DESIGN OF INEXPENSIVE AND EASY TO USE DIY INTERNET OF THINGS

PLATFORM

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science in Electrical Engineering

by

Samuel Jaffe

June 2016
COMMITTEE MEMBERSHIP

TITLE: Design of Inexpensive and Easy to Use DIY Internet of Things Platform

AUTHOR: Samuel Jaffe

DATE SUBMITTED: June 2016

COMMITTEE CHAIR: Vladimir Prodanov, Ph.D.
Associate Professor of Electrical Engineering

COMMITTEE MEMBER: Bridget Benson, Ph.D.
Assistant Professor of Electrical Engineering

COMMITTEE MEMBER: Jeffrey Gerfen, M.S.
Lecturer of Electrical Engineering
ABSTRACT

Design of Inexpensive and Easy to Use DIY Internet of Things Platform

Samuel Jaffe

This thesis focuses on the design and implementation of a new, inexpensive, and less complex system for a Do-It-Yourself (DIY) Internet of Things (IoT) platform. The hardware aspects focus on a new chip called the ESP8266 which contains both microcontroller and WiFi connectivity capabilities in an extremely affordable package. The system uses the Arduino IDE to program the ESP8266, which is known to be an extremely user-friendly environment. All other software is both free and easy to use. Past methods of creating IoT projects involved either expensive hardware, often ranging from $50-$100 per node, or complicated programming requiring a full computer, or a constant connection to an immobile power source. This method costs as little as $2.50, can last for months or even years off of batteries, can be smaller than a quarter, and only requires a few lines of code to get data moving, making this platform much more attractive for ubiquitous use.
ACKNOWLEDGMENTS

I would like to express my deepest gratitude for the support of my friends, family, and the faculty of the electrical engineering department. From my first EE project I built for my friends in high school, to my first experience with microcontrollers guided by my brother Ben, I’ve had a burning curiosity and excitement for electrical engineering. Without a doubt, I would much less likely be on the path I’m on today if not for these influences.

Additional acknowledgements go to Professors Prodanov, Benson, Gerfen, Taufik, Smilkstein, and Braun. Their passion for teaching and their guidance has enabled me to not only succeed here at Cal Poly, but also have a great time while doing it!

Lastly I’d like to thank my parents for their continued support over the last 22 years. I owe everything to them and I hope to make them proud.
# TABLE OF CONTENTS

| LIST OF TABLES | xi |
| LIST OF FIGURES | xii |

## CHAPTER

1: Introduction .................................................................................................................. 1
   1.1: What is “Internet of Things”? ............................................................................. 1

2: Background and Historical Perspective ........................................................................ 4
   2.1: Background .......................................................................................................... 4
       2.1.1: Implications of IoT ....................................................................................... 4
       2.1.2: Impact Analysis ........................................................................................... 5
           2.1.2.1: Economic Impact .................................................................................. 5
           2.1.2.2: Environmental Impact ........................................................................... 5
           2.1.2.3: Sustainability ......................................................................................... 6
           2.1.2.4: Ethical Impact ......................................................................................... 7
           2.1.2.5: Health and Safety ................................................................................. 8
   2.2: Prior Methods ...................................................................................................... 8
       2.2.1: Full-Fledged Computer .................................................................................. 9
           2.2.1.1: System Overview .................................................................................... 9
           2.2.1.2: Hardware ............................................................................................... 9
           2.2.1.3: Software ............................................................................................... 10
           2.2.1.4: Example: Schwippy Tree ..................................................................... 11
           2.2.1.5: Analysis ................................................................................................ 15
2.2.2: Microcontroller with WiFi Adapter ................................................................. 16
  2.2.2.1: System Overview ......................................................................................... 16
  2.2.2.2: Hardware ................................................................................................. 17
  2.2.2.3: Software ..................................................................................................... 19
  2.2.2.4: Example: Adafruit: Automatic Twitch ON AIR Sign ............................... 20
  2.2.2.5: Analysis ..................................................................................................... 24

2.2.3: Microcomputer – e.g. Raspberry Pi ............................................................... 25
  2.2.3.1: System Overview ......................................................................................... 26
  2.2.3.2: Hardware ..................................................................................................... 26
  2.2.3.3: Software ..................................................................................................... 28
  2.2.3.4: Example: A_Comcast_User ......................................................................... 30
  2.2.3.5: Analysis ..................................................................................................... 32

2.2.4: Comparisons .................................................................................................... 33

3: System Design ....................................................................................................... 34
  3.1: System Overview ............................................................................................... 34
    3.1.1: ESP8266: The Hardware that Makes This All Possible ............................. 34
    3.1.2: The Free Software Used in This Method .................................................. 35
    3.1.3: System Overview ......................................................................................... 36
    3.1.4: Examples ...................................................................................................... 37

3.2: Low-Level Design ............................................................................................... 38
  3.2.1: Hardware: ESP8266 Board Layout Options .............................................. 38
    3.2.1.1: Easy To Use .............................................................................................. 38
    3.2.1.2: Discrete Form Factor ............................................................................... 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1.3</td>
<td>Low-Power Mode</td>
<td>40</td>
</tr>
<tr>
<td>3.2.1.4</td>
<td>Comparisons</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Software: Internet Connectivity</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2.1</td>
<td>Arduino IDE: Hardware Programming</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2.2</td>
<td>Firebase: Data Storage</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2.3</td>
<td>Web App for Human-Firebase Interaction</td>
<td>45</td>
</tr>
<tr>
<td>3.2.2.4</td>
<td>Dropbox: Web Hosting of Web App</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Analysis</td>
<td>47</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Comparison of Previous Methods</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>System Testing</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Latency and Reliability Tests</td>
<td>50</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Keyboard Press with Single ESP8266</td>
<td>50</td>
</tr>
<tr>
<td>4.1.1.1</td>
<td>System Setup</td>
<td>50</td>
</tr>
<tr>
<td>4.1.1.2</td>
<td>Software</td>
<td>50</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Keyboard Delay Test</td>
<td>54</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Keyboard Press with Two ESP8266 Modules</td>
<td>57</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Conclusion</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Example: RC Car</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1</td>
<td>System Setup</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1.1</td>
<td>Software</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1.2</td>
<td>Hardware</td>
<td>63</td>
</tr>
<tr>
<td>4.2.1.3</td>
<td>Battery</td>
<td>65</td>
</tr>
<tr>
<td>4.2.1.4</td>
<td>Speed Controller</td>
<td>68</td>
</tr>
</tbody>
</table>
B2.1: Firebase Setup .......................................................................................................................... 93
B2.2: Dropbox Setup ............................................................................................................................. 97
B2.3: Web App for Firebase Interface .................................................................................................. 99
B2.4: Arduino IDE for Hardware Programming .................................................................................... 101

Appendix C: Code .................................................................................................................................. 103

C1: Web App Javascript .......................................................................................................................... 103
C2: Web App HTML .................................................................................................................................. 111
C3: Arduino code for RC car ..................................................................................................................... 113
C4: Arduino code for Garage Door Sensor ................................................................................................. 117
C5: Latency Test Codes ............................................................................................................................. 118

C5.1: Keyboard press with single ESP ....................................................................................................... 118

C5.1.1: Virtual Keyboard Code .................................................................................................................. 118
C5.1.2: ESP Code .......................................................................................................................................... 120
C5.1.3: Arduino Timer Code ....................................................................................................................... 123

C5.2: Keyboard delay test ............................................................................................................................ 124

C5.2.1: Virtual Keyboard Code ................................................................................................................... 124
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Raspberry Pi Comparison Chart (a few essential data points)</td>
<td>26</td>
</tr>
<tr>
<td>2: Comparison Between IoT Platforms</td>
<td>33</td>
</tr>
<tr>
<td>3: ESP8266 Board Comparisons</td>
<td>42</td>
</tr>
<tr>
<td>4: Comparison Between IoT Platforms</td>
<td>49</td>
</tr>
<tr>
<td>5: ESP8266 Garage Door Sensor Power Consumption</td>
<td>75</td>
</tr>
<tr>
<td>6: Cost for Garage Sensor</td>
<td>77</td>
</tr>
<tr>
<td>7: Bill of Materials for RC Car</td>
<td>85</td>
</tr>
<tr>
<td>8: Bill of Materials for Garage Door Sensor</td>
<td>86</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: High-Level Diagram of Proposed System</td>
<td>2</td>
</tr>
<tr>
<td>2: Schwippy Tree V1 Just After Christmas</td>
<td>11</td>
</tr>
<tr>
<td>3: Schwippy Tree 2014</td>
<td>12</td>
</tr>
<tr>
<td>4: Schwippy Tree Hardware System Diagram</td>
<td>13</td>
</tr>
<tr>
<td>5: Three X10 Lamp Modules Plugged into a Power Strip</td>
<td>13</td>
</tr>
<tr>
<td>6: Microcontroller with WiFi Shield: High Level Block Diagram</td>
<td>16</td>
</tr>
<tr>
<td>7: Microcontrollers (left to right): MSP, Propeller, BASIC Stamp, Arduino</td>
<td>17</td>
</tr>
<tr>
<td>8: Sparkfun's WiFly GSP (left) and WiFi CC3000 (right)</td>
<td>18</td>
</tr>
<tr>
<td>9: Adafruit’s Shield 101 (left), Feather (middle), ATWiNC1500 (right)</td>
<td>18</td>
</tr>
<tr>
<td>10: Adafruit’s Automatic Twitch ON AIR Sign</td>
<td>21</td>
</tr>
<tr>
<td>11: Microcontroller and WiFi Module Connections</td>
<td>22</td>
</tr>
<tr>
<td>12: System Block Diagram</td>
<td>23</td>
</tr>
<tr>
<td>13: Active Adafruit Twitch Stream</td>
<td>23</td>
</tr>
<tr>
<td>14: Inactive Adafruit Twitch Stream</td>
<td>23</td>
</tr>
<tr>
<td>15: Raspberry Pi 2 B</td>
<td>27</td>
</tr>
<tr>
<td>16: Pi Competitors: Edison, APC 8750, Cubie Board, OLinuxino, C.H.I.P.</td>
<td>27</td>
</tr>
<tr>
<td>17: AComcast User Block Diagram</td>
<td>31</td>
</tr>
<tr>
<td>18: Twitter Stream</td>
<td>31</td>
</tr>
<tr>
<td>19: System Block Diagram</td>
<td>34</td>
</tr>
<tr>
<td>20: ESP Models: ESP-09 (left), ESP-01 (middle), ESP-12E (right)</td>
<td>35</td>
</tr>
<tr>
<td>21: High-Level Internet Communication Between Devices</td>
<td>37</td>
</tr>
</tbody>
</table>
45: Motor Speed Controller .................................................................................. 68
46: Garage Door Sensor Overview ......................................................................... 70
47: Fully Assembled System .................................................................................. 71
48: Tilt Switch Used in Example ............................................................................ 72
49: Mercury Tilt Switch with Glass Casing ............................................................ 72
50: System on Open Garage Door .......................................................................... 75
51: System on Closing /Opening Garage Door ....................................................... 76
52: System on Closed Garage Door ......................................................................... 76
53: Battery Life vs. Check Frequency for 850mAh Battery ..................................... 77
54: ESP Models with USB Programming: 12E, Lua, WeMosD1, ESPduino ............... 87
55: ESP-01 Model .................................................................................................. 88
56: Parallel to Breadboard Adapter (bottom) and M-F Wires (top) ......................... 88
57: 3.3V FTDI Adapter for Programming Discrete ESP8266 ..................................... 89
58: ESP-01 to FTDI Connections ............................................................................ 90
59: ESP-01 with Sleep Connection ......................................................................... 91
60: Sleep/Wake Wires for ESP8266 12-E (left) and Sparkfun “Thing” (right) .......... 91
61: Sign In to Firebase ............................................................................................ 94
62: Signing in to Firebase with Google ................................................................... 94
63: Click "Manage App” on Firebase ...................................................................... 95
64: Firebase, Apps Page, First Time ...................................................................... 95
65: Dropbox Log In .................................................................................................. 97
66: Dropbox Icon (on Mac) and Dropbox Login ..................................................... 98
67: Dropbox Drop-Down ......................................................................................... 98
Chapter 1: Introduction

1.1: What is “Internet of Things”?

“Internet of Things,” or IoT, describes a movement toward an increase in smart, web-connected devices in every-day life. Adding technological capabilities to household items holds the potential to greatly improve their performance and with recent innovations making technology smaller, cheaper, and more power efficient, that idea stands to become a reality. The fully realized goal imagines a future where everything that serves a function in day-to-day life has communication with the web. They can be both controlled over the web and they can send data to the web. This gives people control over their devices and gives them status updates on the state of their devices from anywhere in the world.

