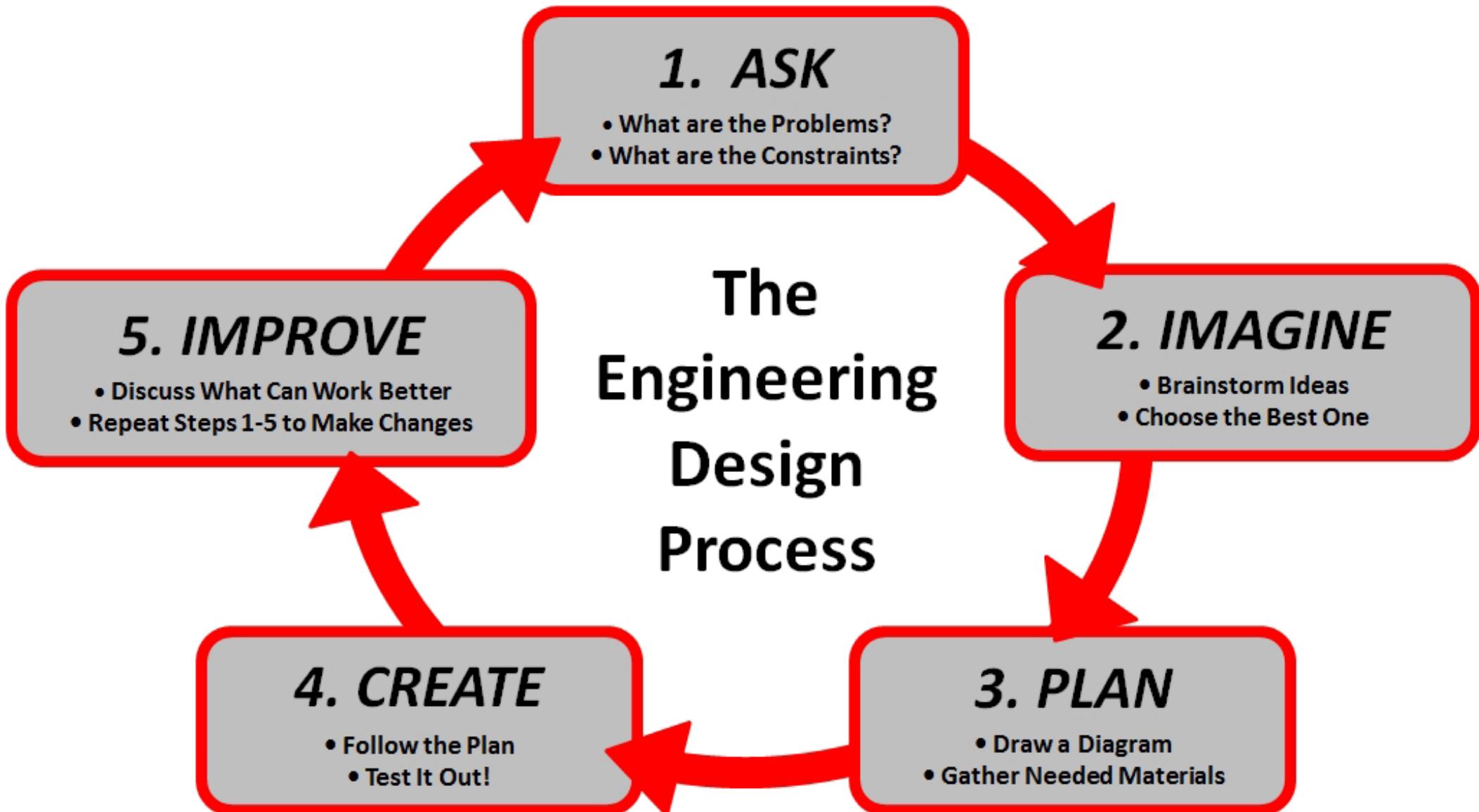
The students have arrived and asked; what do we do?

- Is their robot successfully and reliably completing mini task A or completing the wait for light and starting? Yes, great, now start working on mini task B.
- Many students will successfully complete the mini task once and think that is sufficient.
- Develop a metric to determine success, (e.g., it must wait for the light, then start and work 10 out of 10 times, or 4 out of 5, etc).
- If the task is continually giving them problems, reevaluate. Maybe a mechanical adjustment will make it work or have they double checked their code? Then explore solutions in the workshop examples, the Botball community site or by calling KIPR?



# How do I teach this?

The Engineering Life Cycle provides an overview of the entire process.





# When you come to the tournament.

- Adults are NOT allowed in the pits.
- Bring ALL of your equipment. Especially your charging cables, extra LEGO, etc.
- Bring your computers.
- Bring a power strip.
- Plan on staying for the awards.
- Check the Botball FAQ!
- Make a Checklist!
- Prepare Your Onsite Presentation!
- Money for lunch or snack.
- Bring a CD or flash drive with a back up file of your code.



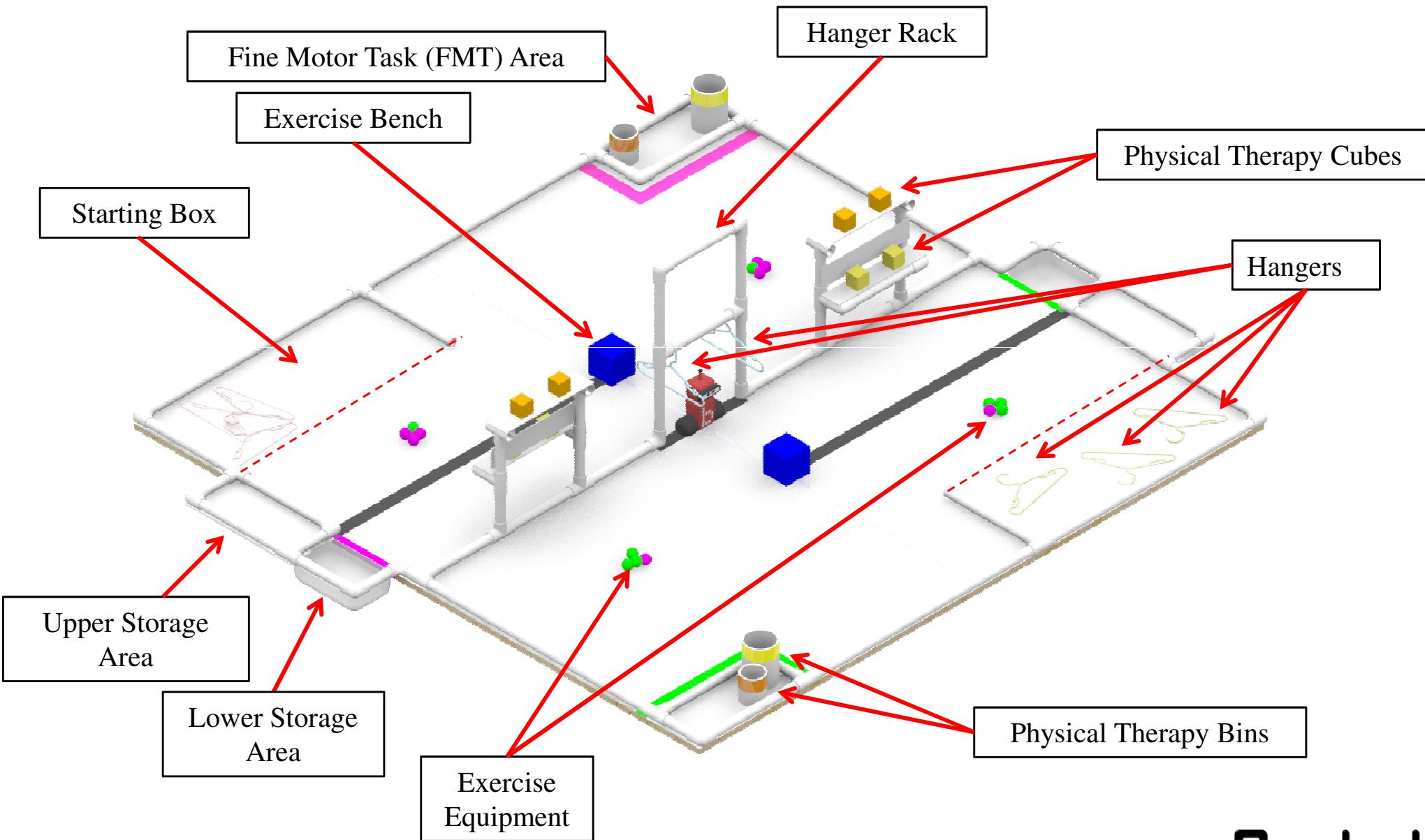
# Preview of This Year's Game

Q&A is Next



# The Game Board

## Q & A Session



# Botball T-Shirts



FRONT



BACK

Team preorder - \$7  
At tournament - \$10

## Note:

T-shirts are not provided  
One preorder per team  
Can be modified up to 1 week prior to tournament

[botball.org/shirts](http://botball.org/shirts)

**Botball®**



# Tournament Awards



**Botball®**



# Tournament Awards

**There are a lot of opportunities for teams to win awards!  
A few are listed below.**

- Tournament Awards
  - Outstanding Documentation
  - Seeding Rounds
  - Double Elimination
  - Overall
- Judges' Choice Awards
  - KISS Award
  - Spirit of Botball
  - Outstanding Engineering
  - Outstanding Software
  - Spirit
  - Outstanding Design/Strategy/Teamwork





# Asus<sup>®</sup> Xtion PRO Depth Sensor

Special thanks to our sponsor **Asus<sup>®</sup>**  
for their donation of Xtion PRO sensors!



# Prep

## Depth Sensor

- Depth sensor setup and overview
- Depth image and image coordinates
- Depth sensor library functions
- World coordinates
- Scanline operations and objects

<code>depth_open()</code>	<code>get_depth_value(&lt;row&gt;, &lt;col&gt;)</code>
<code>depth_close()</code>	<code>get_depth_world_point_x(&lt;row&gt;, &lt;col&gt;)</code>
<code>depth_update()</code>	<code>get_depth_world_point_y(&lt;row&gt;, &lt;col&gt;)</code>
<code>get_depth_world_point_z(&lt;row&gt;, &lt;col&gt;)</code>	
 <code>depth_scanline_update (&lt;row&gt;)</code>	
<code>get_depth_scanline_object_... (&lt;obj&gt;)</code>	



# Depth Sensor Setup

- The **Asus® Xtion PRO** depth sensor in the Botball kit will work in either of the KIPR Link's USB ports
- Plug in the depth sensor and you will be able to see the depth image by going to the *Depth* screen under the *Motors and Sensors* menu
  - If you unplug the depth sensor, the KIPR Link *might* not recognize it again if you plug it back in...
    - You will need to restart the KIPR Link if this happens



# Depth Sensor Debugging

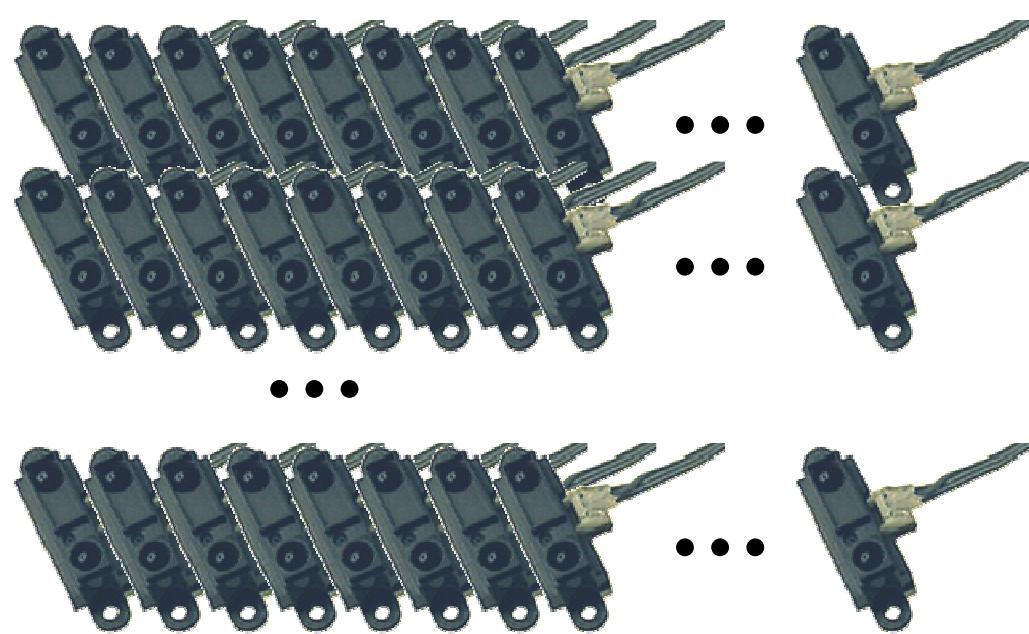
- If you are having problems opening the connection the depth sensor, try the following:
  - Try again.
  - Restart the KIPR Link.
  - Restart the KIPR Link **with the depth sensor unplugged**, then plug the depth sensor back in once the KIPR Link has finished booting.
- If you are using the depth sensor
  - and get the following message when your program ends simply ignore it:  
`pure virtual method called  
terminate called without an active  
exception`
  - and the KIPR Link GUI crashes (the screen goes black) simply wait a few seconds—the KIPR Link GUI will restart on its own
  - And are using the KIPR Link Network and it crashes (the network connection turns off) restart the KIPR Link.



# Overview:

## What do we mean by “depth”?

- Here, “**depth**” means the same as “**distance**” or “**range**” similar to the ET Infrared Rangefinder.
- The **Asus® Xtion PRO** depth sensor provides thousands of depth (distance) measurements...
  - It’s like having a grid of thousands of ET Infrared Rangefinders!



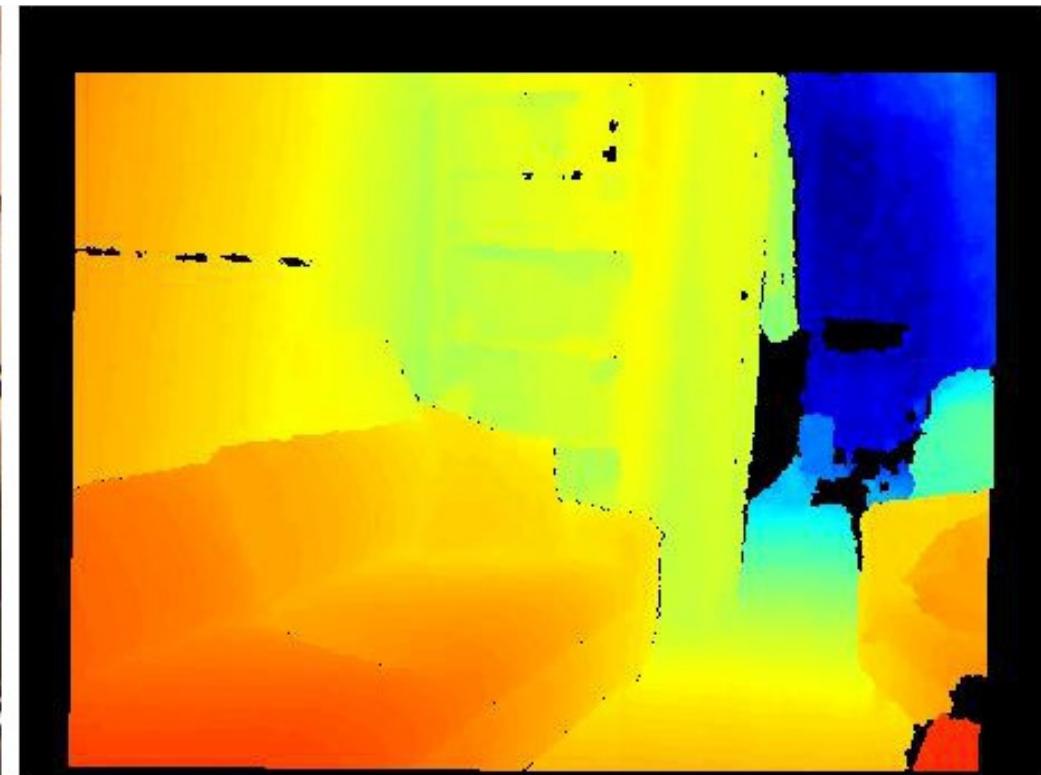


# Overview:

## What do thousands of **depth** measurements look like?



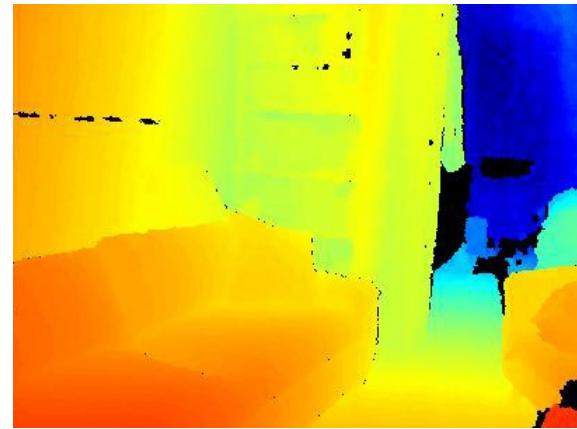
We see this.



The **depth sensor** sees this.  
This is called a “**depth image**”.



# Depth Image



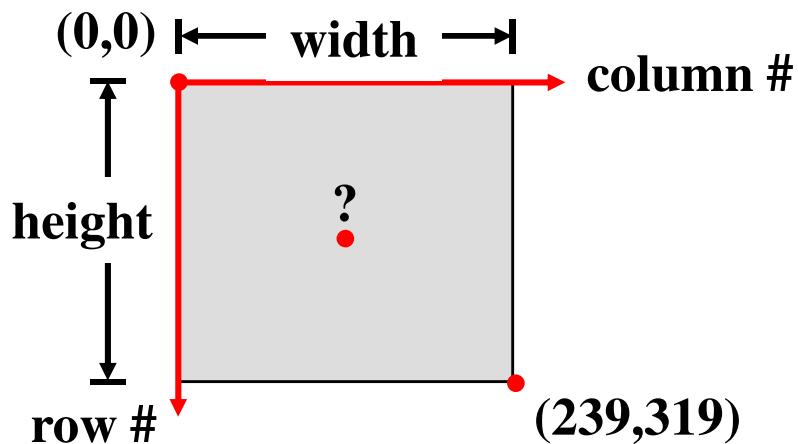
- Wow—it's *sooooo* colorful!
  - The *depth sensor* does not actually see color.
  - The colors are only there to make it easier for *you* to observe differences between depth readings.
  - *Near* depth readings are more **red**.
  - *Farther* depth readings are more **blue**.
  - *Invalid* depth readings (too near/far) are displayed in **black**.
- Each point (“pixel”) in the depth image has an associated **int** value—the depth at that point...
  - The depth value is in **millimeters (mm)**.
  - The *minimum* depth is approx. **500mm** (approx. 20”).
  - The *maximum* depth is approx. **5,000mm** (approx. 200”).
  - *Invalid* depth readings (too near/far) will return 0.



# Depth Image Coordinates

- Each pixel in the depth image is accessed by its (*row#*, *column#*) coordinate...
  - The upper-left pixel has coordinates (0, 0)
  - The lower-right pixel has coordinates (239, 319) at the default depth image resolution (320 x 240)...
    - default depth image ***width*** (# of columns) is 320
    - default depth image ***height*** (# of rows) is 240
  - The Link display might distort the depth image.

