Car Grid Exploration

Snow Li
Sriram Jayakumar
Brown University
12/12/2012
CSCI160: Embedded Systems (Professor Reiss)

Introduction

Common robotic tasks involve exploring an unknown space. For example, the Roomba automatic vacuum cleaner wanders a room cleaning the floor. Locations might be visited multiple times, but all spots are guaranteed to be cleaned. Data collection applications also fall into this category.

The purpose of this project is to build a toy car that will explore a user-specified grid of space autonomously, and collect the location of any obstacles in the grid. More specifically,

Objective: With the car, map a square area of side L, where L is a given length. Explore the area in fixed size steps B. For the case of a 5 unit square grid, produce a map like the following:

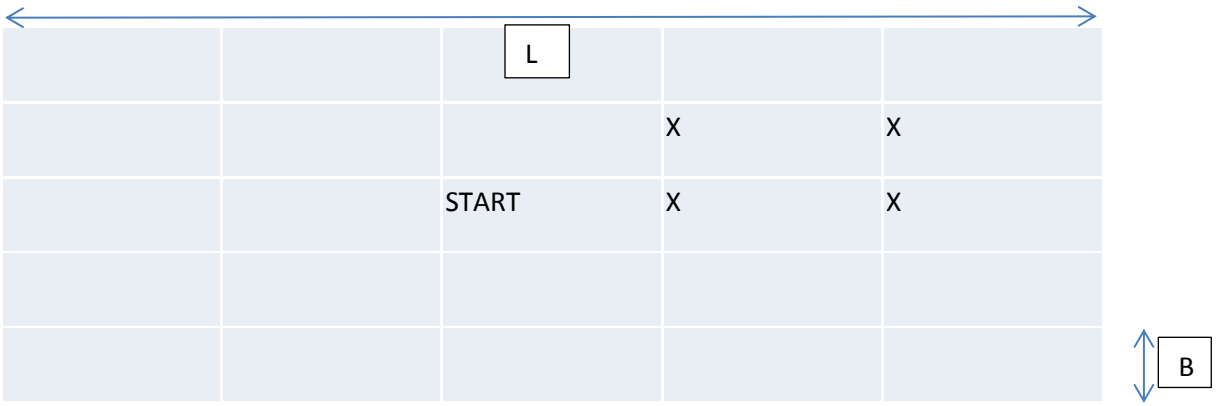


Figure 1 Problem Statement (Not to Scale)

Legend
X := object in map
START := starting point

In our implementation, the car moves in unit steps *u*. The user can specify how large B is in terms of *u*, and how large L is in terms of B. A block may be 8 unit steps, and the area to explore might be 6 blocks by 6 blocks.

Although we have no hard real time constraints, beyond detected obstacles quickly, ideally, the car should do the exploration as quickly and efficiently as possible.

Our system can accomplish these tasks. Starting from a toy rc car, we modified the hardware to provide a car that can be controlled from the Arduino. We added components to make it easy to rotate the car and to detect obstacles.

On the software side, we wrapped the hardware in a movement API that provides unit steps in the forward and reverse directions, and rotates the car in 90 degree increments in place. The depth-first search exploration algorithm is built on top of this. Finally, the user needs a way to grab the collected data from the car. We store the data in EEPROM, which keeps data across power cycles. Once the exploration is complete, the user can read the data over the Arduino serial interface, put the binary format into a parser, and get a user-readable visualization of where the obstacles are in the grid.

See Also

In addition to this report, please take a look at our README in "doc/README" and test plan and results in "tests/test plan.txt" and "tests/results.txt." The test plan, at the very least, will show you the system in action.

Hardware

Before going into the details of the hardware, note that it does not take into account all the vagaries of the real world. We assume that the car moves on a flat surface, does not run out of power, does not get caught on any objects, etc.

The hardware platform encompasses the following features

- Two rotation mechanisms. The first relies on using a compass that tells the Arduino the heading relative to north. The second uses an LED to notify the user when the car wants to rotate; when the rotation is complete the user presses a pushbutton.
- A bumper. Our bumper system is built out of chopsticks and rubber bands. Effectively, when the car bumps into an object, a contact is made between two pieces of metal. For testing, the bumper can be simulated using a pushbutton.
- An rc car. We stripped the car, figured out which pins controlled forward/reverse/left/right and soldered wires to these pins. These pins are now controlled by the Arduino.
- Arduino Uno
- 9V battery to power the Arduino

See the appendix for a detailed procedure on how to construct the hardware.