IoT needs to meet a few baselines before it can become ubiquitous. Besides the general ability to talk with the Internet and things, the platform must be physically small enough to place on objects without disturbing their originally intended function. The platform must be low power to both save energy and run off of batteries for mobility. It must be low-cost to be cost-effective when deciding whether or not to make an object smart. And it must be easy to use. Markets, especially technological markets, flourish when everyone has the ability to build a platform themselves without professional knowledge. By meeting these conditions, an IoT platform can become global.

The platform described in this thesis meets every one of these specifications. This platform utilizes the new ESP8266, a microcontroller/WiFi System on a Chip. The smallest ESP8266 board measures 1cm x 1cm which is smaller than any other board used
for IoT. The ESP8266 power consumption falls to 0.257mW in sleep mode, smaller than mostly every other method. The cheapest ESP8266 boards cost as little as $2.50, lower than any other hardware. These specifications make the ESP8266 the perfect hardware platform for IoT.

The software utilized in this system, as portrayed in Figure 1, in addition to its high-performance functionality, is all free and easy to use. This method uses the Arduino IDE to program the hardware, Firebase to store/send/receive data online, and a web app, hosted on Dropbox, to make human interaction more user friendly with the data. This all offers user-friendly setup and, once set up, fast prototyping due to its lack of complexity.

![Figure 1: High-Level Diagram of Proposed System](image)

This chip coupled with free software enables an engineer to connect any *thing* to the Internet for $2.50 with minimal software complexity. At a $2.50 per node price-
point, mass connectivity becomes much more attainable. This IoT platform takes a huge step forward in realizing the full potential of global *thing* connectivity.

This thesis provides insight as to how IoT has developed, where it is headed, and proposes a new platform. Chapter 2 reviews the importance and potential of IoT as well as prior methods for creating a hardware/software platform. Chapter 3 describes the system design using ESP8266, Arduino IDE, Firebase, the Web App, and Dropbox. Chapter 4 investigates performance including functionality, latency, power consumption, and sample projects. Chapter 5 concludes the thesis and suggests future work to be done. Included in the appendix is a How-To guide on getting the system up and running at home and all code used in testing and examples.
Chapter 2: Background and Historical Perspective

2.1: Background

2.1.1: Implications of IoT

Internet of Things has the potential to revolutionize how we live our lives. A world where every object is web-enabled presents interesting alterations to day-to-day living. It goes without saying that the market for IoT is immense.

A large market for IoT lies in domestic applications. This could be as big as a thermostat. If you leave your house but forget to turn the air conditioner off, you can do so from your phone. Or you could program it to read the GPS location of your phone to determine that you’re out of the house and to automatically shut off the air conditioner. This could be as small as shoes - shoes that detect their own odor and automatically order either odor-eaters or new shoes depending on the state of the shoes. This could be as important as sensing if an elderly person has fallen and can’t get up. It can be set up to automatically call a family member’s number or an ambulance. Domestic IoT has the potential to save energy, time, and even lives.

Applications are not just domestic. IoT holds the potential to give our cities the ability to think. Traffic can be better controlled. If every car knows the position, destination, and desired route of every other car, computers can reroute to prevent gridlock. The electrical grid can prevent blackouts. With the ability to modify on-times of large electronic appliances and industrial plants, blackouts can be a thing of the past. The potential is limitless - forest fire detection, industrial structural health, water quality, waste management, item location, intelligent shopping, smart lighting and more. Anything that needs to be monitored or sensed has a spot in IoT and with all this
potential, some estimate that we’ll have anywhere from 26 billion to 100 billion connected devices by the year 2020 [1]. IoT has the ability to revolutionize how we live our lives as did smartphones or the Internet itself.

2.1.2: Impact Analysis

2.1.2.1: Economic Impact

IoT, as a broader system, is all about collecting valuable data and using that data to improve some part of life. Many jobs to do with surveying or monitoring stand to be assisted or even replaced by IoT. If a $15/hour worker is replaced with a hundred dollar sensor and $3 of connectivity hardware, that employer can make their money back in less than a day. With a conservative lifespan of 5 years the employer could have around 6000x more jobs being done with IoT over humans at the same price. By some estimates, a workforce filled with robots (IoT, sensors…) may “replac[e] the work of approximately 100 million workers” by 2025 [2]. These statistics and estimations stand to save billions.

Individual households can save money at home with smart AC. If the AC turns itself off when nobody’s in the house, that household will save on their electric bill. The same goes for lights and other household energy consumers.

Smart cities can prevent traffic. Americans spent 8 billion hours stuck in traffic in 2015 [3]. At a rough average of $25/hr that’s $200 billion lost every year to something that can be reduced or prevented by IoT.

2.1.2.2: Environmental Impact

Domestic environmental impacts include reduced energy consumption via smart thermostat, automatic lights, and more. Food items can have their own individual devices
to tell the owner when it’s going to expire or what one can cook with the ingredients in
the kitchen. This will increase awareness of the status of one’s food and reduce food
waste.

Negative environmental impact includes the energy in production and operation,
both to power the device and to power the data centers which handle the information
being stored and transmitted. Another negative environmental impact is the material
production and disposal cost. There was 53 million metric tons of e-waste worldwide
just in 2013 before any widespread IoT [4]. This number stands to accelerate with the
rise of IoT.

2.1.2.3: Sustainability

At first glance adding billions of electronic devices to our day to day lives seems
unsustainable. We as a society are verging on a serious e-waste problem. If done
without thought, IoT could contribute greatly to this problem. If done with forethought,
however, we can have IoT in a sustainable way. The system described in this thesis is
highly versatile and future systems have the potential to be just as, if not more versatile.
Each device has generic digital inputs and a reprogrammable microcontroller. This
means that each node can be programmed to talk to literally any sensor. Say a device is
being used to keep an eye on the expiration date of a banana. Once that banana is eaten,
the device can be removed and recycled or placed on another object. Either it can be
returned to the retailer where it came from to be repurposed or reprogrammed by the
consumer to connect any other thing to the Internet. With the proper infrastructure and
planning IoT can benefit sustainability more than it hurts.
A later section in this thesis discusses low-power methods to conserve power. The proposed platform performs at significantly lower power consumption than nearly all other methods. The number of IoT nodes is growing and each must be powered. The lower this power consumption is, the less of an energy impact IoT will have on the world.

The aforementioned low-power section discusses these methods with the main aim of getting a system to run on batteries for an extended period of time. At the present time more than 350 million rechargeable batteries and nearly 3 billion dry-cell single-use batteries are purchased every year in the USA alone[5]. With IoT this number will only go up. But with low-power IoT this additional number can be greatly reduced by responsibly using rechargeable batteries to maintain a long lifespan or by simply using dry-cells more slowly or by using energy harvesting to eliminate the need for significant batteries completely.

2.1.2.4: Ethical Impact

One of the largest potentials of IoT lies in the ability to cheaply and easily replace jobs. Taking away someone’s job is, of course, an ethical issue. People need jobs to make a living and taking that away takes away their ability to live. But, without getting too political, this issue cannot stand in the way of progress. Without automation we’d be decades behind in technology. As someone on the Internet said, “Pretty soon we’ll learn that robots are meant to work. We’re meant to live.”

IoT also has the potential to save lives. Not often is there an argument that the extension of life constitutes a negative ethical impact.
2.1.2.5: Health and Safety

IoT has the potential to save lives. It could sense when an elder has fallen and can’t get up. Or monitor an irregular heart to give the wearer advanced warning of problems. Or monitor glucose in a person’s blood stream. Or basically act as a star trek tricorder but without having to take out a device to scan someone. Just 24-hour health data ready to warn you of anything that might be wrong.

IoT also has the ability to monitor industrial sites. This would give us safer, cleaner, more reliable water as well as waste management. Not to mention food creation. In grad seminar in Winter quarter 2016 a man in viticulture spoke about how he wanted to use IoT devices to monitor water levels in his crops. This can be extended to all farming leading to more reliable food production.

Or picture a pilot in a small plane. IoT could monitor the medical status of the pilot and relay that information back to ground control. And if for some reason the pilot is unable to fly the plane or communicate with ground control over radio, ground control would be alerted and the plane, using high-tech technology, could be set to autopilot to land itself.

2.2: Prior Methods

The following section delves into the options of prior methods of IoT. An IoT system consists of a computing object of some sort (computer, microcontroller, microcomputer, ESP8266) which connects the things to the Internet. These things can be sensors, GPS, switches, and anything else that gathers data from the world around or about the system. All comparisons in this and future sections consider only the computing device unless otherwise stated. Comparisons of cost and size do not include
the hardware to power the device, sensors connected, or the wires and other hardware necessary to complete the system. This is because most of these peripheral costs are the same from platform to platform. And power hardware introduces variability. The power supply in a full-fledged computer usually comes with the cost of the computer but the cost of the power hardware for a microcontroller probably does not, though if the microcontroller is assisting a device that already has power, it can tap into that for free. Complications like this mean these comparisons only consider the hardware cost of the computing device.

2.2.1: Full-Fledged Computer

2.2.1.1: System Overview

2.2.1.2: Hardware

The hardware for this method consists mainly of a computer, like a desktop PC running Windows or Mac or Linux or some other operating system. Back before microcontrollers or other smaller self-contained electronic processors, computers were the only way to connect to the Internet so if someone wanted to make a thing connect to the Internet, they’d have to connect that thing to a computer.

The main drawbacks to this method arise with cost, size, and power. The first mac in 1984 cost $2,500 [6] which is around $5,800 in 2016 dollars [7]. In the early 2000s, when Arduino started up [8] a computer cost around $800 [9] or about $1000 in 2016 dollars [7]. In 2016 dollars each node, that is each thing you wanted to connect to the Internet, would cost between $1000 and $5800 making investment payoff unreasonable. Size was also an issue. A desktop computer is a wholly impractical object
to carry around for any extended period of time. Thirdly power is a consideration. The computer would have to be plugged into the wall to remain working. This prevents mobility, if the sheer size wasn’t enough. Not to mention the power consumption. It is hard to find something where the payback of connecting it to the Internet is worth not only the upfront cost but the cost of powering the system when it’s running.

2.2.1.3: Software

Software at the time was not at all user friendly. First one would have to get the computer talking to some hardware on the outside – the thing. Then one would need to know how to send that information over the Internet to some place where it could be retrieved and used again. In what was the stone age of the Internet, applications were extremely limited.

In modern day this method is better and worse. It’s worse in that essentially the only way to talk to a thing is over USB. Modern computers have done away with anything but consumer data ports so interfacing with things has gotten harder. Connecting to the Internet, however, has gotten much easier. Software such as node.js allows javascript, an extremely web-friendly programming language, to control aspects of a person’s computer and interact with the Internet. As for data storage, there are many free and user-friendly options such as dropbox, google drive, or in the case of the system described later, Firebase. Any of these free cloud-based operations can store data and deliver it to wherever it needs to go. This particular subset of software has boomed over the last few decades so there are many more ways to send data over the Internet. Another way is shown in the following example.
2.2.1.4: Example: Schwippy Tree

One of the most well-known IoT projects on the Internet is the Schwippy Tree, an Internet-controlled Christmas Tree [10]. The Schwippy Tree started in 2007 and has been an annual appearance on tech blogs and social media every year in the month of December leading up to Christmas. Site visitors see a live video broadcast of the tree and various inputs to control the tree. The first iteration in 2007 had 4 pairs of buttons: Green (on and off), White (on and off), Red (on and off), and Blue (on and off) to turn on or off those respectively colored lights as shown in Figure 2.

![Control the Schwippy Christmas Tree](image)

Visit us in the Schwippy Forums at [http://forums.schwippy.com](http://forums.schwippy.com)

As the years progressed, Schwippy included a star on top of the tree, a train around the tree (only one year due to the noise), scrolling text, an arrangement of lights to make
faces, window lights, light-up gift boxes, and choreographed light-shows with songs, shown in Figure 3.

Figure 3: Schwippy Tree 2014

The Schwippy tree has progressed with the technology. Unfortunately Schwippy has not yet written up how his system works today but his first iteration in 2007 was a great example of an old IoT system.

Originally the system used simple remote-controlled relays to turn on or off the lights as shown in Figure 4.
The system used expensive X10 lamp modules as the relays [11] shown in Figure 5. X10 is a company specializing in home automation. X10 lamp modules are remote controlled switches for devices that plug into the wall.

Schwippy used these modules to turn on or off each strand of lights. The computer controlled the X10 modules wirelessly through the X10 computer interface module. The
computer received commands over the Internet from the buttons on the site, then communicate a command to the X10 computer interface module which told a particular X10 lamp module to turn on or off. Each lamp module cost $12. The computer interface module cost $15. The whole system, disregarding the tree and lights, cost $63 plus the cost of the computer for four one-direction, binary controls.

The software for the original iteration is a bit complex. He states, “The main html file that hosts the tree's page (Located at tree.schwippy.com) is hosted on an off-site server. If you are interested in the client-side scripting, just view the source of that file. On that page are the buttons that control the lights. Instead of the offsite server controlling the lights, it instead contacts a computer I have set up at home. The PHP file that is called on that computer is located [11].

