Goose Chaperone

Final Report

Group Number: 17
Clients: Dr. Randall Geiger & Dr. Degang Chen
Advisor: Dr. Randall Geiger

Team Members:
Weston Berg
Zhihao Cao
Alec Morris
Johnson Phan
Woodrow Scott

Team Email: sddec19-17@iastate.edu
Team Website: https://sddec19-17.sd.ece.iastate.edu/
Revised: 12-10-2019

Project Design

For the structure of the robot, we are using polyvinyl chloride or PVC for short. For efficiency and cost, we are using the 1-inch solid core pipe version instead of sheet as it only requires drilling and cutting. As our design is only intended to scare and chase with no harm to any animal, it is only required to survive the environment. PVC is considerably lighter than most metal and more resistant to corrosion. It is rigid and durable and its pipe form can serve as covering for exposed wires and devices. For the current prototype, the wires have been left outside the PVC for ease of access during troubleshooting.

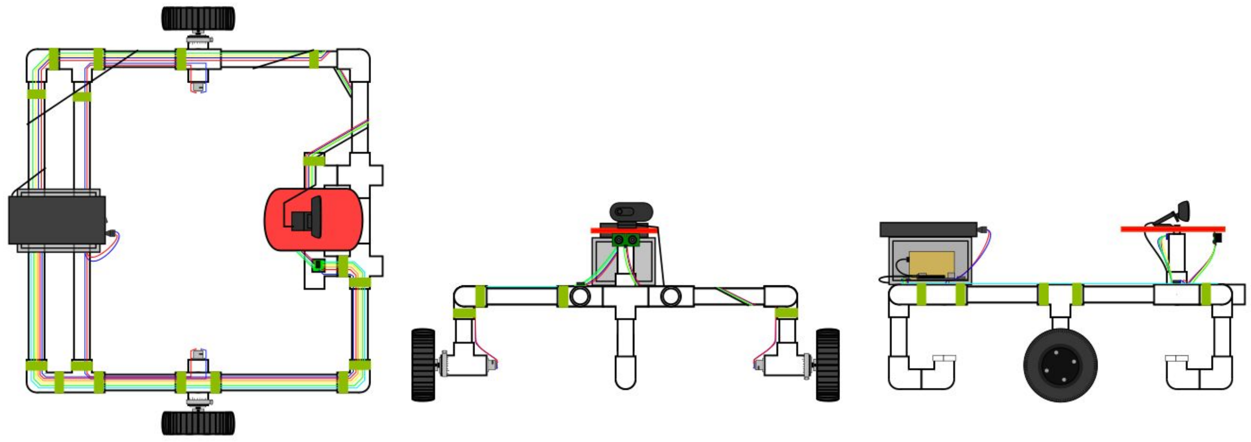

Figure 1: Diagram of chassis structure

The Beaglebone Black is the microcontroller controlling the data flow of the robot and is contained within an enclosed environment. This platform has both the functions required to collect data from the environment and meets the minimum memory needed to store or hold data collected from the environment. All of the following peripherals are connected to and extend the functionality of the BeagleBone Black. For environmental data collection, the Logitech C270 webcam and Ultrasonic Distance Sensor are utilized. The camera is used to identify geese and hazardous objects, such as water or boulders, while the ultrasonic sensor is used to check the distance of objects in the robot's surroundings. Validation of the robot's position is provided by the Adafruit Breakout GPS module. This is the most compatible GPS for our microcontroller. For movement of the robot, we are using two 12V brushed, geared DC motors. There a three different sources which provide power to the system. A 5VDC 2.1A li-ion battery pack provides power to the Beaglebone. Power is supplied to the motors via a 12VDC 3A li-ion battery pack. The 5VDC pins from one of the expansion headers on the Beaglebone power the GPS and ultrasonic sensor.