Materials

Arduino, toy radio control car, 9V battery, 5mm power plug, 22 AWG wires, breadboard, LEDs, 1K resistors, pushbuttons, compass IC

Improvements

The main problems are that the compass rotation mechanism is slow, and that the bumper is unreliable. The car does not have an in-place rotation mechanism, so we have to make many small movements of the car to approximate an in place rotation. Making large sweeping movements would be inaccurate because we have no way to measure absolute position, so it would be likely that after the rotation we would not be where we started.

System Picture

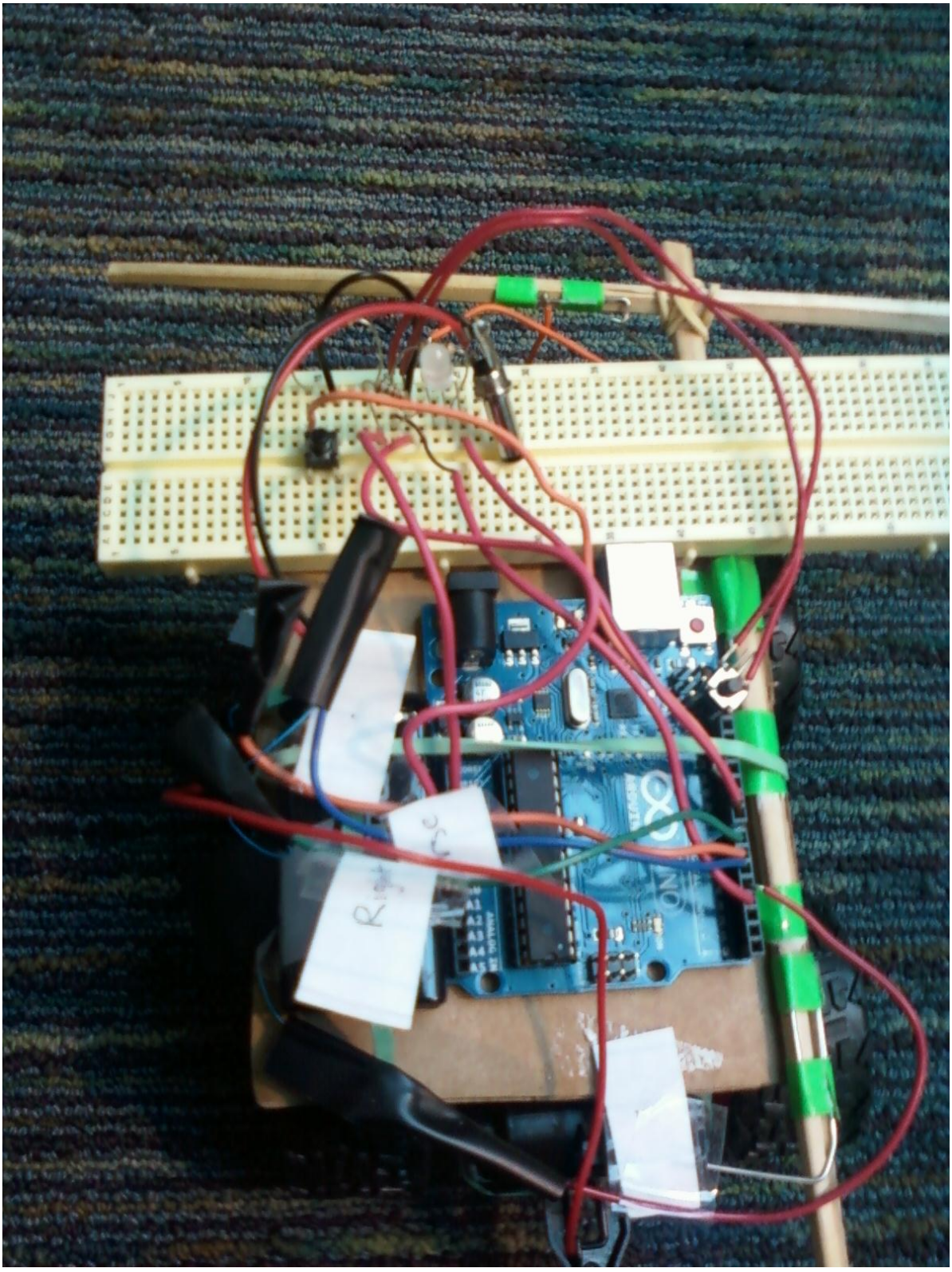
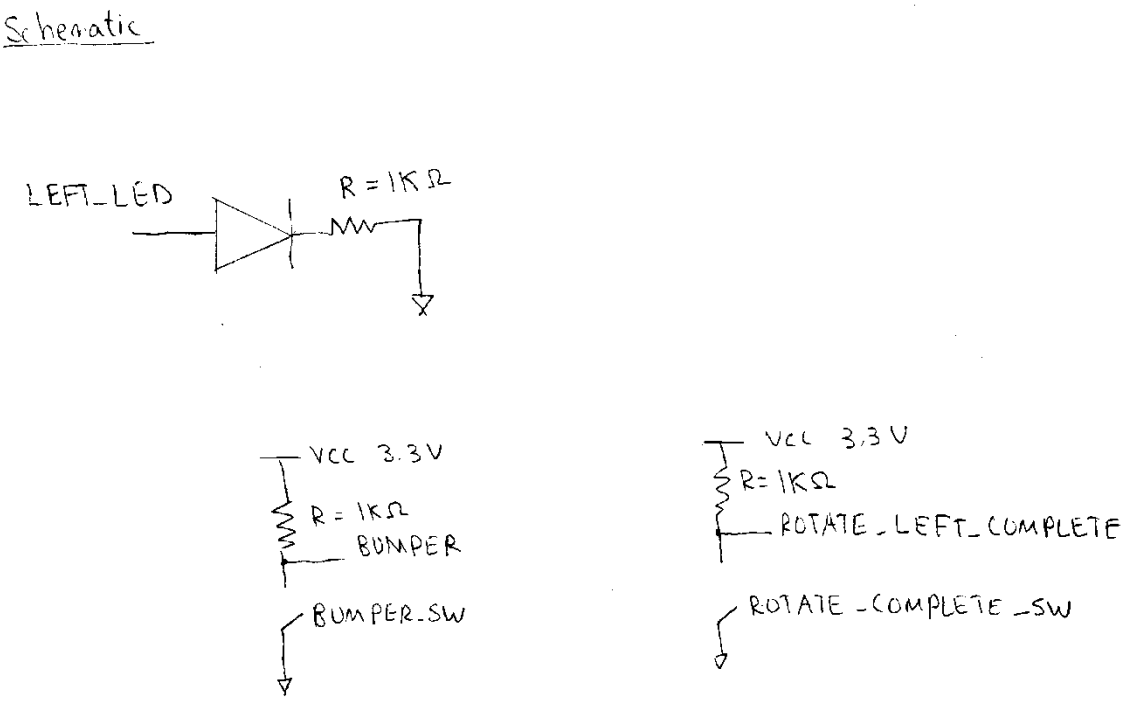