The PHP file for a layman, even for an electrical engineer, is a bit daunting. From his explanation it seems that when his computer receives a command, it executes a command on the shell of the computer telling the X10 computer interface module which light to turn on/off. As simple as the explanation sounds, a layman would have a difficult time recreating this system in their own home. So even though the software exists in a user-friendly graphic operating system, the code and vocabulary used is still very intimidating.

The Schwippy Tree project has grabbed the attention of thousands of viewers every year and provides an excellent example of what IoT can do. Viewers experience the tree from the perspective of spectators, and it’s easy to see why. The write-up on his site is riddled with hardware and confusing code. Internet of Things must have a more
accessible platform, both in complexity and cost, before it can really take off as more than just a gimmick.

2.2.1.5: Analysis

**Pros:** The software is created and used in a user-friendly graphical operating system. And the Internet connection is very reliable.

**Cons:** The software can be daunting and these days one would probably need consumer hardware to interface between a *thing* and the computer like Schwippy’s X10 computer interface box. The Schwippy example was only using 4 binary switches and it required a fair amount of hardware, an entire computer, and lots of power to run the computer, not to mention an off-site server which may not have been free.

**Cost:** The *things* used (the X10 devices) are custom commercial devices with USB interface. They are more expensive because of the necessity of a USB interface due to how the computer talks to *things*. For this reason they add to the cost of the system per node as compared to other systems.

<table>
<thead>
<tr>
<th>Item</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Desktop PC</td>
<td>~$500</td>
</tr>
<tr>
<td>1 X10 computer interface device</td>
<td>$15</td>
</tr>
<tr>
<td>4 X10 lamp modules</td>
<td>$12 each = $48</td>
</tr>
</tbody>
</table>

Total hardware cost for platform: ~$563

Software cost: Time and possible cost of off-site server.

Not to mention powering the computer 24/7.
2.2.2: Microcontroller with WiFi Adapter

2.2.2.1: System Overview

The bare bones of this method include a microcontroller and a WiFi or Ethernet shield/adapter/module, shown in Figure 6.

![High Level Block Diagram](image)

**Figure 6: Microcontroller with WiFi Shield: High Level Block Diagram**

In the last 10 to 15 years microcontrollers have become extremely popular. The market for microcontrollers has grown exponentially with the DIY hobbyist community allowing engineers and non-engineers to easily prototype their electronic designs. Microcontrollers have shaped the way people interact with their environment. With a few lines of code a total beginner can blink an LED or play sounds on a speaker or do any of hundreds of other projects. Microcontrollers, and their shift toward increasingly user-friendly interfaces, have allowed even the least experienced person to program and interact with basically any sensor. With a microcontroller anyone can create the *things* in Internet of Things.

But while microprocessors are great for connecting to *things*, they are not natively able to connect to the Internet. To create IoT with one of these microcontrollers an engineer needs to purchase and interface with a WiFi or Ethernet adapter/module.
Interfacing with a complicated non-native adapter creates software complexity, takes up more space, and adds a non-negligible monetary cost.

2.2.2.2: Hardware

This method requires, first and foremost, a microcontroller. Some of the most popular microcontrollers, as seen in Figure 7, include MSP (from TI), Propeller, BASIC Stamp, and, most commonly known, Arduino. A microcontroller is perfectly suited for communicating with and controlling things. Every microcontroller has simple digital I/O ports which allow it to read digital logic which, by extension, allows it to communicate in any digital communication protocol such as I²C, SPI, and UART. Most microcontrollers also have pins to read analog voltages. This opens the possibility to talk to analog sensors without additional hardware. Since sensors typically talk using these digital methods or analog, microcontrollers are well equipped to read/write from/to basically any sensor.

The Internet part of IoT requires more hardware since Internet is almost never native to the microcontroller. From the start of the rise of Arduino 10-15 years ago developers have created new methods of connecting to the Internet. Sparkfun, one of the frontrunners in microcontroller accessory development, has many products for connecting a microcontroller to the Internet including their WiFly GSX Breakout which

Figure 7: Microcontrollers (left to right): MSP, Propeller, BASIC Stamp, Arduino
sells for $84.95 and their WiFi Breakout – CC3000 which sells for a much more reasonable $34.95, seen below in Figure 8.

![Sparkfun's WiFly GSP (left) and WiFi CC3000 (right)](image)

Figure 8: Sparkfun's WiFly GSP (left) and WiFi CC3000 (right)

Major competitor Adafruit also develops for microcontroller platforms and currently sells the Arduino WiFi Shield 101 ($49.95), WiFi Feather STM32F205 ($34.95) and the Adafruit ATWINC1500 WiFi Breakout ($24.95) among others, as shown in Figure 9.

![Adafruit's Shield 101 (left), Feather (middle), ATWiNC1500 (right)](image)

Figure 9: Adafruit’s Shield 101 (left), Feather (middle), ATWiNC1500 (right)

WiFi breakout modules have been developed for over a decade, now. At first they were bulky and expensive but with each iteration they have gotten better in every aspect. One issue now is that there are too many. Choosing which WiFi breakout to purchase is more complex than selecting a microcontroller. At least with the microcontroller one knows what brand they are familiar with and they can select which board from that line they
need since those decisions are simply concerning speed, memory, number of pins… but it is difficult to know what WiFi adapter will work with a project. The parameters for selection are more obscure and each device has its own long set of unique instructions to get it up and running. Selecting the WiFi hardware has become a nightmare.

2.2.2.3: Software

Software, especially for a newcomer, can be daunting. Connecting to the internet, retrieving data, and sending data, can all be done in many different ways. One is described in the example. As for programming the hardware, there are loads of microcontrollers and they are all programmed differently. So for the sake of simplicity let’s say we go with Arduino. Arduino leads the pack due to its robust hardware and its extremely user-friendly software. Anyone can plug it in and either write a few lines of code or simply look at the myriad of examples online (and in the Arduino IDE itself) to get something up and running. Interacting directly with GPIO ports is as simple as one line of code. Reading an analog voltage is just as simple. And due to libraries and the massive community behind Arduino and its software, communicating with any digital protocol (I²C, SPI, UART) is simple. It has come to the point where anyone who wants to make a project can search the web and see documentation from someone else who has done the same thing before.

Of course having a large community has its drawbacks. In the last few years libraries have fallen victim to the same problem mentioned before regarding WiFi breakout hardware. There is too much out there. For the simple stuff Arduino regulates what people see and what they can use but for all this new hardware developers try to assist their hardware by creating a library. And in their hasty software they release
subpar code and people jump on it so later when something better rolls around people still use the old software because it is what they know. It gets confusing. And even though there are few groups as committed to the community as independent Arduino developers who are excited to develop, there is too much mediocre code out there. And even too much good code can make things more confusing.

This phenomenon of too many resources has a way of paralyzing what it tries to help and of all the examples of this in the electronics community, WiFi breakout software stands above the rest. Not only is the hardware flooded with expensive, complicated boards, each piece of hardware has its own unique software setup and software to get it to run. Sparkfun and Adafruit, among others, do their best to create tutorials and instructional videos and code examples for each new module they release and while it is certainly helpful in some cases, more development just makes the mess worse.

2.2.2.4: Example: Adafruit: Automatic Twitch ON AIR Sign

The following example uses an Arduino microcontroller and the latest and cheapest WiFi breakout on the market, the Adafruit ATWINC1500 ($24.95). In an attempt to make this new hardware seem approachable Adafruit has created a Library, given demos, written a long how-to on getting up and running, and has published an example of what can be created. This example builds an ON AIR sign that lights up whenever they are streaming on the website Twitch, seen in Figure 10.
The process to get this project up and running starts with their overview of the WiFi module on learn.adafruit.com [12]. This long tutorial goes over everything one needs to know about the module. From the start, one needs to solder the pins to the module, connect 6 pins (Vin, GND, SCK, MISO, MOSI, CS) to the Arduino microcontroller (see Figure 11), download the Adafruit library, run example code to update firmware, (then if the user has trouble with SSL connections they have to go through more firmware updates and certificates and updates), and then they are ready to start their project.
From here they use a URL to directly see the status of the Adafruit Stream on Twitch via the Twitch API. The URL, https://api.twitch.tv/kraken/streams/adafruit, simply displays a line of text (JSON package) stating the status of the Adafruit stream. The system diagram is shown in Figure 12. Figure 13 and 14 show the two states of the twitch API link.
Figure 12: System Block Diagram

Figure 13: Active Adafruit Twitch Stream

Figure 14: Inactive Adafruit Twitch Stream
They simply place this link into their code and parse the start of that string. If they see `{“stream”{“_id” then the stream is active and they turn/keep the light on. If they see `{“stream”:null then the stream is inactive and they turn/keep the light off. This example is as simple as that.

2.2.2.5: Analysis

**Pros:** There exists a populous community behind microcontrollers and these types of projects so no matter the hardware there will be software and example projects to follow. Microcontrollers are exceptional at interacting with things which cuts the difficulty in half when making IoT. There are lots of hardware and software options which allows for customization and optimization.

This example shows how extremely simple such a project can be. This project is not intimidating for newcomers.

**Cons:** Microcontrollers are great but they can be very limited in both memory and speed meaning the microcontroller has to use its primitive processor to not only interact with the things but to also send and receive data through this non-native adapter which could become slow for complex systems. Each microcontroller company has its own unique software which makes switching from, say, MSP to Arduino, challenging. Every WiFi module uses different software which complicates things and often limits which WiFi device can be used with which microcontroller.

This particular example works for Twitch but how does it work with other sites? Usually data does not come so easily. They even said:

Lucky for us, instead of forcing us to use OAuth or API keys or sacrificing chickens, twitch makes it really easy to see what's going on with any stream. Yay!
All you have to do is go to [https://api.twitch.tv/kraken/streams/adafruit](https://api.twitch.tv/kraken/streams/adafruit) and you'll be able to see the current status of the adafruit stream [13].

As convenient as this is for this particular case, it kind of leaves the reader in the dark if they want to use this system with a less convenient website – one that does not provide a direct link to the desired information.

**Cost:** These adapters typically fall in the price range from $25 [14] to $85 [15] which, combined with the price of the microcontroller, could be anywhere from $45 to over $100 per node. This is certainly a step up from a several hundred dollar computer but $45 per node is still too expensive to deliver on the full potential of IoT.

2.2.3: Microcomputer – e.g. Raspberry Pi

A microcomputer is essentially a barebones computer motherboard at an extremely affordable price with the GPIO (general purpose Input/Output) availability nearing that of a microcontroller. This allows the user to plug into a monitor or TV and run it just like a full-fledged computer while still communicating directly to digital logic. They usually consist of a CPU, memory, a spot for long-term storage (SD card), and plugs for peripherals. They are typically less powerful than a regular full-fledged computer but essentially work the same way in terms of compatible peripherals, operating systems, and programs, but at a cost around $30 and recently as low as $9. Microcomputers allow for us to revisit the full-fledged computer method but without the monetary cost and power consumption of a laptop or desktop.
2.2.3.1: System Overview

2.2.3.2: Hardware

Microcomputers have been on the rise for the past half a decade. Perhaps the most well known microcomputer on the market, and arguably the board that started the microcomputer craze, is the Raspberry Pi. Raspberry Pi released their first board in August of 2011 [16]. Since then their boards have improved in speed, memory, and peripherals, while maintaining comparable pricing as shown in Table 1:

Table 1: Raspberry Pi Comparison Chart (a few essential data points)

<table>
<thead>
<tr>
<th></th>
<th>Model 1 B</th>
<th>Model 1 B+</th>
<th>Model 2 B</th>
<th>Model 3 B</th>
<th>Pi Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Release Date</strong></td>
<td>Apr 2013</td>
<td>July 2014</td>
<td>Feb 2015</td>
<td>Feb 2016</td>
<td>Nov 2015</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>700MHz</td>
<td>700MHz</td>
<td>900MHz</td>
<td>1.2GHz</td>
<td>1GHz</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>512MB</td>
<td>512MB</td>
<td>1GB</td>
<td>1GB</td>
<td>512MB</td>
</tr>
<tr>
<td><strong>USB Ports</strong></td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td><strong>Price</strong></td>
<td>$35</td>
<td>$25</td>
<td>$35</td>
<td>$35</td>
<td>$5</td>
</tr>
</tbody>
</table>

The first four boards are all the same size, slightly larger than a credit card, and were released in consecutive years. The specs improved in every category while maintaining the $35 (or less) price point. The Raspberry Pi 2 is shown in Figure 15.
Figure 15: Raspberry Pi 2 B

The popularity of the Raspberry Pi sparked a whole industry of startups and companies trying to capture some of the market. As seen in Figure 16, there is Intel’s Edison, APC 8750, Cubie Board, OLinuXino, C.H.I.P. and many others. Each offers a specialty, be it faster CPU or GPU, native WiFi, or more peripherals, but nobody has been able to beat Raspberry Pi’s all-around specs for the price.

