

Getting Started with the XBC v 2.0 and IC

KISS Institute for Practical Robotics
1818 W Lindsey, Bld D Suite 100
Norman, OK 73069 USA
1-405-579-4609



Using this documentation

- This is a getting started manual.
- More complete software documentation is available under the help manual when running IC
- The software examples in this manual are for illustration only. They may not be absolutely correct or complete.



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 downloaded onto robot



To Install IC

- On a Mac OSX (10.1 and higher)
 - Double click on InteractiveC5xxx.tar.bz2 file
 - The IC5 folder can be placed in your Applications folder, or anywhere else convenient
 - Note: keep the app and the library folders in the same IC5 folder (programs you write can be kept wherever you wish)
- On Windows (Win 98 and higher)
 - Double click on InteractiveC5xxx.exe
 - IC5 will be added to your program menu
 - An IC5 shortcut will be placed on your desktop
- On Linux
 - Contact support <at> kopr.org



XBC (v2.0) Checklist

- XBC
 - XBC main board with camera
 - XPORT FPGA (small circuit board screwed on top)
 - Game Boy Advance (XPORT slides into game slot)
 - Battery box (mounted on bottom) w/Lego holes
- 9 pin M-F serial cable
- AC Adapter (little. Note: +15v center, 900ma)



XBC (V2.0) Board Setup



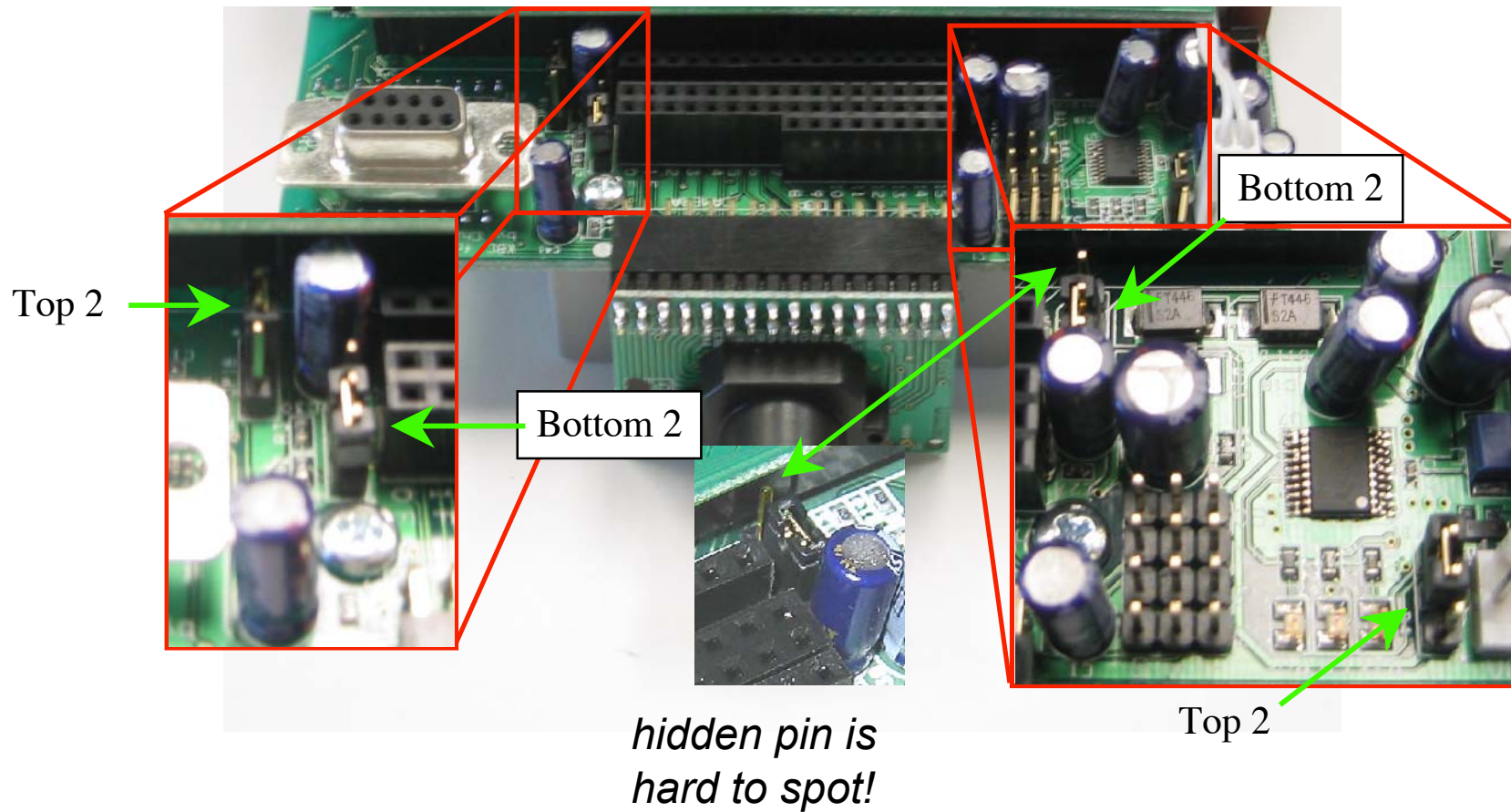
Serial Cable

XBC w camera & GBA:
7.2V NiMH 6 cell Battery



XBC (v1.0) Jumper Settings

(visually inspect to see if settings are correct)

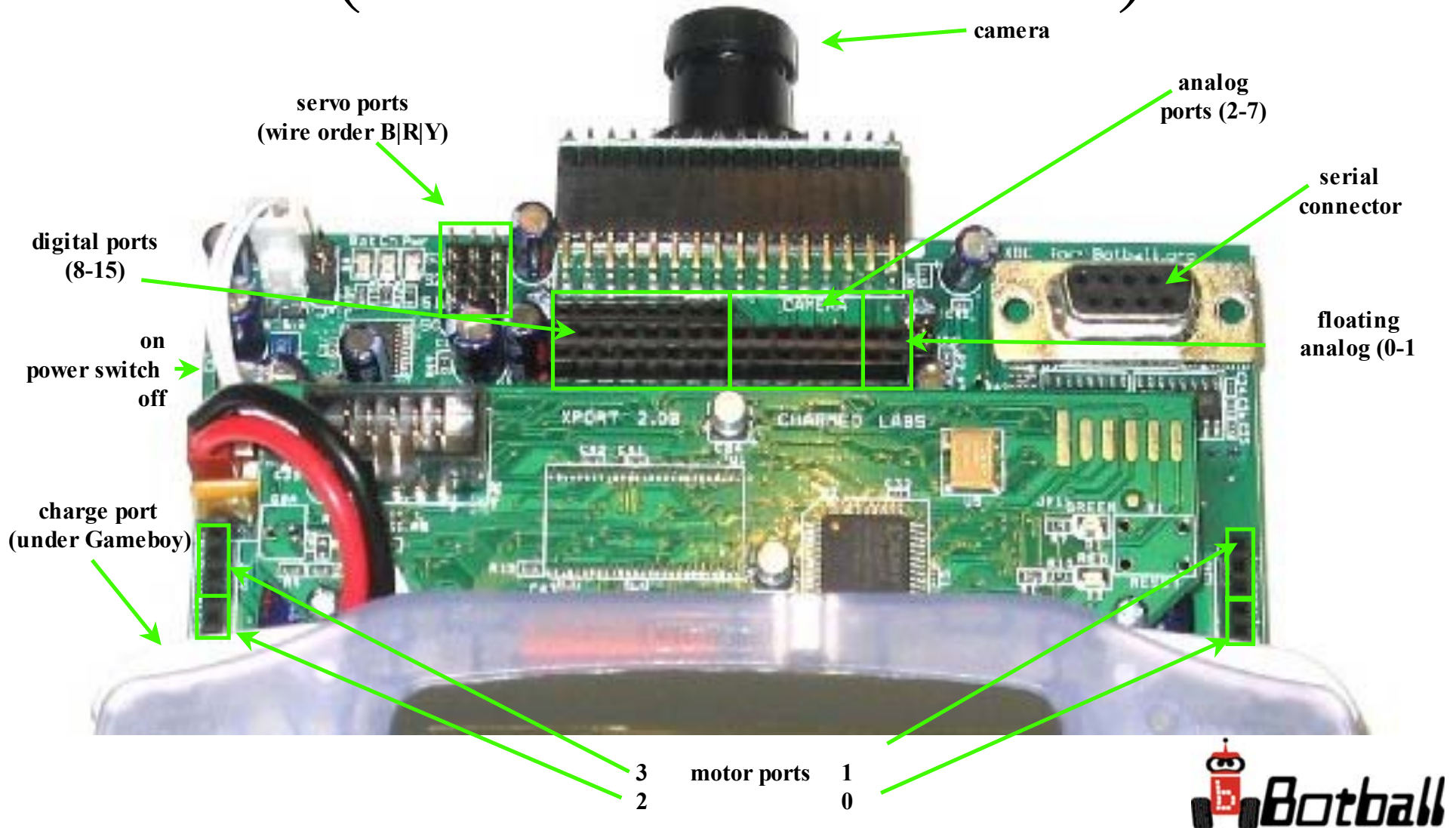


Charging XBC (v2.0) Battery Pack

- Charging is best accomplished using XBC 15v AC charger plugged into XBC with XBC turned off.
- Yellow charging light flashes when battery pack is being charged.
- Yellow charging light turns solid on when battery pack is 60% charged.
- 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.



