ECE Team Report

Intel-Cornell Cup USA

Spring 2015

Made for students,
by students

Tyler Walker • Steve Bryden
Alex Rucker • Ashwath Laxminarayana • Claire Chen • George Li
Harshit Gulati • Judy Stephen • Laura Ng • Mikhail Rudinskiy
Rameez Qurashi • Shela Wang • Siyu Liu • Sungjoon Park
Syed Tahmid Mahbub • Tony Chau • Vaidehi Garg
Xiaohan Liu • Yu-Che Eric Hsueh • Yuqi Mark Zhao

REPORT | INTEL-CORNELL CUP USA | ACADEMIC ADVISOR: DR. DAVID SCHNEIDER

Table of Contents

1 FPGA

1.1 PID Controller

1.1.1 Introduction

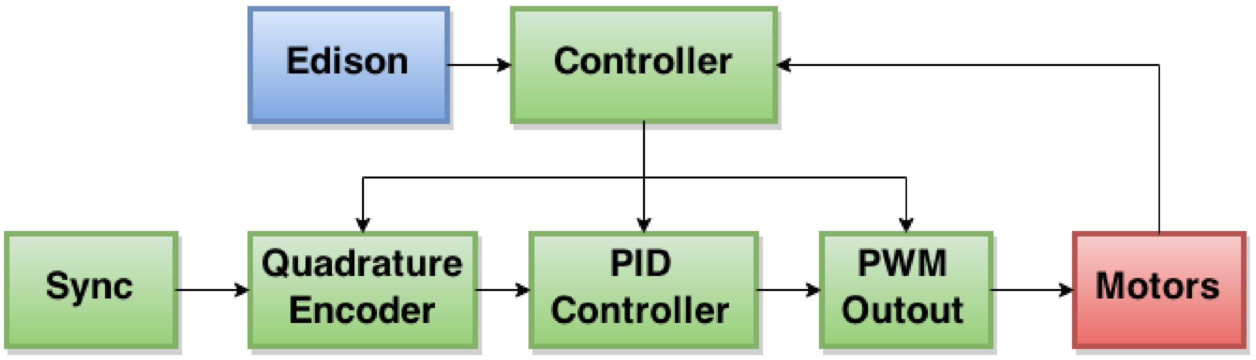
For this project, we chose to implement a PID controller in an FPGA. PID control is a common scheme used in many applications to control physical systems, in which a control signal derived from the error, integral of error, and derivative of error is applied to a system. To calculate the feedback, the equation

$$K_P * e + K_I * \int_{-\infty}^{t} e \, d\tau + K_D \frac{de}{dt} \text{ .... (1)}$$

is used. Each term in this equation has a specific purpose. The first term is one of the simplest control schemes possible: measure the error (distance of the system from the desired position), multiply it by some value, and use that as the correction factor. This scheme is not perfect, however. Consider the impact of a constant external force, such as gravity. There is a point where the impact of the external force on the system is balanced by the control input. At this point, the system will settle into at a point away from what is desired.

To counteract this problem, the second term is added. As the system remains away from a set point, the integral term will accumulate, and, eventually, force the output back to the desired output. The derivative term is not always necessary, but it can help to lead to a faster response from the system. To determine the correct values for the PID coefficients, a tuning method is required. Although there are many techniques, including mathematical analysis of the system, to determine these coefficients, one simple one is Ziegler-Nichols tuning.

The overall block diagram of the PID controller system is shown below.



1.1.2 Quadrature Encoder Interface

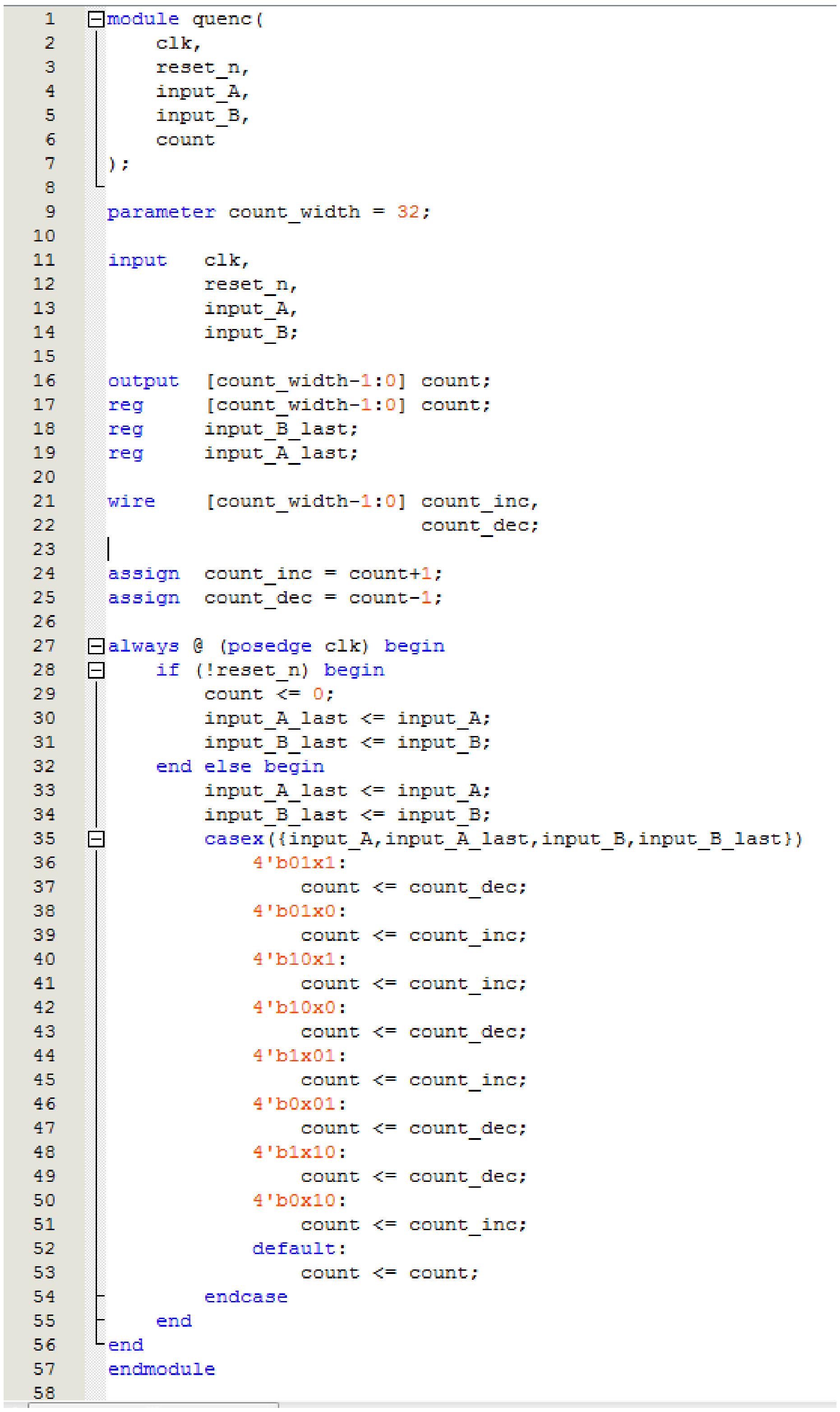
To determine the current position, a quadrature encoder is used. This encoder generates two square waves, A and B. The frequency of the square wave is related to the speed with which the encoder rotates, and the phase of the square waves is related to the direction of rotation (if the encoder is rotating one way, A will go high before B, otherwise, B will go high before A). These characteristics can be used to build a very simple interface to a quadrature encoder, in order to measure the distance traveled. Inputs are synchronized into the internal clock domain, and are fed into a block which checks for rising and falling edges. If a certain transition is found, the encoder's counter is either incremented or decremented, depending on whether the transition is associated with forwards or backwards movement.

1.1.2.1 Input Synchronization
This is a key portion of any hardware design that deals with inputs from the outside world. In Verilog, everything is handled relative to a clock. This allows the design tools, such as Quartus, to analyze the final design to see how well it meets timing requirements, because the fixed clock period means that the tools know when it is possible for signals to change. However, signals from the outside can change at any time, potentially arriving at forbidden times, resulting in a problem known as *metastability,* where a signal takes on a state between zero and one. To prevent this, the signals are delayed using a chain of flip-flops. This gives time for the signal to settle before it is used.

1.1.2.2 Edge Detection and Decoding
For simplicity, detecting which signals had edges and whether the transitions indicated by the edges were encoded in one *casex* statement, using the current and previous states of the signals as inputs. Using the assumption that both signals do not transition at the same time (reasonable, considering that the transitions are well-spaced relative to the extremely fast FPGA clock), it was possible to encode each (transition, direction) pair using the old and new values of one signal as well as the new value of the other signal. These then select whether the value of the counter is the old value, the old value decremented by one, or the old value incremented by one. The value of the counter is then the position, as measured by the encoder. Using every applicable transition for measurement has the advantage that it is more accurate.

1.1.2.3 Reset Logic
This module uses a negative reset, *reset_n.* Therefore, when the reset is low, the module will enter a reset condition. This initializes the counter to zero. However, it is important to note that it does not zero the registers used for edge detection. Instead, they are initialized to the value of the inputs. This is important for the accuracy of the module: if they were initialized to a constant value, then there is a 50 percent chance that there would be an edge detected immediately. There is also the more threatening possibility that there would be two simultaneous edges detected, which would violate one of the fundamental design assumptions made for this module.

1.1.2.4 Source Code



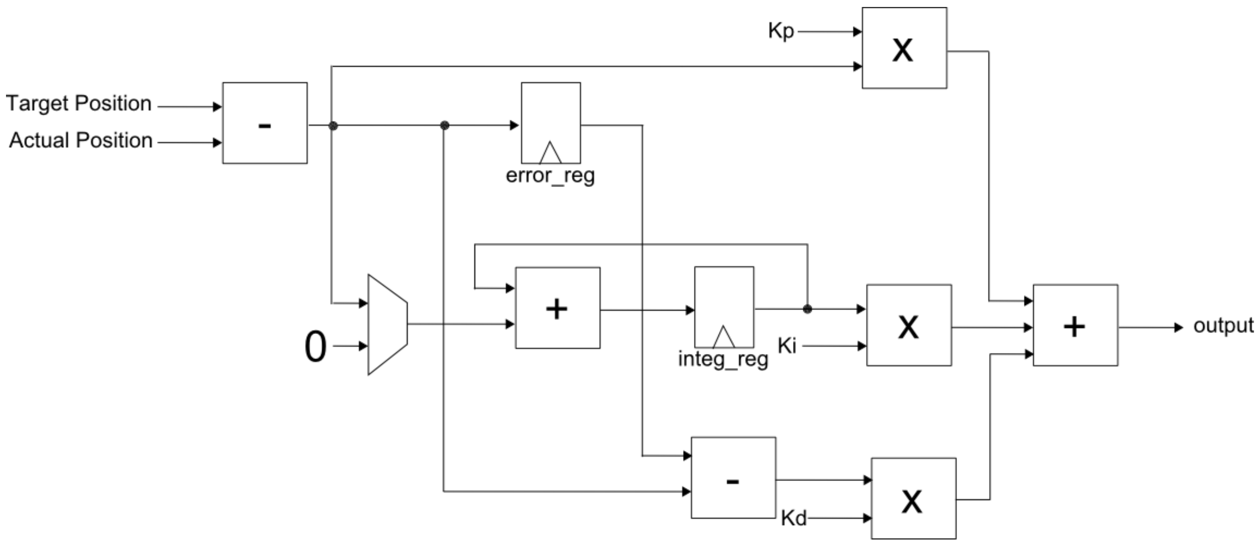
```verilog
module quenc(
    clk,
    reset_n,
    input_A,
    input_B,
    count
);

parameter count_width = 32;

input   clk,
        reset_n,
        input_A,
        input_B;

output  [count_width-1:0] count;
reg     [count_width-1:0] count;
reg     input_B_last;
reg     input_A_last;

wire    [count_width-1:0] count_inc,
                          count_dec;

assign  count_inc = count+1;
assign  count_dec = count-1;

always @ (posedge clk) begin
    if (!reset_n) begin
        count <= 0;
        input_A_last <= input_A;
        input_B_last <= input_B;
    end else begin
        input_A_last <= input_A;
        input_B_last <= input_B;
        casex({input_A,input_A_last,input_B,input_B_last})
            4'b01x1:
                count <= count_dec;
            4'b01x0:
                count <= count_inc;
            4'b10x1:
                count <= count_inc;
            4'b10x0:
                count <= count_dec;
            4'b1x01:
                count <= count_inc;
            4'b0x01:
                count <= count_dec;
            4'b1x10:
                count <= count_dec;
            4'b0x10:
                count <= count_inc;
            default:
                count <= count;
        endcase
    end
end
endmodule
```

Figure 1: Source code for the quadrature decoder.

1.1.3 PID Controller Interface

After the Quadrature Encoder Interface determines the current position, it feeds in the current position value to the PID Controller module whose task is to calculate the error in the current position and then determine the feedback value which it transfers to the PWM output module. In order to determine the error in the current position, the PID controller module takes in the actual position (desired position) as an input from the main Controller module. It also takes in the constants Kp, Ki and Kd as inputs from the controller module. The output of the PID Controller module is a value which tells both the direction and the speed with which the motors should rotate to get closer to the desired position.

The datapath diagram for the PID Controller module is shown below.



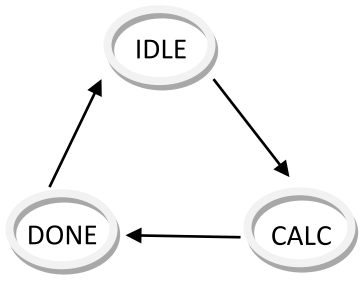
The first step in determining the overall error is calculation of the Proportional error. This is done by a Subtractor subA which takes the actual and target positions as the input and calculates the error in the current position. The output of this subtractor is fed into three places. First, it goes into a multiplier which multiplies the error with the proportional constant Kp and outputs the net proportional error. Second, the output from the subtractor goes into a 2:1 mux which helps in finding the integral error and third, the output is latched into a register to help in the calculation of the differential error.

The next step in determining the overall error is the calculation of the Integral error. The Integral error correction is nothing but the error correction taking the accumulated error into the account. To store the running sum of the error, a register “integ_reg” is instantiated. The input of this register is the output of an adder which adds the current error to the previous error. Since just after reset the error will be undefined, a 2:1 mux is placed before the adder. For the first time, the mux outputs a 0 and from then on the mux outputs the current error determined by the Subtractor subA. The output of the integ_reg goes into a multiplier which multiplies the integral error with the constant Ki.

The next step in feedback calculation is determining the differential error. The differential error is calculated by subtracting the current error with the previous error. The value of the previous error is read from the register error_reg. Then another subtractor sub calculates the differential error and feeds it into a multiplier which then multiplies it with the constant Kd.

Finally, the three errors, proportional error, integral error and the differential error are all added up by an adder. The output of this adder is the output of this module which goes into the PWM output module.

The control logic for the PID controller module uses a simple 3-states state machine shown below. The default state is the IDLE state. As soon as there is a valid request (req_val goes high) to calculate the feedback from the main controller module, the PID controller module goes into the CALC state where it triggers the Kp, Ki and Kd calculations. After all the calculations are over, the state machine goes into the DONE state where it is ready to output the result. The PWM output module waits for the “resp_val” signal to go high before sending the outputs to the motors.



1.1.4 PWM Output Interface

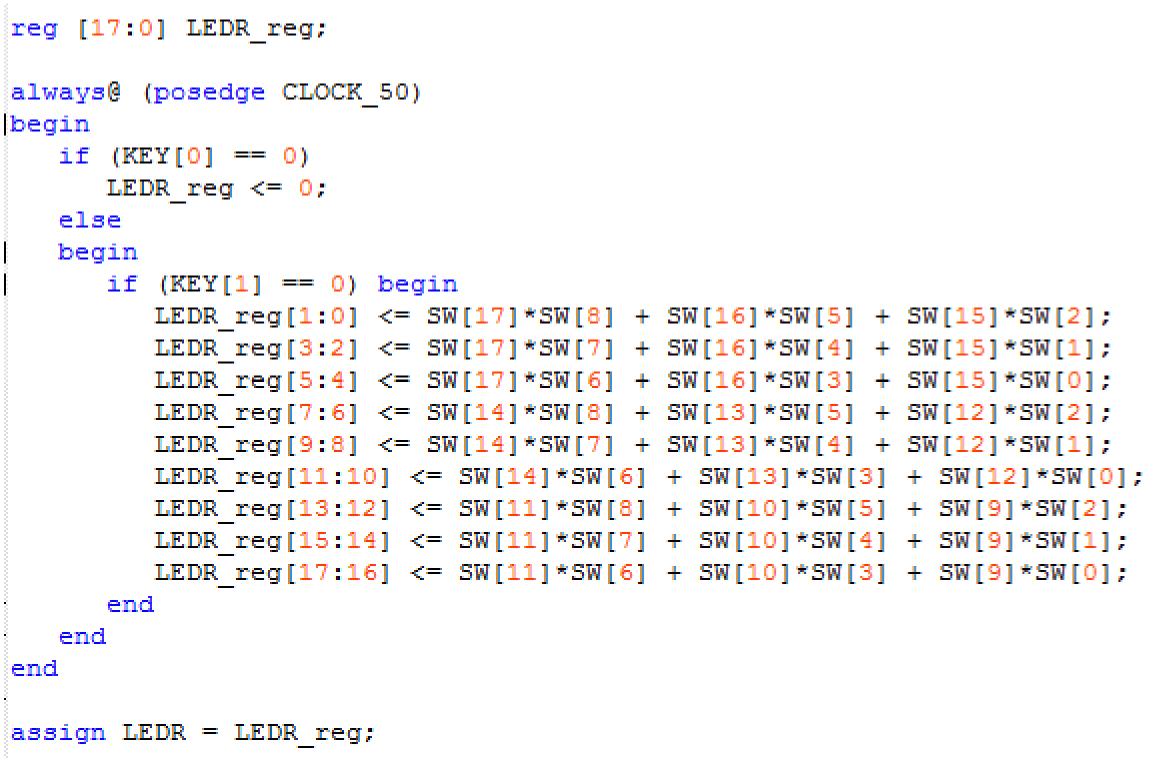
The input to the PWM output module is an 8-bit value. The MSB of this value tells the direction (anticlockwise or clockwise) in which the motors should rotate in order to correct itself and bring to the actual position. The remaining 7 bits are used to generate two PWM outputs to control the two motors.

1.2 Matrix Multiplier

This is a simple example code that we implemented as a part of FPGA guide for the students who will be joining the team next year. Since there was no proper example code for the FPGA team, we implemented two versions of matrix multiplication.

This matrix multiplier will calculate the product of two 3 by 3 matrices. 18 switches on the board will act as inputs and the red LEDs will show the output. More specifically, from the switch 17 to 9, the first matrix inputs are set up, and from the switch 8 to 0, the second matrix inputs are set up. Once the inputs are all put in, KEY[1] will be pressed to show the values in the register that holds the calculated value on the red reds. these will be read from red LEDs 0 to 17. To clear up the register and turn off the red LEDs, KEY[0] (reset) can be pressed.

1.2.1 Single Cycle Version Code



```
reg [17:0] LEDR_reg;

always@ (posedge CLOCK_50)
begin
   if (KEY[0] == 0)
      LEDR_reg <= 0;
   else
   begin
      if (KEY[1] == 0) begin
         LEDR_reg[1:0] <= SW[17]*SW[8] + SW[16]*SW[5] + SW[15]*SW[2];
         LEDR_reg[3:2] <= SW[17]*SW[7] + SW[16]*SW[4] + SW[15]*SW[1];
         LEDR_reg[5:4] <= SW[17]*SW[6] + SW[16]*SW[3] + SW[15]*SW[0];
         LEDR_reg[7:6] <= SW[14]*SW[8] + SW[13]*SW[5] + SW[12]*SW[2];
         LEDR_reg[9:8] <= SW[14]*SW[7] + SW[13]*SW[4] + SW[12]*SW[1];
         LEDR_reg[11:10] <= SW[14]*SW[6] + SW[13]*SW[3] + SW[12]*SW[0];
         LEDR_reg[13:12] <= SW[11]*SW[8] + SW[10]*SW[5] + SW[9]*SW[2];
         LEDR_reg[15:14] <= SW[11]*SW[7] + SW[10]*SW[4] + SW[9]*SW[1];
         LEDR_reg[17:16] <= SW[11]*SW[6] + SW[10]*SW[3] + SW[9]*SW[0];
      end
   end
end

assign LEDR = LEDR_reg;
```

The code for the single cycle multiplier is shown above. This design has a total of 27 multipliers since all multiplication operations are done in parallel in a single clock cycle. First, a register for storing the calculated matrix multiplication is generated that will show on the red LEDs later. The register is cleared when KEY[0] is pressed so that it turns off all the LEDs. If KEY[1]] is pressed it will do the calculation and store its value to the register. Finally, the result is going to be passed onto the red LEDs from the registers that we created in the beginning. This design is fully parallelized and fast since everything is done in a single clock. However, it uses total 27 multipliers which requires more area. Also, the multiplication and addition operations create a huge combinational block which may affect the overall timing of the design and the synthesis tool might lower the clock frequency to meet the timing.

1.2.2 Multiple Cycle Version Code

```
reg [3:0] k;
reg  result_done;
reg start;
always @(posedge CLOCK_50)
begin
      if(KEY[0] == 0)
      begin
         start <= 1'b0;
         k <= 0;
         result_done <= 1'b0;
      end
      else if (KEY[1] == 0)
      begin
         start <= 1'b1;
         k <= 0;
         result_done <= 1'b0;
      end
      else if (start)
      begin
         if (k == 0) begin
             LEDR_reg[1:0] <= SW[17]*SW[8] + SW[16]*SW[5] + SW[15]*SW[2];
         end
         else if (k == 1) begin
             LEDR_reg[3:2] <= SW[17]*SW[7] + SW[16]*SW[4] + SW[15]*SW[1];
         end
         else if (k == 2) begin
             LEDR_reg[5:4] <= SW[17]*SW[6] + SW[16]*SW[3] + SW[15]*SW[0];
         end
         else if (k == 3) begin
             LEDR_reg[7:6] <= SW[14]*SW[8] + SW[13]*SW[5] + SW[12]*SW[2];
         end
         else if (k == 4) begin
             LEDR_reg[9:8] <= SW[14]*SW[7] + SW[13]*SW[4] + SW[12]*SW[1];
         end
         else if (k == 5) begin
             LEDR_reg[11:10] <= SW[14]*SW[6] + SW[13]*SW[3] + SW[12]*SW[0];
         end
         else if (k == 6) begin
             LEDR_reg[13:12] <= SW[11]*SW[8] + SW[10]*SW[5] + SW[9]*SW[2];
         end
         else if (k == 7) begin
             LEDR_reg[15:14] <= SW[11]*SW[7] + SW[10]*SW[4] + SW[9]*SW[1];
         end
         else begin
             LEDR_reg[17:16] <= SW[11]*SW[6] + SW[10]*SW[3] + SW[9]*SW[0];
             start <= 1'b0;
             result_done <= 1'b1;
         end
      k <= k + 1;
      end
end

assign LEDR = result_done ? LEDR_reg : 0;
```

This is the code for the multiple cycle multiplier which uses a priority encoder. This design does not have as many multipliers as the single cycle one because same multipliers would be used multiple times. This time, three more registers were created: one for counter, one for result signal and one for a simple state machine. Similar to the single cycle version, when KEY[0] (reset) is pressed, all the registers are cleared and when KEY[1] is pressed all the calculations are computed and the result is stored in the LEDR_reg register. when KEY[1] is pressed, the start state goes to 1, counter resets and result signal goes to 0. This result_done register is set once the calculation is done and the resultant matrix output is latched to the output LEDs. Hence, in this design the output latency is 9 clock cycles.

These two examples were implemented not only to compare the single cycle and the multi cycle calculations on matrix multiplications but also to give the new FPGA team members general ideas to run simple FPGA codes such as use of registers, combinational blocks, simple state machines and assigning different input/outputs to the various wires. Also, it shows how to use LEDs, KEYs and switches to show the results and use them as deb

2 Introduction to Linux

The Intel Edison that will be covered in the sections below has an Operating System of Yocto (an embedded platform that is a Linux Operating System). This section will describe a few commands that are generally used in Linux and will help you do you basic operations in Linux.

Note: This is not the whole list of commands in Linux, there are many tutorials and guides online that can help you find specific commands that are needed. Listed here are the ones that were used the most and the ones that give you a basic understanding on how to get comfortable in a Linux environment.

2.1 List Command in Linux

The 'ls' Command

The 'ls' command is a list command. It lists all the files and directories in the folder or path you are currently in. The 'ls' command and some of its useful addendums and explanations are listed below.

The ls command has the format:

Command	Action
ls *dir*	list files in directory *dir*; if *dir* is left empty, ie the command is just ls, the action is to list files in the current working directory
ls -l *dir*	list files in directory *dir* in long format, with all the permissions, ownerships, date and size of the file; if *dir* is left empty, the action takes place on the current working directory

2.2 Directory Commands in Linux

2.2.1 The ‘mkdir’ command

The ‘mkdir’ command is a ‘make directory’ command. It creates a directory(folder) in the specified directory *dir*. If dir is left out/empty, a directory(folder) is created in the current working directory.

Example : To make a directory called CornellCup in the current working directory

```
mkdir CornellCup
```

After typing this command and hitting the ‘Enter’ button a directory called CornellCup is created in the current folder. To see if this directory exists use the ‘ls’ command to display the files and directories in the current directory.

Example: To make a directory called CornellCup in the directory /home/

```
mkdir /home/CornellCup
```

2.2.2 The ‘rmdir’ command

The ‘rmdir’ command is a remove directory command. It removes(deletes) the mentioned directory.

Example: To delete the CornellCup directory

```
rmdir CornellCup
```

After typing this command and hitting the ‘Enter’ button the directory called CornellCup is deleted from the current folder. To see if this directory is deleted use the ‘ls’ command to display the files and directories in the current directory. Use this command with caution.

Example: *To delete the /home/CornellCup directory from a different directory*

```
rmdir /home/CornellCup
```

2.2.3 Moving between Directories

To move between directories we use the ‘cd’ command, the ‘cd’(change directory) command is a useful command and is necessary for when moving between directories. Commands and their uses are listed below.

Example : Create a directory called CornellCup and go into it and then go back to home directory

```
mkdir CornellCup  - Makes the CornellCup directory

cd CornellCup     - goes into the CornellCup directory

cd ..             - goes back up one directory
```

cd /sys/class - changes current working directory to /sys/class

2.3 File Commands in Linux

2.3.1 Creating a file in Linux

To create a file in Linux, we use an editor, the most commonly used editor is a VI editor. This editor has its own list of commands and its mode of operations that are necessary to know to use it comfortably. These commands will be listed in the 'VI Editor' Section below.

To create a file in Linux the command is

vi *filename*	if you want to make a file without an extension
vi *filename.extension*	if you want to make a file with an extension

Example: Create a file called hello and a C file called hello.

vi hello (to quit file go to the 'Quit File' subsection in 'VI Editor' Section)

vi hello.c (to quit file go to the 'Quit File' subsection in 'VI Editor' Section)

Hit the 'Enter' button after typing each command, both files can be found when you type 'ls' in the terminal. If a text file is needed to be created instead, the extension for that is ".txt".

Note: hello and hello.c are two different files, even though they have the same name, they can be distinguished in Linux by their extension.

2.3.2 Removing a file in Linux

To remove a file in Linux, we type in the command 'rm' followed by a filename and extension if the file has one. This command will not work on directories.

Let us remove the files that we created in the section above.

Example: Remove files made in the previous section

rm hello

This command will remove the file called hello

rm hello.c

This command will remove the file called hello.c

After typing each line of command and entering it into the system. The file specified will not be found in the system. Typing 'ls' will show you the directory of the folder without the files mentioned above.

Like previously, it is possible to take action on a file not in the current working directory by specifying its location. Eg:

rm /usr/scripts/hello.sh

2.3.3 VI Editor

The VI Editor is an editor where programs can be created, notes taken, etc… It can be considered as a file creator and editor and it has many commands associated with it.

Because it is a terminal editor, sometime using it can be tricky.

There are three modes in the VI Editor, command mode, insert mode and command line mode. We will cover a basic understanding and use of each mode and a few of their important commands.

2.3.3.1 Command Mode

When you created your file 'hello' by using the vi command a different screen showed up with a blinking cursor and - "hello" [new file] - at the bottom of your screen.

This is still in the command mode where you can command the file by using some keyboard prompts.

An example of this is the 'ZZ' command which saves the file and closes it. However there must be text in the file else the 'ZZ' command will not save the file.

Some commands to use in the command mode:

2.3.3.1.1 Movement Commands:

While not commands, you can move the cursor across text and edited lines by using the arrow keys on the keyboard. However you must have inserted text or edited lines in the terminal.

2.3.3.2 Insert Mode

To enter text in VI, insert mode must be activated. There are several commands to enter insert mode, however we will only cover two essential commands. To exit this mode you must hit the 'escape' key on your keyboard to back into Command Mode.

When in the VI terminal, hit the 'i' key and start typing. The 'i' key is an insert key and after you hit it, it will allow you to insert in characters to the vi.

The second command that will allow you to enter text in the terminal is the 'a' key meaning append the text.

The difference between the ‘i’ key and the ‘a’ key is where you want the text to appear, if you wish to insert text before previous text, the ‘i’ key is useful here, if you wish to type text after a certain point the ‘a’ key should be used.

To go to a next line, hit the ‘enter’ key on your keyboard.

To navigate from line to line, see *Movement Commands* in Command Mode.

2.3.3.3 Command line mode

This mode is to see commands in the command line, used mostly for saving the file and quitting the file. To enter this mode, you need to be in the command mode and then hit the ‘:’ button (Shift + ;), after hitting the ‘:’ button the cursor will go to the bottom of the screen with a ‘:’ showing up. You can enter commands into this line and they will be executed after hitting enter.

2.3.3.4 Saving and Quitting Commands

There are several save and quit commands in VI, all commands follow a ‘:’ that is not shown below.

Commands	**What it does**
w	‘writes’ the file, a save function
q	quits the file, if the file has not been saved, an error will show up [E37: No write since last change(add ! to overwrite)]
wq	‘write and quit’, this saves the file and then quits it
q!	If you want to quit the file without saving changes to it, use this command.

2.4 Other Useful Commands

This section will cover some more useful commands that will be seen in the sections later on. The commands covered here are: mv, cp, shutdown, pwd, head, and tail commands.

2.4.1 The ‘mv’ Command

The ‘mv’ command is a move command and can be applied to files and directories. It can also be used to rename files. The format for this command is

Examples:

mv filename ~/directory/filename	- moves file to new directory location
mv filename newfilename	- renames file to newfilename

mv directory ~/directory2/directory	- moves directory into another directory
mv directory newdirectoryname	- renames directory to newdirectoryname

2.4.2 The 'cp' Command

The 'cp' command is a copy command that is used for copying files or directories. This is useful if there is a file you need copied to another directory or another file format. When the 'cp' command is invoked if the file that you want to copy to exists, it will overwrite all the content of the intended file with the content of the copy file. If the intended file does not exist then it will create a new file and copy the contents of the old file.

Examples:

cp filename filename2 -copy contents to filename to filename2

cp filename1 filename2 ~/directory -copy contents to directory, if this directory is opened it will have filename1 and filename2 existing in it.

cp -r directory1 directory2 -copy directory1 and all it's contents into directory2.

Note that to copy a directory '-r' must be used '-r' is an option flag which tells the copy command to copy the directory recursively.

2.4.3 The 'shutdown' Command

To shutdown from the terminal we use a 'shutdown' command with option flags. This is helpful when using microcontrollers where there is no GUI. The 'shutdown' command has the following format.

Examples:

sudo shutdown -h now	-this shuts down the linux OS immediately
shutdown –h now	-this is the command you use for the Edison

The shutdown command can only be initiated by the super-user, thus the 'sudo' command is issued here. If your OS has a password associated with it, after calling this command, a password prompt will be given, if you do not have a password then the system should shutdown immediately.

Note for Edison you have to be logged in as the root user, so you do not need the *sudo* command before the shutdown command.

2.4.4 The 'pwd' Command

The 'pwd' command is used to find out which directory you are working in. This is a useful command because you can use it for finding out where you are relative to the home directory.

Example:

pwd - This is the pwd command entered

/user/home/directory	- This is the resultant
cd ..	- Going up one folder
pwd	- pwd command again
/user/home	- resultant

2.4.5 The 'head' Command

The 'head' command displays a number of lines from the beginning of the file to the line specified of the file. This can help find out the initial contents of a file.

Example:

head filename	- This command will output the first 10 lines of filename
head -n20 filename	- This command will output the first 20 lines of filename

2.4.6 The 'tail' Command

The 'tail' command displays a number of lines from the end of the file working backward to the line specified of the file. This can help find out the ending contents of a file.

Example:

tail filename	-This command will output the last 10 lines of filename
tail -n20 filename	-This command will output the last 20 lines of filename

2.5 Further Readings

There are many resources online for commands on Linux and Unix operating systems. These are just a few to give you a flavor and to cover the commands you might encounter later on in this documentation.

3 Intel Edison

3.1 MRAA Library Installation

The MRAA library allows the user to abstract the Intel Edison operations to high level software function calls, without having to deal with them in Linux or through file descriptors. Intel provides the detailed API description in **(Reference [1])**.

