Autonomous Pothole Detection and Cataloging for Bikes

By

Harshvardhan Bhatia

Jesse Chen

Andy Sun

Final Report for ECE 445, Senior Design, Spring 2018

TA: Xinrui Zhu

May 2018
Project No. 62

Abstract

We designed a bicycle mounted system to automatically detect potholes through computer vision and accelerometer data and catalog the coordinates to a database. At the same time the system will also check the location of the rider and warn them of nearby potholes stored in the database.

We successfully implemented and integrated most of the features, but the computer vision aspect still needs tuning to improve the accuracy to practical levels.

Contents

1. Introduction

Potholes are an issue which plague cities all around the world. While damaging to cars, potholes are particularly dangerous, even fatal, for bikers and can lead to millions of dollars in lawsuits for a city if not patched [1]. Bikers need to be very aware of the surroundings around them while on the road, but with so much going on, it can be easy to miss potholes, both at day and at night. However, for cities to be able to fix potholes, they need to first know where they are. Currently, the city of Champaign is starting to utilize a phone application for reporting problems like potholes, but few bikers are going to stop, get off their bikes, and pull out their phone to fill in a report [2]. Most cities simply utilize a telephone number or website for pothole reporting, which is even more inconvenient.

Our project aims to ease the issues of both pothole cataloging for municipalities as well as pothole warning for bikers. We aim to do this by creating a device which can detect potholes via computer vision (for long-range detection) as well as accelerometers (for potholes hit). In addition, a user can report a pothole they ride past by pressing a button. Once a pothole is detected by the system, the device's current GPS location and time is sent to a database for municipalities to access. Devices will also host a local copy of this database, which will be used to warn bikers via haptic feedback if they are riding towards an area with many potholes. In addition, when the computer vision portion detects a pothole, the rider will also be alerted, in case they didn't spot it themselves. Please refer to Figure 1 while reading the following subsections.

1.1 Power

This module oversees powering the entire system. The Li-Ion battery will be a standard phone battery pack with a USB connection. We need a 5V output for the Beaglebone. We need the 3.3V for the rest of the sensors. It is important to note that the Beaglebone tends have a high-power draw which is why even though our battery pack has a voltage of 5V, we run it through a regulator to stabilize the voltage noise so that the Beaglebone doesn't shut down unexpectedly.

1.2 Computer Vision

The computer vision system consists of a camera and a dedicated MCU for image processing. Whenever a pothole is detected, the computer vision module sends a high signal over one of its GPIO pins to the control module. The computer vision module is the only detection method that provides the user with real time warning of new potholes.

1.3 Control/Feedback

The control/feedback module is in charge controlling the entire system depending upon the various sensors as well as computer vision module. It consists of an LPC1114 MCU that is connected to the GPS, Accelerometer, Bluetooth, Button, as well as the Buzzer.

At a high level, its requirement is to use input of the sensors to detect a pothole. Then it uses the GPS coordinates to store the location of the module in a database. For our system the database is in the local flash of the MCU however for future iterations it would be in the cloud, and the MCU would communicate with it using Bluetooth paired with your phone. This module is also in charge of checking current GPS coordinates to those in the database. If the biker is headed towards a pothole then it warns the user using the haptic feedback buzzer. A more quantitative requirement for each specific sensor within this module can be found in R&V table in Appendix A.

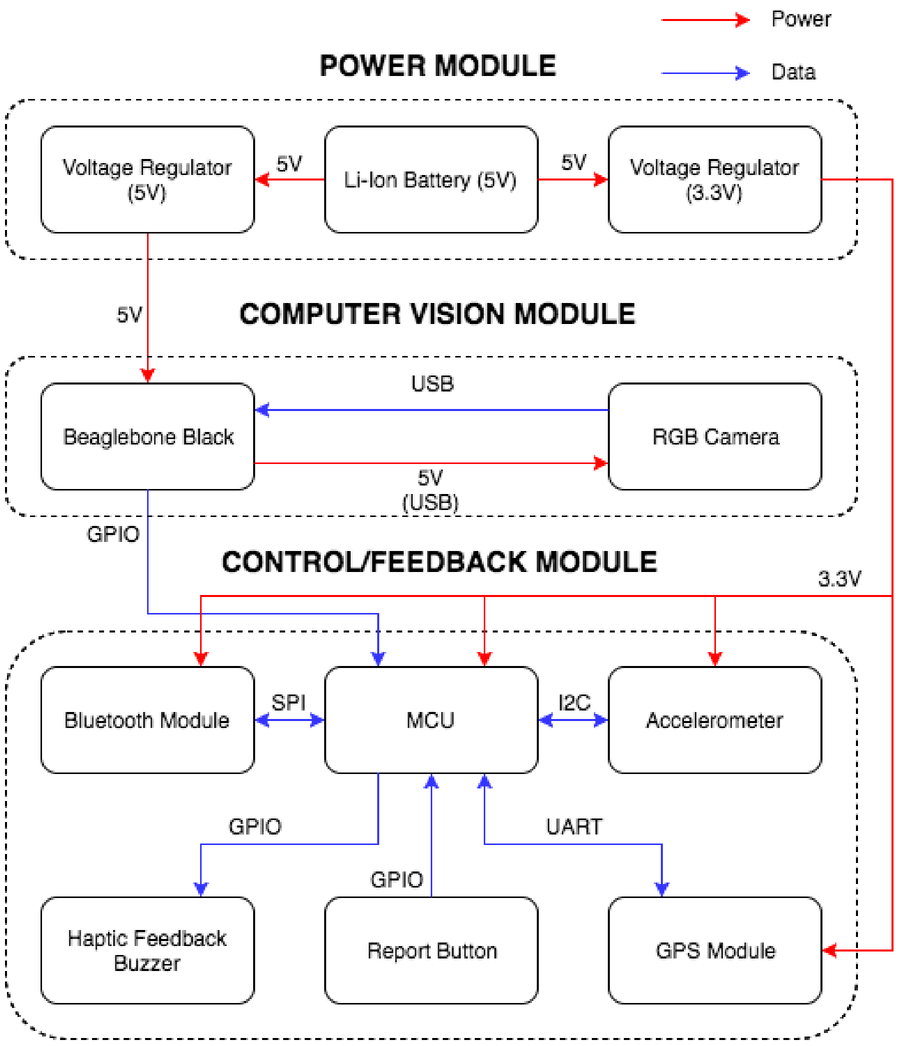


Figure 1: Block Diagram

2 Design

2.1 Control

Controlling the system efficiently was important to us since there are multiple sensors communicating via multiple interfaces. We were looking to avoid interrupts. They tend to mess up the system when there are many of them, however several of the sensors that we were using had certain specifications that made it unavoidable. We decided to go with a main loop that is controlled by a set of flags. As shown in Figure 2 the loop relies on four flags that are manipulated by interrupt and timed functions. As shown by table 1 the two potDet flags are set when a pothole is detected. In this case the canDet flag is used to control the potDet flags so that we don't upload a similar location for potholes repeatedly. The potEnc flag is only set when the biker is about to hit an area with potholes. We went with 2 different potDet flags because we want the computer vision module to be able to interrupt the loop separately so that if the module detects a pothole in advance we can warn the user immediately.

