Botball 2009 Educator's Workshop

While you are waiting:

1. Install KISS-C
2. Install the CBC driver
   1. This is the same FTDI USB to serial driver that was used on the XBC
   2. If this has already been installed on your machine, you do not need to reinstall it. Reinstalling it will not hurt anything
   3. The installer is part of the Botball Tools install for Windows
   4. One the Mac, select the appropriate installer from the folder
3. If you are using a Mac and do not have the developer tools installed, flag down the instructor so it can be taken care of
4. If you are not sure how to do the previous steps please ask one of the staff!

Botball 2009 Educator's Workshop

v1.5 2009.2.13

2009 Botball Sponsors

Yahoo! Employee Foundation

Kauai Economic Development Board

KISS Institute thanks the following venue sponsors who support Botball by donating workshop and/or tournament venues free of charge:

- **University of North Florida -** Florida Region
- **Southern Illinois University Edwardsville -** Greater St. Louis Region
- **University of Maryland, College Park -** Greater DC Region
- **Grandville Boosters and Middle School** - Michigan Region
- **Hawaii Convention Center** - Hawaii Region
- **MITRE and the University of Massachusetts Lowell** - New England Region
- **Polytechnic University** - New York/New Jersey Region
- **University of Oklahoma** - Oklahoma Region
- **University of Pittsburgh** - Pennsylvania Region
- **CMU Qatar Campus** - Qatar Region
- **University of San Diego** - Southern California Region
- **University of Houston - CORE** - Texas Region

Botball 2009

written by the
the staff of KIPR and the Botball Instructors Summit participants

Tutorial Schedule:

- Day 1:
  - Robots, Botball and design projects
  - Botball related activities
  - KISS-C programming
  - Your first programs
    - *kissSim* simulator
    - CBC and Create
    - Simulated robot
  - Variables and data in C
  - Program flow/repetition
  - Create movement functions
  - C functions in general
    - Building your own library
    - Move and turn
  - Demobot Build
- Day 2:
  - Game Review
    - Robot Documentation
    - Building Rules
    - Game Challenge
  - Using tournament software
  - About the CBC
  - CBC Color Vision
  - Sensors, Create, Motors
  - Using Demobot
  - Motor functions and robot programming
  - Project design
  - Checkout and wrap up

What is a Robot?

Wikipedia Definition:

- *A **robot** is an electro-mechanical or bio-mechanical device or group of devices that can perform autonomous or preprogrammed tasks.*

Computing Dictionary Definition:

- *A mechanical device for performing a task which might otherwise be done by a human, e.g. spraying paint on cars.*

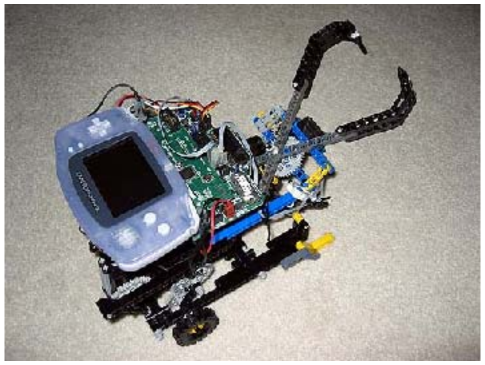
Synonyms:

- *automaton, golem*
  - –see also: android, humanoid, mechanical man, mechanism

Levels of Autonomy

- Remote control
  - Battle bots
- Teleoperation
  - Robot sensor feedback, operator uses a joystick or other device to instruct the control sequence
- Tele-presence control
  - Exoskeleton or VR control (operator mapped to robot)

- Semi-autonomous
  - Operator provides high-level goals and robot on its own seeks to accomplish the goals
- Autonomous
  - Robot on its own with multiple computational processes in parallel; emergent behaviors



Botball®

Basic robot elements

Human vs. Robot Subsystems

People	**Robots**
Bones	Mechanical Structure
Muscles	Effectors
Senses	Sensors
Digestion/Respiration	Power
Brain	Computer
Knowledge	Program

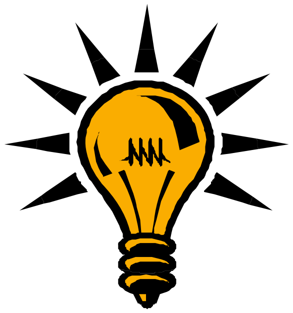
Structure

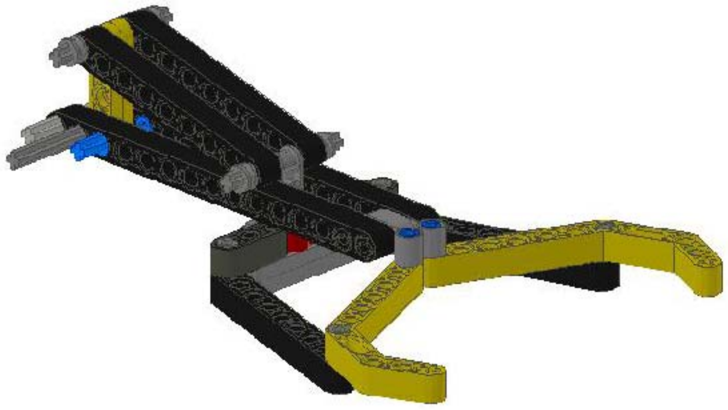
Robot Structure

- Carries all the forces exerted by the robot and the environment
- Joints in structure normally have effectors attached
- Holds sensors in position

Effectors

- Used to change the state of the robot
- Used to change the state of the world
- *Examples:*
  - Motors, thrusters, arms, or legs
  - Voice synthesizers, buzzers, and lights
  - Serial lines, com ports, radios





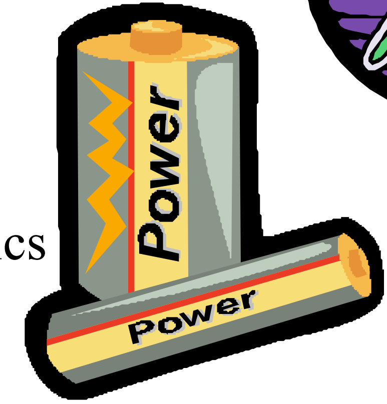
Botball

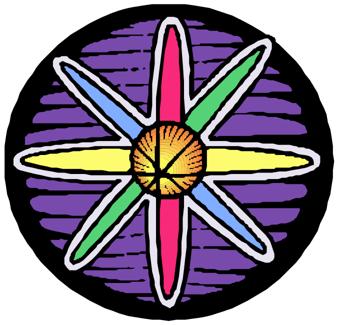
Sensors

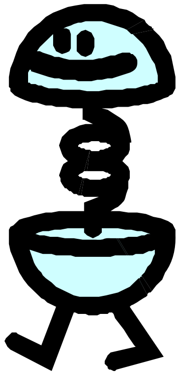
- Proprioceptive sensors
  - Report on the current state of the agent
  - e.g. encoders, gyros, low-voltage sensors
- External sensors
  - Report on the current state of the world
  - light sensors, range sensors, touch sensors,...

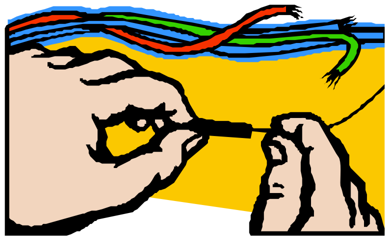
Power

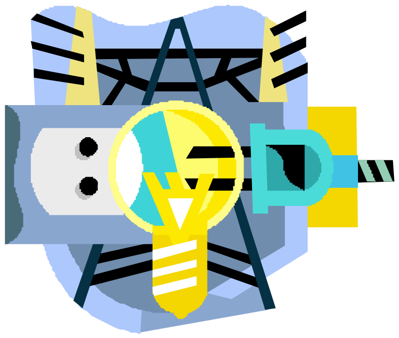
- The power source
  - Batteries, solar panels
  - Springs, hydraulics, pneumatics
  - Nuclear reactor
- Power distribution
  - Wires
  - Busses
- Power management
  - Regulators
  - Converters





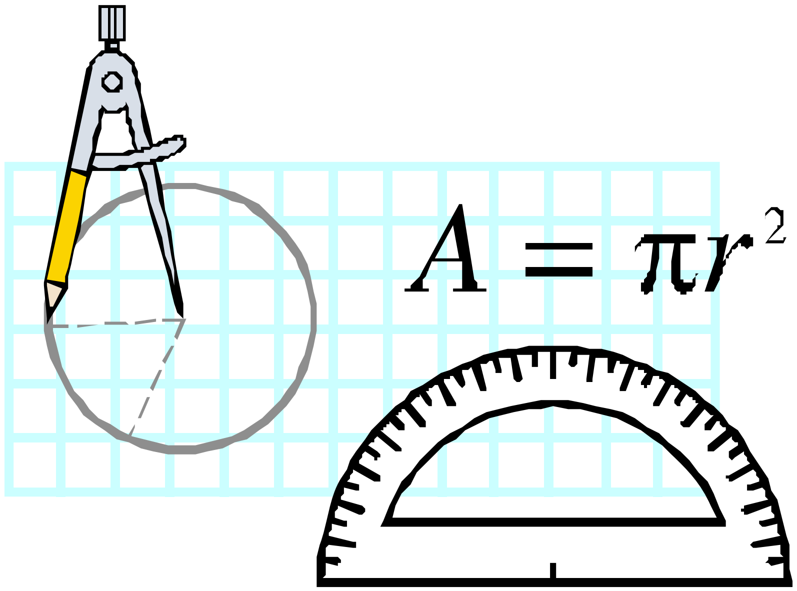






Computation

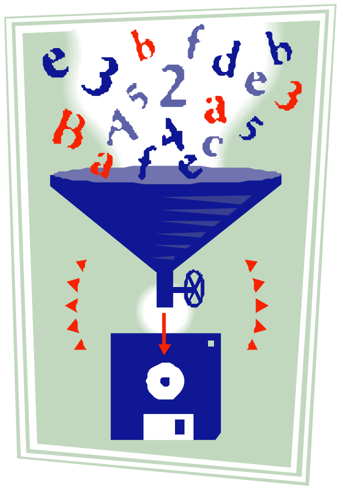
- Used to interpret sensor values; perception
- Used to generate proper effector commands
- Used to project effects and plan actions
- Must be low power
- Must be interfaced to proper sensor and motor circuitry

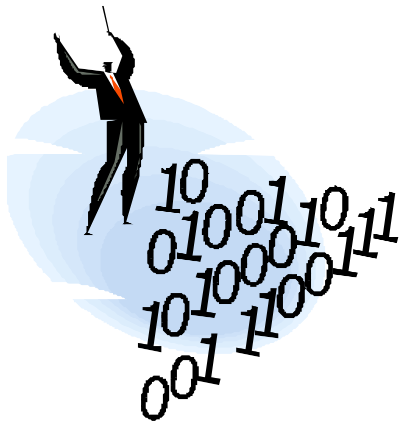


Botball

Information

- Internal Information
  - How to interpret sensor values
  - How to generate effector commands
  - Internal state & history
- External Information
  - World, user & predictive models
- Program
  - Determines robot actions
  - Forms robot plans
  - Debugging - introspection

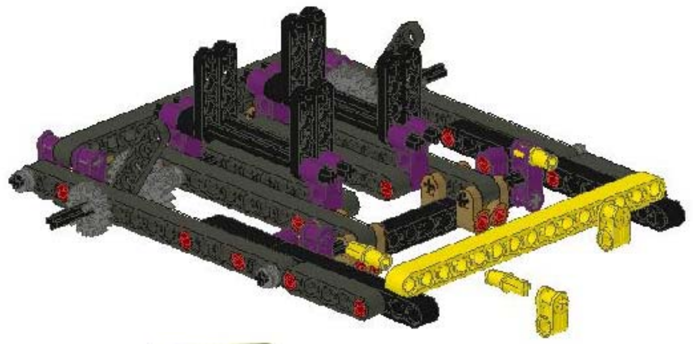




Botball Robotics

Botball Robots Are Real Robots

- Structure:
- Power: rechargeable batteries
- Effectors:
  - Gear head motors
  - Servo motors

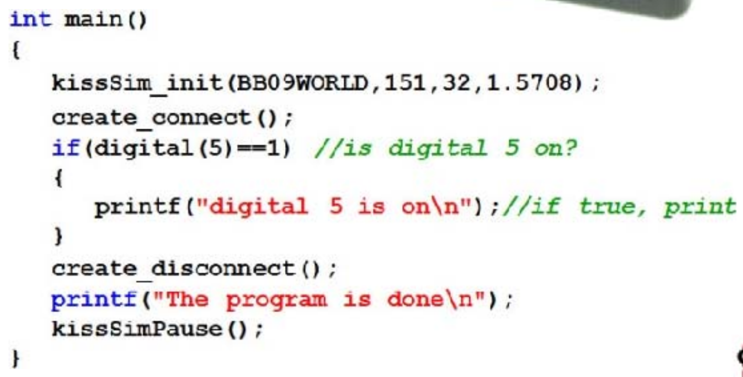


Botball Robots are Real Robots

- Sensors: 30+ sensors of many different types

- Computation: 2 microcontrollers

- Information: C programs written by the students



```
int main()
{
   kissSim_init(BB09WORLD,151,32,1.5708);
   create_connect();
   if(digital(5)==1)  //is digital 5 on?
   {
      printf("digital 5 is on\n");//if true, print
   }
   create_disconnect();
   printf("The program is done\n");
   kissSimPause();
}
```

CBC v.1

- 8 digital IO
- 8 10-bit analog inputs
- 4 servo outputs
- 4 PID motors with BEMF
- 1 TTL serial port
- 2 USB A ports
- USB connection to PC
- 1 physical button
- Integrated battery and charge system
- Speaker & microphone
- 320 x 240 color touch screen
- Runs LINUX

Botball is an R&D Design Project

- Botball involves the *Design Process*
- Botball involves team management and organization
- Botball involves research
- Botball involves documentation
- Botball involves schedule and resources
- Botball produces a product: your team of robots

The Botball program is designed to produce a successful Project Outcome

- Botball provides your team with just about all of the parts and software needed to complete the project.
- Botball provides teams with very specific requirements documentation
- Botball has an infinite number of possible solutions -- if your team runs into a technical problem, there are lots of other ways to solve the problem.
- Through documentation requirements and scheduled updates, the Botball program helps keep your team on a schedule that will yield a functioning robot solution

Botball

- Educational Goals:
  - Technology awareness
  - Systems engineering
  - Mechanical principles
  - C programming
  - Design and creativity
  - Scientific method
  - Learning the invention process (iteration…)
  - Documenting your work (on-line)

Botball is NOT Battlebots

- Robots are autonomous
  - there is no radio control
  - robot's behavior is based on your program and feedback from sensors
- Robots do not try and destroy other robots (and the game makes attempts counter productive)
  - they try to out maneuver opponents
  - they try to out think their opponents
  - they try and do the main task

Botball is a Student Activity...

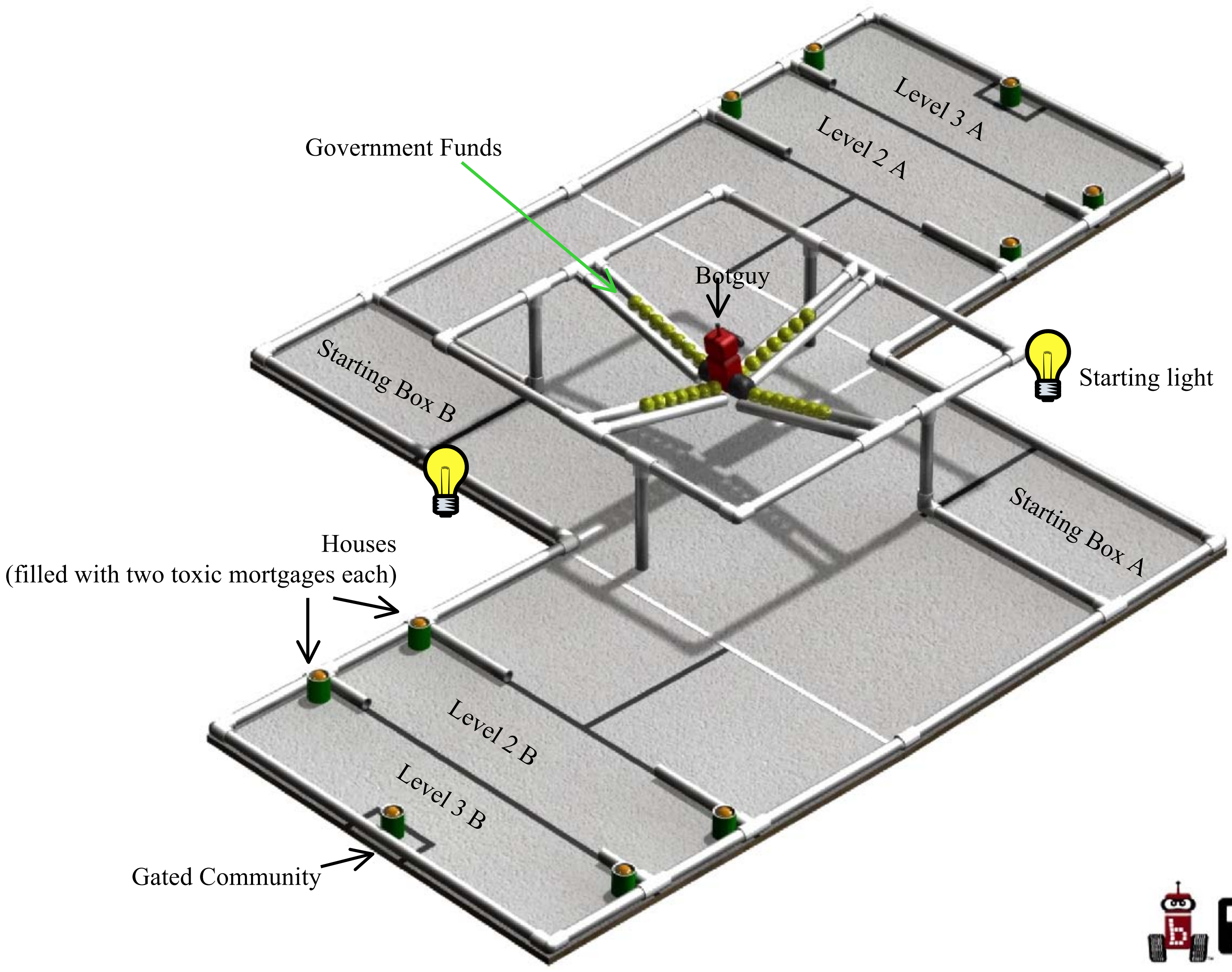
- Botball robots are student designed, built and programmed
- Students work in teams
- Mentors are there to answer questions, point students to resources, act as facilitators, etc.
- Teachers & mentors should be able to complete their role in Botball with their hands tied behind their back!

Botball Related Activities

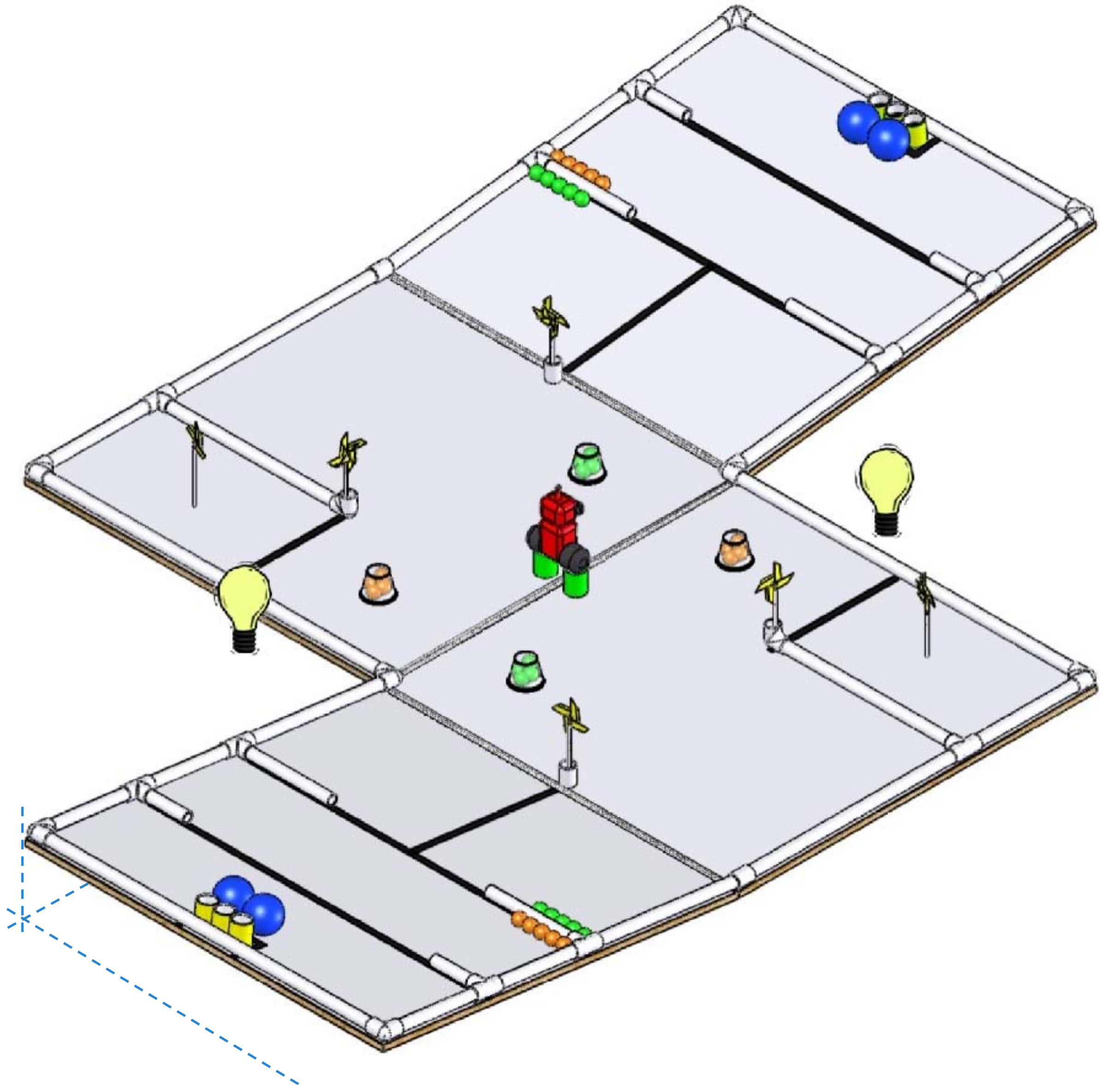
Beyond Botball Tournament

- Beyond Botball is an activity for those beyond Botball age (college students, adults, mentors, teachers, researchers, etc)
- The International Beyond Botball Tournament is held each year in conjunction with the Global Conference on Educational Robotics (GCER).
- Beyond Botball team entry forms and conference registration can be found at www.botball.org
- The 2009 Beyond Botball Tournament will be held in conjunction with GCER 2009 July 1-5, 2009 at the National Conference Center in Leesburg, VA

2009 Beyond Botball Board



And If This Makes You Wonder What the 2009 Botball Game Board is Like

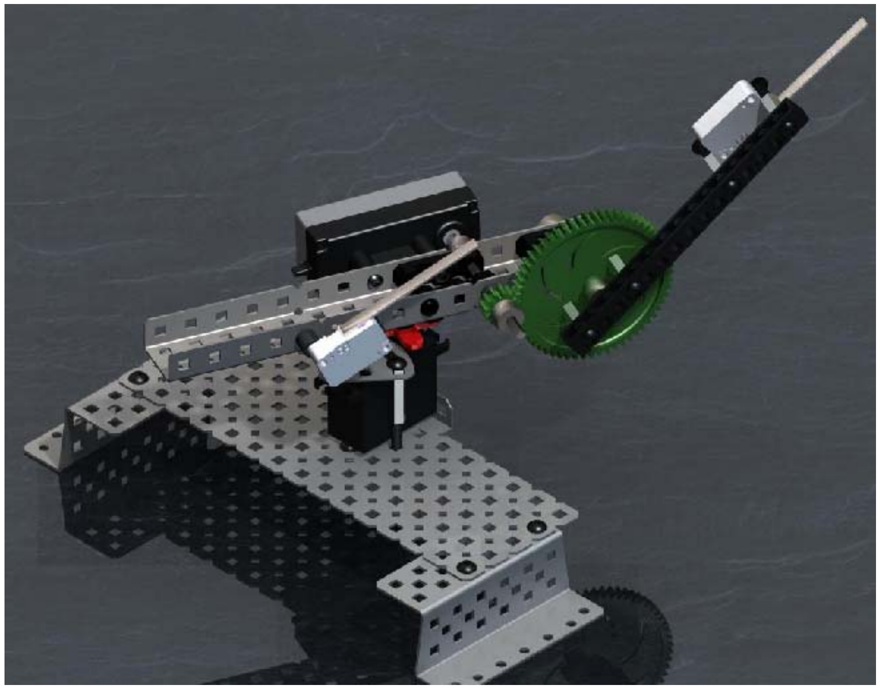


GOING ON NOW!

Objective: Create a CAD model and animation of your Botball® robot in action.
Eligibility: Open to all Botball® teams.

Deadline: June 19, 2009

Winners will be announced at GCER. Check your team Home Base for more information.



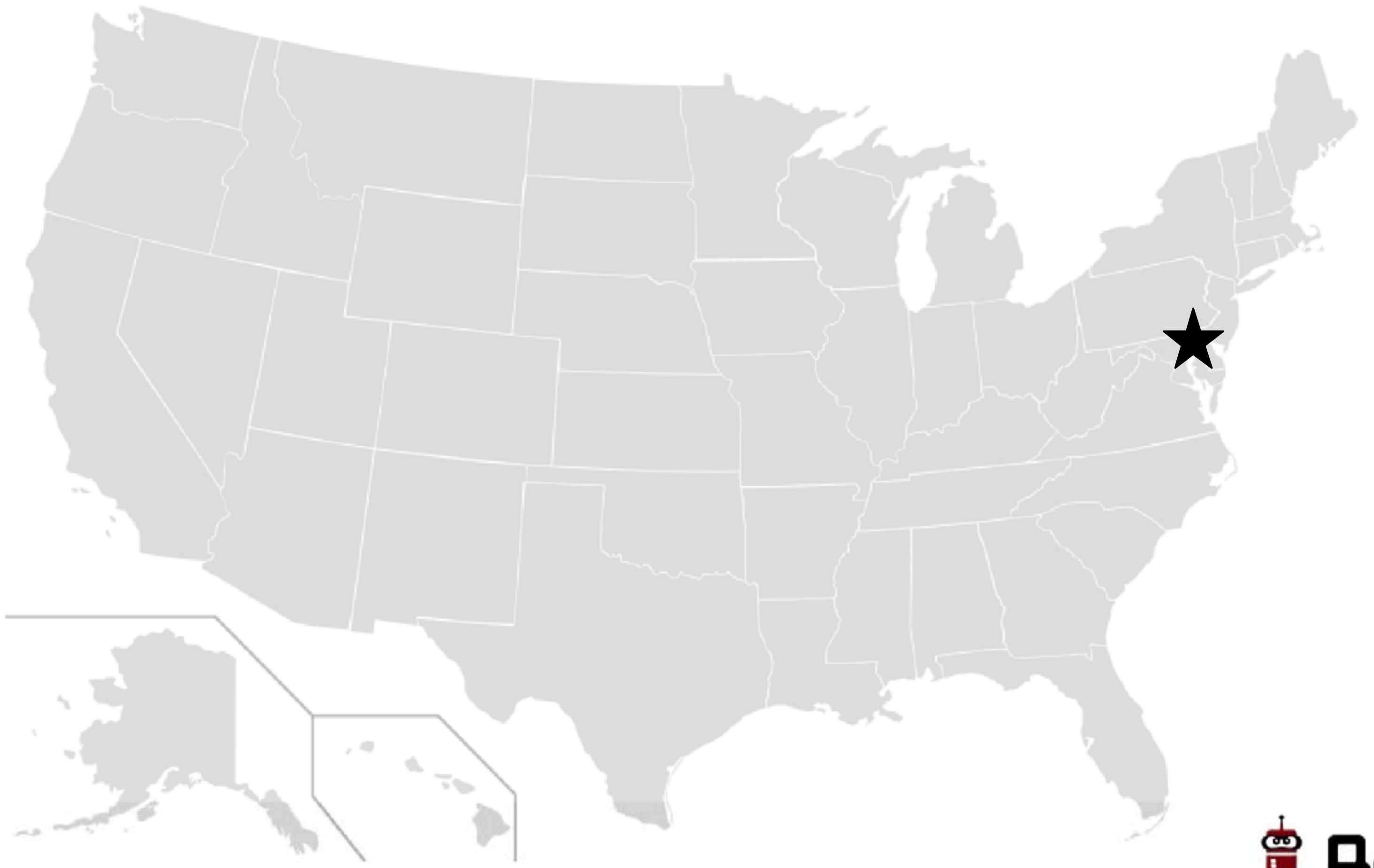
Solidworks CAD models of virtually every kit piece are on your team USB stick. The Solidworks CAD system is included with your kit.

GCER

(Global Conference on Educational Robotics) http://www.botball.org/current-season/GCER/

Where: National Conference Center in **Leesburg, VA**

(about 35 miles from Washington, D.C.)



Botball®

GCER

When: **July 1st - July 5th**. Pre-conference activities **June 30th**

Who: Middle school and high school students, educators, robotics enthusiasts, and professionals from around the world.

ALL TEAMS ARE INVITED!

Why: **Fun** and **Fireworks** in Washington D.C.

Meet and **Network** with students from around the country and world.

Talks by internationally recognized robotics experts from academia, government and industry

Teacher, Student, and Peer Reviewed **Track Sessions**

International Botball Tournament

Beyond Botball™ Tournament (Botball for grown-up kids!)

Autonomous Robotics Showcase

Botball®

Research and Design Website Challenge

- National website design competition
- Middle School and High School Divisions
- Winners receive $1,000 travel grant for GCER
- Winners are automatically accepted to submit a paper for GCER which will be included in conference proceedings

Research and Design Website Challenge

- Previous Challenges included:
  - Bionic prosthesis
  - Feeding the world - Saving the Environment
  - Robotics in Lunar Exploration
  - Household Robotics

Watch for next year's challenge, which will be released this Spring, enter and showcase your talents

KISS-C: the C Programming Environment used in Botball

KISS-C Software Package

- The KISS-C software is “donation ware”
  - It is free and can be freely distributed and used for personal and educational purposes
  - If you like it and are looking for a tax deduction, please consider the KISS Institute
  - If you would like to use KISS-C in a commercial product, contact the KISS Institute about licensing
- The latest version may be found at:
  - http://www.botball.org/educational-resources/
- KISS-C is a C IDE that uses the standard Gnu C compiler (gcc) to produce the program to be run on the robot controller
  - Implements the full ANSI C language
  - Can drive the iRobot Create from your PC
  - Interfaces to the CBC
  - Interactively guides hardware setup requirements
  - Provides an editor and on-line documentation
  - Provides an interactive environment with graphical simulator for testing and debugging

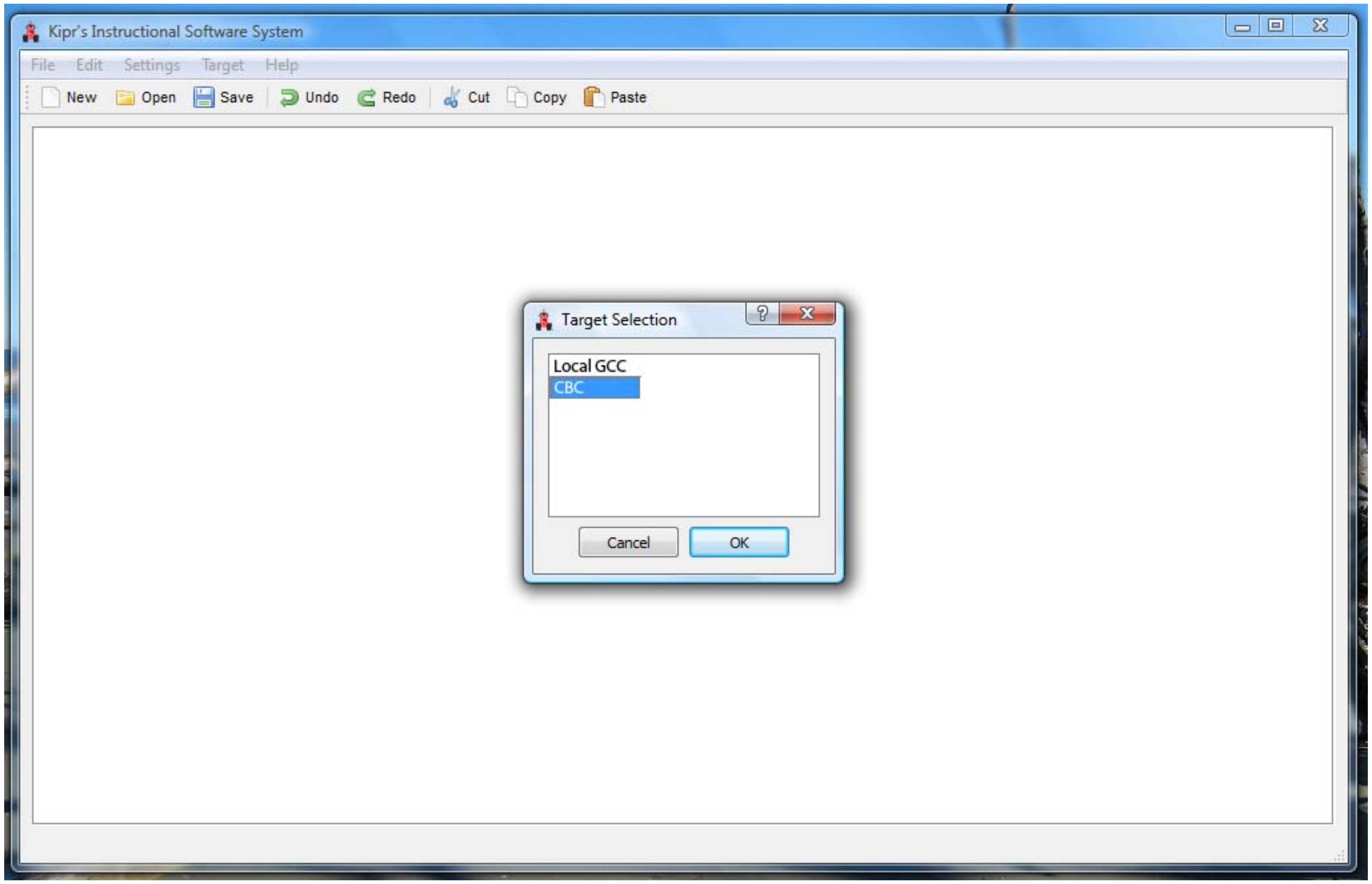
Setting Up

- If a CBC robot controller is being accessed, KISS-C can be used to produce the program for it to use
- The KISS-C 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 KISS-C interface
- To check for logic errors you:
  - Can simulate execution of your program using KISS C's built in graphical simulator for the CBC/Create, or
  - Attach a CBC controller and try running your program on it

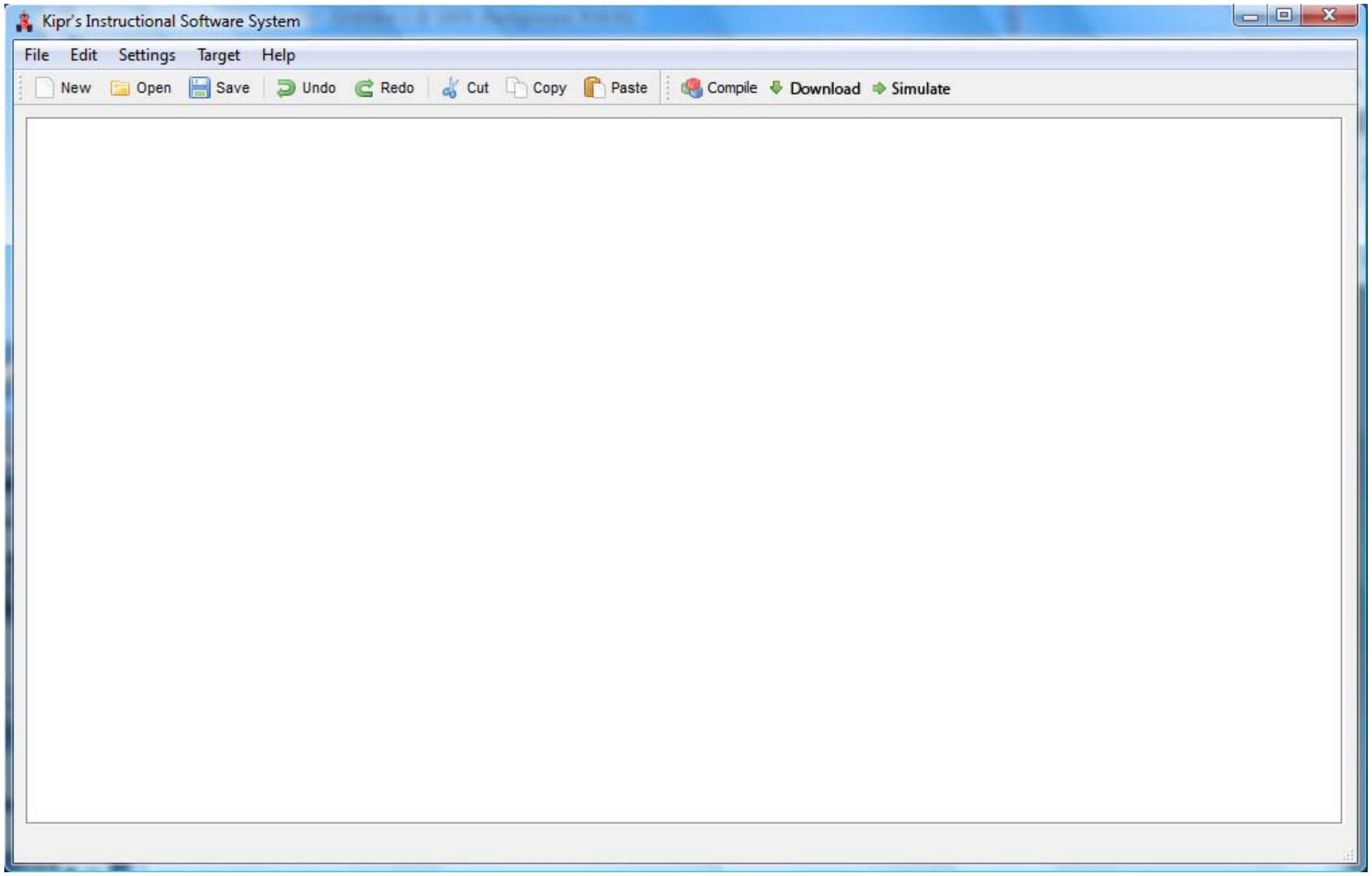
To Install KISS-C

- On a Mac OSX (10.3 and higher)
  - Double click on KISS-2.dmg file
    - The KISS-C folder can be placed in your Applications folder, or anywhere else convenient
    - Note: keep the app and the library folders in the same KISS-C folder (programs you write can be kept wherever you wish)
    - Note: KISS-C on the Mac makes use of the Apple Developer Tools (can be installed from the DVD that came with your OS or computer)
- On Windows (XP and Vista)
  - Double click on KISS-2.exe
    - KISS-C will be added to your program menu
    - A KISS-C shortcut will be placed on your desktop
- On Linux (Ubuntu)
  - If you are using LINUX -- you should know what to do (you may have to download and install some packages)!

