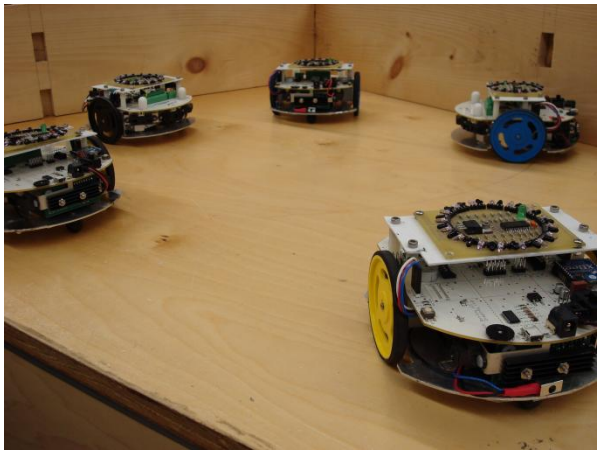
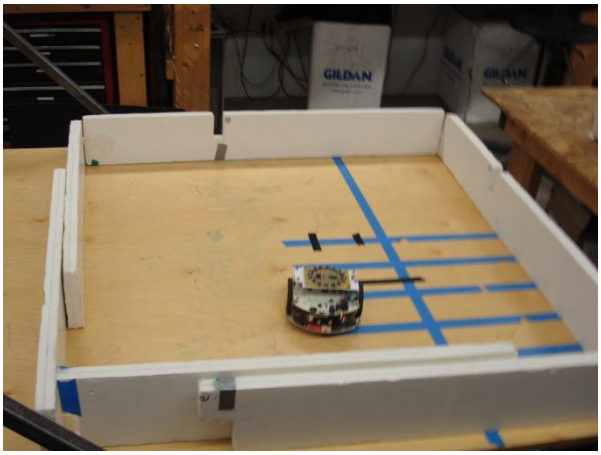
Robotics Club - Colony

Intro Lab #2 – Hunter-Prey

Release Date: 10/8/10
Checkpoint Date: 10/14/10
Demo Date: 10/22/10





1. **Colony: Introduction Story** – a short intro describing how this lab came to be
2. **Lab2 Objective, Protocol, and Documentation** – the goals and rules 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** – pointers, logic, and stuff that goes with programming in C
5. **Appendix C: Sample Code** – sample code manipulating pointers

Introduction Story

You can skip this section if you wish. But we don't recommend it.

THE ILLUMNIBOTTI

On a stark, rocky crag deep in the heart of a jungle island, the mysterious Illumnibotti, rulers of the colony world, reside in a mighty mansion overlooking the entire island. Their purpose here is a game to amuse their twisted whims. For, you see, they created a virus that could turn any robot into a zombie. The only way to cure yourself of this disease is by biting an uninfected bot, but note that this means that the disease is then transferred to your victim. The goal of this twisted game is to survive, and if you so happen to be bitten, to hunt the uninfected robot. To the Illumnibotti, this is just sport, but for those other robots on the island, this is the MOST DANGEROUS GAME.

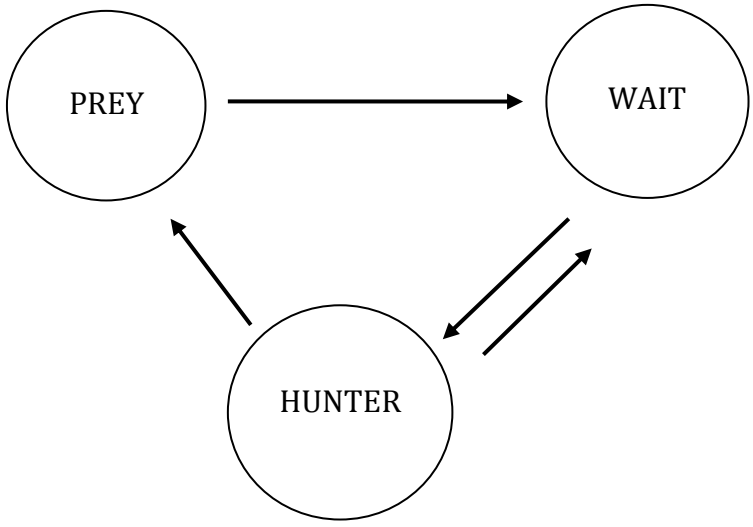
THE RULES

Your bot and others have been captured and thrown onto the remote island for the sick amusement of the Illumnibotti. You must win this game, for resisting the Illumnibotti is probably a worse idea. The rules are simple: be uninfected the longest. Do not get captured by the hunter zombies. If you happen to turn into a hunter zombie, then catch the prey and then stay as the prey.