What are the coordinates of the **center** pixel in the depth image?





# Depth Sensor Library Functions:

**depth\_open**, **depth\_close**

- To *open a connection* to the depth sensor, use the **depth\_open()** function.
  - This enables the depth sensor
    - The depth sensor draws significant power, so should not be started until needed
  - The function will return 1 if it found the sensor, and 0 if it did not  
**(note:** sometimes it fails to find the sensor, so check this value!).

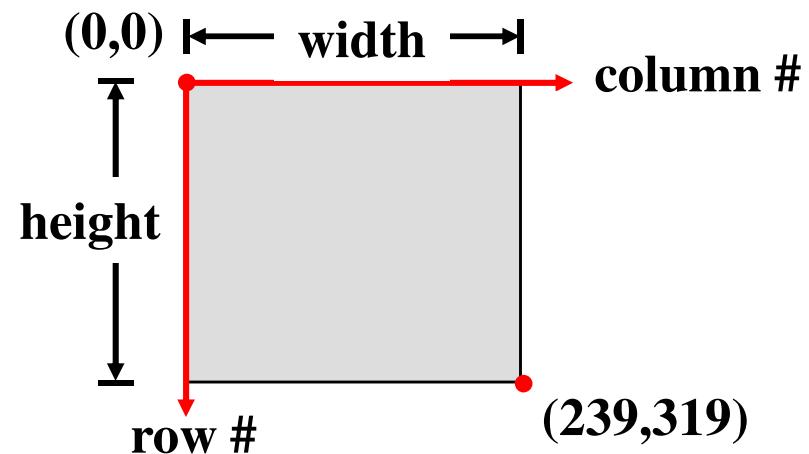
```
if (depth_open() == 0) {
    printf("Failed to connect to depth sensor.\n");
}
```
- To *close a connection* to the depth sensor, use the **depth\_close()** function.
  - This disables the depth sensor (conserving power).
  - Should be run before the end of your program.



# Depth Sensor Library Functions:

`get_depth_image_width`, `get_depth_image_height`

- To get the current *width* and *height* of a depth image, we use the `get_depth_image_width()` and `get_depth_image_height()` functions, respectively.
- Recall that the *default* depth image resolution is (320x240), so...
  - `get_depth_image_width()`  
returns 320 *by default*
  - `get_depth_image_height()`  
returns 240 *by default*



- See the *KIPR Link Getting Started Manual* for information on changing the depth sensor resolution.



# Depth Sensor Library Functions:

## `depth_update`, `get_depth_value`

- The KIPR Link library function `depth_update()` is a command that causes the KIPR Link to capture the most recent depth image from the depth sensor.
- The `get_depth_value(<row#>, <column#>)` function returns the depth value at the specified (row, column)

```
while (b_button() == 0) {  
    depth_update();  
    printf("Depth at center is %d mm\n",  
          get_depth_value(120,160));  
}
```

- The depth value is in **millimeters (mm)**.
- ***Invalid*** depth readings (too close/far) will return 0.



# Demo

Connecting to and using the depth sensor.

```
int main()
{
    // Begin main function code block.
    // 1. Attempt to open a connection to the depth sensor.
    if (depth_open() == 0) {
        printf("Failed to connect to depth sensor\n");
        return 1;
    }

    // 2. Until the B-button is pressed, use the depth sensor.
    while (b_button() == 0) {
        depth_update(); // Update the depth image.

        // Print the depth value (in millimeters) of the center pixel in the depth image.
        // Note: this code assumes the *default* depth image resolution of 320x240.
        printf("Depth at center is %d mm\n", get_depth_value(120, 160));
        msleep(100);
    }

    // 3. Close the connection to the depth sensor.
    depth_close();

    // 4. Tell the user program has finished.
    printf("Done!\n");

    // 5. Exit the program by returning 0.
    return 0;
} // End main function code block.
```



# Activity 8a: Description

## Driving with the depth sensor

- Mount the depth sensor to the DemoBot...
  - You can either loosely mount the depth sensor with tape or UGlu, or you can follow the next set of slides for a stronger mount.
  - Note: consider placing the depth sensor on the back of the robot (rather than the front) to maximize its effective range.
- Program the DemoBot to drive forward if it is farther away than 1 meter from an obstacle, or stop if it is closer or if it sees nothing.



# Activity 8a: Extensions

## Driving with the depth sensor

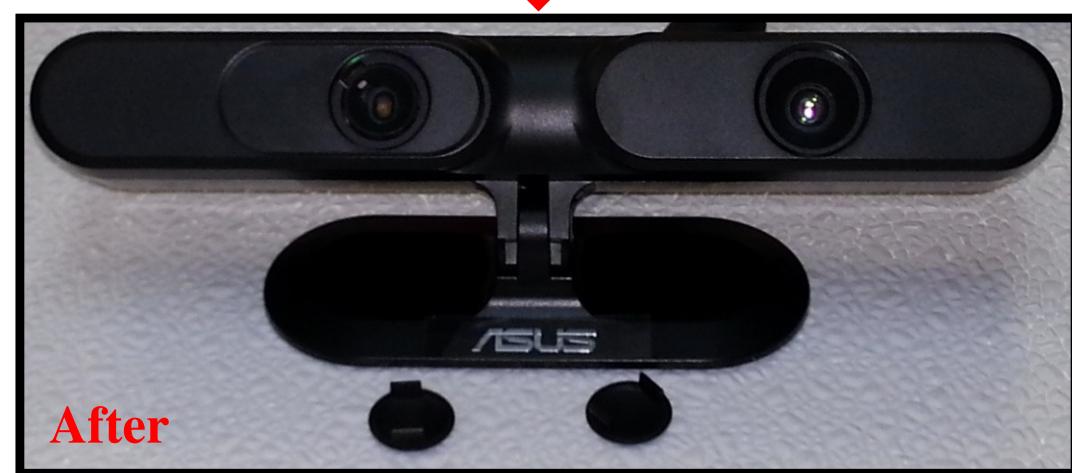
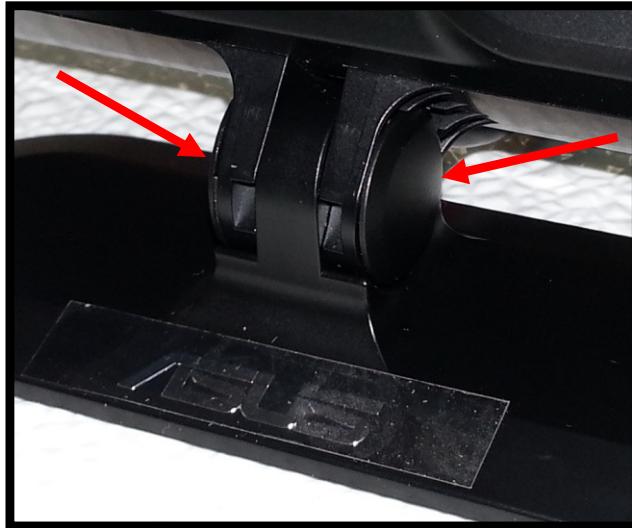
- Measure the distance from the front of the depth sensor to the obstacle after the DemoBot stops. How accurate is it?
- Try the same task using the ET Infrared Rangefinder. How do they compare?
- Make the robot “square up” with a wall in front of it...
  - Hint: use two calls of `get_depth_value` with different column values.
- Make the robot follow you within a certain distance!



# Activity 8a: Build

## Mounting the Asus® Xtion PRO depth sensor to the DemoBot.

**Step 1:** remove screw covers





# Activity 8a: Build

Mounting the **Asus® Xtion PRO** depth sensor to the DemoBot.

**Step 2:** unscrew Phillips head screw, remove nut





# Activity 8a: Build

## Mounting the **Asus® Xtion PRO** depth sensor to the DemoBot.

**Step 3:** remove spacer, remove base

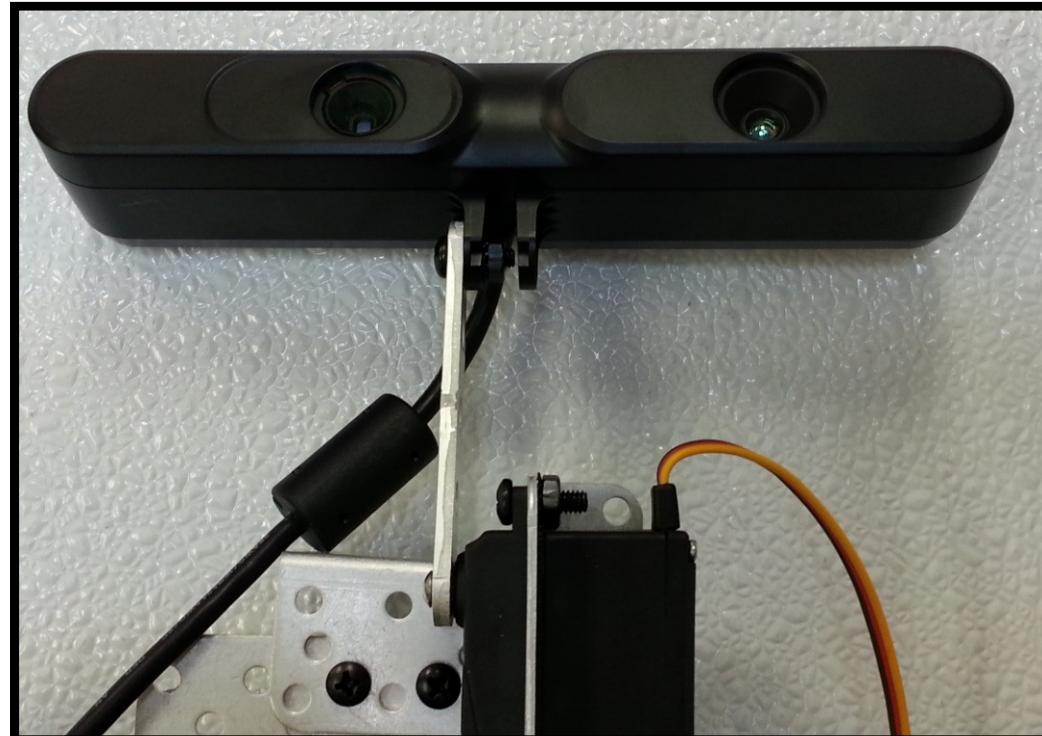
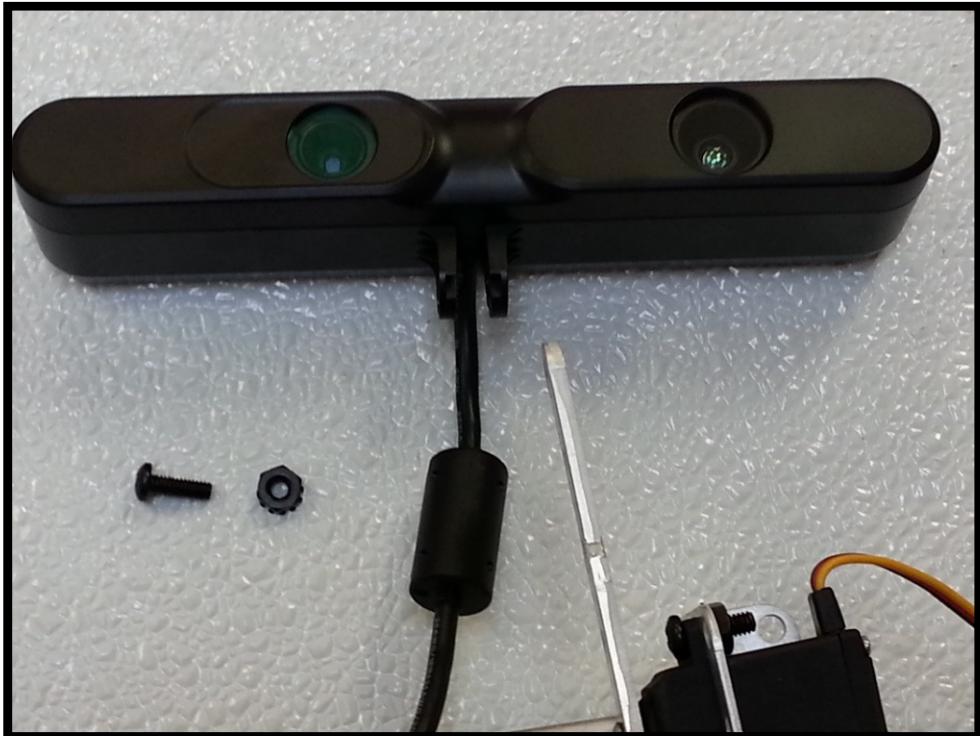




# Activity 8a: Build

## Mounting the **Asus® Xtion PRO** depth sensor to the DemoBot.

**Step 4:** use a medium length screw (1/2" long) and a keeps nut to attach the Xtion to DemoBot servo arm. Note that the metal servo arm is on the outside of the Xtion mounting slot and the nut is on the inside.





# Activity 8a: Solution

## Driving with the depth sensor

```

int main()
{
    // Begin main function code block.
    // Attempt to open a connection to the depth sensor.
    if (depth_open() == 0) {
        printf("Failed to connect to depth sensor\n");
        return 1;
    }
    // Announce the program and wait for the A-button to be pressed.
    printf("Press A-button to drive robot forward until 1m\n");
    while (a_button() == 0) { // Wait for A-button to be pressed.
    };
    // Drive forward until the robot is within 1 meter of an obstacle in front of it.
    // Note: this code is using the depth of the center pixel in the depth image.
    while (b_button() == 0) {
        depth_update(); // Update the depth image.
        printf("Depth is %d mm\n", get_depth_value(120, 160));
        if (get_depth_value(120, 160) > 1000) { // Check if the depth value is greater than 1 meter.
            motor(0, 50); // Note the motor port number -- this might be different for your robot!
            motor(3, 50); // Note the motor port number -- this might be different for your robot!
        }
        else { // The depth value is less than 1 meter or is not valid (i.e., 0).
            ao(); // Stop (turn off all motors).
        }
        msleep(100);
    }
    depth_close(); // Close the connection to the depth sensor.
    printf("Done!\n"); // Tell the user program has finished.
    return 0; // Exit the program by returning 0.
} // End main function code block.

```



# Activity 8a: Reflections

Driving with the depth sensor

- How stable are the depth readings when stopped vs. moving?
- How does the angle of the depth sensor itself affect its depth readings?
  - Hint: think about the floor...
- How accurate is the depth sensor?
- How does this compare to using the ET Infrared Rangefinder?
- How well did the robot follow you? How can you make it better?



# World Coordinates:

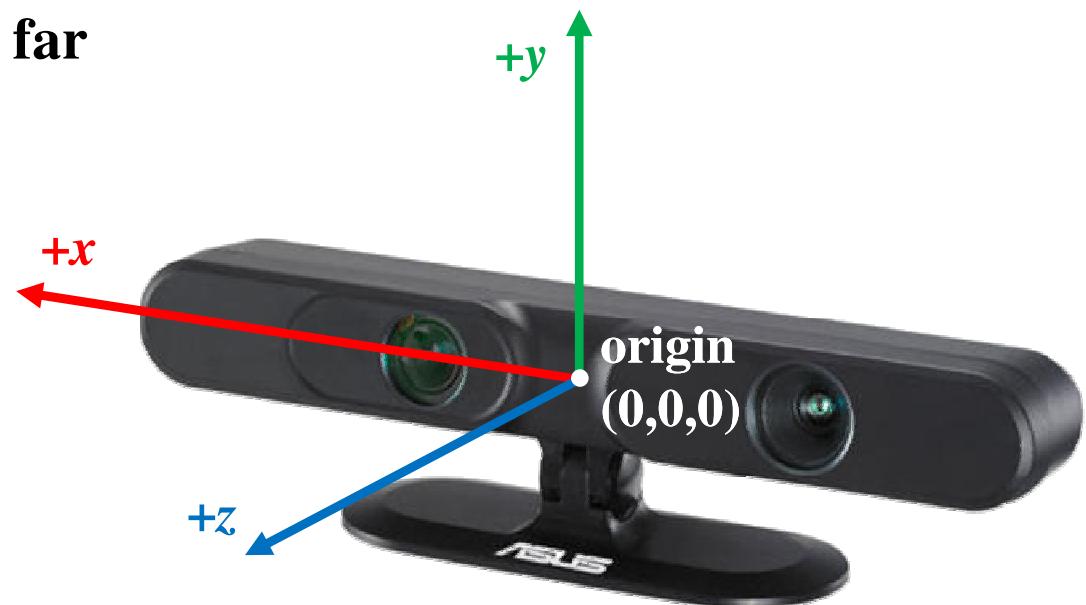
- The previous functions provided us with the depth at a particular pixel (*row*, *column*).
- However, it would be advantageous for us to be able to identify the exact ( $x$ ,  $y$ ,  $z$ ) location of a point (in **millimeters**) in the real world.
- To do this, we use three function for each real-world axis ( $x$ ,  $y$ ,  $z$ ):
  - `get_depth_world_point_x(<row#>, <column#>)`
  - `get_depth_world_point_y(<row#>, <column#>)`
  - `get_depth_world_point_z(<row#>, <column#>)`



# World Coordinates:

What do we mean by  $(x, y, z)$ ?

