Lab Report 5

Final Project

EE 474 Fall 2016
Dr. Allan Ecker
12/9/2016

Bryan Bednarski (1337864)
Justin Allmaras (1329197)
Colin Summers (1332139)

1.0 Introduction

This fifth and final lab for EE 474 involved the combination of a number of peripheral sensors, motors and microcontrollers that we have been developing throughout the quarter. In earlier labs we had developed hardware drivers for and LCD screen, shift register, motor controller H-bridge, analog IR sensors and a kernel-level module for the BeagleBone. For this particular lab, we were expected to implement a bluetooth chip, to allow wireless communication and command sequences from the user. This process also involved developing a boot service routine to initialize and run our main rover-control functionality upon startup of our system. In addition to these required peripheral additions to our system, we also implemented a Wii Nunchuck remote to demonstrate I2C bus communication, a serial to mp3 converter chip to demonstrate SPI busses, and the openCV library to demonstrate object tracking and image processing on the beaglebone.

As we developed our hardware modules throughout the earlier labs in this class, we began to notice that we were heavily limited by the non-precise and power-lacking drive of the motors on the chassis with which we were provided. Therefore for our final project, we elected to do a complete hardware rebuild of the tank system with a new chassis, pair of brushless motors, an 11.1V 3 cell lipo battery, and custom motor control and peripheral printed circuit boards. It was on this system that we implemented our peripheral hardware mentioned above. This process was a huge undertaking for our group, because it involved redesigning much of our analog circuitry. This analog development did pay off in a huge way with the consistency of our analog readings from IR sensors as described later in this paper.

This paper walks through the development process of the hardware and software modules that were most critical in demo and the functionality of our completed system. In our design specification section we walk through the design purpose and process for each of our main communication protocols in play, and the structure of our openCV object tracking module. In each of these sections we describe why these different modules were important to the overall functionality of our system and imply where else they could be applicable in other projects.

2.0 Design Specifications

For our final lab, we built a tank drive rover with a variety of control modes and communication protocols building on our previous labs. For the lab, we implemented bluetooth control over UART, Wii Nunchuck control of I2C, and MP3 audio over SPI. To bring all the components together in an organized fashion, we designed, ordered and assembled a custom printed circuit board which plugs directly into the Beaglebone. We also implemented object tracking using OpenCV, but due to time constraints, we were unable to integrate this with the motor drive circuitry. See Appendix A for a block diagram of our system.

2.1 Printed Circuit Board

For this lab, we designed a printed circuit board using EagleCAD and sent the design to be manufactured and then soldered on the components after receiving the manufactured boards. The board had slots for the majority of the peripherals that we were implementing. These included IR sensors, a Wii Nunchuck, Motor Controls, LCD, as well as breaking out all of the pins on the Beaglebone for further use as needed. A schematic of the board can be seen in Appendix B. The specifics for each module will be discussed in that particular modules section of the report as needed. In addition to the option of powering the Beaglebone over USB, we included a buck converter to provide 5V to the Beaglebone in case we wanted to power the Beaglebone from the battery used to power the motors. For simplicity, this was omitted during the demonstration of our project.

We also implemented on board serial bluetooth using the RN42 bluetooth module. Special precautions were taken to make sure that the antenna would not be interfered with either by our circuit board or with the mounting onto the chassis. The device was positioned such that the antenna was hanging over the edge of the board as much as possible. In addition, the ground plane was removed from underneath the device to minimize signal attenuation through the board.

In addition to our initial peripheral circuit board, or cape, that was mounted to the two pinout rails on our PCB, we also designed a custom motor control PCB in order to drive the current and PWM signals necessary to drive the two motors on our new chassis. This circuit took the pwm and gpio inputs for motor controls, 8 pins in total, passing signals through an optoisolator and surface-mount motor control chip. The output of these motor controller drove a series of 4 PNP and 4 NPN mosfet transistors in a high-power H-bridge configuration. Unfortunately after testing this circuit extensively, we were unable to get it to function well-enough to drive our motors. We eventually elected to use the pre-fabricated motor driver that is described in **Section 2.5** of this report. **Figure 1** below shows the designed circuit boards before and after being soldered with surface-mount elements and connectors.