Set KISS-C to use CBC Target


Kipr's Instructional Software System
File Edit Settings Target Help
New Open Save Undo Redo Cut Copy Paste
Target Selection
Local GCC
CBC
Cancel
OK

KISS-C Interface Should Now Be Active


Kipr's Instructional Software System
File Edit Settings Target Help
New Open Save Undo Redo Cut Copy Paste Compile Download Simulate

Simple Programs

A Simple C Program

```
int main()
{
    printf("This is a C program\n");
}
```

Step by Step Process

- Use the *File* menu .. *New,* or click on the *New* shortcut button (upper left corner)
- Type in the simple program for printing out a text string (change the text to be printed if you wish)
  - Hint: <Ctrl><Shift>= (or <Ctrl>+) enlarges the font
- Save the file using the the *File* menu .. *Save,* or click on the *Save* button
  - The file name prompted for can be whatever you want it to be (just remember it) – the directory where you save becomes the default directory for saving files
- Check your program by clicking on the *Simulate* button
  - This automatically saves your program and runs it if there is no error
- If there is an error, you can use the *Edit* .. *Goto Line* menu item to get to the problem
  - **the error will be on OR before that line**

Working Directory

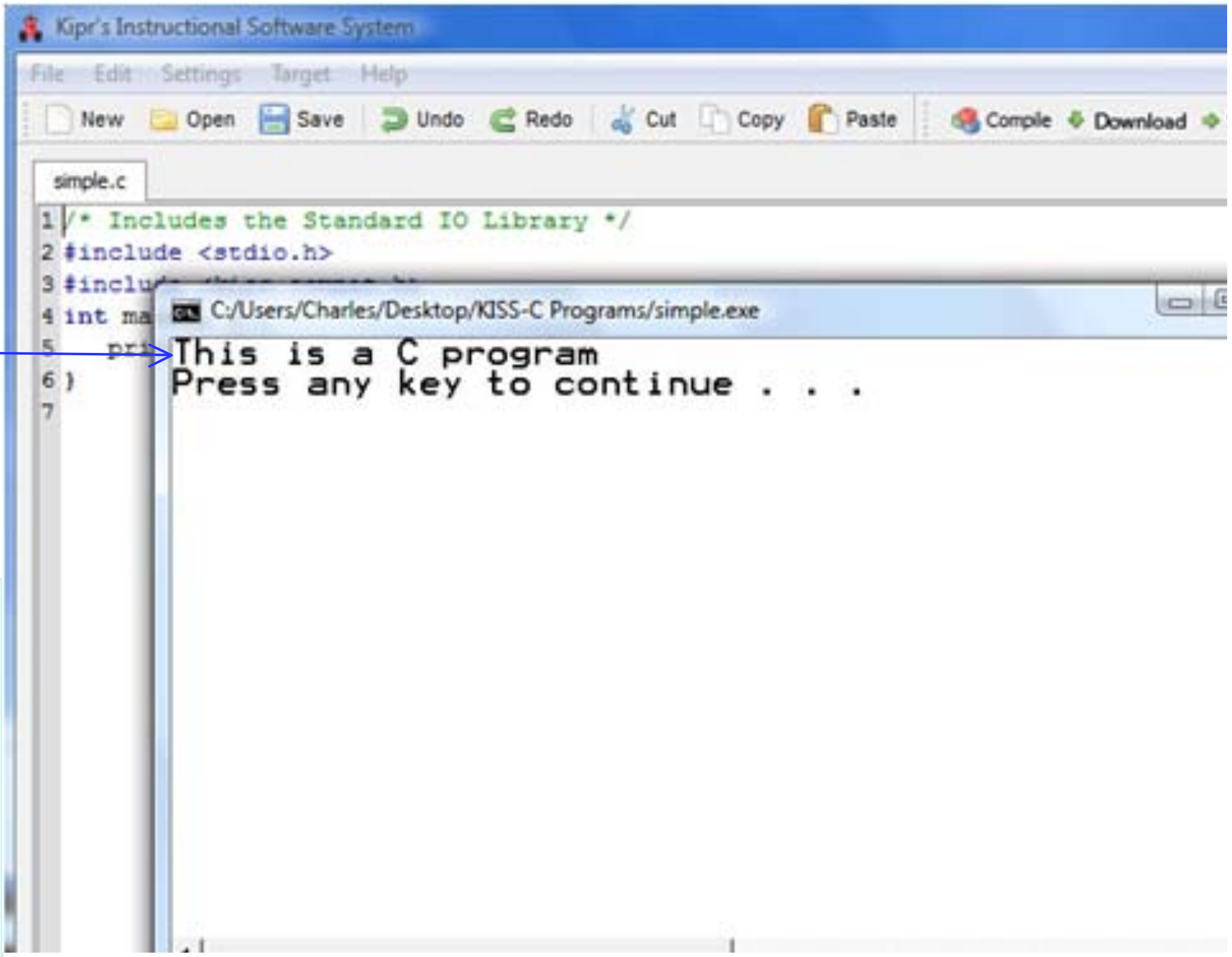
- The default directory for saving files is where KISS-C will save a previously unsaved file unless you change the directory specification
  - Files that have been previously saved will be updated by *Save* and remain in their current directory
- The default directory for saving files is established by either saving a previously unsaved file or by using *Save As*
- Similarly, the default directory for loading files is the directory KISS-C most recently loaded from
- Your working directory is where you have files you are currently working on
  - You usually want this to also be the default for loading or saving

Simulating your Program

- To simulate what your program does on the CBC simply click the *Simulate* button
  - Your program will launch a window on your PC to show your program results

Observing Results

Observed result


Kipr's Instructional Software System
File Edit Settings Target Help
New Open Save Undo Redo Cut Copy Paste Compile Download
simple.c
C:/Users/Charles/Desktop/KISS-C Programs/simple.exe

```
/* Includes the Standard IO Library */
#include <stdio.h>
#inclu
int ma
    pri
}
```

```
This is a C program
Press any key to continue . . .
```

Note: Depending on your operating system, you may be able to change the default properties for the display window opened by KISS-C (i.e., to have a white background and black text). For Windows XP or Vista, right click on the window title bar and select *Defaults;* subsequent display windows will use the new default values. How and whether this can be done varies among operating systems.

Returning to KISS-C

- You can resume your work in KISS-C even while you have a program running
- You can't simulate or download the program again until the current run is done (on Windows)
  - You can work on another program

The Program Explained

(it illustrates most C syntax)

Function Names

```
int main()
{
    printf("This is a C program\n");
}
```

*Every **C** program must have exactly one function named ‘***`main`***’. As a function,* **`main`** *returns a value, which by convention is an integer, specified by starting its definition with* **`int`***. The return value from* **`main`** *is only of potential use to the operating system, since when* **`main`** *has finished your program is done. Some systems may give a warning message if* **`main`** *does not contain a* **`return 0;`**

Blocks of Code

```
int main()
{
    printf("This is a C program\n");
}
```

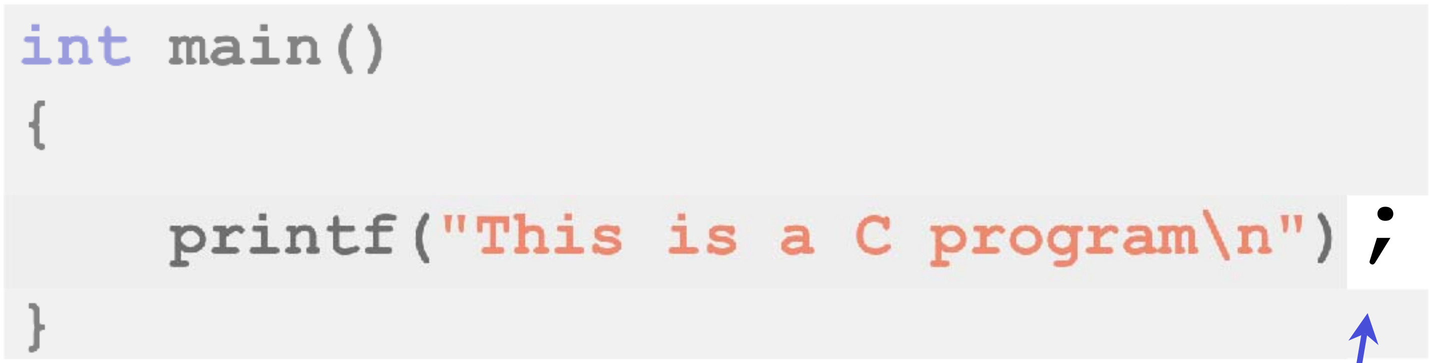
The braces '{ ' and '} ' act as a wrapper that contains a code block

Function Calls

```
int main()
{
   printf("This is a C program\n");
}
```

To call a function, just write its name and a set of parentheses. If there are any arguments, put them in the parentheses, separated by commas. Note that unlike **`printf`**, **`main`** *has no arguments, but you still have to include the parentheses.* **`printf`** *is a function made available to your program by KISS-C.*

Terminating Statements



```
int main()
{
    printf("This is a C program\n");
}
```

All C statements end with a '`;`'
Most statements are either a function call (like this **`printf()`***) or an assignment statement.*

Programming Process

How to Program

- Think about and explicitly state goal
  - Verbalizing the goal will help you to understand and refine it.
- Collect Requirements:
  - Identify "specs" or specifications – those things that are to be done (independent of solution method)
- Break large goals down into smaller sub-goals
  - Break goals down to smaller parts until each part is easy to think about as a set of program statements.
- Develop: Iterate and refine until done:
  - create a small part of the overall program
  - Be sure to document!
  - test syntax
  - test functionality/logic
  - When it works add the next part

Exercise #1a: Step by Step

- Goal: Create a program that prints your name on the screen.
- Start KISS-C, choose the CBC target, and click the *New* button
- Type in a program that prints your name and save it
- Check your program by clicking on the *Simulate* button
- If there is an error, KISS-C reports the line numbers where it is encountering problems
  - Usually, you need to address the first error before anything else
  - You can use the *Edit .. Goto Line* drop down menu to get to the problem
  - Note: the error occurs on OR before that line
- Once debugged, run your program by clicking the *Simulate* button
  - Clicking on *Simulate* automatically saves the most recent version of your program (even if there is an error in it)

Debugging: Syntax & Logic

- C *syntax* is prescribed by a formal grammar that provides the construction "rules" for C programs
  - When you press the *Simulate* button, KISS-C checks to make sure your program is legal
  - If you have a syntax error, it gives an error message
  - The line number of the error message indicates where KISS-C realized something was wrong
  - The actual typo is on that line **or before**
- C *logic (or semantics)* determine the interpretation of a statement
- The program will do what you tell it to do, not necessarily what you meant for it to do
- Fixing program syntax & semantics is the major part of the exercise programmers call "debugging"

Exercise #1b: Experiment a Little

- Try adding more **`printf()`** statements to your program (pay close attention to the syntax, particularly the terminating semi-colon needed by each statement)
- Have your program print out a haiku about robotics
- Run your revised program (*Simulate* button)
- Experiment by leaving off or adding extra "**`\n`**" to the end of the strings in your **`printf()`**

What's Being Simulated?

- The CBC target represents the CBC robot controller
- Programs written for the simulator will also work on the CBC hardware
- Programs written that make use of Create commands (coming up soon), will be visualized on the simulator, and will have similar behavior when downloaded to a CBC connected to a Create

CBC

Create

Built-in CBC Functions

- The CBC target has lots of built-in functions -- special capabilities, written by other people, that you can use in your programs,
- **`printf`** is an example of such a function.
- **`sleep`** is another such function. **`sleep`** pauses your program for however many seconds you give as an argument, e.g.,
  - **`sleep(5);`** pauses your program for five seconds.
- **`msleep`** does the same thing, but pausing for milliseconds, e.g.,
  - **`msleep(250);`** pauses your program for 250 milliseconds (or 1/4 of a second).

Exercise #2

- Open your program from exercise #1 and save it (use the *File .. Save As* menu item) in a file called exercise2.c
  - So you don't overwrite your exercise1.c file!
- Add some additional print statements to your program, but use **`sleep`** or **`msleep`** statements in between the **`printf`** statements, e.g.;

```
printf("Hi ");
sleep(3);
printf("there.");
```

- Click on the *Simulate* button to see how your program behaves

More Built-In Functions

- **`kissSim_init`**: opens up a graphical simulator of the CBC and the Create robot base. This function takes 4 arguments -- a simulated world and the X, Y and Θ of the initial position of the robot in the simulated world, e.g.;

```
kissSim_init(BB09WORLD,151,32,1.5708);
```

  - A Θ (radian measure) value of 0 has the robot pointing east, π/2 south
  - **Important**: you need to close out the graphical simulator before resuming your work in KISS-C
    - Otherwise your program, seemingly mysteriously, won't compile!
- **`kissSimPause`**: this pauses the simulation until the user hits the space bar, e.g.;

```
kissSimPause();
```

More Built-In Functions (2)

- **`create_connect`**: connects the CBC to the Create robot base, e.g.;

  ```
  create_connect();
  ```

- **`create_drive_straight`**: causes the Create to move at a given velocity (measured in mm per second), e.g.;

  ```
  create_drive_straight(200);
  ```

  - The speed range is -500 to +500

- **`create_stop`**: causes the Create to stop moving, e.g.;

  ```
  create_stop();
  ```

- **`create_disconnect`**: disconnects the CBC from the Create robot base, e.g.;

  ```
  create_disconnect();
  ```

- **`create_sensor_update`**: updates the Create's sensor values and makes them available to the CBC, e.g.;

  ```
  create_sensor_update();
  ```

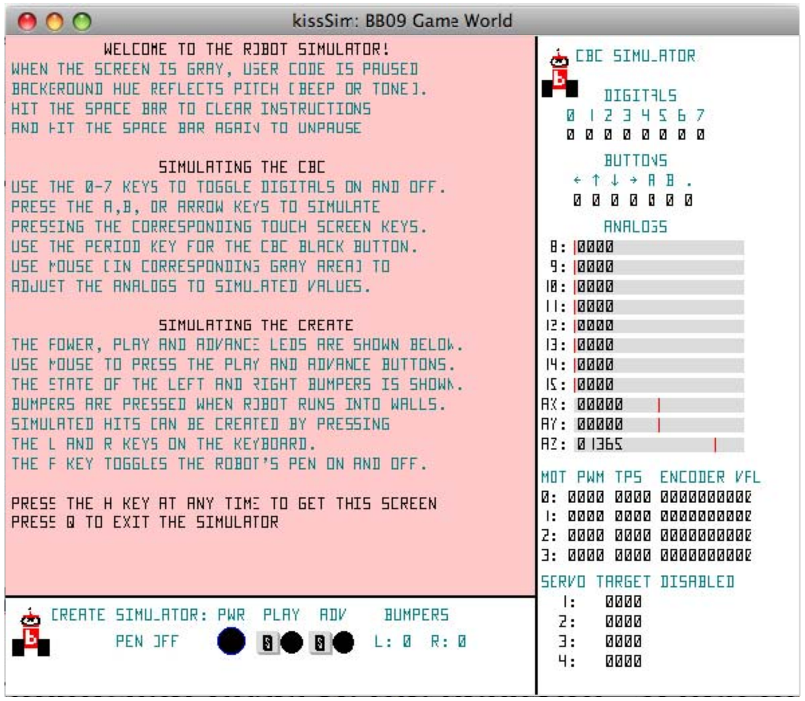
Exercise #3

- Open up a new file in KISS-C, save it as exercise3.c
- Write a program that:
  1. starts up the graphics simulator window,
     - Hint: use **`kissSim_init(BB09WORLD,151,32,1.5708);`**
  2. connects to the Create,
  3. drives straight at 200mm/second,
  4. sleeps for 10 seconds,
  5. stops the Create,
  6. disconnects from the Create, and
  7. pauses the simulator
- Click on the simulate button to see how your program behaves.
  – Read the on-screen instructions when the graphics window comes up.
  – You may have to click on the graphics window before it will listen to your keyboard

Exercise #3: Example (1)

`kissSim_init(BB09WORLD,151,32,1.5708);`

```c
int main()
{
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
    create_drive_straight(200);
    sleep(10);
    create_stop();
    create_disconnect();
    kissSimPause();
}
```



```
            WELCOME TO THE ROBOT SIMULATOR!
WHEN THE SCREEN IS GRAY, USER CODE IS PAUSED
BACKGROUND HUE REFLECTS PITCH [BEEP OR TONE].
HIT THE SPACE BAR TO CLEAR INSTRUCTIONS
AND HIT THE SPACE BAR AGAIN TO UNPAUSE

                 SIMULATING THE CBC
USE THE 0-7 KEYS TO TOGGLE DIGITALS ON AND OFF.
PRESS THE A,B, OR ARROW KEYS TO SIMULATE
PRESSING THE CORRESPONDING TOUCH SCREEN KEYS.
USE THE PERIOD KEY FOR THE CBC BLACK BUTTON.
USE MOUSE [IN CORRESPONDING GRAY AREA] TO
ADJUST THE ANALOGS TO SIMULATED VALUES.

                 SIMULATING THE CREATE
THE POWER, PLAY AND ADVANCE LEDS ARE SHOWN BELOW.
USE MOUSE TO PRESS THE PLAY AND ADVANCE BUTTONS.
THE STATE OF THE LEFT AND RIGHT BUMPERS IS SHOWN.
BUMPERS ARE PRESSED WHEN ROBOT RUNS INTO WALLS.
SIMULATED HITS CAN BE CREATED BY PRESSING
THE L AND R KEYS ON THE KEYBOARD.
THE F KEY TOGGLES THE ROBOT'S PEN ON AND OFF.

PRESS THE H KEY AT ANY TIME TO GET THIS SCREEN
PRESS Q TO EXIT THE SIMULATOR

CREATE SIMULATOR: PWR   PLAY   ADV     BUMPERS
         PEN OFF     ●    0 ●   0 ●   L: 0  R: 0
```

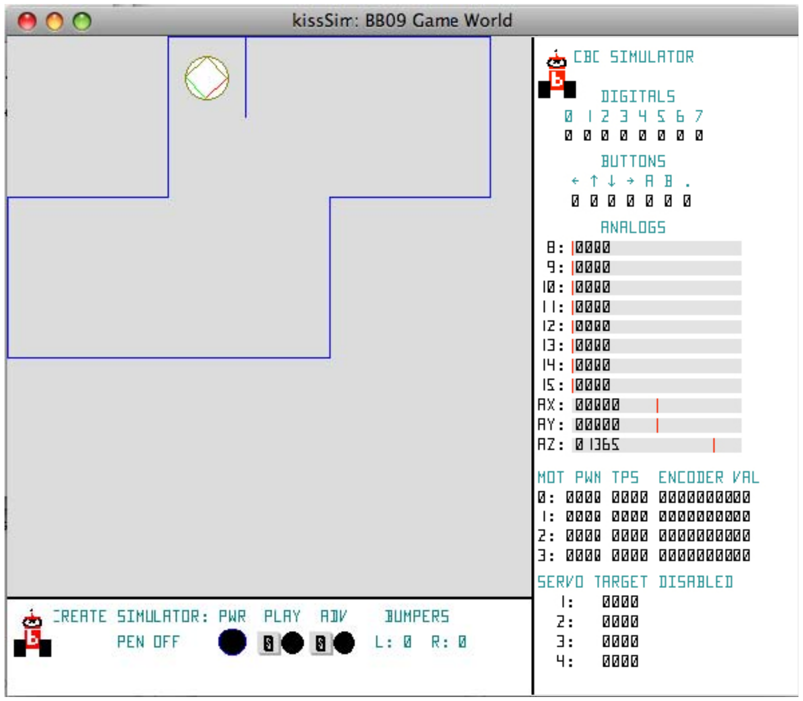
```
CBC SIMULATOR

      DIGITALS
  0 1 2 3 4 5 6 7
  0 0 0 0 0 0 0 0

      BUTTONS
   ← ↑ ↓ → A B .
   0 0 0 0 0 0 0

      ANALOGS
 8: 0000
 9: 0000
10: 0000
11: 0000
12: 0000
13: 0000
14: 0000
15: 0000
AX: 00000
AY: 00000
AZ: 01365

MOT PWM  TPS  ENCODER VEL
0:  0000 0000 0000000000
1:  0000 0000 0000000000
2:  0000 0000 0000000000
3:  0000 0000 0000000000

SERVO TARGET DISABLED
  1:   0000
  2:   0000
  3:   0000
  4:   0000
```

This line of the code creates the simulator window, and brings up this help screen. Select the **kissSim: BB09 Game World** window by clicking on it, and then press the space bar to get to the next screen.

Exercise #3: Example (2)

`kissSim_init(BB09WORLD,151,32,1.5708);`

```
int main()
{
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
    create_drive_straight(200);
    sleep(10);
    create_stop();
    create_disconnect();
    kissSimPause();
}
```



```
CBC SIMULATOR
DIGITALS
0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0
BUTTONS
← ↑ ↓ → A B .
0 0 0 0 0 0 0
ANALOGS
 8: 0000
 9: 0000
10: 0000
11: 0000
12: 0000
13: 0000
14: 0000
15: 0000
AX: 00000
AY: 00000
AZ: 01365
MOT PWN  TPS  ENCODER VAL
0:  0000 0000 0000000000
1:  0000 0000 0000000000
2:  0000 0000 0000000000
3:  0000 0000 0000000000
SERVO TARGET DISABLED
 1:   0000
 2:   0000
 3:   0000
 4:   0000
```

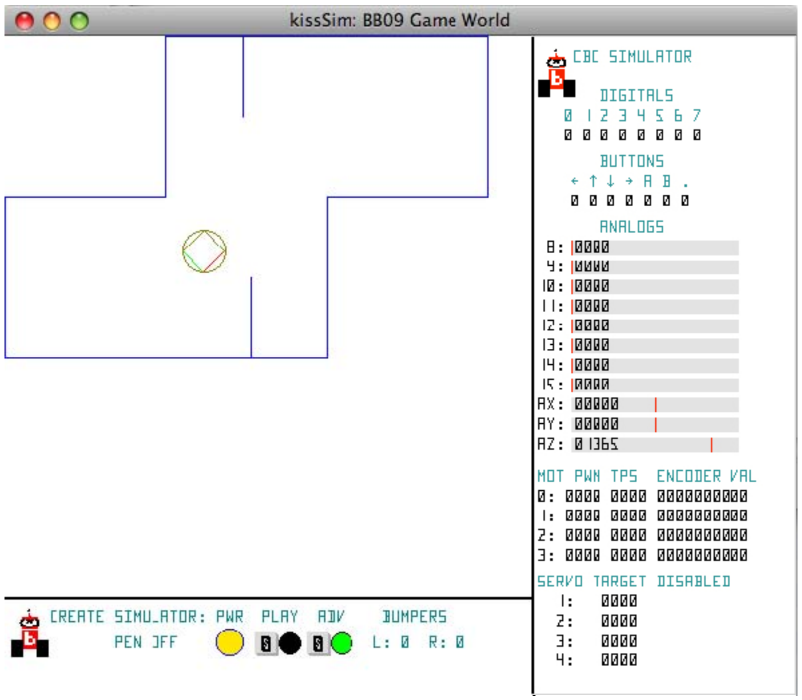
```
CREATE SIMULATOR: PWR  PLAY  ADV   BUMPERS
       PEN OFF     ●   0●    0●   L: 0  R: 0
```

Your program has not yet advanced, but now you can see the world. Note that the gray background indicates that your program is paused. The simulator always starts with your program paused. Press the space bar to start your program running.

Exercise #3: Example (3)

`create_drive_straight(200);`

```
int main()
{
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
    create_drive_straight(200);
    sleep(10);
    create_stop();
    create_disconnect();
    kissSimPause();
}
```

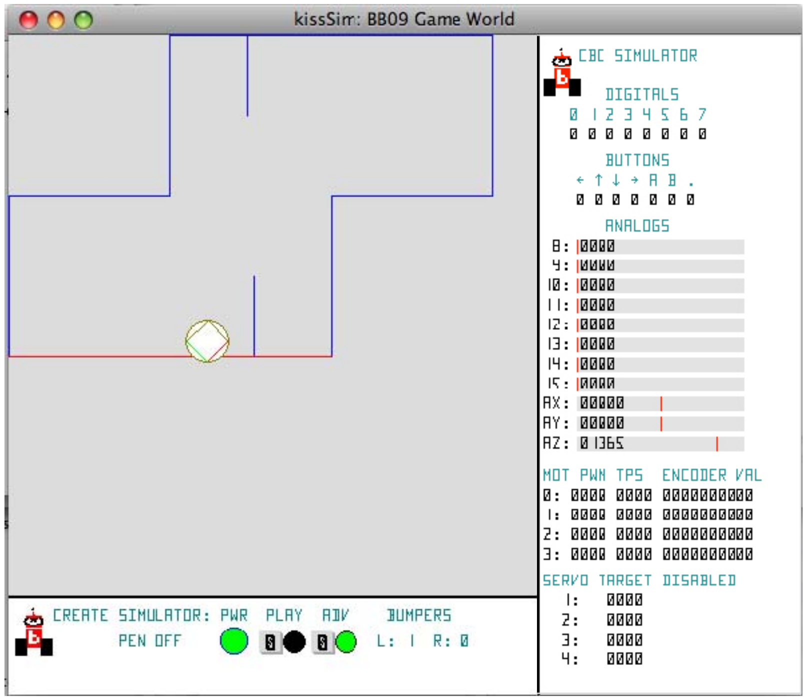


Your program has started. The PWR light turns yellow indicating that the Create is connected to the CBC and in Safe mode (orange indicates that it is in Full mode). The ADV light turns green (default setting when the Create is turned on). Within a millisecond, your program has reached the **`sleep`** statement. The robot continues moving forward until the 10 seconds are up.

Exercise #3: Example (4)

`create_disconnect();`

```
int main()
{
   kissSim_init(BB09WORLD,151,32,1.5708);
   create_connect();
   create_drive_straight(200);
   sleep(10);
   create_stop();
   create_disconnect();
   kissSimPause();
}
```



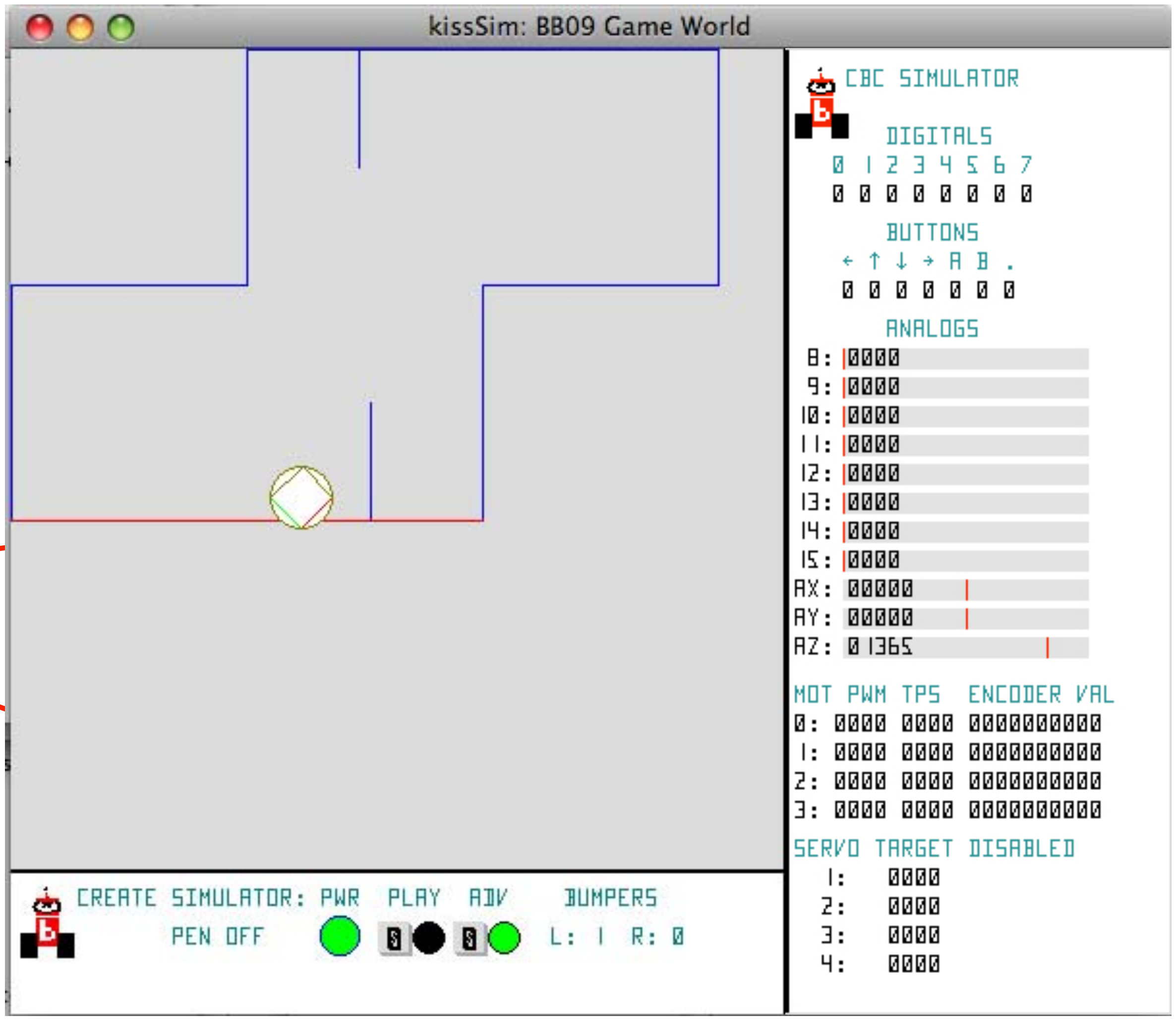
```
DIGITALS
0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0
BUTTONS
← ↑ ↓ → A B .
0 0 0 0 0 0 0
ANALOGS
 8: 0000
 9: 0000
10: 0000
11: 0000
12: 0000
13: 0000
14: 0000
15: 0000
AX: 00000
AY: 00000
AZ: 01365
MOT PWN TPS  ENCODER VAL
0: 0000 0000 0000000000
1: 0000 0000 0000000000
2: 0000 0000 0000000000
3: 0000 0000 0000000000
SERVO TARGET DISABLED
  1:   0000
  2:   0000
  3:   0000
  4:   0000
```

```
L: 1  R: 0
```

The PWR light turns green indicating that the Create is disconnected from the CBC. The bottom wall has turned red indicating that the robot has collided with the wall (note that the robot has gone part way through). The background has turned gray because your program is paused. Hit the space bar to unpause (which will exit the program and close the graphics window).

