SOEN 422/2 Fall 2015

Final Project

Team members :
Samuel Beland-Leblanc (7185642)
Cong Lu (3080021)

Contents

1 Introduction

This is the overview of our final project, which is dedicated to building a wheeled robot that will be able to simulate a car's auto-pilot.The purpose of this project is to introduce our group's conceptual design and practical hands-on work. For instance, from a variety of software and hardware that we have used to build up the vehicle, to how we gathered and assembled them to finally work. What is to be shown is the hardware design, the software design based on the hardware, the tools used and the final working prototype.

2 Project Description

2.1 Name and Purpose

Our unit is called "Sam" and it is an electric car capable of piloting itself (i.e. has an auto-pilot).

2.2 Unit Function and Performance

In this group project, we attempted to build up a vehicle that is capable of moving on a designated path (lane) guided by webcam. The vehicule should drive at a slow to moderate speed centered in the lane, adjust itself if it's crooked and adjust it's direction when it gets in a curve. It should also detect standard object like a stop sign and react accordingly. If more than one lane is present, the vehicule should be able to safely change lane also.

Also, the vehicle should be aware of any obstacles around it. Using a set of sonar, the car should constantly detect objects in a specific range and should stop or adjust its direction (e.g. avoid a still obstacle) accordingly.

2.3 Project Hardware

Here are the major components of the system :

Servo Motor Used to move the webcam's field of vision

Webcam Allows to take pictures of the ground in front of the car to be analysed in order to adjust the car's direction

Ultrasonic Sensor (sonar) Detect objects in a specific range

Battery To power the system independently

H-Bridges To drive the motors (direction and speed)

Motors and Wheels To move the vehicle

Teensy++ 2.0 To drive all the core hardware (H-Bridges,servo and Sonar)

BeagleBone Black Captures image from webcam and analyses it in order to produce a specific instruction and send it to Teensy to be carried out.

For any illustration in regards to these components, see section 3. The battery was listed in case we needed it, but we never actually used it (because we couldn't charge it). We simply used the 5V from a USB power adapter and it was perfect since we didn't want extreme speeds. The eventuality of using the battery has been taken into account in section 3.1 though.

2.4 Project Software

The overall software is split across the Teensy and the Beaglebone. On the Beaglebone, the software takes care of capturing an image and analyzing it to extract specific features through OpenCV. These features are then analyzed and a next action to take is determined and sent to the Teensy via I^2C . On the Teensy, the software reads and analyses the sonar's readings, drives the servo motor to the right angle and drives the H-Bridges to get proper speed and direction based on the instruction received from the beagle. It is also polled by the Beaglebone through I^2C to know if the car should stop because there is an obstacle in front.

2.5 Scope of Completion

At the present, we have successfully made the sonar sensor work as it could react to the object in front of the vehicle and make the car stop. In addition, OpenCV is implemented on the Beaglegone with the webcam to instruct the vehicle (i.e. the Teensy) of any action it should take from the images captured. When the vehicle is crooked or in a curve, it stops and starts looking where it should go. When it detects the angle it needs to turn(i.e. the angle the webcam needed to turn before detecting it was "straight" again), the vehicle would turn as much as the webcam and then continue going forward or readjust itself if it turned too much.

Unfortunately, the software components are not optimized enough and the reaction of the system is somehow sluggish. When the software was tested using a PC, everything worked perfectly but the Beaglebone black ended up being slower than expected. But, technically speaking, it works.

3 Hardware Design

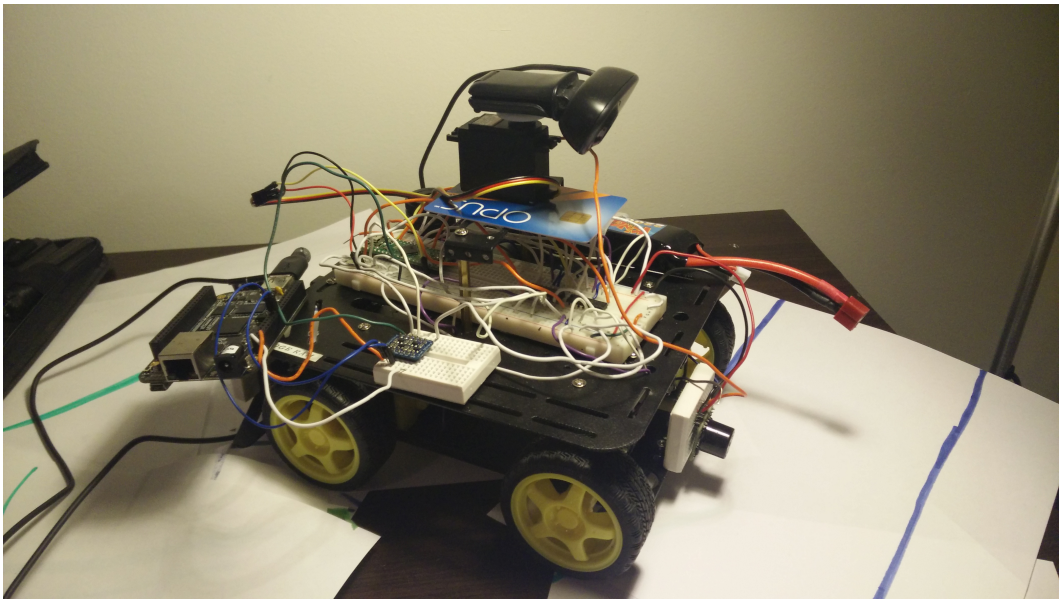


Figure 3.1: Final working prototype

This section will represent the actual hardware of a working prototype. It will not reflect any possible feature but rather the features that are working. For that manner, it will be less complex than our initial proposal.

3.1 System Design

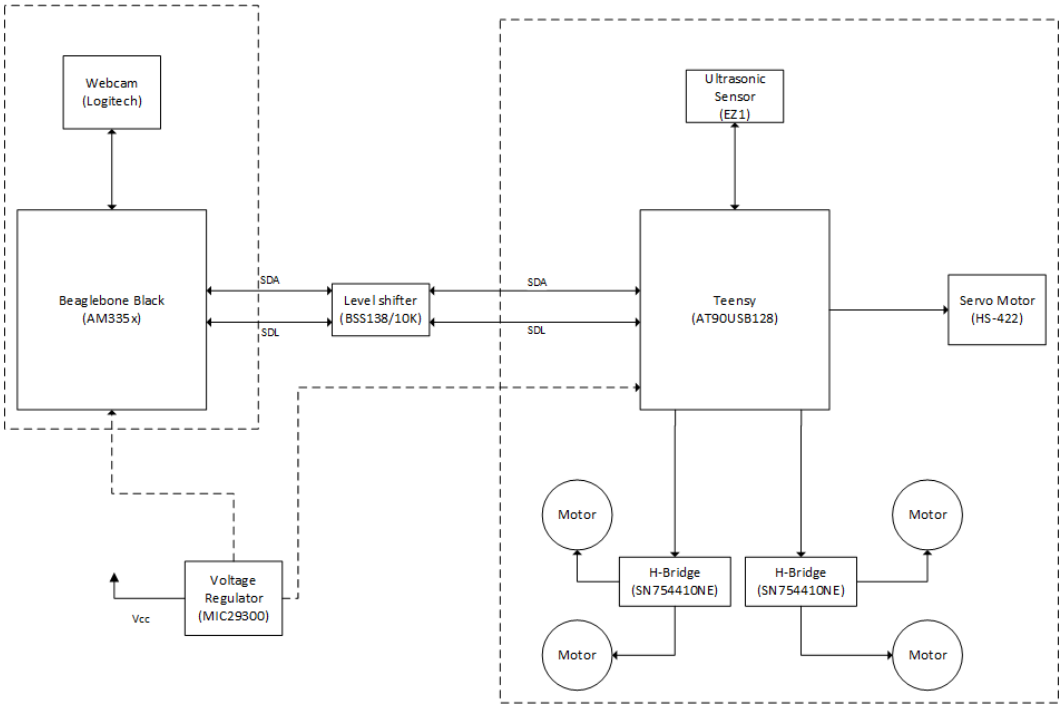


Figure 3.2: Main system's block diagram

The whole system's workload is split across two embedded systems, the BeagleBone Black (with the AM335x ARM processor) and the Teensy++ 2.0 (with the Atmega AT90USB128 processor). Both processor handle their own subsystem (represented by dashed box in figure 3.2). There is an additional voltage regulator in the diagram in case an input voltage over 5.0V is used (e.g. using a battery). Our working prototype does not need voltages higher than 5.0V as per it's functionnalities, so it is not used. Both subsystem are interconnected with an I^2C bus running through a level shifter to adapt between the 5.0V from the Teensy to the Beaglebone's 3.3V operating voltage.

The Beaglebone's subsystem simply uses a Logitech webcam. It's main function is to capture a stream of images from the webcam and analyse them to determine the next action to take. When the data has been analysed, it instructs the Teensy of what it should do through it's 2^{nd} I^2C bus. It also polls the Teensy before each capture loop to see if it should stop analyzing or not (e.g. There is an obstacle in front of the car).

The Teensy's subsystem is the one that is controlling the motors and the sensors. It uses the internal timers of the Teensy to control both H-Bridges, where the motors and wheel are attached. These H-Bridges are connected to the same 5V source as the rest of the subsystem because the motors didn't need extra speed. Again, using the internal timers, it controls the servo motor on which the webcam is mounted. Using the internal ADC, it reads the value from the ultrasonic sensor to detect object in from of the car. This subsystem receives commands through the I^2C

bus to know which action to take. Although, if the sensor detects something close, the subsystem stalls and updates it's internal state so that when the Beaglebone polls, it gets notified.