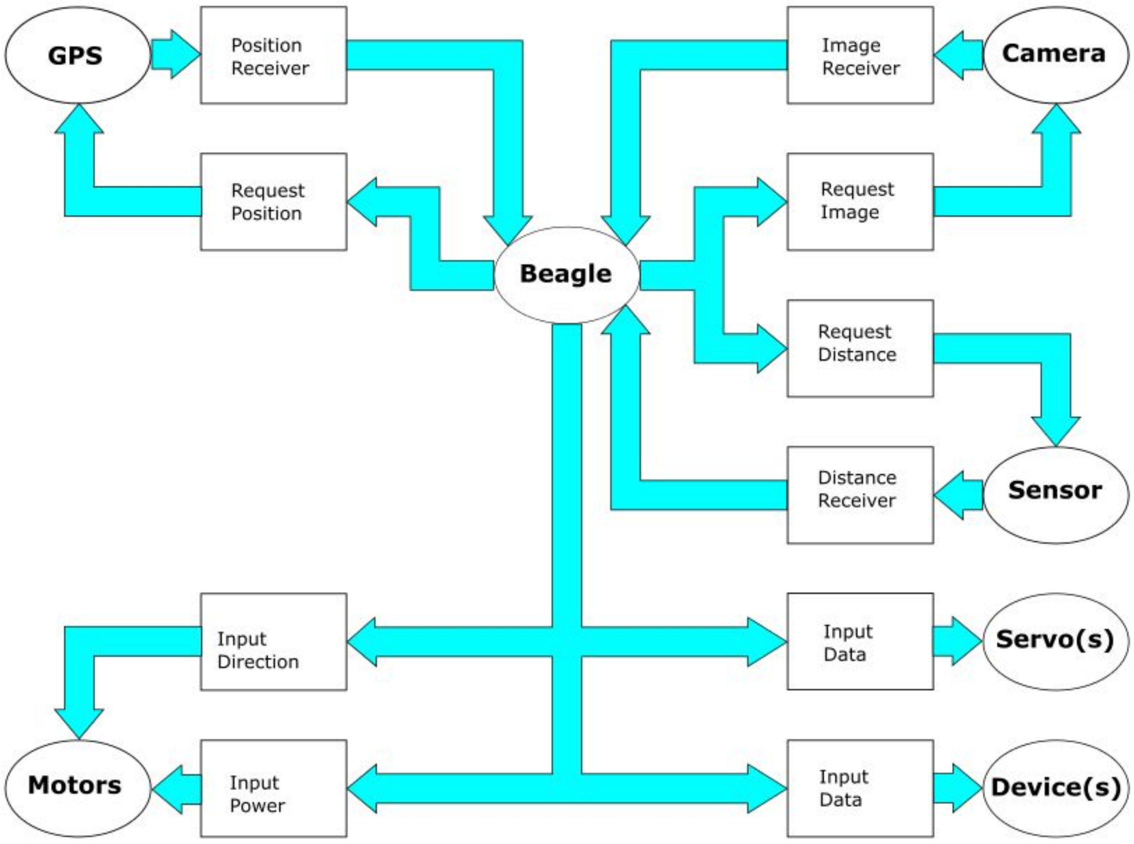

Figure 2: Block diagram of peripheral component connections

The software environment of the robot is relatively simple. Natively, the BeagleBone Black runs a distribution of Debian as its operating system. Debian, a 'Unix-like OS', provides a standardized framework for the rest of the robot's software to run in. The robot's mainloop is implemented in Python. There are several Python libraries that expose the necessary low-level interfaces for configuration, operation, and monitoring of the various external components that are attached to the microcontroller. Interfacing with these components is straightforward thanks to the structure provided by Debian. Two software libraries, Tensorflow and OpenCV, are utilized for image processing. Tensorflow is useful for picture categorization and determining if certain objects are present within and image. The categorized image data from Tensorflow is fed into OpenCV. OpenCV allows the robot to determine where the identified objects are in relation to itself. This is essential for accurate deployment of goose deterrence methods and safely avoiding hazardous objects.