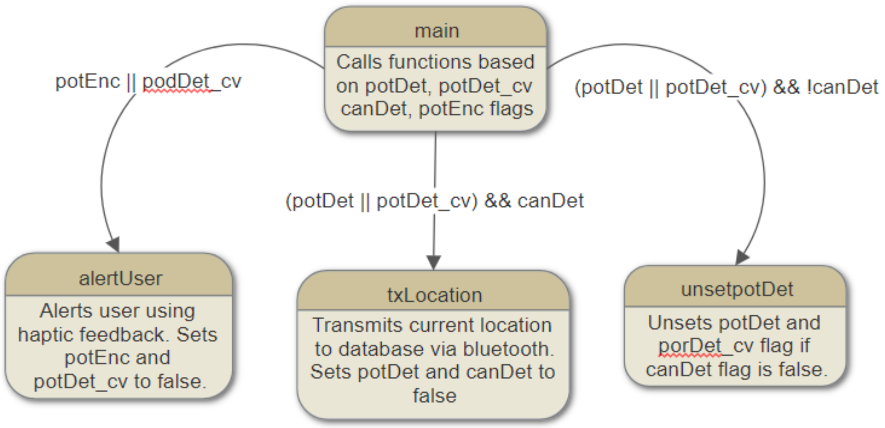

main
Calls functions based on potDet, potDet_cv canDet, potEnc flags
potEnc || podDet_cv
(potDet || potDet_cv) && !canDet
(potDet || potDet_cv) && canDet
alertUser
Alerts user using haptic feedback. Sets potEnc and potDet_cv to false.
txLocation
Transmits current location to database via bluetooth. Sets potDet and canDet to false
unsetpotDet
Unsets potDet and porDet_cv flag if canDet flag is false.

Figure 2: Main Control Loop

Table 1: Control Flags	
Flags	**Function**
potEnc	Set when pothole will soon be encountered
potDet	Set when pothole is detected by button or accelerometer
potDet_cv	Set when pothole detected by cv module
canDet	Set when a pothole has not been detected recently

Figure 3 shows a high-level diagram of the interrupt-based functions that occur. The MCU requests data from the accelerometer at 120Hz and can set the potDet flag if it detects a pothole. New GPS coordinates polled every second. This is will be addressed in the GPS section in more detail.

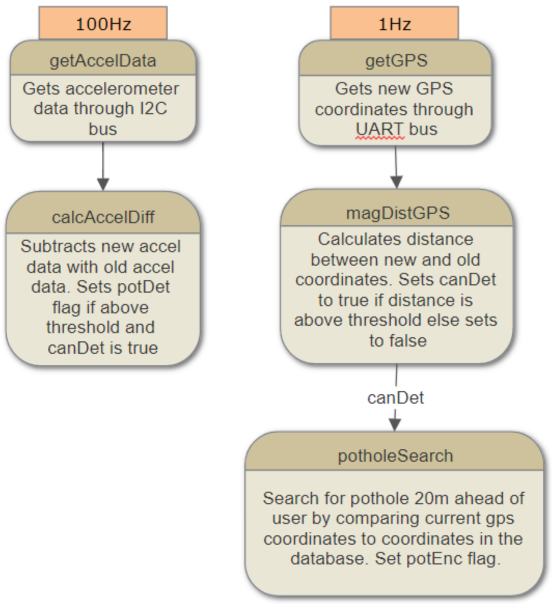

100Hz
getAccelData
Gets accelerometer data through I2C bus
calcAccelDiff
Subtracts new accel data with old accel data. Sets potDet flag if above threshold and canDet is true
1Hz
getGPS
Gets new GPS coordinates through UART bus
magDistGPS
Calculates distance between new and old coordinates. Sets canDet to true if distance is above threshold else sets to false
canDet
potholeSearch
Search for pothole 20m ahead of user by comparing current gps coordinates to coordinates in the database. Set potEnc flag.
Interrupt
cvDetHandler
Sets potDet_cv when beaglebone GPIO port is transitions to high
2Hz
checkButton
Sets potDet when button GPIO port is high

Figure 3: Interrupt and Timer Functions

2.2 Accelerometer

The accelerometer was used for scenarios when the biker hits a pothole. Design considerations were decided by referencing several research papers that used accelerometers for the same purpose.

The main design consideration for the accelerometer was choosing an algorithm for pothole detection. After reading several research papers, it was decided that we should use a Z-Differential algorithm. This takes the current g value and subtracts it from the previous g value as shown in Figure 4. The difference if the difference is higher than a previously set threshold value then the biker has ridden over a pothole. Figuring out the thresholding value proved to be challenging since the flash memory on the MCU was not big enough to store enough values for a test. If we received values through our UART debugger then we would have to bike with the laptop which would likely affect the data. We decided to use the Bluetooth output to find the optimum threshold value. Table 2 describes the observations that we had during our testing and shows why we landed up at 1.95g as our threshold value.

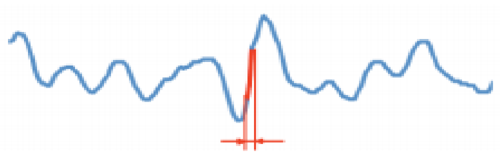


Figure 4: Z-Differential Method

Table 2: Accelerometer Observations

Threshold	Observations
1.95g	Major potholes Accurate and reliable
1.4g - 1.9g	Pothole detected for bumpy pavement Seemingly random detections
<1.4g	Random detections

Another aspect of design had to do with the range of the accelerometer. Our chip had a ±1.5g range with a 6-bit output. However, while testing the system we realized that the accelerometer spread was a lot higher than expected and the 6-bit output was not precise enough for testing purposes. Given more time an accelerometer with greater than ±5g range and 12-bit output would be desirable.

2.3 Computer Vision

When designing the computer vision module the major decisions we made were about how the module would interface with the rest of the project and which detection algorithm to use. The computer vision module operates asynchronously to the main control module and only communicates in one direction from the computer vision module to the control module. We chose this design for its simplicity and is supported by the fact the that potholes will generally be encountered infrequently. We chose the algorithm we used because it closely mirrored our application and seemed to be computationally simple enough to run quickly on an embedded system. Other algorithms were designed to detect potholes from a top down view or did not run in real time.

The detection algorithm we used is adapted from the paper *Pothole Detection System Using a Black-box Camera* by Youngtae Jo and Seungki Ryu. It follows the flowchart in Figure 5.