3.2 Subsystem Design

3.2.1 BeagleBone Black Subsystem

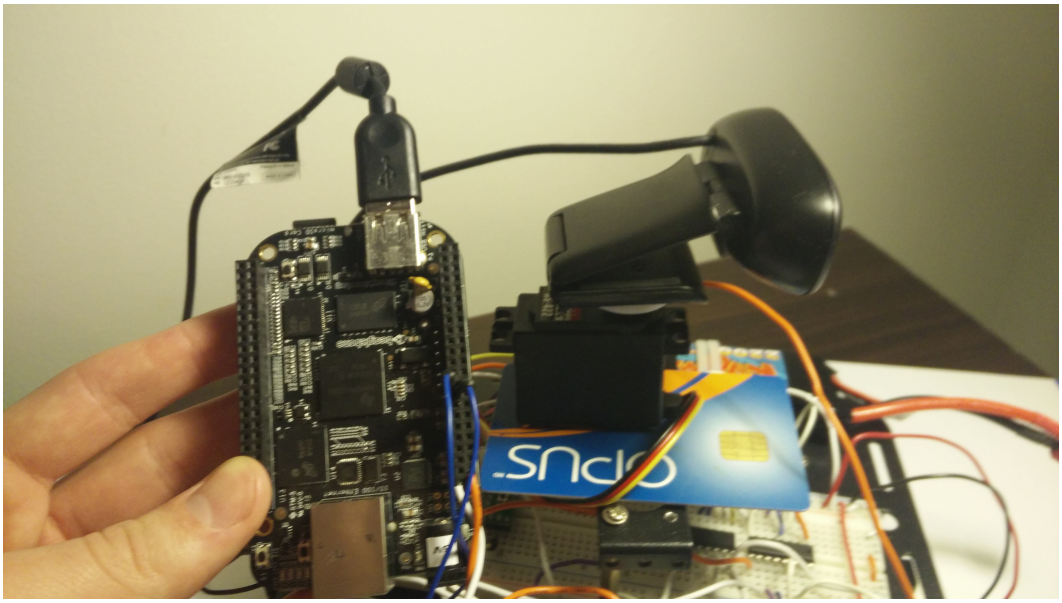


Figure 3.3: BeagleBone Black subsystem

The Beaglebone's subsystem is not complex. It is concise but uses the superioir computation power of the ARM AM355x processor for images analysis.

3.2.1.1 Beaglebone Black

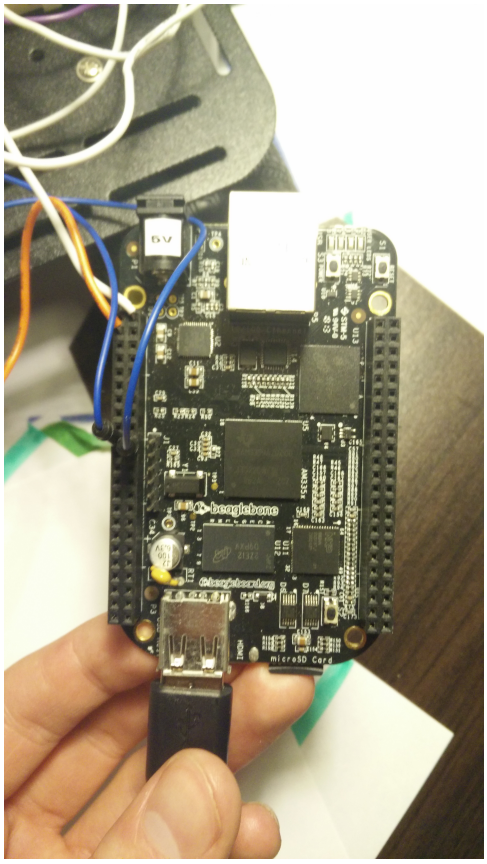


Figure 3.4: Beaglebone black used for the project

Embedded system with the integrated Sitara XAM3359AZCZ100 processor. Amongst the multitude of components contained in the system [2], here are the one used on this prototype (to avoid saturating this document) :

- Sitara XAM3359AZCZ100 processor at **1GHZ**
  - To do heavy image analysis (relative to embedded systems computational power)
  - To be able to run a modern operating system (Linux) and use different programs/libraries
- USB 2.0 Host Port
  - To connect the Logitech webcam
- Ethernet Port
  - To control the Beaglebone Black over a local network
- microSD card reader

  - To load more programs on top of the OS, since the internal memory is limited
- Two I^2C buses
  - To communicate with the Teensy

3.2.1.2 Logitech Camera

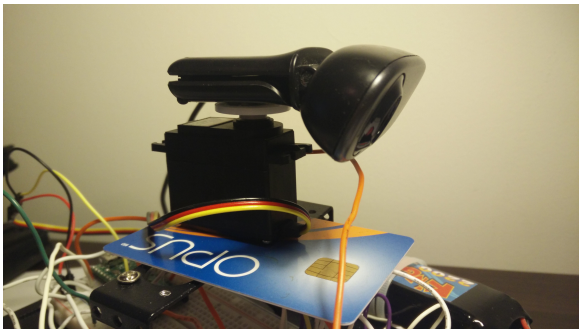


Figure 3.5: HD Logitech webcam mounted on the servo motor

A simple HD webcam powered over USB. Provides a good quality to analyse the surrounding environment. It is mounted on the servo motor in order to have a wider range to capture.

3.2.2 Teensy Subsystem

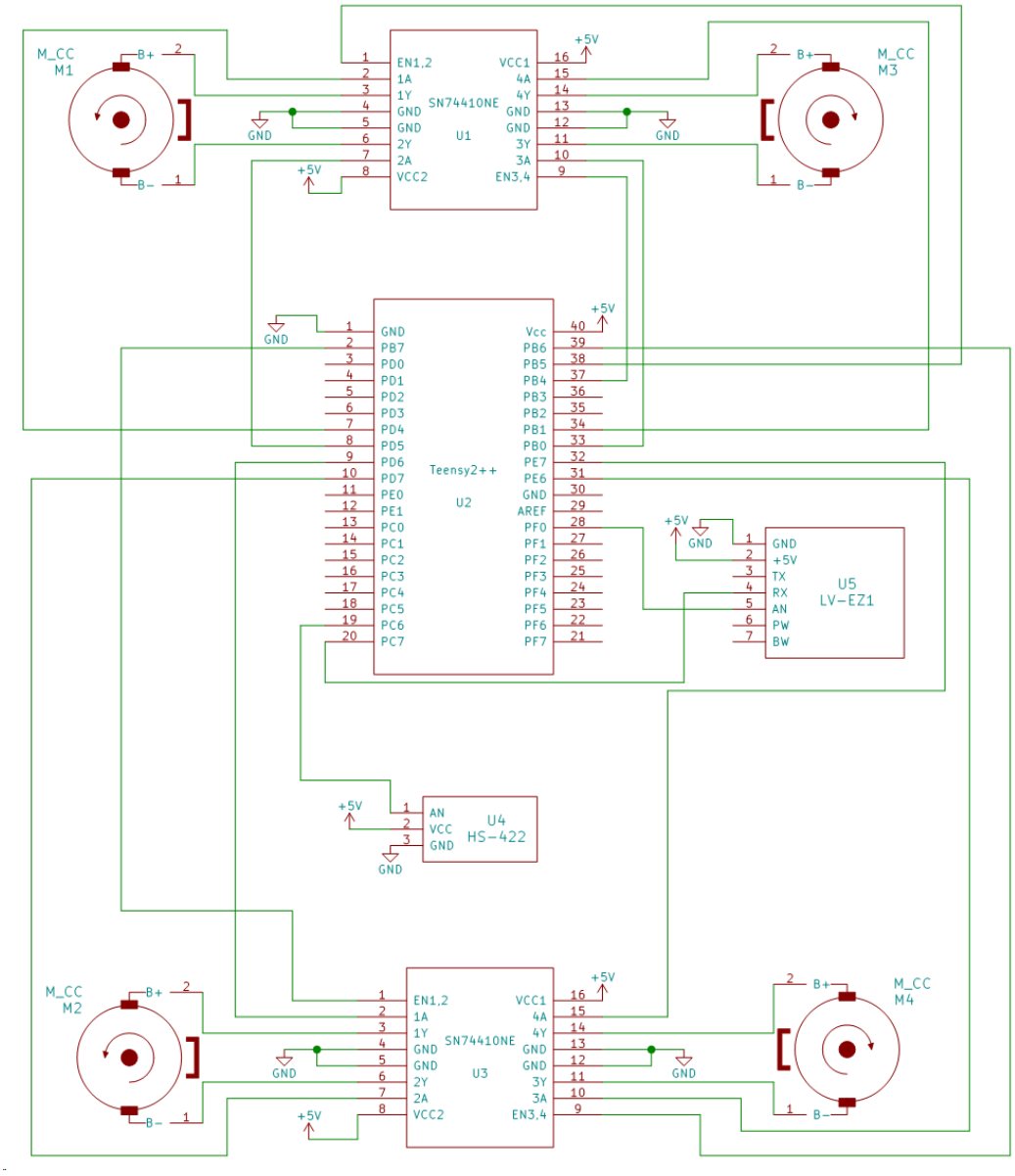


Figure 3.6: Teensy subsystem schematic omitting the communication part

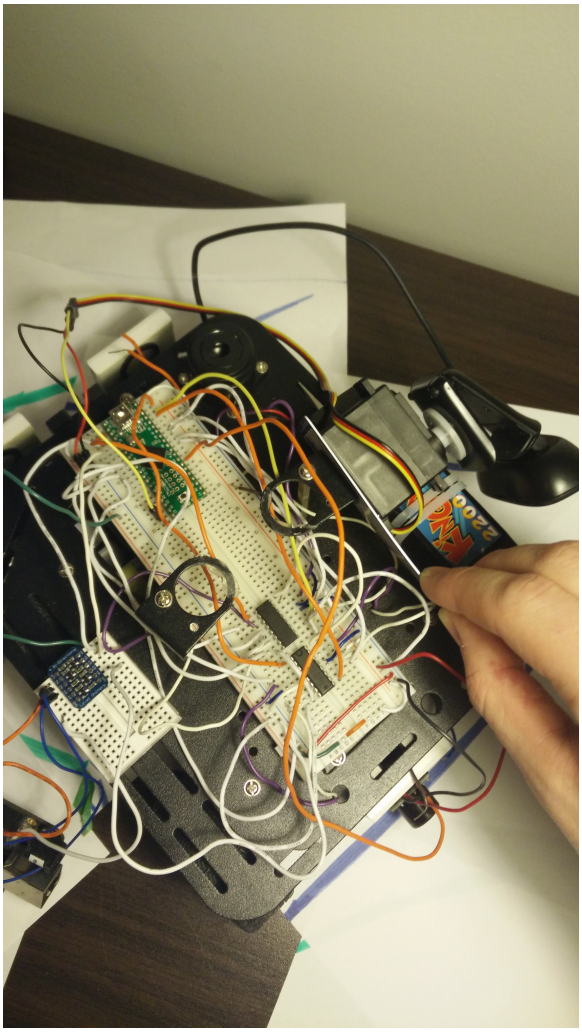


Figure 3.7: Teensy subsystem implementation

3.2.2.1 Teensy

Embedded system with the integrated Atmega AT90USB128 System On a Chip (SoC). Amongst the multitude of components contained in the SoC [1], here are the one used on this prototype (to avoid saturating this document) :

- Four usable timers (two 16-bit timers and 2 8-bit timers)
  - To control the H-Bridges enable pins (Motor Speed) with PWM
  - To control the Servo Motor's angle with PWM
- Analog to Digital (ADC) converter operating on 5V inputs.