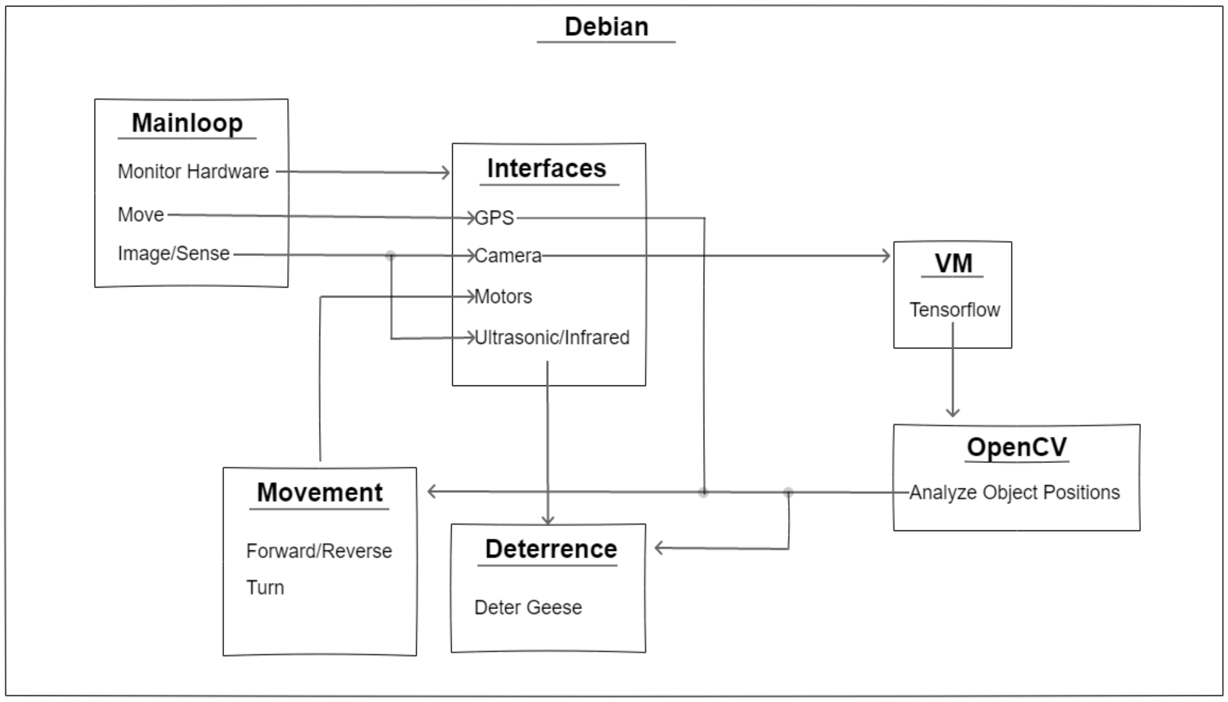

Figure 3: Block diagram of software environment

Implementation Details

Hardware

The structural design of the PVC model is built with intentions to ease friction on movement. Constructing with the wheels placed left and right of the model allows complete, 360 degrees rotation not possible with 4 wheels. Additionally, the resistivity and durability of the PVC pipes allow us to place it in the front and back for balance and less friction on the environment when rotating, backward, or forward movement.

The Beaglebone uses several different methods for controlling the system's peripherals. This is made possible via the two expansion headers and a USB header on the Beaglebone. The expansion headers consist of highly configurable General Purpose Input Output (GPIO) and several power and ground pins. This project used the expansion header pins for GPIO, Universal Asynchronous Receiver/Transmitter (UART), Pulse Width Modulation (PWM), and power/ground and the USB port to connect the camera.

The ultrasonic sensor only needs two GPIO pins to control it. The sensor takes distance measurements by sending out pulses of ultrasonic waves and waiting for the waves to bounce off objects and return to the sensor. Consequently, one GPIO pin is used to trigger the distance measurement and the other is used to receive the echo signal coming back. Once the echo signal is received, calculations by the Beaglebone determine if an object was detected and how far away it is. The Ultrasonic sensor has the capability to detect the distance between 2cm to 5 meters, the sensor will stop measure distance once distance between object and sensor is less than 2 cm. The purpose of the sensor is to get a general idea about what object is on the way to object and how far between object and sensor. The relationship between ultrasonic sensor and Beaglebone can be observed in *Figure 4*.

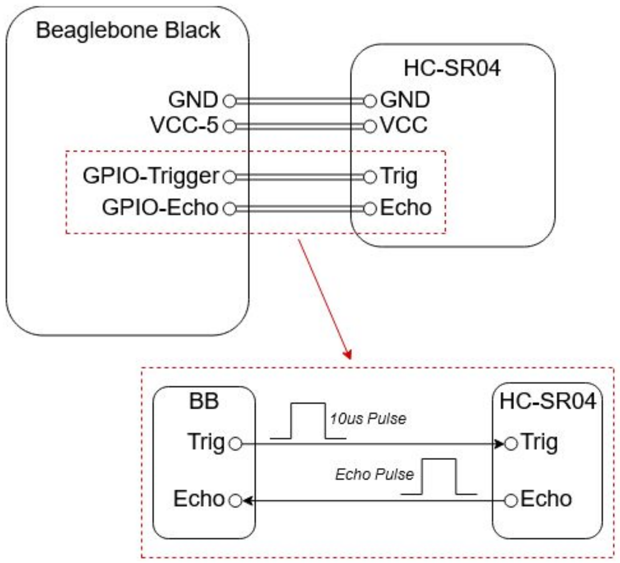

Figure 4: Diagram of connection between Beaglebone and Ultrasonic Sensor

The GPS is connected to the Beaglebone by a serial connection over UART as illustrated in *Figure 5*. The Beaglebone transmits commands to the GPS to configure its functionality. The GPS is configured to work using WAAS, which greatly improves precision as well as accuracy. The GPS streams location data back to the Beaglebone which is then parsed, analyzed, and acted upon. When this data is sent to the Beaglebone, much of it is filtered out through a Python script, and all that remains are longitude and latitude coordinates.