XBC (v 2.0) (with GBA & Camera)



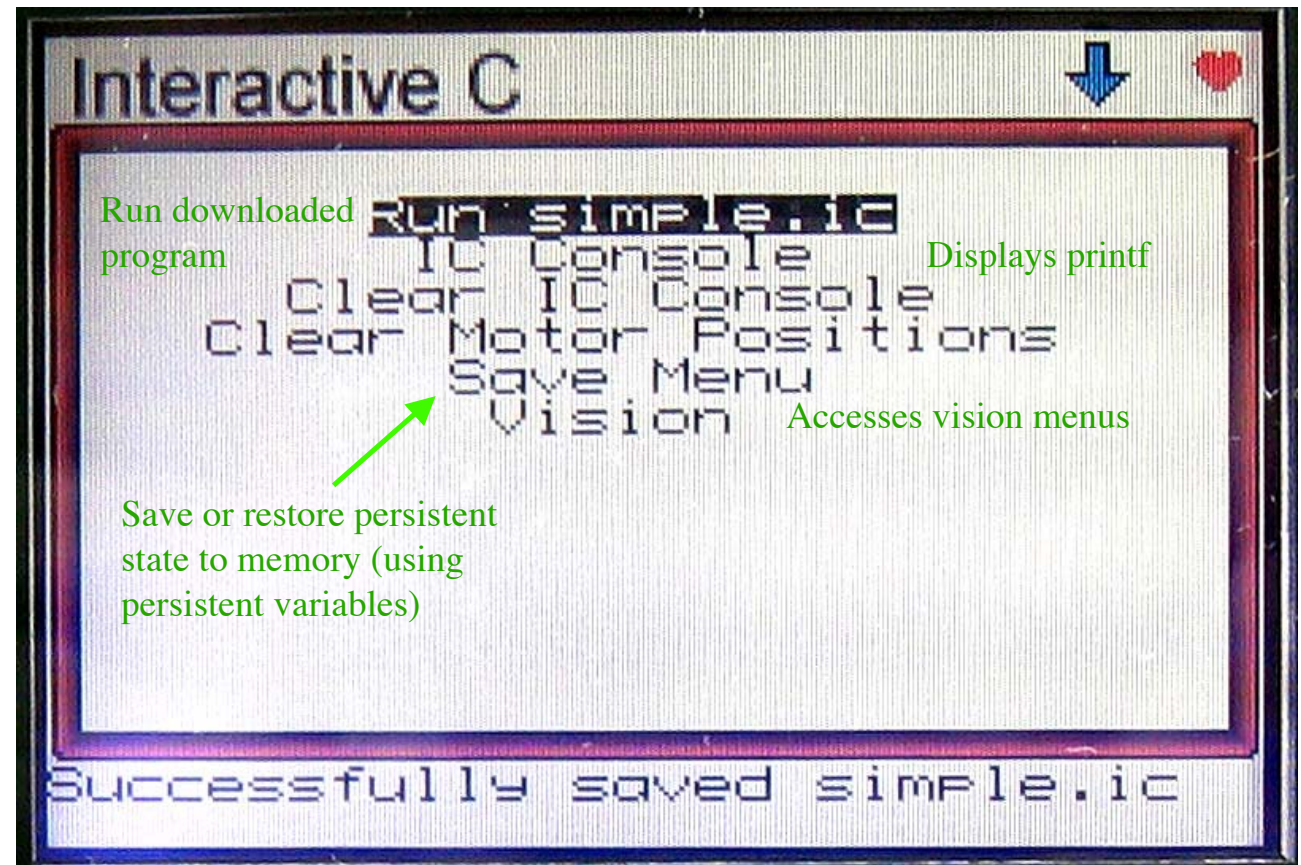
Download the Firmware

- Make sure your XBC is connected to your personal computer via 9-pin serial port cable (and a USB to serial converter if needed)
- Make sure the XBC is turned off
- Select **Change controller type** from the **Settings** menu and select the **XBC PWM/PID**
- Select the appropriate serial port
- Select **Download Firmware** from the **Settings** menu
- Follow the onscreen directions