  - To read analog values from the sonar

- 48 Programmable I/O lines
  - To control the H-Bridges **xA** pins with digital signals (see section 3.2.2.4)
- USB interface
  - To program the AT90USB128
  - To test the video capture logic with a normal computer (replacing I^2C communication by USB serial communication)
- I^2C interface (see section 3.3.1)
  - For intercommunication with the Beaglebone Black
- Can go up to 16Mhz
  - Allows a tighter program loop, thus a more responsive system in general
- Operates on 5V
  - To get ADC readings that are more precise than the BeagleBone Black running on 3.3V

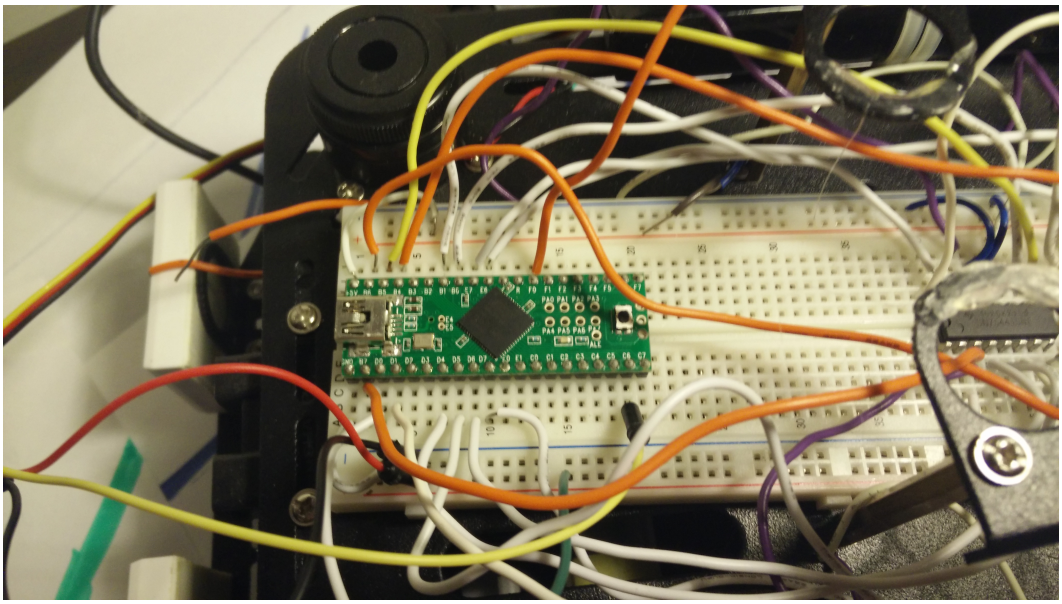


Figure 3.8: Teensy used in the project

3.2.2.2 HS-422

Standard Servo providing up to 180 degrees of rotation. This Servo can operate with voltage between 4.8V and 6V and using the standard 50hz pulse frequency [3]. The camera is actually mounted on rotating part.

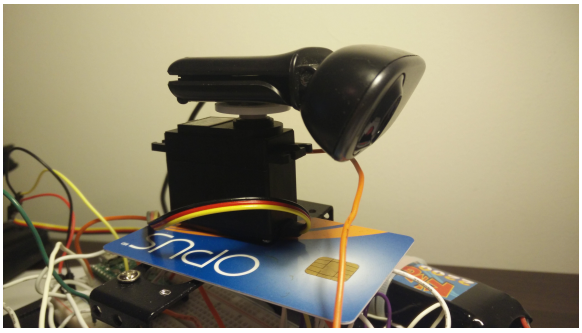


Figure 3.9: Servo used in the project with the webcam mounted

3.2.2.3 LV-EZ1

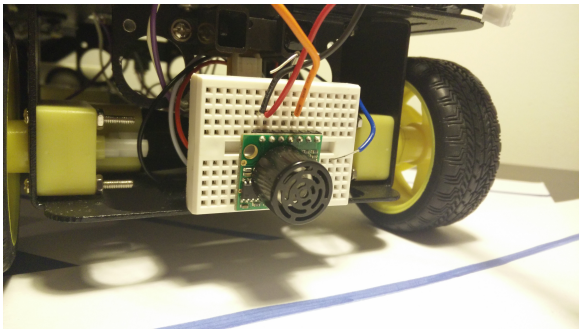


Figure 3.10: Sonar used in the project

Ultrasonic sonar working on 5V input, which is consistent with the rest of the subsystem. It provides reliable distance reading on objects of arbitrary shape (e.g. compared to an IR distance sensor), which is more apropriate in the context of a car simulation. This sonar allows to serially connect multiple sonars together so that they don't all read simultaneously (see figure 3.11), thus avoiding interference. It is not used in this prototype as we didn't have time to completely implement the object detection.

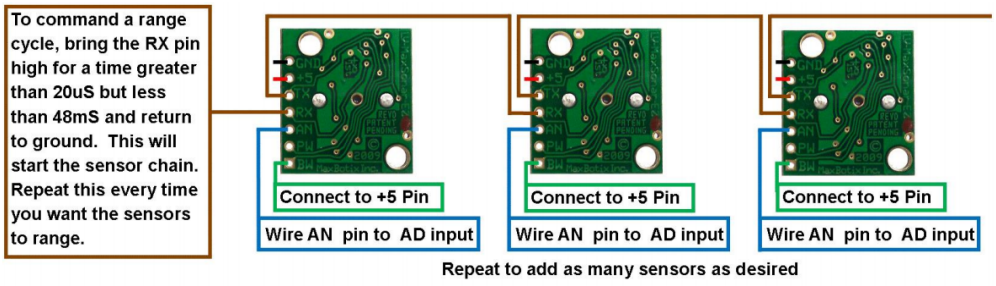


Figure 3.11: Serial interconnection of sonars[5]

3.2.2.4 SN754410

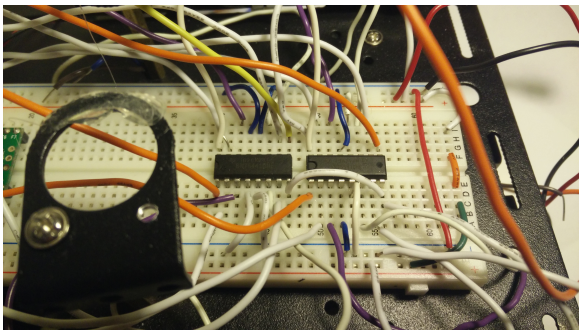


Figure 3.12: H-Bridges setup used in the project

Quadruple Half-H Driver accepting up to 36V of supply voltage, which is more then plenty for driving electrical motors. By it's design, it allows to control the overall intensity (i.e. PWM duty cycle) through the enable pins and control the directions by raising/sinking pairs of **xA** pins (which is what is done for this project).

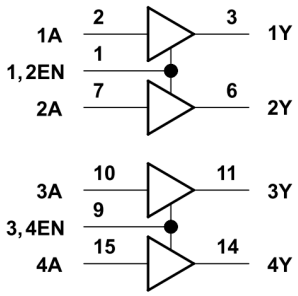


Figure 3.13: SN754400 simplified schematic [8]

3.2.2.5 Motor

Four constant current motors used to control 4 wheels independently.

3.3 System Intercommunication

3.3.1 Teensy/Beaglebone Black

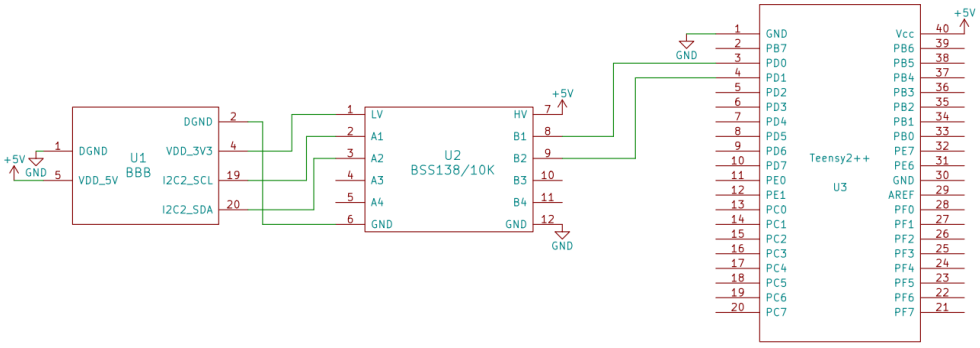


Figure 3.14: Interconnection hardware schematic

Both boards are interconnected through an I^2C bus. The communication protocol requires both the SDA and SCL line to be pulled up, but the Beaglebone Black and the Teensy operate on different voltages. To overcome this problem, some Field-Effect Transistor (FET) must be used on each line in order to pull up both sides to the proper voltage and sink them both at the same time (see figure 3.16). This will create a level shifter. In our case, the level shifter is a board breakout from *Adafruit*, both containing the FETs and the pull up resistors, so we simply needed to match the **Ax** pins with the **Bx** pins and supply the proper V_{DDs}. In order to get 3.3V, we simply took the 3.3V output from the Beaglebone.

