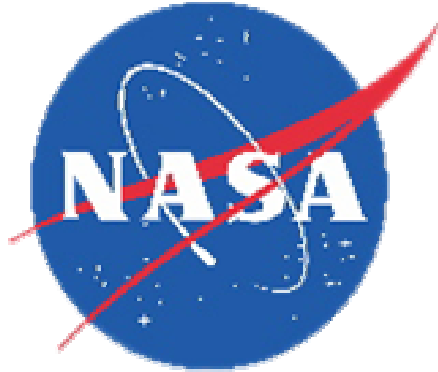


# Botball 2007 Educator's Workshop

v1.6 2007.2.20



# National Botball Sponsors



**NASA Robotics Alliance Project**  
National Diamond Circle Sponsor



**Hawaii Convention Center**  
National In-Kind Sponsor



**SolidWorks**  
National In-Kind Sponsor



# Regional Botball Sponsors



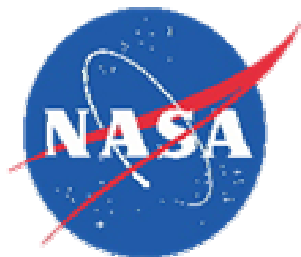
## **Naval Research Laboratory**

Regional Platinum Sponsor - Greater DC Region



## **Hawaii Department of Education**

Regional Platinum - Hawaii Region



## **NASA Goddard Space Flight Center**

Regional Gold Sponsor - Greater DC Region

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

- **University of Arkansas, Little Rock** - Arkansas Region
- **University of North Florida** - Florida Region
- **Mercer University** - Georgia Region
- **Southern Illinois University Edwardsville** - Greater St. Louis Region
- **University of Maryland, College Park** - Greater DC Region
- **Hawaii Convention Center** - Hawaii Region
- **Rose-Hulman Institute of Technology** - Midwest Region
- **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 2007

©1993-2007 KISS Institute for Practical Robotics

Written by:

David P. Miller

and

Charles Winton

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



# Tutorial Schedule:

- Day 1:
  - What is a robot?
  - What is Botball?
  - Design Projects
  - **C** programming language
  - **IC** Simulator
    - simple programming
    - variables and data
    - functions and libraries
    - **IC** Robot Simulator
    - motor functions
    - flow of control, while
    - sensors
    - differential steering
  - Botball electronics
  - Demobot construction
- Day 2:
  - The XBC v3
    - Hardware
    - Firmware Testing sensors & motors
  - Robot challenges
  - Botball functions
  - XBC Color Vision
  - Color Tracking
  - Robot challenges
  - Project management
- Day 2, 4pm:
  - Game Review
    - Robot Documentation
    - Building Rules
    - Game Challenge
  - 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



# Basic robot elements



# Human vs. Robot Subsystems

## People

Bones

Muscles

Senses

Digestion/Respiration

Brain

Knowledge

## Robots

Mechanical Structure

Effectors

Sensors

Power

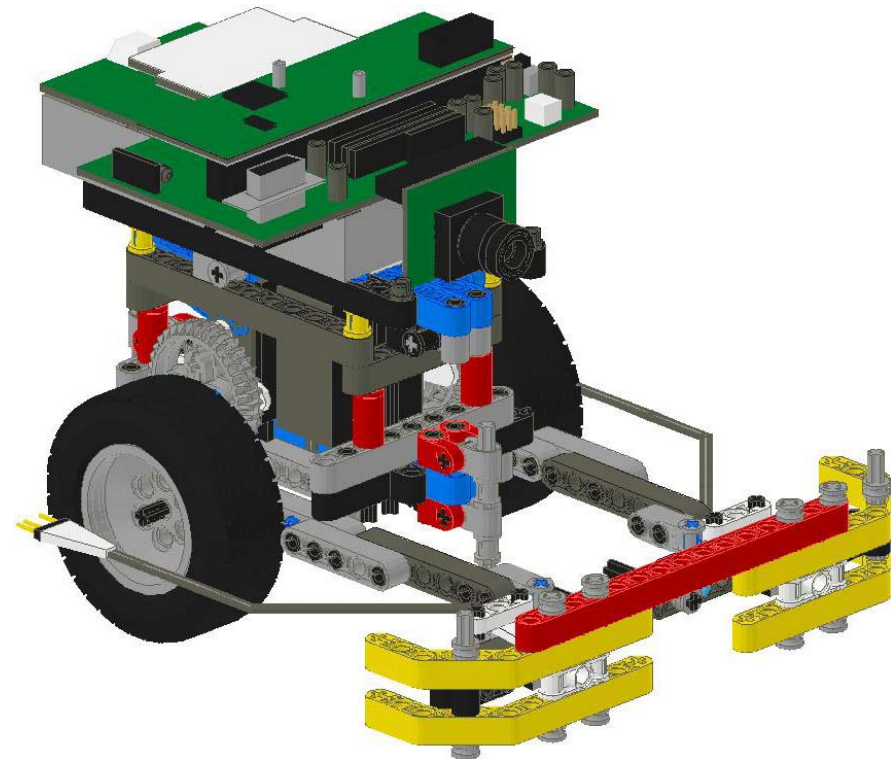
Computer

Program

# Structure

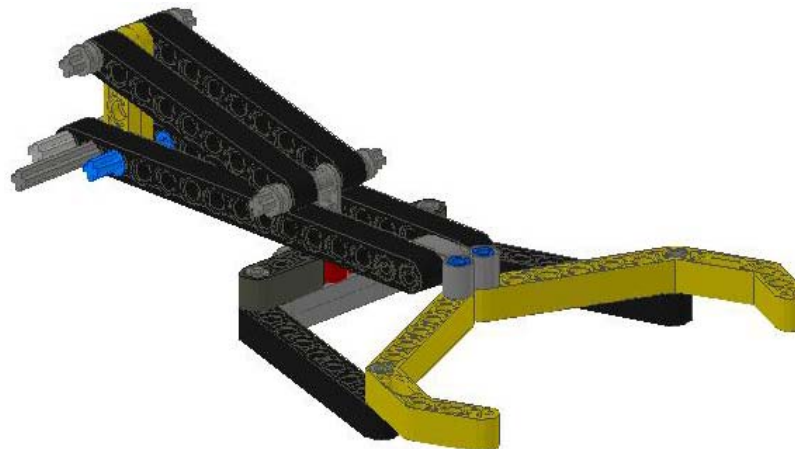
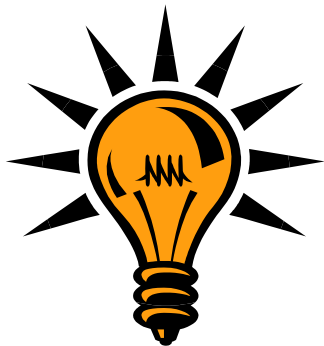
## Robot Structure

- 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





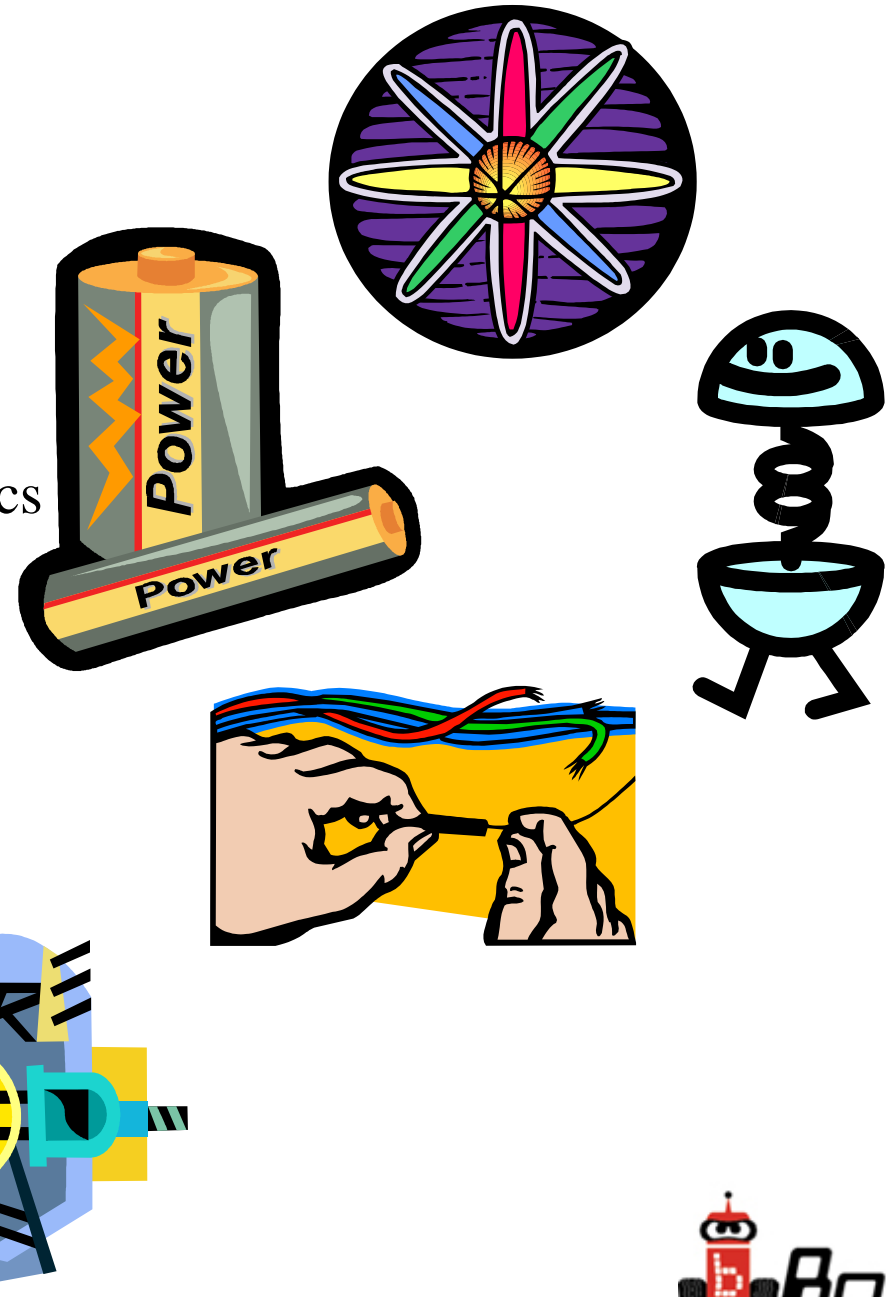
# Sensors

- 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,...



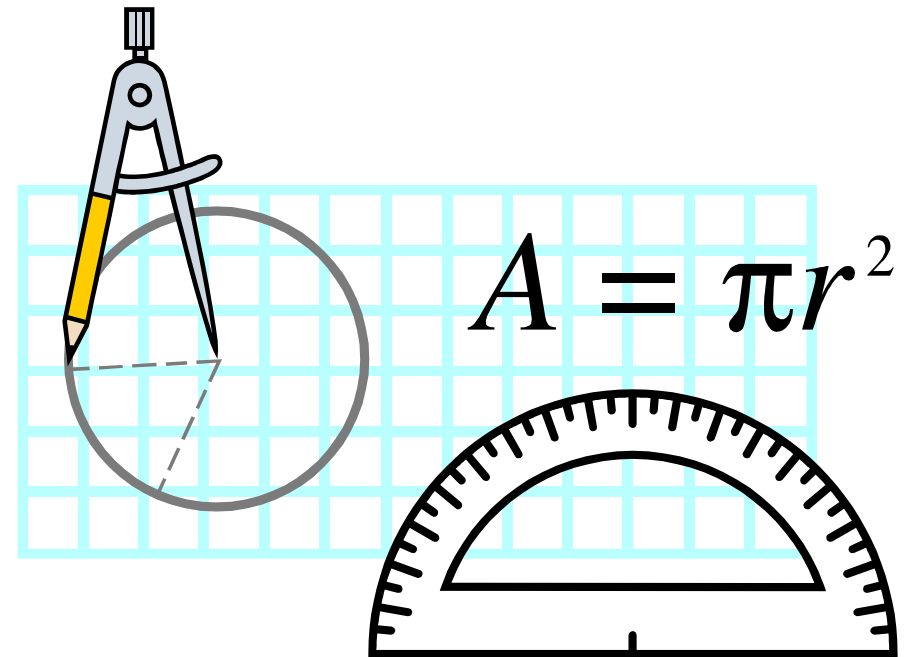
# Power

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



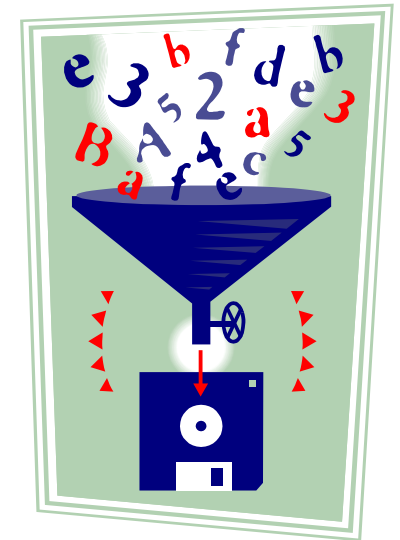
# Computation

- 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



# 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

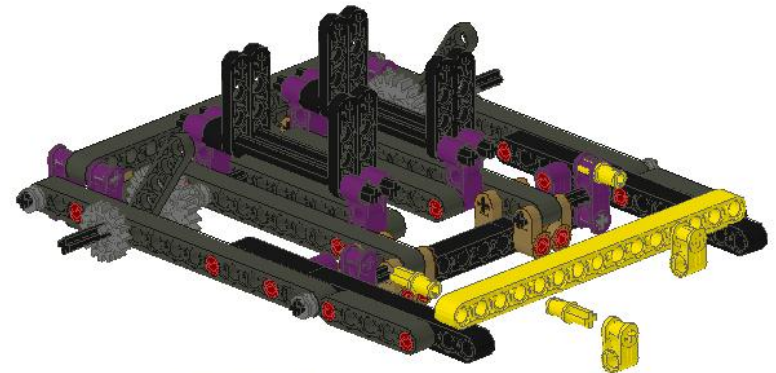


# Botball Robotics



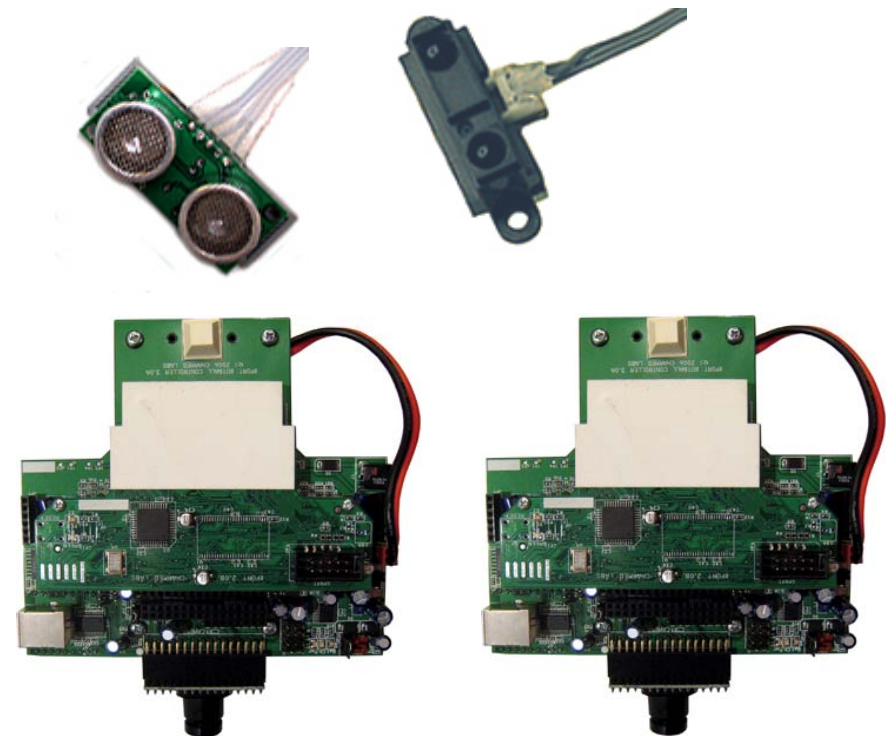
# Botball Robots Are Real Robots

- Structure:
- Power: rechargeable batteries
- Effectors:
  - Gear head motors
  - Direct Drive Motors
  - Servo motors

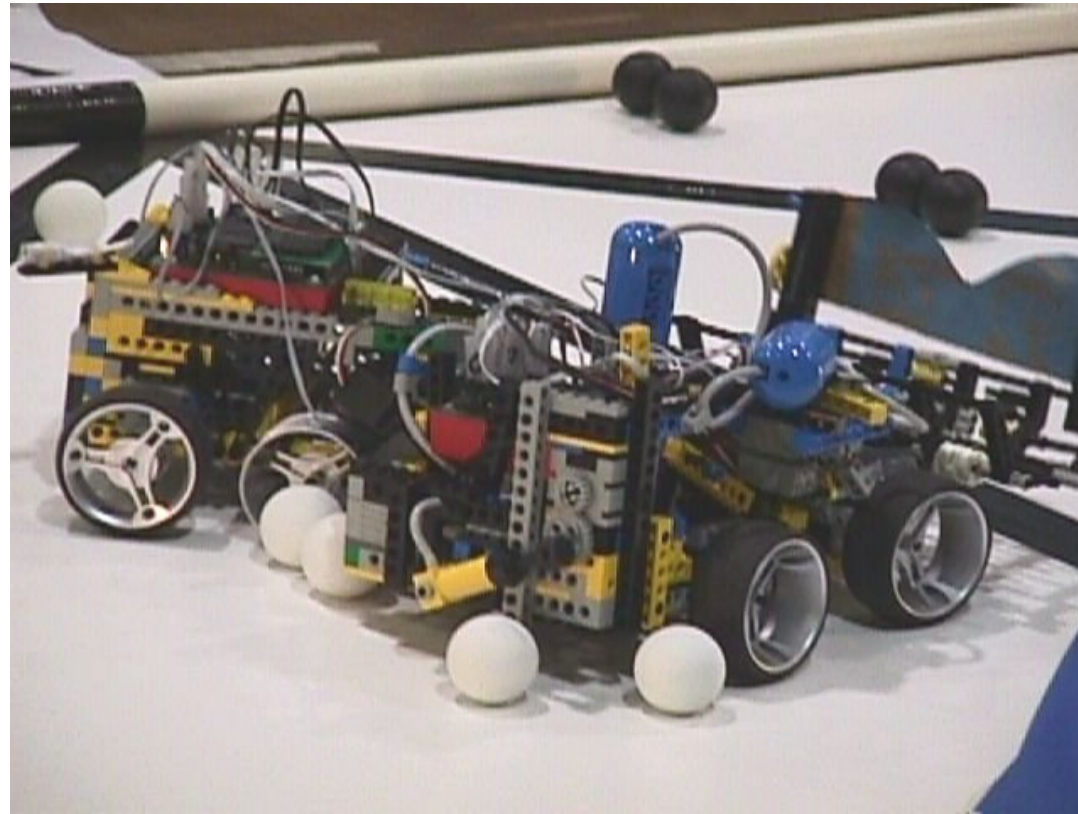


# Botball Robots are Real Robots

- Sensors: 28 sensors of 8 different types
- Computation: 2 microcontrollers
- Information: **C** programs written by the students



```
void repeat_beep(int numbeeps) //function is named repeat_beep
{
    int num=1;                /* declare & initialize counter */
    while (num <= numbeeps) // loop while num is <= numbeeps
    {
        beep();               /* beep once */
        num = num + 1; /* add one to the counter */
    }
}
```

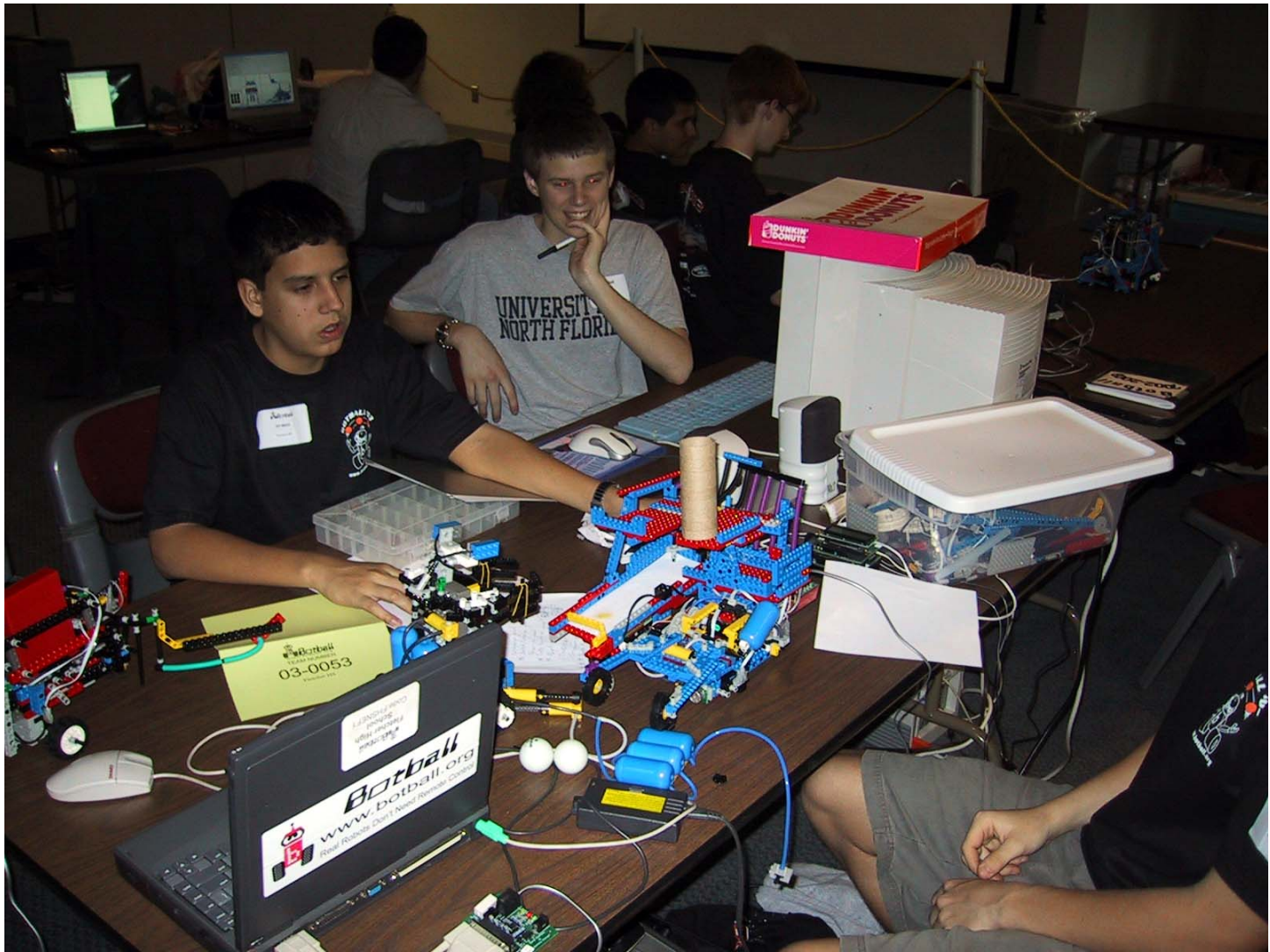


BOTBALL = roBOT + BALL



# Botball Robotics Program

for Middle and High Schools: [www.botball.org](http://www.botball.org)



# 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



# In the real-world, most R&D projects fail

## **Botball is designed to produce a successful project outcome**

- In the real world, projects lack physical resources:
  - Botball provides your team with just about all of the parts and software needed to complete the project.
- Projects do not meet mission requirements
  - Botball provides teams with very specific requirements documentation
- Projects meet a technical road block
  - 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.
- Project parts are not integrated in-time or at all
  - Through documentation requirements and scheduled updates, the Botball program helps keep your team on a schedule that will yield a functioning robot solution





- 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



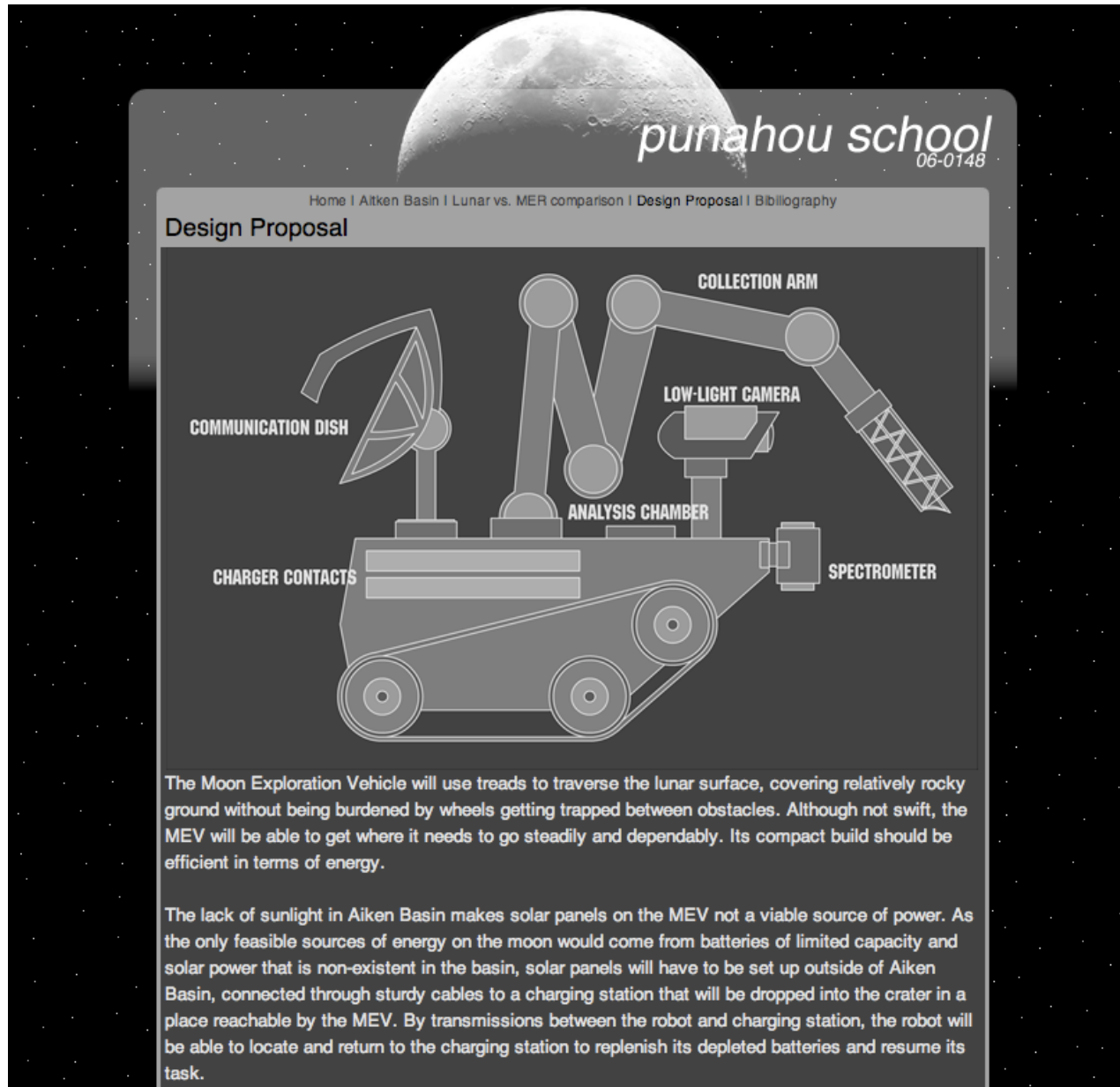
# Botball is a Student Activity...

- 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!



# Fall Research Design Project

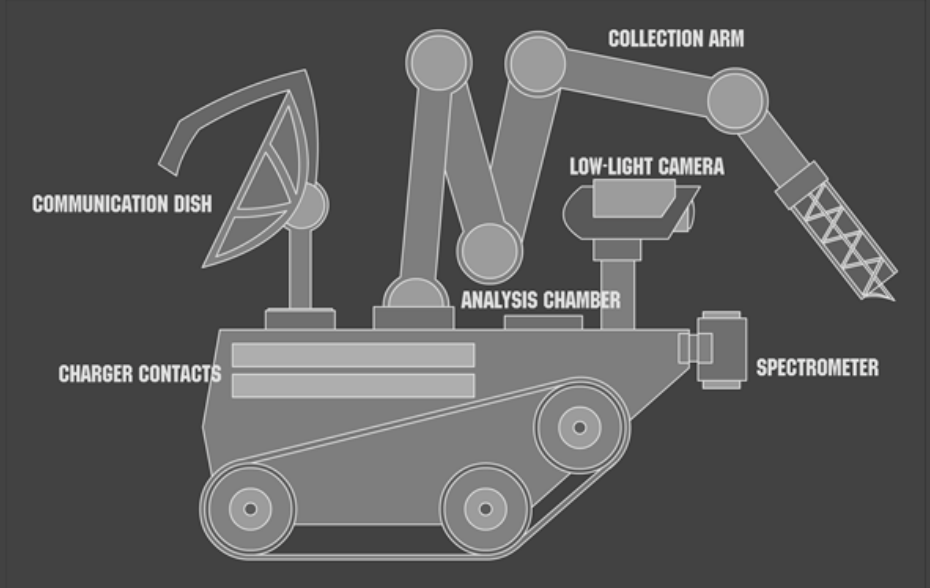
Sign-up by (1/26); Win Valuable Prizes



*punahou school*  
06-0148

Home | Aiken Basin | Lunar vs. MER comparison | Design Proposal | Bibliography

## Design Proposal

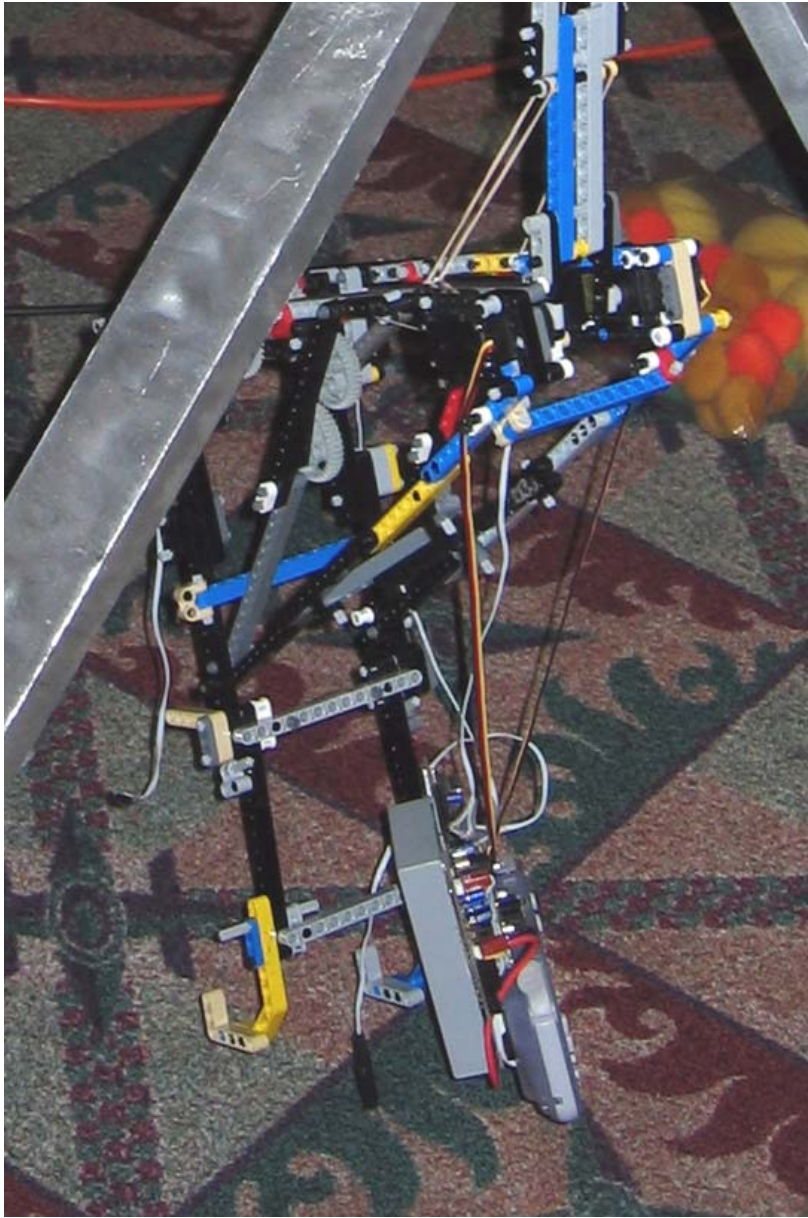
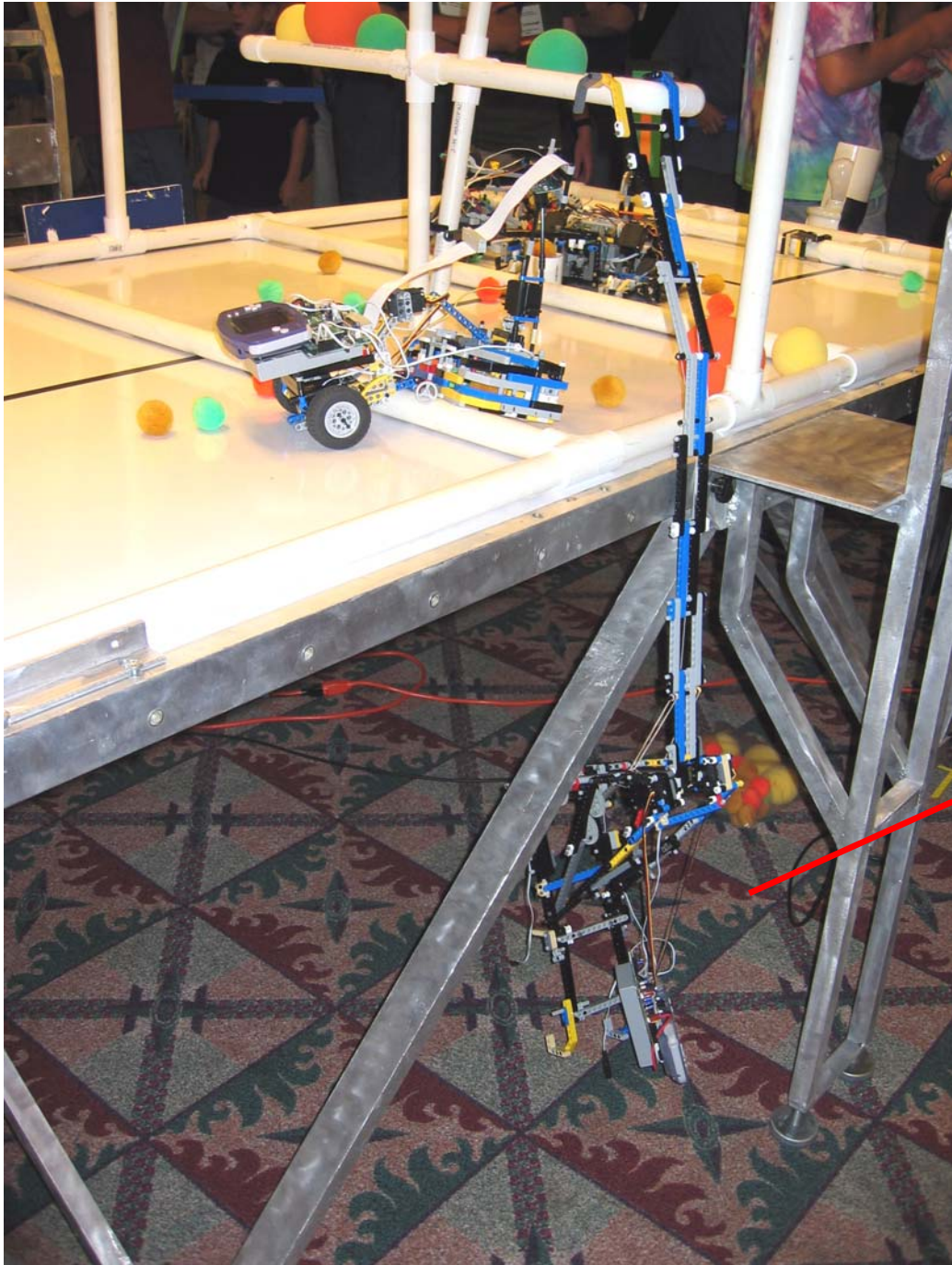


The Moon Exploration Vehicle will use treads to traverse the lunar surface, covering relatively rocky ground without being burdened by wheels getting trapped between obstacles. Although not swift, the MEV will be able to get where it needs to go steadily and dependably. Its compact build should be efficient in terms of energy.

The lack of sunlight in Aiken Basin makes solar panels on the MEV not a viable source of power. As the only feasible sources of energy on the moon would come from batteries of limited capacity and solar power that is non-existent in the basin, solar panels will have to be set up outside of Aiken Basin, connected through sturdy cables to a charging station that will be dropped into the crater in a place reachable by the MEV. By transmissions between the robot and charging station, the robot will be able to locate and return to the charging station to replenish its depleted batteries and resume its task.



# NCER 2006: Final



NCER 2007  
Honolulu: July 9-13

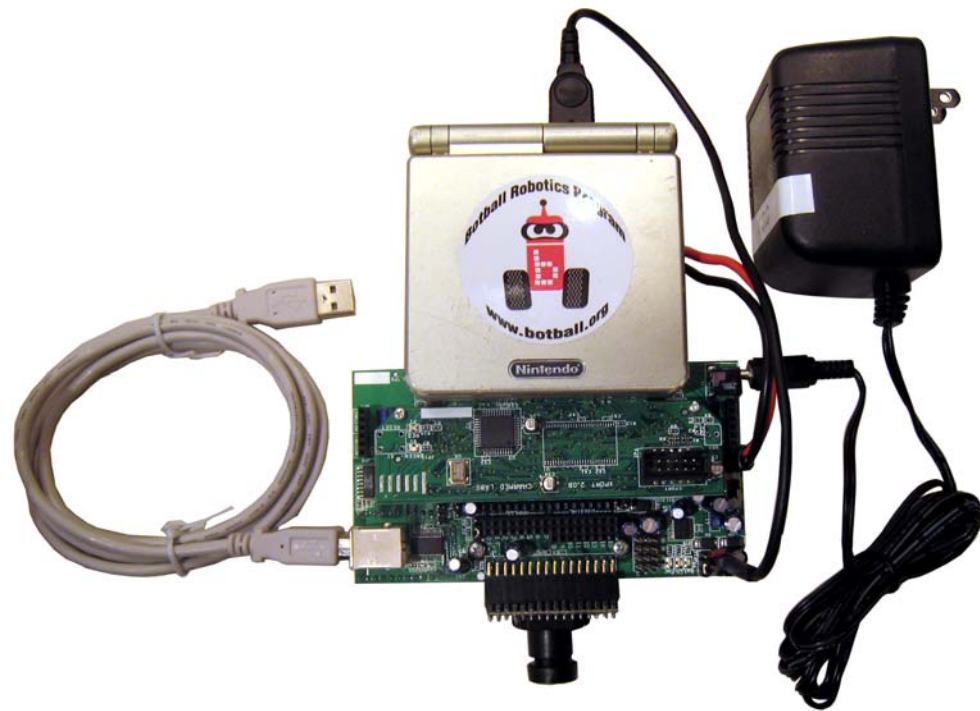




# Botball Today – Botball Tomorrow?

- Xport/XBC

- 16.78 MHz ARM7TDMI 32bit RISC (GBA)
- 2.9" TFT color LCD (GBA)
- Serial port via USB
- Integrated digital camera



- Qwerk

- 200 MHz ARM9 RISC processor
- Linux 2.6
- WiFi
- WebCam video input
- USB 2.0
- 10/100BT Ethernet
- Serial ports



- iRobot Create



# IC: the C Programming Language used in Botball



# IC Software Package

- The **IC** 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 **IC** in a commercial product, contact the KISS Institute about licensing
- The latest version may be found at <http://www.botball.org/ic/>
- **IC** (Interactive **C**) is a **C** compiler/interpreter
  - Implements most of the ANSI **C** language
  - Interfaces to both the Handy Board and XBC (and others)
  - Interactively guides hardware setup requirements, loading firmware (“pcode” interpreter) and key library functions onto the processor board
  - Provides an editor and on-line documentation
  - Provides an interactive environment for testing and debugging



# Setting Up

- **IC** runs partly on the PC and partly on the robot board
- The **IC** editor can be used without an attached robot board
- Code can be checked for syntax errors
- To check for logic errors you:
  - Can simulate execution of your program using built in simulator, or
  - Attach a robot and run your program
- The XBC needs to have firmware loaded before programs can be loaded onto robot



# To Install IC

- On a Mac OSX (10.1 and higher)
  - Double click on InteractiveCxxx.tar.bz2 file
    - The IC folder can be placed in your Applications folder, or anywhere else convenient
    - Note: keep the app and the library folders in the same IC folder (programs you write can be kept wherever you wish)
- On Windows (Win 2000 and XP)
  - Double click on InteractiveCxxx.exe
    - IC will be added to your program menu
    - An IC shortcut will be placed on your desktop
- On Linux
  - If you are using LINUX -- you should know what to do!

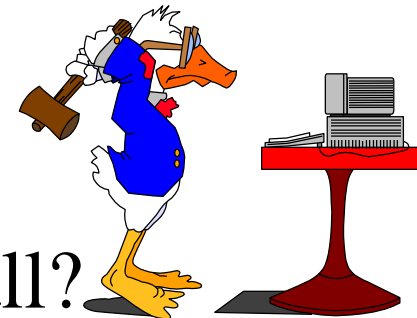


# C Programming

# The C Language



- What is C?
- Where did C come from?
- What is C used for?
- How hard is it to learn C?
- What does C have to do with Botball?
- So, how do you write C programs for robotics or anything else?



# What is C?

**C** is a *high-level block structured language*

- **C** programs appear to be independent of the specific hardware on which they are executed
- **C** programs have separate modules which may be executed independently
- **C** code is organized into blocks, each of which contains its own code and variables
- Unlike a human language, **C** is strictly governed by the rules of a formal grammar





# C: In the Beginning

- **C** is the brainchild of a Bell Laboratories scientist named Dennis Ritchie (circa 1970)
- **C** was preceded by a programming language named B
  - And there was one named BCPL which preceded B
- Bell Labs scientists used **C** to develop the original version of the Unix computer operating system
  - Over 90% of the programming was done in **C**
- Unix continues to be alive and well today
  - Linux is a popular “freeware” version of Unix
  - Mac OS/X is also a free version of Unix (with a moderately priced interface on top)



# The C Language

- C is one of the most widely used computer languages in the world
- C is the basis for C++ and Java
  - So learning C gets you halfway through learning those languages as well
- C was used to write UNIX & LINUX
- C is the primary language for writing applications for MacOS, LINUX and Windows



# Using a C Program

- C programs are just plain text files
- A C program is a collection of one or more C functions in one or more files
- Exactly one C function in each program is named **main**
- The program files are run through a compiler or interpreter
  - These are programs themselves which turn C code into machine executable code (not human readable - at least by most humans)



# C and Botball

- Botball programming is done in a subset of ANSI **C** called **IC**
- **IC** stands for “Interactive **C**”
- **IC** adds support for multiprocessing (a task usually controlled by an operating system)
- The **IC** programming environment includes a software simulator for each controller
  - You don’t have to have a controller to test your programs!