Figure 16: Pi Competitors: Edison, APC 8750, Cubie Board, OLinuXino, C.H.I.P.
The downside of Raspberry Pi to other boards is the lack of on-board long-term storage and native WiFi. All Raspberry Pi models require SD cards which adds about $4 to $30 depending on how much storage the user requires, plus a WiFi dongle (if required) adding another $5-$15. Other boards are starting to catch on and include long-term storage and WiFi making their price comparable to the Pi + these features. One board in particular showed tremendous promise. On May 7th 2015 C.H.I.P. began its Kickstarter campaign to raise funds to create a $9 board with everything a Pi had plus built-in long-term storage, WiFi, and Bluetooth [17]. It looked like C.H.I.P. was going to be the frontrunner in microcomputers but in November of that year Raspberry Pi announced its Pi Zero, same specs as the original Pi but for $5 at half the size. C.H.I.P. may still have a slight advantage but with this new product and Raspberry Pi’s already massive user base, Pi may keep the lead.

2.2.3.3: Software

Raspberry Pi and other microcomputers offer the same IoT capabilities as the full-fledged computer. Working with a Pi gives the same user experience as working on a full Linux Computer, just a bit slower and with fewer apps. This means that any method that works on a full-fledged computer works on a microcomputer. The Schwippy tree example from before would barely need to be modified to run off of a Pi instead of a desktop.

A huge (although admittedly unfair for comparison) software advantage of Pi over the full-fledged computer is all the software that has come out in recent years. Full desktop IoT existed years ago. In recent years companies have sprouted and software has been developed to make computer IoT much more manageable. Even though full
desktops can utilize the methods listed below, the only time one would use a desktop for IoT was in a time when these methods did not exist, making this comparison slightly unfair but logically valid.

Recent software provides a myriad of methods to both send and later retrieve data from the Internet using a microcomputer. One popular method of achieving data transfer over the Internet that people use all the time is via email. A microcomputer can write an email, send it, and later retrieve that data since email can be stored in the cloud. Another popular method is via Twitter. In fact SloOp, the Bouldering Gym in San Luis Obispo, has a door sensor that tracks how many people have entered and exited and posts the running count to Twitter every 10 or so minutes. These well-known methods can be used by people using a microcomputer so they can be used by a microcomputer itself.

A second, less obvious method to achieve the I in IoT is via online file storage. Companies such as Dropbox or Google Drive allow users to store video, audio, PDFs, text files… online for free. With a few lines of Python or JavaScript a microcomputer can create, append, or otherwise modify a text file and save it on one of these cloud storage options. This data can then be retrieved later by another microcomputer or person with access to the Internet and these files. This is free, just like the email/Twitter method, but feels less bulky and allows for better organization of data.

A third, less mainstream method of Internetting a thing is via data hosting websites designed to store small, text-based information. Options include Sparkfun’s Data Stream, ThingSpeak, and Firebase, among others. Let’s talk about Firebase. Firebase allows users to create free web space to store and access text-based data. Users install some software which lets them use JavaScript or Python to push and pull data to
and from that web space with a few lines of code. All a user needs is Node.js (more software which allows JavaScript to access the GPIO among other things) and they can easily interact with their things and use Firebase to connect them to the Internet.

Each method provides benefits and drawbacks. Email and Twitter will store the information forever and already has a user base of billions but may be slow and the environment may have more than the users needs. Dropbox/Google Drive can quickly store lots of data but often requires a specialized app which may run poorly on a minimalistic computer, which basically describes a microcomputer. Firebase and others require some planning in the code to organize and store lots of data but the speed is unparalleled and the lack of complexity is unbeatable. There exist many options for Internet connectivity in microcontroller, each with benefits and drawbacks. The user has the option to choose based on their system requirements.

2.2.3.4: Example: A_Comcast_User

A_Comcast_User is a Twitter account set up to automatically monitor the quality of Comcast Internet service [18]. Reddit User AleskyP explains that he pays Comcast for 150mbps download and 10mbps upload speeds. His Raspberry Pi monitors the downspeed and Tweets whenever it falls below 50mbps (1/3 of what he pays for) [19]. Figure 17 shows a block diagram of the system. Figure 18 shows the final twitter feed.
Figure 17: AComcast User Block Diagram

Figure 18: Twitter Stream
The Raspberry Pi runs some Python code to communicate via Ethernet to the router which requests PING, upload speed, and download speed from XFINITY speed test. The Pi continues the Python code to check if the speeds are insufficient. If they are it uses the Twitter library and REST API to Tweet the message.

This project takes a thing, data gathered from his Internet service provider, and posts it to the Internet via Twitter. He achieves this with a bit of Python code. Every hour the code opens a speed test site, runs a speed test, gathers the data, checks if the downspeed is under 50mbps, and if so, posts to Twitter with the canned message shown in the picture above [20]. This is simple code for an interesting project for around $40.

2.2.3.5: Analysis

Pros: Microcomputers run just like full computers. The user sees a desktop, applications, the mouse curser… the user friendly environment to which they have grown accustomed. And with all this new software a user has many options, many of which they may already feel comfortable using. Plus the physical hardware is much more discrete, usually just bigger than a credit card, and the power supply is minimal, often just a phone charger.

Cons: The power consumption on a typical microcomputer runs at around 6 watts peak and 2-3 watts idle [21]. This draws considerably less than a full desktop but still too much for long-term battery power. At this power a Raspberry Pi will run off of 3+ AAs (with linear regulator) for 2.5 to 5 hours [22]. Microcontrollers on the other hand can be placed in low-power states and last for months or even years off of AA batteries. In addition to power consumption, microprocessors lack the speed of full-fledged computers but that is to be expected for a fraction of the cost.
Cost: Raspberry Pi Zero is as little as $5 but with an SD card and WiFi dongle comes to around $15-$30 per node. C.H.I.P. advertises $9 per node with built-in storage and WiFi. The market is new which promises more innovations in the near future. A last consideration is the setup cost. The per node cost does not include setup cost. Getting the microcomputer up and running forces an initial investment. The engineer must buy or have a keyboard, mouse, monitor/TV, and anything else to get the system up and running.

2.2.4: Comparisons

The past few decades have shown a rapid technological boom in approaching a cheap IoT platform. From full-fledged computer to microcontroller to microcomputer, nearly every aspect of these systems has improved. Table 2 shows a comparison between the methods just mentioned and what will be discussed in the proposed system.

Table 2: Comparison Between IoT Platforms

<table>
<thead>
<tr>
<th></th>
<th>Full Computer (Desktop/Laptop)</th>
<th>Microcontroller (Arduino)</th>
<th>Microcomputer (Raspberry Pi)</th>
<th>Microcontroller/WiFi (ESP8266)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>$500 - $1000</td>
<td>$45 - $100</td>
<td>$9 - $45</td>
<td>$2.50 - $16</td>
</tr>
<tr>
<td>Power</td>
<td>80W – 250W</td>
<td>0.2W – 0.5W</td>
<td>2.5W – 6W</td>
<td>0.00026W – 0.56W</td>
</tr>
<tr>
<td>Hardware</td>
<td>Medium Complexity</td>
<td>Medium Complexity</td>
<td>Low Complexity</td>
<td>Very Low Complexity</td>
</tr>
<tr>
<td>Software</td>
<td>High Complexity</td>
<td>High Complexity</td>
<td>Low Complexity</td>
<td>Very Low Complexity</td>
</tr>
<tr>
<td>Size</td>
<td>Large</td>
<td>Small</td>
<td>Small</td>
<td>Very Small</td>
</tr>
</tbody>
</table>
Chapter 3: System Design

3.1: System Overview

The entire system uses the ESP8266 as the hardware. It uses the Arduino IDE to program the hardware, Firebase to store the data online, and a Web App, hosted on Dropbox, to make a human-interface for data viewing on computers/smartphones as shown below in Figure 19.

![Figure 19: System Block Diagram]

3.1.1: ESP8266: The Hardware that Makes This All Possible

This proposed system takes the first step in a new breed of IoT platform. As previously discussed, prior methods include a full-fledged computer (desktop/laptop), microcontroller (Arduino) with WiFi adapter, and microcomputer (Raspberry Pi). This method combines the low-power thing compatibility of microcontrollers with the
software ease greater than that of microcomputers. And it all lies on a new chip called the ESP8266. Shown below in Figure 20 are a few ESP8266 board models.

![ESP8266 Board Models](image)

Figure 20: ESP Models: ESP-09 (left), ESP-01 (middle), ESP-12E (right)

The ESP8266 is a microcontroller with native WiFi. This is similar to the second method, microcontroller with WiFi but the ESP8266 has WiFi built-in. This means there is no additional size or cost to adding WiFi to a system using ESP8266. The System on a Chip (SOC) design allows the board to communicate via GPIO while seamlessly connecting to, and transmitting data over, the Internet. This is the perfect marriage of Internet and thing. It stands unbeatable at a $2.50 price point, physical size a fraction of even the smaller microcontrollers, and offers ultra low power consumption. This is the best hardware all around and combined with simple, straightforward software, this chip is the future of IoT.

3.1.2: The Free Software Used in This Method

The ESP8266 offers compatibility with many previous software methods of hardware programming and Internet communication. Hardware programming can be achieved via Visual Studio, GCC, or most commonly, the Arduino IDE. The Arduino
IDE offers the least complexity without limitation of any important capabilities so this method will use that. Internet connectivity can be achieved via directly reading data from a website, like the Adafruit ON AIR example, or it can act as a server providing information to a client, but this method be using Firebase for the same reason as Arduino; it offers the least complexity without limitation of any important capabilities. Additional software includes a web app hosted on Dropbox to provide a more user-friendly environment for a user to interact with Firebase data.

Other professional, commercial methods are available for software data storage and communication. The drawbacks of these methods, however can be represented by the drawbacks of one in particular, The Cisco IoT System. Cisco emphasizes security in their IoT software and communication but their video just explaining their systems is 55 minutes long. That’s how long it takes to set up the system proposed in this thesis. So for simple, quick-to-install IoT, the proposed method gets the user up and running faster than Cisco. Not to mention the website makes it seem like one would need a company to start any sort of system, not just a person automating something in their house. And it is this barrier that implies the large potential cost to run a system with Cisco. For DIY IoT, this is not the way to go.

3.1.3: System Overview

Firebase exists in the cloud. A personal Firebase “app” (which is just a personalized homepage to store data) stands ready to send or receive data at any time from any device with the necessary credentials. The ESP8266 communicates directly to this Firebase app, either sending or requesting data. Computers and smartphones also communicate with this Firebase app, giving the engineer a user-friendly presentation to
either view or manipulate (receive or send) the data. All devices access all of the data at any time. When one device sends new data or updates a field, other devices can be notified and act accordingly. This is shown in Figure 21.

![Diagram of High-Level Internet Communication Between Devices]

**Figure 21:** High-Level Internet Communication Between Devices

### 3.1.4: Examples

This nearly instant two-way communication between all devices promises some excellent possibilities. For example an engineer with a smartphone could manipulate data on Firebase to tell an ESP8266 to turn on or off a light, instantly, from anywhere in the world. Or an engineer could set up an ESP8266 to monitor and report on the occupancy of a restroom and trigger another ESP8266 to play a sound when the restroom is available, instantly, from anywhere in the world. Or an engineer could make the equivalent of a nuclear launch-like security requiring 3 simultaneous keys to be turned but instead of keys it can be buttons and instead of 3 it could be 100 pressed simultaneously with the data transmitted instantly from 100 users, from anywhere in the world. Fun project ideas are endless.
3.2: Low-Level Design

3.2.1: Hardware: ESP8266 Board Layout Options

The ESP8266 has branched into many unique form factors over the past year. From discrete to bulky, each design has its pros and cons. Choosing a particular model is all about the project in mind and the experience one has.

3.2.1.1: Easy To Use

If it is your first time working with the ESP8266 I highly recommend using one with built-in USB programming. This negates a whole host of problems one can encounter when attempting to wire an FTDI adapter or other method of programming. The most popular ESP models with built-in USB programming, shown in Figure 22, are the ESP-12E, NodeMcu Lua, WeMosD1, and the ESPDuino.

![Figure 22: ESP-12E, NodeMcu Lua, WeMosD1, ESPDuino](image)

With these modules all one needs to do is plug them into the computer, select the correct board, write some code in the Arduino IDE, and hit upload. These boards offer simplicity that will be beneficial to first-time users. But this simplicity comes at a price. These boards are bulky - just about as bulky as a typical microcontroller. The WeMosD1 and ESPDuino are even designed in the Arduino Uno dimensions, headers and all. The ESP-12E is a bit smaller and it fits onto a breadboard (more simplicity) but it only leaves
one row of pins on either side due to its width as shown in Figure 23.