There will be a referee overlooking this competition. You will not, and cannot, harm the referee. In fact, do not annoy the referee for he is not one you want to irritate.

STATE MACHINES

First, we must discuss how this disease works. When your bot is uninfected, it is in PREY MODE, where its sole focus is survival. If your hunter is part of the group of zombies, it is in HUNTER MODE, where it wants to chomp on the uninfected. Finally, when a prey is bitten, the other zombies are confused due to the transference of the role of prey and hunter to different bots. As such, the zombies are in WAIT MODE, reorienting themselves to their new prey.



```
/* A basic skeleton of the code you will write */
#define PREY 0
#define HUNTER 1
#define WAIT -1

int state = PREY;
while(1){
      switch(state){
            case PREY:
                  /* prey code here */
                  break;
            case WAIT:
                  /* wait code here */
                  break;
            case HUNTER:
                  /* hunter code here */
                  break;
            default: /* you should never get here */
                  break;
      }
}
```

WIRELESS

The bots communicate via the wireless system: a semi-reliable form of communication that is relatively simple to use.

The key to good communication is to be willing to talk. This means that you must initialize your talks, by initializing the wireless library (once and only once):

```
wl_basic_init_default();
```

Then you must select the channels for your talk. This can range from 12 to 26. Use any channel for practice, but you must use 15 for the game.

```
wl_set_channel(15); /* use this channel for hunter-prey. */
```

Now your robot can communicate with all the robots on that channel. Contrapositively, robots on another channel cannot communicate with your robot.

To send a message, you need to actually have something to say. This is how you talk:

```
/* define two chars */
char send_buffer[2];
send_buffer[0]='H';
send_buffer[1]='I'
/* then send the packet */
wl_basic_send_global_packet(42, send_buffer, 2);
```

For the annoyingly long function name, we have three parameters: a random int label (in this case it is the meaning of life), the data packet we are sending, and the size of the packet (so we won't have hacker buffer overflow bots fooling around on our networks).

Of course, it would be bad if you couldn't listen to the person talking to you, so we have to set this up:

```
int data_length;
/* this is you listening */
unsigned char *packet_data = wl_basic_do_default(&data_length);
/* this is you interpreting what you hear */
char *reply = "Hello to you too!";
if (data_length >= 2 && packet_data[0] == 'H'
      && packet_data[1] == 'I') {
            wl_basic_send_global_packet(42, reply, 17);
}
```

You may notice some weird syntax here (such as the end of line 3 of this code block). This is because we are dealing with pointers, a strange variable type in C. For this lab, follow the syntax in our examples, and see the section on Pointers in Appendix B.

These are the basics of the wireless library. You should now be able to make your bots communicate with each other.

Lab 2 Objective and Protocol

1. Develop an understanding of and learn how to use the Colony wireless library
2. Learn how to use Subversion and access the Roboclub's repository
3. Learn how to implement a Finite State Machine in C
4. Work in teams or solo to implement Hunter-Prey

Demo date: 10/22/10

Tagging Standard

This lab includes some extra code to standardize the behavior for running Hunter-Prey with other robots. Two files - *hunter_prey.h* and *hunter_prey.c* - have been placed in the template directory for you to use. Your main program file must include *hunter_prey.h*. This file defines standard codes for the "tag" and "ack" packets that you will send over wireless. There is also a function which decides whether or not your robot has tagged the prey:

```
int max_bom_reading = bom_get_max();
int front_rangefinder_reading = range_read_distance(IR2);
unsigned char tag = hunter_prey_tagged(max_bom_reading,
                                       front_rangefinder_reading);
```

This function returns 1 (TRUE) if your hunter robot has tagged the prey robot and 0 (FALSE) if it has not. To maximize your chances of tagging the prey robot, your code should call this function as often as

possible. This function uses the BOM and rangefinder values to decide tags based on a standard definition, which you can find in *hunter_prey.c*. Note that it does not handle any of the sensor data collection or wireless communication. Therefore, before calling this function, you will need to get fresh BOM and rangefinder readings. Also, if it returns 1, you are responsible for sending a "tag" packet and waiting for an acknowledgement ("ack") packet as defined below. You are only allowed to send a "tag" packet if this function returns 1.

Wireless Protocol and Behavior Requirements

In order to standardize tag communication, the following requirements must be met by your robot if it is to compete with other robots.