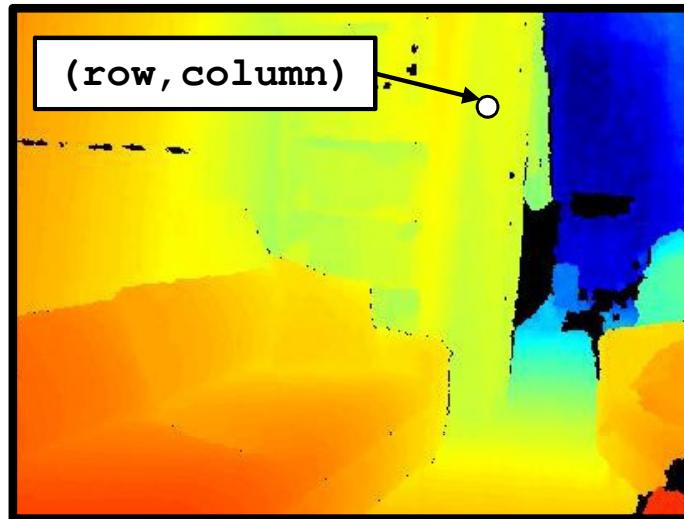
- The ***origin***  $(0,0,0)$  is at the center of the depth sensor.
- From the depth sensor **origin**...
  - $x$  is **positive** from **left** to **right**
  - $y$  is **positive** from **bottom** to **top**
  - $z$  is **positive** from **near** to **far**



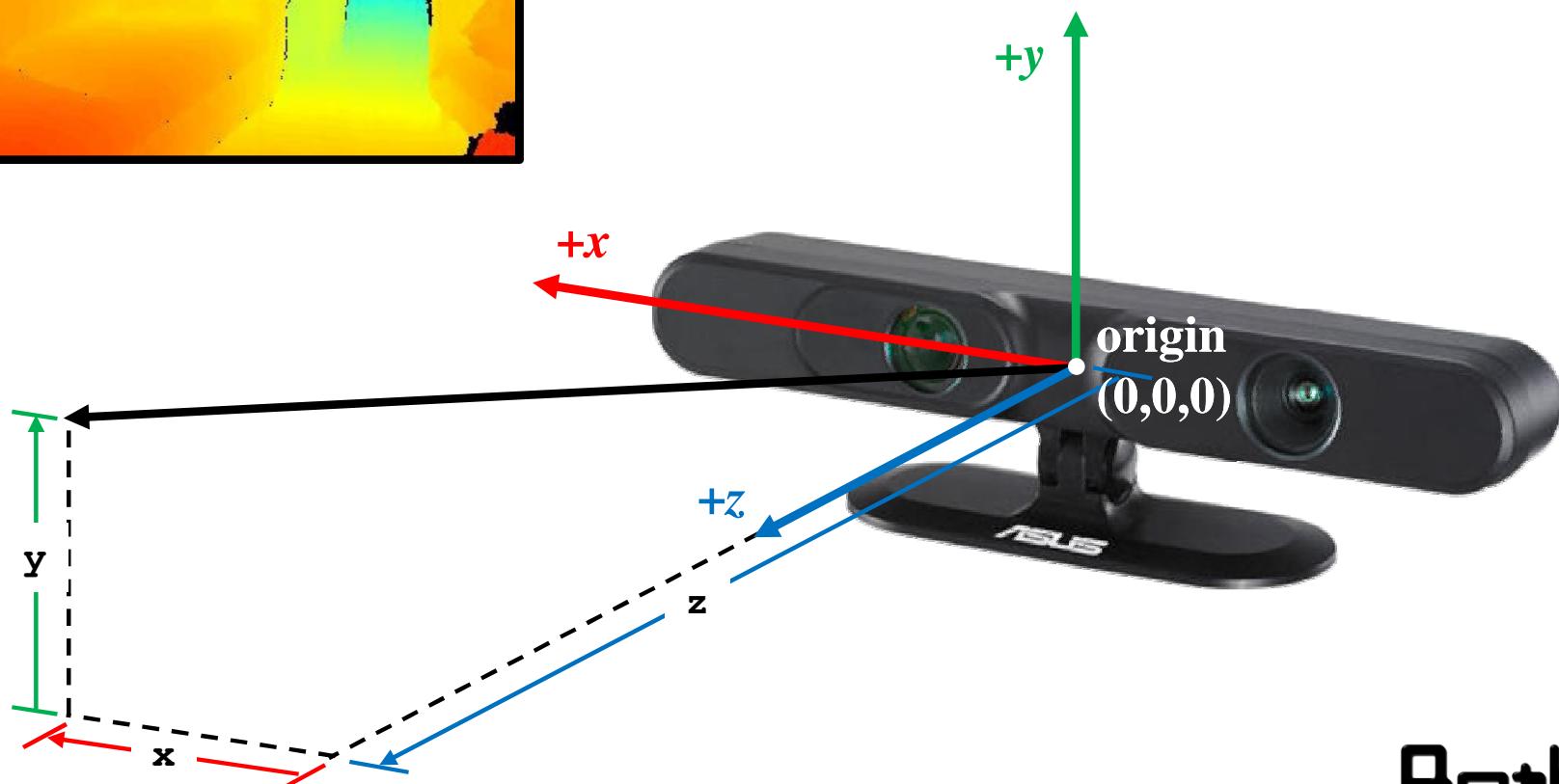


# World Coordinates:

What is the  $(x, y, z)$  of the pixel at  $(row, column)$ ?



```
int x = get_depth_world_point_x(row, column);
int y = get_depth_world_point_y(row, column);
int z = get_depth_world_point_z(row, column);
printf("(x, y, z) is (%d, %d, %d)\n", x, y, z);
```





# Demo

Finding the  $(x, y, z)$  location of a pixel (*row, column*).

```

int main()
{
    // Begin main function code block.
    // 1. Attempt to open a connection to the depth sensor.
    if (depth_open() == 0) {
        printf("Failed to connect to depth sensor\n");
        return 1;
    }

    // 2. Variables for the (x, y, z) location of the center pixel (120, 160) in the depth image.
    int x = 0, y = 0, z = 0, row = 120, column = 160; // Note: change (row, column) to evaluate a different pixel.

    // 3. Until the B-button is pressed, use the depth sensor.
    while (b_button() == 0) {
        depth_update(); // Update the depth image.

        // If the depth reading at the center pixel (120, 160) is 0, then it is not valid.
        if (get_depth_value(row, column) == 0) {
            printf("Depth reading is not valid\n");
        }
        else { // Print the (x, y, z) location of the center pixel (120, 160) in the depth image.
            // Note: this code assumes the *default* depth image resolution of 320x240.
            x = get_depth_world_point_x(row, column);
            y = get_depth_world_point_y(row, column);
            z = get_depth_world_point_z(row, column);
            printf("[%d, %d] is at (%d, %d, %d) mm\n", row, column, x, y, z);
        }
        msleep(100);
    }

    // 4. Close the connection to the depth sensor.
    depth_close();

    // 5. Tell the user program has finished.
    printf("Done!\n");

    // 6. Exit the program by returning 0.
    return 0;
} // End main function code block.

```



# Activity 8b: Description

World coordinate of the point with the minimum depth value.

- Plug the depth sensor into your KIPR Link.
- Copy the example program in the previous slide and download it to your KIPR Link.
- Move an object in front of the center of the depth sensor to get a feel for the **minimum** and **maximum** depths (**500mm** and **5000mm**, respectively).
- Change the **row** and **column** variables in the program to get a feel for where these locations are in real world coordinates.



# Activity 8b: Extensions

World coordinate of the point with the minimum depth value.

- Place an object in the center and 1 meter away from the depth sensor. Look at the depth image using the *Depth* screen under the *Motors and Sensors* menu. Does it look like the object is in the center of the depth image?
- Determine if the point with the minimum depth value is on the left or right of the depth sensor.
- Calculate the horizontal angle (or all angles) of the point with the minimum depth value.
- Mount the depth sensor to the DemoBot and program it to make it follow you within a certain **distance** and **angle**!



# Activity 8b: Solution

World coordinate of the point with the minimum depth value.

```

int main()
{
    // Begin main function code block.
    if (depth_open() == 0) { // Attempt to open a connection to the depth sensor.
        printf("Failed to connect to depth sensor\n");
        return 1;
    }
    int row = 0, curr_col = 0, min_col = 0, min_depth = 5000, curr_depth = 0, x = 0, y = 0, z = 0;
    while (b_button() == 0) { // Until the B-button is pressed, use the depth sensor.
        depth_update(); // Update the depth image.
        curr_col = 0; min_col = 0; min_depth = 5000; // Reset the variables.
        while (curr_col < get_depth_image_width()) {
            curr_depth = get_depth_value(row, curr_col);
            if ((curr_depth != 0) && (curr_depth < min_depth)) {
                min_col = curr_col; min_depth = curr_depth; // Set minimum depth info.
            }
            curr_col = curr_col + 1; // Increment the current column.
        }
        // If the minimum depth is still 5000, then all points evaluated in the above code were not valid.
        if (min_depth == 5000) {
            printf("No valid readings\n");
        }
        else { // Print the world coordinate with the minimum depth value (in millimeters).
            x = get_depth_world_point_x(row, min_col);
            y = get_depth_world_point_y(row, min_col);
            z = get_depth_world_point_z(row, min_col);
            printf("Nearest point at (%d, %d, %d) mm\n", x, y, z);
        }
        msleep(100);
    }
    depth_close(); // Close the connection to the depth sensor.
    printf("Done!\n"); // Tell the user program has finished.
    return 0; // Exit the program by returning 0.
} // End main function code block.

```



# Activity 8b: Reflections

World coordinate of the point with the minimum depth value.

- How does the location of the object in front of the depth sensor affect its depth readings and world coordinates?
- How does the angle of the depth sensor itself affect its depth readings and world coordinates?
- How stable are depth readings and world coordinates as you move around the object or depth sensor?
- If an object is in the center of the depth sensor, is it in the center of the depth image?
  - Hint: check using the *Depth* screen under the *Motors and Sensors* menu
- Where are the boundaries of the depth sensor in the real world?

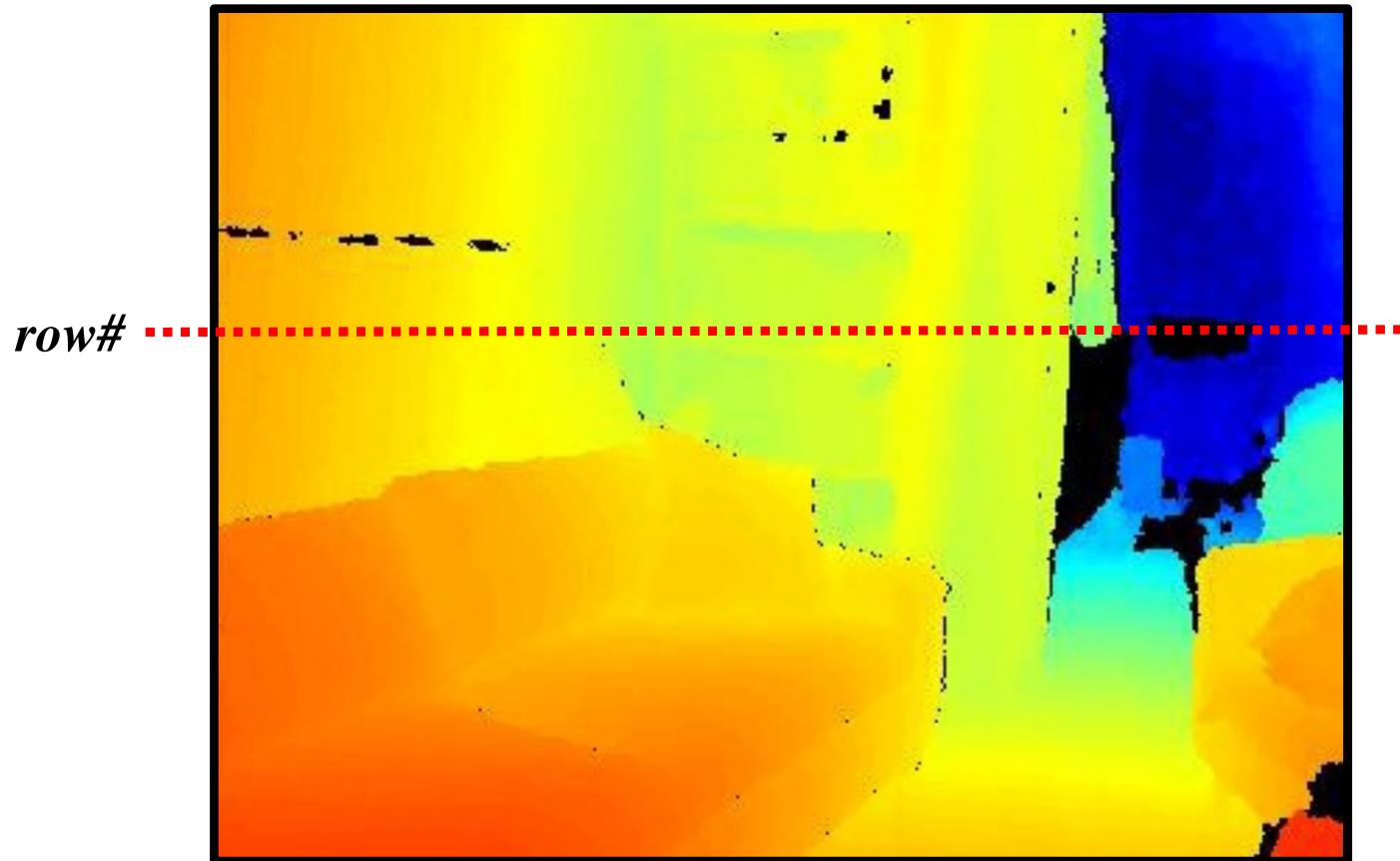


# Scanlines

- Review:
  - Specifying (**row, column**) allows us to access the depth value of a **single pixel** in the depth image.
  - Specifying (**row, column**) also allows us to access the  $(x, y, z)$  coordinate of a **single pixel** in the depth image in the real world.
- What if we want to work with more than just a **single pixel** in the depth image? What if we want to work with an **entire row of pixels** in the depth image?
  - This **entire row of pixels** is called a “**scanline**”.



# Scanlines



- The function **depth\_scanline\_update (<row#>)** gets scanline data for the selected row
  - **row#** is the row in the depth image to be used as the scanline.
  - is usually called shortly after calling **depth\_update ()**.
  - generates “**scanline objects**” and data about them.

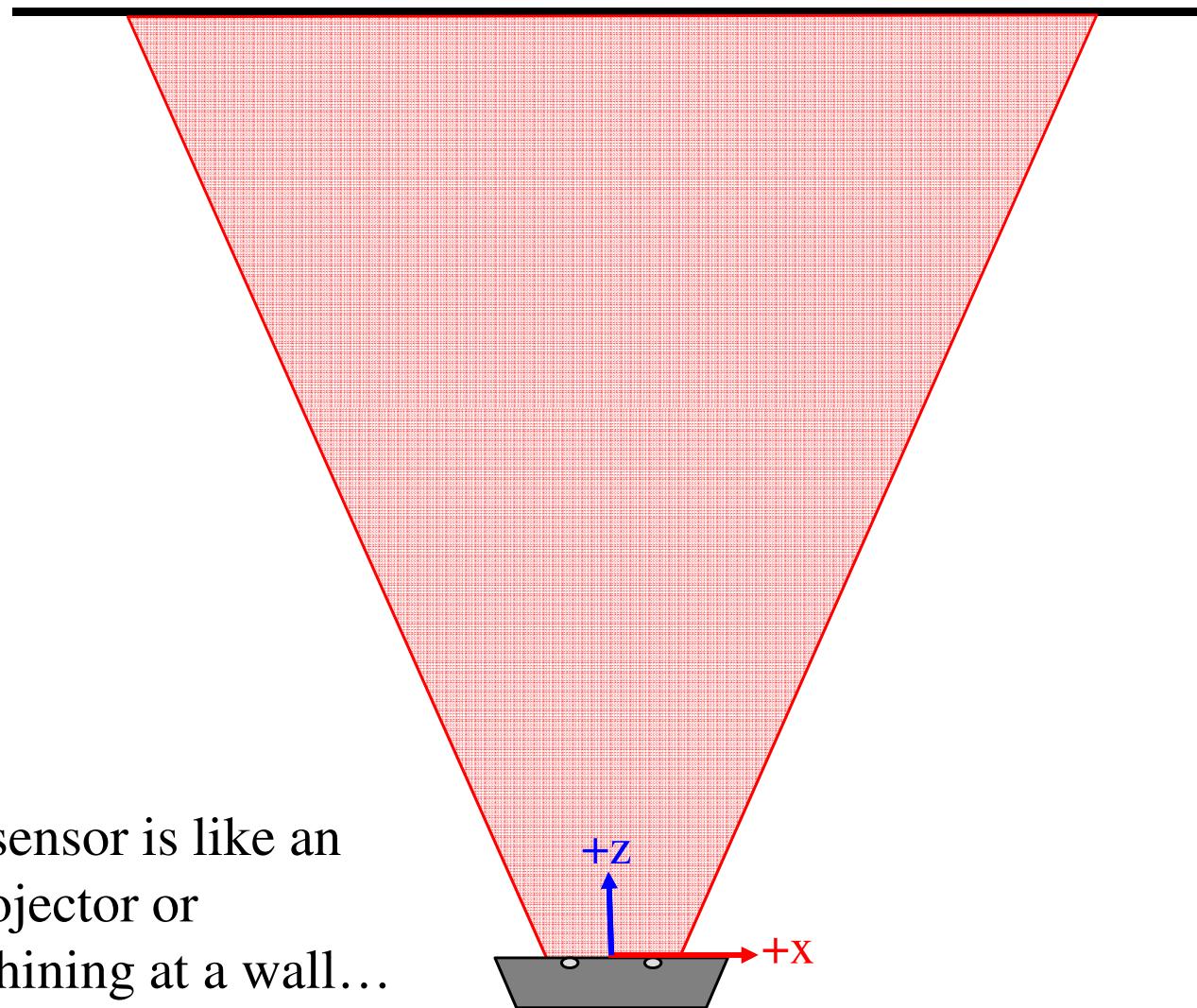


# Scanline Objects

- A “**scanline object**” is a segment of pixels on the **scanline** that are associated with one another based on having nearby depth values.
- The number of scanline objects detected is returned by the function **get\_depth\_scanline\_object\_count ()**.
- The following slides provide an illustration of how scanline objects are determined...



# Scanline Objects



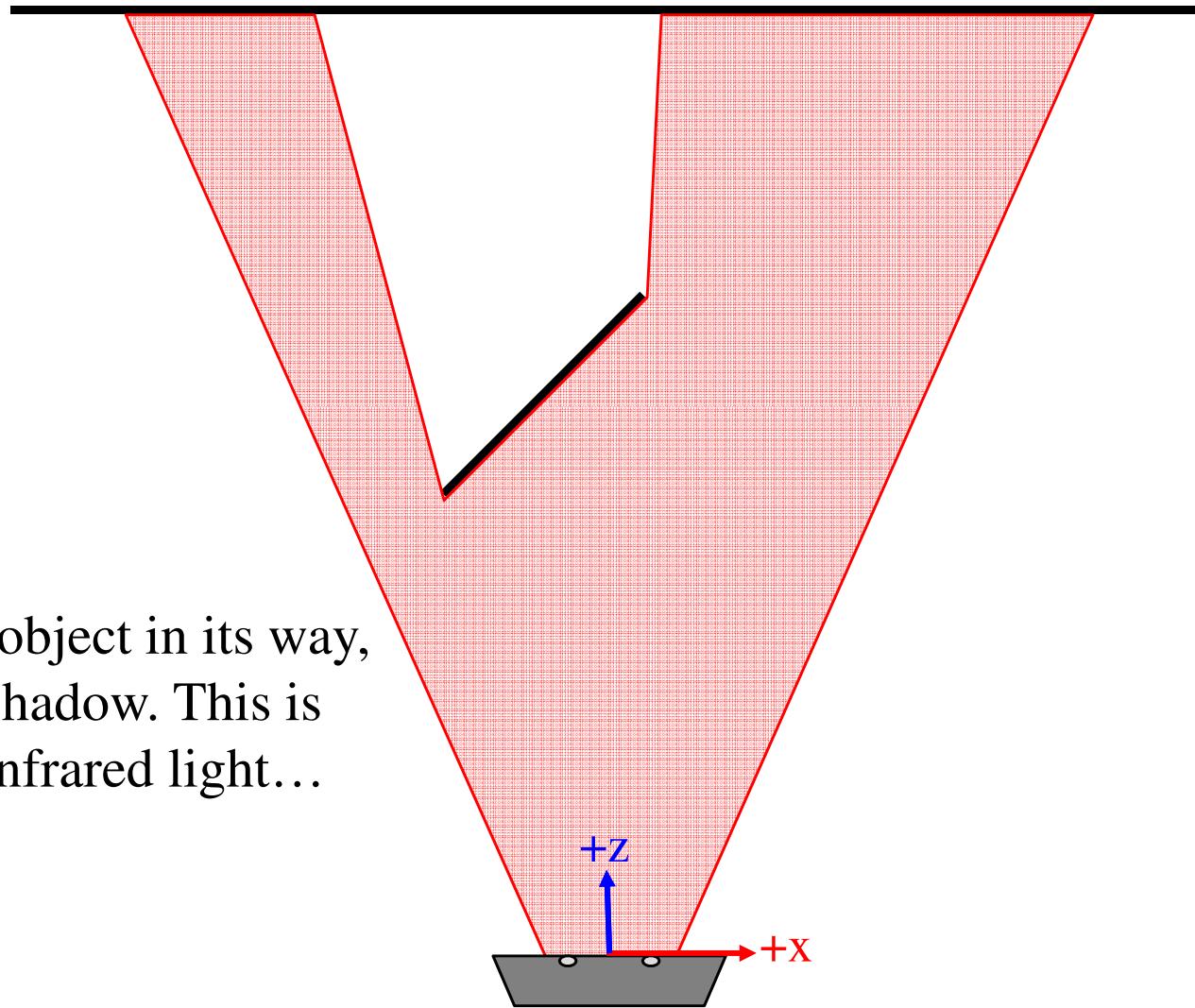
The depth sensor is like an infrared projector or flashlight shining at a wall...