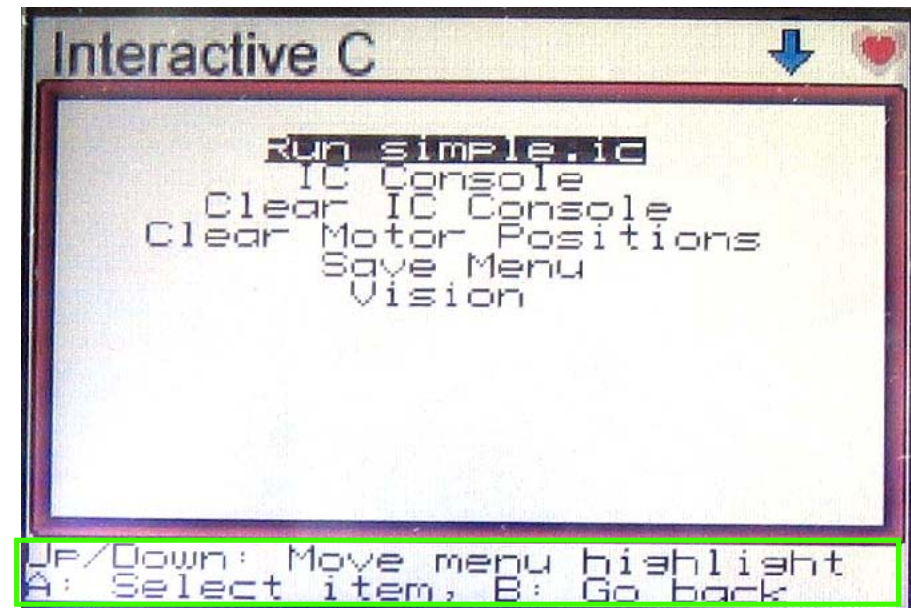
XBC Main Display

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



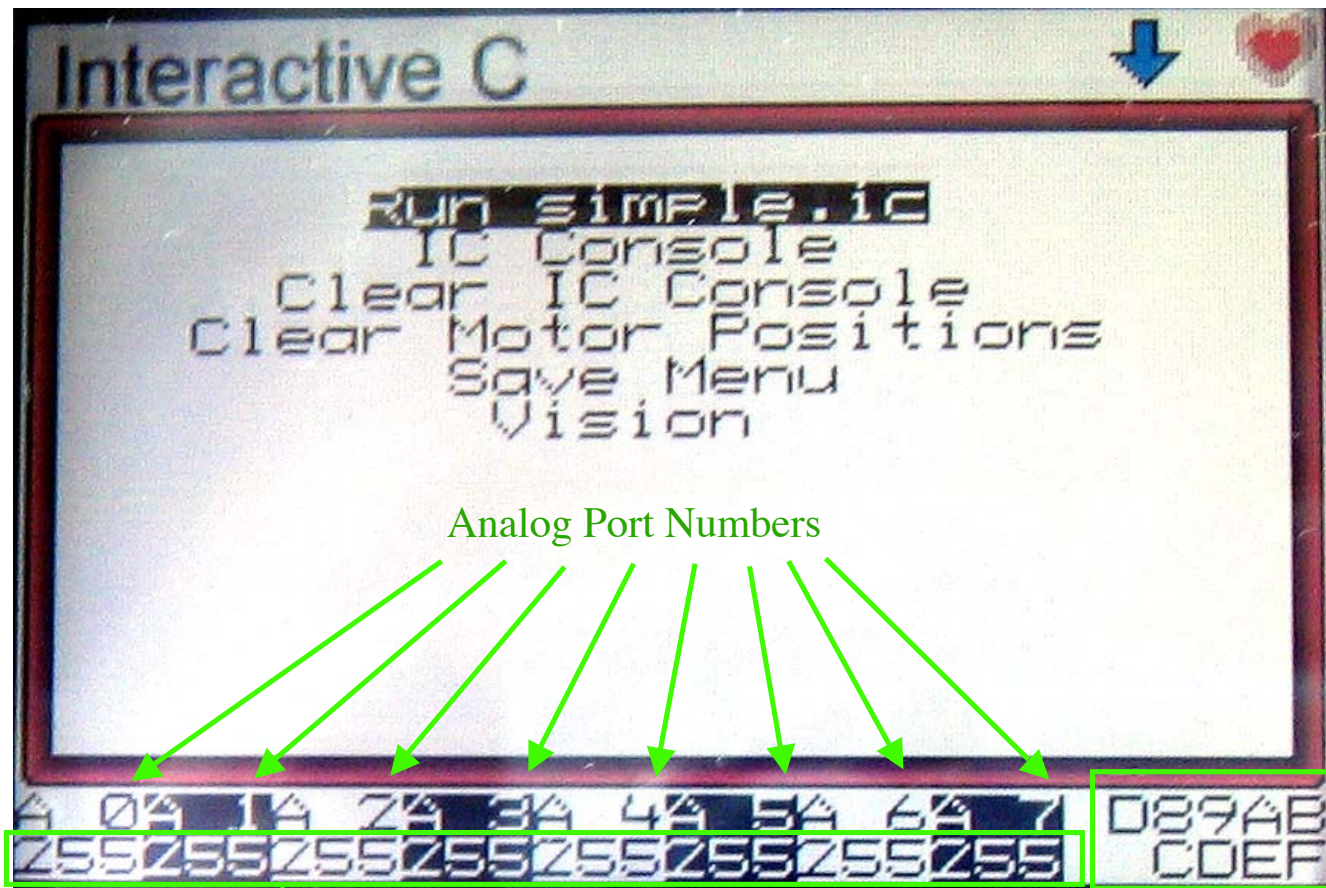
XBC Status Display

- 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

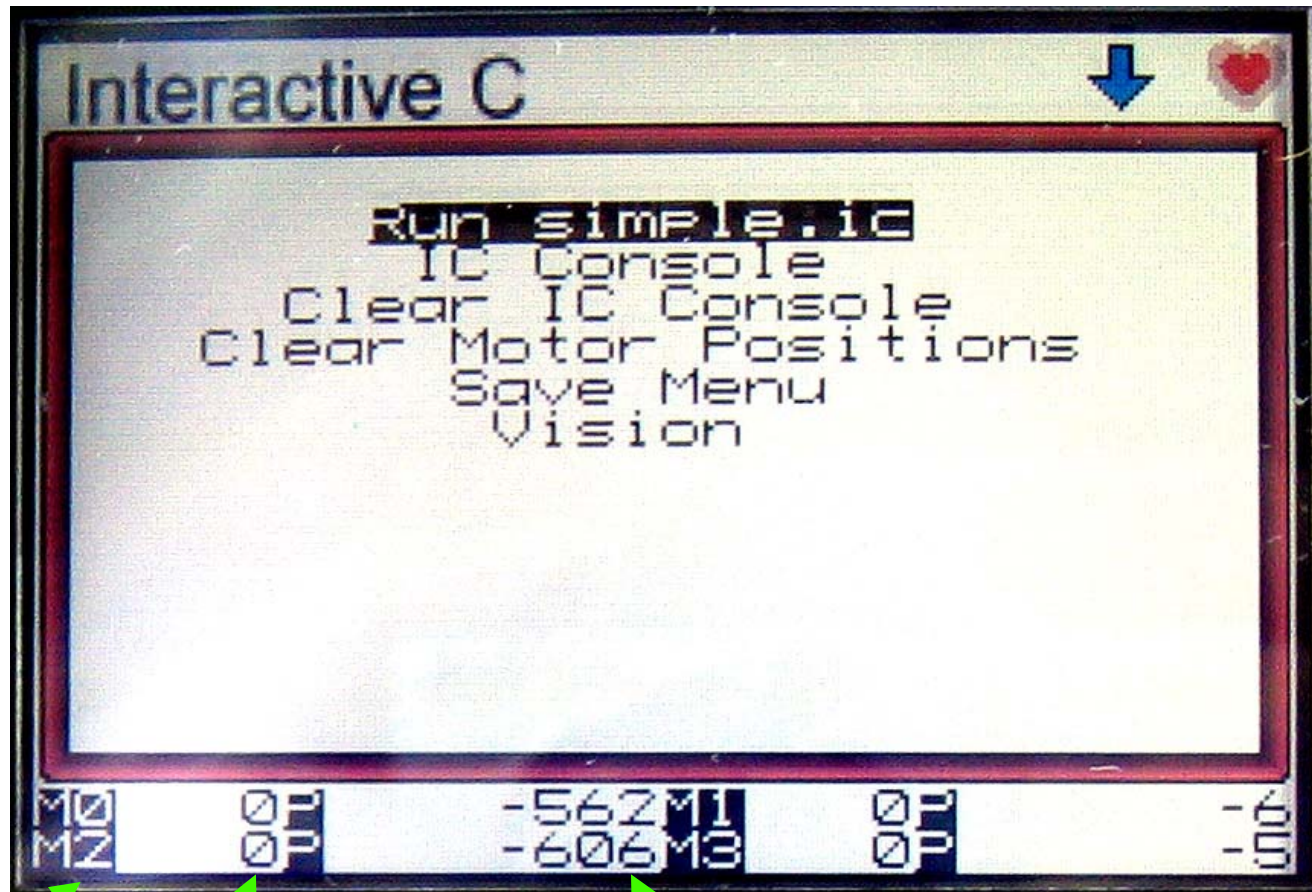


Status window

XBC Sensor Status Display



XBC Motor Status Display



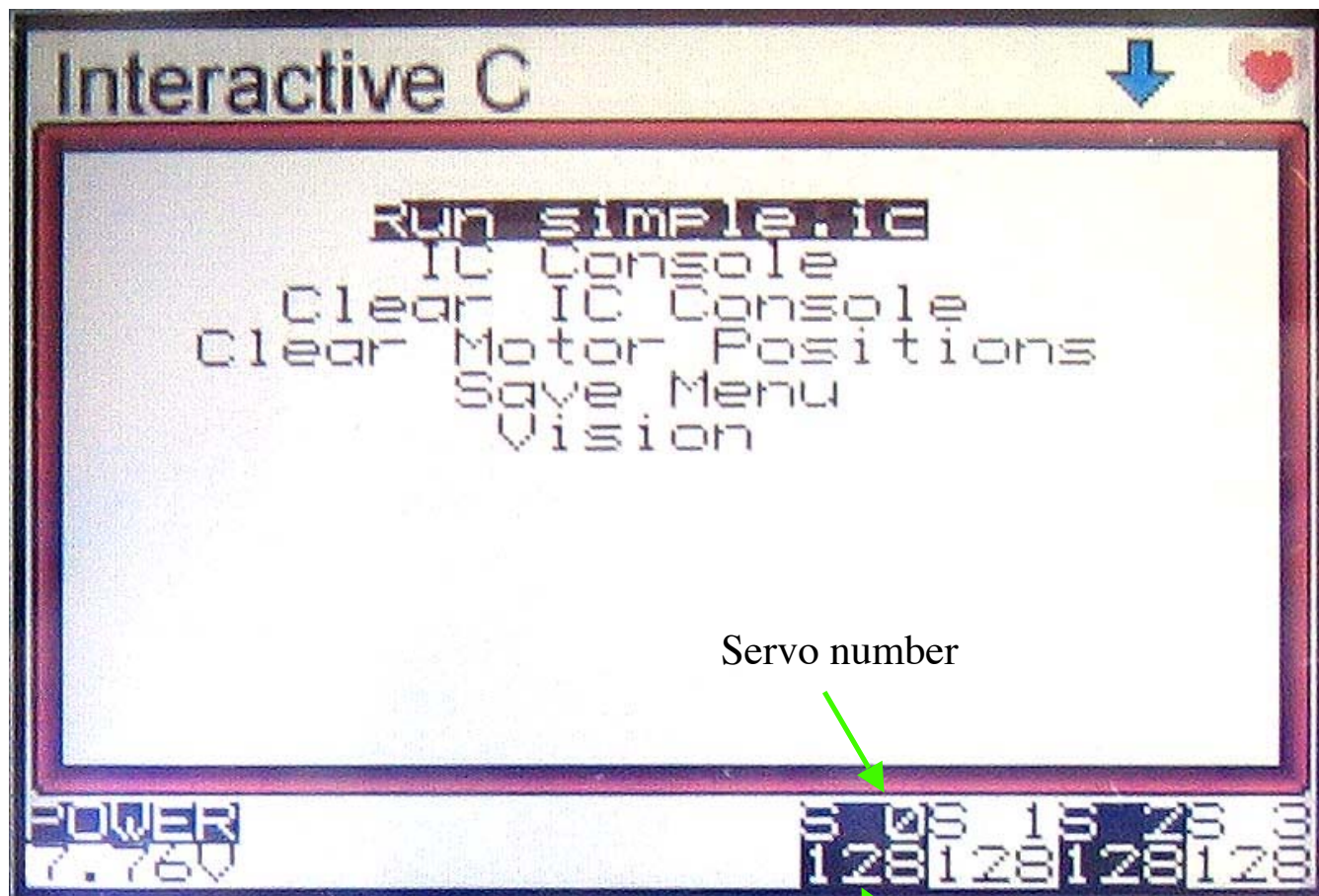
Motor number

Motor speed (0-100)

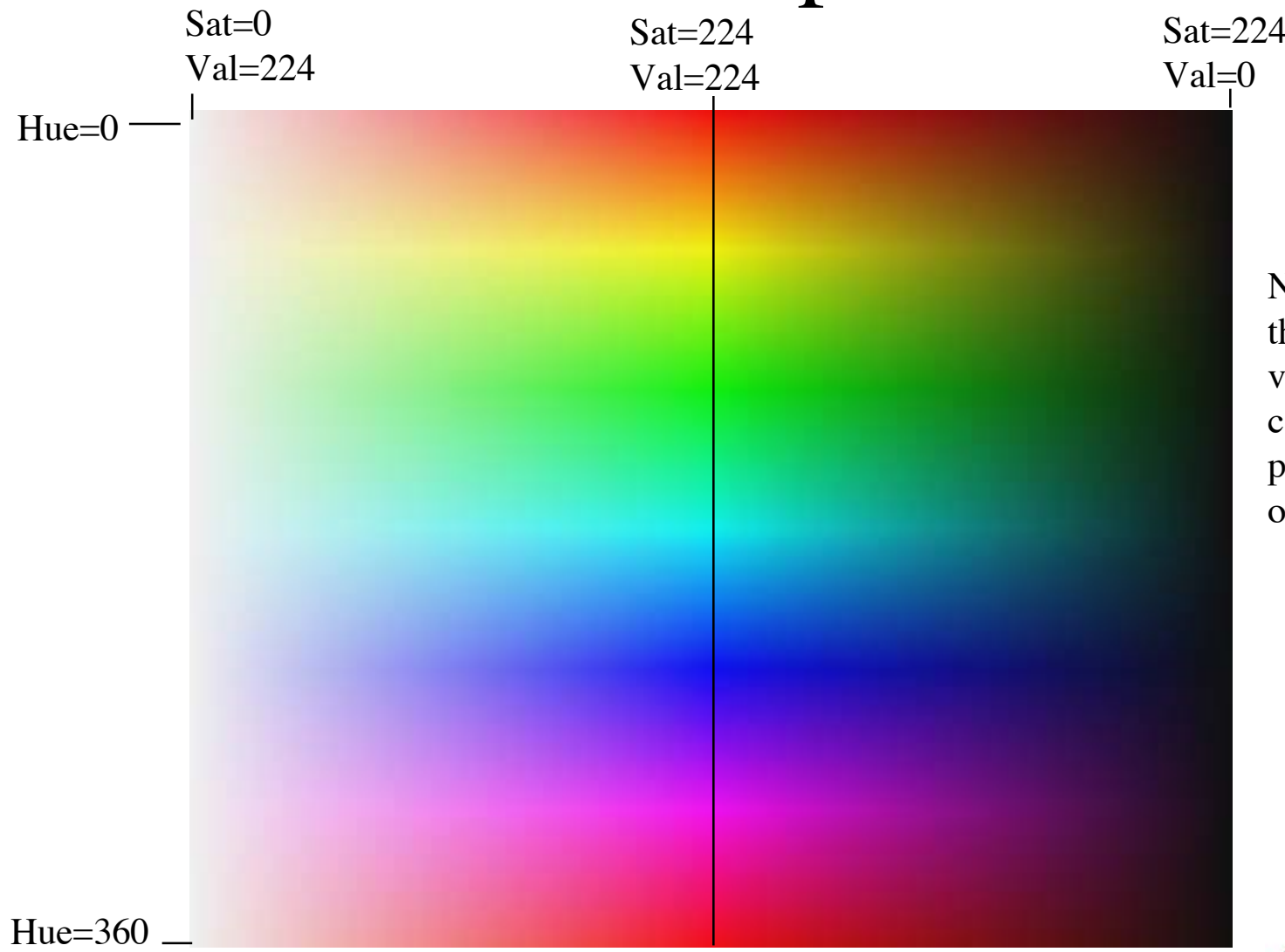
Motor position



XBC Power & Servos Display



Color Space



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



Color Blobs

- In color tracking, one selects a rectangular piece of color space and segments all of the pixels in the image that fall within that piece
- Contiguous pixels are combined into blobs
- Each blob has a size, position, number of pixels, major and minor axis, etc.
- These blobs correspond to objects seen in the image that are the desired color



