University of Virginia
Department of Electrical and Computer Engineering
Charlottesville, VA 22903

ECE 4440 - Embedded System Design
Instructors: Mircea Stan, Professor
Ben Melton, Teaching Assistant

Team LED
Final Technical Report

Automatic Parking Attendant
With License Plate Recognition

Done by

Harrison Brookeman
Joseph DeGuzman
Hong Moon
Zachary Zydron

December 14, 2013

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Student Signature: ____

Approved: __________________________________ Date ______________________
Professor Mircea Stan, Department of Electrical and Computer Engineering

Abstract

This project, going along with the theme of Computer Engineering and Embedded Systems, is a combination of both software and hardware components. We strived to create an automated parking garage attendant system based on Automatic Number Plate Recognition method for payment automation. Automatic Number Plate Recognition (ANPR) is a method of obtaining license plate numbers from image data that contain license plate regions. The system is able to take pictures of vehicles to store their license plate numbers with timestamps in a database, where the stored license plate numbers are used as unique identification numbers to identify vehicles. The amount of time each vehicle has been in the parking lot will be calculated with the timestamps to determine the payment amount for each customer. The ANPR algorithm has been implemented with *OpenCV* computer vision library and *Tesseract OCR Engine* running on top of a Linux operating system.

The hardware components center around *BeagleBone Black*, a low-cost embedded Linux board with ARM processor that greatly suited this project's needs. Also, push button circuits have been set up to provide user-interface. We have a working application which interacts with the user through a command terminal window and push buttons. The system takes pictures of vehicles containing license plates and analyzes the pictures to find possible license plate regions. The application of various image processing algorithms filters incorrect license plate regions.

Task and Purpose

A usual paid parking garage has a system which prompts a customer to reach for a printed ticket after pressing a button. Each customer keeps the small piece of paper ticket for the entire duration of time parking in the garage and hands the ticket to an attendant usually sitting in a cramped environment to calculate the pay the driver owes for parking in the garage. The system created in this project provides a much more automated system which eliminates the easy-to-lose ticket stubs and the cramped human parking lot attendants. Parking lot attendants will no longer have to sit in a confined environment at night or during the weekends for trivial payment services. The automated camera-based system will also result in the overall reduction of maintenance cost.

Through the use of image processing algorithms and Optical Character Recognition (OCR), we designed an ANPR system in which each license plate number is used as a unique identification number for each vehicle without using a printed ticket.

Introduction

Typical parking lot management systems use paper tickets to mark the entrance times of vehicles to the parking lot. This places the responsibility of maintaining the tickets on the users, despite them having to pay for the service. By using the ANPR algorithm to store the plate numbers and time information in a database, that responsibility is lifted off the users as the system has the ability to recall the time information of the users exiting the facility. Our goal was to use a combination of software and hardware components to create a reliable system for maintaining entrance and exit control. We were able to achieve that mainly through two software libraries: *OpenCV* and *Tesseract-OCR Engine*.

OpenCV is a powerful image manipulation and processing library that contains numerous functions for optimizing and locating objects in an image. Once the plate region has been identified by the *OpenCV* software, the image is passed along to the *Tesseract-OCR Engine*. This open source OCR (Optical Character Recognition) engine is a powerful tool for determining the text contained in an image. We managed to implement image processing algorithms with *OpenCV* to localize the plate region in the image and used Tesseract to extract the license plate characters.

Our implementation uses a small embedded processor running a Linux distribution to provide backend support for the two libraries. The libraries have been installed and run through multiple C++ files. The data for each car are stored in an SQLite database. SQLite was chosen because of its lightweight memory usage, high performance searching ability, and fault tolerance upon the event of power outage.

Theory

OpenCV is what provides the main image processing power for this product. The entire image processing process is broken into multiple steps in a pipeline, as shown in *Figure 1:*