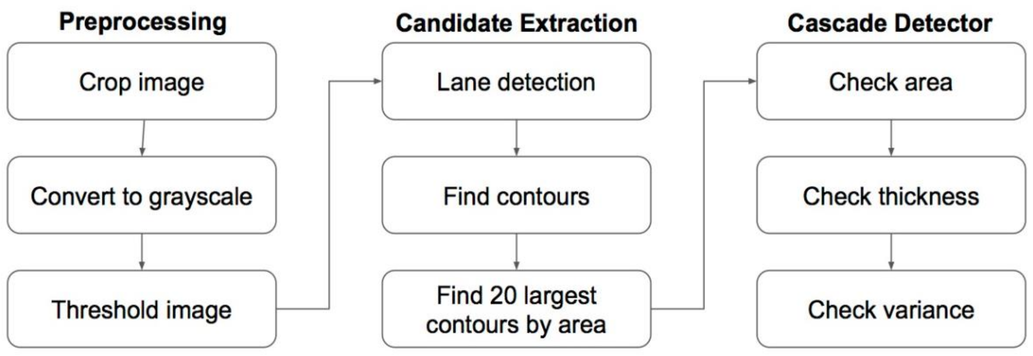


Figure 5: Computer Vision Flowchart

The image is cropped to remove the sky to improve the performance of thresholding in a later step. The image is then converted to grayscale to simplify computations and is possible since we do not consider any color information. Next, the image is thresholded to create a binary image. Every pixel that is brighter than the threshold is changed to zero and every pixel less than the threshold is changed to one. The threshold value is determined using Otsu's method. It searches for the value that minimizes the intraclass variance. Mathematically it can be written as follows.

$$T_b = argmin_{0 \leq x \leq L-1} \{H_1 \sigma_1{}^2(x) + H_2 \sigma_2{}^2(x)\}$$

Where T_b is the threshold value, H_i is the sum of pixel values in class i, and $\sigma_i(x)$ is the variance of class i as created by a threshold value of x. Lane detection finds the boundaries of the road and excludes all regions outside of the boundary. Since potholes only exist on the road, we can ignore all other areas. The paper recommends using automatic lane detection, but in our implementation, we used a hardcoded mask for simplicity and to save on computation. Next, we group neighboring pixels th at passed the threshold to generate a list of candidates for possible potholes. We then take the 20 largest candidates to run through the cascade detector. This step is necessary because early on we ran into issues with an excessive number of trivial candi dates, in the range of tens of thousands, that are only a few pixels large. Twenty allows us to check most of the viable candidates without sacrificing performance. The cascade detector runs each candidate through a series of tests. The candidate must pass every test to be classified as a pothole. First the area of the candidate is checked to make sure it is within a range of common pothole sizes. Then we apply a distance transform to the pothole which calculates the distance of every non-zero pixel to the nearest zero pixel. To compute the average thickness of the pothole we find the mean of the non-zero pixels in the distance transformed image. This is to check the candidate has roughly the correct shape and is not a large network of cracks for instance. The final test is to calculate the variance in brightness of the candidate. Potholes tend to have a higher variance compared to shadows.

2.4 GPS

The GPS was an essential part of our project, as it was the only method we had of locating the user and potholes. We decided upon the FGPMMOPA6H SOC, which utilizes a MediaTek MTK3339 chipset, as it had a patch antenna, so we didn't have to design our own PCB trace antenna and was lost cost. One downside of using a MediaTek chipset is that we couldn't query the chi p for location data; it just outputs the data at a predefined interval (usually 1 second). Other vendors, such as SiRFstar, allow for querying of data, which may have simplified some design portions. The accuracy was rated to be around 3m in open blue-sky conditions, which is standard for consumer GPS modules. Since the chip communicated via UART, the first step was to write a UART driver for our MCU.

The data outputted from the GPS is formatted as an ASCII string following the NMEA protocol, which is essentially the standard for all GPS chips. This protocol consists of several types of messages, some of which are standard among all chips and some of which are proprietary to the vendor. Each message begins with a '$' character and ends with the '\r\n' characters. In addition, every message consists of a variety of data fields which are separated by commas. Figure 6 shows an example of an 'RMC' message.

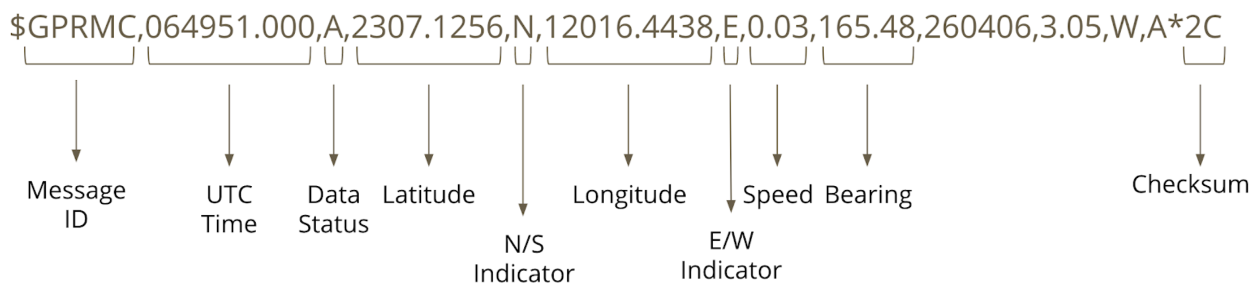


Figure 6: GPS Dataflow

The message ID field tells us that this is an 'RMC' message. The data status and checksum fields are used to tell if we want to parse the message. When the data status field is 'A', that means there is valid data present in the message, so we should parse it. The checksum is calculated by performing a bitwise XOR to all characters between '$' and '*'. If the checksum that we calculate does not match the checksum at the end of the message, that means something was corrupted during transmission and we throw the message away. The remaining fields contain useful information which we store if the data is valid.

By default, the GPS chip outputs this 'RMC' message, plus four other message types, every second. We don't care about these other messages though, so at program bootup, we send a message to the GPS to only output the 'RMC' messages.

In addition, the default baudrate of the GPS is 9600 bits per second, which is a little slow when transmitting such long messages. When considering the start and stop bytes required by the UART protocol, the transfer rate is about 960 characters/second. Since a typical RMC message is around 78 characters long, that leads to a transmit time of nearly 78960=0.08125s every second. This means a minimum of 8.125% of our processing time was used just to read GPS messages. This led to issues where we would be interrupted while reading messages since it took so long to transmit. Therefore, at program bootup we also modify the baudrate to be 115200, which causes the read time to account for only 0.677% of our processing time.

After initialization, we wait for the '$' character to be transmitted, after which we start reading additional characters into a buffer until we find the '\r\n' end characters. A flag is set indicating a new message is ready to be processed. When the main loop finds this flag set, it calls functions to verify the message by checking the data status and checksum fields. If these tests pass, then we parse the message and store the data locally. A summary of this software flow is shown in Figure 7.

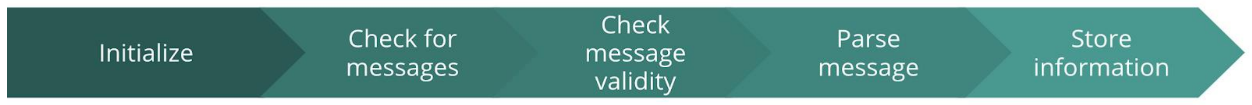


Figure 7: GPS Software Flow

2.5 Bluetooth

Bluetooth is used to simulate a link with a cloud server where the main pothole database would be located. We originally were planning on just having a local database consisting of the potholes the user has detected but realized there was a much greater benefit in having a general database which all users add to. It also provided a very useful tool for debugging, as we can transmit debug information to our phones remotely without having to rely on a wired UART connection.

We explored other ways to simulate this cloud database, such as a SIM card with 2G conne ctivity, but this proved to be very expensive as we would have to purchase a SIM card and data plan.