CBC Simulator: Create Sensors

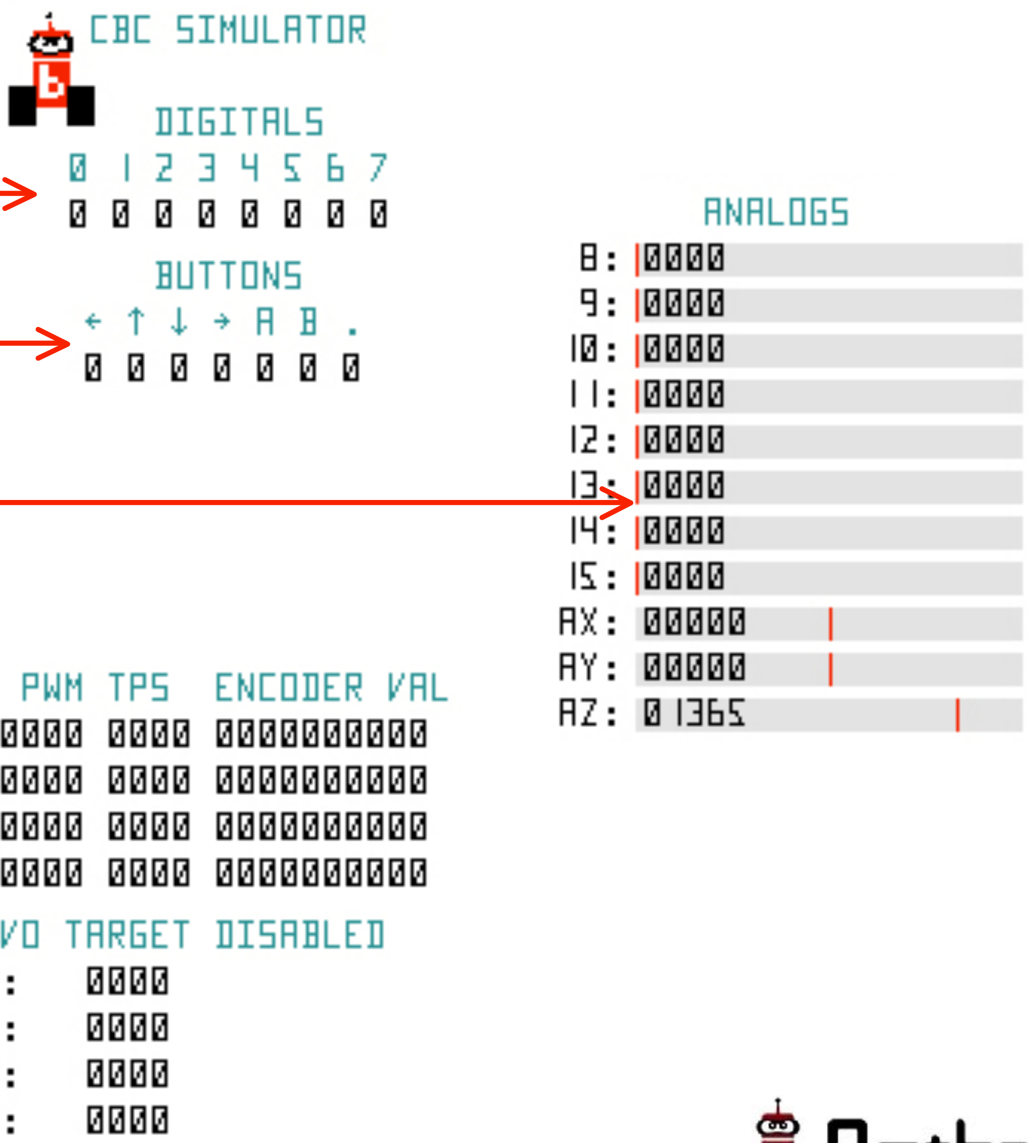
- Two simulated touch sensors correspond to the left and right front bumper on the Create module
  - **`gc_lbump`** and **`gc_rbump`** get the current sensor values for the bumpers when **`create_sensor_update()`** is called
- If the simulator detects the robot running into an obstacle (graphically, a blue line), the appropriate sensor becomes pressed and the obstacle changes color
  - Clicking on the L or R key will activate/deactivate the bumpers
- Note that in the simulation the robot will go right through the wall unless your program tells it to do something else!



```
CBC SIMULATOR

DIGITALS
0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0
BUTTONS
← ↑ ↓ → A B .
0 0 0 0 0 0 0
ANALOGS
 8: 0000
 9: 0000
10: 0000
11: 0000
12: 0000
13: 0000
14: 0000
15: 0000
AX: 00000
AY: 00000
AZ: 01365

MOT PWM  TPS  ENCODER VAL
0:  0000 0000 0000000000
1:  0000 0000 0000000000
2:  0000 0000 0000000000
3:  0000 0000 0000000000
SERVO TARGET DISABLED
 1:    0000
 2:    0000
 3:    0000
 4:    0000

CREATE SIMULATOR: PWR  PLAY  ADV    BUMPERS
         PEN OFF   ●   0 ●   0 ●    L: 1  R: 0
```

CBC Simulator: Sensors & Motors

- The simulator also provides controls to set simulated values for digital and analog sensors
  - Digital sensors are activated/deactivated by pressing a number key, buttons by an arrow, letter, or period (for CBC black button)
  - Analog sensors are changed by using your mouse (click and hold) to drag a thin red settings bar right or left
- If Create motor commands are used, the simulator moves the simulated robot in a manner corresponding to the commands
- For non-Create motors, current motor values set by your program are displayed



```
CBC SIMULATOR

   DIGITALS
0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0

   BUTTONS
← ↑ ↓ → A B .
0 0 0 0 0 0 0

        ANALOGS
 8: 0000
 9: 0000
10: 0000
11: 0000
12: 0000
13: 0000
14: 0000
15: 0000
AX: 00000
AY: 00000
AZ: 01365

MOT PWM  TPS  ENCODER VAL
0:  0000 0000 0000000000
1:  0000 0000 0000000000
2:  0000 0000 0000000000
3:  0000 0000 0000000000

SERVO TARGET DISABLED
 1:    0000
 2:    0000
 3:    0000
 4:    0000
```

Documentation: Comments

- Explanatory comments
  - Can (and should) be added to a program to assist you and your teammates in understanding it later
  - Do not affect the size or performance of the compiled program
- Syntax
  - In-line form: `//` *comment text*
    - The comment ends when a new line is started
  - Multi-line form: `/*` *comment text* `*/`
    - The comment starts with an initial "`/*`" and continues until "`*/`" is encountered
- Using comments to document program changes
  - Particularly useful for
    - A large evolving program
    - Programs being modified by more than one person

```
// simple.c - (c) 2009 David Miller, KIPR
// This program displays a simple character string
/* History:
   Modified 01/7/09 - adjusted comments- dpm
*/

int main() {
    printf("This is a C function\n"); // display the string
}
```

Style: Indentation

- C ignores most white space (spaces, returns, tabs, blank lines)
- Indenting C program text helps to bring out the structure of your program – this is an aspect of programming *style* for improving readability
  - Uniformly indent program text within each program block
    - Start a new indentation after each '{'
    - Shift indentation back to left after each '}'
  - Indent the second line of a single statement (exception: any added white space inside a quoted string will print!)
- KISS-C's built in editor can do most indentation for you!
  - The KISS-C *Edit* menu provides commands for indenting your text

Exercise #4

- After finishing exercise #3, use *Save as* to save the file as exercise4.c and modify the program so it looks like this:

```
int main()
{
  kissSim_init(BB09WORLD,151,32,1.5708);
  create_connect();
  gc_distance = 0;// add this line
  create_drive_straight(200);
  sleep(10);
  create_stop();
  create_sensor_update(); // add this line
  printf("Robot traveled %dmm\n",gc_distance); // add this line
  create_disconnect();
  kissSimPause();
}
```

- Click on simulate and observe the behavior. When the simulator pauses at the end, look at the text window.

gc_distance

- The lines that were added in Exercise #4 manipulate **`gc_distance`**.
- **`gc_distance`** is a variable, which holds an integer value that corresponds to how far, in mm, the robot has traveled.
- Variables are a standard feature of almost all programs.
- **`gc_distance`** is built into the CBC target environment, but you can create variables of your own to hold numbers that your program will work with.

Variables and Data

Variables

- Variables are boxes where data can be stored and retrieved.
- A variable's name can be anything you want, as long as the name starts with a letter and contains only letters, digits and the under score character _.
- A variable's name should be something that relates to how it will be used.
- When a variable is first created (declared), the type of data that it will hold needs to be specified.

Numeric Data in C

- There are two types of numbers commonly used in C
  - Integer
    - a 32 bit binary number in the range -2,147,483,648 to +2,147,483,647
      - On some systems 64 bit integers may also be supported
  - Floating point numbers (fractions)
    - a 32 bit or 64-bit representation in "scientific notation" for numbers such as 3.141659
    - The number of bits used determines the precision of the representation for non-determining decimals such as π
- If mixed data types are used in a computation, C converts arguments to the most complex type for the computation
  - There are built-in means to force how a type conversion is done
- Integer arithmetic is handled by hardware circuits, so it is fast
- Floating point arithmetic is more complex and absent floating point hardware is handled by software, making it slower than integer arithmetic (although not slow in comparison to humans!)

C Variables & Data Types

- Variables retain data for later use
  - Variables are employed in constructions such as arithmetic expressions
- Each variable represents a location in computer memory, so its type must be specified for C to know how it is to be interpreted
- Syntax: *<data-type> <variable-name>* `;`
- Numeric data types
  - `int numOfWheels;`
    - specifies **`numOfWheels`** to be an integer variable
    - 32 bit memory location
    - Range -2,147,483,648 to +2,147,483,647
  - `float radius;`
    - specifies **`radius`** to be a floating point variable
    - 32 bit memory location to be interpreted as representing a number in "scientific notation" (exponent & mantissa format)
- There are also non-numeric data types (see documentation)

Integer Arithmetic (`int`)

- `+` is used for addition

  `X + Y` means X plus Y
- `-` is used for subtraction

  `X - Y` means X minus Y
- `*` is used for multiplication

  `X * Y` means X times Y
- `/` is used for division

  `X / Y` means X divided by Y with the decimal truncated
- `%` is used for modulus (when the arguments are positive, this is just the remainder)

  `X % Y` means the remainder when X is divided by Y (assuming that X and Y are positive)

Floating Point Arithmetic

- The result of an operation on floating point numbers is a value of type `float`
- Any integers used must have a decimal point so that they are of type `float` (e.g., 3.0 instead of 3)
  - The `+`, `-`, `*` operations work as before except that the result is a `float` (retains a decimal fraction, even if it is 0)
  - With floating point arguments the `/` operation retains the decimal fraction rather than truncating it
- **%** (remainder operator) is not defined for floating point numbers

Using Variables

- Variables only exist inside the block of code (defined by the **{ }**) in which the variable's declaration statement exists.
- To aid in program readability, declare all of your variables for your function at the beginning of the function (in the line(s) immediately after the opening **{**).
- The value of a variable is undefined until some value has been specifically assigned to it.
- It is a good idea to initialize your variables after you have defined them, e.g.;

```
int myVariable;
float myOtherVariable, anotherVariable;
myVariable = 0;
myOtherVariable = 2.5;
anotherVariable = 3.0;
```

- In **C**, you can combine the declaration and initialization statements; e.g.,

```
float myOtherVariable=2.5, anotherVariable=3.0;
```

Displaying Variable Values using `printf();`

- The values of variables used within a program can be displayed using the C library function **`printf`**
- Syntax: **`printf(`**<text-string>**`,`** <arg1>**`,`** <arg2>**`,`** **`...);`**
- For each argument, a corresponding "% code" is embedded as a "placeholder" within the <text-string>, to be replaced by the value of the argument when the **`printf`** is executed
  - Example:

```
printf("volts=%f R=%d\n", voltage, resistance);
```

  - **`%d`** is used to correspond to an integer argument (**`int`**)
  - **`%f`** is for a floating point argument (**`float`**)
  - **`\n`** positions you at the start of the next line in the display window
  - Documentation has a more complete description of **`printf()`**

Getting Data into Variables (recap)

- A data value can be assigned to a variable when it is declared

```
int j=10;          //integer variable j, initialized to 10
float pi=3.1416;   /*floating point variable, initial value
                    approximately π */
```

- A data value can be assigned to a variable as part of program execution

```
int i,j=10; //integer variables i and j, j initialized to 10
i = 2;      //assign (or store) the value 2 in i
```

- Variables can be used in expressions and assigned new values
  - Continuing the previous example

```
int i,j=10; //integer variables i and j, j initialized to 10
i = 2;      //assign (or store) the value 2 in i
i = i + j;  //compute i+j and store the value in i
```

  - What is the value of `i` now?

Exercise #5

- Use *Save As* to save Exercise #4 in a file called exercise5.c
- Right after the **`{`** in your main function, declare an **`int`** variable named **`cmDist`** and a float variable named **`inchDist`**.
- Right after the **`printf`** statement, add a couple statements that fill these variables with the correct values, e.g.,

```
inchDist=gc_distance/25.4; //convert dist in mm to inches
```

- Write a similar statement to convert mm to cm (divide the value in mm by 10)
- Add a print statement that prints out the distance in cm and inches, so that if your robot had moved 1817mm it would print out: **`Robot traveled 181cm or 71.5354inches.`**
- Remember to use **`%d`** to print out an **`int`** variable or value and **`%f`** for float variable or value.

Program Flow/Repetition

Program Flow

- When a program is run, the control moves from one statement to the next
- Calculate $j^2 + 1$ when $j = 3$

```
int main()
{
   int r,j;         // declare r and j
   j = 3;           // assign a value to j
   r = j*j + 1;     // calculate and assign
   printf("result is %d\n",r);
}
```

Program Flow (2)

- C offers other program flows than doing one step after another
- Sometimes you want your program to do something more than once:
  - do a task until a condition is true (move until robot has traveled 2 meters)
  - wait until some condition is true (don't start moving until a button is pushed)
  - repeat a step a specific number of times (wave arm 3 times)
- Example: Instead of having the robot travel for 10 seconds, let's have it travel until it has traveled 90cm (900mm). This distance would put it about halfway between the top and bottom blue lines.
- To do this, we want to check the value of **`gc_distance`** over and over again until it is greater or equal to 900, and then stop the robot
- A *while loop* is a control structure for repeating a series of statements while a condition is true

The `while` Statement

- Syntax: `while (`*<test>*`) {`*<statements>*`}`
  - The *test* is something that is either true or false
  - While the test is true, the block of statements are repeated
  - Each time the last statement in *statements* is completed, the *test* is checked
    - if the *test* is still true, the block of *statements* is repeated
    - if the *test* is false, the program skips past that block of *statements* and continues on
  - A true/false test is called a *Boolean Expression*

Boolean Expressions

- Boolean expressions result in either

  `0` *(false)* or `1` *(true)*

  [technically, **C** treats any non-zero value as true]
- Boolean operators:

  `==` (two equals signs together, not one)

  `<`, `<=`, `>`, `>=` (the usual comparators)

  `!=` (not equal)

  `||` (or), `&&` (and)

  `!` not
- The *while* statement uses boolean expressions:

  `while` (*<boolean expression>*)

Using Repetition to Check for Change

- The `while` statement is commonly used in robotics to wait for an event. e.g:

```
//wait till A is pressed
while(a_button()==0){msleep(10);}
```

or

```
//drive until left or right bumper is pressed
//check bumpers then update sensors and repeat
create_drive_straight(50);
create_sensor_update();
while(!(gc_lbump || gc_rbump)) {
   msleep(50); //let Create catch its breath
   create_sensor_update();
}
create_stop();
```

Some Things That Change on a Botball Robot

- Variables holding Create sensor values
  - **`gc_lbump`**
  - **`gc_rbump`**
  - **`gc_distance`**
  - **`gc_total_angle`**
  - **`gc_advance_button`**
  - **`gc_play_button`**
- Button functions for the CBC
  - **`a_button()`**
  - **`b_button()`**
  - **`left_button()`**
  - **`right_button()`**
  - **`up_button()`**
  - **`down_button()`**
  - **`black_button()`**

CBC Sensor Functions

- **`digital(`*`<port>`*`)`**
  - <port> is an `int` value corresponding to the location where the sensor is plugged in (0-7)
  - the function returns an `int` 1 or 0 (true or false) depending on the state of the sensor
- **`analog10(`*`<port>`*`)`**
  - <port> is an `int` value corresponding to the location where the sensor is plugged in (8-15)
  - the function returns an `int` value of 0-1023, depending on the state of the sensor
  - ports 13-15 are reserved for the range sensors (the ET and sonar sensors) which use a floating signal line
- **`accel_x()`**
  - returns an `int` value between -2047 to 2047 representing acceleration along the X axis (uses the CBC internal accelerometer)
  - a value of 1365 is approximately 1 G (9.8 m/sec^2 - force of gravity)
  - negative values indicate acceleration in the negative direction
- **`accel_y()`**
  - same as **`accel_x`**, but in the Y axis
- **`accel_z()`**
  - same as **`accel_x`**, but in the Z axis

Example: Using Repetition

- Syntax: **while (***<test>***) {***<statements>***}**
- Move 90cm

```
int main()
{
     kissSim_init(BB09WORLD,151,32,1.5708);
     create_connect();
     create_drive_straight(200);
     create_sensor_update();// get Create's current values
     gc_distance = 0; // reset gc_distance
     while(gc_distance < 900){// loop while this true
         msleep(50);// wait 1/20 sec between updates
         create_sensor_update();// update gc_distance
     }
     create_stop();
     printf("Robot traveled %dmm\n",gc_distance);
     create_disconnect();
     kissSimPause();
}
```

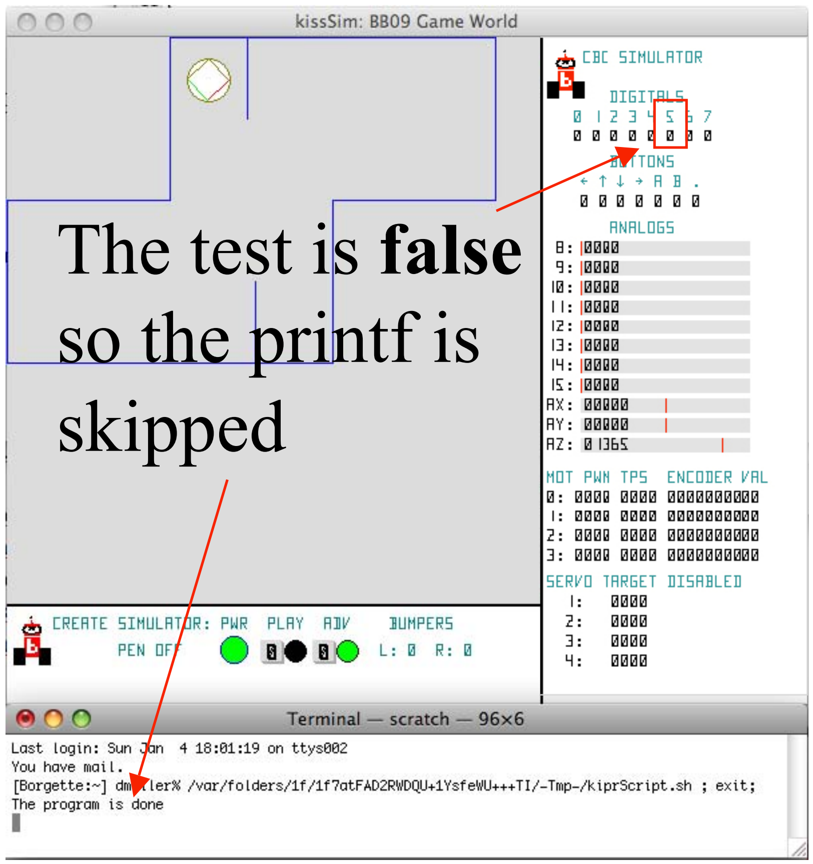
Conditional Execution
Making a Decision with `if`

- Syntax `if` (<*test*>) {*statements*}
- < *test* > is a boolean expression for testing whether or not a test criteria is met
- The statements are skipped if the *test* is false. e.g.:

```
int main()
{
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
    if(digital(5)==1) //is digital 5 on?
    {
       printf("digital 5 is on\n");//if true, print
    }
    create_disconnect();
    printf("The program is done\n");
    kissSimPause();
}
```

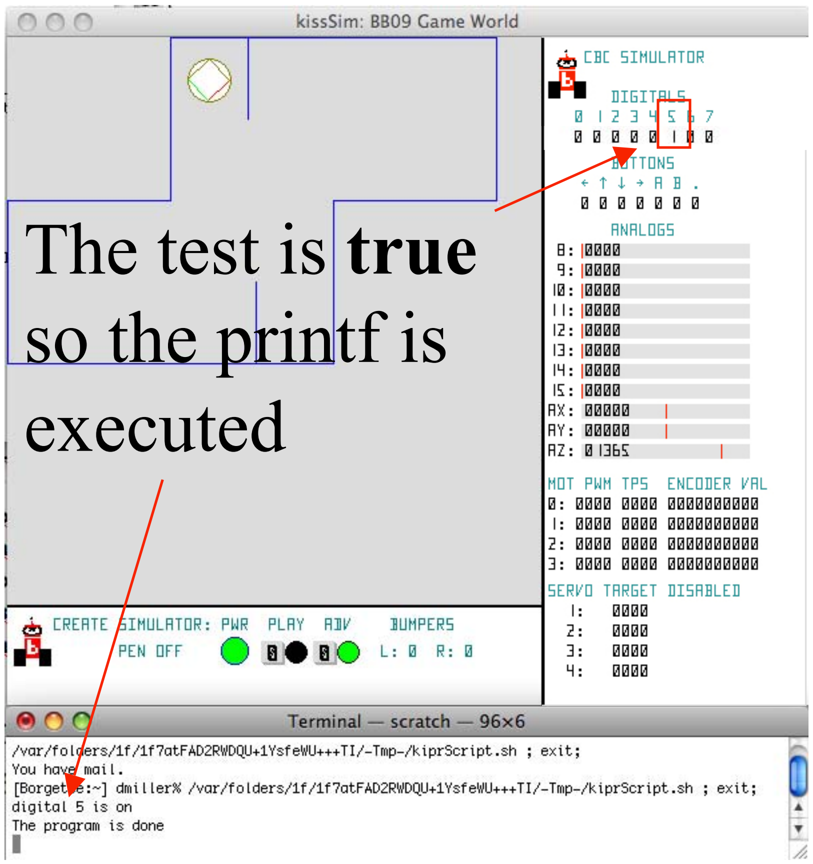
Example using `if`

- Run the program
- After help screen, press the 5 key to toggle digital 5 on and off
- Press space to execute program, and then space again to exit



The test is **false** so the printf is skipped

```
Last login: Sun Jan  4 18:01:19 on ttys002
You have mail.
[Borgette:~] dmiller% /var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
The program is done
```



The test is **true** so the printf is executed

```
/var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
You have mail.
[Borgette:~] dmiller% /var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
digital 5 is on
The program is done
```

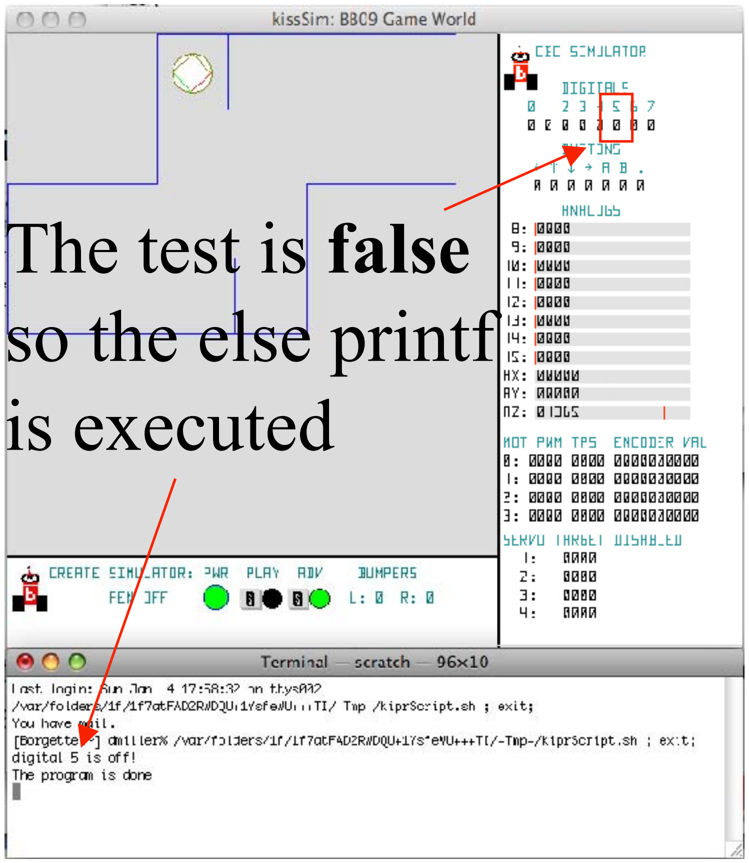
Conditional Execution
Making a Decision with `if else`

- Syntax `if` (<*test*>) {*statements*} `else` {*statements*}
- If *test* is true do first block of statements, if it is false do the second block of statements. e.g.:

```
int main()
{
   kissSim_init(BB09WORLD,151,32,1.5708);
   create_connect();
   if(digital(5)==1) //is digital 5 on?
   {
      printf("digital 5 is on\n");//if true, print
   }
    else //do this block when digital 5 is off
   {
      printf("digital 5 is off!\n");//if false, print
   }
   create_disconnect();
   printf("The program is done\n");
   kissSimPause();
}
```

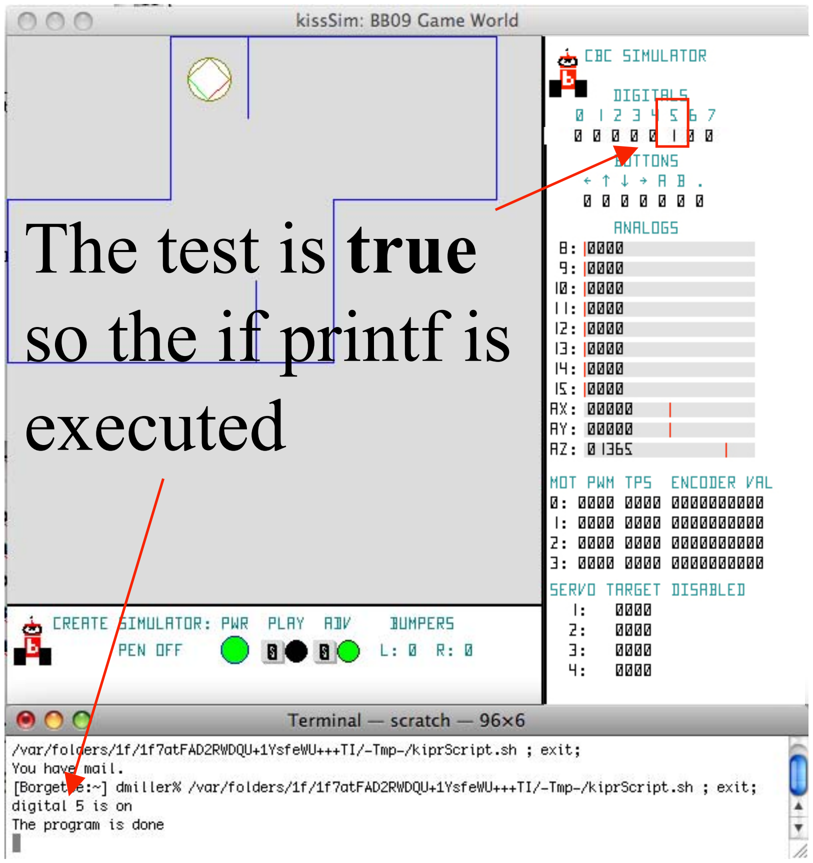
Example using `if else`

- Run the program
- After help screen, press the 5 key to toggle digital 5 on and off
- Press space to execute program, and then space again to exit



The test is **false** so the else printf is executed

```
Last login: Sun Jan  4 17:58:32 on ttys002
/var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
You have mail.
[Borgette:~] dmiller% /var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
digital 5 is off!
The program is done
```



The test is **true** so the if printf is executed

```
/var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
You have mail.
[Borgette:~] dmiller% /var/folders/1f/1f7atFAD2RWDQU+1YsfeWU+++TI/-Tmp-/kiprScript.sh ; exit;
digital 5 is on
The program is done
```

Example: if else

```
int main()
{
  kissSim_init(BB09WORLD,151,32,1.5708);
  create_connect();
  create_drive_straight(200);
  create_sensor_update(); //update values for bumpers before testing
  while(gc_lbump==0 && gc_rbump==0){//loop while not bumped
    msleep(50);//wait 1/20 sec between updates
    create_sensor_update();// update gc_distance
    if(digital(5)==1){create_play_led(1);}//if true, turn on LED
    else{create_play_led(0);}//turn off LED when digital 5 is off
    if(accel_z()<0) break;//if robot is upside down, exit loop
  } //end while
  create_stop(); //a bumper was pressed or robot flipped so stop
  printf("Robot traveled %dmm\n",gc_distance);
  create_disconnect();
  kissSimPause();
}
```

KISS-C Create Movement Functions

- To see all the KISS-C Create movement functions see the KISS-C help manual for the CBC target
- Here are major ones (some of which we've already used)
  - Speed in mm/sec, radii in mm
  - **`create_drive(int speed, int radius)`**
    - Drive the robot along a curve with given radius; positive radius turns left, negative right
    - Example: **`create_drive(300, -650);`**
  - **`create_drive_straight(int speed)`**
  - **`create_spin_CCW(int speed)`**
  - **`create_spin_CW(int speed)`**
  - **`create_drive_direct(int r_speed, int l_speed)`**
    - Right motor speed and left motor speed
  - **`create_stop()`**

Exercise #6

- Open a new file (press the *New* button on KISS-C window) and save it as exercise6.c
- Write a program for the robot that goes straight until it runs into something; if the left bumper is pressed the robot should turn clockwise for 2 seconds at speed 200, if the right bumper is pressed the robot should turn CCW for 2 seconds at speed 350.
- The robot should stop and your program exit if the digital port 0 ever has a value of 1.
- Hints:
  - Test digital port 0 for your while loop
  - Inside your loop, have two if statements, one of which is executed if **`gc_lbump==1`** and the other if **`gc_rbump==1`**
  - If the right bumper is pressed your if statement should: **`{create_spin_CCW(350); sleep(2);}`**

Where We're Headed

Example KISS-C Program for Robot Control

```
#include "my_lib.c"
int main() {
    int i=0;
    kissSim_init(BB09WORLD,151,32,1.5708); // initialize w/world
    create_connect();       // communicate with Create if present
    while(i<4) // do a square
    {
        printf("Moving Forward\n");
        my_move(600, 250);
        printf("Turn Left\n");
        my_turn(90, 100);
        i = i+1;
    }
    create_disconnect();    // stop communication with Create
    kissSimPause();         // end of simulation
}
```

Go 600 mm at 250 mm/sec and turn 90^o
Do 4 times to trace a square

- Idea: use similar logic to what we did for moving /turning a Create to construct functions and save them in our own function library and use them like any built-in function (a labor saving device!)

Functions in General

About Functions

- Remember your math functions?
  - A typical function
    - Area of a circle is a function of the radius
      - The "area function" for a circle is the Greek circle constant π times the radius r squared, or $A(r) = \pi r^2$
  - 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
    - $A(50) = 2500\pi$ which is approximately 7853.981634

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 can return nothing (is **`void`**) 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
  - For the area function this would appear in the form:

    function name *argument name*

    ```
    float circ_area(float r)
    ```

    data type returned *data type for the argument*

  - This is called the function's *prototype*, since it clues you as to how to use the function
  - You may have noticed that the documentation for each library function provides the function's prototype

Writing Your Own C Functions

- The general function syntax in C is

  <*type*> fn-name (<*type*> arg1 , <*type*> arg2 , …)

  {<*function-body*> }

- The area function requires an argument and returns the area of a circle that has the radius given by the argument

```
float circ_area(float r) {
   float pi = 3.14; // approx value of pi
   return(pi * r * r);
}
```

- Create a new file, copy the **circ_area** function into KISS-C, and save as "**area_test.c**"
- Test your function
  - Add a **main** function to "**area_test.c**"

```
int main()
   {printf("area = %f\n",circ_area(50.0));}
```

Note the effect approximating π has on the result!

(compare your answer to the better approximation of 7853.981634)

About `#include`

- `#include` is used to instruct KISS-C to add the contents of some other file (a "library") to your program when you compile it
- KISS-C automatically includes most C and Botball libraries having functions you are likely to need (such as **`printf`** and **`msleep`**)
- If the file name is enclosed in quotes, KISS-C will look for the file in your default input directory (normally your working directory)
  - Or you can specify the file path for KISS-C to use
  - If "pointy" brackets (**<>**) enclose the file name, KISS-C will look for the file in the system directory

Setting Up a Function Library

- We want a function to move the Create a given distance, so we should name it; e.g., "my_move", and use an argument for how far in mm to move and second argument for what speed
  - This is a function "spec"
- Since we want our move function and a turn function in the same library, we should use a representative file name
  - Click on *New* in KISS-C to start a new file
  - *Save* as **`my_lib.c`**
- This is a library, **`main`** should not be defined in this file!
  - Other programs may also need these functions
- The library can have as many functions defined in it as you wish

Move Function: A First Version

```
void my_move(int x, int vel) { // move x mm, vel in mm/sec
    create_sensor_update(); // update Create sensor readings
    gc_distance=0;       // and initialize distance variable
    create_drive_straight(vel);
    while(gc_distance < x)
    {
        create_sensor_update();
        msleep(50); // pause between sensor updates
    }
    create_stop();  // stop
}
```

- Comments: What if you try to use this function to move backwards? (negative velocity)
  - An opportunity to write a more general version!

Exercise #7: Turning

- We know we can get the Create to turn in place by using **`create_spin_CCW`**
- The variable **`gc_total_angle`** keeps track of how many degrees the Create has turned through since it was initialized
  - We can use it in a manner analogous to **`gc_distance`**!
- So, we want a function to turn the Create a given number of degrees, so we should name it "my_turn" and use as an argument for how far in degrees to turn and a second argument for what speed
- It should be added to your library file with the my_move function
- Test your library with the program on the next slide

A Program That Uses the Library

```
#include "my_lib.c"
int main() {
    int i=0;
    kissSim_init(BB09WORLD,151,32,1.5708); // initialize w/world
    create_connect();      // communicate with Create if present
    while(i<4) // do a square
    {
        printf("Moving Forward\n");
        my_move(600, 250);
        printf("Turn Left\n");
        my_turn(90, 100);
        i = i+1;
    }
    create_disconnect();   // stop communication with Create
    kissSimPause();        // end of simulation
}
```

Add in your library functions

Use your move function!

Use in your turn function!

So Why Use Libraries?