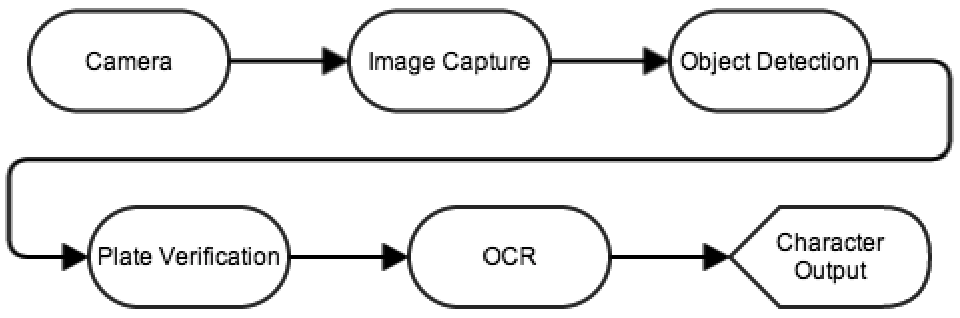


Figure 1: System Layout

The image is captured from the camera by recording a stream through *OpenCV* and taking a single frame from that stream as the image. For our application of taking a picture of a stationary object, this method works appropriately by using a very short stream length. Next, the image is fed into a program which creates a vector of all possible plate regions. As decribed by Escrivá (2012), the initial step taken is to apply a variety of filters on the image in order to remove noise and information that might hinder the plate detection process (pp.167-168). This is done by first creating a grayscale version of the image and removing all colors. Next, a Gaussian blur filter is applied to smooth the edges of the image and remove image noise. Finally, a Sobel filter, which accentuates the edges between objects, is used. As Myler and Weeks (1993) have described, a Sobel filter is an edge detection spatial filter that returns the magnitude and direction of edges (p.218). License plates have high densities of vertical lines, so the Sobel filter allows us to locate potential license plates in the image by identifying those high-density regions. After applying the Sobel filtering, we apply image processing algorithms to find rectangular contours where possible plate regions may exist. As Escrivá (2012) has explained, for all possible plate regions, histogram equalization is applied to distribute the intensities of pixels across the images (p.173).

After all potential plate regions are detected, these sections need to be verified to determine their validity as license plates. Escrivá (2012) has explained that this can be done by looking at the aspect ratio of the detected region as well as the area (p.169). Standard license plates in the United States have an aspect ratio of 2:1, width to height. Due to the different angles and heights of cars that may be imaged, error margins are included to allow for slight variations in the ratio. In addition to the aspect ratio verification, the area of the plate region is also calculated to verify that it falls within the license plate range of regions.

The final step in the image processing pipeline is the Optical Character Recognition (OCR) algorithm. It is contained in a library called *Tesseract OCR Engine*. *Tesseract* is able to read an image and outputs a text document of the characters contained in the image. It is significantly used in reading and interpreting scanned documents, but we were able to utilize its features for our purpose. This library comes with training features for the English character set and multiple fonts. Using *OpenCV*, the plate characters were isolated from other background information and passed into the *Tesseract* software, returning a text file containing the license plate characters from the image.

Apparatus and Instrumentation

BeagleBone Black

The hardware component of our project involves a development board capable of running an operating system that supports all of our software for the project. This board is the BeagleBone Black. Figure 2 shows the board and many of its features and external ports.