The Bluetooth module communicated with the MCU via a modified SPI protocol. Due to the hardware limitations of the underlying nRF51822 SOC, a 100us delay had to be added between asserting the chip select (CS) line and writing any data on the SPI bus. In addition, rather than toggling the CS line every byte, the CS line had to be asserted for the entirely of a packet, which could be up to 20 bytes.

Whenever we want to send data over Bluetooth, we store it into a character buffer and transfer it to the Bluetooth module using Adafruit's Simple Data Exchange Protocol (SDEP). If the message is greater than 16 characters, we must break it into separate SDEP packets and send them separately, indicating in each packet if there is another packet following it.

2.6 Database

In addition to a cloud database, we wanted to implement a local database since bike rides often take place in the countryside where cellular connection is not reliable. The concept was that the rider would download the most recent version of the database beforehand and store it in local memory before their ride. Since this database needed to be saved between power cycles, it had to be stored in non-volatile memory.

2.6.1 Flash Memory

The flash memory present on our MCU proved to be a convenient storage location. It consists of 8 sectors of 4 kB each, totaling 32 kB. However, much of that is used to store the code, so we can only reserve the last 4kB of memory for our database. This corresponds to around 240 pothole entries, where each entry consists of two doubles corresponding to the latitude and longitude of the pothole.

One issue was that the MCU utilized NAND flash memory, which doesn't allow writes to individual memory addresses like SRAM or other volatile memories do. We can only write in 256-byte chunks and only on 256-byte word boundaries. In addition, NAND memory writes can only flip 1's to 0's. To turn 0's to 1's requires erasing an entire sector to all 1's. For us, this means erasing our entire database. Therefore, the challenge was to somehow store all potholes as well as our current index in the database without writing any 0's to 1's. The index is needed to know where to store the next pothole location.

We accomplish this by using the first 256 bytes to store our current index in the database. These 256 bytes all start off as 0xFF. Every time we write a new pothole location, we rewrite this 256-byte section with an additional 0x00 byte and fill the rest with 0xFF. The next time the device is started, it reads the number of 0x00 bytes at memory location 0x0007000 (where the database is stored) which gives it the current database index.

Whenever we write a new pothole location, to avoid flipping any unnecessary 1's to 0's, we must pad our new pothole data with data from previous potholes before it and 0xFF's after it. This is shown in Figure 8.

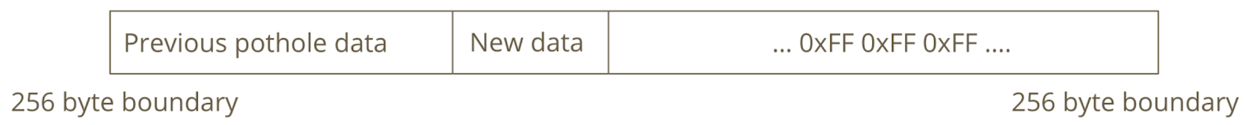


Figure 8

This way, only the bits at the memory location where the new data is located are changed.

2.8 Pothole Search

One of the main functionalities of our device was to warn the user of any nearby potholes. This essentially consisted of us taking the user's current location and comparing it to potholes in our database.

The search was done in two stages. In stage 1, we check if the user is within 25m of any pothole in our database. If it is, we store it in a list of risky potholes. We continue until three potholes are stored or until the database is exhausted.

If we stored any potholes, we go to stage 2 once we receive the next GPS coordinate. In stage 2, we check if we are still within 25 m of any risky potholes. If we are, we calculate the direction the user should be heading to hit the pothole and compare it to the user's actual direction (taken from the GPS). If they are within 60 degrees of each other, we alert the user.

Calculating the distance between two GPS coordinates is not a trivial task, as the coordinates represent points on an ellipsoid surface. We tried three equations to do this calculation, shown below:

Haversine

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$
$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R \cdot c$$

Law of Cosines

$$d = \operatorname{acos}(\sin\varphi_1 \cdot \sin\varphi_2 + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \cos\Delta\lambda) \cdot R$$

Equirectangular Approximation

$$x = \Delta\lambda \cdot \cos \varphi_m$$
$$y = \Delta\varphi$$
$$d = R \cdot \sqrt{x^2 + y^2}$$

All three equations gave similar accuracy for our use case, but equirectangular approximation proved to be the fastest as shown in Figure 8.

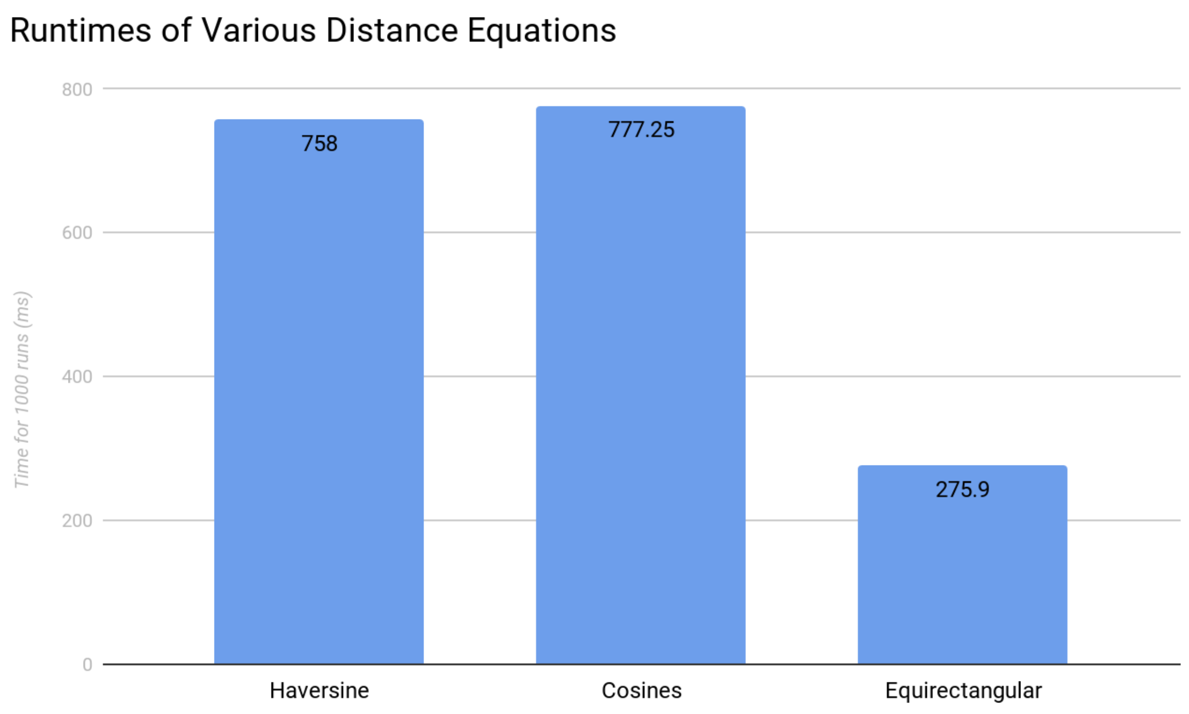


Figure 9

The equation used to calculate the bearing between two GPS coordinates is shown below:

$$\theta = \operatorname{atan2}(\sin \Delta\lambda \cdot \cos \varphi_2 , \cos \varphi_1 \cdot \sin \varphi_2 - \sin \varphi_1 \cdot \cos \varphi_2 \cdot \cos \Delta\lambda)$$

2.9 PCB

While designing the PCB we decided to go with a relatively large board that is easy to modify and debug. We realized that we would prefer to debug through UART since it is easier to look at data as the system is running, however the MCU only has one UART interface. To solve this problem, we added two electrically controlled single pole double throw switches to the TX and RX lines of the MCU. Whenever the MCU wants to use printf statements for debugging it sets a line high the turns the switch from the GPS UART to the debugger UART.

