**Robotics Club - Colony**

**Intro Lab #1 – Target Practice**

Release Date: 9/24/10

**Demo Date: 10/8/10**

 

1. **Colony: Introduction Story** – a short intro describing how this lab came to be

2. **Lab1 Objective and Documentation** – the goals of this lab and how to accomplish them

3. **Appendix A: Colony Library Documentation** – more specific information on different things you can program

4. **Appendix B: C Documentation** – comments, logic, and stuff that goes with programming in C

5. **Appendix C: Sample Code** – sample reading from sensors and printing to your computer

**Introduction Story**

*You can skip this section if you wish.*

Fact: robots like to make their creators happy. Last week it came to the attention of the robots that some of their creators liked to eat cupcakes. Using their superior coordinating skills and using none of their superior moral skills, the colony robots conspired to steal overpriced cupcakes from Entropy+. They successfully filed in single-file and were poised to steal cupcakes, when they realized they needed a lookout.

The robots needed to move one at a time through a critical point, and wanted to keep a lookout there at all times in case the police showed up. Unfortunately, they weren't able to execute this switch-off maneuver – where the next robot drove to where the previous one was standing – with enough efficiency to get out in time. As it happened, the manager walked in while a slow switch-off was happening, the robots were caught, and they all wiped their memory for the sake of the mission.

Luckily, colony is modular, so new robots easily took the place of their fallen companions and carried out the mission cleanly the next day, to the great enjoyment of the Colony Project members.

To this day the switch-off maneuver has been retained as a training exercise.

**Lab 1 Objective and Documentation**

Your robot will be placed approximately 3 feet away from the target robot, facing a random orientation. Your robot must rotate to face the target robot, and then drive towards it, stopping at a range of 1 foot. The target robot will be removed. When Button 2 is pressed, your robot will drive forward, and stop in the location where the target robot was sitting. For testing, code has been provided for the target robot. Go into the beacon folder, and compile and download it onto a second robot.

Demo date: 10/1/10

**Section 1: Setting up your computer to read data off the robots**

**Linux**

1. Install gtkterm using your favorite package manager.
2. When using gtkterm, set the port to /dev/ttyUSB0, and the speed to 115200. If you are calling it from a terminal, the call is: gtkterm –p /dev/ttyUSB0 –s 115200
3. Close the connection to the robot (by changing the port or closing gtkterm) when done or programming won’t work.

**Mac OSX**

1. Download and install Goserial.
2. When using goserial, click on the light switch and set the speed to 115200, and select the USB options under connections.

**Microsoft Windows**

1. Download and install PuTTY from : http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
2. Run PuTTY, and set the Connection Type to Serial.
3. Set Serial Line to whatever COM port your robot is on (probably COM4).
4. Set Speed to 115200.
5. Click Open to start the connection.
6. Close the PuTTY window when done, or programming the robot won’t work.

**Appendix A: Colony Library Documentation**

**USB**

These functions print to USB so you can read data from your serial terminal. Be aware that they are NOT like printf, and you must break up strings into pieces by type.

usb\_puti(INTEGER)

usb\_puts(STRING)

usb\_putc(CHARACTER)

INTEGER: An integer number.

STRING: A string. It does not accept formatting arguments like printf. To print a newline, use \r\n.

CHARACTER: A character.

**Encoders**

Encoders measure the distance turned and rate of turning of the wheels (the large colorful ones, not the potentiometer). Be aware that their accuracy is very questionable.

encoders\_init()

Initializes the encoders. This must be done before they can be used. It is already included in lab1.c.

return: nothing.

encoder\_read(ENCODER)

return: An integer from 0 to 1024 measuring the instantaneous value of the selected encoder. -1 indicates an error, usually low battery. 1025 or greater indicates other error.

encoder\_get\_dx(ENCODER)

return: An integer measuring the total distance traveled by the selected encoder. Count is in encoder clicks, which is about 1/1024 of the wheel circumference. But experiment to determine how far the encoder must travel to measure a distance, don't trust math.

encoder\_get\_v(ENCODER)

return: An integer from -1024 to 1024 measuring the instantaneous velocity of the selected encoder. Returns 2048 if error occurs. Results are inconsistent, so use at your own risk.

encoder\_rst\_dx(ENCODER)

Resets the total distance that the selected encoder traveled to 0.

return: nothing

ENCODER: LEFT or RIGHT, for the encoder on the left or right wheel respectively.

**Rangefinders**

Rangefinders emit IR light and measure the brightness of the reflected signal to measure the distance of the nearest object. There are 5 rangefinders in different positions around the robot. The positions are:

IR1:Front-Left, IR2:Front, IR3:Front-Right, IR4:Left, IR5:Right.

range\_read\_distance(RANGEFINDER)

RANGEFINDER: one of the 5 rangefinders (IR1, IR2, IR3, IR4, IR5)

return: An integer from 0-255, or -1 (bad reading) representing a distance to an object in front of the rangefinder in arbitrary units on an arbitrary scale.