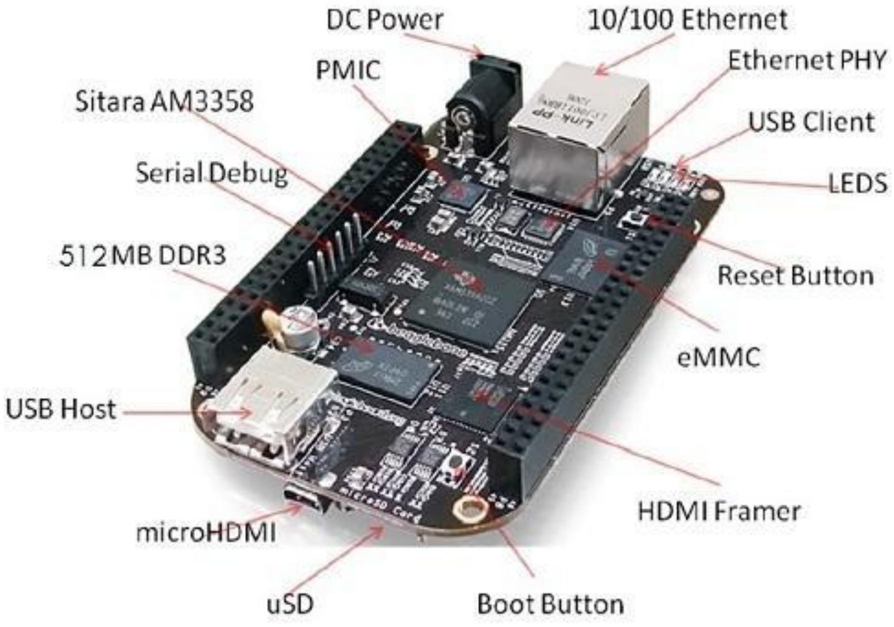


Figure 2: BeagleBone Black layout. Source: BeagleBoard.org., 2013

We took advantage of many of the board's interfaces for our project. The official organization for BeagleBone Black, provides a detailed specification for the board. We were able to easily access the internet with the 10/100 Ethernet port in which the network was available upon plugging a network Ethernet cable into the port. The USB host interface, along with a four-to-one USB hub allowed us to use a mouse, keyboard, and our webcam on the board. For monitor interface, we used the micro-HDMI port to hook our board to an external monitor, through DVI or HDMI, so that we could be able to interface with the board properly. Also, its many GPIO ports provided us with a good method to interface the system with a user through push button circuits, since there are no available buttons on the board. Its processor, the Sitara XAM3359AZCZ100 Cortex A8 ARM by Texas Instruments™, features good performance with running an operating system (BeagleBoard.org, 2013, Black Description, para. 1).

We ran the Angstrom distribution of Linux on top of the board through a downloaded image on a 64 GB micro SD card inserted into the microSD port on the board. The image runs almost instantly when the BBB is powered on after the micro SD card is inserted. The reason we chose Angstrom was that it had pre-installed drivers available for our webcam to taking pictures (Molloy, 2013).

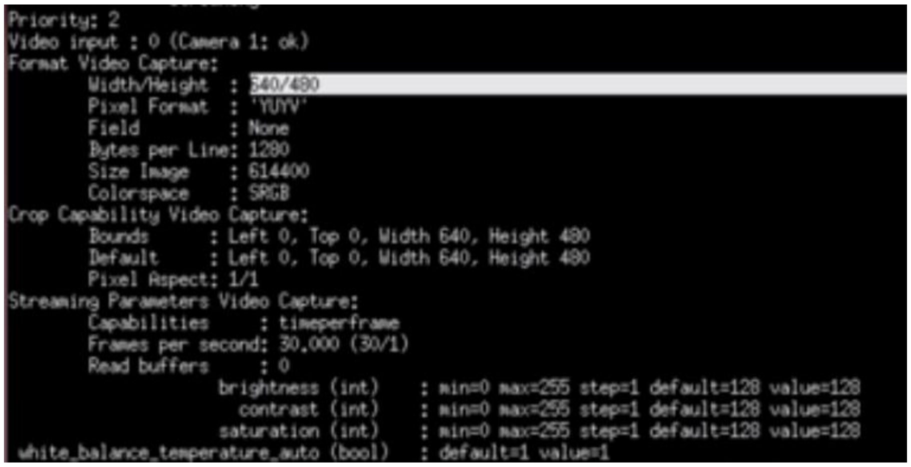
Angstrom is also suitable for embedded systems because of its light use of memory. Also, the image came with a nice window manager GNOME, which we actually replaced it with *Xfce* after realizing the GNOME window manager had not been powerful enough to display processed images to the screen. In spite of its useful features, many different dependencies, which are not readily available in the Angstrom distribution, have to be installed on the board.

<u>The Camera</u>

For taking pictures for processing with our system, we integrated a webcam that interfaces with the board through USB. The webcam we used is the Logitech C920, which has HD capabilities and preprocessing functions with its driver, which as stated before is able to work very well with Angstrom Linux. *Figure 3* displays the Logitech C920 webcam.

Figure 3. Source: *PCMAG*, 2013

Within Angstrom we are able to configure different parameters for taking videos and pictures using the *v4l* (video for Linux) camera interface, as seen in Figure 4.



```
Priority: 2
Video input : 0 (Camera 1: ok)
Format Video Capture:
        Width/Height  : 640/480
        Pixel Format  : 'YUYV'
        Field         : None
        Bytes per Line: 1280
        Size Image    : 614400
        Colorspace    : SRGB
Crop Capability Video Capture:
        Bounds      : Left 0, Top 0, Width 640, Height 480
        Default     : Left 0, Top 0, Width 640, Height 480
        Pixel Aspect: 1/1
Streaming Parameters Video Capture:
        Capabilities     : timeperframe
        Frames per second: 30.000 (30/1)
        Read buffers     : 0
                     brightness (int)    : min=0 max=255 step=1 default=128 value=128
                       contrast (int)    : min=0 max=255 step=1 default=128 value=128
                     saturation (int)    : min=0 max=255 step=1 default=128 value=128
 white_balance_temperature_auto (bool)   : default=1 value=1
```