We also added several test pads around the PCB for easier debugging and testing. Please check Appendix C for the PCB diagram.

2.10 Physical Design

To mount the device on the bike we decided to 3D print a base for the PCB, camera and Beaglebone. The base had to be big enough to contain the 2 MCUs as well as have the frontside long enough for a camera mount. As shown in figure 9 we were able to mount the PCB on the base, however we never mounted the Beaglebone since we did not use the Beaglebone for the computer vision part in our final demo. The yellow armband that the user wears on the right is for the haptic feedback buzzer. We expect further iterations to be waterproof as well as more compact, as the PCB becomes smaller.

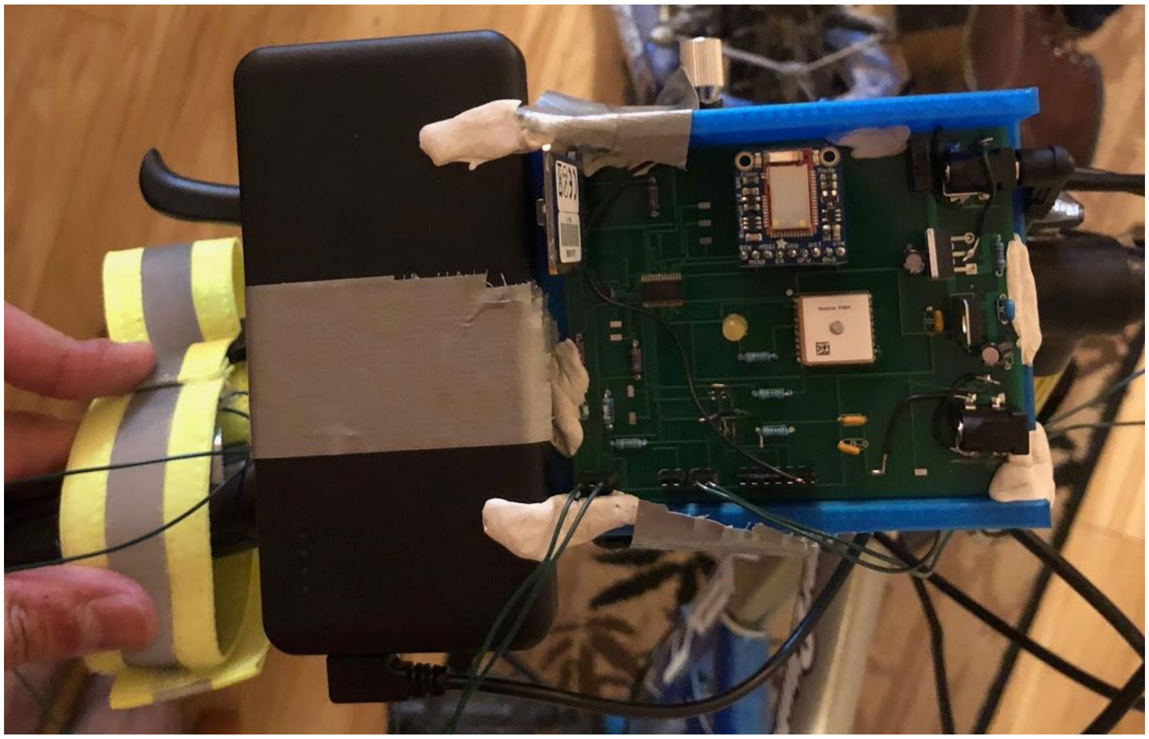


Figure 10: Photo of prototype model

3. Verification

3.1 Accelerometer

The main requirement for the accelerometer was to detect potholes with a true positive rate of 80%. We tested this at a parking lot which simulated three various parts of a road. A pothole, bumpy paved road, and smooth paved road. We then rode the bike around all three parts and counting that as one lap. We took the bike around for 5 laps and recorded the true positive rate shown in Table 3. We tested for several thresholds finally using 1.95g. Since this part of our requirements worked, we were able to confirm the verification of the I2C communication, and other accelerometer requirements on the R&V table.

Table 3

Threshold	True Positive Rate
1.95	100%
1.4-1.9	30%-70%
<1.4	<10%

3.2 Computer Vision

The main requirements for our computer vision module were to run at real time and be accurate. We computed the average speed for 20 frames and found the system took about 79ms per frame to process or about 13 frames per second, satisfying our real time requirement. In terms of accuracy, the computer vision was a bit too sensitive for practical use. In broad noon daylight, we were able to achieve a 100% sensitivity, but only a 28.6% precision. The algorithm does an excellent job identifying potential potholes, but still needs work to distinguish potholes from other dark aberrations on the road's surface.

3.3 Pothole Search

The main requirement for the database system was to be able to warn a user if they were within 25 m of a known pothole and are traveling towards it. To test this, we set two potholes on Clark St. using the manual report feature. The pothole locations are shown in Figure 11: We will refer to pothole 1 as the one on Clark and 6th and pothole 2 as the one between Clark and 6th and Clark and Wright.

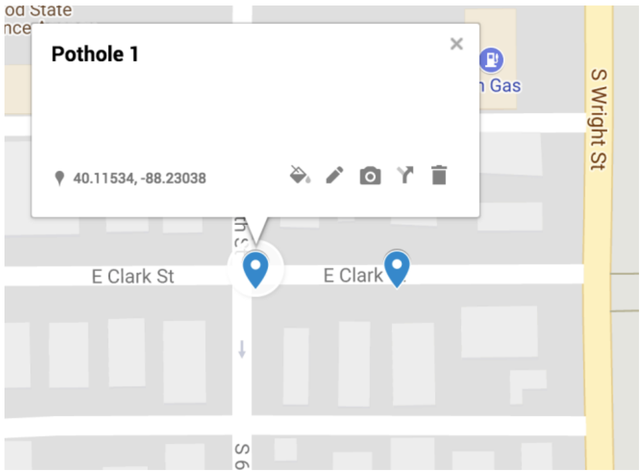


Figure 11

We then walked from Clark and Wright heading west, past Clark and 6th (where pothole 1 is located). As expected, as we neared pothole 2, the haptic buzzer started going off. Once we passed pothole 2, the buzzer stopped for a bit until we got closer to pothole 1, when it started going off again. After we walked past pothole 1, the buzzer stopped once more. We then turned around and repeated the route with the same results. We did an additional test walking south down 6th street towards pothole 1 and another test walking south down Wright St. Both produced the expected results.