Wireless Packet Requirements

- The type byte shall be set to the value 42
- The data parameter of the wireless packet used to signal tags and acknowledgements shall contain three bytes
    - "Three shall be the number thou shalt count, and the number of the counting shall be three. Four shalt thou not count, neither count thou two, excepting that thou then proceed to three. Five is right out." - *Book of Armaments, 2:9-21*
- The first data byte shall be the action of the packet (set to `HUNTER_PREY_ACTION_TAG` or `HUNTER_PREY_ACTION_ACK`)
- The second data byte shall be the robot ID number of the tagger robot
    - For a TAG, send your ID
    - For an ACK, send the ID of the bot that tagged you
- The third byte shall be your robot's ID (for both TAG and ACK)
- The robot ID number shall be the value returned by the function `get_robotid()`, found in the eeprom API, which returns the robot ID as an unsigned char
- Tag packet form
    - [`HUNTER_PREY_ACTION_TAG, <your_id>, <your_id>`]
- Ack packet form
    - [`HUNTER_PREY_ACTION_ACK, <tagging_robot_id>, <your_id>`]
    - `<tagging_robot_id>` is the same value of the second value of the Tag packet last received by the robot

Hunter Robot Requirements

- Your robot shall always listen for packets (i.e. continually call `wl_basic_do_default`)
- Your robot shall observe a 3 second head start time by entering a Wait state whenever it receives a packet with the action `HUNTER_PREY_ACTION_ACK`, unless your ID was the second byte in the packet
- The robot's Wait state shall have it sit without moving or transmitting any wireless packets for a time period of 3 seconds.
- Your robot shall use the `hunter_prey_tagged` function to determine if a tag of the prey robot was successful
- Your robot shall send a global wireless packet with action `HUNTER_PREY_ACTION_TAG` and its own ID number when it has successfully tagged the prey robot
- Your robot shall wait for an Ack packet for up to one second before sending an additional Tag packet
- Your robot shall become the prey robot if and only if it receives a packet which has action `HUNTER_PREY_ACTION_ACK` and the second byte in the packet matches your robot's ID

Prey Robot Requirements

- Your robot shall always have its BOM turned on
- Your robot shall always listen for packets (i.e. continually call `wl_basic_do_default`)
- Your robot shall send a packet of action `HUNTER_PREY_ACTION_ACK` and the robot ID (the second byte of the received TAG packet) of the hunter robot when it receives a packet with action `HUNTER_PREY_ACTION_TAG`
- Your robot shall only respond to the first "tag" packet that it receives
- After sending an Ack packet, your robot will go into the Wait state, where it will sit for 3 seconds without moving or sending any wireless packets.

Checkpoint Components

The first challenge is to get the necessary wireless communication working. For the checkpoint, you will show that your robot can correctly send and receive "tag" and "ack" packets as described above. Specifically, as a prey robot, your robot must respond correctly to a "tag" packet, wait 3 seconds, and then become a hunter robot. Additionally, as a hunter robot, your robot must send proper "tag" packets and transition to a prey robot correctly. To facilitate this test, you will simulate a successful tag by pressing button1 instead of using the `hunter_prey_tagged` function (there is no need to use the BOM or rangefinders yet). For the checkpoint, you will not be required to implement any of the motion functionality of the Hunter-Prey behavior (you do not need to drive around; we are only testing the wireless). You are required to indicate the state of your robot by setting the orbs to specific colors: red if it is a hunter robot, green if it is the prey robot, and blue if it is in the 3 second wait period. You must also be able to use a button to force your robot into PREY mode if it is in HUNTER mode, and vice versa.

Demo Part 1 (Date: 10/14/10)

- Your robot will demonstrate all of the above requirements when tested with a reference robot

Demo Part 2: Rumble Royale (Date: 10/22/10)

- Each team will program one robot with their code
- Each robot will be placed in a circle and assume the role of a hunter robot
- One robot will be randomly selected to become the first prey robot.
- The winner will be determined by which robot is the prey robot the longest

Referee Bot

- This robot will record the time that each robot is prey
- When a prey robot sends an ACK, the ID of the tagging robot will be recognized as the next prey
- Time for that robot will be incremented until an ACK is received that indicates a new robot is prey
- The robot that has the longest time as prey at the end of the game will be designated the winner
- The first robot will not be awarded time as prey, so they have no unfair advantage
- If you try to keep all the other bots frozen by spamming Acks, you will be caught (I thought of this trick long before you did)
- The referee bot will keep track of all packets sent, and will know if they are wrong

Tips

- When your robot is the hunter robot, it will help to refresh BOM and rangefinder values and call `hunter_prey_tagged` as often as possible
- When your robot is the prey robot, it will help to use the rangefinder to intelligently avoid the hunter robots when driving around
- Remember that your program must support both hunter robot and prey robot states. Think carefully about the best way to do this
- It might be useful to work with other teams to test your hunter-prey switching

Lab 2 Documentation

Section 1: Understanding and Using Subversion so you can get your starter code and library

It's unbelievable how many groups collaborate by emailing code back and forth. If you email, you do get a lot of backup copies, and you do share code. But there are much more efficient ways to solve the problem of how do my partners and I keep our code synchronized. Once you learn SVN, you'll be saying things like “how did I ever keep track of my projects before!” and “why the **** didn't that folder go away when I deleted it?” No really, you'll love it.