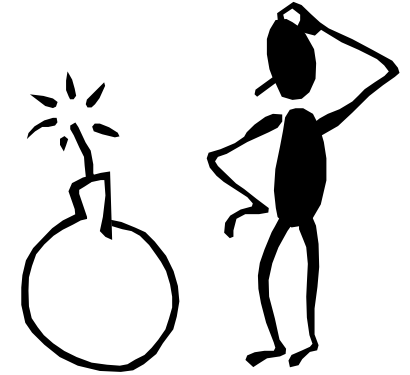


# C is a Small Language

- There are only a few built-in structures and operations
- Most of the utility is provided by libraries of functions
  - These are just C functions already written and debugged by someone else
  - IC automatically loads the key libraries you usually need, but more are available



# So What's the Catch?



- From concept to a working program requires both design and extensive iterative refinement
- Rigid rules of syntax and semantics for **C** produce “sins of omission and commission”
- Programming discipline, like writing discipline, has a learning curve



# A Simple C Program

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



# Function Names

```
void main( )
```

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

*Functions that don't return anything start with 'void'*

*Every **C** program should have exactly one function named 'main'*





# Blocks of Code

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

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

# Function Calls

```
void 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.*



# Terminating Statements

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

All **C** statements end with a ‘;’  
A statement is either a function call  
(like this *printf()*) or an assignment  
statement.



# Commenting Programs

- Explanatory comments
  - Can (and should) be added to a program to assist in understanding it later
- 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
- Example:

```
/* simple.ic -(c) 2003 David Miller, KIPR */
/* This program displays a simple character string
   -- It is a good idea to start each function with a
   comment that explains what the function does --
[this comment and the previous one are in multi-line form] */
void main()
{
    printf("This is a C program\n"); // display the string
} // end of main [this comment and the previous one are in-line]
/* Hmm ... the last in-line comment is pretty superfluous;
   it's a waste of valuable eye time! */
```



# Indentation...

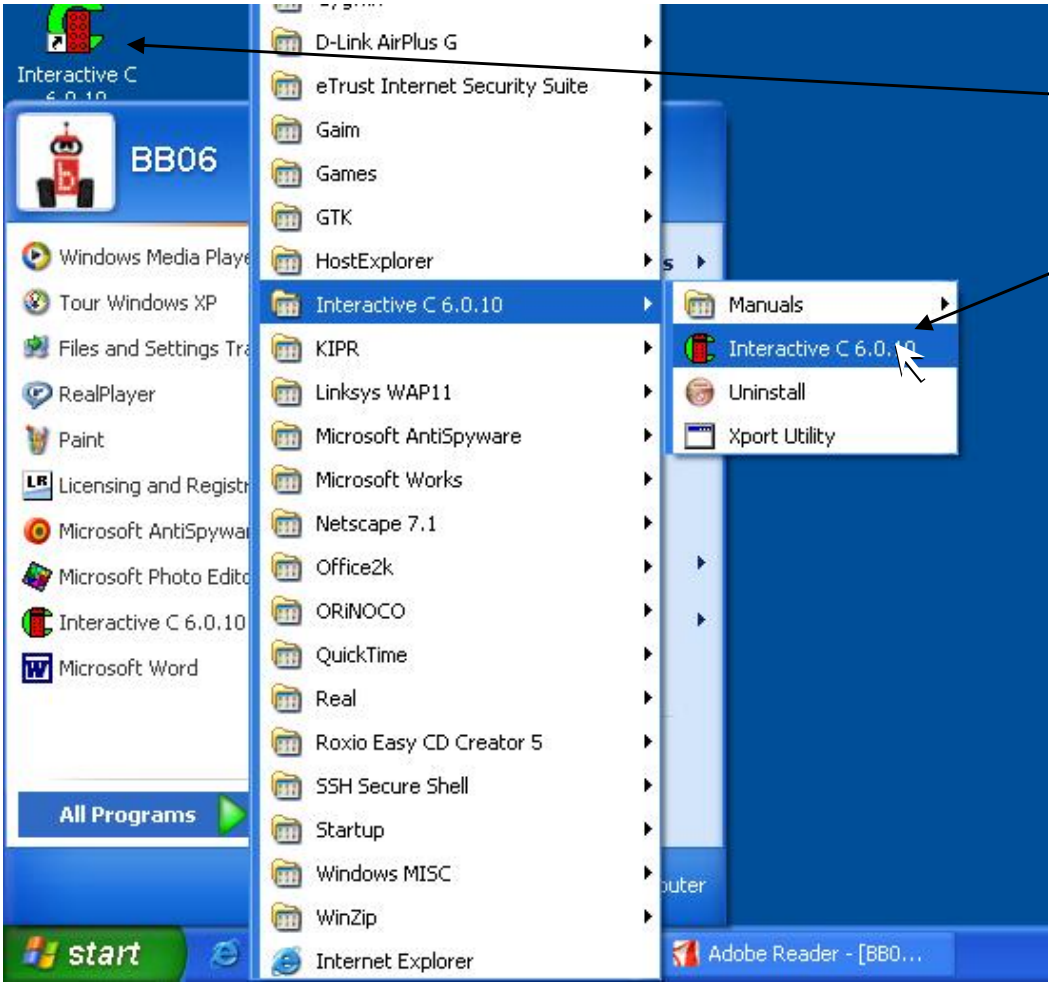
- **C** ignores most white space (spaces, returns, tabs, blank lines)
- Indenting **C** program text helps to bring out the structure of your program
  - Indent code after each '{'
  - Shift back to left after each '}'
  - Indent the second line of a single statement (exception: any added white space inside a quoted string will print!)
- **IC**'s built in editor does most indentation for you!
  - Programs are automatically indented when loaded
  - The **IC** *edit* menu provides commands for indenting your text without saving and reloading



# IC Environment and Simulator



# Start Up IC

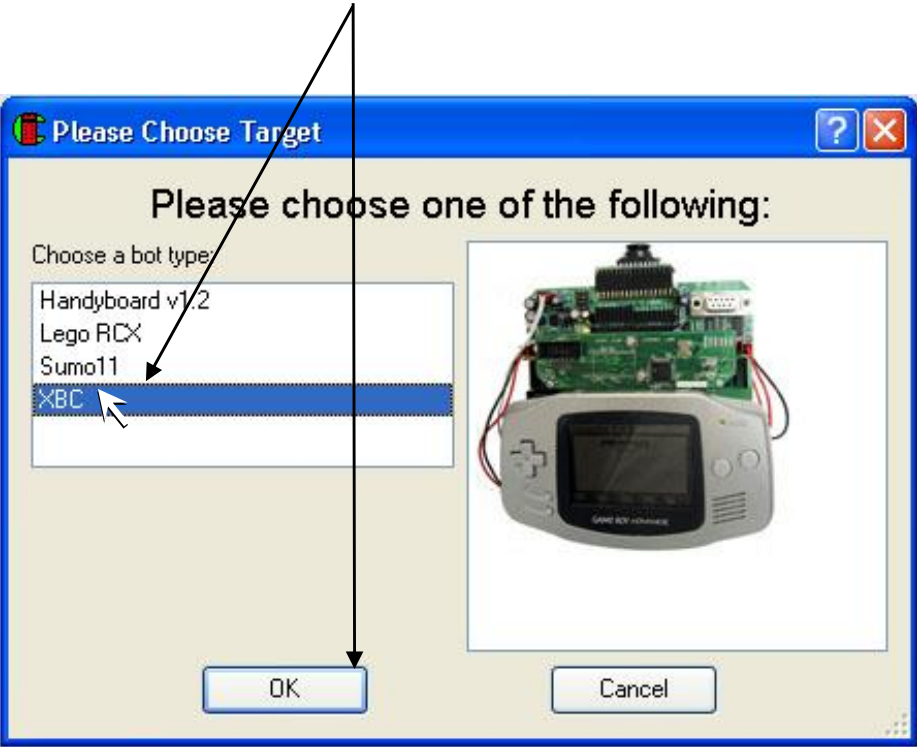


*Launch IC*

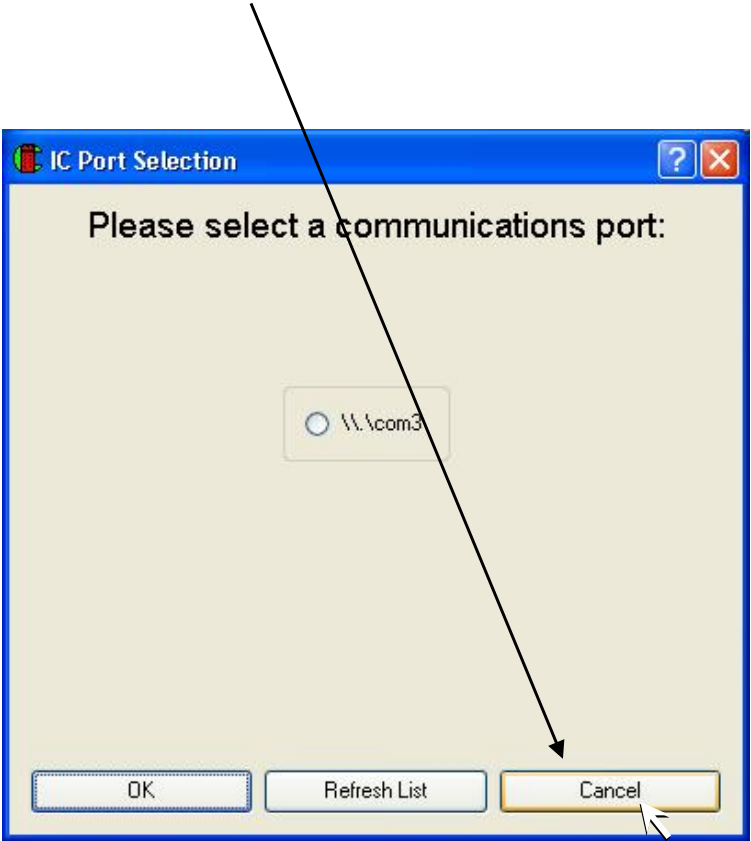


# Set **IC** to use the XBC Controller

STEP 1 – Choose XBC then OK:



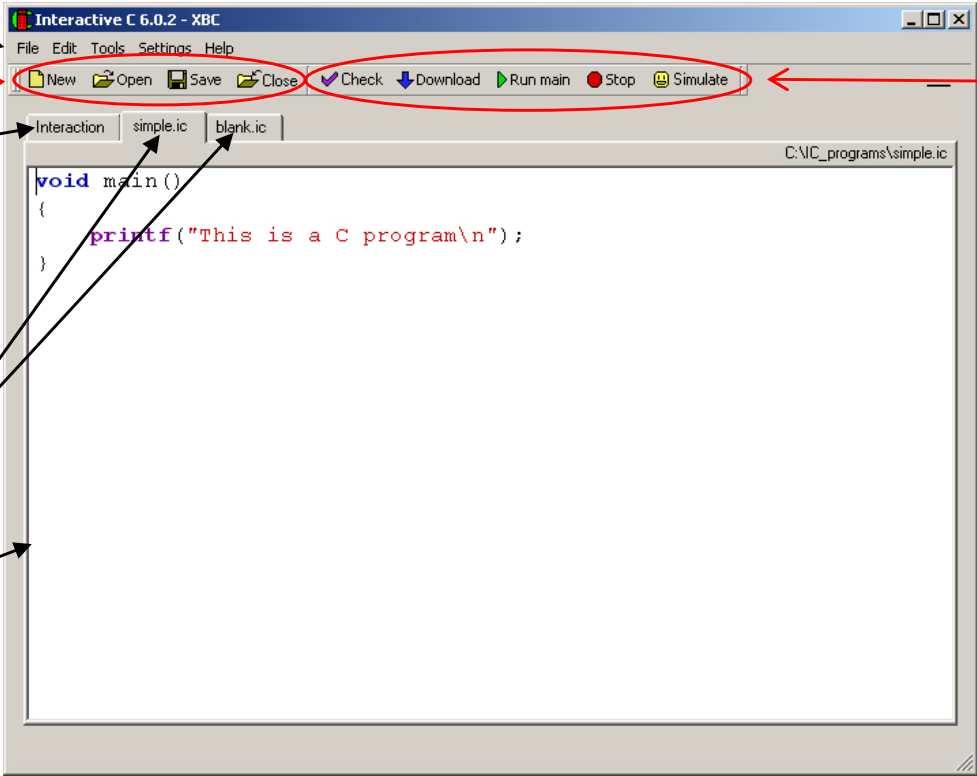
STEP 2 – **Cancel** Communications for now:





# IC Environment

- Drop down menu bar
- File shortcuts
- Tab for interaction window
  - Only works if a controller is attached
  - Simulator has its own interaction window
- File tabs
  - One per open file
- Program entry/edit window for active file tab



## Program services

- Compilation *check* of active file
- Communication with attached controller
  - Download
  - Run main
  - Stop
- Launch *simulator*
  - Has to be from a file tab with a valid **IC** file (which may be blank)



# IC Drop Down Menus

The screenshot shows the 'Interactive C 6.0.2 - XBC' application window. The menu bar includes File, Edit, Tools, Settings, and Help. The toolbar contains icons for New, Open, Save, Close, Check, Download, Run main, Stop, and Simulate. Below the toolbar are tabs for 'Interaction', 'simple.ic', and 'blank.ic'. The main text area shows a C program with `void main()` and `program(" ")`. Several drop-down menus are open, with red arrows pointing from the corresponding menu items in the toolbar to the open menus:

- File menu:** New (Ctrl+N), Open... (Ctrl+O), Close, Save (Ctrl+S), Save As..., Print... (Ctrl+P), Exit.
- Edit menu:** Undo (Ctrl+Z), Redo (Ctrl+Y), Cut (Ctrl+X), Copy (Ctrl+C), Paste (Ctrl+V), Select All, Find... (Ctrl+F), Find Next, Goto Line, Indent, Indent All.
- Tools menu:** Check Pairs, Upload Array, List functions, List global variables, List loaded files, List #defines, View diagram, Download firmware, Download libraries, Sim Lib Check.
- Settings menu:** Increase Font Size, Decrease Font Size, Change controller type, Change serial port.
- Help menu:** Manual..., About.

Blue arrows provide additional context:

- A blue arrow points from the text *Matched to controller selected* to the 'Manual...' option in the Help menu.
- A blue arrow points from the text *Sets simulator configuration* to the 'Change serial port' option in the Settings menu.
- A blue arrow points from the text *Not needed when working with the simulator* to a bracket grouping 'Download firmware' and 'Download libraries' in the Tools menu.

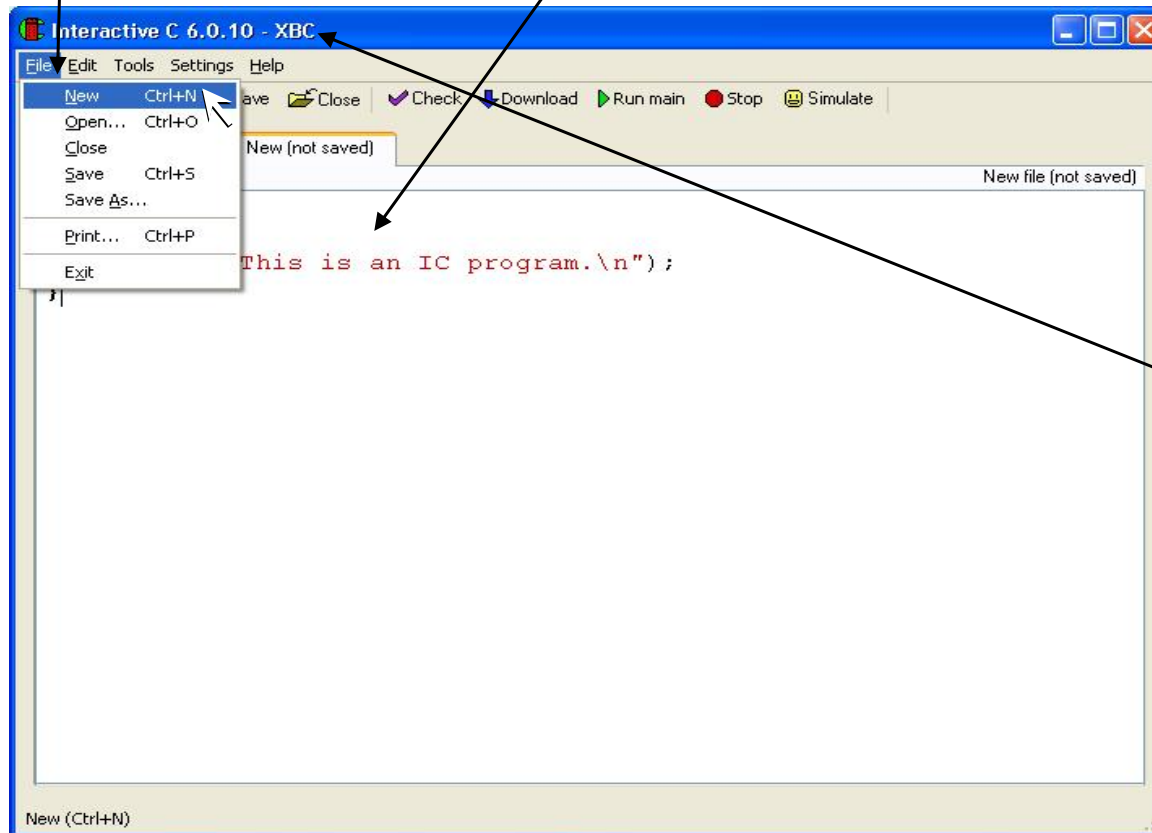
# Create, Edit and Save a New File

1) Create a New file:

2) Type your program into the file:

3) Save the file:

**NOTE 1:** remember  
where you saved your file

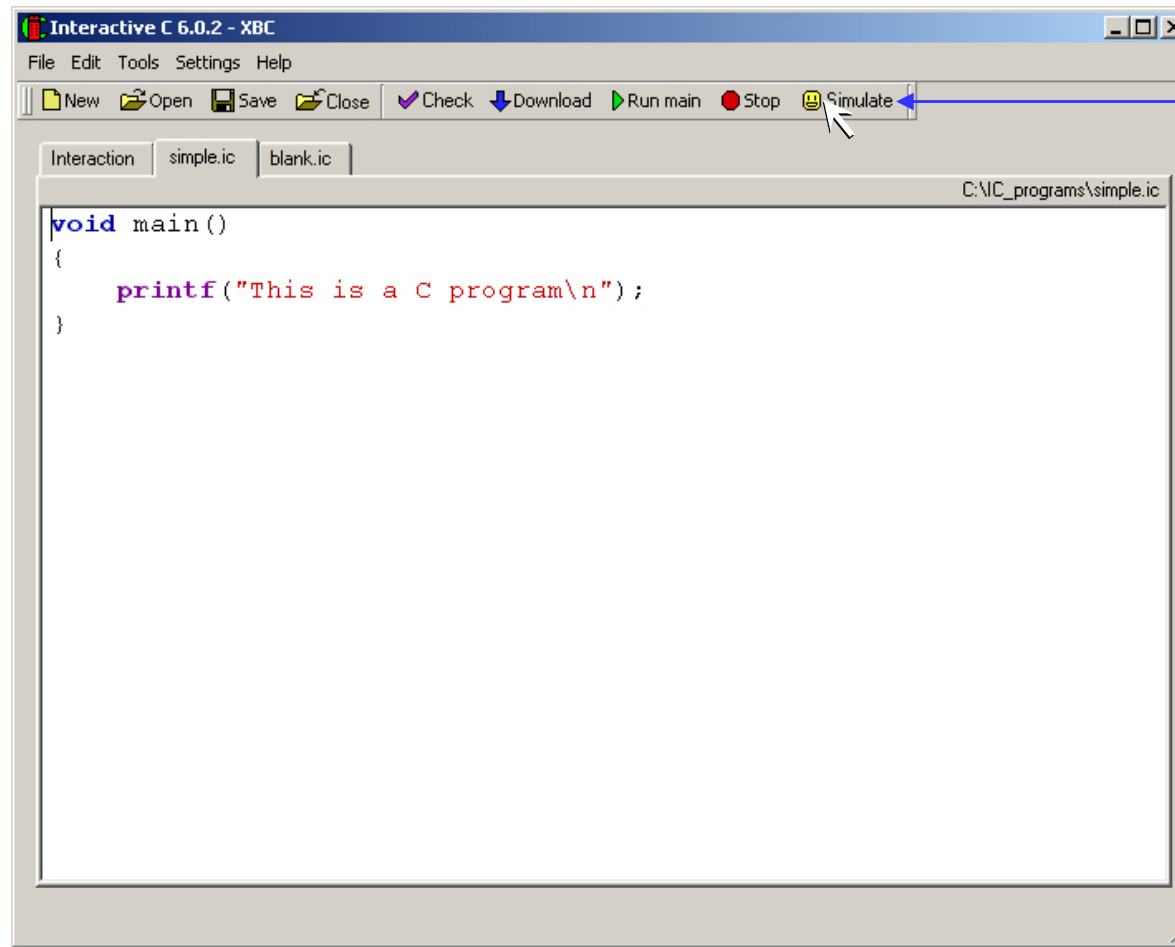


**NOTE 2:** you can VERIFY that the XBC is the “current active controller type” by viewing the window title which tells you:

- 1) The version of IC running
- 2) The controller type selected (if any)
- 3) The serial port selected (if any)

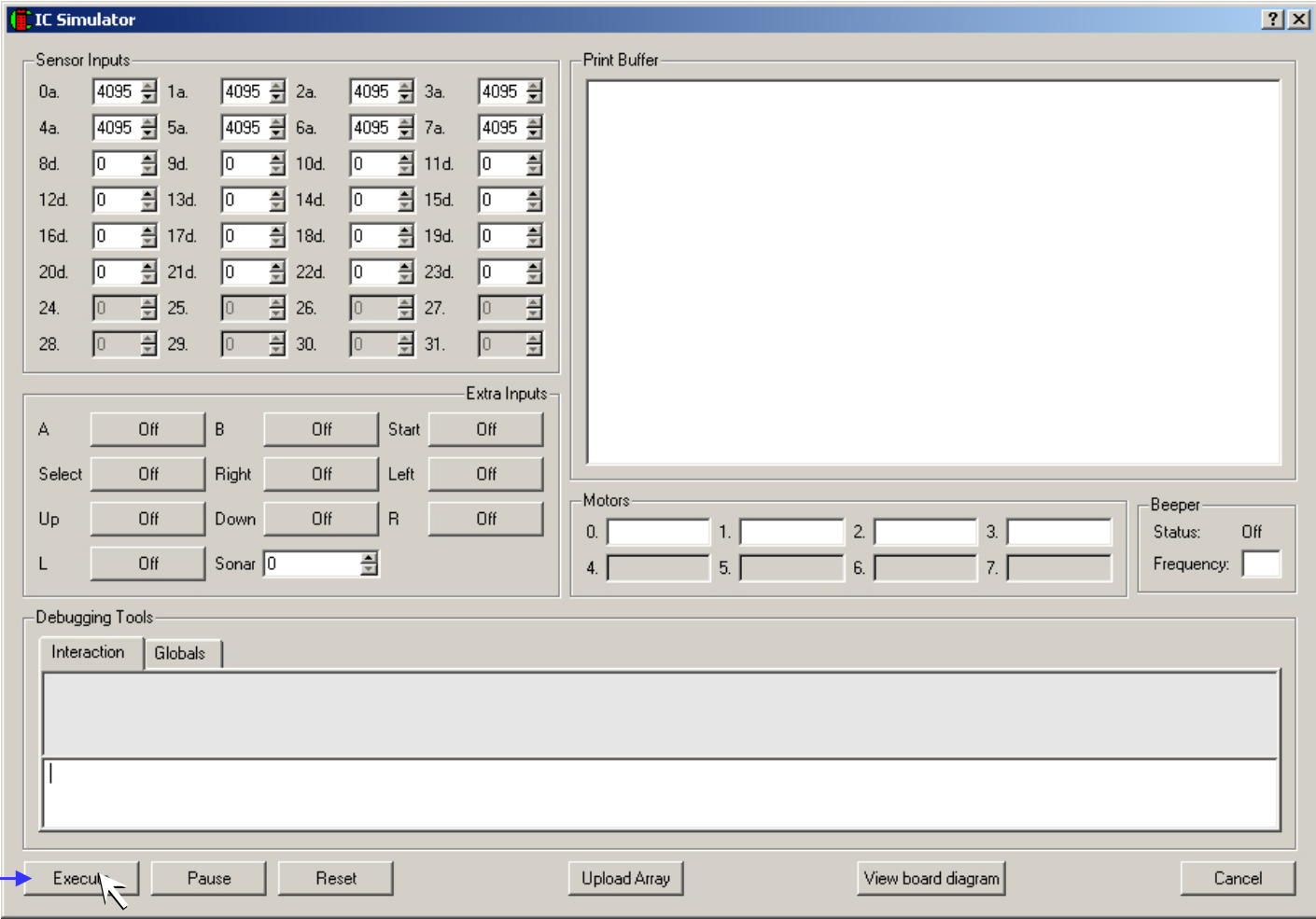


# Launching the Simulator

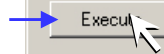


*Launch  
simulator*

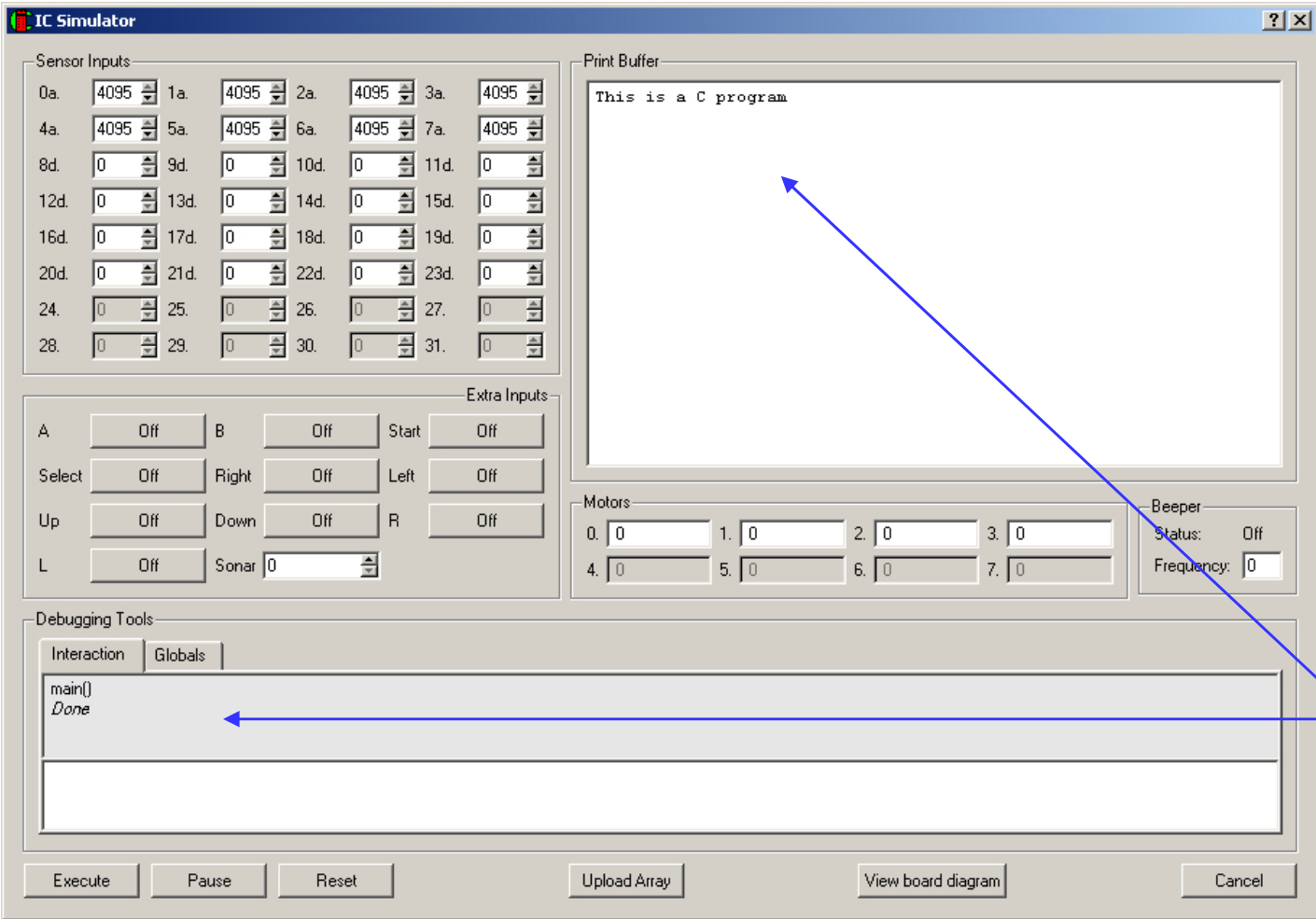
# Running your Program



*Run Program*



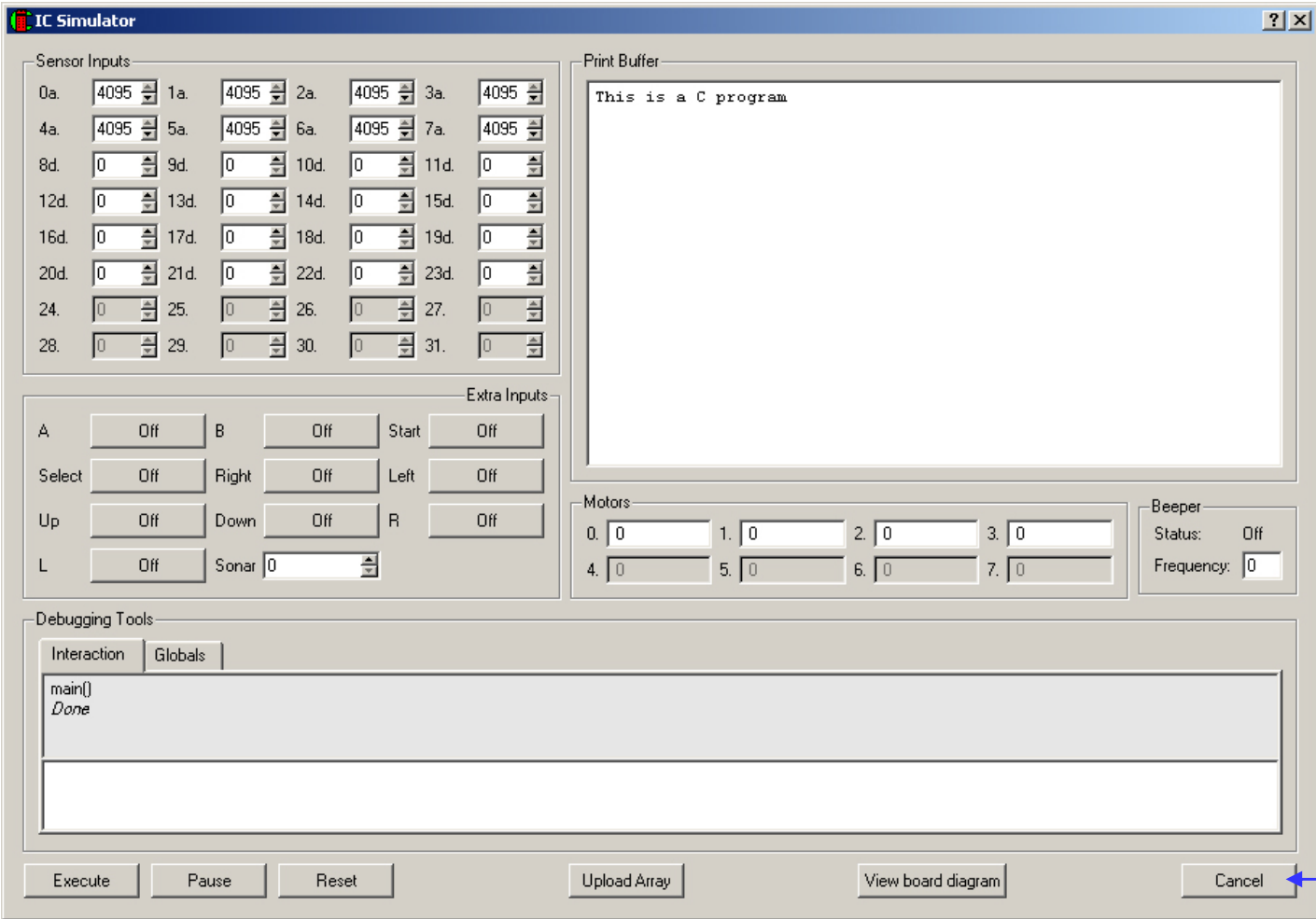
# Observing Results



Observe  
result



# Returning to IC



*You must cancel the Simulator before you can use IC again*



# How to Program

- Think about and explicitly state goal
- Collect Requirements:
  - Identify things that must be done (independent of solution method)
- Method of Execution:
  - Think of relevant examples or similar problems and their solutions
  - Choose your tools (in this case it is **IC**)
  - Iterate and refine until done:
    - create the program to address the requirements
    - test syntax
    - test functionality/logic
  - Celebrate!





# Exercise #1: Step by Step

- Goal: Create a program that prints your name on the screen.
- Start **IC**, choose the XBC, and click the "New" button on the IC window
- Type in a program that prints your name and save it
- Check the syntax of your program by clicking on the "Check" button
- If there is an error, **IC** stops checking and reports the line number where it encountered the problem
  - you can use the *Edit .. Goto Line* drop down menu to get to the problem
  - Note: the error occurs on OR before that line
- Load the program to the simulator by clicking on the **IC**'s "Simulate" button
- Run the program by pressing the "Execute" button on the simulator pane



# Variables and Data



# Numeric Data in C

- There are three types of numbers implemented in C
  - Integer
    - in IC, a 16 bit binary number in the range -32,768 to +32,767
  - Long integer
    - in IC, a 32 bit binary number in the range -2,147,483,648 to +2,147,483,647
  - Floating point numbers (fractions)
    - in IC, a 32 bit representation in “scientific notation” for numbers such as 3.141659
- Arithmetic can only be performed on items of the same type
  - There are built-in means for working with “mixed” expressions
- Integer arithmetic (integer or long integer) is handled by hardware circuits, so it is fast
- Floating point arithmetic is handled by software, so it is 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 i;**
    - specifies **i** to be an integer variable
    - 16 bit memory location
    - Range -32,768 to +32,767
  - **long j;**
    - specifies **j** to be a long integer variable
    - 32 bit memory location
    - Range -2,147,483,648 to +2,147,483,647
  - **float k;**
    - specifies **k** 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 **IC** documentation)



# Integer Arithmetic (**int** & **long**)

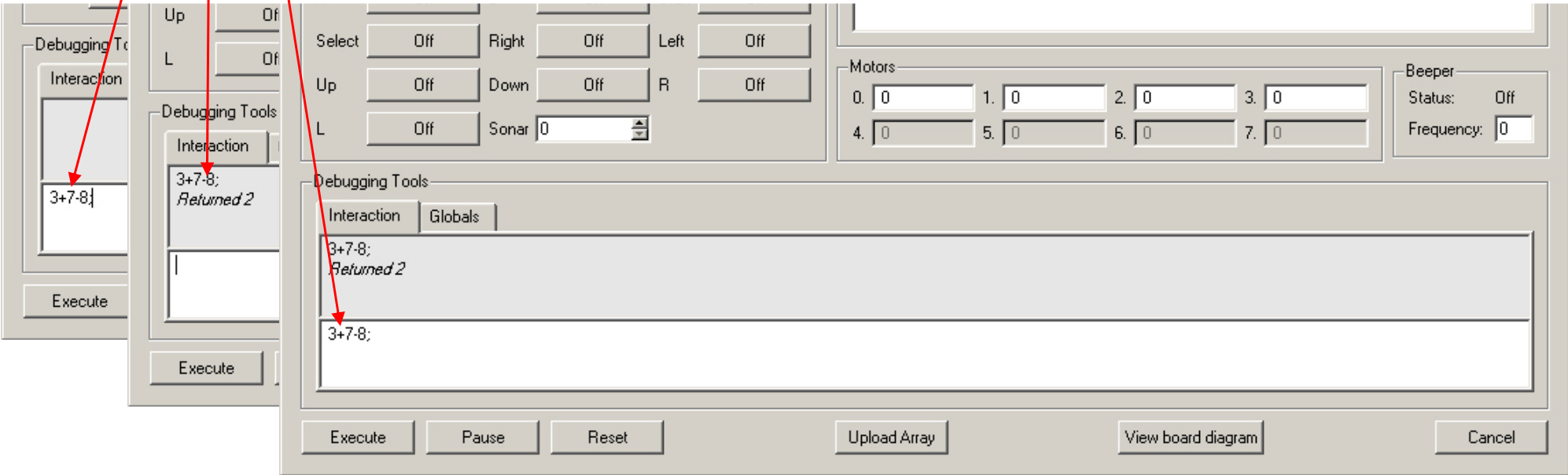
- **+** is used for addition  
 $X + Y$  means X plus Y
- **-** is used for subtraction  
 $X - Y$  means X minus Y
- **\*** is used for multiplication  
 $X * Y$  means X times Y
- **/** is used for division  
 $X / Y$  means X divided by Y with the decimal truncated
- **%** is used for modulus (when 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)



# Arithmetic Using the Simulator

## Interaction Window

- Enter an arithmetic expression
  - **3 + 7 - 8;**
  - and press Enter
  - Observe result
  - Use cursor “up arrow” to scroll interaction window up to most recent entry



# 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)
    - the work is now being done in software instead of at the hardware level
  - With floating point arguments the **/** operation retains the decimal fraction rather than truncating it
- **%** is not defined for floating point numbers



# Getting Data into Variables

- 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  $\pi$ */
```

- 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 new value in i
```

- What is the value of `i` now?





# “Mixed” Expressions

- Remember that for **IC** **int**, **long**, and **float** are different types of data
  - Humans expect  $5.1 + 3$  to add to 8.1
  - Try it using the simulator
    - **IC** produces an error because from its point of view 5.1 and 3 are two different kinds of data and the expression mixes them
- You could just enter  $5.1 + 3.0$  to correct the problem, but your numbers could be in variables in which case adding a decimal point would not be possible
  - Using the simulator try  
`5.1 + (float)3;`
  - This is call a “cast” since you are casting the **int** 3 to its closest **float** equivalent (which is 3.0)
    - Casting is a built-in operation in **IC** used when you need to operate on data that is of different types



# Classification of Variables

- Recap: each **C** variable used in a program must be specified (or declared)

```
int i;           //integer variable
int j=0;         //integer variable, initialized to 0
float pi=3.1416; /*floating point variable, initial
                  value approximately  $\pi$ */
```

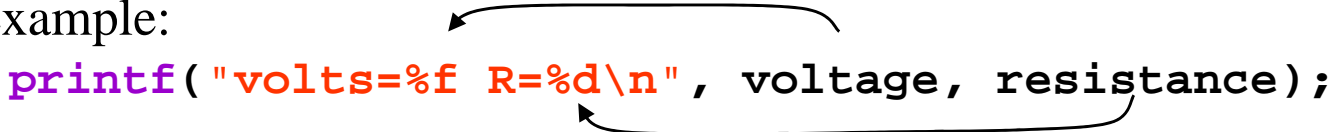
- This will create variables **i**, **j**, and **pi**
- There are two major classifications for variables
  - *Global* variables
  - *Local* variables
- A global variable is one whose specification is outside of all functions, including **main( )**
- A local variable is one whose specification is inside a block or function
  - Local variables for a block must be specified at the beginning of the block (right after the “{“)
  - A local variable is valid only within the block in which it is declared (the *scope* of the variable)
    - A local variable name can be the same as a global variable name, in which case only the local meaning is recognized
    - The basic “scope rule” is that if a variable name is used in a block, the most local version is the one recognized



# Displaying Variable Values using `printf ( ) ;`

- The values of variables used within a program can be displayed using the **IC** library function `printf`.
- `printf` is a version of the `printf` function in the standard **C** library
  - Limited by the abbreviated display capabilities of the robotics controller
  - Only prints to the robot controller's display
  - For the simulator, the output is to the simulator's **Print Buffer** window
- 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` and `%l` correspond to **int** and **long** arguments
  - `%f` corresponds to a **float** argument
  - `\n` positions you at the start of the next line in the **Print Buffer** window
  - The on-line documentation has a more complete description of `printf ( )`



# Try it on the Simulator

- In the simulator interaction window try a simple expression such as **2+3\*5;**
  - Is the result  $(2+3)*5$  or  $2+(3*5)$ ?
    - **C** semantics determines the implied order of operation if the expression is not fully parenthesized
- Next try printing the value of the arithmetic expression
  - **printf("%d\n", 2+3\*5);**
- Now try
  - **printf("%d\n", 114/5);**
    - For **C** semantics, the result of an integer division is truncated (as opposed to rounded)
- Now try
  - **printf("%f\n", 114.0/5.0);**
- Now try something more complicated, such as
  - **{int i=7,j; j=i\*9; i=i\*j/2; printf("%d\n",i);}**
    - Did you get  $(i*j)/2$  or  $i*(j/2)$ ?
  - Or try something of your own choosing!



# 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  $\pi$  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

- 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
  - You've already written one of these, **main**!
- 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 **IC** Functions

- The general function syntax in **C** is

```
<type> fn-name ( <type> arg1, <type> arg2, ... )
    { <function-body> }
```
- **main** is a function with no arguments, and returns nothing

```
void main() {
    printf ( "a very simple C function" );
}
```
- 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 = 3.141593; // approx value of pi
    return(pi * r * r);
}
```
- Copy the **circ\_area** function into **IC**, save as “**circle\_fns.ic**”, and launch the simulator
- Test your function
  - Enter **circ\_area(50.0)** in the simulator’s interaction window
  - Note the effect approximating  $\pi$  has on the result! (compare to the better approximation of 7853.981634)





# Libraries

- A library is an **IC** file that contains a set of functions but no **main** function
- A library serves as an organizational tool
- **IC** automatically loads some libraries to give you basic functions like **printf** and robot motor commands
- You can *use* the functions in additional libraries by telling **IC** to *use* them



# Using **IC** Functions, **#use**

- To use your area function, you call it the same way you would call any of the **IC** library functions
- To provide **IC** with the definition of your area function, you could just copy its definition in with your **main** function; eg.,

```
void main() {
    float r=50.0;
    printf("circle of radius %f\n",r);
    printf("area: %f",circ_area(r));
}
float circ_area(float r) {
    float pi = 3.141593; // approx value of pi
    return(pi * r * r);
}
```

- The **IC** “preprocessor directive” **#use** provides the means for giving **IC** the definitions of functions you are maintaining in a separate file

```
#use "circle_fns.ic"
void main() {
    float r=50.0;
    printf("circle of radius %f\n",r);
    printf("area: %f\n",circ_area(r));
}
```

- The **IC** library is just a file of **IC** functions (“**libxbc.ic**” for the XBC) that is automatically used with the controller selected



# Exercise

- Add a function named **circ\_circum** to your **circle\_fns.ic** “library” file. Modify the main function on the previous slide to print the circumference as well as the area.
  - Remember that the circumference function is  $C(r) = 2\pi r$
  - Note that **#use** requires one space between the word **use** and the file name.
  - Hint: copy/paste the **circ\_area** function, change the name of the pasted copy to **circ\_circum** and fix the formula used in the **return** statement to compute the circumference



# IC Robot Simulator: *iROBOSim*

# But What About Robots?

- Robots are just computational devices that interact with the world by side effect.
- Robots can operate in the physical or virtual world.
- Botball provides an **IC** library that simulates a simple robot, *iROBOsim*, and some simple worlds
- Programs written for *iROBOsim* can be run on the **IC** simulator, and with very minor changes, also be run on a physical Botball robot.



# Install *iROBOsim*

- Copy the contents of the *iROBOsim* folder (from the team CD) into the *InteractiveCxxx/lib/xbc* folder.
- When writing programs for *iROBOsim* always do a:

`#use "iROBOsim.ic"`

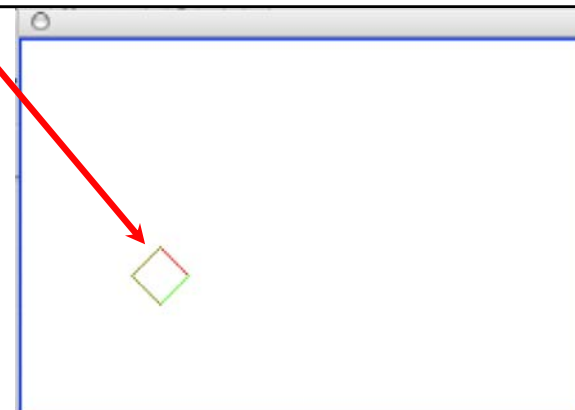
at the beginning of your program file



# Example *iROBOSim* Program

- This program runs the simulated robot for 10 seconds.
- Try typing it in! (you can also copy/paste from the e-copy of these slides)
- Save in a file named `move10.ic`
- The diamond shaped robot is red on the front left and green on the front right
- `rsim_mav` powers a motor
- `rsim_ao` turns all motors off
- We'll get to what this code means but first some background info

```
#use "iROBOSim.ic"
void main()
{
    //initialize simulator
    //use world keyword
    iROBOinit(EMPTYWORLD);
    //motor 0 is left motor
    rsim_mav(0,200);
    //motor 1 is right motor
    rsim_mav(1,200);
    //leave motors running 10 secs
    sleep(10.0);
    //turn a[ll motors] o[ff]
    rsim_ao();
}
```



# Motor Functions



# XBC Motor Functions

- XBC has four motor ports (0, 1, 2 & 3)
- The *iROBOsim* robot also has four ports. 0 is for the left motor, 1 is for the right motor, and 2 and 3 are TBD
- *iROBOsim* motor functions are the same as XBC motor functions except they have "**rsim\_**" prefixing the function name



# XBC Motor Functions

- The XBC 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 black gear motors included with the XBC electronics kit, one full rotation = about 1100 “ticks” -- a unit of measure for rotation
- The velocity measure is ticks per second (magnitude 0-1000)



# XBC Motor Functions

- Get the current tick count of motor 0:  
`get_motor_position_counter(0);`
- Move the motor at a speed of 0.5rps (550 ticks/sec) to position 30000:  
`mtp(0,550,30000L); //move to position`
- Move the motor backwards at a speed of 0.25rps (275 ticks/sec) for 1 revolution: `mrp(0,275,-1100L); //move relative position`
- Move a motor indefinitely at a given velocity (e.g., backwards at 1rpm (18 ticks/sec) (wow that is slow!) `mav(0,-18); //move at velocity`
- Turn motor 0 off: `off(0);`
- Turn all motors off: `ao();`
- *iROBOSim* has versions of these functions:  
`rsim_get_motor_position_counter`, `rsim_mtp`,  
`rsim_mrp`, `rsim_mav`, `rsim_off` and `rsim_ao`.
- See **XBC Motors** in the Appendix for more details



# Library Functions and *iROBOsim*



# IC Library Functions

[e.g., `sleep(seconds);`]

- All of the functions below work on the XBC, simulator and iROBOsim
- **IC** provides a number of library functions specifically to support use of the XBC (see **IC**'s on-line documentation for the complete list)
  - We've already looked at one of these, `printf()`
  - **IC** library functions are downloaded to the XBC when the **IC** environment connects
  - **IC** library functions are provided to the simulator when it is launched
- After the simulator has been started, the functions can be listed back in **IC** by using the  
*Tools..List Functions* feature of the **IC** toolbar
- `sleep()` is another one of these – it pauses the function's execution for an amount of time equal to the number of seconds given by its argument (expressed as a `float`)
  - On-line documentation appears as follows:

`sleep` [Category: Time]

Format: `void sleep(float sec)`

Waits for an amount of time equal to or slightly greater than `sec` seconds. `sec` is a floating point number. Example:

```
/* wait for 1.5 seconds */ sleep(1.5);
```



# Example Using `rsim_mrp`

- This example moves the robot 2000 ticks forward and then back half the distance
- The `sleep` function gives time for the robot to move before moving on to the next command.

```
#use "iROBOsim.ic"
void main()
{
    //initialize simulator
    //use world keyword
    iROBOinit(RULERWORLD);
    //move both motors fwd
    rsim_mrp(0, 400, 2000L);
    rsim_mrp(1, 400, 2000L);
    //this sleep is longer
    //than should be needed
    sleep(8.0);
    //move both motors back
    rsim_mrp(0, 300, -1000L);
    rsim_mrp(1, 300, -1000L);
    //leave motors running no more
    // than 5 secs
    sleep(5.0);
}
```



# Example Using `rsim_bmd`

- The previous example used `sleep` to give time for the `mrp` commands to execute.
- The “block motor done” or `bmd` command pauses your program just long enough for the motor command to complete
- Using `bmd` eliminates the need to calculate in advance how long you think the motor command will take.

```
#use "iROBOSim.ic"
void main()
{
    //initialize simulator
    //use world keyword
    iROBOinit(RULERWORLD);
    //move both motors fwd
    rsim_mrp(0, 400, 2000L);
    rsim_mrp(1, 400, 2000L);
    //Now wait until motors
    //have reached goal positions
    rsim_bmd(0); rsim_bmd(1);
    //move both motors back
    rsim_mrp(0,300, -1000L);
    rsim_mrp(1,300, -1000L);
    //leave motors running till
    //motors reach positions
    rsim_bmd(0); rsim_bmd(1);
}
```



# Some More Useful Functions

- All of the functions below work on the XBC, simulator and *iROBOSim*
- **beep( ) ;**
  - The simulator is silent, but blinks the **Beeper Status** On/Off, flashes the **Print Buffer**, and shows the **Frequency** value (500)
- Built-in sensors
  - The simulator has “Off/On” toggle buttons for each built-in sensor
    - button “a” simulates the Gameboy A button, “b” the B button, etc
  - **a\_button( )** returns the current value of the A button (which will be 0 for “off” and 1 for “on”)
    - Toggle the button labeled “a” on the simulator by clicking it
    - Execute **printf( "%d\n", a\_button( ) ) ;** from the simulator’s interaction window to verify the value
  - **b\_button( )** returns the current value of the B button





# Moving a Distance

# Calculating Distance

- To find out how far a robot has traveled you need to know:
  - diameter of the wheels
  - the number of wheel rotations (could be a fraction)
- Traveling a specific distance can be very useful
- The distance traveled in one wheel rotation provides you with another way to calculate the diameter of the wheels
  - $\text{distance} = \pi \times \text{diameter}$
  - $\text{diameter} = \text{distance} / \pi$
- For a physical robot:
  - You can check the motor position counter on the XBC (motor display activated by using the shoulder buttons)
    - The motor position counter counts the “ticks” used to measure motor rotation
  - By rolling your bot and observing the change in the motor position counter you can determine how many “ticks” it takes to roll 10cm
    - Divide by 10 to calculate “ticks per cm”
    - Do multiple times and average
- For *iROBOsim* ticks/revolution and diameter of wheels are pre-set:
  - 1100 ticks/rev
  - 5.6cm diameter



# Exercise: Calibrate Your Robot

- In *iROBOsim*
  - each pixel is equivalent to 1cm.
  - Your simulated robot has wheels that are 5.6cm in diameter
  - Each wheel is connected directly to a simulated black gear motor, which requires 1100 ticks per revolution
  - RULERWORLD has marks along the bottom of the world that are 30cm apart (about 1 foot).
  - The motor movement initiated by an **rsim\_mrp** or **rsim\_mtp** command continues until the motor position gets within 100 ticks of the goal position or until another motor command is executed.
  - Your robot starts out with its center at (30, 100) facing right where (0, 0) is the upper left corner and X increases to the right and Y increases downwards.
- Write a program that will move your robot to about position (120, 100) -- which is the 4th hash mark from the left -- and have the robot stop there.
  - Use your knowledge of geometry to calculate the correct arguments to the **rsim\_mrp** function.
  - Have your robot **beep** when it is done.
  - Test your program.
  - If you finish early, calculate the maximum difference that there might be, in cm, between different runs of your program.