Figure 23: ESP 12-E On breadboard - Tight Fit

Another downside is power consumption. These models come with LEDs to indicate if the device is on or transmitting/receiving data which helps the user know if the board is working properly but each LED draws a considerable amount of current when considering long-term battery operation. Another power-consuming component is the FTDI adapter. The FTDI adapter allows the ESP8266 to communicate to the USB port which, again, simplified the programming aspect but also draws current. Lastly these models run for up to $9. This price beats the best of the other IoT platform methods but is still too steep for large-scale IoT. These models are fantastic for first time users or crude prototyping but for more advanced usage we will need to look at other models.
3.2.1.2: Discrete Form Factor

For the more experienced developer the ESP8266 comes in more compact and discrete layouts. With these designs an engineer can have a board smaller than the smallest microcontroller board. Some of these models include the ESP-01 (the original and most popular model), ESP-03, and the ESP-09 [23] displayed in Figure 24.

![Figure 24: Discrete ESP Boards: ESP-01, ESP-03, ESP-09](image)

Each of these models performs just like any other ESP8266 model. Drawbacks include a lower pin count, programming requires additional hardware, there is no on-board voltage regulator, and most are not breadboard friendly. Benefit, of course, is the much smaller size as well as a price range; as low as $2.50 (before bulk discount).

3.2.1.3: Low-Power Mode

The ESP8266 has the internal hardware/firmware to enter sleep mode which draws as few as 78 micro amps. To put this in perspective, a sleeping ESP8266 running on AA batteries could last 3.5 years [24]. Finally, batteries are an option for IoT.

All ESP8266 boards have this capability. One line of code puts the board to sleep for a specified amount of time. When that time is up the board sets GPIO16 HIGH. Connect GPIO16 to the RESET pin and the board will reset/wake up after sleep and start executing code again. Many of the ESP8266 models have GPIO16 broken out (the small
integrated circuit connects to larger, easily accessible pins, which a developer uses to connect to circuits). The Sparkfun “Thing” calls this pin DTR. The ESP8266-12E calls it GPIO0, as do most others. Simply connect this pin to RESET and the ESP8266 can not only sleep but wake up, too!

A problem arises when GPIO16 is not broken out. This happens in some of the smaller, more discrete models. The ESP-01 for instance only has 8 pins broken out: VCC, GND, RESET, CH_PD (chip power down), and 4 GPIO pins. None of these pins are GPIO16 so the ESP-01 cannot be woken from sleep with the broken-out pins. These particular pins were probably selected to appeal to the highest number of customers. Not many might take advantage of sleep capabilities and GPIO 16 (the pin required to wake from sleep) has limited functionality compared to other GPIO (permanent pull-down resistor, no PWM) [25]. The designers must have thought 4 fully-functioning GPIO appealed to the audience more than 3.5 fully-functioning GPIO with sleep/wake capabilities. There are ways to break out this pin but they require careful soldering. More on this in the Getting it Up and Running section.

A comparison between boards is shown in Table 3.
3.2.1.4: Comparisons

Table 3: ESP8266 Board Comparisons

<table>
<thead>
<tr>
<th>Board</th>
<th>Picture</th>
<th>Size</th>
<th>Pin (GPIO) Count</th>
<th>USB Prog</th>
<th>Easy Sleep?</th>
<th>Breadboard Friendly?</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP-12E</td>
<td></td>
<td>Medium</td>
<td>14digital 1analog</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>$6.55</td>
</tr>
<tr>
<td>NodeMcu Lua</td>
<td></td>
<td>Medium</td>
<td>14digital 1analog</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>$7.35</td>
</tr>
<tr>
<td>WeMosD1</td>
<td></td>
<td>Medium-Large</td>
<td>14digital 1analog</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$8.49</td>
</tr>
<tr>
<td>ESPDuino</td>
<td></td>
<td>Medium-Large</td>
<td>14digital 1analog</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$9.25</td>
</tr>
<tr>
<td>ESP-01</td>
<td></td>
<td>Small</td>
<td>4digital</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>$3.21</td>
</tr>
<tr>
<td>ESP-03</td>
<td></td>
<td>Very Small</td>
<td>9digital</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>$2.90</td>
</tr>
<tr>
<td>ESP-09</td>
<td></td>
<td>Ultra Small</td>
<td>8digital</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>$3.24</td>
</tr>
<tr>
<td>Thing Dev.</td>
<td></td>
<td>Medium</td>
<td>10digital 1analog</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>$16</td>
</tr>
<tr>
<td>Thing</td>
<td></td>
<td>Medium</td>
<td>10digital 1analog</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>$16</td>
</tr>
</tbody>
</table>

3.2.2: Software: Internet Connectivity

3.2.2.1: Arduino IDE: Hardware Programming

Arguably the most important piece of software is the software used to program the ESP8266. This software controls how the ESP interacts with the things and how it sends/receives data to/from the Internet. The software chosen for hardware programming in this thesis is the Arduino IDE.
The Arduino IDE has provided a platform for newcomers and professionals alike to program Atmel microcontrollers. In recent years the IDE has expanded to allow users to program a myriad of chips including ATmega, ATtiny, 32-bit ARM, Digistump AVR, Intel i686 boards, and most importantly ESP8266 boards.

A user connects the board, selects the proper board and serial port, and can access the GPIO just like a regular Arduino microcontroller. This is the same fantastic user experience with thing interaction that made microcontrollers so attractive to IoT development.

The Internet part of this IoT system uses an extremely new Arduino library which utilizes Firebase; free and fast cloud data storage. With a few lines of setup code a user is ready to send and receive data to and from Firebase. One more line of code and they can send a value or a message. Another line and they can read a value or message. It’s just that easy.

The Arduino IDE provides a comfortable, familiar, user-friendly environment for development. With ESP8266 board definitions a user can program it just like an Arduino. And with the new Firebase library a user can communicate with the Internet. Just a few lines of code and this IoT platform is ready to go.

3.2.2.2: Firebase: Data Storage

Firebase is a free online data storage utility. Every user has a Firebase “app,” essentially a webpage which holds all of their data. Firebase stores data as keys (the data identifier, like the name of a variable in C) and values (the actual numbers/letters stored or associated with that key) for example I could have a key called Name with the value of Sam or the key howBrightIsIt and the value of 52 (example value between 0 and 100).
To access this data I could program the ESP8266 to GET the *value* of Name (which would return Sam) or SET the *value* of howBrightIsIt to some other value, presumably correlated to that ESP8266’s sensors. Any ESP8266 can be programmed to GET or SET any *value* from any *key* or make up new *keys* and *values*. See Figure 25 for a sample Firebase app.

![Sample Firebase App](image)

**Figure 25: Sample Firebase App**

One benefit of Firebase over other methods is how it achieves nearly instant communication. Usually communication over the internet requires a request for information which opens a connection and closes it once it is finished transmitting. Firebase, however, uses web sockets [26], a form of communication that keeps a connection open, effectively eliminating the time needed to open/close a connection. For any high-tech Firebase connection like a computer or smartphone, web sockets make
communication incredibly fast. The ESP8266 itself is not so high-tech so it only benefits through web sockets via the computer/smartphone talking with it. The ESP8266 uses the Firebase REST API [27] which is not quite as quick as web sockets but definitely fast enough for our purposes.

Firebase is great for talking to the ESP8266 and other code-based processes but what about human interaction? Well the Firebase homepage “app” shows in plain text a list of the users keys and values in real time (see picture above). The user can click and type to modify the values of keys or create or delete keys. This is a rudimentary but fully functioning method of interacting with IoT as a human but it lacks user-friendliness. To make this system sufficiently user-friendly we use more software.

3.2.2.3: Web App for Human-Firebase Interaction

I, with the help of my brother Ben Jaffe, have created a web app for computer, smartphone, tablet… which interacts with Firebase and presents the user’s data in a more straightforward, easy-to-use manner. It reworks every functionality of Firebase to make the user experience faster and more intuitive. Figure 26 shows an example of keys and values created by the web app and the corresponding Firebase “app.”
The Firebase “app” homepage lets users click on the values of keys and type in new values. This is a time-consuming, annoying way to interact with data. To remedy this, the web app splits all data into four categories.

1. Message: A message is a key whose value is a string of letters like a word or a sentence. This is useful for sending text-based messages. See the Name key in the figure above.

2. Slider: A slider is a key whose value is selected based on the position of a physical sliding dot on a line. Users can click and drag to get a value between any user-defined numbers (0 to 255, -100 to 100…). They can also single click, then use the keyboard to move the slider one integer at a time. This is useful for applications such as light dimming or precise positioning of a servo. See the howBrightIsIt key in the figure above.

3. Momentary Button: A momentary button is a key whose value is TRUE or FALSE, like a Boolean in C. When the button is clicked, it sets the value as TRUE. When released it sets it as FALSE. Users can also create a Momentary Button whose key
is a single letter. Pressing this letter down on the keyboard will cause that corresponding
\textit{value} to go true until the user releases. This is useful for controls on a remote controlled
car or repositioning a security camera.

4. Toggle Button: A toggle button is just like a momentary button but the \textit{value}
toggles, meaning when a user clicks the button, the \textit{value} of the button inverts (goes from
\text{TRUE} to \text{FALSE} or \text{FALSE} to \text{TRUE}). This is useful for turning and keeping things on
or off like lights.

The web app code is provided in chapter 5.

3.2.2.4: Dropbox: Web Hosting of Web App

The web app lives on the web which is what makes it accessible to any Internet-
connected device, and it is personalized to the users Firebase “App” URL. For these
reasons it needs to be hosted somewhere to be accessible via the Internet. Usually web
hosting costs money and requires hours of complicated setup but we can get around this
with another web utility.

Dropbox, as mentioned in earlier sections, lets users store files online like audio,
video, text files, HTML… We’re most interested in the HTML. Dropbox not only lets
users see the HTML code, it actually runs the code. So to have a free hosted website
online a user just needs to put HTML in a shared folder on Dropbox and go to the shared
link. It’s as simple as that.

3.3: Analysis

The ESP8266 is the perfect combination of microcontroller and WiFi
connectivity. It is a fraction of the price and a fraction of the size of even the most
competitive microcontrollers (alone without WiFi) with comparable low-power states.
The Arduino IDE programs the ESP8266 to interact with *things* and the Internet via Firebase. Firebase stores the data and ensures its availability to any web-enabled device with the proper credentials. The web app makes this Firebase data user friendly, and Dropbox hosts the web app. The complete system diagram can be seen in Figure 27.

Figure 27: Diagram of Communication Between Devices

**Pros:** Hardware: The ESP8266 offers an unbeatable $2.50 price point with the size smaller than the smallest microcontrollers and a power consumption less than or equal to any other previously mentioned method. All around the hardware is the best of any method.

Software: This particular software system is not only fast but its free. And once put together the system is extremely easy to work with. No more complicated software.
**Cons:** Hardware: Programming the ESP8266 can be difficult. The discrete boards require an FTDI adapter and a half-dozen connections to make it work. And sometimes the boards with USB programming require the installation of a particular driver to communicate with the FTDI chip, but that solution is as simple as googling the part number.

Software: This method utilizes four different utilities, Dropbox, Firebase, Arduino, and the Web App. This may take a bit of time to set up and keeping the entire system in the head of a beginner might be challenging.

**Cost:** Hardware: $2.50 to $16 per node (not including power, *things*, and other hardware)

Software: Free

3.3.1: Comparison of Previous Methods

See Table 4 for comparison of methods.

<table>
<thead>
<tr>
<th></th>
<th><strong>Full Computer</strong> (Desktop/Laptop)……</th>
<th><strong>Microcontroller……</strong> (Arduino)</th>
<th><strong>Microcomputer</strong> (Raspberry Pi)</th>
<th><strong>Microcontroller/WiFi</strong> (ESP8266)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timeline</strong></td>
<td>1990’s - 2000’s</td>
<td>2005 - Present</td>
<td>2012 - Present</td>
<td>2014 – Present</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>$500 - $1000</td>
<td>$45 - $100</td>
<td>$9 - $45</td>
<td>$2.50 - $16</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>80W – 250W</td>
<td>0.2W – 0.5W</td>
<td>2.5W – 6W</td>
<td>0.00026W – 0.56W</td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>Medium Complexity</td>
<td>Medium Complexity</td>
<td>Low Complexity</td>
<td>Very Low Complexity</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>High Complexity</td>
<td>High Complexity</td>
<td>Low Complexity</td>
<td>Very Low Complexity</td>
</tr>
<tr>
<td><strong>Size</strong></td>
<td>Large</td>
<td>Small</td>
<td>Small</td>
<td>Very Small</td>
</tr>
</tbody>
</table>
Chapter 4: System Testing

4.1: Latency and Reliability Tests

4.1.1: Keyboard Press with Single ESP8266

4.1.1.1: System Setup

This first example tests the latency of a typical setup. The web-app has functionality to trigger momentary button objects with keys on the keyboard. If a momentary button is created on the web app and its name is a single letter, that key on the keyboard will trigger that key’s value to go TRUE when the key is pressed and FALSE when released. An ESP8266 is set to listen for the state of that key online and set a pin HIGH if TRUE and LOW if FALSE. This system measures the latency from the key press to the ESP8266 reaction.