Figure 4: V4L output of connection to Logitech C920. Source: Molloy

Much of these features though are not used in our project, and we actually used the *OpenCV*'s *VideoCapture* function to take pictures with the webcam and preprocessing of the image.

Push-button Setup

In regard to a user interface, there were no available buttons on the board that a user could push to enter the parking garage. This meant that we had to take advantage of the many GPIO ports on the BBB (BeagleBone Black) and create a pushbutton circuit to provide the interface needed for a user to interact with the system. The circuit diagram for the push-button setup is shown in Figure 5.

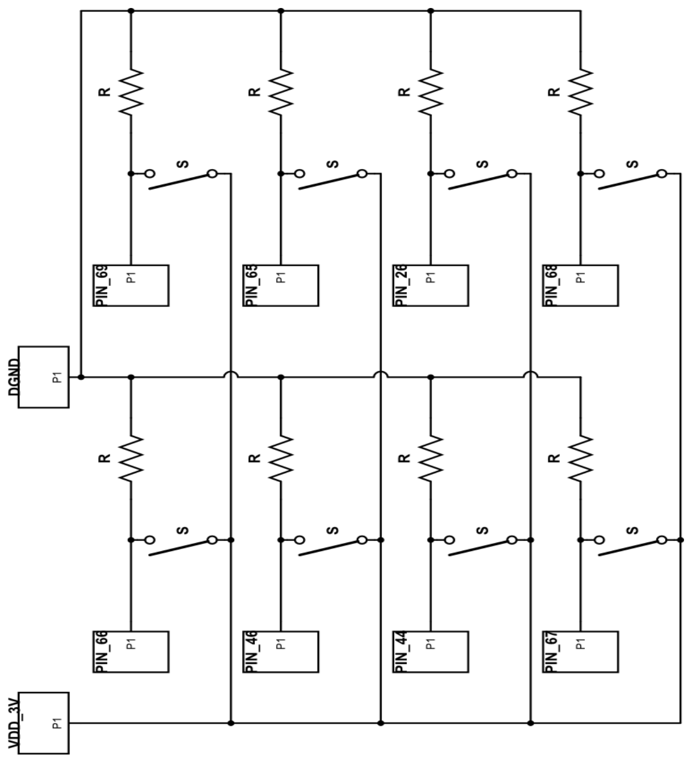


Figure 5: Push-button Configuration

As you can see from Figure 5, there are multiple button setups which are there for demo purposes, showing different cars entering and exiting the garage. The buttons are primarily served to provide user confirmation for incorrect character reading by the OCR component. Each push button is wired between VDD and the GPIO port on the BBB along with a pull-down resistor (of arbitrary values) attached to ground. When pressed while the system is running on the board, the switches send signals to their respective GPIO ports on the BBB, which then reads the high signal and performs whatever action that port is assigned to in the code, which can be vehicle entrance, or exit. For missed OCR reads, however, we implemented a system using keyboard input that lets a customer confirm or deny that the picture taken is of his/her car.

Experimental Procedure

Moving into the design of this project, five main aspects of code required development and research: (1) using the GPIO Pins for button controllers, (2) capturing images with the camera, (3) identifying license plate regions on an image with *OpenCV*, (4) discerning characters from a detected license plate using an OCR, and finally (5) storing consumer data in a database.