**BOM (Bearing and Orientation Module)**

A Roboclub creation, the BOM enables Colony robots to find each other. A BOM can either be broadcasting or reading at any time. A BOM detects the IR brightness in 16 directions and can return either a brightness from an individual direction or the brightest direction. Theoretically, the brightest direction is the direction to a robot that has a BOM that is broadcasting. Readings are often jumpy, and the brightest direction can fluctuate. Additionally, only one robot can have its BOM broadcasting at a time in order for other robots to find a direction easily.

bom\_init(TYPE)

Initializes the BOM so it can be used. It is already in lab1.c, but you may have to change the argument.

TYPE: The type of BOM you have. If it is yellow, it is BOM10, if it is green, it is BOM15.

bom\_on()

Turns the BOM on. Only used to transmit your position to other robots. Do not try to read your BOM when it is on.

bom\_off()

Turns the BOM off. Other robots will not be able to find you. You must do this to read values from the BOM.

bom\_refresh(BOM\_ALL)

Refreshes the readings from the BOM. Do this each time before using bom\_get or bom\_get\_max to get updated values. BOM\_ALL is the argument you want to pass in to the function, not a placeholder as in the rest of the documentation.

bom\_get(WHICH)

return: An integer value from 1-1024 representing the brightness of the selected BOM sensor.

WHICH: Which sensor on the BOM you want the value of. Keep in mind that the labels on the BOM are from 1-16, but the labels in the chip are 0-15 (same starting sensor and direction).

bom\_get\_max()

return: An integer from 0-15 indicating which BOM sensor is reporting the highest reading. This should usually be in the direction of another robot that has its BOM on. Keep in mind that the labels on the BOM are from 1-16, but the labels in the chip are 0-15 (same starting sensor and direction). This is usually accurate, but subject to interference. It will return -1 if there is no significant max.

**Motors**

Sets the direction and speed of the motors. Motors keep their direction and speed until changed. Be

aware that setting both motors at the same speed is no guarantee that your robot will drive straight.

motor\_r\_set (DIRECTION, SPEED)

motor\_l\_set (DIRECTION, SPEED)

DIRECTION: Either FORWARD or BACKWARD

SPEED: An integer value from 0-255. Motors may not work below 170.

Or one of the following values: SLOW\_SPD, HALF\_SPD, NRML\_SPD, FAST\_SPD,

FULL\_SPD.

return: nothing

**Orbs**

Sets the color of the Orbs. Orbs keep their color until changed. Calling these functions repeatedly

without delay may cause Orbs to not light up.

orb1\_set\_color(COLOR)

orb2\_set\_color(COLOR)

orb1\_set( R, G, B )

orb2\_set( R, G, B )

COLOR: Any color from RED, ORANGE, YELLOW, LIME, GREEN, CYAN, BLUE, PINK,

PURPLE, MAGENTA, WHITE, ORB\_OFF

R,G,B: An integer value from 0-255. R = red value, G = green value, B = blue value

return: nothing

**Buttons**

Reads whether the buttons are being pressed or not.

button1\_click()

button2\_click()

return: 0 immediately if button not pressed. If button is pressed, returns 1 as soon as button is released.

**Potentiometer**

Reads the value of the potentiometer.

wheel()

return: the value of the potentiometer (wheel), as an integer from 0-255.

**Delay**

Delays code execution for the specified amount of time.

delay\_ms(TIME)

TIME: Time in milliseconds (1/1000 seconds) that code execution will be delayed.

return: will return when TIME has elapsed.

**Appendix B: C Documentation**

**Comments**

Comments are text that is not executed as commands. They’re just there to keep you organized.

// This is a legal single line comment. It is proceeded by //.

// Anything before // on this line is part of the code,

// anything after is part of the comment.

code // comment

/\*This is a legal multi-line comment. Anything between the /\*

and \* / (without a space) is part of the comment. Anything else is part of the code. Notice it works over multiple lines.\*/

code /\* comment

comment

comment \*/ code

code

**Control Structures**

Control structures make decisions and make your program not progress in a linear fashion. The will

cause blocks of code to be executed, skipped, or repeated based on tests. These tests are done with

logic statements, that result in a 1 (true) or 0 (false) when evaluated. More detail on logic statements is

below.