In order to install MRAA on an Edison, enter the following lines of code **(Reference [2])**:

```
1. echo "src mraa-upm http://iotdk.intel.com/repos/1.1/intelgalactic" >
/etc/opkg/mraa-upm.conf
2. opkg update
3. opkg install libmraa0
4. opkg upgrade
5. npm install mraa
```

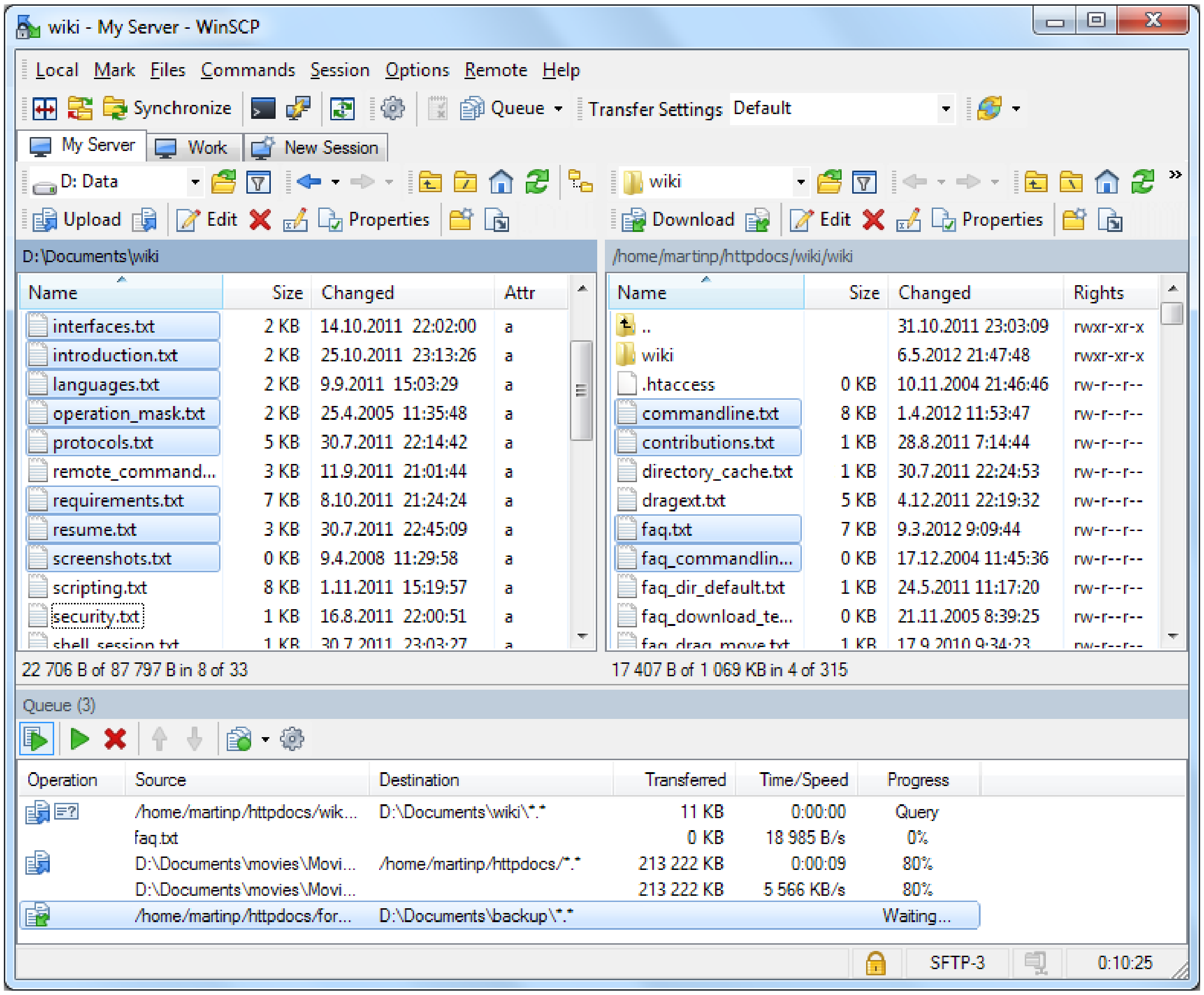
References:

[1] http://iotdk.intel.com/docs/master/mraa/index.html

3.2 Transferring files to the Edison

****Reference Fall 2014 documentation for more information. This section builds on the findings established there.*

The easiest way to transfer files has been to use WinSCP **(Reference [1])** to transfer files from Windows to the Edison. WinSCP allows file transfer through a drag-and-drop GUI. On Linux the equivalent functionality can be obtained using the scp command through the CLI. Here is a screenshot of WinSCP from the website **(Reference [2])**:


wiki - My Server - WinSCP
Local Mark Files Commands Session Options Remote Help
Synchronize
Queue
Transfer Settings Default
My Server
Work
New Session
D: Data
wiki
Upload
Edit
Properties
Download
Edit
Properties

D:\Documents\wiki

Name	Size	Changed	Attr
interfaces.txt	2 KB	14.10.2011 22:02:00	a
introduction.txt	2 KB	25.10.2011 23:13:26	a
languages.txt	2 KB	9.9.2011 15:03:29	a
operation_mask.txt	2 KB	25.4.2005 11:35:48	a
protocols.txt	5 KB	30.7.2011 22:14:42	a
remote_command...	3 KB	11.9.2011 21:01:44	a
requirements.txt	7 KB	8.10.2011 21:24:24	a
resume.txt	3 KB	30.7.2011 22:45:09	a
screenshots.txt	0 KB	9.4.2008 11:29:58	a
scripting.txt	8 KB	1.11.2011 15:19:57	a
security.txt	1 KB	16.8.2011 22:00:51	a
shell_session.txt	1 KB	30.7.2011 23:03:27	a

22 706 B of 87 797 B in 8 of 33

/home/martinp/httpdocs/wiki/wiki

Name	Size	Changed	Rights
..		31.10.2011 23:03:09	rwxr-xr-x
wiki		6.5.2012 21:47:48	rwxr-xr-x
.htaccess	0 KB	10.11.2004 21:46:46	rw-r--r--
commandline.txt	8 KB	1.4.2012 11:53:47	rw-r--r--
contributions.txt	1 KB	28.8.2011 7:14:44	rw-r--r--
directory_cache.txt	1 KB	30.7.2011 22:24:53	rw-r--r--
dragext.txt	5 KB	4.12.2011 22:19:32	rw-r--r--
faq.txt	7 KB	9.3.2012 9:09:44	rw-r--r--
faq_commandlin...	0 KB	17.12.2004 11:45:36	rw-r--r--
faq_dir_default.txt	1 KB	24.5.2011 11:17:20	rw-r--r--
faq_download_te...	0 KB	21.11.2005 8:39:25	rw-r--r--
faq_drag_move.txt	1 KB	17.9.2010 9:34:23	rw-r--r--

17 407 B of 1 069 KB in 4 of 315

Queue (3)

Operation	Source	Destination	Transferred	Time/Speed	Progress
	/home/martinp/httpdocs/wik...	D:\Documents\wiki\*.*	11 KB	0:00:00	Query
	faq.txt		0 KB	18 985 B/s	0%
	D:\Documents\movies\Movi...	/home/martinp/httpdocs/*.*	213 222 KB	0:00:09	80%
	D:\Documents\movies\Movi...		213 222 KB	5 566 KB/s	80%
	/home/martinp/httpdocs/for...	D:\Documents\backup\*.*			Waiting...

SFTP-3 0:10:25

Image Source: (Reference [3])

References:

[1] http://winscp.net/eng/index.php

[2] http://winscp.net/eng/docs/ui_commander

[3] http://791840461.r.cdn77.net/data/media/screenshots/commander.png?v=178

3.3 PS4 controller via Bluetooth

****Reference Fall 2014 documentation for more information. This section builds on the findings established there.*

The PS4 controller is a Bluetooth HID and is extremely easy to connect with, using Linux. To start it in pairing mode, the PS and share buttons should be held together until the light starts flickering quickly. Using Bluetoothctl, the device can then be paired, trusted and connected the following way:

```
bluetoothctl
agent KeyboardDisplay
default-agent
pairable on
scan on
```

At this point, look for the controller ID for the PS4 controller. Once the ID is located:

```
pair <ID>
trust <ID>
connect <ID>
```

At this point, the white light should stop flickering and turn a solid white. Then exit bluetoothctl:

```
exit
```

This sequence of pairing and connecting is required only once. After this is done, whenever the PS4 controller is turned on (short press on the PS button), it will search for a device to connect to. When the Edison is powered on, the connection will be done automatically. The application code can then proceed to use the PS4 controller. It is a device that can be read from /dev/input/event2 in the Linux environment. The Linux header file to go through in detail will be <linux/input.h> that is very important for the events associated with the PS4 controller as an input device.

All gamepad related functions and operations are presented in the two files: ps4.h and ps4.c. The function gp_init initializes the gamepad. It does so by trying to open /dev/input/event2 as read-only and non-blocking. It continues attempting to do so until opening it no longer fails. This means that, the initialization routine attempts to open the PS4 related event infinitely until the PS4 connects to it successfully. For this to happen, it must already be paired to the Edison using the steps mentioned above.

Closing the “file” /dev/input/event2 closes the gamepad, as is done in the gp_quit function.

The most important function with using the gamepad is gp_getEvent. The function gp_getEvent attempts to read the file /dev/input/event2 if any valid information is present, ie any gamepad related event has occurred. If the read fails (if the file read returns number of bytes read as lower than that expected based on the size of the event struct), the program registers that as no event occurring. However, if the read does succeed, it means that an event has occurred. The input_event struct is then checked to retrieve information about the event.

From <linux/input.h> the input_event struct is shown below:

```
/*
 * The event structure itself
 */

struct input_event {
        struct timeval time;
        __u16 type;
        __u16 code;
        __s32 value;
};
```

The <linux/input.h> file also defines the events and button presses, as well as absolute axes and many other useful definitions.

The type field in the input_event struct is checked to see if it was of type EV_ABS (one of the digital direction pins was pressed) or of type EV_KEY (one of the buttons was pressed). The results are stored in an event_t struct:

```
typedef struct
{
        evtype_t        type;
        int                     btn;    // keys / mouse/joystick buttons
        int                     x;              // mouse/joystick x move
        int                     y;              // mouse/joystick y move
} event_t;
```

The type field is not used. The x and y fields are populated when a digital direction button is pressed, using the value field from the input_event struct. The btn field is populated when a button is pressed, using the code field from the input_event struct.

Once these are done, the gp_postEvent function is called that performs a certain action based on the event that occurred.

Thus, the PS4 operation via Bluetooth is simplified to file read and structure handling operations due to the abstraction provided by Linux made possible by the Edison drivers.

The code base for the PS4 is developed off of the Edidoom application mentioned in the references section.

References: http://2ld.de/edidoom/

Intel Edison Bluetooth Guide: https://communities.intel.com/docs/DOC-23618

Source for <linux/input.h>: source: http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/include/linux/input.h?v=2.6.11.8

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>       // SUPER IMPORTANT - FOR EVENTS
#include "ps4.h"

const char* gamepadName = "/dev/input/event2";
volatile int gamepad = -1;
event_t joystickEvent = { ev_joystick };
int rcv = -1;
int upd_x = -1, upd_y = -1;

int jx=0,jy=0,jbtn=0;

int get_rcv(void)
{
        return rcv;
}

int get_updx(void)
{
        return upd_x;
}

int get_updy(void)
{
        return upd_y;
}

void get_joystickEvent(int* x, int* y, int* btn)
{
        *x = jx;
        *y = jy;
        *btn = jbtn;
}

void gp_postEvent(event_t joystickEvent)
{       // joystickEvent contains 3 ints: btn, x, y
        static int oldx=0, oldy=0;

        upd_x = -1;
        upd_y = -1;

        jx = joystickEvent.x;
        jy = joystickEvent.y;

        //if (joystickEvent.btn)
        //{
                jbtn = joystickEvent.btn;
        //}

        if (joystickEvent.x != oldx)
        {
                upd_x = 0;
        }

        if (joystickEvent.y != oldy)
        {
                upd_y = 0;
        }

        oldx = joystickEvent.x;
        oldy = joystickEvent.y;
}

void gp_init(void)
{
        do {
                gamepad = open(gamepadName, O_RDONLY | O_NONBLOCK);
                //if (gamepad < 0)
                //fprintf(stderr, "Cannot access gamepad at %s\n",
gamepadName);
        } while (gamepad < 0);
        //else{
                //fprintf(stdout, "Accessed gamepad at %s\n", gamepadName);
        //}

}

void gp_quit(void)
{
        if (gamepad >= 0)
                close(gamepad);
}


void gp_getEvent(void)
{
                struct input_event ev[64];
```

```
		joystickEvent.btn = 0;

		int rd = read(gamepad, ev, sizeof(ev));

		if (rd < (int) sizeof(struct input_event))
		{
			rcv = -1;
			return;
		}

		int i;
		for (i = 0; i < rd / (int)sizeof(struct input_event); i++)
		{
			struct input_event e = ev[i];

			switch (e.type)
			{
			case EV_ABS:
				if (e.code == ABS_HAT0X)		//
Digital pins L and R
					joystickEvent.x = e.value;
				else if (e.code == ABS_HAT0Y)  // Digital pins
U and D
					joystickEvent.y = e.value;
				break;
			case EV_KEY:
				// Map keys (SEE HEADER FILE):
				joystickEvent.btn = e.code;		//
Button pressed
				break;
			}

			gp_postEvent(joystickEvent);
			// Do something with this info
			rcv = 0;
		}
}
```

3.4 Edison PWM with MRAA

DC motor control with PWM is made simple by the MRAA PWM library. This abstracts the PWM operation to library calls that setup the PWM module, its frequency and duty cycle. The main functions are described in the Intel MRAA PWM API Description (Reference [1]).

The steps for PWM usage are:

- Initialize the PWM module using mraa_pwm_init or mraa_pwm_init_raw. It is easier to use the mraa_pwm_init() function and pass as the pin parameter the PWM pin to use. For the Arduino board, the PWM pin parameter is the pin number on the board. For the mini breakout board, the PWM pin parameter is the MRAA pin number as specified in the Intel breakout board physical to MRAA pin number mapping (Reference [2]). The function result is of type mraa_pwm_context, and is passed to the functions for setting period (frequency), duty cycle and enable.
- The PWM period is defined by using mraa_pwm_period (period defined as a float, in seconds), mraa_pwm_period_ms (period defined as int, in milliseconds) or mraa_pwm_period_us (period defined as int, in microseconds).
- The PWM duty cycle is defined by using mraa_pwm_write (duty cycle defined as a percentage indicated by the float parameter between 0.0 and 1.0), mraa_pwm_pulsewidth (duty cycle defined by specifying pulse width as float, in seconds), mraa_pwm_pulsewidth_ms (duty cycle defined by specifying pulse width as int, in milliseconds) or mraa_pwm_pulsewidth_us (duty cycle defined by specifying pulse width as int, in microseconds).
- The PWM channel is enabled using mraa_pwm_enable.
- Instead of separately specifying the period and duty cycle, they can done together using mraa_pwm_config_ms or mraa_pwm_config_percent.
- The PWM channel can be closed using mraa_pwm_close. One thing to be careful of is that, this will disable the PWM channel wherever it was in the period. So, the output could be left high. The way to deal with this would be to make the corresponding pin a digital output after closing the PWM channel and writing it low.

The MRAA library also provides other fairly useful PWM functions. One odd observation with the PWM module has been that setting the PWM period to a value "too precise" (this showed up when the period was to be set for 44.1 kHz frequency), the frequency reverted to a "default" of 660 Hz. However, operation at lower resolution frequencies such as 1 kHz and 2 kHz have worked fine.

Once the PWM module has been initialized, the parameters can be changed on the fly using one of the functions mentioned above, for changing period and duty cycle, or enabling/disabling the module.

By using the MRAA PWM functions for speed control and the MRAA IO functions for direction control, motor control can be easily done with the Edison.

References:[1] http://iotdk.intel.com/docs/master/mraa/pwm_8h.html

[2] http://iotdk.intel.com/docs/master/mraa/edison.html

3.5 Edison IO with MRAA

Similar to the PWM, IO operations are abstracted to simple function calls using the MRAA library (Reference [1]).

The steps for initializing and using a pin for IO operations are:

- Initialize the GPIO pin using mraa_gpio_init or mraa_gpio_init_raw. For the pin to pass as parameter, if the Arduino board is being used, the pin refers to the pin labelled on the Arduino board; if the mini breakout board is used, the pin refers to the MRAA pin number as specified in the Intel breakout board physical to MRAA pin number mapping (Reference [2]). The function result is of type mraa_gpio_context, and is passed to other GPIO functions.
- The direction should be set using the mraa_gpio_dir function.
- If output, the value can be written to high or low using the mraa_gpio_write function. If input, the value can be read using the mraa_gpio_read function. A GPIO external interrupt can be configured using mraa_gpio_isr (see Edison interrupt section).

The MRAA GPIO library provides other useful functions laid out in (Reference [1]).

References: [1] http://iotdk.intel.com/docs/master/mraa/gpio_8h.html

[2] http://iotdk.intel.com/docs/master/mraa/edison.html

3.6 Mini Modbot Application

A small robot, the "mini modbot," was developed using the PS4 controller for user input (direction and speed control) and the IO and PWM functionalities of the Intel Edison mentioned above. The Arduino board was used for this purpose.

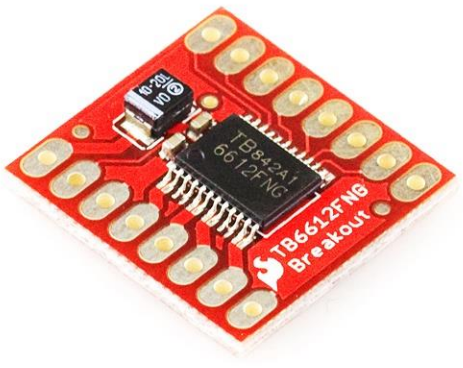


Image source: [1], The Sparkfun TB6612 dual H-bridge motor controller used to control the 2 wheel motors.

The mini modbot is meant to be modular, and easy to implement system. Our hopes are that it aids novices in the field of robotics to create an affordable, and simple robot, with basic controls. However, we want to emphasize that it has the potential to be expanded upon, and incorporated into various projects. The chassis for the mini modbot were purchased from hobbyking.com. We looked into various options based on control schemes. We found two very affordable options that come with motors already mounted on the chassis.

Image source (left): HobbyKing 4WD chassis→
http://www.hobbyking.com/hobbyking/store/__26250__4WD_Robot_Chassis_KIT_.html
Image source (right): HobbyKing magician chassis→
http://www.zagrosrobotics.com/shop/item.aspx?itemid=859

The inputs AIN1/BIN1 and AIN2/BIN2 determine the direction control for the motors while the PWMA input determines the speed control (see figure below). The motors are connected between A01 and A02, and B01 and B02.

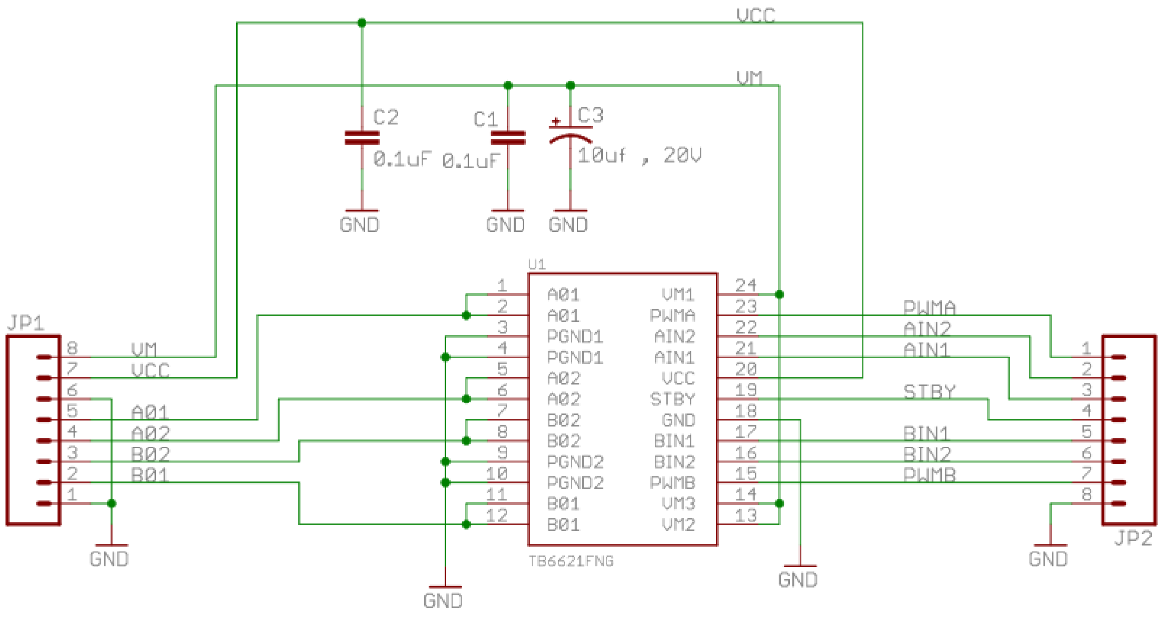


Image source: [2], Schematic of the H-bridge driver

Besides that, all of the motor control functionality are described in the motor_control.c and motor_control.h files, which can be found in the appendicies.

The digital buttons on the PS4 controller are mapped to the robot motion. Pressing up or down ramps the motor speed up/down and makes it go forward/backwards. Pressing left or right causes the robot to turn in the correct direction. Pressing the cross button makes the robot brake. When the up and down buttons are pressed, two corresponding LEDs go on and are turned off when they are released. All code is attached.

References: [1] https://www.sparkfun.com/products/9457 [2] https://www.sparkfun.com/datasheets/Robotics/TB6612FNG%20Breakout%20v10.pdf

3.7 Edison Timing

In order to initialize a Linux timer for use, the steps followed are:

1. Establish handler for signal

2. Block timer signal temporarily
3. Create timer
4. Start timer
5. Unlock the timer signal

1. In establishing the handler for the timer signal, the handler is passed to the sa_sigaction field of a struct sigaction in <signal.h> [1]. Then, the sigaction system call [3] is performed to change the action taken when the timer signal is received. This sets up the timer handler which is analogous to a timer ISR in a microcontroller.

2. The sigprocmask function [5] is called – it is used to change signal mask of the calling thread. For the caller, the signal mask is the set of signals whose delivery is blocked – SIG_SETMASK is passed as a parameter to do this.

3. In a sigevent struct, the notify field is set to signal the event to allow the handler to deal with it. The associated signal is given a signal number (sort of like an identifier) that is an integer value between the two definitions SIGRTMIN and SIGRTMAX defined in <signal.h> for application use. A timer_t object timer_sw (defined as a global variable) is the timer used and a pointer to it is passed to the sival_ptr field in the same sigevent struct. Then using the timer_create function, the timer itself is created.

4. An itimerspec struct [6] defines the timer value and the period. It is then passed as a parameter, along with the timer_t object, to timer_settime function that configures the timer and starts it.

The itimerspec struct has two fields:

```
struct timespec  it_interval        //timer period

struct timespec  it_value           //timer value
```

The timespec structs themselves have only two fields : tv_sec and tv_nsec to specify time as seconds and nanoseconds.

When the it_value in the itimerspec struct specifies a nonzero value, then timer_settime starts the timer (calling arming the timer), setting it to initially expire at the provided time. If the timer was already armed, then the previous settings are overwritten. If the it_value specifies a zero value (both the tv_sec and tv_nsec fields are zero), then the timer is stopped (disarmed).

5. To unlock the timer signal, the sigprocmask function is again called to unblock the masks previously blocked in step 2 by passing SIG_UNBLOCK as a parameter.

The function clock_gettime retrieves the time of the specified clock clk_id.

Types of clocks that can be used for clk_id (passed to timer_create) [2]:

CLOCK_REALTIME
System-wide realtime clock. Setting this clock requires appropriate privileges.
CLOCK_MONOTONIC
Clock that cannot be set and represents monotonic time since some unspecified starting point.
CLOCK_PROCESS_CPUTIME_ID
High-resolution per-process timer from the CPU.
CLOCK_THREAD_CPUTIME_ID
Thread-specific CPU-time clock.

Since CLOCK_MONOTIC gets time from an unspecified starting point, it is necessary to get an initial time in the main code and subtract that time from whatever reading is taken relative to the unspecified starting point.

Setting the it_value of the timer to zero disarms the timer, but it doesn't close or delete the timer- just stops it. When the timer is no longer needed, it can be deleted:

1. First disarm the timer. This is done by setting the it_value of the timer to zero and calling the timer_settime function.
2. Delete the timer by using the timer_delete function.

The timer handler is written as a regular function; the prototype used was:

static void timer_handler(int sig, siginfo_t *si, void *uc)

The signal to the timer can be ignored after that if the signal function [8] is called, passing the int sig and an additional parameter – the disposition. If it is SIG_IGN, the signal is ignored. If the disposition is set to SIG_DFL, the default action associated with the signal [7] occurs.

The timer related functionality used by the team is detailed in the code in timers.h and timers.c. The overall concepts are the same as mentioned here; the code has detailed comments.

References:

[1] http://pubs.opengroup.org/onlinepubs/007908799/xsh/signal.h.html

[2] http://linux.die.net/man/2/timer_create

[3] http://man7.org/linux/man-pages/man2/sigaction.2.html

[4] http://linux.die.net/man/3/sigemptyset

[5] http://linux.die.net/man/2/sigprocmask

[6] http://pubs.opengroup.org/onlinepubs/007908775/xsh/time.h.html

[7] http://man7.org/linux/man-pages/man2/timer_settime.2.html

[8] http://man7.org/linux/man-pages/man2/signal.2.html

[9] http://man7.org/linux/man-pages/man7/signal.7.html

3.8 Edison External Interrupts

To configure the Edison for IO external interrupts, after the GPIO pin is configured to be a digital input, the mraa_gpio_isr function is called passing as parameter, the pin on which to accept interrupts, the interrupt edge, the IO handler, and any arguments to pass. The function details are provided in the MRAA GPIO API description. [1]

```
mraa_result_t mraa_gpio_isr (mraa_gpio_context dev, gpio_edge_t edge, void(*fptr)(void *), void *args)
```

The interrupt edge can be defined as:

- No interrupt: MRAA_GPIO_EDGE_NONE
- Interrupt on both rising and falling edges: MRAA_GPIO_EDGE_BOTH
- Interrupt only on rising edge: MRAA_GPIO_EDGE_RISING
- Interrupt only on falling edge: MRAA_GPIO_EDGE_FALLING

The interrupt handler is just treated like an ISR in a microcontroller. No flag management is necessary, as is sometimes necessary to clear interrupt flags in a microcontroller. A sample call is shown:

```
mraa_gpio_isr(gpio_context, MRAA_GPIO_EDGE_FALLING, INT_HANDLER, NULL);

void INT_HANDLER (void* args) {

}
```

References:

[1] http://iotdk.intel.com/docs/master/mraa/gpio_8h.html

3.9 Miscellaneous – Edison

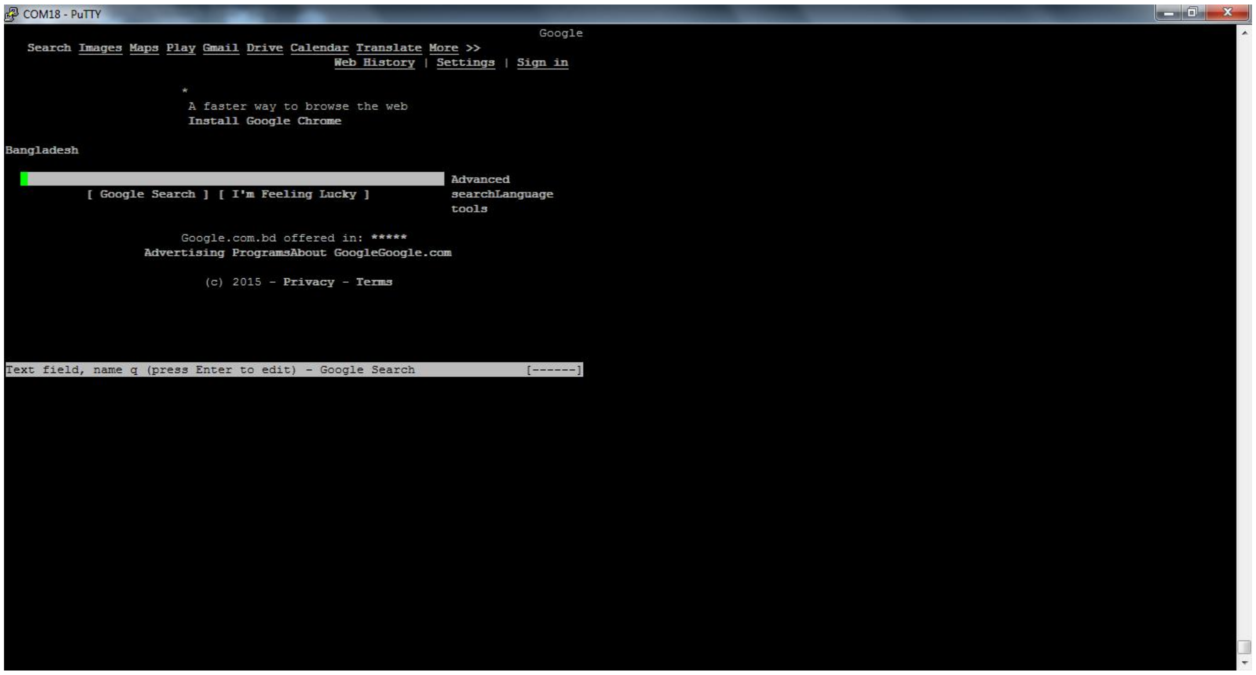
3.9.1 Elinks web browser

It is possible to browse the internet through the Intel Edison by adding a text browser. Since the Edison does not have any native video output, text browsers are the simplest way to access the internet for browsing. A fairly popular (as popular as text browsers go) option is Elinks **(Reference [1])**. The steps to installing the Elinks browser are:

- Download tar ball from : http://elinks.or.cz/download.html
- Using the tar.gz: **wget** http://elinks.or.cz/download/elinks-0.11.7.tar.gz
- Uncompress the tar ball: **tar -xzvf elinks-0.11.7.tar.gz**
- **cd** into the uncompressed directory (can check using **ls**)

Then, all that is required is to configure, make and install: **./configure && make && make install**

A screenshot of the Elinks browser viewing Google is shown below:



```
                                                                  Google
   Search Images Maps Play Gmail Drive Calendar Translate More >>
                                            Web History | Settings | Sign in

                        *
                         A faster way to browse the web
                         Install Google Chrome

Bangladesh

                                                           Advanced
         [ Google Search ] [ I'm Feeling Lucky ]           searchLanguage
                                                           tools

                        Google.com.bd offered in: *****
                   Advertising ProgramsAbout GoogleGoogle.com

                            (c) 2015 - Privacy - Terms



Text field, name q (press Enter to edit) - Google Search            [------]
```

References:

[1] http://elinks.or.cz/

3.9.2 Installing other applications

Since the Intel Edison has an x86 CPU, Linux applications can be downloaded and installed. An easy method of doing this is locating the source code for the software, downloading them using wget, decompressing the files if necessary (TAR balls are common), configuring, making and installing them. These are usually laid out in a README file in the installation directory.

Other useful applications that have been attempted on the Edison are:

- DTRX (Do The Right Extraction) – an intelligent archive extraction tool **(Reference [1])**
- Apache Web Server **(Reference [2])**

References:

[1] http://brettcsmith.org/2007/dtrx/

[2] http://httpd.apache.org/

4 Localization

4.1 iBeacons

One of the team's goals this semester was to create a wireless sensor network (WSN) that could be used to locate the R2-I2 robot within a room. The advantage of such a system would be that data regarding R2's location from the motor encoders and accelerometers, for example, could be verified against the localization information obtained from this network. Furthermore, the application could be extended in the future so that any human holding a certain device would also be locatable, opening up the possibility of the robot being able to find and move towards a user who calls for it.

4.1.1 Design Approach

The scheme that was decided upon was a network of transmitters – set-up with known locations around an area – that transmitted signals that could be picked up by a receiver on R2. Once these signals were received by R2, the distance to each transmitter would be calculated, and a triangulation algorithm would be applied to get R2's absolute location within the network.

The team considered several options for the hardware of this network. One technology that was considered was Wi-Fi. This would involve setting up various routers around a room and measuring the signal strength from each router to a receiver on R2. Another technology considered was radio beacons, which would be set-up similarly.

Ultimately, the technology decided upon was Bluetooth Low Energy (BLE) because pre-packaged BLE transmitters already existed (see the "Estimote" section below), and because there seemed to be a well-established protocol for getting signal strength, which correlates to distance, from BLE packets.

4.1.2 RSSI

Received signal strength indication (RSSI) refers to the power of a radio signal that is perceived by a receiver. RSSI is usually expressed in terms of decibels (dB) because the signal strength is expressed relative to the received signal strength when the receiver is 1 meter away from the transmitter. (This one-meter-away value is generally referred to as TX Power. "TX" is short for "transmit" or "transmission") Thus, in theory, an equation for converting RSSI to distance could be formulated.

4.1.3 Estimote

The Estimote Beacons[1] are pre-packaged iBeacons[2]. They regularly broadcast packets using BLE, and these packets can be read by BLE receivers (such as a phone that supports BLE) in order to get received signal strength indication (RSSI) readings. As previously mentioned, this signal strength correlates to distance.

The main use for the Estimote seems to be proximity sensing (i.e., "Is this person/object near or far from this beacon?"); however, for this project, the team hoped to use the RSSI readings to get distances from a receiver to Estimotes that were in the area.

4.1.3.1 Estimote App

There are apps created by the Estimote company for iOS (iPhone) and Android, which allow users to see the signals sent out by Estimotes. To find these apps, simply search "estimote" in the App Store or Google Play Store.

These apps also allow the user to modify how each Estimote sends its data, in addition to seeing the signals. To make any such modifications, you must sign into the app. The login information is as follows:

Email address: ch659@cornell.edu
Password: CornellCupRocks

A note about the iOS version of the Estimote app: When prompted, make sure to allow the app to use your location even when the app is closed; otherwise, the app might not discover nearby Estimotes.

While the Estimote comes with pre-made apps for sensing distance using mobile phones, the team hoped to build a custom receiver that either could be plugged into the Windows computer attached to R2 or could send data to some other central computer. This way, the team could use that data in other parts of the system.

[1] Estimote Beacon product page: http://estimote.com/
[2] Brief overview of iBeacons: http://www.ibeacon.com/what-is-ibeacon-a-guide-to-beacons/

4.1.3.2 Estimote Packet Structure

The Estimote can transmit a packet several times per second, and this frequency can be adjusted using the Estimote App. The packet structure for all iBeacons contains the same fields. These fields include:

- **UUID**: Estimotes all use the same UUID, which is B9407F30-F5F8-466E-AFF9-25556B57FE6D.
- **Major and Minor**: These are two identifying numbers; for example, all beacons in one building might have the same Major value, but each beacon in that building would have a different Minor.
- **Bluetooth Address**: This is unique for each beacon and is formatted like a MAC address; for example, E7:44:89:31:ED:4E.
- **TX Power**: This is the measured RSSI from 1 meter away. (See "RSSI" section above for how this value is used.)

More-detailed information on the Estimote packet structure can be found at the following URL: http://makezine.com/2014/01/03/reverse-engineering-the-estimote/

4.1.4 BLE Mini

The team's first attempt at building a receiver for the iBeacon packets used the BLE Mini. The BLE Mini[3] is a BLE module that can act as a receiver of iBeacon packets and that can communicate with a computer or microcontroller (such as an Arduino) over a serial port.

Using this module, the team found that it is possible to get RSSI readings approximately once every 10 seconds. The reason behind this delay is likely due to the way that the BLE Mini scans for nearby Bluetooth devices. The team hypothesized that after scanning and discovering devices, the module waits 10 seconds to see if it will discover any new devices before allowing a new scan to begin. In addition, it appears that this 10-second timeout is not user-configurable and is probably part of the device's firmware. According to the BLE Mini website, it may be possible to develop custom firmware for this device; however, the team decided that it would be more productive simply to consider other approaches.

Thus, as our intended application for this system was to triangulate/locate an object or person, which could possibly be moving, the BLE Mini was far too slow for our purposes.

4.1.5 Intel Edison

The team later found that the Intel Edison has the ability to obtain RSSI readings from BLE devices such as the Estimotes. This section details the steps for doing that.

4.1.5.1 Enable Bluetooth on Edison

Use PuTTY to start a serial connection with an Intel Edison. After logging into the Edison, enter the following commands:

- rfkill unblock bluetooth
- bluetoothctl

[3] BLE Mini product page: http://redbearlab.com/blemini/

    - Edison will enter “Bluetooth control” mode
- agent KeyboardDisplay
- default-agent
- scan on
    - Check that there is some output and no error.
- scan off
- quit
    - Edison will exit out of bluetoothctl

4.1.5.2 RSSI via Command Line

Getting RSSI readings via the Edison is accomplished using the “hcitool” and “hcidump” command line tools. Running these two commands at the same time allows us to continuously see information about BLE packets that are being received by the Edison. The command to do this is as follows:

```
hciconfig hci0 reset && hcitool lescan --duplicates >/dev/null & \
hcidump -a | grep -B 0 -A 3 -E 'FA:|ED:|D2:'
```

After entering this command, information on the three beacons should be printed to the terminal. There should be a new reading about once every second, for each beacon, until the processes are terminated. (See the end of this subsection for directions on quitting the processes.)

The breakdown of this command is as follows:

- The hciconfig part resets the Bluetooth receiver, hci0.
- The && means that the second command will only run if the first is successful
- The hcitool part starts a scan for BLE signals.
    - The --duplicates flag allows the scan to display multiple signals received from the same transmitter.
    - >/dev/null means to hide the output of the hcitool scan.
- The & sends the hcitool process to the background so that we can start a new command.
- The hcidump part displays the contents of the received BLE packets.
- | grep starts a command that will filter the output of the hcidump.
    - ‘-E FA:|ED:|D2:’ means to look for lines containing “FA:”, “ED:”, or “D2”. **Note:** These are the beginning parts of the Bluetooth addresses of the three Estimotes available in lab; thus, this part would need to be edited should the Estimotes change.
    - -B 0 -A 3 means that, if a line is found that contains “FA:”, “ED:”, or “D2”, show 0 lines before that line and 3 lines after it.

To quit the processes:

- Ctrl+C (Quits the hcidump process)
- fg <enter> (Moves the hcitool process from the background to the foreground)
- Ctrl+C (Quits the hcitool process)

4.1.5.3 RSSI via Python Script

Any programming language that can manipulate serial COM ports can be used to communicate with the Edison command line and receive data from it. The team chose to use Python for this purpose because of the relative speed with which simple programs can be developed.

The Python script that was developed utilizes the commands discussed in the previous subsection. The script should be saved and run on the computer that the Edison is plugged into (the "host computer"). This host computer should have Python installed[4], as well as a Python library called pySerial[5], which contains the "serial" module.

Before running the script, ensure that the following steps are completed:

1. Connect the computer to the Edison and open a serial connection to the Edison, using PuTTY.
2. Log into the Edison and enable Bluetooth as described in the "Enable Bluetooth on Edison" section above.
3. Save the Python script (shown below) to the computer that the Edison is plugged into (*not* onto the Edison itself).
4. In the script, change the COM_PORT_NUMBER variable to match the number of the COM port to which the Edison is connected. On Windows, this number can be found by using the Device Manager.
5. Close PuTTY. This step is required before running the Python script because only one program can use a single COM port at one time, and both PuTTY and the script require the use of the COM port.

If the script is saved with the file name rssi.py, then it can be run on the host computer via the command line with the command `python rssi.py`. The rest of this document will refer to the script as rssi.py. The code for rssi.py is listed below:

```
# rssi.py
# Python script that communicates with Edison over a serial port and prints
# on the host PC the RSSI readings of three specific Estimote beacons.

import re
import serial

# @USER: Change this to match the COM port that the Edison is on.
# Make sure no other program (including PuTTY) is using this port.
COM_PORT_NUMBER = 7

print 'Starting rssi.py ...'