Setting up the GPIO Pins

For our project and for the sake of the demo, we required a method of user input that would not require a keyboard. The simplest solution was to create buttons with various GPIO pins available on the Beaglebone Black. In order to interface the hardware of the Beagle board and software, the status of the pins corresponding to each button had to be constantly exported to a file, which could be continuous poled by our C++ code. From some simple research, we were able to determine the simplest method for exporting all the required pins would be through the use of two shell commands:

```
echo 68 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio68/direction
```

**for this example, pin 68 is being exported*

In order to run these shell commands from with a C++ program, the commands were hard coded into strings and then executed using the ***int system(string command)*** Linux system call. A default method ***void prepareButtons()*** was created to detect whether the necessary pins were exported and export them if necessary. The following table, *Figure 6*, displays what GPIO pins were used for each button and function:

Figure 6: GPIO Pin Configuration

GPIO Pin:	**67**	**68**	**44**	**26**	**46**	**65**
Demo 1 Function:	Car AIN	Car AOUT	Car BIN	Car BOUT	Car CIN	Car COUT
Demo 2 Function:	Car Enter	Car Exit	Unused	Unused	Unused	Unused

Capturing Images with the camera

The operating system we selected for the design was already equipped with packages for camera control, such as *v4l* (video for Linux). Also, there was already support for detecting and manipulating the camera without installing drivers. In order to handle image capturing in C++, we wrote this simple method:

```
string captureImage() {
        VideoCapture cap(0);                          // 1.

        cap.set(CV_CAP_PROP_FRAME_WIDTH,960);         // 2.
        cap.set(CV_CAP_PROP_FRAME_HEIGHT,720);

        if(!cap.isOpened()) return "error";           // 3.

        Mat frame;
        cap >> frame;                                 // 4.
        imwrite("capture.jpg", frame);                // 5.
        return "capture.jpg";
}
```

__Method execution__:

1. Create a *VideoCapture* object that locates camera 0. Camera 0 must be a Logitech camera because it is the only one connected to this system.
2. Change the resolution of the image captured to 960 x 720 pixels
3. Handle an error if camera is either not available or is currently being used by another program
4. Save the current image sensor data to a matrix
5. Write the image matrix data to a usable .jpg file, titled *capture.jpg*

The method identifies the completed execution upon returning the "capture.jpg" string, which also signifies the file that will be used for the plate detection.

__Method augmentations__:

We found that there were some internal errors thrown when accessing the camera. Given that these errors did not inhibit the function of the camera, we decided to prevent them from displaying in the terminal. The following arguments were added within the ***captureImage()*** method to prevent the errors from showing:

**this occurs prior to the creation of the VideoCapture object*

```
int back_fd, new_fd;
fflush(stdout);
back_fd = dup(2);
new_fd = open("/dev/null", O_WRONLY);
dup2(new_fd,2);
close(new_fd);
```

**this occurs after calling imwrite() on fram, but before the return statment*

```
fflush(stdout);
dup2(back_fd,2);
close(back_fd);
```

Using the redirection technique, these two sections of code work to send the error flags in the terminal to another location instead of the standard output. Following the ***imwrite("capture.jpg", frame)***, the standard output location is returned to normal so that further debugging can take place without error declarations being missed.

Detecting Plate Regions with *OpenCV*

In this section, the process for identifying plate regions will be discussed. This process includes the following aspects: contour detection, Support Vector Machine plate verification, and finally character region detection.

The foundation for this aspect of the project was the Chapter 5 Number Plate Detection project expressed in the resource *Mastering OpenCV with Practice Computer Vision Projects* (Escrivá , 2012, pp. 161-188). In this chapter, a small program was used to detect the regions of Spanish license plates for character detection. Code for this application was found in an online repository cited by the book.

Plate Contour Detection

The first task to detect plate regions from a license plate is to find all the regions with a certain geometric pattern within an image. We found that the best way to do this is to create a C++ vector object in which we can store the regions of the image that satisfy certain license plate constraints. In order to customize this function to work with our situation, i.e. finding the Virginia license plates, we needed to alter the aspect ratio of license plate rectangles and the generalized location of the characters on the plates. In the United States we know that the license plate size must be 12’’ x 6.3’’ which can then vary to 12” x 6”. This gives us a pretty good idea of what an acceptable aspect ratio would be – 2. Since we can set the error tolerance for the detection, the aspect does not have to be set to an exact value, which is 1.905.

Various image operations for the possible plate regions, including relations to aspect ratio, can be found in the *DetectRegions.cpp* file. The operations include thresholds, morphological operations and filtering. The file *DetectRegions.cpp* is used with *main.cpp* to create a list of all possible plate regions. Then additional operations determine which region is the most likely region.