3.4 System Level

Our design really depends on the system components all working in harmony to deliver quick feedback to the user. As a result, many of our requirements involved measuring the time it took for system input to transform into system output.

We made a stopwatch module within the software for this purpose, allowing us to measure the time it took to get from one line of code to another line of code and print it. For example, for the requirement of warning the user within 100ms of the computer vision module detecting a pothole, we simulated the pothole detection with a button press: so, we measured the time it took to get from reading that the button press had occurred to the line of code activating our warning system. This time was on ~50 ms on average. The rest of our timing requirements were easily met as well.

4. Costs

4.1 Parts

Part	Distributer	Part Number or Name (if applicable)	Cost
GPS Module	Adafruit	FGPMMOPA6H	$30
Button	Adafruit	B3F-1000	$2.50 for pack of 20
Bluetooth Module	Adafruit	Bluefruit LE SPI Friend	$17.50
Piezo Buzzer	Adafruit	Buzzer 5V - Breadboard friendly	$0.95
Accelerometer	Digikey	MMA7660FCT	$1.81
MCU	Digikey	LPC1114FDH28	$2.58
Computer Vision MCU	Digikey	BeagleBone Black	$56.25
USB Webcam	Amazon		$45.99
Battery	Amazon	Anker PowerCore II Slim	$25.99
3.3V Voltage Regulator	Digikey	LD1117AV33	$0.53
5V Voltage Regulator	Digikey	LP3856ET-5.0/NOPB	$5.04
Barrel Jack Connector x 2	Digikey	PJ-002AH-SMT-TR	$1.44 per unit
Labor[1]	N/A	N/A	$45,000
TOTAL COSTS			$45,189.65

Labor Cost = 3 people $30/hour 20 hours/week 10 weeks 2.5 = $45,000

4.3 Schedule

Week	Andy	Harsh	Jesse
2/5	Work on project proposal	Work on project proposal	Work on project proposal
2/12	Research computer vision Work on design document	Research accelerometer and MCU Work on design document	Research power, GPS, and bluetooth Work on design document
2/19	Finish design document Order camera	Finish design document Order accelerometer, MCU	Finish design document Order GPS, bluetooth module, buzzer, button, battery, voltage regulators, barrel jack connectors, etc
2/26	Complete first python prototype of detection algorithm	Build circuit on breadboard Set up I2C protocol Set up Accelerometer tests	Write SPI and UART protocols Build initial test circuit on breadboard
3/5	Optimize algorithm to improve speed and accuracy	Design PCB and order Finalize Accelerometer pothole detection algorithm Design PCB case	Write firmware to communicate with GPS and store location data
3/12	Optimize algorithm to improve speed and accuracy	Assemble and test PCB Assemble and test PCB Case Begin to integrate GPS module to MCU Start to implement control workflow	Write firmware to communicate with Bluetooth
3/19 (spring break)			
3/26	Copy program onto MCU	Begin testing pothole detection system Integrate Bluetooth with control workflow	Design algorithm to detect if user is approaching stored potholes based upon current location

4/2	Continue to debug and integrate detection unit	Test, debug and optimize system	Write piezo buzzer and pothole report button code Test and debug GPS, bluetooth, and haptic feedback modules
4/9	Test, debug and optimize system	Test, debug and optimize system	Test and debug pothole detection module interface with feedback modules
4/16 (mock demo)	Test to ensure complete functionality	Test to ensure complete functionality	Test to ensure complete functionality
4/23	Start final report Prepare for final demo	Start final report Prepare for final demo	Start final report Prepare for final demo
4/30	Complete final report	Complete final report	Complete final report

5. Conclusion

5.1 Accomplishments

Overall the system works well given a controlled environment and good weather. If the biker hits a pothole, it is highly likely to be recorded accurately. The GPS module and database work as expected, storing pothole locations correctly. The pothole encounter algorithm checks the database to warn the user of any incoming potholes. The Bluetooth module works well as a debugger for now, giving the user location updates and pothole warnings.

5.2 Uncertainties

While the system works well, it does so only in a controlled environment. If the biker enters the road from the curb it is likely that the system will detect that as a pothole. Given some more time it is likely that we would be able to make the accelerometer detection more robust to differentiate potholes from other jerk like movements.

The computer vision detects too many false positives to be fully integrated. 28.6% precision is not a useful accuracy level. Our current method of manual lane detection could be improved by automating the process. In addition, our current system seems to be sensitive to daylight conditions and appears to work better later in the evening when the Sun is lower.

5.3 Ethical considerations

In the course of our work we were mindful to adhere to the IEEE code of ethics. We strove to "be honest and realistic in stating claims or estimates based on available data" and to "seek, accept, and offer honest criticism of technical work, acknowledge and correct errors, credit properly the contributions of others," in accordance to point 3 and 7[4]. There are some ethical questions involved with privacy as well since we are continuously recording video. However, none of the data is stored and is immediately overwritten after it is processed.

5.4 Future work

The next step for our project is to make the computer vision aspect more consistent and to fully flesh out the database system.

The computer vision needs more work to improve its accuracy. Automatic lane detection would be a major improvement. More time is needed to find out how to tweak the algorithm to make it more specific to potholes. Most likely more tests will need to be added to the cascade detector. Using the same inputs as the paper we followed, we were able to achieve similar results. The foundation is there, but adjustments still need to be made.

In the fully realized system, we would like to have a cloud database where users can contribute to a crowdsourced map of pothole locations. We also do not currently have a way to automatically remove potholes that have been filled in from the database.

Since our project is meant to be used outside, we will need to make it waterproof to protect it from rain and other water hazards. Finding a way to shrink the design will also make it more user friendly.

6 References

[1] R. Annis, "$6.5 Million Settlement Given to LA Cyclist for Injuries from Pothole", *Bicycling*, 2017. [Online]. Available: https://www.bicycling.com/news/when-cyclists-sue-the-city. [Accessed: 08- Feb- 2018].

[2] "Report A Problem - City of Champaign", *City of Champaign*. [Online]. Available: http://champaignil.gov/public-works/report-a-problem/. [Accessed: 08- Feb- 2018].

[3] Y. Jo and S. Ryu, "Pothole Detection System Using a Black-box Camera", *Sensors*, vol. 15, no. 12, pp. 29316-29331, 2015.

[4] "IEEE Code of Ethics", *IEEE*. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html. [Accessed: 20- Feb- 2018].

[5] R. Keim, "UART Baud Rate: How Accurate Does It Need to Be?", Allaboutcircuits.com, 2017. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/the-uart-baud-rate-clock-how-accurate-does-it-need-to-be/. [Accessed: 25- Mar- 2018].

[6] NXP Semiconductors "LPC111x/LPC11Cxx User manual," LPC111x/LPC11Cxx datasheet, July. 2010 [Revised Dec. 2016].