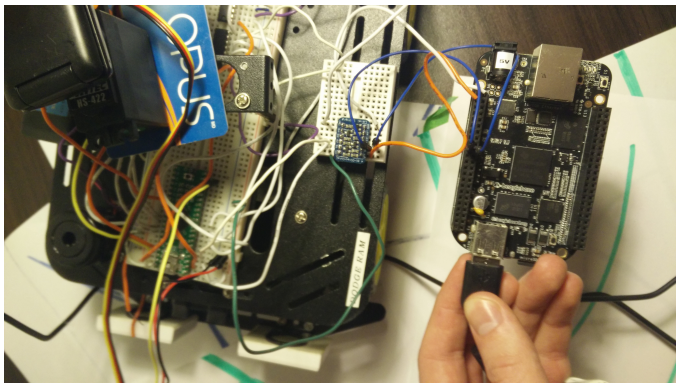


Figure 3.15: Intercommunication implementation in the project

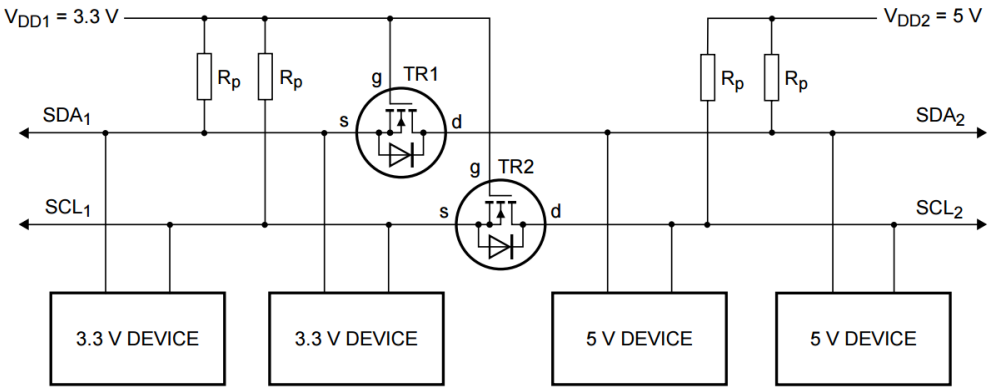


Figure 3.16: Bidirectional level shifter circuit connecting two different voltage sections in an I^2C bus system [4]

3.3.2 Webcam/Beaglebone

The Logitch webcam is connected to the Beaglebone using the onboard USB port. No USB hub was used because it is the only device using USB.

4 Software Design

4.1 System Software Design

If we picture the system as a whole, the software's function can be seen as this basic loop:

1. Capture an image

2. Extract features from the image
3. Decide what should be the next action from the feature extracted
4. Carry out this new action

The first three steps are computed on the Beaglebone Black running the latest stable version of Ubuntu. The software is using python with the OpenCV (Open Source Computer Vision) library python bindings in order to access the webcam and make computation on the images captured. In order for OpenCV to use the webcam, it is built with the famous *ffmpeg* video library in conjunction with the *V4L* (Video For Linux) package to to easily access video devices detected by the linux kernel. The *smbus* python library is also used to interface with the *SMBus* API of the kernel.

The last step is carried out by the Teensy. There is no operating system on the Tennsy. There is only a small program used to drive the motors (H-Bridges), the servo and read the sonar's values.

4.2 Software Subsystem Designs

4.2.1 Image Analysis (BeagleBone Black)

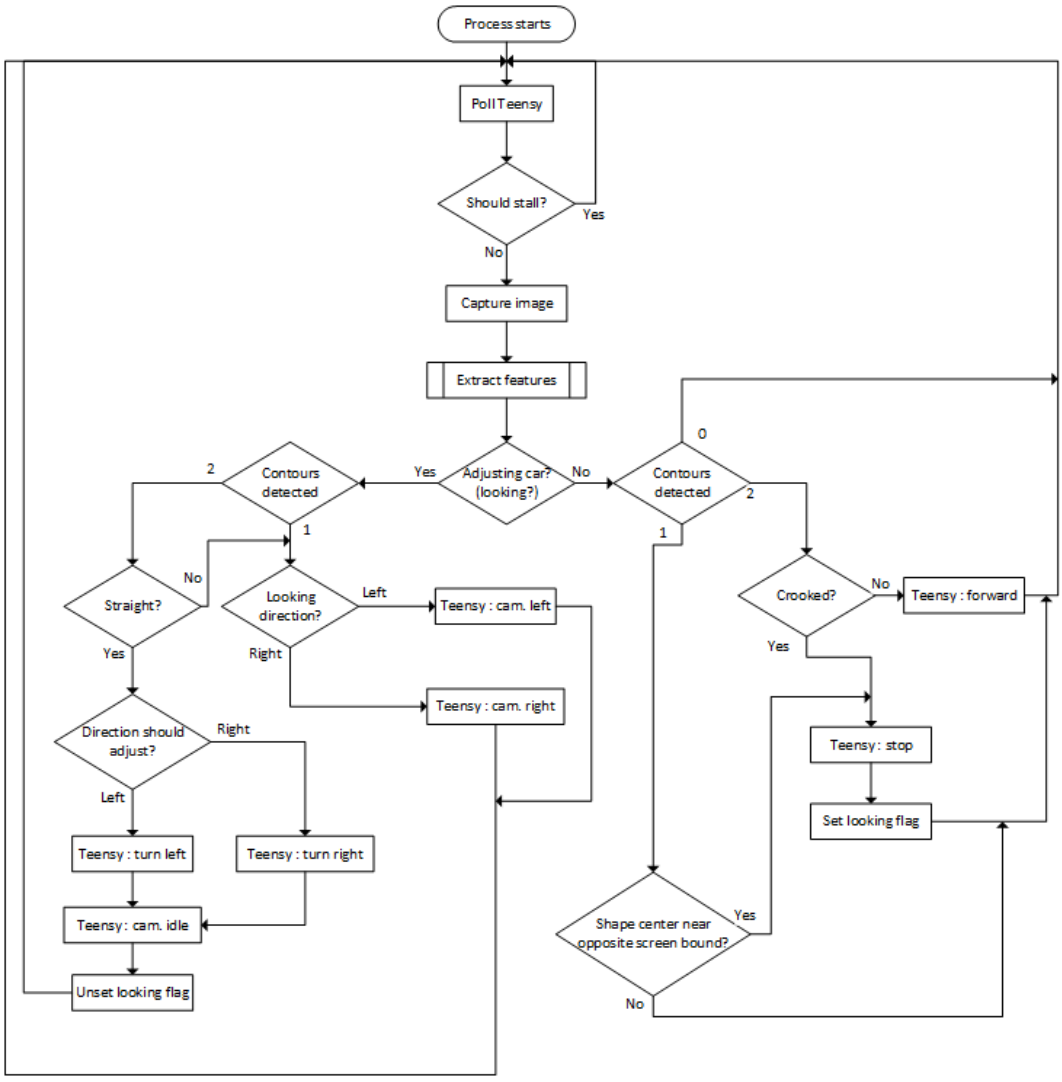


Figure 4.1: BeagleBone's software subsystem flowchart

This software module's job is to capture an image, extract some features out of this image, analyse these features and take a decision on what should the car do. It is installed on the Beaglebone Black on top of Ubuntu. As stated before, this subsystem's interface with the OS is the OpenCV library using the ffmpeg library (for video feed) and the smbus library (for I^2C communications) on top of python. Python will take care of everything that is "low level" (i.e. any interaction with the Linux kernel).

The main function of this subsystem is illustrated in figure 4.1. Any process of the type *Teensy:xxx* means it's a communication operation with the Teensy subsystem. The software is relatively simple. All it's power comes from the *Extract features* subprocess that is done right after capturing an image from the webcam. Before any process is done, the Beaglebone will poll the Teensy to see if the system should stall (e.g. the sonar detected something). This would have been optimally implemented as a interrupt, but because of time constraints, polling was used instead becaue of its simplicity.

The feature extraction is actually demanding and complex. The software will take the RAW pixel data and modify them it order to get computer fiendly representation of what is happening around the car. The idea is to detect the lane's lines in front of the car. The feature extraction is done this way (everything touching images is done with OpenCV) :

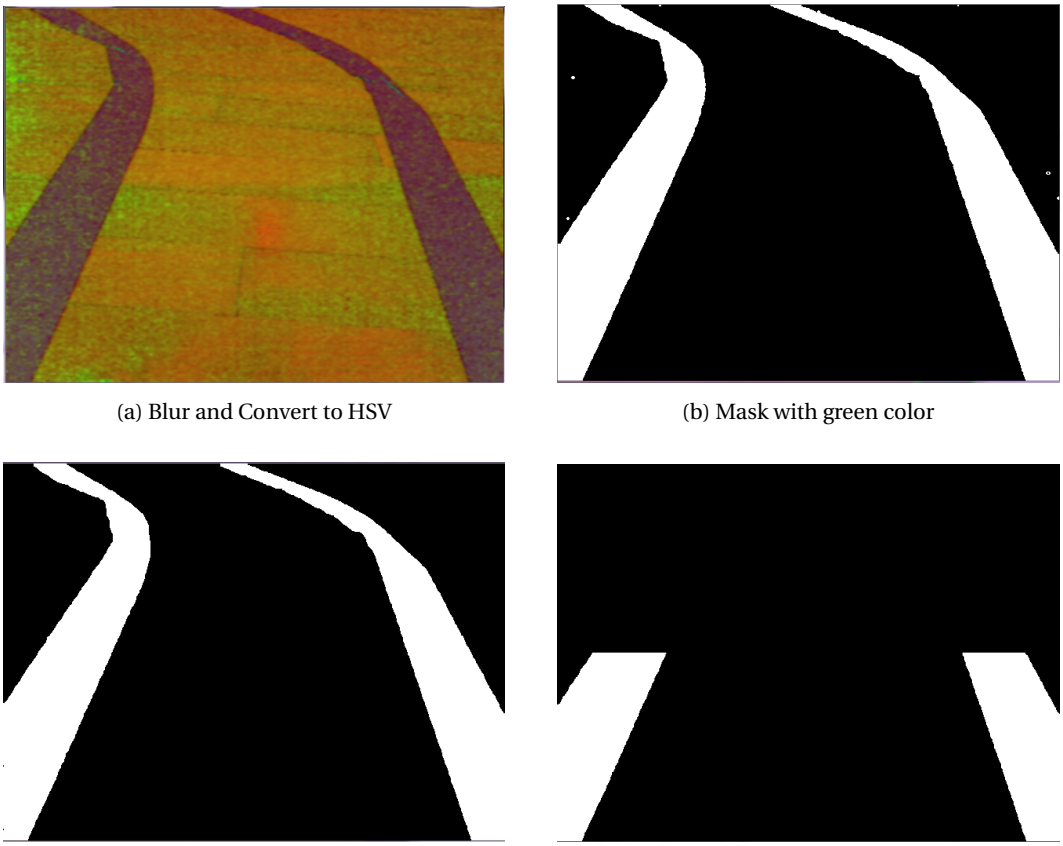
1. Blur the captured image to remove noise coming from the webcam
2. Convert the color values from an RGB representation to an HSV representation. An HSV representation is easier to work with.
3. Extract the pixels in the range of color we want to a binary reprentation. This means that with an HSV upper bound and lower bound, check every pixel to see if they are in range, If the pixel is in range, mark it as 1, else mark it as 0. This then gives a black and white mask of the pixels of the wanted color
4. Perform an opening operation on the mask. Thi means shrink every group of pixels and then dilate every group of pixel. This will have for effect to remove noise in the mask because the small group of pixels will disappear during the shrink, and the wanted group of pixel will regain their shape during the dilate operation.
5. Cut the image in two using a simple bitwise_and with a dummy half black, half white image.
6. Detect contours and approximate the shapes from the contours detected using a convex hull algorithm.