Top-down view



# Scanline Objects

If you put an object in its way, it will cast a shadow. This is also true for infrared light...

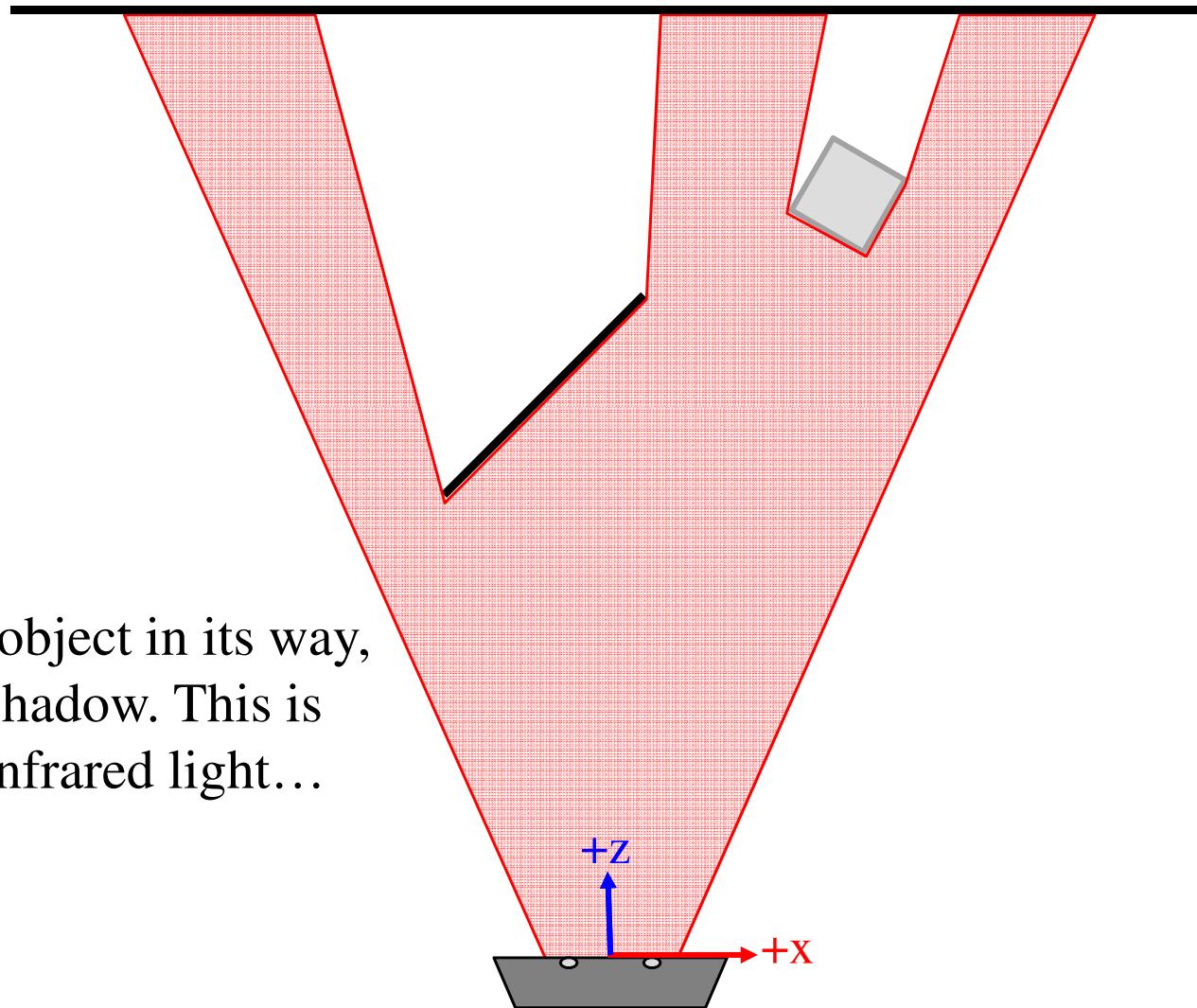


Top-down view



# Scanline Objects

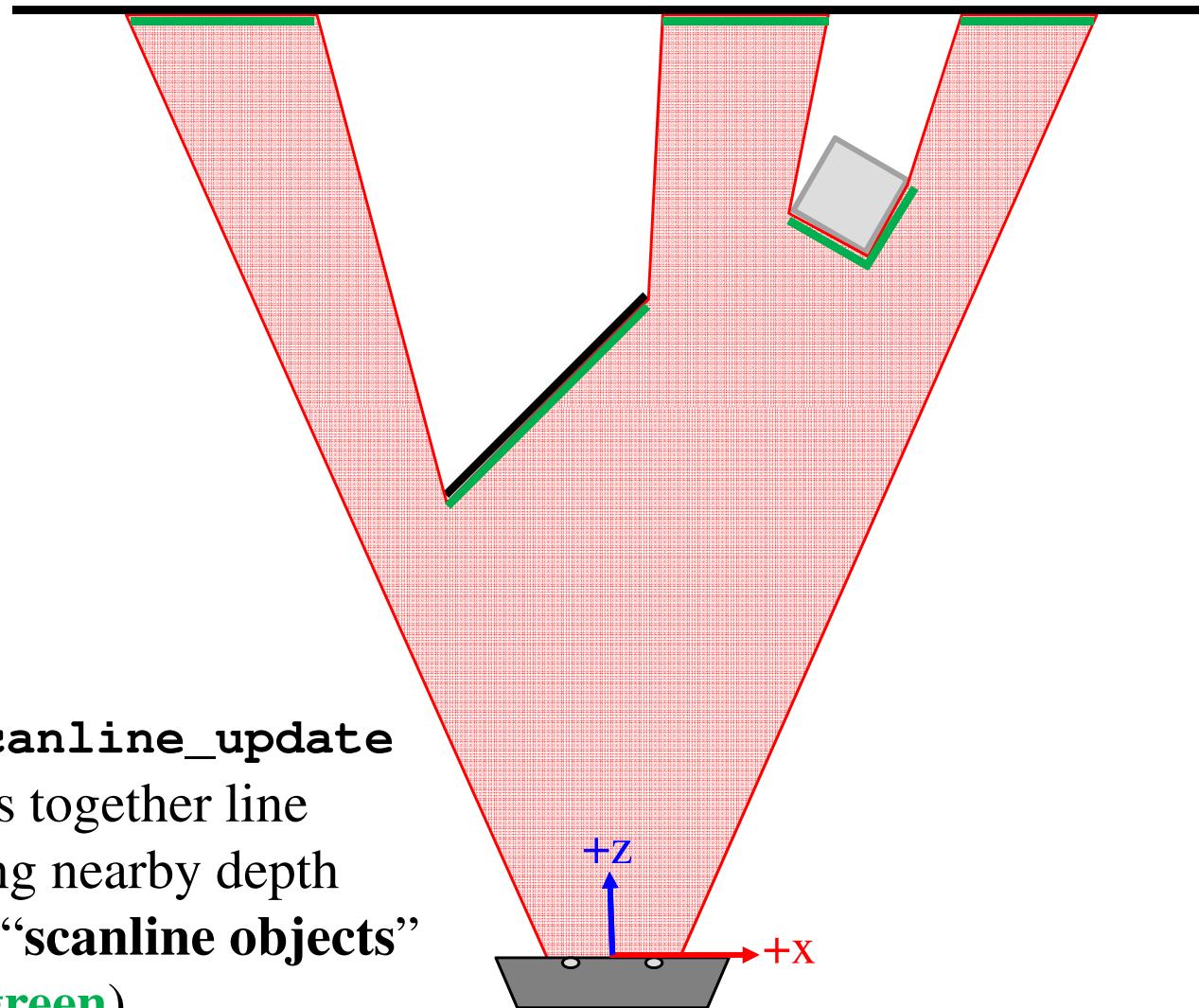
If you put an object in its way,  
it will cast a shadow. This is  
also true for infrared light...



Top-down view



# Scanline Objects



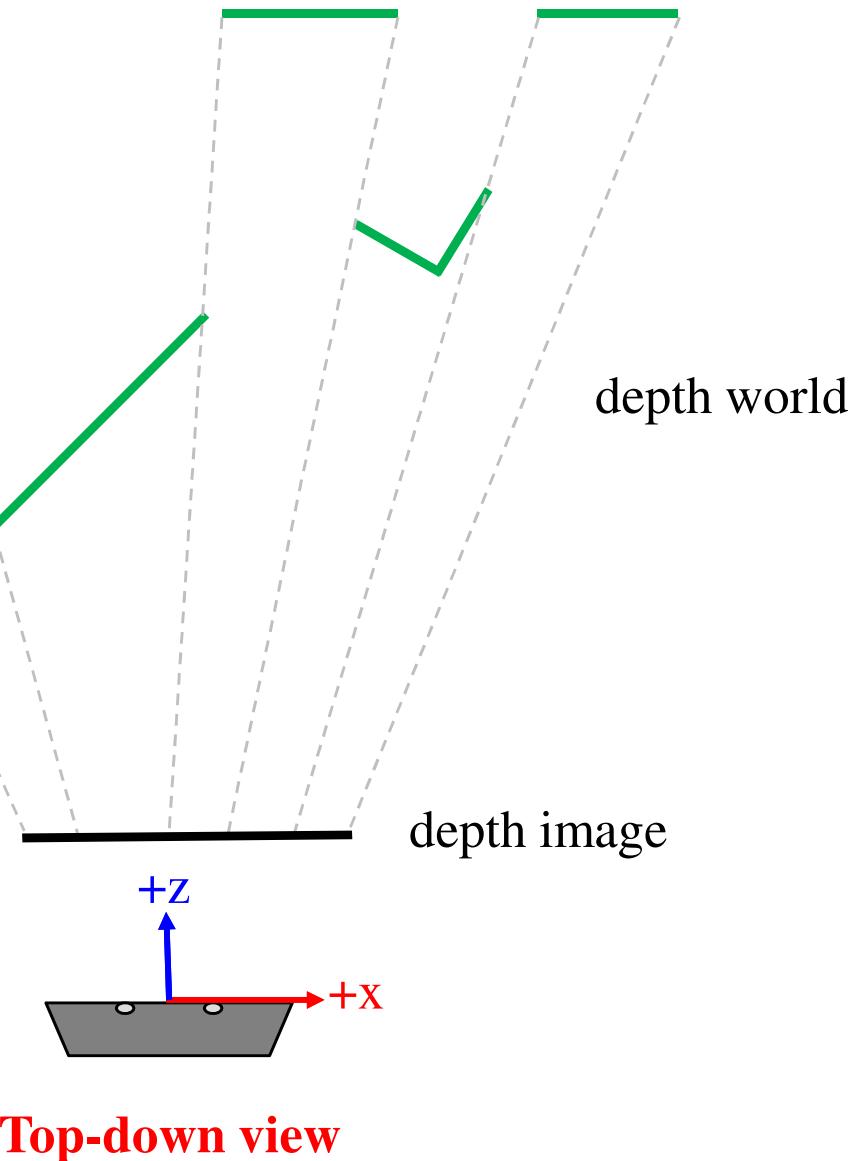
The `depth_scanline_update` function groups together line segments having nearby depth values to form “scanline objects” (illustrated in **green**)...

Top-down view



# Scanline Objects

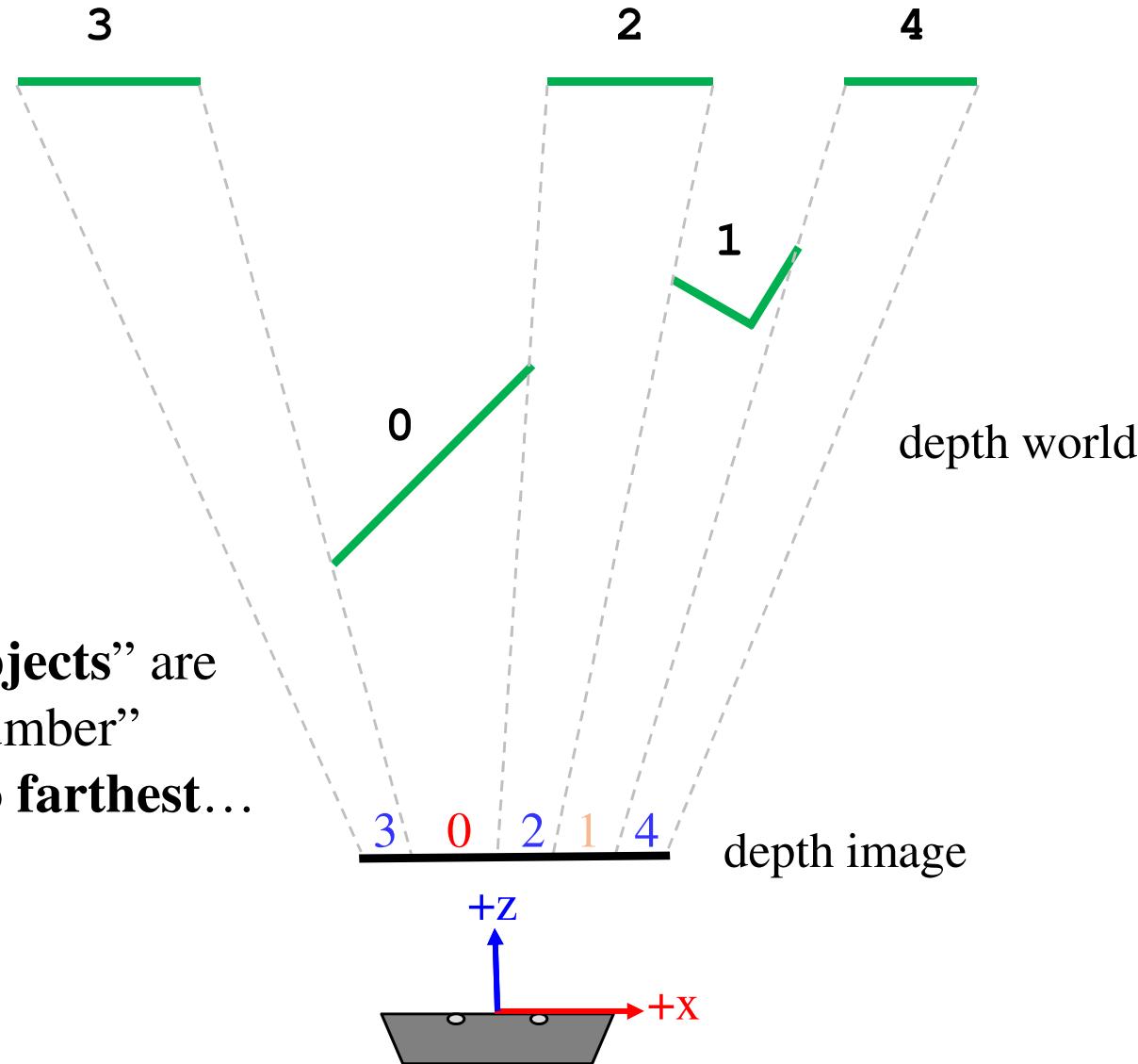
The `depth_scanline_update` function groups together line segments having nearby depth values to form “scanline objects” (illustrated in green)...





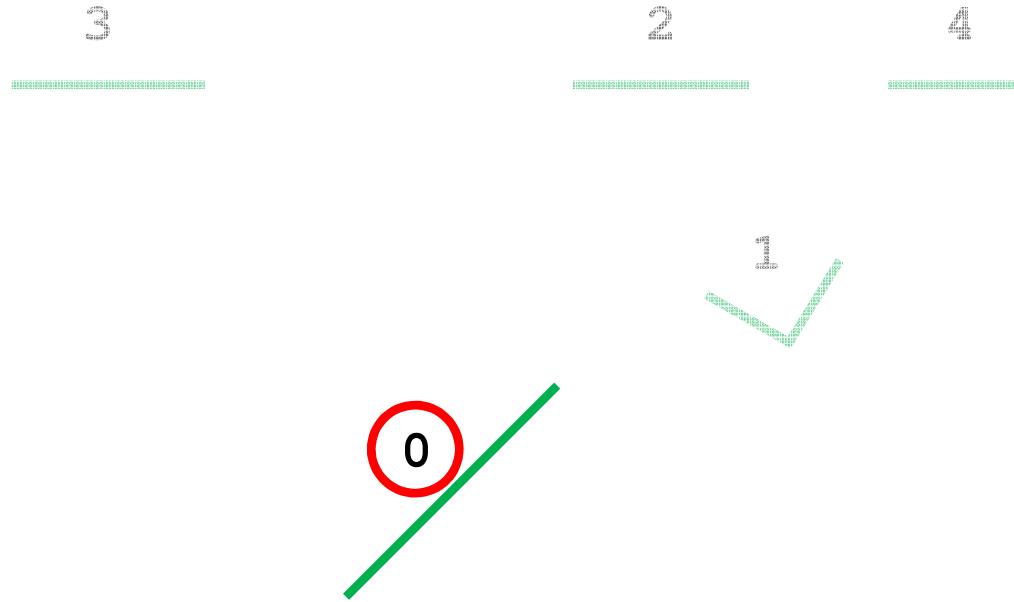
# Scanline Objects

These “**scanline objects**” are given an “object number” assigned **nearest to farthest**...

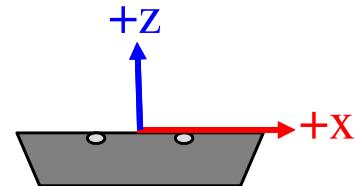




# Scanline Objects



Each “**scanline object**” has properties that can be accessed using a function of the format  
**`get_depth_scanline_object_...(<obj#>)`**.  
 Let’s look at object #0 now...

