Robotics Club – Colony
Introductory Lab #2 – Hunter Prey

Release Date: 10/3/09

Checkpoint Date: 10/14/09

Final Demo Date: 10/23/09

Prerequisite

This lab assumes that you have already learned how to set up the programming environment. It also assumes that you have learned the basics of programming the robot to move around (see Introductory Lab #0), as well as how to use the BOM and rangefinders (see Introductory Lab #1).

Objectives

1. Develop an understanding of and learn how to use the Colony wireless library
2. Work in teams to implement Hunter-Prey

The Demo: Hunter-Prey

First, your hunter-prey robot will have to demonstrate that it can play tag with other hunter-prey robots by using the Colony wireless library to communicate who is tagging and who is being tagged. The functionality of your hunter-prey robot will be tested against a reference robot (we will provide the reference robot) to ensure that your robot behaves as specified in the Requirements section. Once your robot passes our initial tests, it will be entered into a competition with all other hunter-prey robots. The robots will play one large game of Hunter-Prey, and the robot which remains the prey robot for the longest period of time will be declared the winner.

To ensure that you are making progress, we will have a checkpoint a week before the final demonstration. For the checkpoint, you need to show us that you have the entire wireless infrastructure working for your robot. This means that your robot should be able to communicate that it has been tagged by another robot, and your robot should be able to communicate a "tag" to another robot. A more detailed description of the requirements is included below.

Objective 1: Communication Between Robots

The Colony wireless library allows you to develop more complicated behaviors that involve multiple robots. In this lab, you will be using wireless communication to send and receive a standard "tag" packet. In order to communicate, our robots are equipped with wireless modules, which broadcast data over radio frequencies. Data sent over the network is referred to as a "packet". The Colony robots use the popular ZigBee 802.15.4 wireless protocol, which is implemented by the blue XBee wireless chip mounted on each robot. This standardized protocol connects each individual robot to all the others, allowing a robot to broadcast a global message to all robots or selectively talk to a single robot.

To avoid interference and confusion, each XBee can be set to a specific channel (much like a standard 802.11 wifi module). These channels allow multiple groups of robots to communicate with each other without interfering with other groups. Robots can only receive packets from other devices

transmitting on the same channel. During testing, each group will have to use a unique channel so that one group's robots do not interfere with those of other groups.

A Necessary Aside: Data Representation and Pointers

Use of the Colony wireless library requires knowledge of basic memory management techniques in C. The library uses the notion of pointers to generalize the type of data a wireless packet can contain. This added flexibility can be very useful, but with the added freedom comes a greater risk of incorrectly using the program memory and causing your program to malfunction.

A program uses storage containers called variables to store information. This information is saved in a large data storage array which is informally called memory. Every location in this array has an address (or index) and a corresponding piece of data which is stored at that address. For example, when you create an integer and assign it the value 42, a chunk of the memory array is selected (with its own address), and the number 42 is stored at this location.

```
int theAnswer = 42;   // Store 42 at a location in memory which we can
                      // access later by using the variable "theAnswer"
```

If you want to know where in the memory data array this variable is being stored, you can access the address of the variable by using the '&' character.

```
int *address = &theAnswer;  // Obtain the address of "theAnswer" and
                            // store it in a variable named "address"
```

This will allow you to determine the address where `theAnswer` is stored. Notice that to store off this address in a new variable, you have to declare the variable as a special type: **int ***. This will be explained shortly. First, suppose you now have a memory address, and you would like to extract out the data stored at this memory address. To do this, use the '*' character to take a variable, treat it as a memory address, and look up the value stored at this memory address.

```
// Treat the variable "address" as a memory address (which points to
// the location of our original variable, "theAnswer"), extract out
// the data at this memory location, and store it in a new variable
// called "hoursSinceSleep"
int hoursSinceSleep = *address;
```

Now, what about the data type **int ***? This data type simply labels the variable `address` as a pointer (which is just a memory address) that happens to point to a piece of memory that should be treated as an integer. If you wanted to treat a variable as an address for a character, use a **char ***; for a long, use **long ***.

The use of the '&' and the '*' gives a programmer considerable freedom in how they control a program. You will see more on these simple memory management tools in later programming courses. Be wary any time you start playing around with memory, though, as it can get very confusing very quickly. If you have further questions about pointers, please be sure to ask someone about them or look

online for help. A good tutorial for pointers (that assumes a basic knowledge of Java) can be found here: http://www.comp.lancs.ac.uk/~ss/java2c/diffs.html#pointers.