# Programming in IC

Repetition using **while**



# Program Flow

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

```
void 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)

- If you wanted to beep 20 times in a row you could type in  
`beep ( ) ;`  
20 times
- But **C** offers other flows than one step after another
- A better way to beep 20 times is to use a *while loop*



# Repetition Using **while**

- Syntax: **while** (<test>) {*statements*}
- Beep 20 times

```
void main()  
{  
    int num;           /* declare counter */  
    num = 1;           /* initialize counter */  
    while (num <= 20) /* loop while num is <=20*/  
    {  
        beep();        /* beep once */  
        num = num + 1; /* add one to the counter */  
    }  
}
```



# Digital Sensors



# Digital Sensors

- Digital sensors are ones which produce an “off” (0) or “on” (1) signal.
- Most GameBoy buttons are accessible as built-in digital sensors when using **IC**
- There are library functions specific to each built-in sensor
- The XBC has digital “ports” to use with two-state plug-in sensors
  - accessed using the **digital** or **extra\_digital** library function



# The GBA SP Buttons & Library Functions

All of these  
functions work on  
the XBC, simulator  
and *iROBOsim*

L “shoulder” button  
(under the hinge)

`l_button()`

back-light off/on button

up button

`up_button()`

left button

`left_button()`

right button

`right_button()`

down button

`down_button()`

select button

R “shoulder” button  
(under the hinge)

`r_button()`

power indicator LED

recharge indicator LED

A button  
(choose)

`a_button()`

B button  
(escape)

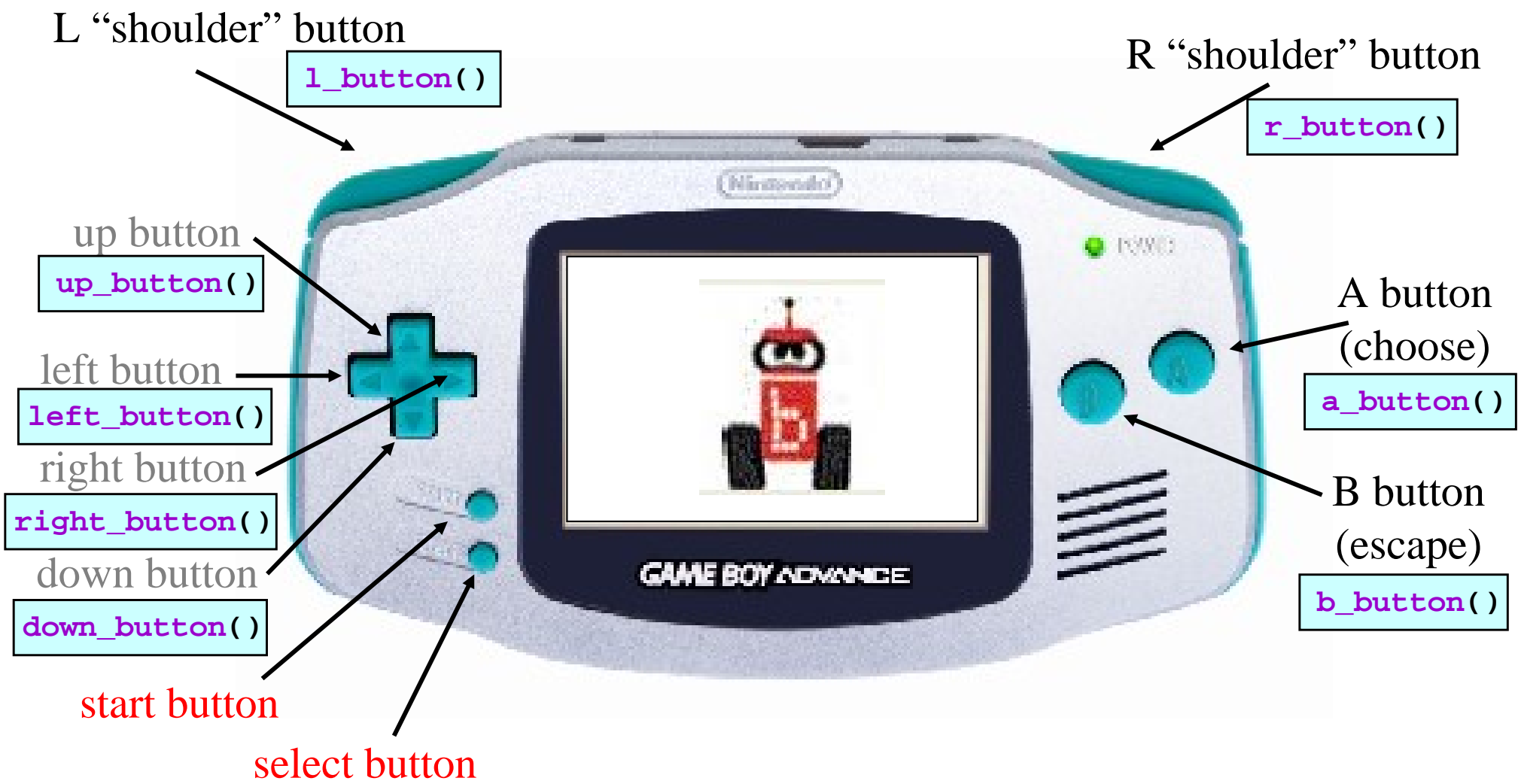
`b_button()`

start button



# Gameboy Buttons & Library Functions

All of the functions below work on the XBC, simulator and *iROBOsim*



# Digital Sensor Port Simulation

- All of the functions below work on the XBC, simulator and *iROBOsim*
- The Simulator allows you to set the values for each of the digital ports
- Digital ports
  - Used with plug-in two-state sensors such as switches
  - Have value either 0 or 1
  - Are the ports numbered 8-15
  - The library function **digital** ( *<port-number>* ) accesses ports 8-15



# Analog Sensors

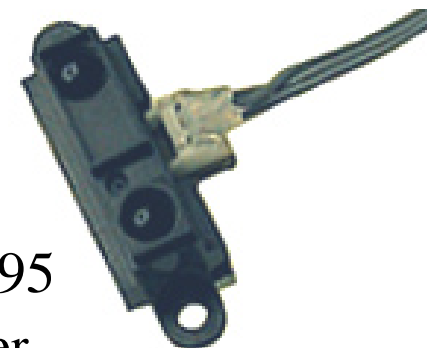
# Analog Sensors

- Analog sensors are ones which produce a range of integer values, not just 0 and 1
- The XBC has analog ports and floating analog ports for plug-in sensors, which are accessed using either the **analog** library function or the **analog12** library function
  - With the **analog** function the return values are scaled to the range is 0-255, with **analog12**, to 0-4095
- There are two types of analog sensors used with the XBC: regular analog sensors (such as light sensors), and floating analog sensors (such as distance sensors)



# Analog Sensor Port Simulation

- All of the functions below work on the XBC, simulator and *iROBOsim*
- The Simulator allows you to set the values for each of the analog ports
- Analog ports
  - Used with sensors that produce a range of values
  - Have integer value scaled to the range 0 through 4095
    - Analog value is represented by a 12 bit binary number
  - Are the ports numbered 0-7
    - Ports 0 and 1 are “floating analog” for use with floating analog sensors
    - Ports 2-6 are used with regular analog sensors.
    - Port 7 is used for monitoring the battery volts.
  - The **IC** library function **analog12**( <port-number> ) is used to access the analog ports
  - The library function **analog**( <port-number> ) scales the return value to the range 0 to 255



# *iROBOsim* sensors



# *iROBOsim* Sensors

- The *iROBOsim* robot has 4 touch sensors, which are pre-set for 4 digital ports:
  - **digital(8)** returns 1 if the right front bumper is pressed and 0 if it is not being pressed.
  - **digital(9)** does the same for the right rear bumper.
  - **digital(14)** does the same for the left rear bumper.
  - **digital(15)** does the same for the left front bumper.
- If *iROBOsim* detects the robot running into an obstacle (graphically, a blue line), the appropriate sensor becomes pressed.
- Note that in the simulation the robot will go right through the wall unless your program tells it to do something else!



# Mode Switching Using **while**

- Syntax: **while** (<test>) {statements}
- Move forward until **B** button is pressed

```
#use "iROBOsim.ic"
```

```
void main()
```

```
{
```

```
  iROBOinit(RULERWORLD);
```

```
  rsim_mav(0,200);           //start left motor
```

```
  rsim_mav(1,200);           // start right motor
```

```
  while (b_button() == 0)
```

```
    { /* just waiting for button press */ }
```

```
  rsim_a0();                 // stop all the motors
```

```
  beep(); // beep once to alert user robot is stopped
```

```
  //report dist traveled
```

```
  printf("Stopped after %1 ticks\n",
```

```
    rsim_get_motor_position_counter(0));
```

```
}
```

**==** means *is equal to*

**||** means *or*

**&&** means *and*

**!** means *not*



# Exercise

- Modify the program above to create a ping pong robot that bounces between walls

– HINT:

- go forward; while *neither* front bumper is hit  
- wait;
- go backwards; while *neither* rear bumper is hit - wait;
- stop;
- to check if *X* and *Y* are both 0 do (  $X==0$   
&&  $Y==0$  )

$==$  means *is equal to*

$||$  means *or*

$\&\&$  means *and*

$!$  means *not*



# Differential Steering and Turns

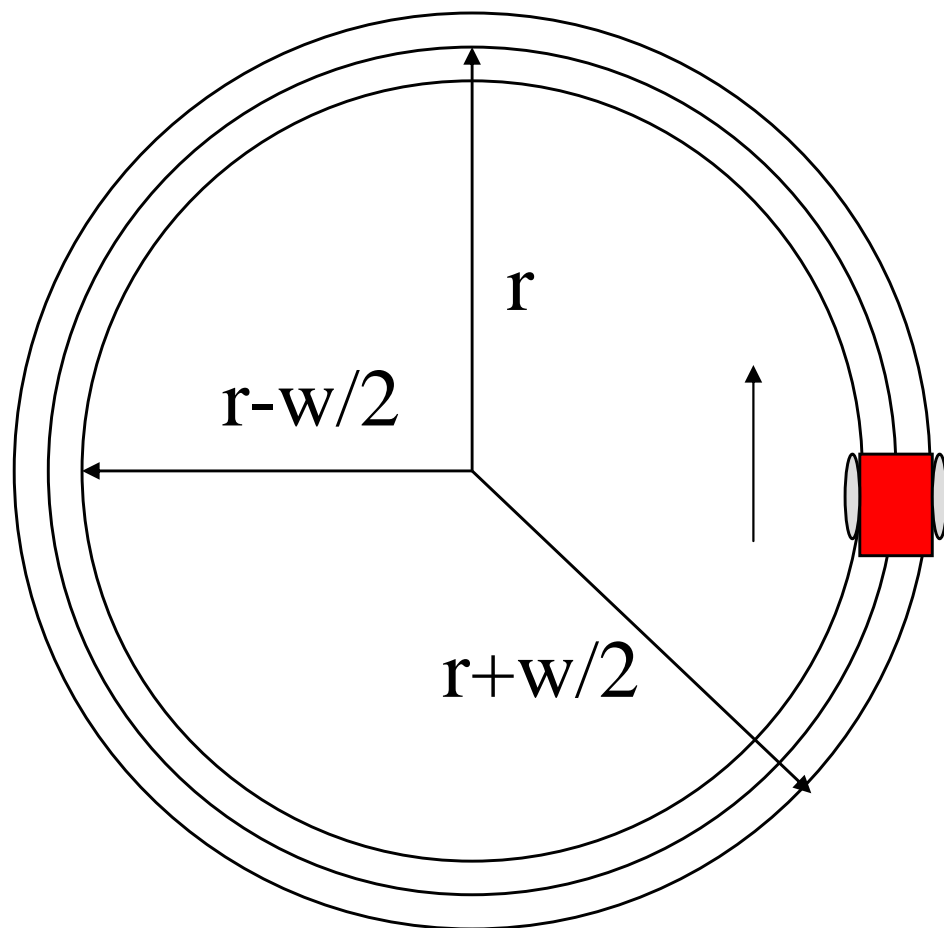


# Differential Steering

- Steering is accomplished by having 2 independently powered wheels or treads mounted along a single axis
  - If the motors are operated at the same speed, the vehicle goes straight
  - If the motors are operated at different speeds, the vehicle turns, or spins



# Calculating Turns – Differential Steering

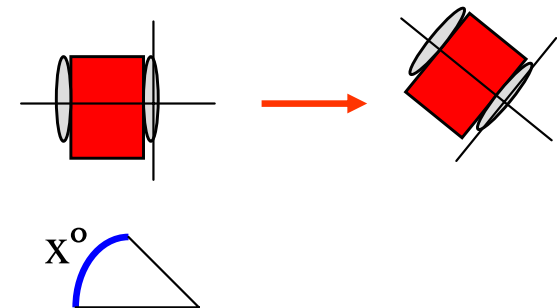


**For a robot with a wheel separation of  $w$ ...**

- When traveling a complete circle the robot will travel  $2\pi r$
- The left wheel will travel:  $2\pi r - \pi w$
- The right wheel will travel  $2\pi r + \pi w$
- If the right travels  $2\pi w$  further than the left, the robot's made a complete CCW circle
- If the left goes  $\pi w$  more than the right, the robot has turned CW  $180^\circ$
- This determines the degree of turn for any angle measured in radians (e.g.,  $\pi/2 = 90^\circ$ )

# Example: Turn $x^\circ$ Using 1 Motor

- Measure your robot's wheel base  $w_b$   
(distance between the center of the tire treads)
  - for *iROBOSim*  $w_b = 12\text{cm}$
- You should already have determined “ticks per cm” for your bot.
- If you power only the left wheel your robot will turn like this
- and trace out an  $x^\circ$  arc
- $x/360$  is the fraction of the full circle being traced, so the tire needs to move



■ ■

$2\pi \cdot w_b \cdot x/360$  for an  $x^\circ$  turn

# Example **IC** Code to Turn $x^\circ$ (using 1 motor)

*// global variables used*

**float** pi=3.1416;

**float** wb=12.0;           *// wheelbase in cm*

**int** tickspercm=62;      *// dependent on wheel size*

**int** mL=0, mR=1;        *// motor ports for left and right motors*

**int** vel=250;            *// motor speed for turn*

*// Note that this is a function. A main is needed to call it!*

**void** turn\_mL(**float** x) { *// turn using L wheel (x >= 0)*

**long** pos, curr;

**float** dist;

    dist = 2.0\*pi\*wb\*x/360.0; *// distance in cm to roll along*

    pos = (**long**) (dist\*(**float**) tickspercm); *// ticks needed to do it*

    rsim\_mrp(mL,vel, pos); *// operate wheel at specified velocity*

*// move until the motor position counter is pos more*

*// bmd keeps function from returning till done*

    rsim\_bmd(mL);

}





# Example: Turn in Place $x^\circ$

- Turning in place simply requires running one wheel in the opposite direction from the other
- Taking the computation for turning  $x^\circ$  and dividing by 2 gives the amounts to move each wheel
  - One tire needs to move  $\pi \cdot w_b \cdot x/360$
  - The other needs to move  $-\pi \cdot w_b \cdot x/360$



# Example IC Code to Turn in Place $x^\circ$

## **if – else** to select direction

```
//global variables used
float pi=3.1416;
float wb=12.0;      // wheelbase in cm
int  tickspercm=62; // dependent on wheel size
int  mL=0, mR=1;    // motor ports for left and right motors
int  vel=250;       // motor speed for turn

// Note that this function handles CW and CCW turns. Also, it tracks motor position
// itself rather than using mrp and bmd.
```

```
void turn(float x) { // x > 0 for left turn, x < 0 for right
    long pos, currL, currR;
    int dir; // used to keep pos > 0
    if (x < 0.0) dir = -1;
    else dir = 1;
    pos = (long)((float)tickspercm*((float)dir*pi*wb*x/360.0));
    currL=rsim_get_motor_position_counter(mL);
    currR=rsim_get_motor_position_counter(mR);
    if (x > 0.0) { // turn left
        rsim_mav(mR,vel); rsim_mav(mL,-vel);
        while (rsim_get_motor_position_counter(mR) < pos+currR);
    }
    else { // turn right
        rsim_mav(mR,-vel); rsim_mav(mL,vel);
        while (rsim_get_motor_position_counter(mL) < pos+currL);
    }
    rsim_ao();
}
```

*1/2 the distance  
used before*



# Exercise: Make a Square

- Use the functions for turning and your calibration experience and write a program to have your robot move in a square 60cm on a side.
- In your main routine, put the line:  
**iROBOTrail=1;**  
right after the call to **iROBOinit(RULERWORLD)**.
  - This will have your robot paint a trail behind it making it easier to evaluate the quality of your geometry.
- Once you have mastered the square, try having your robot make a 60cm circle -- note that you may have to move the robot first to give it some room



# *if-else* Selection Within *while*

(wandering robot)