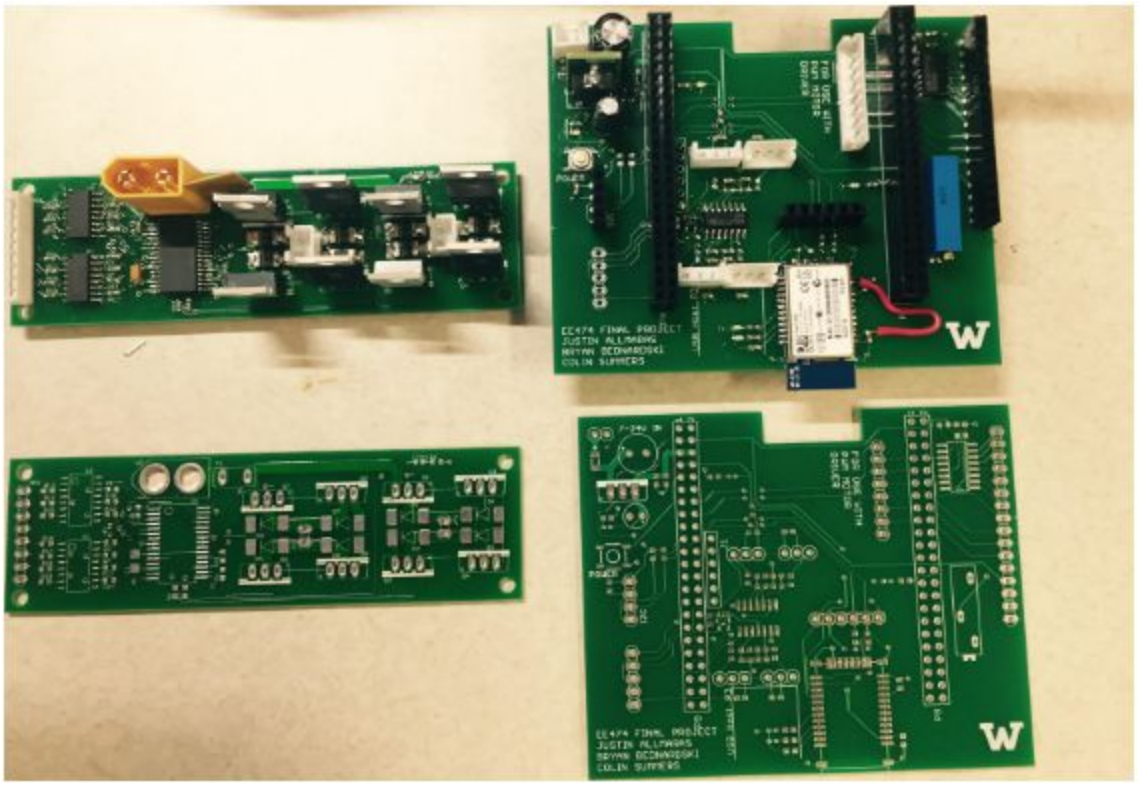


Figure 1: Printed circuit boards designed for Beaglebone peripherals and motor control

2.2 Bluetooth

Bluetooth communication was implemented using a UART communication bus between the RN42 bluetooth module and the Beaglebone. We used a baud rate of 115,200 baud, which was specified in our setup code for the bluetooth. The bluetooth module is set up such that there is a configuration timer during which it is possible to configure the settings of the module and after that time period, the module enters broadcasting mode and other bluetooth masters can connect to it. When the bluetooth module was in configuration mode, an orange LED would blink 10 times per second, and upon entering broadcasting mode, the orange LED would blink 1 time per second. After a device connects to the bluetooth module, the orange LED shuts off and a solid blue LED turns on, indicating a successful connection.

We used a bluetooth joystick app that had a two axis joystick and 6 buttons for use. The app send joystick data that ranged from 100 to 300 for both x and y and sent characters A-L for the button presses. For button 1, it send ‘A’ when it transitioned from off to on and ‘B’ when it transitioned from on to off and likewise for the other buttons. We used the joystick for driving

and the buttons to switch modes and send stop signals to the rover as needed. The bluetooth app does not send newline characters when it sends a data packet, so we needed to read the UART port in noncanonical (raw) mode so that the data would become available for us to consume and use. The following **Figure 2** below shows an example of a user interfacing with the Joy BT Commander app from the google play store to control our rover via bluetooth UART.

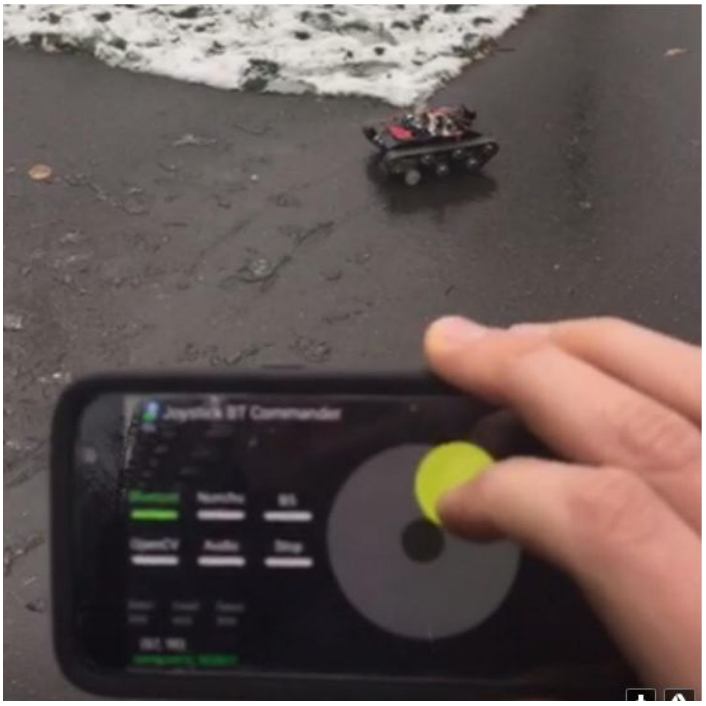


Figure 2: User interface between Joy BT Commander app and rover motor control

We read the bluetooth data on a 75ms timer interrupt to make sure that we had quick response to our input commands. On program startup, the rover was put into idle mode and the rover could only be taken out of this mode using a bluetooth command from the controller. This acts as a safety feature that does not allow the rover to issue any motor commands until it is explicitly told to do so. We also had a stop button that took top priority when determining a control mode, which was very important as our rover motor battery could not be easily accessed without taking the cover off, so being certain that the rover would not move was very important.

2.3 IR Sensors

On our rover, we mounted 4 infrared proximity sensors to detect objects near the rover in front, back or on either side. We used Sharp 2Y0A21 sensors for this purpose. According to the

datasheet, the sensor has a read range from 10 cm to 80 cm. In our experimenting, we found this to be accurate for our sensors. **Figure 2.3.1** referenced in the appendix shows the voltage/distance curve for these sensors.

From the AM335 datasheet, the maximum input value for the analog input pins depends on whether or not the internal voltage reference or the external voltage reference is used. From the Beaglebone schematic, see **Figure 3**, the external reference is used, so the the full-scale input range is from VREFN to VREFP, which is 0V to 1.8V.