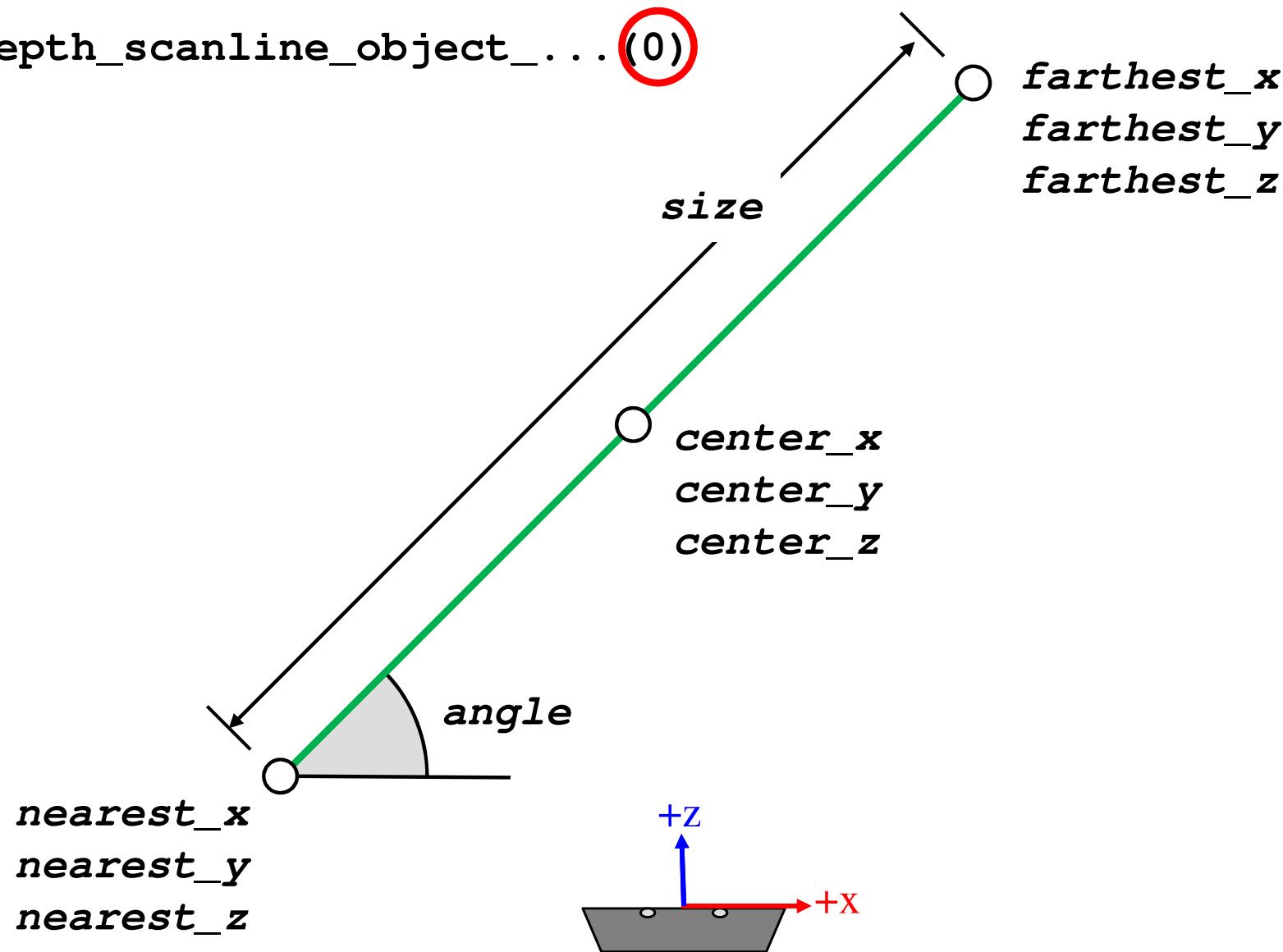


**Top-down view**



# Scanline Objects

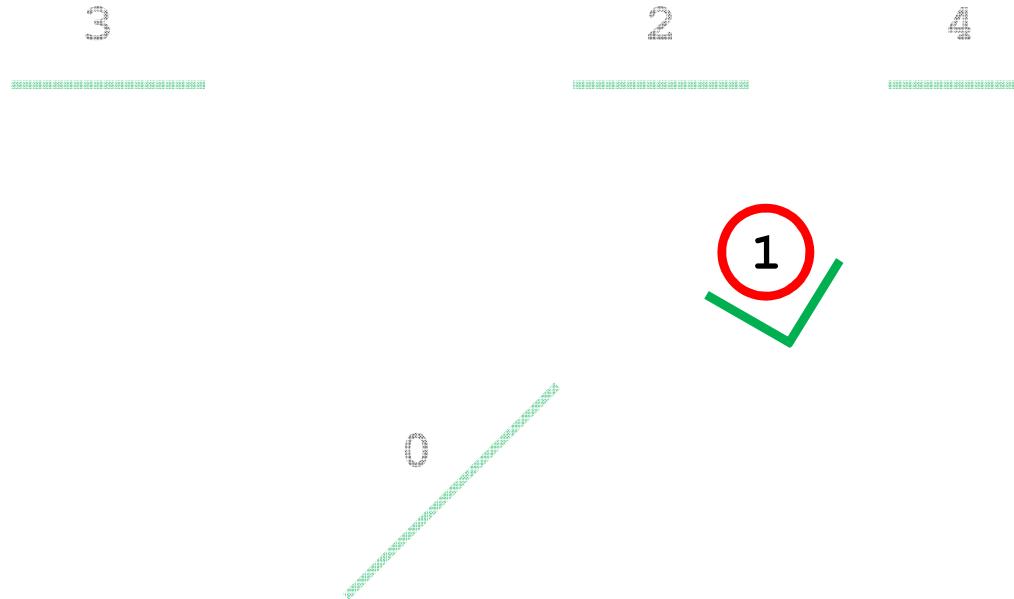
`get_depth_scanline_object_...()`



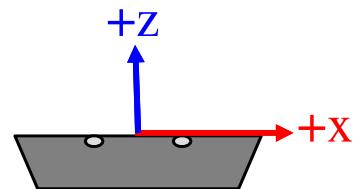
Top-down view



# Scanline Objects



We just looked at object #0.  
**Let's look at object #1 now.**

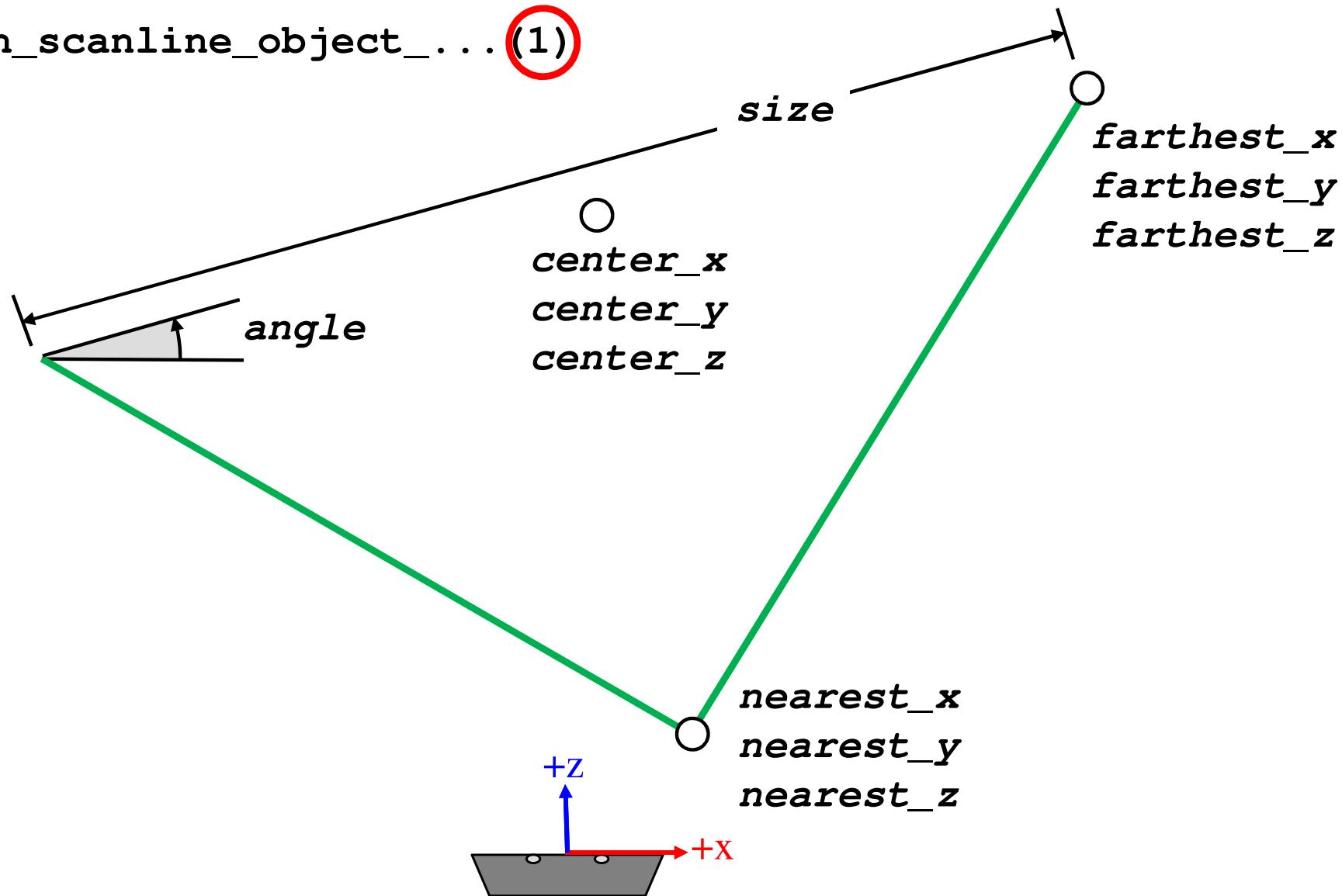


**Top-down view**



# Scanline Objects

get\_depth\_scanline\_object\_... (1)



Top-down view



# Scanline Object Functions

## `get_depth_scanline_object_...`

- Return value for `get_depth_scanline_object_suffix`
  - For *suffix*
    - `count()` is the number of scanline objects being tracked
    - `size(<obj>)` is the size of the scanline object (distance from leftmost to rightmost point)
    - `angle(<obj>)` is the angle (in `degrees`) of the scanline object (between leftmost and rightmost point) – a positive angle is counter clockwise
    - `center_x(<obj>)`, `center_y(<obj>)`, `center_z(<obj>)` are the respective world coordinates (x, y, or z) of the scanline object's *center point*
    - `nearest_x(<obj>)`, `nearest_y(<obj>)`, `nearest_z(<obj>)` are the respective world coordinates (x, y, or z) of the scanline object's *nearest point*
    - `furthest_x(<obj>)`, `furthest_y(<obj>)`, `furthest_z(<obj>)` are the respective world coordinates (x, y, or z) of the scanline object's *furthest point*



# Demo

## Display properties of the nearest object in a scanline

```

int main()
{
    // Begin main function code block.
    // Attempt to open a connection to the depth sensor.
    if (depth_open() == 0) {
        printf("Failed to connect to depth sensor\n");
        return 1;
    }

    // Variables for the properties of the center world coordinate in the scanline.
    // Note: this code assumes the *default* depth image resolution of 320x240.
    int x = 0, y = 0, z = 0, size = 0, angle = 0, row = 120; // Note: change row to evaluate a different scanline.

    // Until the B-button is pressed, use the depth sensor.
    while (b_button() == 0) {
        depth_update(); // Update the depth image.
        depth_scanline_update(row); // Update the depth scanline.
        display_clear(); // Clear the screen.
        // If there are no scanline objects detected, print a message.
        if (get_depth_scanline_object_count() == 0) {
            printf("No scanline objects found\n");
        }
        else { // Print the (x, y, z) location of the center pixel (120, 160) in the depth image.
            x = get_depth_scanline_object_center_x(0);
            y = get_depth_scanline_object_center_y(0);
            z = get_depth_scanline_object_center_z(0);
            size = get_depth_scanline_object_size(0);
            angle = get_depth_scanline_object_angle(0);
            printf("Object center at (%d, %d, %d) mm\n", x, y, z);
            printf("Size is %d mm\n", size);
            printf("Angle is %d degrees\n", angle);
        }
        msleep(100);
    }
    depth_close(); // Close the connection to the depth sensor.
    printf("Done!\n"); // Tell the user program has finished.
    return 0; // Exit the program by returning 0.
} // End main function code block.

```



# Activity 8c: Description

Playing with scanlines

- Plug the depth sensor into your KIPR Link.
- Copy the example program in the previous slide and download it to your KIPR Link.
- Observe how each of the values changes as the angle of the depth sensor and angle of the scanline object change.



# Activity 8c: Extensions

## Playing with scanlines

- Identify the coordinates of a cube...
  - Hint: this would be very useful for finding PT cubes in this year's game
- Make the Demobot center on a scanline object and approach it as near as possible without touching it...
  - Hint: use the *center* point ( $x, y$ ) coordinates, but the *nearest* point  $z$  coordinate
- Make the DemoBot “square up” with a wall in front of it...
  - Hint: use the **get\_depth\_scanline\_object\_angle** function.
- Make the DemoBot follow a wall next to it 1 meter away...
  - Hint: use the **get\_depth\_scanline\_object\_angle** function.



# Activity 8c: Reflections

Playing with scanlines

- How does the location of a scanline object in front of the depth sensor affect its properties?
- How does the angle of the depth sensor itself affect the properties of a scanline object?
- How stable are scanline object readings as you move around the object or depth sensor?
- Does the **size** of the scanline object change at different distances or orientations?



# Depth Sensor Recap

- The depth sensor is the **Asus® Xtion PRO**.
  - We thank **Asus®** for their donation of these sensors for Botball 2014!
- To open and close the depth sensor, use `depth_open()` and `depth_close()`, respectively.
- To get new depth sensor information, use `depth_update()`.
- There are three ways to use the depth sensor:
  1. **Depth value:** `get_depth_value(<row>, <column>)`
  2. **World coordinates:**  
`get_depth_world_point_x(<row>, <column>)`  
`get_depth_world_point_y(<row>, <column>)`  
`get_depth_world_point_z(<row>, <column>)`
  3. **Scanlines:**  
`depth_scanline_update(<row>)` gets new scanline information  
`get_depth_scanline_object...(<obj>)` accesses scanline object properties



# Depth Sensor Recap

- Remember the **minimum** and **maximum** depth values for the depth sensor (**500mm** and **5000mm**, respectively)...
  - Consider placing the depth sensor on the back of the robot (rather than the front) to maximize its effective range.
  - The ET infrared rangefinder could be used for close-range sensing (between **approx. 100mm** and **750mm**).
- The depth sensor **emits infrared light**...
  - **Thus, it interferes with other sensors that detect light!**
  - Consider **shielding** your light sensors, as well as your reflectance, top hat, and ET infrared rangefinder sensors.
  - It also interferes with **other depth sensors!** However, we have determined that **the interference between depth sensors is negligible**.
- The depth sensor **uses a lot of power** from the KIPR Link...
  - Thus, the **KIPR Link battery will drain faster** when the depth sensor is plugged in—**even more so when in heavy use!**
  - **We recommend testing with the KIPR Link plugged into its charger!**

# Lunch Time!





# Welcome back!

Can you find Botguy and the PT Cubes?  
Play with the Xtion!





# Vision



# Prep

## Vision

- Vision setup
- HSV color selection and color blobs
- Training the Link to use an HSV color model
- Library functions for using the camera

<code>camera_open()</code> <code>camera_close()</code> <code>camera_update()</code> <code>get_object_count(&lt;ch&gt;)</code>	<code>get_object_bbox_...(&lt;ch&gt;, &lt;obj&gt;)</code> <code>get_object_center_...(&lt;ch&gt;, &lt;obj&gt;)</code> <code>camera_load_config(&lt;name&gt;.conf)</code>
--	--

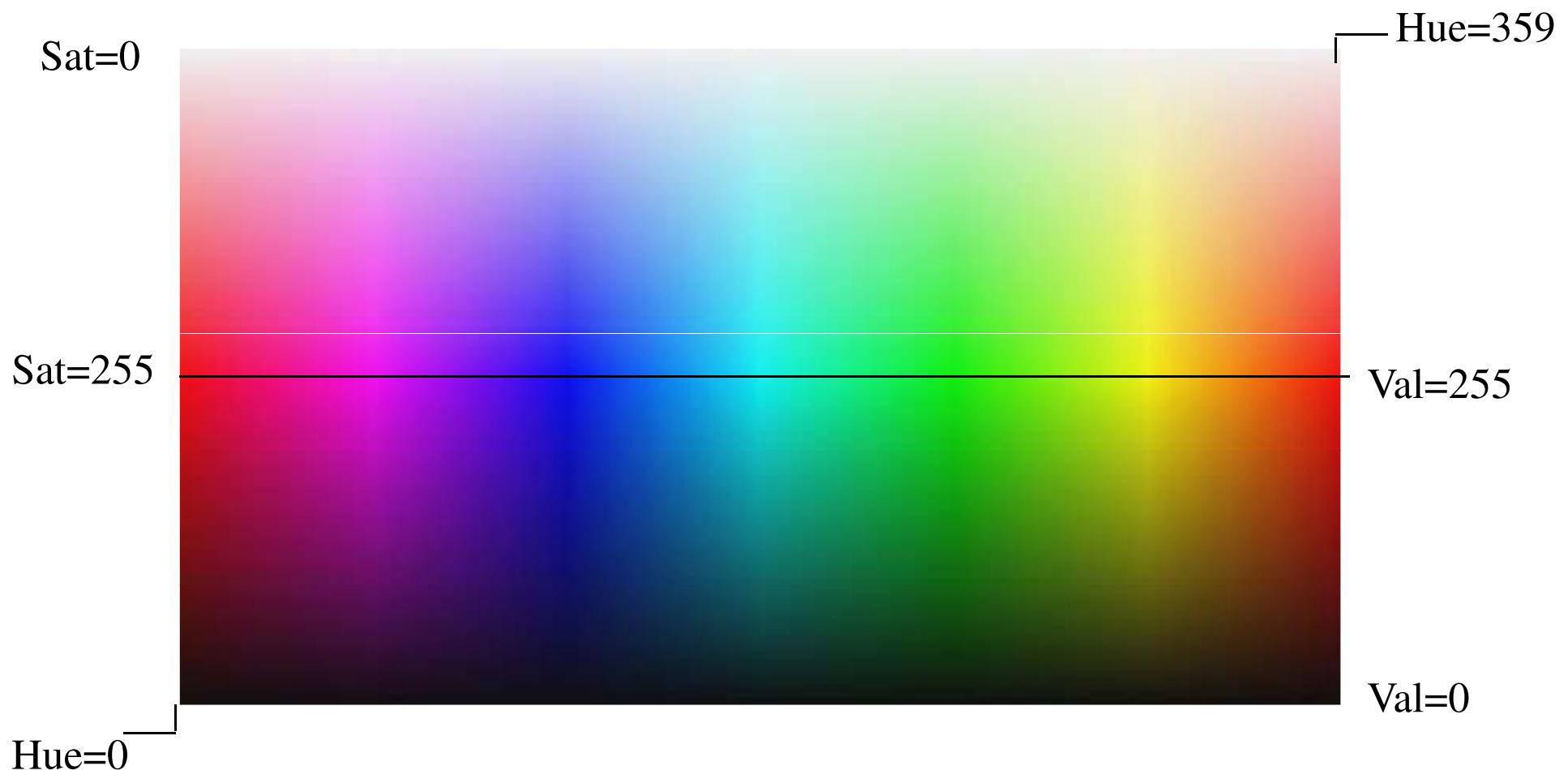


# Vision Setup

- The USB cameras in the Botball kit will work in either of the Link's USB ports
- Plug in a camera and you will be able to see the camera image by going to the *Camera* screen under the *Motors and Sensors* menu
  - If you unplug the camera, the Link may no longer recognize it if you plug it back in
    - You will need to restart the Link if this happens