```
#use "iROBOsim.ic"
//initialize the simulator with EMPTYWORLD, RULERWORLD, ARENAWORLD
void main() {
    iROBOinit(ARENAWORLD); iROBOtrail=1;
    rsim_mav(0,250); rsim_mav(1,221); //left motor & right motor
    while(!digital(10) || iROBOstopSim) {
        if(digital(8)) { //right bumper
            rsim_mav(0,-200); rsim_mav(1,-200); sleep(.5);
            rsim_mav(1,300); rsim_mav(0,-300); sleep(.5); // spin CCW
            rsim_mav(0,400); rsim_mav(1,400);
        }
        else {
            if(digital(15)) { // left bumper
                rsim_mav(0,-400); rsim_mav(1,-400); sleep(.5);
                rsim_mav(0,400); sleep(.5); //spin CW
                rsim_mav(1,400);
            }
        }
    }
    iROBOend();
    rsim_ao();
}
```

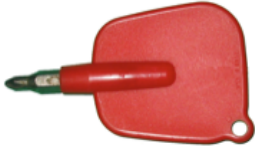





# The 2007 Botball Kit Electronics







# The Botball Electronics Kit (1)

## Tool + Digital Sensors

| Description        | Piece   | # in kit |
|--------------------|---|----------|
| Screwdriver        |    | 1        |
| lever sensor       |    | 3        |
| large touch sensor |  | 6        |
| small touch sensor |  | 4        |

## Analog Sensors

| Description         | Piece   | # in kit |
|---------------------|---|----------|
| range finder        |    | 3        |
| ir "top hat" sensor |   | 4        |
| light sensor        |  | 5        |
| sonar sensor        |  | 1        |



# The Botball Electronics Kit (2)





## Servos & Motors

| Description      | Piece   | # in kit |
|------------------|---|----------|
| standard servo   |     | 4        |
| black gear motor |     | 4        |
| White gear Motor |   | 4        |
| Silver Motor     |   | 1        |
| Ferrite Beads    |  | 4        |



# The Botball Electronics Kit (3)

## XBC Controller

| Description                     | Piece  | # in kit |
|---------------------------------|--|----------|
| XBC v3 AC adapter               |    | 2        |
| XBC v3 optional GBAsp & charger |   | 2        |
| XBC USB cable                   |  | 2        |
| XBC Camera Extension Cable      |  | 2        |





# Building with Lego: Demobot



# Building With LEGO

- Like programming, mastering the art of building with LEGO takes time – the following are a few tips to get you started:
  - Adding MORE Lego to a structure does NOT automatically make the structure stronger!
  - Use pins and “axle joiners” – with the newer Lego pieces, pins and joiners (all sorts of types) are your new best friends
  - If the Lego pieces aren’t fitting together “naturally”, don’t force them – there are many other combinations of pieces that might solve the problem
- Borrow “tricks” from examples – for instance examine the gearing and joints in the Demobot example (next)



# Build Demobot

- Three building aids
  1. Solidworks eDrawing of Demobot
    - Use eDrawing Viewer
    - Hide and reveal pieces in assembly order
  2. Powerpoint step by step instructions
  3. Real-time step by step assembly movie (see an expert build a demobot that is almost the same as the one in the directions)



# Preparing Motors for Demobot

- Demobot uses a DC motor for each wheel (2)
  - Plugs into the XBC motor ports
    - gray cable with 2 prong plug
  - Remove the current (red) mounting plate and replace as directed (supplied screwdriver)
    - Use supplied brass screws (in a separate bag) with the brass collars from the servo motor bag
- Demobot uses a DC motor for the claw (1)
- Demobot uses a servo motor for the arm (1)
  - Plugs into the servo motor ports
    - Black (-), Red (+), Yellow (S) cable
  - Remove the current mounting plate and replace with a red mounting cross



# Motor Ports

2 DC motor ports on each side  
(2 and 3 on this side)



To match with *iROBOsim*, plug the left motor into DC motor port 0 and the right into 1; we will also assume the gripper is in DC motor port 2 and the arm in servo motor port 0

We will determine motor polarity for the DC motors shortly (ie., which way the plug should be inserted)

4 servo motor ports  
(S0, S1, S2, S3)



-  
+  
S



# Day 2



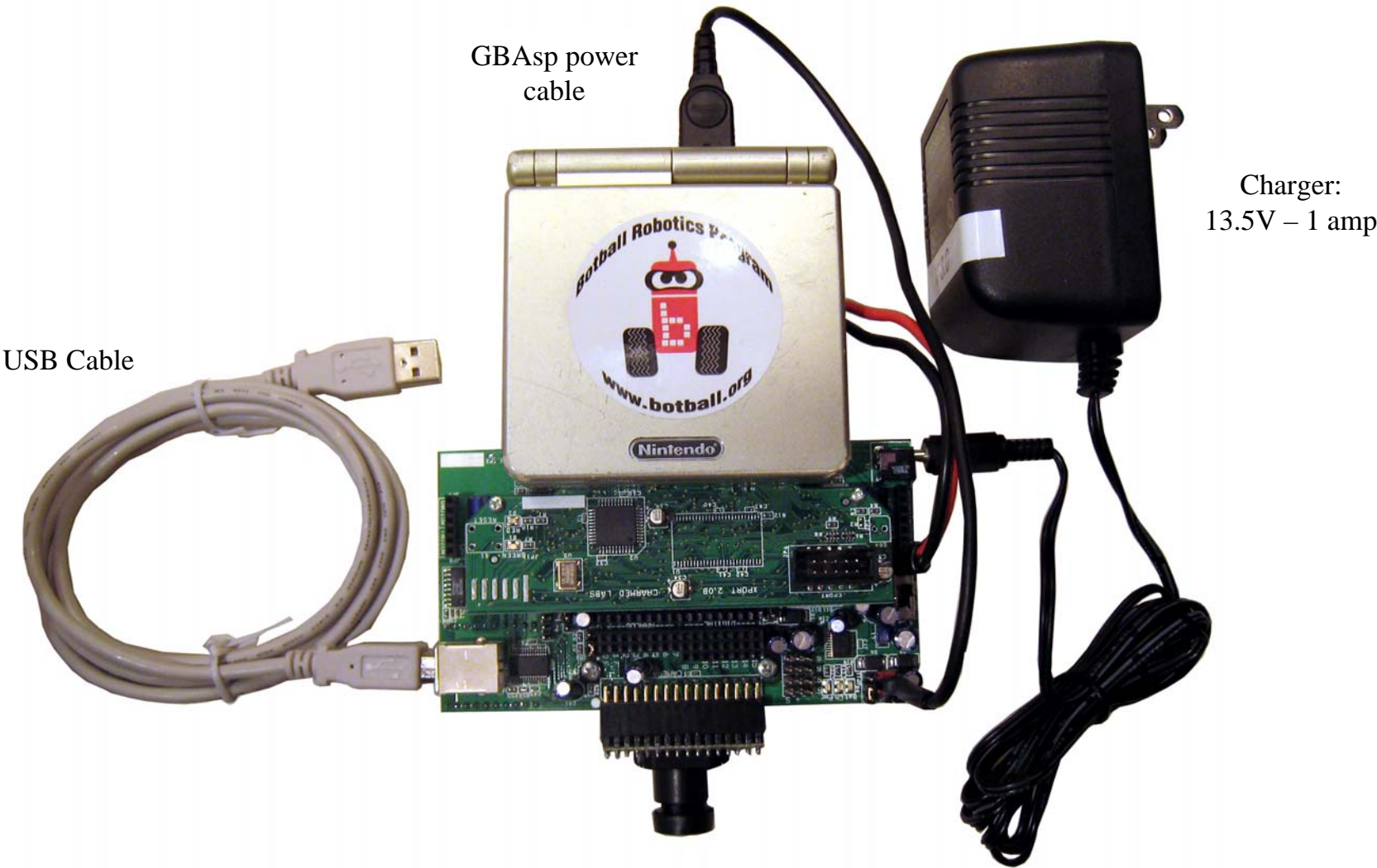
# XBC Checklist

- XBC
  - XBC main board with camera
  - XPORT FPGA (small circuit board screwed on top)
  - Battery boxes (mounted on bottom) w/Lego attachments
  - 7.2v NiMH battery pack
  - |                                       |   |  |
|---------------------------------------|---|--|
| – GBAsp (XPORT slides into game slot) | } | <i>Except for teams supplying<br/>their own Gameboys</i> |
| – GBAsp power cable and wall charger  |   |  |
- USB-AB cable
- AC Adapter 13.5v





# XBC v3 Board Setup

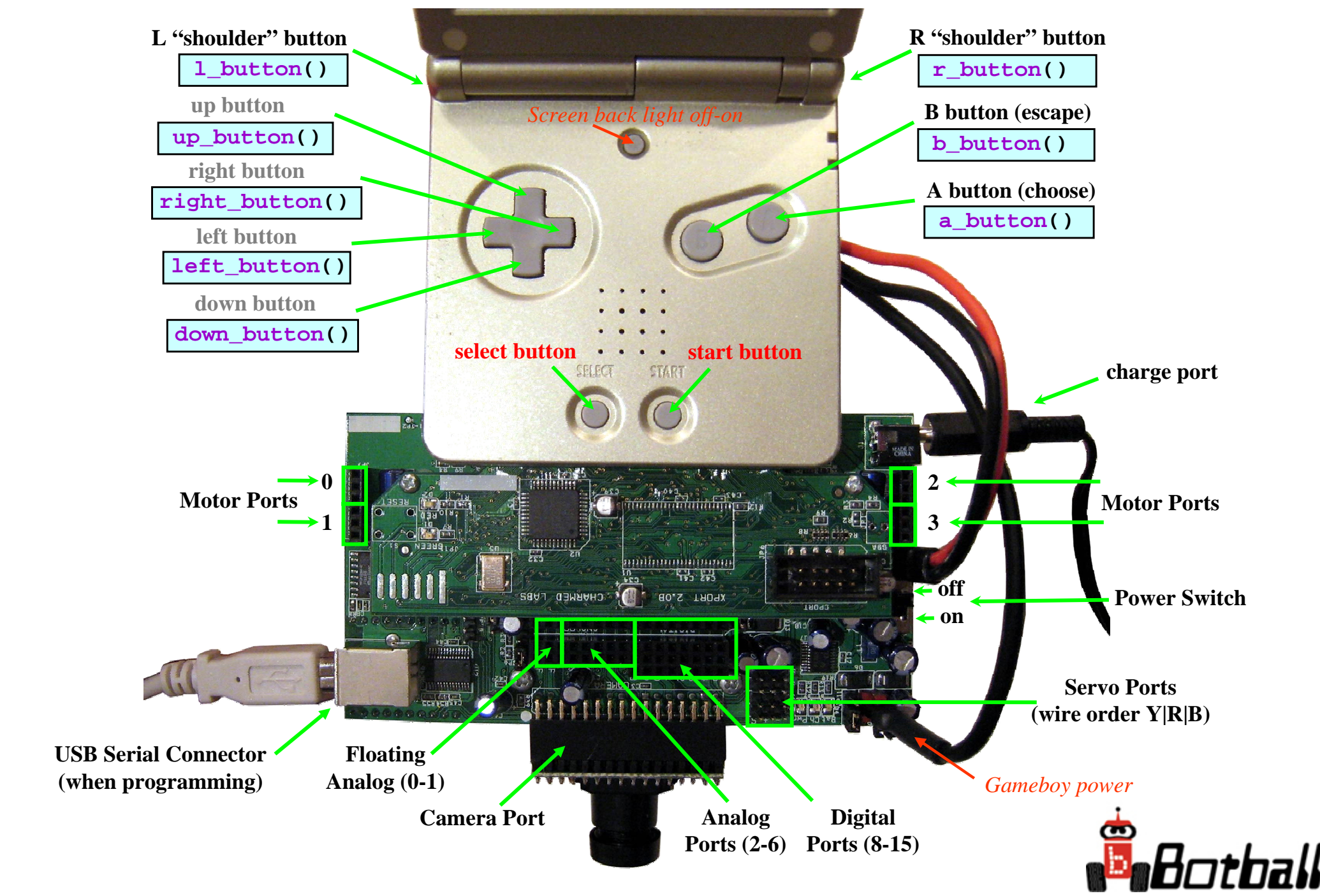


XBCv3 w camera & GBAsp:  
7.2V NiMH 6 cell Battery





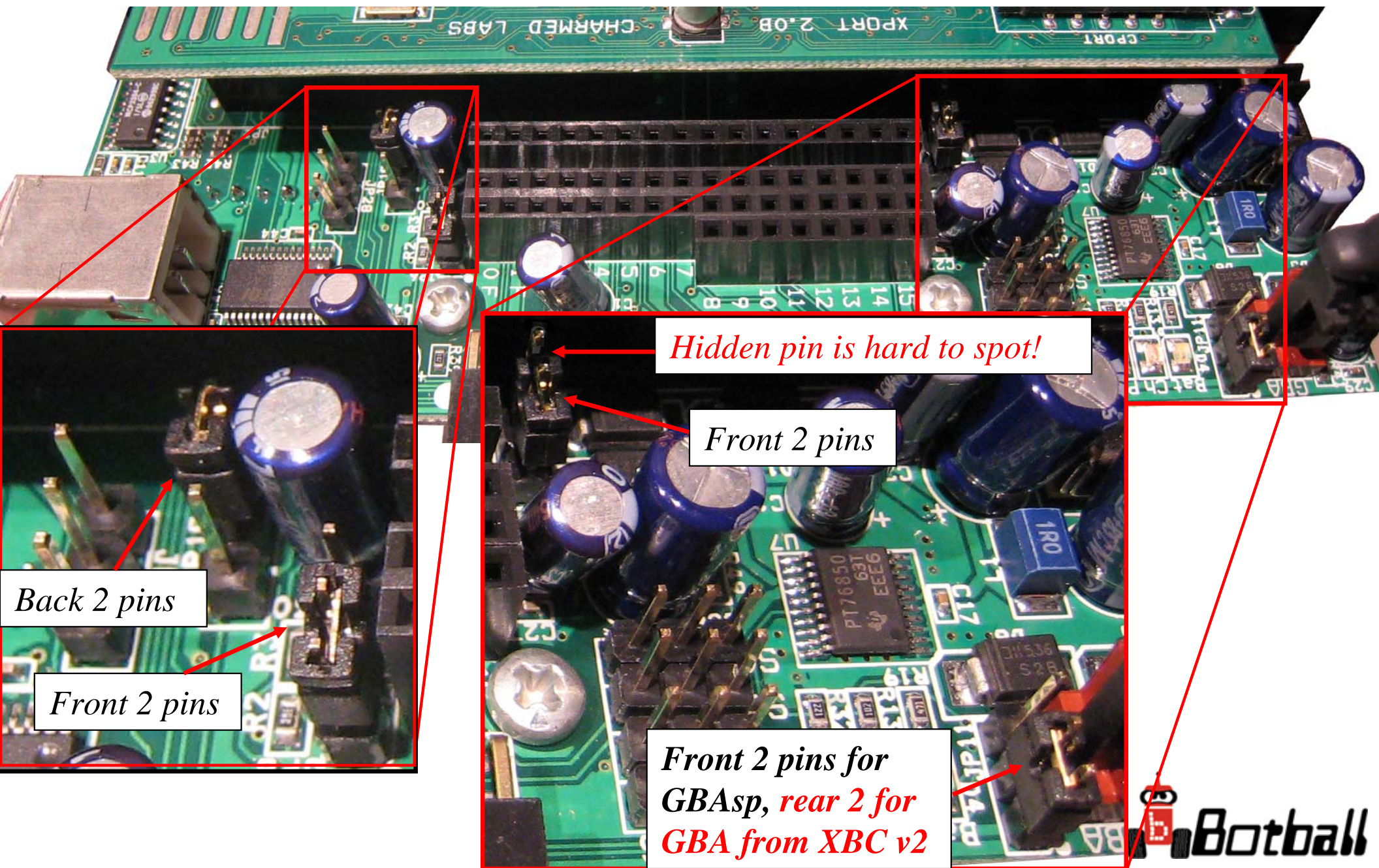
# XBC (v.3 w/GBA SP and Camera)





# XBC v3 Jumper Settings

(visually inspect to see if settings are correct)



# Charging XBC v3 Batteries

- Charging is best accomplished using the XBC 13.5v 1000mA AC charger plugged into the XBC with the XBC turned off
  - If your Gameboy has functional batteries, you should turn it off, too!
- Yellow charging light flashes when battery pack is being rapidly charged
- Yellow charging light turns solid on when battery pack is 60% charged and indicates a slower, maintenance level charge
- To achieve a full charge, allow the XBC to charge for 3 hours
- The battery pack installed in your XBC is a 2000mAh 7.2V NiMH Pack
- The XBC will charge while it is on, but the times will be longer and vary depending on how the XBC is being used



# A Note on Gameboy Options

- For Botball, you can use any one of the following Gameboy models with the XBC:
  - Gameboy Advance (GBA)
  - Gameboy Advance SP (GBA<sub>SP</sub>)
  - Gameboy Micro
- Dual Screen (DS) Nintendo models may not be used in place of the above models for Botball competitions
- If you use the Gameboy power connection on the XBC
  - If your Gameboy has functional batteries and you turn off the XBC before turning off the Gameboy, the XBC won't turn off until you turn off the Gameboy (drawing power from the Gameboy batteries!)
  - The Gameboy batteries will charge only if the XBC is on, but this will slow down XBC charging (so Gameboy batteries should be charged independently)
  - It is best to remove the Gameboy from the XBC for charging (in particular, trying to charge an installed Gameboy while the XBC is charging is not a good idea)





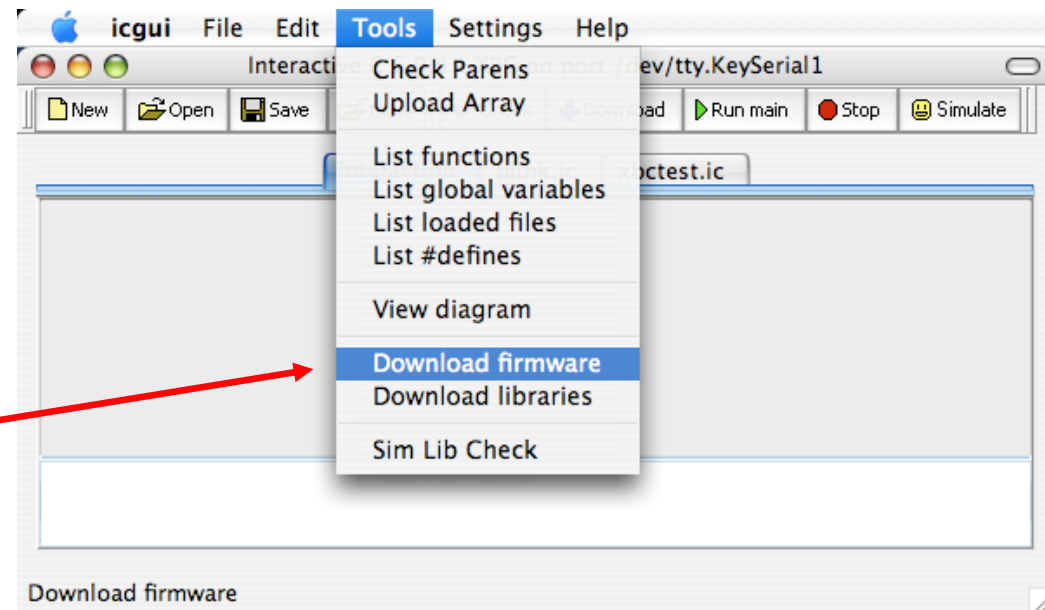
# Connecting to the XBC v3

- This year's XBC has a USB to serial converter built into the XBC
- You must have the proper USB driver installed on your system
- Get the XBC USB driver from the team CD (or via the web at <http://www.ftdichip.com/Drivers/VCP.htm>)
- Install the driver
- Connect the XBC via USB cable to your computer
- In **IC**, select the new serial port that appears (typically COM 5 or higher on PC or /dev/tty.usbserial-xxx on Mac)



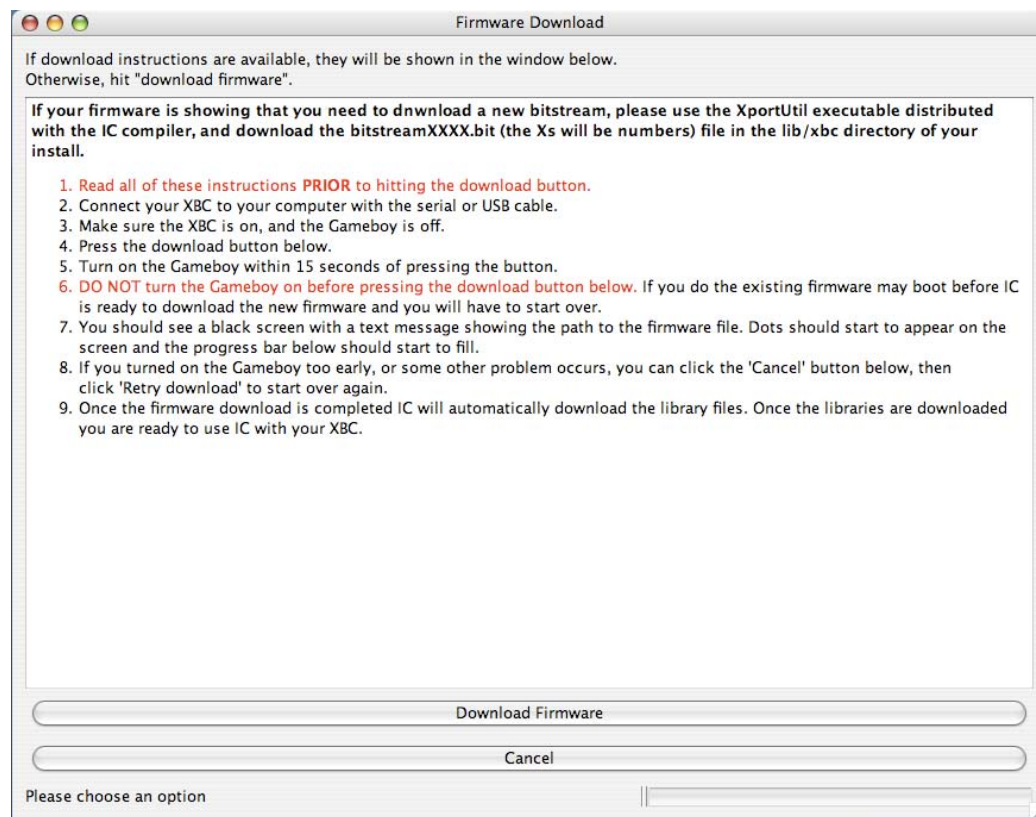
# Loading Firmware from **IC** (1)

- When you download a program to the XBC, you may get a firmware warning. If so:
  - Select “Download Firmware” from the “Tools” menu



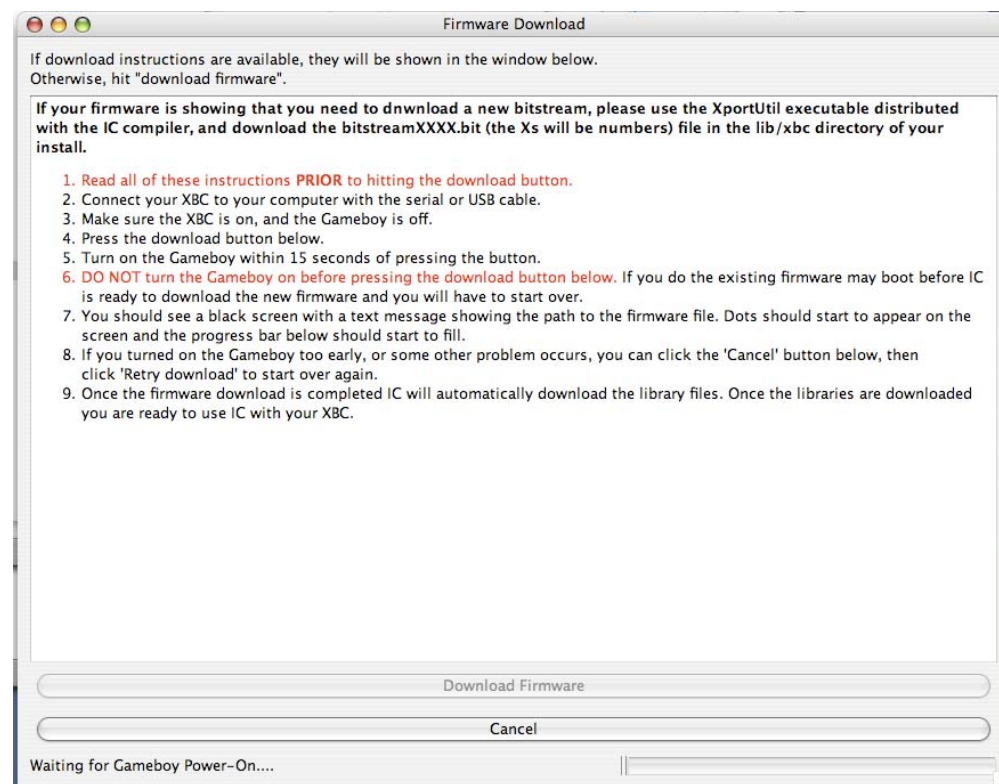
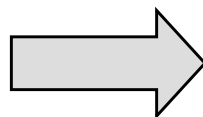
# Loading Firmware from **IC** (2)

- The “Firmware Download” window will appear
- Read the instructions
- Turn off the GameBoy
- Click the “Download Firmware” button



# Loading Firmware from IC (3)

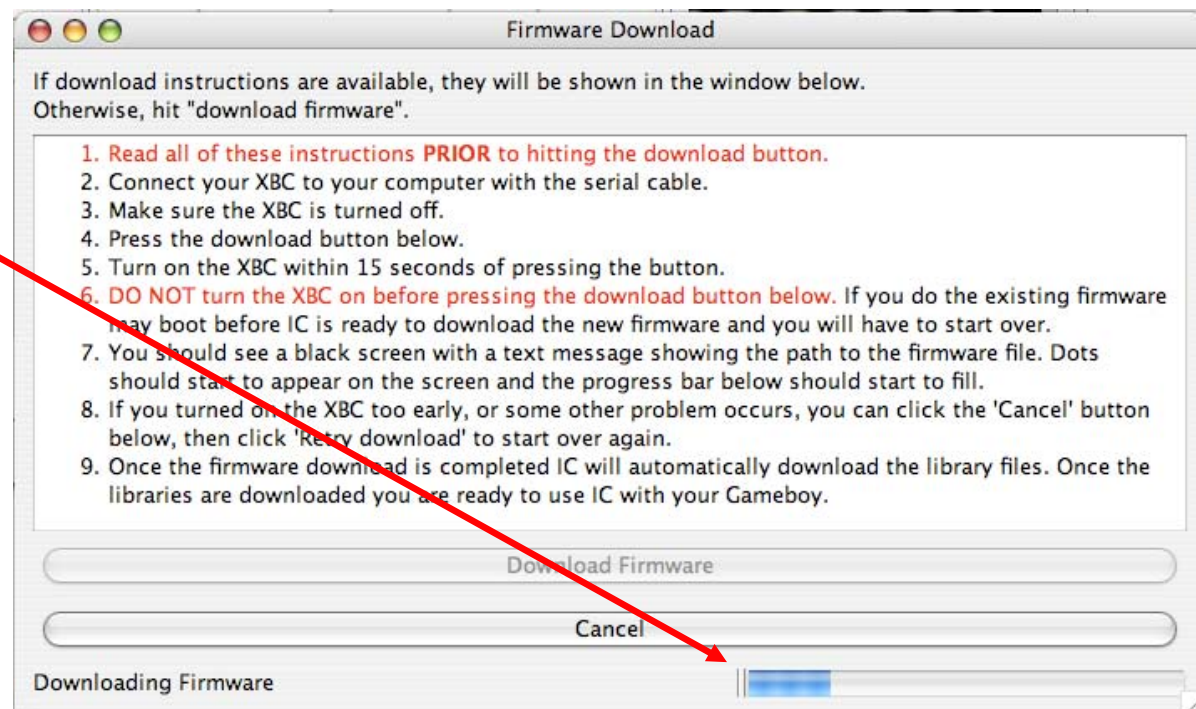
- When the “Waiting for Gameboy Power On...” message appears at the bottom...
- Turn on the Gameboy





# Loading Firmware from IC (4)

- The progress bar will advance to the right
- The bar and the window will disappear when the firmware load is complete
- Do not turn off the XBC or disconnect things until either the firmware load has been cancelled or it has been completed and the Gameboy has reset



# What to Load and When

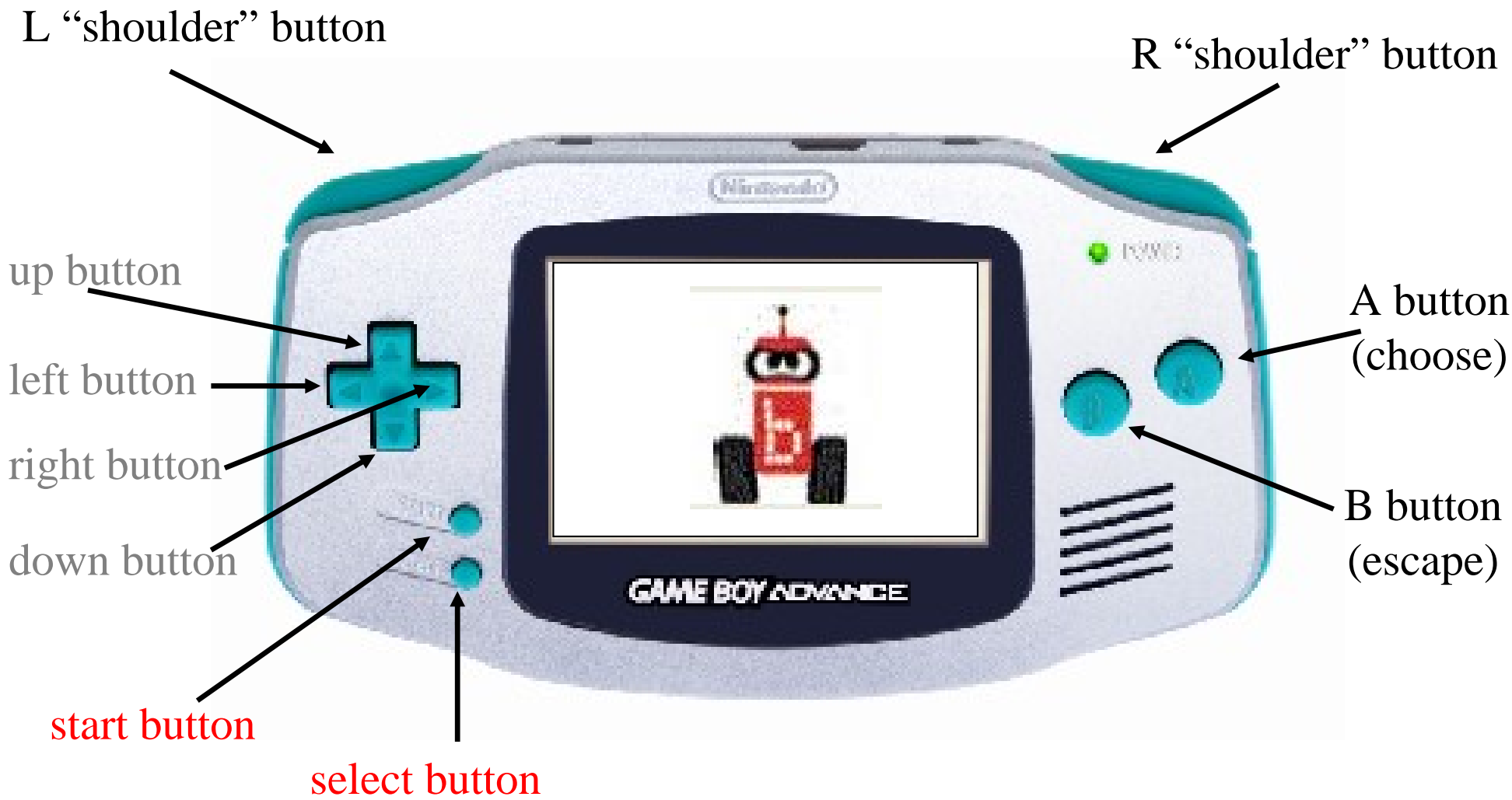
- Bitstream: You may never have to load it -- bitstreams are updated very rarely
  - Current version is: bitstream8d05.bit
  - This version requires firmware version 3.0 or higher
  - **IC** will tell you when the bitstream needs to be replaced
  - Bitstream is updated using the Xport utility in the **IC** folder (see appendix)
- Firmware: The firmware version is coordinated with the version of **IC**. **IC** will give a warning if the two don't match (in which case you should load the matching firmware version from **IC** via *Tools..Download firmware*)
- Programs: When you are ready to run your program, use the "Download" button to put your program onto the XBC.



# Gameboy Buttons



# The Basic GBA



# The GBA SP



# XBC Buttons

(with Gameboy installed)

- With a Gameboy installed, the XBC has 6 buttons and one “D-pad” (directional pad) which is actually just 4 buttons – thus there are a total of 10 buttons on the XBC
- The **start** button only starts (or stops) the program loaded on the XBC
- The **select** button moves you back and forth between the *program* window and the *menu* window
- When in *menu mode* the other buttons have *miscellaneous* uses (for example A usually selects that menu option and B moves back to the prior menu, the up button moves up, down moves down, ...)
- The GBA SP backlight on/off button is not used by the XBC





# XBC Buttons & More

- In an IC program, the programmer can use 8 (of the 10) XBC buttons – the only two that the programmer can NOT use/access are the start and select buttons
- `a_button()`, `b_button()`, `r_button()`,  
`l_button()`, `up_button()`, `down_button()`,  
`left_button()`, `right_button()`
  - All return 1 if currently pressed down, 0 otherwise
- Analog port 7 is reserved for battery voltage and the function `power_level()`

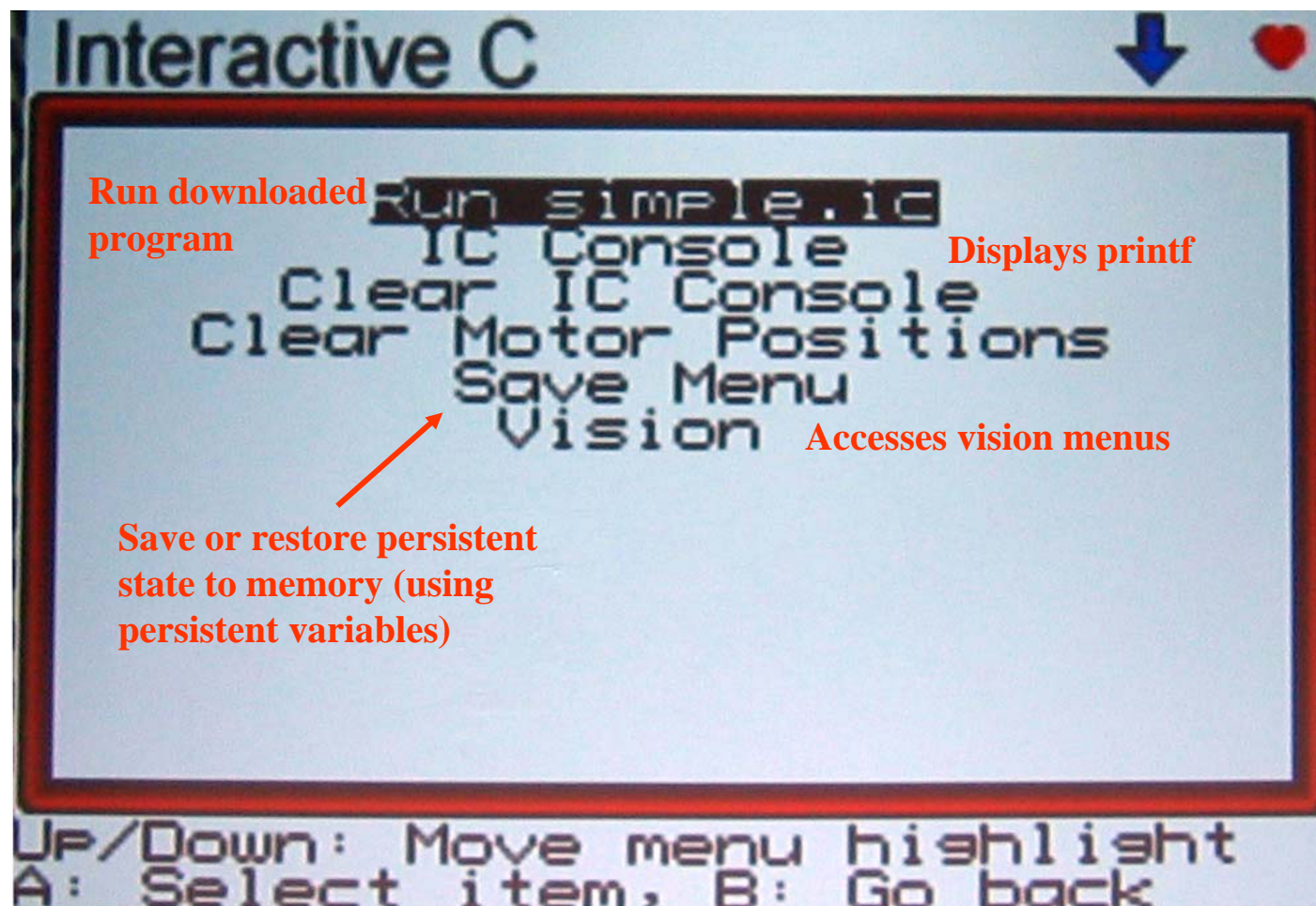


# XBC Display Menus



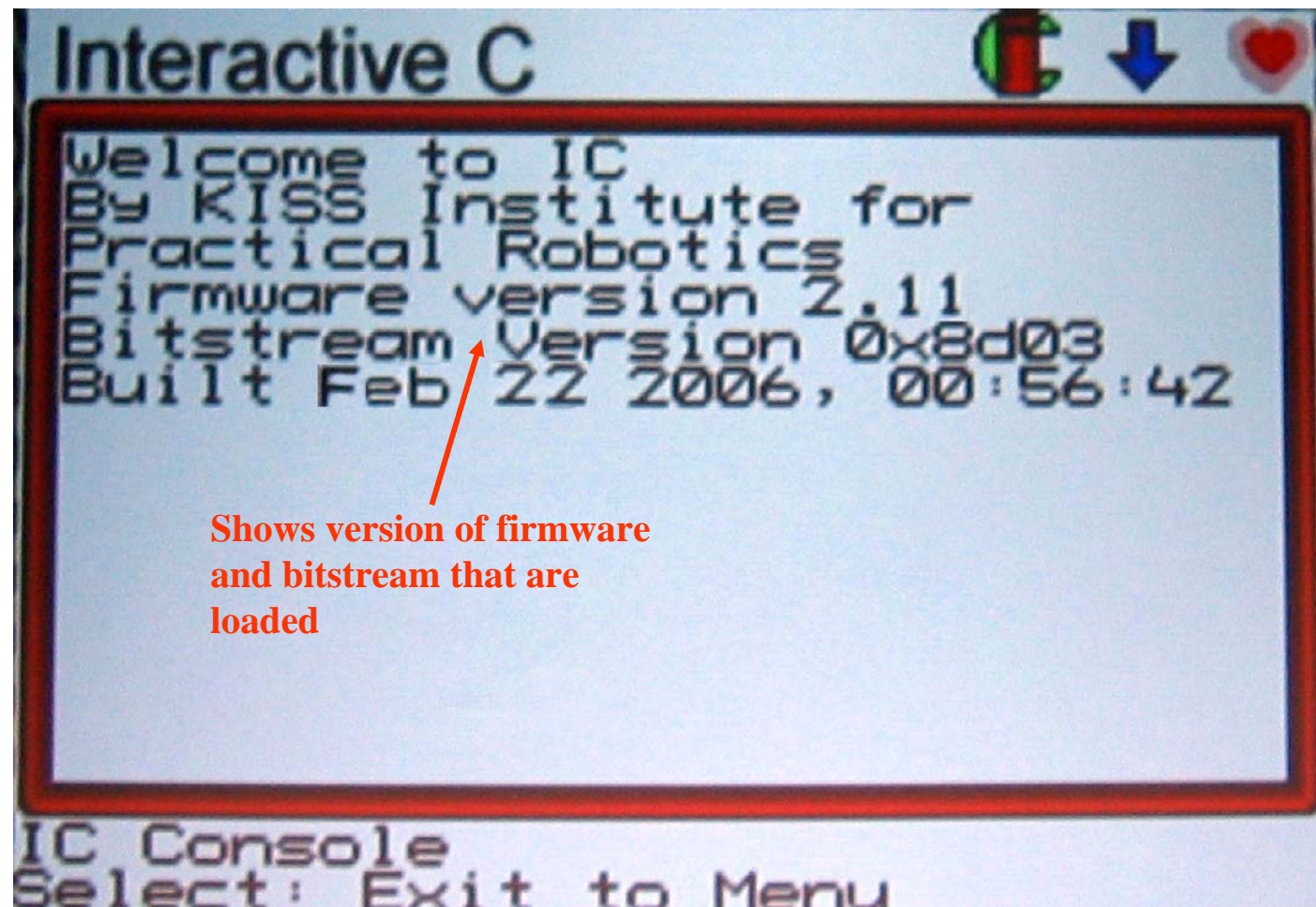
# XBC Main Display

- The XBC menu is navigated using the up and down buttons of the direction pad (left of display) and the **A** button to select



# Console Display at Boot

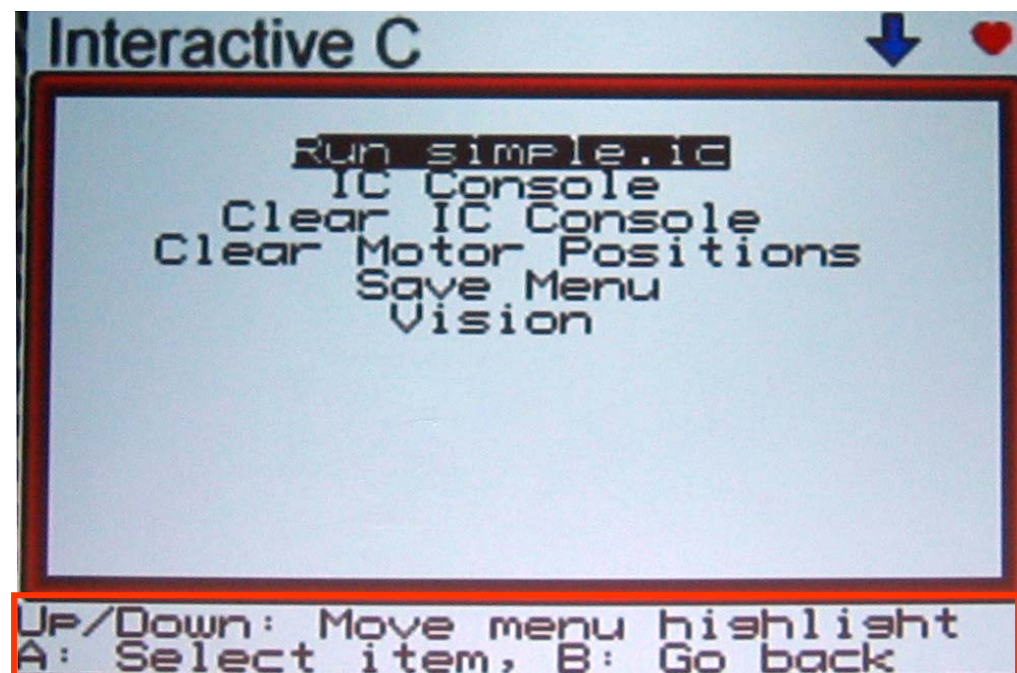
- After starting the XBC, use the directional pad to select the IC Console, which will have a display showing what versions of the firmware and bitstream are being used





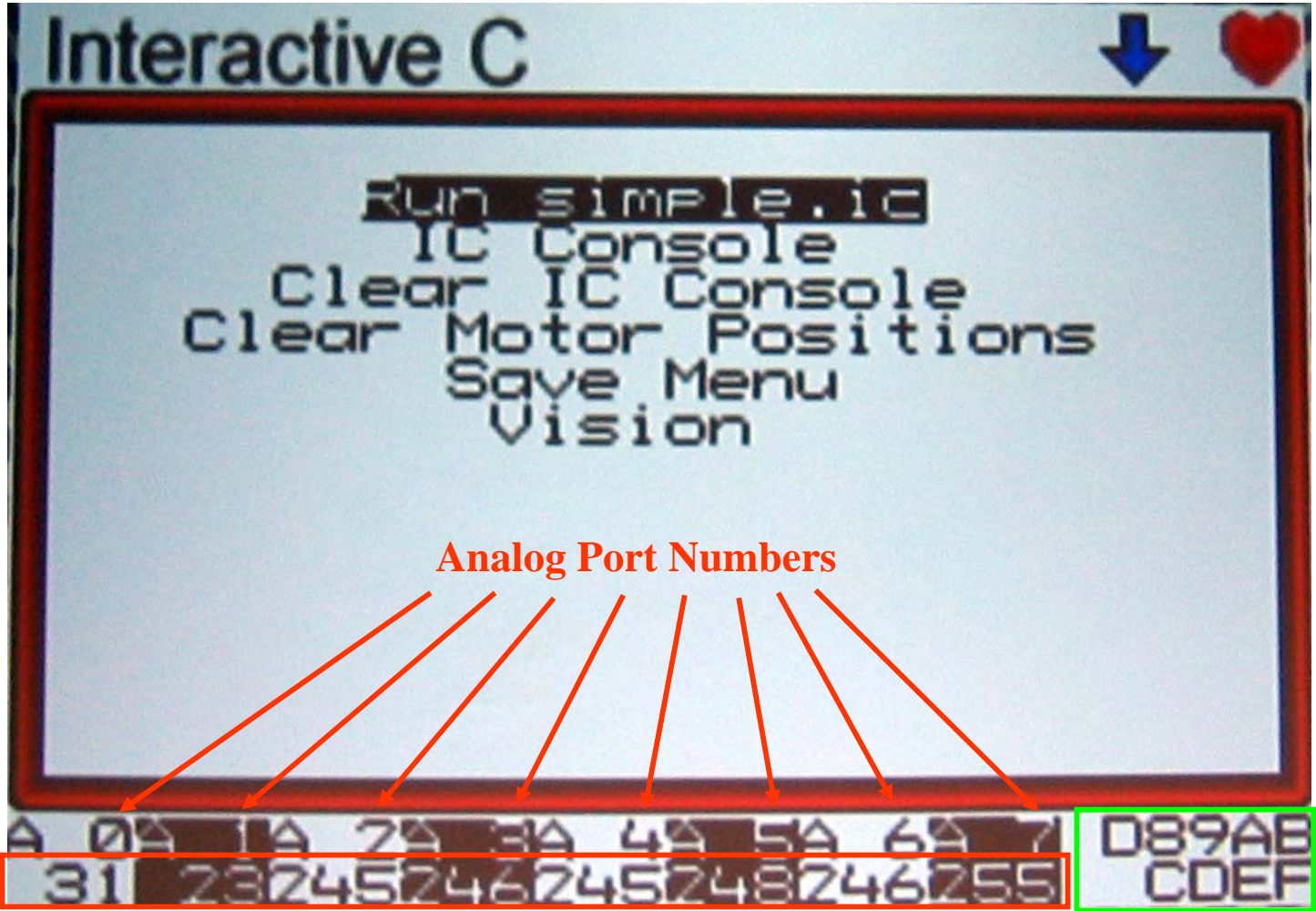
# XBC Status Display

- Pressing L & R buttons (top edge of GBA) scroll through three displays at the bottom of the GBA screen
  - IC Status
  - Sensor Status
  - Motor Status
  - Power & Servos



Status window

# XBC Sensor Status Display



*Use this display to test a plugged in sensor to see if its value changes when you manipulate it!*

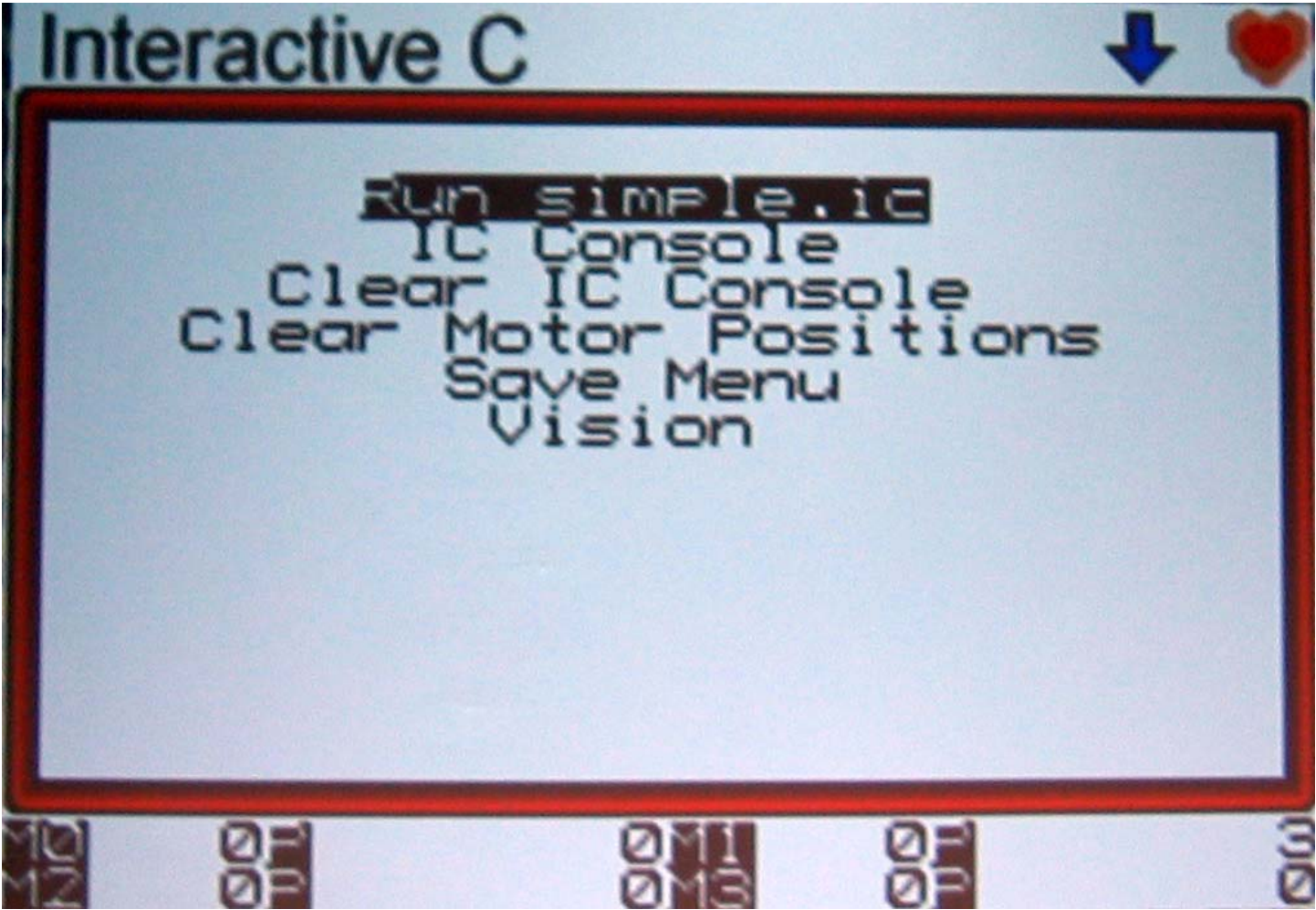
Analog port values

Digital port numbers (hex)  
Background color indicates state





# XBC Motor Status Display



Motor number

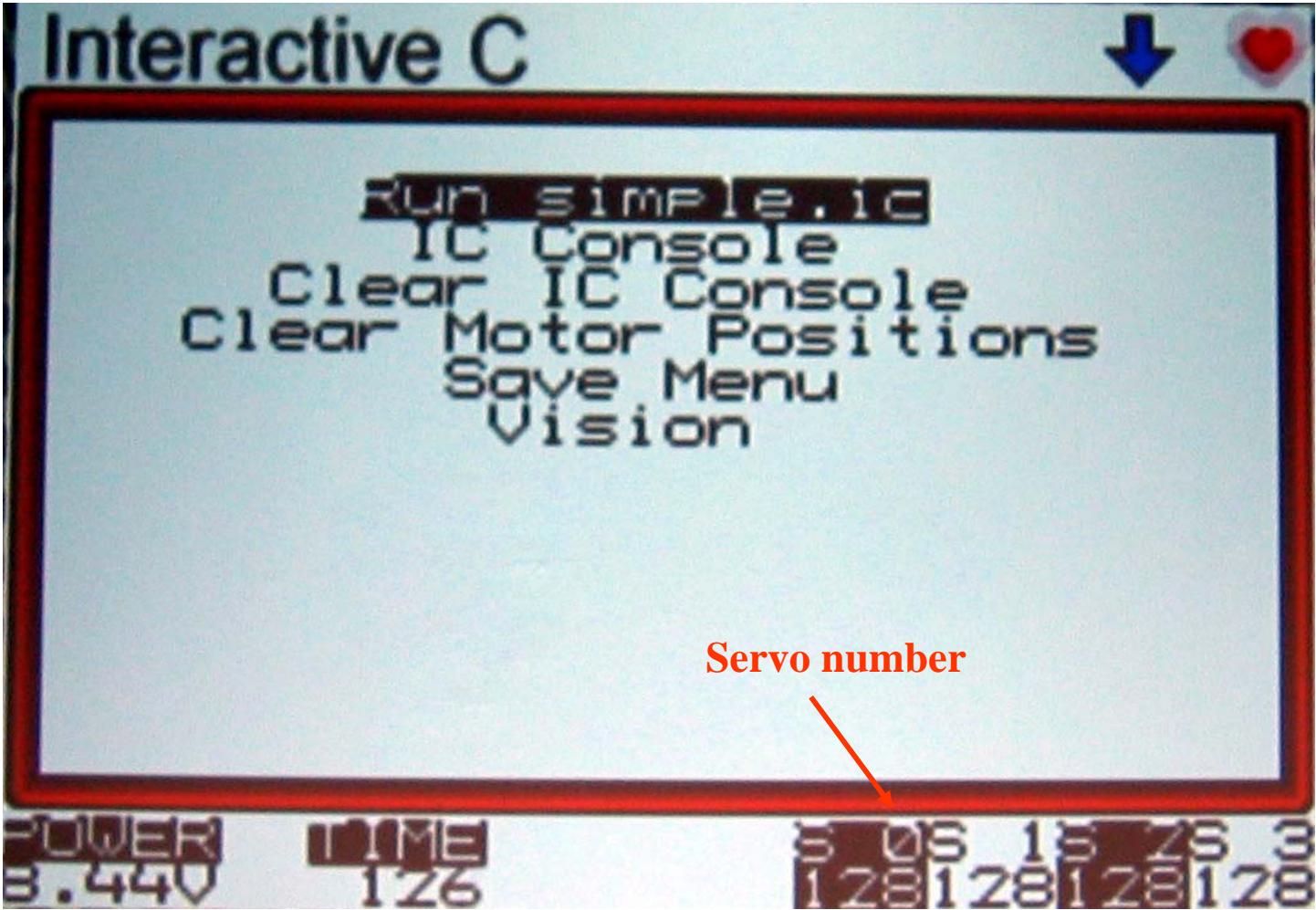
Motor speed (0-100)

Motor position

*If you rotate a motor forward and its position counter changes negatively, reverse your motor plug!*



# XBC Power & Servos Display



Unlike sensors and DC motors, servo motors provide no feedback, so the position is relevant only when a servo command is executed

if less than 7V --  
Past time to recharge!

Seconds since XBC was booted

Servo position



# Determine DC Motor Polarity for Demobot

- Use the shoulder buttons to cycle the display to the XBC Motor Status Display
  - You want the motor position counter for each wheel to increase when you turn the wheel forward
    - If it does not, reverse the motor plug for the wheel
  - You want the gripper to open when you rotate its motor to increase the motor position counter
    - Start with the gripper closed
    - If it does not open when rotating to increase the counter, reverse the plug



# Using Sensors





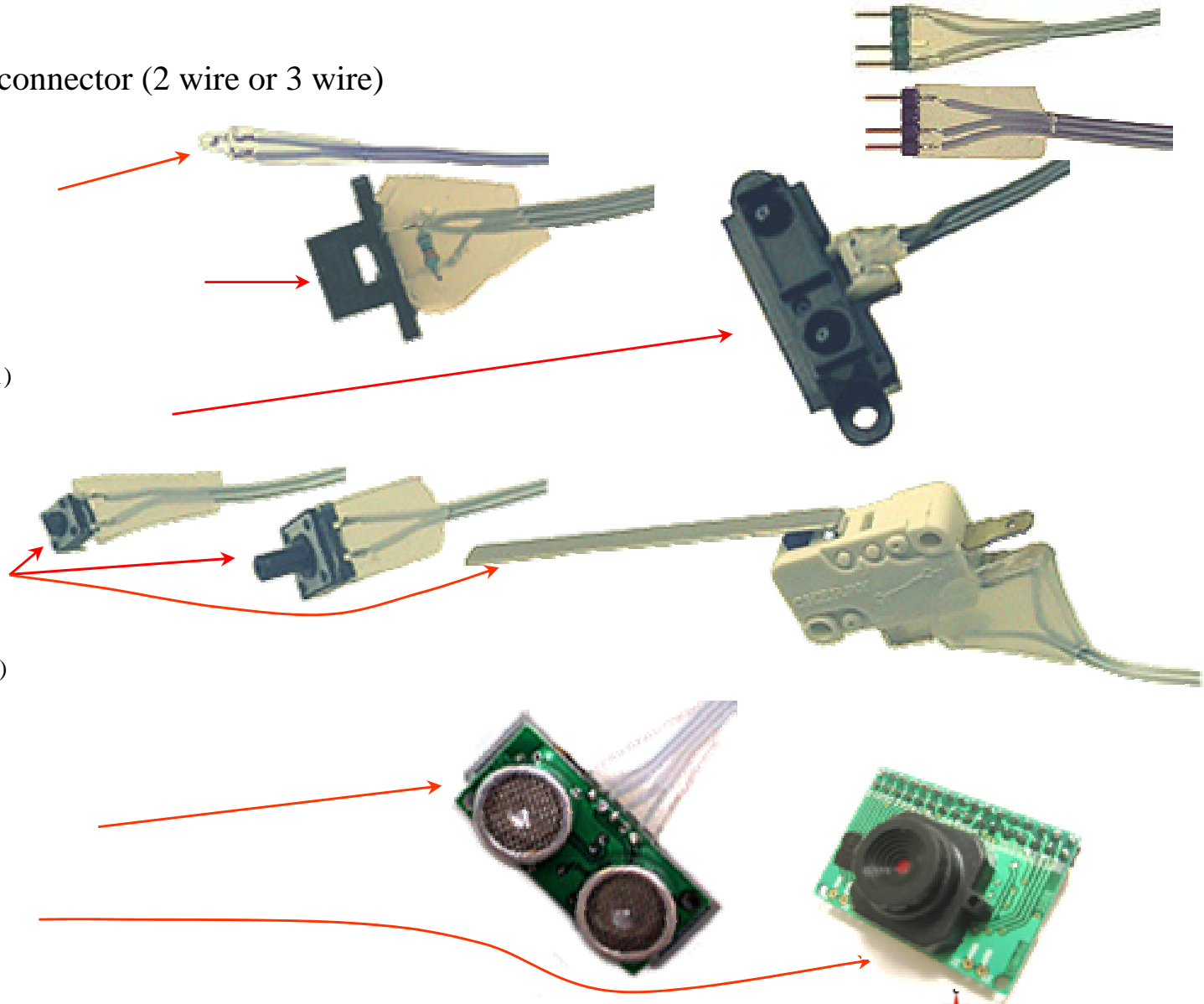
# Sensor Types

- Digital:
  - Return a 0 or a 1
  - Switches or bumpers are an example (open: 0, or closed: 1)
- Analog:
  - Sensor returns a continuum of values
  - Processor digitizes results (8 bits give values of 0-255)
  - XBC also has 12 bit analog functions
  - e.g. light sensors




# Detachable Sensors

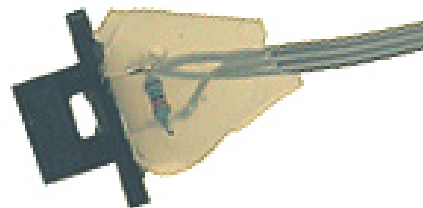
- Analog sensors:
  - Light (ports 2-6)
  - IR reflectance (ports 2-6)
- Floating analog sensors:
  - Optical rangefinder (ports 0-1)
- Digital sensors:
  - Touch (ports 8-15)
- Special sensors:
  - Ultrasonic rangefinder (sonar)
    - (ports 8-15)
  - XBC Camera
    - (camera port on XBC)



# Light Sensors

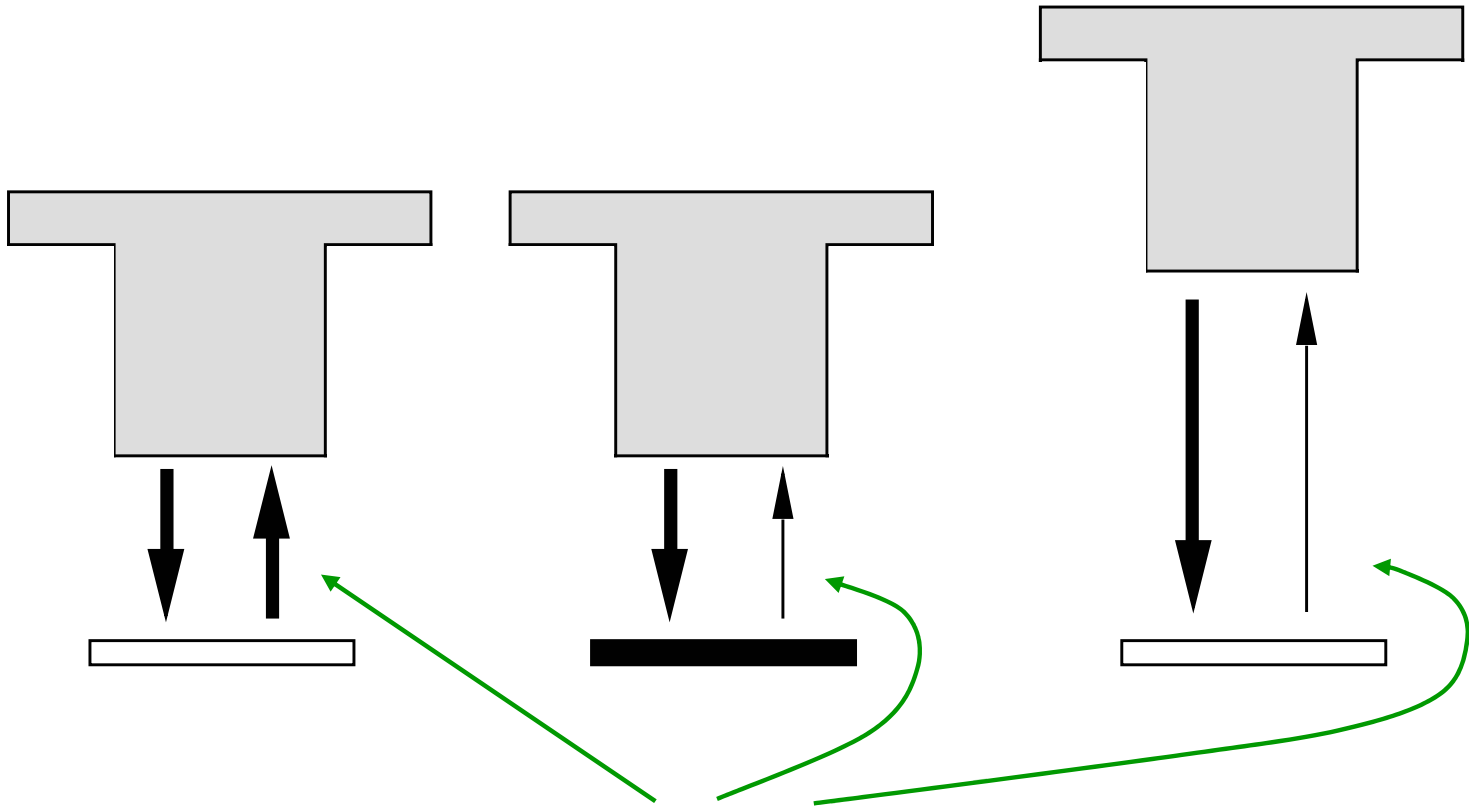
- Analog sensor 
- Connect to ports 2-6
- Access with library function `analog12(port#)`
  - You can also use `analog(port#)` for lower resolution
- Low values (near 0) indicate bright light
- High values (near 4095 for `analog12`, 255 for `analog`) indicate low light
- Sensor is somewhat directional 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 2-6
- Access with library function `analog12(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 (4095) indicate low light, dark color, or distance of several inches
- Sensor has a reflectance range of about 3 inches

# IR Reflectance Sensors

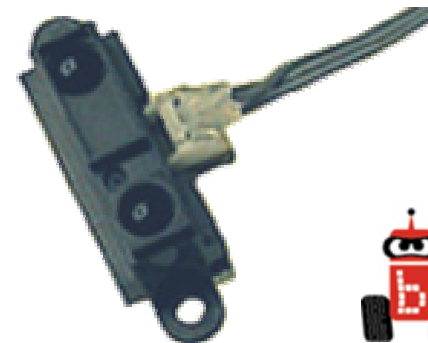


Amount of reflected IR depends on surface texture, color, and distance to surface

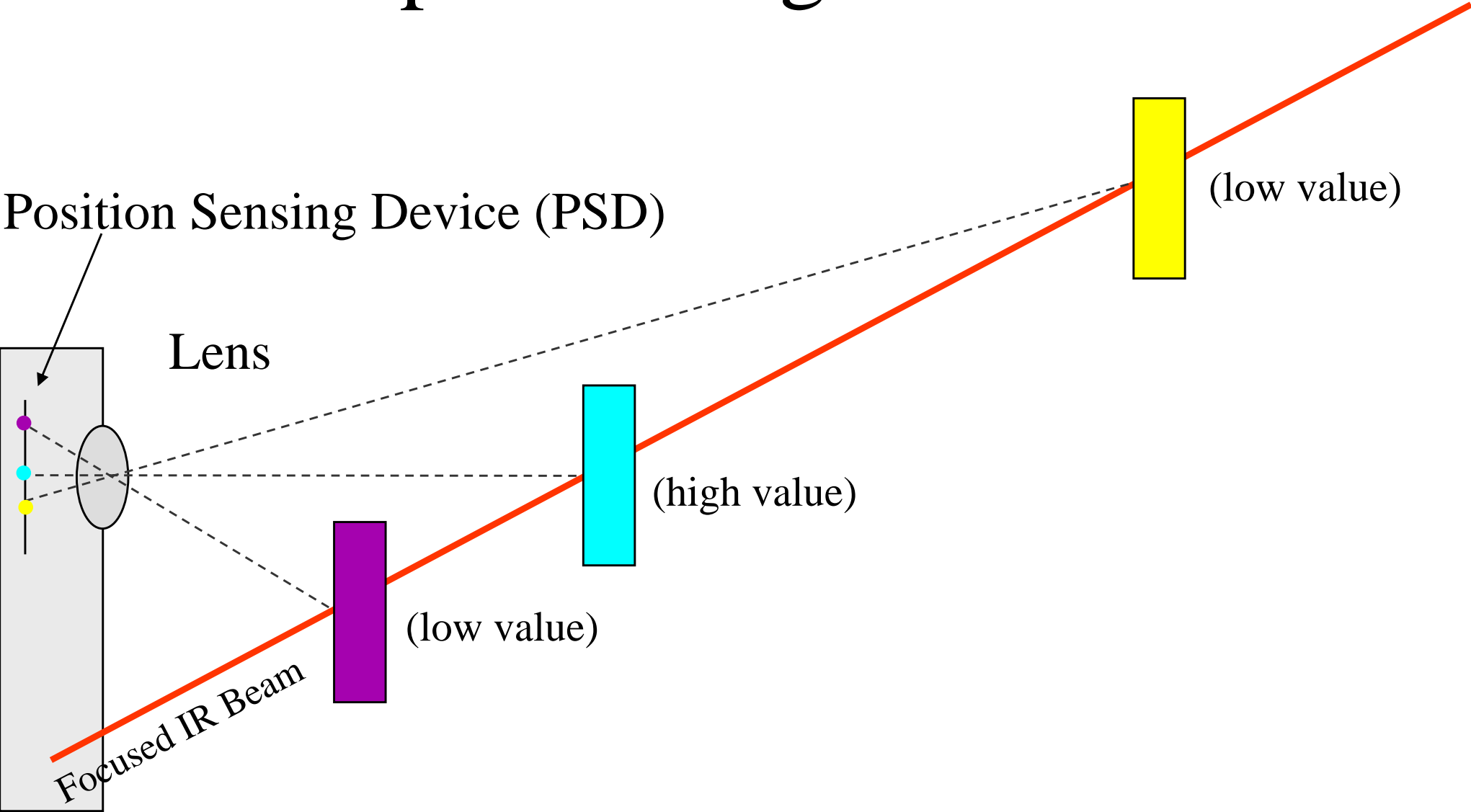


# Optical Rangefinder “ET”

- Floating analog sensor
- Connect to ports 0-1
- Access with library function `analog12(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)

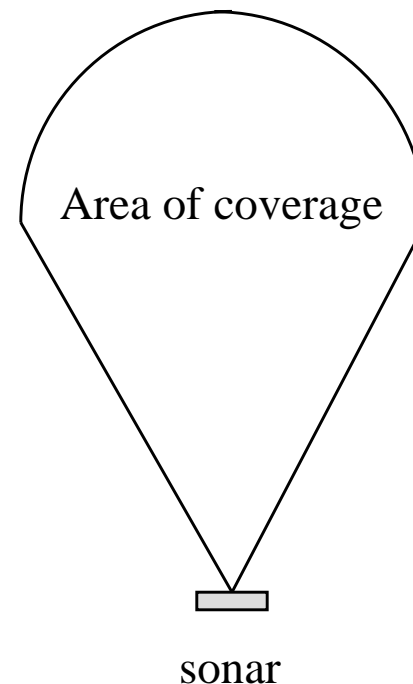
- Timed analog sensor
- Connect: port 8-15
- Access with library function **sonar** (*port#*)
- Returned value is distance in mm to closest object in field of view
- Range is approximately 30-2000mm
- No return (because objects are too close or too far) gives value of 32767





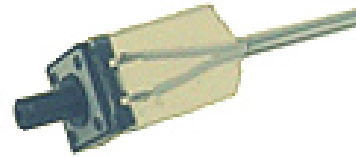
# Ultrasonic Sensors

- 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 a 30° (3-dim) teardrop



# Touch Sensors

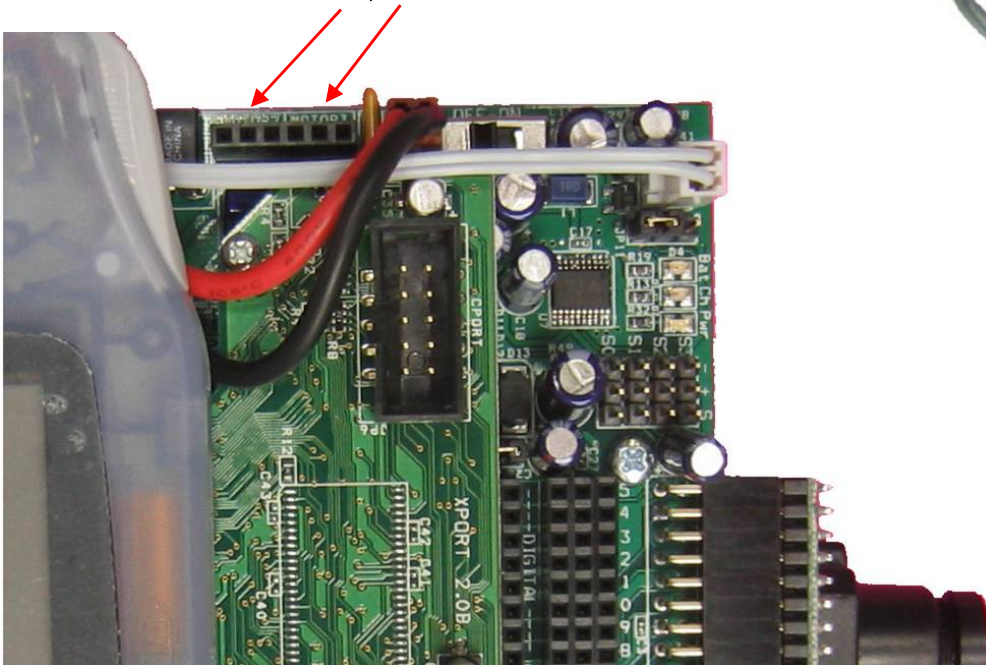
- Digital sensor
- Connect to ports 8-15
- Access with library function **digital**(*port#*)
- Three form factors in kit
- 1 indicates switch is closed
- 0 indicates switch is open
- These make good bumpers and can be used for limit switches on an actuator



# Motors & Servos

# DC Motors

- DC motors (gray cables with 2 prong plugs) plug into the XBC motor ports
- The XBC has 4 motor ports number 0,1,2,3
  - 0, 1 are on the right side
  - 2, 3 are on the left side



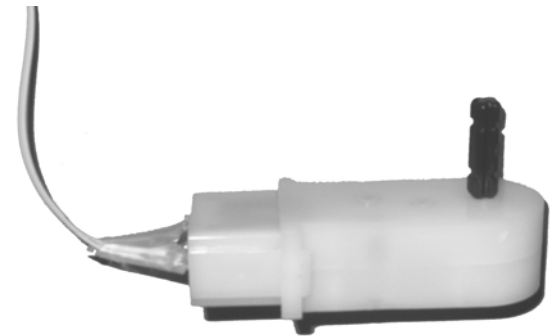
# DC Motors

- DC motors with gray cable plug directly into XBC motor ports
- If the motor position counter on the Motor XBC Status Display decreases when the motor is manually turned in the forward direction, simply flip the plug 180 degrees to correct
- The black motors have a high stall torque of about 48 in-oz
  - Each black motor has a servo horn mounted to the motor shaft. Different servo horns can be attached by removing the screw in the motor shaft and lifting off the old servo horn.
  - Lego pieces can be attached to the servo horns using either glue 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

- These motors have a LEGO axle for mounting wheels or gears
- The white gear motors are a little faster but have a little less torque than the black gear motors
- The silver motors have no built in gears. They can turn very fast, but have low torque.
- Use LEGO gear trains to trade speed for torque



# Ferrite Bead Installation on White DC Motors



# EMI & CPU Problems

- The white Botball motors do not have EMI shielding the way the other kit motors do
- It is possible that electrical and magnetic pulses from these motors could cause data corruption in an XBC that was very close to the motors
- In extreme cases, this could cause an XBC to crash when the motor is in use
- Ferrite beads greatly reduce EMI interference caused noisy sources such as unshielded motors
- The following slides show how to install the ferrite beads on your white motors





# Parts

- Gather your two white motors and two ferrite beads



# Install the Bead

1. Slip the bead over the motor connector
2. Then loop the connector through a second time
3. Pull the wire snugly against the bead
4. Repeat for the other Motor.
5. In the unlikely event that you still have EMI problems in the future, pull the wire through for another loop

1



2



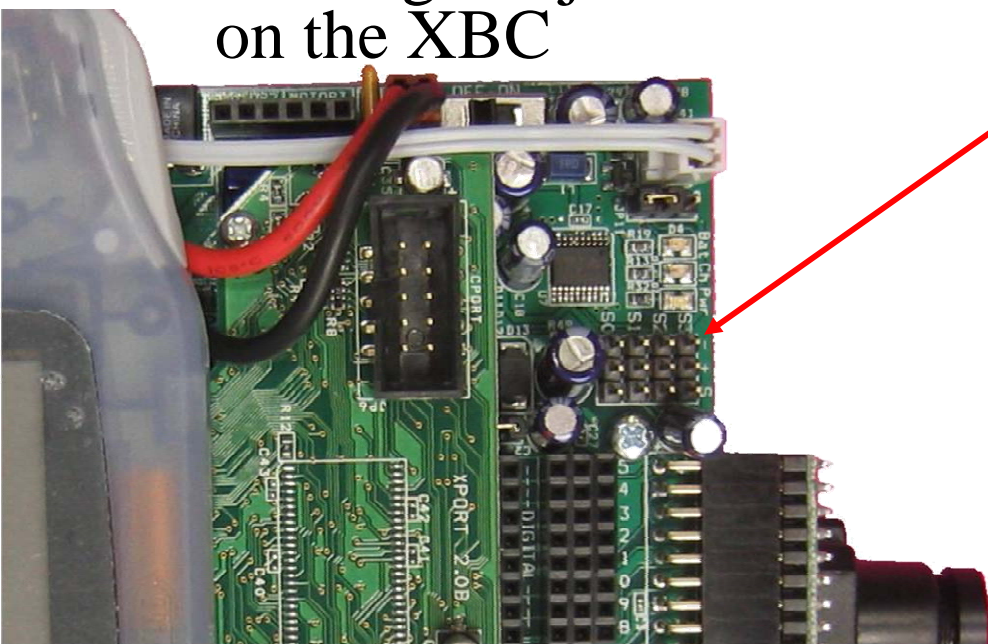
3



# Servo Motors

# Servos

- Servo motors (black-red-yellow cables with 3 prong receptacle) plug into the XBC servo ports
- The XBC has 4 servo ports numbered 0,1,2,3
- Plug-in order is yellow, red, black with black toward the right when looking into the camera
- The plugs for the servo ports are the pins sticking out just to the left of the camera on the XBC



# Position Servos

- Position 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 , 123 ) ;`
    - Sets the position for servo port 2 to 123
    - If servos are enabled, the servo in port 2 rotates to position 123
  - Position range is 0-255
  - You can preset a servo's position before enabling servos
  - Default position when servos are first enabled is 128
- `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

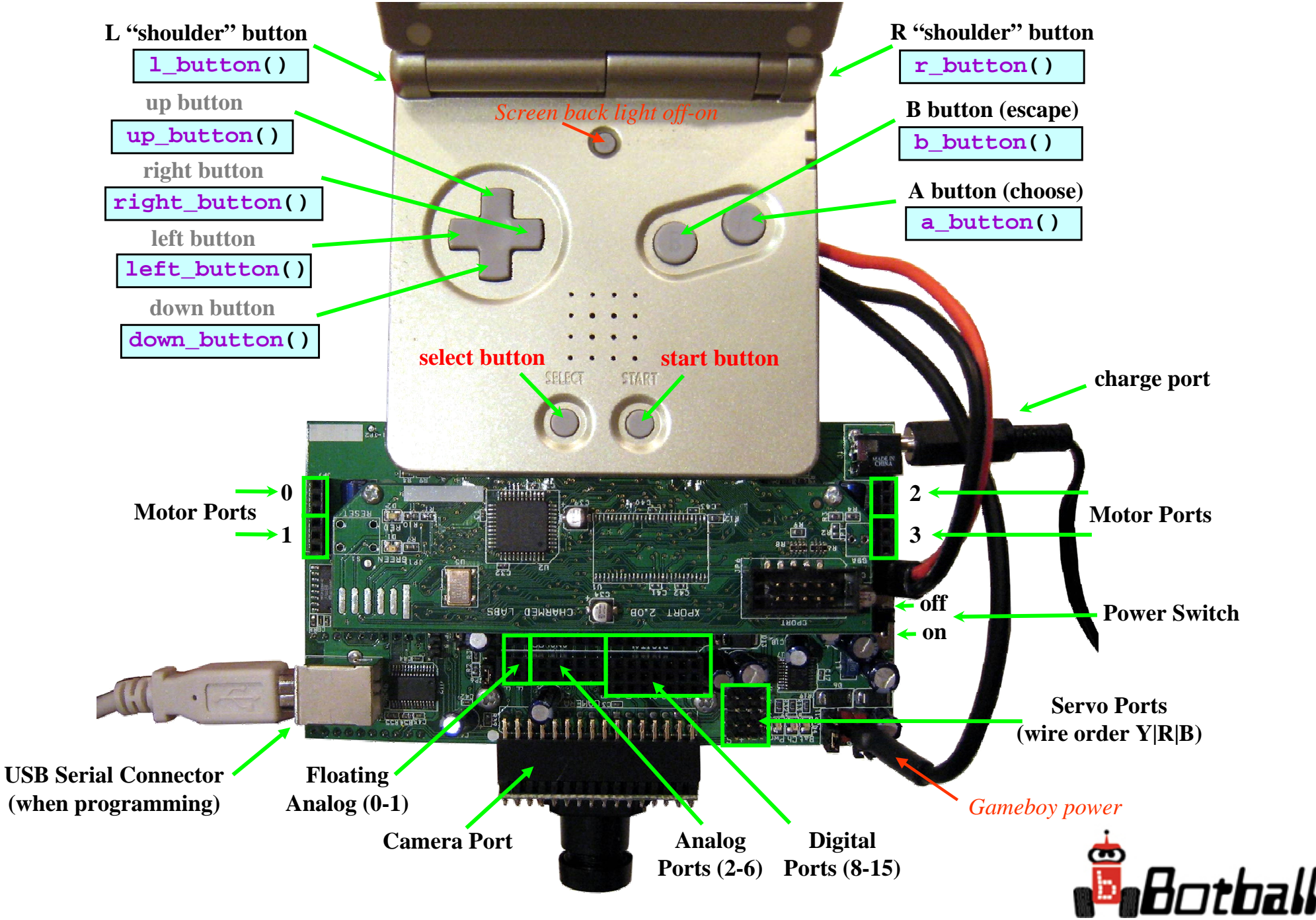
```
void main(){  
    enable_servos(); // turn servos on and  
                      // rotate to default position (128)  
    printf("moving to 64\n");  
    set_servo_position(3,64); // rotate servo 3 to 64  
    sleep(2.0); // give plenty of time for servo to move  
    printf("moving to 200\n");  
    set_servo_position(3,200); // rotate servo 3 to 200  
    sleep(2.0);  
    printf("moving to 64\n");  
    set_servo_position(3,64); // rotate servo 3 back to 64  
    sleep(2.0);  
    disable_servos(); // turn servos off  
    printf("Done\n");  
}
```



# XBC Reference Sheet



# XBC (v.3 w/GBA SP and Camera)





# Testing Motors and Sensors on the XBC



# You Can Use the Interaction Window for Testing

- For example, to test a motor
  - Plug the motor into a motor port
  - Execute **ma****v**( *<port-nmbr>*, *<vel>* ) in the interaction window for the port you have selected
  - Execute **ao**( ) to stop the motor
  - Particularly useful for setting motor plug orientation
- Or, you can use the **xbctest.ic** in **IC**'s XBC folder



# Testing Your XBC, Sensors & Motors

- Turn on your XBC and make sure the serial cable is connected
- Use settings menu in **IC** to switch controller to XBC
- Open **xbctest.ic** from **IC**'s XBC folder and download it
- Use the scroll down button to select *Run xbctest.ic* from the Gameboy screen
- Press the A button or the scroll right button to run the program
- Follow the onscreen instructions to menu through the motor, servo, sonar and vision tests
- Connect and test 4 motors (connect to ports 0, 1, 2 and 3)
- Connect and test the servos (one at a time)
- Connect (one of ports 8 through 15) and test the sonar
- Connect and test the digital sensors (look at status screen)
- Connect and test the remaining analog sensors (be sure you have the right type of sensors in the right ports) (look at the status screen)
  - Note that the status screen reports the scaled **analog** function (0-255)
- Test the vision system, focus can be adjusted by gently turning the front of the lens



# Experiments with Demobot



# Mobile Robot Experiment (1)

- Plug a touch sensor into port 8 for a front bumper
- Plug a touch sensor into port 14 for a rear bumper
- You should already have the wheel (ports 0 and 1) and gripper (port 2) DC motors plugged in, with correct polarity
- You should already have the arm/shoulder servo plugged into servo port 0 (verify polarity – black is -, red is +, yellow is S)
- Take your ping pong program from day 1 and modify it to run on this robot
  - Hint: you will only need to check for a single front and single rear sensor
  - Hint: get rid of all **rsim\_** prefixes.



# Mobile Robot Experiment (2)

- Question: is reality as accurate as simulation?
- Modify your square program from Day 1 so that it will run on the XBC and demobot
- Mark where the robot moves.
- Compare the quality and accuracy of the squares in simulation and reality



# Calibrating the Shoulder/Arm

- You should be able to raise and lower the arm freely until servos are enabled
  - In the interaction window enter
    - `set_servo_position(0,127)`
      - This instructs servo 0 to move to position 127, when the servo is enabled
    - `enable_servos()`
      - This powers on all of the servos.
      - Servo 0 moves to the last position instructed (127)
    - `set_servo_position(0,100)`
      - Try different values to determine what value best raises the arm up (no motor straining) and what value lowers it to the surface and no further
        - » Hint: use the up-arrow key to scroll the interaction window back to the previously entered command, so you only have to change the number rather than re-typing the whole command
      - Write down these values!



# Calibrating the Gripper

- Close the gripper claw by manually rotating its DC control motor
  - In the interaction window enter
    - `clear_motor_position_counter()`
      - 0 becomes the motor position for gripper being closed!
    - `mtp(2, 100)`
      - Try different values to determine what value best gives you an open gripper claw (write these down!)





# Mobile Robot Experiment (3)

- Write functions to control the arm (no peeking ahead until you've tried to do this)
- Now use these to do a task, such as
  - Move some distance
  - Turn some amount
  - Move some more
  - Open the claw and lower the arm
  - Close the claw and raise the arm
  - Return to where you started
  - Lower the arm and open the claw to release whatever you picked up



# Compare Your Solutions to These

```

int arms=0;  // arm servo in S0
// servo positions for up and down
int up=10, down=100;
int clawm=2;  // claw motor
long open=250L, closed=0L;
// claw(250L) opens claw(0L) closes

void arm_init() {
    arm(up);  // initialize to up
    enable_servos();
}
void arm(int pos) {
    set_servo_position(arms,pos);
}
// manually close claw for claw_init
void claw_init() {
    clear_motor_position_counter(clawm);
}
void claw(long pos) {
    mtp(clawm,250,pos);
}

```

*Example usage -*

```

int mL=2, mR=0;
void main() {
    display_clear(); while(a_button());
    printf("BE SURE CLAW IS CLOSED\n");
    printf("press A when ready\n");
    while(!a_button()); // wait for press
    claw_init(); // claw must be closed
    arm_init(); // pre-calibrated up/down
    move(6.0); turn(-95.0); move(6.0);
    claw(open); arm(down); sleep(2.0);
    claw(closed); sleep(2.0);
    arm(up); sleep(2.0);
    turn(180.0); move(-6.0); . . .
}

```

*(download these and try using  
arm and claw in the interaction  
window!)*



# Mobile Robot Experiment (4)

- Copy the program *color-line.ic*, from your team CD onto your hard disk
- Open the program in **IC** and then load it onto your robot
- Place the robot at one end of a strip of red tape
- Press start and follow the onscreen instructions
  - They will tell you to press the rear bumper when you are ready for it to move
- The robot should follow the red tape and stop when you press the B button, or it gets near the end of the tape. Look at the program and try and figure out how it works.



# Feedback and Control

- Bang bang control
- P-loops



# Using a Sonar in Bang-Bang

- Bang-Bang is a control scheme that uses extremes
  - If the sensor reads too close go in reverse
  - If the sensor reads too far go forward
  - In some circumstances Bang-Bang can damage gear trains, motors and motor drive electronics

– e.g.:

```
int dir = 0;  /* direction switch */
while(b_button()==0)
{
    if (sonar(15) < 300) {
        if (dir == 0) {
            mav(1,-500); mav(-500); dir = 1;
        }
    }
    else {
        if (dir == 1) {
            mav(1,500); mav(2,500); dir = 0;
        }
    }
    sleep(.03); // give sonar time between pings
}
```



# Using a Sonar in P-Loop

*P-Loop* or proportional control is a control scheme that adjusts the output proportionally to the changes in input

If the sensor reads close to the target distance move slowly. If it is far away, move quickly.

e.g.