(after all you could just put all of the code in the same file)

- Try this scenario: Each of Mary, John, and Sue are writing functions for move and turn, so they put them in libraries:
  “Mary.c”, “John.c”, and “Sue.c”
- So long as Mary, John, and Sue follow the naming conventions for move and turn (“my_move” and “my_turn”), you can use their work with the program you just finished simply by changing in turn

```
#include "my_lib.c"  to #include "Mary.c"
                     to #include "John.c"
                     to #include "Sue.c"
```

and running the program for each

Exercise #8 DemoBot Build

Instructions are on your workshop flash drive

END OF DAY 1

Day 2

Tutorial Schedule:

- Day 1:
  - Robots, Botball and design projects
  - Botball related activities
  - KISS-C programming
  - Your first programs
    - *kissSim* simulator
    - CBC and Create
    - Simulated robot
  - Variables and data in **C**
  - Program flow/repetition
  - Create movement functions
  - **C** functions in general
    - Building your own library
    - Move and turn
  - Demobot Build
- Day 2:
  - Game Review
    - Robot Documentation
    - Building Rules
    - Game Challenge
  - Using tournament software
  - About the CBC
  - CBC Color Vision
  - Sensors, Create, Motors
  - Using Demobot
  - Motor functions and robot programming
  - Project design
  - Checkout and wrap up

Game Session

- Instructor will reveal password at the end of the game presentation

Utilities for Botball
`wait_for_light`

- In Botball robots start when signaled by a starting light and have to stop 120 seconds later
- The function **`wait_for_light`**`(`*<port>*`)` for the CBC runs a calibration function for the light sensor you have plugged into *<port>*
  - Calibration determines what sensor reading corresponds to the light being on and what corresponds to it being off
  - On-screen prompts on the CBC guide you through the calibration process
  - A successful calibration means your sensor is (1) in the port you specified, is (2) shielded well enough, and (3) is aimed at the light sufficient for the sensor to determine whether or not the light is on
    - If calibration is successful your program will be paused until the game controller activates the starting light
    - If unsuccessful, you will be given another shot at calibration (**fix your shields!**)
- Using **`wait_for_light`** for your tournament program is highly advised!

Utilities for Botball

`shut_down_in`

- When executed, the function

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

  will stop motors, freeze servos, and send a **`create_stop`** command (in case a Create is being used). Use of this program is optional, and you can devise your own version if you wish, but it is at your own risk.
  - If you do devise your own version, TEST multiple times before the tournament
- If you are using the Create and your CBC loses its serial connection to the Create (probably the result of 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!)

Note: Normally a **`float`** number (with a decimal point) is used for *game_secs*

Game Program Template

```
#include "my_lib.c"
```
– add an include for each library your program needs –

```
int main() {
```

– specify any local variables your programs uses here –

– then start the robot simulation (with your own values) if you want to use it –

```
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
```
– if using Create –

– and put in the following two lines –

```
    wait_for_light(11);
```
– port between 8 and 12 –

```
    shut_down_in(119.5);
```
– choose your own number of seconds –

– put your main program statements here –

```
}
```

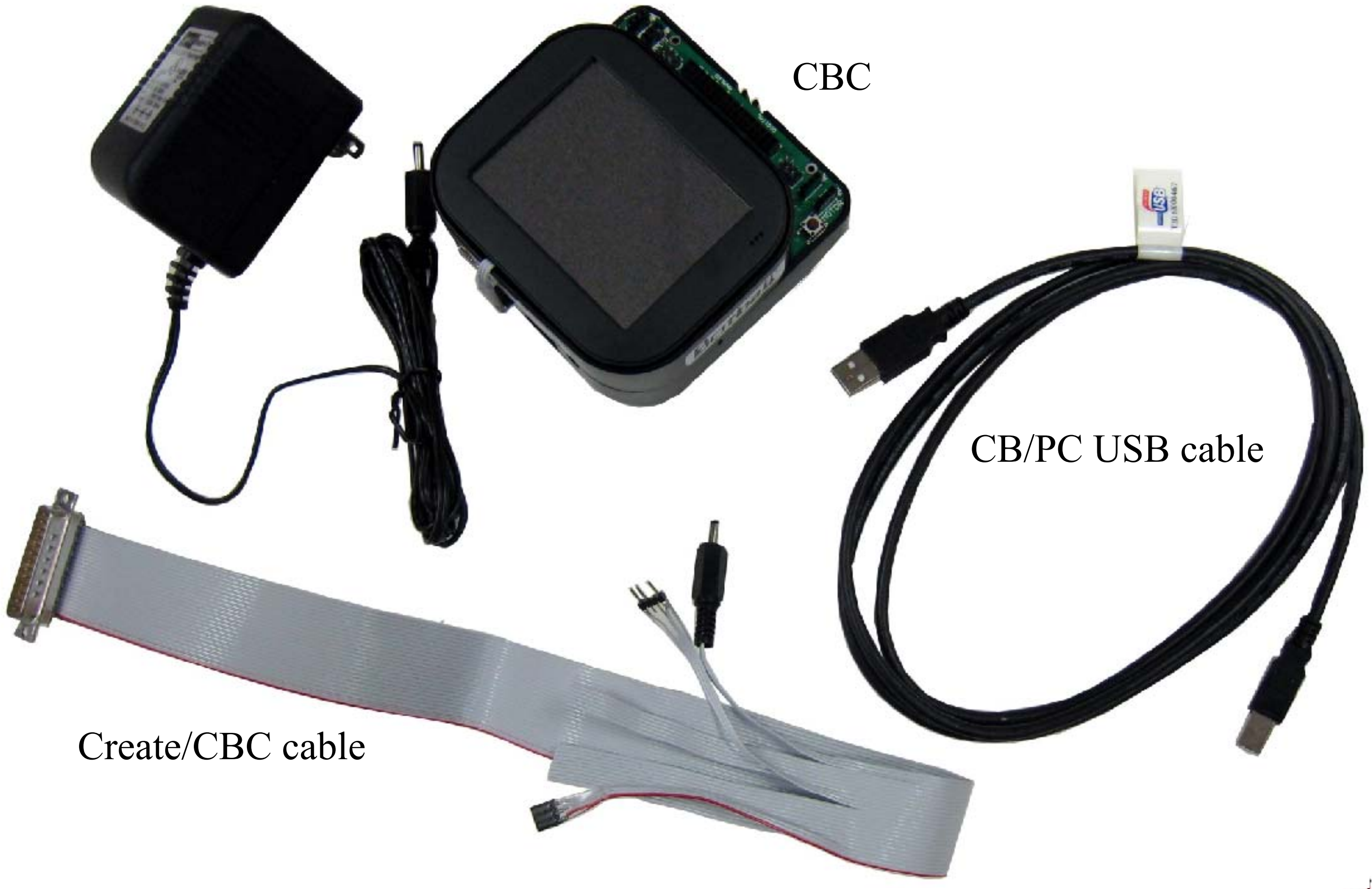
Botball®

Exercise #9: Game Program

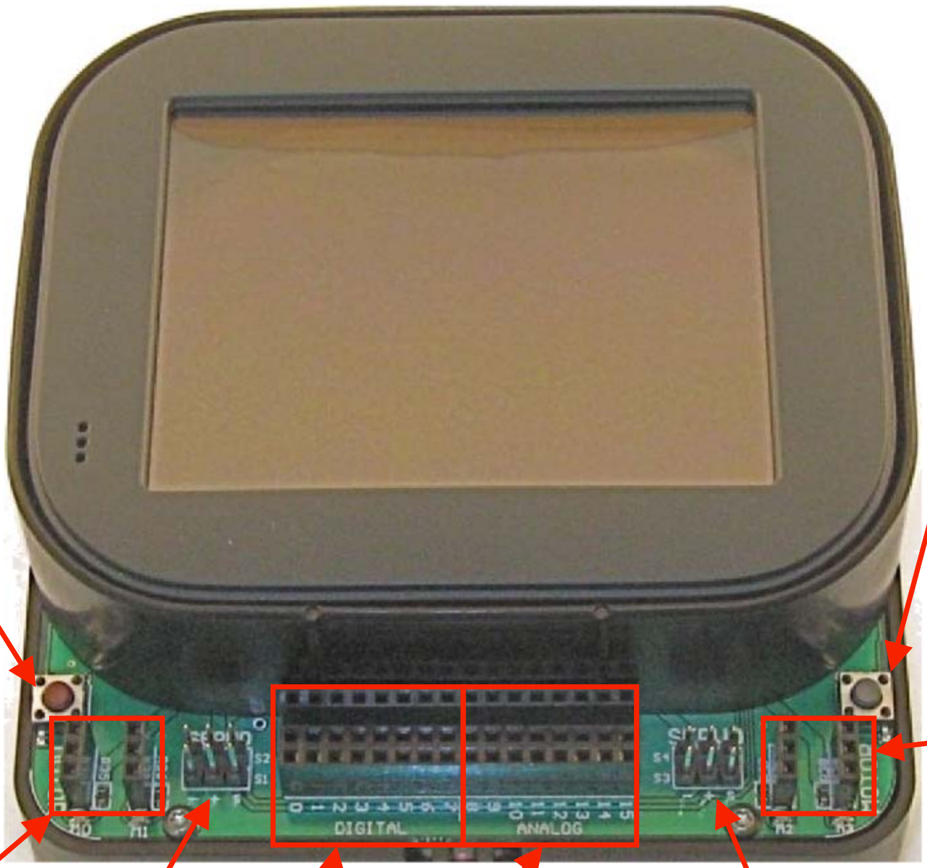
- Convert the program you wrote for drawing a rectangle to work as a game program
  - Use the template
- *Simulate* your program
  - When calibrating, *Left* is the left arrow key and *Right* is the right arrow key
  - For the simulator, use the analog slider for port 11 to first set the light on (low) value and then to set the light off (high) value
    - If these are sufficiently different, you have good calibration!
    - Simulate the start light turning on by sliding port 11 to low
- Experiment – see what happens if calibration isn't successful

The CBC

CBC and Cables



CBC (front ports)



Black button
`black_button()`

Boot/off button
(boots the CBC
or shuts it down)

Motor ports 2 & 3

Motor ports 0 & 1
`motor(<m>,<vel>)`
`mav(<m>,<vel>)`
`mrp(<m>,<vel>,<target>)`

Servo ports 1 & 2

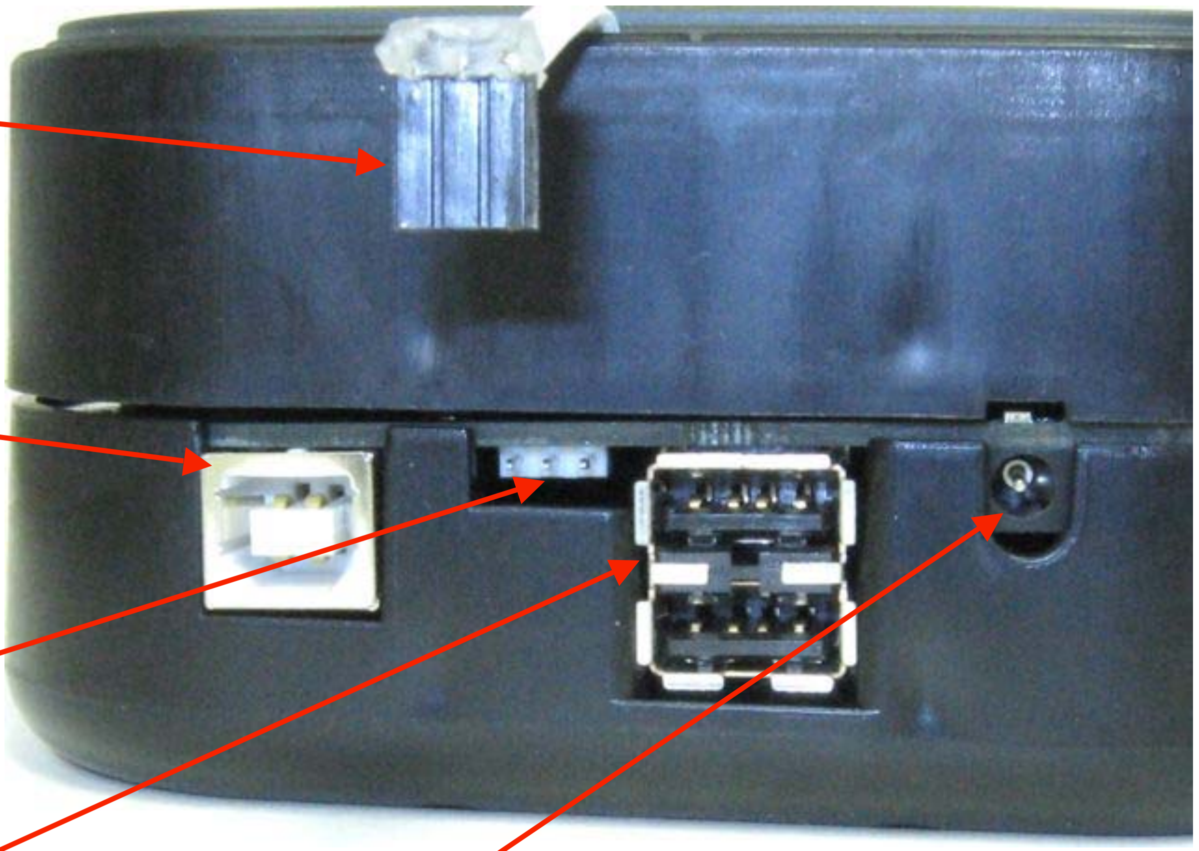
Analog ports 8-12
Floating analog 13-15
`analog10(<port>)`

Servo ports 3 & 4
`enable_servos()`
`disable_servos()`
`set_servo_position(<s>,<pos>)`

Digital ports 0-8
`digital(<port>)`

Botball®

CBC (rear ports)



Power lock dongle, insert in serial port for charging and transport

USB connector for cable to your PC (used for downloading your programs)

TTL level serial connector for communications with Create robot base

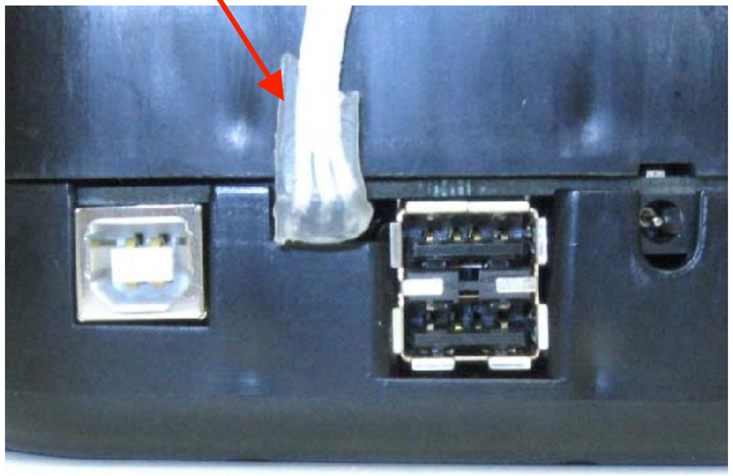
USB ports for camera and future peripherals

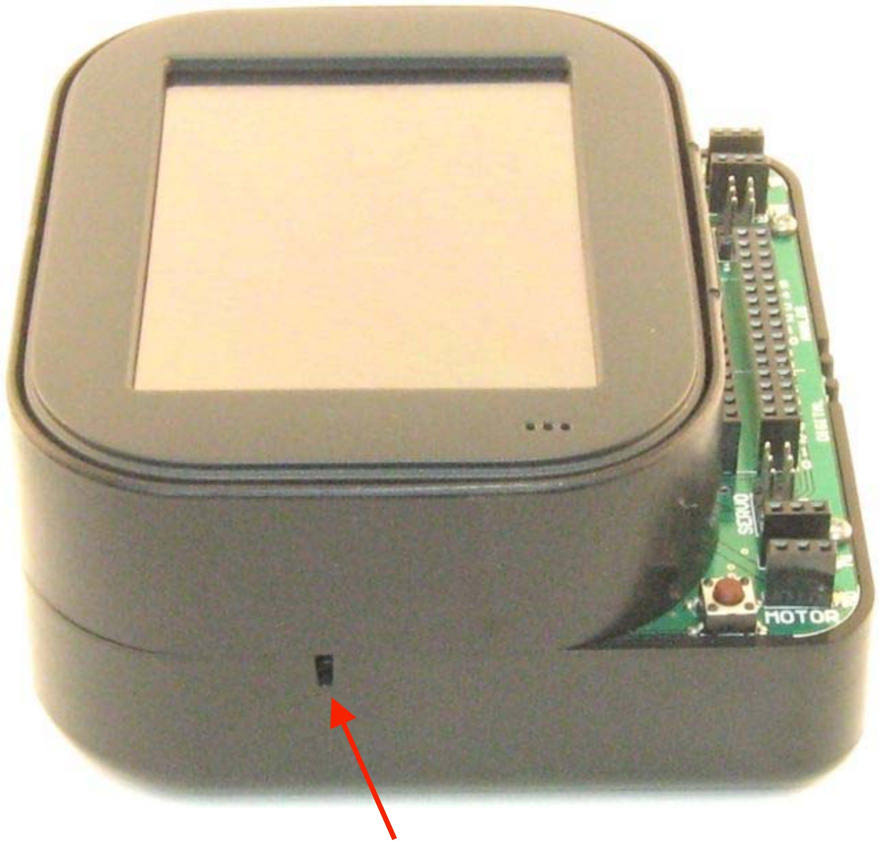
Charge port: use center positive, 13.5v DC at 1 amp charger

Botball

CBC Main Power

Insert power lock dongle in serial port to fully power down system for long term storage and transport





Use paper clip or Botball screwdriver to press recessed main power on button

The CBC Update Process

CBC Updates

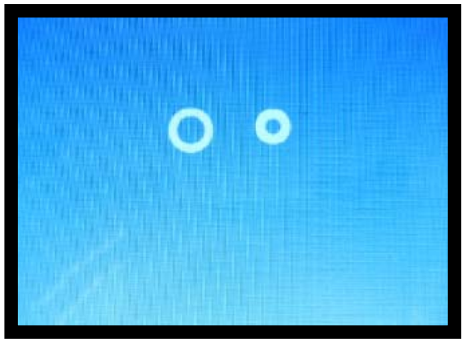
- The CBC is a brand new system, that first became operational in January 2009
- New capabilities are being added to the CBC on a regular basis
- In order to be able to use these new capabilities, you need to update the onboard software
- This process involves placing a file called ***userhook0*** on a flash drive, and inserting it into the CBC

Updating Firmware (1)

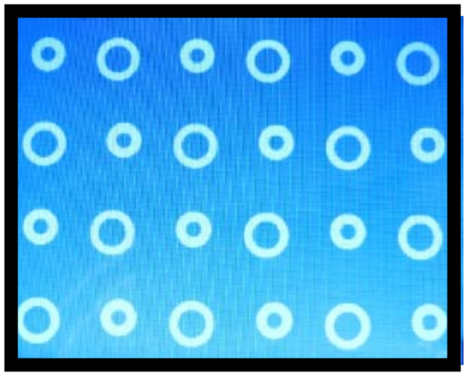
- The firmware on your CBC may need to be updated
  - This is analogous to upgrading the ROM BIOS on a PC or upgrading firmware on a MAC, which as you may know is a process that should be not be interrupted once started
  - You must insure there is adequate power for the process to complete, so you are strongly urged to plug in your CBC power supply **before** proceeding with an update
    - The process takes about 5 minutes, so this a very serious consideration!
  - **DO NOT INTERRUPT FIRMWARE UPDATE ONCE IT HAS BEEN STARTED**
- Continue to the next slide for the process
- Once the process has been started, do not (un)plug the power supply (do not plug in the power supply once the process has started -- better to hope that the battery will last).

Updating Firmware (2)

- **ONCE THE FIRMWARE UPDATE IS STARTED DON'T INTERRUPT IT!!**
- Plug the power adapter into your CBC
- Boot the CBC (push the recessed button for power on, if necessary, then press the (red) boot/off button twice to start boot)
- You should see this screen immediately while the CBC checks for its boot software; plug in the flash drive at this point.



- Then you should see this screen to indicate the CBC is booting (if you get colored circles, press reset and try again)
- Try to plug the flash drive with the firmware update into a CBC USB port before this screen first appears



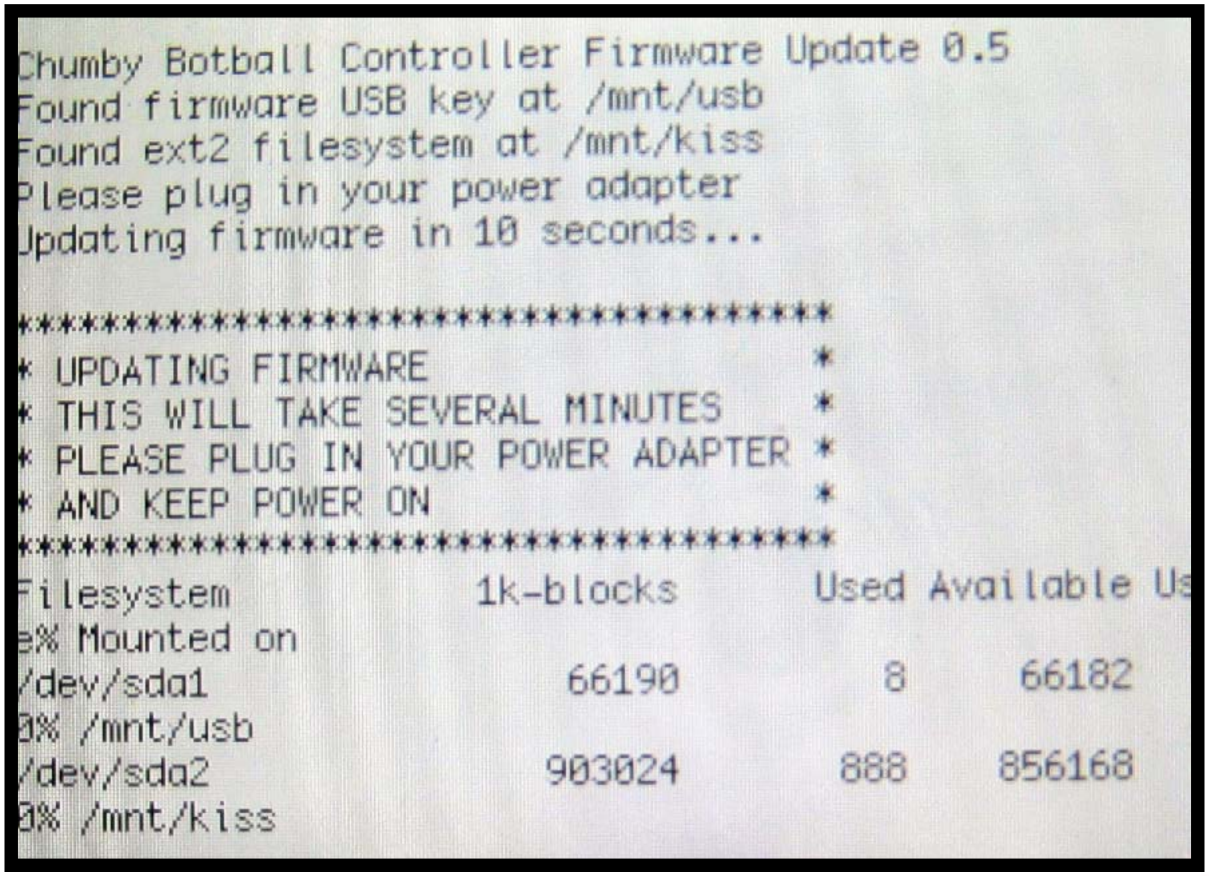
Updating Firmware (3)

- **Warning**: If the firmware is in the root directory of a flash memory plugged into the CBC, the boot software will probably find it when you boot the CBC and so will start the update process
  - Know what is on flash drive!!
- If the firmware upgrade fails, put a fresh copy onto the root of your flash drive and try it once more, then ask for help
  - **ONCE THE FIRMWARE UPDATE IS STARTED IT CAN'T BE INTERRUPTED!!**
  - You will see the screen on the following slide at first and then the system will start showing its progress

Updating Firmware (4)

ONCE THE FIRMWARE UPDATE IS STARTED IT CAN'T BE INTERRUPTED!!

When firmware update begins this screen briefly appears, and is then followed by update progress information such as given on the next slide

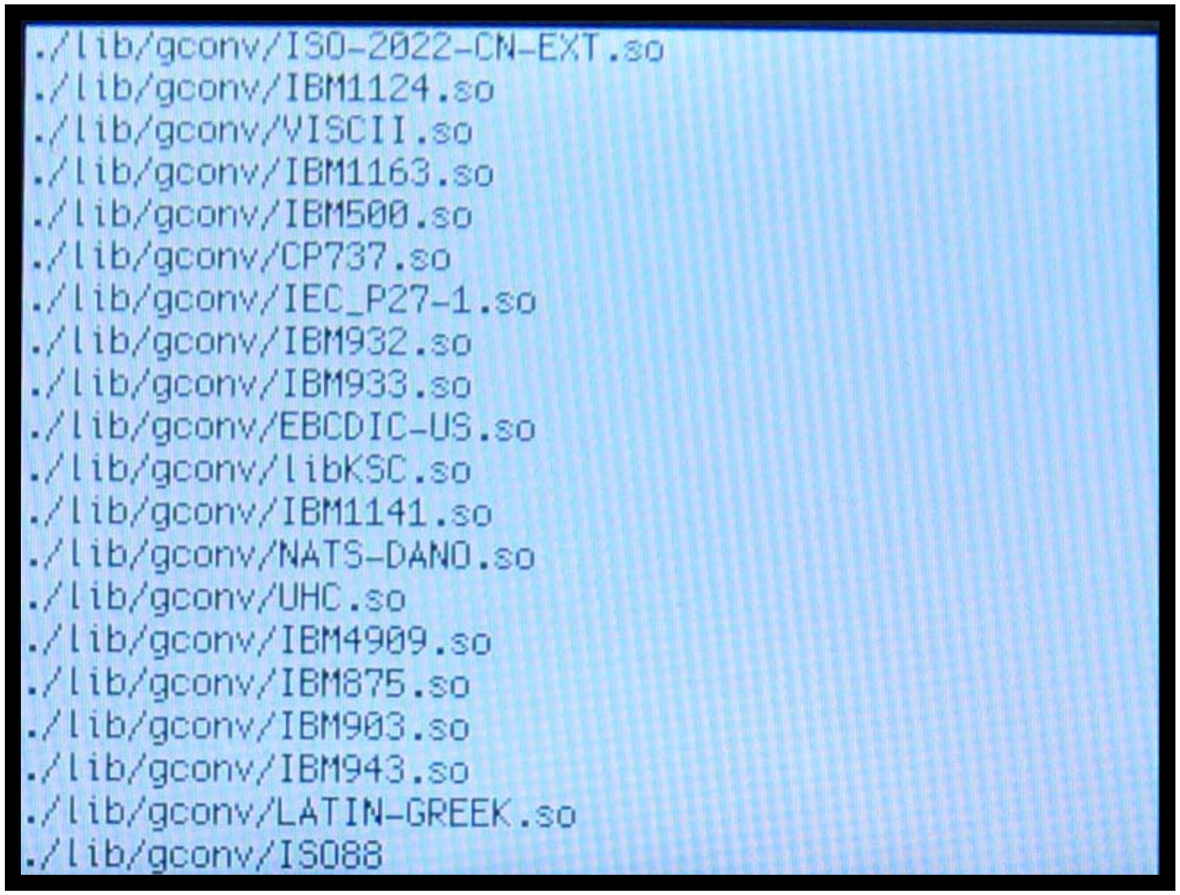


```
Chumby Botball Controller Firmware Update 0.5
Found firmware USB key at /mnt/usb
Found ext2 filesystem at /mnt/kiss
Please plug in your power adapter
Updating firmware in 10 seconds...

***********************************
* UPDATING FIRMWARE               *
* THIS WILL TAKE SEVERAL MINUTES  *
* PLEASE PLUG IN YOUR POWER ADAPTER *
* AND KEEP POWER ON               *
***********************************
Filesystem           1k-blocks      Used Available Us
e% Mounted on
/dev/sda1                66190         8     66182
0% /mnt/usb
/dev/sda2               903024       888    856168
0% /mnt/kiss
```

Updating Firmware (5)

ONCE THE FIRMWARE UPDATE IS STARTED IT DO NOT INTERRUPT IT!!

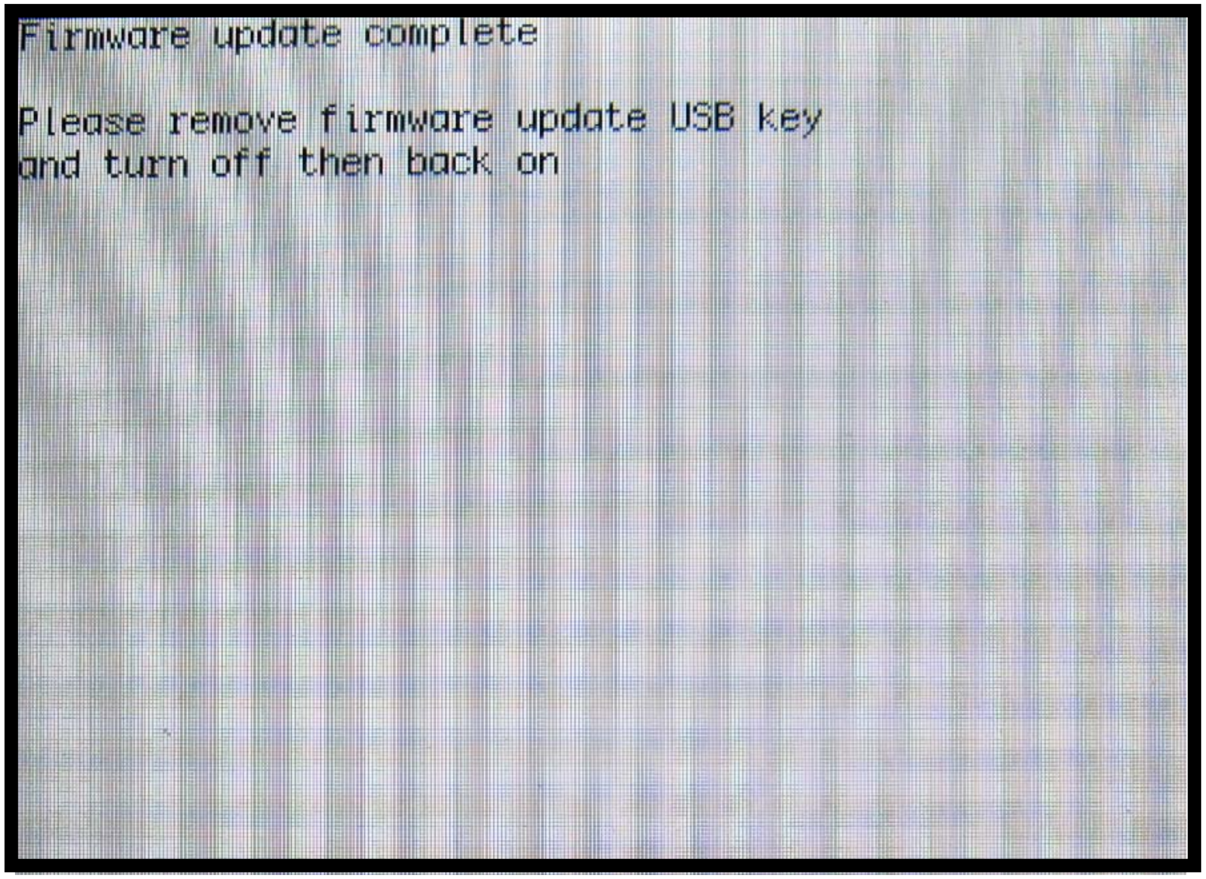
While firmware update is in progress, you will see information like this on the screen



```
./lib/gconv/ISO-2022-CN-EXT.so
./lib/gconv/IBM1124.so
./lib/gconv/VISCII.so
./lib/gconv/IBM1163.so
./lib/gconv/IBM500.so
./lib/gconv/CP737.so
./lib/gconv/IEC_P27-1.so
./lib/gconv/IBM932.so
./lib/gconv/IBM933.so
./lib/gconv/EBCDIC-US.so
./lib/gconv/libKSC.so
./lib/gconv/IBM1141.so
./lib/gconv/NATS-DANO.so
./lib/gconv/UHC.so
./lib/gconv/IBM4909.so
./lib/gconv/IBM875.so
./lib/gconv/IBM903.so
./lib/gconv/IBM943.so
./lib/gconv/LATIN-GREEK.so
./lib/gconv/IS088
```

Updating Firmware (5)

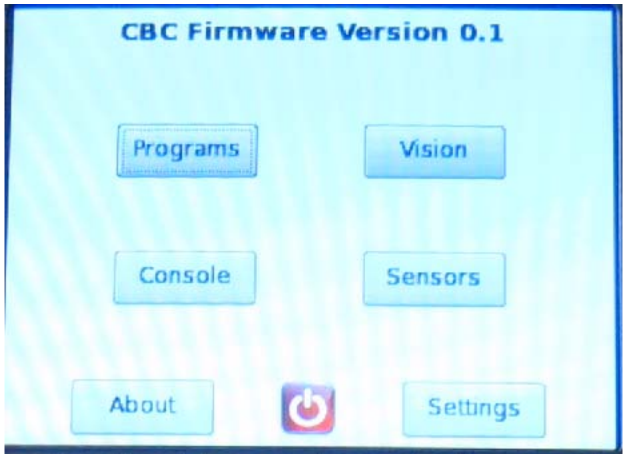
- ONLY when you reach the following screen is firmware update over!!



```
Firmware update complete

Please remove firmware update USB key
and turn off then back on
```

- Now **REMOVE THE FLASH DRIVE**, press the boot/off button once to turn off the CBC

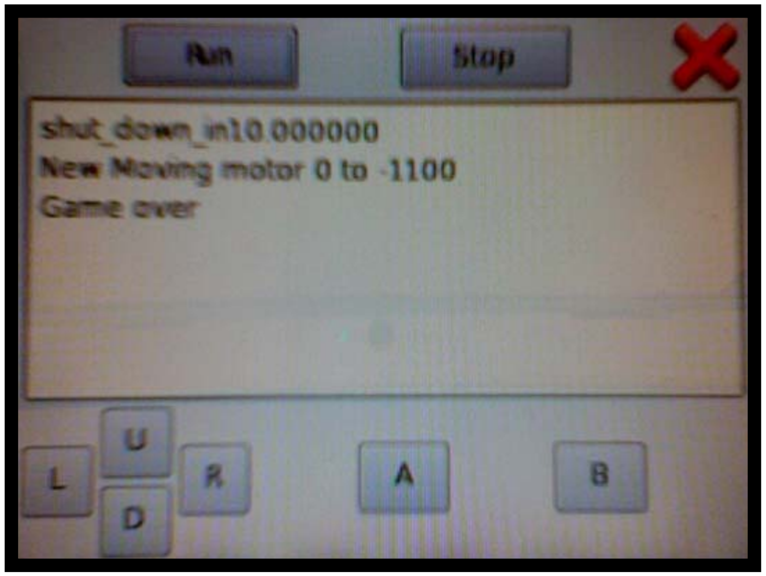
CBC User Interface



- Programs, Vision, and Sensors
  - Programs
    - At present, to run a program, copy the file with **`main()`** in it to your USB flash drive and rename the file **`robot.c`**
    - Copy any of your libraries the program uses to your flash drive
    - Make sure the CBC is already booted, and then insert the flash drive into the CBC
    - Press "Programs" to load your program and bring up the screen to "Compile" and "Run" it
  - The Console screen come up automatically when running a program, and can also be used to rerun programs already compiled
  - The camera configuration screen will come up when you press "Vision"
    - You will need to install your Botball camera in one of the CBC's USB ports
  - The sensor test screen will come up when you press "Sensors"
    - We will be using this later
- The remaining buttons are not working at this time