# Open serial port
ser = serial.Serial()
ser.baudrate = 115200
ser.port = COM_PORT_NUMBER-1
ser.open()

print 'Serial communication opened on %s' % ser.name

# Send the command that will start the Edison getting RSSI readings
ser.write("""hciconfig hci0 reset && hcitool lescan --duplicates >/dev/null & hcidump -a | grep -B 0 -A 3 -E
'FA:|ED:|D2:'\n""")
```

[4] Python download page: https://www.python.org/downloads/
[5] pySerial documentation: http://pyserial.sourceforge.net/index.html

```
EMPTY_PAYLOAD = {'addr': '', 'rssi': ''}
payload = EMPTY_PAYLOAD

while True:
  try:
    # Read each line from the serial port
    inline = ser.readline()

    if not inline:
      print 'No Bluetooth data available.'
      break

    if 'addr' in inline:
      # Parse out the Bluetooth address from the line
      found_addr = re.findall(r"\b((?:\w\w:){5}\w\w)\b", inline)
      # re.findall gets back a list
      if len(found_addr) != 1:
        # If address NOT found, reset payload and start over
        payload = EMPTY_PAYLOAD
        continue
      else:
        # If address found, put it in the payload
        payload['addr'] = found_addr[0]

    elif 'RSSI' in inline:
      # Parse out the rssi from the line
      found_rssi = re.findall(r"(?<=RSSI: )-*\d\d", inline)
      # re.findall gets back a list
      if len(found_rssi) != 1:
        # If rssi NOT found, reset payload and start over
        payload = EMPTY_PAYLOAD
        continue
      else:
        # If rssi found, put it in the payload
        payload['rssi'] = get_dist(found_rssi[0])
        # Send the information only if both addr and rssi are there
        if payload['addr'] != '':
          print str(payload)
          payload = EMPTY_PAYLOAD # Reset payload
  except (KeyboardInterrupt, SystemExit):
    break
  except Exception as ex:
    print ex
    break

# Cancel the processes
ser.write('\x03') #ctrl-c
ser.write('fg\n')
ser.write('\x03')
ser.close()
```

```
print "Bluetooth scan stopped."
```

This script will print a JSON string with the following format:

{'rssi': '<RSSI in dB>', 'addr': '<MAC address/ID of Estimote>'}

For example: {'rssi': '-70', 'addr': 'FA:56:D1:5E:1A:11'}.

4.1.5.4 Sending Data over Wi-Fi

A client/server connection can be used between the host computer running rssi.py and another computer to communicate RSSI readings between the two machines. If such functionality is desired, add the following lines to rssi.py:

```
# @USER: Put IP address of host computer here
hostIP = "192.168.1.220"
hostPort = 9999

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.connect((hostIP, hostPort))

def sendMessage(msg):
  sock.sendto(msg, (hostIP, hostPort))


# ...

# @USER: Place this line at the very end of the script
sock.close() # Close the socket
```

Then, replace the line in rssi.py that says `print str(payload)` with `sendMessage(str(payload))` so that the information is sent over Wi-Fi instead of simply printed to the console.

On the receiving computer, the following script can be run to display the data that is sent:

```
import socket

# @USER: Put host (transmitting) computer's IP address here
UDP_IP = "192.168.1.220"
UDP_PORT = 9999

sock = socket.socket(socket.AF_INET, # Internet
          socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))

print 'Listening on %s : %d ...' % (UDP_IP, UDP_PORT)

while True:
  data, addr = sock.recvfrom(1024) # buffer size is 1024 bytes
  print "received message:", data
```

Note that the variable hostIP and the variable UDP_IP in the above two snippets have the same value. Similarly, hostPort and UDP_PORT also have the same value. Note also that the two computers should be on the same Wi-Fi network in order for the communication to work.

4.1.6 Data Collection

The team used the Edison to receive iBeacon packets and extract the RSSI readings of each of the three Estimotes in lab at various distances. The data collected from this process would be used to formulate an equation to calculate the distance based on the received signal strength indication (RSSI).

4.1.6.1 Process

1. First, we set up a metric scale (using a tape measure) that was 5 meters long to show the distance between the Estimotes and the Edison.
2. Second, we put the three iBeacons, which have different colors in blue, green, and purple, at different distance points on the metric scale. For example, the blue iBeacon would be placed at 1.0 meter, the green at 3.0 meters, and the purple at 4.5 meters.
3. Third, we ran code on the computer (similar to the code for rssi.py, discussed in the previous section) to receive data packets from the three iBeacons. For each beacon and at each distance, we collected 200 data points. The data collection took only around 2 minutes for each run, which is very fast.
4. We re-named the three files after collecting data each time, according to the color of the iBeacons and the distances. For example, with the configurations listed in section 2, the file names would become blue_100.txt, green_300.txt, purple_450.txt.
5. We moved the three iBeacons to different locations, at intervals of 50 cm, and then did the above process again.
6. We repeated the steps until we had data of each iBeacon for each 50-cm point, from 0 to 5 meters.

4.1.6.2 Data Analysis

Using the data collected in the steps above, we obtained the tables shown below.

color	blue									
distance (cm)	50	100	150	200	250	300	350	400	450	500
average	-57.44	-63.59	-70.66	-73.22	-72.18	-75.97	-80.34	-75.90	-73.41	-80.76
median	-57	-63	-69	-73	-72	-75.5	-80	-76	-73	-80

color	green									
distance (cm)	50	100	150	200	250	300	350	400	450	500
average	-57.08	-64.11	-70.20	-74.38	-74.19	-75.76	-79.15	-74.34	-74.58	-78.70
median	-57	-64	-69	-73	-75	-77	-79	-74	-75	-78

color	purple									
distance (cm)	50	100	150	200	250	300	350	400	450	500
average	-57.39	-62.74	-68.92	-73.80	-72.02	-75.37	-77.44	-73.15	-71.92	-78.73
median	-58	-62.5	-68	-73	-72	-75	-77	-72	-72	-78

4.1.6.3 Results

The data displayed in the tables show that the RSSI values are ***not*** monotonically decreasing, as we had expected and hoped them to be. This fact hindered the team's efforts to find a "best-fit line" for converting RSSI to distance, and it casts some doubt on the feasibility of using RSSI to determine distance.

Using the collected data and an RSSI-to-distance equation found online[6], the team's best guess for calculating distance from RSSI is as follows:

$$\text{Distance (in meters)} = A * \left(\frac{RSSI}{TX\ Power}\right)^{B} + C$$

where **A = 0.75, B = 6.0, and C = (1 – A) = 0.25**. These numbers can of course be further refined and calibrated. Recall that *TX Power* is the RSSI measured when the receiver is 1 meter away from the beacon. In the equation above, ***TX Power* = -63**.

C is set to be (1 – A) because when the distance is 1 meter, then *RSSI* = *TX Power*, so *RSSI* / (*TX Power*) = 1. Thus, the right-hand side of the above equation would become *A* + *C* = *A* + (1 – *A*) = 1.

The team also found that the RSSI readings obtained from the Estimote iBeacons fluctuated significantly, up to +/- 5 or even 10 dB. While we are not certain about the reason for this fluctuation, we hypothesize that it may have something to do with interference from other signals around the room or from objects near the beacons and receiver. It is also possible that, even in ideal conditions, Bluetooth Low Energy technology is simply not reliable enough to produce steady readings.

Thus, when this data is sent to the Computer Science team, the readings should be averaged over the most recent 5 or 10 readings to obtain a more stable value.

[6] "Understanding iBeacon distancing":
http://stackoverflow.com/questions/20416218/understanding-ibeacon-distancing

4.1.7 Future Work

Based on the work done this semester, it is uncertain whether the team should continue to pursue BLE and iBeacons as an approach for localization. In fact, certain papers even claim that RSSI is not a reliable measurement for localization applications[7].

However, Estimote does offer a feature called "Indoor Location SDK," which the company claims can be used to determine the location of a person in a room[8]. Four Estimotes are required to use this SDK, and the lab currently only has three, so more beacons would need to be purchased if this project is to continue developing in that direction. This SDK could possibly be leveraged for use in this project if the team can discover a way to export data from the SDK to the computers and other devices used in the R2-I2 system.

Different equations for calculating distance could also be tested, such as the one below[9]:

$$\text{Distance} = 10^{A-\text{RSSI}/K}$$

where A and K are constants that can be calibrated.

4.2 Extended Kalman Filter

The problem of navigation can be summarized into answering three questions: "where am I?", "where am I going?" and "how should I get there?". Localization answers the first question "where am I?", finding a reliable solution to this problem paves essential way for solving the remaining two questions.

Localization is one of the fundamental problems in mobile robot navigation and motion planning. In an indoor environment with a flat floor plan, localization is identified as a problem of estimating the pose, i.e. position and orientation of a mobile robot, when the map of the environment, sensor readings, and executed actions of the robot are provided. Odometry is one of the most popular solutions for wheeled mobile robots. The encoders mounted on the wheels provide robot motion information to update the mobile robot pose. However, odometric pose estimation unboundedly accumulates errors due to different wheel diameters, wheel-slippage, wheel misalignment, and finite encoder resolution. Experiment results presented in this report, concur that the largest factor in odometry error is due to the rotation of the robot.

The odometry errors can be eliminated using an auxiliary sensor to observe landmarks in an environment. Different sensors, such as cameras, sonar sensors, and laser range finders have been used to detect landmarks and obtain the required measurements. The robot itself can only measure odometry through onboard sensors which can include noise and error. Using a laser scanner, camera,

[7] "Is RSSI a Reliable Parameter in Sensor Localization Algorithms": http://www.cse.buffalo.edu/srds2009/F2DA/f2da09_RSSI_Parameswaran.pdf
[8] Estimote Indoor Location: http://estimote.com/indoor/
[9] "Is RSSI a Reliable Parameter in Sensor Localization Algorithms": http://www.cse.buffalo.edu/srds2009/F2DA/f2da09_RSSI_Parameswaran.pdf

and other sensors, the environment can be sensed to help give a correct position of the robot. This is done through landmark extraction and reevaluating the location of these landmarks after the robot has moved.

In most robotic applications, determining what to do is relatively easy if one only knew certain quantities. For example, moving a mobile robot is relatively easy if the exact location of the robot and all nearby obstacles are known. Unfortunately, these variables are not directly measurable. Instead, a robot has to rely on its sensors to gather this information. Sensors carry only partial information about those quantities, and their measurements are corrupted by noise. State estimation seeks to recover state variables from the data. Kalman Filter, as well as extended Kalman Filter is a tool to recover the state variables from the data.

In this report, we propose an accurate mobile robot localization method using odometry and a Kinect sensor. The odometry and the Kinect sensor measurements are fused using the extended Kalman filter (EKF) to provide more accurate localization results. The correct detection of landmarks using the Kinect sensor has significantly contributed to the better performance of the robot localization. The experiments are carried out with R2D2 mobile robot and the results are provided in order to interpret the accuracy of the proposed method. The basic process is given below:

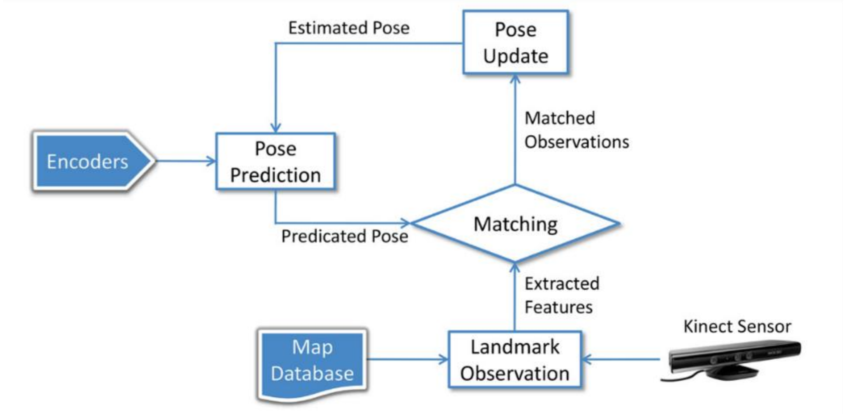


Figure 1. Overview of EKF localization

When the odometry changes because of the robot moving, the uncertainty pertaining to the new position is updated in the EKF using odometry update. Fixed-point landmarks are extracted from the environment and are associated with observations of the same landmarks at the previous positions. These re-observed landmarks are now used to update a better estimation of both the robots position and the landmarks location in the EKF. If a new landmark is found, it is added to the EKF for later landmark extraction and comparison.

4.2.1 Extended Kalman Filter

The Kalman filter is used because of its ability to incorporate all the measurements and provide a least squares estimate of the process state. On the basic level the localization problem is the signal plus noise problem; the signal (true location of robot) and the noise (measurement error). The Kalman Filter will be used for estimating under the assumption that all sensor errors (noise) has a zero mean process and is Gaussian. This assumption can prove to be problematic for some accounts, but in most cases, sensor noise will in fact be white. Another assumption made by the KF that can pose a problem is that the process model is linear. This limits the use of KF from most complex problems, as they are usually highly nonlinear. Using an Extended Kalman Filter helps to tackle some more complex problems, but if the process deviates from the set point at which it was linearized, it will also become problematic.

4.2.1.1 State

Environments are characterized by state. State can be regarded as the collection of all aspects of the robot and its environment that can impact the future. State may change over time, such as the movement of robot, is called dynamic state. State can keep same, called static state. Typical state variables are:

- The **robot pose**, which comprises its localization and orientation. For our R2D2 robot, the pose is given by three variables, its two location coordinates in the plane and its heading direction.
- The configuration of the robot's **actuators**
- The location and features of surrounding objects in the environment. Such as **landmarks**.

4.2.1.2 Environment Interaction

There are two fundamental types of interactions between a robot and its environment: The robot can influence the state of its environment through its actuators. And it can gather information about the state through its sensors.

- **Sensor measurements**. Perception is the process by which the robot uses its sensors to obtain information about the state of its environment. The result of such a perceptual interaction will be called a measurement
  The **measurement data** will be denoted as z_t.
- **Control actions** change the state of the world. They do so by actively asserting forces on the robot's environment. Even if the robot does not perform any action itself, state usually changes. Thus, for consistency, we will assume that the robot *always* executes a control action, even if it chooses not to move any of its motors.
  Control data get its source from *odometers*. Odometers are sensors that measure the revolution of a robot's wheels. As such they convey information about the change of the state. Control data will be denoted as u_t.

4.2.1.3 Formation of Extended Kalman Filter

EKF can be described by the following method:

- Update the current state estimate using the odometry data
- Update the estimate state from re-observing the landmarks

- Add new landmarks to the current state

Kalman Filter is based on very strong assumptions: it has to be linear state dynamics and observations have to be linear in state. Kalman Filter has the following equations:

$$x_t = Ax_{t-1} + Bu_{t-1} + w_{t-1}$$

$$z_t = Hx_t + v_t$$

if the system is not linear, then there will be non-linear state dynamics and non-linear observations.

$$x_t = f(x_{t-1}, u_{t-1}, w_{t-1})$$

$$z_t = h(x_t, v_t)$$

So we need to linearize the non-linear equation above as follows:

$$x_t \approx \widetilde{x_t} + A(x_{t-1} - \hat{x}_{t-1}) + Ww_{t-1}$$

$$z_t \approx \widetilde{z_t} + H(x_t - \tilde{x}_t) + Vv_t$$

where

$$\widetilde{x_t} = f(x_{t-1}, u_{t-1}, 0)$$

$$\widetilde{z_t} = (\widetilde{x_t}, 0)$$

where A, H, W, V are Jacobians defined by

$$\mathrm{A}(x_t)=\begin{pmatrix} \frac{\partial f_1(x_t,u_t,0)}{\partial x_1} & \cdots & \frac{\partial f_1(x_t,u_t,0)}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(x_t,u_t,0)}{\partial x_n} & \cdots & \frac{\partial f_n(x_t,u_t,0)}{\partial x_n} \end{pmatrix}$$

$$\mathrm{W}(x_t)=\begin{pmatrix} \frac{\partial f_1(x_t,u_t,0)}{\partial w_1} & \cdots & \frac{\partial f_1(x_t,u_t,0)}{\partial w_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(x_t,u_t,0)}{\partial w_n} & \cdots & \frac{\partial f_n(x_t,u_t,0)}{\partial w_n} \end{pmatrix}$$

$$\mathrm{V}(x_t)=\begin{pmatrix} \frac{\partial h_1(x_t,0)}{\partial v_1} & \cdots & \frac{\partial h_1(x_t,0)}{\partial v_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_n(x_t,0)}{\partial v_n} & \cdots & \frac{\partial h_n(x_t,0)}{\partial v_n} \end{pmatrix}$$

$$\mathrm{H}(x_t)=\begin{pmatrix} \frac{\partial h_1(x_t,0)}{\partial x_1} & \cdots & \frac{\partial h_1(x_t,0)}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_n(x_t,0)}{\partial x_n} & \cdots & \frac{\partial h_n(x_t,0)}{\partial x_n} \end{pmatrix}$$

4.2.1.3.1 System State: X

The system state, X, contains the position of the robot (x, y, and θ) and the position of each of the n landmark (x,y). This gives a state equation that takes on the following form

$$x = \begin{Bmatrix} x_r \\ y_r \\ \theta_r \\ x_1 \\ y_1 \\ :: \\ x_n \\ y_n \end{Bmatrix}$$

4.2.1.3.2 Covariance Matrix: P

The covariance matrix P is where the covariance on the robot position, the covariance of each of the landmarks, the covariance between robot position and landmarks and the covariance between the landmarks is contained. The matrix is broken up into several main subsections and takes the following form:

$$\begin{bmatrix} [A] & [E] & \cdots & \cdots \\ [D] & [B] & \cdots & [G] \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & [F] & \cdots & [C] \end{bmatrix}$$

The A cell contains the covariance on the robot position and is a 3 x3 matrix (x, y, and θ). B contains the covariance for the first landmark and is a 2x2 matrix (x and y). The covariance on the landmarks continues down the diagonal to C which is the covariance for the last stored landmark. Cell D contains the covariance between the robot state and the first landmark while cell E contains the covariance between the first landmark and the robot state, with E being the transpose of D. F contains the covariance between the last landmark and the first landmark, while G is the covariance between the first land mark and the last landmark, with G being the transpose of F. The rest of the covariance matrix is just a combination of covariance between the robot state and landmarks and the covariance between each landmark. When the robot is traveling without seeing any landmarks, the P matrix only contains the A matrix and is simply a 3x3, but as each landmark is identified, the P matrix grows by 2 in each dimension.

4.2.1.3.3 Kalman Gain: K

The Kalman gain, K, is computed by the way of:

$$K_k = P_k^- H_k^{tr} \left(H_k P_k^- H_k^{tr} + D_k D_k^{tr} \right)^{-1}$$

Which is used to find out how much the current estimation of the landmarks is trusted. This gain is what will help the robot know its true position using the landmarks as a reference. If the devise used to measure the landmarks is sub optimal, there will be more confidence in the odometry data therefore the Kalman gain will be low i.e. won't correct the current predication of the location very much away from the odometry data. On the contrary if the device used to measure landmarks is very good compared to the odometry performance, the gain will be high i.e. correcting the current prediction of the location a significant amount away from the odometry data.

4.2.1.3.4 The Jacobian of the Measurement Model: H

The measurement model defines how to compute an expected range and bearing from the measurements and is done with the following formula:

$$\begin{bmatrix} \text{range} \\ \text{bearing} \end{bmatrix} = \begin{bmatrix} \sqrt{(\lambda_x - x)^2 + \left(\lambda_y - y\right)^2} + v_r \\ \tan^{-1}\left(\dfrac{\lambda_y - y}{\lambda_x - x}\right) - \theta + v_\theta \end{bmatrix}$$

Where λ_x and λ_y are the x and y location of the landmark and x, y, and θ are the current estimate of the robot position and rotation. This gives a Jacobian, H, with respect to x, y, and θ as:

$$\begin{bmatrix} \dfrac{x - \lambda_x}{r} & \dfrac{y - \lambda_y}{r} & 0 \\ \dfrac{\lambda_y - y}{r^2} & \dfrac{\lambda_x - x}{r^2} & -1 \end{bmatrix}$$

H shows how much the range and bearing will change as x, y, and θ change. This is the formulation of the usual H for regular EKF state estimation, but it is also necessary to add in additional values to account for the landmarks. As an example, for landmark number two the H matrix there for takes the form of:

$$\begin{bmatrix} A & B & C & 0 & 0 & -A & -B & 0 & 0 \\ D & E & F & 0 & 0 & -D & -E & 0 & 0 \end{bmatrix}$$

Where the first three columns are for the x, y, and θ of the robot and the next six columns would be for each pair of x and y landmark coordinates. The first three columns are the same as for regular EK state estimation, and each new landmark just adds two new columns.

4.2.1.3.5 Jacobian of the Prediction Model: F

Just like Jacobian of the measurement process was related to the measurement process, the Jacobian of the prediction model is closely related to the prediction model. The prediction model which gives the relationship between previous position, control input, and movement is shown below.

$$f = \begin{bmatrix} x + \Delta t\cos\theta + q\Delta t\cos\theta \\ y + \Delta t\sin\theta + q\Delta t\sin\theta \\ \theta + \Delta\theta + q\Delta\theta \end{bmatrix}$$

Where again x and y are the position of the robot and θ is the rotation of the robot. Δt is the change in thrust and q is the error term. The linearized version of the Jacobian is given as follows:

$$\begin{bmatrix} 1 & 0 & -\Delta t\sin\theta \\ 0 & 1 & \Delta t\cos\theta \\ 0 & 0 & 1 \end{bmatrix}$$

4.2.1.3.6 Process and Measurement Noise: Q , W, and V

The process is assumed to have a Gaussian noise proportional to the controls, $\Delta x, \Delta y$ and Δt.

$$W = [\Delta t\cos\theta \quad \Delta t\sin\theta \quad \Delta\theta]^{tr}$$

And the Q matrix given as:

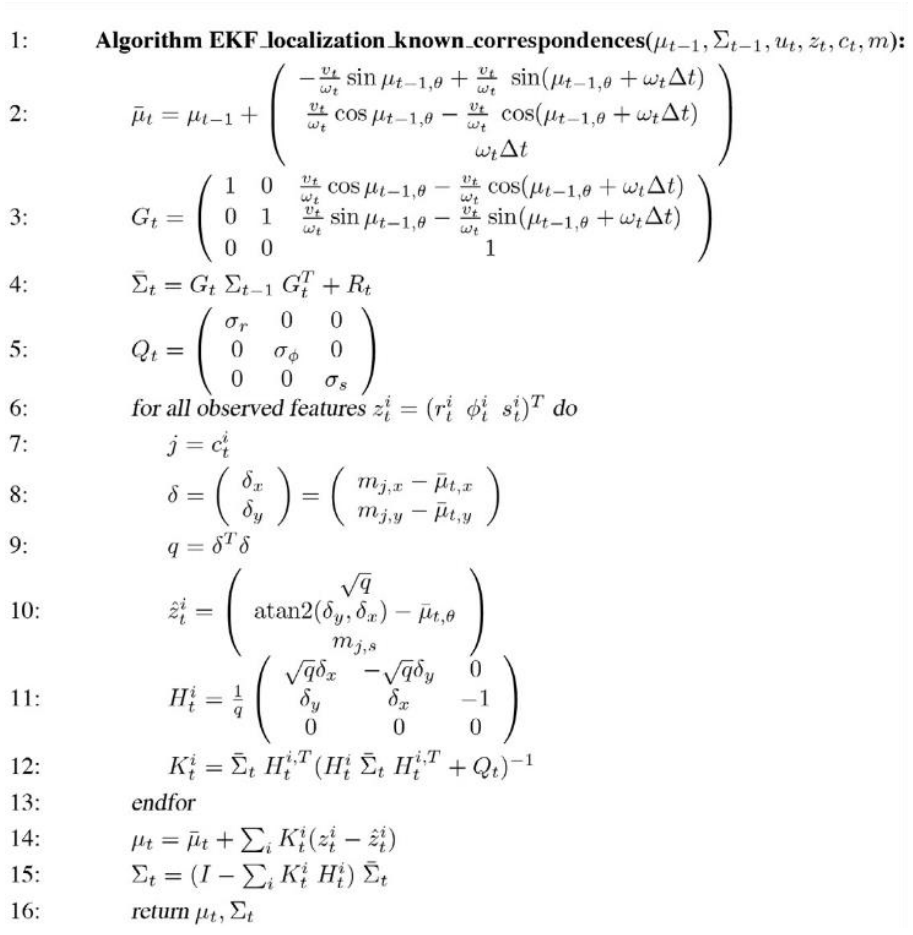
4.2.2 $\mathbf{Q = WW^{tr}}$

The range measurement device is also assumed to have Gaussian noise proportional to the range and bearing.

4.2.2.1 EKF information

This algorithm is derived from the EKF mentioned above. It requires as its input a Gaussian estimate of the robot pose at time t − 1, with mean μt−1 and covariance Σt−1. Further, it requires a control ut, a map m, and a set of features zt = {zt1, zt2, . . .} measured at time t, along with the correspondence variables ct = {c1t , c2t , . . .}. It output is a new, revised estimate μt, Σt.

The algorithm for coding EKF is listed as follows:



```
1:  Algorithm EKF_localization_known_correspondences($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, c_t, m$):
2:      $\bar{\mu}_t = \mu_{t-1} + \begin{pmatrix} -\frac{v_t}{\omega_t} \sin \mu_{t-1,\theta} + \frac{v_t}{\omega_t} \sin(\mu_{t-1,\theta} + \omega_t \Delta t) \\ \frac{v_t}{\omega_t} \cos \mu_{t-1,\theta} - \frac{v_t}{\omega_t} \cos(\mu_{t-1,\theta} + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix}$
3:      $G_t = \begin{pmatrix} 1 & 0 & \frac{v_t}{\omega_t} \cos \mu_{t-1,\theta} - \frac{v_t}{\omega_t} \cos(\mu_{t-1,\theta} + \omega_t \Delta t) \\ 0 & 1 & \frac{v_t}{\omega_t} \sin \mu_{t-1,\theta} - \frac{v_t}{\omega_t} \sin(\mu_{t-1,\theta} + \omega_t \Delta t) \\ 0 & 0 & 1 \end{pmatrix}$
4:      $\bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$
5:      $Q_t = \begin{pmatrix} \sigma_r & 0 & 0 \\ 0 & \sigma_\phi & 0 \\ 0 & 0 & \sigma_s \end{pmatrix}$
6:      for all observed features $z_t^i = (r_t^i \;\; \phi_t^i \;\; s_t^i)^T$ do
7:          $j = c_t^i$
8:          $\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} m_{j,x} - \bar{\mu}_{t,x} \\ m_{j,y} - \bar{\mu}_{t,y} \end{pmatrix}$
9:          $q = \delta^T \delta$
10:         $\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \text{atan2}(\delta_y, \delta_x) - \bar{\mu}_{t,\theta} \\ m_{j,s} \end{pmatrix}$
11:         $H_t^i = \frac{1}{q} \begin{pmatrix} \sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & \delta_x & -1 \\ 0 & 0 & 0 \end{pmatrix}$
12:         $K_t^i = \bar{\Sigma}_t \, H_t^{i,T} (H_t^i \, \bar{\Sigma}_t \, H_t^{i,T} + Q_t)^{-1}$
13:     endfor
14:     $\mu_t = \bar{\mu}_t + \sum_i K_t^i (z_t^i - \hat{z}_t^i)$
15:     $\Sigma_t = (I - \sum_i K_t^i \, H_t^i) \, \bar{\Sigma}_t$
16:     return $\mu_t, \Sigma_t$
```

Figure 2. EKF Algorithm for Localization

Lines 2 through 4 implement the familiar motion update, using a linearized motion model. The predicted pose after motion is calculated as μ̄t in Line 2, and Line 4 computes the corresponding uncertainty ellipse. Lines 5 to 15 implement the measurement update. The core of this update is a loop through all possible features i observed at time t. In Line 7, the algorithm assigns to j the correspondence of the i-th feature in the measurement vector. It then calculates a predicted measurement zˆt(i) and the Jacobian Ht(i) of the measurement model. The Kalman gain Kt(i) is then calculated in Line 12 for each observed feature. The sum of all updates is then applied to obtain the new pose estimate, as stated in Lines 14 and 15. Notice that the last row of Ht(i) is all zero. This is because the signature does not depend on the robot pose. The effect of this degeneracy is that the observed signature sit has no effect on the result of the EKF update. This should come at no surprise: knowledge of the correct correspondence zt(i) renders the observed signature entirely uninformative.

Other notation:

- ωt is a rotational velocity
- Δt is the time window over which the robot motion is executed

- Mean μt−1
- Covariance Σt−1
- Jacobian Gt
- Covariance Rt is a function of the velocities vt and ωt and the mean μt−1

4.2.3 System Models

4.2.3.1 Driving System:

Wheel Encoders:

Wheels and other driving mechanics are controlled by motors. These motors can make wheels move either forward or backward. By means of wheel encoder sensors we can monitor the number of wheel rotations of a specific wheel. We can use the wheel encoder readings from these sensors together with knowledge about the diameter of the wheel to estimate its displacement. Wheel encoder sensors provide relative position measurements.

In general we are not interested in the relative displacement of each wheel, but in the displacement of the robot as a result of the displacements of all wheels in the drive system together. Depending on the configuration of the guidance system, the conversion between output from the wheel encoders from the different wheels and the relative displacement of the robot is different.

4.2.3.1.1 The perfect system model

In order to model the way in which a drive system changes the location of a robot, we first take a look at the ideal situation in which there is no noise in the environment. Once we have a noise-free system model, we extend the model to take into account noises that influence the system.

In our case, the system model that we are looking for describes how the location of the robot changes due to the dynamics of the guidance system. Since we can monitor the relative dynamics of the guidance system with wheel encoders, we are looking for a system model that given the last location and a relative displacement determines the new location. In order to derive this model, let us formalize a location and a relative displacement more precisely.

Location:

Equation below shows schematically how we define the location of a robot. The location of a robot is defined in a global coordinate system and consists of the x and y coordinates of the center of the robot, the center point, and the orientation ϕ of the robot in that general coordinate system at a certain time k,

$$x_k = \begin{bmatrix} x_{[x],k} \\ x_{[y],k} \\ x_{[\phi],k} \end{bmatrix}$$

Relative Displacement.

The relative displacement is the displacement of the center point of the robot over a certain time. We assume that we can convert the separate displacement from the wheels of the robot into a relative displacement measurement u_k of the center point of the robot using some function that takes as argument raw data d_k from the wheel encoders. We then define the relative displacement as:

$$u_k = j(d_k) = \begin{bmatrix} u_{[\Delta D],k} \\ u_{[\Delta\phi],k} \end{bmatrix}$$

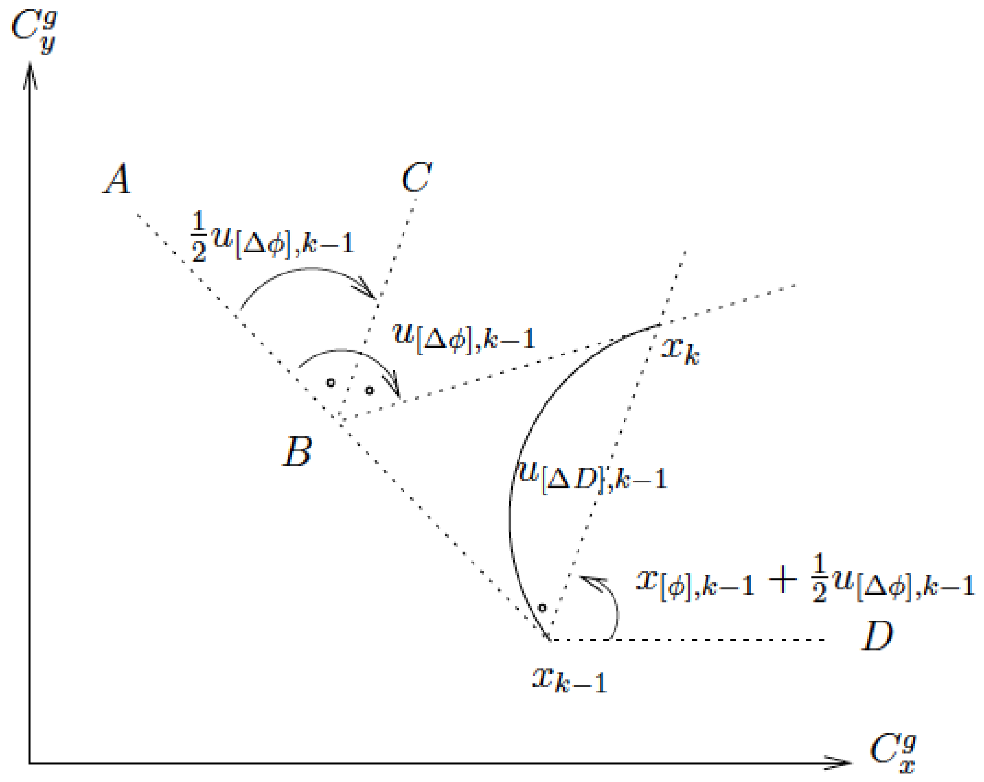
where $u_{[\Delta D],k}$ is the distance over which the center point of the robot travels, and $u_{[\Delta\phi],k}$ is the change in orientation during the travelling.

Location Update

Given the relative information u_{k-1}, and given the last location of the robot x_{k-1}, we want to express the current location of the robot. That is, we look for the function f,

$$x_k = f(x_{k-1}, u_{k-1}) = \begin{bmatrix} f_x(x_{k-1}, u_{k-1}) \\ f_y(x_{k-1}, u_{k-1}) \\ f_\phi(x_{k-1}, u_{k-1}) \end{bmatrix}$$

Coordinates:



C_y^g
A
C
$\frac{1}{2}u_{[\Delta\phi],k-1}$
$u_{[\Delta\phi],k-1}$
x_k
B
$u_{[\Delta D],k-1}$
$x_{[\phi],k-1} + \frac{1}{2}u_{[\Delta\phi],k-1}$
D
x_{k-1}
C_x^g

We derive expression for f_x and f_y by means of Figure above. The figure shows the path that the robot drives from x_{k-1} to x_k. The robot starts at x_{k-1} facing direction A; thus $x_{[\phi],k-1} = \angle A x_{k-1} D$. The robot then drives a path of length $u_{[\Delta D],k-1}$ to x_k, whereby the orientation changes $u_{[\Delta\phi],k-1} = \angle ABx_k$. It can be shown that $\angle Ax_{k-1}x_k = (1/2)u_{[\Delta\phi],k-1}$, which gives us:

$$x_k x_{k-1} D = \angle Ax_{k-1}D - \angle Ax_{k-1}x_k = x_{[\phi],k-1} + (1/2)u_{[\Delta\phi],k-1}$$

Besides this angle, we need to know the length of $x_{k-1}x_k$ in order to determine the coordinates $x_{[x],k}$ and $x_{[y],k}$. In general this length is not known, though a common way to deal with this is to use $u_{[\Delta D],k-1}$ as an approximation. We then calculate the coordinates of x_k as:

$$x_{[x],k} = f_x(x_{k-1}, u_{k-1}) = x_{[x],k-1} + u_{[\Delta D],k-1} \cdot cos(x_{[\phi],k-1} + (1/2)u_{[\Delta\phi],k-1})$$

$$x_{[y],k} = f_y(x_{k-1}, u_{k-1}) = x_{[y],k-1} + u_{[\Delta D],k-1} \cdot sin(x_{[\phi],k-1} + (1/2)u_{[\Delta\phi],k-1})$$

Orientation.

The expression for f_ϕ is trivially found. If the relative displacement information indicates that the orientation of the robot changes by $u_{[\Delta\phi]}$, then the updated orientation of the robot is the sum of this and the last orientation of the robot. Thus,

$$x_{[\phi],k} = f_\phi(x_{k-1}, u_{k-1}) = x_{[\phi],k-1} + u_{[\Delta\phi],k-1}$$

4.2.3.1.2 The Noisy System Model:

In the derivation of the system model so far we assumed that the driving of the robot was not distributed by any noise sources. We assumed that the wheel encoders worked perfectly; we assumed that we could perfectly map wheel encoder countings to traveled wheel distance; we assumed that there were no influences from the environment that disturbed the driving, and that thus our approximated system function perfectly describes the movement of the robot over time. In the real world most of these assumptions do not hold, and thus we will introduce noise in the system model.

Noise Modeling:

Relative Displacement Noise:

Assume that we can model the noise in the relative displacement by a random noise vector q_k being zero-mean independent Gaussian distributed,

$$q_k \sim N(\widehat{q_k}, U_k),$$

where

$$\widehat{q_k} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$U_k = E[(q_k - \widehat{q_k})(q_k - \widehat{q_k})^T]$$

$$= \begin{bmatrix} \sigma^2_{q[\Delta D],k} & \sigma_{q[\Delta\phi],k}\sigma_{[\Delta D],k} \\ \sigma_{q[\Delta\phi],k}\sigma_{[\Delta D],k} & \sigma^2_{q[\Delta\phi],k} \end{bmatrix}$$

In the covariance matrix U_k, the off-diagonal elements are zero, since we assume that the noise sources are independent of each other. We adjust the noise-free relative displacement by including the term q_k to obtain the noisy relative displacement,

$$u_k = j(d_k) + q_k = \begin{bmatrix} u_{[\Delta D],k} \\ u_{[\Delta\phi],k} \end{bmatrix} + \begin{bmatrix} q_{[\Delta D],k} \\ q_{[\Delta\phi],k} \end{bmatrix}$$

Since we include the random noise term q_k in the model, the relative information vector u_k also becomes a random vector. If we assume that j(_) is deterministic, the uncertainty in u_k equals the uncertainty in the noise term q_k.

System Noise.

We can model noise sources that are not directly related to the relative displacement with a system noise term. Assume that we can model these noise sources with a term w_k, that is independent Gaussian distributed,

$$w_k \sim N(\widehat{w_k}, Q_k),$$

where

$$\widehat{w_k} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$Q_k = E[(w_k - \widehat{w_k})(w_k - \widehat{w_k})^T]$$

$$= \begin{bmatrix} \sigma^2_{w[x],k} & \sigma_{w[y],k}\sigma_{w[x],k} & \sigma_{\omega[\phi],k}\sigma_{w[x],k} \\ \sigma_{w[x],k}\sigma_{w[y],k} & \sigma^2_{w[y],k} & \sigma_{\omega[\phi],k}\sigma_{w[y],k} \\ \sigma_{w[x],k}\sigma_{\omega[\phi],k} & \sigma_{w[y],k}\sigma_{\omega[\phi],k} & \sigma^2_{w[\phi],k} \end{bmatrix}$$

we assume that the noise sources in the location elements are independent of each other, and thus the off-diagonal elements of the covariance matrix Q_k are zero. The diagonal elements of the covariance matrix contain the variances of the noise vector w_k. Including this noise term in the noise-free system model from

$$x_k = f(x_{k-1}, u_{k-1}) + w_{k-1}$$

$$= \begin{bmatrix} f_x(x_{k-1}, u_{k-1}) \\ f_y(x_{k-1}, u_{k-1}) \\ f_\phi(x_{k-1}, u_{k-1}) \end{bmatrix} + \begin{bmatrix} w_{[x],k-1} \\ w_{[y],k-1} \\ w_{[\phi],k-1} \end{bmatrix}$$

Instead of a deterministic location x_k, the location is now a random vector. The variance of the location grows with every time step, since the location x_k at step k depends on the location x_{k-1} one step earlier and since at every time step the system noise increases the variance. Besides this, the relative displacement vector u_{k-1} is also a random vector with influence on the uncertainty in x_k.

4.2.4 Extended Kalman Localization

In this part, we give combine the theory mentioned above to put it into practice to show how to implement EKF to R2D2 localization.

4.2.4.1 Predictive position tracking:

Prediction equations:

Since the system model that we obtained in nonlinear in the orientation of the robot, we use the EKF to perform the state estimation. Recall that the prediction equations of EKF are

$$\hat{x}_k^- = f(\hat{x}_{k-1}^+, \hat{u}_{k-1}) + \widehat{w}_{k-1}$$

$$P_k^- = A_{x,k} P_{k-1} A_{x,k}^T + A_{u,k} U_{k-1} A_{u,k}^T + Q_{k-1}$$

where $\hat{x}_{k-1}^+$ is the posterior state estimate of the previous time step with covariance matrix P_{k-1}; $\hat{u}_{k-1}$ is the control input that comes from a relative measurement with covariance U_{k-1}; f() is the nonlinear system function of the guidance system; and $\widehat{w}_{k-1}$ is the best estimate of the system noise with covariance Q_{k-1}. The matrices $A_{x,k}$ and $A_{u,k}$ are the Jacobian matrices with the partial derivatives of the system function f() with respect to the state x and the control input u respectively, evaluated at the last state estimate $\hat{x}_{k-1}^+$ and control input $\hat{u}_{k-1}$. We instantiate the EKF prediction equations by calculating these Jacobians as:

$$A_{x,k} = \frac{\partial f(x)}{\partial x} |_{x=\hat{x}_{k-1}^+, u=\hat{u}_{k-1}}$$

$$= \begin{bmatrix} \frac{\partial f_x}{\partial x_{[x]}} & \frac{\partial f_x}{\partial x_{[y]}} & \frac{\partial f_x}{\partial x_{[\phi]}} \\ \frac{\partial f_y}{\partial x_{[x]}} & \frac{\partial f_y}{\partial x_{[y]}} & \frac{\partial f_y}{\partial x_{[\phi]}} \\ \frac{\partial f_\phi}{\partial x_{[x]}} & \frac{\partial f_\phi}{\partial x_{[y]}} & \frac{\partial f_\phi}{\partial x_{[\phi]}} \end{bmatrix}_{x=\hat{x}_{k-1}^+, u=\hat{u}_{k-1}}$$

$$= \begin{bmatrix} 1 & 0 & -u_{[\Delta D]} \cdot \sin(x_{[\phi]} + \frac{u_{[\Delta\phi]}}{2}) \\ 0 & 1 & u_{[\Delta D]} \cdot \cos(x_{[\phi]} + \frac{u_{[\Delta\phi]}}{2}) \\ 0 & 0 & 1 \end{bmatrix}_{x=\hat{x}_{k-1}^{+}, u=\hat{u}_{k-1}}$$

and

$$A_{u,k} = \frac{\partial f(x)}{\partial u} |_{x=\hat{x}_{k-1}^{+}, u=\hat{u}_{k-1}}$$

$$= \begin{bmatrix} \frac{\partial f_x}{\partial u_{[\Delta D]}} & \frac{\partial f_x}{\partial u_{[\Delta\phi]}} \\ \frac{\partial f_y}{\partial u_{[\Delta D]}} & \frac{\partial f_y}{\partial u_{[\Delta\phi]}} \\ \frac{\partial f_\phi}{\partial u_{[\Delta D]}} & \frac{\partial f_\phi}{\partial u_{[\Delta\phi]}} \end{bmatrix}_{x=\hat{x}_{k-1}^{+}, u=\hat{u}_{k-1}}$$

$$= \begin{bmatrix} \cos(x_{[\phi]} + \frac{u_{[\Delta\phi]}}{2}) & u_{[\Delta D]} \cdot \sin(x_{[\phi]} + \frac{u_{[\Delta\phi]}}{2}) \\ \sin(x_{[\phi]} + \frac{u_{[\Delta\phi]}}{2}) & u_{[\Delta D]} \cdot -\cos(x_{[\phi]} + \frac{u_{[\Delta\phi]}}{2}) \\ 0 & 1 \end{bmatrix}_{x=\hat{x}_{k-1}^{+}, u=\hat{u}_{k-1}}$$

These two matrices compute the relative change in the variables of the state, when respectively the last state estimate and control input change. The columns of $A_{x,k}$ and $A_{u,k}$ determine per element of x_{k-1} and u_{k-1} respectively what their contribution to the elements of x_k is. Notice that the x and y variables of the state are the nonlinear components in the system function due to nonlinear relationships with the orientation control input variable $u_{[\Delta\phi]}$ and orientation state variable $x_{[\phi]}$. If these variables are constant, then the system function is linear in the state variables. Moreover, if the distance control input variable $u_{[\Delta D]}$ also does not change, then the Jacobians do not have to be recomputed.

Initialization. Assuming that we can measure the true initial location x_0 of the robot with some precision, we initialize the EKF with the true position of the robot together with the uncertainty. We furthermore assume that we have access to relative measurements $\hat{u}_k$ monitoring the control commands executed by the driving system at every time step. Although there is uncertainty in this, we do not model this explicitly. Thus we let the uncertainty U_k in the relative measurements be zero. Instead, we incorporate the uncertainty in the relative measurements in the system noise. We assume that the system noise mean $\widehat{w_k}$ is zero at all times, but with some covariance Q_k.

4.2.4.2 Corrective Position Tracking

Correction Equations. Recall that the correction equations of the EKF are

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-))$$

$$P_k^+ = (I - K_k H_k)\, P_k^-$$

where

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$$

where the Jacobian matrix H_k with partial derivatives of the measurement function h() with respect to the state x is evaluated at the prior state estimate $\hat{x}_k^-$, which in this case is

$$H_k = \frac{\partial h(x)}{\partial x}\Big|_{x=\hat{x}_k^-}$$

$$= \begin{bmatrix} \frac{\partial h_x}{\partial x_{[x]}} & \frac{\partial h_x}{\partial x_{[y]}} & \frac{\partial h_x}{\partial x_{[\phi]}} \\ \frac{\partial h_y}{\partial x_{[x]}} & \frac{\partial h_y}{\partial x_{[y]}} & \frac{\partial h_y}{\partial x_{[\phi]}} \\ \frac{\partial h_\phi}{\partial x_{[x]}} & \frac{\partial h_\phi}{\partial x_{[y]}} & \frac{\partial h_\phi}{\partial x_{[\phi]}} \end{bmatrix}_{x=\hat{x}_k^-} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notice that since the Jacobian only contains constants, the measurement function is linear in all the state variables. We do not have to recomputed the matrix at every time step.

4.2.5 Testing

To test the efficiency of EKF, a simulation is in Matlab. This simulation is based on the EKF algorithm mentioned above based on other people’s work.

4.2.5.1 Map

As shown in Figure 3, a map is uploaded before testing EKF. The black line stands for the obstacles, which may the walls, the desk, etc. The blue “+” stands for the localization which is known before testing. Also, the control data needs to be given, as the red line. This is the expected path for R2D2 to move.

Data: the system noise is 2 cm, 1 degree; the sensor range is 0.2-5.6 m; the angle of sensor is pi. The measurement noise is 3 cm, 1 degree; the sensor’s angular resolution is 1 degree; control is 0.1m/s translational velocity, and 0.02 rad/s rotational velocity.

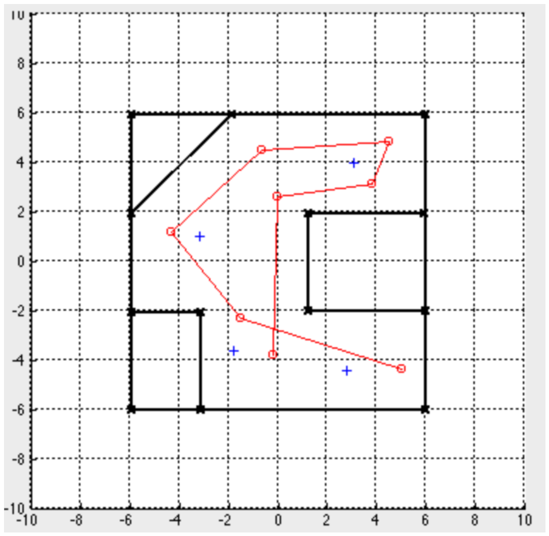


Figure 3. Simulation Map

4.2.5.2 Simulation

The green line stands for the sensor detect the landmark process. The red ellipses around landmarks stand for the landmark covariance while the robot is moving. The red ellipses around robot stand for robot position covariance while the robot moving. The true robot position is blue line and blue triangular. The estimated robot position is the red line and red triangular. It can be seen from the Figure 4 that, the position error gets largest when the robot turns around. Figure 5 gives the ending condition of the robot and the path it moves and estimates.

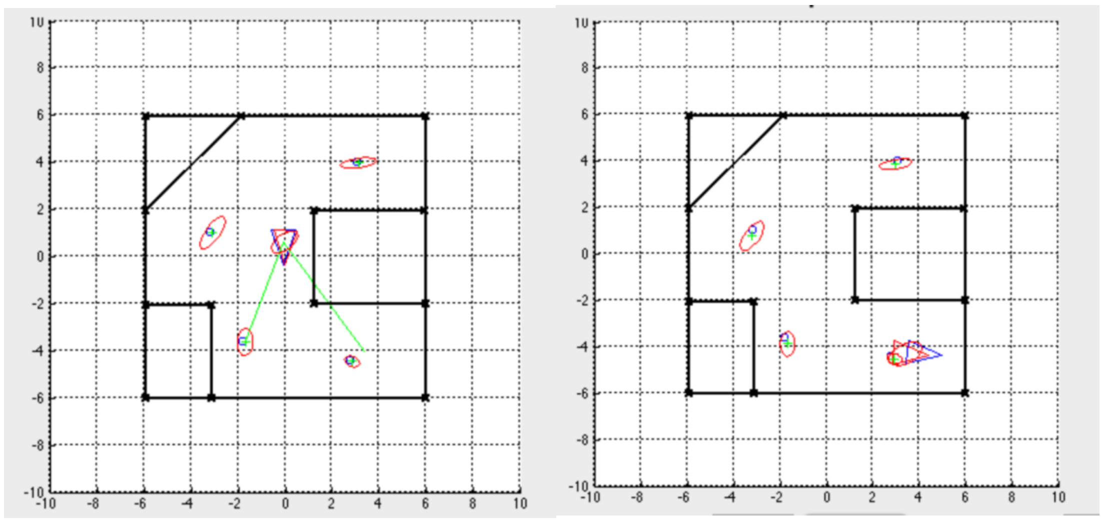


Figure 4. EKF simulation

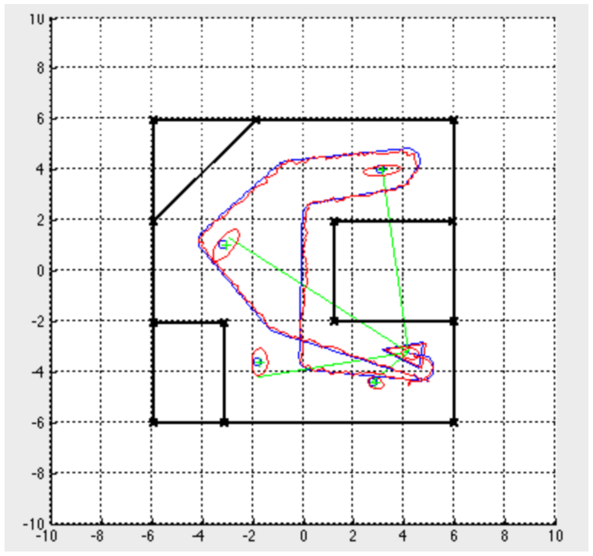


Figure 5. EKF ending

4.2.5.3 Result Analysis:

As shown in Figure 6, the error and deviation are analyzed in the accumulated time.

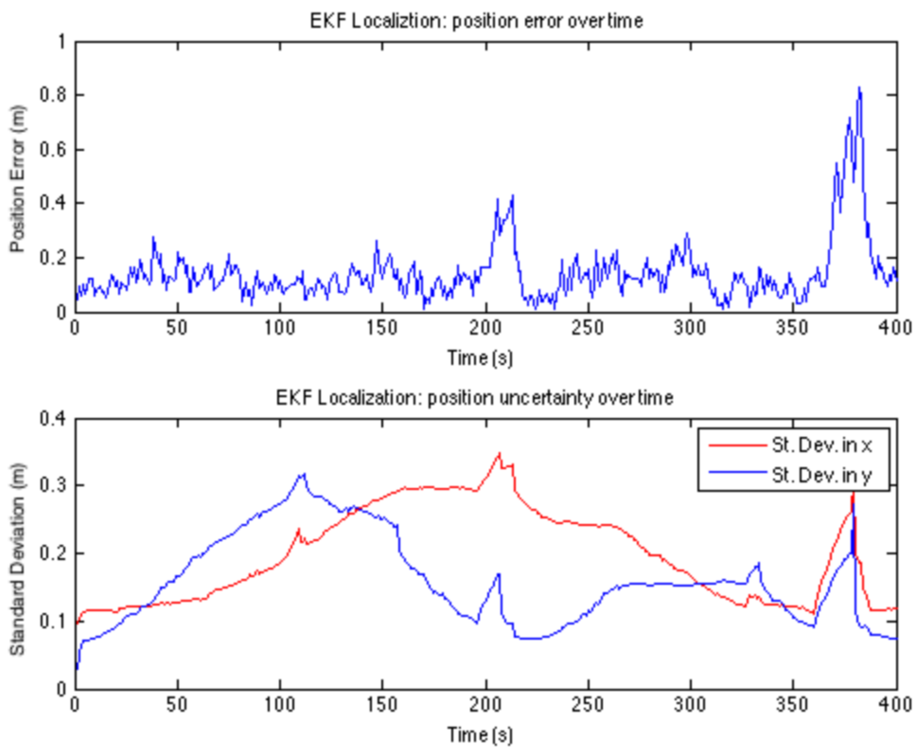


Figure 6. Position Error & Standard Deviation

The figure 6 describes the x deviation and y deviation according to the world map. The mean values of the estimated results are close to the way points while their standard deviations are very small. It confirms that the EKF is capable of providing more accurate localization results. It should be noted that some pose errors in x and y direction. These errors may due to the turning executed by the controller.

If the modeled system noise is lower than the actual system noise, then the EKF will incorrectly assume that the uncertainty in the state of the system increases with the modeled system noise. In our R2D2, modeling lower system noise than the true system noise has as consequence that the location estimates have less true system noise has as consequence that the location estimates have less uncertainty than they should have. This can result in decision making problems, since these can be based on the location of the robot. Having lower system uncertainty can result in the true state not being captured by the confidence intervals. The information content of the location estimates is higher, but it may not be consistent.

If the modeled system noise is higher than the actual system noise, then the EKF will assume more noise in the system than that there is and increase the uncertainty in its state estimates unnecessary much. In localization context, modeling higher system noise than the true system noise makes the uncertainty in the state estimates increase quicker than that they strictly should. Thus, location estimates are more unsure and therefore decision making controllers that base their decisions on the location estimates will be more conservative. The information content of the location estimates is lower.

The figure 7 shows the scan error of the sensor and the odometry error of odometer in accumulated time. It can be seen the largest error happens when the robot turns around the corner.

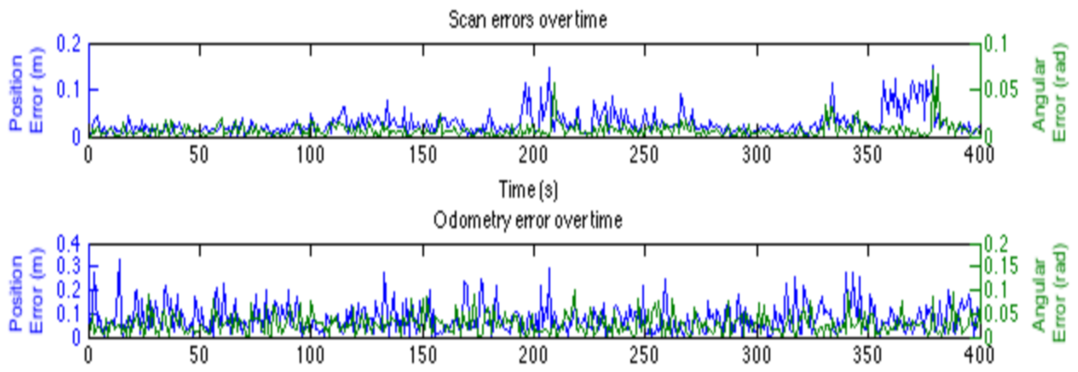


Figure 7. Scan & Odometry Error.

From the analysis above, it can be seen that EKF is reliable in localization.

4.2.6 Conclusion

In order of appearance we discussed the following topics.

4.2.6.1 Robot Localization.

From a practical point of view we discussed the need for a robot to localize itself while navigating through an environment. We identified three different localization instantiations, position tracking, global localization, and the kidnapped robot problem. In order to determine what information a robot has access to regarding its position, we discussed different sources of information and pointed out advantages and disadvantages. We concluded that due to the imperfections in actuators and sensors due to noise sources, a navigating robot should localize itself using information from different sensors.

In order to formalize this, we considered the localization problem in probabilistic context as a Bayesian estimation problem. We defined the belief of the robot as the probability density over the space of possible locations; conditioned on types of location information it has access to. With understanding and application of Bayes' rule and the Markov assumption, we obtained a localization formula that formalizes the incorporation of relative and absolute position measurements into this belief. We found that in order to implement the localization formula we need a concrete representation of the location space, along with models that describe the influence of actions on the location, and that describe the relation between measurements and locations. Discussing several methods that implement the formula, we found different ways of implementing these.

4.2.6.2 Extended Kalman Filters.

We thoroughly discussed the basics of the Kalman Filter. We looked at the assumptions that the Kalman Filter poses on the system of which it estimates the state: a linear dynamic system with linearly related measurements, corrupted by Gaussian distributed, white, zero-mean noise. We looked at the implications of these assumptions, which led us to the equations that form the Linear Kalman Filter. We created better understanding of the Kalman Filter by describing the meaning of the different equations.

Since the Linear KF only works for systems that can be described with linear system and measurement models, we discussed how we can use linearization techniques to obtain KF extensions that estimate the state of nonlinear problems. We argued that the Extended Kalman Filter improves this by incorporating the latest state estimates into the linearization process. In line with this, the Iterated Extended Kalman Filter repeatedly linearizes a corrected state estimate to find the best state estimate used for inclusion of a measurement. We showed how we can extend the formulation of the EKF to systems and measurements with any number of parameters, as long as the uncertainties in them are independent of each other. We furthermore pointed out the key ideas of alternative extensions, which do not require

Jacobian matrices to be calculated and allow for other than Gaussian distributions. These extensions may further improve the estimation performance.

4.2.6.3 EKF Localization

In order to show how we can use Extended Kalman Filters in the Robot Localization context we derived models for acting and sensing and discussed the true behavior of these systems in the light of noise. Among others, we concluded that the Gaussian noise assumption is not always valid, and that uncertainty in the orientation of the robot causes the uncertainty in the location to increase.

The EKF has been successfully implemented in our R2D2 robots and works perfectly.

Right now, we are trying to combine the kinect and odometry and IMU to do a better sensor fusion. We currently successfully combine the “fake” sensor data from kinect that CS vision team gives me. And we already successfully combine the IMU with odometry to get better performance. Hopefully in next semester, the vision team can get kinect work in R2D2 and we could test the real kinect and do more data fusion using EKF to get better localization.

4.3 GSM communication for R2 voice recognition

In order to be able to use GSM communications, we needed to setup a cheap mobile plan that would provide us with a SIM card and the required texting service. Fortunately, AT&T offers a no contract, month-to-month GoPhone plan with unlimited texting for $30/month. After going to the local store and explaining to the employees our need, we received our SIM card and account and were able to begin working almost immediately.

The GSM shield for the Arduino allows the use of a SIM card to communicate over a network carrier’s GSM network, using the Arduino. The interface uses UART that makes it easy to use the device. Furthermore, the shield has pre-written library functions that make it easy to use.

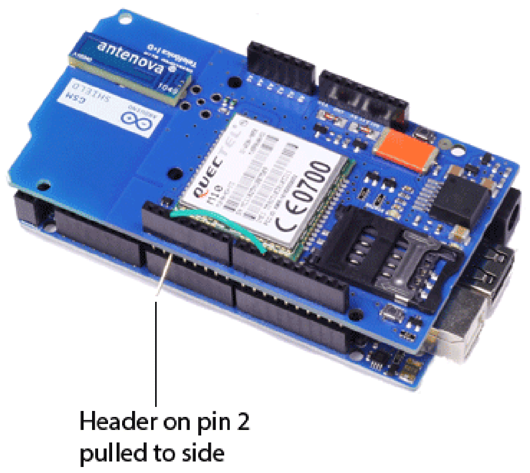
The key idea for the voice recognition was that, a smart watch or smart phone with speech to text service would be used that would send text messages that the Arduino can read and respond to. The text message would contain the command to send to R2D2, meaning that only certain pre-defined commands are allowed to be spoken. As long as these words are in the user’s phone’s dictionary, the speech to text conversion worked very reliably. To make the application more portable, commands chosen are regular English words. So, an example command is “Hello”.

The base application relied on the sample code provided as part of the library for receiving text messages. The main difference is that the received message is forward onto a second serial port. The GSM library uses software serial for communication to ensure that it does not interfere with operation with the other serial ports. Initially the default Serial port (for PC communications) is used for debugging messages in the demo code.

The initial testing was performed by checking the received message to see if it started with either the string "on" or the string "off". If it started with "on", an LED connected to pin A5 was lit up (A5 set high) and if it started with "off", the LED was turned off (A5 set low). Once this was confirmed, the debug messages on the Serial port were removed since it was intended to use Serial for sending the messages to the computer for the CS team to process. However, removing the serial debug messages from the GSM library made the GSM reception not work and the initialization to fail. This was mysterious since it indicated some sort of timing issue in the library that was fixed by additional delays because of the serial debug message printing.

Since the Arduino Mega (the Arduino being used for this) had another serial port available for use, it was determined that it would be quickest to just keep the debug messages enabled to fix the timing issues and use the other serial port (Serial1) to forward the received message. This worked without issues.

Hardware wise, there were two things to be careful of. According to the GSM shield website, the Arduino relies on an interrupt on pin 2 of the shield (GSM_TX pin) "to know when information is available to read". This is pin 2 on the Arduino Uno. The GSM library will redefine the GSM_RX pin automatically depending on the board selected. However, this is not done for the GSM_TX pin that needs to be manually re-routed. For the Mega, the GSM library uses pin 10. So, a jumper needs to be placed between pins 2 and 10 on the Mega to re-route the signal properly. Additionally to ensure that the signal is not interfered by or interferes with pin 2, the header pin on the shield is bent to disable connection to the Mega.



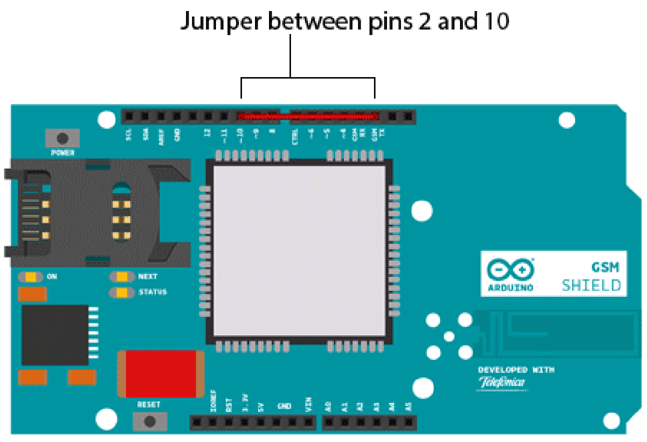


Image source: http://www.arduino.cc/en/uploads/Guide/Mega_GSMShield2.jpg

There is another hardware consideration to make. During peak usage, the shield can have a peak current draw of about 2A. To help deal with this current surge, the large orange capacitor is used. This is why it is recommended to use an external power supply rated between 700mA and 1000mA capacity to power the shield and the Arduino since the required current cannot be provided over USB connection with the shield "under heavy use".

Reference for description and image: http://www.arduino.cc/en/Main/ArduinoGSMShield

```
/*
 SMS receiver


 This sketch, for the Arduino GSM shield, waits for a SMS message

 and displays it through the Serial port, along with all debug

 messages during GSM operation. Additionally, it has an external

 LED that can be turned on or off by sending text messages starting

 with "on" and "off" respectively.


