LAB 2

IROBOT OPEN INTERFACE AND MOVEMENT

INTRODUCTION

This week you will program the CyBot to implement basic movement actions. Your attention will be focused on two components of the CyBot: the iRobot Create and a serial I/O interface on the microcontroller. Communication between the iRobot and microcontroller is supported over a standard RS232 serial interface using a UART (Universal Asynchronous Receiver Transmitter) on the microcontroller (UART4). You will learn implementation details for UART communication in a later lab. In this lab, communication is abstracted using the iRobot's "Open Interface" specification, which allows you to send commands to the iRobot. The commands that you will primarily use are those related to controlling movement and getting the state of the iRobot, such as sensor values. A simple API (Application Programming Interface) has been written for you that handles such actions as setting the wheel speed and retrieving sensor data.

REFERENCE FILES

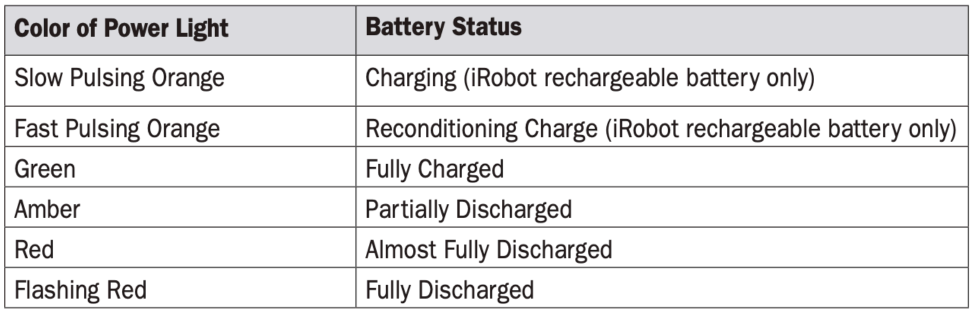
The following reference files will be used in this lab:

- open_interface.c, API functions for basic Open Interface functions
- open_interface.h, header file for open_interface.c
- lcd.c, program file containing various LCD functions
- lcd.h, header file for lcd.c
- timer.c, program file containing various wait commands
- timer.h, header file for timer.c
- iRobot Create 2 Open Interface Specification

The code files are available to download.

CYBOT CHARGING

Turn off the power to the CyBot platform before charging and check the charging status. The TI Launchpad board consumes power, and the battery will not charge properly when the board is left on. Additionally, ensure that the CyBot is turned off completely. The battery will not charge if the iRobot is in Full Mode, which it enters after calling oi_init. The battery status of the iRobot is shown with these LED colors:



Color of Power Light	Battery Status
Slow Pulsing Orange	Charging (iRobot rechargeable battery only)
Fast Pulsing Orange	Reconditioning Charge (iRobot rechargeable battery only)
Green	Fully Charged
Amber	Partially Discharged
Red	Almost Fully Discharged
Flashing Red	Fully Discharged

Double-Check When Writing Your Code

To save memory, call oi_alloc() and oi_init(…) only once at the beginning of the program. You will need to pass the pointer (oi_t *) to your functions. The call to oi_init() will also initialize the serial connection between the microcontroller and iRobot, and this is necessary before sending commands.

You must call oi_free() at the end of your program to free program memory and allow the bot to charge.

Prelab

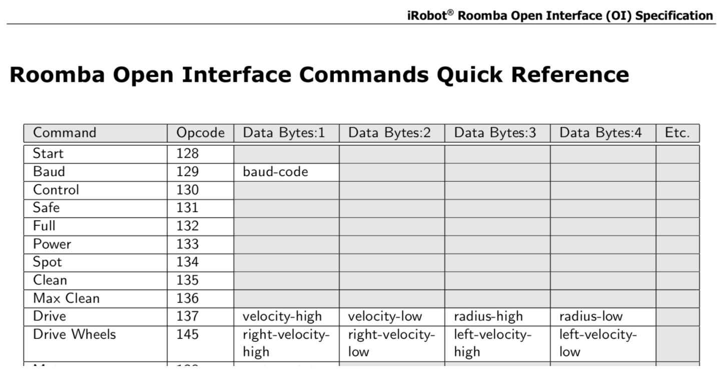
See the prelab assignment in Canvas and submit it prior to the start of lab.