This should then give a distinct set of shapes. We can then detect the middle of the shapes and determine where the car needs to go. To detect if the car is crooked, we check the difference in y-values of both shapes detected (both shapes representing the lane's lines). If this difference exceeds a certain threshold, we know the car is crooked and we need to adjust. To detect an curve, we wait until we lose track of one lane line. This means we are entering curve. Then, when the center of the remaining shape exceeds a certain threshold, we know we have to turn.

To clarify the flowchart, here is how we handle turning whenever we detect that we need to turn (i.e. when the *looking* flag is set):

1. Stop the car completely
2. Start moving the camera in the proper direction until we are considered "straight" again.
3. Depending on the angle the camera moved, make the car turn in the proper direction while bringing back the camera to idle position

4. Restart the whole detecting algorithm. This way, if after turning the car is still crooked, it will adjust itself again.



(a) Blur and Convert to HSV

(b) Mask with green color

(c) Perform opening on the image

(d) Limit the field of view with a bitwise_and

Figure 4.2: The process of extracting features with OpenCV

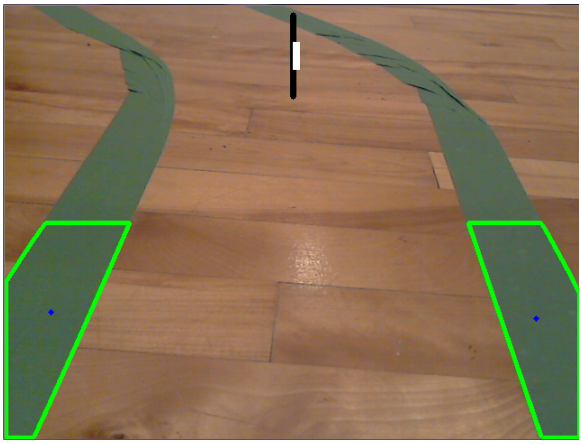


Figure 4.3: Final result of the feature extraction. Blue dots are the centers

4.2.2 Hardware manipulation (Teensy)

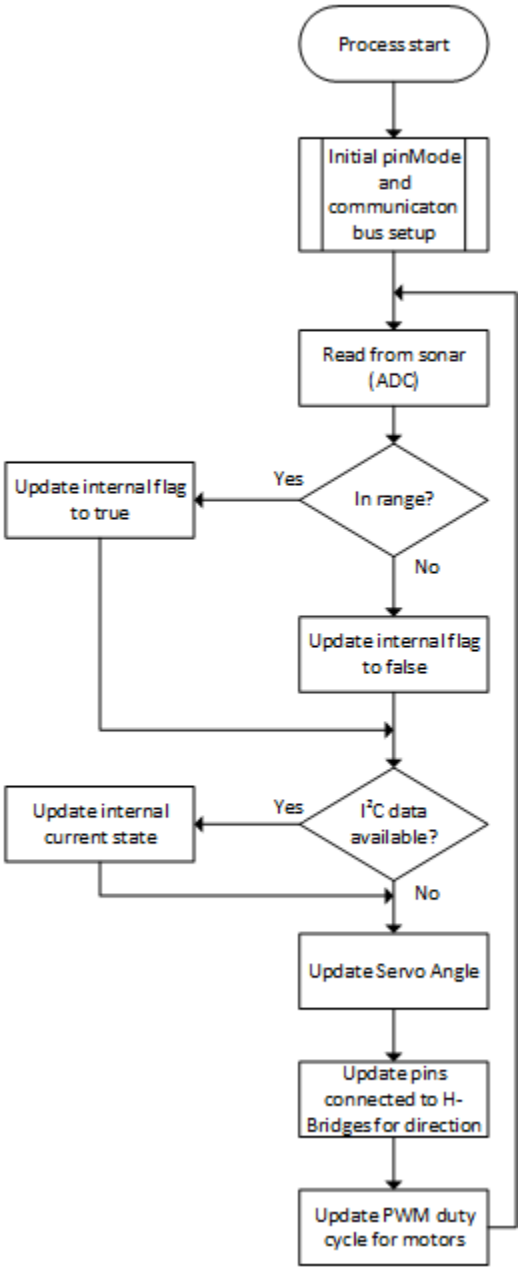


Figure 4.4: Teensy's software subsystem flowchart

This module's job is to drive the wheel motor, the servo motor and the sonar. As there are no operating system on the Teeny, the only interaction with other module is from the use of Arduino

libraries. But in the end, it ends up being a single software module (i.e. when compiled). The **Servo** library is used to control the servo motors and the **Wire** library to control I^2C , otherwise standard arduino functions are used.

The main function of this subsystem is illustrated in figure 4.4. Using the Wire library, an handler is attached on the I^2C data request from a master device (i.e. for when the Teensy needs to be a slave sender). This is used for the BeagleBone to poll the Teensy. When it is polled, it returns the internal *inRange* flag. The three update methods are affected by the internal current status of the car. This status is updated by the Beaglebone after the analysis of an image. The possible status are :

Listing 4.1: Internal status enum used in the Teensy software

```
enum car_status{
        car_idle = 0,
        car_forward = 1,
        car_left = 2,
        car_right = 3,
        car_vision_rotation_left = 4,
        car_vision_rotation_right = 5,
        car_vision_idle = 6
};
```

Based on the current status, the H-Bridges will be adjusted for the direction or the PWM duty cycle will be adjusted for appropriate speed. The angle of the servo will also adjust depending on the status. For the project, since four motors were used, the turning algorithm basically make half of the wheels go in one direction and the other half in the opposite direction.

4.3 System Software Communication

Only I^2C is used to interconnect both subsystem. To control the I^2C communications, SMBus is used in Linux and the Wire library is used in Arduino for the Teensy. Due to time contraints, the communication protocol used is simple. During the polling of the Teensy by the Beaglebone, a single byte is sent back, which is either 1 or 0 indicating if something is in the range of the sonar. During the polling, the Beaglebone Black goes into master receiver mode and the Teensy goes into slave sender mode. After that, the Teensy goes back into slave receiver mode and the Beaglebone returns to being in master writer mode. After analysing an image, the latter sends a single byte to the Teensy indicating the next state in which the Teensy subsystem should be (see listing 4.1).

Otherwise, there is USB communication between the BeagleBone Black and the Logitech webcam, but it is handled by the kernel. The Beaglebone's software subsystem simply request an image from the webcam and gets the result back without any low level code involved.

5 Development Software

5.1 BeagleBone Black

The principale software installed on the BeagleBone Black is the latest Ubuntu 14.04 image for ARM platforms. Since the DTC was broken, a corrected version [7] was compiled and installed in order for evertything to work correctly.

On top of Linux, the packages that were installed are :

1. Python and it's dependencies
2. Python-smsbus (for I^2C support)
3. OpenCV and it's dependencies [9]
4. FFMPEG
5. V4L (Video for Linux)

For the development of the python software, **PyCharm**, a professional python IDE by *Jet-Brains*, was used. This allowed us to debug the python scripts before trying them on the Beagle-Bone Black . For installation instruction, refer to section 7.1.

5.2 Teensy

For the Teensy, simply the adapted **Arduino IDE** (i.e. *Teensyduino*) was used in conjunction with the Teensy Loader. The libraries used are :

1. Basic digital and analog functionalities
2. The **Wire.h** library (included with the Arduino IDE) for I^2C
3. The **Servo.h** library (included with the Arduino IDE) for the servo motor

6 System Performance

This part of the discussion will only reflect what we were able to actually achieve. Anything related to expected functionalities is not present to avoid confusion. We also deem our actual work, even though it's not complete, to be complex enough to give a discussion about it.

6.1 Component Testing

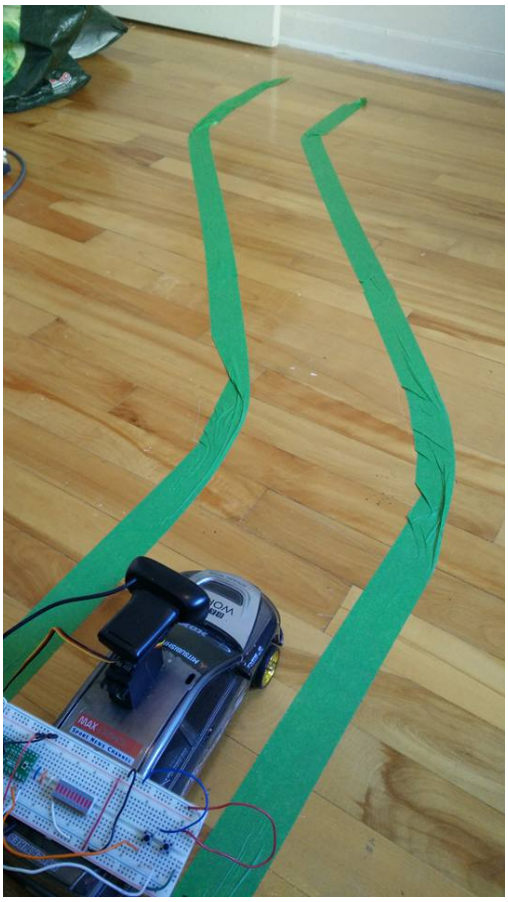
6.1.1 Webcam / Image Analysis

This was the trickiest part because we had to find a way to get reliable results. The key here is *reliable*. Multiple different techniques were used to detect the lines forming the lane in front of the car. Some of these techniques were :

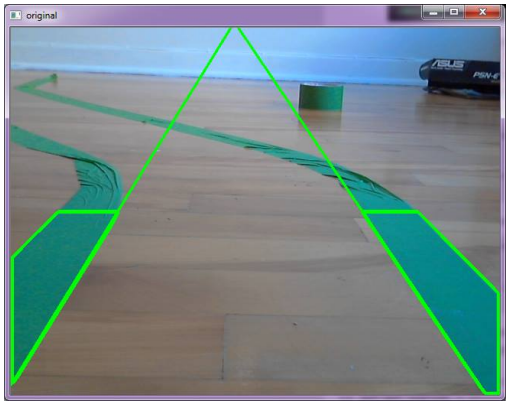
1. Approximating lines through the detected shapes and detect their intersection. This yielded nice result only if the car was going in a really straight line. As soon as the car became a bit crooked, the lines were going crazy.
2. Trying to detect the general orientation of the detected shapes using the shape moments data. This simply never worked.
3. Trying a multitude of viewing angle for the webcam.