 Circuit:

 * GSM shield attached to and Arduino (on Mega2560, connect pin 2

   on shield to pin 8 on Mega and remove pin 2 connection from Mega)

 * SIM card that can receive SMS messages


 Modified 02/28/15

 by Syed Tahmid Mahbub (Cornell Cup USA)
```

```
*/


// include the GSM library

#include <GSM.h>


// PIN Number for the SIM

#define PINNUMBER ""


// initialize the library instances

GSM gsmAccess(true);

GSM_SMS sms(true);


// Array to hold the number a SMS is retreived from

char senderNumber[20];


String str_on("on");

String str_off("off");


char c_array[50];


#define LED       A5

#define LED_on()  digitalWrite(LED, HIGH)

#define LED_off() digitalWrite(LED,  LOW)
```

```
void setup()
{
 // initialize serial communications and wait for port to open:
 Serial.begin(9600);
 Serial1.begin(9600);
 pinMode(LED, OUTPUT);  LED_off();

 Serial.println("SMS Messages Receiver");

 // connection state
 boolean notConnected = true;

 // Start GSM connection
 while (notConnected)
 {
  if (gsmAccess.begin(PINNUMBER) == GSM_READY)
   notConnected = false;
  else
  {
   Serial.println("Not connected");
   delay(1000);
  }
 }
```

```
  Serial.println("GSM initialized");

  Serial.println("Waiting for messages");

}


void loop()

{

  char c;


  // If there are any SMSs available()

  if (sms.available())

  {

    Serial.println("Message received from:");


    // Get remote number

    sms.remoteNumber(senderNumber, 20);

    Serial.println(senderNumber);


    // An example of message disposal

    // Any messages starting with # should be discarded

    if (sms.peek() == '#')

    {

      Serial.println("Discarded SMS");

      sms.flush();

    }
  int index = 0;


  // Read message bytes and print them
  while (c = sms.read())
  {
   Serial.print(c);
   c_array[index] = c;
   if (index++ > 45) break;
  }


  c_array[index] = '\0';


  Serial1.println(c_array);
  /*
  {
   String txt(c_array);
   txt.toLowerCase();
   if (txt.startsWith(str_on))
//      Serial.println("\n\n\n\nGOT ON!!!!\n\n\n");
    LED_on();
   else if (txt.startsWith(str_off))
//      Serial.println("\n\n\n\nGOT OFF!!!!\n\n\n");
    LED_off();
  }
  */

  Serial.println("\nEND OF MESSAGE");

  // Delete message from modem memory
  sms.flush();
  Serial.println("MESSAGE DELETED");
 }

 //sms.flush();
 delay(1000);

}
```

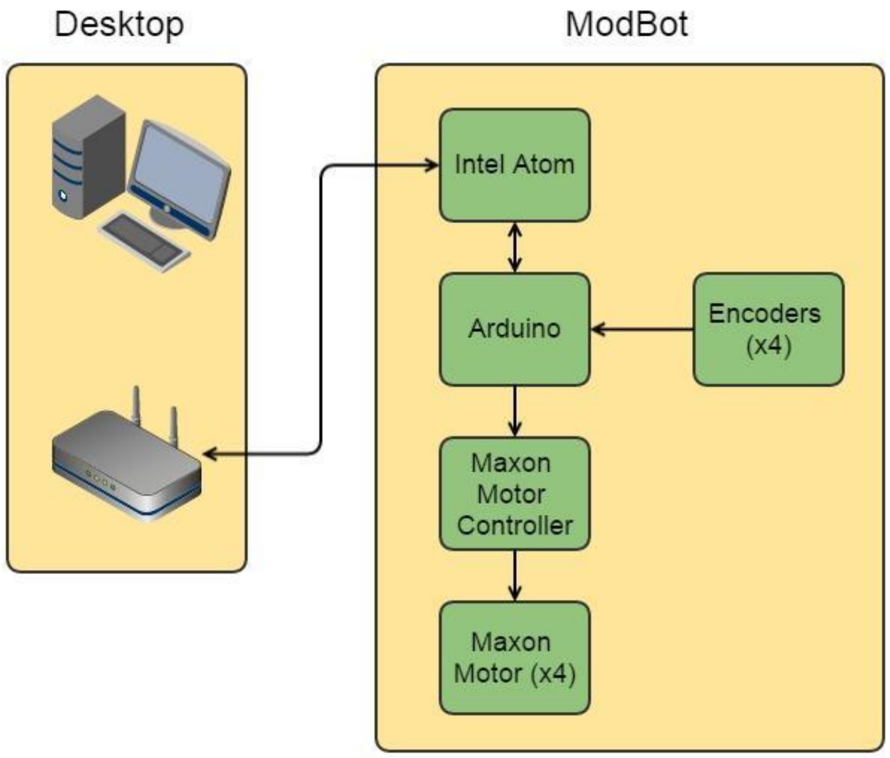
5 Modbot Motor Control

The Cornell Cup previously used Maxon brushless motors in several of its designs including the Modbots. Modbots use these motors as the main drive system and therefore need precise characterization of their performance for optimal motion prediction. The Cornell Cup team envisioned these Modbots driving autonomously which requires better acceleration and deceleration characteristics. The old controllers were tested and do not have all of the characteristics necessary for our purposes.

5.1 High Level Design

We selected new controllers and configured them with appropriate settings. This includes synchronization to the current communication protocol from the Atom board on the robot. Based on where the robot needs to go, the motherboard sends a string of character string to an Arduino microcontroller. This Arduino reads the string, parses the data, and sends it in a proper format to the ESCON36/3c controller. The controller is set up to run the motors based on this signal.

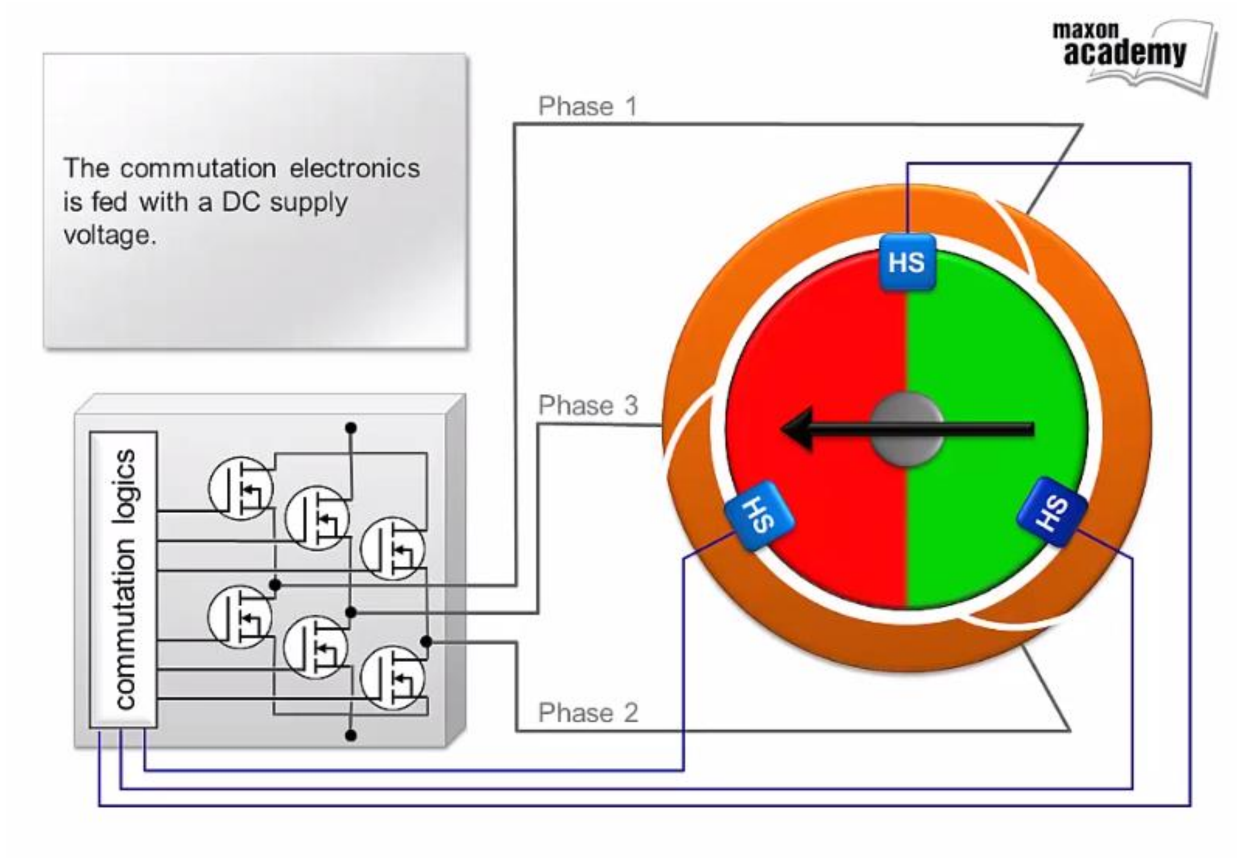
The Arduino is also used to read and transmit the values of 4 encoders from the Maxon controllers. This data is sent back to the base desktop and integrated into the high level path planning decisions.



5.1.1 Program/ Hardware Design

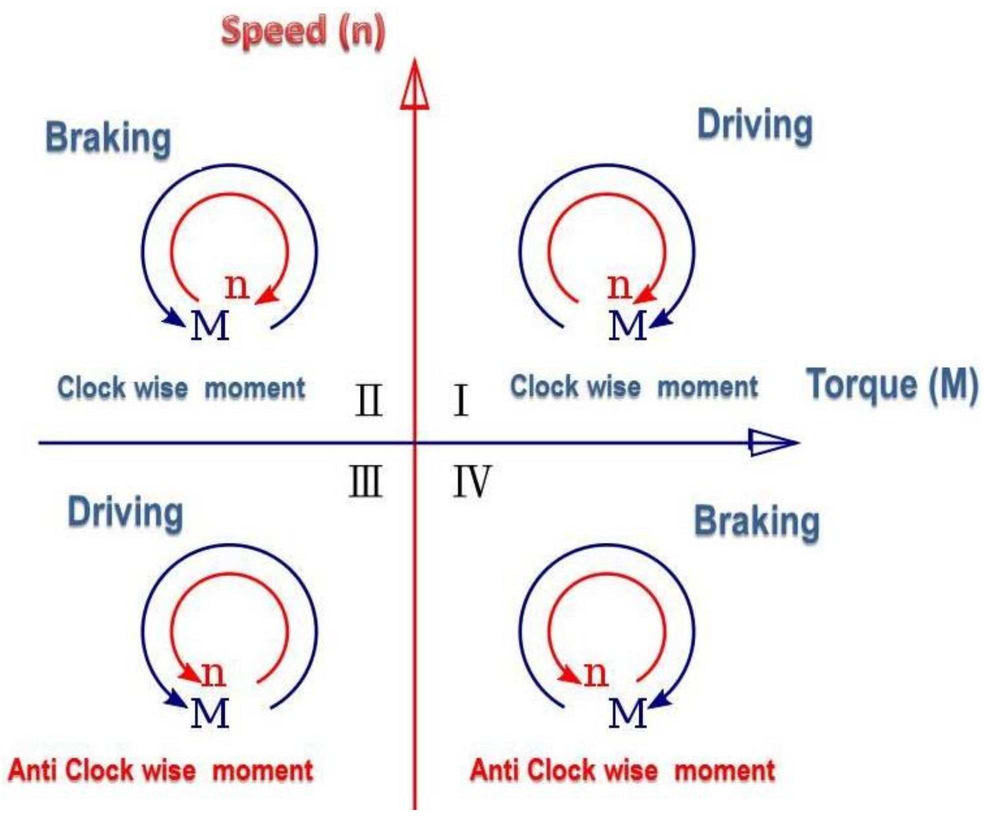
Maxon brushless motors are currently installed on the Modbot and Dunebot robots. The motors are 40 watt brushless motors with hall sensors. They include a 33:1 gearhead and a 3 channel line encoder. The hall sensors were used by the old controller as a feedback loop for the given velocity. They are used in the same capacity for the new control system.

The basic principles of a brushless motor are demonstrated below. A permanent magnet serves as the motor rotor and three coil windings are turned switched at various times to rotate the rotor. Each of the three phase wires can serves either as the power supply or ground in this configuration. This requires a controller with the logic capable of performing this action. The controller must read input from three hall sensors (HS) to ensure smooth operation. Each hall sensor can detect whether the magnetic field of the rotor is pointed toward or away from it. Using this information, the controller determines the position and speed of the rotor. The Modbot uses Maxon brushless motors as the main drive components


maxon academy
The commutation electronics is fed with a DC supply voltage.
Phase 1
Phase 3
Phase 2
commutation logics
HS
HS
HS

https://www.youtube.com/watch?v=ywAB7UPOFTg

We chose new ESCON 36/2 controllers because of their ability to provide 4 quadrant control (compared to 2 quadrant control for the old controllers). The following image demonstrates the benefits of 4 quadrant control.



http://en.wikipedia.org/wiki/File:Four_quadrant_motion_control_of_a_motor.jpg

While our old controllers can operate in region I and III. ESCON 36/2 controllers can operate in all four regions. The issue with the old controllers is that they cannot provide characterized deceleration or acceleration data when changing speeds (1 quadrant operation). This data is necessary to calculate the current position of the robots. Without knowing the deceleration characteristics, the robots amass a significant amount of drift after continuous operation. Since our goal is to maneuver these robots precisely even in confined spaces, such uncertainties are unacceptable. As shown in the figure above, these controllers are able to apply negative torque and absorb power at a controlled rate. This allows for characterized and adjustable deceleration. Unlike the old system, the Modbots do not coast to a stop but rather decelerate.

5.1.2 Hall Effect Sensors for Velocity Measurement

In order to effectively localize the modbot in space, it is critical that we obtain an accurate measurement of the rotation speed of each of the four motors. The motors give us two options for doing so: Encoders and Hall Effect Sensors.

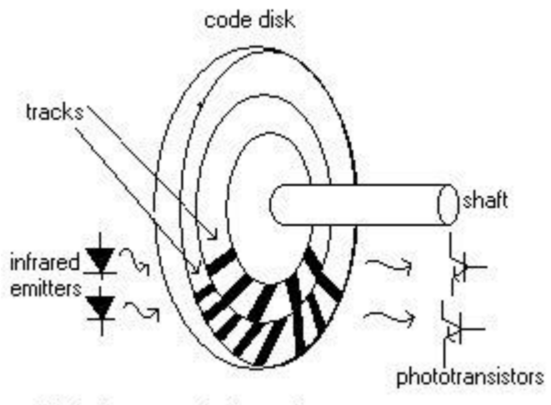
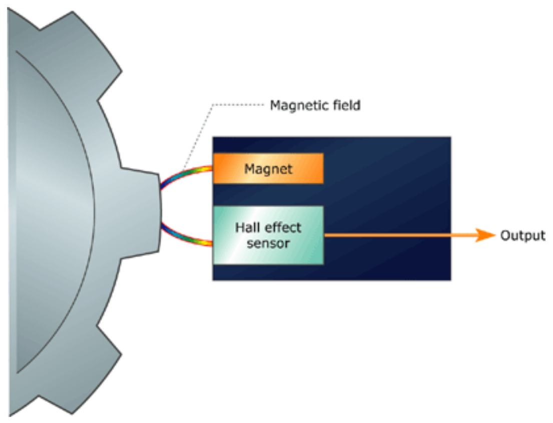


Fig 1. A rotary optical encoder

A schematic of a basic optical encoder



A high-level overview of a hall-effect sensor

While there are several different versions of encoders, the one on our ModBot (and generally the most common), is an optical encoder as shown in the figure above. Optical encoders work based on optoelectronic principles. In this case, an LED is shown through a finely screened wheel that is attached to the motor shaft. As the shaft rotates, light intermittently passes through the slits in the screened wheel with the frequency of the "flashes" of light depending on how fast the motor shaft rotates. A phototransistor circuit is then able to count the number of flashes in a given time, and because the screened wheel is accurately calibrated, the circuit thus is able to output several digital signals corresponding to the speed of the motor. From these signals, an accurate speed and direction of the motor can be calculated. While optical encoders are generally very accurate, they are easily susceptible to dust, and as a result, malfunction easily in harsh or dusty environments.

On the other hand, our ModBot is also equipped with hall effect sensors (as seen in the second figure on the previous page). These sensors work based on the hall effect (hence the name). A permanent magnet is mounted next to a hall effect sensor. Accordingly, a gear is mounted on the shaft which rotates. When the teeth of the gear passes between the magnet and the hall sensor, it changes the magnetic field produced by the magnet. The hall sensor is able to measure the changing magnetic flux caused by the rotation of the gear. The sensor is then connected to a circuit which outputs an analog voltage corresponding to the speed of the shaft. Hall sensors are generally less accurate (changing flux is harder to measure compared to crisp blinks of light) than optical sensors. However, the main advantage of hall sensors is their durability, as the contact-less system removes mechanical wear and fatigue, and the system is not as susceptible to dust as encoders. Furthermore, hall sensors integrate well with our motor controller.

Thus, while the motors are equipped with both encoders and hall sensors, we decided to utilize the hall sensors for velocity measurement. We did not use the encoders because of their redundancy and fragility. Both the hall sensors (and this model of encoder) output information as a velocity. As mentioned before, the advantage of hall sensors is that they are durable and are able to output a signal through the motor controller, and thus allows us to customize the output format of the hall sensor to suit our needs.

5.1.2.1 Set-up of the hall sensor

In our case, we chose to output the velocity of the motor from the hall sensor. The motor controller allows us to output a voltage between -4V and 4V corresponding to a maximum negative rpm (reverse direction) and a maximum positive rpm of the actual motor (pre-gearing). Originally, we programmed the motor controller to output -4V at -10000 rpm, and 4V at 10000 rpm (with a linear scaling between). However, we encountered a problem when reading the analog voltage using an analog-to-digital converter on an Arduino Mega (which we will discuss in detail later). For now, it is **critical to note that the ADC converter on the Arduino cannot read negative voltages**. Thus, while we originally read incorrect values on the Arduino, we recognized our mistake and corrected the error by outputting only positive voltages from the motor controller (hence the benefit of the customizability of our hall sensor).

As a result, our motor controller encoded the motor speed as a function of a voltage. More specifically, the motor controller output between 0 to 4V, corresponding to -10000 rpm and 10000 rpm respectively and with a linear fit in between. Thus, a 2V signal from the hall sensor would indicate that the system was stationary. From this voltage, we are thus able to calculate the velocity of the motor.

(As a test, we did perform readings of the encoder information and verified that the encoder was sending correct pulses. All that is needed is for a program to calculate the speed and direction of the motor based on these pulses, which can be implemented in the future. For pin information, see the datasheet for the HEDL 5540).

5.1.2.2 Reading and transmitting hall sensor data

The second component of the control system is the ability to read and transmit hall sensor data back to the base station. This was necessary to track the position of our robot at the wheels. To do so, we calculated the velocity of the motor from the analog voltage supplied by the hall sensors. In order to convert our analog voltage to a numeric value of rpm, we utilized an Analog to Digital Converter (ADC) on the Arduino Mega located on our ModBot. In the high-level overview of the system, each of the four motors on the ModBot is controlled by a motor controller. In turn, each motor controller outputs an analog voltage corresponding to the speed of its respective motor. Thus, to compile the velocities of all four motors, we wired each motor controller to an Analog Pin of the Arduino Mega (A0 to A3 in our case). As a preface, we will use the function analogRead() (see documentation at http://www.arduino.cc/en/Reference/AnalogRead) to access the Arduino Mega's 16 channel, 10-bit ADC. This function outputs a digital value between 0-1023, corresponding to 0-5V (i.e. we obtain a resolution of .0049 V per unit). We will demonstrate our calculations for voltage to speed later.

Because our motor controller code needs to accomplish many tasks continuously (i.e. receive and process commands from the base station, write voltages to control the motors, etc.), it is difficult to read our hall sensor voltages without affecting the flow of the code. We wanted to read the hall sensor values at regular intervals (e.g. 10 Hz) while also being able to fluidly control our robot without interference. Initially, we attempted to do so via interrupts.

For full documentation about interrupts, please see the datasheet about your specific microcontroller. It will provide the necessary information about setting up interrupts. In our case, we used an atmega2560 located on the Arduino Mega. Moreover, we used a timer overflow interrupt (See figure).

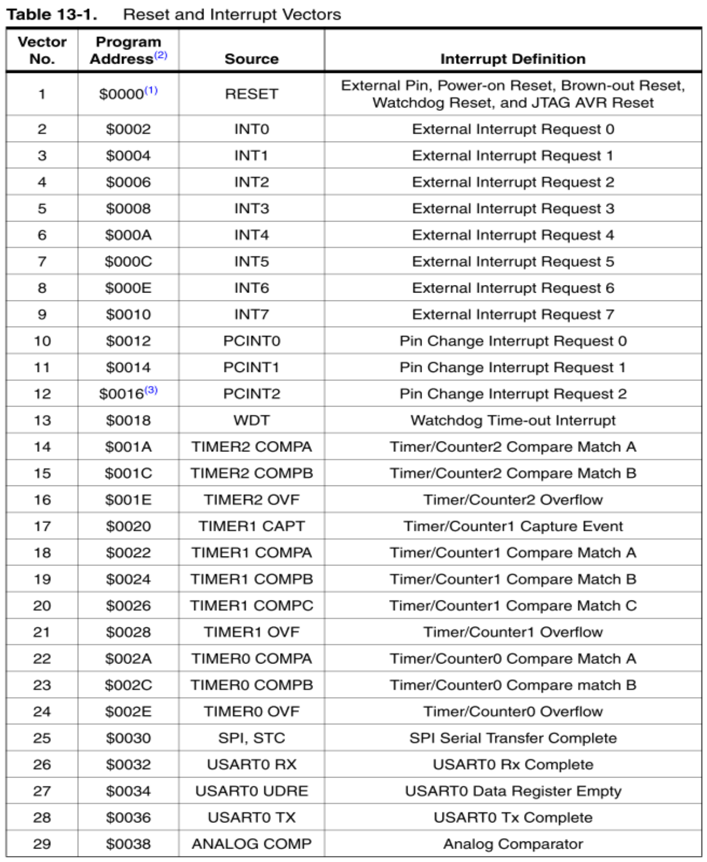


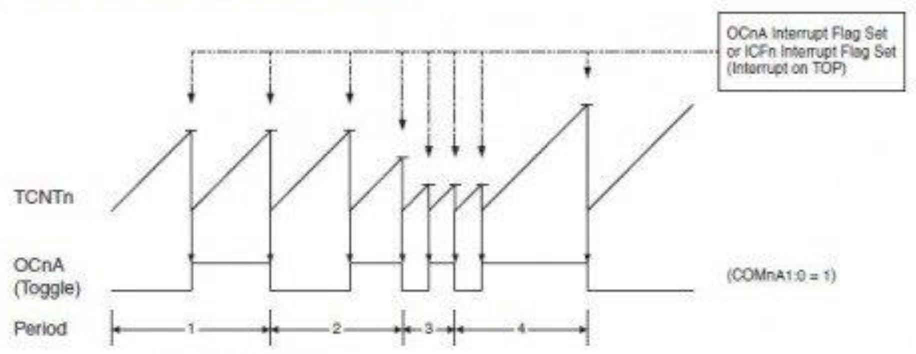
Table 13-1. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	PCINT1	Pin Change Interrupt Request 1
12	$0016[3]	PCINT2	Pin Change Interrupt Request 2
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	$001E	TIMER2 OVF	Timer/Counter2 Overflow
17	$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	$002E	TIMER0 OVF	Timer/Counter0 Overflow
25	$0030	SPI, STC	SPI Serial Transfer Complete
26	$0032	USART0 RX	USART0 Rx Complete
27	$0034	USART0 UDRE	USART0 Data Register Empty
28	$0036	USART0 TX	USART0 Tx Complete
29	$0038	ANALOG COMP	Analog Comparator

Some possible interrupts on the ATMEGA2560

The ATMEGA2560 comes with five timers integrated into the chip. Since we wanted to sample the hall sensors at regular intervals, a timer interrupt was fitting. More specifically, we used timer 5's overflow interrupt (TIMER5_OVF_vect). This interrupt operates on the principle seen in the figure below. Depending on a prescaling factor (i.e. how slowly the clock runs), the timer will increment a counter at a set rate. Once the counter hits 66536 (the maximum for a 16 bit timer), the interrupt will trigger and execute the code inside of the vector. The timer will then be reset and allowed to overflow again.

Figure 16-6. CTC Mode, Timing Diagram



Operating principle behind timer interrupts

To set up our overflow interrupt, we referenced the atmega2560 datasheet. In the setup() function of our code, we implemented the following.

```
/**********************************SetupInterrupts**********************************/
TCCR5A = 0;     //set entire TCCR5A register to 0
TCCR5B = 0;     //same for TCCR5B

TCNT5 = 34286;  //preload timer 66536 - 16MHz/256/2Hz (will overflow and trigger interrupt

TCCR5B |= (1 << CS52);  //Set prescaler to 256
TIMSK5 |= (1 << TOIE5); //Enable timer overflow interrupt

UCSR0A |= (1 << RXC0);
```

Much of our code was simply based on the setup information stated in the datasheet. First we simply had to clear timer 5's registers to prepare for the setup. TCNT5 is the actual counter of the timer. Thus, we had to preload the timer with a specific value to ensure that it will overflow at an appropriate rate (in our case 2 Hz). To determine this value, we calculated that 66536 is the point at which the timer overflows. The atmega2560 runs at 16MHz, and we set the prescaler at 256 (i.e. the timer increments every 256 cycles). We wanted to trigger once every two seconds. Thus, we calculated that the timer should be initialized at 66536-16*10^6/(256*2)=34286. Afterwards, the remaining code is simply initializing the prescaling factor and telling the timer to run the overflow vector, as was instructed on the datasheet.

After the timer is setup, we must write a method stating what to do when the timer is triggered. The method name must be ISR, and it must be passed TIMER5_OVF_vect, as shown below.

```
ISR(TIMER5_OVF_vect){
  analogInRF = analogRead(1);
  analogInRB = analogRead(3);
  analogInLF = analogRead(2);
  analogInLB = analogRead(0);
  TCNT5 = 34286;       //preload timer

  dataAvailable = 1;
}
```

Thus, once the timer is triggered, we read and then store each of the voltages (see above for information about analogRead()) to a volatile variable. We then reset the timer so that it will trigger again in 0.5 seconds, and then finally set a “flag” (a boolean variable) that indicates when we have data available.

```
        if(dataAvailable == 1){
          float rpmRF = rpmCalc(analogInRF);
          float rpmRB = rpmCalc(analogInRB);
          float rpmLF = rpmCalc(analogInLF);
          float rpmLB = rpmCalc(analogInLB);

          Serial.print(rpmLF);
          Serial.print(" ");
          Serial.print(rpmRF);
          Serial.print(" ");
          Serial.print(rpmLB);
          Serial.print(" ");
          Serial.println(rpmRB);

          dataAvailable = 0;
        }
```

In our main loop() of the code (see above), we then simply calculate the actual motor speed from our analog values every time we see that we have data available. We then print out those values and reset the flag. To perform our calculations on motor speed, we know that analogRead() gives values of

0-1023 corresponding to 0-5V. From there, you can convert the 0-1023 values to values between -10000 to 10000 RPM. (So when the wheel isn't moving, you should get 1023/5 * 2 = 409). After that, the motor has a 33.0 gear ratio, so that RPM value needs to be divided by 33 to obtain the final RPM of the wheel. Thus, we used a function to perform our calculations as seen below.

```
float rpmCalc(float analogVal){
  float r = (rMax / 409.2) * analogVal - rMax; //linear equation for motor's rpm
  float rpm = r / 33.0; //divide by 33.0 due to gear ratio to find true rpm
  return rpm;
}
//In this case, rMax = 10000, which is what we have it set currently on the motor controller.
```

After implementing our code, we discovered a problem after intensive testing. While controlling our ModBot with a joystick, the code seemed to perform well until ~5-10 minutes into testing. In very rare instances, our ModBot would not get the “stop” command sent from our base station. Thus, this caused the ModBot to continue driving at its previous speed, causing it to crash. It is critical to note that bugs such as this occur very sporadically (~once every 10 minutes). However, it is a critical failure that must be addressed. Thus, *while programs may seem to work initially, it is critical that extensive testing be performed with ample time to identify and address such problems*. This is especially important when encountering bugs that happen so infrequently, it requires many hours of testing to even begin to identify the source of the problem. Furthermore, there is no one set way to fixing these bugs, only simply many rounds of iterative testing to gradually identify the source of the error.

In our case, we concluded that the modbot was not reading the commands sent through serial and processed in our main loop (as seen below).

```
while (Serial.available()) {
  Serial1.println(Serial.peek());
  if (Serial.peek()=='M'){
    Serial.readBytes((char*) motor,7);
    if (motor[6]=='#'){
    setMotor();
    }
  }
  else {Serial.readBytes((char *) garb,1);}
```

To alleviate this problem, we assumed that processing the hall sensor data and reading the analog voltages was taking too much processor time, and that caused the code to miss processing some of the serial buffer. Thus, we outsourced the rpm calculations to our base station. Furthermore, instead of performing all four readings per interrupt, we spread out the readings so that one wheel is read per trigger. Thus, we increased the frequency at which the interrupt triggered to 8 Hz. As a result, we were able to check for available commands in the main loop more frequently, while also being able to output the same information.

However, while our solution did seem to alleviate the problem, it did not completely fix it. After more extensive testing, we discovered that we now only encountered the missed command bug approximately once every 20+ minutes. This effectively made the bug harder to troubleshoot, as we had to test for much longer periods of time per iteration of the code. However, our solution did point us in the right direction. It verified that some part of our code was holding up how the serial commands were being read, and as a result, we were missing commands.

Finally, to fully address this problem, we asked for outside feedback from more experienced individuals. As a result, and after much online research, we discovered that ***serial commands work via interrupts***. Thus, it is very inadvisable to use additional interrupts that will trigger when we are receiving serial commands. For example, if we are processing our serial buffer, which occurs via interrupts, and our timer interrupt in turn interrupts our serial processing, we may in turn lose part of our serial data. Hence, the only effective solution to remedy our problem was to not use an interrupt.

We thus approached our problem via a method of using polling loops. The idea behind a polling loop is just to simply check a condition every time our main loop executes. Usually this condition is some sort of timer or counter. If this condition has been satisfied, we will execute the commands in the body of our polling loop. While this solution is less elegant than interrupts (and generally less efficient due to having to check a conditional once every loop), it does resolve the problem of interrupts conflicting while reading serial data. Thus, we implemented our polling loop as follows.

```
 if(TCNT5 >= 1562){
   analogInLB = analogRead(0);
   analogInRF = analogRead(1);
   analogInLF = analogRead(2);
   analogInRB = analogRead(3);
   Serial1.print("%");          //Send % symbol to indicate
   Serial1.print(analogInLB);
   Serial1.print(" ");
   Serial1.print(analogInRF);
   Serial1.print(" ");
```

```
  Serial1.print(analogInLF);
  Serial1.print(" ");
  Serial1.println(analogInRB);
  TCNT5 = 0;
 }
```

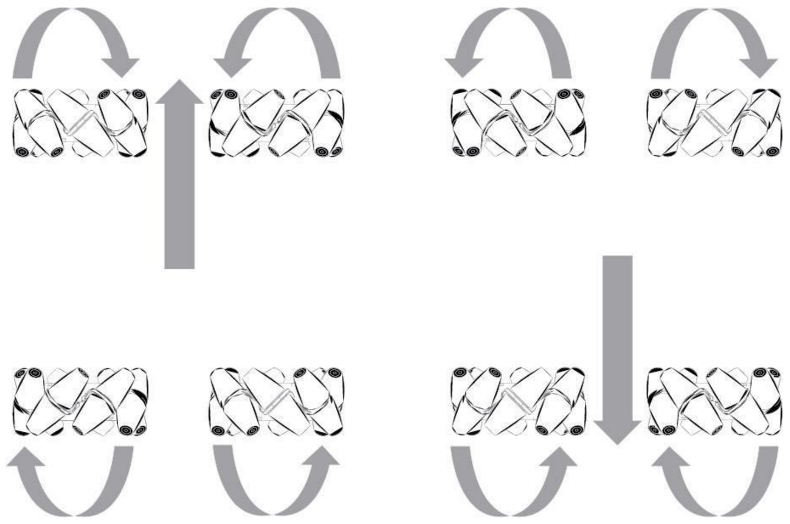
The polling loop still uses the timer5 on the atmega2560. While the basic principles behind this and our original interrupt remain the same, we removed the ISR method and instead started the timer at 0, and made it “overflow” at 1562 with a 1024 prescaler.