Classifying the Plate Region

After the input image from the camera has been separated into image segments and added to the list of possible plate regions accordingly, operations must be done to determine which region most likely contains the plate. Our next decision was to get a Support Vector Machine (SVM) that could be trained so that we could iteratively go through the list of plate regions and determine whether a plate was present or not (Escrivá, 2012, pp. 173-176). This process required a collection of numerous pictures of actual license plates from which we could crop out the correct regions.

In order to train the SVM algorithm, we needed to collect sample images that would be good examples of vehicles in the parking garage scenario. We decided to go into the parking lot outside of Rice Hall and collect images of the license plates available using the Logitech camera that would be used in real life. We made sure only to capture images of Virginia license plates from the front of the vehicle to reduce some of the great variation in vehicle license plates. Following this task, we have accumulated about 136 images for which we could crop and use for either plate or non-plate regions to train the SVM.

An SVM uses pre-selected training images to create a neural network that is able to distinguish a plate region from non-plate regions. This means that the more images used to train the SVM, the larger the XML file for the SVM (SVM.xml). By trial and error, we found that SVM.xml, although accurate enough to detect plate regions, would be too large for the Beagle Board to process without the RAM filling. For instance, using about 1300 images to train the SVM, we created an SVM.xml file with the size of about 92.1MB, which produced issues with the available RAM on the device. In addition, when the max size of the SVM.xml was acquired, the amount of time needed to identify plate regions increased significantly. The SVM trained with only 46 images (about 3.2MB) created a delay of more than 15 seconds alone. This delay would potentially put a large wait time in the finalized program and, without a doubt, discourage customers. Unfortunately, despite the work put into implementing a Support Vector Machine in our program, we decided to abort and try to find a new way to identify plate regions from the list.

The SVM was removed from plate region list operations, but we found that, more often than not, the image located in the 0th location of the list was the correct plate region. In this case we decided to press on assuming that we could always identify plate region zero as the correct region. We also added an error-handling routine in case the assumption was incorrect.

Character Region Detection

The final step in identifying the plate before sending the results to the OCR for character identification is the specific region identification of the characters on the plate. Based on the images that we had collected earlier in the project, we identified the characters as shown in Figure 7:

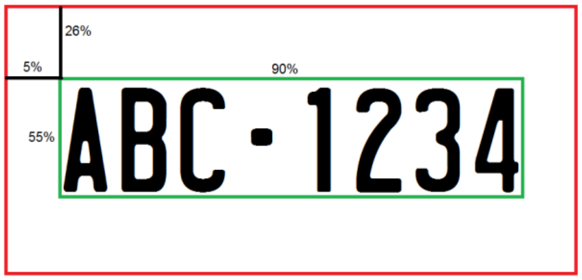


Figure 7: License Plate Character Regions

Essentially the final region sent to the OCR (in relation to the detected license plate region width and height) is 90% by 55%, with the origin brought down by 26% from the top and the width by 5% from the left.

Identifying Characters with an OCR

The OCR procedure used to identify characters on the plate regions from Escrivá (2012) was only set to observe the Spanish language and constraints on characters used for their license plate (pp. 177-184). This made the included OCR not compatible with our project as we needed a specific list of characters to register within our image only. As a result we decided to use *Tesseract* for OCR support.

Since *Tesseract* is a generic package that is designed to simply detect characters from images, we needed to optimize it for our code, given the constraints from Virginia license plates. The first change necessary was to create a “white-list” that would tell *Tesseract* what characters would be allowed from the input images. This method prevents the OCR from identifying characters that cannot appear on a license plate (like punctuation and special characters) but replaces them with the next best characters that we determined as acceptable. To create this “white-list” we had to add a config file to the *Tesseract* package location in our root directory. This was an extension-less file entitled “letters” with the following contents:

```
tessedit_char_whitelist ABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789
```

In order to implement this into our code, we simply added the “letters” flag to the end of the typical *Tesseract* command line argument to identify that this configuration would be used. Since *Tesseract* is used with a command shell argument, the only way to implement this within our C++ code is with another ***system(string command)***. This will then send the string identified by the OCR to a text document named according to our designation (we used “out.txt”) from which we can read the detected license plate number.

Some further simple if/then statements were added to the code to correct the wrong characters. This logic was deduced from the following facts:

1. A license plate cannot have numbers before a dash.
2. It cannot have letters after the dash.

Therefore we could replace incorrectly located characters with their next best candidates. For example, we replaced “S” found after the dash with the number 5 because it would be the most likely replacement.