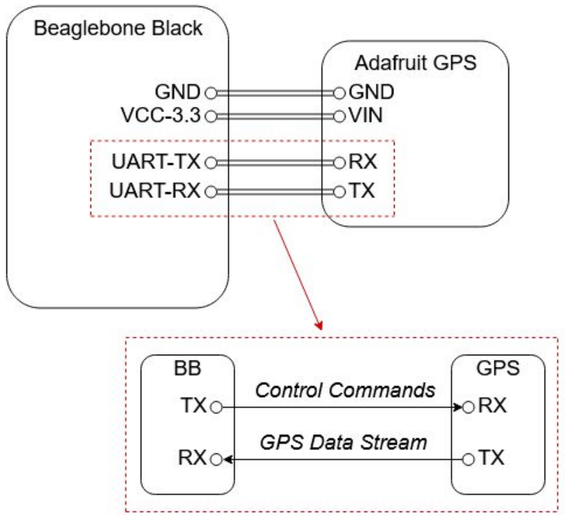

Figure 5: Diagram of connection between Beaglebone and GPS

A combination of GPIO and PWM signals are used to control the two DC motors through a motor driver board. The GPIO pins are used to select which direction the motors should turn and if the motors should brake. The PWM signals provide the actual power to the motor terminals and determine how fast the motors should spin. The Toshiba TB67H420FTG is the specific motor driver used on this prototype. The TB67H420FTG has two output channels that can provide 1.7A to each motor which is sufficient for the DC motors used. Changing the GPIO select signals changes whether the positive or negative line of each output channel is delivering the PWM. Applying the PWM to the positive terminal will cause the motor to spin forward and the negative terminal will cause the motor to spin in reverse. The entire motor control system is diagrammed in *figure 6*.

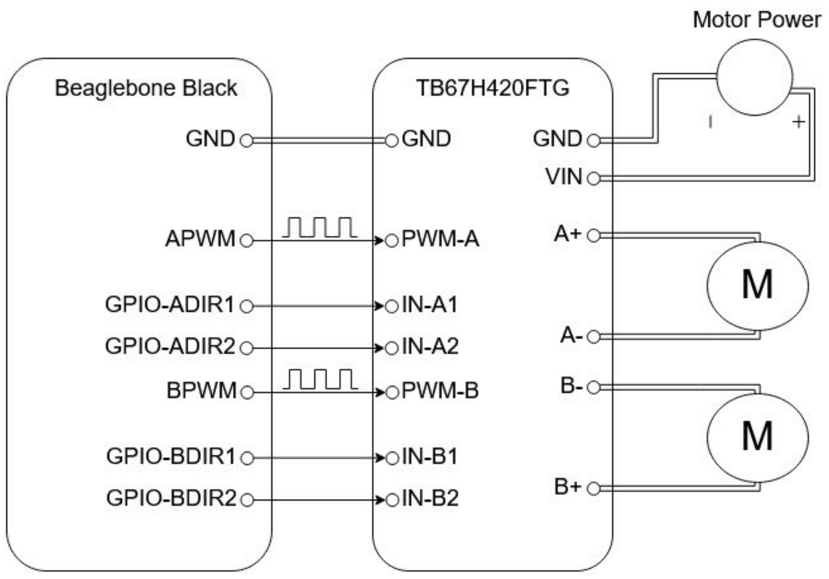

Figure 6: Diagram of the motor control system on the robot

Brushed DC motors were chosen to drive the robot because they are simple and inexpensive in comparison to brushless which makes brushed ideal for prototyping. A future design for this project may want to swap for brushless due to improved performance. The motors are also geared to a medium ratio, 1:100, to trade speed for more torque. The 1:100 gearing brings the motor's RPM down to 53. This gives us a theoretical robot speed of:

$$Rotation\ Distance\ =\ Wheel\ Circumference\ =\ \pi * 6in. = 18.85in.$$

$$Speed\ = RPM\ *\ Rotation\ Distance\ =\ 53\frac{rotations}{minute}\ *\ 18.85\frac{in.}{rotation} = 999.05\frac{in.}{minute} = 83.25\frac{feet}{minute}$$

This is on the slow side but for the purposes of this prototype this speed is sufficient. Future designs may want to increase the robot's speed since it will be travelling long distances across golf courses.