Unfortunately, none of these were near acceptably reliable, but the experience it gave us lead us to our current algorithm, because we became more familiar with image manipulations.

Also, we placed the car in a multitude of positions in a curved section of a lane in order to understand the visual cues that were detectable to know whether we had to start turning or no. This also proved to be tricky, because depending on the angle the webcam had towards the ground, the cues would change and would become very misleading. Again, with trial and error, we finally found how to position the webcam so that we can get a good field of view to extract features from.



(a) Curve cues detection



(b) Line approximation method tried

Figure 6.1: Some tests examples

6.1.2 Webcam / Motors coordination

First, when we want to turn, we stop, look around, turn and continue. Coordinating the camera's rotation and the car's rotation was too complexe for the time we had. We ended up bringing back the camera to idle before turning, tell the Teensy to blindly start turning, wait a certain amount of time and tell the Teensy to stop turning. In order to get the car to turn on itself to the desired angle, we simply did some trial and error trying to find a corelation between the angle of the camera and the time the system needed to wait until it had to tell the Teensy to stop.

6.1.3 I^2C

The main challenge was to understand how Linux's *smbus* and *Arduino*'s Wire library could interact together. So we made a simple python software using the smbus python's binding and a

simple arduino sketch to run on the Teensy. We tested both the master sender/slave receiver pair and the master receiver/slave sender pair. When we had this basic communinication going, we simply had to do a protocol to use between both systems.

6.1.4 Sonars

For the sonar, since we had plans to use more than one, we tested a technique to wire them so that they read serially automatically (see figure 6.2). To test the wiring and the readings, we used LEDs and serial output to get a visual feedback of what was happening. We were able to make it work without interference, but due to time constraints, we only used one sonar. So we simplified the code we developed doing our tests.

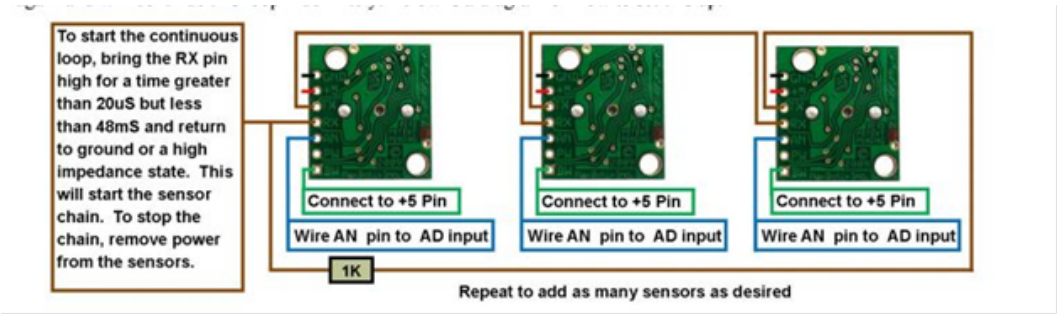


Figure 6.2: Serial technique we tried to avoid interference [5]

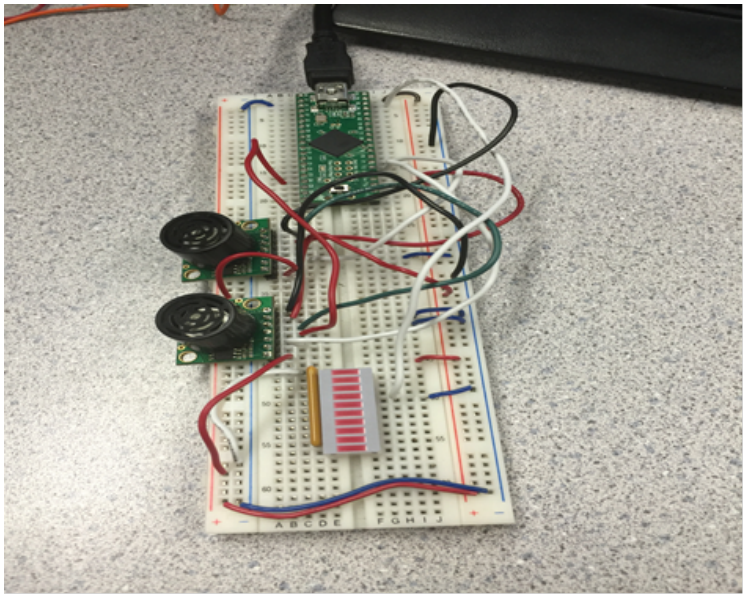


Figure 6.3: The actual test setup

6.2 System Test

The initial "whole" system tests where done using a normal PC. The only difference was that we used serial communication through USB instead of I^2C and that it was not the BeagleBone Black doing the computation. The Teensy subsystem and every software components were identical.

So, to test the whole system, we simply put it at the beginning of the lane and let it do "it's thing". We first made the car start completely straight to see how it reacted when it got to a curve. As we observed how the car handled the curve, we adjusted the software until we got consistent results. We then started the car crooked to see how much it would straighten up by itself. Then again, we adjusted the software until we had consistent results.

After that, we moved the pyhton software on the BeagleBone Black and migrated to I^2C . We then ran similar tests. We also added the sonar functionalities Even though we had the exact same software, we realized that the BeagleBone Black was slower than we expected. Even though we had very consistent results using the computer in the exact same environment, we started getting sluggish reaction times and weird behaviour. We could not debug at all because we had no way to see what the BeagleBone Black was actually seeing. From the tests, we came up with the hypothesis that the video was internally buffered by Linux, because the system would react with a certain lag. It is as if the camera was looking at 75°, be the system was still analyzing the image took at 78-79°. Also, the polling and the sonar reading turned out to work. The car would stall when something was in front. Unfortunately, we realized that maybe 1 to 2% of the sonar reading were invalid. This proved to be just enough to make the car start moving again sometimes, which was a problem. We tweaked both software and retried the same tests with objects in the way.

The systems tests also showed that the Arduino code seems to hide some shady setup for the timers. At some point, when the car was going forward, the webcam started wiggling left and right, showing that the PWM signal that was sent to the Servo became extremely noisy. We verified the wiring and nothing was shorting. It could also be an hardware problem since all the timers are used on the Teensy. Maybe there is some current leakage. We unfortunately never found the cause. Bare metal code would have been the best for the Teensy, but again, a time constraint made us use a faster and easier technology.

We unfortunately couldn't get results as consistent as we wanted because of the time, but at least it technically worked. With some optimization, we are confident we could get way more consistent results.

7 System Delivery

7.1 System Initialisation

This section will only explain how to set up the software on the BeagleBone Black and Teensy. For everything regarding hardware set up, refer to section 3. Here are the required steps to get the whole system up and running:

1. Install the latest version of *Ubuntu* and do the inital set up of the operating system (see

appendix A)

2. Compile and install the *ffmpeg* video library and the kernel video interface *V4L* (Video for Linux) required for OpenCV (see appendix B).
3. Install *Pyhton* and all its dependencies in order to be able to compile and install *OpenCV* (see appendix C).
4. Copy the python script for the car locally on the BeagleBone Black .
5. Open the Teensy's subsystem source code into the *Arduino IDE*, compile it and transfer the binaries on the Teensy.
6. Adjust the HSV color range in the python script.
7. Start the python script on the BeagleBone Black .

Due to the currently limited control over the whole system, a lane of green tape should already have been made and the car should have been put at the beginning of it facing more or less towards the direction it should go. If it's crooked a little, it will adjust itself when the script starts. Also, the car can currently only take left curve even though it will adjust itself whether it is crooked to the left or to the right. For the source code, you can consult appendix D and E.

It is also important to know that there is currently no ending condition, so the car will try to follow a lane up until the program crashes or is explicitly closed by the user.

7.2 System Operation

The first thing that should be done is to make a lane with two lines of green tape (see figure 6.1a). Green is specified because it is a color that will give the lowest amount of noise during the features extraction. It is important to note that the car can currently only take a left curve. Figure 6.1a shows a right curve near the end, but it is not used, though it was intended to be used. The car should then be placed at the beginning of the lane.

Before starting the script, some software adjustment must be done to get optimal results. Unfortunately, these changes must be hardcoded for the moment. It was intended to have an actual LCD screen interface with some buttons, but we didn't have time to implement it.

Teensy - Line 24 You can adjust the the maximum range before something is detected as *In Range* by the sonar.

Teensy - Line 65 and 83 You can set the servo's idle angle. If the webcam and the servo were mounted crooked, this value can be changed to compensate.

Python - Line 10-15 Here **you have to set** a proper HSV color range for the feature detection to work correctly (see appendix F).

Python - Line 18 You can adjust the capture device number if ever there is a problem. Sometime, if the program crashes or doesn't close properly, the device number 0 locks and we need to use 1 instead. This problem can be detected when trying to launch the script and the capture device is never able to open and the script stops.

When everything is ready, the script can be started and the car should start moving by itself. When the path has been followed properly, we can stop the script explicitly.

As stated earlier, this section won't contain expected features setting. Even though the system is very limited, it is technically working so we rather present the working parts instead.

8 Conclusion

In conclusion, we have more profound understanding and hands-on experience towards assembling different parts of hardware and software together to get a working system. Building this car led us to see that embedded systems are very sensitive to any small hardware (e.g. bad wiring) and software (e.g. write to wrong register) mistakes. It was a nice experience to concretely use everything learned from the beginning in a complex project with an actual purpose. From the communication tests we did, we also learned that even though intercommunication standards exists, they are not managed the same way on every SoC (e.g. I^2C through Linux vs Bare Metal), so we need to fully understand how each SoCs used in the project work in order to get a system working fluently. As computer scientist major, we often code without even acknowledging the hardware on which we code. However, this class made us realize the close relation software has with hardware event though we use high level languages like *Java*.