Structured Pairing

Read about structured pairing (see documents posted with the lab). During lab, you and your lab partner will exchange roles for each part of the lab. The roles you will switch between are "Driver" and "Navigator." Your TA(s) will review the expectations of these roles and help facilitate your use of these roles during lab.

Open Interface Commands and Sensors

The Open Interface Specification is essentially an instruction book for how to program the iRobot Create.

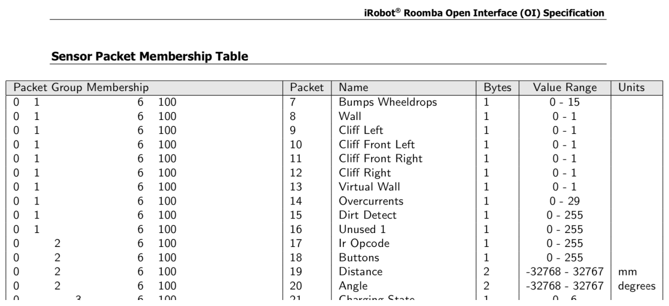


iRobot® Roomba Open Interface (OI) Specification

Roomba Open Interface Commands Quick Reference

Command	Opcode	Data Bytes:1	Data Bytes:2	Data Bytes:3	Data Bytes:4	Etc.
Start	128					
Baud	129	baud-code				
Control	130					
Safe	131					
Full	132					
Power	133					
Spot	134					
Clean	135					
Max Clean	136					
Drive	137	velocity-high	velocity-low	radius-high	radius-low	
Drive Wheels	145	right-velocity-high	right-velocity-low	left-velocity-high	left-velocity-low	

(page 33)



iRobot® Roomba Open Interface (OI) Specification

Sensor Packet Membership Table

Packet Group Membership	Packet	Name	Bytes	Value Range	Units
0 1 6 100	7	Bumps Wheeldrops	1	0 - 15	
0 1 6 100	8	Wall	1	0 - 1	
0 1 6 100	9	Cliff Left	1	0 - 1	
0 1 6 100	10	Cliff Front Left	1	0 - 1	
0 1 6 100	11	Cliff Front Right	1	0 - 1	
0 1 6 100	12	Cliff Right	1	0 - 1	
0 1 6 100	13	Virtual Wall	1	0 - 1	
0 1 6 100	14	Overcurrents	1	0 - 29	
0 1 6 100	15	Dirt Detect	1	0 - 255	
0 1 6 100	16	Unused 1	1	0 - 255	
0 2 6 100	17	Ir Opcode	1	0 - 255	
0 2 6 100	18	Buttons	1	0 - 255	
0 2 6 100	19	Distance	2	-32768 - 32767	mm
0 2 6 100	20	Angle	2	-32768 - 32767	degrees

(page 39)

Part 1: Moving Forward

Your first task is to build a program executable in CCS that will make the CyBot drive forward 1m.

1. To get started quickly, open Code Composer Studio and copy your project from Lab 1.

Right click on Lab1 in the project workspace and select 'copy'. Next, right click in the project workspace and select 'paste'. When prompted, name the project 'Lab2'. Modify main.c to call functions for movement that you will write in your movement API.

****Tip:** If you do not see a window in CCS titled project explorer, open it by selecting View → Project Explorer.

****Tip:** When opening CCS, if you accidentally select the wrong folder as your workspace, would like to change the folder that you are currently working in, or you open CCS and none of your files are showing, then ensure that you are in the correct workspace by selecting File → Switch Workspace → Other and selecting the correct directory as your workspace. CCS will close and relaunch using the workspace you selected.