The DC motors are physically connected to the wheel via an axle. The axle ends in a circular hub which is bolted to the wheel's central hub. This provides a secure connection between motor, axle, and wheel. A single bearing is used at the connection between the axle and the chassis to take most of the robot's weight. Without this bearing all of the robot's weight would be on the motor's axle. The motor's axle is not designed to have that big of a load on it and would likely break. Two shaft collars are used on either side of the bearing to prevent axial load on the motor from the axle either getting pushed in or pulled out.

Software

There are three main parts to the software running on the Beaglebone: peripheral drivers, image processing, and the robot's mainloop. The peripheral drivers, tensorflow model, and mainloop run in the Beaglebone's Debian environment.

The peripheral drivers follow a singleton or object oriented (OO) pattern. In embedded systems, it is easy to think of peripherals in terms of these patterns. Take the system's camera for example. There is a single camera on the system that can be instantiated so it is intuitive to think of the camera like a singleton. If you need access to the camera's functionality all that needs done is to import the 'camera' module and call one of its methods (e.g. camera.capture()). For peripherals that could potentially have multiple instances on the system the singleton pattern will not work. This type of situation lends itself to an OO approach. A class defines the peripheral and its functionality and each instantiation operates independently of the others. The DC motors on the system are a good example of where this pattern makes sense. Each motor is an instance of the 'DCMotor' class and can change speed independent of the other instance. Source code examples from this project of both of these patterns can be found in *Appendix IV*.

Regardless of a singleton or OO pattern, most of the peripheral drivers utilize dependency injection. Dependency injection is used to specify which pins on the Beaglebone the peripheral will utilize. This simplifies initializations as a whole and makes changing which peripherals use which pin easier.

Tensorflow is instantiated in the main python file and opens the camera in an OpenCV session, as well as loading a tensorflow model and its associated detection tags. Images are taken at most in 100 millisecond intervals. Converted to numpy arrays and resized to a 300x300 pixel resolution, and are fed into the tensorflow model. The output is then converted into a 2D array. The first dimension being a list of detected objects, of which each is an array consisting of three parts in order: the id of the identified object, the confidence value of that object, and lastly its bounding box coordinates.

Testing Process and Results

Phase One Testing

The first phase of testing for the project consisted of components testing. Testing the components individually before integrating them into the robot achieved two goals:

1. Confirm compatibility of each individual component with the Beaglebone
2. Test functionality of software controlling component

The following are descriptions of how we tested each component and the results of that testing.

<u>DC Motor Testing</u>

Process

The plan to test the DC motor was to start on a set of motors smaller than what the final motors would be to see if PWM control through a dual motor driver board would be acceptable. Once this test was completed the full-scale motors and driver board would be tested.

Result

The small-scale test passed without problems. Control through a driver board was simple and accurate enough for this project. There were only a few changes needed to the software and the wiring to make the full-scale parts operable.

Stepper Motor Testing

Process

The stepper motor had required little testing. The plan was to simply hook up the driver board to the Beaglebone and test how accurately it can change angle.

Result

The stepper motor functioned as expected.

Camera, Tensorflow and OpenCV Testing

Process

Initial testing has largely been performed on computers with a USB webcam attached (of the same model used on the robot). Several architecture models were trained and tested. Initially, tests were carried out by feeding in a series of test images from our collection. Afterwards, with some kinks worked out, testing moved on to live camera feed, also testing various models. This was often difficult, as different architectures have different means of outputting results, and so a lot of additional code had to be written and tested to account for these differences.

Result

The final model currently installed uses the SSD Mobilenet V1 architecture trained on the COCO image base. Oddly, retraining the model breaks visually representing live recognition with bounding boxes. However, its output tensors are still available, and code was able to be written to extract all detection instances and their confidence scores. Bounding box coordinates are also extracted, but at this time are unused.

Ultrasonic Sensor Testing

Process

The initial test was done by connecting the ultrasonic sensor to an Arduino. This was to make sure the sensor itself worked properly. The next step was testing the sensor on the Beaglebone with Python code.

Result

The test with Arduino was successful, the sensor reflect the distance by meter. This test makes sure that the ultrasonic sensor do not have any hardware defect. Then test with Beaglebone by

Python code to obtain distance from 2cm to 5 m. and once the distance goes below 2cm the measurement program would be terminated.