```
int speed;
while(b_button() == 0)
{
    speed = 10 * (sonar(15) - 300);
    if (speed > 1000) speed = 1000;
    mav(1, speed); mav(2, speed);
    sleep(.03);
}
```

**Exercise:** Take this code fragment and write the rest of the **main** function to make the robot stay near distance 300.



# Exercise: Constant Distance

- Have a robot maintain a constant distance (about 25 cm) from the object (your hand) in front of it
- When your hand moves towards the robot, the robot should back away
- When your hand moves away from the robot, the robot should go forwards
- Use a p-loop to control your robot's velocity
- Question/Experiment: What happens if you move your hand suddenly very close to the bot (less than three inches from the sensor)?



# Botball Utilities

## From the XBC Library





# Utilities for Botball

- The library file *botball.ic* contains special routines to help you write your Botball programs
- In Botball robots start when signaled by a starting light and have to stop 90 seconds later
- the function **wait\_for\_light** ( *<port>* ) runs the calibration routine for the XBC with the light sensor plugged into *<port>* and then waits for the light to come on
- When using this function write:

```
#use "botball.ic"
```

at the top of your program file (before main)



# Timing for Botball

- When executed, the function  
`shut_down_in(<game_secs>);`  
starts a process that turns off all motors after *game\_secs* has elapsed and keeps any new commands from being processed until the XBC is power cycled (servo motors, if running, are turned off). You can edit this function to leave servos powered (see the file *botball.ic* in the *lib/xbc* directory).
- If you are not sure what your changes will do (or even if you are) TEST multiple times before the tournament

Note: *game\_secs* must be a **float** number (with a decimal point)

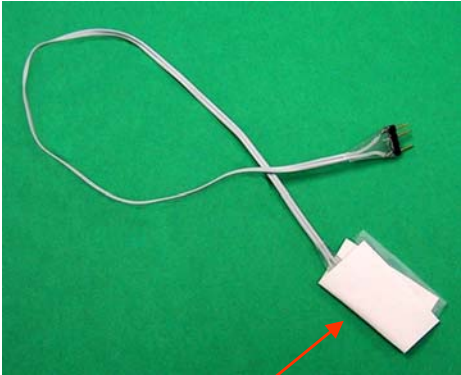


# YOU MUST SHIELD YOUR LIGHT SENSOR

- The table will be lit with 8 - 60 Watt replacement (800 lumen, 13 watt) compact fluorescent light bulbs in 9" reflectors.
- 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



No!!

Yes!

1

2

3

Wrap segment of plastic straw in tape

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

The image shows a three-step process for correctly shielding a light sensor. Step 1 shows the materials: a white cable with a three-pin connector, a roll of black tape, and a small red segment of plastic straw. Step 2 shows the straw wrapped in black tape. Step 3 shows the straw being slid over the sensor head, with the tape being secured. An orange arrow points to the straw in step 3.

# Exercise:

## Create A Botball Program

**Have your Team Do this the First Week!!!**

- Modify one of your challenge programs to use the Botball utilities:
  - Add the **#use** line to load the file botball.ic
  - Add a light sensor to your robot and use the **wait\_for\_light**( ) function to calibrate it
  - Use the **shut\_down\_in**( ) function to turn your robot off after 30 seconds
  - Remember: In the real situation, the light sensor will require shielding



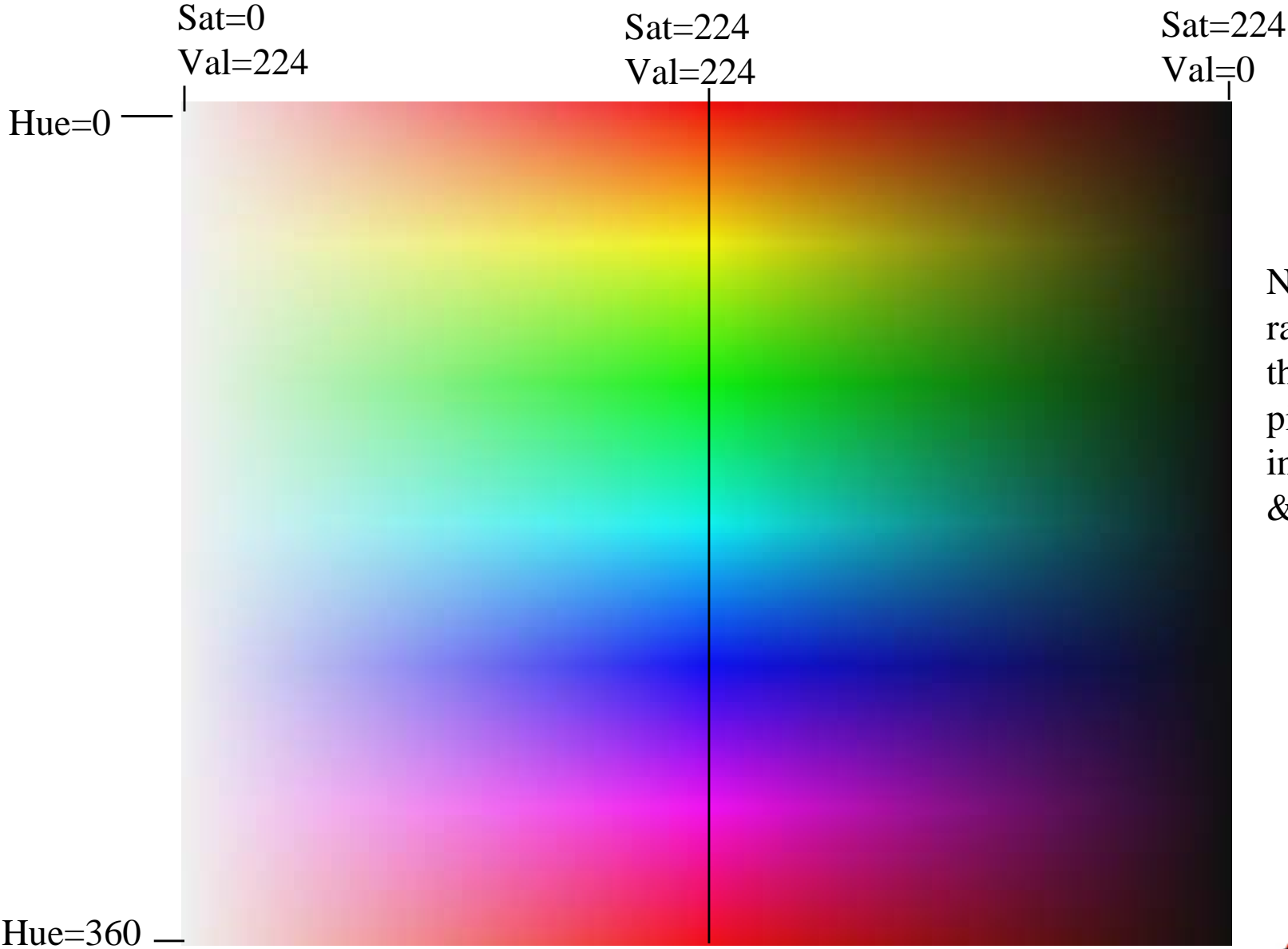
# Color Vision System

# 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
- Challenge exercises



# 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 **IC** is  $357 \times 293 = 104,601$  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 XBC can segment the image using three different pieces of the 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)
  - Processed (pixels not selected by a color model are dark)
  - Combined (processed combined with raw show you can see where the blobs are on the live video)




# More on Color Models

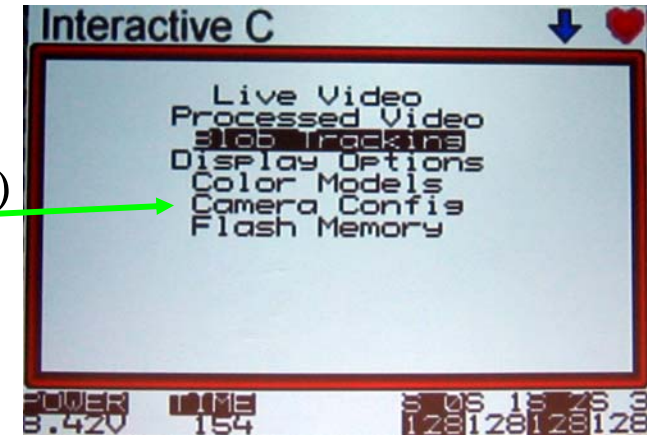
- A Color Model-HSV specifies a bounding box in the color selection plane
- Moving either edge towards the center line constrains the range of accepted color values to only include more vivid colors (i.e. only accept things that are more like *Astro Brights* paper).
- If everything you want is being 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.



# Trying Out the Vision System

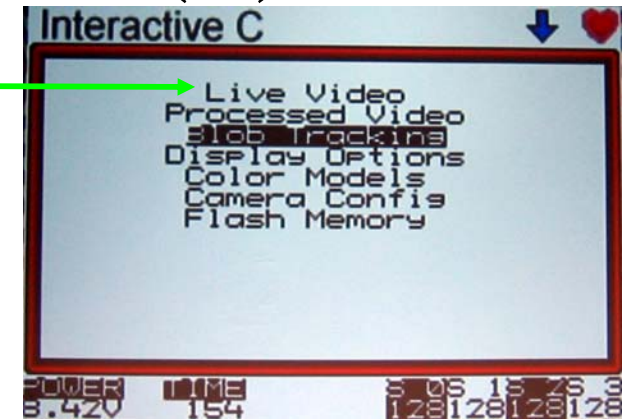
# Trying Out Color Vision (1)

1. Parts: XBC; White piece of paper; Solid colored object
2. Turn on XBC and select the *Vision* menu   
(use the pad to scroll down, then press the A button when the *Vision* menu is highlighted)
3. Select *Camera Config* and live video
  - current values will be displayed
4. Point the camera at the white piece of paper
5. Press the Start button to initiate white balance calibration
  1. “STARTING CALIBRATION” will print at the bottom of the screen
  2. After a few seconds when it says “DONE” the calibration is complete and the red/blue color temperature values are locked in.
  3. If unsatisfied, retry calibration or experiment with turning AWB (Auto White Balance) to 1 or 0 with the right and left direction pad buttons. While AWB=1 the red/blue values will react to what the camera sees, when AWB changes to 0 it locks in the red/blue values.
  4. When satisfied, press B to go back to the menu.
  5. To preserve the camera settings to still apply after reboot, select *Flash Memory* under the *Vision* menu. Scroll right until *Camera Config* appears in the setting slot. Then select *Save to Flash* and press A.



# Trying Out Color Vision (2)


1. Press the B button till you get to the menu that has *Live Video* as its top item
2. Select *Color Model* and then *Restore to Default*
3. Press B then select *Live Video* and see what the camera sees
  - The A button cycles the display through live, processed, and combined video
4. Press B and then select *Processed video* to see the image segmented
  - The A button cycles the display through processed, combined, and live video
5. Press B and then select *Blob tracking* to see how those segments are broken into blobs
  - The A button cycles the display through combined, live, and processed video with bounding boxes for blob tracking displayed
  - The color model for each bounding box is identified by the color of the corners and center cross-hairs that mark the box (green, blue, and red for 0, 1, and 2)






# Blob Tracking Views

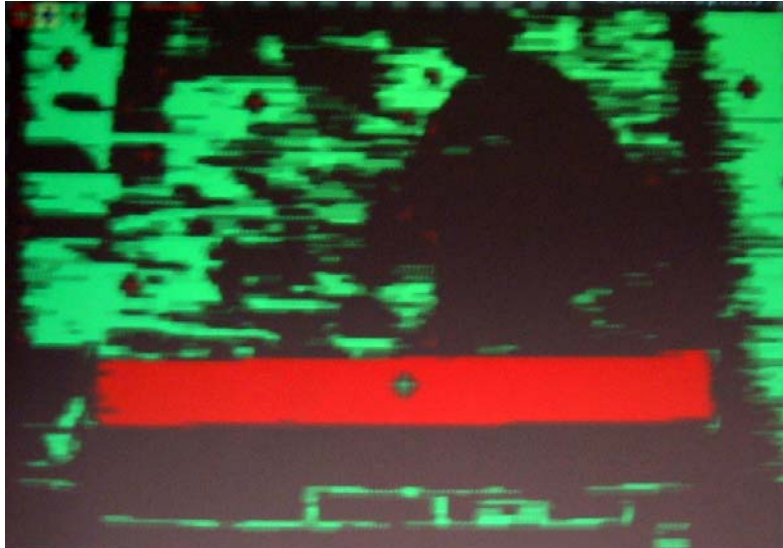
Channel Legend



Raw View




Processed View



Channel 0 Blob 0 Bounding Box

Combined View



Use the A button to cycle through the three types of views





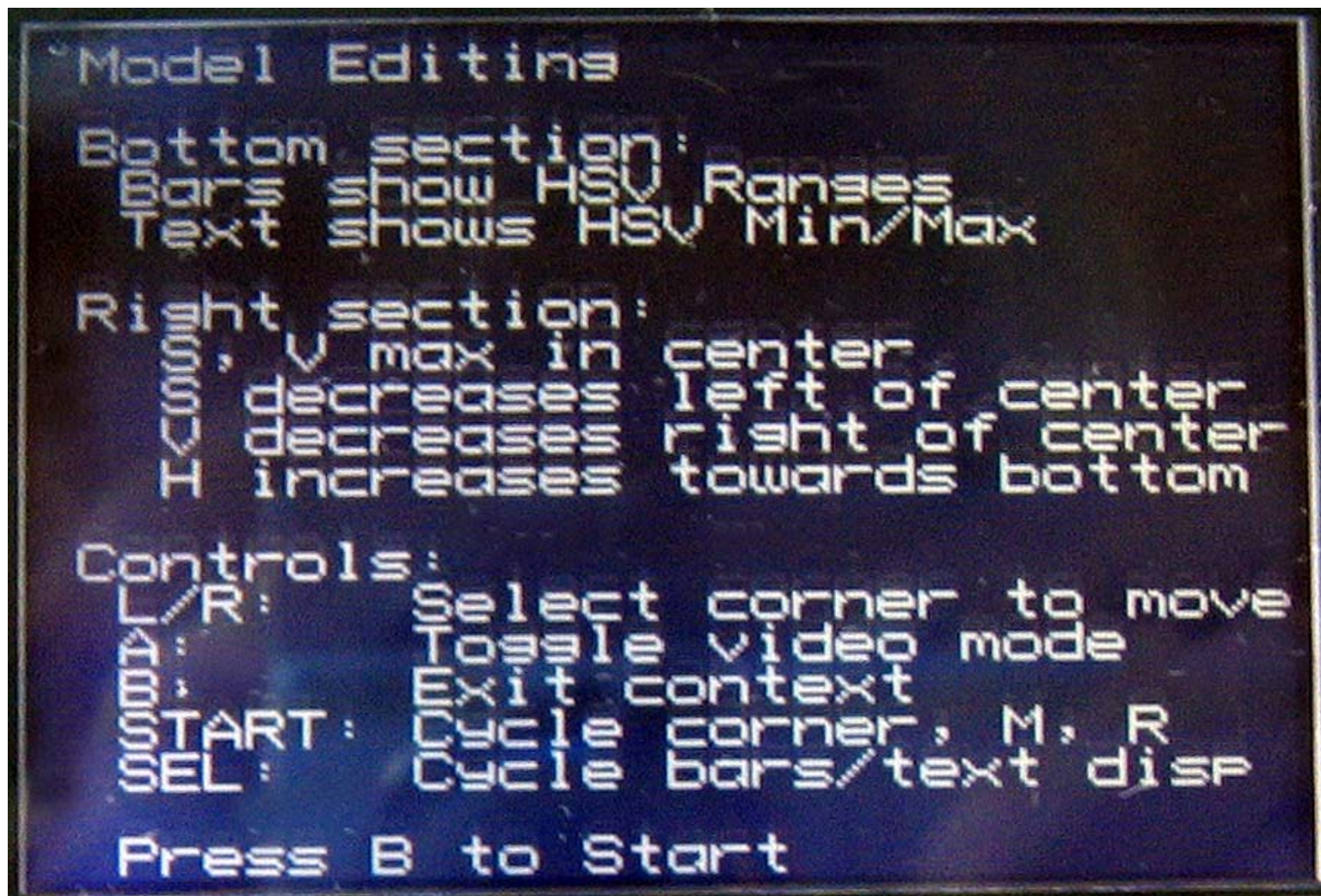
# Onscreen Instructions

## for setting color models



# Onscreen Instructions

for setting color models



# Trying Out Color Vision (3)

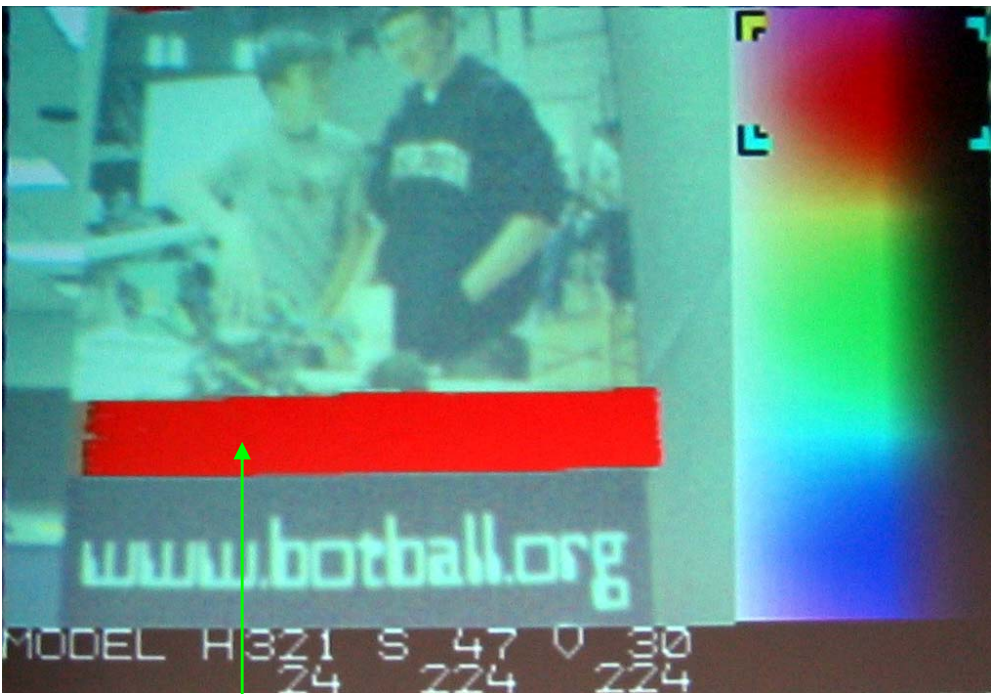
1. Press the B button and select *Color Model* and then *Modify Model 0*
2. Follow the onscreen instructions to modify the color model:
  1. The start button chooses symmetric **M**ove or **R**esize modes for the box
    - indicated by M or R in upper right corner of the display
  2. L & R buttons switch you to corner (upper left or lower right) move mode
    - indicated by upper left or lower right corner highlighted
    - only one side of the box moves, depending on the corner selected
  3. The D-pad is used to **M**ove the box, **R**esize the box, or move the corners
3. The A button cycles through live, processed, or combined video
4. Do training
  1. Open up the S and V ranges by moving the side edges outwards
  2. Open up the top and bottom edges as far as they go (MAX\_HRANGE),
  3. Move the whole range up and down until it includes what you want to accept.
  4. Then close down the top and bottom edges until they're as close together as they go before cutting out part of what you want to keep.
  5. After you have good top and bottom settings, start moving the side edges closer to the center until you have cut out everything you want to get rid of.



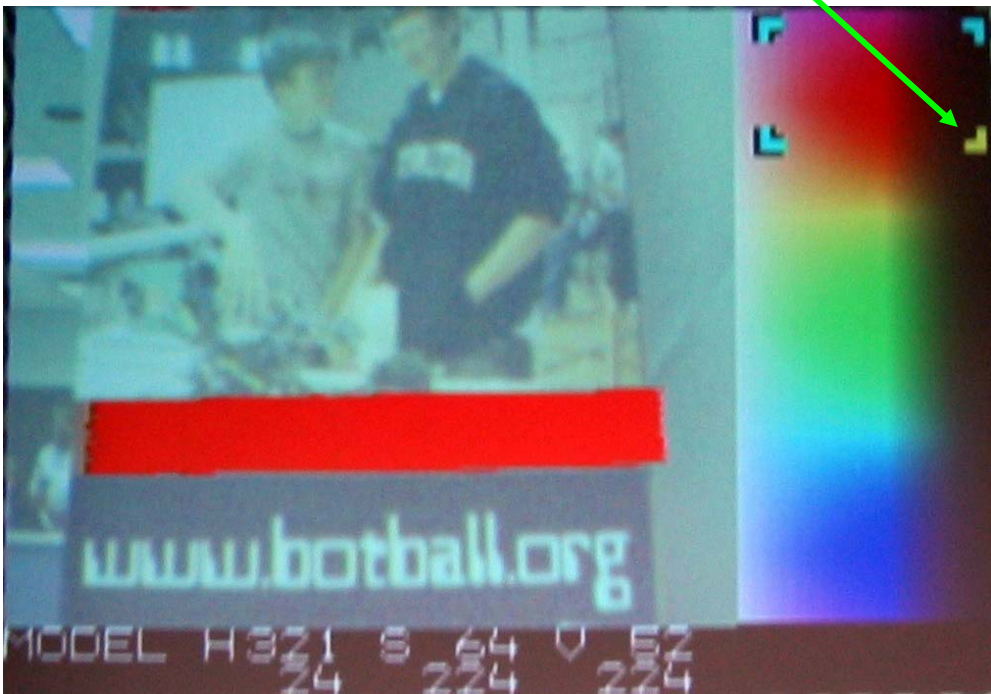


# Corner Modes

- L mode
  - Selected by L button
  - Directional pad moves left side of bounding box to left or right and top side up or down
  - Identified by highlighted upper left corner



- R mode
  - Selected by R button
  - Directional pad moves right side of bounding box to left or right and bottom side up or down
  - Identified by highlighted lower right corner



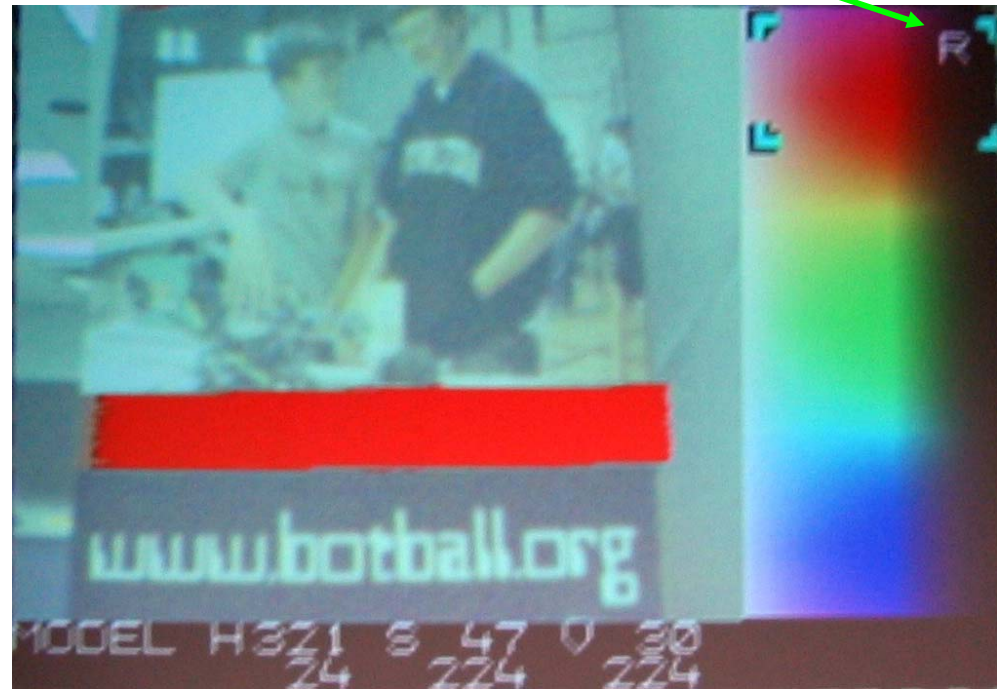
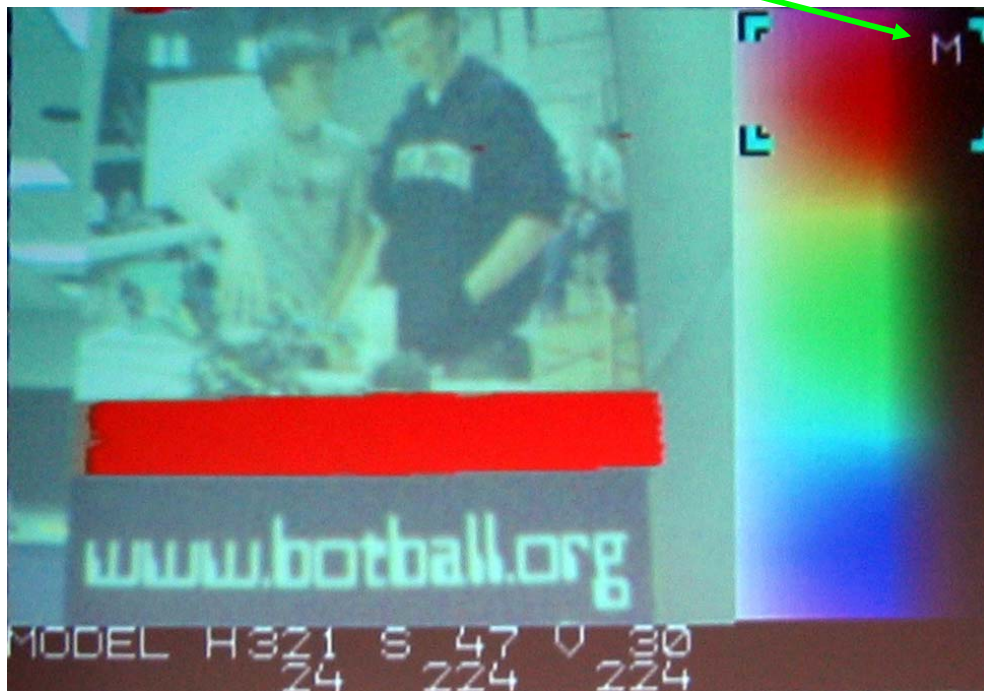
Pixels being selected  
by this model

*(note: this is a composite view; use the A button to cycle the display through Raw, Processed, & Composite views when setting the color model)*



# Move and Resize Modes

- Move mode (M)
  - Selected by Start button (toggles between M and R)
  - Directional pad moves whole bounding box to left or right and the whole box up or down
  - Identified by M on the display
- Resize mode (R)
  - Selected by Start button (toggles between M and R)
  - Directional pad shrinks the bounding box inward or outward either horizontally or vertically
  - Identified by R on the display



*(note: this is a composite view; use the A button to cycle the display through Raw, Processed, & Composite views when setting the color model)*

# Camera Check

# XBC Camera Check

1. Create a color model for channel 0 that sees a red screw driver
2. Load **xbctest.ic** onto your XBC
3. Run the program
4. Select the vision test
5. Select the correct channel/model to see red
6. Follow on screen directions to get data on the blobs
7. If you like your model, save it to flash
8. For more info, see XBC Camera in **IC** Help





# Which Color Model?

- The XBC stores 3 channels of color (0, 1, 2)
- There are 3 models maintained for each channel:
  1. The currently active model (the one you see in the *modify model* menu)
  2. The model in flash (you can save the currently active model and camera settings using the *save to flash* menu item or make them active by loading from flash)
  3. The default model and camera settings which are made active by selecting *restore to default*





# Vision Library Functions

# Loading the Vision Library

- The vision library is a separate library in the */lib/xbc* folder named *xbccamlib.ic*
- If you haven't changed the default file by saving to another location, the vision library is loaded by including in your program

```
#use "xbccamlib.ic"
```

- Otherwise you will need to include the full file path to the library for `#use` to find it



# Initializing the Camera

- The camera functions will not work until your program initializes the camera by calling the `init_camera( )` function
  - If your program is using the camera, you usually put this early in `main( )`



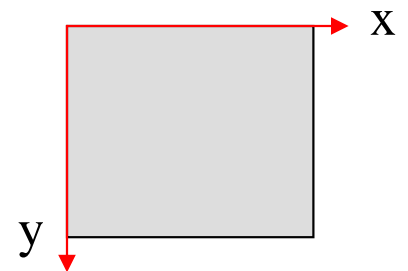
# Vision Functions

- 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 XBC
- There are vision functions for using a program to access or alter the camera configuration (see **IC** help)
  - For most purposes, you can accomplish all the camera configuration that is necessary using the vision interface accessed from the XBC menu display (i.e., using these functions is for advanced users!)



# Color Tracking

- There are three channels numbered 0, 1, 2 for processing color models
  - In the original configuration the models are set up to roughly track red/orange, yellow, and green
  - “cooler” colors such as green and blue are hard for the camera to track
  - 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 (356,292)
    - The XBC display does not show the camera’s full field of view



# Color Tracking Functions

- To keep the camera data from overwhelming the XBC processor, current camera tracking data is captured only when the **track\_update** ( ) function is called
  - Normally called prior to using other tracking functions
- 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)=(356,292) 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
- There are additional tracking functions for reducing the processor load by turning off unused channels, etc (see **IC** help)



# *color-line.ic* Program

```
#use "xbccamlib.ic"
// follow colored line, color in channel 0; stop if correct color not in view
void main() {
    init_camera(); display_clear(); // initialize camera, clear display
    printf("Point robot at red line\nPress rear bumper when ready\n ");
    while(digital(14)==0) {} // wait for rear bumper press
    display_clear();
    while(b_button()==0) { // continue until B button pressed
        track_update(); // update info from camera queue
        // show (channel 0, blob 0) data: x position, y position; also blob count
        display_set_xy(0,0); printf(" x=%d\n y=%d\n count=%d\n",
                                   track_x(0, 0), track_y(0, 0), track_count(0));
        if (track_count(0)>0) { // are there any blobs on channel 0?
            // if largest (blob 0) is centered (x=176) move at 400 ticks/sec
            // if off center, turn using speed proportional to 5*offset
            // from blob's centroid x position (given by track_x(0,0))
            mav(3, 400+5*(track_x(0,0)-176)); // if blob is left slow m 3 (left)
            mav(1, 400+5*(176-track_x(0,0))); // if the blob is right slow m 1 (right)
        }
        else { // there are no blobs being detected on channel 0!
            ao(); // stop (until someone extends the line)
            printf("Can't see the red line\n");
            sleep(0.1); display_clear(); // sleep to get fresh data; clear the display
        }
    }
    ao(); beep(); // stop motors, sound horn
    display_clear(); printf("Program Exit.\n");
}
```





# Challenge Exercises

## Vision



# Vision Challenges

- The next three slides have challenges for you to implement.
- They are arranged in approximate order of increasing difficulty
- All programs should start with `wait_for_light` and timeout after a set time
- You will need to add a light sensor to your robot to activate the robot with `wait_for_light`
  - You can cover the sensor with your hand when the light is supposed to be off



# Challenge 1

- Modify the example “color-line.ic” program (from the CD) so that when your bot gets to/near the end of the line it will turn and then follow the line back to the start
- Break the problem into a **main** and two helper functions
  - One function follows a line until it cannot see it any longer
  - The other function turns around until it sees the line while facing in the other direction



# Challenge 2

- Plug a servo into the XBC and mount a servo horn that looks like a pointer (or tape a pencil to the round horn already on the servo)
- put the servo next to the camera
- Write a program that has the pointer point to the red screwdriver and “follows” the screwdriver as it moves back and forth in front of the camera

# Challenge 3

- Red light/Green light: Have the robot move forward when it sees green and stop when it sees red/orange
- Extra challenge: go slow (or speed up if that is how you behave) while it sees yellow
- Use the colored paper from your registration envelope (but do not destroy that paper -- you'll need it later)

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



# Template for Tournament Code

```
/* load the Botball utilities */
#use "botball.ic"
/* Botball.ic includes the functions: wait_for_light(int port)
   and shut_down_in(float seconds)
*/
void main()
{
    wait_for_light(2);    // light sensor in port 2
    shut_down_in(89.9);   // kill the robot after 89.9 seconds
    my_botball_program(); // call the function that does everything
}