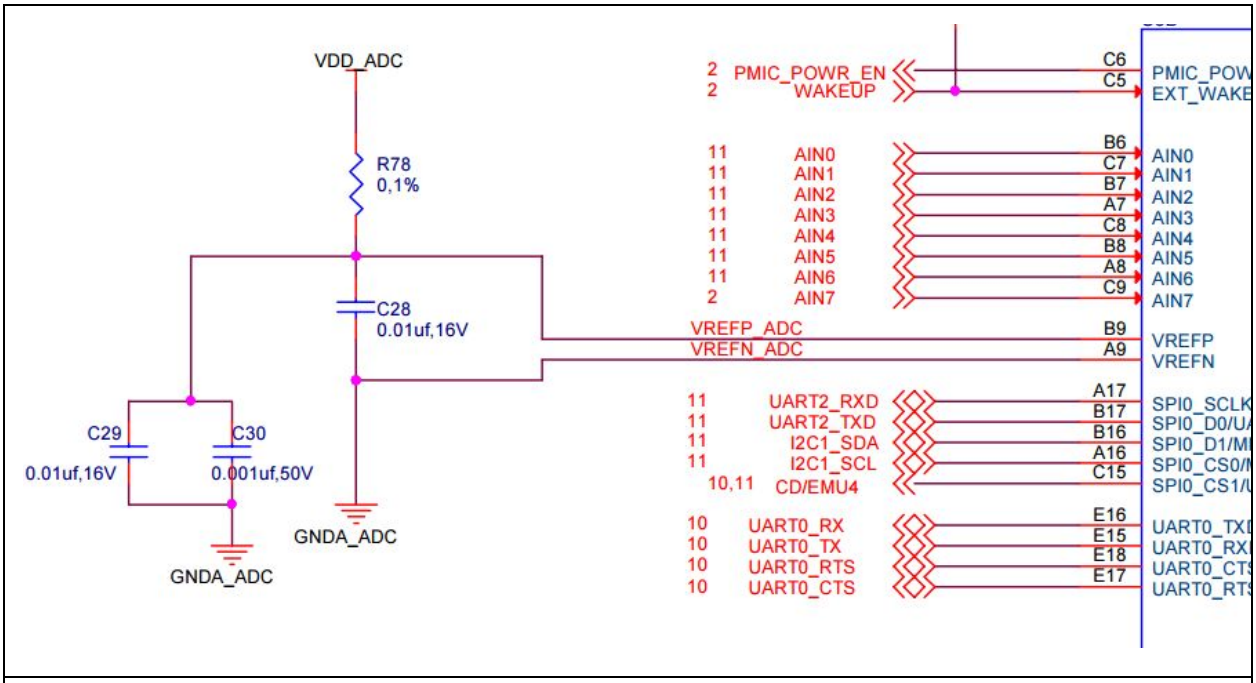


Figure 3: AM335 Analog Reference Voltage Structure

From **Figure 2.3.1**, we know that the IR sensors are capable of outputting voltages of over 3V, which could possibly damage the ADC of the AM335, so we needed to figure out a way to keep the input voltage to the ADC less than 1.8V. In lab 4 we first attempted to do this using a voltage divider with both legs set to 1kΩ. When viewing the output on the benchtop multimeter, this proved to be a good solution, but when reading the values on the Beaglebone analog input pins, we were unable to get a consistent reading. We changed resistor values multiple times and in the end were unable to get clean results using a voltage divider. We ended up implementing an op amp chopping circuit, see **Figure 4**, that allowed the signal to pass until it got to 1.8V at which point it held the value constant. Using this circuit, we still were encountering erroneous values being read by the Beaglebone ADC and we found that there was a significant amount of noise and ripple on the outputs of the sensors. We were unable to determine the root cause of the

noise, but we used a low pass filter to try and mitigate the effects as much as possible. We also used several decoupling capacitors at the input of the op amp power supply which helped reduce the noise as well.

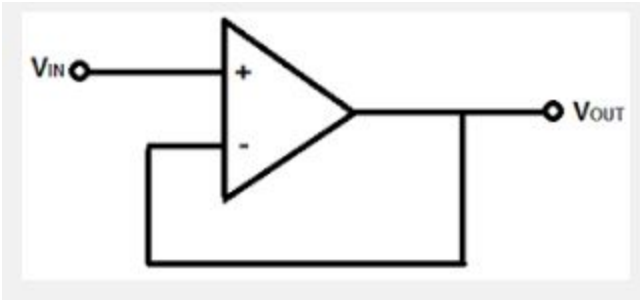


Figure 4: Op Amp Buffer Circuit

For this final lab, we implemented each of these solutions in series between theIR sensors and input ADC pins on the Beaglebone directly on our peripheral PCB cape. First, we used the 1.8 V reference from the beaglebone. We then used a voltage divider to drive that max output to 0.9V. Using a single op-amp clamp, we clamped the output value at a max 0.9V. We then used a quad op amp IC to buffer the input of each ADC pin transient noise that would enter our beaglebones reading pins. Ultimately, these methods were no different from the combinations that we built in Lab 4, other than that they were all used together. From this circuit, we read extremely clean IR sensor values and had consistent interrupts for our navigation algorithm. This analog solution also allowed us to revise our original code for interpreting the ADC values read from the sensors, and we were able to reduce the size of our "deadband" interrupt reaction buffer by over 75%!

The infrared sensors are read on a 100 ms timer interrupt and if the value is above or below a specified value, an interrupt is sent to the motor driver process to alert it of an impeding object or the removal of a previously impeding object. The reading of the analog inputs with the timer interrupt allows the rover to run uninterrupted as long as no objects impede its progress which makes for a more efficient process.