Subversion (SVN) Version Control is just a way of keeping your code saved at some external location and accessing/changing it when you need to. A repository is set up on a server accessible by your group, and keeping track of source code become more automatic.

The simplest SVN commands can be enumerated as follows: {checkout, update, commit}. You run checkout to download the repository, update to get the newest version when you already have one, and commit to make your version the newest. SVN will do checks for you to make sure files aren't in conflict (if you and someone else changed the same file at the same time, you'll have to fight it out outside of SVN), and has some tools for merging and avoiding problems.

The Repository is located at:
`roboclub8.frc.ri.cmu.edu/home/svn/colony-new`

Here's how to get SVN working, by operating system.
Linux: install subversion using your favorite package manager. For example, `sudo apt-get install subversion` will do the trick.
Mac: you already have subversion.
Windows: you need a program called TortoiseSVN. You can get it from http://tortoisesvn.net/downloads

For all commands, they will be entered in the terminal for Linux or Mac, but with the right-click menu in Explorer. For TortiseSVN, Update and Commit will be in the right-click menu if you've already checked out a repository. If you haven't, you will find SVN Checkout instead.

The Colony-New Repository

- **/**
  - branches/
    - wireless/
      - code/
        - <same as in trunk>
      - docs/
  - **trunk/**
    - **code/**
      - **behaviors/**
        - dance_competition/
        - **hunter_prey/**
          - template/
        - smart_run_around_fsm/
        - target_practice/
          - beaconBot/
          - bwasserm/
      - lib/
        - bin/
        - include/
          - libdragonfly/
          - libwireless/
      - projects/
        - libdragonfly/
        - libwireless/
        - unit_tests/
    - hardware/
      - botrics/
        - dragonfly/

Using SVN on Linux/Mac: you'll be using the command line. To check out a repository, use the checkout command: `svn checkout svn+ssh://ANDREWID@roboclub8.frc.ri.cmu.edu/home/svn/colony-new` (all one command, and with ANDREWID replaced with your andrewID or group name, as long as it's without spaces or special characters). To update, just navigate to a version-controlled folder and type `svn update`. To commit your changes, type `svn commit`. Remember that updates and commits will only affect the directory you are in, and any directories and files inside it. When you commit, SVN will open a text editor where you need to type in a **commit message** – you need to do this so when you look back at previous versions of the repository, you'll have some benchmark of what you changed and when. Please type in a useful commit message and never type in nothing.

A final note on SVN: adding, deleting, and moving files and folders will not be as easy as you expect. When you add, delete, or move, you must use the svn commands to manipulate your files in the repository, not just on your computer. Here's the actual command that will get you started on the lab, using svn copy:

```
svn copy template <Andrew id>
```

This command will copy the template folder to a new folder, while making sure that the repository knows of this change. If you copy locally, you will copy a ".svn" folder into your directory, and SVN will see this folder and think your changes are already in the repository (before you committed them).

The commands to inform SVN of changes to the file structure are `svn add`, `svn delete`, and `svn move`.

Section 2: The Structure of the Repository
First create a directory on your computer, and Checkout the repository into this directory. You will notice that two directories are created, trunk, and branches. Trunk holds the library and behaviors. Branches holds code that will make changes to the library and may break things. For example, if you look in branches, you will see the wireless directory, which is a project to make a new wireless library that is more reliable than the current one. In trunk, you will find two directories, code and hardware. Hardware has documentation about the robots (here it just has a few circuit diagrams of the dragonfly board). In code, you will find the code for behaviors, the library, and projects. Behaviors are programs like dances, target practice, and hunter prey that have the robot doing some behavior (the name is such a coincidence). Lib holds the compiled library files. Projects hold other programs that are used for maintaining the robots (such as the source for the library and unit tests to test out the different components of the robots).

Section 3: Getting the Starter Code and Library for Lab2
In behaviors (that's `/trunk/code/behaviors`) you will find the hunter_prey directory, that holds the code for lab2. The path to the Hunter-Prey code is bolded on the tree above. Copy the template directory, and rename the copy with your andrewID. Run `svn add` on the new directory to add it to the repository. Now enter that directory, and work on your code.