Sqlite Database and Error Correction

In our system, there are several major benefits from using the *Sqlite* database to store the license plate numbers and the times when vehicles enter the parking garage. The most appealing feature of the

Sqlite database is that it is very light-weight. Unlike standard database management systems commonly used on general purpose computers, *Sqlite* requires a very little amount of memory for storage, which makes it a very attractive choice for embedded applications. Also, by storing data inside the database, we can provide fault-tolerance upon the event of power outage. If we use local text files or application-level data-structures, such as a hash table, all the data that have been previously stored will be lost. However, by storing data inside the database, we can securely keep all the data without any loss, and run the application back to continue providing services to users.

Also, instead of using text files to store information, storing data inside the *Sqlite* database allowed us to have faster search results and flexible search options using SQL queries. To make the interface to the *Sqlite* database with SQL queries cleaner and simpler, we created an object called *DBManager*, which has methods abstracting the execution of SQL queries, such as insert, delete, and select. In a similar fashion, we created a method called *selectWithWildCard (string)*.

For example, if a given license plate number is "ABC-1234," the following queries, each including a single wildcard character, will be executed in our error correction algorithm with *selectWithWildCard (string)* method, as shown in Figure 8:

```
_BC-1234
A_C-1234
AB_-1234
ABC_1234
ABC-_234
ABC-1_234
ABC-12_4
ABC-123_
```

Figure 8. Wildcard search queries that will be executed for a plate number "ABC-1234"

From our experimental results, we found that while *Tesseract OCR Engine* yielded mostly correct results, there were many cases in which only one character was missed. So we used the SQL wildcard search queries, assuming that we had obtained results that were almost correct but one character. By iterating through each character in the license plate, we could improve the accuracy of the license plate recognition. For every result obtained from the wildcard searches, we show the image of the license plate stored within the file system and ask the user whether the license plate is correct using a keyboard command interface.

Conclusion and Discussion

The results of the project involve a working system with user interface through the command terminal on Angstrom Linux and the push button setup. The accuracy of the license plate recognition is quite satisfactory with many of the images taken with the camera for demonstration purposes.

Errors that did arise in our program resulted from images that would otherwise not occur in our ideal parking garage scenario. In this parking garage scenario such variables as light and camera angle will all be controlled to remove glare and rotation of the license plate in the captured image. These corrections based on environment controls would result in a more accurate system but further precision improvements could be made with the use of an infrared camera.

The Optical Character Recognition, through *Tesseract*, also works quite well and only has slight inaccuracies for every few runs of the system. Our error correction algorithms (incorrect character replacement discussed earlier in Experimental Procedure) make these inaccuracies very minimal.

The runtime for each run of the system is also quite ideal. Initially, when adding a Support Vector Machine (SVM) the program would take a very long time to read the training xml file, usually around 10 MB in size, and would have to run each time a car enters or exits. We decided to remove this feature because our license plate detection through filtering and edge detection worked quite well, and through the use of area verification for the possible plate regions we were able to find the actual plate, without the SVM. After removing the SVM features, the program runs at less than 5 seconds after user interface is detected, e.g. a button is pressed for entrance or exit.

All in all, our system works quite well and has all basic functionality decided on in the beginning of the project. It runs quite fast and returns reliable results for most runs and with error handling these inaccuracies are miniscule.

References

BeagleBoard.org. (2013, December 13). *Beagleboard:BeagleBoneBlack* (Tech.). Retrieved December 14, 2013, from BeagleBoard.org website: http://elinux.org/Beagleboard:BeagleBoneBlack#BeagleBone_Black_Description

Escrivá, D. M. (2012). Number plate recognition using SVM and neural networks. In *Mastering OpenCV with practical computer vision projects* (pp. 161-188). Birmingham, UK: Packt Publishing.

Logitech C920. (n.d.). *PCMAG*. Retrieved from http://www.pcmag.com/article2/0,2817,2400114,00.asp

Molloy, D. (Director). (2013, May 25). *Beaglebone: Video capture and image processing on embedded Linux using OpenCV* [Video]. Retrieved December 14, 2013, from http://www.youtube.com/watch?v=8QouvYMfmQo#t=525

Myler, H. R., & Weeks, A. R. (1993). *The pocket handbook of image processing algorithms in C*. Englewood Cliifs, NJ: Prentice Hall.