```
  TCCR5A = 0;// set entire TCCR2A register to 0
  TCCR5B = 0;// same for TCCR2B
  TCNT5  = 0;//initialize counter value to 0
  // Set CS21 bit for 1024 prescaler
  TCCR5B |= (1 << CS50);
  TCCR5B |= (1 << CS52);
```

As a result, our polling loop executes at roughly 10 Hz. Thus, in the main loop, every iteration of the loop will check to see if the timer has overflowed a predetermined value. If so, we will simply read the analog voltages associated with each wheel and print the values to our base computer through serial. After every execution of the loop, we will reset the timer to 0.

However, we encountered one final problem while sending serial data back to the computer. If we attempted to send serial data through the serial port while our base computer sent serial data to our microcontroller, we would effectively block out the incoming serial commands and render the ModBot uncontrollable. To address this problem, we used another serial port on the Arduino Mega to send the hall sensor readings. The Arduino Mega has three other serial ports, so we decided to use Serial1 located on pins 19 (RX) and 18 (TX). Thus, we wired the TX pin to the RX line of a USB plug and inserted the USB into one of the other COM ports located on our Atom board. After more extensive testing, we verified that the wheel speeds were accurately reported at a good frequency, and moreover, we verified that there are no missing command issues with the ModBot. Thus, we are now able to read the wheel velocity effectively from our base computer. From this data, we can effectively provide information that helps localize the location of the ModBot.

Although this give a very accurate speed at the wheel, there is slip between the wheel and the ground which has to be accounted for. Mecanum wheels such as the ones used on the ModBot operate with the assumption of slippage. Therefore, reading hall sensor velocity at the motor shaft does not give the speed of the robot. The figure below demonstrates the operation mode of Mecanum wheels.[10]



http://robomatter.com/Shop-By-Robot/VEX-Robots/Hardware/Mechanics/Mecanum-Wheel-4-4-pack

The main tracking and localization system used by the ModBot is the vision system; however, having an extra source of data is useful to improve position estimation.

5.1.3 Maxon Motor Results & Conclusion

The first step in solving this issue was to understand the operation of the old controller. The step was complicated because we were missing code from the previous implementation. This required writing new code on an Arduino using the old controller datasheet. We characterized the brake operation and understood how the controller should be wired and commanded.

This lead to the realization that the old controller was not capable of applying a negative torque in reference to the velocity of the motor. Such an operation is only possible with 4 quadrant controllers. We researched controller options and ordered new controllers. These new controllers have a controlled ramping functionality. The new controllers are mounted and interface with the motherboard.

We also interfaced with the hall sensors available on the motors and relayed that information back to the desktop computer. The system now works in a characterized manner and we are better suited to make autonomous path planning decisions.

[10] Please see the Appendix for a complete version of the motor controller source code

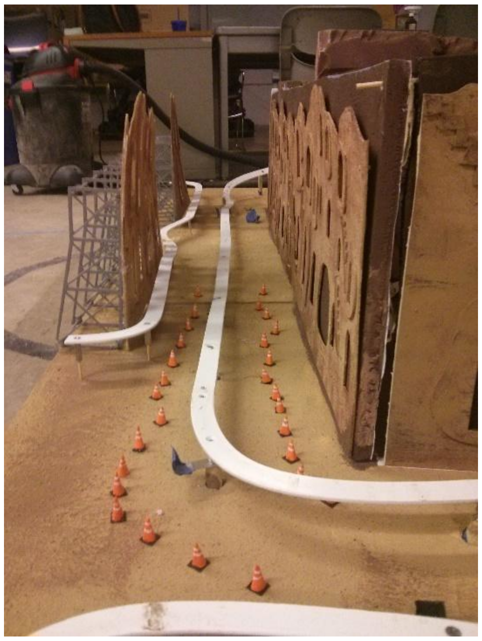
6 Pod-Racer Simulator Based on Small Scale Models

The task for this year's Intel-Cornell Cup demonstration was to create a model Disney roller coaster with a focus on prototype simulation. The miniature roller coaster sends data to a Moog motion generator to mimic the gravitational forces and experience of a real roller coaster. A precise control and sensor system was designed by the ECE team to verify the forces on this coaster.

6.1 High Level Design

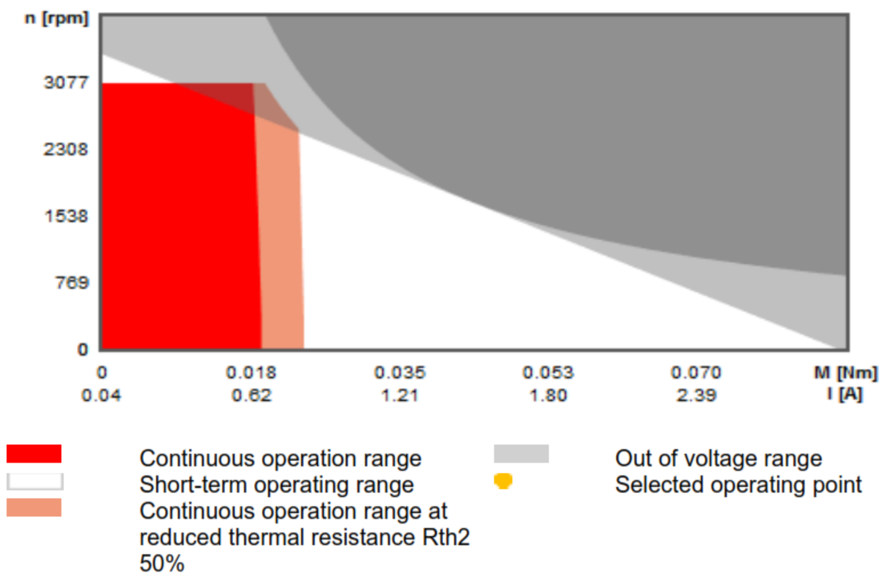
The purpose of this project was to showcase the new Intel Edison board on a roller coaster model ride. The ride emulates a Star Wars themed park as the pod loops through different scenes from the movie. The utility of the small scale model is twofold. First, it gives viewers a bird's eye view of the whole park and system. Second, a camera mounted to the front of the cart allows users to follow the ride as though they are actually ridding it. This video is saved and displayed on an oculus rift for users to experience the ride. The pod is designed to follow each section of track at the appropriate speed for the part. There is a photogate calibration system onboard which recalibrates the pod during the run. In addition, sensor data is collected from model track segments to provide data for actuation of the user's seat using a Moog Motion generator.

The following photographs depict sections of the small-scale ride track. The ECE team did not focus on the track but rather the pod, so discussion will be limited to this component.



The whole small pod system was designed around a small but very powerful Maxon motor. The motor was designed to be provide realistic acceleration and deceleration for the small pod in order to have the camera film a believable video. Since the track is scaled down from a full size coaster, we also scaled the gravity by value by the same amount. Therefore, our pod does not provide equal acceleration but rather a scaled value of the necessary value.

The motor is equipped with a 3.9:1 single stage gearhead that we use to decrease the shaft speed and increase the torque. It is also equipped with a 256 count encoder that the controller uses to monitor the motor position and close the feedback loop. We budgeted the motor with a nominal voltage of 12V due to the limit of our battery. Although a higher voltage battery would be able to provide faster speed and higher torque, this was not necessary for our application. Furthermore, limiting the battery weight and volume was a significant concern on the team. The following graph displays the operating characteristics our motor at various currents.



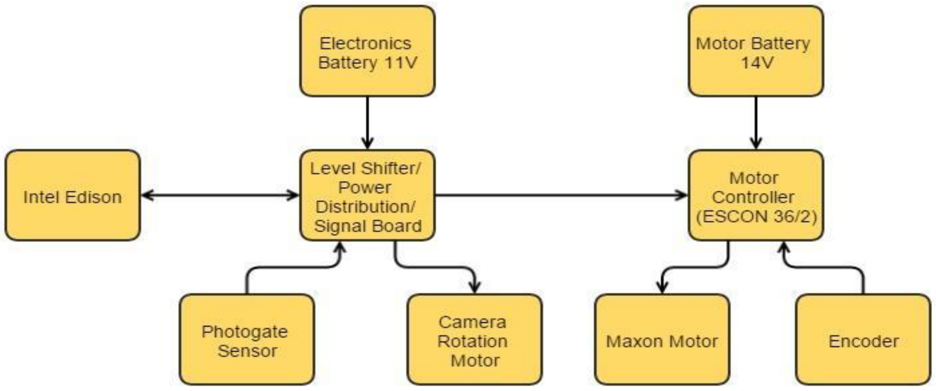
The encoder on this motor is a 256 count 3 channel encoder. We had the option to select encoders with higher resolutions; however, at the speed that we are operating, a higher resolution encoder would likely have led to hysteresis.

The controller used to operate this system is an ESCON 36/2 DC brushed motor controller. It is specifically designed to work with this line of Maxon motors and is able to form a closed loop feedback control with the motor and encoder.

The controller has configurable and static I/O pins which we used for the following purpose:

- Battery power (14V)
- Motor out (power/ground)
- Encoder input (from motor)
- Digital I/O to/from Intel Edison
- Analog I/O to/from Intel Edison

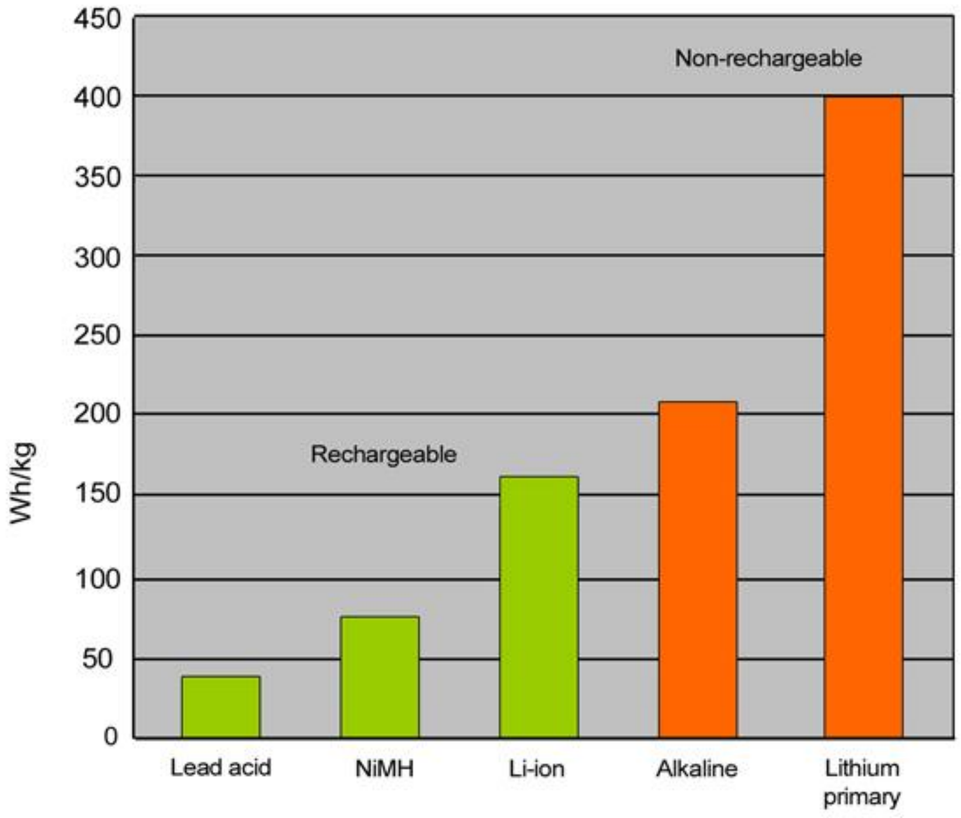
The diagram below shows the component connections on the small scale pod.



6.2 Power System

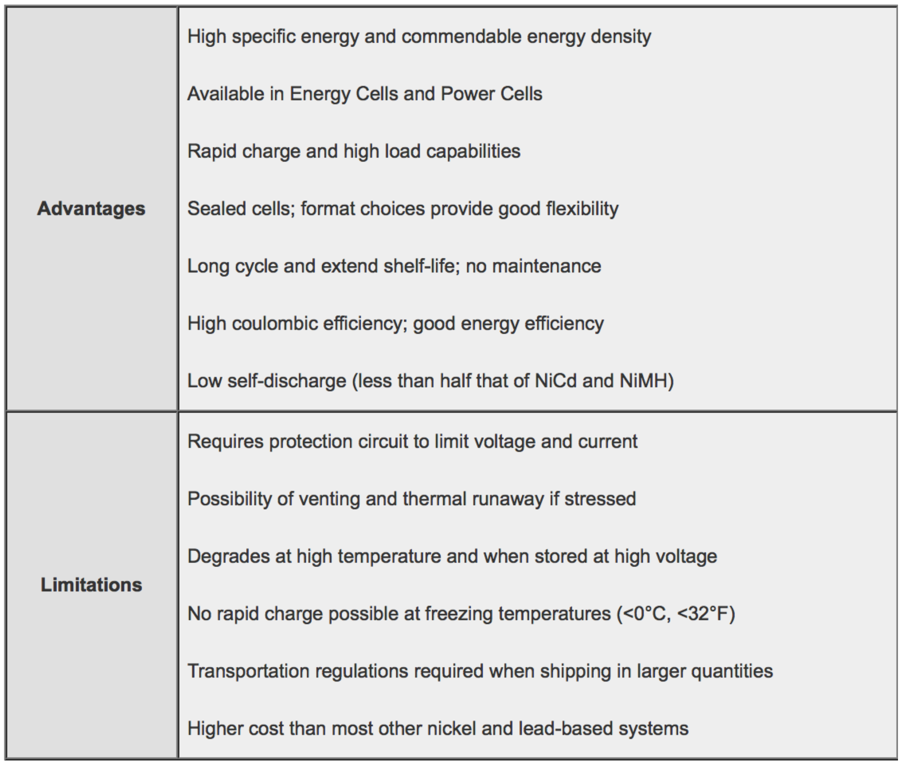
While we originally planned to utilize a bus-bar system to power the pod (See documentation for Fall 2014), that system proved to be difficult to transport, and as a result, the adjustments we would have to make would impair the reliability of our pod. For example, we would effectively lose all power to the pod if the skates on the pod skipped off of the track for a short duration, which is not desirable. While we considered an RC circuit to provide power to the pod as redundancy in the event of power loss, this method was cumbersome and not as effective as simply using batteries. Thus, we decided to rely on battery power for our pod. This method was deemed to be the most reliable, and in the event of a power failure, was easily addressed by simply replacing the battery.

However, we had to ensure that voltage spikes generated by the motor did not interfere with our Intel Edison and sensor circuits. This is because motors can create large voltage spikes when stopping, and accordingly damage sensitive circuitry like the Edison. As a result, we used two separate batteries (and thus effectively two separate circuits). Furthermore, the Intel Edison uses 1.8V logic which requires a level shifter to integrate with the motor controller and camera motor. The camera rotation motor is used to tilt the camera in the direction that the rider would be looking. It is operated with a PWM signal (as well as a power supply). Several of the components on our system use a 5V supply, so the 11V battery is converted to 5V on the board. We then used a 14.8V battery to power the motor controller as a separate circuit.



Comparative Energy Densities of Different Batteries

For our application, we decided to use Lithium-Ion batteries. Our small pod was very constrained by size, so every component had to be minimized in size. Thus, we required a battery that had a high energy density. Furthermore, we needed a battery that was rechargable. From the chart above comparing energy densities, we evaluated that Lithium Ion batteries were the most effective for our application. As shown in the chart below, the only major disadvantage of lithium-ion batteries is that if not properly handled, Lithium-Ion batteries can fail catastrophically and explode. Thus **it is always necessary to have a protection circuit on an Lithium-Ion battery and never leave a Lithium-Ion battery charging unattended.**



Advantages	High specific energy and commendable energy density Available in Energy Cells and Power Cells Rapid charge and high load capabilities Sealed cells; format choices provide good flexibility Long cycle and extend shelf-life; no maintenance High coulombic efficiency; good energy efficiency Low self-discharge (less than half that of NiCd and NiMH)
Limitations	Requires protection circuit to limit voltage and current Possibility of venting and thermal runaway if stressed Degrades at high temperature and when stored at high voltage No rapid charge possible at freezing temperatures (<0°C, <32°F) Transportation regulations required when shipping in larger quantities Higher cost than most other nickel and lead-based systems

Advantages and Disadvantages of Li-Ion Batteries

For our application, we used a 3S and a 4S battery. Lithium Ion batteries come in voltages that are multiples of 3.7V. This is because each cell has a 3.7V rating, and by wiring separate cells in series, we are able to increase the voltage of the battery pack (hence the name 3Series). Thus, our 3S pack had a voltage of 11.1V, and our 4S pack had a voltage of 14.8V. Furthermore, each battery pack is rated for a charge capacity in mAh, or milliamp-hours. Effectively, this is the current that the battery can provide in one hour. For example, our batteries were rated at 750mAh.

Thus, it is important to spec a battery according to your intended application. In our case, the 750mAh was more than enough to run our motor (which we evaluated as needing ~1A) and the rest of our circuitry (~700mA) for many runs through our track. Furthermore, it is critical to select a battery that is suited to discharge enough current to suit our load. If a battery is not rated to supply a high enough current, then our components will not get enough power to function properly, and moreover, the battery may overheat. In this case, Lithium Ion batteries are rated in terms of C. To evaluate the discharge current supplied by a battery, simply multiply the number in front of the C (e.g. 5C) by the maximum charge capacity of the battery. If our battery was 4C, and the capacity is 750mAh, the maximum discharge current is 3A. Furthermore, it is important to note that some batteries have a continuous discharge current (the current which a battery can continuously discharge) and a maximum discharge current (the current the battery can supply in small peaks). In our case, our batteries were limited to 4A discharge current by the protection circuitry.

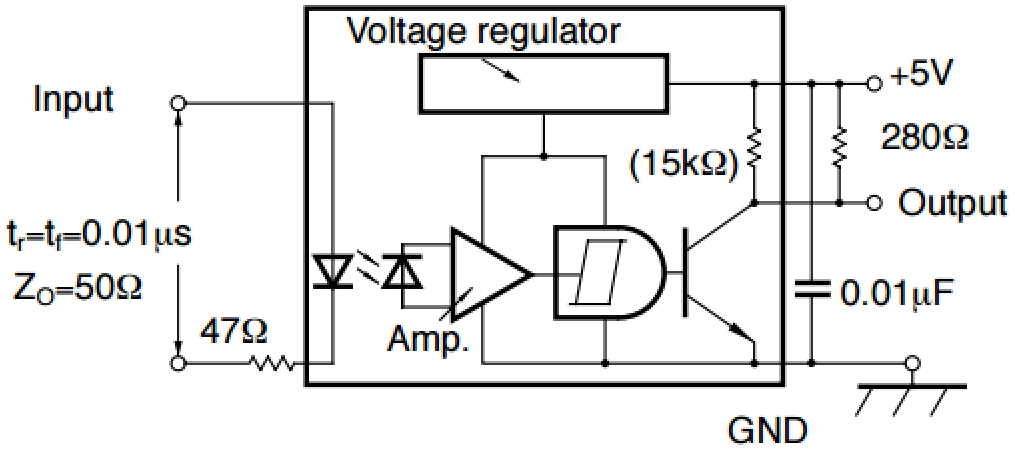
Once we had all the specifications of our battery determined (i.e. how much charge we need, the voltage we need, how much current will the circuit draw, how long will the battery discharge current, and the size constraints), we ordered custom batteries from *batteryspace.com.* We decided to use custom batteries because we are able to minimize the size of each battery if we customize the specifications to suit our needs.

Finally, as a backup system, we ordered several 1S Lithium-Polymer batteries from Sparkfun (https://www.sparkfun.com/products/10718). We were able to wire up these batteries in series to form a 3S battery pack. However, it is critical that we ensure that each battery has roughly the same voltage (within .1V), and furthermore, the batteries output a similar current across a 300 Ohm resistor. By checking these values, we know that we are able to charge this battery pack equally, as each battery has a similar impedance. Thus, by wiring up the batteries in series, and supplying a 12V charge across the terminals, we know that the voltages divide equally between all batteries. **However, it is important that the batteries are monitored while charging**, and that current is limited. Moreover, it is highly recommended to use an official balance charger that balances the voltage between each cell equally. While it is better to have professionals create a battery pack, our batteries did contain protection circuits. Furthermore, since our series packs were backups in the event of a main-battery failure, we deemed them suitable for our needs.

Thus, we have an effective means to power our small pod and multiple backups in the event of a battery failure.

6.3 Localization

A photogate interrupt sensor is used to detect the position of the cart on the track and recalibrate it. It works by sending an infrared signal between two LEDs. An event is triggered when the signal is broken. The output is pulled high meaning that it is normally 5V unless there is a trigger event. In this case, the output is pulled to ground. We utilized the photogate system on our small-scale pod as a means to localize the pod on the track. As the pod passed through different points on the track, a physical object would break the signal in the photogate. Thus, by counting the number of signals (aka checkpoints) passed along the track, the pod would know exactly where its location is along the track.

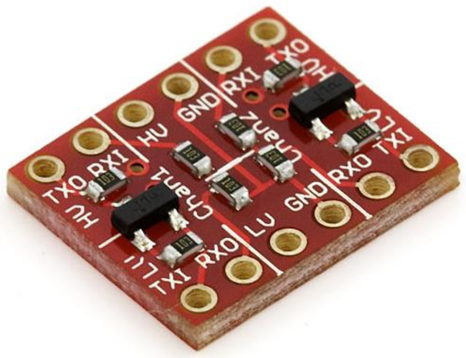


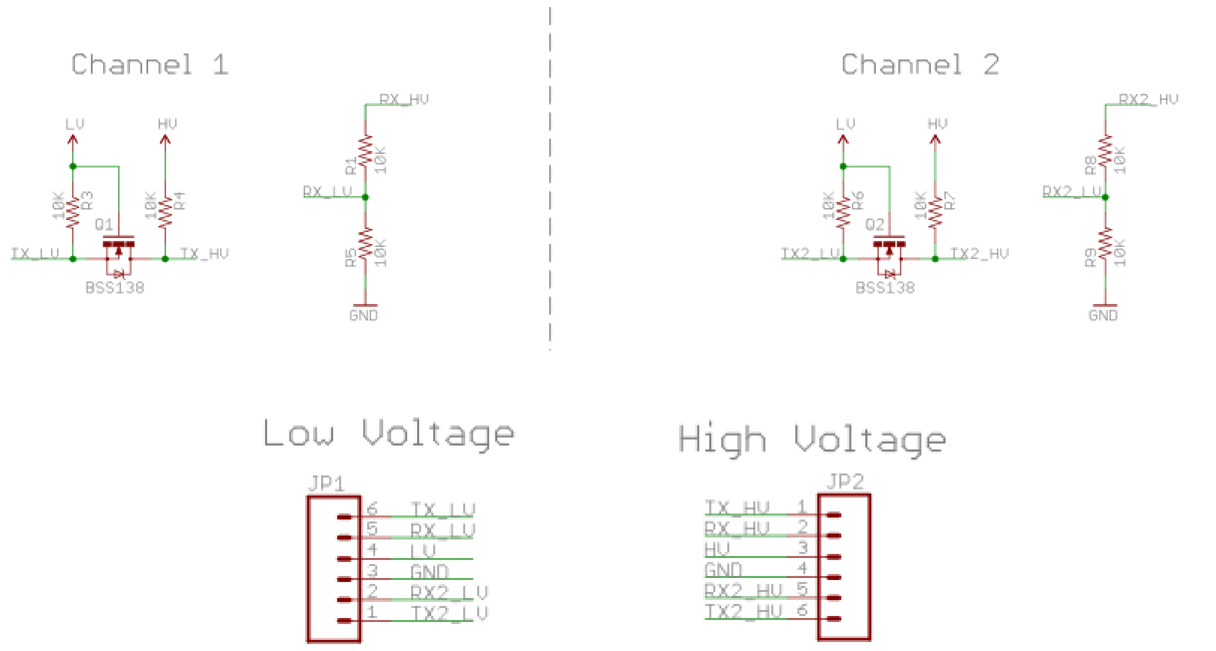
A Circuit Schematic for the Photogate System

6.4 Intel Edison for Motor Control and Photo-gate Operation

The Pod requires many combined aspects of the Edison mentioned above, in section 3—Intel Edison (pg. **Error! Bookmark not defined.**).

The Edison mini breakout board is used here. Care must be taken in dealing with the IO since it operates at 1.8V logic. Level translation is performed using the Sparkfun BOB-08745 [1]. The 1.8V to 5V logic works flawlessly and requires only to input the Edison signal on the 1.8V side, taking the output from the 5V side. The 5V to 1.8V step down is "hacked". The level translator step down only halves the voltage. So, it would step down the 5V to 2.5V. However, by adding a series resistor, this is changed to 1.8V. The BOB-08745 is pictured below (Reference [1]) along with details (Reference [2]).





TXI = TX_LV, TX2_LV
RXO = RX_HV, RX2_HV
TXO = TX_HV, TX2_HV
RXI = RX_HV, RX2_HV

The MRAA PWM and IO are used for motor control. The external interrupt is used for detecting a photogate input. The timer is used to update the speed according to a speed profile.

A speed profile file is stored locally on the Edison. The file is a txt file transferred to the Edison through SCP using WinSCP on Windows or SCP on Linux. It contains information about the speed (corresponding to PWM duty cycle) at 500 millisecond intervals. Thus, the timer is used to "interrupt" every half a second and update the speed from the speed profile. The values are read in to an array at program startup. Here are the first few lines of the speed profile:

0	0.00057107
0.50004	0.067129
1.0001	0.09903
1.5001	0.11431
2.0002	0.12163
2.5002	0.12512
3.0003	0.12679

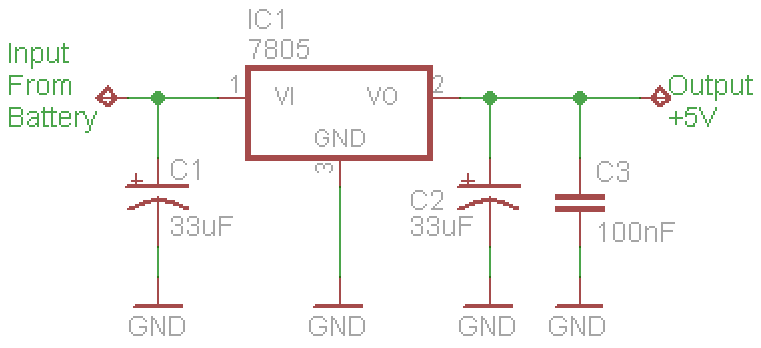
3.5003	0.12751
4.0003	0.12785
4.5004	0.12611
5.0004	0.12589
5.5005	0.12793
6.0005	0.12824
6.5005	0.12832
7.0006	0.12823
7.5006	0.12784

Unlike the mini-Modbot, the PWM does not require H-bridge drive. As previously mentioned, a motor controller is used to drive the single motor on the pod.

No direction control is required. However, there is a controller enable through a GPIO pin. The motor controller requires an analog input with the voltage corresponding to the speed of the motor. The PWM output from the Edison is level translated to 5V and then it is filtered to give an analog output that is fed to the controller.

The photogate output is translated down to 1.8V logic for the Edison and is configured to interrupt on a falling edge (using IO interrupt as mentioned above). The falling edge is when the photogate is interrupted with an object. When a photogate interrupt occurs, the speed profile is adjusted. There is another text file stored on the Edison that corresponds to the index of the speed profile to revert to when the photogate interrupt occurs. This is used for speed correction since the speed profile timing is essentially open-loop. Thus, the photogate input acts as feedback to the speed control.

The 1.8V supply is taken from the Edison breakout board header. The 5V supply is obtained by stepping down the battery voltage using a linear regulator (7805) as shown below.



The Edison connections are point-to-point connections, where the Edison connections are:

- PWM pin – J18-7
- Motor enable pin – J17-11
- Photogate interrupt input pin – J17-14
- Vin (5V) – J
- 1.8V – J19-2
- Ground – J19-3
- The Edison pinouts are described in (Reference [3]).

References:

[1] https://www.sparkfun.com/products/retired/8745

[2]https://www.sparkfun.com/datasheets/BreakoutBoards/Level-Converter-v10.pdf

[3] http://iotdk.intel.com/docs/master/mraa/edison.html

6.5 Sensors for Actuating the Moog Motion Generator

6.5.1 High Level Design

Before sending sensor data to the Moog Motion Generator to actuate a user, we needed to verify that the forces experienced by these track sections are what we would predict in the simulations, as part of ensuring the safety of our design. We created two separate systems, one on the Intel Edison shown in figure 14, and one on the Arduino Uno shown in figure 15, to exploit the strengths of each platform. The orientation calculation code was provided by Peter Bartz and extended by us to work with the Intel Edison drivers, to compensate accelerometer readings against gravity, and to save orientation and acceleration data to an SD card.

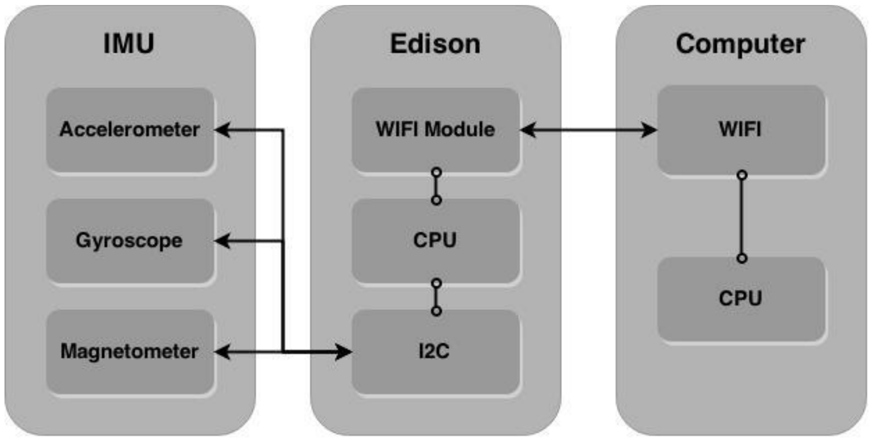


Fig. 14. Block diagram showing high level architecture of Edison system

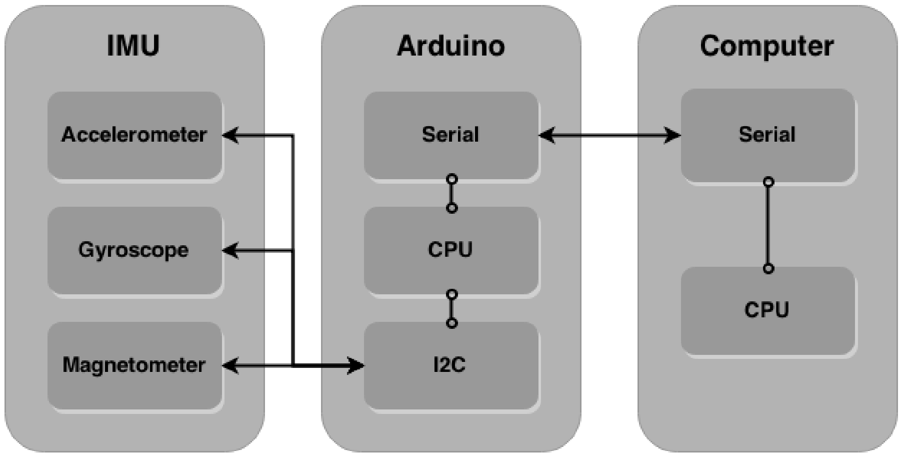


Fig. 15. Block diagram showing high level architecture of Arduino system

The 9DOF Razor IMU incorporates three sensors - an ITG-3200 (MEMS triple-axis gyro), ADXL345 (triple-axis accelerometer), and HMC5883L (triple-axis magnetometer) - to give nine degrees of inertial measurement. The outputs of all sensors are processed by an on-board CPU, and output to a base station via either serial in the Arduino system or WiFi in the Edison system.

The sensor conditioning code takes orientation data from the gyroscope, and uses magnetometer and accelerometer data for drift correction. More information on the sensor conditioning and drift correction can be found in this paper. The orientation data is then used to remove the gravity component from the accelerometer data using the system of equations described in equations 1-3.

$$A_{compx} = A_x + \sin(pitch)$$

Equation 1: Gravity compensation of x acceleration vector

$$A_{compy} = A_y - \sin(roll)*\cos(pitch)$$

Equation 2: Gravity compensation of y acceleration vector

$$A_{compz} = A_z - \cos(roll)*\cos(pitch)$$

Equation 3: Gravity compensation of z acceleration vector

The final Arduino setup we used for testing and in the results below is shown in figure 16.

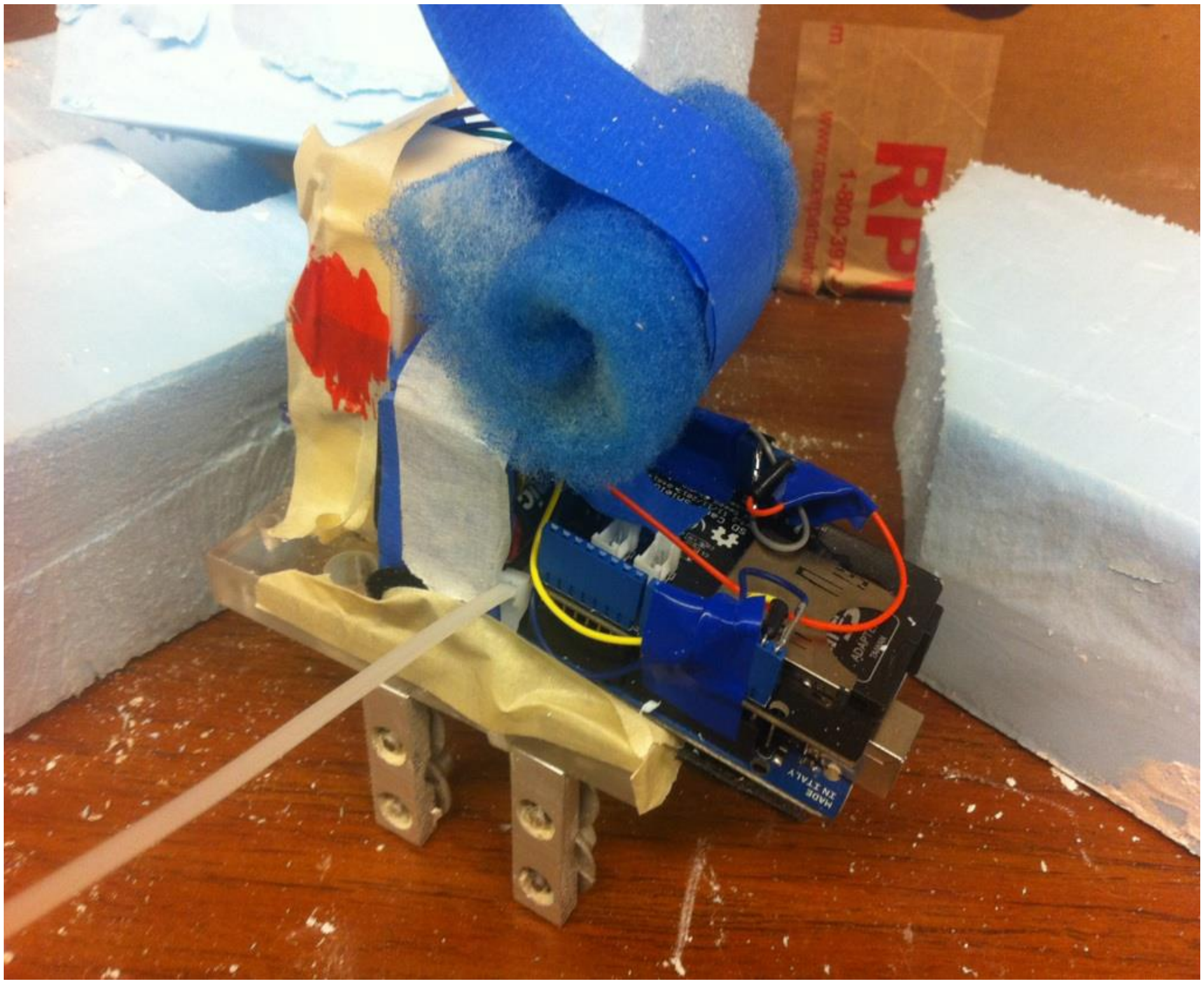


Figure 16: Final Arduino setup used to collect data

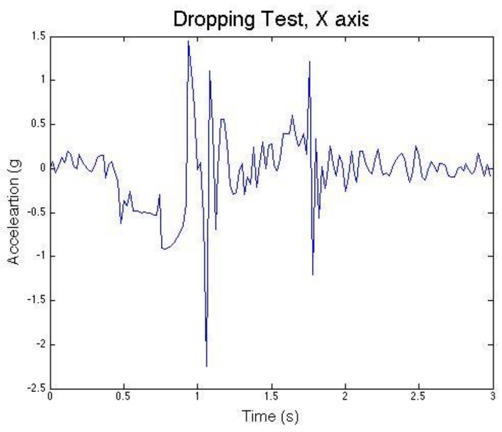


Fig. 17. Drop test X axis results

6.5.2 Results

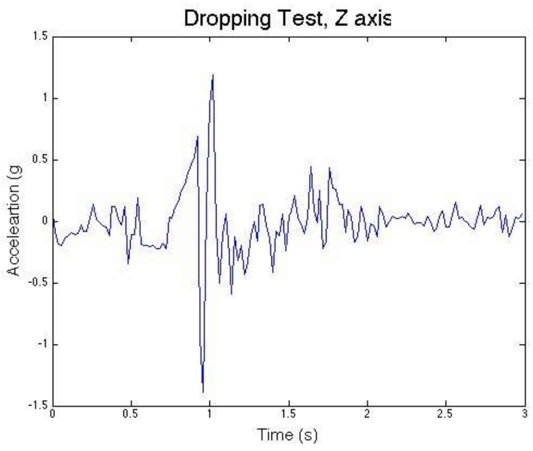


Fig. 18. Drop test Y axis results

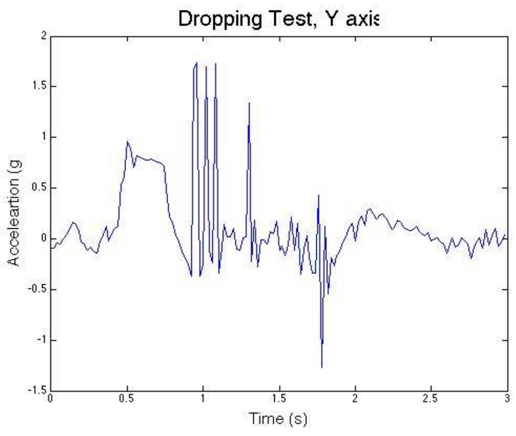


Fig. 19. Drop test Z axis results

We were not able to use the Edison to collect samples with the IMU quickly enough to make accurate orientation readings. However, with the Edison we were able to transmit data to another Linux computer via netcat. Since the Edison was too slow to make orientation readings, we switched the project to Arduino. In Arduino we were able to get good gravity compensated acceleration data and orientation data. In figures 17-19 the X, Y, and Z accelerometer data from the a simple drop test are shown. We dropped the pod setup shown in figure 3. The Y axis was pointing down, the X axis was pointing right relative to the pod, and the Z axis was pointing towards the front of the pod. In the Y axis from the .5 to .75s mark, a step response is visible with a peak of 1g, which is what we would expect from a simple drop. In the X axis, in the .5 to .75s period we see a step response with a peak of -.5g. The Z axis shows minor fluctuations until the .75s point. This is most likely due to the XZ plane of the accelerometer not being perfectly parallel to the ground plane. This also explains why the Y axis peaks at 1g briefly but never reaches that high point again during the drop phase. At the .75s mark, the pod collided with the surface and bounced back up, and also highlighted one of the issues with our
accelerometer: it is very sensitive to vibration. At the 1.75s mark, a second collision occurred. The pod settled back to steady state at the 2.5s mark.

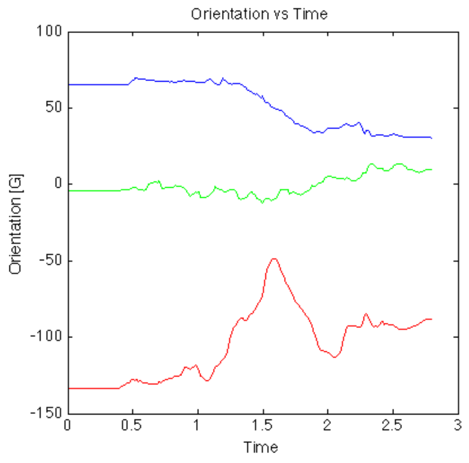


Fig. 20. Orientation v.s. Time average over 20 Trials on medium scale track

Note: orientation units should be [°] not [G]

Figures 21-23 show the results of the accelerometer and Figure 20 of the orientation data for the pod on the medium scale track section averaged over 20 trials. When the pod is in the default position, the roll angle is 0 degrees, the pitch angle is -90 degrees, and the yaw angle is relative to magnetic north. The coaster starts on the track soon after the ~.3-.4s mark. The pitch follows the track section as we would expect, and the roll and yaw change due to the instability of the track. The acceleration data is very noisy, again since the track is not perfectly stiff and the accelerometer is very sensitive to vibration. We tried to surround the accelerometer in foam to dampen the effect of vibration but that did not work. Lowpass filtering the data did not help either. In order to at least determine the safety of the coaster, we ran the data through a 7-point running average filter and then took the euclidean distance of the X, Y, and Z data at every point, the results of which are shown in figure 24. The acceleration values after this filtering do not exceed 1.5g, and below filtering none of the values exceed 3g, both well below the 6g limit that causes discomfort and grey out in populations with certain illnesses[1] and thus the coaster is safe to ride.

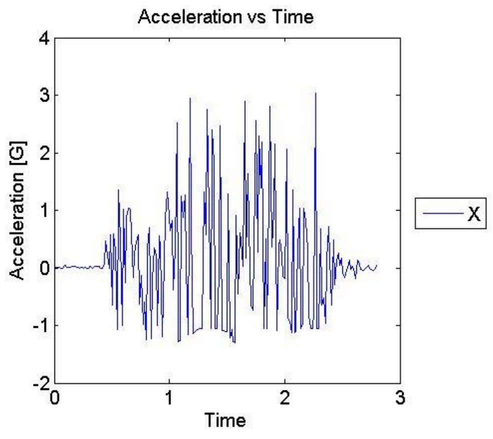


Fig. 21. Acceleration vs. time results in X direction averaged across 20 track trials

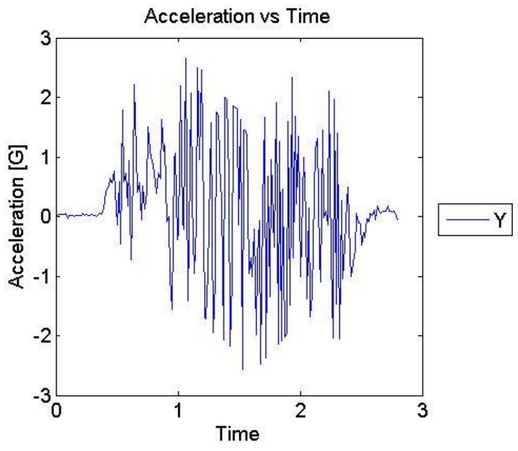


Fig. 22. Acceleration vs. time results in Y direction averaged across 20 track trials

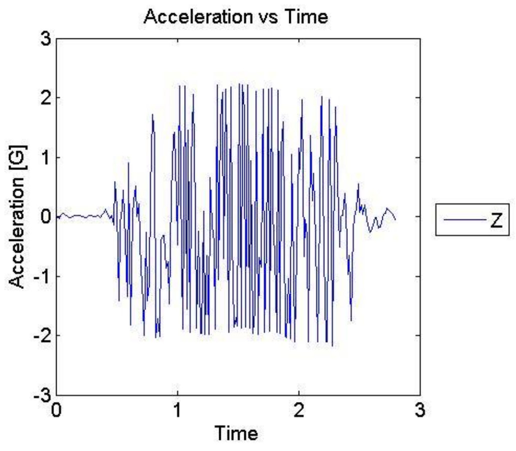


Fig. 23. Acceleration vs. time results in Z direction averaged across 20 track trials

6.5.3 Conclusions

Currently, the only way to access the I2C pins on the Edison is via file operations. Once the MRAA I2C libraries on the Edison are completely fixed, which we will have to wait on Intel to finish, we will easily be able to use the Edison for orientation and acceleration data collection. The MRAA libraries allow one to access the hardware on the Edison without the latency of having to go through multiple layers of abstraction like the file operations method. In addition, since the Edison has a smaller profile than Arduino and allows for easy wireless information transmission, it ultimately will better serve the needs of the roller coaster subteam.

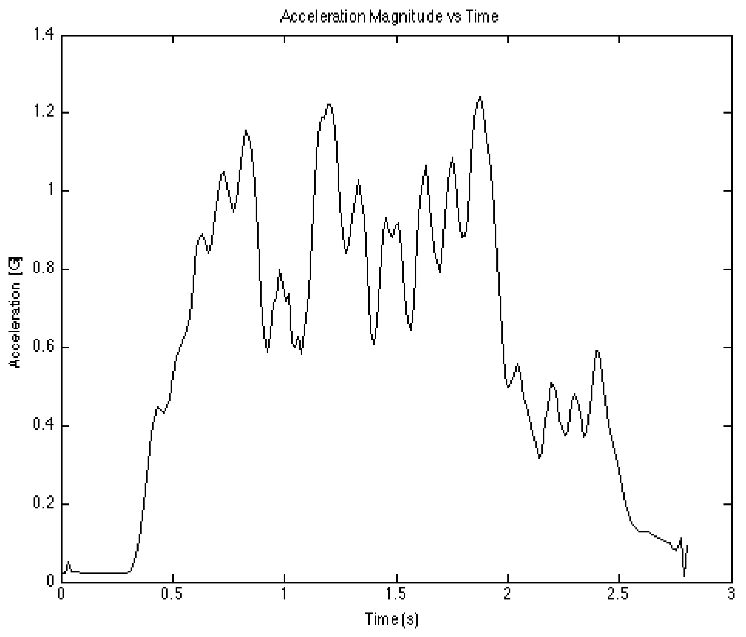


Fig. 24. Lowpass-filtered Euclidean distance of X, Y, Z components of acceleration gathered from average of 20 track trials

The accelerometer is guaranteed by the datasheet to have a certain sensitivity, and our preliminary tests confirm the code works as expected. The gyroscope and magnetometer are being used by the code as expected as well since the orientation data of the trial track runs matches our expectations. However, we must ensure that our application will be free of high-frequency vibrations since this increases the SNR to the point of being unable to extract useful data. For future testing, we must either ensure our track is stiffer, create a good shock absorption system to shield the accelerometer from vibration, or purchase a different accelerometer that is more tolerant of vibrations.

7 Appendicies

7.1 Intel Edison Code

7.1.1 Control of the mini modbot

7.1.1.1 Main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "ps4.h"
#include "motor_control.h"

#define threshold_x    2000
#define threshold_y    2000

mraa_gpio_context gpio2, gpio4;

int velocity=0;//+ve is forward, -ve is reverse
int theta=0;

void LEDs (int x, int y)
{
        if (x != 0)
        {
                mraa_gpio_write(gpio2, 1);
        }
        else
        {
                mraa_gpio_write(gpio2, 0);
        }

        if (y != 0)
        {
                mraa_gpio_write(gpio4, 1);
        }
        else
        {
                mraa_gpio_write(gpio4, 0);
        }
}

void motorLogic (int x, int y, int btn)
{
        float speed;
        float angle;

        if (y == -1)            // DPAD up
        {
                if (velocity >= 0)
                        velocity++;
                if (velocity < 0)
                        velocity += 50;

                if (velocity > threshold_y)
```

```
                        velocity = threshold_y;
        }
        else if (y == 1)        // DPAD down
        {
                if (velocity > 0)
                        velocity -=50;
                if (velocity <= 0)
                        velocity --;

                if (velocity < -threshold_y)
                        velocity = -threshold_y;
        }
        else if (y == 0)
        {
                if (velocity > 0)
                {
                        if (--velocity < 0)
                        {
                                velocity = 0;
                        }
                }
                else if (velocity < 0)
                {
                        if (++velocity > 0)
                        {
                                velocity = 0;
                        }
                }
        }

        if (x == -1)
        {
                if (theta > 0)
                        theta --;
                if (theta <= 0)
                        theta --;

                if (theta < -threshold_x)
                        theta = -threshold_x;
        }
        else if (x == 1)
        {
                if (theta >= 0)
                        theta ++;
                if (theta < 0)
                        theta ++;

                if (theta > threshold_x)
                        theta = threshold_x;
        }
        else if (x == 0)
        {
                theta=0;
        }
        speed = ((float)velocity/(float)threshold_y);
        //printf("%d %d %.2f\n", velocity, threshold_y, speed);
        // between -1 and 1

        if (velocity > 0)
        {
                goForward(speed);
        }
        else if (velocity < 0)
        {
                goReverse(-speed);
        }
        else if (velocity == 0)
        {
                goForward(0);
        }

        if (btn == BTN_CROSS)
        {
                stopCar();
        }

//      angle = ((float)theta/(float)threshold_x);
        angle = 0.95;

        if (theta > 0)
        {
//              goForward(angle);
                turnRight(angle, angle); // 0, angle
        }
        else if (theta < 0)
        {
                //goReverse(-angle);
                turnLeft(angle, angle); // angle, 0
        }
        else if (theta == 0)
        {
                //goForward(0);
        }


}

void main (void)
{
// Assume that PS4 controller is already paired and connected
        gp_init();      // Initialize gamepad control
        motor_control_init();

        int x=0, y=0, btn=0;


        gpio2 = mraa_gpio_init(2);
        gpio4 = mraa_gpio_init(4);

        mraa_gpio_dir(gpio2, MRAA_GPIO_OUT);
        mraa_gpio_dir(gpio4, MRAA_GPIO_OUT);
        while (1)
        {
                gp_getEvent(); // Event handling
                get_joystickEvent(&x, &y, &btn);

                motorLogic(x,y,btn);
                LEDs(x,y);

        }
        gp_quit();
}
```