void my_botball_program()
{
    /* my botball robot just beeps every 3 seconds */
    while(b_button()==0){
        beep();
        sleep(3.0);
    }
}
```



# YOU MUST SHIELD YOUR LIGHT SENSOR

- The table will be lit with 8 - 60 Watt replacement (800 lumen, 13 watt) compact fluorescent light bulbs in 9" reflectors.
- 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.

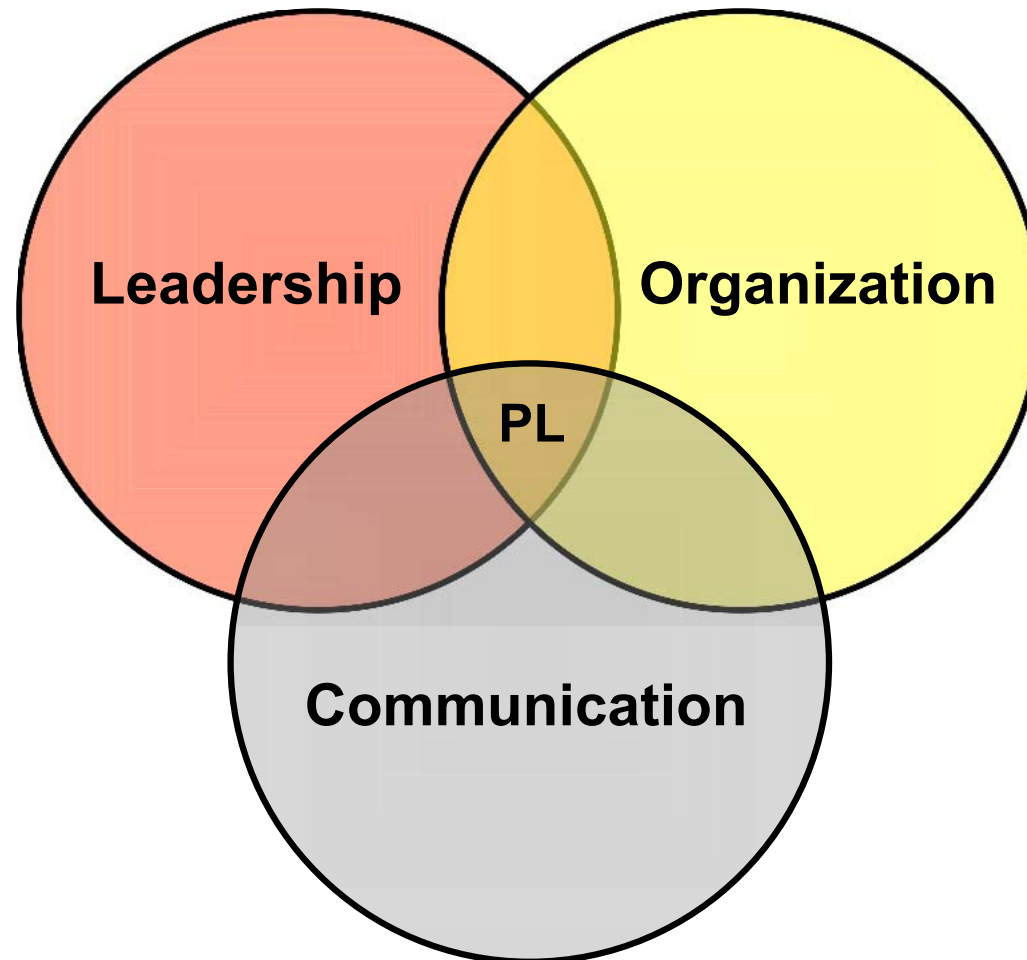




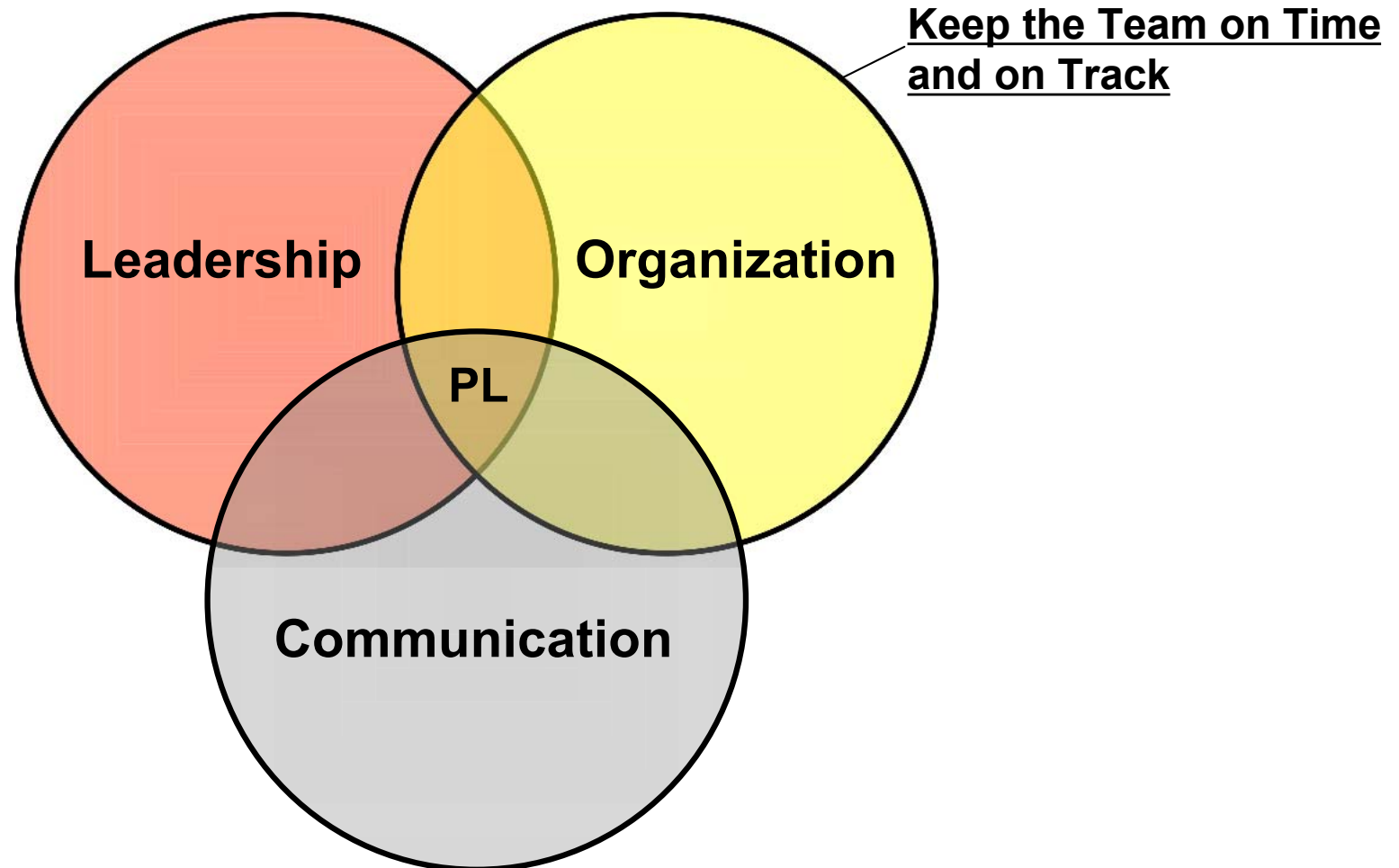
# Botball is an R&D Project



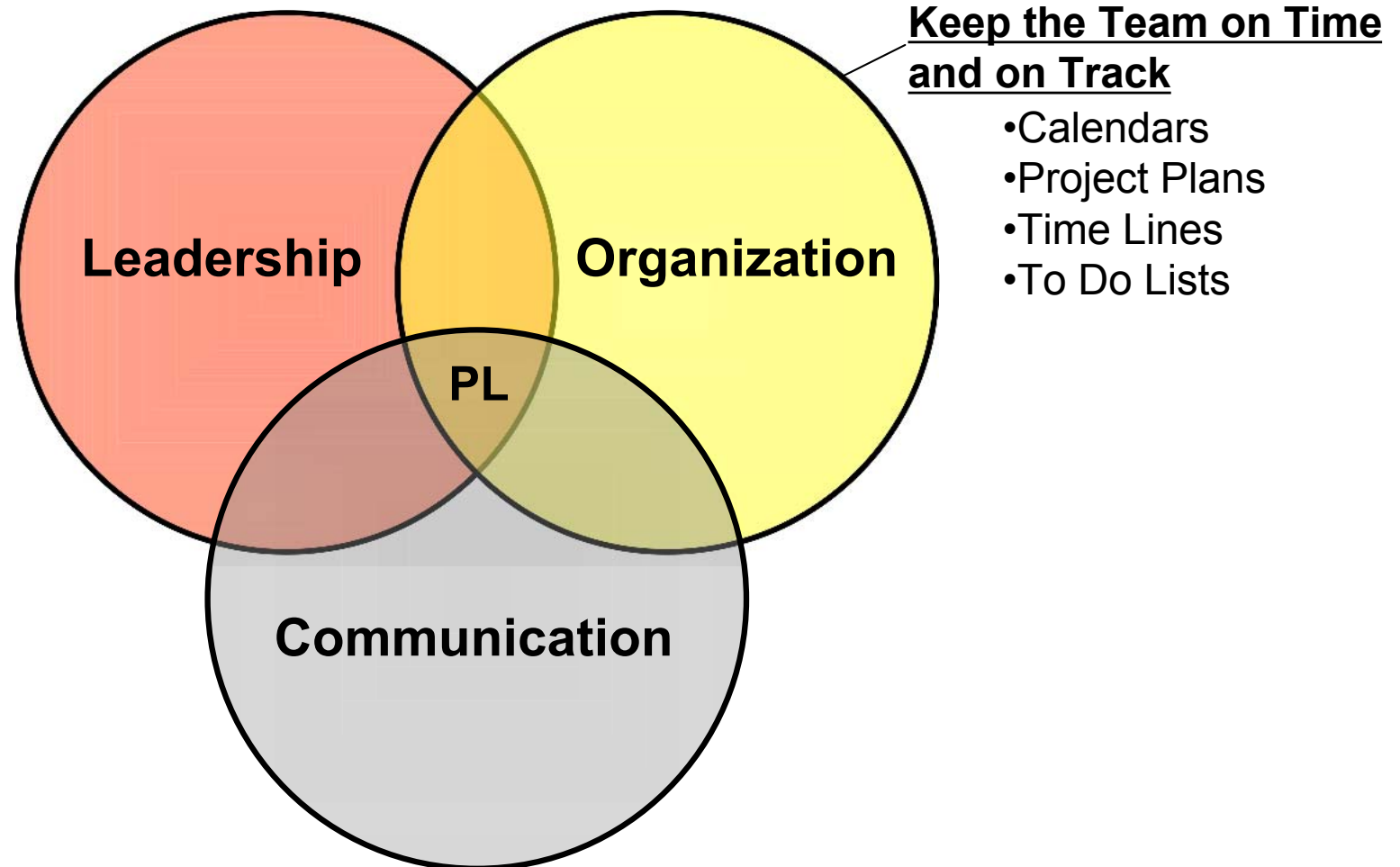
# Project Leadership



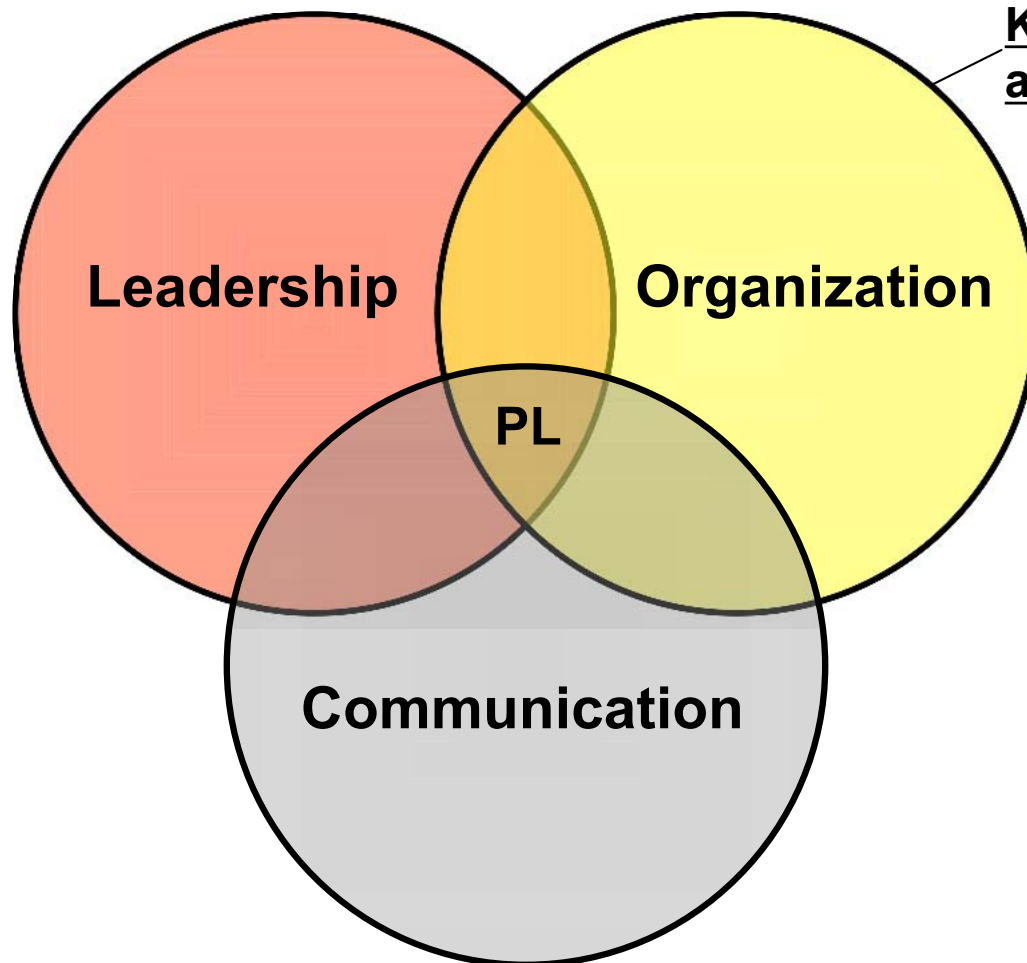
# Project Leadership



# Project Leadership



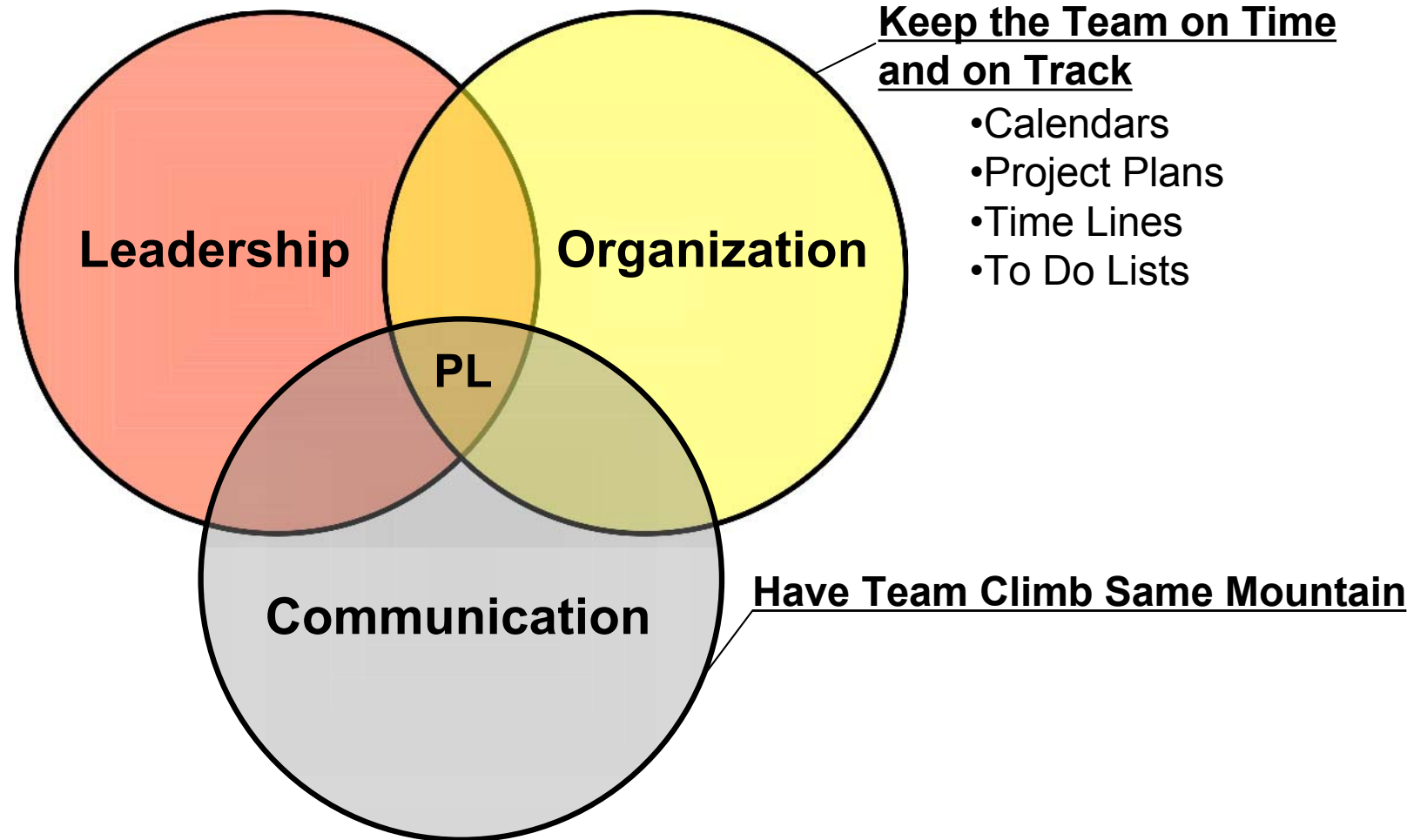
# Project Leadership



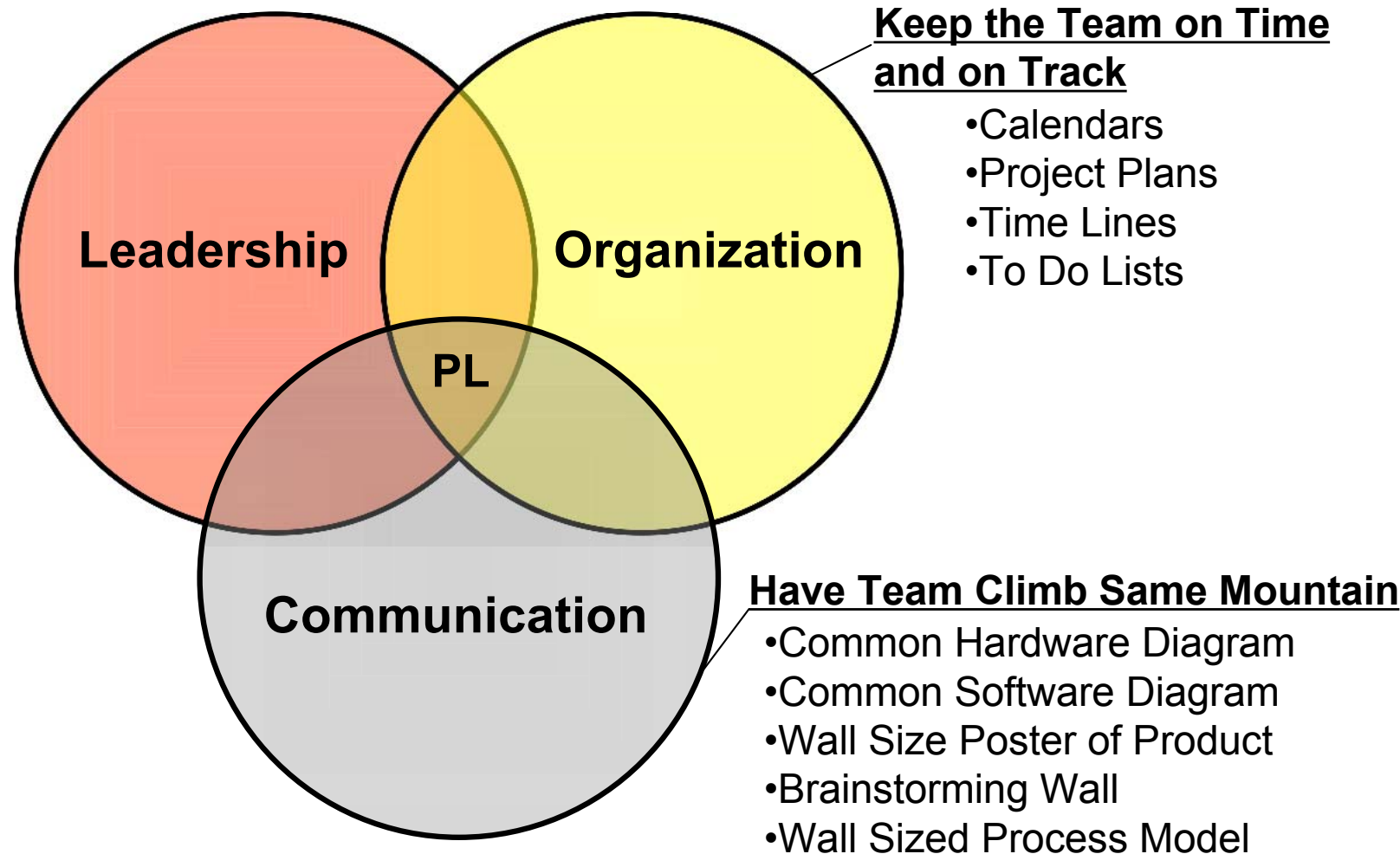
## Keep the Team on Time and on Track

- Calendars -Post publicly in team space.
- Project Plans – 1 or 2 names per task.
- Time Lines – Post large in team space.
- To Do Lists – One per team mate.

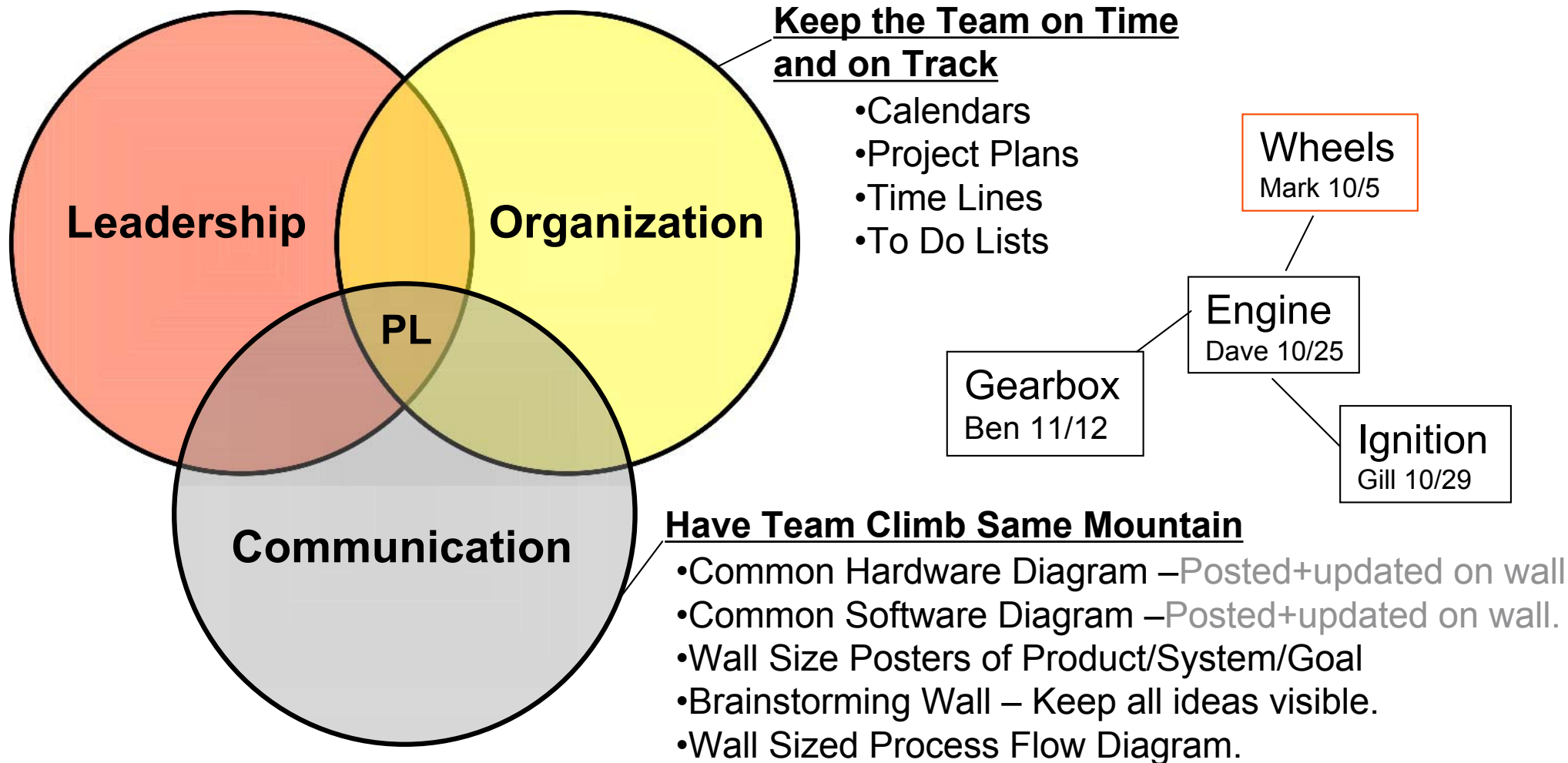
# Project Leadership



# Project Leadership

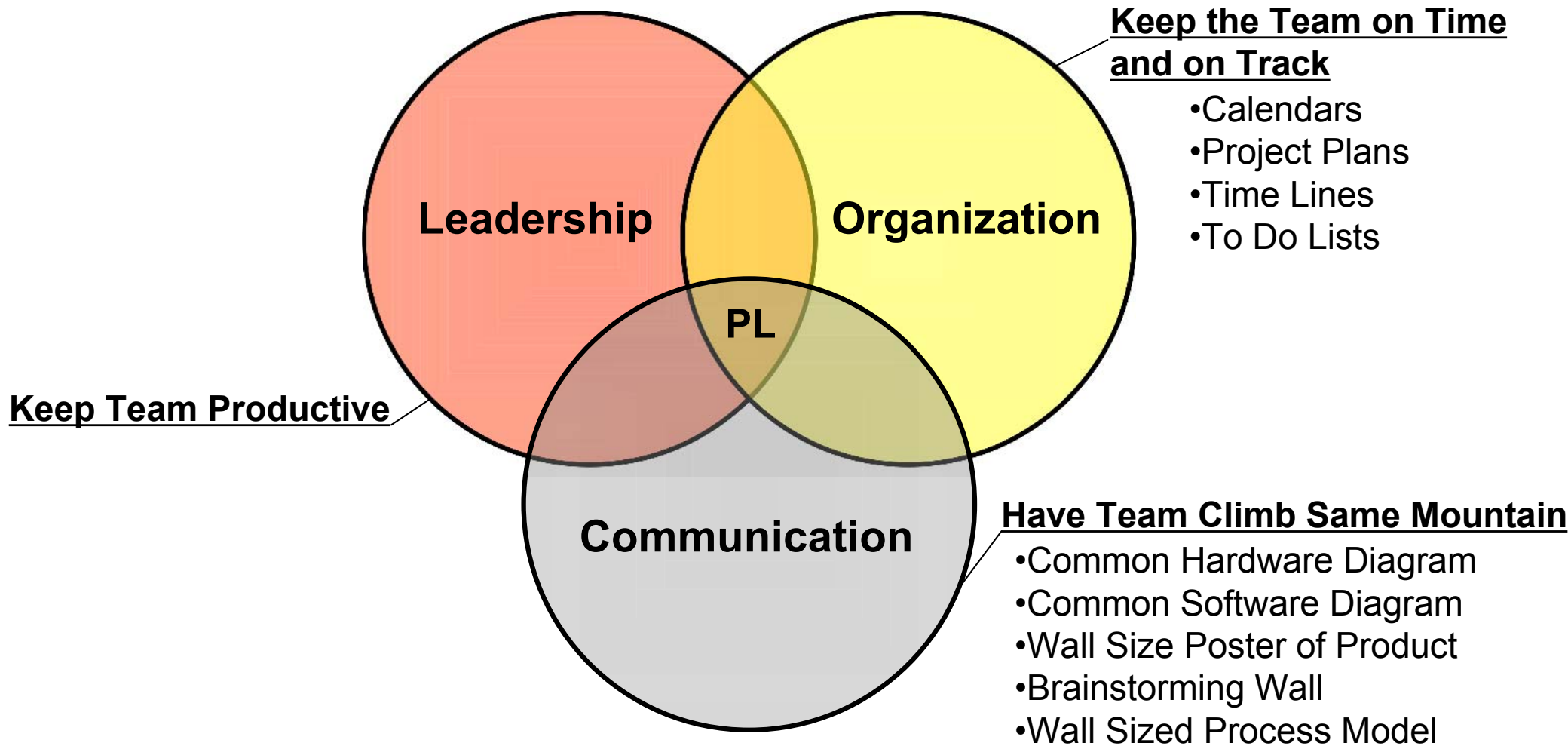


# Project Leadership

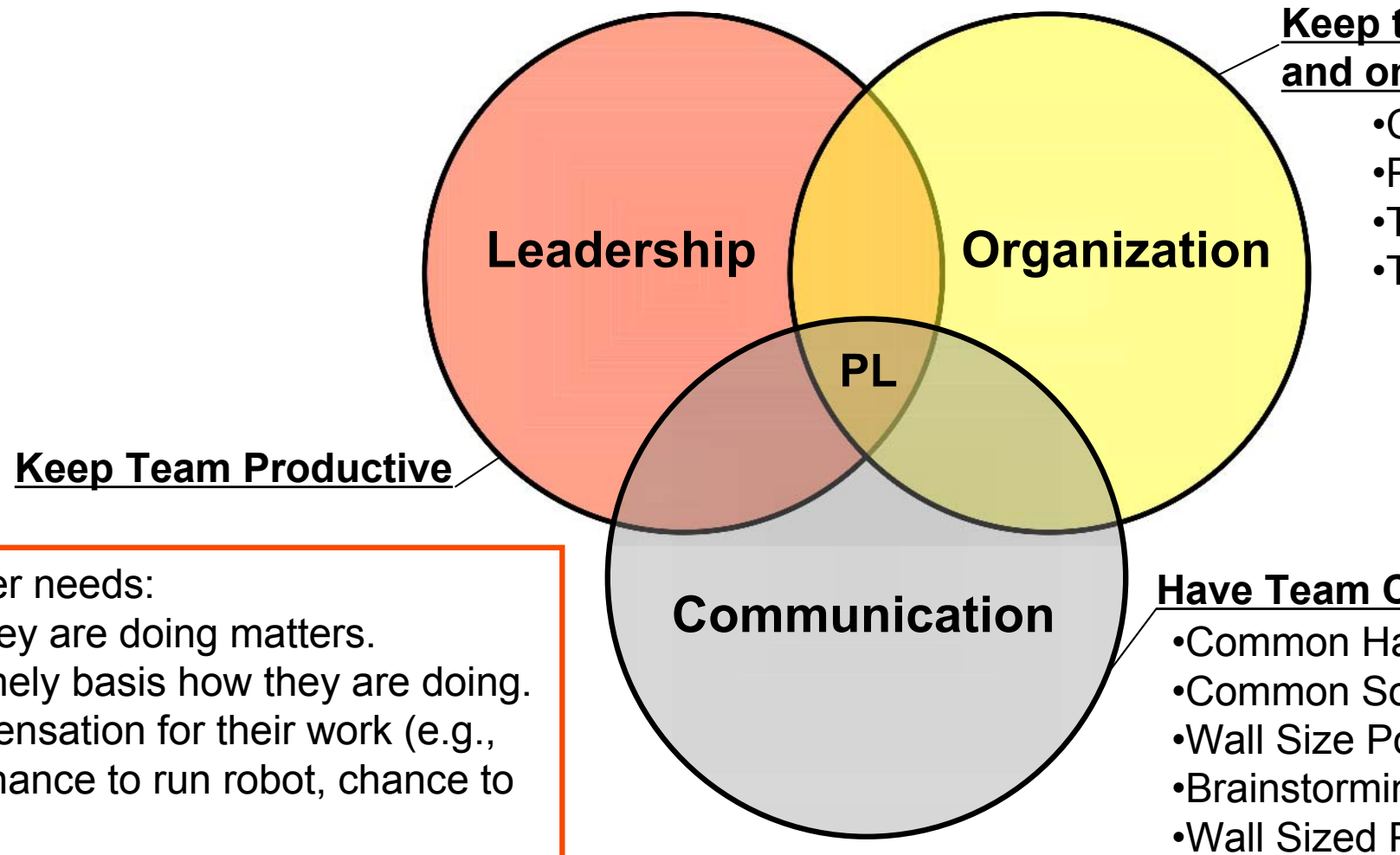




# Project Leadership



# Project Leadership



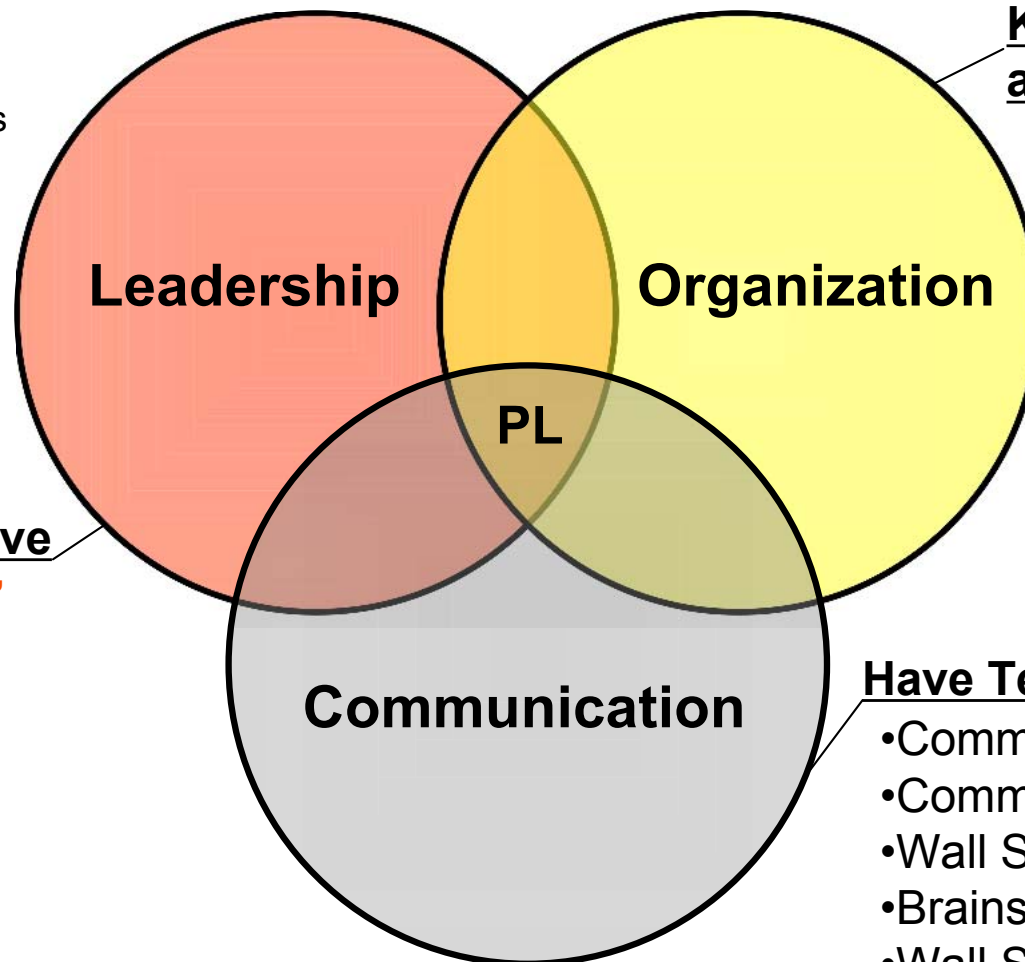
Each Team member needs:

1. To know what they are doing matters.
2. To know on a timely basis how they are doing.
3. To get fair compensation for their work (e.g., credit attribution, chance to run robot, chance to attend NCER, etc.)

# Project Leadership

Each Team member needs:

1. To know what they are doing matters.
2. To know on a timely basis how they are doing.
3. To get fair compensation for their work.



**Keep the Team on Time and on Track**

- Calendars
- Project Plans
- Time Lines
- To Do Lists

**Keep Team Productive**  
“Lead Don’t Manage”

**Have Team Climb Same Mountain**

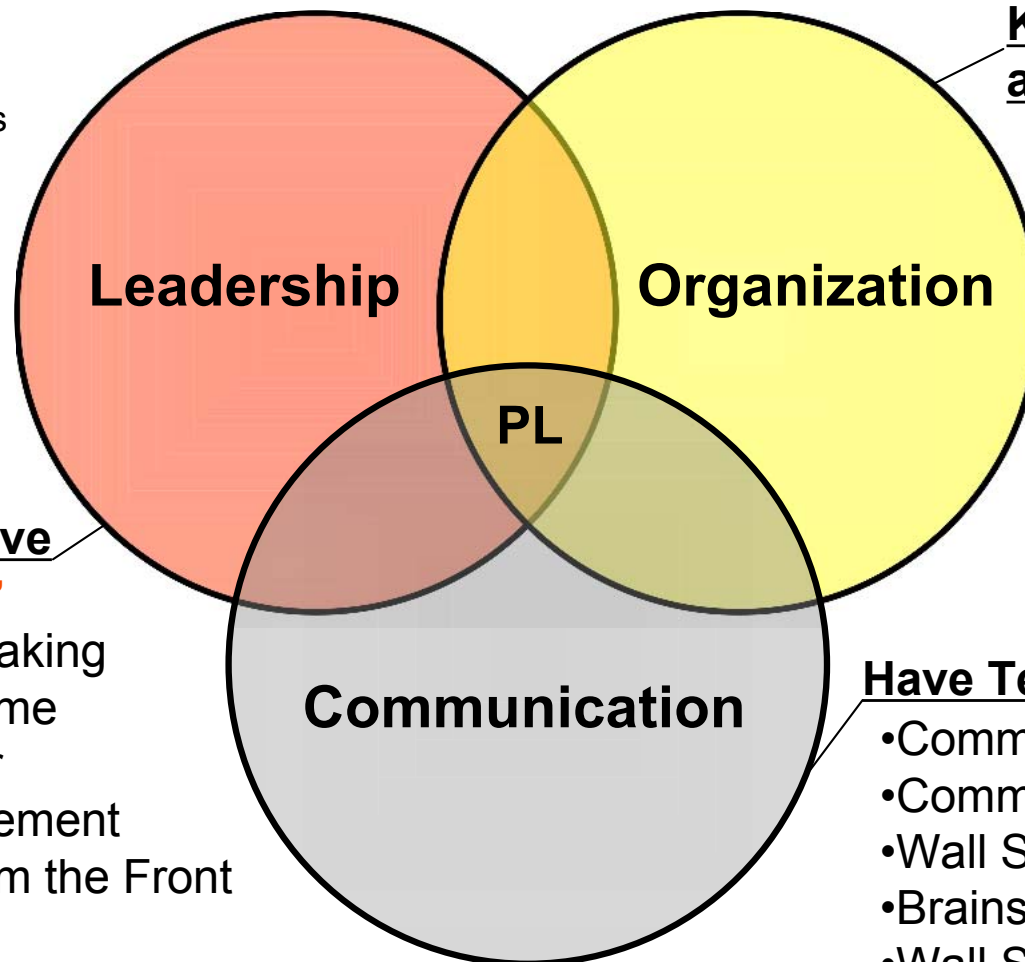
- Common Hardware Diagram
- Common Software Diagram
- Wall Size Poster of Product
- Brainstorming Wall
- Wall Sized Process Model



# Project Leadership

Each Team member needs:

1. To know what they are doing matters.
2. To know on a timely basis how they are doing.
3. To get fair compensation for their work.



## Keep the Team on Time and on Track

- Calendars
- Project Plans
- Time Lines
- To Do Lists

## Keep Team Productive

### **“Lead Don’t Manage”**

- Delegate Decision Making
- Provide Tools and Time
- Hold Team to Deliver
- Call in Upper Management
- Lead by Example from the Front

## Have Team Climb Same Mountain

- Common Hardware Diagram
- Common Software Diagram
- Wall Size Poster of Product
- Brainstorming Wall
- Wall Sized Process Model



# Project Leadership

Each Team member needs:

1. To know what they are doing matters.
2. To know on a timely basis how they are doing.
3. To get fair compensation for their work.

## Keep Team Productive

### **“Lead Don’t Manage”**

- Delegate decision making to the teammate closest to the work to be done.

**(\*\*Don’t Micro-Manage\*\*)**

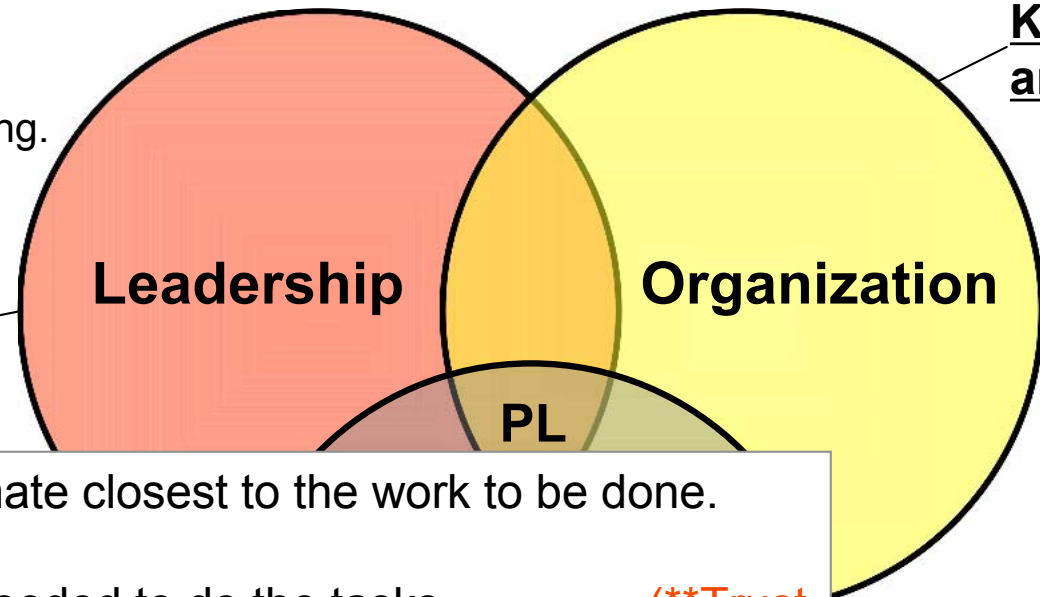
- Provide Team with the tools and time needed to do the tasks.

**(\*\*Trust them to do it right\*\*)**

- Hold Team (via honor and integrity) to deliver what they promise, when they promise it. **(\*\*Treat Team as Peer Collaborators\*\*)**

- Call in Upper Management (Faculty Advisor, mentor) for help early. **(\*\* Don’t try and solve every problem yourself & Don’t hide problems \*\*\*)**

- Lead by example from the front. **(\*\*Don’t punish from the rear\*\*\*)**



**Keep the**  
**and on T**

- Cal
- Pro
- Tim
- To

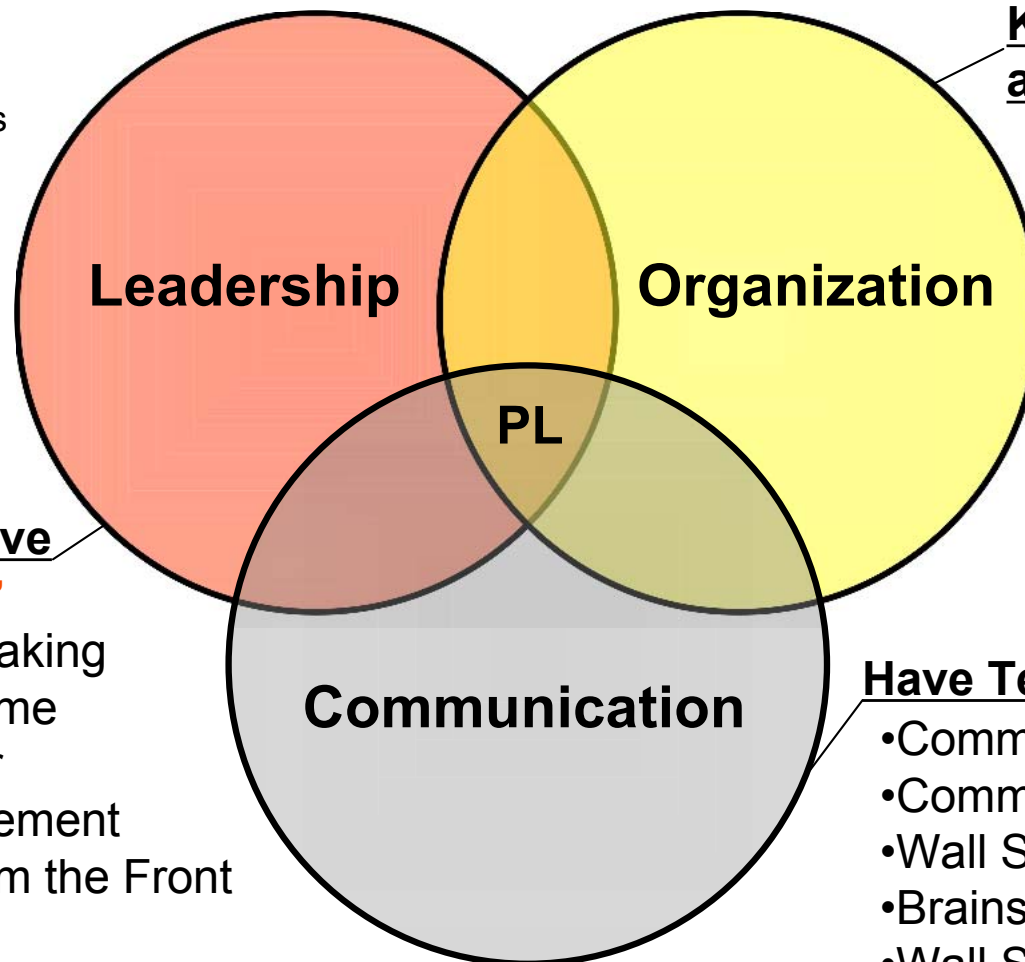
## **Have Team Clir**

- Common Hard
- Common Softw
- Wall Size Post
- Brainstorming
- Wall Sized Pro

# Project Leadership

Each Team member needs:

1. To know what they are doing matters.
2. To know on a timely basis how they are doing.
3. To get fair compensation for their work.



## Keep the Team on Time and on Track

- Calendars
- Project Plans
- Time Lines
- To Do Lists

## Keep Team Productive

### **“Lead Don’t Manage”**

- Delegate Decision Making
- Provide Tools and Time
- Hold Team to Deliver
- Call in Upper Management
- Lead by Example from the Front

## Have Team Climb Same Mountain

- Common Hardware Diagram
- Common Software Diagram
- Wall Size Poster of Product
- Brainstorming Wall
- Wall Sized Process Model



# File History & Comments

## A Way to Promote Communication

```
/* simple.ic - (c) 2003 David Miller, KIPR */

/* History:
    Modified 6/15/03 - shortened the initial comments &
    changed the word program to function
    in the printf text - dpm
    Modified 6/16/03 - removed silly comments at end - dpm
*/
/* This program displays a simple character string
*/
void main()
{
    printf("This is a C function\n"); // display the string
}
```



# Now Add 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

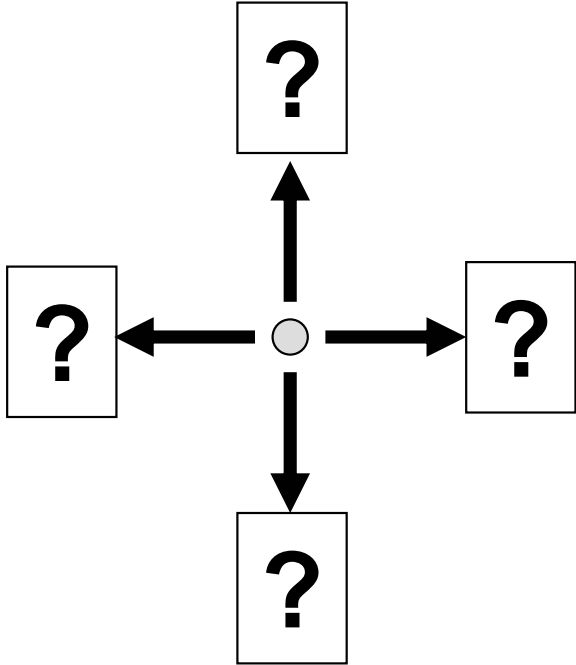




# Start with a task



# What are the requirements?



**start when the light goes on**

**turn off all motors at the end of the round**

**don't go onto the other side for the first 30 seconds**



**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.
      1. 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

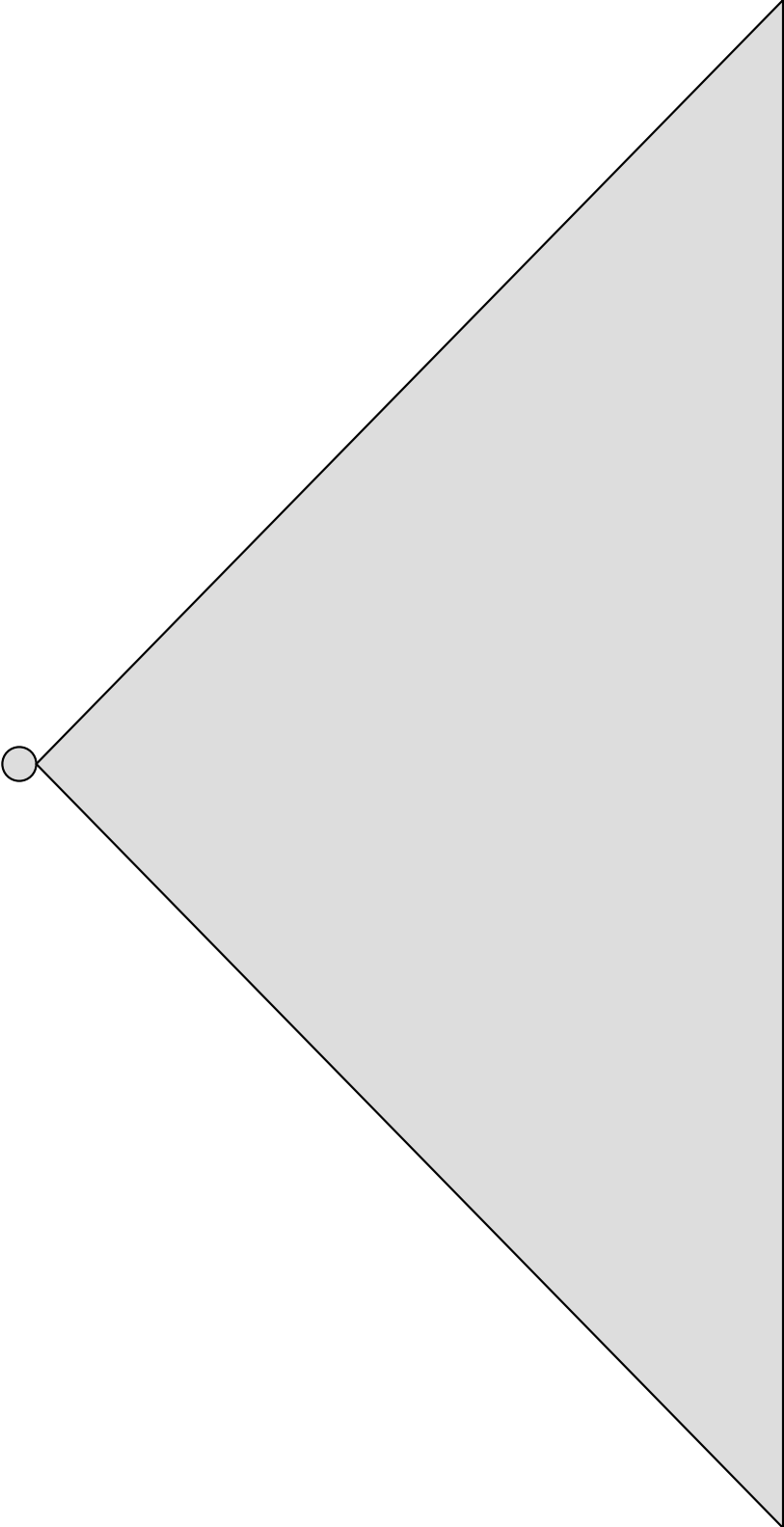
1. formulate the task as a question “How can we ....?”
2. take a few minutes in silence to individually write down ideas
3. keep your ideas short and snappy
4. each person reads out one idea
5. no criticism of ideas is allowed - reserve judgment - crazy ideas welcome (THIS REQUIRES DISCIPLINE)
6. build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





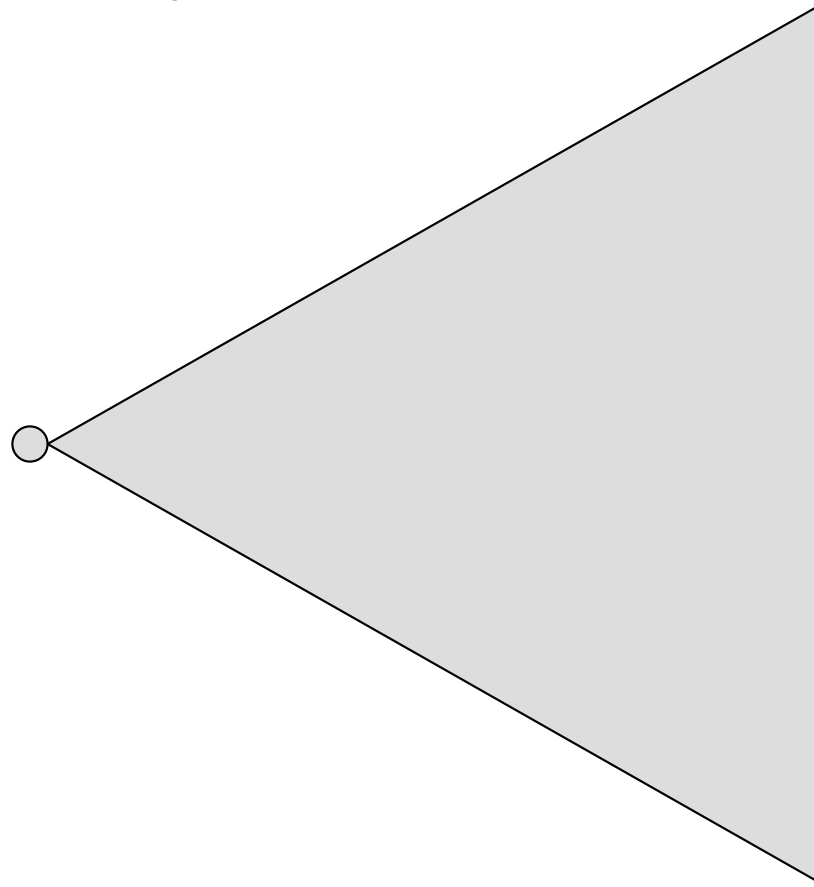
# Evaluation

- 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

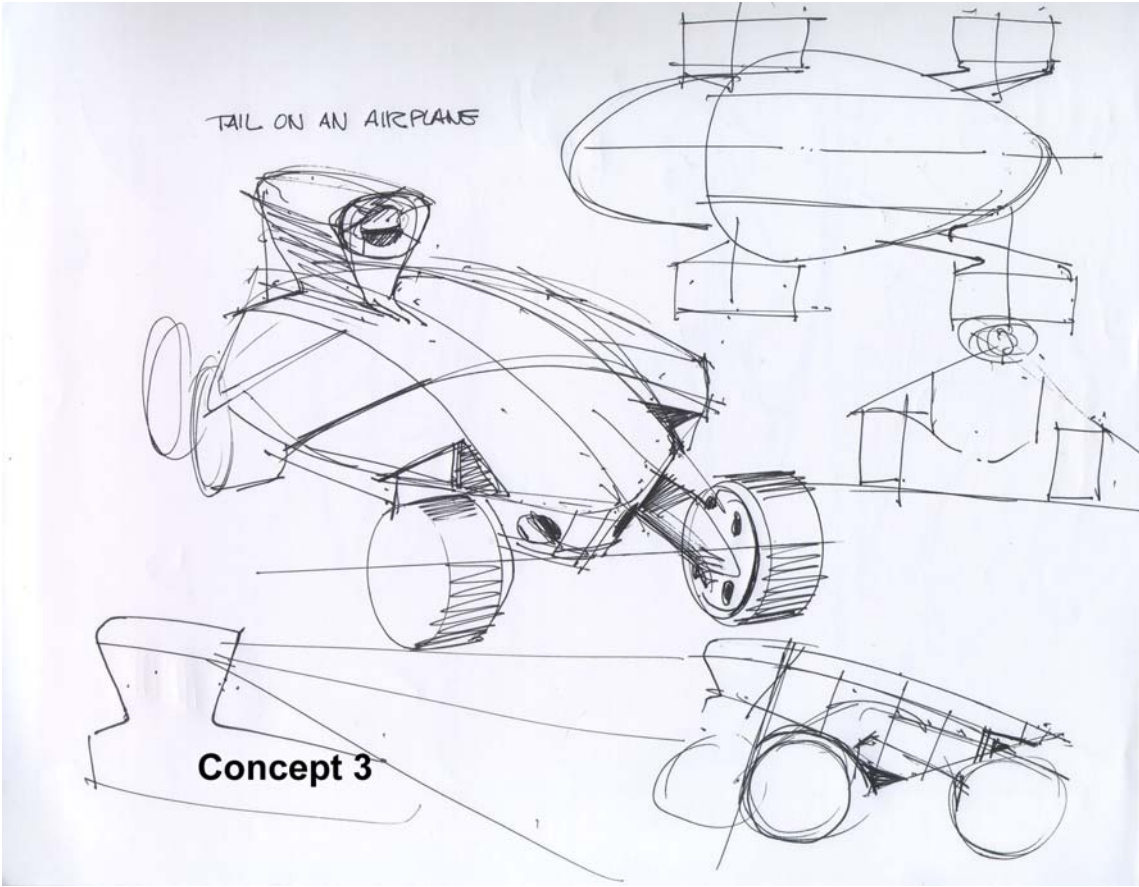
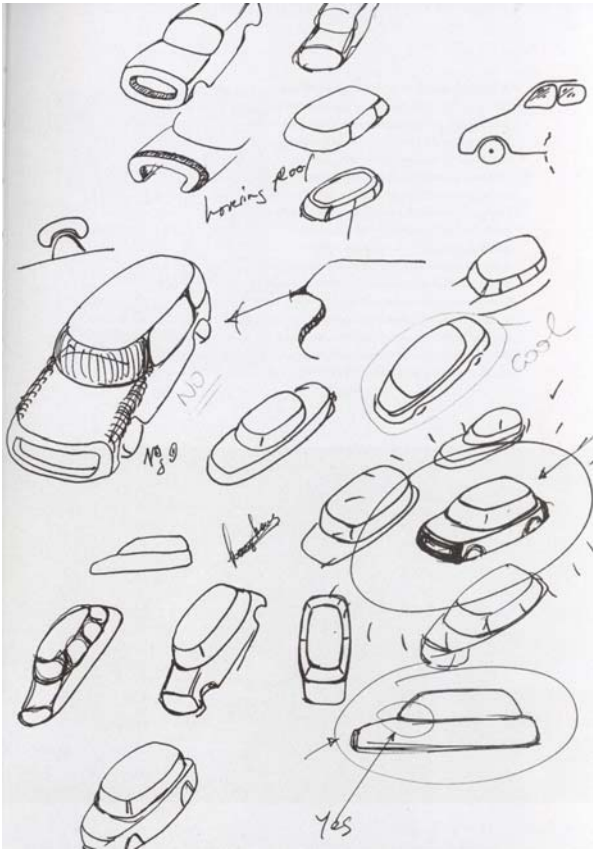


# Decide what you want to do

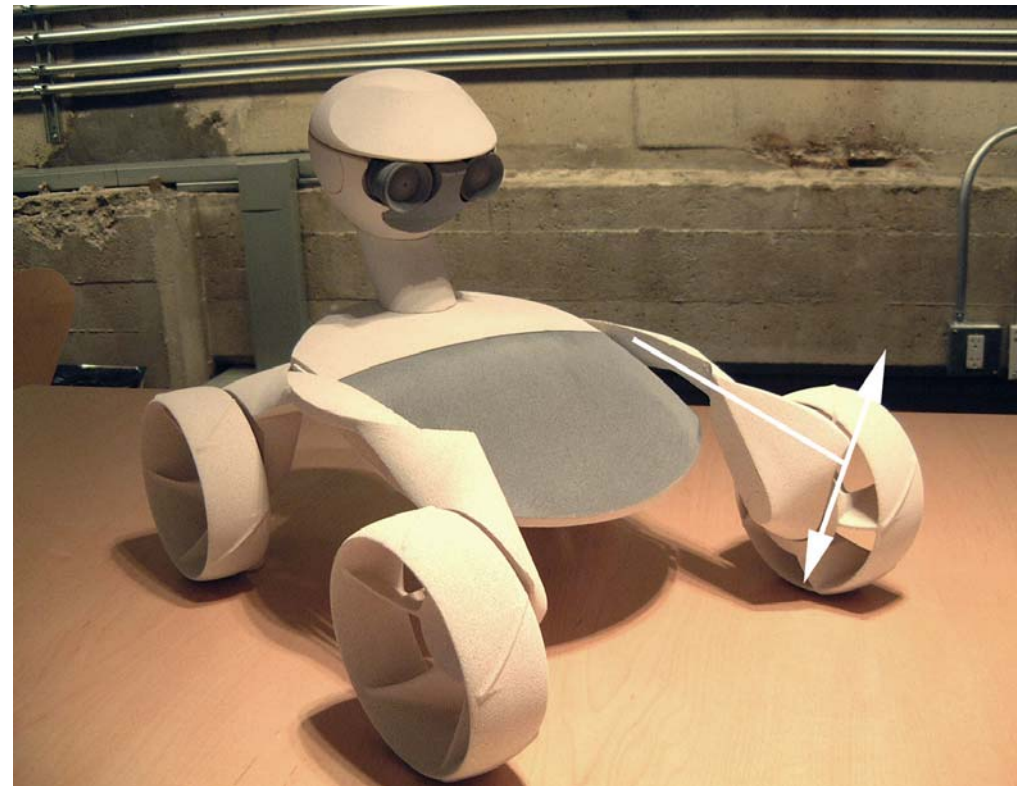
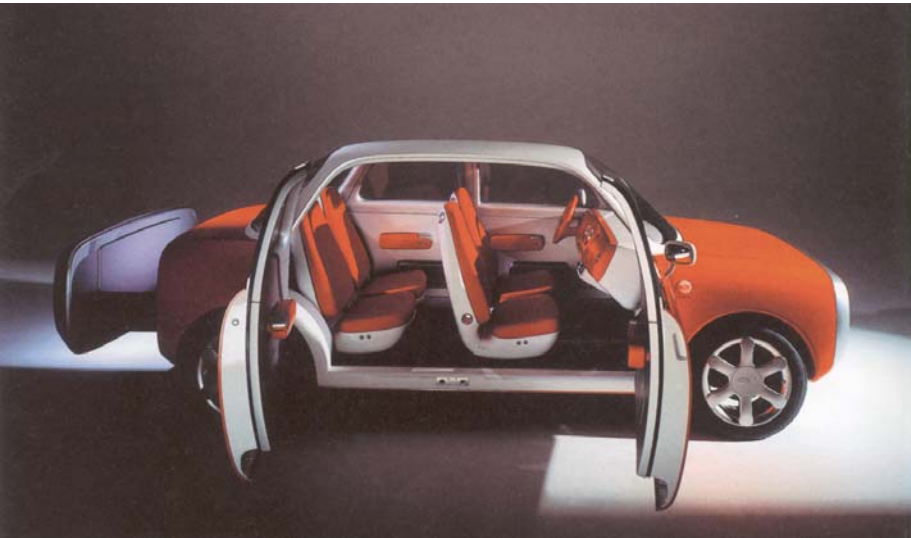
**Your team's goals will  
limit what you should do**



# 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
- Create a crude CAD model and perform a kinematic and/or dynamic simulation
- Have team members act out the parts of the robot to make sure the ideas make sense
- Simulation will help eliminate some ideas and spark new ones









# Six Hats



## Six Thinking Hats Worksheet

- For instructions about Six Thinking Hats, visit <http://www.mindtools.com/rs/SixHats>.
- For business decision-making, visit <http://www.mindtools.com/rs/DecisionMaking>.
- For more business leadership skills, visit <http://www.mindtools.com/rpages/HowtoLead.htm>.

| Thinking Hat   | Focus   | Insights |
|--|---|----------|
|    | <ul style="list-style-type: none"><li>• Available data</li><li>• Past trends</li><li>• Gaps in the data</li></ul> |          |
|    | <ul style="list-style-type: none"><li>• Intuition</li><li>• Gut reaction</li><li>• Emotion</li></ul>              |          |
|   | <ul style="list-style-type: none"><li>• The pessimistic viewpoint</li><li>• Why might it NOT work?</li></ul>      |          |
|  | <ul style="list-style-type: none"><li>• The optimistic viewpoint</li></ul>  |          |
|  | <ul style="list-style-type: none"><li>• Creativity</li><li>• Other ways of doing things</li></ul>                 |          |
|  | <ul style="list-style-type: none"><li>• Process control</li></ul>   |          |

- Group evaluation method
- Everyone wears each hat in turn
- Only the one with the hat can speak
- Comments are limited to the focus area of the hat currently being used in the group





# Multi-Voting

1. Generate a numbered list of items
2. Combine similar items and renumber list
3. Give each member a number of votes equal to about  $\frac{1}{3}$  of the number of (remaining) items on the list (but not less than 2 votes)
4. Conduct voting round: each person can distribute their votes in any combination across the list
5. Eliminate items with the fewest votes
6. Repeat 3-5 until a single solution is the obvious choice



# Implementation

- Before implementation, there needs to be team buy-in on the selected solution
- Different pieces need to be assigned to different people
- Individuals or very small groups complete the pieces
- Keep a schedule to track progress -- update it as each piece gets done
- Work on pieces in parallel
- Create interface agreements among workers so that the pieces will fit together when they are completed
  - e.g., Make a table of sensor and motor ports that shows what goes where
- Use stand up meetings to keep all team members in the loop





# Create a full experimental prototype

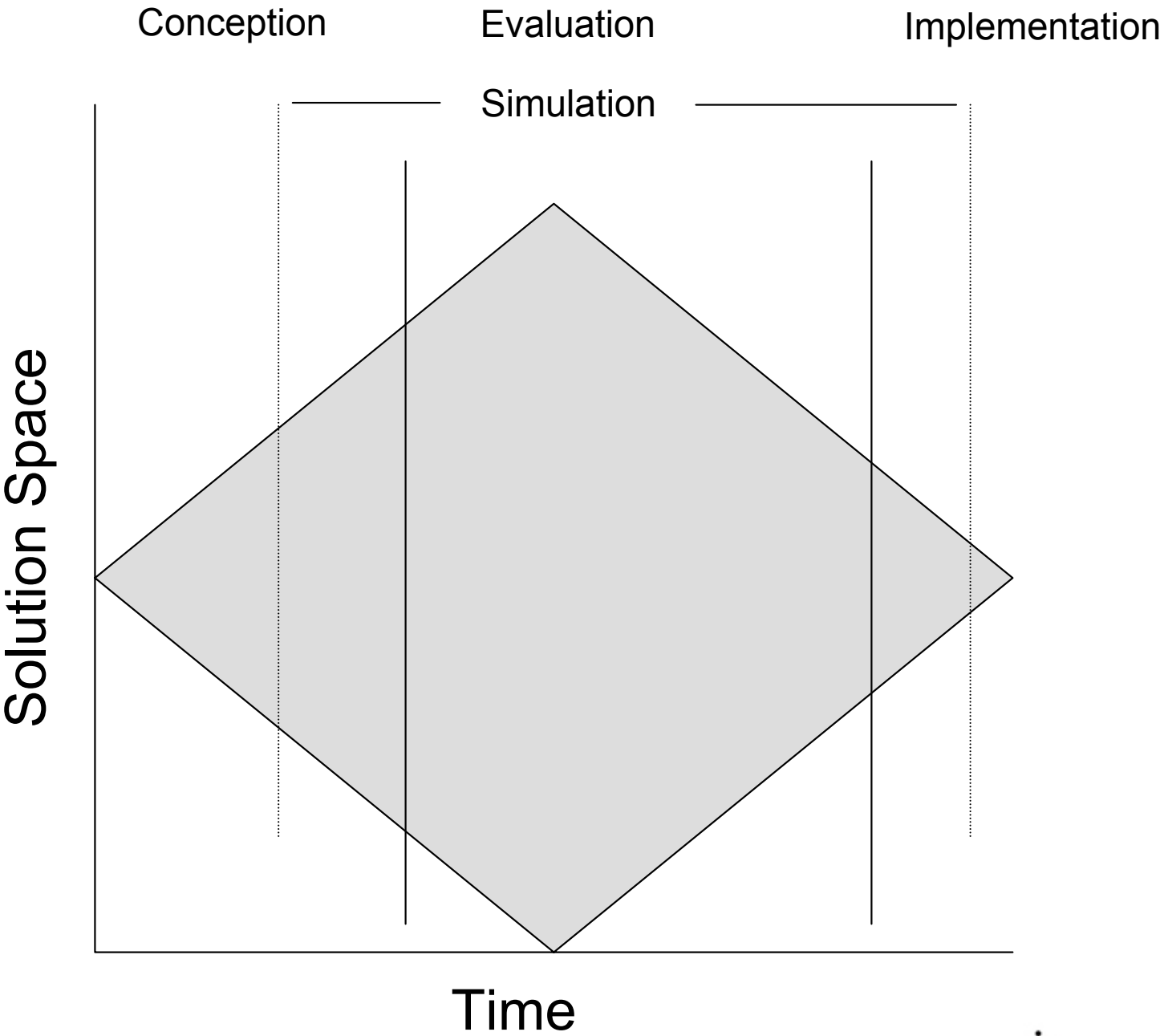
- Do a first draft of the full system
  - Mechanics
  - Software
  - Electronics
- Don't worry about “polish” but include all major functions
- Plan on making major modifications from the lessons you learn from this experimental system



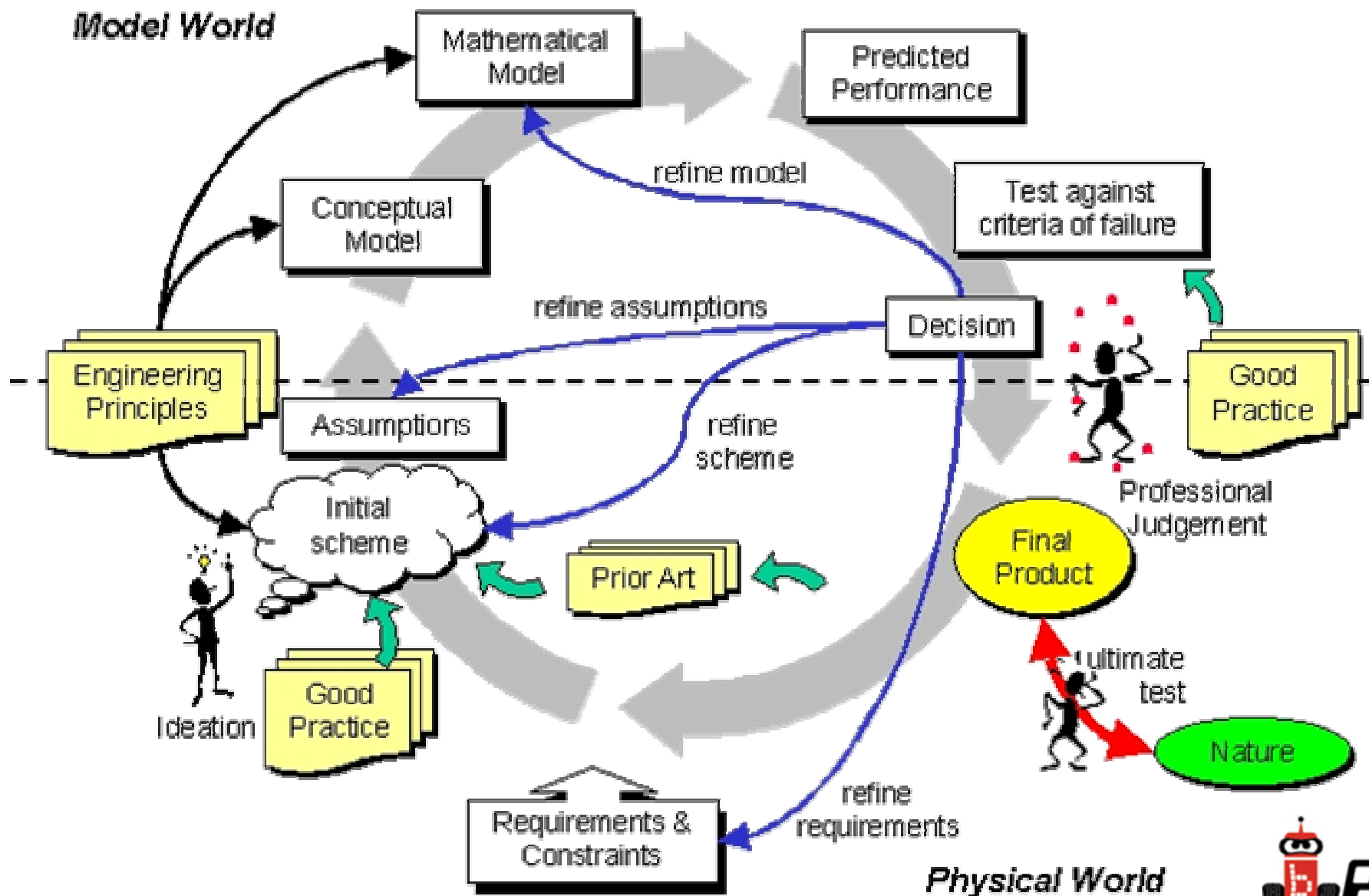
# Create a Final System

- 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





# The Design Cycle



# Handy References

- NASA Robotics Alliance Project: XBC 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](http://www.mindtools.com/pages/article/newTED_07.htm)



# Appendices

# Additional Topics (homework!!)

- Boolean expressions in **C**
- Selection: **if** and **if-else** (+ some programming “gotchas”)
- Repetition using **for** (an alternative to **while**), **break** statement
- Functions and **#use**
- Printing in place on the XBC
- Communication between functions
- **#define** for **IC**
- Variables and scope for **IC**
- Mixed expressions: casting
- Programming style
- XBC motor functions
- XBC motor examples
- Memory & arrays
- Uploading global arrays
- Line following using a reflectance sensor
- Camera example: turn in direction of arrow
- Processes
- Loading the bitstream using the Xport utility
- Loading firmware using the Xport utility
- XBC Camera extension cable
- Challenge Exercises



# Boolean Expressions in C





# Boolean Expressions

- Boolean expressions result in either **1** (*true*) or **0** (*false*)
- 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>*)



# Programming in IC

Selection using **if-else**



# Conditional Execution

## Making a Decision with **if**

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

```
void main()  
{  
    int j = -2;  
    if (j < 0)           /* skip if >= 0 */  
    {  
        j = -j;          /* change sign of x */  
    }  
    printf("magnitude is %d\n",j);  
}
```

# Gotcha (“=” instead of “==”)

- What we intended

```
void main()  
{  
    int i = 2;  
    if (i == 1)  
    {  
        beep();  
    }  
}
```

- What we did

```
void main()  
{  
    int i = 2;  
    if (i = 1)  
    {  
        beep();  
    }  
}
```

# Unintended “;” is a “gotcha”, too

```
void main()  
{  
    int i = 2;  
    if (i == 1);  
    {  
        beep();  
    }  
}
```



```
void main()  
{  
    int i = 2;  
    if (i == 1)  
    {  
        beep();  
    }  
}
```

*What we  
intended*

# Either/Or Selection: **if-else**

- Syntax **if** ( *<TFcheck>* ) { *<statements for true case>* }  
    **else** { *<statements for false case>* }
- If the *<TFcheck>* is true then { *statements for true case* } are selected
- If the *<TFcheck>* is false then { *statements for false case* } are selected

```
void main()  
{  
    int reading;  
    reading = analog12(3);  
    if (reading < 1000) // select < 1000 case  
    {  
        printf( "RED ALERT: %d\n", reading );  
    }  
    else // select >= 1000 case  
    {  
        printf( "System normal\n" );  
    }  
}
```



# IC: **if** (<*sensor-test*>) ... **else**

- Syntax recap: **if** (<*TFcheck*>) { ... } **else** { ... }

```
void main()  
{  
    if(digital(15)==1)      /* select "forward" case */  
    {  
        mav(3,500);  
        printf("Forward\n");  
    }  
    else                    /* select "reverse" case */  
    {  
        mav(3,-750);  
        printf("Reverse\n");  
    }  
    sleep(2.0);             /* let motor run 2 secs */  
    off(3);                 /* stop motor */  
}
```



# IC: Selection Nested in Loop

```
void main() {  
    int dir = 0;                                /* dir switch gives current direction */  
    while (b_button()==0) /* loop until b_button() */  
    {  
        if (a_button()==0) /* select "forward" case */  
        {  
            if (dir == 0) /* if going reverse, forward motor */  
            {  
                mav(3,500);  
                dir = 1; /* dir = 1 indicates going forward */  
            }  
            printf("Forward\n");  
        }  
        else /* select "reverse" case */  
        {  
            if (dir == 1) /* if going forward, reverse motor */  
            {  
                mav(3,-250);  
                dir = 0; /* dir = 0 indicates going in reverse */  
            }  
            printf("Reverse\n");  
        }  
    }  
    off(3); /* turn off motor */  
}
```





# Programming in IC

Repetition using **for**,  
**break** statement



# for Loops

- **for** loops incorporate the loop control initialization, the condition test, and the loop control modification within one statement
- The syntax of a **for** loop is the following:  
`for(<expr-1>; <expr-2>; <expr-3>) { <statements> }`
- The **for** loop construction is equivalent to the following **while** loop construction:

```

<expr-1>;           // loop control initialization
while ( <expr-2> ) { // condition test
    <statements>
    <expr-3>;       // loop control modification
}

```

- Example:

| Using <b>for</b>  | Using <b>while</b>  |
|---|---|
| <pre> for (i=0; i&lt;10; i=i+1){     printf( "%d\n", i); } </pre> | <pre> i=0; while(i&lt;10) {     printf( "%d\n", i);     i=i+1; } </pre> |



# Ending a Loop Early: **break**

- Use of the **break** statement provides an early exit from a **while** or a **for** loop.