2.4 Nunchuck

To demonstrate the I2C communication protocol on our rover, we added a mode that allowed the user to control the rover using a wired Wii Nunchuck controller. While this operation was not as flashy as the wireless UART control of the bluetooth joystick, it worked well and reacted smoothly to this new protocol. I2C is useful for a robotics system, because it allows multiple "slave" devices to send data via a singular data bus to the system's "master"node, which

interprets each block's address and data package. While this system only used one I2C "slave" device, the procedure would be similar to adding additional devices. We registered this nunchuck device under the ioc-2 at a unique I2C address 0x52. The I2C breakout chip has four inputs: 5V, ground, System Data Clock and System Data Bus. The system Data Clock is necessary to sync all data busses in the system so that a device does not try to write at a different frequency than the Master node. A device cannot write to the bus while the bus is occupied by another device, but it will wait until the bus is clear and send its data.

The Wii Nunchuck remote sends the state of 9 different sensors in each of its data packets to the master node. The device has a 2-axis x-y joystick that we used to directly control the motors with the same algorithm to the bluetooth joystick. Each value for x and y form the joystick is between 0-255, being zeroed at 127 in both directions. With this data we were able to center the values between -127-128 and pass current xy data to out differential controller algorithm. This algorithm was critical to the smooth control of our over rover, because it could turn left at right motors at different rpms to account for banking rather than relying on simple turn-in-place navigation.

Over I2C the nunchuck also passed data from its "Z" and "C" buttons, and a three-axis accelerometer. We did not need this information for out rover, but it was being sent over I2C regardless as part of the nunchuck controller's normal operation. For reference, **Appendix D** provides a simple schematic of the I2C protocol and shows how data is transferred safely over a common system bus.

2.5 Motor Controls

We used a dual 15A motor controller to drive the motors. The motor controller took UART packages at baud rate of 9600 bits per second. The controller has a specific way of interpreting serial bytes sent. If the byte sent was between 1 and 127, it was used to control motor output 1 and if the number was between 128 and 255, the number was used to control motor 2. Sending 0 stopped both motors. Sending the midpoints of the input ranges (64 and 192) would stop that motor. Using zero as a stop value allows for the signal to the motor controller to be removed without generating spurious motor turn on.

We used an algorithm for controlling the tracks that allowed for simple motion such as forward, backward, left and right turns in place as well as banking turns in a continuous spectrum between forward and turn in place. The algorithm took in an x and y coordinate pair that we were able to get from either our virtual bluetooth joystick or our physical Nunchuck joystick and converts that to left and right track values that we were able to scale and send to our motor controller. This algorithm will also work for integrating OpenCV, where our module is designed to give an x

coordinate that corresponds to the location of the object. We would then determine a y value to send as a speed that we wanted the rover to run and the algorithm would generate appropriate left and right track values that we could use to drive our motors.

The motor controller uses 5V logic, so a logic level shifter was used to translate the 3.3V UART of the Beaglebone to 5V UART for the motor controller.

2.6 Computer Vision

Challenges

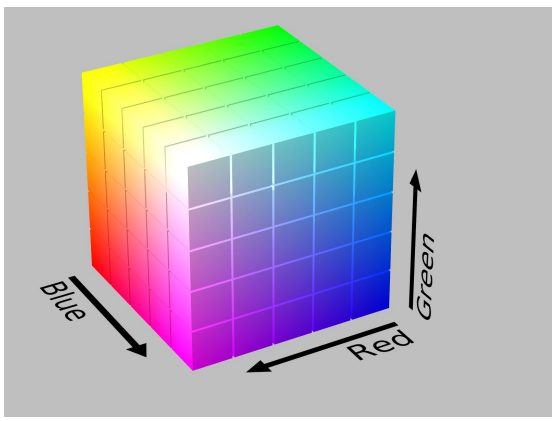
Our design specification included the ability to autonomously navigate by tracking a Frisbee. Our rover mainline, however, operates on a an interrupt every 75ms and checks for data from one of the control sources depending on which mode is toggled (nunchuck, Bluetooth, etc.). The data check is a simple system call to the memory object mapped to each data source and thus is very quick. Our tracking algorithm yields only 4-5 FPS, however, and thus it would be impractical to initiate a search from this mainline.

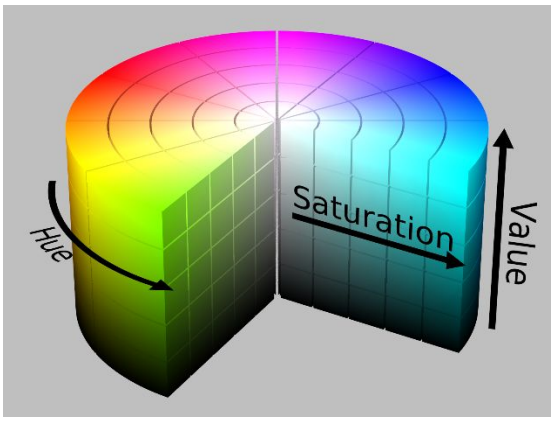
The solution to this would have been to have a separate process for object tracking, initiated on some command, that writes to a memory-mapped IO location or pipe that can be quickly checked by the rover mainline, thus decoupling the two actions. Unfortunately, due to some last minute design challenges we were unable to implement this feature, this preventing us from following the object. Our tracking algorithm still functions as expected, however, and is described below.

Procedure

Our procedure for tracking an object is as follows:

1. Open up a video steam from /dev/video0, the location of our webcam.
2. Capture a frame and convert it from the RGB color space to HSV. We do this because it separates color from intensity. HSV allows us to identify a color at any saturation level, while disregarding its value or brightness. This increases our ability to track objects in varying light conditions.