7.1.1.2 Motor_control.c

```
/*
Sample Code to run the Sparkfun TB6612FNG 1A Dual Motor Driver using Arduino
UNO R3

 This code conducts a few simple manoeuvres to illustrate the functions:
 - motorDrive(motorNumber, motorDirection, motorSpeed)
 - motorBrake(motorNumber)
 - motorStop(motorNumber)
 - motorsStandby

 Connections:
 - Pin 3 ---> PWMA
 - Pin 8 ---> AIN2
 - Pin 9 ---> AIN1
 - Pin 10 ---> STBY
 - Pin 11 ---> BIN1
 - Pin 12 ---> BIN2
 - Pin 5 ---> PWMB

 - Motor 1: A01 and A02
 - Motor 2: B01 and B02
*/

#include "motor_control.h"
#include "mraa.h"

void motor_control_init(void)
{
        pin_ain1 = mraa_gpio_init(pinAIN1);
        if (pin_ain1 == NULL) fprintf(stderr, "Couldn't open AIN1\n");

        pin_ain2 = mraa_gpio_init(pinAIN2);
        if (pin_ain2 == NULL) fprintf(stderr, "Couldn't open AIN2\n");

        pin_pwma = mraa_pwm_init(pinPWMA);
        if (pin_pwma == NULL) fprintf(stderr, "Couldn't open PWMA\n");

        pin_bin1 = mraa_gpio_init(pinBIN1);
        if (pin_bin1 == NULL) fprintf(stderr, "Couldn't open BIN1\n");
        pin_bin2 = mraa_gpio_init(pinBIN2);
        if (pin_bin2 == NULL) fprintf(stderr, "Couldn't open BIN2\n");
        pin_pwmb = mraa_pwm_init(pinPWMB);
        if (pin_pwmb == NULL) fprintf(stderr, "Couldn't open PWMB\n");

        mraa_gpio_dir(pin_ain1, MRAA_GPIO_OUT);
        mraa_gpio_dir(pin_ain2, MRAA_GPIO_OUT);

        mraa_gpio_dir(pin_bin1, MRAA_GPIO_OUT);
        mraa_gpio_dir(pin_bin2, MRAA_GPIO_OUT);

        pin_stby = mraa_gpio_init(pinSTBY);
        mraa_gpio_dir(pin_stby, MRAA_GPIO_OUT);

        mraa_pwm_period_us(pin_pwma, 1500);    // ~660Hz
        mraa_pwm_period_us(pin_pwmb, 1500);

        mraa_pwm_enable(pin_pwma, 1);
        mraa_pwm_enable(pin_pwmb, 1);

        mraa_pwm_write(pin_pwma, 0);           // 0% initial duty cycle
        mraa_pwm_write(pin_pwmb, 0);
}

void goForward(float motorSpeed)
{
  motorDrive(motor1, turnCCW, motorSpeed);
//motor1 is slightly faster - fix later
  motorDrive(motor2, turnCW, motorSpeed);
}

void goReverse(float motorSpeed)
{
  motorDrive(motor1, turnCW, motorSpeed);
//motor1 is slightly faster - fix later
  motorDrive(motor2, turnCCW, motorSpeed);
}

void turnRight(float motor1Speed, float motor2Speed)
{
  motorDrive(motor1, turnCCW, motor1Speed);
  motorDrive(motor2, turnCCW, motor2Speed);//CW
}

void turnLeft(float motor1Speed, float motor2Speed)
{
  motorDrive(motor1, turnCW, motor1Speed);//CCW
  motorDrive(motor2, turnCW, motor2Speed);
}

void stopCar(void)
{
  //Apply brakes, then standby
  motorBrake(motor1);
  motorBrake(motor2);
  motorsStandby();
```

```
}

int getMotorSpeed(char motorNumber)
{
/*
  if (motorNumber == motor1)
    return analogRead(pinPWMA);
  else
    return analogRead(pinPWMB);
*/
}

//============================================================================

void motorDrive(char motorNumber, char motorDirection, float motorSpeed)
{
  /*
  This Drives a specified motor, in a specific direction, at a specified
speed:
   - motorNumber: motor1 or motor2 ---> Motor 1 or Motor 2
   - motorDirection: turnCW or turnCCW ---> clockwise or counter-clockwise
   - motorSpeed: 0 to 1 ---> 0 = stop / 1 = fast
   */

  char pinIn1;  //Relates to AIN1 or BIN1 (depending on the motor number
specified)


  //Specify the Direction to turn the motor
  //Clockwise: AIN1/BIN1 = HIGH and AIN2/BIN2 = LOW
  //Counter-Clockwise: AIN1/BIN1 = LOW and AIN2/BIN2 = HIGH
  /*
  if (motorDirection == turnCW)
    pinIn1 = HIGH;
  else
    pinIn1 = LOW;
  */
  pinIn1 = (motorDirection == turnCW);

  //Select the motor to turn, and set the direction and the speed
  if(motorNumber == motor1)
  {
        mraa_gpio_write(pin_ain1, pinIn1);
        mraa_gpio_write(pin_ain2, !pinIn1);
        mraa_pwm_write(pin_pwma, motorSpeed);
  }
  else
  {
        mraa_gpio_write(pin_bin1, pinIn1);
        mraa_gpio_write(pin_bin2, !pinIn1);
        mraa_pwm_write(pin_pwmb, motorSpeed);
  }

  //Finally , make sure STBY is disabled - pull it HIGH
  mraa_gpio_write(pin_stby, HIGH);

}
```

```
void motorBrake(char motorNumber)
{
  /*
This "Short Brake"s the specified motor, by setting speed to zero
   */

  if (motorNumber == motor1)
        mraa_pwm_write(pin_pwma, 0);
  else
    mraa_pwm_write(pin_pwmb, 0);

}


void motorStop(char motorNumber)
{
  /*
  This stops the specified motor by setting both IN pins to LOW
   */
  if (motorNumber == motor1) {
        mraa_gpio_write(pin_ain1, LOW);
        mraa_gpio_write(pin_ain2, LOW);
  }
  else
  {
        mraa_gpio_write(pin_bin1, LOW);
        mraa_gpio_write(pin_bin2, LOW);
  }
}


void motorsStandby(void)
{
  /*
  This puts the motors into Standby Mode
   */
  mraa_gpio_write(pin_stby, LOW);
}
```

7.1.1.3 Motor_control.h

```
#include "mraa.h"


#ifndef HIGH
        #define HIGH           1
#endif

#ifndef LOW
        #define LOW                    0
#endif

//Define the Pins
```

```
//Motor 1
#define pinAIN1         9       // Direction
#define pinAIN2         8       // Direction
#define pinPWMA         3       // Speed

mraa_gpio_context pin_ain1, pin_ain2;
mraa_pwm_context pin_pwma;

//Motor 2
#define pinBIN1         11      // Direction
#define pinBIN2         12      // Direction
#define pinPWMB         5       // Direction

mraa_gpio_context pin_bin1, pin_bin2;
mraa_pwm_context pin_pwmb;

//Standby
#define pinSTBY         10

mraa_gpio_context pin_stby;

//Constants to help remember the parameters
#define turnCW          0       //for motorDrive function
#define turnCCW         1       //for motorDrive function
#define motor1          0       //for motorDrive, motorStop, motorBrake
functions
#define motor2          1       //for motorDrive, motorStop, motorBrake
functions

#define leftMotor       motor1
#define rightMotor      motor2

void motor_control_init(void);
void goForward(float motorSpeed);
void goReverse(float motorSpeed);
void turnRight(float motor1Speed, float motor2Speed);
void turnLeft(float motor1Speed, float motor2Speed);
void stopCar(void);
int getMotorSpeed(char motorNumber);
void motorDrive(char motorNumber, char motorDirection, float motorSpeed);
void motorBrake(char motorNumber);
void motorStop(char motorNumber);
void motorsStandby(void);
```

7.1.1.4 PS4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>        // SUPER IMPORTANT - FOR EVENTS
#include "ps4.h"
```

```
const char* gamepadName = "/dev/input/event2";
volatile int gamepad = -1;
event_t joystickEvent = { ev_joystick };
int rcv = -1;
int upd_x = -1, upd_y = -1;

int jx=0,jy=0,jbtn=0;

int get_rcv(void)
{
        return rcv;
}

int get_updx(void)
{
        return upd_x;
}

int get_updy(void)
{
        return upd_y;
}

void get_joystickEvent(int* x, int* y, int* btn)
{
        *x = jx;
        *y = jy;
        *btn = jbtn;
}

void gp_postEvent(event_t joystickEvent)
{       // joystickEvent contains 3 ints: btn, x, y
        static int oldx=0, oldy=0;

        upd_x = -1;
        upd_y = -1;

        jx = joystickEvent.x;
        jy = joystickEvent.y;

        //if (joystickEvent.btn)
        //{
                jbtn = joystickEvent.btn;
        //}

        if (joystickEvent.x != oldx)
        {
                upd_x = 0;
        }

        if (joystickEvent.y != oldy)
        {
                upd_y = 0;
        }

        oldx = joystickEvent.x;
        oldy = joystickEvent.y;
```

```
}

void gp_init(void)
{
        do {
                gamepad = open(gamepadName, O_RDONLY | O_NONBLOCK);
                //if (gamepad < 0)
                //fprintf(stderr, "Cannot access gamepad at %s\n",
gamepadName);
        } while (gamepad < 0);
        //else{
                //fprintf(stdout, "Accessed gamepad at %s\n", gamepadName);
        //}

}

void gp_quit(void)
{
        if (gamepad >= 0)
                close(gamepad);
}


void gp_getEvent(void)
{
                struct input_event ev[64];
                joystickEvent.btn = 0;

                int rd = read(gamepad, ev, sizeof(ev));

                if (rd < (int) sizeof(struct input_event))
                {
                        rcv = -1;
                        return;
                }

                int i;
                for (i = 0; i < rd / (int)sizeof(struct input_event); i++)
                {
                        struct input_event e = ev[i];

                        switch (e.type)
                        {
                        case EV_ABS:
                                if (e.code == ABS_HAT0X)                //
Digital pins L and R
                                        joystickEvent.x = e.value;
                                else if (e.code == ABS_HAT0Y)  // Digital pins
U and D
                                        joystickEvent.y = e.value;
                                break;
                        case EV_KEY:
                                // Map keys (SEE HEADER FILE):
                                joystickEvent.btn = e.code;             //
Button pressed
                                break;
                        }
                        gp_postEvent(joystickEvent);
                        // Do something with this info
                        rcv = 0;
                }
}
```

7.1.1.5 PS4.h

```
#include "mraa.h"

typedef enum {false, true} boolean;
typedef unsigned char byte;

#define MAXCHAR         ((char)0x7f)
#define MAXSHORT        ((short)0x7fff)
// Max pos 32-bit int.
#define MAXINT          ((int)0x7fffffff)
#define MAXLONG         ((long)0x7fffffff)
#define MINCHAR         ((char)0x80)
#define MINSHORT        ((short)0x8000)

// Max negative 32-bit integer.
#define MININT          ((int)0x80000000)
#define MINLONG         ((long)0x80000000)

 #define BTN_SQUARE                0x130
 #define BTN_CROSS                 0x131
 #define BTN_CIRCLE                0x132
 #define BTN_TRIANGLE              0x133
 #define BTN_LB                    0x134
 #define BTN_RB                    0x135
 #define BTN_LT                    0x136
 #define BTN_RT                    0x137
 #define BTN_SHARE                 0x138
 #define BTN_OPTIONS               0x139
 #define BTN_LSTICK                0x13a
 #define BTN_RSTICK                0x13b
 #define BTN_PS                    0x13c
 #define BTN_TOUCHPAD              0x13d

#define MAXEVENTS              64

// Input event types
typedef enum
{
    ev_keydown,
    ev_keyup,
    ev_mouse,
    ev_joystick
} evtype_t;

// Event structure
typedef struct
{
        evtype_t        type;
        int                     btn;    // keys / mouse/joystick buttons
        int                     x;              // mouse/joystick x move
        int                     y;              // mouse/joystick y move
} event_t;

/*
  event_t                       events[MAXEVENTS];
  int                   eventhead;
  int                   eventtail;
*/

  mraa_pwm_context pwm0;
  mraa_pwm_context pwm1;
```

7.1.2 Control of the Small Pod

7.1.2.1 Main.c

```
/* GPIO external interrupt and PWM test program
 * Cornell Cup USA presented by Intel
 * Spring 2015
 * Programmed by: Syed Tahmid Mahbub
 */

#include "initializer.h"
#include <time.h>
#include "pwm.h"
#include "logging.h"
#include "timers.h"

 //#define CAMERA_RUN


//#define period 1515  // initial period
#define period 1000 // 1 kHz

//#define PRINT_DEBUG

mraa_gpio_context INT; // external interrupt
mraa_pwm_context PWM; // PWM output
mraa_gpio_context ME; // motor enable

logging_t time_log;
time_sns_t init_time;
time_sns_t time_struct, time_struct2;

// Speed profile
float speed[number_of_points];
unsigned int camera[number_of_points];

#define photogate_points 12
unsigned int photogate[photogate_points];

void INT_handler(void * args){
        // if (interrupt_enabled) {
                usleep(5000); // "debounce delay"

        // Ensure that the voltage level is correct - that the interrupt
        // wasn't just entered because of bouncing
                if (mraa_gpio_read(INT) == 0){

                        interrupt_enabled = 0; // disable interrupt

                        // Status update: show updated duty cycle
                        #ifdef PRINT_DEBUG
                                printf("Setting duty = %.2f\n",
speed[speed_index]);
                        #endif

                        // Update PWM duty cycle
                        speed_index = photogate[photogate_index++];

                        #ifdef PRINT_DEBUG
                                printf("Interrupt!\n");
                        #endif
                }
        // }
}

int main(void){

        int pwm_pin = 20; // J18-7
        int pwm_period = period; // this sets the initial period
        int int_pin = 13; // J17-14
        int motor_enable_pin = 10; // J17-11
        int camera_pin = 21; // J18-8

        float profile_data[number_of_points][2];

        int row, col, item;

        speed_index = 0;
        camera_index = 0;
        photogate_index = 0;

        interrupt_enabled = 1;
        int_counter = 0;

        //sleep(3);

        int i = 0;
        FILE *fp = fopen("vel_history.txt", "r");
        if (fp == NULL){
                        fprintf(stderr, "Could not open speed profile
file!\n");
                        return -1;
        }

        for(row=0; row<number_of_points; row++) {
                col = 0;
                item = fscanf(fp, "%f %f", &profile_data[row][col],
&profile_data[row][col+1]);
                speed[row]=profile_data[row][1];
        }
        if (fclose(fp)){
                fprintf(stderr, "Could not close speed profile file!\n");
        }

        #ifdef CAMERA_RUN
        // Camera stuff
        fp = fopen("yawangle_profile.txt","r");
        if (fp == NULL){
                fprintf(stderr, "Could not open yaw angle profile file!\n");
                return -1;
        }

        float camera_value;
        for(row=0; row<number_of_points; row++) {
                col = 0;
                item = fscanf(fp, "%f %f", &profile_data[row][col],
&profile_data[row][col+1]);
                camera_value = profile_data[row][1];
                // camera_value = 1800;
                //camera_value = (camera_value-1500.0)/500.0 * 900.0 +
1500.0;
                #ifdef PRINT_DEBUG
                        printf("%u\n", (unsigned int) camera_value);
                #endif
                camera[row] = (unsigned int)camera_value;
        }

        if (fclose(fp)){
                fprintf(stderr, "Could not close yaw angle profile
file!\n");
        }
        #endif

        fp = fopen("photogate_timeindices.txt","r");
        if (fp == NULL){
                fprintf(stderr, "Could not open photogate info!\n");
                return -1;
        }

        for (row=0; row<photogate_points; row++){
                fscanf(fp, "%u", &photogate[row]);
        }

        if (fclose(fp)){
                fprintf(stderr, "Could not close photogate info file\n");
        }

        mraa_gpio_context PWM_GPIO;

        if (mraa_failed(mraa_init())){
                fprintf(stderr, "MRAA failed to initialize!\n");
                return -1;
        }

/* Initialize GPIO for external interrupt with the following settings:
 *              int_pin = MRAA pin. Look at the pin mapping here:
 *                      http://iotdk.intel.com/docs/master/mraa/edison.html
 *              MRAA_GPIO_EDGE_FALLING -> interrupt on falling edge
 *              INT_handler -> interrupt handler for the external interrupt
 */
        INT = init_GPIO_INT(int_pin, MRAA_GPIO_EDGE_FALLING, &INT_handler);

/* Initialize GPIO for motor enable with the following settings:
 *              motor_enable_pin = MRAA pin. Look at the pin mapping here:
 *                      http://iotdk.intel.com/docs/master/mraa/edison.html
 * The pin defaults to 1 to enable the motor initially.
 */
        ME = init_GPIO_motorenable(motor_enable_pin);

/* Initialize PWM for motor control use with the following settings:
 *              pwm_pin = MRAA pin. Look at the pin mapping here:
 *                      http://iotdk.intel.com/docs/master/mraa/edison.html
 *              pwm_period = initial PWM period
 *              speed[0] -> initial duty cycle
 */
        PWM = init_PWM(pwm_pin, pwm_period, speed[0]);

        offset = -100;

        #ifdef CAMERA_RUN
        if (init_camera(camera_pin)){
                return -1;
        }
        #endif

        init_timer();

        while (1){
                if (speed_index >= 0 && speed_index < number_of_points){
                        mraa_pwm_write(PWM, speed[speed_index]);
                }
                #ifdef CAMERA_RUN
                if (camera_index >= 0 && camera_index < number_of_points) {
                        #ifdef PRINT_DEBUG
                                printf("Camera index = %d, value: %d\n",
(int)camera_index, (int)camera[camera_index]);
                        #endif
                        mraa_pwm_pulsewidth_us(PWM_camera,
camera[camera_index]+offset);
                }
                #endif

                if (getchar() == '0') break;
        }
// Close peripherals to exit
        motor_stop(PWM, pwm_pin, ME);
        close_timer();
        #ifdef CAMERA_RUN
                close_camera(camera_pin);
        #endif
// Done! Return successfully
        return 0;
}
```

7.1.2.2 Initializer.c

```
#include "initializer.h"

int mraa_failed(mraa_result_t result){
/* Check if MRAA function was successfully done
 * This is done when the result of an MRAA function
 * (mraa_result_t type) is one of the two:
 *              MRAA_SUCCESS
 *              MRAA_ERROR_PLATFORM_ALREADY_INITIALISED
 * This means that the operation was successfully done
 * or the aimed initialization has happend already.
 * -----------------------------------------------------
 * Return 0 if operation successfully.
 * Return -1 if operation failed.
 */
        if (result == MRAA_SUCCESS || result ==
MRAA_ERROR_PLATFORM_ALREADY_INITIALISED)
                return 0;
        else{
                return -1;
        }
}

mraa_gpio_context init_GPIO_INT(int gpio_pin, gpio_edge_t int_edge, void
handler (void*)) {
// gpio_pin = gpio pin that will be used
// int_edge = which edge to interrupt on
// handler = interrupt handler
// return the GPIO context upon success, NULL upon failure

        mraa_gpio_context gpio = mraa_gpio_init(gpio_pin);
        if (gpio == NULL) {
                fprintf(stderr, "Could not intialize GPIO on pin %d\n",
gpio_pin);
                return NULL;
        }

        if (mraa_failed(mraa_gpio_dir(gpio, MRAA_GPIO_IN))){
                fprintf(stderr, "Could not set GPIO pin to input on pin
%d\n", gpio_pin);
                return NULL;
        }

        if (mraa_failed(mraa_gpio_isr(gpio, int_edge, handler, NULL))){
                fprintf(stderr, "Failed to initialize interrupt for pin
%d\n", gpio_pin);
                return NULL;
        }

        return gpio;
}

mraa_pwm_context init_PWM(int pwm_pin, int period, float duty_init) {
// pwm_pin = pin on which to have PWM output
// period = period of PWM in microseconds
// duty_init = initial duty cycle to set (0.05 to 0.95 -> 5% to 95%)
// return the PWM context upon success, NULL upon failure

        mraa_pwm_context PWM = mraa_pwm_init(pwm_pin);

        if (PWM == NULL) {
                fprintf(stderr, "Could not initialize PWM on pin %d\n",
pwm_pin);
                return NULL;
        }

        if (period < 50) period = 50;

        if (mraa_failed(mraa_pwm_period_us(PWM, period))){
                fprintf(stderr, "Could not set period on pin %d\n",
pwm_pin);
                return NULL;
        }

        // Set bounds on duty cycle
        if (duty_init < 0.05) duty_init = 0.05;
        if (duty_init > 0.95) duty_init = 0.95;

        if (mraa_failed(mraa_pwm_write(PWM, duty_init))){
                fprintf(stderr, "Could not set duty cycle on pin %d\n",
pwm_pin);
                return NULL;
        }

        if (mraa_failed(mraa_pwm_enable(PWM, 1))) {
                fprintf(stderr, "Could not enable PWM on pin %d\n",
pwm_pin);
                return NULL;
        }

        return PWM;
}

int close_PWM_GPIO(mraa_pwm_context pwm, mraa_gpio_context gpio, int
pwm_pin) {
// return 0 upon successfully closing PWM and GPIO
// return -1 if PWM output couldn't be guaranteed to be 0

        mraa_pwm_write(pwm, 0);
        mraa_pwm_enable(pwm, 0);
        mraa_pwm_close(pwm);

/* When the PWM is disabled and closed, its value on the pin
 * stops at whatever value it was at that point. This could be
 * high or low. That may be a problem since it isn't necessarily
 * acceptable to have an output always be high. Thus, what is
 * needed is to manually drive the pin low. This is done by
 * switching from PWM to GPIO on that pin and driving it low asctime
 * an output.
 */
        mraa_gpio_context PWM_GPIO = mraa_gpio_init(pwm_pin);
        if (PWM_GPIO == NULL) {
                fprintf(stderr, "Could not initialize GPIO on pin %d\n",
PWM_GPIO);
                return -1;
        }

        if (mraa_failed(mraa_gpio_dir(PWM_GPIO, MRAA_GPIO_OUT))){
                fprintf(stderr, "Could not set GPIO pin to output on pin
%d\n", PWM_GPIO);
                return -1;
        }

        mraa_gpio_write(PWM_GPIO, 0);
        mraa_gpio_close(PWM_GPIO);

        // Stop interrupt watcher on GPIO and set interrupt
        // edge to none (MRAA_GPIO_EDGE_NONE)
        mraa_gpio_isr_exit(gpio);
        mraa_gpio_close(gpio);

        return 0;
}


mraa_gpio_context init_GPIO_motorenable(int gpio_pin) {
// gpio_pin = gpio pin that will be used
// return the GPIO context upon success, NULL upon failure

        mraa_gpio_context gpio = mraa_gpio_init(gpio_pin);
        if (gpio == NULL) {
                fprintf(stderr, "Could not intialize GPIO on pin %d\n",
gpio_pin);
                return NULL;
        }

        if (mraa_failed(mraa_gpio_dir(gpio, MRAA_GPIO_OUT))){
                fprintf(stderr, "Could not set GPIO pin to input on pin
%d\n", gpio_pin);
                return NULL;
        }

        mraa_gpio_write(gpio, 0); // default to disabled

        return gpio;
}

int init_camera(int pwm_pin) {
        PWM_camera = mraa_pwm_init(pwm_pin);
        if (PWM_camera == NULL) {
                fprintf(stderr, "Could not initialize PWM on pin %d\n",
pwm_pin);
                return -1;
        }

        if (mraa_failed(mraa_pwm_period_us(PWM_camera,19000))){
                fprintf(stderr, "Could not set period on pin %d\n",
pwm_pin);
                return -1;
        }
        if (mraa_failed(mraa_pwm_pulsewidth_us(PWM_camera, 1500+offset))){
                fprintf(stderr, "Could not set duty cycle on pin %d\n",
pwm_pin);
                return -1;
        }

        if (mraa_failed(mraa_pwm_enable(PWM_camera, 1))) {
                fprintf(stderr, "Could not enable PWM on pin %d\n",
pwm_pin);
                return -1;
        }

        return 0;
}


void close_camera(int camera_pin) {
        mraa_pwm_write(PWM_camera, 0);
        mraa_pwm_enable(PWM_camera, 0);
        mraa_pwm_close(PWM_camera);

        mraa_gpio_context GPIO_camera = mraa_gpio_init(camera_pin);
        mraa_gpio_dir(GPIO_camera, MRAA_GPIO_OUT);
        mraa_gpio_write(GPIO_camera, 0);
        mraa_gpio_close(GPIO_camera);
}
```

7.1.2.3 Initializer.h