The Console Screen

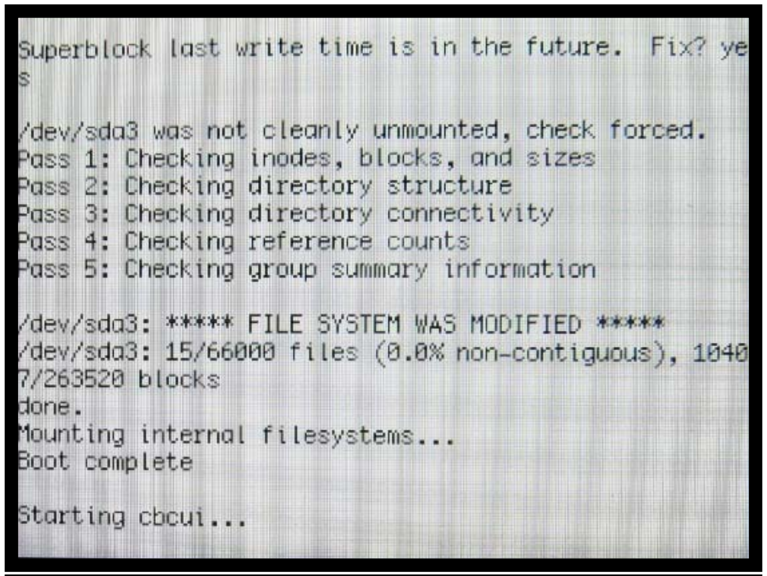
- This screen is used when running user programs
- The a, b, up, down, left and right buttons are also on this screen
  - User code can get the status of these buttons by calling `a_button()`, `up_button()`, etc
- You can switch among screens while your program is running, to monitor sensor values, motor positions, or the vision system



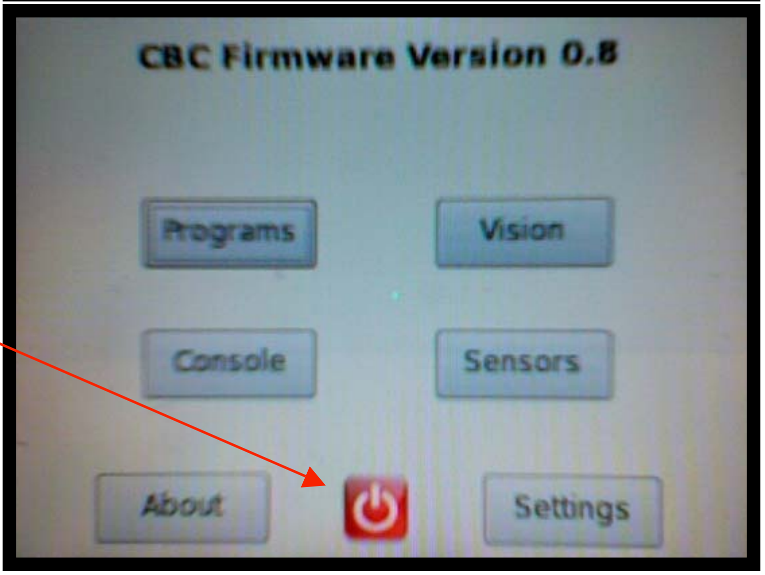
```
shut_down_in10.000000
New Moving motor 0 to -1100
Game over
```

Booting the CBC

- The CBC is fully off if the power lock dongle is plugged into its serial port
- To power on the CBC pull the dongle loose and use a small tool to press the recessed power button on the side of the CBC, then press the (red) boot/off twice (turning the screen off then back on).
  - **During boot be sure you don't mess with any motors you have plugged in** (could mess up port calibration)
- If boot is successful you will see a message indicating "Boot complete"
  and that the CBC User Interface is starting
- To power down, use the **touch screen** power button. In an emergency (run away robot, or system crash) the red physical button (used to boot the system) may be used to power down -- however, the next boot may take twice as long then if the soft power button had been used.



```
Superblock last write time is in the future.  Fix? ye
s

/dev/sda3 was not cleanly unmounted, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information

/dev/sda3: ***** FILE SYSTEM WAS MODIFIED *****
/dev/sda3: 15/66000 files (0.0% non-contiguous), 1040
7/263520 blocks
done.
Mounting internal filesystems...
Boot complete

Starting cbcui...
```



CBC Power Management (1)

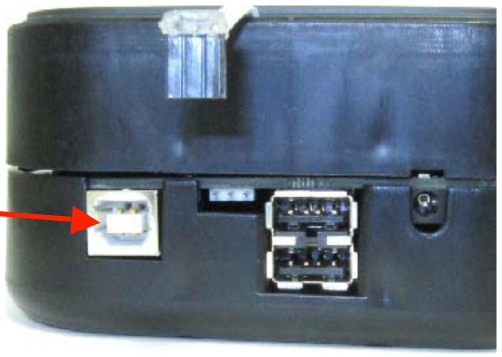
- When not running the robot, leave the CBC plugged in
  - The batteries have several mechanisms to prevent overcharging, so it is safe to leave the CBC on charge overnight
  - The CBC does not need to be shut down, if it is on the Charger (the processor and display can run 24/7 without damage)
  - Keep the CBC on the charger when you can; if the charge light goes out before batteries are fully charged, unplug the charger, give it a short rest, and plug it back in
    - If the batteries are fully charged, the light will go back out again quickly
- The CBC is best stored with the power lock dongle plugged in
  - Plugging in the dongle completely turns off power to the CBC
  - At the end of the Botball season or when shipping the CBC, reinsert the dongle

CBC Power Management (2)

- If you plug the power lock dongle while the charge light is blinking, it will continue to blink after the charger stops charging, and even if the charger is unplugged!
  - This will not result in anything bad and the light will turn off the next time you turn on the CBC
- The CBC can be plugged in to the power adapter or to the Create to for charging, BUT servos and motors will behave differently when the CBC is plugged into a charger (to the wall or Create) then it will when running off batteries.
- **Best not to turn on motors or servos when the CBC is plugged into a charger**

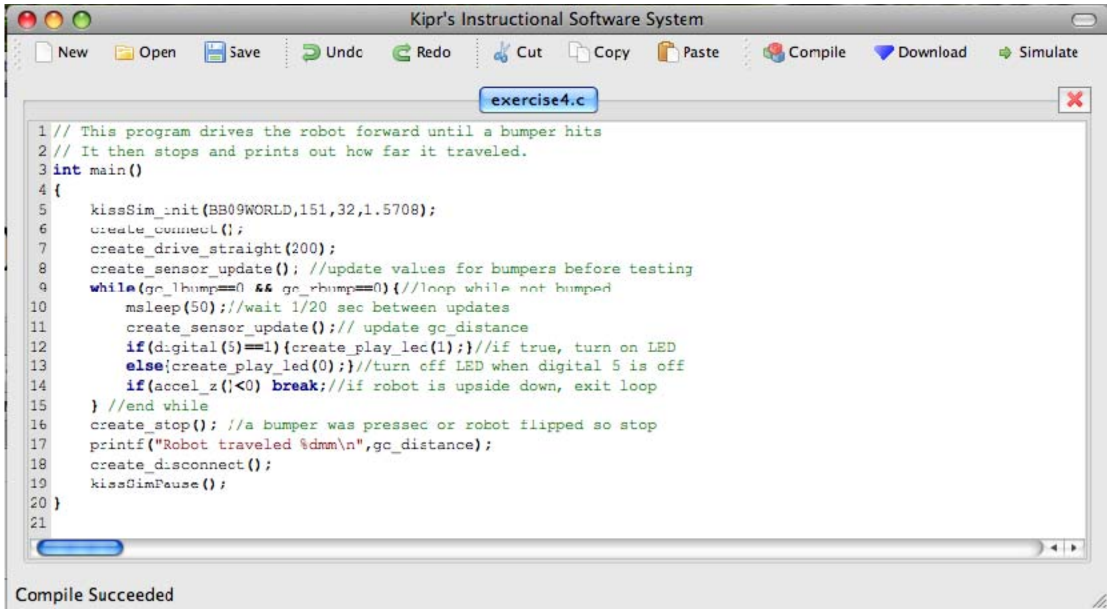
Running Programs on the CBC

- The CBC can be connected to your PC using the included USB cable
  - The flat end (A) of the USB cable goes into your PC
  - The squarish end (B) plugs into the CBC


USB (B) connector for cable to your PC (used for downloading your programs)

Running Programs on the CBC (2)

- Compile and simulate your program in KISS-C to make sure it is bug free

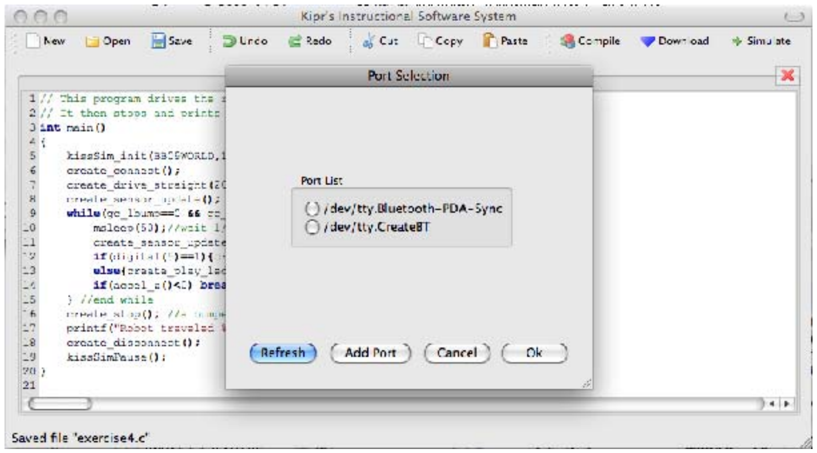


```
// This program drives the robot forward until a bumper hits
// It then stops and prints out how far it traveled.
int main()
{
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
    create_drive_straight(200);
    create_sensor_update(); //update values for bumpers before testing
    while(gc_lbump==0 && gc_rbump==0){//loop while not bumped
        msleep(50);//wait 1/20 sec between updates
        create_sensor_update();// update gc_distance
        if(digital(5)==1){create_play_led(1);}//if true, turn on LED
        else{create_play_led(0);}//turn off LED when digital 5 is off
        if(accel_z()<0) break;//if robot is upside down, exit loop
    } //end while
    create_stop(); //a bumper was pressed or robot flipped so stop
    printf("Robot traveled %dmm\n",gc_distance);
    create_disconnect();
    kissSimPause();
}
```

Compile Succeeded

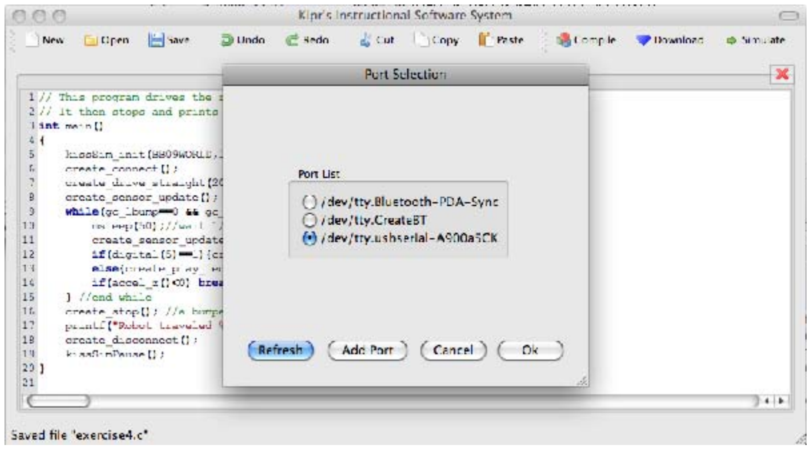
Running Programs on the CBC (3)

- Connect the CBC to your PC using the USB cable
- Set the USB port using the Settings menu
  - If you are not sure which port is the correct one:
    - disconnect the CBC, press refresh,
    - connect the CBC and press refresh
  - The port for the CBC is the one that appears



Port List:
- ( ) /dev/tty.Bluetooth-PDA-Sync
- ( ) /dev/tty.CreateBT

Saved file 'exercise4.c'

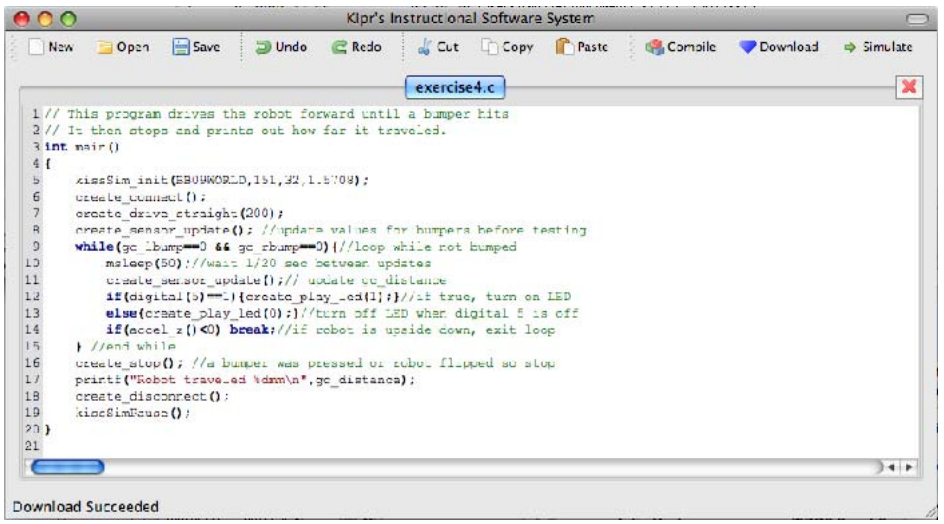


Port List:
- ( ) /dev/tty.Bluetooth-PDA-Sync
- ( ) /dev/tty.CreateBT
- (●) /dev/tty.usbserial-A900a5CK

Saved file 'exercise4.c'

Running Programs on the CBC (4)

- Press the "Download" button on KISS-C
  - The status message on the lower left corner of KISS-C will switch to "Downloading…"
  - When the download is complete, the status message will switch to "Download Succeeded"
  - The CBC will automatically switch to the programs menu and then compile your program on the CBC

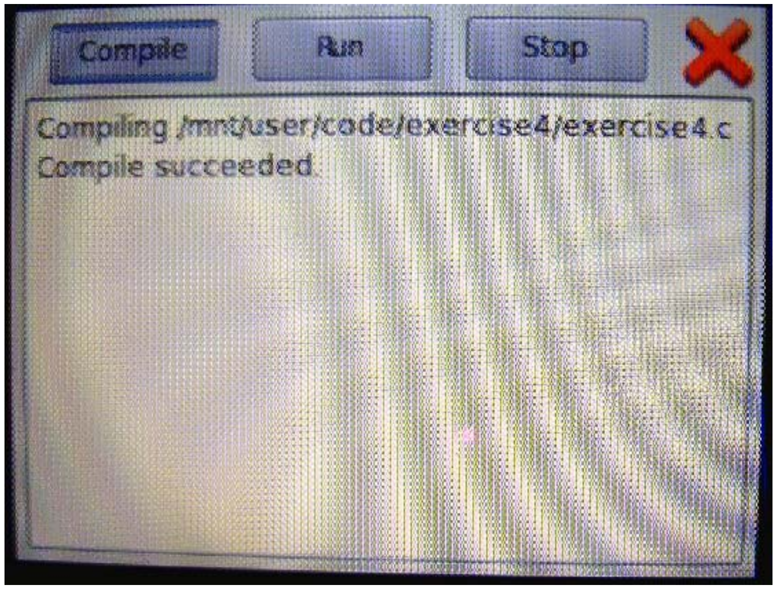


```
// This program drives the robot forward until a bumper hits
// It then stops and prints out how far it traveled.
int main()
{
    kissSim_init(BB09WORLD,151,32,1.5708);
    create_connect();
    create_drive_straight(200);
    create_sensor_update(); //update values for bumpers before testing
    while(gc_lbump==0 && gc_rbump==0){//loop while not bumped
        msleep(50);//wait 1/20 sec between updates
        create_sensor_update();// update gc_distance
        if(digital(5)==1){create_play_led(1);}//if true, turn on LED
        else{create_play_led(0);}//turn off LED when digital 5 is off
        if(accel_z()<0) break;//if robot is upside down, exit loop
    } //end while
    create_stop(); //a bumper was pressed or robot flipped so stop
    printf("Robot traveled %dmm\n",gc_distance);
    create_disconnect();
    kissSimPause();
}
```

Download Succeeded

Running Programs on the CBC (5)

- The screen on the CBC will indicate a successful compilation
  - At this point you can press the "Run" button on the CBC to execute your program
- Alternatively, you can copy your program onto a USB flash drive and rename it "robot.c". This program can be loaded by placing the flash drive into the CBC and pressing the compile button in the Programs window of the CBC
- Note: If your program `#include`s any other files, they must be in the same directory as the file you downloaded from KISS-C



```
Compiling /mnt/user/code/exercise4/exercise4.c
Compile succeeded.
```

Exercise #9b: Game Program

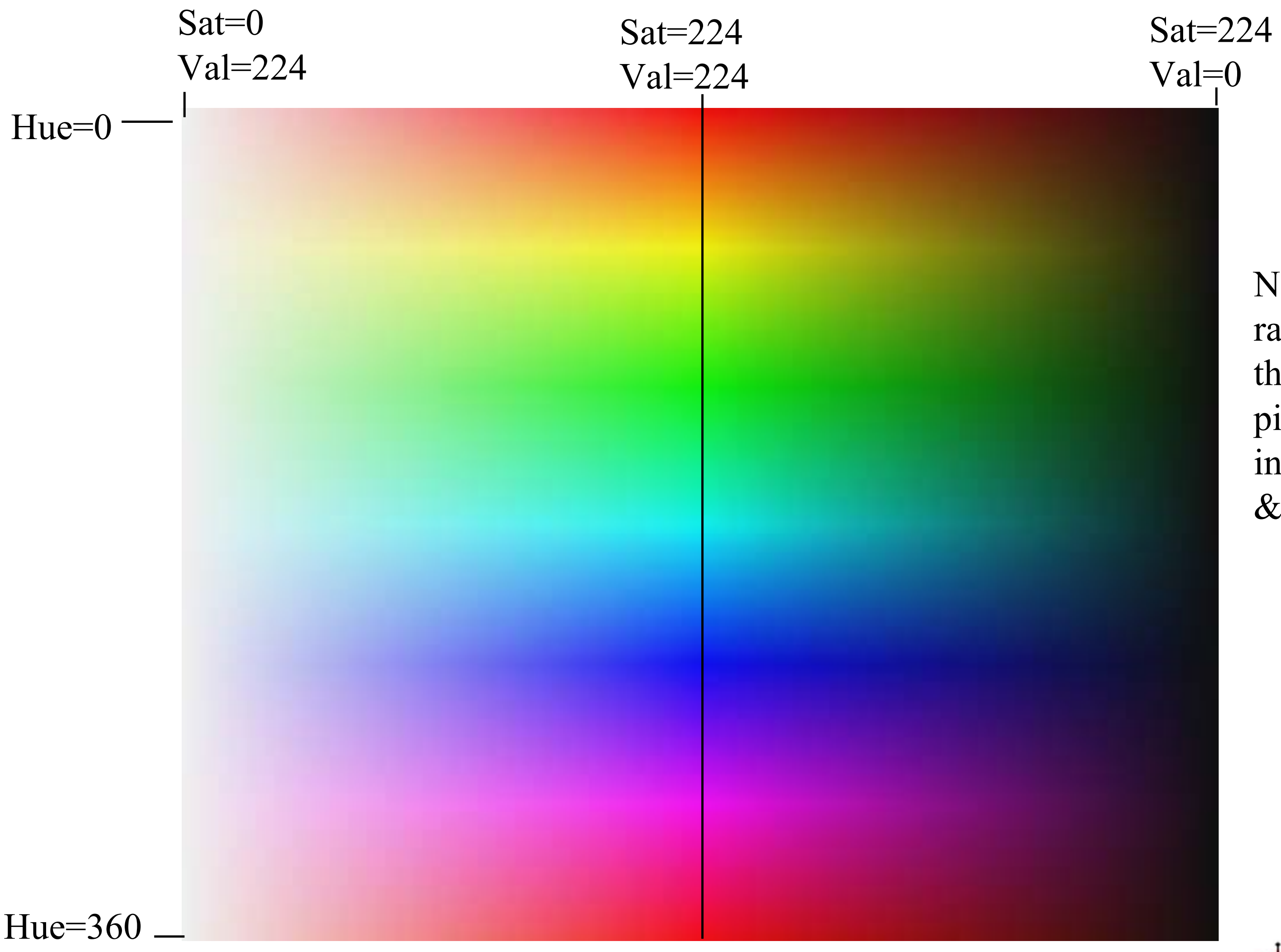
- Take your haiku program from Day 1 (or any other program that prints things out, but does not use the Create) and download it to the CBC
- You can add in the Botball utilities to your program and use a light sensor to test those out as well.

Color Vision System

Color Vision System

- Color Space
- Finding color blobs in an image
- Trying out the vision system
- Making a color model
  - On screen instructions
  - Camera check with test program
- Tuning the camera

HSV Color Selection Plane



Note: 224 is the range of values the camera pixels put out in each of R, G & B

Finding Color Blobs in an Image

Color Blobs

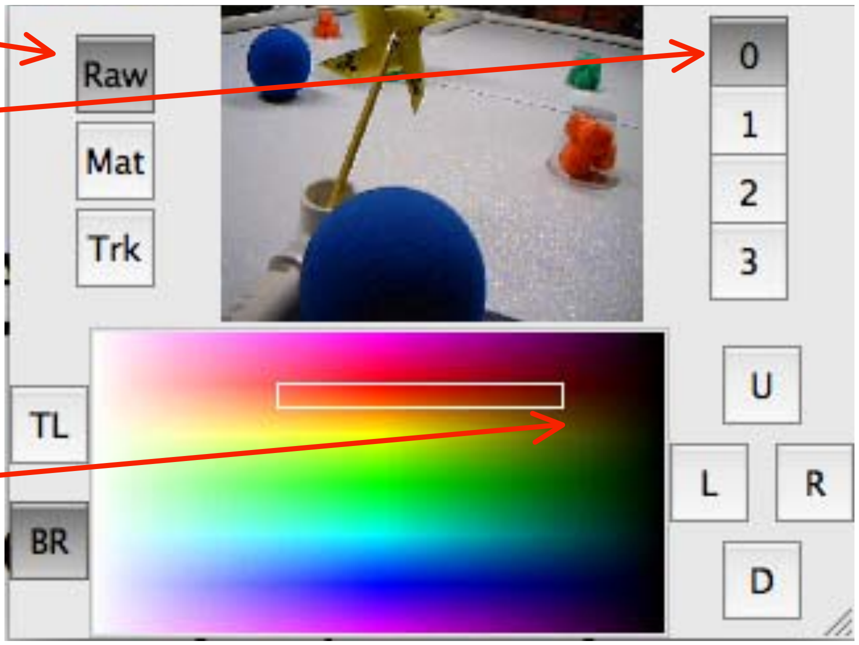
- For color tracking, a rectangular piece of the color selection plane is selected. All of the pixels in the image whose color falls within that piece are selected.
  - The camera resolution using the CBC is 160x120 = 19,200 pixels
- Selected pixels that are contiguous are combined as blobs
- Each blob has a size, position, number of pixels, major and minor axis, etc.
- The blobs correspond to objects seen in the image that are the desired color (as given by the specified piece of the color selection plane).

Color Models

- The CBC can segment the image using four different pieces of the HSV color selection plane (each is called a color model) simultaneously
- It can track a number of blobs from each color model
- It can display the video in any one of three ways
  - **Raw** (live video)
  - **Mat**ch (pixels matching the color model are highlighted)
  - **Tr**ac**k**ed (highlight matching pixels and show blob boundaries and centroids)

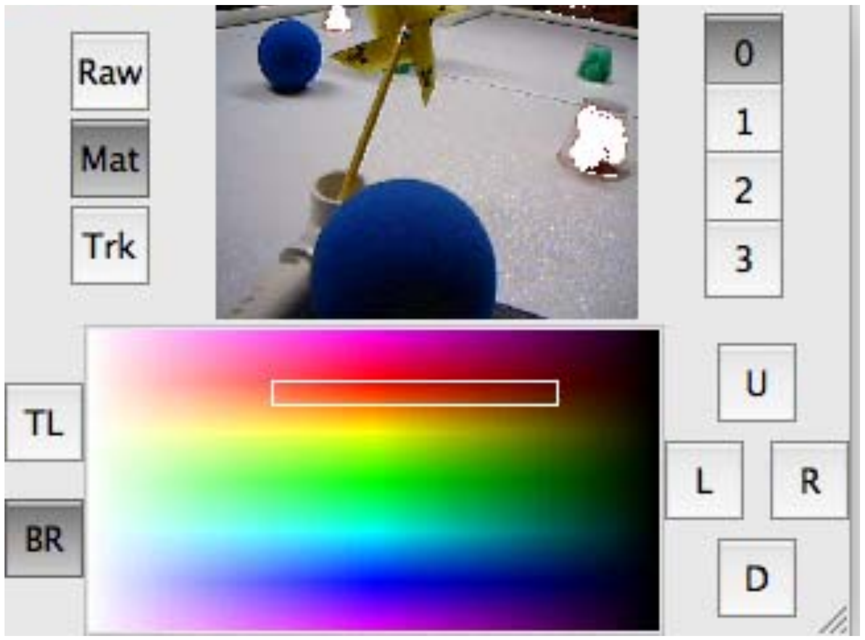
Color Vision Interface

- **Raw** image is displayed
- Color Model **0** is being manipulated
- The **B**ottom **R**ight corner of the color selection box is being adjusted
- It can be moved **L**eft, **R**ight **U**p or **D**own



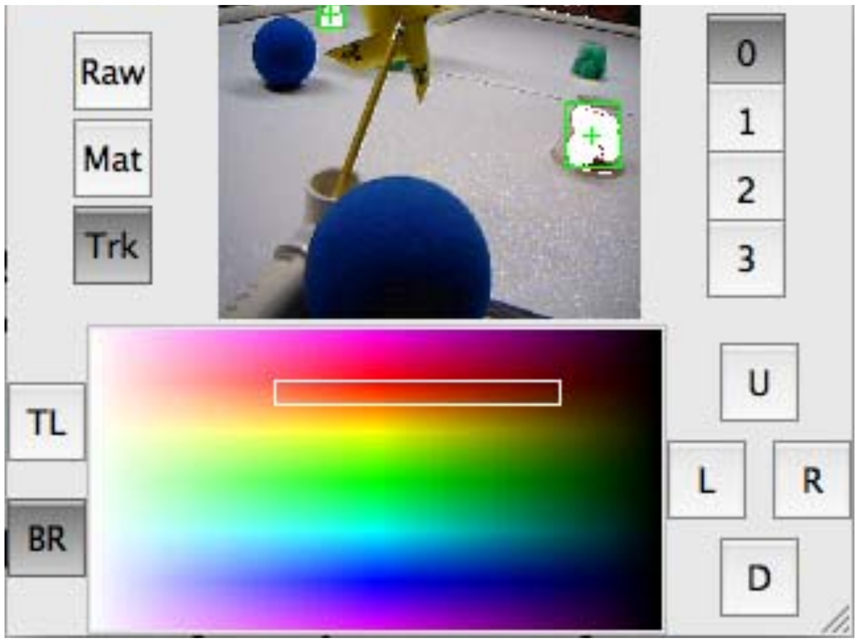
Color Vision Interface

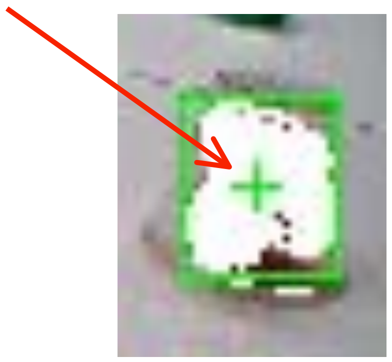
- **Mat**ched image is being displayed
- Pixels that correspond to selected color region are shown highlighted



Color Vision Interface

- **Tracked** image is displayed
- The bounding boxes of the tracked blobs are displayed (look closely!), along with the centroid of each blob





More on Color Models

- A Color Model-HSV specifies a bounding box in the color selection plane
- Moving either edge towards the center line constrains the range of accepted color values to only include more vivid colors (i.e. only accept things that are more like *Astro Brights* paper).
- If everything you want is being accepted but so is a lot of other junk you don't want, move the corners closer to the center.
  - Moving either edge away from the center of the selection plane includes less vivid colors in the color model.
    - Moving the left edge away from center includes colors that are closer to pastel than what is currently accepted.
    - Move the right edge away from center includes darker colors than what is currently accepted.
  - Moving the top and bottom edges up and down changes the range of hues accepted by the model.

Built-in Vision Functions

Vision Functions

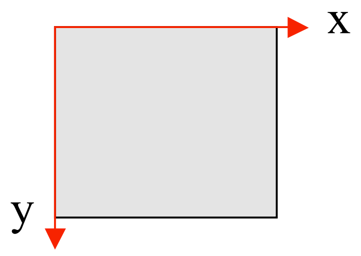
- The normal use of the camera is for blob tracking
- Blob tracking functions use the active color models you have established
    - Color models are initialized from flash when you start the CBC
- There are vision functions for using a program to access or alter the camera configuration (see KISS-C help)
    – For most purposes, you can accomplish all the camera configuration that is necessary using the vision interface accessed from the CBC menu display (i.e., using these functions is for advanced users!)

CBC Camera

- The camera should be plugged into one of the CBC's USB ports
  - Inserting or removing camera when CBC is on may cause CBC to crash
  - Best to plug in or remove camera when CBC is turned off
  - Camera uses a lot of power, so keep CBC plugged in when creating color models, where possible

Color Tracking

- There are four channels numbered 0, 1, 2, 3 for processing color models
  - In the original configuration the models are set up to roughly track red, yellow, green, and blue
  - The camera field of view is treated as an x-y (column,row) coordinate array
    - The upper left corner has coordinates (0,0)
    - The lower right corner has coordinates (159,119)
    - The CBC display does not show the camera's full field of view



Color Tracking Functions

- Current camera tracking data is captured only when the **`track_update()`** function is called
  - Normally called prior to using other tracking functions
  - If you don't call it, no other data will change
- The function **`track_count(`***<ch #>***`)`** is used to find out how many blobs a channel currently has in view
  - If the value is 0, the camera isn't detecting any blobs for the channel
  - The blobs are numbered starting from 0, with blob 0 being the largest blob
- **`track_size (`***<ch #>***`,`***<blob#>***`)`**
  - gets the number of pixels in the blob
  - maxes out (saturates) at 32,767 if the area gets that large
- **`track_confidence (`***<ch #>***`,`***<blob#>***`)`**
  - gets the confidence for seeing the blob as a percentage of the blob pixel area/bounding box area (range 0-100, low numbers bad, high numbers good)

Getting Blob Data

- **`track_x (`***<ch #>***`,`***<blob#>***`)`**
  `track_y (`*<ch #>***`,`***<blob#>***`)`**
  - Gets the x coordinate (column) or y coordinate (row) of the centroid of the blob
  - The total visual field has (x,y)=(0,0) as the upper left and (x,y)=(159,119) as the lower right
    - These give the limiting values for track_x and track_y
- **`track_bbox_left(`***<ch #>***`,`***<blob#>***`)`**
  `track_bbox_right (`*<ch #>***`,`***<blob#>***`)`**
  - Gets the x coordinate of the leftmost pixel or rightmost pixel in the blob
- **`track_bbox_top(`***<ch #>***`,`***<blob#>***`)`**
  `track_bbox_bottom (`*<ch #>***`,`***<blob#>***`)`**
  - Gets the y coordinate of the topmost pixel or bottommost pixel in the blob
- **`track_bbox_width(`***<ch #>***`,`***<blob#>***`)`**
  `track_bbox_height (`*<ch #>***`,`***<blob#>***`)`**
  - Gets the width of the bounding box
    - same as track_bbox_right - track_bbox_left
  - Gets the height of the bounding box
    - same as track_bbox_bottom - track_bbox_top

color-line.c Program for Create

```
// Makes robot follow object being tracked
// on color channel 0
int main()
{
   int forwardSpeed=150, x;
   create_connect(); // starts comm btw CBC and Create. LED should turn orange
   wait_for_light(12); // Use light sensor in port 12
   shut_down_in(60.25); // end program is 60.25 seconds
   create_sensor_update(); // update create sensors (bumpers)
   while(!gc_lbump && !gc_rbump){// loop until a bumper is pressed
      track_update();//update the camera image
      if(track_count(0)>0){// are there blobs on color 0?
         x = track_x(0,0);//get x of largest blob on color channel 0
         // drive direct drives right and left wheel speeds
         create_drive_direct(forwardSpeed+2*(80-x), forwardSpeed+2*(x-80));
         printf("Object at %d, %d\n",x,track_y(0,0));
      }// end if
      else {// no blobs on color 0
         create_stop();
         printf("No red seen\n");
      }// end else
      msleep(200);// slow loop so we can read prints
      create_sensor_update();// update the bumper status
   }//end while
   printf("All done \n");
   create_disconnect();//turn off comm to create
   return 0;// gets rid of warning message on CBC compile
}
```

Sensors

Digital Sensors

- Digital sensors are ones which produce an “off” (0) or “on” (1) signal.
- Graphical buttons serve as built-in digital sensors for both the CBC and *kissSim*
- There are library functions specific to each built-in sensor
- The CBC has digital “ports” to use with two-state plug-in sensors
  - accessed using the **`digital`** library function

Analog Sensors

- Analog sensors are ones which produce a range of integer values, not just 0 and 1
- The CBC has analog ports and floating analog ports for plug-in sensors, which are accessed using either the **`analog`** library function or the **`analog10`** library function
  - With the **`analog`** function the return values are scaled to the range is 0-255, with **`analog10`**, to 0-1023
- There are two types of analog sensors used with the CBC: regular analog sensors (such as light sensors), and floating analog sensors (such as distance sensors)

Detachable Sensors

- Detachable sensors use a keyed connector (2 wire or 3 wire, except for USB connector on the camera)
  - Analog sensors:
    - Light (ports 8-12)
    - IR reflectance (ports 8-12)
  - Floating analog sensors:
    - Optical rangefinder (ports 13-15)
  - Digital sensors:
    - Touch (ports 0-7)
  - Special sensors:
    - Ultrasonic rangefinder (sonar)
      - (ports 13-15)
    - CBC Camera
      - (either USB port on CBC)

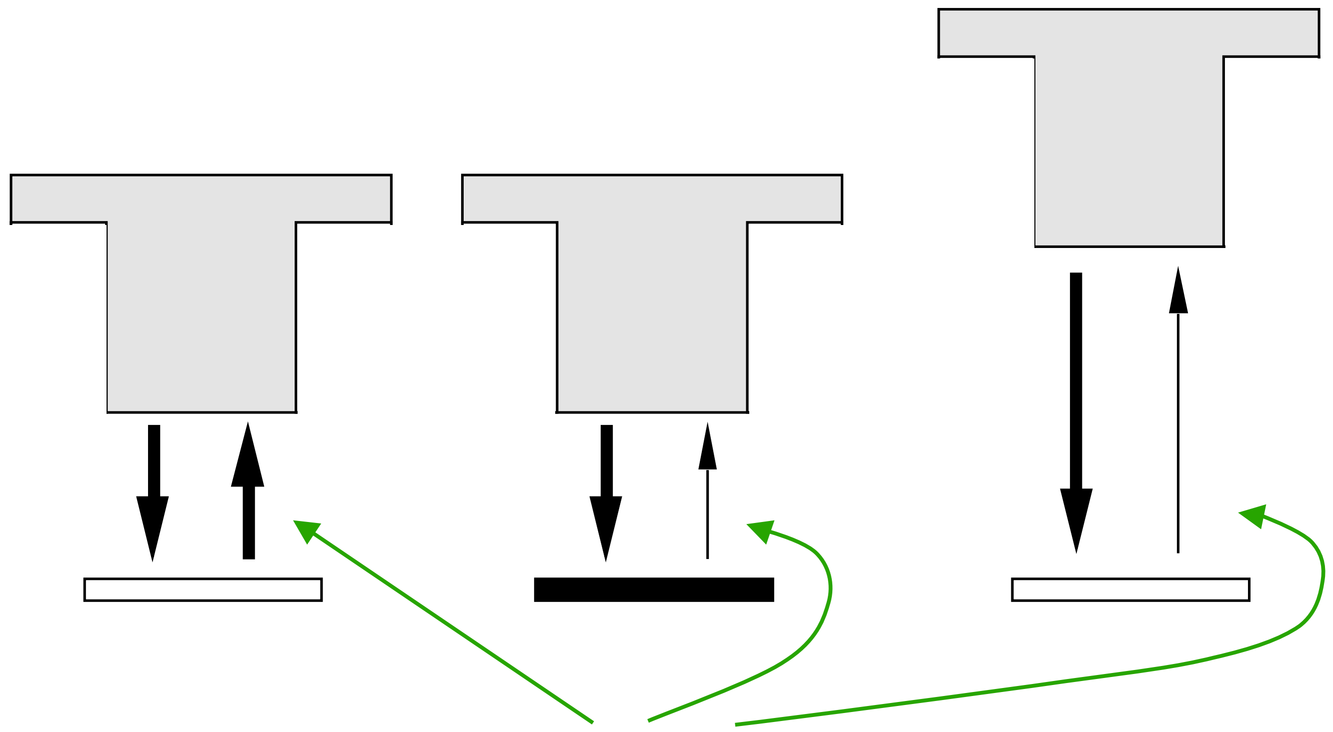
Light Sensors

- Analog sensor
- Connect to ports 8-12
- Access with library function **`analog10(`***port#***`)`**
  - You can also use **`analog(`***port#***`)`** for lower resolution
- Low values (near 0) indicate bright light
- High values (near 1023 for **`analog10`**, 255 for **`analog`**) indicate low light
- Sensor is somewhat directional and can be made more so using black paper or tape or an opaque straw or lego to shade extraneous light. Sensor can be attenuated by placing paper in front.