4.1.1.2: Software

First a momentary button key called ‘w’ is created on the web app. See Figure 28.

Figure 28: Web App Steps for Creating a key
This creates the *key* ‘w’ in the “data” folder on Firebase and gives it a value FALSE (since it is not clicked, nor is the ‘w’ key held down) on Figure 29.

![Firebase After Web App Activity](image)

**Figure 29: Firebase After Web App Activity**

Next an Arduino Micro, acting as a virtual keyboard, presses down the ‘w’ key and sets a GPIO pin HIGH. The press of the ‘w’ key signals the web app to set the ‘w’ key as TRUE. The web app communicates this information to Firebase. The ESP8266 sees this change and sets one of its GPIO HIGH. The same goes for releasing ‘w’ and setting GPIO LOW. A Separate Arduino Uno observes these two pins and counts the number of milliseconds between the two rising or falling transitions to see the total time delay of the process.

\[
t_{\text{delay}} = t_{\text{keyboard to web app}} + t_{\text{web app to firebase}} + t_{\text{firebase to ESP8266}}
\]

All other time delays (like setting GPIO HIGH or LOW, delay of signal propagation through wires…) is negligible. See full test diagram in Figure 30.
After 20,000 cycles of pressing and releasing the ‘w’ key the system has 40,000 time delay data points. The statistics are as follows (distribution in Figures 31 and 32):

- Mean: 91.8 ms
- Median: 89 ms
- Mode: 88 ms
- Standard Deviation: 21.5 ms
- Minimum: 81 ms
- Maximum: 2495 ms
The data show a high consistency of between 81 and 100 milliseconds time delay.

There was an instance, in the roughly one hours of data retrieval, where the propagation
reached 2.495 seconds but this data point stands alone. It is probably due to an ISP or
router issue, not from the system under test, since the next highest data point is at 0.685
seconds and the 99.9th percentile is at 317ms.

In a purely random process, which is the starting point for any statistical analysis
such as this, the data will form a Gaussian/Normal/Bell Curve. The data in this
experiment forms what appears to be a skewed bell curve. Further experiments attempt
to break down the cause of this and delve into the characteristics of each time delay in the
system.

4.1.2: Keyboard Delay Test

In an attempt to narrow down the effect the keyboard has on the total delay I’ve
devised a test to isolate the keyboard latency. I have an app called Processing run a
program to start a timer and tell me at what time it registers a pressed key on the
keyboard. I programmed the Arduino Micro virtual keyboard to press and release the ‘w’
key every 15 ms for 10 repetitions, then 16 ms, then 17, up until 100 ms. The data
plotted in Figure 33 and 34 below shows the programmed delay (red) vs the delay of the
received signal (blue).
Figure 33: Delay of Key Pressed (red) vs. Time Delay of Received Key Press (blue)

Here is one section zoomed in.

Figure 34: Fig. 33 Zoomed In
About half the time the blue line is below the red which seems impossible since the blue has to happen after the red. This is due to the keyboard serial buffer. If a keyboard signal is just barely missed, it will be received on the next check making it look like it’s received before it’s sent. This discrepancy of comparing one signal to the previous signal shows up in the difference in average delays between the two data sets. The time delay of the key press (red) has an average of 56.97 ms while the time delay of received key press is 57.03 ms insinuating an average latency of 0.06 ms with quantization noise.

Figure 35 shows the data in another graph to better calculate the specifics of the quantization noise.

![Time Delay of Keyboard Press](image)

**Figure 35: Time Delay of Keyboard Press to show Quantization Characteristic**

The data show a clear categorization of time delays. No time delay falls between 2 and 11 milliseconds, nor in any other trough. Keyboards only ever interact with humans and humans do not require particularly high refresh rates to keep up. In fact computer screens only change 30-60 times per second. If we take the center of the 6th
peak and divide it by 6 we can get a good estimation of the space between peaks. This 6th peak lies at around 100ms. 100ms/6 gives 16.67 ms between peaks giving a frequency of 1/0.01667 = 59.98 Hz. This suggests the keyboard is only checked every 16.67 milliseconds or 60Hz. With the latency from before we conclude that the t_keyboard to web app has a uniform distribution of between 0 and 16.7 milliseconds with an additional 0.06 ms, or in statistical terms:

\[ t_{\text{keyboard to web app}} = U(0.06, 16.7) \text{ [ms]} \]

This same test was done by mashing the keys on a physical keyboard and the results were the same therefore comparing this virtual keyboard to a real keyboard holds validity.

4.1.3: Keyboard Press with Two ESP8266 Modules

Test 6.1.1 provides a decent baseline of expected total latency. Test 6.1.2 provides a solid model of the expected time delay of the keyboard. The next test looks in to the time delay from the web app to Firebase and from Firebase to the ESP8266.

To do this we repeat the same experiment from 6.1.1 but with two ESP8266 modules. We observe the time difference between the latencies of each module and attempt to draw more conclusions about the characteristics of the time delays. The results are shown in Figures 36 and 37.
Figure 36: Time Difference in Arrival of Button Signal through Firebase (Log Scale)

Figure 37: Time Difference in Arrival of Button Signal through Firebase

Breakdown of the 27823 data points:

Mean = 1.142 ms
Median = 0.820 ms
Mode = 0.256 ms
Standard Deviation = 1.126 ms
99.9th Percentile = 10.022 ms

These data represent the variance of the time delay from Firebase to ESP8266. The total time delay from Firebase to ESP8266 is still unknown but by observing the difference between two identical receivers with identical code in nearly identical physical positions with the same power source and the same connections we can see the stochasticity of this delay. And by the breakdown we can conclude a typical variance of around 1 ms. This affects the ~80 to 100 ms total delay but only in a small way, around 1%.

4.1.4: Conclusion

The time delay from keyboard press to ESP8266 receipt is roughly 85ms. Somewhere between 0 and 16 ms is attributed to the computer’s latency in reading the keyboard. And between 1 and 5 ms is attributed to stochasticity of the delay from Firebase to the ESP8266 as found by the comparison of two identical systems. The computer talks to Firebase with web sockets which adds negligible delay which means the rest of the delay can be attributed to Firebase relaying the information to the ESP8266.

This latency is amazingly small considering how many steps the data goes through to get from beginning to end. It should also be known that this system was tested on Charter’s 30mbps plan in the 93405 area which, in my experience, has been good but not great. A few speed tests from Ookla gives an average PING to a good Los Angeles server as 16ms.

All in all this latency is small enough for most human reaction to hardly notice. In the First Person Shooter video game community, where latency can mean life or death
in the game, an acceptable ping (latency) is anything under 100(ms). According to PINGTEST.NET a decent broadband connection will have a ping below 100ms [28]. This minimal latency and its lack of negative effect is shown in the first example: RC Car.

4.2: Example: RC Car

4.2.1: System Setup

This example demonstrates a possible application for IoT. The system uses an ESP8266 to control a remote controlled car from the user’s keyboard from anywhere in the world.

4.2.1.1: Software

The software consists of everything listed in previous sections – Web App, Firebase, Dropbox, and Arduino.

The web app contains keys “w,” “a,” “s,” and “d” to control the direction of the car (“w” and “s” for forward and backward, “a” and “d” for steering left and right). The user uses the keys on their keyboard to easily manipulate the data to easily drive the car. In addition to WASD the web app has a “Steering_Trim” key and a “Speed_Trim” key. These keys appear as sliders which can be clicked and dragged to straighten the tires (“Steering_Trim”) and make sure the motor speed is zero when it should be zero (“Speed_Trim”). The web app view is shown in Figure 38.
Dropbox hosts the web app.

Firebase holds these keys and transmits any changes made as shown in Figure 39.
Arduino code requires the most consideration. The ESP8266 streams the “data” folder which contains the WASD and trim keys. This means the ESP8266 receives a notification any time a key is modified in the “data” folder and since the web app places its data in the “data” folder, the keyboard commands will transmit directly to the car. Whenever a key is modified the ESP8266 gets a message and parses the data into a key and value. Based on the key and value the code spins the motors or turns the servo.

Another consideration falls on how to control the car. The front wheel steering runs via servo and the drive speed runs on a speed controller which also takes servo-like controls. The typical Servo library interferes with the ESP8266 WiFi functions so it cannot be used. Instead the code simply sets the servo/speed controller data pins HIGH, waits the appropriate amount of time (between 1 ms and 2 ms corresponding to the
position/speed of the servo/motor driver), then sets it LOW. It does this occasionally which, by the method in which servos work, works just fine.

This software took a few hours to get everything tested and working.

4.2.1.2: Hardware

Hardware, as shown in Figure 40, 41 and 42, includes the ESP8266 for control, a breadboard and wires for connections, the car chassis (with motors, wheels, mechanisms…), the motor speed controller, and the servo for steering. Not including the car assembly, the hardware took around 30 minutes to wire completely.

Figure 40: RC Car Black Box Diagram
4.2.1.3: Battery

On the left, in blue, sits an 1800mAh 14.8V Lithium Polymer battery. This battery provides power to all of the electronics. Unfortunately all of the electronics, from the speed controller to the servo to the ESP8266, cannot handle 14.8V so power is drawn from two of the four cells giving a total of 7.4V nominal. This is done by tapping in to the charging cable of the LiPo.

LiPo batteries are manufactured into 3.7V cells. The bigger the cell, the greater the capacity. This gives the RC car battery its 1800mAh rating. To achieve greater voltage, manufacturers wire identical cells in series. LiPo voltage ratings are measured in
number of cells in series (S). 1S means 3.7V, 2S means 7.4V and so on. This RC car battery has a rating of 4S, or 14.8V. Individual cells can be seen in Figure 43.

Figure 43: LiPo Battery Without a Cover Showing Three Cells in Series

The cells discharge from one end of the series chain to the other to give maximum voltage. Due to non-idealities such as leakage or slightly different capacity ratings between cells, each cell may have a unique voltage from the others after discharge, even if they started at the same voltage. If left unbalanced and recharged via the discharge port, these voltage differences can build and cause the battery to fail. To remedy this, LiPo batteries are charged with what are called balancing chargers. These chargers charge each cell individually, ensuring they all reach a safe, fully charged voltage regardless of initial voltage. See a LiPo battery with both discharge cable and balancing charge cable in Figure 44.
Figure 44: Battery showing Charge Cable (white) and Discharge Cable (yellow)

The battery above shows the high-power discharge cable (red and black with yellow connector) and the balanced charging cable (4 wires with white connector). The discharge cable and connection design allows for high-current discharge, enough such that the limiting factor will be the chemistry of the battery, not the wires or connectors. The charging cable, however, cannot handle as much current. The battery used in this example has a capacity rating of 1800mAh and a discharge rating of 30C meaning $30 \times 1800\text{mA} = 54\text{ amps}$. Charging is a different story. The convention to maintain a long lifecycle for a LiPo is to charge at no more than 1C, or in this case, 1.8 amps. Lucky the entire system draws less than 1.8 amps so I do not have to worry about the connectors faulting from the high current.
4.2.1.4: Speed Controller

The speed controller, shown in Figure 45, takes controls just like a servo. A servo position in the middle corresponds to a stopped motor. A servo position full one direction corresponds to full speed in one direction. And a servo position full in the other direction corresponds to full speed in the other direction. This speed controller outputs DC PWM to control the speed of the brushed DC motor. It can handle 6-24V at 3.5A continuous (10A peak) [29].

![Motor Speed Controller](image)

Figure 45: Motor Speed Controller

4.2.1.5: Steering Servo

The RC car came with a hefty (~3A current draw at stall) servo to control the steering of the front wheels. This servo takes typical servo commands to point the servo head in a particular direction. A control pulse of 1 ms means full left. A pulse of 2 ms means full right. A pulse of 1.5 ms means dead ahead.
Servos get this pulse and compare it against the position of the head. If the head is too far to one side, the power electronics in the servo kick it in the direction the pulse wants it to be. Enough kicks gets the head in the correct position. This means the servo does not care about off-time of the control signal. The control pulse can come around 50 times a second or once a minute. As long as there is some low-time, the servo will get the message and get where it needs to go (though it may take some time in the low frequency case). This fact made the software much easier. Usually an engineer would use a Servo library or configure their own timers to control servos but the Servo library interfered with the ESP8266’s WiFi protocols and messing with the timers sometimes causes the ESP8266 to crash. This helpful servo fact meant I just had to control the on-time, from 1 to 2 milliseconds, and not worry about timers. Not too much to devote a GPIO write, wait, GPIO write.

4.2.2: Performance

The performance is astounding. From person experience, and comparing against the pre-modified system using the factory controller, the control latency feels comparable to a regular RC car. The system rarely glitches. The only downside is the lack of power to the motors but that’s only due to the low 7.4V voltage source and the undersized motor controller (which was all I had laying around at the time). But this system is a proof of concept. The system under test was the ESP8266 and its ability to control the car via keyboard control over the Internet with minimal latency, and to that it did amazingly well. Anyone with the link to that web app can control this car from anywhere in the world with just as little latency (depending on the quality of their internet connection).
And not including the price of the hardware that goes into a regular RC car, this only cost $7 in parts (and can be reduced to $3).