GPS module Testing

Process

The GPS module was tested using the following protocol; After being connected to the Beaglebone board, and a proper Python program was developed, GPS location data was harvested from the module over the course of several minutes in a few distinct locations. This data was then compared to the actual coordinates provided by Google maps.

Result

The test was successful, as the average GPS reading was determined to be within an acceptable radius of the actual location. The distance that was determined to be acceptable was a 10 foot radius around the actual location.

Phase Two Testing

The second phase of testing for the project consisted of integration testing. All components were integrated onto the chassis and hooked into the Beaglebone.

The following are the integration tests which we performed and the results of those tests:

Movement Testing

Process

Once the motors and the control system are integrated onto the chassis the ability for the robot to go forward, reverse, and turn by a certain angle would be tested.

Result

After testing and troubleshooting, the motors are unable to move the robot around. This could be due the motors being too weak to move the chassis, improper connection of the motor shaft to the wheel axle, and/or bad wiring to the motor limiting the power provided to the motors.

Navigation Testing

Process

Navigation would have been tested by creating a route for the robot to take via GPS coordinates, uploading the coordinates onto the robot, and confirming that the robot followed the route with some margin of error.

Result

Navigation was not able to be tested.

Safety Testing

Process

Test the system to make sure no functionality could cause harm to humans or the robot's operating environment. These tests include ensuring the robot can recognize humans and would never deploy tactics if a human was detected and verifying the robot cannot deviate from its set route into prohibited or potentially dangerous areas, such as onto a neighboring road.

Result

The image processing software has been proven to recognize humans; however, we do not have any deterrence capabilities on the robot yet so that portion remains untested. Route verification was also not able to be tested.

Related Products

Goose Guardian

http://www.gooseguardian.com/

This product has a very similar goal to ward off geese from properties. Technology-wise, it also utilizes a camera to identify geese. Upon detection, it triggers a 'hazing' device that spins a rod to scrap a flexible material on its surface. According to the manufacturer, this is to train geese to "associate their behavior with the scary motion and sound produced". Where this product diverges from our project is that the 'Goose Guardian' does not patrol but is a stationary deterrence device.

Bird Control Robots

https://smprobotics.com/

This series of products are visually much more similar in design. They employ autonomous robots on wheels that use image recognition to detect birds and employ scare tactics to disperse them. They also follow guided routes when patrolling. The scare tactics described by the manufacturer depends on the model, but three listed on their website give the choice of lasers, a gas gun, and a scare-call mimic. This product is essentially identical to our project.

Appendix I

Appendix I consists of a manual detailing how to setup and operate the robot.

Setup

1. Power on ROMOSS (smaller white) power bank
2. Power on TalentCell (larger black) power bank
3. USB cable connection from laptop or computer to Beaglebone
4. If user does not have Putty, install Putty
5. Set IP address : 192.168.7.2, Port: 22 if not at default and open
6. Login as: debian, password as: temppwd
7. Command prompt: cd goose_chaperone/software
8. Edit the GPS.csv file to populate with sequential GPS coordinates to create a guide route to follow (not implemented).

Operation

1. Flip the power switch to both batteries to the 'On' position
2. The logic board will begin booting to Linux immediately. Once ready, the main robot logic program will be launched.
3. While the program loads its dependencies, there are several LED signals that will flash on the board indicating its status. These are as follows:
    a. BOOT UP : This is the first signal that should flash as the program starts. It is indicated by six quick flashes.
    b. START: This indicates that the programs logic loop has started. This is indicated by ten very fast flashes, afterwards the light should remain lit.
    c. START FAILED: If this occurs following the boot sequence, then an exception has occurred and the program is about to exit. If this occurs, attempt powering down the Beaglebone, unplug it from the power supply, hold the reset button down for 10 seconds, and then power it back on. This is indicated by five long flashes, and afterwards the indicated LED should turn off.
    d. TARGET DETECTED: This occurs when the robot detects a target, as reported from tensorflow. This is defined by a constant stream of very quick flashes while a target is in sight. Afterwards, the LED will remain lit.
    e. OBSTACLE DETECTED: Taking precedence over the TARGET DETECTED signal, this will flash slowly while the ultrasonic sensor detects an obstacle nearby.
    f. SHUTDOWN: While shutting down (not currently implemented without powering down the beaglebone, but works while debugging) the indicator LED will flash slowly five times, and remain off afterwards.
4. The robot will begin searching for targets to chase.
5. To shut down, turn the power to the board and motors off by flipping their respective power switches on their batteries.