```
void main(){  
    int i;  
    for(i=0; i<=100; i=i+1) {  
        beep(); sleep(.1); // beep once  
        if (b_button()) break;  
        // had enough? then stop  
    }  
}
```



# Loop “Rule of Thumb”

(when to use a for-loop rather than while-loop)

- Use a for-loop when you know you want to execute something up to  $X$  times (where  $X$  may be a constant or a formula that can be executed at the start of the loop to determine the exact number of times it will be executed)
- Use a while-loop when the loop may execute 0 or more times but it is non-deterministic when the loop may end – for example driving forward until you bump into something.



# Programming in **IC**

`++` and `--` operators

# ++ and -- Arithmetic Operations

- ++ and -- are “unary operators”
  - i++ is equivalent to i = i + 1
  - i-- is equivalent to i = i - 1

- Useful with **for** loops

```
void main() { // print an X pattern on the display
    int i;
    display_clear();
    for (i=0;i<14;i++) { // print the \ slash
        display_set_xy(2*i,i);
        printf("x");
    }
    for (i=13;i>=0;i--) { // print the / slash
        display_set_xy(26 - 2*i,i);
        printf("x");
    }
}
```



# Programming in IC

## Functions and #use



# IC: Writing your Own Functions

- In **IC**, click on *New*
- Enter a function; e.g.,

```
void pause()  
{  
    printf ("press A when ready\n"); // A button  
    /* wait for A button */  
    while (a_button() == 0){}; //wait for button press  
    while (a_button() == 1){}; //wait for button release  
    printf ("Started\n");  
}
```

- Do *Save As* and name the file “pause”
- Do *download* to send the function to the Board
- Note that the tab is labeled “pause.ic”
- Click on *Tools..List functions* to verify “pause” is present
- Try calling **pause( )** from interaction window





# Variables Must Be Declared at Beginning of Block

**WRONG!**

```
float circ_area(float r){  
    printf("area");  
    float pi = 3.141593;  
    return(pi * r * r);  
}
```

**RIGHT!**

```
float circ_area(float r){  
    float pi = 3.141593;  
    printf("area");  
    return(pi * r * r);  
}
```



# #use Preprocessor Statement

- **#use** is a preprocessor directive
  - Preprocessor directives start in column 1 with the “#” symbol
- Purpose of **#use**
  - Program blocks may be saved in multiple files
  - Before **IC** can compile a user’s program, the **IC** preprocessor has to gather together all of the program files
  - The **#use** directive is needed to tell the preprocessor where to find files when multiple files are being used
- Example

```
#use "globals.ic"
#use "pause.ic"
void main () { . . .
```
- **WARNING!!** Preprocessor directives are terminated by the end of the line, not a semi-colon. Don’t put comments on a preprocessor directive's line (it will get pulled in as part of the preprocessor command!).



# Using Functions to Repeat

- Beep any number of times

```
// load my pause function
#use "pause.ic"
void main() // this is executed when program starts
{
    pause(); // wait for button to be pressed and released
    printf("Beep 15 times\n"); // print message
    repeat_beep(15); // call our beep with 15 as parameter
    printf("Do it again, but now 30!!\n"); // print message
    repeat_beep(30); // call our beep with 30
}

void repeat_beep(int numbeeps) // function is named repeat_beep
{
    int num=1; /* declare & initialize counter */
    while (num <= numbeeps) // loop while num is <= numbeeps
    {
        beep(); /* beep once */
        num = num + 1; /* add one to the counter */
    }
}
```



# Programming in **IC**

Printing in place



# Function for Print in Place

- Printing in place is a two step process using `display_set_xy` and `printf`. You can write a function that combines these two actions.

```
void print(char text[], int row, int col) {  
    display_set_xy(col,row);  
    printf(text);  
}
```

- Test this on the simulator in the interaction window using something like

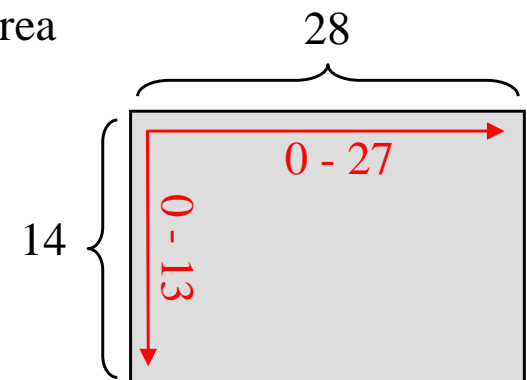
```
print( "Botball", 3, 5 );
```

- What happens if you make a mistake and use a row or column value that is too large?
  - What if you want to print a number rather than text?
  - It can be tricky to build a library function that has general usefulness!
- You probably noticed that the first argument for the function (`text[ ]`) has brackets and data type `char`. This is how you indicate to **IC** that the argument holds a “string” (or array) of characters.
  - Arrays are a topic to be covered elsewhere



# Printing in Place on the XBC

- The **IC** library for the XBC provides functions to take advantage of the rectangular layout of the XBC display to allow **printf** to start from any position on the display.
  - The XBC display provides a 14 row, 28 column character area
    - Rows are numbered 0 to 13, columns 0 to 27
  - **display\_clear( )**
    - Function to clear the display
  - **display\_clear\_at**
    - **display\_clear\_at(3,5,7);**
      - erases 7 characters on row 5 starting at position 3
  - **display\_set\_xy**
    - **display\_set\_xy(0,0);**
      - positions IC for the next **printf** to start from the upper left hand corner of the display
    - **display\_set\_xy(0,10);**
      - positions IC for the next **printf** to start from the left end of row number 10 (ie., the 11<sup>th</sup> row)
  - **display\_get\_xy**
    - **display\_get\_xy(&col,&row);**
      - Returns in variables col and row the position for the next **printf**
      - Seldom needed



# Try it on the Simulator

- First execute the program

```
void main() {  
    display_clear();  
    display_set_xy(4,10);  
    printf( "BALL" );  
    sleep(3.0);  
    display_set_xy(1,9);  
    printf( "BOT" );  
}
```

- Now from the simulator interaction window do something like

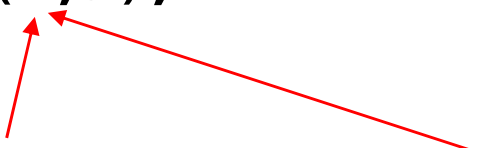
```
{display_set_xy(9,11); printf( "!!!!" );}
```



# Program Flow

- When a program is run, statements execute in order from one statement to the next (*program flow*)
- The program we did earlier illustrates this:

```
void main() {  
    display_clear();  
    display_set_xy(4,10);  
    printf("BALL");  
    sleep(3.0);  
    display_set_xy(1,9);  
    printf("BOT");  
}
```



- **Warning:** entering **1** (*ell*) instead of **1** (*one*) is a common “Gotcha”!
  - Hard to spot, especially if you have a variable name **1**!
- **C** has constructions that allow you to modify the normal program flow





# Communication Between Functions



# Communicating Using Parameters

- The **print** function

```
void print(char text[], int row, int col) {  
    display_set_xy(col,row);  
    printf(text);  
}
```

- Arguments must use the type specified for each parameter

- Other functions communicate with the **print** function by calling it, supplying a value (of the correct type) for each argument; e.g.,

- `print("Botball", 3, 5);`

↑                    ↑    ↑  
*text string, integer, integer*



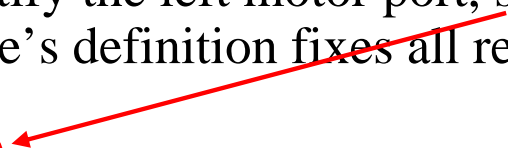
# Communicating Using Global Variables

- Global variables provide means for sharing data among functions

```
int left_motor = 2, right_motor = 0; // globals
void ahead_full() {
    fd(left_motor); fd(right_motor);
}
void reverse_full() {
    bk(left_motor); bk(right_motor);
}
```

- If you decide to move the left motor to port 3, assuming you always use the global variable to identify the left motor port, simply changing the value of the global variable's definition fixes all references that are for the left motor port.

```
int left_motor = 3, right_motor=0; // globals
. . .
```



# Communicating Using Global Variables - Flags

- Global variables provide means for flagging an action's completion

```
#use "pause.ic"
int flag = 0; // global to flag if a function has printed
void main() {
    display_clear(); pause(); printf("\n\n"); // open some space
    printb(); printg();
    if (flag == 0) printf("(no one was here)\n");
    printf("\nDONE\n");
}
void printb() {
    if ((flag == 0) && random(2)) { // if no one has printed maybe I will
        printf("BotGuy was here\n");
        flag = 1; // change flag to let everyone know I've printed
    }
}
void printg() {
    if ((flag == 0) && random(2)) { // if no one has printed maybe I will
        printf("BotGal was here\n");
        flag = 1; // change flag to let everyone know I've printed
    }
}
```

- `random(int x)` is a library function returning a random integer between 0 and x-1, inclusive
  - `random(2)` will randomly be either 0 or 1



# Programming in IC

## `#define` for IC



# #define Preprocessor Statement

- Purpose of **#define**
  - Equate a meaningful name to repeatedly encountered text
    - **#define** READING **analog**(3)
      - Before compiling, the preprocessor replaces all occurrences of **READING** with **analog**(3)
      - eg.  

```
if (READING < 30) { . . .
```

is equivalent to  

```
if (analog(3) < 30) { . . .
```
    - May reduce overhead (see the **IC** on-line programmer reference manual)
- Has a limited macro capability
- In IC **#define** affects all code loaded from **#use**



# Programming in **IC**

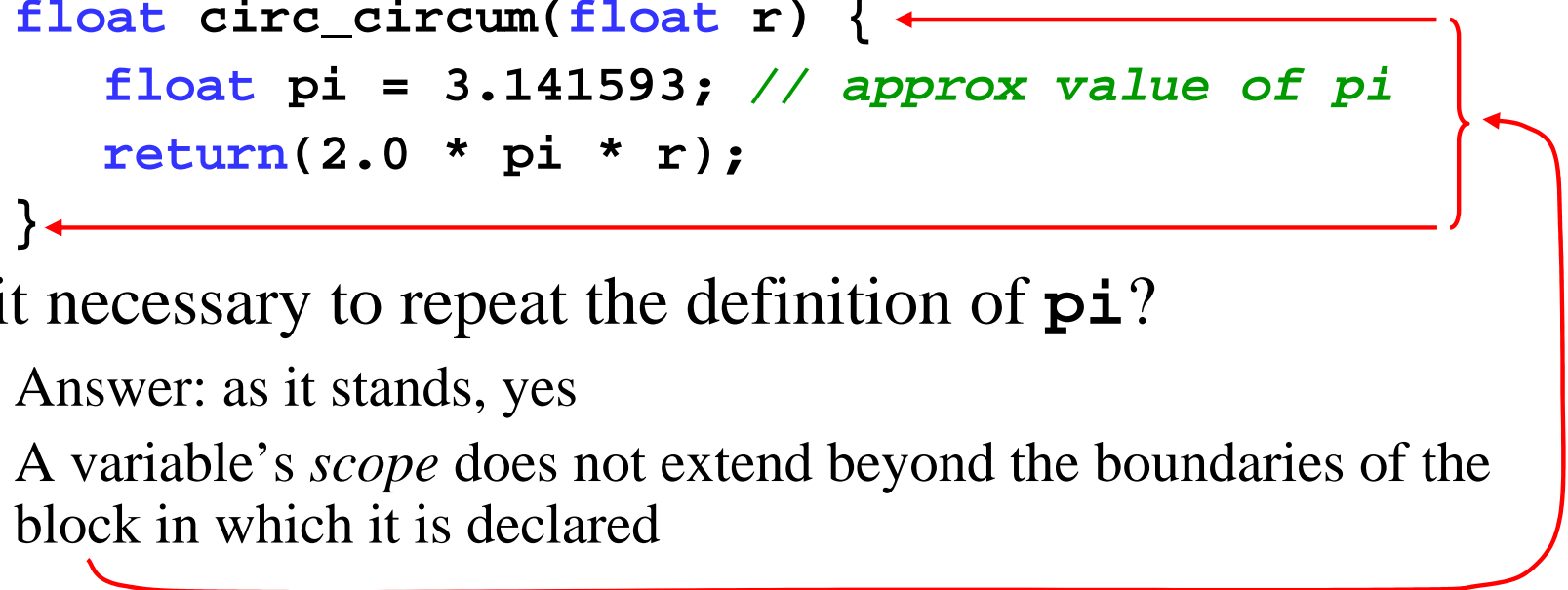
## Variables and Scope



# Variable Scope

- Your area and circumference functions probably look like this

```
float circ_area(float r) {  
    float pi = 3.141593; // approx value of pi  
    return(pi * r * r);  
}  
float circ_circum(float r) {  
    float pi = 3.141593; // approx value of pi  
    return(2.0 * pi * r);  
}
```

A red bracket on the right side of the code block groups the two function definitions. A red arrow points from the opening curly brace of the second function to the `pi` variable definition inside it. Another red arrow points from the closing curly brace of the second function to the `pi` variable definition inside the first function, illustrating that the `pi` variable in the second function is not visible to the first function.

- Is it necessary to repeat the definition of `pi`?
  - Answer: as it stands, yes
  - A variable's *scope* does not extend beyond the boundaries of the block in which it is declared



# Using Global Variables for Constants

- If the (approximate) value for the constant  $\pi$  is stored in a global variable **pi**, its definition isn't needed in functions that use it

```
float pi = 3.141593; // approx value of pi
float circ_area(float r) {
    return(pi * r * r);
}
float circ_circum(float r) {
    return(2.0 * pi * r);
}
```

- **pi** is global since its specification is outside of all functions
  - **circ\_area** and **circ\_circum** use the global variable **pi** in their calculations
- Global variables should be declared at the beginning of the program file
- Other typical constants are motor and sensor port numbers, motor power, and servo positions



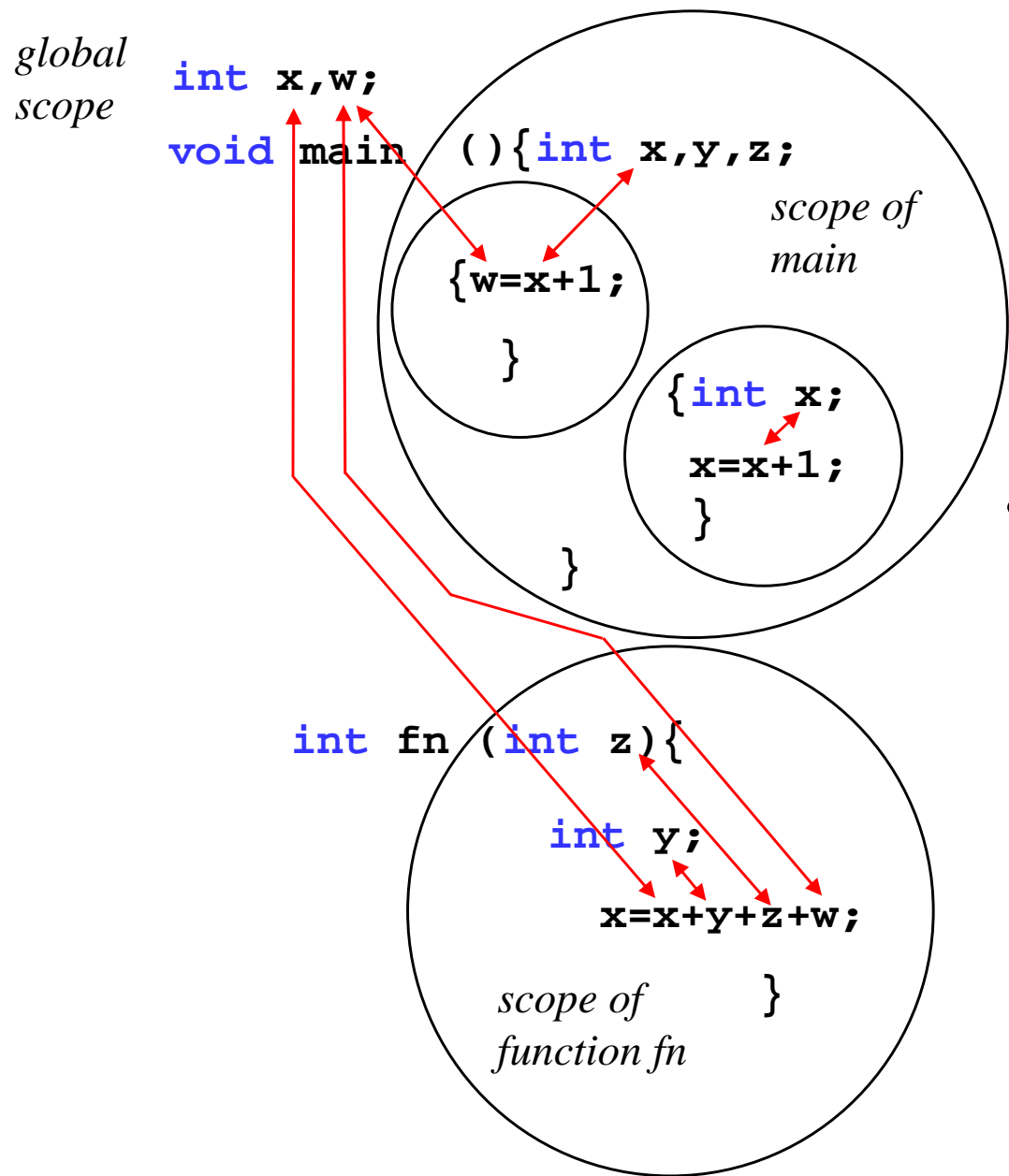
# Determining a Variable's Scope

- Each block is like a one-way mirror
  - You can't look in to locate a variable
  - You can look out to locate a variable
    - The most local version is the one used
    - If not found all the way out to the global level, you have an error!
- Global variables provide the means for interprocess communication

```
int imaglobal;  
void f1()  
{  
    int imalocal, imalocal2;  
    //This will work  
    imalocal2=imalocal+imaglobal;  
}  
  
void f2()  
{  
    //error: undefined imalocal2  
    imaglobal=imalocal2;  
}
```



# Super Ninja Scoping



- Avoid this situation, use unique variable names!!



# Programming in **IC**

## Mixed Expressions - Casting



# Mixed Expressions in C: *Casting*

- All of the operands in an expression (in **IC**) must be of the same data type.
- There is a special (unary) operator known as the *cast* operator which can be used to “coerce” its operand into a different data type (thus allowing “mixed” expressions).
  - For example, if you have an integer variable *r* and want to calculate  $x=2\pi r$  your program code would need to be similar to:

```
int r;  
float x, pi=3.1416;  
x = 2.0 * pi * (float)r
```
  - *r* has been cast to a (temporary) floating point location in the appropriate format so it can be used in the computation of  $2\pi r$
- An expression involving casting can become quite complicated.
  - For example,

```
(int) ((float) mseconds() / 1000.0 * range) + offset;
```
  - Two cast operators are used – (**int**) and (**float**)
  - What must the type of **range** and **offset** be, respectively?



# Programming in **IC**

## Programming Style



# Program Style: Commenting

- Explanatory comments
  - Can (and should) be added to a program to assist in understanding it later
  - Do not affect the size of the compiled program on the XBC
- 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
- Example:

```
/* simple.ic -(c) 2003 David Miller, KIPR */
/* This program displays a simple character string
   -- It is a good idea to start each function with a
   comment that explains what the function does --
[this comment and the previous one are in multi-line form] */
void main()
{
    printf("This is a C program\n"); // display the string
} // end of main [this comment and the previous one are in-line]
/* Hmm ... the last in-line comment is pretty superfluous! */
```



# File History & Comments

```
/* simple.ic - (c) 2003 David Miller, KIPR */

/* History:
    Modified 6/15/03 - shortened the initial comments &
    changed the word program to function
    in the printf text - dpm
    Modified 6/16/03 - removed silly comments at end - dpm
*/
/* This program displays a simple character string */
void main()
{
    printf("This is a C function\n"); // display the string
}
```





# Program 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!)
- **IC**’s built in editor does most indentation for you!
  - Programs are automatically indented when loaded
  - The **IC** *edit* menu provides commands for indenting your text without saving and reloading



# Syntax and Semantics

- **C** *syntax* is prescribed by a formal grammar that provides the construction “rules” for **C** programs
- **C** *semantics* determine the interpretation of a statement that is syntactically OK
  - **a = b - c;**  
is interpreted to mean “*subtract the contents of variable **c** from those of variable **b**, storing the result in variable **a***”.
  - Note that **C** attempts to “do what we expect”; i.e., in the above case, subtract **c** from **b**, not the other way around
- Fixing program semantics is the major part of the exercise programmers call “debugging”



# XBC Motor Functions

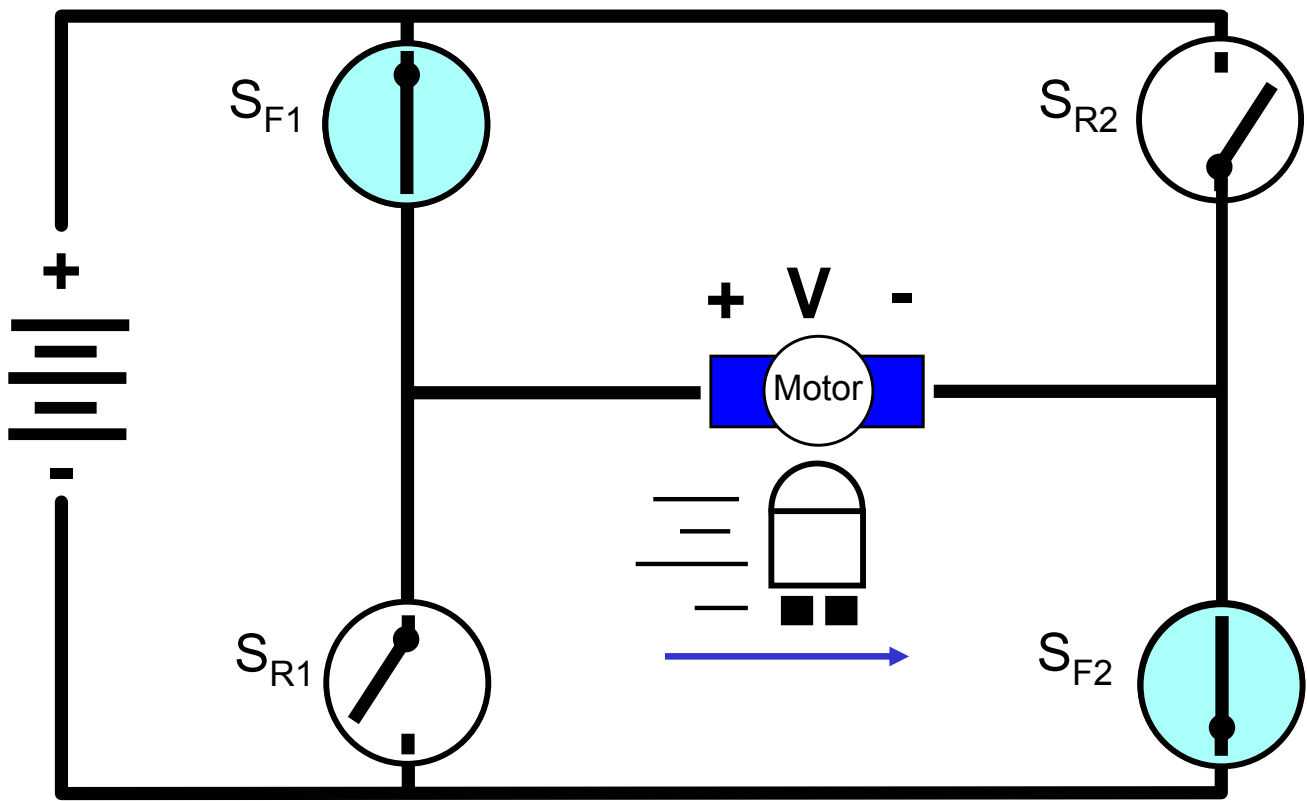
# How do you connect motors to computers?

- Computers operate at constant voltage and very low current
- Motors draw high current and their speed and direction depends on the voltage
- Robot controllers use:
  - H-Bridge circuit to handle current and direction
  - PWM to control speed



# H-Bridge

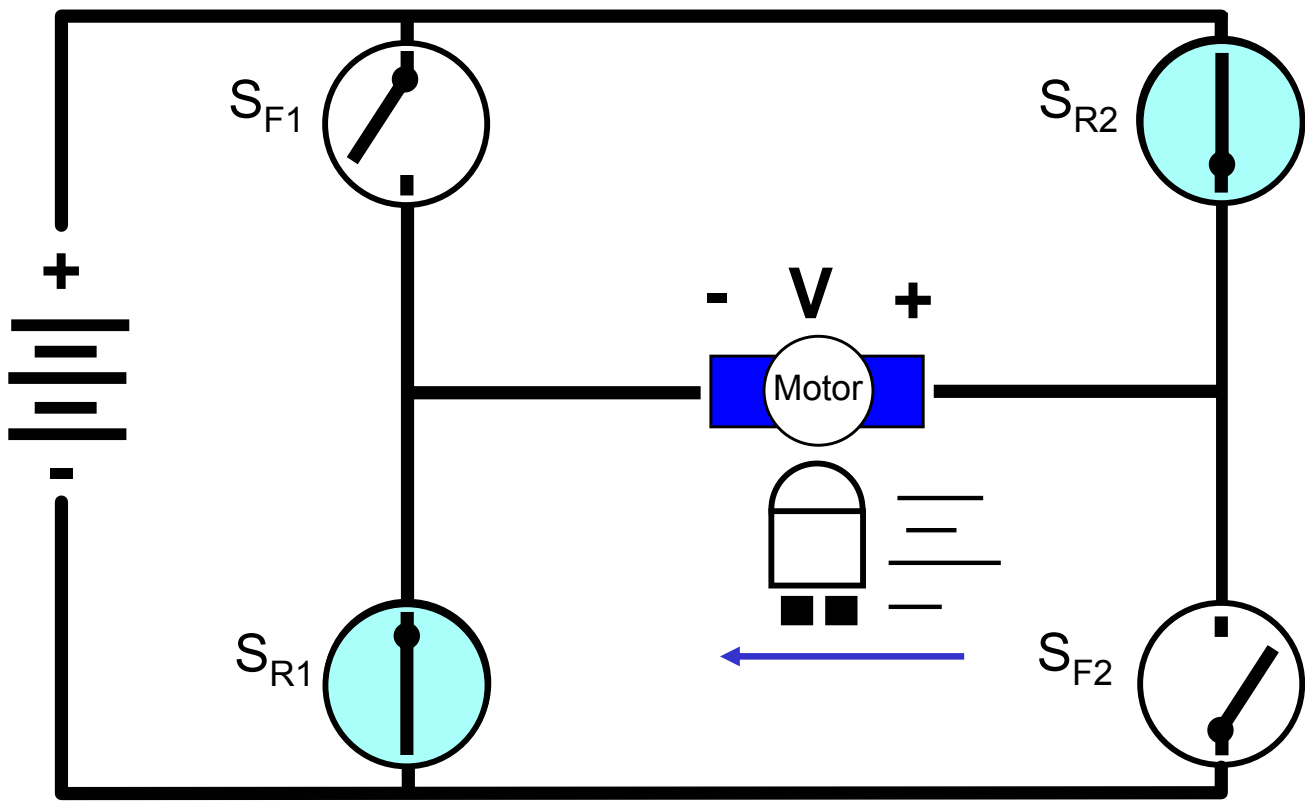
Separate Logic Current from Motor Current



Pulse On

# H-Bridge (reversed)

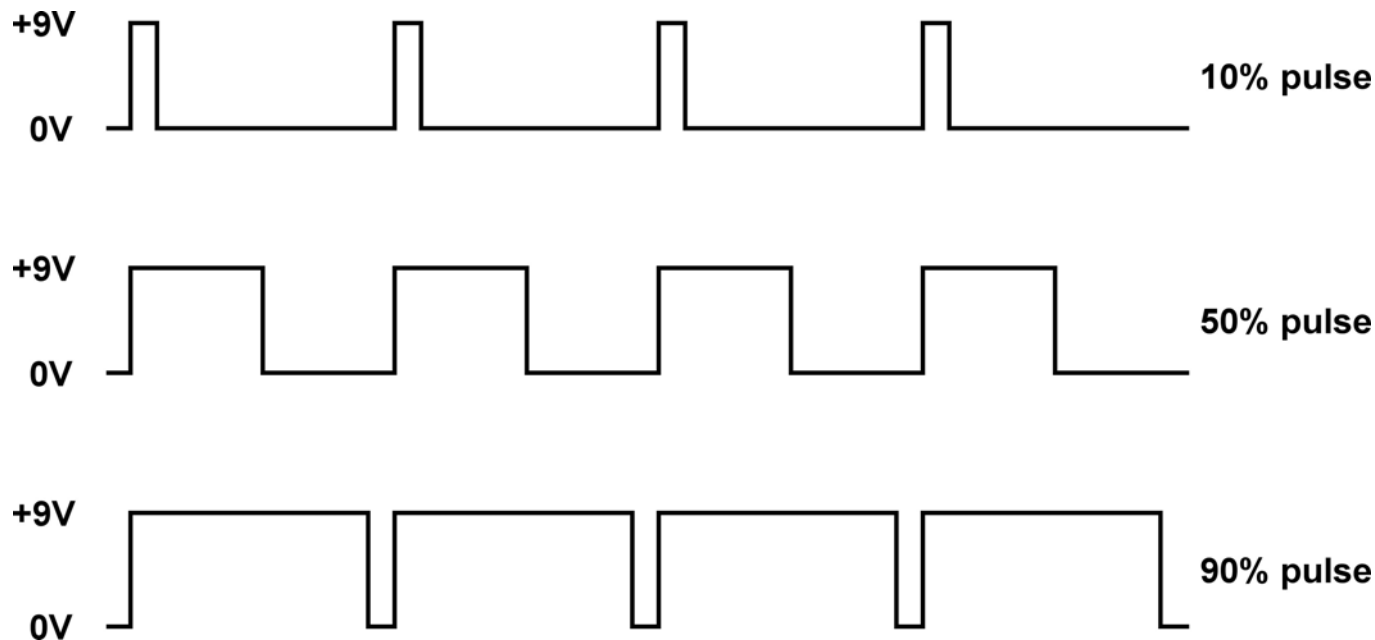
Separate Logic Current from Motor Current



Pulse On

# Pulse Width Modulation (PWM)

- Pulse motor at fixed frequency
  - Maintains voltage level supplied to motor
  - Duty cycle governs speed



# DC Motors - PWM

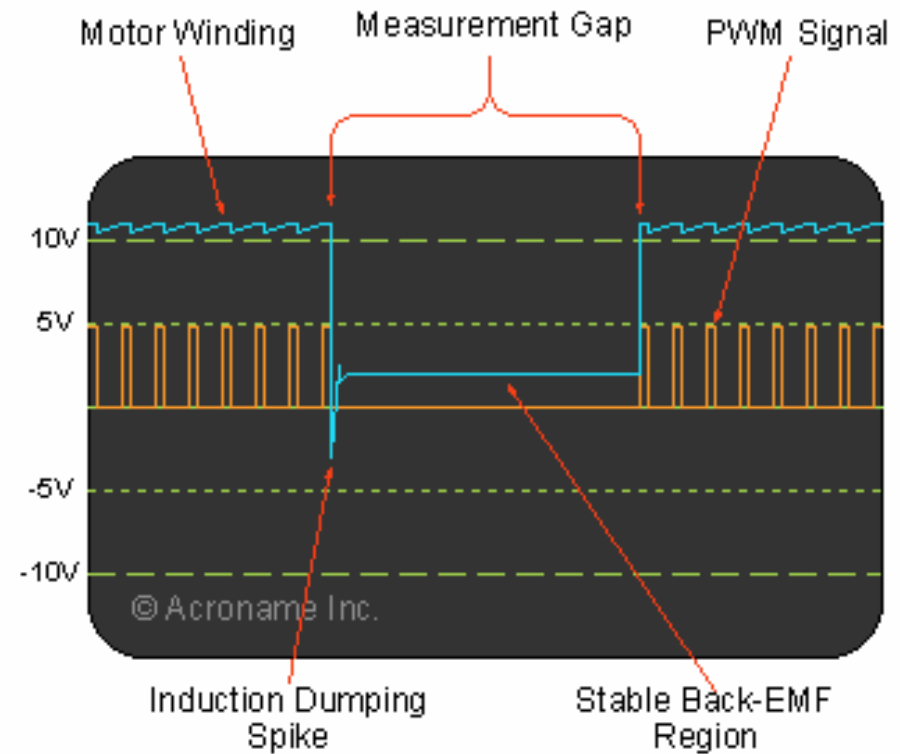
- Black gear motors resemble servos -- but they are not (identify by gray cable color)
- Other motors (white & silver) also have gray cables
- Use Motor channels 0, 1, 2 & 3
- Full Forward PWM: `fd(3);`
- Full Reverse PWM: `bk(3);`
- 2/3 Reverse PWM: `motor(3,-66);`
  - On XBC this means run the motor in reverse at 66% scaled duty cycle
- Turn off motor: `off(3);`
- Turn off all motors: `ao();`





# Back EMF

- XBC can measure EMF generated back from spinning motor
- BEMF is proportional to actual motor speed



25% Duty Cycle Simulated Scope

from [acroname.com](http://acroname.com)

# BEMF Commands (1)

- `clear_motor_position_counter(3)`
  - This clears the position counter for motor 3 to be 0
- `get_motor_position_counter(3)`
  - This returns a **long** integer which is the values of the position counter for motor 3
- `set_motor_position_counter(3,200L)`
  - This sets the position counter for motor 3 to the **long** value 200. This function is rarely used -- usually the only value you want to set a counter to is 0, which is best done using `clear_motor_position_counter`



# BEMF Commands (2)

- `move_at_velocity(3,123)`
  - This will try and move motor 3 forward at 123 ticks per second. If the motor velocity is affected by outside forces, the duty cycle of the PWM being sent to the motor will be changed as needed to try and keep the motor velocity will be changed the clears the position counter for motor 3 to be 0
  - Note that velocity values are integers between -1000 and 1000
  - Note that this command is terminated by any other PWM or BEMF command that moves this motor.
  - Note that performing a `set_motor_position_counter` or a `clear_motor_position_counter` while `move_at_velocity` is active might cause erratic behavior.
- `mav(3,123)`
  - This is shorthand way of doing `move_at_velocity`



# BEMF Commands (3)

- `move_relative_position(3,123,-369L)`
  - This will try and move motor 3 backwards at 123 ticks per second until it has moved 369 ticks behind where it was when the command was issued. The second argument (123 in the example) is a speed, not a velocity. The third argument is a **long** corresponding to the distance to move. The sign of the position indicates whether or not the motor should turn forwards or backwards.
  - Note that speed values are integers between 0 and 1000
  - Note that this command does not block but takes time (the example values should take about 3 seconds). This command will finish when the destination position is reached or will be terminated early by any other PWM or BEMF command that moves this motor before the goal is reached.
  - Note that performing a `set_motor_position_counter` or a `clear_motor_position_counter` while `move_relative_position` is active might cause erratic behavior.
- `mrp(3,123,-369L)`
  - This is shorthand way of doing `move_relative_position`



# BEMF Commands (4)

- **move\_to\_position(3,123,-369L)**
  - This will try and move motor 3 in whichever direction is needed at 123 ticks per second until the motor counter has reached -369. The second argument (123 in the example) is a speed, not a velocity. The third argument is a **long** corresponding to the goal position as specified by the motor counter.
  - The sign of the position does not indicate the direction of movement. Movement direction is automatically determined by the sign of goal position minus current position.
  - Note that speed values are integers between 0 and 1000
  - Note that this command does not block but takes time (the time is roughly the difference between the goal and current positions divided by the speed). This command will finish when the destination position is reached or will be terminated early by any other PWM or BEMF command that moves this motor before the goal is reached.
  - Note that performing a **set\_motor\_position\_counter** or a **clear\_motor\_position\_counter** while **move\_relative\_position** is active might cause erratic behavior.
- **mtp(3,123,-369L)**
  - This is shorthand way of doing **move\_to\_position**



# BEMF Commands (5)

- **get\_motor\_done(3)**
  - This function returns 0 if a BEMF command is in progress on motor 3 and 1 otherwise
  - If a motor is moving under velocity control (e.g., **mav**) then **get\_motor\_done** will return 0 until that motor command is terminated
  - If a motor is moving under position control (e.g., **mrp** or **mtp**) then **get\_motor\_done** will return 0 until that motor command is terminated or the motor reaches the goal position.
  - If a motor is moving under PWM control (e.g., **fd**, **bk** or **motor**) then **get\_motor\_done** will return 1
- **block\_motor\_done(3)**
  - This function blocks (i.e., your program will not go onto the next statement) until the currently executing BEMF motor command terminates.
  - WARNING! If the robot is stalled before getting to where you told it to move, this function will keep the rest of your program from running indefinitely
- **bmd(3)**
  - This is shorthand way of doing **block\_motor\_done**
  - If a **bmd(3)** immediately follows a **mrp(3,500,3000L)** then the bmd command will block until the motor has moved all 3000 ticks (about 6 seconds for this example)
  - If a **bmd(3)** immediately follows a **mav(3,123)** then the bmd will **not** terminate (unless killed by another process) and your program will hang with the motor running



# BEMF Commands (6)

- **freeze(3)**
  - This function immediately stops the motor then tries to keep the motor at its current position moving at zero velocity then **off** in that it actively powers the motor to stay where it is. It will resist backdriving. The motor may drift slowly due to BEMF errors.
  - This function uses power
  - **freeze** will continue to control the motor until it is terminated by another motor command.



# XBC Motor Examples



# XBC Motor Examples (1)

```
/* This function moves the motor 2 2000 ticks at a  
   speed of 500 ticks per second. This function  
   will take about four seconds to terminate  
*/  
void move2000ticks()  
{  
    mrp(2,500,2000L);  
    bmd(2);  
}
```



# XBC Motor Examples (2)

```
/* This function moves the motor 2000 ticks at a  
speed of 500 ticks per second. This function  
will take about four seconds to terminate unless  
digital(15) returns 1 first, in which case the  
motor will immediately stop and the function will  
return  
*/  
void move2000ticks_bump()  
{  
    mrp(2,500,2000L);  
    while(!get_motor_done(2) && !digital(15)){  
        off(2); //only needed if digital(15) is hit  
    }  
}
```



# XBC Motor Examples (3)

```
/* This function moves the motor forward 3300 ticks  
   (about 3 revs) and then moves it backwards the  
   same amount  
*/  
void back_and_forth_three()  
{  
    mrp(2,500,3300L);  
    bmd(2); // wait for mrp to finish moving  
    mrp(2,500,-3300L);  
    bmd(2); // wait for mrp to finish moving  
}
```



# XBC Motor Examples (4)

```
/* This is a GOTCHA!!!!!!  
   This function will only turn backwards about  
   3 revs (it will never turn forwards) because  
   there is no delay between the forward and the  
   backwards mrp commands, so the first mrp command  
   is immediately overridden by the second mrp  
*/  
void back_and_forth_three()  
{  
    mrp(2,500,3300L); // start going forward  
    mrp(2,500,-3300L); // cancel that, go backwards  
    bmd(2); // wait for mrp to finish moving  
}
```



# XBC Motor Examples (5)