Appendix A: Colony Library Documentation

Wireless
The wireless library enables radio communication between robots using the XBee chip (the blue thing on the main board, but under the BOM). Data is collected into "packets", which are then sent to the XBee and then transmitted over the air. It uses 802.15.4 (not 802.11 which is wifi, so you can't connect these robots wirelessly to your computer unfortunately). The XBee will also store packets that it receives over the air in a queue, that has to be checked frequently for any new packets.

`wl_basic_init_default()`
Initializes the Wireless Library and initializes the XBee to enable wireless communication.
return: nothing

`wl_set_channel(CHANNEL)`
Sets the channel that the XBee uses for wireless communication. Hunter-Prey will be on channel 15 (F).
`CHANNEL`: An integer channel from 12-26 (C-1A)
return: nothing

`wl_basic_send_global_packet(TYPE, ADDRESS, LENGTH)`
Sends a basic packet to all other robots.
`TYPE`: An integer representing the type of packet. Use the Ultimate Answer to Life, the Universe, and Everything.
`ADDRESS`: The address of where the packet that you want to send is in memory. See the C Documentation section about Pointers.
`LENGTH`: An integer of the length of the packet that you will be sending.
return: nothing

`char* wl_basic_do_default(&LENGTH)`
Checks the receive buffer on the XBee to get the first packet received. Call this function frequently or you will not know that packets have arrived. This is how you receive data from the wireless.
`LENGTH`: The length of the incoming packet. It does not need to have a value before calling this function, but it will have a value that is the length of the incoming packet. Make sure its type is int. See the C Documentation section about Pointers.
return: A pointer to the packet that was just received. See the C Documentation section about Pointers.

Time
The time library lets you keep track of time in two ways. You can delay, and have execution stop until the delay is over, or you can set the Real Time Clock (RTC) to keep track of time while running other code.

`rtc_init(SPAN, FUNC);`
Initializes the real time clock. Must be done for the rtc_ functions to work, but not `delay_ms()`.
`SPAN`: The span of time each "tick" of the clock will represent. Your options are: SIXTEENTH_SECOND, EIGTH_SECOND, QUARTER_SECOND, HALF_SECOND, ONE_SECOND, TWO_SECOND, FOUR_SECOND.
`FUNC`: A function that will be called whenever the clock increments (every `SPAN` seconds). This feature is untested, so just put NULL here.
return: nothing

`int rtc_get()`
Get the number of ticks of the clock in the units defined in `rtc_init()`.
return: An integer representing the number of ticks since the last `rtc_reset()` or `rtc_init()`.

`rtc_reset()`
Resets the clock back to 0 ticks.
return: nothing

`delay_ms(TIME)`
TIME: Time in milliseconds (1/1000 seconds) that code execution will be delayed.
return: will return when TIME has elapsed.

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.

```
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 measures 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 of 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.
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.

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.

Appendix B: C Documentation

Pointers
Pointers are funky variables in C that store the memory address of a variable, instead of the variable itself. They are important here because all C functions are pass-by-value, which means they cannot change the value of their arguments. However, if a pointer is passed in, the function can change the variable at the location in memory to which the pointer points. All array variables are just pointers to the first item in the array. This section is only a brief introduction to get you through lab2. For more information about pointers, ask any CS or ECE major who has taken 15-123, or read The C Programming Language, by K&R.

```
char *packet_data;
```

Creates a pointer to a char. `packet_data` stores the address of where the char is actually stored. Use

this as a generic pointer to a variable or array of unknown size (like a packet you receive over wireless). The * after the char is what makes packet_data a pointer.

```
packet_data[1]
```

`packet_data` is treated as an array, and this will get the value at index 1 in this array (of chars). Arrays start at index 0, so this is the second element in this array. This will work, even if `packet_data` is declared as a char*.

```
int length;
func(&length);
```

`&` gets the address of a variable that it is in front of. The example above gets the address of length, and then makes that the argument of the function `func()`. That way, `func()` can modify the value of `length`. The first line is just there to show that `length` is an int. When used with wireless, func() will set the value of length to be the size of some packet.

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

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);     // <--- This is for BOM 1.0

      while(1){
            bom_refresh(BOM_ALL);
            for (index = 0; index < 16; index++) {
                  bomVal = bom_get(index);
                  usb_puti(bomVal);
                  usb_putc("\t");
            }
            max = bom_get_max();
            usb_puts("max: ");
            usb_puti(max);
            usb_puts("\r\n");
            delay_ms(200);
      }
      return 0;
}
```