Appendix II

Appendix II consists of discussion on design iterations prior to the final design.

Initially, designs for the robot involved the use of a pre-built chassis. The team was partial to using a pre-built chassis over constructing our own from scratch. Using a pre-built would allow us to focus our time and effort on the more important details of the project, such as the image processing software. The chassis under consideration were made up of a frame, usually of aluminum, and DC motors with wheels. The frames had easy attachment points for the motors/wheels as well as any sensor that we needed mounted.

There were several problems with these pre-built platforms. The first problem was the size of the chassis. Many of the pre-builts were designed for small, hobby robots that will operate indoors. The more robust platforms that could be utilized outdoors were harder to find, still had too small of a profile for our client, and cost a lot relative to our budget. This leads us to the second problem of cost. From our research, the smaller, robust platforms which can operate outdoors with a load of 15-20lbs were priced around $150-$200 at a minimum. To get the bigger profile that our client desired the cost would have gone up at least another several hundred dollars, consuming our entire budget for the project.

Due to these problems, we concluded that the chassis for the robot must be built entirely from scratch to save money and get the size that the client desired. This was unfortunate because it meant the team must now designate a significant amount of time to the design and implementation of a chassis, something no team member had attempted before.

Appendix III

Appendix III consists of discussion on miscellaneous topics related to the project not mentioned above.

Appendix IV

Appendix IV consists of relevant examples from the project's source code.

Important API callables

comvis.py
get_detections()
Parameters: None
Return: A 2D array of lists detailing detected objects. Each element of the array (r) contains three values:

r[0] contains a class id, referenced by the mscoco_label_map.pbtxt file, in which filtered results currently allow ids 1 (humans) and 16 (birds).

r[1] contains its confidence score as a floating point between 0 (no confidence) and 1 (total confidence). Currently, only scores 0.5 and above are returned.

r[2] is an array of four values representing bounding box corners.

get_box_area(r[2])
Parameters: An array of bounding boxes from element r.
Returns: A floating point that calculates the area (size) of the object detected on screen.
Note: Does not currently give reliable results.

distance_sensor.py
init(trigger, echo)
Parameters:
    Trigger: Integer referencing pin for ultra-sonic sensor trigger line (input)
    Echo: Integer referencing pin for ultra-sonic sensor echo line (output)
Result: Initializes sensor

detect_distance()
Parameters: None
Result: Distance from an object in range in meters.

TB67H420FTG_motor_driver
init(l_channel, l_select1, l_select2, r_channel, r_select1, r_select2)
Parameters:
    l_channel: PWM pin controlling left DC motor's PWM signal
    l_select1: GPIO pin controlling direction of left DC motor
    l_select2: GPIO pin controlling direction of left DC motor
    r_channel: PWM pin controlling right DC motor's PWM signal
    r_select1: GPIO pin controlling direction of right DC motor
    r_select2: GPIO pin controlling direction of right DC motor
Result: Initializes motor drivers

set_speed(speed_setting, direction)
Parameters:
    Speed_setting: Set motor speed
    Direction: 0 for forward, 1 for reverse
Result: Set speed and direction

brake()
Parameters: None
Result: Break all wheels

turn(degree, direction)
Parameters:

degree: Degrees to turn
Direction: direction to turn (TurnDirectionEnum.LEFT or TurnDirectionEnum.Right)

cleanup()
Parameters:
None
Result:
Free resources

uln2003_stepper
rotate(degrees, rpm)
Parameters:
degrees: Degrees to turn
rpm: Speed to turn
Result:
Rotate stepper motor by degree and speed

Utility Programs

ImageScraper
Arguments:
Query: Image query to search for
StartPage: Google result page to skip to
Count: Number of pages to search
Result:
Searches the internet and downloads into folders named after the query input. Limited to 100 results per day.