3. Apply algorithm and compute the center coordinates of an object. We then scale the x coordinate of this center to a value between -127 and 127 which is fed into a differential equation used for generating arcing travel paths by our motor controller.

Algorithm

We utilized two different algorithms for tracking objects for this demo. They are described below.

Color Range and Smoothing

This is the first algorithm that we developed as requires very little computational time. As a byproduct, it is also not very robust. The algorithm is as follows:

1. Apply the function inRange to the image which returns a 2D array the same dimensions as your image with a 1 in the (i, j) index if the pixel in that image was within some lower and upper bounds on the HSV color space and 0 otherwise. In other words, we generate a B/W image representing the pixels that were within the color region that we are looking for.
2. After doing so, we smooth the image by applying a series of dilating and eroding transformations. A dilation takes some kernel or restructure element (typically a square or circle) and scans it across the entirety of the image. If the restructuring element overlaps with at least one element in the foreground, the pixel at the center or anchor of the kernel is set to the maximum value within the kernel region. Erosion is similar, except the minimum value is selected. The effect is the following:

   Where the original element is in the center, the image on the right is after dilation, and the image on the right is after erosion. By applying a series of erosions, we get rid of small, independent blobs/noise that made it past our HSV filter, while the dilations smooth out the edges of each blob. This process yields an effective noise filter.
3. Calculate the (x,y) coordinate of this region. To do so, we borrow a concept from physics and generate the spatial moments of the image. These are defined as:

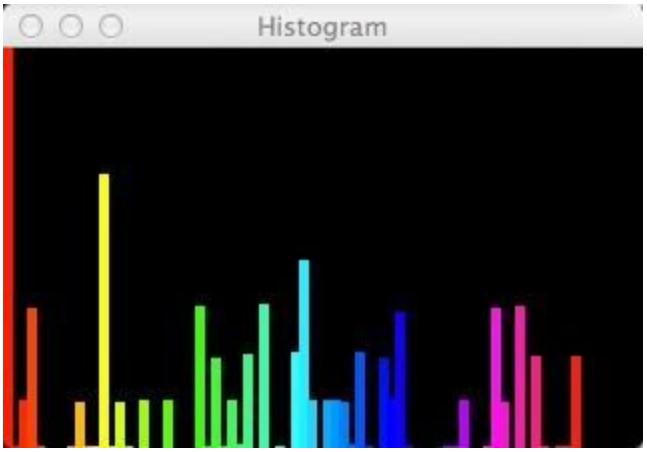
$$\mathtt{m_{ji}} = \sum_{\mathtt{x,y}} \left(\mathtt{array}\left(\mathtt{x},\mathtt{y}\right) \cdot \mathtt{x}^{\mathtt{j}} \cdot \mathtt{y}^{\mathtt{i}}\right)$$

   Array(x,y) is either 0 or 1. Thus m10 corresponds to the x moment, the m01 moment corresponds to the y moment, and m00 corresponds to the total area of the region of interest. Thus the x and y coordinates of the centroid of our region can be calculated as:
   X = M10 / M00
   Y = M01 / M00
4. We then scale this x value to between -127 and 127.

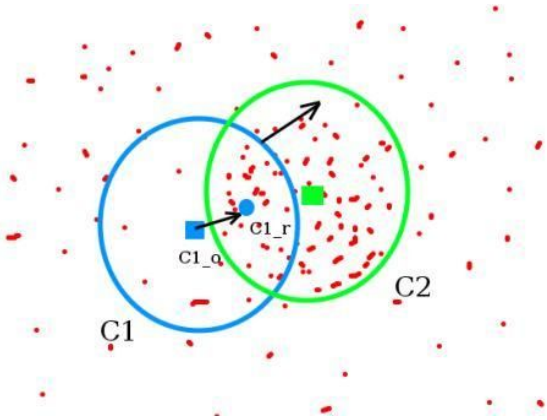
Camshift (Adaptive Meanshift)

This algorithm relies on more than a simple color pass and is thus slightly more CPU intensive. The benefit, however, is that we don't apply any noise filters and thus this process is less IO intensive. The net result is that it performs on par with the above algorithm on the BeagleBone with greater accuracy and robustness.

1. Calculate a histogram of a sample. In our case, our sample was a picture of just a Frisbee. A histogram simply scans the H and S color spaces, and bins values across each space (which range from 0-180 and 0-255 respectively). The process for the H channel, as an example, yields a histogram like this:



2. We can then create what's called a histogram back projection. Let us suppose the above histogram indicates that 5% of the pixels lie within the leftmost red bin. We then scan over the H channel of our image and replace all pixels that lie within that bin's range with the value 0.05, indicating that that pixel has a probability of 0.05 of bellowing to our reference image from which the histogram was computed. We then normalize these values to between 0 and 255. Doing this for each pixel in the H and S channels, we create a probability map ranging from 0-255, where regions that are more likely to below to our reference sample have larger values.
3. Now we apply the meanshift algorithm to our backprojection.
    a. We first select an region of interest where we expect the object to be (in this case the center of the image in a 50 by 50 pixel box).
    b. We then calculate the centroid of this region using the method described in the previous algorithm.
    c. Next we set our region of interest to the new centroid.
    d. Repeat steps a-c until convergence or a maximum number of iterations is met.



4. Next, we apply the camshaft algorithm which updates the size of our region of interest by

$$s = 2 \times \sqrt{\frac{M_{00}}{256}}.$$

Which allows for our window to grow and shrink as the object moves closer and farther away.

The net effect is a robust and lightweight algorithm which can effectively track a unique colored region in 3 dimensions. If the target exits the image, the algorithm resets back to a region of interest 50x50 in size centered in the middle of the image until a target is required.