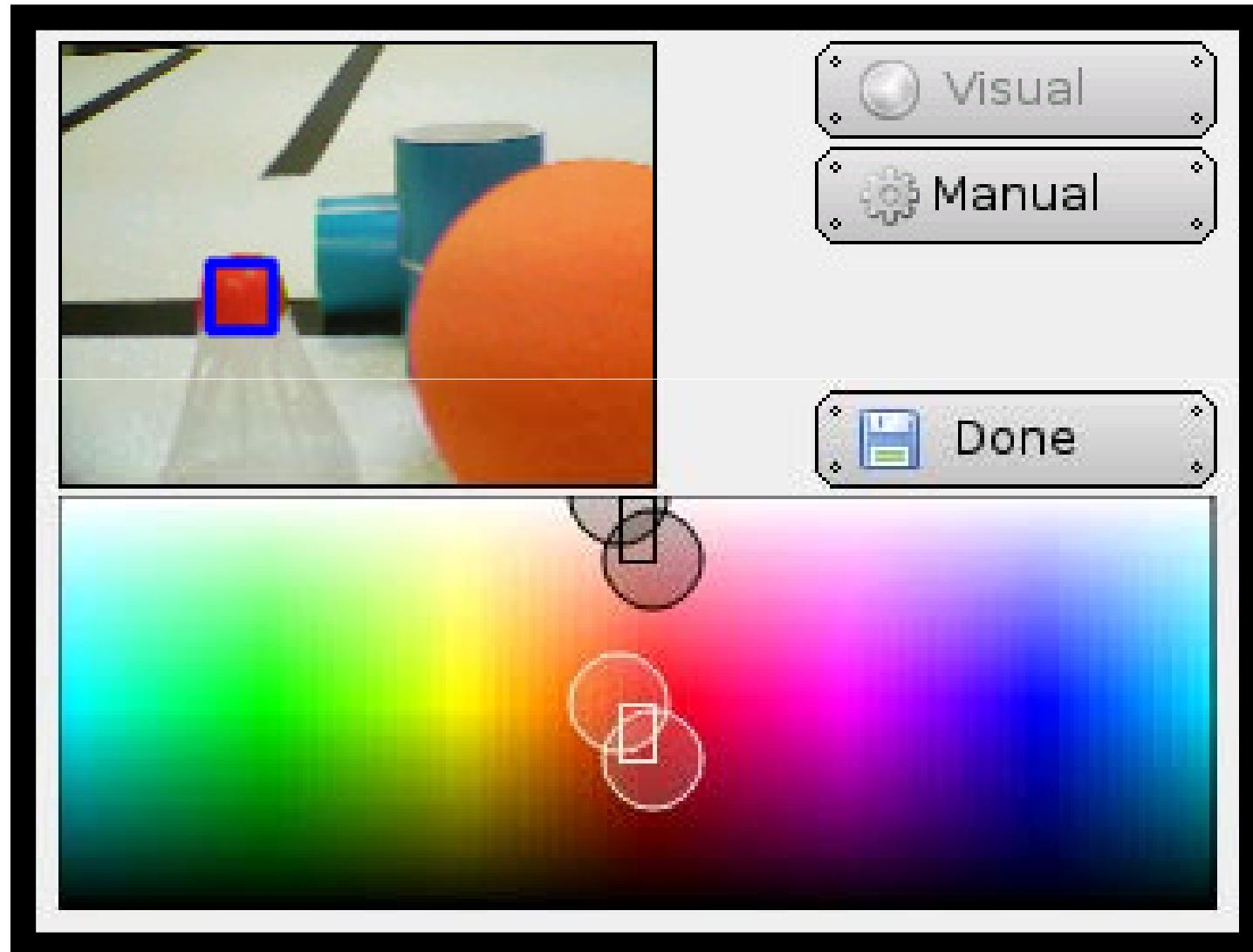


# HSV Color Selection Plane





# Channels Interface



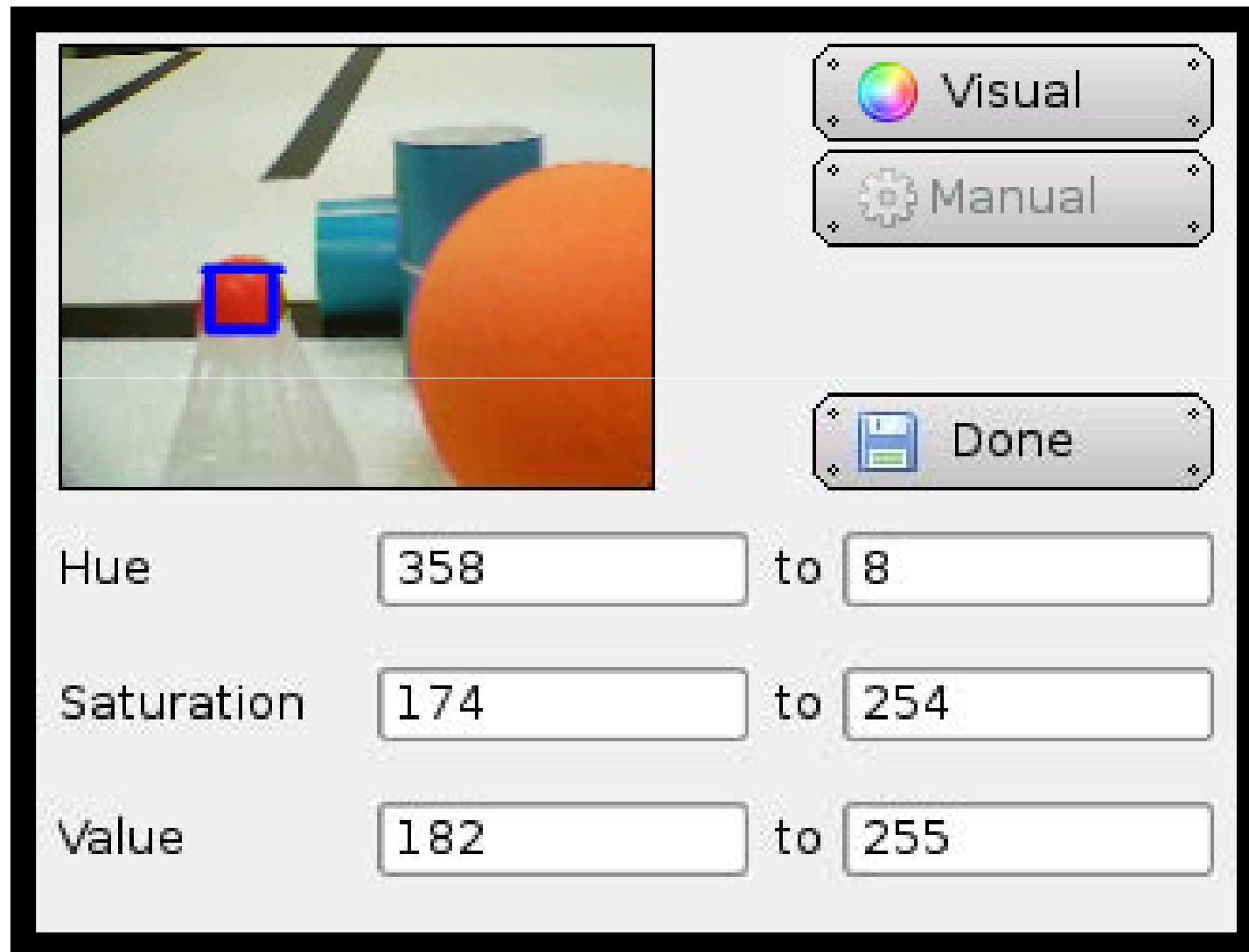


# Color Blobs

- Each pixel on the screen has an HSV color
- When we say "red", we really mean a range of HSV colors on the color selection plane that are approximately red
- Two rectangular pieces of the color selection plane that correspond to being "red" specify the range of HSV colors to be viewed as red by the KIPR Link
  - This is called an HSV color model
- A red *blob* is all contiguous pixels matching one of the HSV colors in the red range
- A blob has a bounding box, a center, etc.
- If you want to find Botguy with the camera, you look for a big red blob



# Manual Channel Interface





# Demo/Video of Setting Color Models

<http://youtu.be/nSszFa7opMA>



# Performance Factors

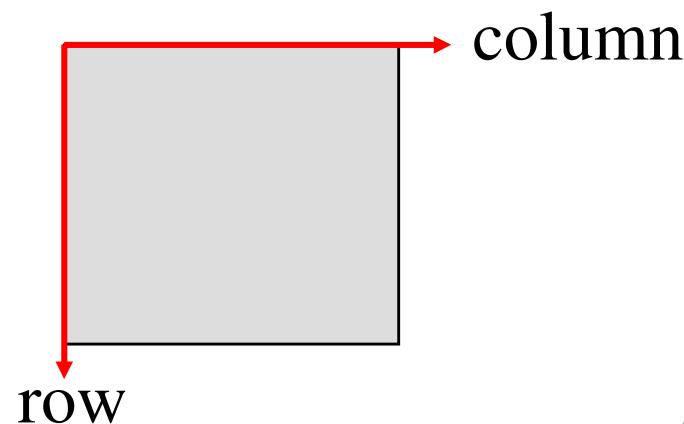
- Focus:
  - A slightly blurred image smooths out colors and can improve some tracking reliability
  - Sharp focus is important for separating adjoining blobs
  - Adjust focus by turning the focus ring on the camera
- Image Resolution:
  - The lower the resolution the higher the frame rate
  - This is set by the argument given to `camera_open()`



# Image Coordinates

- The camera's processed field of view is treated as an (row, column) coordinate array
  - The upper left corner has coordinates (0,0)
  - The lower right corner has coordinates (119,159) by default
  - The Link display may distort the camera's field of view

What are the coordinates of the center?





# Vision System Color Models

- You can create multiple vision system configurations
  - Each configuration can have up to 4 channels
- **YOU MUST SET ONE CONFIGURATION AS THE DEFAULT**
  - The KIPR Link *Settings* screen is where you do this
- The KIPR Link can handle 4 Channels simultaneously
- Each channel can be either a HSV blob tracking channel or a QR code scanner channel
  - QR won't be discussed for this season, but are still supported



# Vision System Library Functions

- The KIPR Link library function **camera\_update()** ; is a command that causes the KIPR Link to capture the most recent camera frame for analysis
  - Frame analysis determines objects properties such as the (row, column) coordinates of the center of the object
- **get\_object\_count(3)** ; provides how many objects are being seen by channel 3 **in the default configuration**
  - If the count is 0 there are no objects; if -1 the channel does not exist
  - Objects are numbered 0,1,2, ... from largest to smallest
- **get\_object\_center\_column(3, 0)** ; for channel 3, object 0, returns the value of the center column coordinate of the largest object



# More Object Functions

## `get_object_suffix`

- Coordinate values (*row*, *column*) for an object's center or bounding box (upper left, or bottom right), or the object's width, height, and area; *suffix* is one of:
  - **center\_column** (*<ch>*, *<obj>*)
  - **center\_row** (*<ch>*, *<obj>*)
  - **bbox\_ulc** (*<ch>*, *<obj>*)
  - **bbox\_ulr** (*<ch>*, *<obj>*)
  - **bbox\_brc** (*<ch>*, *<obj>*)
  - **bbox\_brr** (*<ch>*, *<obj>*)
  - **bbox\_width** (*<ch>*, *<obj>*)
  - **bbox\_height** (*<ch>*, *<obj>*)
  - **area** (*<ch>*, *<obj>*)



# Selecting the Action to Perform

## if - else

- For **while**, an action is performed so long as the condition check is true
- In contrast, for **if - else** , one action is performed if the condition is true and another if it is false
- Example:

```
if (get_object_count(0) > 0)
    { printf("There's a red blob\n"); }
else
    { printf("Don't see a red blob\n"); }
```

- The **if** control structure is a special case of **if - else**



# Example Using Vision Functions

```

// Set up a camera configuration that is calibrated so that it recognizes
// a red colored object for color channel 0 before running the program
// and make sure that configuration is the default
int main()
{
    // Start up the camera and specify the resolution
    int col, row, color = 0; // set up for color channel 0 (red)
    camera_open();
    printf("Looking for red\nPress A when ready\n\n");
    printf("Press B button to quit\n");
    while (a_button() == 0) { // wait for A button
    }
    while (b_button() == 0) { // run till B button is pressed
        camera_update(); // process the most recent image
        if (get_object_count(color) > 0) {
            //get x, y for the biggest blob the channel sees
            col = get_object_center_column(color, 0);
            row = get_object_center_row(color, 0);
            printf("Biggest blob at (%i,%i)\n", row, col);
        }
        else {
            printf("No color match in Frame\n");
        }
        msleep(200); // give user time to read
    }
    printf("Program is done.\n");
    return 0;
}

```



# Activity 9a: Description

## Vision

- Plug the camera into your KIPR Link
- Calibrate your vision system so that color model 0 picks up nearby pink colored objects
- Copy the example program and download it to your KIPR Link
- Move a pink object around in front of the camera to get a feel for the boundaries of the camera's field of view
- Modify the program so it prints out whether the blob is to the right or to the left
- Change the color channel for the program and repeat using a different color



# Activity 9a: Reflections

## Vision

- How do lighting and shadows affect your color model?
  - Be sure to calibrate your camera under the same lighting in which it will be used
- How stable are blobs as you move an object or camera around while training?
- How close to the boundaries for columns (0-159) and rows (0-119) could you get the centroid reading for your object?



# Activity 9b: Description

## Vision Tracking

- Modify the line following program (Activity 7) to track the largest object on a vision channel
  - Code should be the same whether or not your channels is for a colored object
- Improvements:
  - Divide the vision field into three regions where if the object's center is to the left, turn left, if it is to the right, turn right and if it is in the center region, go straight forward
  - If no object detected on that channel is in view stop and wait



# Activity 9b: Solution

## Vision Tracking

```

/* Move the robot towards the largest object on channel 0.
   Robots stops if no object is detected*/
int main()
{
    int ch = 0, leftmtr = 0, rghtmtr = 3; // identify channel and motors
    int high = 100, low = -10;           // set wheel powers for arc radius
    camera_open();
    printf("Move towards object on channel 0\n");
    printf("Press B button when ready\n\nPress side button to stop\n");
    while(b_button() == 0) {           // wait for button press
    }
    while(side_button() == 0) {        // stop if button is pressed
        if(get_object_count(ch) > 0) { // if object is seen...
            if(get_object_center_column(ch,0) < 65) { // if object is on left...
                motor(leftmtr,low); motor(rghtmtr,high); // arc left
            }
            else {
                if(get_object_center_column(ch,0) > 95) { // if object is on right...
                    motor(rghtmtr,low); motor(leftmtr,high); // arc right
                }
                else {
                    motor(rghtmtr,high); motor(leftmtr,high); //go straight
                }
            }
        }
        else {
            ao();
        }
    }
    ao(); // stop because button pressed
    printf("done\n"); return 0;
}

```



# Additional Activities

- [Activity 10](#): Point Servo at Colored Object
- [Activity 11](#): Bang-Bang Control
- [Activity 12](#): Proportional Control
- [Activity 13](#): Bang Bang Control with DemoBot Arm
- [Activity 14](#): Proportional Control with DemoBot Arm
- [Activity 15](#): Accelerometer for Bump Detect
- [Activity 16](#): Graphics for Custom User Interfaces
- [Activity 17](#): Reduce Accumulated Errors
- [Activity 18](#): Reading QR Codes



# From Now Until Lunch

- Work on any earlier activities you haven't finished
- Try out one or two of the additional activities listed on the previous slide (activity details are located at the end of today's presentation)



# Remainder of the Day

- Continue working on Activities
- Take advantage of having experts in the room - ask them your technical questions
- Make contacts with other schools
  - Schedule joint practice sessions
  - Schedule mini tournaments
  - Set up joint fund raising activities
  - etc.



# Things for ALL Teams to Remember

1. Review "New for 2014" on your Team Home Base.
2. Review the BOPD Manual (Botball Online Project Documentation).
3. 2014 Documentation expects:
  - Online (3 submissions) and Onsite Presentation (you can't win without good documentation and a practiced presentation). A scored example is on your Team Home Base.
4. Side A and Side B for this year's game have different colors
  - Robots using colors should be designed and programmed for running on either A or B sides (KIPR Software will determine what side you will run on).
5. Teams are allowed to use at most 1 camera on their entry.
6. Use the manuals and "hints for new teams" on your Team Home Base.



# Suggestions for New teams

1. Read the "Hints for New Teams Manual" on your Team Home Base.
2. Hit the ground running (don't wait to get started).
3. It is okay to ask for help - use the team home base forums, community site and KIPR.
4. If possible, build a practice board (instructions are on your Team Home Base - this is a great parent/mentor/student activity).
5. Keep It Simple Students (start out with one task and do it well before adding tasks - a simple robot is easier to build, repair and program).
6. Don't forget the documentation - read and follow the rubrics.
7. Check out the "construction hints" pictures of drive trains, effectors and sensor mounts on your Team Home Base to help generate ideas.
8. Use the DemoBots as a good starting point and modify them as you go.
9. HAVE FUN!



# Wrap-up: Avoid Embarrassing Problems at the Tournament

- Test your robots **from start to end**:
  - Shield your starting light sensors.
  - Go through the entire starting sequence.
    - Calibrate your light sensor(s) to the starting light.
    - Make sure the robots stop when they are supposed to.
      - verify with a stop watch!
- Have a check list of what to bring.
  - Bring on-site documentation materials.
  - Make backups of software.
  - Bring backups of software.
  - Bring a power strip, laptop power supply, and chargers for Create and KIPR Links.



Check

[www.botball.org](http://www.botball.org)

and your Team

Home Base

*regularly*

Good Luck!

**Botball®**



# Additional Activities

- [Activity 10](#): Point Servo at Colored Object
- [Activity 11](#): Bang-Bang Control
- [Activity 12](#): Proportional Control
- [Activity 13](#): Bang Bang Control with DemoBot Arm
- [Activity 14](#): Proportional Control with DemoBot Arm
- [Activity 15](#): Accelerometer for Bump Detect
- [Activity 16](#): Graphics for Custom User Interfaces
- [Activity 17](#): Reduce Accumulated Errors
- [Activity 18](#): Reading QR Codes



# Point Servo at Colored Object



# Activity 10: Prep

## Point Servo at Colored Object

- Calibrate your vision system so channel 0 matches for a colored object
- Remember from the vision activities how the vision coordinate system works and download the program below
- Move the object around in front of the camera to get a feel for the boundaries of the camera's field of view

```

int main()
{
    // Activity 9 prep
    int row, col;
    camera_open();
    printf("Looking for blob\n\nPress side button to quit\n");
    while(side_button() == 0) { //run until side is pressed
        camera_update(); // process the most recent image
        if(get_object_count(1) > 0) { // any blobs of the trained color?
            col = get_object_center_column(1,0); row = get_object_center_row(1,0);
            // store row, col of biggest blob
            printf("Color Blob at (%i,%i)\n", row, col); msleep(200);
        }
        else {
            printf("No Blob in Frame\n");
        }
    }
    printf("Program is done.\n");
    return 0;
}

```



# Activity 10

- Write a program to have the arm on DemoBot keep pointing at an object moved up and down in front of the camera (hint: use the y position of the blob as a factor in your position determination)
- The camera has a vertical angle of view of approximately 60 degrees, or 1/3 of the range of motion of a servo
  - Servo range is 180 degrees -> 0-2023 servo-tics
  - y-coordinate covers 60 degrees from 0-119 pixels
  - Each pixel represents about 6 servo-tics



# Activity 10: Solution

```

#define ARMPORT 0

int main()
{
    int offset=662, yFactor=6, x, y, sPos;
    enable_servos(); // servos powered on
    while(side_button() == 0) {
        camera_update(); // get most recent camera image and process it
        if(get_object_count(0) > 0) { // there is a blob
            x = get_object_center(0,0).x; // x coordinate
            y = get_object_center(0,0).y; // and y
            display_printf(0,4,"Blob is at (%i,%i)\n",x,y);
            set_servo_position(ARMPORT, offset + 6*y); // assumes center position of servo...
            // is aligned with vertical center of camera
        }
        else {
            display_printf(0,4,"No colored object in sight\n");
        }
        msleep(200); // don't rush print statement update
    }
    disable_servos(); // servos powered off
    printf("All done\n");
    return 0;
}

```



# Bang-Bang Control



# Bang-Bang Control

- Bang-bang control, as its name implies, is a control strategy that changes power to a new value without a transition such as first slowing down
  - The effect is like bumper car bouncing back and forth between two walls; i.e., you slam into reverse when you hit one wall (bang) and then slam into forward when you hit the other (bang), never slowing down to soften the blow
  - Activity 7, line following, used bang-bang with the robot turning either hard left or hard right. So did 8c



# Activity 11: Objectives

## Bang-Bang Control

Write a program that monitors the floating analog "ET" sensor (Day 1, slide ~163) to keep the robot a certain distance away from a moving obstacle using bang-bang control.