Example of the camera module using a singleton pattern:

```
'''Module for controlling the capture device on the system'''
# NOTE  'Frame' objects returned from VideoCapture.read() consist of
#       a matrix of data representing the image. This matrix can be
#       manipulated/analyzed with various OpenCV functions.
import cv2

_g_CAPTURE_DEVICE_INDEX = 0  # Only one camera on system so '0' index
_g_INIT_FRAMES = 5  # Number of throwaway frames to initialize camera

def init(cap_delay):
   '''Creates handle and initializes capture capture device'''
   global _g_vcap  # Handle for capture device
   global _g_cap_delay_ms  # Delay between batch captures in ms
   # Create the object for capturing frames
   _g_vcap = cv2.VideoCapture(_g_CAPTURE_DEVICE_INDEX)
   if not _g_vcap.isOpened():
      raise RuntimeError('Unable to connect to camera.')
```

```
    _g_cap_delay_ms = cap_delay
    # Set device properties here if necessary (i.e. frame width/height)
    # Initialize device by capturing dummy frames
    for i in range(_g_INIT_FRAMES):
        _g_vcap.read()
        cv2.waitKey(_g_cap_delay_ms)  # Delay

def release():
    '''Releases the handle on the capture device'''
    _g_vcap.release()

def single_capture():
    '''
    Captures a single frame from the capture device
    :returns: True if frame read correctly along with corresponding frame
    '''
    return _g_vcap.read()

def batch_capture(num):
    '''
    Captures a series of frames with fixed delay between each frame. If an
    invalid frame is encountered in will be thrown away with no retry.
    :param num: Maximum number of frames to capture
    :returns: List of valid frames captured
    '''
    frame_list = []
    if num:
        for i in range(num):
            success, frame = single_capture()
            if success:
                frame_list.append(frame)
            cv2.waitKey(_g_cap_delay_ms)  # Delay
    return frame_list
```

Example of DCMotor class using object oriented pattern:

```
'''Driver for initializing/utilizing DC motor with TB67H420FTG motor driver'''
import Adafruit_BBIO.GPIO as GPIO
import Adafruit_BBIO.PWM as PWM

_g_SPD_DIR_VALUES = [0, 1, None]    # Valid values for speed direction

class DCMotor(object):
    # PWM frequency in HZ
    # Max PWM freq for TB67H420FTG Motor Driver = 100 kHz
    _PWM_FREQ_HZ = 20000

    def __init__(self, channel, select1, select2, dcycle_map):
        '''
        Initializes DC motor for PWM communication with specified pin
        :param channel: PWM pin controlling DC motor's PWM signal
        :param select1: GPIO pin controlling motor direction
        :param select2: GPIO pin controlling motor direction
        :param dcycle_map: Mapping between speed setting and duty cycle
        '''
        self.channel = channel
        self.select1 = select1
        self.select2 = select2
        self.dcycle_map = dcycle_map
        self.speed_setting = MotorSpeedEnum.STOP

        # Configure PWM channel with initial duty cycle of 0
        PWM.start(channel=self.channel, duty_cycle=0, frequency=DCMotor._PWM_FREQ_HZ)
        # Configure GPIO pin controlling motor direction
        GPIO.setup(self.select1, GPIO.OUT)
        GPIO.setup(self.select2, GPIO.OUT)
        # Initially set motor direction to forward
        GPIO.output(self.select1, GPIO.HIGH)
        GPIO.output(self.select2, GPIO.LOW)

        print('DC Motor Initialized - PWM: %s IN1: %s IN2: %s'
              % (self.channel, self.select1, self.select2))

    def set_speed(self, speed_setting, direction=None):
        '''
        Sets the motor speed for the motor
        :param speed_setting: Speed setting to set motor to
        :param direction: Motor direction
                    0 for forward, 1 for reverse, None for no change
        '''
        # Ensure proper function usage (can be removed/refactored later)
        assert (direction in _g_SPD_DIR_VALUES), 'Invalid direction' \
                                                 ' for motor speed'
        # Get the appropriate duty cycle
        dcycle = self.dcycle_map[speed_setting]
        # Set direction if necessary
        if direction is not None:
            if direction == 0:  # Forward
                GPIO.output(self.select1, GPIO.HIGH)
                GPIO.output(self.select2, GPIO.LOW)
            else:  # Reverse
                GPIO.output(self.select1, GPIO.LOW)
                GPIO.output(self.select2, GPIO.HIGH)

        # Perform speed change
        PWM.set_duty_cycle(self.channel, dcycle)
        # Track active speed setting
        self.speed_setting = speed_setting
```

```
        print('Set PWM \'%s\' duty cycle: %d%%' % (self.channel, dcycle))

    def brake(self):
        '''Brakes the motor'''
        GPIO.output(self.select1, GPIO.HIGH)
        GPIO.output(self.select2, GPIO.HIGH)
        PWM.set_duty_cycle(self.channel, 0)

    def cleanup(self):
        '''
        Shuts down DC motor
        '''
        PWM.stop(self.channel)
        GPIO.output(self.select1, GPIO.LOW)
        GPIO.output(self.select2, GPIO.LOW)
        print('DC Motor shutdown')
```