Appendices

Appendix A Installing Ubuntu on the BeagleBone Black

The only hardware required is a microSD card with at least 4GB and a SD card reader with a microSD to SD adapter(or a microSD card reader). The easiest way to load Ubuntu on a microSD card is using the latest pre-built image at `https://rcn-ee.com/rootfs/2015-11-13/elinux/ubuntu-14.04.3-console-armhf-2015-11-13.tar.xz`. This image contains a script that will setup the microSD card correctly with a bootable partiton containing the *U-Boot* bootloader and another partition containing Ubuntu [6].

1. Download the image

```
wget https://rcn-ee.com/rootfs/2015-11-13/elinux/ubuntu-14.04.3-
    console-armhf-2015-11-13.tar.xz
```

2. Unpack the image using the **tar** command and navigate to the unpacked folder

```
tar xf ubuntu-14.04.3-console-armhf-2015-11-13.tar.xz
cd ubuntu-14.04.3-console-armhf-2015-11-13
```

3. Find where the SD card has been mounted by Linux

```
sudo ./setup_sdcard.sh --probe-mmc
```

4. Launch the loading script with the location of the SD card (/dev/sdX)

```
sudo ./setup_sdcard.sh --mmc /dev/sdX --dtb beaglebone
```

After this, the microSD card can be inserted in the BeagleBone Black and we can boot Ubuntu.

Before we can use the system properly, we have to update the packages and install a proper Device Tree Compiler, since the original one is broken. After connecting the BeagleBone Black with a network cable and booting up we can start. Using *PuTTY* on Windows or any terminal in Linux, we can ssh on the BeagleBone Black and login with the user `ubuntu` with the password `ubuntu`. When we are connected, the first thing we need to do is update the package list and the installed packages using the following commands:

```
sudo apt-get update
sudo apt-get upgrade
```

When this procedure is done, we are ready to install a proper dtc for the BBB [7].

1. Search and install the latest kernel image

```
apt-cache search linux-image
apt-get install linux-image-4.1.1-bone10 -> Depending on the latest version
```

2. Get the latest dtc source code and the latest device tree overlays

```
cd /usr/src
git clone https://github.com/RobertCNelson/bb.org-overlays
cd bb.org-overlays
```

3. Compile and install the new dtc using the proper script

```
./dtc-overlay.sh
```

4. Compile and install the new device tree overlays

```
./install.sh
```

At that point, the base installation is completed. Software can then be installed and the BeagleBone Black can be used as needed.

Appendix B Installation Procedure for FFMPEG

Since **ffmpeg** is required for video in OpenCV, we have to install it. In the 14.XX versions of Ubuntu, Canonical removed the official package from their repositories, so we need to build it. We first need to get the source code from git, compile it and install the x264 video codec using the following commands :

```
git clone git://git.videolan.org/x264.git
cd x264
./configure --enable-shared --prefix=/usr
make
make install
```

When this is installed, we can do the same thing with the ffmpeg sources :

```
git clone git://git.videolan.org/ffmpeg.git
cd ffmpeg
./configure --enable-shared --enable-libx264 --enable-gpl
git remote set-url origin git://source.ffmpeg.org/ffmpeg
make
make install
```

We also need a small utility (Video for Linux, *V4L*) to interface with the kernel :

```
sudo apt-get install v4l-utils
```

Finally, to avoid some dynamic library linking problems, we need to add "`/usr/local/lib`" to the file "`/etc/ld.so.conf`", then run the `ldconfig` command.
Ffmpeg is ready to be used at that point.

Appendix C Installing Python and OpenCV

Before installing OpenCV, we must be sure we have python installed with all the apropriate libraries. We also need to install OpenCv's dependencies. We can install everything in one command. This command also include the SMBus library for python so that we can use I^2C properly.

```
sudo apt-get install python-setuptools python-pip python-smbus build-
    essential cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-
    dev libswscale-dev python-dev python-numpy libtbb2 libtbb-dev libjpeg-
    dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev
```

After the installation of all the packages, OpenCV is ready to be installed using the following steps:

1. Clone the OpenCV git repository

   ```
   git clone https://github.com/Itseez/opencv.git
   ```

2. Go into the directory and create a new folder named "release"

   ```
   cd opencv
   mkdir release
   cd release
   ```

3. Configure the compilation. This will take over an hour on the BeagleBone Black .

   ```
   cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D
       BUILD_EXAMPLES=ON ..
   ```

4. Compile and install OpenCV

   ```
   make
   sudo make install
   ```

At that point, OpenCV should be installed and ready to be used.

Appendix D Teensy subsystem Source Code

```
#include "Servo.h"
#include "Wire.h"

enum car_status{
    car_idle = 0,
    car_forward = 1,
    car_left  = 2,
    car_right = 3,
    car_vision_rotation_left = 4,
    car_vision_rotation_right = 5,
    car_vision_idle = 6
};

Servo serv;
int  servoAngle;
unsigned char inRange = 0;
car_status currStatus;

//Sonar variables
const int anPin = 38;
const int led_pin = PIN_C7;
long anVolt, cm;
long read_var;
const int RANGE_MAX = 18;

void dataHandler(int lol){
    //Needed for the Wire lib., to work properly
}

void requestHandler(){
    if (inRange){
        Wire.write(0);
    }
    else{
        Wire.write(1);
    }
}

void setup() {
    Serial.begin(9600);
    Wire.begin(100);
    Wire.onRequest(requestHandler);
    Wire.onReceive(dataHandler);

    pinMode(PIN_F0, INPUT);
    //To control the motors direction
    pinMode( PIN_D4, OUTPUT);
    pinMode( PIN_D5, OUTPUT);
    pinMode( PIN_D6, OUTPUT);
    pinMode( PIN_D7, OUTPUT);
    pinMode( PIN_B1, OUTPUT);
    pinMode( PIN_B0, OUTPUT);
    pinMode( PIN_E7, OUTPUT);
    pinMode( PIN_E6, OUTPUT);

    //To Control the bridges enable pin
    pinMode(PIN_B4, OUTPUT);
    pinMode(PIN_B5, OUTPUT);
```

```
59    pinMode(PIN_B6, OUTPUT);
60    pinMode(PIN_B7, OUTPUT);
61
62    //To control the servo
63    pinMode(PIN_C6, OUTPUT);
64    serv.attach(PIN_C6);
65    servoAngle = 93;
66    serv.write(servoAngle); //idle position
67
68    currStatus = car_idle;
69
70 }
71
72 void updateServoAngle(){
73    switch (currStatus){
74       case car_vision_rotation_left:
75          servoAngle--;
76          currStatus = car_idle;
77          break;
78       case car_vision_rotation_right:
79          servoAngle++;
80          currStatus = car_idle;
81       break;
82       case car_vision_idle:
83          servoAngle = 93;
84          currStatus = car_idle;
85       break;
86    }
87 serv.write(servoAngle);
88 }
89
90 void updateDirection(){
91    switch (currStatus){
92       case car_forward:
93          digitalWrite( 4, LOW);
94          digitalWrite( 5, HIGH);
95          digitalWrite( 6, LOW);
96          digitalWrite( 7, HIGH);
97          digitalWrite( 21, HIGH);
98          digitalWrite( 20, LOW);
99          digitalWrite( 19, HIGH);
100         digitalWrite( 18, LOW);
101         break;
102      case car_left:
103         digitalWrite( 4, LOW);
104         digitalWrite( 5, HIGH);
105         digitalWrite( 6, LOW);
106         digitalWrite( 7, HIGH);
107         digitalWrite( 21, LOW);
108         digitalWrite( 20, HIGH);
109         digitalWrite( 19, LOW);
110         digitalWrite( 18, HIGH);
111         break;
112      case car_right:
113         digitalWrite( 4, HIGH);
114         digitalWrite( 5, LOW);
115         digitalWrite( 6, HIGH);
116         digitalWrite( 7, LOW);
117         digitalWrite( 21, HIGH);
118         digitalWrite( 20, LOW);
```

```
            digitalWrite( 19, HIGH);
            digitalWrite( 18, LOW);
        break;
    }
}

void updateMotorSpeed(){
    switch (currStatus){
        case car_forward:
            analogWrite(PIN_B4, 255);
            analogWrite(PIN_B5, 255);
            analogWrite(PIN_B6, 255);
            analogWrite(PIN_B7, 255);
            break;
        case car_right:
        case car_left:
            analogWrite(PIN_B4, 255);
            analogWrite(PIN_B5, 255);
            analogWrite(PIN_B6, 255);
            analogWrite(PIN_B7, 255);
            break;
        case car_idle:
        case car_vision_rotation_left:
        case car_vision_rotation_right:
        case car_vision_idle:
            analogWrite(PIN_B4, 0);
            analogWrite(PIN_B5, 0);
            analogWrite(PIN_B6, 0);
            analogWrite(PIN_B7, 0);
        break;
    }
}

void loop() {
    //Detecting given range
    anVolt = analogRead(anPin);

    if (RANGE_MAX > anVolt){
        inRange = 1;
    }else{
        inRange = 0;
    }

    if (Wire.available()){
        unsigned char data = Wire.read();
        currStatus = static_cast<car_status>(data);

    }
    updateServoAngle();
    updateDirection();
    updateMotorSpeed();
}

```

Appendix E BeagleBone Black subsystem Source Code

This code is far from perfect. This version of the code is the version with which we got the most consistent result. In order to do so, we tweaked it a bit so that it would take a left curve more consistently. We then neglected the ability to take a right curve.

```
import numpy as np
import cv2
import math
from time import sleep
import smbus as s

def nothing(x):
    pass

Hh = 73
Hl = 40
Sh = 255
Sl  = 64
Vh = 160
Vl  = 22

teensySerialComm = s.SMBus(2)
cap = cv2.VideoCapture(0)
teensySerialComm.write_byte(0x64,6)

if  (cap.isOpened()):
    print 'Opened'
else:
    print 'not opened'

angle = 0;
openingKernel = np.ones((7,7), np.uint8)
looking = False
lr  = False
ll  = False
flat  = 0;

while(1):
    val  = teensySerialComm.read_byte(0x64)
    print ll
    print lr
     if  val  == 0:
   if  not looking:
        teensySerialComm.write_byte(0x64,0)
   print 'stopping'
         continue

    _,  frame = cap.read()
    gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    blurred = cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)
    blurred = cv2.GaussianBlur(blurred, (5,5),0)
     flat  += 1
     if  flat  < 50:
        continue

    lower_b = np.array([Hl,Sl,Vl])
    upper_b = np.array([Hh,Sh,Vh])