Color Models

- The XBC can segment the image using three different pieces of color space (each is called a color model) simultaneously
- It can track a number of blobs from each color model
- It can display the video (raw, processed, alternating (flashes between raw & processed), and segmented into blobs) on the GBA display



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 (ie 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 towards the edges loosens the model to include less vivid colors.
 - Moving the left edge out accepts colors that are closer to pastel than what is currently accepted.
 - Move the right edge out accepts 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 Color Vision (1)

1. Parts: XBC; White piece of paper; Solid colored object
2. Turn on XBC and select (use the pad to scroll down, then press the A button when the right menu is highlighted) the *Vision* menu
3. Select *Camera* and live video and 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 about 10-20 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 across reboot, select *Save to Flash* under the *Color Model* menu. Whenever you do this both the camera settings and color models will be preserved



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
4. Press B and then select *Processed video* to see the image segmented
5. Press B and then select *Blob tracking* to see how those segments are broken into blobs

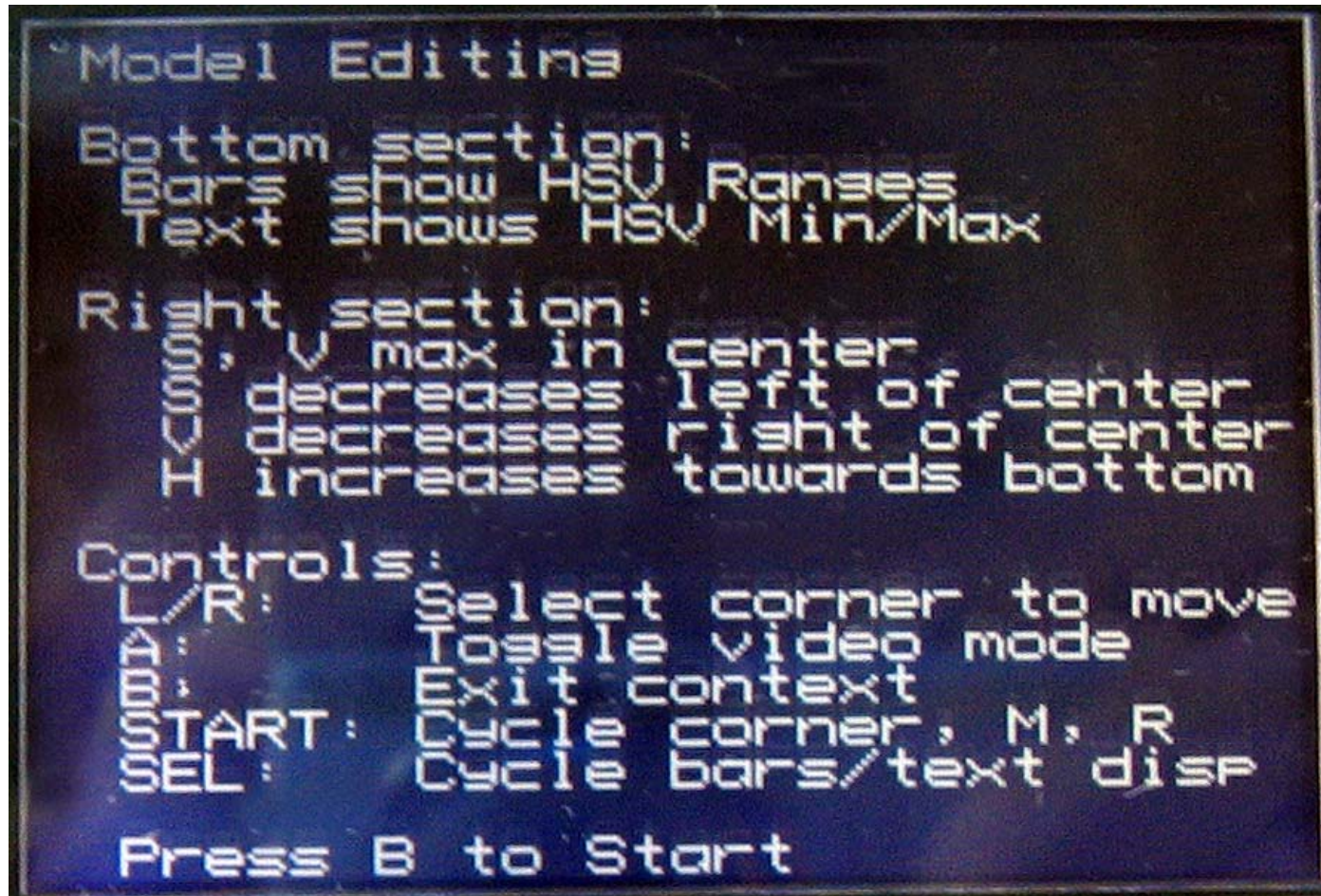


Trying Out Color Vision (3)

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 symmetrical **M**ove or **R**esize modes for the box
 2. L & R buttons switch you to a corner move (upper left or lower right) mode
 3. The D-pad is used to **M**ove the box, **R**esize, or move the corners
3. The A button cycles between live, processed, or combined video
4. Do training by
 1. opening up the S and V ranges by moving the side edges outwards
 2. opening up the top and bottom edges as far as they go (MAX_HRANGE),
 3. then moving 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 the top and bottom set up well, start moving the side edges closer to the center until you have cut out everything you want to get rid of.



Onscreen Instructions



Move Mode

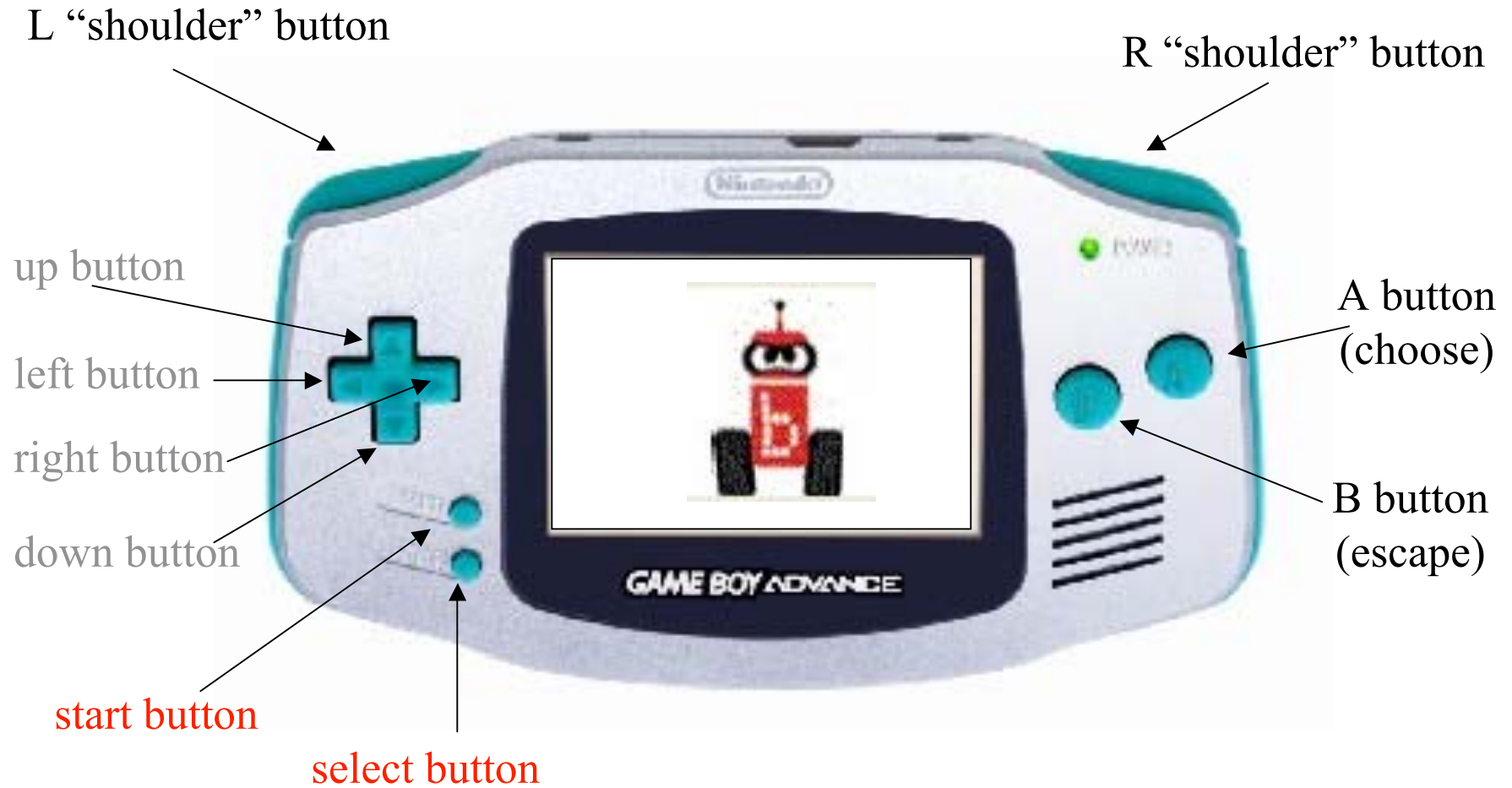
Mode Designator



Processed Video



The GBA



XBC Buttons

- 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, ...)