IR Reflectance Sensor "Top Hat"

- Connect to ports 8-12
- Access with library function **`analog10(`***port#***`)`**
  - You can also use **`analog(`***port#***`)`** for lower resolution (0-255)
- Low values (0) indicate bright light, light color, or close proximity
- High values (1023) indicate low light, dark color, or distance of several inches
- Sensor has a reflectance range of about 3 inches

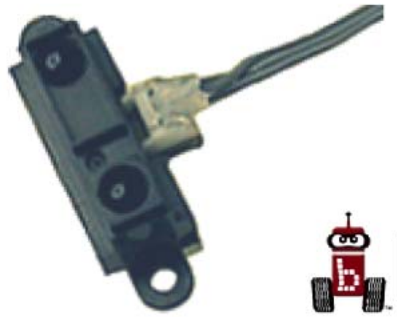
IR Reflectance Sensors



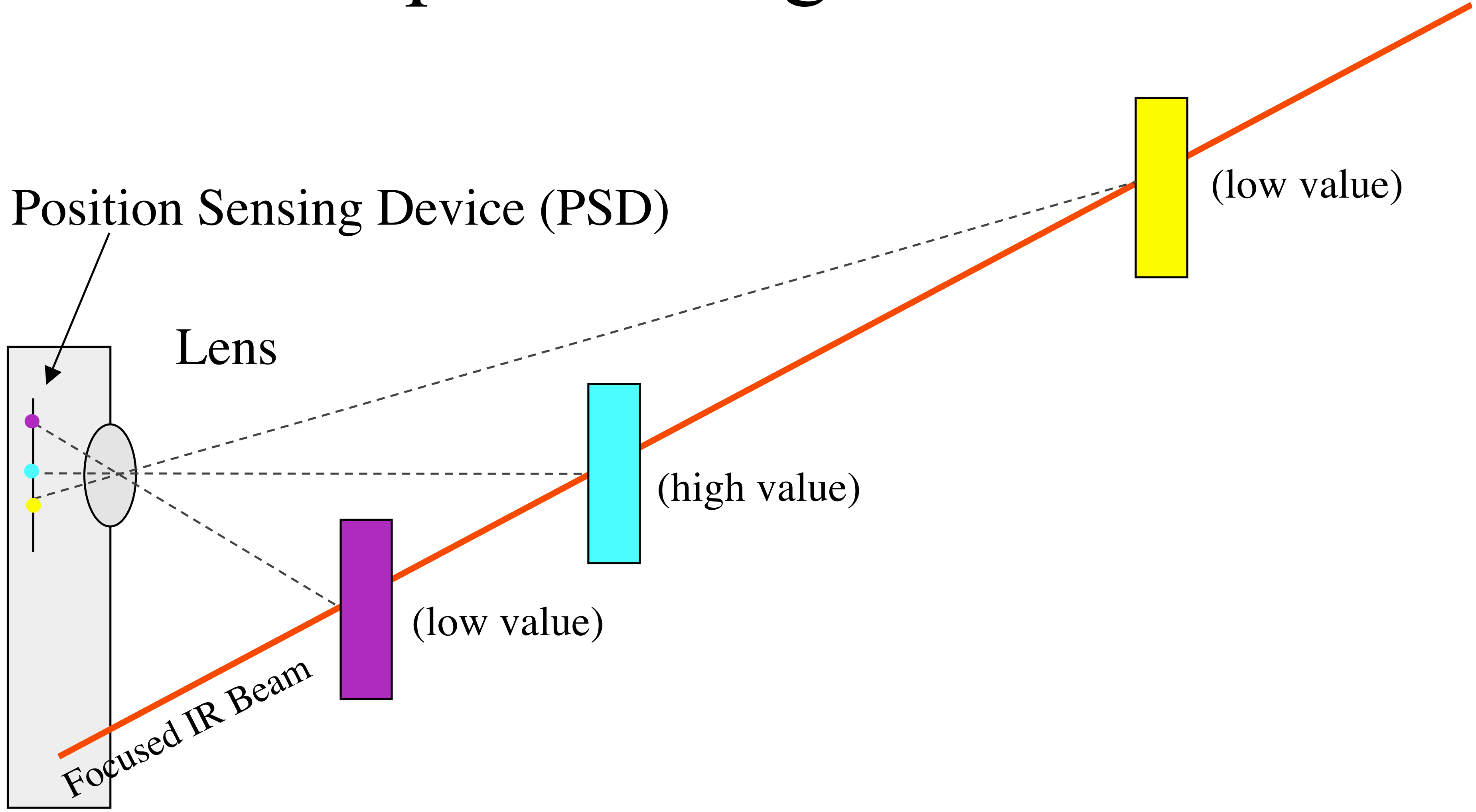
Amount of reflected IR depends on surface texture, color, and distance to surface

Optical Rangefinder “ET”

- Floating analog sensor
- Connect to ports 13-15
- Access with library function **`analog10(`***port#***`)`**
  - You can also use **`analog(`***port#***`)`** for lower resolution
- 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



Optical Rangefinder

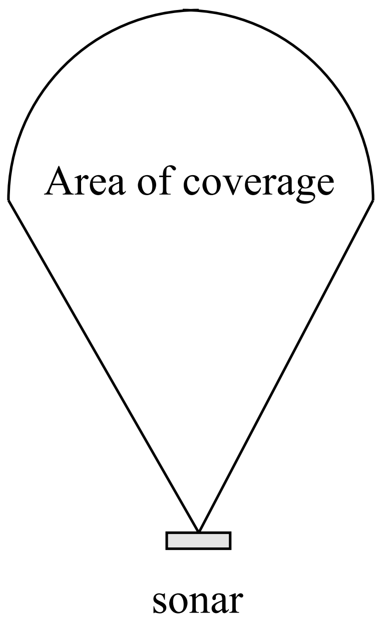


Ultrasonic Rangefinder (Sonar)

- Timed analog sensor
- Connect: port 13-15
- Access with library function **`sonar`** `(`*port#*`)`
- Returned value is distance in mm to closest object in field of view
- Range is approximately 150-10000mm
- **Important**: when first powered on (when CBC master power is turned on (if sonar is plugged in) or when sonar is plugged in, if master power is on), the sonar performs an auto-calibration; the sonar should have a clear view for at least 6 inches (15cm) at this time
- Objects closer than 150mm will return values of about 150mm.

Ultrasonic Sensors

- Puts out a short burst of high frequency sound
- Listens for the echo
- Speed of sound is ~300mm/ms
- **`sonar()`** times the echo, divides by two and multiplies by speed of sound
- The sonar field of view is an approximately 30^o cone

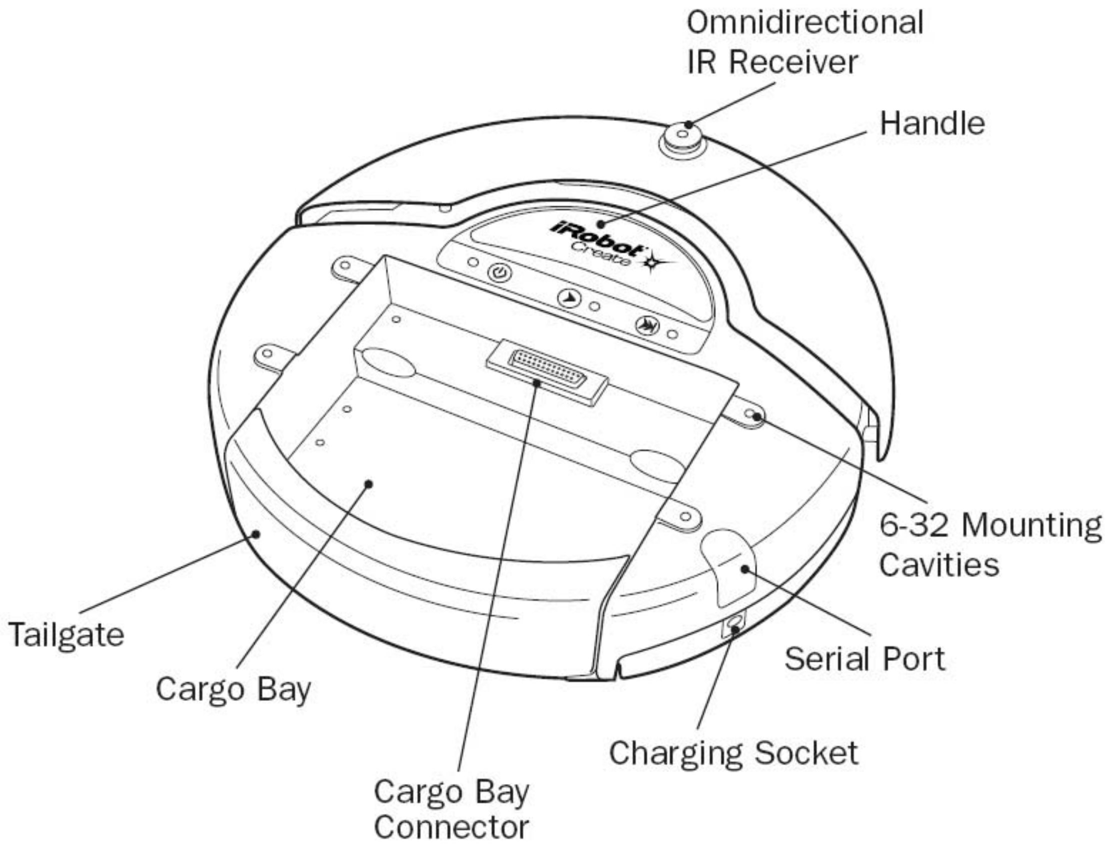


Touch Sensors

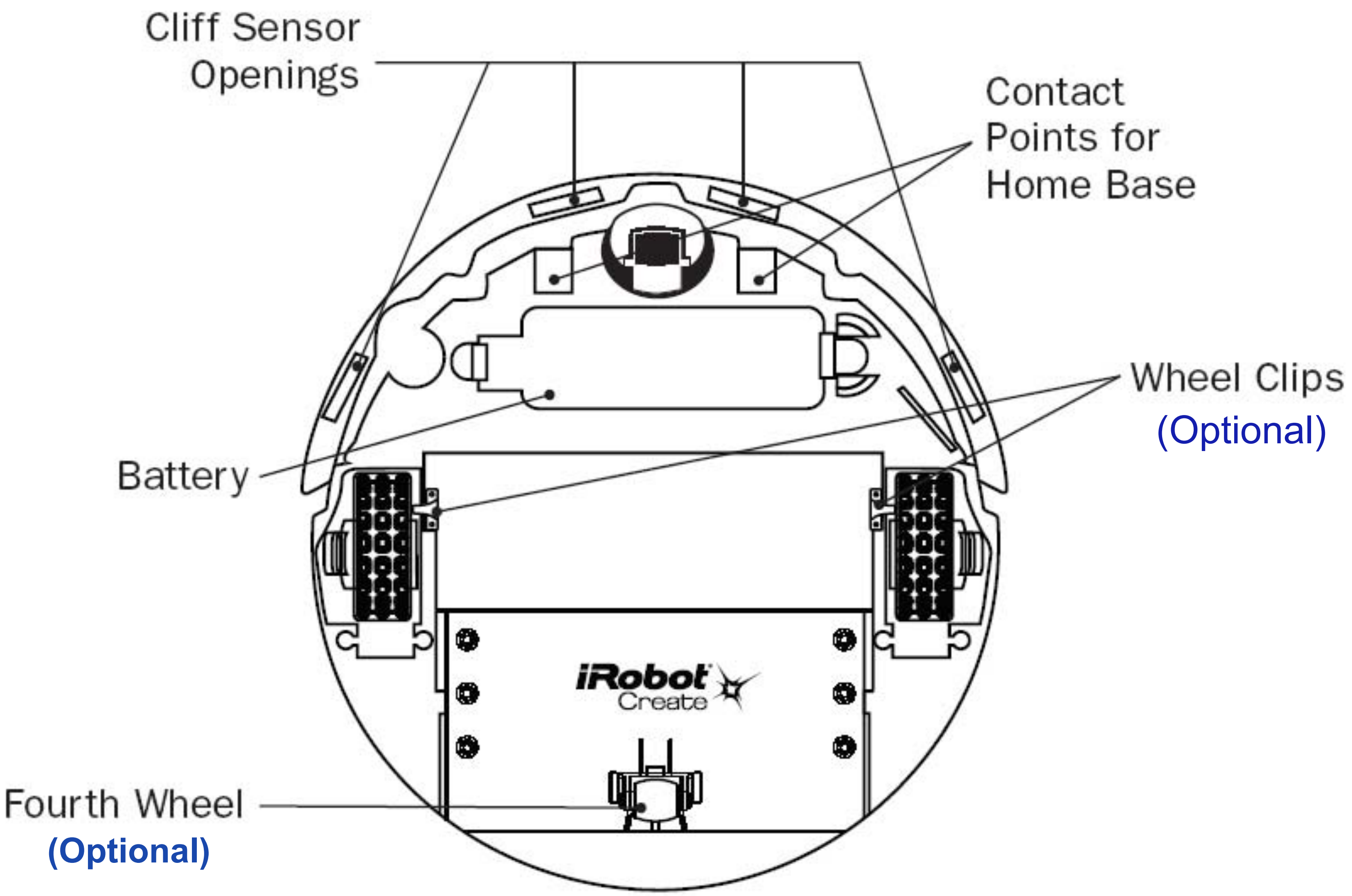
- Digital sensor
- Connect to ports 1-7
- Access with library function **`digital(`***port#***`)`**
- Three form factors in kit
- 1 indicates switch is closed
- 0 indicates switch is open
- These make good bumpers and can be used for limit switches on an actuator

Create sensors

Create Module



(from iRobot Create Owner's Guide)

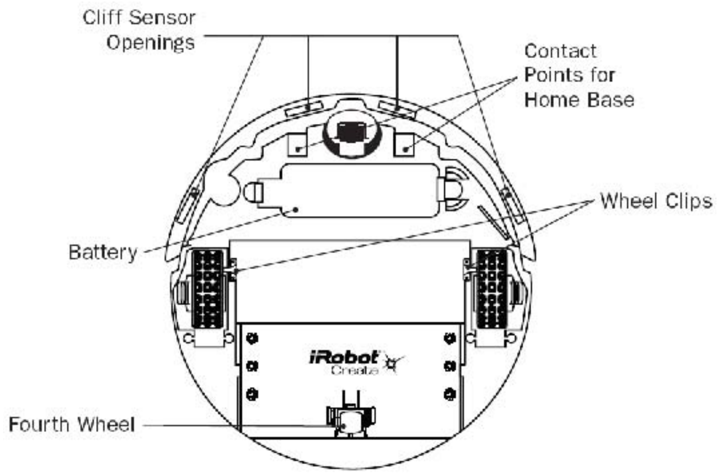


(from iRobot Create Owner's Guide)

Botball

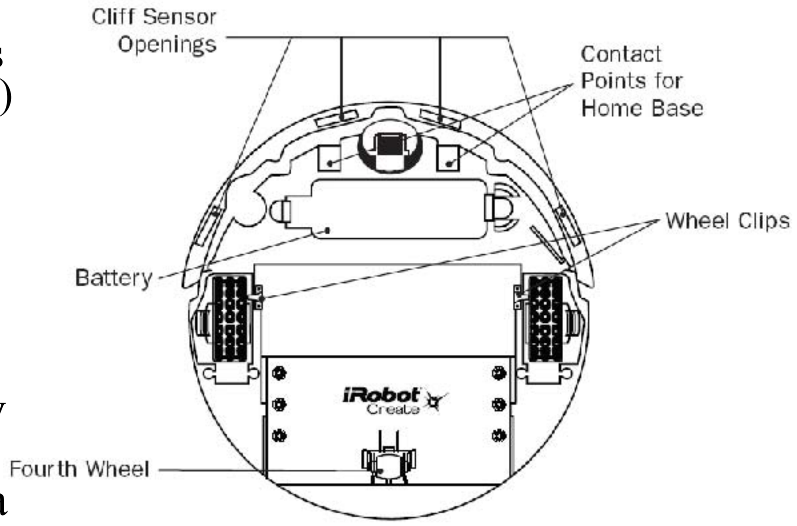
Bumper and Wheel Drop Sensors

- Left and right bump sensors
- Left, right and caster wheel drop sensors
- The **`create_sensor_update()`** library function updates the bumper and wheel drop globals with the current values from the Create
  - Values are *True* (1) or *False* (0)
- Globals are
  - **`gc_fdrop`** *(caster)*
  - **`gc_ldrop`** *(left wheel)*
  - **`gc_rdrop`** *(right wheel)*
  - **`gc_lbump`** *(left bumper)*
  - **`gc_rbump`** *(right bumper)*
- If in "safe" mode (the mode initially established by **`create_connect`**), detection of a wheel drop (on any wheel) will cause Create to stop all motors and revert to "passive" mode
  - In passive mode actuator commands are ignored until returned to "safe" or "full" mode by
    - **`create_safe()`** or **`create_full()`**
  - **`create_mode()`** sets the global **`gc_mode`** which can be checked to determine if something has caused the Create to switch to passive mode



Cliff Sensors

- Located left, left-front, right-front, and right
- The **`create_sensor_update()`** library function updates the cliff sensor globals with the current sensor values from the Create
  - Values are *True* (1) or *False* (0) for "base" variables and 0-4095 for "amt" variables (reflectance strength)
- Globals are
  - **`gc_lcliff`**, **`gc_lcliff_amt`** *(left)*
  - **`gc_lfcliff`**, **`gc_lfcliff_amt`** *(left front)*
  - **`gc_rfcliff`**, **`gc_rfcliff_amt`** *(right front)*
  - **`gc_rcliff`**, **`gc_rcliff_amt`** *(right)*
- If in "safe" mode (the mode initially established by **`create_connect`**), detection of a cliff sensor while moving forward (or moving backward with a small turning radius) will cause Create to stop all motors and revert to "passive" mode
  - In passive mode actuator commands are ignored until returned to "safe" or "full" mode by
    - **`create_safe()`** or **`create_full()`**
  - **`create_mode()`** sets the global **`gc_mode`** which can be checked to determine if something has caused the Create to switch to passive mode



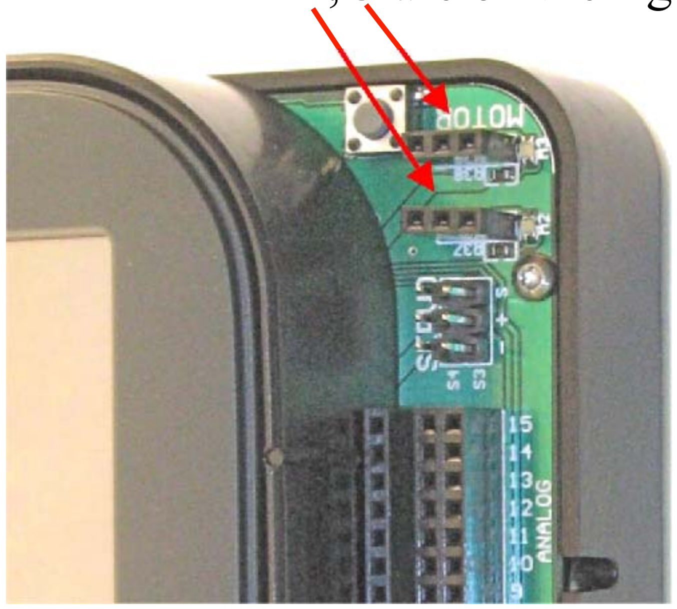
Other

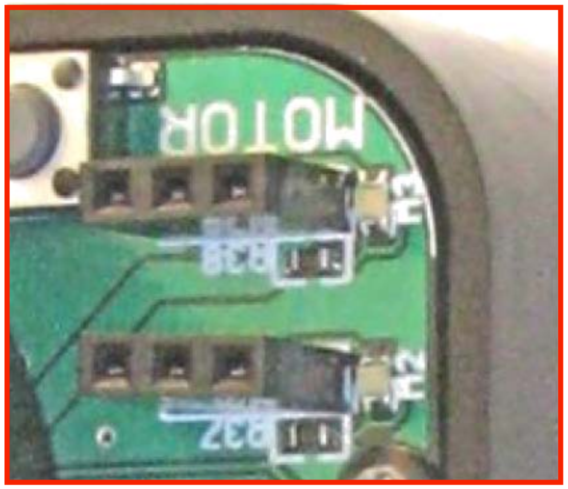
- See the KISS-C (CBC target) manual for information on additional Create functions
- See Create Open Interface manual for gory details on how commands work

Motors & Servos

DC Motors

- DC motors (gray cables or red and black cables with 2 prong plugs) plug into the CBC motor ports
- The CBC has 4 motor ports numbered 0,1,2,3
  - 0, 1 are on the left side
  - 2, 3 are on the right side





DC Motors

- DC motors (gray cable or red and black cable) plug directly into CBC motor ports
- If the motor position counter on the Motor CBC Status Display decreases when the motor is manually turned in what you want to be the forward direction, simply flip the plug 180 degrees to correct
- The gray cable motors have a high stall torque of about 48 in-oz
  - They come with a red horn mounted to the motor shaft. Different horns can be attached by removing the screw in the motor shaft and lifting off the old horn.
  - Lego pieces can be attached to the servo horns using either U-glu or screws. The screwdriver included in your kit can be used for this purpose (the screwdriver cannot be used as part of a robot!)

More DC Motors

- The motors with a red and black cable have a socket for an IFI axle for mounting wheels or gears
- LEGO and/or IFI parts may be attached to the IFI axle
- This motor is a little slower and has slightly higher torque than the black gear motor with the gray cable

Motor Functions

CBC Motor Functions

- The CBC uses intermittent measurements of the motor back EMF (electromagnetic force) to estimate motor position and velocity. This can be considered magic as far as Botball is concerned. See appendix.
- For the DC motors (red and black wire or gray wire) included with the CBC electronics kit, one full rotation = about 1100 "ticks" -- a unit of measure for rotation

Useful CBC Motor Functions

- Get the current tick count of motor 0: **`get_motor_position_counter(0);`**
- Set the current tick count of motor 0 to 0L: **`clear_motor_position_counter(0);`**
- Move the motor backwards at 50% power: **`motor(0,-50);`**
- Move a motor forward at 100% **`motor(0,100);`**
- Turn motor 0 off: **`off(0);`**
- Turn all motors off: **`ao();`**
- Run a motor at a specific velocity (0.5rps) **`mav(0,550);`**
- Run a motor to specific position **`mtp(0,550,5000);`**
- Run a motor about one rev back from current position **`mrp(0,550,-1100);`**
- Wait for the previous command to finish before advancing in your program **`bmd(0);`**
- See **CBC Motors** in the Appendix for more details

Example Using **`motor`**

- This example moves the robot 2200 ticks forward
- The **`while`** loop checks the motor position and then exits when the motor is in the right place.

```
int main()
{
  //show robot world
  kissSim_init(BB09WORLD,
               151,32,1.5708);
  clear_motor_position_counter(0);
  //move both motors fwd
  motor(0, 70);
  motor(1, 70);
  //Now wait till motor 0
  //has rotated about twice
  while(get_motor_position_counter(0)<2200){}
  //turn both motors off
  ao();
  kissSimPause();
}
```

Example Using BEMF

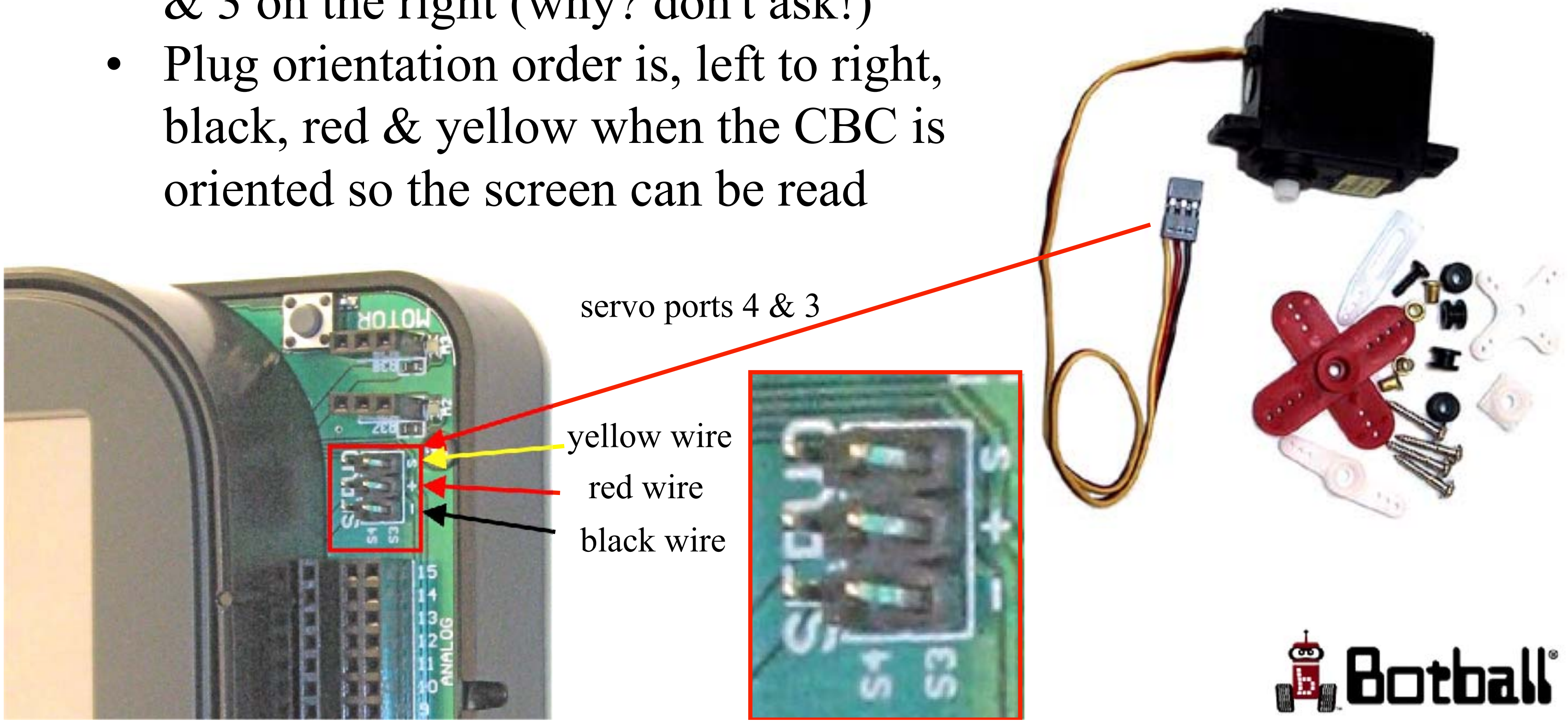
- This example moves the robot 2200 ticks forward, then one motor turns back
- Notice the difference between moving to and moving relative to a position
- Block motor done halts your program until the motor reaches the desired position.
- Move at velocity moves at a velocity, forever or until changed.
- Motors, once they reach their specific destination, will actively stay there until moved somewhere else or turned off.

```
int main()
{
  //show robot world
  kissSim_init(BB09WORLD,
               151,32,1.5708);
  clear_motor_position_counter(0);
  clear_motor_position_counter(1);
  //move both motors fwd
  mtp(1,550,2200); //move both motors fwd
  mav(0,550);
  bmd(1); //wait for motors to reach position
  off(0); //stop the mav motor
  mrp(0,55,-2200); //move motors back to 0
  bmd(0); //wait for motor to reach position
  off(1); //Motor 1 is now limp
  // motor 0 is holding its position actively
  sleep(10); // wait 10 seconds
  ao(); // now everything is off
  printf("All done\n"); // now everything is off
  kissSimPause();
}
```

Servo Motors

Servos

- Servo motors (black-red-yellow cables with 3 prong receptacle) plug into the CBC servo ports
  - These are the ones with a white horn installed
- The CBC has 4 servo ports numbered 2 & 1 on the left and 4 & 3 on the right (why? don't ask!)
- Plug orientation order is, left to right, black, red & yellow when the CBC is oriented so the screen can be read



Servos

- Servos are motors that are designed to rotate to a specified position and hold it
- **`enable_servos();`**
  - Activates all servo ports
- **`disable_servos();`**
  - De-activates all servo ports
- **`set_servo_position(`***<s#>*`,`*<pos>*`);`**
  - Rotates servo in the specified port to the specified position
  - **`set_servo_position(2,1234);`**
    - Sets the position for servo port 2 to 1234
    - If servos are enabled, the servo in port 2 rotates to position 1234
  - Position range is 0-2047
  - You can preset a servo's position before enabling servos
  - Default position when servos are first enabled is 1024
- **`get_servo_position(`***<s#>*`)`**
  - Returns an **`int`** for the specified servo whose value is the current position for which the servo is set
- Note: Servos may run up against their stops at low or high position values. Giving a servo such a position command will suck power at an alarming rate!
- Note: Servos acting weird or not working is an indication the battery is low

Example Servo Program

```
int main(){
   enable_servos(); // turn servos on and
                    // rotate to default position (1024)
   printf("moving to 640\n");
   set_servo_position(3,640); // rotate servo 3 to 640
   sleep(2); // give plenty of time for servo to move
   printf("moving to 1800\n");
   set_servo_position(3,1800); // rotate servo 3 to 1800
   sleep(2);
   printf("moving to 640\n");
   set_servo_position(3,640); // rotate servo 3 back to 640
   sleep(2);
   disable_servos(); // turn servos off
   printf("Done\n");
}
```

Charging and CBC Motors

- When charging, the battery (7.2v) is disconnected and the system is powered by the charger (13.5v)
- Motors and servos run faster at higher voltage
- BEMF motor commands will try and run at the correct speed, but the PWM commands will run much faster with the charger plugged in
- Motors and servos will jitter more and behave slightly differently when the CBC is plugged into the charger or is charging from the Create then they will when running on battery

Create Motor Functions (1)

- The Create uses differential steering with two motorized drive wheels
  - The drive wheels and the forward pivoting wheel are spring loaded so that if the Create is picked up (or runs off of a cliff) the wheel drop can be detected
    - Clips can be attached to the drive wheels to keep them from dropping
  - An optional (stabilizer) non-pivoting wheel can be installed at the rear
    - Keeps Create from doing a "wheely" causing a forward wheel drop
- The library function **`create_drive_direct`** allows independent control of wheel speeds
  - `void create_drive_direct(int r_speed, int l_speed);`
    - **`r_speed`** for the right motor and **`l_speed`** for the left motor
    - Speeds can range between +/-500mm/sec

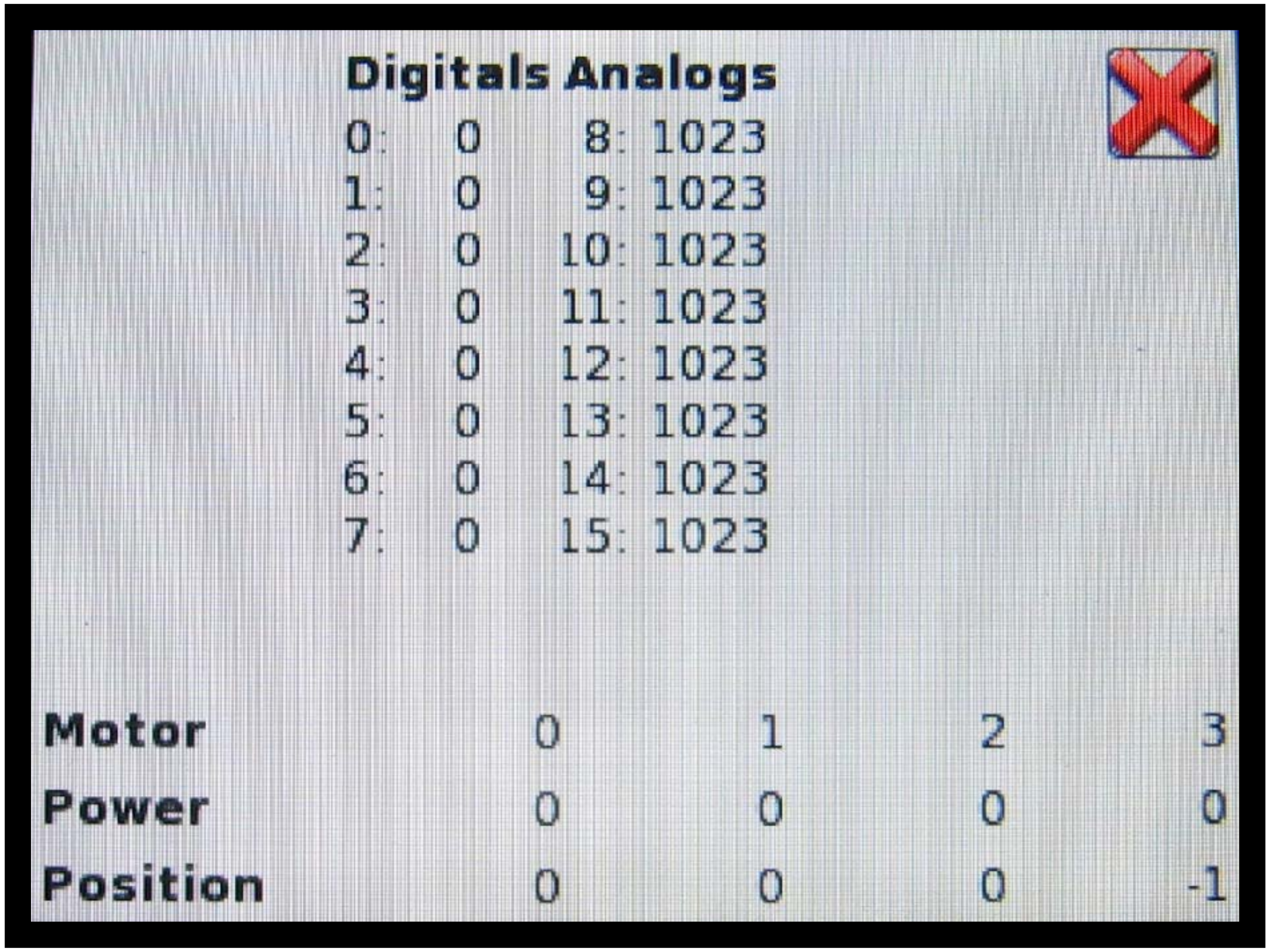
Create Motor Functions (2)