```

```
    mask = cv2.inRange(blurred,lower_b,upper_b)
    opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, openingKernel)

    h,w,_ =  frame.shape
    cutoff1 = np.zeros((h/2,w), np.uint8)
    cutoff2 = np.ones((h/2,w), np.uint8)
    cutoff = np.vstack((cutoff1,cutoff2))

    opening = cv2.bitwise_and(opening, opening, mask=cutoff)

    i, cnts, lol = cv2.findContours(opening,cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

    if looking:
        if len(cnts) != 2:
            if lr:
                teensySerialComm.write_byte(0x64,5)
            elif ll:
                teensySerialComm.write_byte(0x64,4)
            angle+= 1
        else:
            angle +=1
            if lr:
                teensySerialComm.write_byte(0x64,5)
            elif ll:
                teensySerialComm.write_byte(0x64,4)
            hull = cv2.convexHull(cnts[0])
            M = cv2.moments(hull)
            c1x = int(M['m10']/M['m00'])
            c1y = int(M['m01']/M['m00'])
            hull = cv2.convexHull(cnts[1])
            M = cv2.moments(hull)
            c2x = int(M['m10']/M['m00'])
            c2y = int(M['m01']/M['m00'])

            if c1x > c2x:
                c1x, c2x = c2x, c1x
                c1y, c2y = c2y, c1y

            diff = (c2y - c1y)
            print diff
            if abs(diff) < 35:
                print angle
                teensySerialComm.write_byte(0x64,6);
                if lr:
                    teensySerialComm.write_byte(0x64,3)
                elif ll:
                    teensySerialComm.write_byte(0x64,2)
                sleep((angle/2.7) /10)
                teensySerialComm.write_byte(0x64,0)
                angle = 0
                looking = False
                ll=False
                lr=False
    else:
        print len(cnts)
        if len(cnts) == 2 :
            hull = cv2.convexHull(cnts[0])
            M = cv2.moments(hull)
            c1x = int(M['m10']/M['m00'])
            c1y = int(M['m01']/M['m00'])
```

```
            hull = cv2.convexHull(cnts[1])
            M = cv2.moments(hull)
            c2x = int(M['m10']/M['m00'])
            c2y = int(M['m01']/M['m00'])

            if c1x > c2x:
                c1x, c2x = c2x, c1x
                c1y, c2y = c2y, c1y

            diff = (c2y - c1y)
            if abs(diff) < 30:
                teensySerialComm.write_byte_data(0x64,0,1);
                print 'forward'
            else:
                if diff > 0:
        print c1x
        print c2x
                    looking = True
                    ll = True
                elif diff < 0 and c1x > 55:
        print c1x
        print c2x

                    looking = True
                    lr = True
    else:
        continue
                teensySerialComm.write_byte(0x64,0);
                print 'stop'

        elif len(cnts) == 1:
            hull = cv2.convexHull(cnts[0])
            M = cv2.moments(hull)
            c1x = int(M['m10']/M['m00'])
            c1y = int(M['m01']/M['m00'])
            if c1x > 270:
                teensySerialComm.write_byte(0x64,1)
            else:
                teensySerialComm.write_byte(0x64,0)
                looking = True
                ll = True
        else:
            print 'nothin'
```

Appendix F Early Python Code Prototype with Trackbars For Setting the HSV Color Range

This is an incomplete piece of code. It is useful though if we want to find out what HSV color range we need to set up on the BeagleBone Black . We just need to plug in the webcam in the computer, adjust line 13 depending on the number of capture devices plugged in the computer and run the script. We can then use the trackbar to find an HSV range giving an apropriate mask and opening.

```
1  import numpy as np
2  import cv2
3  import serial
4  import math
5
6
7
8  def nothing(x):
9      pass
10
11
12 cap = cv2.VideoCapture(0)
13
14
15
16
17 Hh = 90
18 Hl = 48
19 Sh = 255
20 Sl  = 51
21 Vh = 255
22 Vl  = 0
23 cv2.namedWindow('test', cv2.WINDOW_AUTOSIZE)
24 cv2.createTrackbar('Hh', 'test', Hh, 179, nothing)
25 cv2.createTrackbar('Hl', 'test', Hl, 179, nothing)
26 cv2.createTrackbar('Sh', 'test', Sh, 255, nothing)
27 cv2.createTrackbar('Sl', 'test', Sl, 255, nothing)
28 cv2.createTrackbar('Vh', 'test', Vh, 255, nothing)
29 cv2.createTrackbar('Vl', 'test', Vl, 255, nothing)
30 cv2.createTrackbar('t1', 'test', 8, 20, nothing)
31
32
33 kernel = np.ones((7,7), np.uint8)
34
35 while(1):
36     _, frame = cap.read()
37
38     gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
39     blurred = cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)
40     blurred = cv2.GaussianBlur(blurred, (5,5),0)
41     cv2.imshow("HSV",blurred)
42
43     Hh = cv2.getTrackbarPos('Hh', 'test')
44     Hl = cv2.getTrackbarPos('Hl', 'test')
45     Sh = cv2.getTrackbarPos('Sh', 'test')
46     Sl  = cv2.getTrackbarPos('Sl', 'test')
47     Vh = cv2.getTrackbarPos('Vh', 'test')
48     Vl = cv2.getTrackbarPos('Vl', 'test')
49     t1  = cv2.getTrackbarPos('t1', 'test')
```

```
50
51  lower_b = np.array([Hl,Sl,Vl])
52  upper_b = np.array([Hh,Sh,Vh])
53
54  mask = cv2.inRange(blurred,lower_b,upper_b)
55  opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
56  cv2.imshow("mask",mask)
57
58
59  h,w,_ =  frame.shape
60  cutoff1 = np.zeros((h/2,w), np.uint8)
61  cutoff2 = np.ones((h/2,w), np.uint8)
62  cutoff = np.vstack((cutoff1,cutoff2))
63
64  cv2.imshow("opening before",opening)
65  opening = cv2.bitwise_and(opening, opening, mask=cutoff)
66  cv2.imshow("opening after",opening)
67
68  i, cnts, lol = cv2.findContours(opening,cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
69
70
71  print len(cnts)
72  if len(cnts) == 2:
73      #print 'There are 2 points on the screen : ',
74      hull = cv2.convexHull(cnts[0])
75      M = cv2.moments(hull)
76      c1x = int(M['m10']/M['m00'])
77      c1y = int(M['m01']/M['m00'])
78      hull = cv2.convexHull(cnts[1])
79      M = cv2.moments(hull)
80      c2x = int(M['m10']/M['m00'])
81      c2y = int(M['m01']/M['m00'])
82
83      if c1x > c2x:
84          c1x, c2x = c2x, c1x
85          c1y, c2y = c2y, c1y
86
87      #print 'x1 : %d, y1 : %d || x2 : %d, y2 : %d' % (c1x, c1y,c2x, c2y)
88      diff = (c2y - c1y)
89      print 'The difference is %d' % diff
90      frame = cv2.line(frame, (320, 10), (320, 100), (0,0,0), 5)
91      if diff < 0:
92          p1 = (320,40)
93          p2 = (320 + diff, 70)
94          frame = cv2.rectangle(frame,p1,p2,(255,255,255), cv2.FILLED)
95      else:
96          p1 = (320 + diff, 70)
97          p2 = (320,40)
98          frame = cv2.rectangle(frame,p1,p2,(255,255,255), cv2.FILLED)
99
100 #cv2.drawContours(frame, cnts, -1,(255,0,0), 2)
101 for cnt in cnts:
102     if cv2.contourArea(cnt) > 1000:
103         epsilon = 0.1*cv2.arcLength(cnt,True)
104         approx = cv2.approxPolyDP(cnt,epsilon,True)
105         hull = cv2.convexHull(cnt)
106         M = cv2.moments(hull)
107         cx = int(M['m10']/M['m00'])
108         cy = int(M['m01']/M['m00'])
109         cv2.circle(frame,(cx,cy),1,(255,0,0),3)
```

```
110                 # rows,cols = frame.shape[:2]
111                 # [vx,vy,x,y] = cv2.fitLine(hull,cv2.DIST_L2,0,0.01,0.01)
112                 # lefty = int((-x*vy/vx) + y)
113                 # righty = int(((cols-x)*vy/vx)+y)
114                 # frame = cv2.line(frame,(cols-1,righty),(0,lefty),(0,255,0),2)
115                 cv2.drawContours(frame,[hull],-1,(0,255,0),3)
116             approx = cv2.approxPolyDP(cnt,cv2.arcLength(cnt,True) * 0.005,True)
117
118     cv2.imshow("original", frame)
119
120
121     k = cv2.waitKey(5) & 0xFF
122     if k == 27:
123         break
124
125 cv2.destroyAllWindows()
```

References

[1] *8-bit Atmel Microcontroller with 64/128Kbytes of ISP Flash and USB Controller*. 7593L. ATMEL. Sept. 2012. URL: http://www.atmel.com/images/doc7593.pdf.

[2] Gerald Coley. *BeagleBone Blask System Reference Manual*. BBONEBLK_SRM. Rev. A5.2. Beagleboard. Apr. 2013. URL: https://www.adafruit.com/datasheets/BBB_SRM.pdf.

[3] *General Servo Information*. Ver. 2.0. Hitec. Mar. 2002. URL: http://users.ece.utexas.edu/~valvano/EE345M/Servomanual.pdf.

[4] *Level shifting techniques in I2C-bus design*. AN10441. Rev. 1. NXP Semiconductors. June 2007, p. 4. URL: https://www.adafruit.com/datasheets/AN10441.pdf.

[5] *LV-MaxSonar-EZ Series High Performance Sonar Range Finder*. PD11832c. MaxBotix Inc. URL: http://maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf.

[6] Robert C. Nelson. *BeagleBoardUbuntu Wiki Page*. 2015. URL: http://elinux.org/BeagleBoardUbuntu.

[7] Robert C. Nelson. *Device Tree Overlays for bb.org boards*. 2015. URL: https://github.com/RobertCNelson/bb.org-overlays.

[8] *SN754410 Quadruple Half-H Driver*. SLRS007C. Texas Instrument. Jan. 2015. URL: http://www.ti.com/lit/ds/symlink/sn754410.pdf.

[9] OpenCV Dev Team. *Installation in Linux*. OpenCV. Dec. 2015. URL: http://docs.opencv.org/2.4/doc/tutorials/introduction/linux_install/linux_install.html.