2. Copy in the project files open_interface.c and open_interface.h that have been provided to you on Canvas (link to shared folder with code files).

3. Create a new API (two new files: movement.c, movement.h) that contain new functions you will write to move the CyBot.

To create the C file, select File → New → Source File and name the file "movement.c" in the box titled 'Source file'.

Create the h file in a similar way. Select File → New → Header File and name the file "movement.h".

Remember that a .h file is a header file and is associated with a .c file. A header file contains C function declarations and macro definitions that can be shared between source files.

Use a header file in your program by including it with the C preprocessing directive #include. By using #include, the compiler will access the functions from another source file.

The header file should contain:

```
#ifndef HEADER_FILE

#define HEADER_FILE



Function headers and macro definitions



#endif
```

4. In order to make the CyBot move forward 1m, write a function called
   double move_forward (oi_t *sensor_data, double distance_mm)
   in your movement.c file that takes in a distance, in mm, that your robot should move. The function should be written so that it can be called from your main file and only requires one modification to change the distance that the robot will move.

The Open Interface API has already been written for you that handles setting the wheel speed and retrieving sensor data. You will need to use the oi_setWheels() function to tell the CyBot to move. The oi_setWheels() function calls the "Drive Direct" command (opcode 145) as given in the Open Interface specification (page 14) (or "Drive Wheels" in the Commands Quick Reference table on page 33). As you can see in the OI specification, the wheel motors can be set to run independently from each other. The velocity ranges from -500 mm/s (backward) to 500 mm/s (forward).

Beyond making the CyBot move, you will also need to track how far it has moved. The same Open Interface API supports fetching sensor data from the iRobot Create. The oi_update(oi_t * self) function should be passed a pointer to an oi_t structure. The function will then update every member of the structure with the current sensor value. You should attempt to gain an understanding of the oi_t structure described in open_interface.h and its relationship with the sensor packets listed in the Open Interface specification document.

Here is some sample code (it is not complete):

```
#include "open_interface.h"

void main() {

        oi_t *sensor_data = oi_alloc(); // do this only once at start of main()

        oi_init(sensor_data); // do this only once at start of main()

        // the following code could be put in function move_forward()

        double sum = 0; // distance member in oi_t struct is type double

        oi_setWheels(500,500); //move forward at full speed

                while (sum < 1000) {

                        oi_update(sensor_data);

                        sum += sensor_data -> distance; // use -> notation since pointer

                }

        oi_setWheels(0,0); //stop

        oi_free(sensor_data);  // do this once at end of main()

        }

}
```

****Note:** Ensure that the iRobot Create is turned on by making sure the power button is lit.

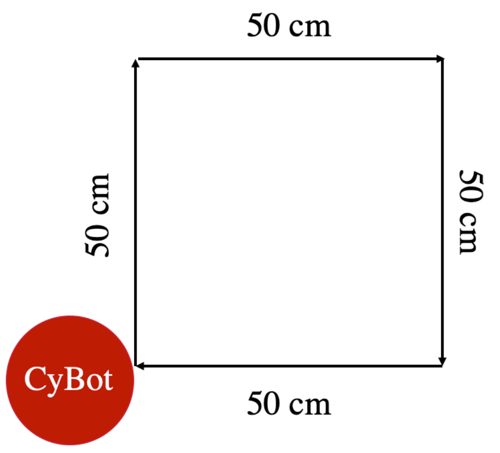
If it is not lit, press the 'Clean' button in the center of the robot. It will turn green to indicate that it is on. If the color is red or orange, it is an indication that the CyBot has low battery and will need to be charged. To charge the CyBot, turn the power to the CyBot off and ensure that it is on the charger properly. The CyBot will slowly pulse orange to indicate that it is charging.

Reminder: You must call oi_free() at the end of your program to free program memory and allow the bot to charge.

CHECKPOINT:

Move the CyBot forward 1 m and display the distance that the robot has traveled as it moves.