Appendix A: Requirements and Verification Table

Requirements	Verification
Power Bank	
> 5000 mAh of capacity	1. Connect fully-charged battery (as indicated by LED on commercial power bank) 2. Discharge battery at 500 mA +/- 5% for 10 hours using constant current load circuit 3. Check if commercial power bank will still allow current to flow using ammeter, as they will automatically shut current off when voltage is too low.
Voltage Regulator	
Provide 3.3V +/- 5% output from 4.7V - 5.3 V source at 0.5A +/- 5%	1. Connect the input to a power supply 2. Draw 0.5A from power supply using constant current load circuit 3. Sweep the power supply from 4.7 to 5.3 V 4. Measure the output is 3.3V +/- 5% at 0.5A +/- 5% with a multimeter
Provide 5V +/- 5% output from 5V - 5.3 V source at 2.5A +/- 5%	1. Connect the input to a power supply 2. Draw 2.5A from power supply using constant current load circuit 3. Sweep the power supply from 5 to 5.3 V 4. Measure the output is 5V +/- 5% at 2.5A +/- 5% with a multimeter

MCU	
Must be able to communicate over I2C in at least standard mode (100 kHz)	1. Set-up I2C on MCU. 2. Set up oscilloscope to trigger mode 3. Connect positive probe to SCL pin 4. Connect negative probe to GND 5. Send any data via the I2C bus 6. When oscilloscope is triggered find the frequency of the clock waveform by measuring any 2 rising edges.
Must be to communicate over UART at 9600 baud	1. Connect MCU to a computer terminal using UART and PuTTY 2. Set PuTTY baud rate to 9600 and make connection 3. Send a test string from MCU to computer upon button press 4. Ensure that string on terminal matches test string sent

MCU + Button	
Button and debouncing software should register press within 500 ms	1. Connect button to MCU GPIO pin 2. Set MCU to output to computer terminal via UART if press is detected 3. Start timer and press button simultaneously 4. Wait for output to appear on terminal 5. Once output appears, stop timer 6. Check if time allotted is less than 500 ms
Button should register press accurately 19 out of 20 times	1. Connect button to MCU GPIO pin 2. Set MCU to output to computer terminal via UART if press is detected 3. Press button 20 times 4. Check if at least 19 outputs are displayed on terminal
MCU + Bluetooth Module	
Must have range greater than 2m	1. Place the receiver 2m away from the module 2. Ensure that phone can see module while it is advertising using Bluetooth scanner app
Must transmit latest GPS location after pothole detection within 1s	1. Connect phone to bluetooth module 2. Open Serial Bluetooth Terminal app 3. Start timer and simulate pothole detection using user report button 4. Stop timer once coordinate appears on phone 5. Ensure time allotted is less than 1s
MCU + GPS	
Location accuracy of 5m (best case, open sky scenario)	1. Determine the precise longitude and latitude of a location using Google Maps (such as a street lamp). 2. Take the GPS module there. 3. Compare that to the output of the GPS module.
Warn user if they are within 25 m of a known pothole and are traveling towards it	1. Manually set pothole location using user report button 2. Mark location 25 m away from pothole 3. Walk 30 m away, then start walking towards pothole 4. See if haptic feedback sensor goes off
MCU + Accelerometer	
Must have sampling rate of 100 Hz	1. Connect accelerometer to MCU 2. Create timer interrupt at a 100 Hz on the MCU

	3. Set AMSR[2:0] of accelerometer to 111 (120 samples/sec) 4. Set accelerometer to Active Mode 5. Read either X,Y or Z register at every MCU interrupt while moving the accelerometer 6. Output data to any platform and check for consecutive duplicate values
±1g range with .1g sensitivity	1. Measure the weight of the accelerometer 2. Place it on one end of a balance and put known weights on the other side 3. Record the output of the accelerometer and compare with the expected computed values.
MCU must detect potholes with the accelerometer. True Positive rate of 80%	1. Find 3 potholes 2. Run bike through pothole track at least 5 times 3. Keep track of each detected event as well as each true positive event 4. Divide true positives by detected events to get rate
MCU + Piezo Buzzer	
Must be able to be detected on bare skin	1. Connect buzzer to GPIO pin on MCU 2. Place the piezo buzzer on wrist 3. Set GPIO pin high 4. Observe if buzzer can be felt
Must warn user within 100 ms of pothole detection from computer vision module	1. Connect control module to computer via PuTTY and UART-USB connector 2. Place function to output system clock count when the GPIO pin connected to the computer vision module goes HIGH 3. Place function to output system clock count when code to activate buzzer is entered. 4. Attach SPDT switch and hardware debouncer to GPIO pin normally connected to computer vision module. Attach GND to one input and 3.3V to the other. 5. Set switch to GND. 6. Run control module normally. 7. Simulate computer vision pothole detection by flipping switch to 3.3V and immediately flipping back to GND. 8. Look at output values on computer terminal. Calculate time elapsed based upon MCU clock speed and compare to 100ms.
Detection Microcontroller	
The camera captures at least 10 frames per second	1. Connect the camera over USB to a computer and capture 5 seconds of video. 2. Use a python program to count the number of frames and divide by the video length

The module should have a sensitivity of 70% and a precision of 70%	1. Bike the system past 5 potholes 2. Record the outputs of the camera module 3. Repeat 3 times for each pothole

Appendix B: UART Development

Writing the UART driver involved setting a couple of clock dividers to get as close as possible to our desired baudrate. Since UART has no clock line, it is just expected that each device is operating at an agreed upon baudrate. If each baudrate is within 3.5% of each other, the UART should work as expected [5]. The process for calculating the clock divider values DLH, DLL, MULVAL, and DIVADDVAL is shown in Figure 12.

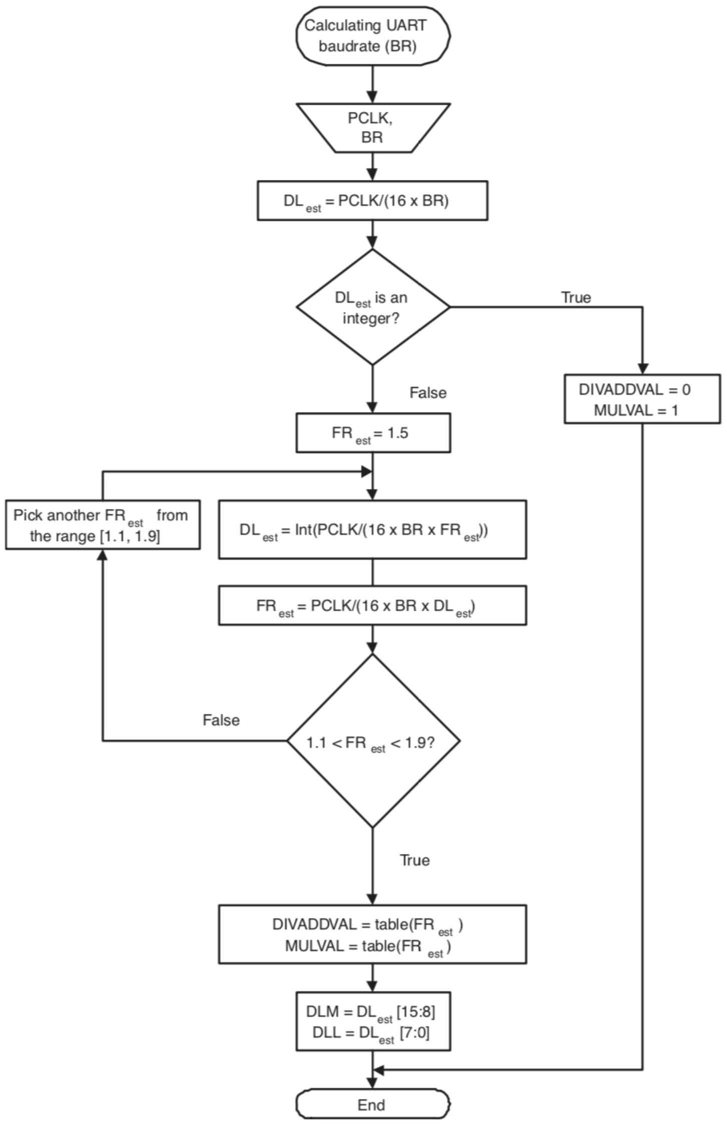


Figure 12

The table mentioned in the above flowchart is shown here:

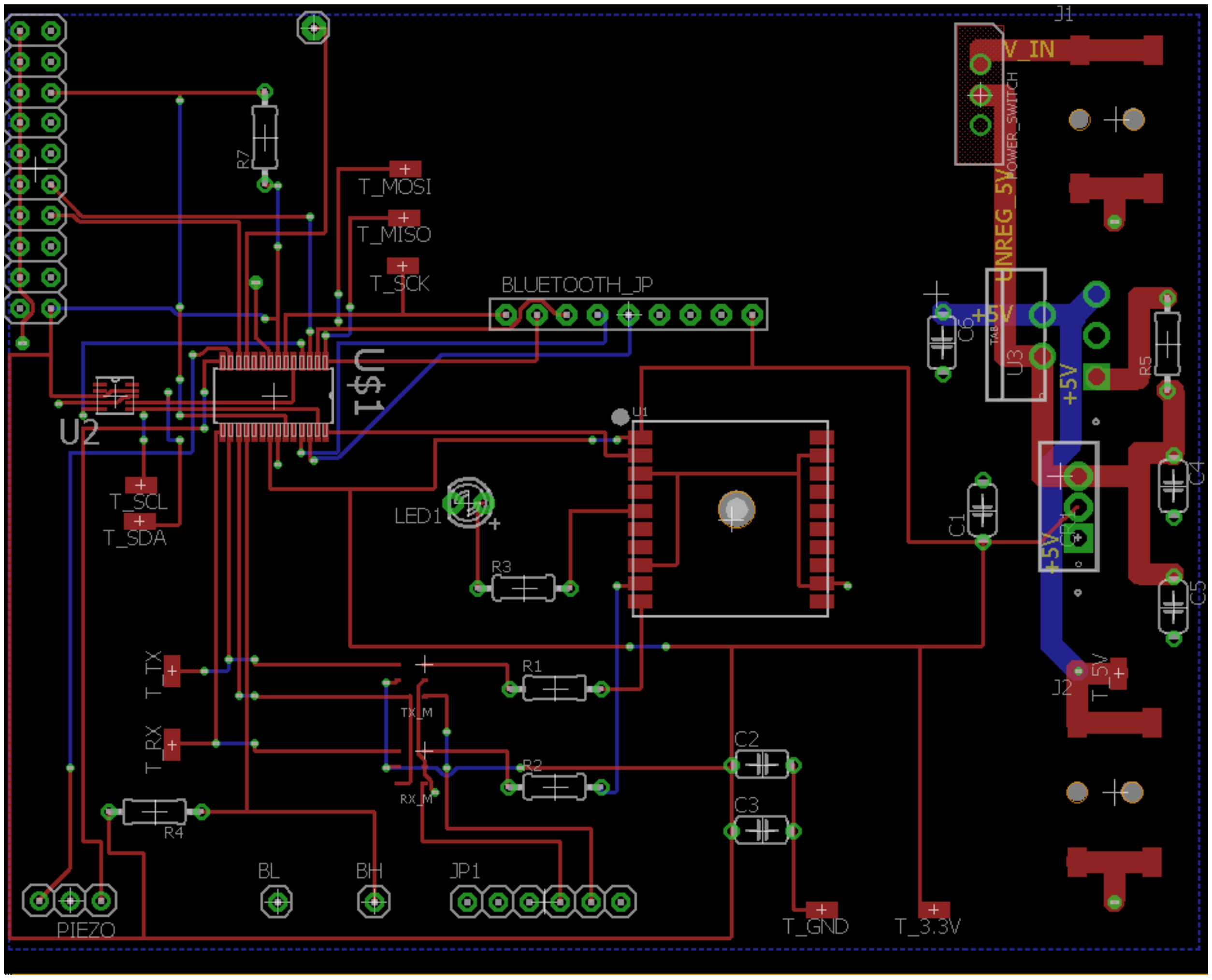
FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal
1.000	0/1	1.250	1/4	1.500	1/2	1.750	3/4
1.067	1/15	1.267	4/15	1.533	8/15	1.769	10/13
1.071	1/14	1.273	3/11	1.538	7/13	1.778	7/9
1.077	1/13	1.286	2/7	1.545	6/11	1.786	11/14
1.083	1/12	1.300	3/10	1.556	5/9	1.800	4/5
1.091	1/11	1.308	4/13	1.571	4/7	1.818	9/11
1.100	1/10	1.333	1/3	1.583	7/12	1.833	5/6
1.111	1/9	1.357	5/14	1.600	3/5	1.846	11/13
1.125	1/8	1.364	4/11	1.615	8/13	1.857	6/7
1.133	2/15	1.375	3/8	1.625	5/8	1.867	13/15
1.143	1/7	1.385	5/13	1.636	7/11	1.875	7/8
1.154	2/13	1.400	2/5	1.643	9/14	1.889	8/9
1.167	1/6	1.417	5/12	1.667	2/3	1.900	9/10
1.182	2/11	1.429	3/7	1.692	9/13	1.909	10/11
1.200	1/5	1.444	4/9	1.700	7/10	1.917	11/12
1.214	3/14	1.455	5/11	1.714	5/7	1.923	12/13
1.222	2/9	1.462	6/13	1.727	8/11	1.929	13/14
1.231	3/13	1.467	7/15	1.733	11/15	1.933	14/15

These values come together to determine the actual baudrate using the following equation:

$$UART_{baudrate} = \frac{PCLK}{16 \times (256 \times U0DLM + U0DLL) \times \left(1 + \frac{DivAddVal}{MulVal}\right)}$$

For our final baudrate of 115200, we used DLM = 0, DLL = 4, DIVADDVAL = 5, and MULVAL = 8. This led to an actual baudrate of 115384, which is 0.16% from the specified baudrate of 115200. This meets our requirements.

Appendix C PCB Diagram


J1
V_IN
POWER_SWITCH
UNREG_5V
R7
T_MOSI
T_MISO
T_SCK
BLUETOOTH_JP
U$1
U2
T_SCL
T_SDA
+5V
C6
U3
R5
U1
LED1
R3
C1
C4
C5
T_TX
T_RX
R1
R2
TX_M
RX_M
C2
C3
R4
J2
T_5V
PIEZO
BL
BH
JP1
T_GND
T_3.3V