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()`



Compatible Button Functions

- The function **choose_button()** returns 1 when the following button is pressed
 - The **A** button on the XBC
- The function **escape_button()** returns 1 when the following button is pressed
 - The **B** button on the XBC



Useful Library Function:

sleep (*seconds*) ;

- **sleep** () delays the function's execution for an amount of time equal to the number of seconds (expressed as a float) given as an argument
- **msleep** () delays the function's execution for an amount of time equal to the number of milliseconds (expressed as a long) given as an argument. Note the time passed to msleep must either be a variable declared as type **long** or a number written as a long, e.g., 123456L, or 42L



Interacting with IC

- Click on the **Interaction** tab
- Make sure controller is connected and on
- Just type into area at bottom of IC window
- Simple expressions
`9/5;`
- Making noise
`beep();`
- Printing to the LCD screen
`printf("I'm printing!!\n");`



Helpful Features of IC

- Tools menu
 - List functions
 - List global variables
 - List loaded files
 - Upload Arrays
- Settings
 - Variable font size
 - Download firmware
- Simulator
- IC Manual



Sensors



Detachable Sensors

- Detachable sensors use a keyed connector (2 wire or 3 wire)

- Analog sensors:

- Light (XBC 2-6)
 - IR reflectance (XBC 2-6)

- Floating analog sensors:

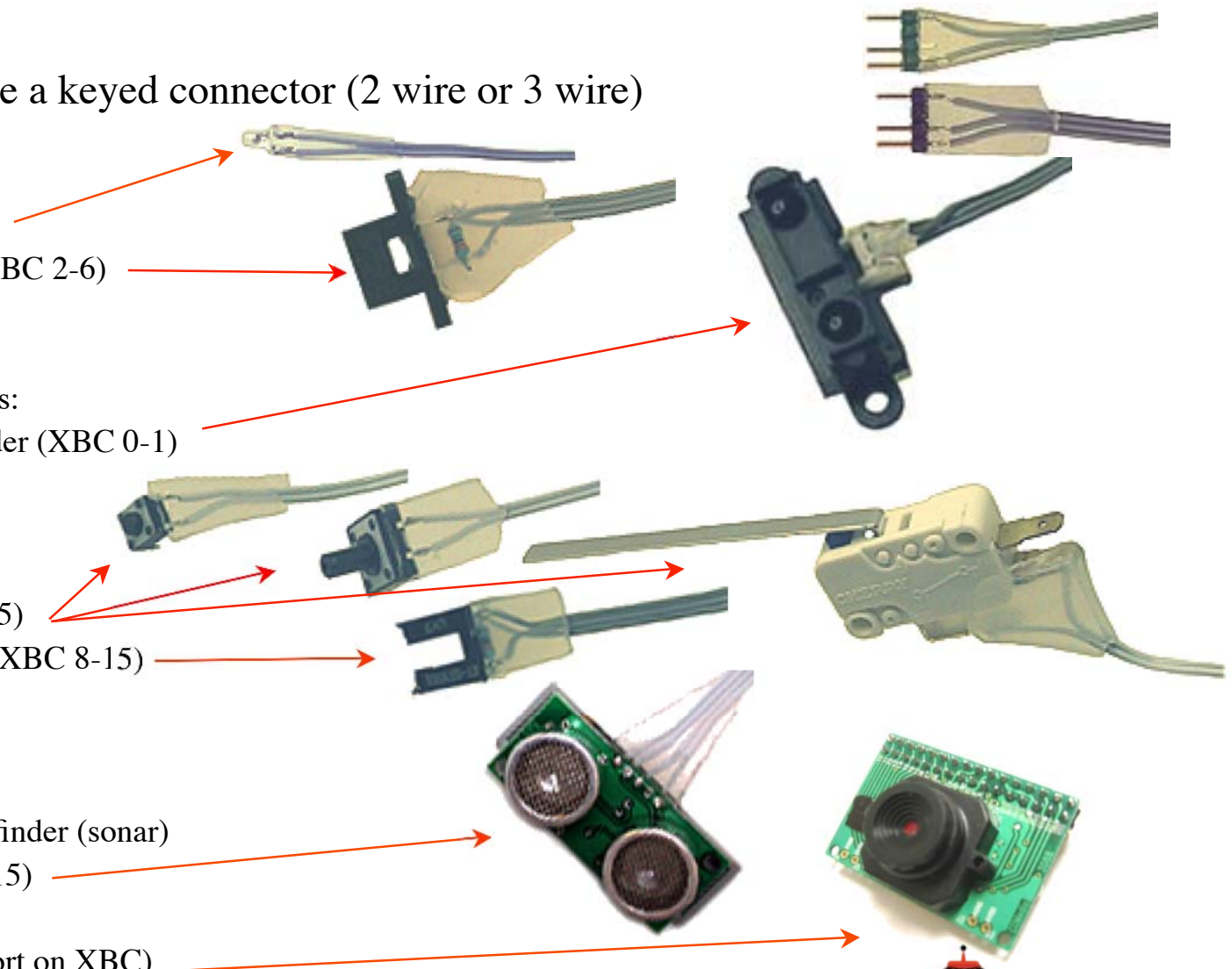
- Optical rangefinder (XBC 0-1)

- Digital sensors:

- Touch (XBC 8-15)
 - Slotted encoder (XBC 8-15)

- Special sensors:

- Ultrasonic rangefinder (sonar)
 - (XBC 8-15)
 - XBC Camera
 - (camera port on XBC)



Light Sensors

- Analog sensor
- Connect to ports
 - XBC 2-6
- Access with library function **analog** (*port#*)
 - On XBC you can also use **analog12** (*port#*) for higher resolution
- Low values indicate bright light
- High values 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
 - XBC 2-6
- Access with library function **analog** (*port#*)
 - On XBC you can also use **analog12** (*port#*) for higher resolution
- Low values indicate bright light, light color, or close proximity
- High values indicate low light, dark color, or distance of several inches
- Sensor has a reflectance range of about 3 inches

Optical Rangefinder “ET”

- Floating analog sensor
- Connect to ports
 - XBC 0-1
- Access with library function **analog** (*port#*)
 - On XBC you can also use **analog12** (*port#*) for higher resolution
- Low values 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



Ultrasonic Rangefinder (Sonar)

- Timed analog sensor
- Another Position Sensing Device
- Ultra sonic rangefinder for XBC (1 plug with 4 pins)
 - On XBC plug into any digital port
- Access with library function **sonar** (*port#*) on XBC
- Returned value is distance in mm to closest object in field of
- Range is approximately 30-2000mm
- No return (because objects are too close or too far) gives value of 32767



Touch Sensors

- Digital sensor
- Connect to ports
 - 8-15 on XBC
- 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



Break Beam Sensors

- Digital (break beam) sensor
- Connect to ports
 - 8-15 on XBC
- Access with library function **digital** (*port#*)
- 1 indicates slot is empty
- 0 indicates slot is blocked
- These can be used much like touch sensors (if the object being touched fits in the slot)
- Special abilities when used as **encoders**
 - See encoder section for more details



DC Motors

- Black gear motors resemble servos -- but they are not (identify by gray cable color)
- Motors with gray cables plug into the Handy Board or XBC
- Use Motor channels 0, 1, 2 & 3
- Full Forward: `fd(3)` ;
- Full Reverse: `bk(3)` ;
- 2/3 Reverse: `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()` ;



XBC BEMF Motor Functions

- The XBC uses intermittent measurements of the motor back EMF to estimate motor position and velocity. On the XBC, BEMF is used to implement velocity and position control.
- For the black gear motors, one rotation = about 1100 “ticks”
- The velocity measure is ticks per second (magnitude 0-1000)
- Get the position of motor 3:
`get_motor_position_counter(3);`
- Set the value of motor 3 counter to a desired value:
`set_motor_position_counter(3, 0L);`
- Move the motor backwards at a speed of 15rpm (275 ticks/sec) for 1 revolution : `move_relative_position(3, 275, -1100L);`
or `mrp(3, 275, -1100L);`
- Move a motor indefinitely at a given velocity (e.g., backwards at 1rpm (18 ticks/sec) (wow that is slow!) `mav(3, -18);` or `move_at_velocity(3, -18);`
- Stop and hold motor 3 at its current position: `freeze(3);`
 - Note the motor is powered but not moving. Use `off` when you do not need to keep a motor frozen.
- See **XBC Motors** in the Appendix for more details



Position Servos

- Plug-in order in the servo ports is black, red, yellow with black toward the left
 - Ports are 0-3 on XBC
- Enable Servos:
`enable_servos () ;`
(activates all servo ports)
- Disable Servos:
`disable_servos () ;`
(de-activates all servo ports)
- Set servo position:
`set_servo_position (2, 127) ;`
(moves servo 2 to position 127, position range is 0-255)
- Get servo position:
`get_servo_position (2) ;`
(returns an int corresponding to the position at which that 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 indicates the battery is low



Using Encoders

- For a striped encoder wheel
 - use the top hat reflectance sensor
- For a perforated encoder wheel, use the slot sensor
- XBC: Connect to ports 8-15 (*encoder#* is the port number)

enable_encoder(*encoder#*) ;

- enable an encoder only once...unless you disable it between enables

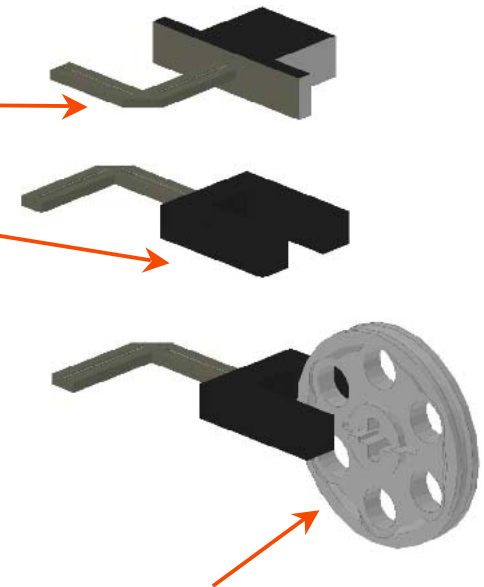
disable_encoder(*encoder#*) ;

read_encoder(*encoder#*) ;