Run the program on the KIPR Link  
using the DemoBot





# Activity 11: Pseudocode

## Bang-Bang Control

1. Set analog port 0 to floating analog.
2. Loop until side button is pressed.
  - a. If the floating analog rangefinder on port 0 has a reading indicating less than about 6" then back up
  - b. Otherwise, drive forward
3. Stop the DemoBot when side button is pressed



# Activity 11: Solution

## Bang-Bang Control

```
// **** Bang Bang Control
int main()
{
    int distVal = 600; // change this value to be sensor reading at 6 inches
    // Step 1: Set analog port 0 to floating analog
    set_analog_pullup(0,0); // disable pullup on port 0 so port is "floating"
    printf("Back up if obstacle too close\n  otherwise go forward\n");
    printf("Press A button to start\n\n");
    while (a_button() == 0);
    printf("Press side button to stop\n");
    while (side_button() == 0) // Step 2: Loop until side button is pressed
    {
        // Step 2a: If the floating analog rangefinder on port 0 reads
        // greater than distVal, back up.
        if (analog(0) > distVal) { motor(0, -50); motor(2, -50); }
        // Step 2b: Otherwise, drive ahead
        else { motor(0, 50); motor(2, 50); }
    }
    // Step 3: Stop
    ao();
    printf("done\n");
    return 0;
}
```



# Activity 11: Experiments

## Bang-Bang Control

- What is the behavior as you move an obstacle towards or away from the robot?
- Increase or decrease the distance the robot should stay from the wall. Do you notice a difference in sensor performance?



# Activity 11: Reflections

## Bang-Bang Control

- What else could bang-bang control be used for?
- Describe the behavior of the robot using bang-bang control.
- How can bang-bang control be improved
  - Think about activity 8c



# Proportional Control



# Proportional Control

- For bang-bang control, motion values change instantly when a target is reached.
- For proportional control, motion values are changed proportionally to the difference between an *actual* and a *desired* value
- Proportional control works best when controlling motors with velocity commands (e.g., **mav**) rather than power commands (e.g., **motor**)
- If we want the robot to maintain a distance equivalent to sensor value of 600 then we can set the robot's velocity to:  
**velocity = kP \* (600 - analog(0));**
  - range sensor values > 600 mean that the robot is too close and generate negative velocities
  - range sensor values < 600 indicate For proportional control, motion values are changed proportionally to the difference between an actual and a desired value
  - **kP** can be set to adjust the responsiveness as desired



# Activity 12: Objectives

## Proportional Control

Write a program (or modify Activity 10) that monitors the "ET" sensor (Day 1, slide ~163) to keep the robot a certain distance away from a moving obstacle using proportional control.

Run the program on the DemoBot.





# Activity 12: Pseudocode

## Proportional Control

1. Set analog port 0 to floating analog.
2. Loop until side button is pressed.
  - a. Set the velocity of both motors of the robot to a value proportional to the difference between the reading from the analog rangefinder on port 0 and its reading for about 6 inches.
3. Stop the DemoBot



# Activity 12: Solution

## Proportional Control

```
// ***** Proportional Control
int main() {
    int velocity, distVal = 600;
    double kP = .15; // adjust
    // Step 1: Set analog port 0 to floating analog
    set_analog_pullup(0,0);
    printf("Back up if obstacle too close\n  otherwise go forward\n");
    printf("Press A button to start\n\n");
    while (a_button() == 0);
    printf("Press side button to stop\n");
    // Step 2: Loop until side button is pressed
    while (side_button() == 0)
    {
        // Step 2a: move the motors proportional to the distance to the obstacle
        velocity = kP * (distVal - analog(0));
        motor(0, velocity); motor(2, velocity);
    }
    // Step 3: Stop
    ao();
    printf("done\n");
    return 0;
}
```



# Activity 12: Experiment

## Proportional Control

- What is the robot's motor behavior as it gets close to the stopping point? Observe the motor lights.
  - How does it change if the value of **kP** is changed?
- Increase or decrease the distance the robot should stay from the obstacle. Do you notice a difference in sensor performance?
- What happens when you move the obstacle inside about 4 inches from the ET sensor?



# Activity 12: Reflections

## Proportional Control

- How does proportional control compare to bang-bang control?
- What else could proportional control be used for?
  - would it improve color object tracking?
  - How about line following?
- Describe the behavior of the robot using proportional control.
- How can proportional control be improved?



# Bang-Bang Control and Arm



# Activity 13: Objectives

Using a servo motor to operate an arm, write a function to operate the servo that raises or lowers the arm on DemoBot



# Activity 13: Prep

- Using servo motors
  - Review Day 1 servo motor section
- Bang-bang control
- Arm function



# Using Servos Recap

- The KIPR Link library functions for enabling (or disabling) all servo ports:
  - **enable\_servos();** activates all servo ports
  - **disable\_servos();** de-activates all servo ports
- **set\_servo\_position(2, 925);** rotates servo 2 to position 925
  - You can preset a servo's position before enabling servos so it will immediately move to the position you want when you enable servos
  - Default position when servos are first enabled is 1024
- **get\_servo\_position(2);** provides the current position of servo 2
  - Works only when servos are enabled
- The KIPR Link *Servo Test* screen can be used to center a servo or determine what position values to use once the servo is installed on a robot



# Servos and DemoBot

- There is one servo on DemoBot for raising or lowering its arm
- The *Servo Test* screen can be used for each servo to determine limit settings
  - Arm fully *up* and arm fully *down*
- Record these values for later use



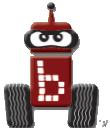
# Bang-Bang Control

- Bang-bang control is a control strategy that changes power to a new value without a transition such as first slowing down
  - With a moving robot, it is bang-bang control if you slam into reverse when you hit a wall (bang) going forward and then slam into forward when you hit a wall going backward (bang), never slowing down to soften the blow
  - Snapping an arm up or down is also a form of bang-bang control



# Arm Function (Bang-Bang)

- Assuming servos have been enabled
  - To raise the arm (bang)  
`set_servo_position (<armport>, <raised-position>) ;`
  - To lower the arm (bang)  
`set_servo_position (<armport>, <lowered-position>) ;`
- Function prototype  
`void arm(int up_down) ;`
- Function strategy
  - **if** the parameter **up\_down** is 1 raise the arm
  - **else** lower the arm



# Selecting the Action to Perform

## if - else

- For **while**, an action is performed so long as the condition check is true
- In contrast, for **if - else**, one action is performed if the condition is true and another if it is false
- Example:

```
if (up_down != 0) { // bang
    set_servo_position(ARMPORT, UPOS);
}
else { // bang
    set_servo_position(ARMPORT, DPOS);
}
```



# C Preprocessor

## #define

- Before your program is compiled it is first examined by the C preprocessor for preprocessing commands
- **#define**
  - Equates a meaningful name to repeatedly encountered text
    - `#define LMOTOR 0`
    - `#define GET_PC get_motor_position_counter`
  - The preprocessor will replace all occurrences of `LMOTOR` with `0` and `GET_PC` with `get_motor_position_counter`; for example,  
`if (GET_PC(LMOTOR) < 30) { . . . }`  
is equivalent to  
`if (get_motor_position_counter(0) < 30) { . . . }`



# Program Steps

Using a servo motor to operate an arm

- Create **#define** statements for using servos
  - Names specifying limits of servo travel,  
**UPOS** is *<up-position>*, **DPOS** is *<down-position>*
  - Names for arm function action  
**UP** is 1, **DOWN** is 0
  - Names to remember the ports for the arm servo  
**ARMPORT** is 0
- Arm function prototype  
**void arm(int up\_down);**
- In your **main** function
  - **enable\_servos();**
  - Repeat several times: raise the arm, sleep a bit, lower it, sleep a bit
  - **disable\_servos();**
- Arm function definition



# Activity 13

- Write a function with prototype  
**void arm(int up\_down);**  
that uses bang-bang control to raise/lower the arm on DemoBot and uses **#define** to assign more meaningful names to constant values
- Start and stop with button presses
- Test your function using a program as outlined in the program steps
- Adjust your arm's **UPOS** and **DPOS** values as necessary to improve the accuracy of the arm's movement



# Activity 13: Reflections

Bang-bang control with Demobot arm

- How would you attach and operate a claw attached to the arm?
- Would arm modifications be needed?
- When would it be better to use a control strategy that raised/lowered the arm gradually?



# Activity 13: Solution

## Bang-bang control with Demobot arm

```

// Using a servo motor to operate an arm using bang-bang control
#define DOWN 0      // arm is raised
#define UP 1        // arm is lowered
#define UPOS 200    // servo position arm raised
#define DPOS 1200   // servo position arm lowered
#define ARMPORT 0   // servo port for arm
void arm(int up_down); // prototype for arm function
int main()
{
    arm(UP);           // initialize arm position as up
    enable_servos();   // start servos with arm up
    printf("Lower and raise arm until side button pressed\n");
    printf("Press A button to start\n\n");
    while(a_button() == 0);
    while(side_button() == 0)
    { // repeat until user presses side button
        msleep(2000); arm(DOWN); // leave up for 2 seconds, then lower it
        msleep(2000); arm(UP);   // leave down for 2 seconds, then raise it
    }
    disable_servos(); // shut down servos
    printf("Done!\n");
    return 0;
}
void arm(int up_down)
{
    if (up_down != 0) set_servo_position(ARMPORT, UPOS);
    else set_servo_position(ARMPORT, DPOS);
}

```



# Proportional Control and Arm



# Activity 14: Objectives

Using proportional control to operate an arm, write functions to operate the servo controlling the arm servo on DemoBot, one to raise the arm and one to lower the arm

- Have each function slowly speed up the servo after it starts, then slow it down as the limit of arm travel is neared



# Activity 14: Prep

Using proportional control to operate an arm

- Proportional control
- Speeding up then slowing down proportionally
- Function for raising an arm



# Proportional Control

- For bang-bang control, motion values change instantly when a target is reached
- For proportional control, motion values are lowered proportionately as the target is neared
  - Assume the servo position for fully down is given by the **#define** names **DPOS** and for fully up by **UPOS**, where **UPOS < DPOS**
  - These are best determined using the *Servo Test* screen
- As an example, assuming **enable\_servos();** here's a loop that slows down the servo as it moves closer to being fully raised

```
while (srvpos > (UPOS + 5)) { // quit if close enough
    // reduce srvpos by a smaller amount each time
    srvpos = srvpos - sqrt(srvpos - UPOS);
    set_servo_position(ARMPORT, srvpos);
    msleep(100); // give time to move
}
set_servo_position(ARMPORT, UPOS); // finish up
```



# Speeding Up, Then Slowing Down

- For an arm, it is useful to gradually increase speed as the arm moves, then slow it back down as it nears its target position
  - The midpoint of arm travel is

```
midpt = UPOS + (DPOS-UPOS) / 2;
```

or half way between **UPOS** and **DPOS**

- When moving away from **UPOS**, **sqrt (srvpos - UPOS)** increases and **sqrt (DPOS - srvpos)** decreases

```
if (srvpos < midpt) {
  amt = 5 + sqrt (srvpos - UPOS);
}
else {
  amt = 5 + sqrt (DPOS - srvpos);
}
srvpos = srvpos + amt; // amt to change srvpos
```

- amt** is higher towards the midpoint and lower toward the up and down positions (**UPOS** and **DPOS**)
- 5 is added to **amt** to ensure when using amt to move the servo it is at least 5 from its current position



# Function to Raise Arm

Proportional control

From the current position of the servo, if less than halfway to being raised, speed up and once past halfway begin slowing down

```
void raise_arm()
{
    int amt, srvpos = get_servo_position(ARMPORT);
    int midpt = UPOS + (DPOS - UPOS)/2;
    while (srvpos > (UPOS + 5) ) { // quit if close enough
        if (srvpos < midpt) {
            amt = 5 + sqrt(srvpos - UPOS);
        }
        else {
            amt = 5 + sqrt(DPOS-srvpos);
        }
        srvpos = srvpos - amt;      // move closer to UPOS
        set_servo_position(ARMPORT,srvpos); // move arm
        msleep(100); // give time to move
    }
    set_servo_position(ARMPORT, UPOS); // finalize at UPOS
}
```



# Activity 14

- Write functions with prototypes

```
void raise_arm();
```

```
void lower_arm();
```

that use proportional control to operate the arm on DemoBot, speeding up through the midpoint of travel then slowing down

- Rework the `raise_arm` example to get `lower_arm`
- Test your functions by writing a `main` function to raise the arm and then lower the arm, repeating until the side button is pressed
  - Don't forget to put in `#define` names and values for `UPOS`, `DPOS`, and `ARMPORT`



# Activity 14: Reflections

Using proportional control to operate an arm

- Are there any advantages for using non-linear scaling instead of linear scaling for proportional control?
  - What would be the effect of changing the constant 5 used in the examples to other values?
- You could raise and lower the arm using bang-bang control – how would this affect being able to hold something in a claw attached to the arm?
- How would you write a single function for raising/lower the arm (like done for bang-bang raising/lowering the arm), and make it independent of whether **UPOS < CPOS** or vice-versa?



# Activity 14: Solution

Using proportional control to operate an arm

```
// Using proportional control to operate an arm
#define UPOS 200 // servo positions for arm up
#define DPOS 1200 // servo positions for arm down
#define ARMPORT 0 // servo port for arm
void raise_arm(); // prototype for arm function
void lower_arm(); // prototype for arm function
int main() {
    set_servo_position(ARMPORT, DPOS); // initialize arm down
    enable_servos(); // and start servos
    printf("Lower and raise arm until side button pressed\n");
    printf("Press A button to start\n\n");
    while(a_button() == 0) {
    }
    while(side_button() == 0) { // repeat until user presses button
        msleep(1000); raise_arm(); // leave down briefly, then raise it
        printf("Arm is up\n");
        msleep(1000); lower_arm(); // leave up briefly, then lower it
        printf("Arm is down\n");
    }
    disable_servos(); // shut down servos
    printf("DONE!");
    return 0;
}
```

*(Continues on next page)*



# Activity 14: Solution, Cont'd

Using proportional control to operate an arm

```
void raise_arm()
{
    int amt, srvpos = get_servo_position(ARMPORT);
    int midpt = UPOS + (DPOS - UPOS)/2;
    while (srvpos > (UPOS + 5)) { // quit if close enough
        if (srvpos < midpt) {
            amt = 5 + sqrt(srvpos - UPOS);
        }
        else {
            amt = 5 + sqrt(DPOS - srvpos);
        }
        srvpos = srvpos - amt;      // move closer to UPOS
        set_servo_position(ARMPORT, srvpos); // move arm
        msleep(100); // give time to move
    }
    set_servo_position(ARMPORT, UPOS); // finalize at UPOS
}
```

*(Continues on next page)*



# Activity 14: Solution, Cont'd

Using proportional control to operate an arm

```
void lower_arm()
{
    int amt, srvpos = get_servo_position(ARMPORT);
    int midpt = UPOS + (DPOS - UPOS)/2; // point of fastest travel
    while (srvpos < (DPOS - 5)) { // quit if close enough
        if (srvpos < midpt) {
            amt = 5 + sqrt(srvpos - UPOS); // move amount
        }
        else {
            amt = 5 + sqrt(DPOS - srvpos); // (at least 5)
        }
        srvpos = srvpos + amt; // move srvpos closer to DPOS
        set_servo_position(ARMPORT, srvpos); // move arm
        msleep(100); // give time to move
    }
    set_servo_position(ARMPORT, DPOS); // finalize at DPOS
}
```



# Accelerometer



# Activity 15: Objectives

The Create has no rear bumper, so use the KIPR Link accelerometer to determine when it hits something while backing up



# Activity 15: Prep

Use the *Graph* screen to initially determine values the accelerometer generates when it is hit in the rear



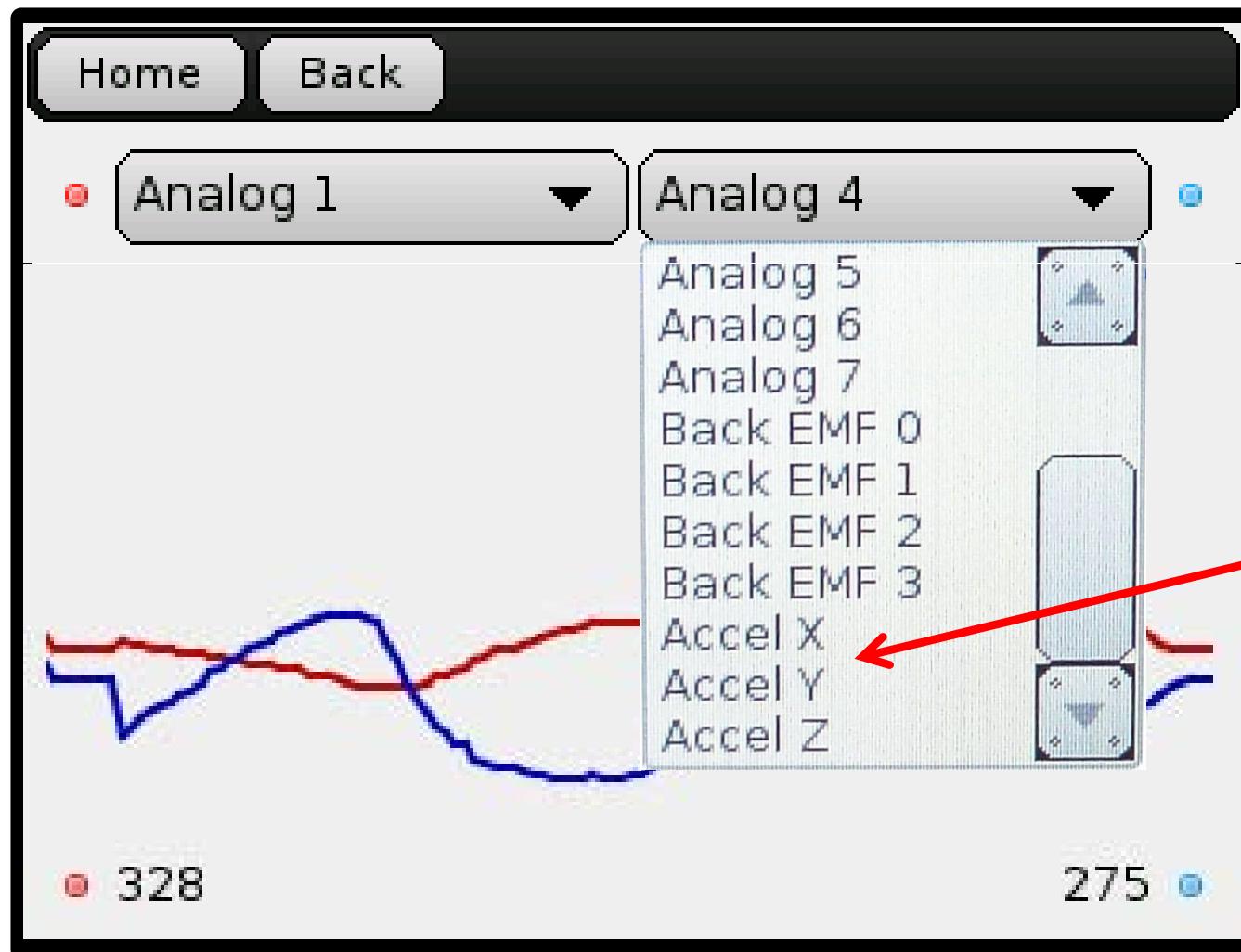
# Accelerometer

- An accelerometer measures force accelerating an object in 3 directions (vertical z, horizontal x, and horizontal y)
  - The y direction is front-to-back on the KIPR Link, x is left-to-right
  - Midpoint of 0, -512 to 511 range
- For an object at rest or moving on a flat surface at a constant speed the accelerometer measures no force for x and y
  - Gravity always exerts a force, so  $z > 0$
  - Lining the KIPR Link up on the Create to look forward, it's rear is to the front of the Create and moving forward is the y direction
- The *Graph* screen shows this behavior for the KIPR Link's built in accelerometer (and can be used for other sensors as well)
  - scaling for the accelerometer has gravitational force (z value) at around 256 (perhaps off by 10-15%)
  - Suddenly stop the KIPR Link while moving forward to see y spike