4.3: Example: Garage Door Sensor

This next setup tests the low-power capabilities of the ESP8266. The system acts as a garage door sensor. The ESP8266 connects to a tilt switch which acts as either a short or an open based on its orientation. This system attaches to the garage door. When the door is open the system is horizontal with the door so the switch acts as a short pulling an ESP8266 GPIO pin LOW. When the door is closed the system is vertical with the door so the switch acts as an open and the GPIO is pulled HIGH by a 1MΩ resistor. This system is portrayed in Figure 46.

![Garage Door Sensor Diagram](image)

Figure 46: Garage Door Sensor Overview

The ESP8266 relays this information to Firebase over WiFi. So if a user backs their car out and drives away and forgets if they closed the garage door they can pull over and look on their phone to check. This system checks the garage door every 30 seconds, just enough time to drive out of sight and pull up the page on a phone.
This test focuses on the low-power aspect of an ESP8266 system. The hardware sits on the garage door which makes running power wires inconvenient. So this system runs off of a battery. The software utilizes deep-sleep capabilities and periodic waking to last as long as possible on that battery.

4.3.1: System Setup

Figure 47 shows the final hardware finished product.
4.3.1.1: Software

The software for this system is just like every other ESP8266 system. The ESP8266 talks to Firebase through the Dropbox-hosted Web App. The only difference is in the hardware code.

4.3.1.2: Orientation Sensor (Tilt Switch)

The orientation sensor, or tilt switch (Figure 48, 49), is a binary switch which is either a short circuit or open circuit depending on orientation.

![Figure 48: Tilt Switch Used in Example](image1)

![Figure 49: Mercury Tilt Switch with Glass Casing](image2)
The physical mechanism involves a small sealed container with a small amount of mercury inside. The mercury ball rolls back and forth from forces, namely gravity. On one side of the container rests two exposed pins sitting side by side which connect to the two external wires. When the mercury rolls in contact with the pins the external wires form a short through the mercury. When the mercury is not contacting the pins the wires form an open.

This sensor serves the system’s purposes perfectly due to its extremely low-power consumption. It is either an open circuit (drawing no power at all) or a short circuit connecting 3.3V to GND through a 1MΩ resistor. And it can be checked as quickly as the ESP8266 can read a digital pin (microseconds).

4.3.1.3: Sleep Code

The ESP8266 software libraries make sleep extremely simple. To put the ESP8266 to sleep simply call:

```c
ESP.deepSleep(time_sleep, WAKE_RF_DEFAULT);
```
where time_sleep is a number representing microseconds (1000000 = 1 second). Calling this will put the microcontroller to sleep for that many microseconds, then set GPIO 16 HIGH. By wiring GPIO 16 to the reset pin the ESP will reset itself at the end of the sleep timer, wake up, and start over at the beginning of the code.

With an already set-up Firebase/Web App/Dropbox/Arduino, creating this project took 30 minutes to code and 30 minutes to wire.

4.3.1.4: Power Considerations

The ESP8266 power consumption depends on its state. At typical active running conditions, meaning booting up, idling, reading data, connecting to WiFi, it draws about
70 milliamps. In sleep mode it draws about 77 microamps. So to minimize power consumption we have to minimize the amount of time the ESP8266 is actively executing code. Or alternatively it must sleep as much of the time as possible.

The system runs through one garage-door check cycle every 30 seconds. A cycle consists of waking, booting up, checking the sensor, connecting to WiFi, updating the garage door status online, and going back to sleep. Waking/booting takes about 0.44 seconds before it starts executing code. Checking the sensor takes no time at all. WiFi takes anywhere from 1.2 seconds to 3.5 seconds to connect based on quality of Internet available. And setting a new value to a key on firebase takes around 0.17 seconds.

The major contributor to active time turns out to be WiFi connectivity and upload. Luckily the system does not have to connect every time it wakes. It only has to connect to update the value if the state of the garage door has changed. To do this it simply checks the current state and compares it against the previous state.

Unfortunately this sleep mode works by resetting the ESP8266 which erases all volatile memory. The most elegant solution to this problem involves the use of the built-in non-volatile EEPROM. The ESP8266 board definition comes with libraries to interact with EEPROM with just a few lines of code. After the library import and the EEPROM.begin(size) there is EEPROM.write(address, value) which writes that value (an integer) to the address specified (a number from 0 to size), EEPROM.commit() which solidifies the data written, and EEPROM.read(address) which returns the value at that address. This solution reduces the active time attributed to WiFi to a few seconds a few times a day instead of a few seconds 2,880 times per day.
4.3.2: Performance

Table 5 shows the time and average current draw of each important state. Figures 51-53 show the system in action.

Table 5: ESP8266 Garage Door Sensor Power Consumption

<table>
<thead>
<tr>
<th>State</th>
<th>Time [ms]</th>
<th>Current [mA]</th>
<th>Executions per 850mAh battery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot (from Wake to Code Execution)</td>
<td>437</td>
<td>80</td>
<td>87,000</td>
</tr>
<tr>
<td>EEPROM Code</td>
<td>0.0085</td>
<td>70</td>
<td>5 billion</td>
</tr>
<tr>
<td>Connect to WiFi, Firebase, Set Value</td>
<td>3,430</td>
<td>90</td>
<td>10,000</td>
</tr>
<tr>
<td>Sleep</td>
<td>NA</td>
<td>0.078</td>
<td>454 days continuous</td>
</tr>
</tbody>
</table>

4.3.2.1: Results:

Figure 50: System on Open Garage Door
Figure 51: System on Closing /Opening Garage Door

Figure 52: System on Closed Garage Door
The system works as expected. If the garage door is opened or closed, Firebase updates with the new state in 35 seconds or less. After running for 24 hours the system has worked flawlessly. And with the sleep and low-power sensors and energy-conscious coding, this system will run for a month, or two months if checked every minute. Figure 54 shows this trend.

![Battery Life vs Check Frequency](image)

**Figure 53:** Battery Life vs. Check Frequency for 850mAh Battery

Table 6 shows the monetary costs for this project:

**Table 6: Cost for Garage Sensor**

<table>
<thead>
<tr>
<th>Item</th>
<th>$ for one</th>
<th>$ in bulk</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP8266-12E</td>
<td>$6.43</td>
<td>$5.81</td>
</tr>
<tr>
<td>Breadboard</td>
<td>$2.15</td>
<td>$1.74</td>
</tr>
<tr>
<td>Tilt Switch</td>
<td>$0.40</td>
<td>$0.35</td>
</tr>
<tr>
<td>Battery</td>
<td>$4.29</td>
<td>$3.78</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>$11.68</strong></td>
</tr>
</tbody>
</table>
Chapter 5: Conclusion and Future Work

5.1: Summary and Conclusion

Internet of Things (IoT) holds the potential to revolutionize the way we live our lives. From smart thermostats to smart cities to smart anything, so much of a typical life can be automated or improved. IoT has the potential to save time, money, and lives.

IoT methods have been around for a few decades. From desktop computers to Arduino to Raspberry Pi, these methods have improved in cost, power, size, and complexity over time. Using modern methods an IoT platform can cost as little as $9 per node or draw minimal power to last for days off of batteries, or fit in a pocket. These specs open automation to much more of the world but the true potential of IoT requires a cheaper, lower power, smaller, simpler, better system.

The method described in this thesis, using the ESP8266 chip with Firebase and Arduino (and optionally Dropbox and the Web App) takes us one large step closer to the full potential of IoT. This method costs as little as $2.50, lasts for months or even years off of batteries, can be smaller than a quarter, and only requires a few lines of code to get data moving.

The ESP8266 microcontroller/WiFi chip interacts with things, objects that can be sensed, detected, monitored… and sends that data up to Firebase. Firebase is an online cloud storage utility which stores this data. Then another ESP8266 can read this data and write back. This data is also accessible to humans via the Firebase dashboard or a custom Web App. The Web App (hosted on Dropbox) interacts with Firebase to make data manipulation on a smartphone or computer more user-friendly. The entire software
system consists of free programs making the price of the hardware the only cost of the entire system.

Benchmark tests show time latency from key press from anywhere in the world to ESP8266 receipt to be around 85ms. This opens up the possibility of fine control of remote devices such as a remote controlled car.

Benchmark tests also show the impeccable low-power capabilities of the ESP8266, sleeping at 78 microamps. This allows an IoT system utilizing ESP8266 sleep to survive for years off of a few AA batteries.

IoT holds the potential to revolutionize our way of life. This system offers a per-node cost of as low as $2.50 compared to the previous absolute minimum of $9 (and more usually $15-$40) per node. There are millions of devices one would pay an extra $2.50 to automate but not an extra $9 (or $15-$40). This new system opens all of these new possibilities. With a computer, this free software, and an ESP8266, anyone can contribute to the IoT revolution.

5.2: Future Considerations and Recommendations

Security considerations arise in essentially any new technological endeavor. This paper does not take security into consideration. Everything in this paper, if set up as described, can be accessed by anyone in the world. The only security against this is them not having the URL. There are, however, many security measures included in the software used in this app.

Security must be taken into consideration before an IoT platform goes global. Having remote control over every aspect of one’s life provides great new possibilities but those possibilities become terrifying if that control falls into the wrong hands. It could be
as simple as somebody shutting off your lights or disabling air conditioning, but the potential for these hacks gets pretty scary pretty quickly. In 2013 two men, Charlie Miller and Chris Valasek, demonstrated how they could take over a Prius, “stopping the brakes from working, fiddling with the gas gauge, turning the steering wheel, honking the horn and tightening the seatbelts – all from a laptop in the backseat while a journalist [drove]” [30]. The potential for more remote hacks expands as cars become smarter, like with Tesla’s mobile app. Hackers wouldn’t even need to be near the car to control it. Without proper security hackers could do all this and more. They might control traffic lights or modify data from, say, a remote weather station creating false warnings of storms, or just making lots of expensive data invalid. With the growing IoT market security must be a top concern.

As for this proposed system, Firebase has an entire section on security. Their site tells a user how to write code to allow certain users and not allow others. The default allows any computer (like the ones using the Web App) to read/write data. These settings can be changed to require a username/password, google login, facebook login, and more. The ESP8266 library requires a secret/token to gain access. This is another secure method of gaining access to the Firebase data. If security is a concern, Firebase has no shortage of options.

Other security issues to explore are protecting the hardware, encryption, tamper detection, and more.

Another future work consideration lies in systems for mass production. If IoT really takes off as people predict, simple DIY IoT with a half-dozen nodes will not be the norm. People will have hundreds of nodes. Certain applications might even require
thousands or millions. The per-node cost in this thesis assumed a single purchase of a single board. Purchasing thousands or millions at a time, however, reduces that cost greatly.

Once millions of devices are purchased, new systems must be developed for programming, connecting, and more. DIY systems take for granted the time of the person putting the system together. Mass production, however, requires much more careful consideration of time. Time becomes a real part of the cost.
REFERENCES


Appendix A: Bill of Materials

Table 7: Bill of Materials for RC Car

<table>
<thead>
<tr>
<th>Item</th>
<th>Cost</th>
<th>Detail</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car Chassis (w/ motor and servo)</td>
<td>~$20</td>
<td>Car chassis with large servo for steering control and large motor for speed</td>
<td></td>
</tr>
<tr>
<td>(Parts were found in attic)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP8266 12-E</td>
<td>$6.43</td>
<td>Microcontroller/WiFi Module: Reads data from online and relays commands to the car</td>
<td></td>
</tr>
<tr>
<td>Breadboard and wires</td>
<td>$2.35</td>
<td>To make wire connections</td>
<td></td>
</tr>
<tr>
<td>Battery</td>
<td>$23.24</td>
<td>To power system</td>
<td></td>
</tr>
<tr>
<td>Motor Driver</td>
<td>$28.75</td>
<td>6-24V 3.5A continuous 10A peak.</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>~$80.77</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cost for Example 2: Garage Door Sensor

Table 8: Bill of Materials for Garage Door Sensor

<table>
<thead>
<tr>
<th>Item</th>
<th>Cost</th>
<th>Detail</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP8266 Sparkfun “Thing”</td>
<td>$15.95</td>
<td>Microcontroller/WiFi Module: Reads tilt switch, relays info to Firebase</td>
<td></td>
</tr>
<tr>
<td>Tilt Switch</td>
<td>~0.40</td>
<td>Switch: Short or Open depending on orientation</td>
<td></td>
</tr>
<tr>
<td>1 MΩ Resistor</td>
<td>~0.10</td>
<td>Pull-down resistor for Tilt Switch circuit</td>
<td></td>
</tr>
<tr>
<td>Breadboard</td>
<td>$2.15</td>
<td>Board to make wire connections</td>
<td></td>
</tr>
<tr>
<td>Wires</td>
<td>~$0.50</td>
<td>Various: To make connections</td>
<td></td>
</tr>
<tr>
<td>Battery</td>
<td>$9.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Magnet</td>
<td>~$1.00</td>
<td>Attach system to metallic door</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>~$30.06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix B: Getting it Up and Running

B1: Hardware

B1.1: First Time Users

If this is your first time using the ESP8266 I highly recommend using one with built-in USB programming such as the ESP-12E, NodeMcu Lua, WeMosD1, or the ESPDuino, shown in Figure 55.