- returns the number of transitions

reset_encoder(*encoder#*) ;

- sets that encoder's count back to 0



The slot sensor will fit across a Lego wedge belt wheel

XBC Camera



XBC Camera

1. Create a color model for channel 0 that sees something orange
2. Load **xbctest.ic** onto your XBC
3. Run the program
4. Select the vision test
5. Select the correct channel/model to see orange
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*



Example Using XBC Camera Functions

```

/* print out the following information
   for the first (largest) blob on channel 0
       x position of center, y position of center, size
   and the total number of blobs being tracked on channel 0 */
/* load required library for vision */
#include "xbccamlib.ic"
void main() {
    init_camera();           // initialize camera
    while(!b_button()) {
        track_update();      // update info from camera queue
        printf("%d  %d  %d  %d",
                track_x(0, 0), // arguments are color channel...
                track_y(0, 0), // ...and blob number (largest first)
                track_size(0, 0), // size of largest blob
                track_count(0)); // total # of blobs on channel 0
        sleep(0.1);          // sleep to get fresh data
        display_clear();      // keep the display nice
    }
}

```



#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 IC5 **#define** affects all code loaded from **#use**



Uploading Arrays



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
- Global arrays can be uploaded from the XBC and placed in spread sheet
- 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();
}
```



Uploading Global Arrays

1. Run a program to create or gather data in an array
2. After the program is finished or paused, connect XBC to computer
3. From the Tools tab choose Upload Array
4. Pick the array to upload

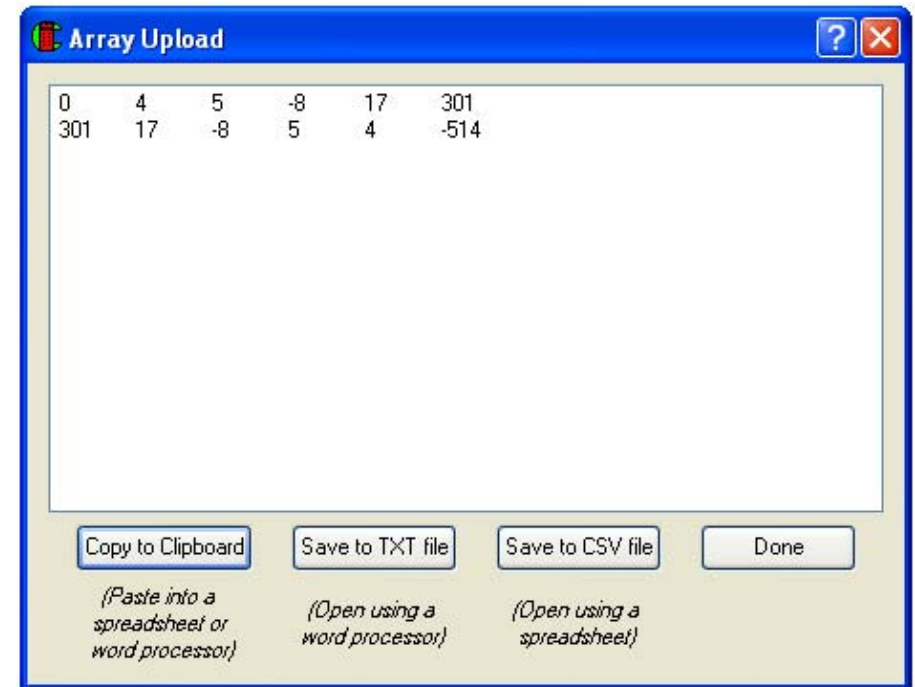


Uploading Global Arrays (cont.)

5. Click Upload Array
6. Choose how to save the data
7. View data

Microsoft Excel - test.csv

	A	B	C	D	E	F
1	0	4	5	-8	17	301
2	301	17	-8	5	4	-514
3						
4						



Notes on Uploading Global Arrays

- After uploading program to XBC, leave IC window open
- The Array must be a Global Array
- Only one array can be uploaded at a time, but you can go back to upload multiple arrays
- To open a CSV file in Excel, you must select “Text Files” file types in the Open file window