```
#include "initializer.h"

int mraa_failed(mraa_result_t result){
/* Check if MRAA function was successfully done
 * This is done when the result of an MRAA function
 * (mraa_result_t type) is one of the two:
 *              MRAA_SUCCESS
 *              MRAA_ERROR_PLATFORM_ALREADY_INITIALISED
 * This means that the operation was successfully done
 * or the aimed initialization has happend already.
 * ------------------------------------------------------
 * Return 0 if operation successfully.
 * Return -1 if operation failed.
 */
        if (result == MRAA_SUCCESS || result ==
MRAA_ERROR_PLATFORM_ALREADY_INITIALISED)
                return 0;
        else{
                return -1;
        }
}

mraa_gpio_context init_GPIO_INT(int gpio_pin, gpio_edge_t int_edge, void
handler (void*)) {
// gpio_pin = gpio pin that will be used
// int_edge = which edge to interrupt on
// handler = interrupt handler
// return the GPIO context upon success, NULL upon failure

        mraa_gpio_context gpio = mraa_gpio_init(gpio_pin);
        if (gpio == NULL) {
                fprintf(stderr, "Could not intialize GPIO on pin %d\n",
gpio_pin);
                return NULL;
        }

        if (mraa_failed(mraa_gpio_dir(gpio, MRAA_GPIO_IN))){
                fprintf(stderr, "Could not set GPIO pin to input on pin
%d\n", gpio_pin);
                return NULL;
        }

        if (mraa_failed(mraa_gpio_isr(gpio, int_edge, handler, NULL))){
                fprintf(stderr, "Failed to initialize interrupt for pin
%d\n", gpio_pin);
                return NULL;
        }

        return gpio;
}

mraa_pwm_context init_PWM(int pwm_pin, int period, float duty_init) {
// pwm_pin = pin on which to have PWM output
// period = period of PWM in microseconds
// duty_init = initial duty cycle to set (0.05 to 0.95 -> 5% to 95%)
// return the PWM context upon success, NULL upon failure

        mraa_pwm_context PWM = mraa_pwm_init(pwm_pin);

        if (PWM == NULL) {
                fprintf(stderr, "Could not initialize PWM on pin %d\n",
pwm_pin);
                return NULL;
        }

        if (period < 50) period = 50;

        if (mraa_failed(mraa_pwm_period_us(PWM, period))){
                fprintf(stderr, "Could not set period on pin %d\n",
pwm_pin);
                return NULL;
        }

        // Set bounds on duty cycle
        if (duty_init < 0.05) duty_init = 0.05;
        if (duty_init > 0.95) duty_init = 0.95;

        if (mraa_failed(mraa_pwm_write(PWM, duty_init))){
                fprintf(stderr, "Could not set duty cycle on pin %d\n",
pwm_pin);
                return NULL;
        }
        if (mraa_failed(mraa_pwm_enable(PWM, 1))) {
                fprintf(stderr, "Could not enable PWM on pin %d\n",
pwm_pin);
                return NULL;
        }

        return PWM;
}

int close_PWM_GPIO(mraa_pwm_context pwm, mraa_gpio_context gpio, int
pwm_pin) {
// return 0 upon successfully closing PWM and GPIO
// return -1 if PWM output couldn't be guaranteed to be 0

        mraa_pwm_write(pwm, 0);
        mraa_pwm_enable(pwm, 0);
        mraa_pwm_close(pwm);

/* When the PWM is disabled and closed, its value on the pin
 * stops at whatever value it was at that point. This could be
 * high or low. That may be a problem since it isn't necessarily
 * acceptable to have an output always be high. Thus, what is
 * needed is to manually drive the pin low. This is done by
 * switching from PWM to GPIO on that pin and driving it low asctime
 * an output.
 */
        mraa_gpio_context PWM_GPIO = mraa_gpio_init(pwm_pin);
        if (PWM_GPIO == NULL) {
                fprintf(stderr, "Could not initialize GPIO on pin %d\n",
PWM_GPIO);
                return -1;
        }

        if (mraa_failed(mraa_gpio_dir(PWM_GPIO, MRAA_GPIO_OUT))){
                fprintf(stderr, "Could not set GPIO pin to output on pin
%d\n", PWM_GPIO);
                return -1;
        }

        mraa_gpio_write(PWM_GPIO, 0);
        mraa_gpio_close(PWM_GPIO);

        // Stop interrupt watcher on GPIO and set interrupt
        // edge to none (MRAA_GPIO_EDGE_NONE)
        mraa_gpio_isr_exit(gpio);
        mraa_gpio_close(gpio);

        return 0;
}


mraa_gpio_context init_GPIO_motorenable(int gpio_pin) {
// gpio_pin = gpio pin that will be used
// return the GPIO context upon success, NULL upon failure

        mraa_gpio_context gpio = mraa_gpio_init(gpio_pin);
        if (gpio == NULL) {
                fprintf(stderr, "Could not intialize GPIO on pin %d\n",
gpio_pin);
                return NULL;
        }

        if (mraa_failed(mraa_gpio_dir(gpio, MRAA_GPIO_OUT))){
                fprintf(stderr, "Could not set GPIO pin to input on pin
%d\n", gpio_pin);
                return NULL;
        }

        mraa_gpio_write(gpio, 0); // default to disabled

        return gpio;
}

int init_camera(int pwm_pin) {
        PWM_camera = mraa_pwm_init(pwm_pin);
        if (PWM_camera == NULL) {
                fprintf(stderr, "Could not initialize PWM on pin %d\n",
pwm_pin);
                return -1;
        }

        if (mraa_failed(mraa_pwm_period_us(PWM_camera,19000))){
                fprintf(stderr, "Could not set period on pin %d\n",
pwm_pin);
                return -1;
        }

        if (mraa_failed(mraa_pwm_pulsewidth_us(PWM_camera, 1500+offset))){
                fprintf(stderr, "Could not set duty cycle on pin %d\n",
pwm_pin);
                return -1;
        }

        if (mraa_failed(mraa_pwm_enable(PWM_camera, 1))) {
                fprintf(stderr, "Could not enable PWM on pin %d\n",
pwm_pin);
                return -1;
        }

        return 0;
}


void close_camera(int camera_pin) {
        mraa_pwm_write(PWM_camera, 0);
        mraa_pwm_enable(PWM_camera, 0);
        mraa_pwm_close(PWM_camera);

        mraa_gpio_context GPIO_camera = mraa_gpio_init(camera_pin);
        mraa_gpio_dir(GPIO_camera, MRAA_GPIO_OUT);
        mraa_gpio_write(GPIO_camera, 0);
        mraa_gpio_close(GPIO_camera);
}
```

7.1.2.4 Logging.c

```
#include "logging.h"

void print_log(logging_t* log_array, time_sns_t* init_time) {
        int i;
        time_sns_t temp;
        temp = subtract_times(init_time, init_time);
        printf("Starting time: %04ld:%09ld\n", temp.tv_sec, temp.tv_nsec);

        for (i=0; i<log_array->time_counter; i++){
                temp.tv_sec = log_array->tv_sec[i];
                temp.tv_nsec = log_array->tv_nsec[i];
                temp = subtract_times(&temp, init_time);
                printf("Interrupt # %3ld: %04ld:%09ld\n", i, temp.tv_sec, 
temp.tv_nsec);
        }
}

time_sns_t subtract_times(time_sns_t* larger_time, time_sns_t* smaller_time)
{
// return result of larger_time-smaller_time

        time_sns_t difference;
        difference.tv_sec = larger_time->tv_sec - smaller_time->tv_sec;
        difference.tv_nsec = larger_time->tv_nsec - smaller_time->tv_nsec;
        // deal with underflow issue
        if (difference.tv_nsec < 0){
                difference.tv_sec--;
                difference.tv_nsec += (long int)1000*(long int)1000*(long 
int)1000;
        }

        return difference;
}
```

7.1.2.5 Logging.h

```
#include <stdio.h>
#include <time.h>

#define MAX_TIMELOG_COUNT 100000

typedef struct logging_array logging_t;

// reuse struct defined in time.h
typedef struct timespec time_sns_t;

struct logging_array{
// time stored as seconds and nanoseconds:
        long tv_sec[MAX_TIMELOG_COUNT];
        long tv_nsec[MAX_TIMELOG_COUNT];
        long time_counter;
};

void print_log(logging_t* log_array, time_sns_t* init_time);
```

```
time_sns_t subtract_times(time_sns_t* larger_time, time_sns_t*
smaller_time);
```

7.1.2.6 Pwm.c

```
#include "pwm.h"

int motor_stop(mraa_pwm_context pwm, int pwm_pin, mraa_gpio_context ME) {

        mraa_pwm_write(pwm, 0);
        mraa_pwm_enable(pwm, 0);
        mraa_pwm_close(pwm);

/* When the PWM is disabled and closed, its value on the pin
 * stops at whatever value it was at that point. This could be
 * high or low. That may be a problem since it isn't necessarily
 * acceptable to have an output always be high. Thus, what is
 * needed is to manually drive the pin low. This is done by
 * switching from PWM to GPIO on that pin and driving it low asctime
 * an output.
 */
        mraa_gpio_context PWM_GPIO = mraa_gpio_init(pwm_pin);
        if (PWM_GPIO == NULL) {
                fprintf(stderr, "Could not initialize GPIO on pin %d\n",
PWM_GPIO);
                return -1;
        }

        if (mraa_failed(mraa_gpio_dir(PWM_GPIO, MRAA_GPIO_OUT))){
                fprintf(stderr, "Could not set GPIO pin to output on pin
%d\n", PWM_GPIO);
                return -1;
        }

        mraa_gpio_write(PWM_GPIO, 0);
        mraa_gpio_close(PWM_GPIO);

        mraa_gpio_write(ME, 0); // disable motor

        return 0;
}
```

7.1.2.7 Pwm.h

```
#include "initializer.h"

int motor_stop(mraa_pwm_context pwm, int pwm_pin, mraa_gpio_context ME);
```

7.1.2.8 Timers.c

```
#include "timers.h"

/* http://pubs.opengroup.org/onlinepubs/007908799/xsh/signal.h.html
 */

int init_timer(void) {
```

```
/*
 *      http://linux.die.net/man/2/timer_create
 *
 *      Steps:
 *      1) Establish handler for signal
 *      2) Block timer signal temporarily
 *      3) Create timer
 *      4) Start timer
 *      5) Unlock the timer signal
 */

 //**************************************************************
 // Step 1: Establish handler for signal

        sa.sa_sigaction = timer_handler;
        sa.sa_flags = SA_SIGINFO;
/* Causes extra information to be passed to signal handlers at
 * the time of receipt of a signal
 */

 /* The sigaction() system call is used to change the action taken
 * by a process on receipt of a specific signal.
 *
 * int sigaction(int signum, const struct sigaction *act, \
 *                     struct sigaction *oldact);
 *
 * signum specifies the signal and can be any valid signal except
 * SIGKILL and SIGSTOP
 *
 * http://man7.org/linux/man-pages/man2/sigaction.2.html
 */
        if (sigaction(SIG_TIMER, &sa, NULL)) {
                fprintf(stderr, "Failed to estabilsh handler for timer
signal!\n");
                return -1;
        }
 //**************************************************************
 // Step 2: Block timer signal temporarily

 // http://linux.die.net/man/3/sigemptyset
        sigset_t mask;

        sigemptyset(&mask);
        /* int sigemptyset(sigset_t *set);
         *
         * Initializes the signal set given by set to empty, with all
         * signals excluded from the set.
         */

        sigaddset(&mask, SIG_TIMER);
        /* int signaddset(sigset_t *set, int signum);
         *
         * Add signal signum to set
         *
         */
        /* http://linux.die.net/man/2/sigprocmask
         *
         * int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
         *
         * Used to fetch and/or change the signal mask of the calling
thread.
         * The signal mask is the set of signals whose delivery is currently
         * blocked for the caller.
         *
         * Returns 0 on success and -1 on error.
         */
        if (sigprocmask(SIG_SETMASK, &mask, NULL)) {
                fprintf(stderr, "Couldn't block timer signal
temporarily!\n");
                return -1;
        }
 //****************************************************************
 // Step 3: Create the timer

        sev.sigev_notify = SIGEV_SIGNAL;
/* Notify the process by sending the signal specified in sigev_signo
 *  If the signal is caught with a signal handler that was
    registered using the sigaction(2) SA_SIGINFO flag, then the
        following fields are set in the siginfo_t structure that is
        passed as the second argument of the handler:

        si_code   This field is set to a value that depends on the
        API delivering the notification.

        si_signo  This field is set to the signal number (i.e., the
        same value as in sigev_signo).

        si_value  This field is set to the value specified in
        sigev_value.

        Depending on the API, other fields may also be set in the
        siginfo_t structure.

        The same information is also available if the signal is
        accepted using sigwaitinfo(2).
 */

        sev.sigev_signo = SIG_TIMER;
/* <signal.h> also declares the macros SIGRTMIN and SIGRTMAX,
 * which evaluate to integral expressions and, if the Realtime
 * Signals Extension option is supported, specify a range of
 * signal numbers that are reserved for application use and for
 * which the realtime signal behaviour specified in this specification
 * is supported. The signal numbers in this range do not overlap any
 * of the signals specified in the following table.
 *
 * The range SIGRTMIN through SIGRTMAX inclusive includes at least
 * RTSIG_MAX signal numbers.
 *
 * It is implementation-dependent whether realtime signal behaviour
 * is supported for other signals.
 */
```

```c
       sev.sigev_value.sival_ptr = (void*) &timer_sw;

       if (timer_create(CLOCK_REALTIME, &sev, &timer_sw)){
                       fprintf(stderr, "Could not create timer!\n");
                       return -1;
       }

//*************************************************************
// Step 4: Start the timer

       struct itimerspec its;
       /* http://pubs.opengroup.org/onlinepubs/007908775/xsh/time.h.html
        *
        * struct timespec  it_interval  timer period
        * struct timespec  it_value     timer expiration
        *
        */
       its.it_interval.tv_sec = TMR_PERIOD_sec;
       its.it_interval.tv_nsec = TMR_PERIOD_ns;
       its.it_value.tv_sec = its.it_interval.tv_sec;
       its.it_value.tv_nsec = its.it_interval.tv_nsec;

       /* http://man7.org/linux/man-pages/man2/timer_settime.2.html
        *
        * int timer_settime(timer_t timerid, int flags, \
        *                     const struct itimerspec *new_value,
        *                     struct itimerspec *old_value);
        *
        * Arms or disarms the timer identified by timerid. The
        * new_value argument is pointer to an itimerspec structure
        * that specifies the new initial value and the new interval
        * for the timer.
        *
        * If new_value->it_value specifies a nonzero value (ie either
        * subfield is nonzero), then timer_settime() arms (starts) the
        * timer, setting it to initially expire at the given time. (If
        * the timer was already armed, then the previous settings are
        * overwritten.) If new_value->it_value specifies a zero value
        * (ie, both subfields are zero), then the timer is disarmed.
        *
        * The new_value->it_interval field specifies the period of the
        * timer, in seconds and nanoseconds. If this field is nonzero,
        * then each time that an armed timer expires, the timer is reloaded
        * from the value specified in new_value->it_interval. If
        * new_value->it_interval specifies a zero value, then the timer
        * expires just once, at the time specified by it_value.
        *
        * Returns 0 on success, -1 on error.
        */

        if (timer_settime(timer_sw, 0, &its, NULL)){
               fprintf(stderr, "Failed to start timer!\n");
               return -1;
        }

//*************************************************************
```

```
 // Step 5: Unlock the timer signal

        if (sigprocmask(SIG_UNBLOCK, &mask, NULL)){
        // if (sigprocmask(SIG_UNBLOCK, &sa.sa_mask, NULL)){
                fprintf(stderr, "Failed to unlock timer!\n");
                return -1;
        }

        #ifdef PRINT_DEBUG
                fprintf(stdout, "All timer settings done!\n");
        #endif

        return 0;

}

int close_timer(void){

 // Disarm timer:

        struct itimerspec its;
        its.it_interval.tv_sec = 0;
        its.it_interval.tv_nsec = 0;
        its.it_value.tv_sec = its.it_interval.tv_sec;
        its.it_value.tv_nsec = its.it_interval.tv_nsec;

         if (timer_settime(timer_sw, 0, &its, NULL)){
                 fprintf(stderr, "Failed to kill timer!\n");
                 return -1;
         }

// Delete timer:

        if (timer_delete(timer_sw)){
                fprintf(stderr, "Failed to delete timer!\n");
                return -1;
        }

        return 0;
}

static void timer_handler(int sig, siginfo_t *si, void *uc) {
        // Toggle LED

        if (++speed_index == number_of_points) speed_index = -1;
        if (++camera_index == number_of_points) camera_index = -1;

        if (++int_counter == 2) {
                int_counter = 0;
                interrupt_enabled = 1;
        }

        /* http://man7.org/linux/man-pages/man2/signal.2.html
         *
         * sighandler_t signal(int signum, sighandler_t handler)
         *
         * signal() sets the disposition of the signal signum
         * to handler, which is either SIG_IGN, SIG_DFL or the
         * address of a programmer-defined function (a "signal
         * handler").
         *
         * If the signal signum is delivered to the process, then
         * one of the following happens:
         *
         * If the disposition is set to SIG_IGN, then the signal
         * is ignored.
         *
         * If the disposition is set to SIG_DFL, then the default
         * action associated with the signal (see:
         * http://man7.org/linux/man-pages/man7/signal.7.html)
         * occurs.
         *
         * If the disposition is set to a function, then first
         * either the disposition is reset to SIG_DFL, or the
         * signal is blocked, and then handler is called with
         * argument signum. If invocation of the handler caused
         * the signal to be blocked, then the signal is unblocked
         * upon return from the handler.
         *
         * The signals SIGKILL and SIGSTOP cannot be caught or ignored.
         *
         * Returns the previous value of the signal handler, or SIG_ERR
         * on error. In the event of an error, errno is set to indicate
         * the cause.
         */


        //signal(sig, SIG_IGN);
}
```

7.1.2.9 Timers.h

```
// link with -lrt

#include "initializer.h"
#include <time.h>
#include <signal.h> // needed for struct sigevent sa
// Please read: http://man7.org/linux/man-pages/man7/signal.7.html

/* http://linux.die.net/man/2/timer_create
 * http://man7.org/linux/man-pages/man2/timer_create.2.html
 * http://man7.org/linux/man-pages/man7/sigevent.7.html
 */

 #define SIG_TIMER SIGRTMIN


 #define number_of_points 585

 #define TMR_PERIOD_sec 0
 #define TMR_PERIOD_ns 500000000

int init_timer(void);
int close_timer(void);
static void timer_handler(int sig, siginfo_t *si, void *uc);

timer_t timer_sw;

struct sigevent sev;
/*
struct sigevent {
        int          sigev_notify; // Notification method
        int          sigev_signo;  // Notification signal
        union sigval sigev_value;  // Data passed with notification
        void       (*sigev_notify_function) (union sigval);
        // Function used for thread notification (SIGEV_THREAD)
        void        *sigev_notify_attributes;
        // Attributes for notification thread (SIGEV_THREAD)
        pid_t        sigev_notify_thread_id;
        // ID of thread to signal (SIGEV_THREAD_ID)
};
*/

/*
union sigval {           // Data passed with notification
        int     sival_int;         // Integer value
        void   *sival_ptr;         // Pointer value
};

*/

struct sigaction sa;
/*
 * The header provides a declaration of struct sigaction,
 * including at least the following members:
 *
 * void     (*sa_handler)(int)
 *             what to do on receipt of signal
 * sigset_t   sa_mask
 *             set of signals to be blocked during execution
 *             of the signal handling function
 * int        sa_flags
 *             special flags
 * void (*)(int, siginfo_t *, void *) sa_sigaction
 *             pointer to signal handler function or one
 *             of the macros SIG_IGN or SIG_DFL
 */

 volatile int speed_index;
 volatile int camera_index;
 volatile int photogate_index;

 volatile char interrupt_enabled;
 volatile char int_counter;
```

7.1.2.10 Text files that MUST be included in the same directory

photogate_timeindices.txt

```
62

92

114

143

168

257

304

347

367

381

524
```

vel_history.txt

```
**filled by edison
```

yawangle_profile.txt

```
**filled by edison
```

Edison stuff

==

/home/root/pod_pwm/pod

/home/root -> default directory

```
gcc -o pod *.c -lmraa -lrt -lm
```

```
./pod
```

===

list of files that must be in the same directory:

main.c

initializer.c, initializer.h

logging.c, logging.h

pwm.c, pwm.h

timers.c, timers.h

photogate_timeindices.txt

vel_history.txt

yawangle_profile.txt

===

configure_edison -w   -> setting up wifi

===

username: root

password: password

COM port: ---

Baud: 115200

===

Reflash Edison

Download the Yocto image from Intel software downloads.

Plug both USB cables into computer. Edison shows up as flash drive. Remove everything in Edison drive/directory.

Copy-paste the extracted Yocto image onto that drive.

Go into Edison terminal and type:

```
reboot ota
```

==

After you flash, you must update MRAA:

```
echo "src mraa-upm http://iotdk.intel.com/repos/1.1/intelgalactic" > /etc/opkg/mraa-upm.conf

opkg update

opkg install libmraa0

opkg upgrade

npm install mraa
```

==

7.1.3 Motor Control Code for the Modbots

MODBOT Motor Controller Code

```
/* Modbot motor controller code to be used on the *.118 modbot.
This code was developed for the ESCON 36/3 motor controllers.
The Arduino should be connect on COM 8 on the modbot. In order to
check this, log on remotely onto the modbot (using Remote Desktop
Connection) with user: remote and no password. In device manager,
verifty that the arduino is on COM 8 with the appropriate baud
rate, change if necessary.

A string is sent from the moterboard to the Arduino in the following
format: 'M direction LF RF LB RB #' All values are sent as ASCII
characters. M specifies that the command is to be used by the motors
the direction is sent as an ASCII with its 4 least significatn bit
values corresponding to motor direction (1 or 0). The speed of each
motor is sent as an ASII ranging from 0-255 (LF RF LB RB). The #
terminates the string.

The ESCON 36/3 controller can only accept a speed pwm range between
10-90%. The original 0-255 speed value must be mapped between 26-229.

In order to program the ESCON 36/3 controllers, open the ESCON studio
```

```
program and select parameters. Alternatively, upload the preset
parameters from the Cornell Cup google drive.
*/

unsigned char motor [8];
char garb;
int drct;
unsigned int a;
int b;
unsigned int mot;
float rMax = 10000.0; //Maximum RPM for motor at 4.0V
int analogInRF = 0; //Analog voltage for hall effect RPM Speed RF
int analogInLF = 0; //Analog RPM LF
int analogInRB = 0; //Analog RPM RB
int analogInLB = 0; //Analog RPM LB


void setup() {
  //cli(); //disable interrupts while setting up
  Serial.begin(57600);
  Serial1.begin(57600);

  //--------------------

  pinMode(3, OUTPUT);    //LB Motor PWM
  pinMode(49, OUTPUT);   //LB Enable
  pinMode(51, OUTPUT);   //LB Directionz
  pinMode(41, OUTPUT);   //LB STOP
  pinMode(0, INPUT);     //LB RPM Read

  pinMode(4, OUTPUT);    //RF PWM
  pinMode(50, OUTPUT);   //RF Enable
  pinMode(48, OUTPUT);   //RF Direction
  pinMode(37, OUTPUT);   //RF STOP
  pinMode(1, INPUT);     //RF RPM Read

  pinMode(5, OUTPUT);    //LF PWM
  pinMode(45, OUTPUT);   //LF Enable
  pinMode(47, OUTPUT);   //LF Direction
  pinMode(39, OUTPUT);   //LF STOP
  pinMode(2, INPUT);     //LF RPM Read

  pinMode(6, OUTPUT);    //RB PWM
  pinMode(46, OUTPUT);   //RB Enable
  pinMode(44, OUTPUT);   //RB Direction
  pinMode(22, OUTPUT);   //RB STOP
  pinMode(3, INPUT);     //RB RPM Read

  /**************************Set Motors speed at 10% pwm (0% speed)************/
    analogWrite(6,27);
    analogWrite(4,27);
```

```
     analogWrite(3,27);
     analogWrite(5,27);

/*************************** Enable Motors ************************************/
    digitalWrite(46,HIGH); //RB enabled, high is enabled, connect to digital pin 2 on ESCON 36/3
controller
    digitalWrite(49,HIGH); //LB enabled
    digitalWrite(50,HIGH); //RF enabled
    digitalWrite(45,HIGH); //LF enabled

    /*************************** Disable Stop ************************************/
    digitalWrite(22,LOW); //RB disable stop
    digitalWrite(41,LOW); //LB disable stop
    digitalWrite(37,LOW); //RF disable stop
    digitalWrite(39,LOW); //LF disable stop


    TCCR5A = 0;// set entire TCCR2A register to 0
    TCCR5B = 0;// same for TCCR2B
    TCNT5  = 0;//initialize counter value to 0
    // Set CS21 bit for 1024 prescaler
    TCCR5B |= (1 << CS50);
    TCCR5B |= (1 << CS52);

  //C code implementation of PWM, can be used if different PWM settings are required (ex increase
frequency)
  //So far it works with 490Hz, but if need be, we can increase to 1.95KHz
  //TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20); //setup fast PWM mode
using timer 2 on pins 9,10
  //TCCR2B = _BV(CS21) | _BV(CS20); //32 prescaler: (16MHz/32/256)= 1.95KHz
  //OCR2A = 100; //pin 10, write 100/255= 39% duty cycle

//  /************************************Setup Inturrupts****************************************/
//  TCCR5A = 0;     //set entire TCCR5A register to 0
//  TCCR5B = 0;     //same for TCCR5B
//
//  TCNT5 = 34286;  //preload timer 66536 - 16MHz/256/2Hz (will overflow and trigger interrupt
//
//  TCCR5B |= (1 << CS52);  //Set prescaler to 256
//  TIMSK5 |= (1 << TOIE5); //Enable timer overflow interrupt

  //UCSR0A |= (1 << RXC0);

  //sei();          //Enable all interrupts
}
void setMotor(){

/*************************** Set Direction ************************************/
    digitalWrite(44, bitRead((int)motor[1], 0)); //RB direction high is clockwise, connect to digital pin
3 on ESCON 36/3 controller
    digitalWrite(51, abs(bitRead((int)motor[1], 1)-1)); //LB direction
    digitalWrite(48, bitRead((int)motor[1], 2)); //RF direction
    digitalWrite(47, abs(bitRead((int)motor[1], 3)-1)); //LF direction


/*************************** Turn Conditions **********************************/

    //if (RB=0                    && LF= 0                    && LB=1                   && RF=1)
    if (bitRead((int)motor[1], 0)==0 && bitRead((int)motor[1], 3) && bitRead((int)motor[1], 1)==0 &&
bitRead((int)motor[1], 2)){
    digitalWrite(44, HIGH); //RB
    digitalWrite(51, HIGH); //LB
    digitalWrite(48, HIGH); //RF
    digitalWrite(47, HIGH); //LF

    }


    if (bitRead((int)motor[1], 0) && bitRead((int)motor[1], 3)==0 && bitRead((int)motor[1], 1) &&
bitRead((int)motor[1], 2)==0){
    digitalWrite(44, LOW); //RB
    digitalWrite(51, LOW); //LB
    digitalWrite(48, LOW); //RF
    digitalWrite(47, LOW); //LF

    }

/**************************** Set Speed ****************************************/
    //If RB motor speed less then 36, disable motor
    if ((unsigned int)motor[5] <= 1){
    digitalWrite(22,LOW); //stop
    //analogWrite(6,27);
    }
    else{
    digitalWrite(22,HIGH); //disable stop
    analogWrite(6, (unsigned int)(map(motor[5],0,255,26,229))); //RB speed, pwm ranges from 10%-
90% 10%=0rpm, 90%=150000rpm, connect to digital pin 1 on ESCON 36/3 controller
    }

    //If LB motor speed less than 36, disable motor
    if ((unsigned int)motor[4] <= 1){
    digitalWrite(41,LOW);
    //analogWrite(3,27);
    }
    else{
    digitalWrite(41,HIGH); //disable stop
    analogWrite(3, (unsigned int)(map(motor[4],0,255,26,229))); //LB speed, pwm ranges from 10%-
90% 10%=0rpm, 90%=150000rpm, connect to digital pin 1 on ESCON 36/3 controller
    }

//    If RF motor speed less than 36, stop motor
```

```
    if ((unsigned int)motor[3] <=1){
      digitalWrite(37,LOW); //stop
      //analogWrite(4,26);
    }
    else{
      digitalWrite(37,HIGH); //disable stop
      analogWrite(4, (unsigned int)(map(motor[3],0,255,26,229))); //RF speed, pwm ranges from 10%-
90% 10%=0rpm, 90%=15000rpm, connect to digital pin 1 on ESCON 36/3 controller
    }

//    If LF motor speed less than 36, disable motor
    if ((unsigned int)motor[2] <= 1){
      digitalWrite(39,LOW); //stop
      //analogWrite(5,26);
    }
    else{
      digitalWrite(39,HIGH); //disable stop
      analogWrite(5, (unsigned int)(map(motor[2],0,255,26,229))); //LF speed, pwm ranges from 10%-
90% 10%=0rpm, 90%=150000rpm, connect to digital pin 1 on ESCON 36/3 controller
    }

}

 /*********************************Perform Calculations for motor RPM*******************/
 /*
 float rpmCalc(float analogVal){
     float r = (rMax / 409.2) * analogInRF - rMax; //linear equation for motor's rpm
     float rpm = r / 33.0; //divide by 33.0 due to gear ratio to find true rpm
     return rpm;
 }
 */
 /*************************************Inturrupt to Read Hall Sensors**********************/
//Stores to analogIn the analog voltage given by the rpm of the motor
//ISR(TIMER5_OVF_vect){
//  TCNT5 = 34286;        //preload timer
//  dataAvailable = 1;
//}
void loop(){
  //Triggers every 10Hz based upon prescaler set above
  if(TCNT5 >= 1562){
    analogInLB = analogRead(0);
    analogInRF = analogRead(1);
    analogInLF = analogRead(2);
    analogInRB = analogRead(3);
    Serial1.print("%");           //Send % symbol to indicate
    Serial1.print(analogInLB);
    Serial1.print(" ");
    Serial1.print(analogInRF);
    Serial1.print(" ");
    Serial1.print(analogInLF);
    Serial1.print(" ");
    Serial1.println(analogInRB);
    TCNT5 = 0;
    //Serial.flush();
  }
  while(Serial.available() > 0) {
//    Serial1.println(Serial.peek());
    if (Serial.peek()=='M'){
      Serial.readBytes((char*) motor,7);
      if (motor[6]=='#'){
      setMotor();
      }
    }
    else {Serial.readBytes((char *) garb,1);}
}}
```

7.1.4 HUD LED Display for the Simulator chair

For the LED display, we selected a digital RGB strip, LPD8806. Each LED can be addressed individually. The strip has a data in pin and a clock in pin, as well as a +5V and ground pin. Each LED can draw up to 60 mA of current from a 5V supply (assuming that all LEDs are set on full brightness). The LPD8806 contains a built-in 1.2 MHz high speed 7-bit PWM for each channel. The chip is capable of doing 21-bit color per LED; however, for our purposes, we only set the LEDs to change to red or green and blink on and off.

An Arduino MicroController was used to control the LED strip by connecting two of its digital pins to the the data in and clock in pins of the strip. Ann LPD8806 library was used in the Arduino code. This library can be found at this GitHub link: https://github.com/adafruit/LPD8806.

In the code, each LED is set to a specific color based on a 6 character sequence read from the Serial monitor.

```
#include "LPD8806.h"
#include "SPI.h" // Comment out this line if using Trinket or Gemma
#ifdef __AVR_ATtiny85__
 #include <avr/power.h>
#endif
int off = 46;
int green = 103;
int red = 114;
int nLEDs = 32;
int dataPin = 2;
int clockPin = 3;
LPD8806 strip = LPD8806(nLEDs, dataPin, clockPin);
char inputBuffer[7];

int upL;  int midL;  int downL;
int upR;  int midR;  int downR;
```

```
//The following block of code identifies the LED numbers corresponding to each arrow.  For example the  start LED of
the upper left arrow (upLS) corresponds to LED 0, and the end LED of the upper left arrow (upLE) corresponds to
LED 3 on the strip.
int upLS = 0; int upLE = 3;
int midLS = 4; int midLE = 7;
int downLS = 8; int downLE = 11;
int upRS = 12; int upRE = 15;
int midRS = 16; int midRE = 19;
int downRS = 20; int downRE = 23;

void setup() {
  // Start up the LED strip
  strip.begin();
  // Update the strip, to start they are all 'off'
  strip.show();
  Serial.begin(9600);
  }
void loop(){
 if(Serial.available() > 0){
    for(int i = 0; i < 6; i++){
      inputBuffer[i] = Serial.read();
    }
  }
  upL = inputBuffer[0];
  midL = inputBuffer[1];
  downL = inputBuffer[2];
  upR = inputBuffer[3];
  midR = inputBuffer[4];
  downR = inputBuffer[5];

  //LEFT UP
  if(upL == off){
    RGBlight('O',upLS,upLE);
  }
  else if(upL == green){
    RGBlight('G',upLS,upLE);
  }
  else if(upL == red){
    RGBlight('R',upLS,upLE);
  }
//Repeat same if-statement block of LEFT UP code for each of 6 arrows

  strip.show();
  delay(100);
  Serial.flush();
}

void RGBlight(char color, int pinStart, int pinEnd){
   for(int i=pinStart; i<=pinEnd; i++){
     if(color == 'R') {strip.setPixelColor(i,strip.Color(127,0,0));}
     if(color == 'G') {strip.setPixelColor(i,strip.Color(0,127,0));}
     if(color == 'O') {strip.setPixelColor(i,strip.Color(0,0,0));}
   }
}
```