**if (if-else)**

Will run or skip a block of code depending on when test is true. Always use braces to delineate a block of

code for an if or if-else statement.

if( test )

{

/\*code here will run if test is true\*/

}

else

{

/\*code here will run if test is false\*/

}

/\*code here will always run\*/

**while**

Loops through the code as long as test is true. Always use braces to delineate a block of code for a

while loop.

while( test )

{

/\*code here will run if test is true\*/

}/\*when the execution reaches this line, it jumps back to the

while line \*/

/\*code here will always run\*/

**for**

A loop that iterates through a counter. Always use braces to delineate a block of code for a for loop.

int counter; /\*you must declare your variable before the loop\*/

for( counter = 0; counter <= 250; counter++)

{

/\*code here will run once for each iteration of the loop\*/

}/\*when the execution reaches this line, it jumps back to the for line\*/

/\*code here will always run\*/

**True/false expressions:**

Computer programs make decisions by testing whether a condition statement is true or false. In C,

expressions that evaluate explicitly to false or to the number 0 are false, and all other expressions are

true. As an example, 5 == 3 is an expression. The symbol == tests whether the number on the left equals

the number on the right. This expression is false, because 3 does not equal 5. Other operators that

may be used in the same way:

a == b true if a equals b

a != b true if a does not equal b

a < b true if a is less than b

a > b true if a is greater than b

a <= b true if a is less than or equal to b

a >= b true if a is greater than or equal to b

You may stick these expressions in the "test" area of the if, while, and for statements described below,

and the expression will only execute if the expression evaluates to true.

**Combining Logic Expressions**

Once you've gotten that down, you may try combining expressions using the AND, OR, and NOT

operators. They work like this:

a && b true if both expression a AND expression b are true

a || b true if either expression a OR expression b are true

!a true if expression a is false

An example: (5 > 3) && (10 == 10) is true, because 5 is greater than 3, and 10 is equal to 10.

A note should be made here on the appropriate use of parentheses. Operations inside parentheses will

happen before operations outside. A brief example of the use of parentheses:

(1 + 2) \* 10 evaluates to 30, because the 1 + 2 happens first, then that 3 gets multiplied by 10.

1 + 2 \* 10 evaluates to 21, because the 2 \* 10 happens first, then 1 is added to that 20.

If you followed all of that, it may interest you to know that large, complicated logical expressions may be

created using AND, OR, NOT, and multiple layers of parentheses.

**Appendix C: Sample Code**

**Encoders sample code:**  
int encoders(void)  
{  
 int encoder\_left,encoder\_right;  
 int x\_left, x\_right;  
 int v\_left, v\_right;

while(1) {

/\* button1 is pressed \*/  
 if (button1\_read()) {

/\* reset dx \*/  
 encoder\_rst\_dx(LEFT);  
 encoder\_rst\_dx(RIGHT);

}  
 encoder\_left = encoder\_read(LEFT);  
 encoder\_right = encoder\_read(RIGHT);  
 usb\_puts("Encoder values (left, right): ");  
 usb\_puti(encoder\_left);  
 usb\_puts(", ");  
 usb\_puti(encoder\_right);  
 usb\_puts("\r\n");  
  
 x\_left = encoder\_get\_x(LEFT);  
 x\_right = encoder\_get\_x(RIGHT);  
 usb\_puts("Total Distance (left, right): ");  
 usb\_puti(x\_left);  
 usb\_puts(", ");  
 usb\_puti(x\_right);  
 usb\_puts("\r\n");  
  
 v\_left = encoder\_get\_v(LEFT);  
 v\_right = encoder\_get\_v(RIGHT);  
 usb\_puts("Velocity (left, right): ");  
 usb\_puti(v\_left);  
 usb\_puts(", ");  
 usb\_puti(v\_right);  
 usb\_puts("\r\n");  
 delay\_ms(500);

}  
 return 0;  
}  
  
**Rangefinders:**  
int rangefinders(void)  
{  
 int rangefinders[5] = {IR1, IR2, IR3, IR4, IR5};  
 int range;  
 int index;  
  
 while(1){  
 for(index = 0; index < 5; index++){  
 range = range\_read\_distance(rangefinders[index]);  
 usb\_puts(" IR");  
 usb\_puti(index);  
 usb\_puts(": ");  
 usb\_puti(range);  
 usb\_putc("\t");  
 }  
 usb\_puts("\r\n");  
 delay\_ms(200);  
 }  
 return 0;  
}  
  
**BOM:**  
int bom(void)  
{  
 int index;  
 int bomVal, max;

bom\_init(BOM10);  
  
 while(1){  
 bom\_refresh(BOM\_ALL);  
 for (index = 0; index < 16; index++) {  
 bomVal = bom\_get(index);  
 usb\_puti(bomVal);  
 usb\_puts(" \t");  
 }  
 max = bom\_get\_max();  
 usb\_puts("max: ");  
 usb\_puti(max);  
 usb\_puts("\r\n");  
 delay\_ms(200);  
 }  
 return 0;  
}