```
/* This function moves the motor forward at full
   speed for 5 seconds, prints out the distance
   traveled and then runs the motor in
   reverse at 100 ticks/sec the exact same amount
   */
void back_and_forth_the_same_dist()
{
    // clear motor counter (set to 0)
    clear_motor_position_counter(2);
    mav(2,1000); // turn forward at full speed
    sleep(5.0);  // keep turning for 5 secs
    off(2); // turn the motor off
    printf("dist=%1\n",
           get_motor_position_counter(2));
    mtp(2,100,0L); // move back to position 0
    bmd(2); // wait for mtp to finish moving
}
```



# XBC Motor Examples (6)

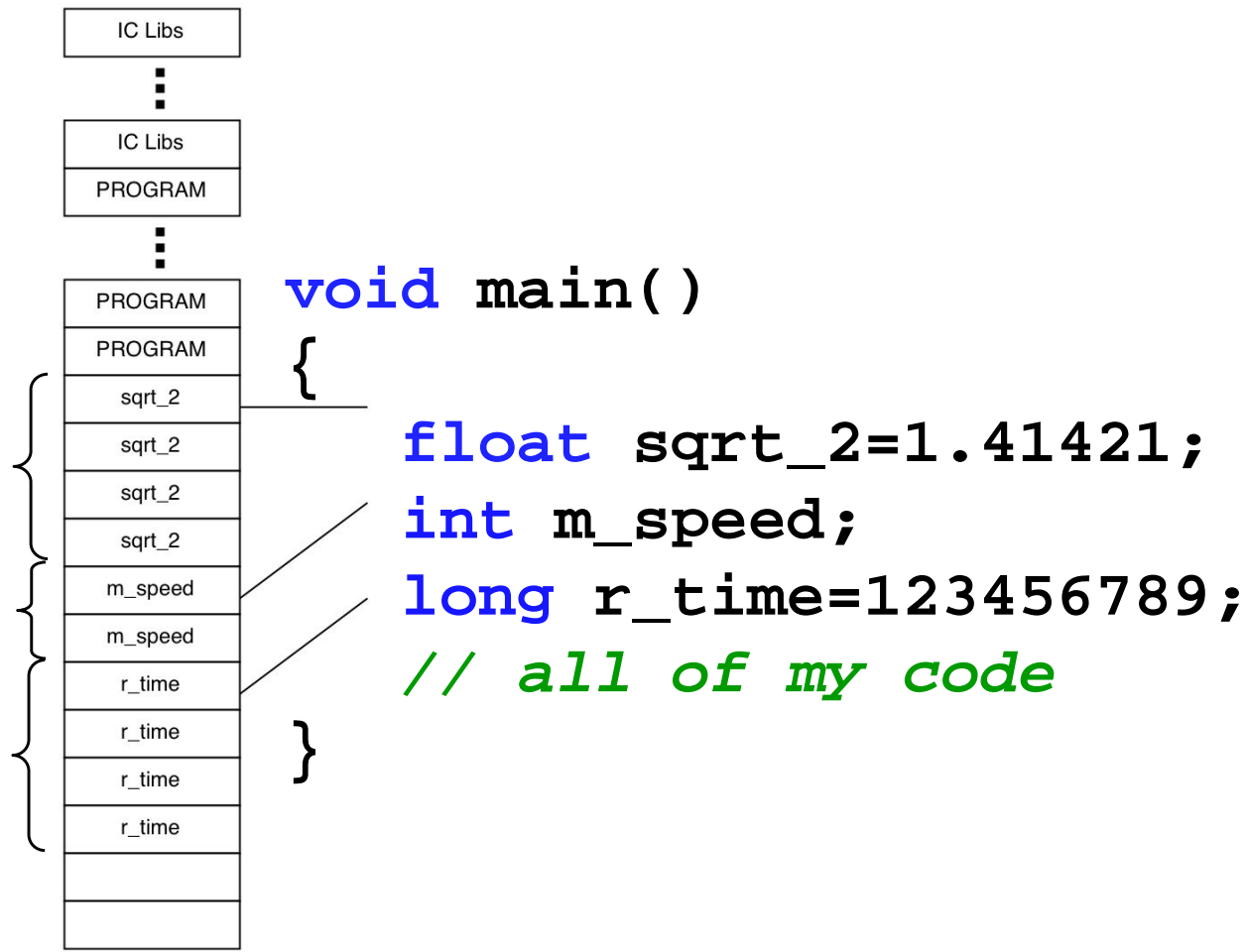
```
/* This function takes a float as argument and turns the bot  
clockwise that much in place. Assume wheel radius is 50mm, left  
and right wheels are 200mm apart, left wheel is on motor 1  
and right wheel is on motor 2 and one wheel rotation is 2000 ticks  
*/  
void rotate_bot(float turns)  
{  
    float half_width, radius, pi, ticks_per_robot_rev;  
    float ticks_per_wheel_rev, robot_rev_circum, wheel_circum;  
    long ticks_to_turn;  
    //put in pre-defined values  
    half_width=100.; radius=50.; pi=3.1416; ticks_per_wheel_rev=2000.;  
    //calculate circumferences and ticks  
    wheel_circum=pi*radius*2.0;  
    robot_rev_circum=half_width*pi*2.0;  
    ticks_per_robot_rev=  
        ticks_per_wheel_rev*robot_rev_circum/wheel_circum;  
    ticks_to_turn=(long)(turns*ticks_per_robot_rev);  
    mrp(1,400,ticks_to_turn); //move left motor forward  
    mrp(3,400,-ticks_to_turn); //move right motor backwards  
    bmd(1); bmd(3); //wait for both motors to complete moves  
}
```



# Computer Memory, Arrays



# Computer Memory





# Arrays

- What is an array?
  - Contiguous blocks of memory, all of the same type
    - Each block of memory is an array cell
  - Predetermined in size at compile time
    - The **IC** function `_array_size( <name_of_array> )`  
can be used to determine the number of cells in the array



# Declaring Array Variables

Syntax:

*<data-type> <var-name> [ <size> ] ;*

or

*<data-type> <var-name> [ <size> ] = { <initial-data> } ;*

- Examples

- `int BBdata[12];`
- `int sensor[3] = {1, 2, 3};`

- Arrays are **zero-indexed** and must be accessed one element at a time using the array [ ] subscript operator.

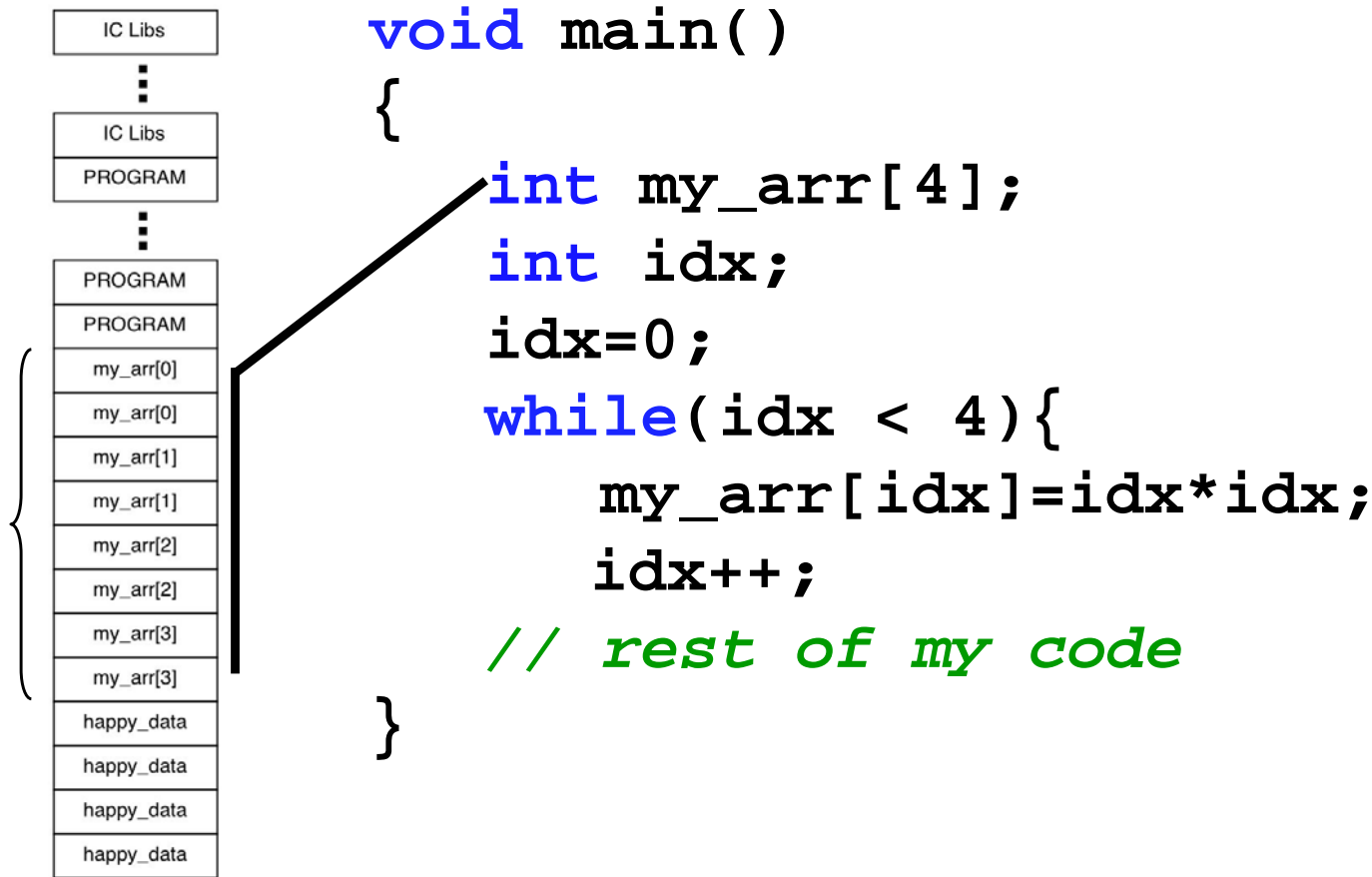
- Example

- `printf ("%d %d %d\n", sensor[0], sensor[1], sensor[2]);`

- Be **very careful** not to go out of the array bounds (otherwise a system error occurs and the processor crashes/stops).



# Arrays in Memory



# Loops with Array

```
void main()
{
    int iArray[10]; // make an array of 10 cells
    int i;
    for(i = 0; i < 10; i++) // fill it with values
    {
        iArray[i] = (i + 1) * 10;
    }
    printf("Size of array: %d\n", _array_size(iArray));
    for(i = 0; i < 10; i++) // print out the values that
                            // were filled in
    {
        printf("%d ", iArray[i]);
    }
}
```

**Output:**

Size of array: 10

10 20 30 40 50 60 70 80 90 100



# Multidimensional Arrays

- What is a Multidimensional Array?
  - Arrays that go in multiple directions
  - Example:
    - Matrix
    - Cube Storage
  - A multidimensional array is an array of arrays
    - Example:  

```
int a_matrix[3][10];
```

is an array having 3 cells, each of which is an array of 10 integers
  - Accessed the same way as an array, with one added level of reference



# Looping Thru a 2D Array

```
void main()
{
    int i2DArray[2][10]; // make an array of 2 cells, each of
                        // which is a 10 cell array

    int i,j;
    for(i = 0; i < 2; i++) // iterate through the
                        // 2 cell array (each cell is a 10 cell array)
    {
        for(j = 0; j < 3; j++) // iterate through the
                                // 1st 3 in each 10 cell array
        {
            i2DArray[i][j] = (i+1) * (j+1);
            printf("%d ", i2DArray[i][j]);
        }
    }
}
```

Output:

1 2 3 2 4 6



# Global Arrays

- Arrays can be global just like other data types
- You can read or modify global arrays using the interaction window
- Global arrays maintain their value after the program has stopped running

# Uploading Global Arrays



# Uploading Global Arrays

- Global arrays are still in memory after the program exits
  - **IC** doesn't clear the stack until next execution of program entry point
- Use “Upload Array” on tools menu in order to upload an array in text or Excel format
- You can also list global variables, functions, etc...



# Programming Example

- Create a global array:  
`int sensors[200][2];`
- Write a program that will loop through the rows of the array and put the current sonar value in `sensors[j][0]` and the ET sensor value in `sensors[j][1]`.
- Attach the sensors to the board, run the program
- Upload the array
- Paste the data in Excel and chart the data
- Compare what values of the ET sensor correspond to distances as measured by the sonar



# ET vs Sonar

```
#use "pause.ic" /* load pause function */
int sensors[200][2]; //data array

void main()
{
    int idx;
    pause();
    for(idx=0; idx<200; idx++) // start the loop
    {
        sensors[idx][0]=analog(0); // ET sensor on XBC
        sensors[idx][1]=sonar(15); // sonar on XBC
        sleep(0.05);
    } // end the loop
    beep();
}
```



# Line Following

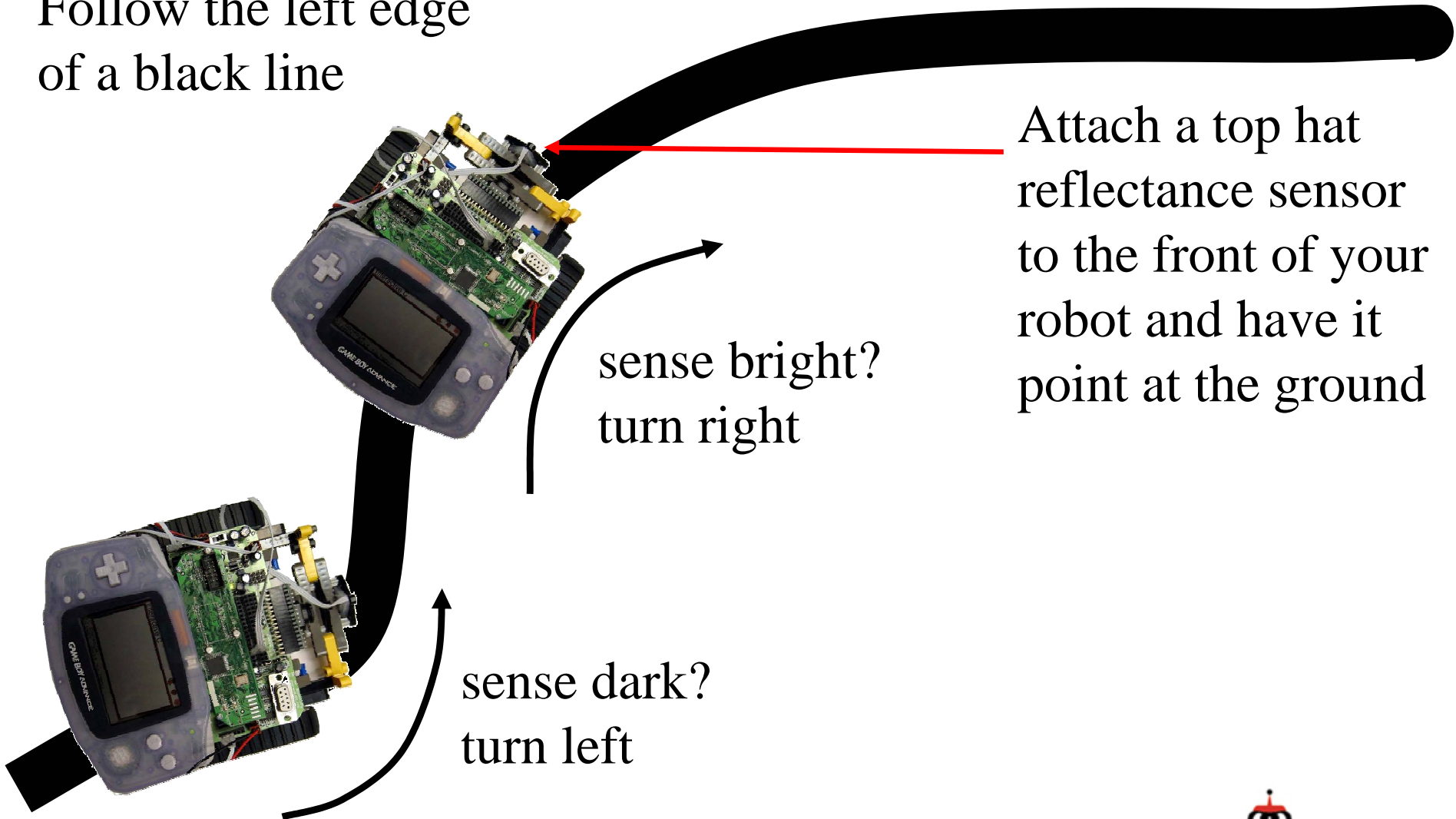
# Line Following (if time permits)

Follow the left edge  
of a black line

Attach a top hat  
reflectance sensor  
to the front of your  
robot and have it  
point at the ground

sense bright?  
turn right

sense dark?  
turn left



# Line Following (in English)

- Human: Aim the robot along the line with the sensor centered on the line (so it is seeing black). Press **start** or **A** to start; press the **stop** or **B** button to stop
- Use top hat sensor
- Program: measure the light sensor on black
  - store this measurement in the variable named **dark**
  - while the stop button is not pressed
    - if the light sensor reads less dark (i.e., something brighter is less than **dark**, perhaps **dark-10**?), then turn right (left tread forward, right stopped)
    - otherwise turn left

*Exercise:* write the code to do this and run it on your robot



# Example User Interface

```
/* This main function demonstrates some nice features of having the
robot wait until it is in the right place and the user is ready,
before actually starting to move. The functions turn_right and
turn_left are left as an exercise */
void main(){
    int black_line;           // place to store best sensor value of blk line
    while(a_button()==0) { // tell user what to do, repeatedly
        black_line = analog(6); // read blk and display it; wait for Strt
        printf("Robot on Blk=%d A if ready\n", black_line);
        sleep(.05);           // wait briefly to stabilize display
    }
    printf("Press Bwhen ready to end\n"); // so user knows what
    while(b_button()==0) { // to do to stop
        if(analog(6)< black_line - 15) {
            turn_left(); // call the function to turn the robot
        } // back towards the black line (You write this!)
        else {
            turn_right(); // turn the robot away from the line
        }
    }
    ao(); // make sure all of the motors are off
    printf("All done\n"); // let the user know you are done
}
```



# Camera Example

## Turn in Direction of Arrow

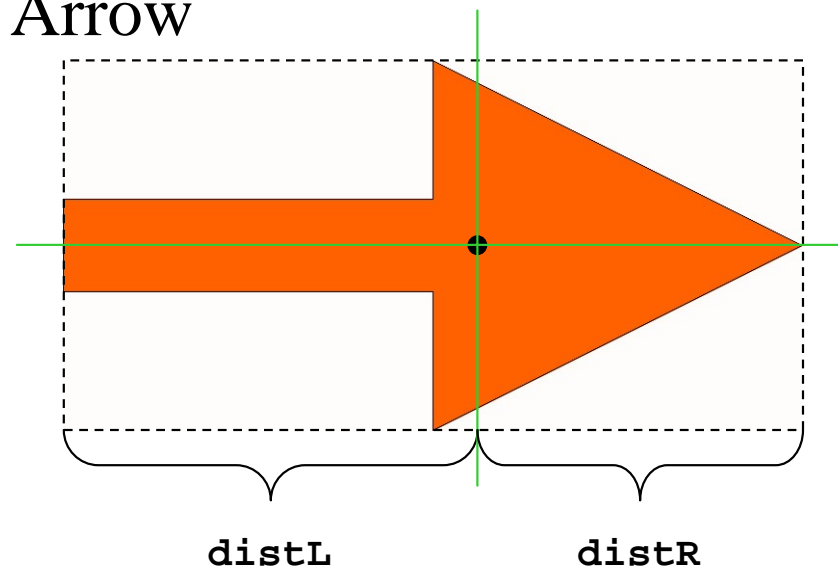




# Example Using XBC Camera Functions

## Turn in Direction of Arrow

```
// spin 360 degree in direction arrow points
// load the vision library
#include "xbccamlib.ic"
// load miscellaneous routines
// turn function defined earlier
#include "pause.ic" // defined earlier
#include "print.ic" // defined earlier
#include "turn.ic" // defined earlier
void main() {
    int distL, distR;
    init_camera(); display_clear(); sleep(1.);
    pause(); display_clear();
    while(!b_button()){ // exit with B button
        track_update(); // get fresh track data
        if(track_count(0)>0 && track_size(0,0) > 100) { // big enough
            distL = track_x(0,0)- track_bbox_left(0,0);
            distR = track_bbox_right(0,0)- track_x(0,0);
            if (distR > distL) // points left
                {print(1,5,"Turning left "); turn(360.0);}
            if (distL > distR) // points right
                {print(1,5,"Turning right "); turn(-360.0);}
            if (distL == distR) // don't see orange
                print(1,5, "Nothing there");
        } // loop when turn is done (or if nothing there)
    }
    printf("\n\nDONE\n");
}
```



# Processes

# IC: Processes

- **IC** functions can be run as processes operating in parallel (along with main)
  - The computer processor is actually shared among the active processes
  - **main** is always an active process
  - Each process, in turn, gets a slice of processing time (5ms)
- A process, once started, continues until it has received enough processing time to finish (or until it is “killed” by another process)
- Global variables are used for interprocess communications



# IC: Functions vs. Processes

- Functions are called sequentially
- Processes can be run simultaneously
  - **start\_process**(*function-call*) ;
    - returns the *process-id*
    - processes halt when function exits or parent process exits
  - processes can be halted by using  
**kill\_process**(*process\_id*) ;
- **hog\_processor**( ) ; allows a process to take over the CPU for an additional 250 milliseconds, cancelled only if the process finishes or defers
- **defer**( ) ; causes process to give up the rest of its time slice until next time



# IC: Process Example

```
#use "pause.ic"
int done;  /* global variable
            for interprocess communication */
void main()
{
    pause();
    done=0;
    start_process (ao_when_stop());
    while (!done) {
        . . . more code (involving motor operation) . . .
    }
}
void ao_when_stop()
{
    while (b_button() == 0); /* wait for B button */
    done=1;                  /* signal other processes */
    ao();                    /* stop all motors */
}
```



# Simple Process Example

```
#use "pause.ic"
int done;  /* global variable for interprocess communication */
void main() {
    pause();
    done = 0;
    start_process (ao_when_stop());
    while (done == 0) { /* loop until stop */
        motorav(3,500);
        sleep(5.0);
        if (!done) {
            mav(3,-500);
            sleep(5.0);
        }
    }
}

void ao_when_stop() /* stops motors at button press even if
                    main function is in middle of sleep
                    statement */
{
    /* wait for stop signal */
    while ((a_button() == 0) && (done == 0));
    done = 1; /* signal other processes */
    ao(); /* stop all motors immediately*/
}
```



# Exercise

- Look at the file `processtest.ic`
- Load the program into your XBC
- Run the program and press the L button to see several processes in action
- Read the program comments and code to see how to manage multiple processes
- The actual code for the 5 processes are in the file `processesforpt.ic` which is automatically loaded by a **#use**



# Loading the Bitstream Using the Xport Utility



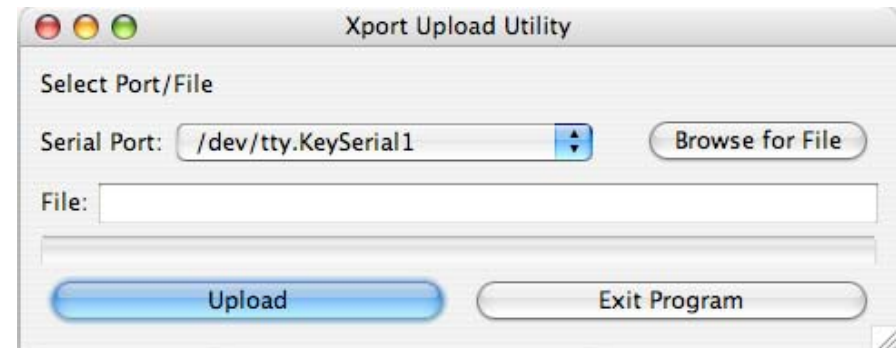
# Bits: Soft & Firm

- The XBC computes using the CPU from the Gameboy and an FPGA in the Xport (small board that plugs into the Gameboy)
- In order for your XBC to do something it needs:
  1. A **bit-stream** which configures the FPGA
  2. The **firmware**, which runs on the Gameboy and acts as interpreter/interface between the FPGA and **C** software running on the Gameboy
  3. Your **IC Programs** (software) written by you that run on the Gameboy and cause your robot to actually do something



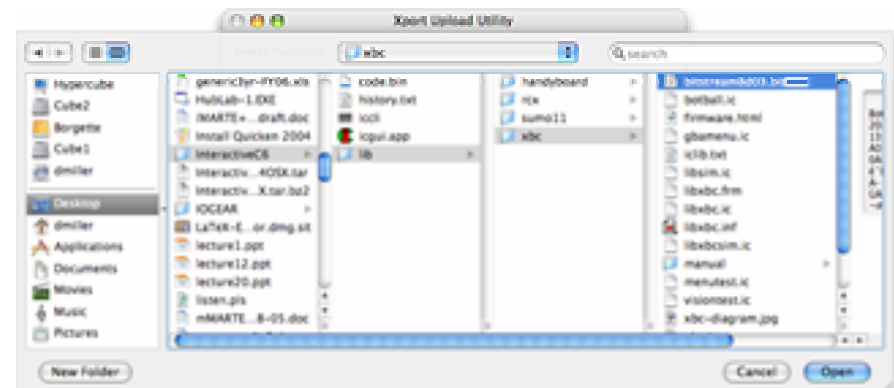
# The Bitstream

- A bitstream must be resident on your XBC for it to do anything at all, including using the serial port
- A bitstream can be updated using the Xport utility program on your CD



# Using the Xport Upload Utility (1)

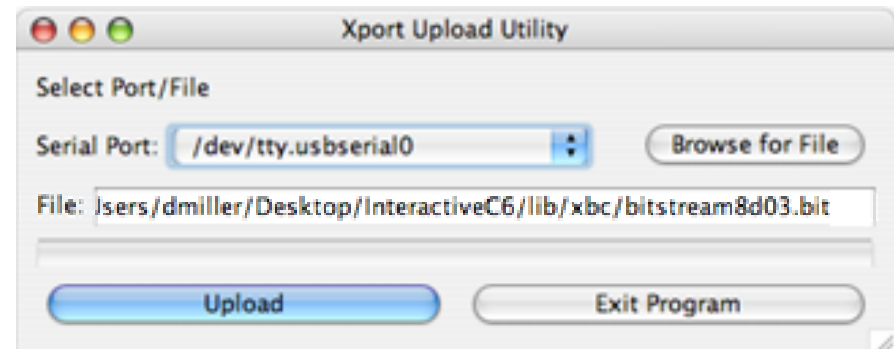
- Click on the browse button
- Navigate to the **bitstream8d0?.bit** file (current version is 8d03)
- The current version is located in the lib/xbc folder in your IC program folder



# Using the Xport Upload Utility

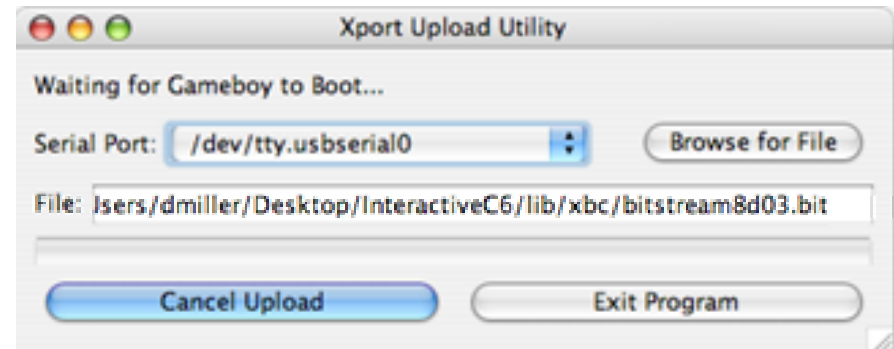
## (2)

- Make sure your computer is connected by a serial cable to the XBC
- Verify that the serial port is set correctly
- Turn off the XBC
- Click the Upload button



# Using the Xport Upload Utility (3)

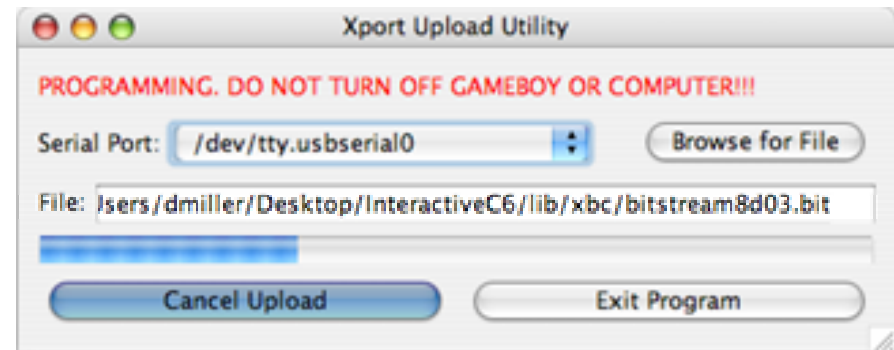
- When the utility shows: “Waiting for Gameboy to Boot...” turn the XBC back on



# Using the Xport Upload Utility

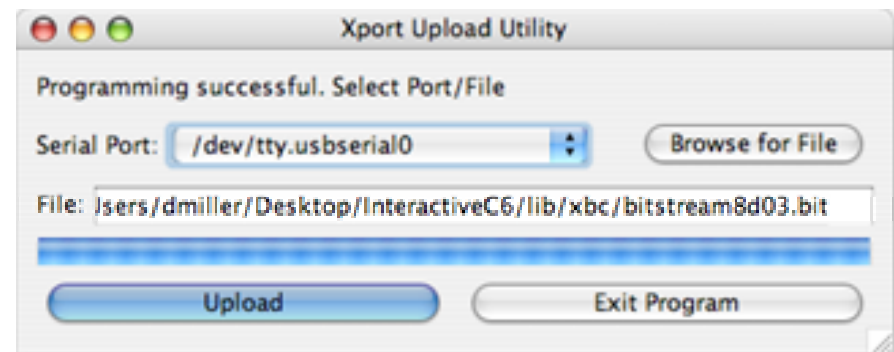
## (4)

- While the bitstream is being loaded, do not turn off the XBC, Gameboy or your computer; do not disconnect the serial cable
- The “Cancel Upload” button can safely cancel the bitstream change and leave your XBC unaltered
- Your XBC will show dots on the screen as the upload progresses



# Using the Xport Upload Utility (5)

- When the loading is complete it will take a few seconds at the end for the Gameboy to reset
- When “Programming successful” comes up on your computer you can:
  - Exit the program, or
  - Click on the “Browse for File” to...



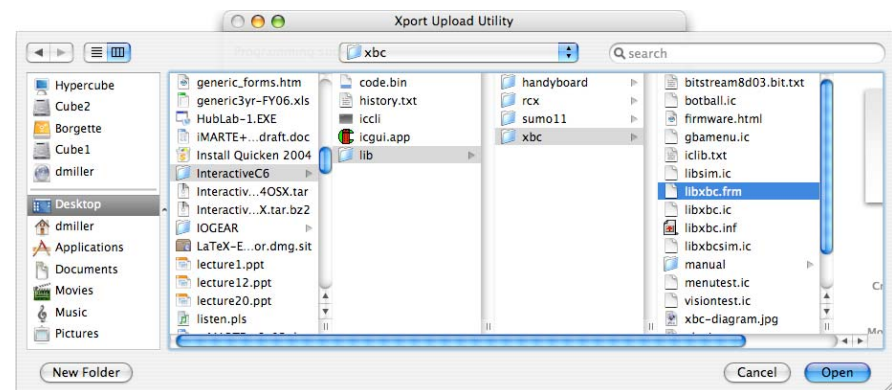
# Loading Firmware Using the Xport Utility





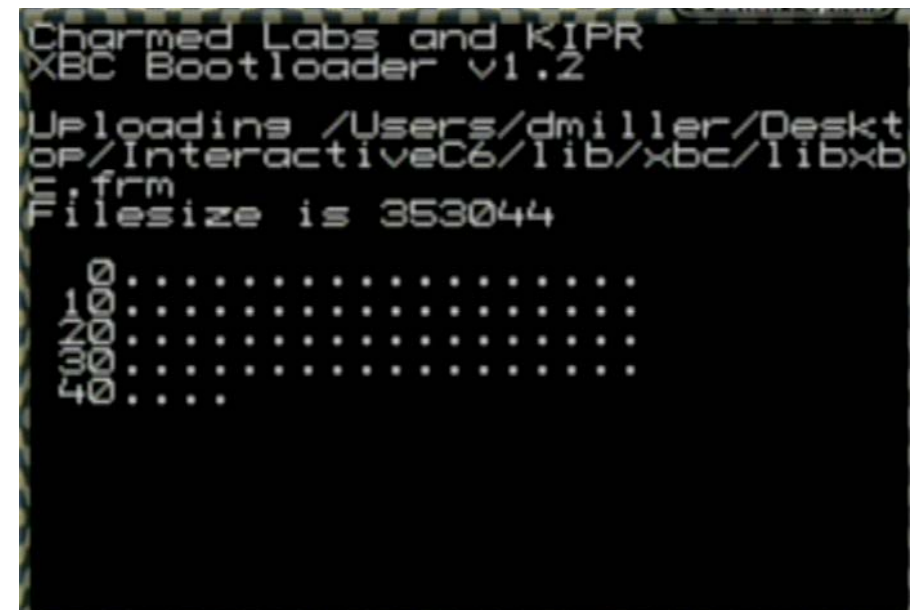
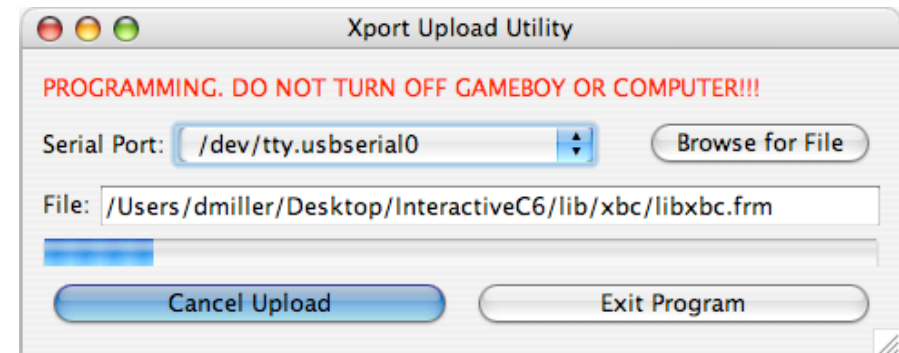
# Using the Xport Utility for Firmware (1)

- Select the **libxbc.frm** file located in the lib/xbc folder which is inside the IC program folder
- Turn off the XBC
- Click the “Upload” button
- Turn the XBC on



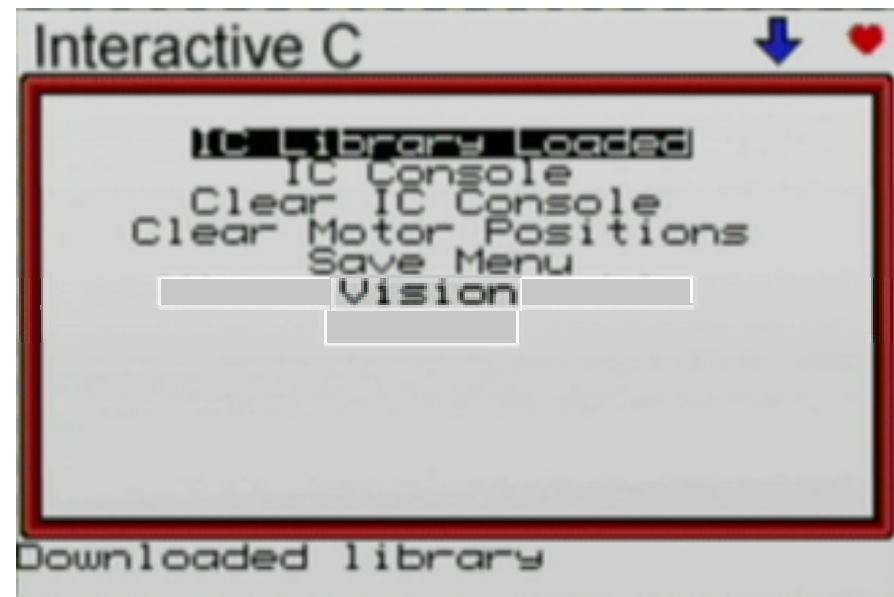
# Using the Xport Utility for Firmware (2)

- When loading the firmware do not disturb the connection
- The “Cancel Upload” button will safely cancel the firmware change and leave your XBC unaltered
- Your XBC will display progress dots on the Gameboy display



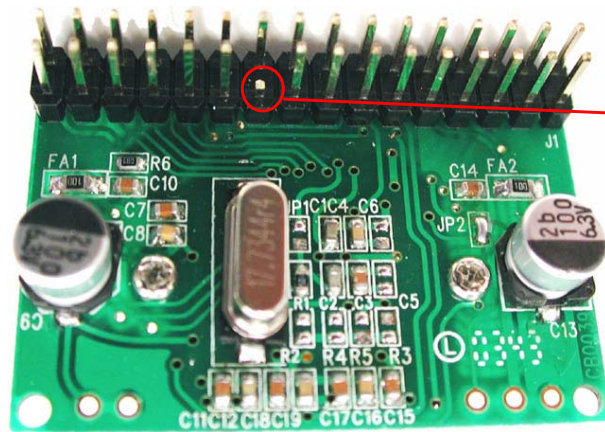
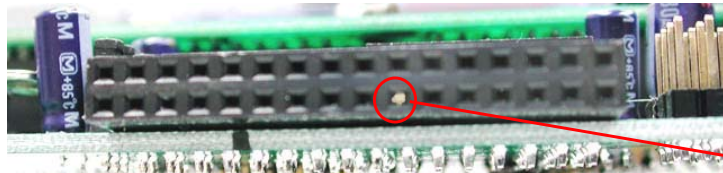
# Using the Xport Utility for Firmware (3)

- When the firmware download is complete the Gameboy will reset and the screen will look something like this -->
- Do not turn off the XBC or the Gameboy or disconnect things until either the firmware load has been cancelled or it has been completed and the Gameboy has reset



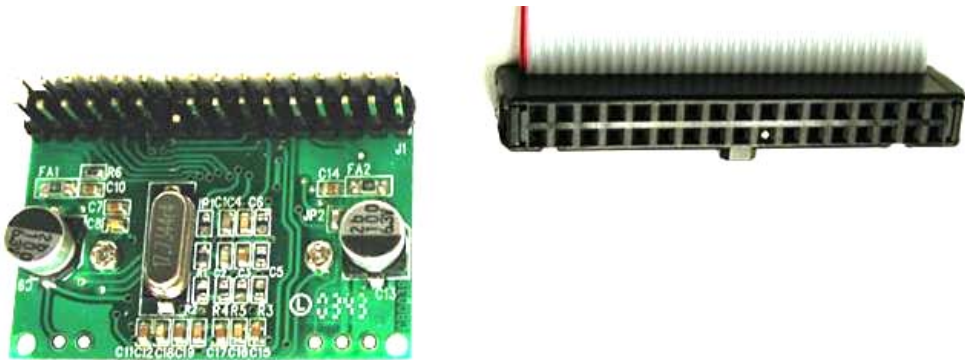
# XBC Camera Extension Cable

# XBC Camera has Keyed Connector

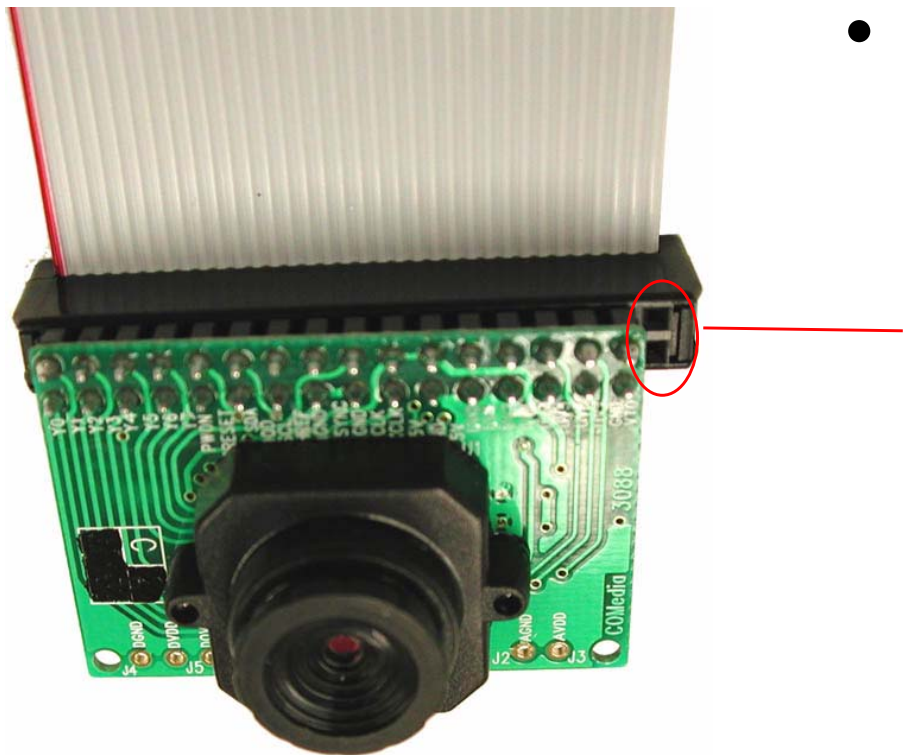


- XBC Connector has pin 19 filled in
- Camera is missing pin 19
- Camera only fits in one way (hanging down)
- Never force the camera into connector
- **Always unplug and turn off XBC before connecting or disconnecting Camera!**

# Extension Cable (1)

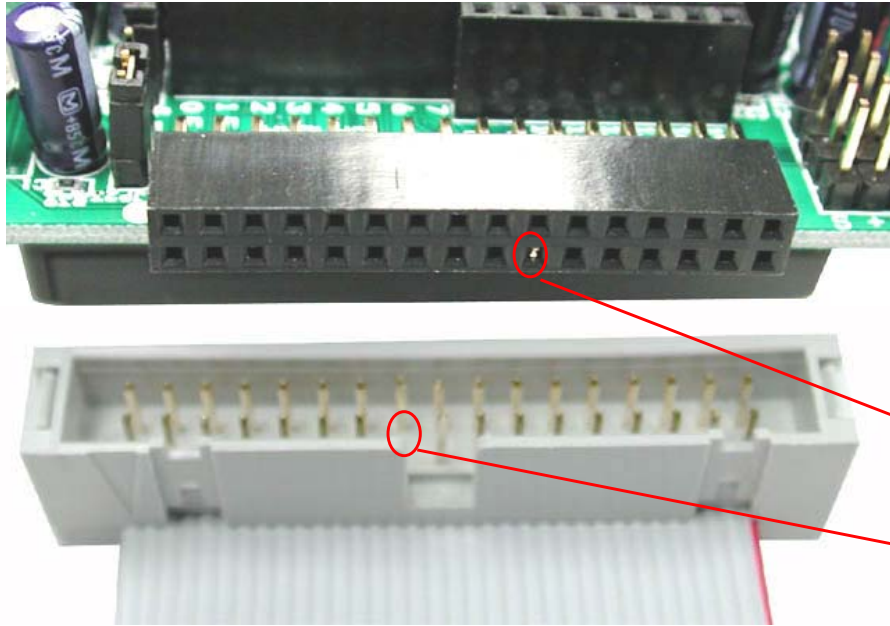


- Camera has 32 pin connector
- Cable is 34 connector
- When looking into camera lens, extra column of pins is on the right

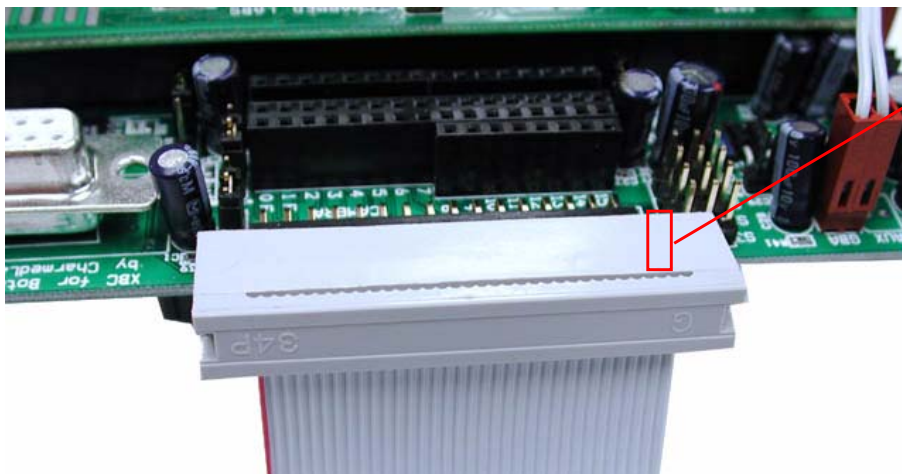




# Extension Cable (2)

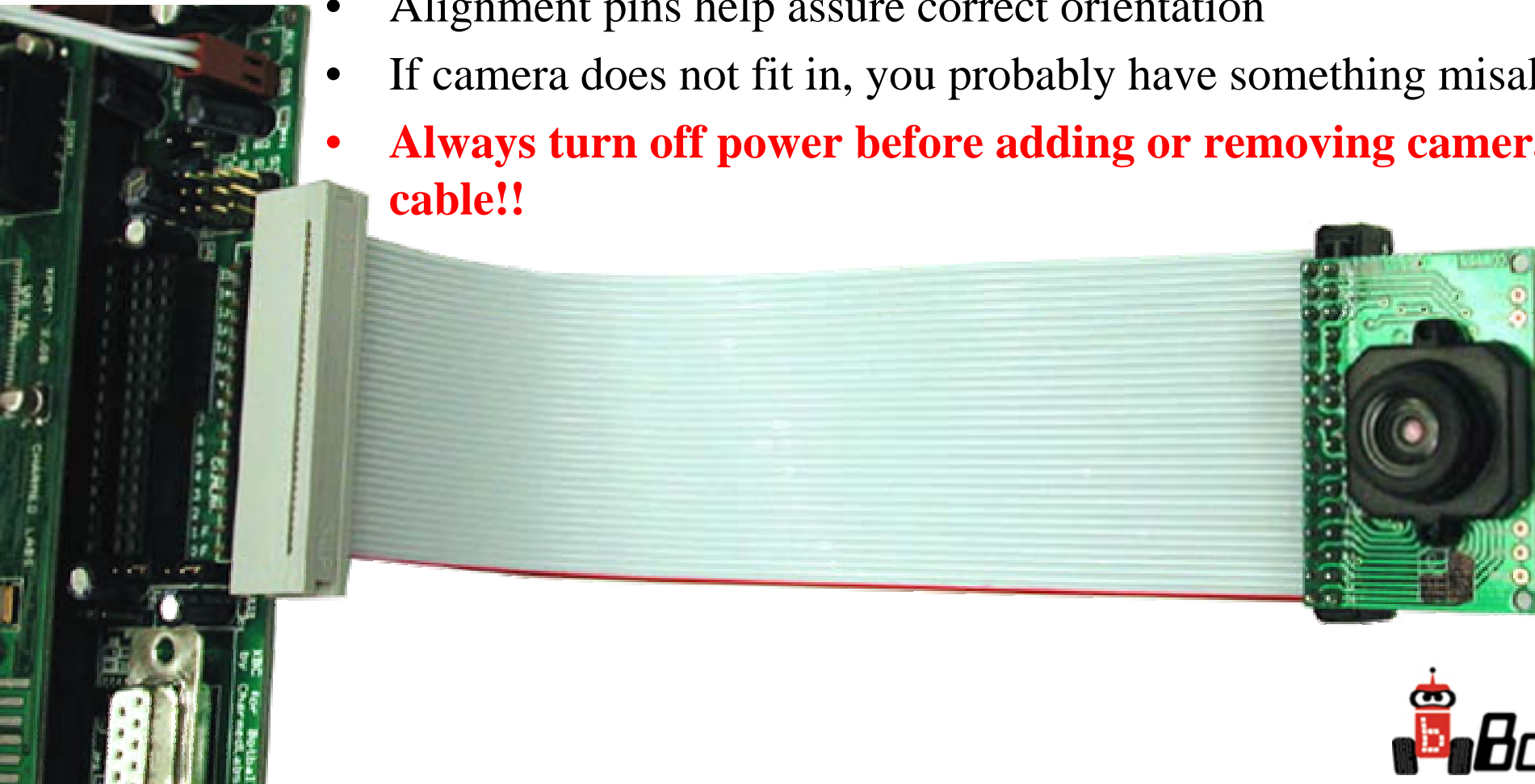


- Camera connector on XBC has 32 holes
- Cable connector has 34 pins
- Filled hole 19 & missing pin help with alignment
- Extra pins go off to the right, when looking into camera (pins are hidden by connector in this view)



# Extension Cable (3)

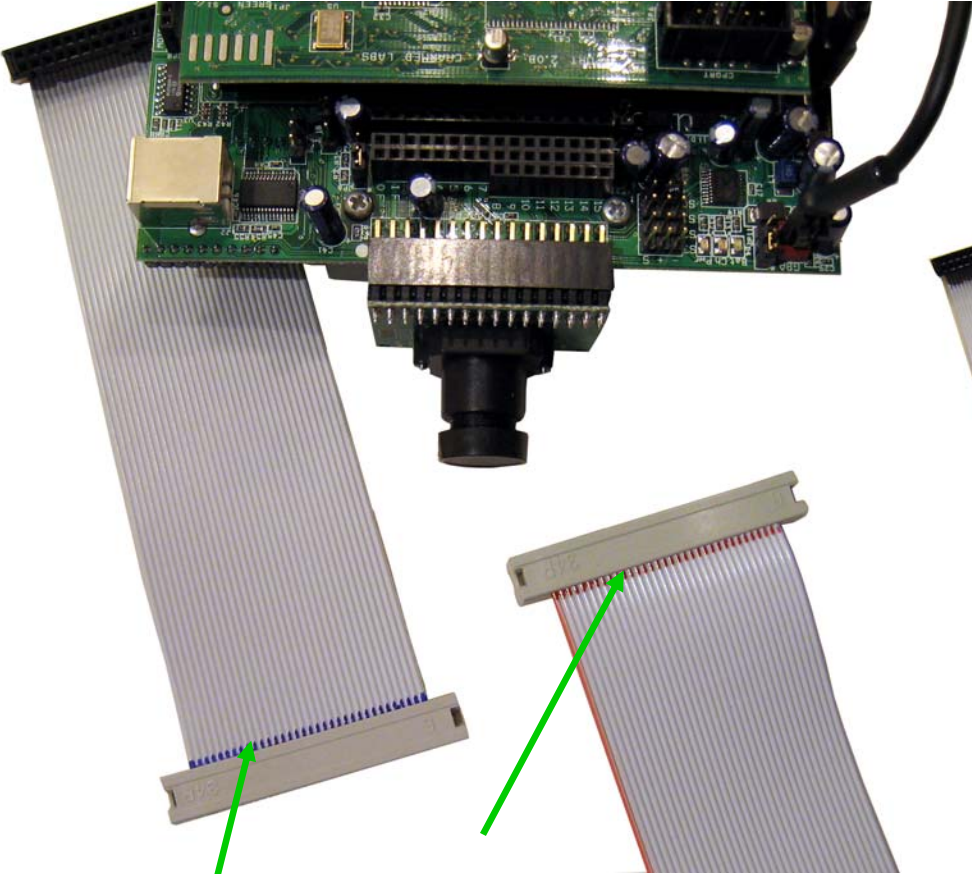
- Cable allows camera to be pointed independently of the position of XBC
- Alignment pins help assure correct orientation
- If camera does not fit in, you probably have something misaligned
- **Always turn off power before adding or removing camera or cable!!**





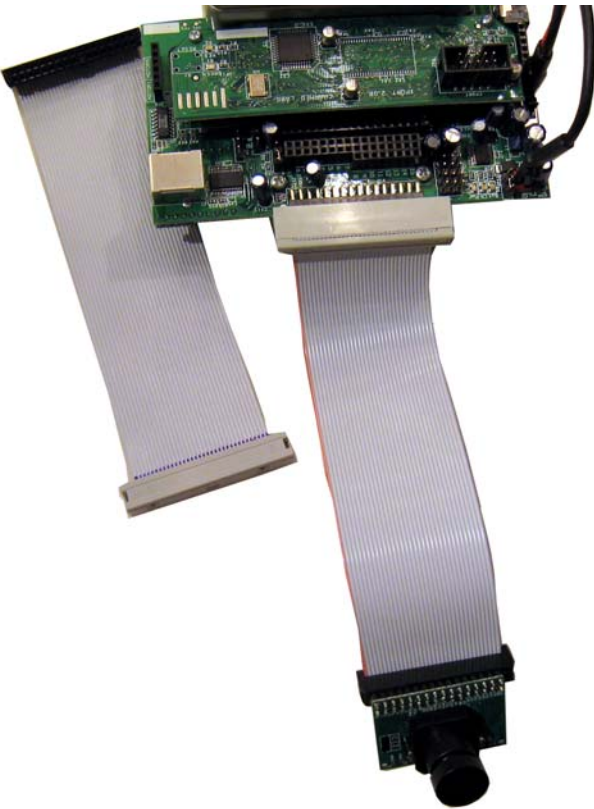
# There are 2 Types of Cables

## Cable Up (red) and Cable Down (blue)

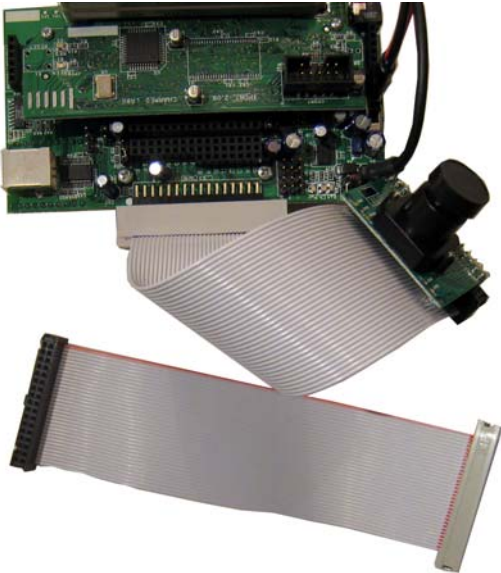


Blue and red stripes identify cable types

Cable Up (red)



Cable Down (blue)



# Challenge Exercises

# Challenges

- The next 4 slides show different challenges for you to program your robot to perform
- They are arranged in approximately increasing order of difficulty
- Take your pick
- If time allows, do more than one
- Drive in a single circle
- Turn in place 45 degrees

# Challenge 1

- Make a robot that goes forward until the range sensor returns a value, using analog12, greater than 3000.
- Have the robot go backwards until the rear bumper is pressed
- Have the robot repeat this forward/backward cycle until the B button is pressed
- We call this a Ping Pong Bot

# Challenge 2

- Have a robot maintain a constant distance (about 25 cm) from the object (your hand) in front of it
- When your hand moves towards the robot, the robot should back away
- When your hand moves away from the robot, the robot should go forwards
- Use a p-loop to control your robot's velocity
- Question/Experiment: What happens if you move your hand suddenly very close to the bot (less than three inches from the sensor)?



# Challenge 3

- Have your robot drive in a square
- Create a function, using the BEMF functions and the examples shown earlier, that has your robot turn in-place 90 degrees
- Extra challenge: Write an interface for the user that uses the up and down buttons to change the length of the sides of the square



# Challenge 4

- Have your robot drive in a circle with a diameter of about two feet
- Have the robot stop where it started (after completing a single circle)
- Extra challenge: Write an interface for the user that uses the up and down buttons to change the diameter of the circle
- What factors do you think effect the repeatability?



END