# KIPR Link Sensor Scope Screen

- Go to the *Sensor Scope* screen
  - Under the *Motors and Sensors* tab on the opening screen, then under *Sensors*





# Sample **accel\_x** Test

- Assume a variable **fwd\_bk** is being used to determine if we're looking for a hit while going forwards (+1) or while going backwards (-1)
- For the case going backwards, **fwd\_bk** is -1, so when positive acceleration (acceleration in the forward direction) is detected something has been hit

```
create_drive_straight(-250);
msleep(500); // get to constant speed
while (fwd_bk == -1) { // switch if hit
    if (accel_y() > 100) {
        fwd_bk = 1;
    }
}
```



# Activity 15

- Use the accelerometer to detect when the Create hits something while going forward or backward
- Pick the accelerometer axis (x, y, or z) that is aligned with the direction of motion
- When something is hit reverse direction
- Stop when the side button is pressed



# Activity 15: Reflections

Detect a hit when backing up the Create module

- What happens if the accelerometer fails to detect a bump?
- What if you try to use different values (larger or smaller than 100) to detect a bump?



# Activity 15: Solution

Detect a hit when backing up the Create module

```
// Using the accelerometer to detect a hit when moving the Create module
int main() {
    int fwd_bk = -1;
    int threshold = 100;
    create_connect();
    printf("Forward and back reversing direction on impact\n");
    printf("Press A button when ready\n\nPress side button to quit\n");
    while (a_button() == 0) {}
    while (side_button() == 0) {
        // monitor for a hit while going backward
        create_drive_straight(-250); // start going backwards
        msleep(500); // give time to reach constant velocity
        while (fwd_bk == -1) {
            printf("Moving backwards\n");
            if (accel_y() > threshold) {
                fwd_bk = 1; // hit, reverse direction
            }
            if (side_button() == 1) {
                break;
            }
        }
    }
}
```

*(Continues on next page)*



# Activity 15: Solution, Cont'd

Detect a hit when backing up the Create module

```
// monitor for a hit while going forward
create_drive_straight(250); // start going forwards
msleep(500); // give time to reach constant velocity
while (fwd_bk == 1) {
    printf("Moving forward\n");
    if (accel_y() < -threshold) {
        fwd_bk = -1;
    }
    // hit detected, reverse direction
    if (side_button() == 1) {
        break;
    }
}
create_disconnect();
printf("done\n");
return 0;
}
```



# Using Graphics for a Custom Interface



# Activity 16 Background

- KISS IDE includes libraries for 2D graphics and mouse (and keyboard) input.
- The Link supports graphics display and limited mouse (touch) input.
- The simulator (and Computer target) support graphics and input as well – for up to whatever size window your display supports
- The Link has a maximum window of 320 x 240
- Smaller graphics windows are centered on the Link display



# Key Graphics Functions

```
graphics_open(<width>, <height>)
graphics_close()
graphics_fill(<r>, <g>, <b>)
graphics_update()
graphics_pixel(<x>, <y>, <r>, <g>, <b>)
graphics_line(<x0>, <y0>, <x1>, <y1>, <r>, <g>, <b>)
graphics_circle(<x0>, <y0>, <radius>, <r>, <g>, <b>)
graphics_circle_fill(<x0>, <y0>, <radius>, <r>, <g>, <b>)
graphics_triangle(<x0>, <y0>, <x1>, <y1>, <x2>, <y2>, <r>, <g>, <b>)
graphics_triangle_fill(<x0>, <y0>, <x1>, <y1>, <x2>, <y2>, <r>, <g>, <b>)
graphics_rectangle(<xUL>, <yUL>, <xLR>, <yLR>, <r>, <g>, <b>)
graphics_rectangle_fill(<xUL>, <yUL>, <xLR>, <yLR>, <r>, <g>, <b>)
```

- Note that x is the column and y is the row for the graphics window; width is number of columns (max x), while height is the number of rows (max y). Origin is in the upper left corner of the window



# Key Mouse Functions

**get\_mouse\_location(<\*x>, <\*y>)**

This function takes two pointer as arguments. Just create two **int** variables and put an "&" in front of them. e.g.,

```
int mx, my;  
get_mouse_position(&mx, &my);
```

**mx** now contains the x coordinate and **my** contains the y coordinate value of the cursor position on the Link screen.

**get\_mouse\_left\_button()**

returns the value 1 when the screen is being touched and 0 otherwise.



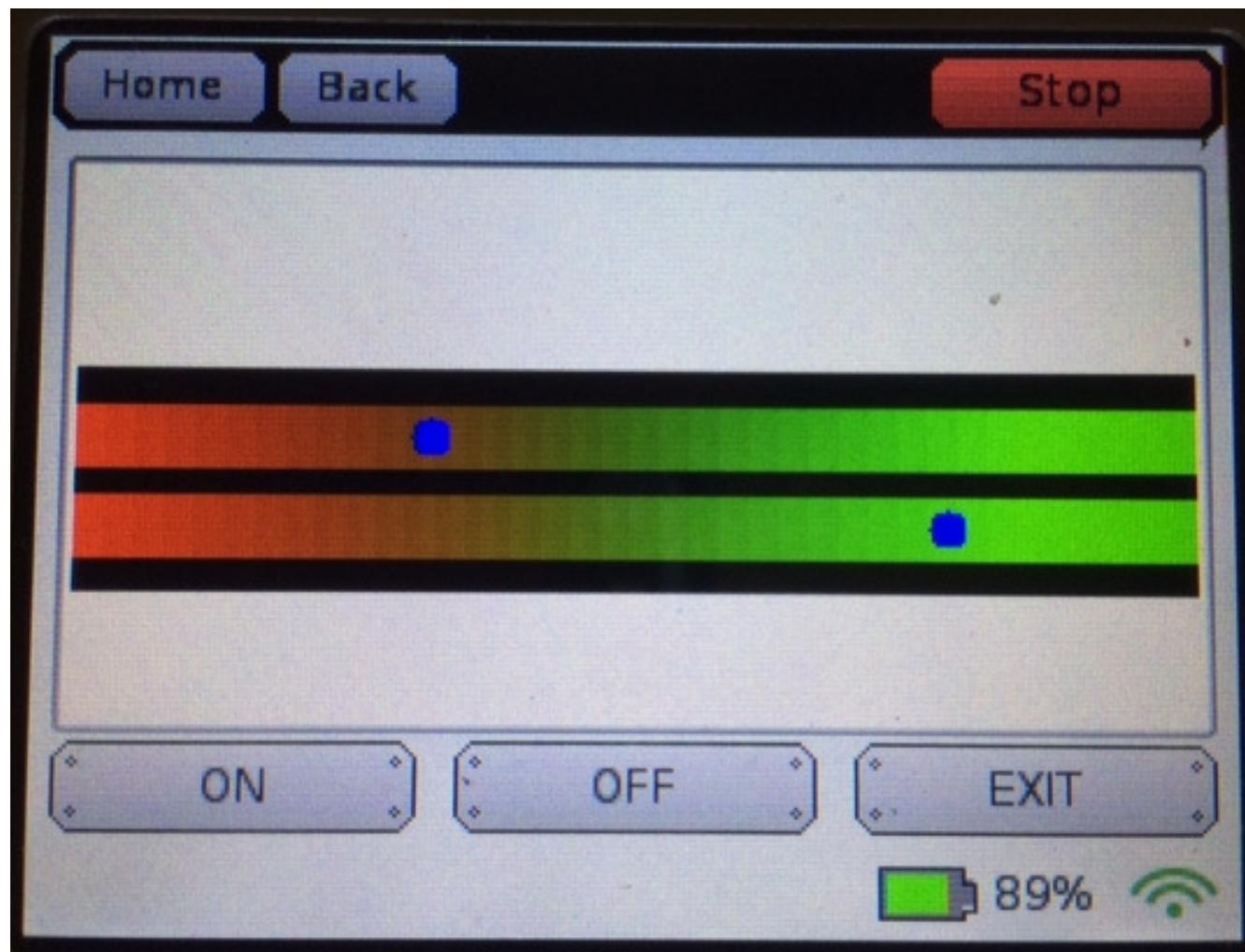
# Program Example Pseudo-Code

Create a graphics interface for controlling two motors. Make two sliders whose value controls the corresponding motors. The user touches along the slider to change the value.

- 1) Create a graphics window
- 2) Label the soft buttons: **start**; **stop**; **exit**
- 3) Loop until exit button is clicked:
  - 1) draw the sliders as red/green gradients (unless off)
  - 2) check if screen is being touched
  - 3) check each slider to see if it is the location of the touch
    - 1) If slider is touched, update the slider bar
    - 2) If slider is touched and motor state is on, update speed of motor



# Example UI for Motors





# Program Example C-Code

```

int main()
{
    int c, mx, my, v0 = 0, v2 = 0, motorPower = 1;
    set_a_button_text("ON"); set_b_button_text("OFF"); set_c_button_text("EXIT");
    graphics_open(300,60); // open graphics window
    graphics_fill(0,0,0); // make it black
    while(!c_button()) { // loop until exit is clicked
        if (a_button()) { // set motor on/off state
            motorPower = 1;
        }
        if (b_button()) {
            motorPower = 0;
        }
        if (motorPower) {
            for(c = 0; c < 300; c++) { // draw red/green gradient
                graphics_line(c,10,c,26,255-c,c,0); // M0 controller
                graphics_line(c,34,c,50,255-c,c,0); // M2 controller
            }
            motor(0,0.4*v0); motor(2,0.4*v2); // update motor speeds
        }
    }
}

```

*(Continues on next page)*



# Program Example C-Code

## Cont'd

```

else { // motors are powered down
    ao(); // sliders are red
    graphics_rectangle_fill(0,10,299,26,255,0,0);
    graphics_rectangle_fill(0,34,299,50,255,0,0);
}
graphics_circle_fill(v2,42,5,0,0,255); // draw previous slider position
graphics_circle_fill(v0,18,5,0,0,255);
graphics_update(); // paint the new image to screen
if (get_mouse_left_button() == 1) { //is screen touched?
    get_mouse_position(&mx,&my); // if so get the position
    if (my >= 10 && my <= 26) { // and update the approiate slider
        v0 = mx;
    }
    if (my >= 34 && my <= 50) {
        v2 = mx;
    }
}
graphics_close();
return 0;
}

```



# Remove Accumulated Heading Errors



# Activity 17

- Remove Accumulated Heading Errors
- As your robot moves about its heading will drift from the robot's "idea" of its heading
  - When moving straight, the robot's heading will drift
  - When making a turn, the actual angle will differ from the angle specified
- You can have your robot occasionally do a maneuver to remove these accumulated errors using such methods as:
  - Have the robot follow a line of known heading
  - Have the robot physically align itself with a known object – which is the method we will explore in this activity



# Physically Aligning Robot With a Wall

- Extended surfaces such as the border of the Botball field are ideal for reducing rotation errors
- Several methods can be used including:
  - Having a large flat front or rear of the robot and driving the robot so that the flat surface of the robot is forced against the wall. The disadvantage of this method is that there is no verification it has succeeded
  - Use two contact sensors on the front or rear edge of the robot (whichever end is going to be against the wall). When the first sensor makes contact with the wall, rotate the robot such that the second sensor contacts the wall while maintaining contact with the first sensor
  - Use the Create bumper against flat surface: there are two switches, if you find out range where both are pressed, and then move halfway through that range, Create should be aligned (assuming bumper activation is symmetric).



# Activity 17

## Using Two Sensors to Align to Wall



- Build a two bumper sensor (an example is on the left)
- Make sure the sensors can activate independently
- Attach it to one of your robots
- Write a program so that if the robot runs into a wall with the bumper it rotates until both sensors are activated



# Aligning Robot With a Wall Using Create Bumper

```

// Define the base robot speed and the forward offset to keep the bumper pressed
#define SPEED 50
#define FOR 15
double timeTurn2Bumps(int dir); // function measures how long both bumpers pressed
void turnHalfTime2Bumps(int dir, double time); //turn in direction dir for half of
time
int main()
{
    int dir = 1; // dir = 1 means CCW and -1 means CW
    create_connect(); // connect to Create
    printf("press A to start towards a wall\n");
    while(a_button() == 0); // wait for the A button to be pressed
    while(!get_create_lbump() && !get_create_rbump()) {
        create_drive_straight(2*SPEED); // drive forward till a bumper is pressed
    }
    // if the left bumper and not the right is pressed, reverse rotation direction
    if (get_create_lbump() && !get_create_rbump()) {
        dir = -dir;
    }
    else { // if both bumpers are pressed, spin until only the right is pressed
        if(get_create_lbump() && get_create_rbump()) {
            while(get_create_rbump()) {
                create_spin_CCW(50);
            }
            msleep(100); // and then spin 0.1 sec longer
        }
    } // find how long, during rotation both bumpers are pressed
    turnHalfTime2Bumps(-dir,timeTurn2Bumps(dir)); // then turn back half that time
    printf("Create is orthogonal to wall. Done.\n");
    return 0;
}

```

*(Continues on next page)*



## Aligning Robot With a Wall Using Create Bumper (Cont'd)

```
double timeTurn2Bumps(int dir)
{ // dir determines which direction to rotate
    double startTime, turnTime;
    create_drive_direct (-dir*SPEED+FOR, dir*SPEED+FOR); // spin while pressing forward
    while (!(get_create_lbump() && get_create_rbump())) {}; // wait till both bumpers are hit
    startTime = seconds(); // seconds uses system clock
    while (get_create_lbump() && get_create_rbump()) {}; // wait till both bumpers are hit
    turnTime = seconds() - startTime; // turnTime has the amount of time both bumpers pressed
    msleep(100); // turn a little further
    create_stop(); // stop
    return turnTime; // send back how long both bumpers were pressed
}
void turnHalfTime2Bumps(int dir, double time)
{
    double startTime;
    create_drive_direct (-dir*SPEED+FOR, dir*SPEED+FOR); // spin while pressing forward
    while (!(get_create_lbump() && get_create_rbump())) {} // wait till both bumpers are hit
}
startTime = seconds(); // startTime occurs when both bumpers are pressed
while (seconds() - startTime < time/2.0) {} // keep turning for half of time
}
create_stop(); // stop and Create should be orthogonal against wall
}
```



# Approach a Specific QR Code



# Activity 18: Objectives

Identify and approach a QR code

Identify and approach a QR code that is a 'P'  
or a 'T' based on initial user input.



# Activity 18: Pseudocode

Identify and approach a QR code

- Create two buttons labeled "P" and "T" for the user to press
- Identify all QR codes
- Use a for loop to step through each QR code in the channel
- Check each QR code for information
  - If it matches the user input
    - Approach target – see activity 8c and slide ~277
    - If no matches found, spin to search
- Stop with bump



# Pseudocode for Approach to Target

1. Keep moving until close enough
2. If target is to the left, realign to left
3. If target is to the right, realign to right
4. And otherwise move on toward the target



# The **for** Loop

- The **for** loop is an alternative to while that can be clearer if you are trying to loop a specific number of times.
  - initialize i to 0, loop while  $i <$  takes on the index value of each object representing a QR code, and add 1 to i at the end of each iteration.
  - then check each object and if it is the QR code for 'P'...

```
int i;  
...  
for (i = 0; i < get_object_count(0); i++) {  
    if (get_object_data(0, i)[0] == 'P') {  
        ...  
    }  
    ...  
}  
...  
...
```



# How Close Am I?

A good way to use the camera to determine proximity to a target is to note:

- If the camera is higher off the ground than the target, as your robot approaches the target the y coordinate of the target increases; if the camera is lower – vice versa
- The y coordinate value relates to how close you are!
- Other properties of objects in the camera's view change systematically with distance
- Your program can access these properties (see the Manual for a list of vision functions)



# Activity 18: Reflections

Identify and approach a QR code

- If you use higher speeds how well does the algorithm perform?
- Rather than spin moves, suppose you continue to move forward but angle back toward center
  - performance should improve
  - but are there risks?



# QR Code data

- **get\_object\_data(<ch>, <obj>)**
  - returns a string pointer
  - returns -1 if the channel or object does not exist
- **get\_object\_data\_length(<ch>, <obj>)**
  - returns an int the length of the string
- **get\_object\_data(<ch>, <obj>) [0]**
  - returns the first character of the data (e.g., 'P')



# Need a QR code to test?

- Visit the website: <http://www.qrstuff.com>
- Allows easy generation of QR codes



# Activity 18: Description

## Vision

- Plug the camera into your KIPR Link
- Add a QR Code channel to your camera configuration (note the channel # -- probably channel 1)
- Copy the example program on the next page and download it to your KIPR Link
- Display a QR Code on your laptop or phone and point the Link camera at it
- Modify the printf for QR codes longer than one letter, try using the printf format %s to display the entire data field:

```
printf("QR code begins with %s\n", get_object_data(0,0));
```



# Activity 18: Code

```

// Prints the center coordinate of P QR codes otherwise prints first letter
int main(){ // Assumes that channel 0 of default config is QR code
    int row, col;
    char q;
    camera_open(); // start up camera
    while(a_button() == 0) {
        camera_update(); // get a new image
        if (get_object_count(0) > 0) { // is there a QR code?
            if (get_object_data(0,0)[0]=='P') { // is it a P?
                col = get_object_center_column(0,0);
                row = get_object_center_row(0,0);
                printf("P found at row: %i, col: %i\n", row, col);
            }
            else { // QR code is not P
                q = get_object_data(0,0)[0];
                printf("QR code begins with %c\n", q);
            }
        }
        printf("Done\n");
        return(0);
    }
}

```



# END