2.7 MP3 Audio

Lastly, we implemented an MP3 Audio player centered around the VSLI VS1053B integrated circuit, which takes MP3 data and outputs the corresponding audio. We used linux spidev to communicate with the chip and were able to take a song or audio clip and play it through headphones.

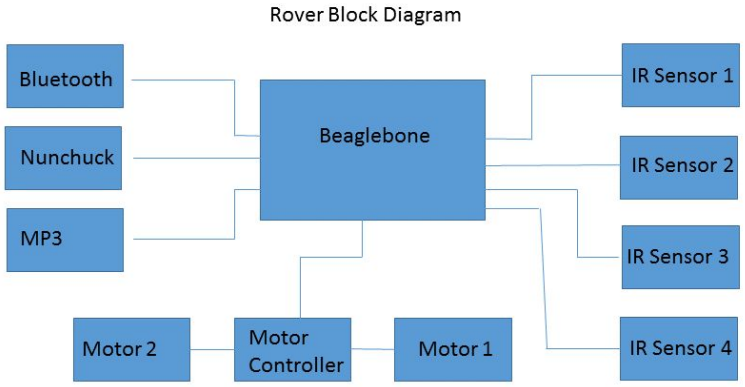
The integrated circuit has two different SPI busses to communicate with. One handles commands to specific registers of the chip and the other accepts data in 32 bytes chucks. There is a GPIO pin as an output of the MP3 chip that lets the processor know when it needs more data. When this pin is high, 32 bytes of data can be pushed to the device. If data is not passed fast enough, the audio becomes choppy.

The chip has enough power to drive headphones of at least 8Ω, which was sufficient for our use, but unfortunately, before demoing, the headphone outputs were shorted together resulting in a near zero resistance which bricked the device and rendered it unusable.

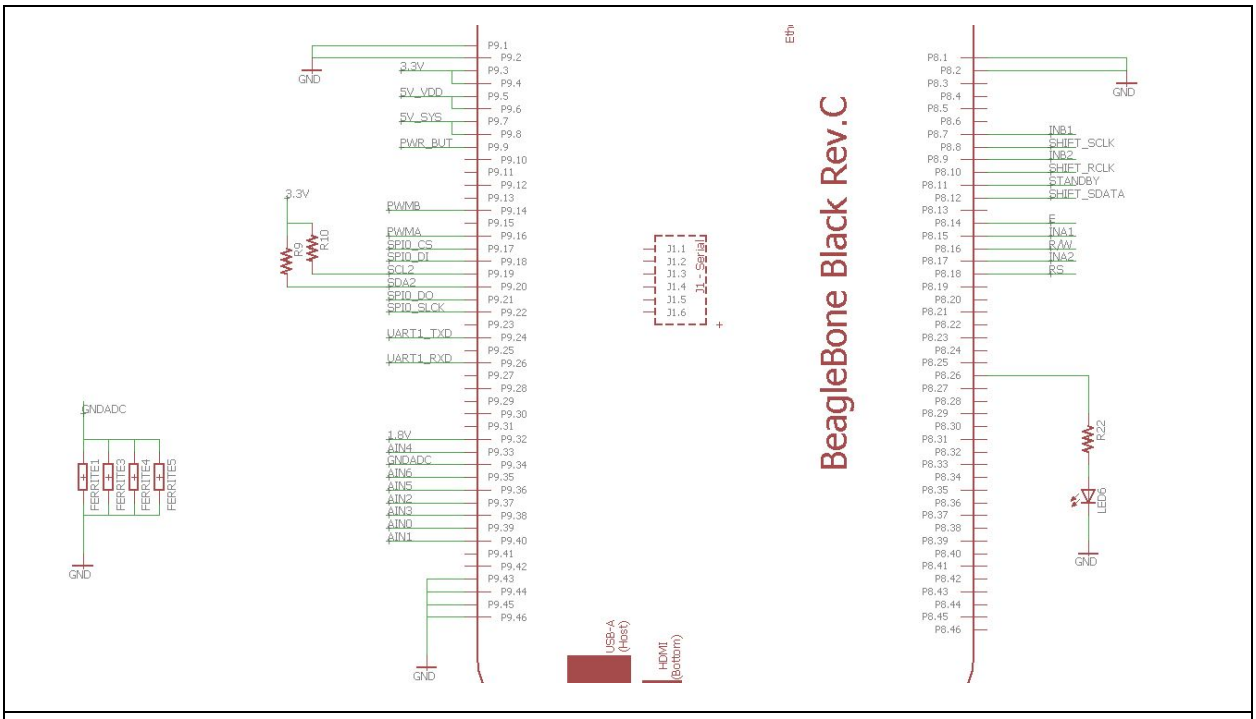
3.0 Conclusion

This lab showed our ability to work with 3 different communication protocols of the Beaglebone and linux in general. We used UART for the motor controller and bluetooth, SPI for the MP3 interface, and I2C for the nunchuck interface. We also successfully implemented analog circuitry to filter the noisy IR sensors to a consistent analog value so that our analog to digital conversion produced a useable value. This constitutes the majority of hardware peripherals included on the Beaglebone and the most common peripherals implemented on today's microcontrollers. The end result of our rover was pretty successful and the knowledge gained from working on the project will be valuable for future embedded systems projects. Knowledge of these communications protocols as well as knowledge of interrupts, GPIOs and PWM will allow us to work with the vast majority of microcontroller and microprocessor peripherals. In addition to the hardware knowledge gained from this lab, there was also significant software/firmware development knowledge gained from this lab. For example, running a script on startup is very important for embedded systems, especially if they are used in consumer products. In addition, we did significant experimentation and development with OpenCV, which is becoming very mainstream computer vision software, so our knowledge gained in this area will no doubt be useful in the future.