Figure 2 System Picture

Schematic

The schematic below shows the manual rotation system (LED and pushbutton), and the bumper.



Software Design

High Level Description

We have implemented a set of libraries for the purpose of interfacing with the rc (radio-control) car. Our libraries provide a movement application programming interface, a depth first search exploration algorithm, and a persistent storage interface for the arduino.

Contents

Our libraries, modeled after the existing standard arduino libraries, sit in three separate directories:

/usr/share/arduino/libraries/movement_api
/usr/share/arduino/libraries/explore
/usr/share/arduino/libraries/protocol

Individual Library Descriptions

movement_api

This library abstracts interaction with the arduino on a pin level to higher level functions including rotateLeft(), unitForward(), and unitReverse(). We envision this library as the most basic layer for any interaction with the rc car and plan to expand the depth of interaction supported. Examples include rotation by a user specified degree and composition of fundamental movements e.g. rotate left *and* move forward.

explore

The purpose of the explore library is to provide an initial algorithm for exploration of an unknown space. The relevant parameters for this library include the exploration radius and the block size. The exploration radius limits the maximum distance from the start location from which to explore and the block size is the number of unitForward() calls that make up a single virtual grid cell in our virtual map of the space to explore.

In this library we have specifically implemented a depth first search algorithm that impels the rc car to explore in a forward, left, right, and backwards order. Upon encountering an obstacle, the rc car moves backwards to the last known unobstructed location and continues searching. Underneath, we abstract the unknown space to be explored as a grid which is itself represented as an array. Each grid cell is assigned two bits of information, one demarcating an explored status and the other demarcating an obstructed status. Upon completion of algorithm, the rc car is left at the start location.

protocol

The protocol library provides an abstraction over the arduino's EEPROM enabling simpler persistent storage. The main use case for this library was storing the results of the exploration algorithm in a persistent manner so that it could later be pulled from the arduino and visualized. Analogously, we also implemented two parsers for visualizing the bit packed data stored on the Arduino's EEPROM.

Improvements

We have identified a few key areas in our system that need improvement.

Algorithm

We see an immediate opportunity for optimization in the depth first search algorithm implemented in the explore library. Currently, the algorithm rotates the rc car to face a particular direction regardless of prior knowledge that could be used to skip these rotations. This optimization is low hanging and could be implemented in a few hours.

Additionally, we could keep track of when the algorithm has explored all possible cells based on the exploration radius and end the exploration process at that point. Rather than relying on recursion to return to the start position, we could use a shortest path graph search algorithm such as Dijkstra's to find a path back to the start position.