Basic Colony Wireless Library

The Colony wireless library is fairly complex and feature-rich; however, it can be difficult to use. For this lab, we have developed a basic wireless library that simplifies the process for packet transmission. To use the basic wireless library you will need to include *wl_basic.h* in your main program file. There are also a series of functions that you will need to learn how to use. The API for the wireless library is not currently posted online, so we have included HTML files which contain documentation for the wireless API with the lab.

The XBee wireless modules must be configured during initialization for them to work properly. Make sure to call the wireless initialization function after you call the `dragonfly_init` function.

```
wl_basic_init_default();
```

Groups of wireless modules must also be assigned to separate channels so that they can communicate with each other without interfering with other groups. Use the `wl_set_channel` function to set the channel of your XBee module. The valid channels range from channel 12 to channel 26.

```
wl_set_channel(12);
```

The meat of wireless networking comes from the ability to send and receive packets. Earlier we described how the XBee modules allow for robot-to-robot communication as well as global communcation to all robots. For this lab, the robots will only be communicating globally to all other robots (if you would like to learn more about how to send packets to individual robots, see the wireless API). The following function will send a packet containing data which, if treated as an array of characters, spells out "HI" to all robots operating on the same channel as the broadcasting robot.

```
char send_buffer[2];  // Create a data array for the function to
                      // wrap into the packet it sends

send_buffer[0] = 'H';
send_buffer[1] = 'I';

// Wrap the packet together and send it
wl_basic_send_global_packet(42, send_buffer, 2);
```

The first parameter is an arbitrary label used for distinguishing what type of packet is being sent (in this case we chose 42). The second parameter is a pointer to the address where the packet data is currently stored. The final parameter specifies the length of the data pointed to by the `send_buffer` variable (in this case, the array is of length 2).

To receive wireless packets, the robot must call the `wl_basic_do_default` function:

```
int data_length;  // Declare a container which will be filled
                  // with the length of the data packet

// Extract information from a recently received packet
unsigned char *packet_data = wl_basic_do_default(&data_length);

// Parse packet_data and execute code if we receive an 'H'
// character and an 'I' character in the first and second byte
if (data_length >= 2 && packet_data[0] == 'H'
      && packet_data[1] == 'I') {

      /* CODE HERE */

}
```

This function will check if a packet has recently been received and, if a packet has been received, extract the information contained in the packet. The `if` statement following the function call will execute the code inside the statement if the first two characters of `packet_data` spell out "HI". Be sure to check the return value of the `packet_data` variable to ensure that a valid packet was received (a null return value indicates that no new packets were received).

These functions will provide you with the tools necessary to communicate using the Colony wireless library. You will use these functions in addition to what you already know about the BOM and the rangefinders to implement the Hunter-Prey behavior. To find an example program that uses these functions, look at the Colony Robot Introduction page on Wireless, found here: http://www.roboticsclub.org/redmine/wiki/colony/Wireless

Objective 2: Hunter-Prey

The end result of this lab will be a behavior called Hunter-Prey. In this behavior, one robot is the prey while other robots (at least two) are the hunters. The hunters will chase the prey, and the prey will run away from the hunters while broadcasting its BOM. When a hunter comes within range (inside a 10 cm radius) of the prey, it "tags" the prey by sending a wireless packet. At this point, the hunter becomes the prey and gets a head start until the other robots all begin to chase it. On the demo day, each team will run their robot against the rest of the teams. The robot who is the prey the longest wins - survival of the fittest bot!

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 Standard and 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 two bytes
- 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
- 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

Hunter Robot Requirements

- Your robot shall always listen for packets (i.e. continually call `wl_basic_do_default`)
- Your robot shall observe a 5 second "head start" time whenever it receives a packet with the action `HUNTER_PREY_ACTION_ACK`, regardless of the ID in the packet
- 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 an ID that matches your hunter 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 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
- Your robot shall, without moving, wait for the requisite 5 second "head start" period before becoming a hunter robot after it sends the "ack" packet

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 5 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 5 second wait period.

Demo Part 1

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

Demo Part 2: Rumble Royale

- 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
- A reference robot will be placed at the center as the initial prey robot and will be removed after it has been tagged
- The winner will be determined by which robot is the prey robot the longest

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

Checkpoint Date: 10/14/09
Final Demo Date: 10/23/09