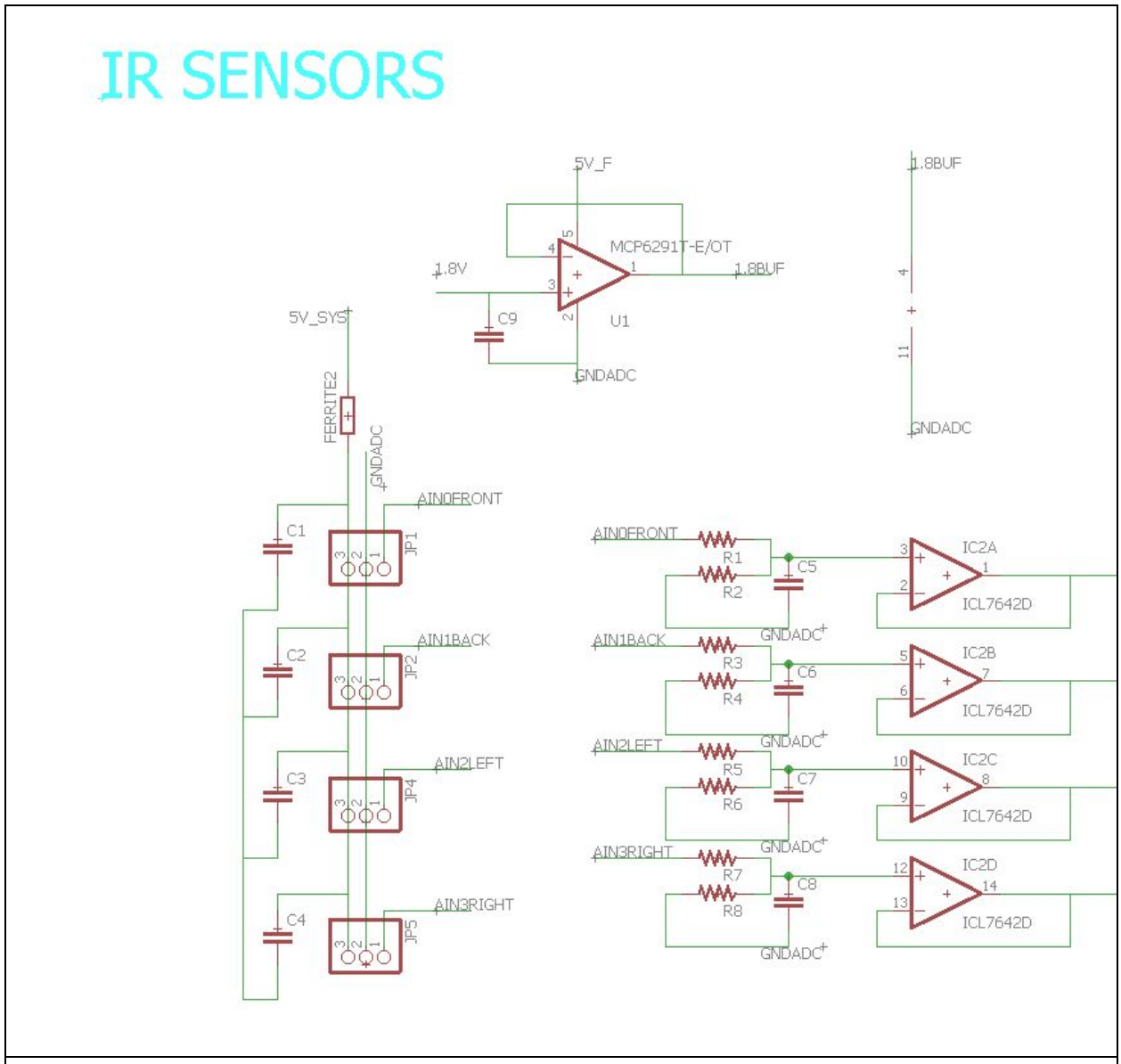
4.0 Appendices



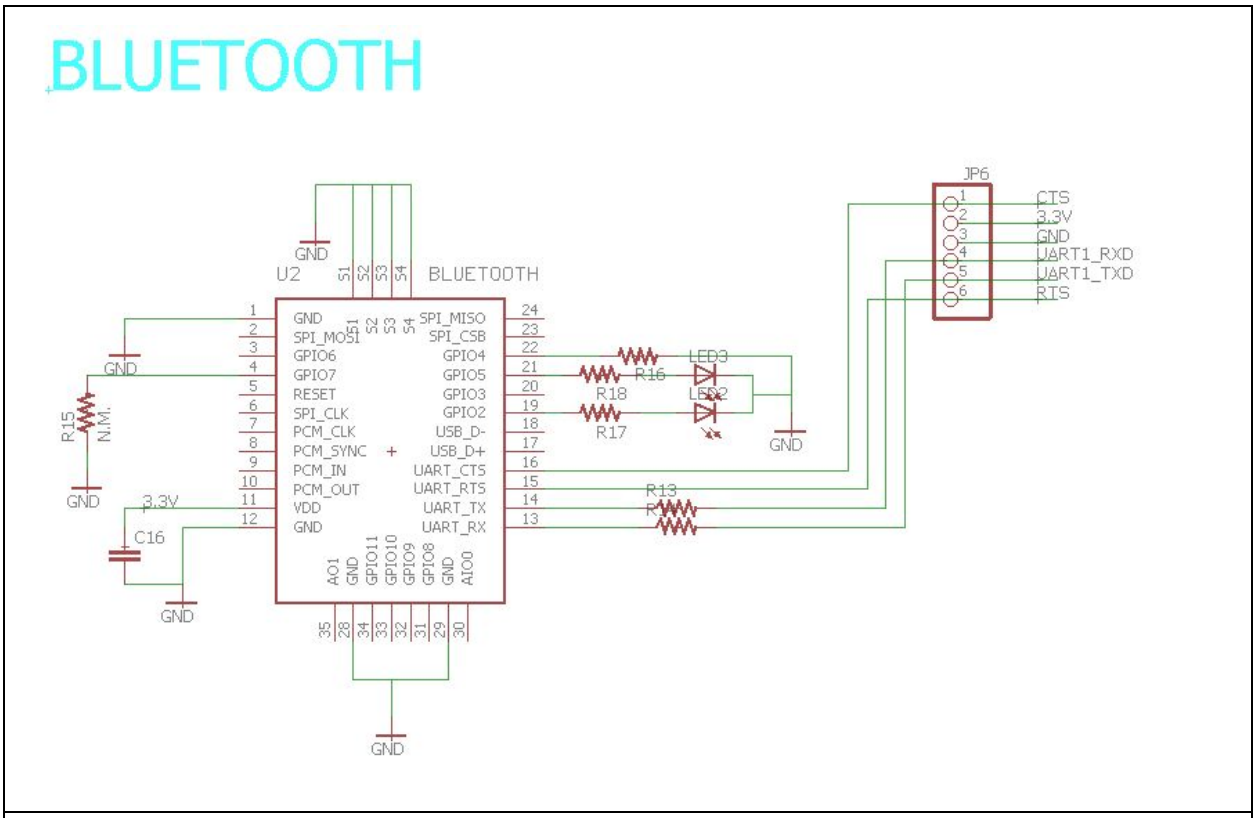
Appendix A: Block Diagram of the Rover System



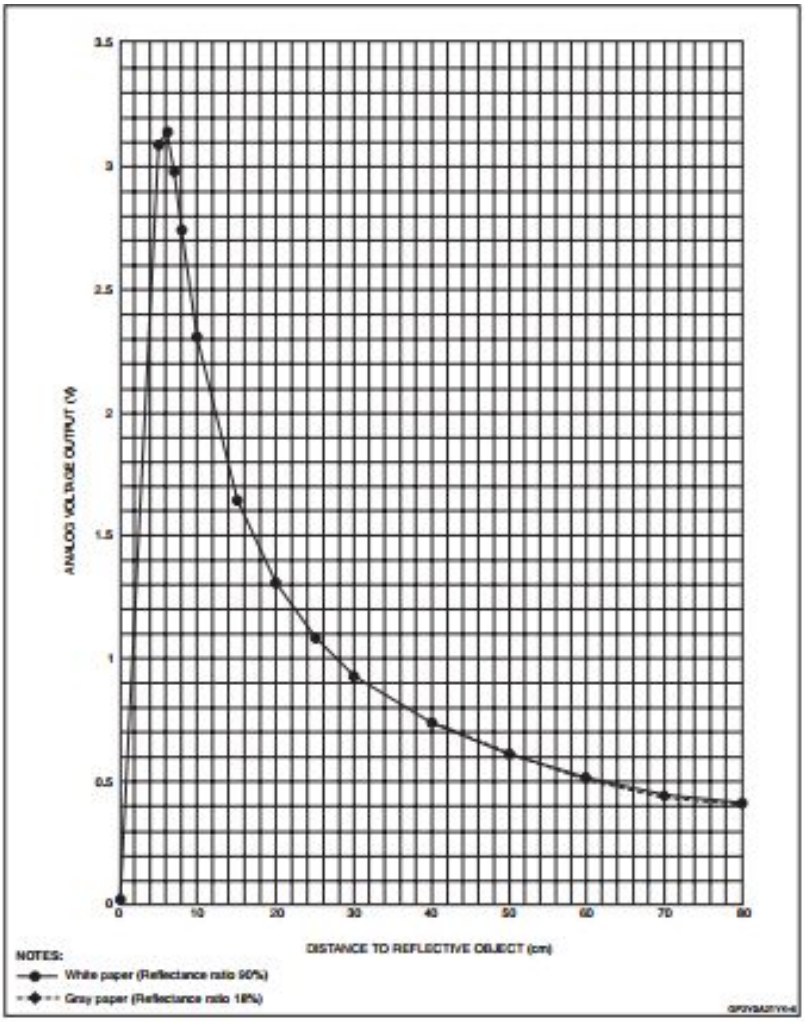
Appendix B: PCB Schematic Beaglebone Connections



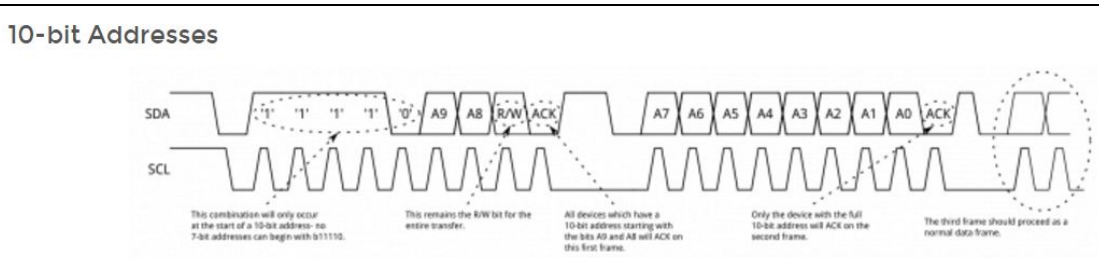
Appendix B: PCB Schematic IR Sensor Circuitry



Appendix B: PCB Bluetooth Circuitry



Appendix C: Output Characteristics of IR Sensor



Appendix D: I2C data bus example