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 (escape_button() == 0); /* wait for stop 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 */
        motor(3,50);
        sleep(5.0);
        if (!done) {
            motor(3,-50);
            sleep(5.0);
        }
    }
}

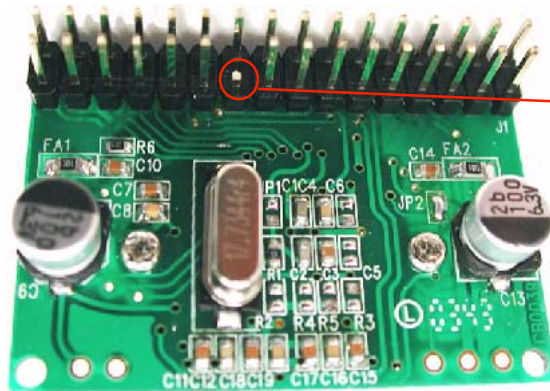
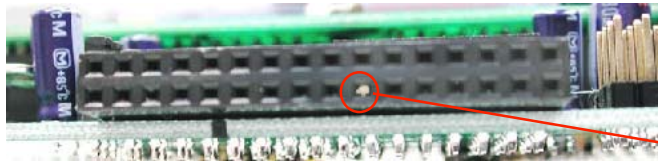
void ao_when_stop() /* stops motors at button press even if
                    /* main function is in middle of sleep
                    /* statement */
{
    /* wait for stop signal */
    while ((choose_button() == 0) && (done == 0));
    done = 1; /* signal other processes */
    ao(); /* stop all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



XBC Camera Extension Cable

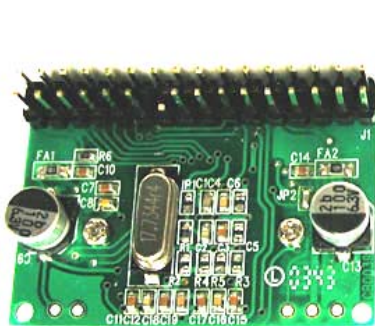


XBC Camera has Keyed Connector

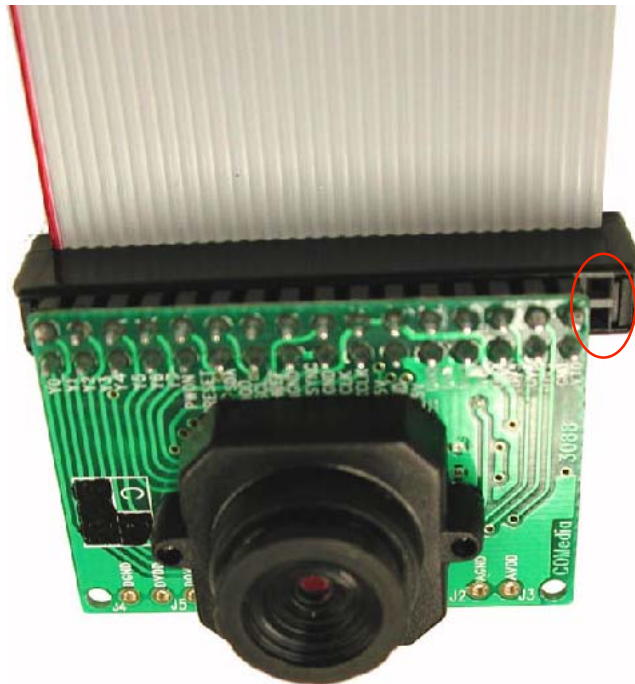


- XBC Connector has pin 19 filled in
- Camera is missing pin 19
- Camera only fits in one way (hanging down)
- Never force the camera into connector
- **Always unplug and turn off XBC before connecting or disconnecting Camera!**

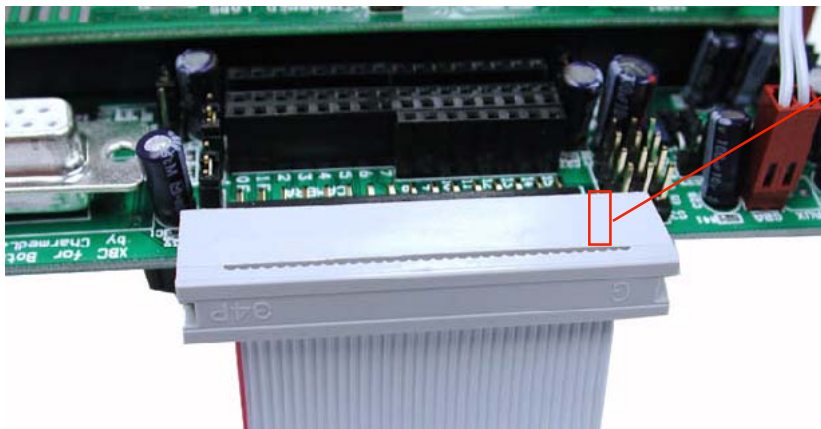
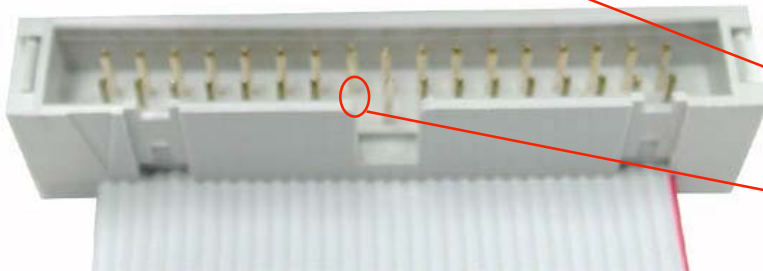
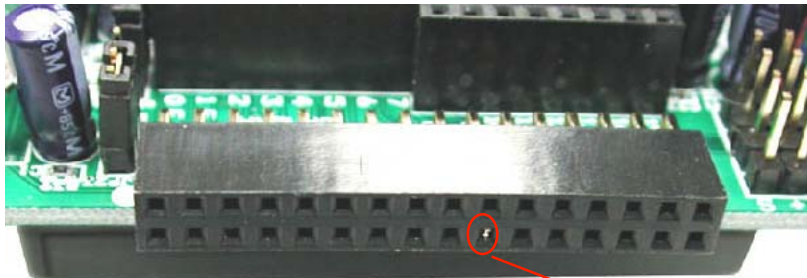
Extension Cable (1)



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



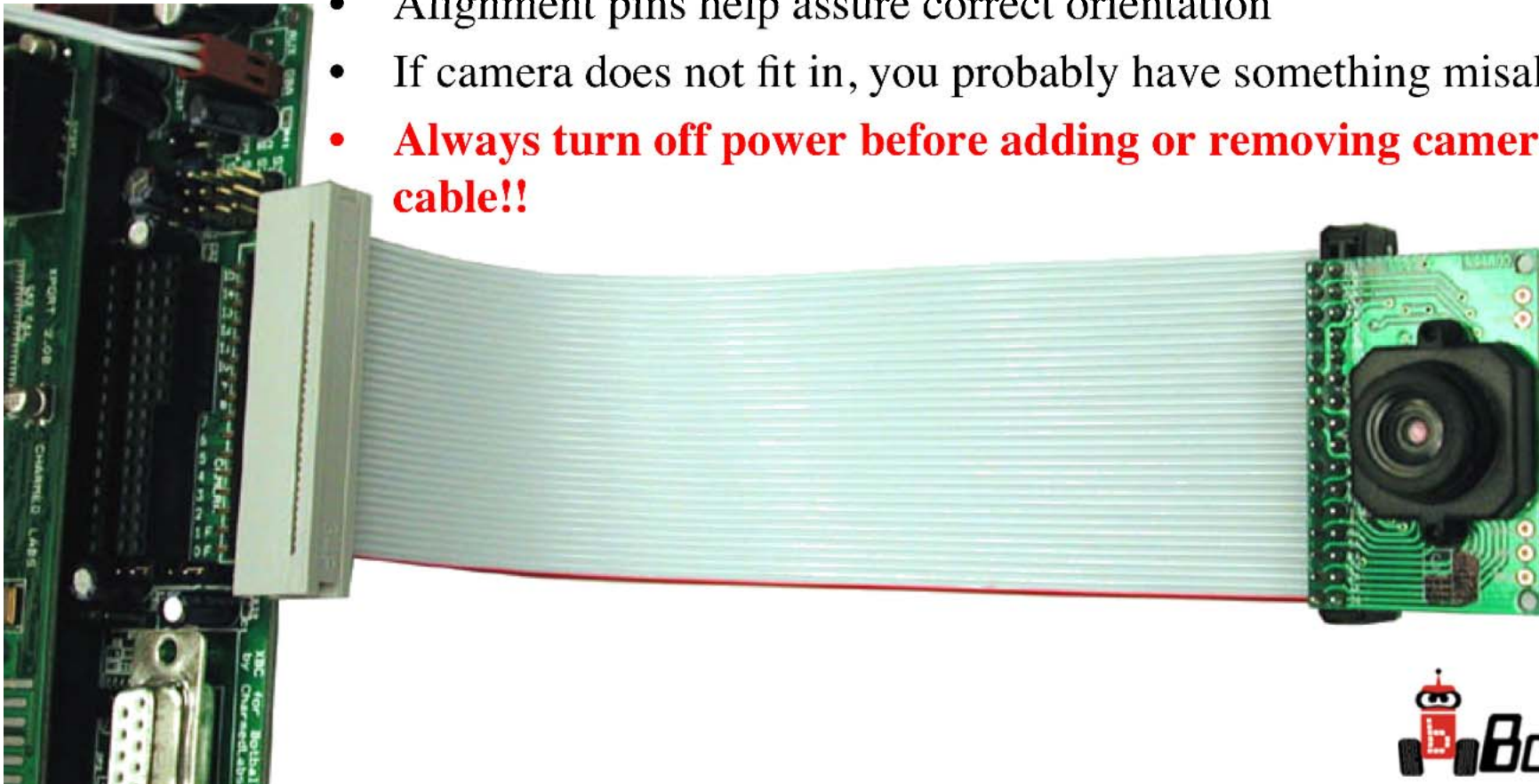
Extension Cable (2)



- 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!!**



IC5 Processor Simulator



Using Simulator

- Select processor you wish to use
- You may cancel out of serial port selection
- Open in IC the program file you want to simulate (the file you would download onto your board)
- Click on the simulate button
- After simulator has loaded, press execute button to run your program
- Click on button or sensor inputs to simulate input to robot
- View motor speeds, servo positions, globals and display as program executes.
- Pause button will halt program execution (to allow time for examining or changing state)
- Hit execute again to resume
- Clock and sleep functions operate on normal time
- The speed of CPU functions (e.g., looping through statements) depends on the speed of your computer



Simulator Limitations

- No camera input
- Many functions that depend on BEMF feedback will not operate as they do on the robot
- Timing of non-clock dependent operations differs from robot
- Simulator library functions may not be quite as up to date as actual processor libraries

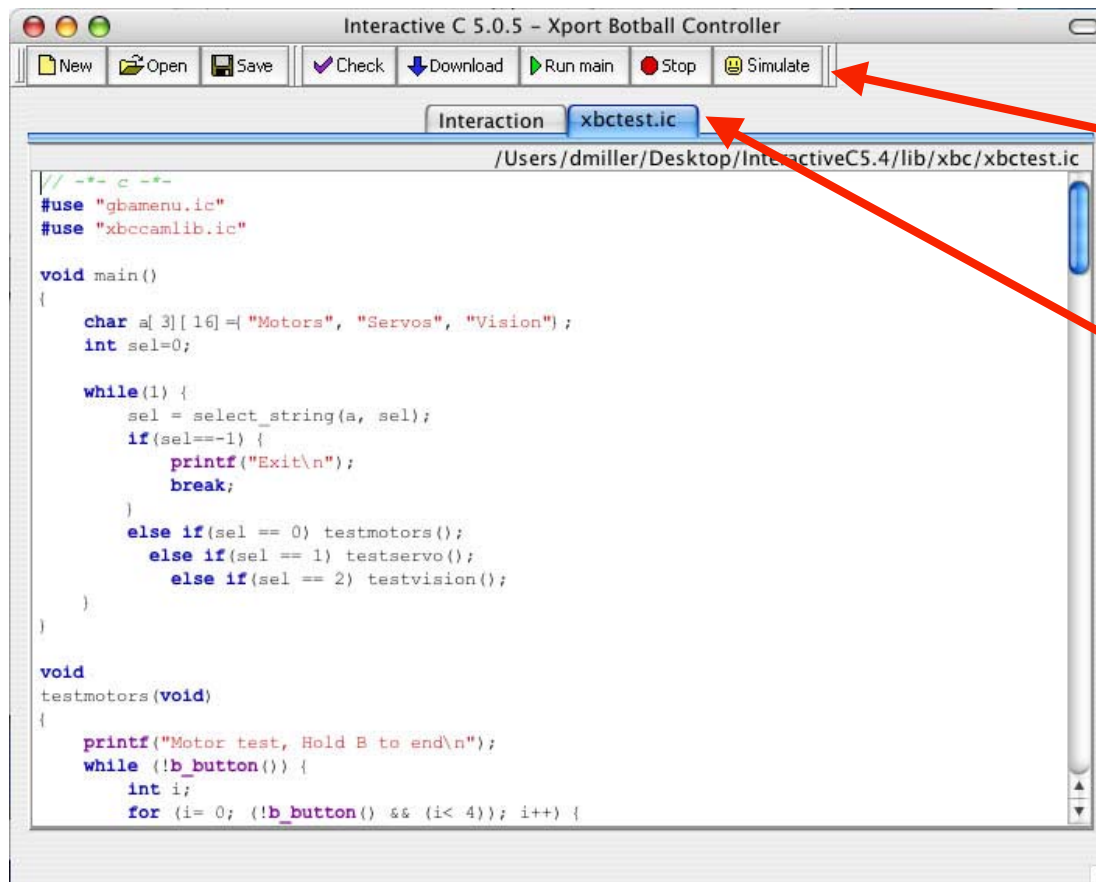


Simulator Benefits

- Allow testing of many aspects of a program's logic without needing robot hardware