PART 2: MOVING IN A SQUARE



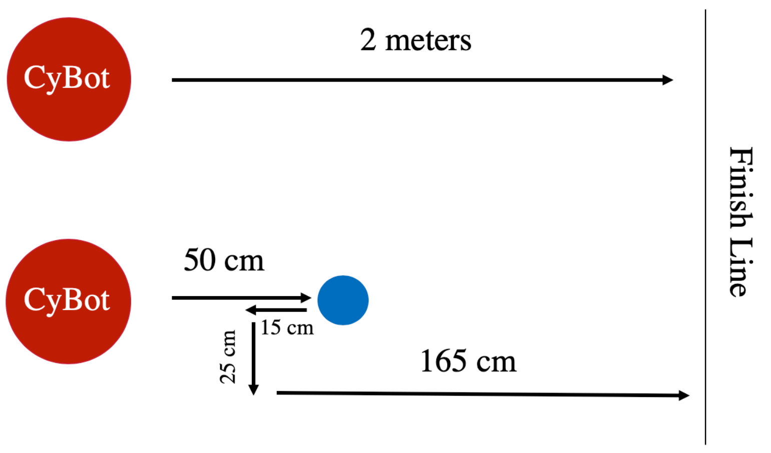
Write two new functions, turn_right(oi_t *sensor, double degrees) and turn_left(oi_t *sensor, double degrees), in your movement API. These functions will allow you to turn the robot right or left by a certain number of degrees. Use these functions in main() so that your robot moves in a 50 cm square and then stops. That is, do the following 4 times: move forward 50 cm, turn 90 degrees. Each turn should be close to 90 degrees. Use the angle member of the oi_t struct.

**Note: Positive angles are counter-clockwise; negative angles are clockwise.

CHECKPOINT:

Move the CyBot in a square.

PART 3: COLLISION DETECTION



The iRobot Create has many sensors. There are two bump sensors in the front of the device. In this part, reuse, modify, and expand the functions in the movement API and write a short program in main so that the CyBot moves forward 2 meters. The CyBot should be able to bump multiple times in its 2-meter run. The specifications of the movement are outlined below.

If no obstacle is present in the path of travel, the robot should simply travel 2 meters forward and stop. However, if the robot comes in contact with an object, the robot should attempt to go around the object by issuing the following commands:

1. Back up 15 cm
2. Turn 90 degrees in the opposite direction of the bump sensor that was triggered
   - If the collision occurs with the right bumper, the robot should initially spin 90 degrees to the left.
   - If the collision occurs with the left bumper, it should spin 90 degrees to the right.
   - If both sensors report a collision, then pick a direction and perform a 90-degree turn.
3. Move laterally 25 cm
4. Turn 90 degrees forward
5. Resume traveling forward the entire 2 meters and stop

The robot should be able to encounter at least two objects and successfully reach the finish line. However, your program does not need to handle the case of a robot hitting an object while in the process of going around an object.

You could design your move_forward() function to stop if it runs into an object and return the current distance travelled.

To get the status of the left and right bump sensors, use the *bumpLeft* and *bumpRight* members of the oi_t structure. These members are boolean values. A nonzero (true) value means that the bump sensor is colliding with an object. For example:

```
oi_update(sensor_data); // get current state of all sensors
if (sensor_data->bumpLeft) {
   // respond to left bumper being pressed
...
}
```

CHECKPOINT:

Move the CyBot a total of 2m, detect collisions, and navigate around them.

DEMONSTRATIONS:

1. **Functional demo of a lab milestone –** Specific milestone to demonstrate in Lab 2: Move the robot forward, detect bumping into an obstacle, move backward and turn 90 degrees in the opposite direction of the bump (right bump, turn left; left bump, turn right). This is a subset of Part 3.
2. **Debug demo using debugging tools to explain something about the internal workings of your system –** The TA will announce any specific debugging requirements at the start of lab; otherwise you will create your own debug demo based on your needs and interests in the lab.
3. **Q&A demo showing the ability to formulate and respond to questions –** This can be done in concert with the other demos.