Finally, there are myriad alternatives to the algorithm that we implemented for exploration. Rather than a depth first search, we could have implemented a breadth first search. This would be beneficial if we wanted to explore closer to the origin of the search before searching far out regions. If we knew something beforehand about the space to be

explored such as a target, we could use an algorithm such as A*.There are also more advanced algorithms for exploration of unknown spaces. A quick search returns the following paper: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00182671

Movement API

As mentioned earlier, we would like to generally expand the library to support more complex but presumably common movements such as diagonal and arcing motions. Finally we would like to add support for rotation for a user specified degree.

System Performance

Overall, the system works. In our simulation and testing (against, see the "test plan"), the car executed the exploration algorithm correctly. Of course, most of our testing involved using the manual rotation mechanism rather than the compass rotation mechanism. Our testing also involved setting off the bumper by hand, rather than using actual objects. We are confident that our system's software works, however the hardware is lacking.

In terms of algorithm size, our average program size is about 8K out of 32K on the Arduino. This increases as the grid we are exploring grows. In terms of speed, we took some measurements on how many squares the algorithm takes to explore an empty space. The grid has $(\text{grid dimension})^2$. Since there are no obstacles, the optimal algorithm can essentially snake through the grid. Our algorithm does slightly worse.

Grid Dimension	Number of Steps	Better Approach	Note
4	30	16	real data
6	70	36	real data
8	126	64	real data
10	198	100	extrapolated
12	286	144	extrapolated
14	390	196	extrapolated
n	2*n^2-2	n^2	extrapolated

Appendix A: Hardware Construction

Wiring

Procedure

- Tips
    - Wash your hands after performing any soldering, especially if the solder has lead.
    - The rc car is cheap, so breaking it isn't too costly.
- Time: 1 day
- Open up the rc car until you reach the circuit board, which takes input from the radio controller and manages the motors on the car. A screwdriver and some force are required. Document any dismantled systems that might be difficult to put back together (e.g. the gearing that connects the motors to the axle).
- To determine which leads control the directions (forward, reverse, left, right), determine which IC is the receiver chip. This will probably be the IC with the most pins. The other blocks are probably just transistors. Walk through each of the pins, and see which one changes when the remote is pushed forward. The directional pins will be digital pins. Follow the same procedure for the other pins. Use a multimeter here, and be sure to connect one lead to ground. Don't let it float.
- Solder some thin wires on top of the leads. To be extra-safe, one could remove the receiver chip entirely. Otherwise, be careful not to use the microcontroller and remote simultaneously to avoid damaging the h-bridges. The best technique with such thin wires is to apply the solder to the wires first, and then stick them on the IC pins. Examine the connections under a microscope for any shorts or opens.

- Insulate any open metal with heat shrink tubing or electrical tape. Label which wires are forward, reverse, etc.
- The Arduino needs a portable power source. We chose a 9V battery. Find a 2.1 mm inner diameter power plug that will plug into the Arduino (outer diameter can be 5 or 5.5 mm). Solder it to the 9V batter clip. The Arduino uses a center-positive connection.

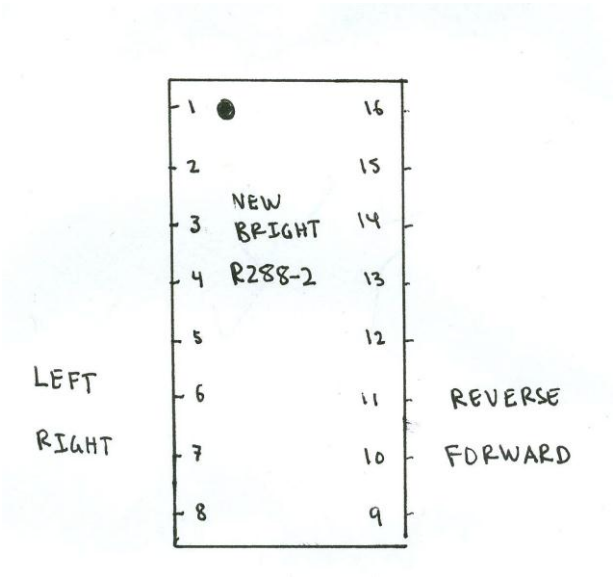


Figure 3: Receiver IC Pinout

References

Search the New Bright chip name to get at some good sources.

http://www.et.byu.edu/~bmazzeo/LTR/tech.phtml. New Bright IC.
http://forum.allaboutcircuits.com/archive/index.php/t-62791.html. New Bright IC.
http://www.clear.rice.edu/elec201/Book/hardware.html. H-bridge.
http://store.curiousinventor.com/guides/How_To_Solder/. Soldering