- Many people can work on software and debug and test their programs at once
- Excellent software teaching tool



Simulator Example (1)

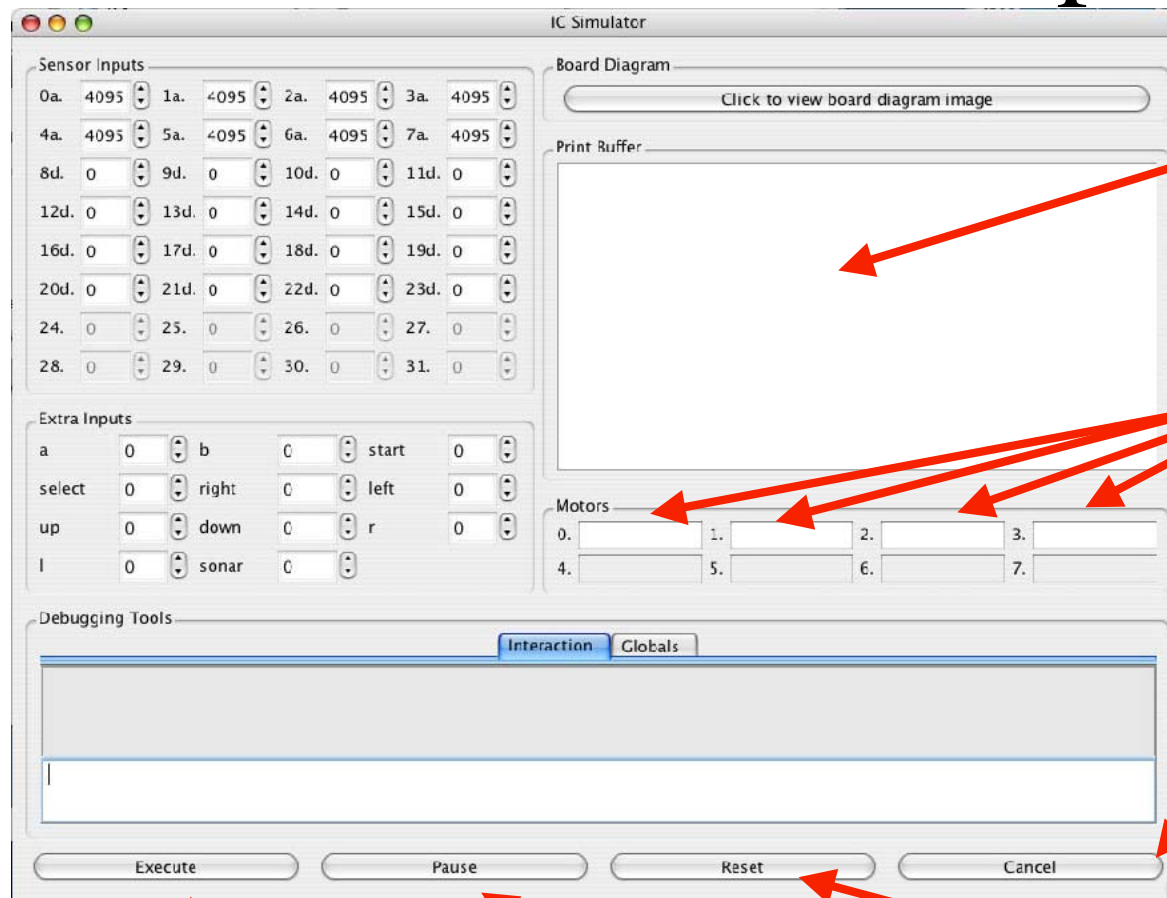


2.) Then press the **Simulate** button

1.) Open the file you want run in the simulator



Simulator Example (2)



Print statements
show up here

Motor speed settings
show up here

Cancel stops the
sim and brings you
back to the normal
IC window

Press execute to
start the program

Pause stops the program, Execute
will restart it where it was

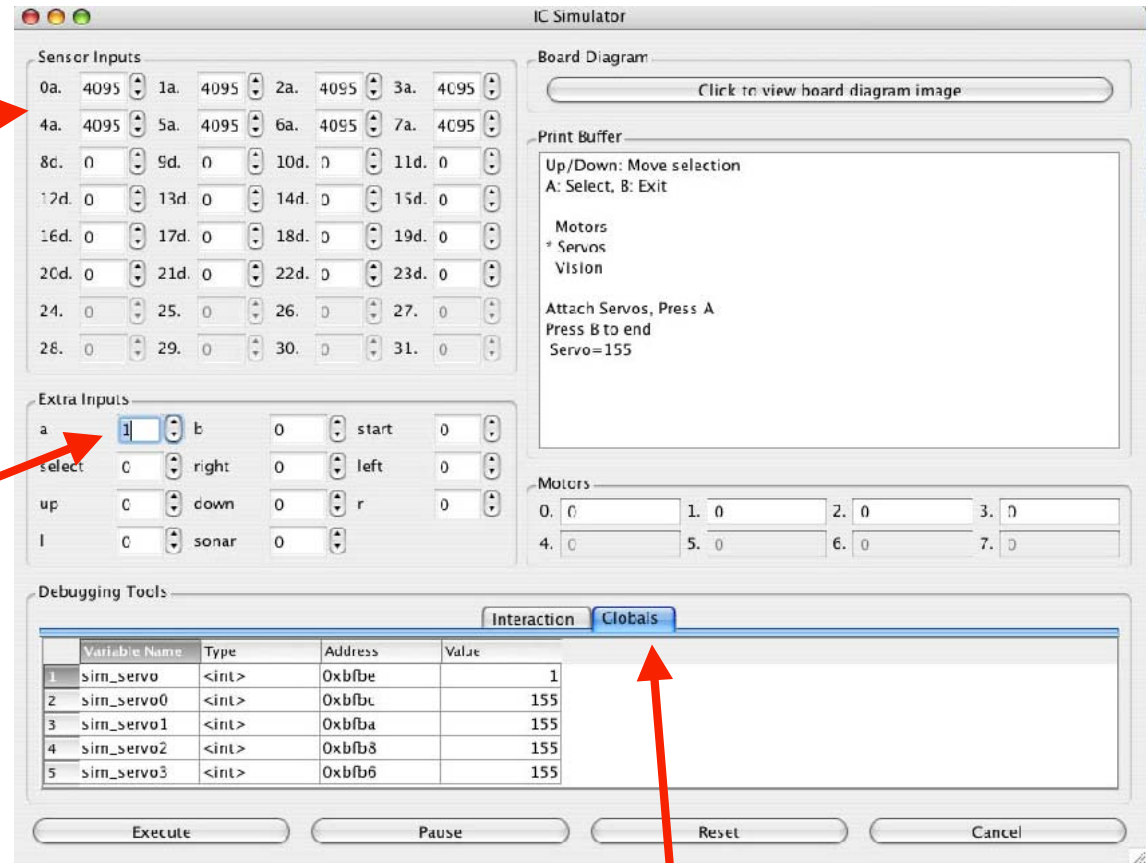
Reset stops the
program and
reloads the code



Simulator Example (3)

Analog and digital sensor values can be input by clicking or typing in the appropriate box

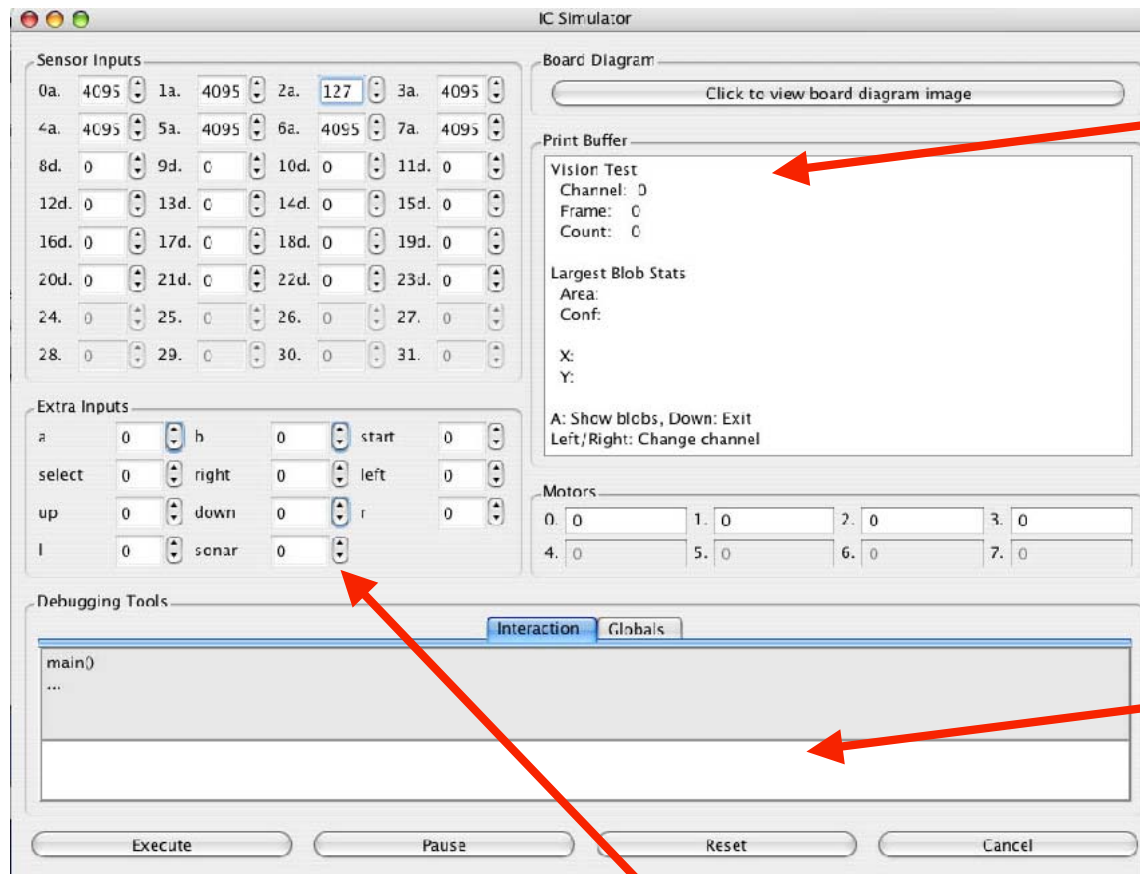
Button A is being pressed by hitting the up arrow by its value. (Don't forget to release the button by clicking on the down arrow)



The globals tab shows the values of globals and allows them to be changed



Simulator Example (4)



Vision functions can be called, but there is no way to update vision data (no camera simulator)

The Sim Interaction window operates similarly to the interaction window when a board is connected

Input for special sensors like knob or sonar can be made at the appropriate box



XBC Motors



XBC Has Two Kinds of Motor Commands

- PWM Commands
 - Similar to those on Handy Board
 - Changes the duty cycle of the PWM to change motor behavior
 - Commands include:
`fd`, `bk`, `motor`,
`off`, and `ao`
- BEMF PID Commands
 - These commands actively adjust to drive motor at desired speed or desired distance
 - PWM signal sent to the motor automatically varies as needed
 - Behavior of commands does not vary with battery level or most environmental factors
 - Commands are detailed on next page



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.
- **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.
 - This function uses power
 - **freeze** will continue to control the motor until it is terminated by another motor command.



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 distance
   */
void back_and_forth_the_same_dist()
{
    // clear motor counter (set to 0)
    clear_motor_position_counter(2);
    fd(2); // turn forward at full speed
    sleep(5.0); // keep turning for 5 secs
    off(2); // turn the motor off
    printf("dist=%d\n",
           (int)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
}

```