- The **`create_drive`** function moves the Create along an arc of specified radius
  - **`void create_drive (int speed, int radius)`**
    - speeds can range between +/-500mm/sec
    - radius is -2000 to +2000 mm
      - radius < 0 for an arc turning right
      - radius > 0 for an arc turning left
    - radius of arc is measured to center of Create between the wheels
- Special cases
  - a radius of 32767 will drive the robot straight
    - Implemented as the **`create_drive_straight(<`***speed***`>)`** library function
  - a radius of 1 will spin the robot CCW
    - Implemented as the **`create_spin_CCW(<`***speed***`>)`** library function
  - a radius of -1 will spin the robot CW
    - Implemented as the **`create_spin_CW(<`***speed***`>)`** library function
  - a radius of 0 and speed of 0 will stop the robot
    - Implemented as the **`create_stop(<`***speed***`>)`** library function

Testing Motors and Sensors on the CBC

CBC Motors and Sensors Screen

for Testing Sensors and DC Motors



```
Digitals Analogs
0:   0    8: 1023
1:   0    9: 1023
2:   0   10: 1023
3:   0   11: 1023
4:   0   12: 1023
5:   0   13: 1023
6:   0   14: 1023
7:   0   15: 1023

Motor        0     1     2     3
Power        0     0     0     0
Position     0     0     0    -1
```

Using the CBC with the Create Demobot

Exercise #10: CBC & Create

1. Gather (see pictures on next 3 slides) :
   a) Your Demobot
   b) A CBC
   c) The Create/CBC cable
   d) USB flash drive
2. Plug the lever sensor at the end of the arm into digital port 1
3. Plug the lever sensor near the motors into digital port 0
4. Plug the servo into servo port 4
5. Plug the motor into motor port 3
6. On the CBC go to the Sensors page
   1. Look at the position listed for motor 0
   2. Gently pull up on the end of the arm and notice how the value shown on the CBC changes
   3. If the value goes down, unplug the motor, rotate the plug 180 degrees and insert it back into motor port 0
   4. If the value goes up when the arm is moved slightly up -- leave the plug as it is

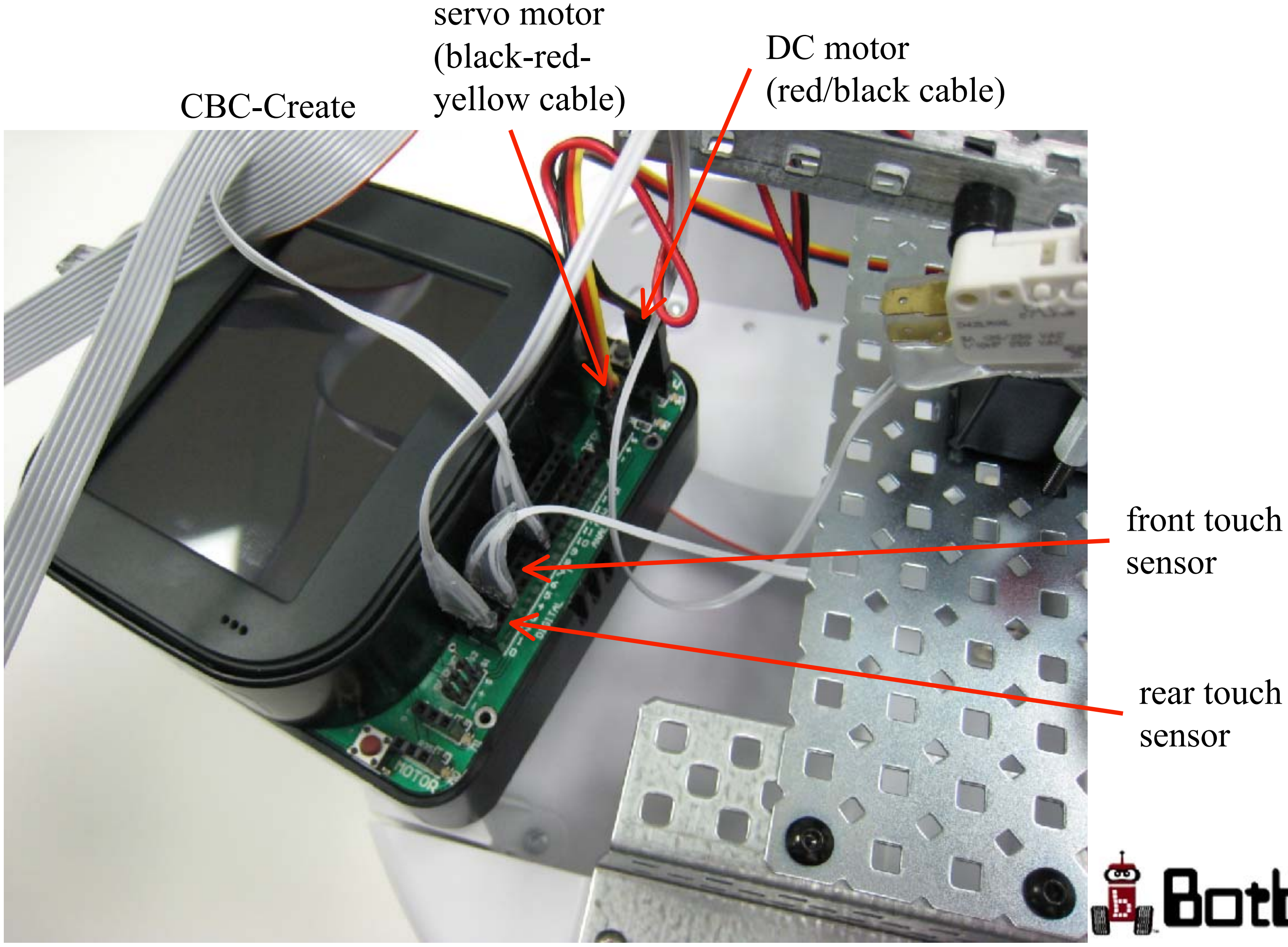
Exercise #10: Continued

1. Make a new program in KISS-C (see code on next slide)
2. Make sure the program compiles in KISS-C
3. Copy the program onto a flash drive, and rename it on the flash drive to `robot.c`
4. Insert the flash drive into the CBC
    1. select the Programs page
    2. press the Compile button
        - If it says flash key not found, wait a few seconds and press Compile again
        - If it reports errors, note what they are, and alter your program accordingly (you will need to remove the flash drive, copy the corrected version onto the flash and recompile it on the CBC
        - If it compiles successfully, remove the key
5. Make sure the CBC is correctly connected to the Create
6. Turn on the Create and unplug the USB cable from the CBC
7. Run your program on the CBC to move the arm
    - The program requires you to press the black button to start the program. If your arm seems to be jamming, or goes to far, press and release the black button as needed to stop the program

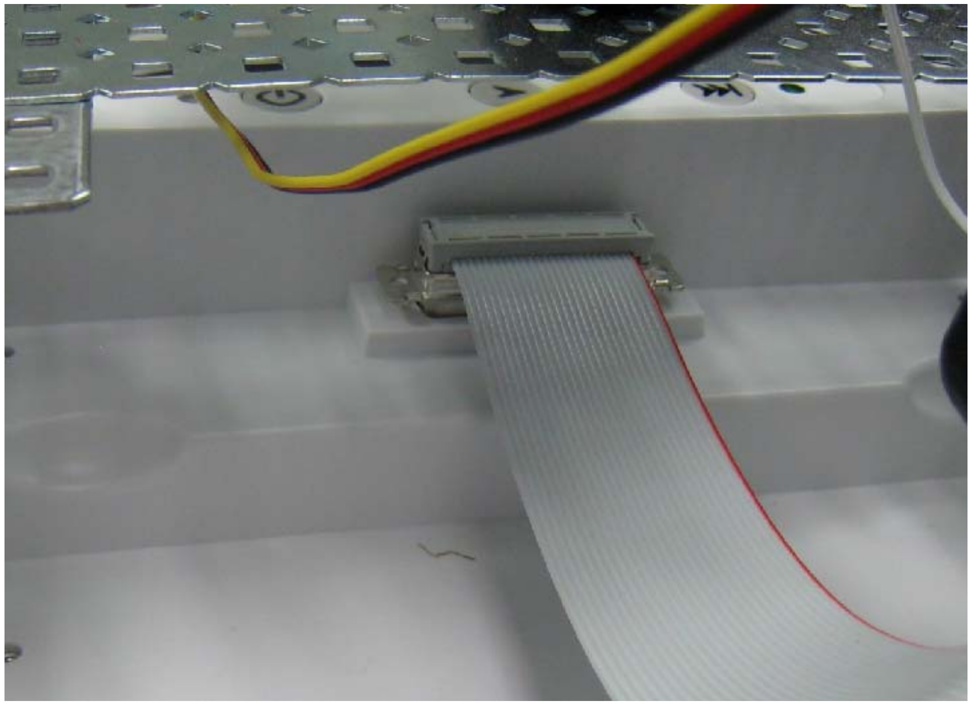
Exercise 10: Code

```
// Moves the demobot arm to the middle position
int main() {
   int upPos, downPos, middlePos;
   printf("Press Black Button to start\n");
   while(!black_button()); while(black_button()); // press & release
   set_servo_position(4,1023); // set servo position for middle
   enable_servos(); // turn on servos
   motor(3,50); // start arm up
   while(!digital(0) && !black_button()) msleep(10); // dig 0 is up sensor
   off(3);
   upPos = get_motor_position_counter(3);
   while(black_button()) {} // if pressed, wait for release
   motor(3,-50); // start arm down
   while(!digital(1) && !black_button()) msleep(10); // dig 1 is down sensor
   downPos = get_motor_position_counter(3);
   off(3);
   while(black_button()) {} // if pressed, wait for release
   middlePos = (upPos - downPos)/2 + downPos;
   mtp(3,500,middlePos); // Move arm to specific position
   bmd(3); // wait until motor has reached destination
   printf("At the middle position\n");
   ao(); disable_servos(); // all motors and servos off
   printf("Program finished\n"); return 0;
}
```

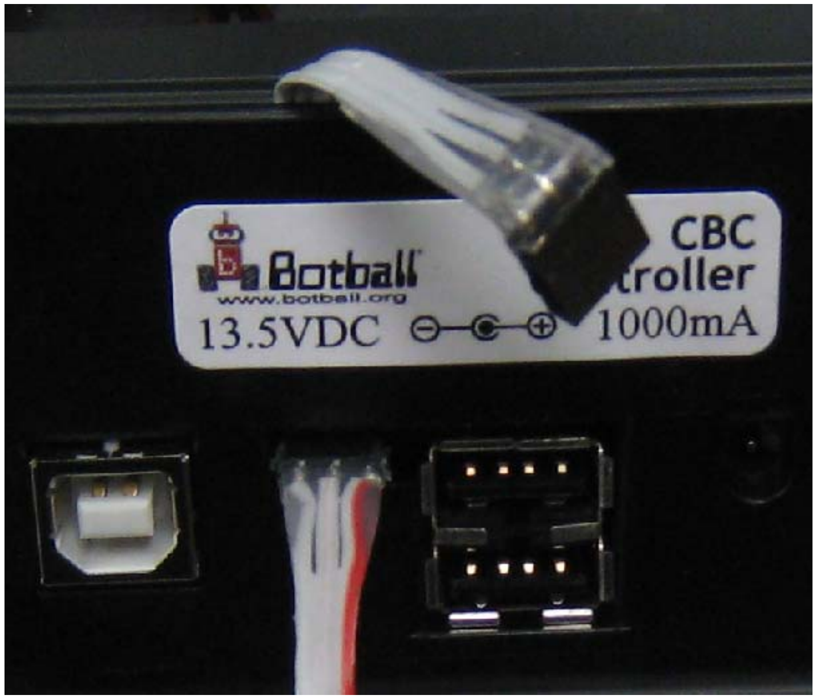
Attach Demobot's Cables


CBC-Create
servo motor (black-red-yellow cable)
DC motor (red/black cable)
front touch sensor
rear touch sensor

Cargo Bay & Serial Connections
CBC-Create Cable

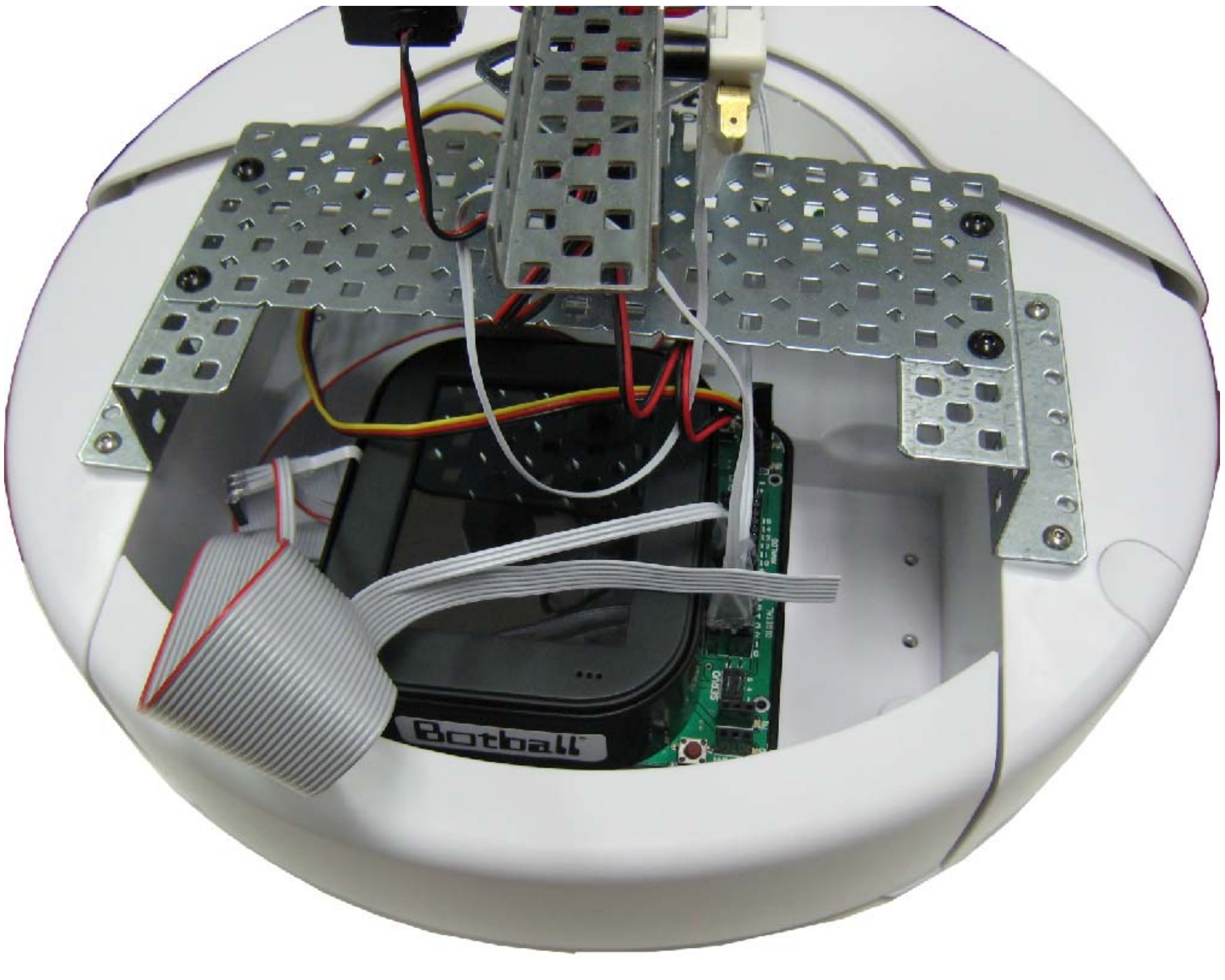


Cargo Bay connection



Serial connection – won't work unless plugged in with red side towards the two USB ports

CBC in Cargo Bay



Exercise #11: Experiments with Demobot

- Type in and run the color_line program given earlier
- You will need to first make a color model on channel 0 that tracks some object you can put in front of your robot
- After this program has been loaded on your demobot and is working
  - Modify it so it slows down when the something gets closer (use a sensor, or the Y position in the image)
  - Add in an arm motion when you get near an object
  - Be creative (or move on to the last exercise)

Exercise #12
Testing Your CBC, Sensors & Motors

- Turn on your CBC
- From the main menu select the motors & sensors screen
  - Connect and test all of your motors; be sure to use all 4 motor ports to ensure all of the ports are working as well (you will need to write a simple program)
  - Connect and test the servos (one at a time)
  - Remember that the digital ports are 0-7 and the analog ports are 8-15, with 13-15 reserved for ET and sonar
  - Connect sonar and ET sensors in ports 13-15 and check for values changing on the status screen in response to moving the sensor
  - Connect and test the digital sensors in ports 0-7 and check to see if triggering them changes corresponding values on the status screen
  - Connect and test the remaining analog sensors (light and reflectance) and note if values change on the status screen in response to light/reflectance
- Select the vision system
  - Install the camera in a USB port and test the vision system; focus can be adjusted by gently turning the front of the lens

Critical Things to Know if You Don't Want to Embarrass Yourself at the Tournament

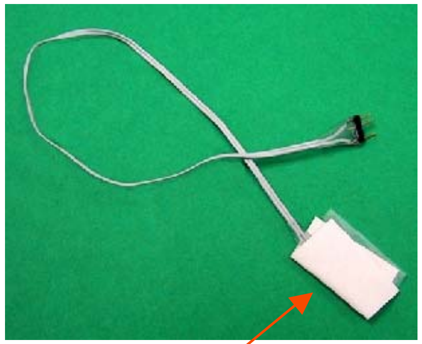
Shielding Light Sensors

YOU MUST SHIELD YOUR LIGHT SENSOR

- The table will be brightly lit
- Overhead lights from the game table will flood an unshielded sensor rendering it incapable of seeing the starting light
- Light sensors only need a little light to work, and it should be shielded from all extraneous sources
- Opaque objects stop 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.

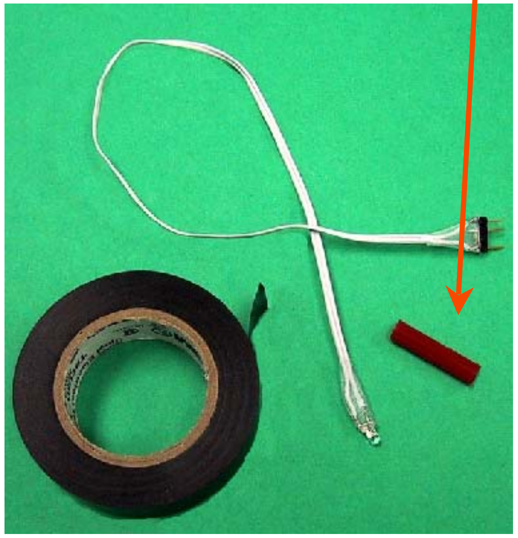
How to Shield a Light Sensor

Wrap segment of plastic straw in tape

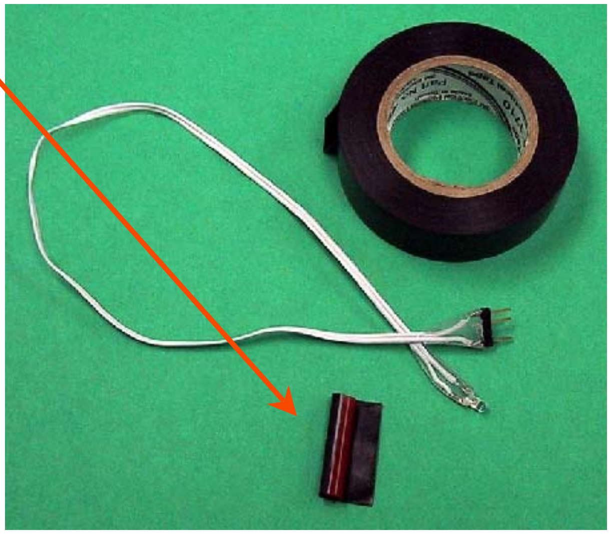


No!!

1

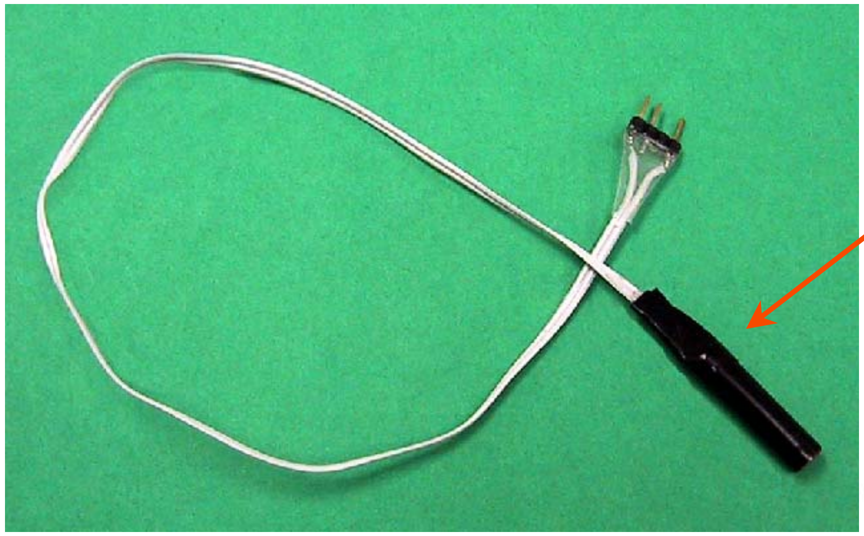


2



3

Yes!



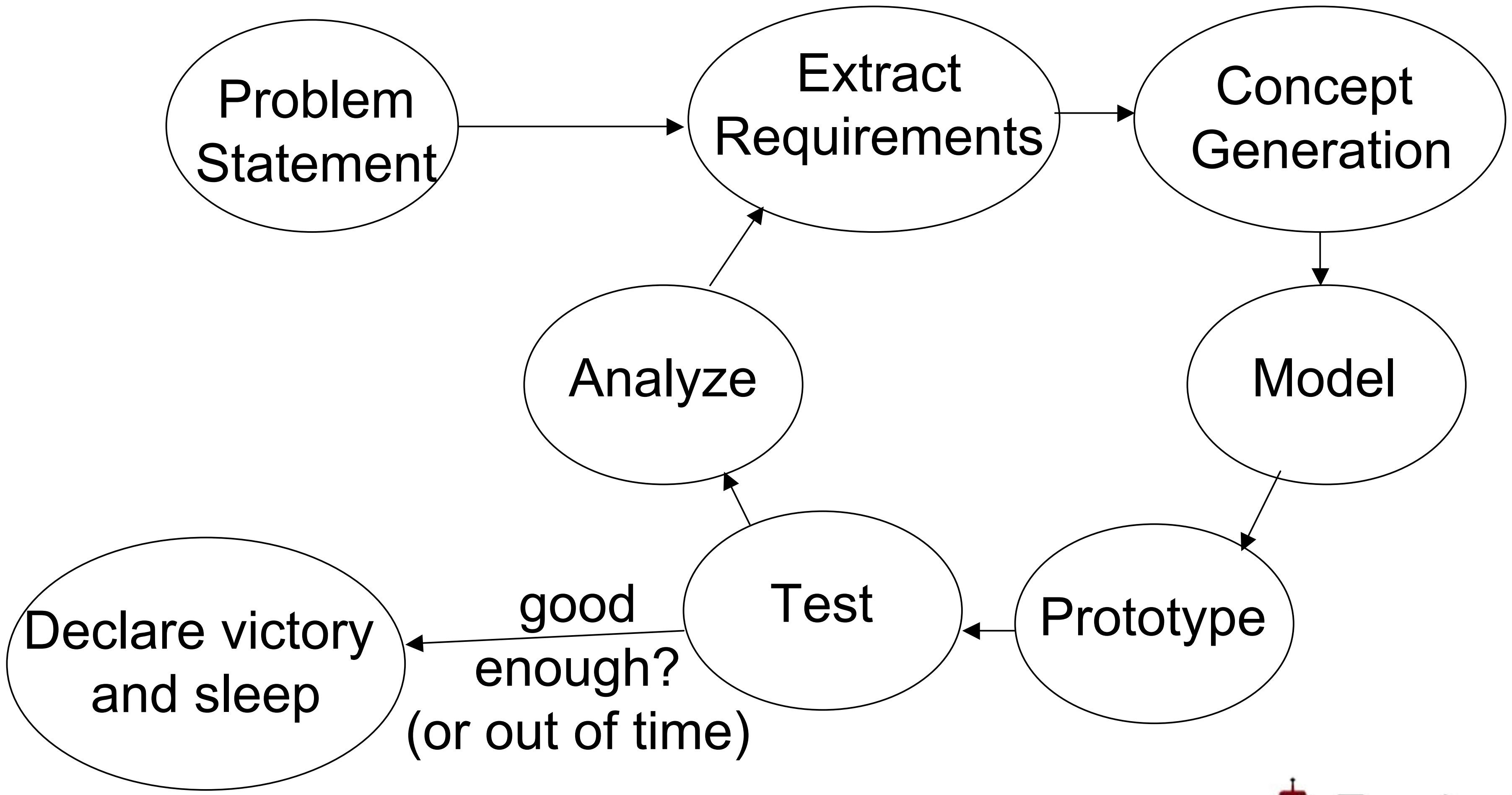
Slide straw over light sensor (leave a gap in the front) and tape in place

Exercise:
Create A Botball Program

Have your Team Do this the First Week!!!

- Modify one of your workshop programs and be sure to incorporate the Botball utilities:
  - Add a light sensor to your robot and use the **`wait_for_light(`***<port>***`)`** function to calibrate it
  - Use the **`shut_down_in(`***<time>***`)`** function to turn your robot off after 30 seconds
  - Remember: In the real situation, the light sensor will require shielding

Engineering Life Cycle



Handy References

- homebase.kipr.org
- NASA Robotics Alliance Project: CBC course
  - http://robotics.nasa.gov/courses/summer06
- Gears
  - http://www.kipr.org/curriculum/gears.html
- Tutorial on how Back EMF works:
  - http://www.acroname.com/robotics/info/articles/back-emf/back-emf.html
- Navigation lessons & integrating Botball in a class
  - http://www.botball.org/educational-resources/curriculum.php
- Good explanation of Multi-voting
  - http://www.ca.uky.edu/agpsd/multivot.pdf
- More details on using 6 Hats (DeBono Hats)
  - http://www.mindtools.com/pages/article/newTED_07.htm

END

Appendices

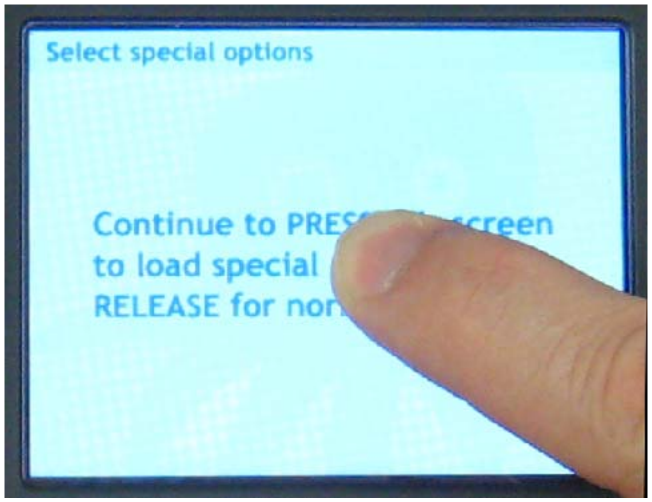
Appendix Index

Kernel Update

- A firmware update may also require doing a kernel update – if this is needed you will be told! (this is a rare event)
- After doing the firmware update and pressing the (red) boot/off button turning off the screen, you access the CBC's special options menu by pressing on the CBC's touch screen while booting the CBC

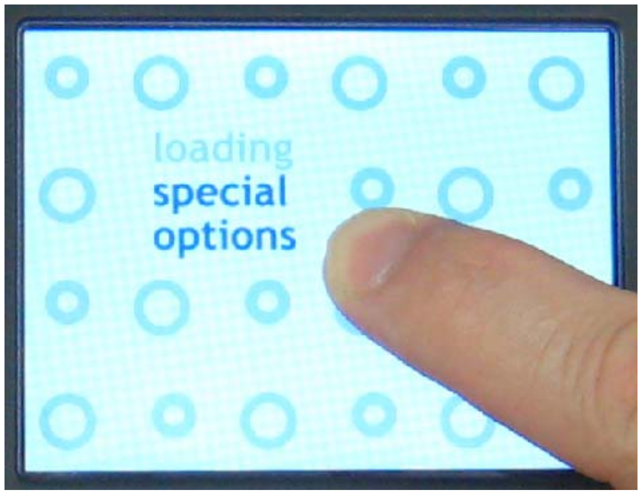
Kernel Update (2)

- Continue pressing the screen until the special options menu appears



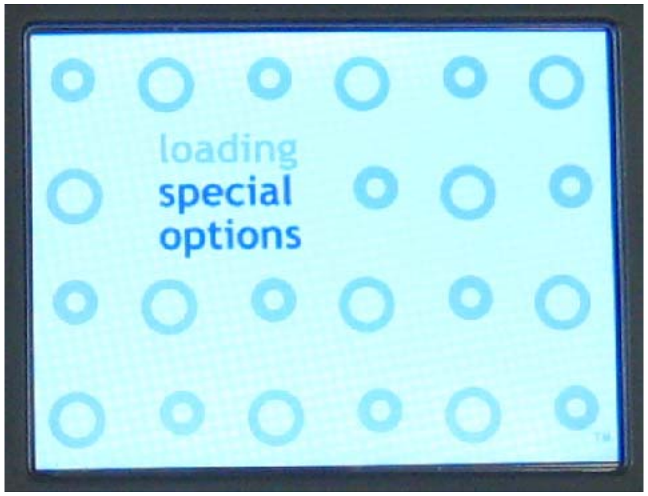
Select special options

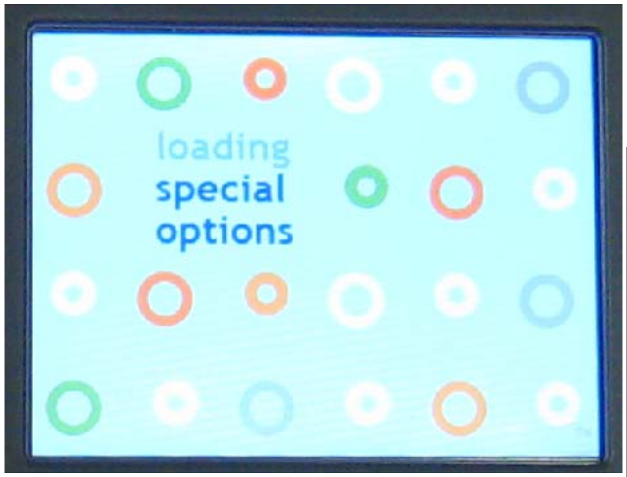
Continue to PRES screen to load special RELEASE for nor

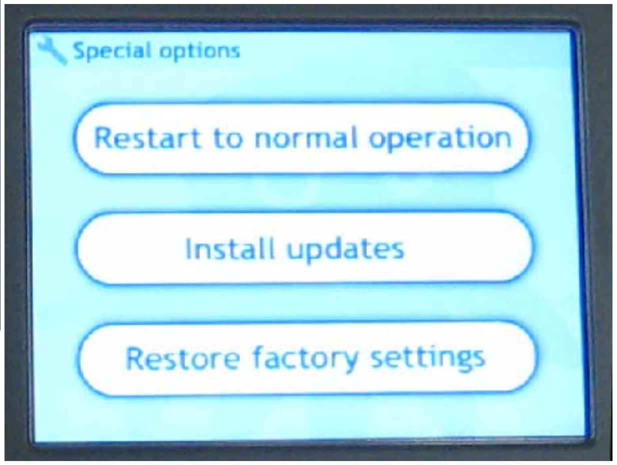


Kernel Update (3)

- Once the special options menu has loaded, press "Install updates"
- The system will install the kernel update from your firmware update and reboot

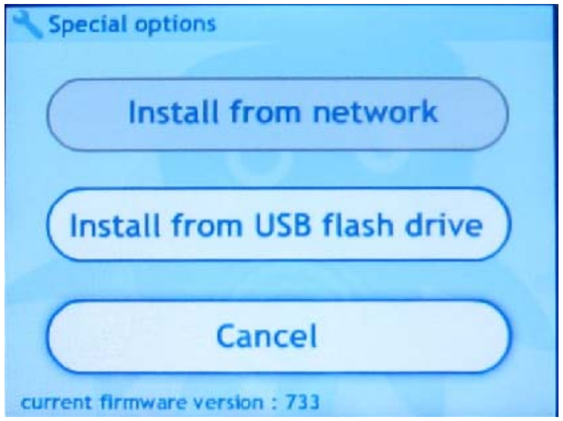




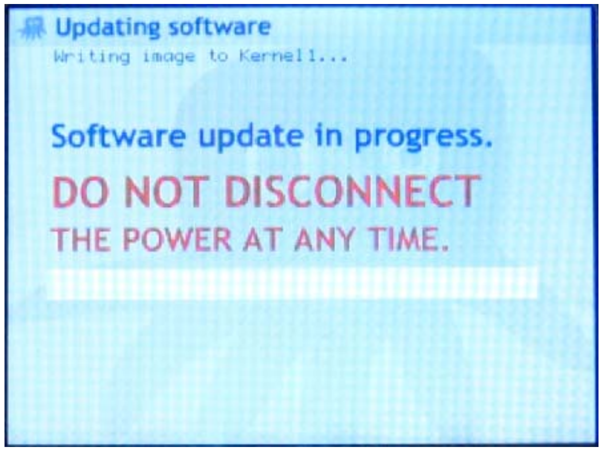


Kernel Update (4)

- Press "Install from USB flash drive"
  - This is the CBC's internal flash drive, not the memory stick you plugged in for firmware update



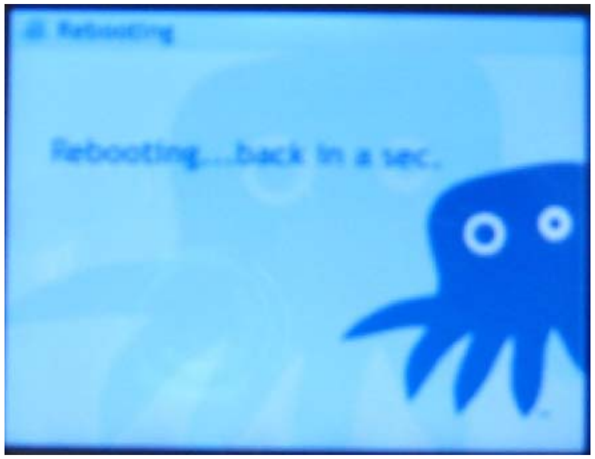
current firmware version : 733



Writing image to Kernel1...

