**University of California, Santa Cruz**

**COSMOS Cluster 6:**

**Networking and Robotics**

**Lab 2: Dead Reckoning: Variables, Loops, and Timing**

## Introduction

Now that you’re confident in the RoachBot toolchain, it’s time to start building some familiarity with the C language.

In this lab, we’ll use loops to implement a very simple navigation algorithm called “Dead Reckoning.” Then, we’ll explore ways to introduce more complicated behavior into the algorithm.

## Loop-based timing:

Open, compile, and load the code in DeadReckoning.X. What does it do?

Clearly, this code has some sort of mechanism for doing timing. It’s using the fact that instructions take a non-zero amount of time to execute, and a whole lot of them add up to a significant amount of time.

### Do the following steps:

1. Do some experimentation to find a value of ONE\_SECOND that accurately times out one second.
   1. This is hard to do if your timing experiment only lasts a second! You’ll probably need to time out about 10 seconds for decent results.

QUESTION: What happens if you add more instructions to the loop? Try, for example, a printf() call, or an LEDSSet() call.

1. Now, write loops that perform simple motions:
   1. Drive forward for one foot
   2. Turn 90 degrees right
   3. Turn 90 degrees left

QUESTION: If your roach attempts to move in a square (IE, one foot forward and 90 degrees right, for four iterations), how close does it get to its original location? How repeatable is this?

## Dead Reckoning:

“Dead reckoning” is attempting to navigate without any information from sensors. A dead reckoning system just executes a set of steps in a fixed order. This would be something like “Move 4 feet forward, then turn 90 degrees right, then move 1 foot forward, then turn left”

### Do the following:

1. There are two mazes on the floor. Write a dead-reckoning program that navigates at least one of them.

When you’re done, be sure to put your code on your COSMOS webpage, along with answers to the above questions.

Extra Challenges:

As usual, you can attempt these in any order you like, or come up with your own challenges.

Code that meets these challenges isn’t hard to *type,* but it is hard to *design.* Therefore, you should spend more time thinking than typing! This often involves drawing on the whiteboard or a piece of paper, talking to your teammates, colleagues, or staff, or taking a walk.

Also, note that these are very hard. Be brave!

## Non-blocking timing:

The loop-based timing structure you used above is very simple, but it comes with some serious problems. The biggest problem is that your roach can’t do anything else while it’s in the timing loop! For this reason, timing loops like these are called “blocking code,” and they prevent your robot from having fast reflexes, or doing two things at once.

Fortunately, the microcontroller has some special circuitry to allow it to do timing (we’ll talk more in class about this later). We provide the timers.c/.h module, which uses these peripheral circuits to implement timers.

For this challenge:

* Look at timers.h to see the functions available in the timers library. In particular, you’ll probably be using TIMERS\_Init(), TIMERS\_InitTimer(), and TIMERS\_IsTimersActive. Here’s a quick example of those functions in action:

|  |
| --- |
| #define TIMER\_A 0  #define TIMER\_B 1  while(1){  TIMERS\_InitTimer(TIMER\_A,1000);  while(TIMERS\_IsTimerActive(TIMER\_A)){};  TIMERS\_printf("A\n");    TIMERS\_InitTimer(TIMER\_B,500);  while(TIMERS\_IsTimerActive(TIMER\_B)){};  printf("B\n");  }; |

* Refactor your code so that it uses the Timers library instead of variable-based timing loops.
* Now, see if you can write your code to be non-blocking. Can you set up your navigation code so that the roach does these two things simultaneously?
  + Navigates the maze
  + Shows the LED sequence from the beginning of the lab

If so, you probably implemented non-blocking code! Check with staff though.

## Bump recovery:

If somebody’s foot is in the way, they can completely derail the roach’s navigation! For this challenge:

* Write a program that will navigate the maze, until it bumps into an object.
  + When a bumper is pressed, it stops moving, and when the bumper is released, it continues navigating the maze.
  + It should successfully complete the maze no matter where, how long, or how often it is stopped.

## Light sensitivity:

Roaches don’t like light, and they try to run away from it. Write a roach that navigates the maze, but whose speed is dependent on the amount of light on its light sensor. If it’s in full light, it goes fast, but if it’s in shadow, it goes slow. Fast or slow, it should navigate the maze.