Software update in progress.

DO NOT DISCONNECT THE POWER AT ANY TIME.



Rebooting...back in a sec.

Screen Calibration

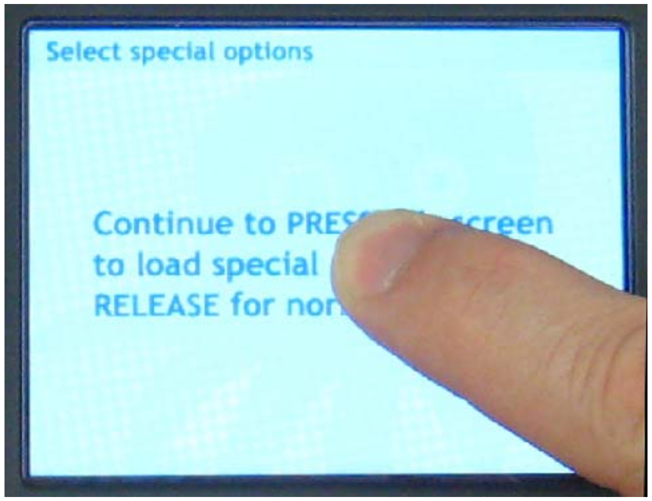
- After doing a firmware update, or a kernel update, the system may require you to re-calibrate the touch screen
  – If this is needed, the system will bring up the calibration screen on its own
- If you calibrate the screen incorrectly, or if you wish to redo the screen calibration, you can start up the calibration process on your own

Manually Initiating Screen Re-Calibration

- Shutdown the CBC using the onscreen power button, or if necessary, pressing the (red) boot/off button.
- You access the CBC's special options menu by pressing on the CBC's touch screen while booting the CBC

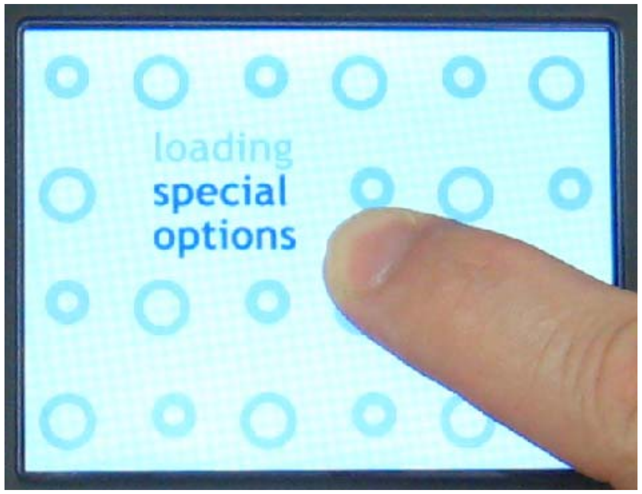
Screen Re-Calibration

- Continue pressing the screen until the special options menu appears



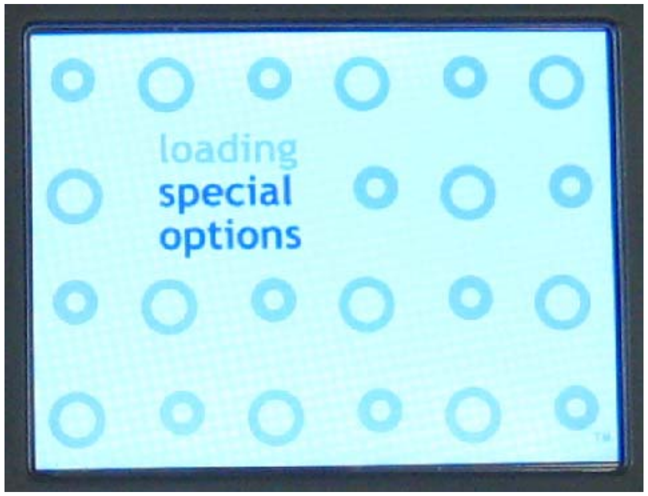
Select special options

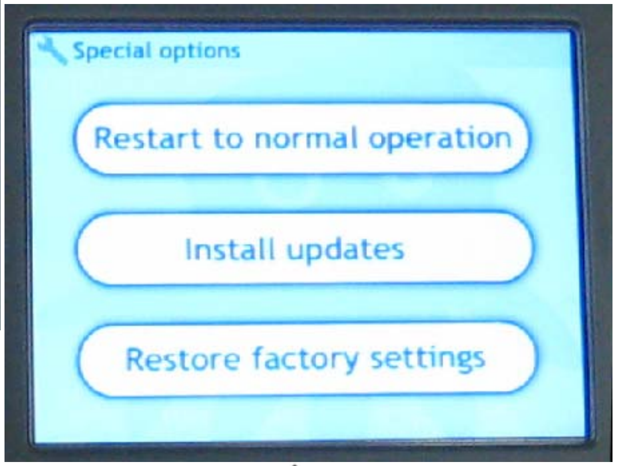
Continue to PRES[illegible] screen
to load special
RELEASE for nor



Re-Calibration

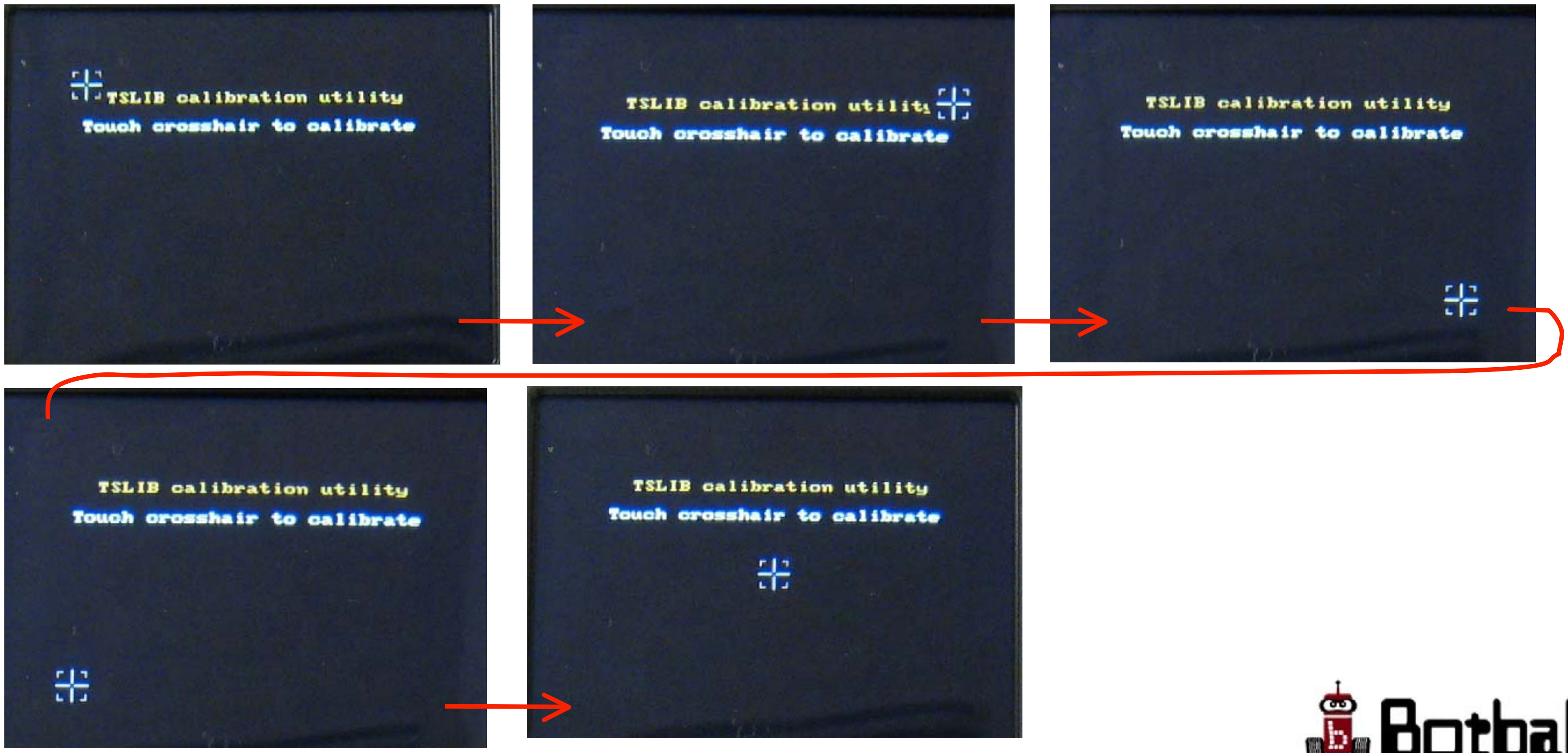
- Once the special options menu has loaded, press "Restore Factory Settings"
- The system will wipe the calibration settings, reboot and bring up the calibration screen





Calibration

- When the CBC reboots, the screen calibration routine will run
- Do calibration carefully
- If calibration is done incorrectly, you can fix it later by booting into special options and pressing "Restore Factory Settings", which will rerun the calibration program



TSLIB calibration utility
Touch crosshair to calibrate

TSLIB calibration utility
Touch crosshair to calibrate

TSLIB calibration utility
Touch crosshair to calibrate

TSLIB calibration utility
Touch crosshair to calibrate

TSLIB calibration utility
Touch crosshair to calibrate

Botball®

Botball is an R&D Project

Project Management

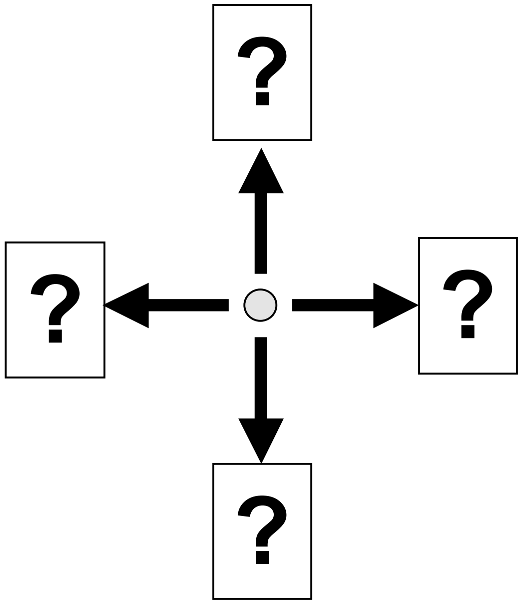
- Projects fail because of poor management at all levels
- In the appendices there are do's and don'ts for organizing and managing projects
- Don't wait until your team members mutiny before trying to implement some or all of these suggestions

The Design Process

- In addition to an organized team, you need a process for the team to follow.
- The engineering design process can provide a good model:
  1. Define the problem through a list of requirements
  2. Explore the solution space
  3. Select a solution
  4. Prototype and refine the solution
     - Repeat 4 at increasing levels of fidelity

Start with a task

What are the requirements?



start when the light goes on

turn off all motors at the end of the round

don’t shoot untethered projectiles

use only the parts supplied in the kit

no entanglement of the other robot

fit in the start box

Design Stages

- Once a problem/task has been defined, and the requirements have been identified, then a design goes through the following:
  1. Conception
     - create ideas come up with specific goals
  2. Evaluation
     - select the promising ideas and interesting goals
     - Derive the design requirements--those unique to your selected idea and goals.
  3. Implementation
     - Create a project plan with a schedule and task assignments.
     - Implement all of the parts; integrate them together; test; test; sleep; test.

Conception

- Look at the problem (both in text and graphics)
- Let it stew for some time
- Then start an idea generation exercise
- There are many different methods, e.g.:
  - Systematic
    - Examine past solutions
    - Examine solutions to similar problems
  - Intuitive
    - Brainstorming
    - Brain Writing

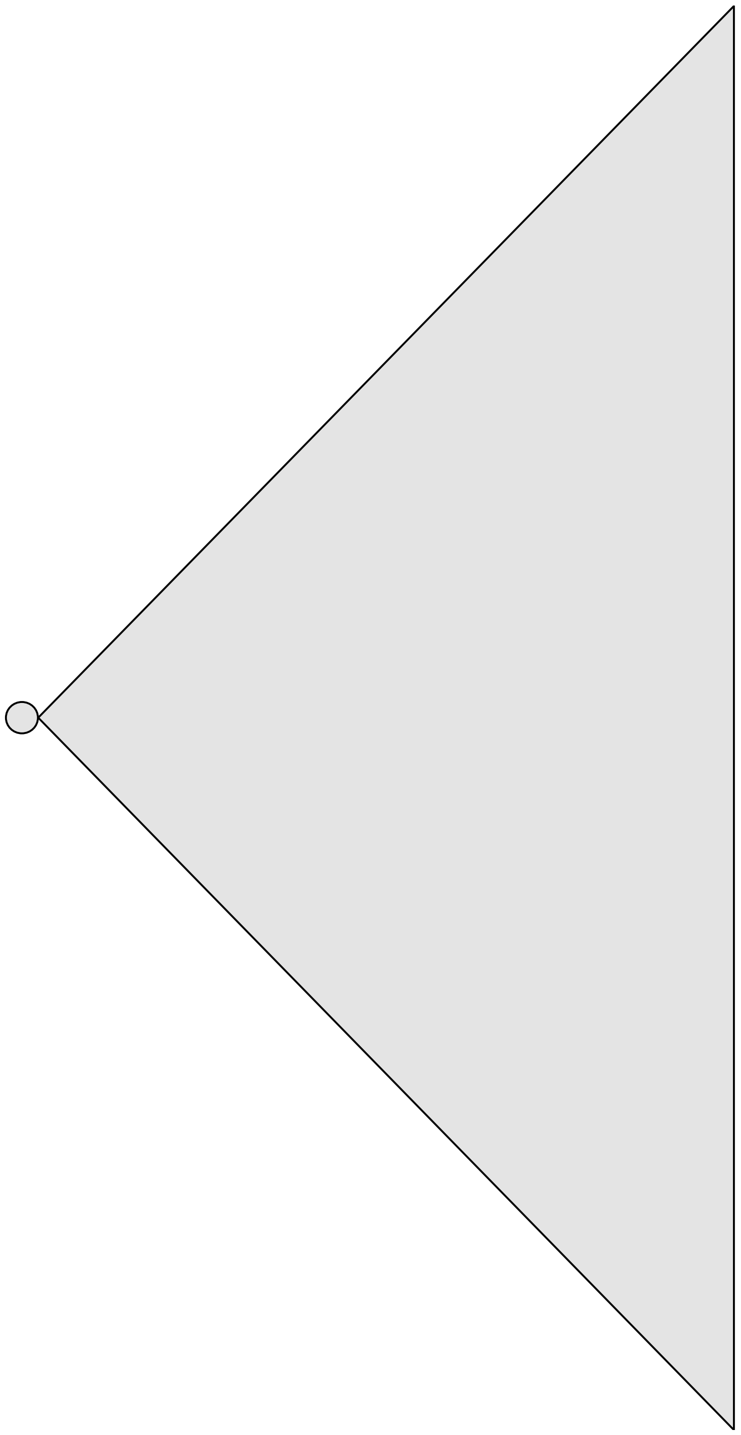
Brainstorming

1. formulate the task as a question “How can we ….?”
2. take a few minutes in silence to individually write down ideas
3. keep your ideas short and snappy
4. each person reads out one idea
5. no criticism of ideas is allowed - reserve judgment - crazy ideas welcome (THIS REQUIRES DISCIPLINE)
6. build on or combine the ideas of others to create additional ideas
7. record ALL the ideas

Brain Writing

1. Identify problem
2. Sit in circle
3. Each person silently generates an idea and writes it down
4. Ideas are passed to the right
5. Person uses the idea they were passed and expands on it or uses it as a primer for a new idea
6. Repeat the process

Idea generation creates the space of what you can do

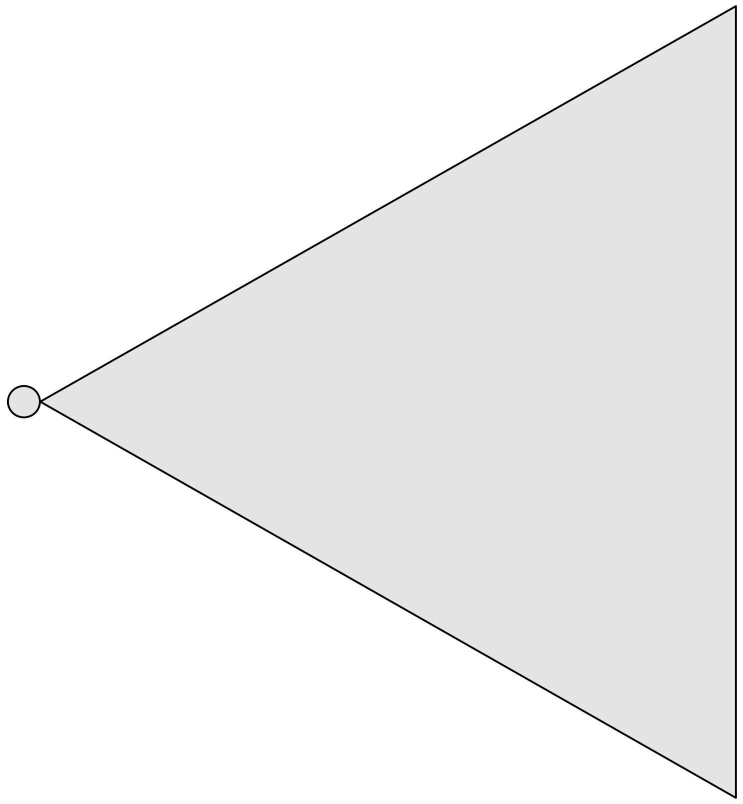


Evaluation

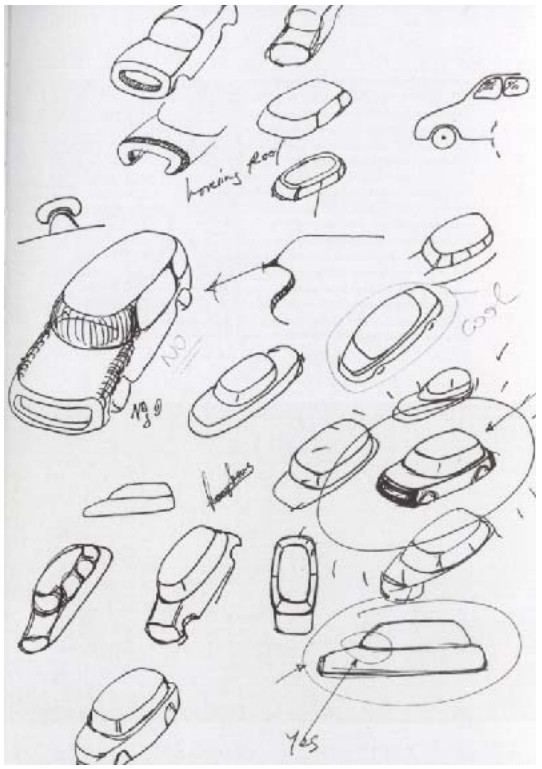
- Go from many ideas to a few promising ideas
- Eliminate those that will not solve problem (clearly do not meet the requirements)
- Eliminate those that clearly cannot work (violate laws of physics)
- Eliminate those that cannot be done by your team (require skills or time that is definitely not available)
- Use evaluation techniques (e.g., six hats) to help with selection
- Use selection technique (e.g., multi-voting) to reach consensus
- If there are several good ideas, all of which will work, vote on the one that the team should pursue
- Do not pursue multiple solutions unless there are adequate resources to complete them -- or unless a firm date to descope to a single solution is agreed upon

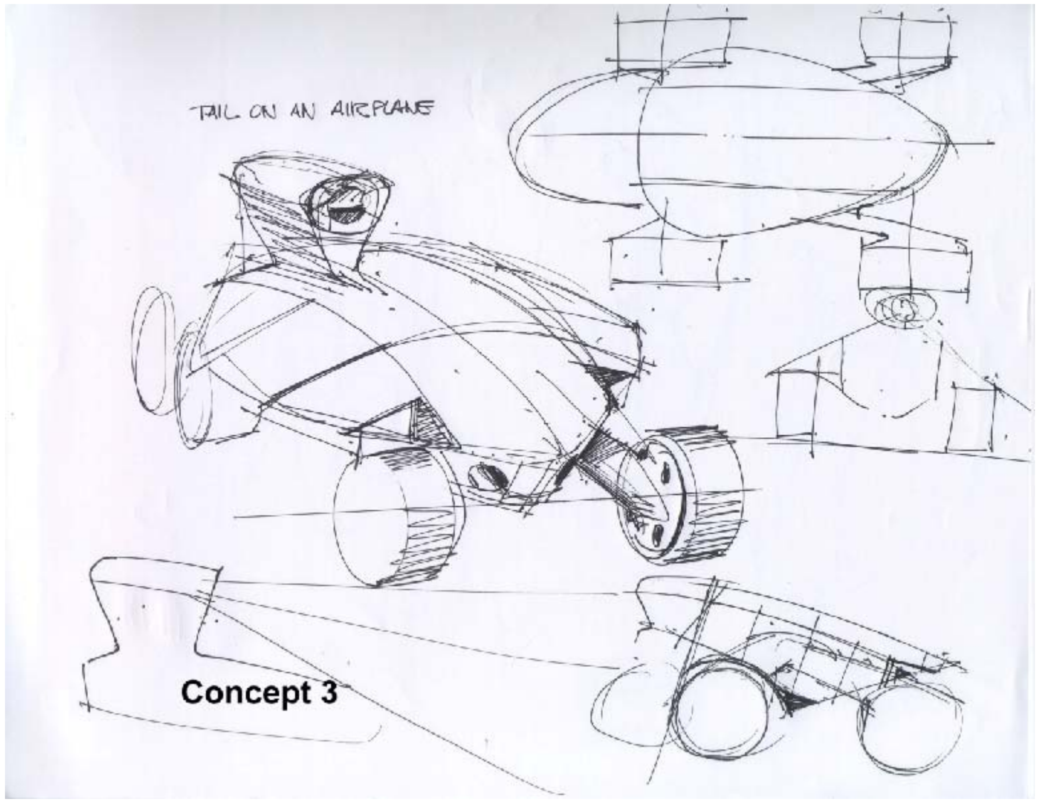
Decide what you want to do

Your team's goals will limit what you should do

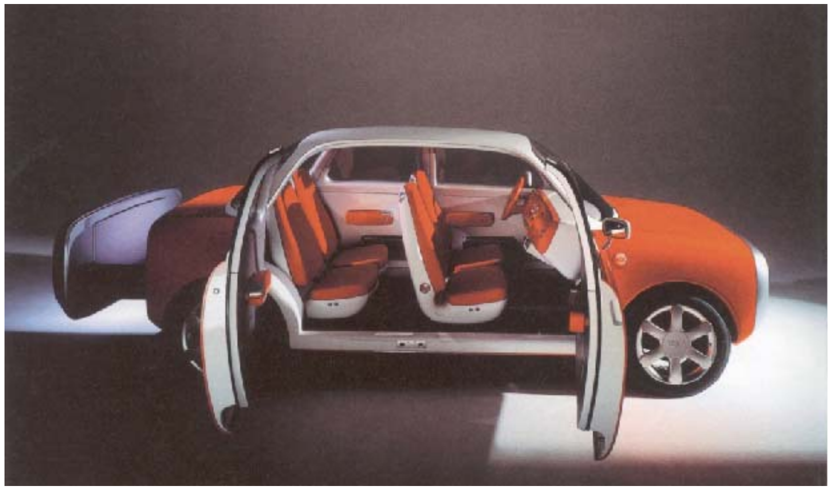


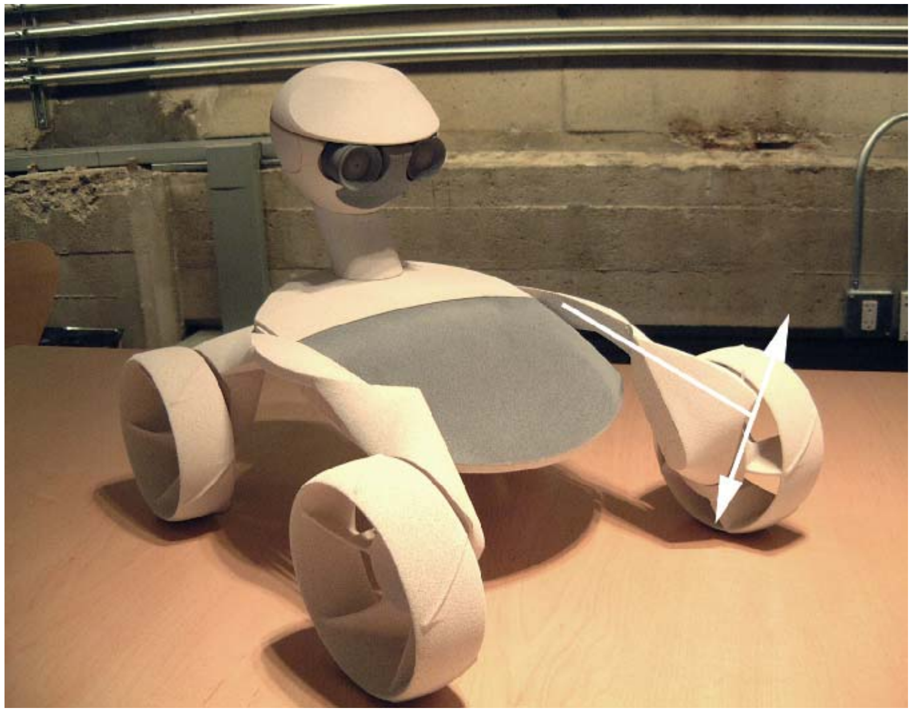
Create Concept Sketches of Promising ideas





Create Sketch Models of the Best Concepts





Perform a preliminary simulation

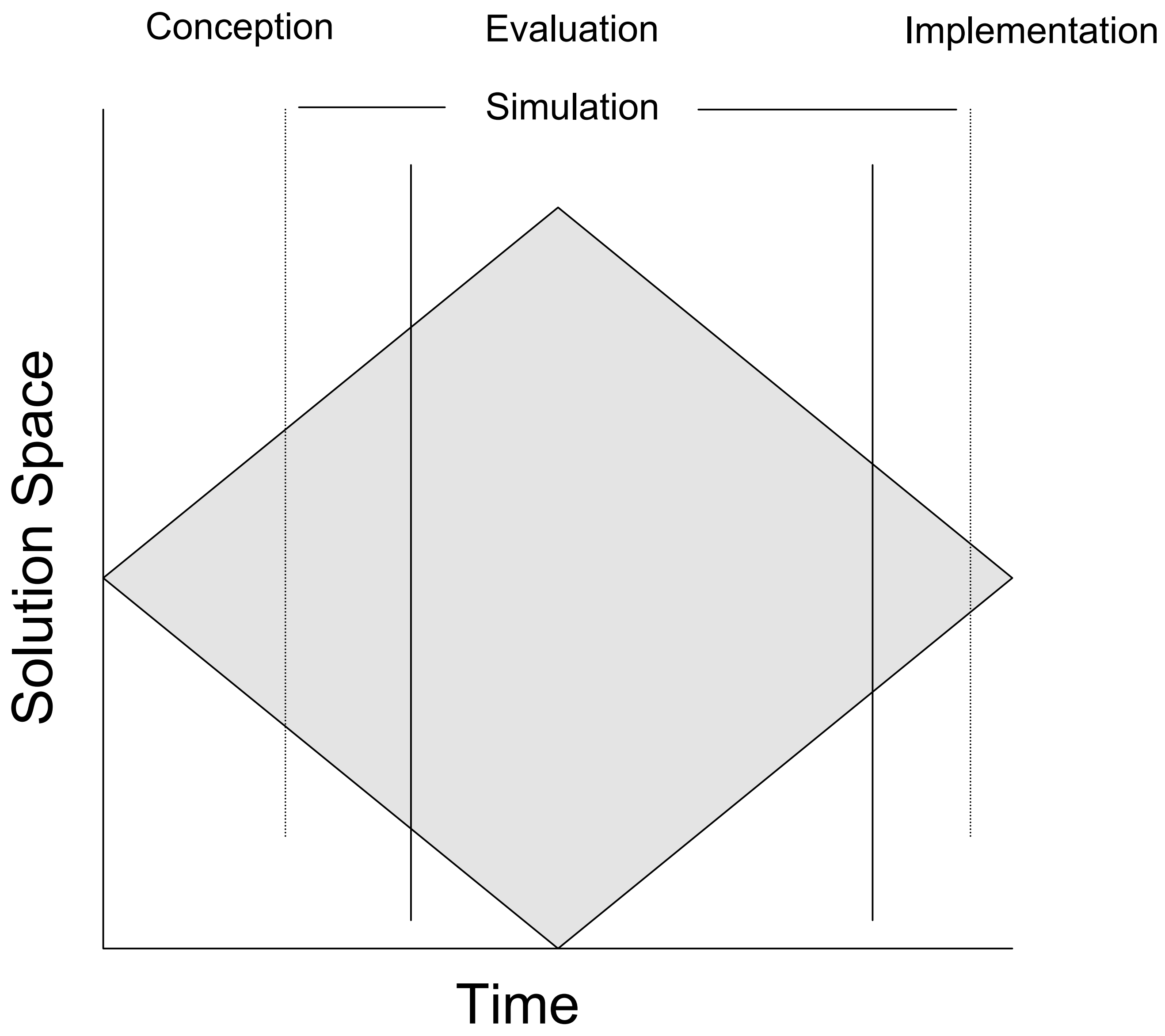
- Move sketch model as a puppet through the motions and through the scenario
- Have team members act out the parts of the robot to make sure the ideas make sense
- Create a CAD model and perform a kinematic and/or dynamic simulation
  - We've provided you with an excellent CAD package (SolidWorks)
  - We've provided you with models of all the kit parts
  - SolidWorks can be used to do both kinematic and dynamic simulation
  - Judges are easily wowed by use of SolidWorks on your documentation
- Simulation will help eliminate some ideas and spark new ones

Create a full experimental prototype

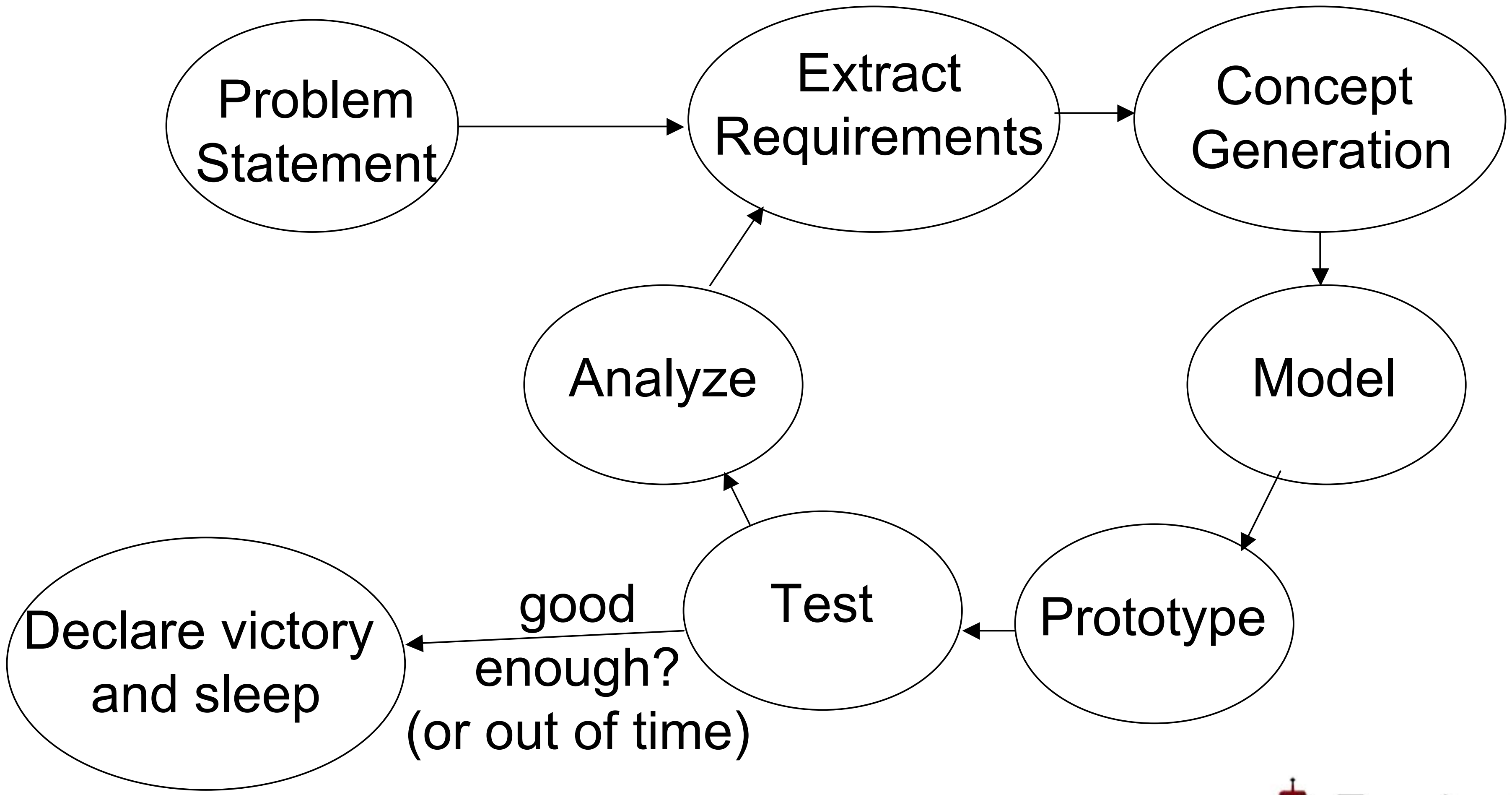
- Do a first draft of the full system
  - Mechanics
  - Software
  - Electronics
- Don’t worry about “polish” but include all major functions
- Plan on making major modifications from the lessons you learn from this experimental system

Create a Final System

- Based on tests of the experimental system
- Incorporates changes and improvements
- There may be several experimental versions before it reaches “final” state
- Allow enough time for at least two iterations
- It does not count as an iteration unless you have tested it thoroughly under realistic conditions


Conception
Evaluation
Implementation
Simulation
Solution Space
Time

Engineering Life Cycle


Problem Statement
Extract Requirements
Concept Generation
Model
Prototype
Test
Analyze
good enough? (or out of time)
Declare victory and sleep