![ESP Models with USB Programming: 12E, Lua, WeMosD1, ESPDuino](image)

That way you can plug the board into the computer via USB without worrying about any confusing wiring. So if something doesn’t work you can be sure that it’s not the programming/power connection to the board that’s the problem.

B1.2: Advanced Users

If you’re looking for a smaller board you should look in to the ESP-01. This is by far the most popular model. It offers 8 pins: Vcc, GND, RESET, CH_PD, RX (GPIO1), TX (GPIO3), GPIO0, and GPIO2 which comes to 4 GPIO total. It measures in at 14.3mm by 24.8mm which is small enough for most applications. These boards can be seen in Figure 56.
From the picture you can see that the wires are 2 rows of 4 pins. While these are spaced perfectly to fit in a breadboard, the pin configuration makes the ESP-01 not breadboard friendly. You can’t plug it directly into a breadboard without shorting either the rows or columns. To remedy this you can use a parallel header to DIP converter or female to male wires as seen in Figure 57.

Figure 55: ESP-01 Model

Figure 56: Parallel to Breadboard Adapter (bottom) and M-F Wires (top)
The ESP-01, and other smaller models, do not have a built-in FTDI chip which means you need an external FTDI chip to program them. You don’t need the FTDI chip once the board is programmed so using an external FTDI chip means it’s not taking up space or power in an already programmed system. You do, however, have to go to some extra measures to program these ESP8266. First you have to acquire an external 3.3v FTDI adapter shown in Figure 58.

![Figure 57: 3.3V FTDI Adapter for Programming Discrete ESP8266](image)

Then you have to wire 3.3v to the 3.3v pin on the ESP8266, GND to GND, RX to RX, and TX to TX (not a mistake), CH_PD HIGH and GPIO 0 LOW (to program. Disconnect DPIO 0 from GND after programming). See Figure 59.
This process is the same for all ESP8266 boards that do not have USB programming built-in. The only exception is the Sparkfun “Thing” (regular, not Dev. Board). It has an auto-reset mechanism which means you don’t need to connect GPIO 0 to GND to program the “Thing.”

B1.3: Sleep Mode

All ESP8266 chips are equipped with internal hardware which can induce an ultra low-power sleep mode. The ESP8266 sets a timer, sleeps, then the timer sets one of its GPIO pins (GPIO 16) HIGH when that timer is up [31]. To wake an ESP8266 from sleep the board must be reset via the RESET pin. Connect GPIO16 to RESET and GPIO 16’s HIGH logic after sleep will reset/wake the board. Unfortunately GPIO 16 is not broken out (wired to a large, easily accessible pin) on all of the boards. Discrete boards with limited GPIO often do not have access to a broken-out GPIO16. Larger boards, like the ESP8266 12-E, the ESPDuino, and the Sparkfun “Thing” do give easy access to GPIO16.

But sleep is not impossible for a board without a broken-out GPIO 16. To get them to sleep you have to find the pinout of the chip and carefully solder a wire directly from RESET to GPIO 16. Figure 59 shows how it looks on the ESP-01 model.
This connection is much easier for boards with this pin broken out, like the Sparkfun “Thing” and the ESP8266 12-E. The connections can be seen in Figure 60:
The connection to allow the 12-E to wake from sleep involves a single wire connecting GPIO0 (labelled as D0 but connects internally to GPIO16) and RST (RESET). This is the same for most boards except the Sparkfun “Thing.” The connection required to allow the “Thing” to wake from sleep involves a single wire connecting DTR (internally connected to GPIO16) to XPD (internally wired to RESET).

Every board has the internal hardware capability to enter this sleep mode. If done properly you can get the current draw down to 78 microamps! This is low enough to run off of some AA batteries for up to 3.5 years! Let’s look at the considerations to achieve this state.

The end goal involves reducing current draw to 78 microamps. To do this we must get rid of any unnecessary component that draws current. The first big component is the FTDI adapter. The bigger boards have an on-board FTDI adapter which allows for convenient USB programming and power. The FTDI adapter used on most cheaper ESP8266 boards is the CH340G [32]. This chip uses between 50 and 80 microamps when USB is suspended. This seems small but when compared against a sleeping ESP8266 this chip can double our power consumption.

Secondly we must consider LEDs. They are great for indicating that things are working or that something is on but these surface-mount LEDs can draw around 8 milliamps each when turned on. That is around 100 times more than the sleeping ESP8266 itself [31]. They must be turned off, if possible. If not possible they must be deactivated by desoldering or cutting traces.
Lastly the *things* the system controls must also be low power. There’s no use getting an ESP8266 from 70mA to 78uA if the sensor uses 10 amps. For low power applications, sensors must also be appropriately low-powered.

B2: Software

The system uses:

- Firebase for data storage
- Dropbox for web hosting
- Firebase Interface Web App
- Arduino for hardware programming

This how-to uses a Mac running OS 10.11.3 (but all recent operating systems will work) and Google Chrome for web activity.

B2.1: Firebase Setup

Go to [firebase.com](http://firebase.com) and create an account (Sign up with Google) (Figure 61)
Enter your Google account information and click allow (Figure 62):

You’ll be taken to a page with your “apps.” You can create a new one any time with a new name. For now let’s stick with the default (yours will be named something different). Click Manage App (Figure 63):
Figure 63: Click "Manage App" on Firebase

You’ll be taken to your app’s page (Figure 64).

Figure 64: Firebase, Apps Page, First Time
As stated, the app page is used for viewing and editing your Firebase data. You can do more later like add security, hosting, and more, but for now we’ll just use it for the data. Feel free to take the tour if you feel so inclined, but you will not need to for this tutorial.

Keys and Values:
Firebase stores data in keys and values. A key is just the name of the data and the value is a number or text assigned to that key. Your main key is shown (mine is called sizzling-torch-4787). It’s current value should be null. You can change this by clicking the box that contains null and typing something new.

Now get it back to the way it was, hover over it and click the red x.

Hover over your main variable (mine is called sizzling-torch-4787) and click the green +. This makes sizzling-torch-4787 a folder which stores the data you’re about to enter.

Enter a name (key) and value to add some data.

You can create folders in Firebase by clicking the green + and giving the data a name (key) but no value. Instead, after entering the key, click the green + next to the key you’ve just created.

Now you can create folders to store your data (keys and values).
Go on your phone or another computer, log on to firebase, and observe what happens when you modify the data in your firebase app. Not only does the other device update with these modifications almost instantly, firebase highlights the data in colors to alert you of what’s been changed, added, deleted, or moved (see the legend on the right).

Now you have Firebase up and running. It will let you store any data you want and access the most current information from any device in milliseconds.

B2.2: Dropbox Setup

Go to dropbox.com. Create a profile, if you haven’t already, by clicking “Sign up for free” or “Sign up free with Google” as shown in Figure 65. After you create a profile, download and install the app.

Figure 65: Dropbox Log In
After installation, enter your login credentials and notice the dropbox icon on the top bar of your computer screen (far left icon next to wifi) (Figure 66).

![Dropbox Icon on Mac and Dropbox Login](image)

**Figure 66**: Dropbox Icon (on Mac) and Dropbox Login

Click on the icon on the top bar of your screen and click the folder icon on the top left of the pop-up window (Figure 67).

![Dropbox Drop-Down](image)

**Figure 67**: Dropbox Drop-Down

A window should pop up containing folders named Public, Photos, and more. You’ll be using your Public folder. If you place a file inside your public folder, you can right-click
it and select, “Copy Public Link.” You can then paste this link into any web browser to access that document. It works with text files, songs, videos, and even HTML for websites. We will be using the HTML feature for our IoT system.

Keep this in mind as we move on to the next step.

B2.3: Web App for Firebase Interface

Firebase is great for storing data but the interface gets a bit annoying when you want to change data quickly or interact with it in a way other than typing, especially on a smartphone. That is why I, with the help of my brother Ben Jaffe, have created a simple website app to more easily interact with Firebase data.

1. Download the code [here](#).
2. Unzip the file and locate app.js
3. Open app.js in any text editor (text edit, sublime, text wrangler…)
4. See the line of code that says “var fbUrl = 'https://sizzling-torch-4787.firebaseio.com/';”
6. Save app.js and close your text editor
7. Drag the entire *FirebaseInterface* folder into your Dropbox Public folder
8. Once in Public, locate index.html, right click, and select “Copy Public Link”
9. Paste this link into any browser and play around with the interface.
This interface lets you to create, name, and set values to your data. You can set them as one of four types of module:

1. Momentary Button - Is TRUE when clicked and FALSE when released.
   a. Momentary buttons can also be controlled by keyboard keys
   b. Name a momentary button a single letter, then press that letter on the keyboard to activate the momentary button.

2. Toggle Button - Click to set as TRUE, click again to set as FALSE, and so on.

3. Slider - This sets the key’s value to a number from 0 to 100 depending on the position of the slider
   a. The range (0-100) can be changed by modifying the app.js file on lines 170 and 171 (min and max).

4. Text - This lets you type messages or simply set Strings of text.

Features:

If you create multiple different types of modules with the same name, each will control that same key. You can use this to see the value of a slider by having a slider and text of the same key, or use a toggle button to occasionally stick a momentary button as FALSE.

Use the same link in other browsers on other computer or smartphones anywhere in the world. Firebase will update instantly as you play around with this web app. Any time something changes in this web app, Firebase will be alerted, will update its data, and will then alert any other person using this web app elsewhere. So even without
microcontrollers, this web app can be used to send messages from one computer to another, or show the boolean status of something not yet automated by a microcontroller.

B2.4: Arduino IDE for Hardware Programming

We will be using the ESP8266 wifi/microcontroller to read sensors or write to other peripherals while talking to Firebase. Luckily the Arduino community has made uploading to the ESP8266 and reading from/writing to Firebase very simple.

If you haven’t already, Download the Arduino IDE at Arduino.cc. Install it and get it up and running.

Next we’ll be following the instructions set by “proppy” on GitHub for interfacing the ESP8266 in the Arduino IDE with Firebase. Since this is an ongoing project at the time of this paper’s completion, I will provide direct links to stable versions of the documents linked on the GitHub.

First we need the board definition. The barebones Arduino IDE doesn’t know what the ESP8266 is, nor how to talk to it so this folder will educate it. Download and unzip this document. Then place the folder in Documents → Arduino → hardware. If there is no hardware folder, make one.

Now we need the Firebase library so the Arduino IDE knows how to tell the ESP8266 to talk to Firebase. Download and unzip this document. Then place the folder in Documents → Arduino → libraries. If there is no libraries folder, make one.
Restart the Arduino IDE.

Create a new sketch in the Arduino IDE called FirebaseStream. Copy and paste this code into the sketch. Replace the WiFi.begin() parameters with your personal SSID and password. Connect the ESP8266 to the computer via USB cable or FTDI adapter (as described in the hardware section). Select the correct board (tools-board) and serial port (tools-port) and upload the sketch. Open the serial monitor (tools-Serial Monitor) to see the realtime bitcoin valuation data pulled from a public firebase site.

In the Arduino IDE to go file → examples → firebase-arduino-master → FirebasePush_ESP8266. Look through the code and replace the SSID, Password, Firebase URL, and the secret or code (in .auth()). Your SSID is your Internet’s name. The password is your Internet’s password. My Firebase URL is https://sizzling-torch-4787.firebaseapp.com/ and yours should be different, and the secret can be found on your firebase dashboard. Go to your firebase URL, on the lefthand side select “Secrets,” click to show secret, then copy and paste that secret into your .auth() in your Arduino code.

Once booted up and connected you should have a new key and value in your firebase app showing a timestamp.

The Web App stores data in a folder called “data.” To get the value of a key simply call fbase.get(“key_name”); and it will return the value. To set the value of a key simply call fbase.set(“key_name”, your_value); and it will set that key as that value. For faster data reading look at the streaming example and stream (“/data”) to quickly receive updates on the data from the Web App.

It’s as simple as that.
Appendix C: Code

The following section provides all code used in and for this thesis.

C1: Web App Javascript

/** @jsx React.DOM */

var types = [
    {
      key: 'momentaryButton',
      readableValue: 'Momentary Button'
    },
    {
      key: 'toggleButton',
      readableValue: 'Toggle Button'
    },
    {
      key: 'slider',
      readableValue: 'Slider'
    },
    {
      key: 'text',
      readableValue: 'Text'
    }
];

var fbUrl = 'https://sample.firebaseio.com/';

var ToggleButton = React.createClass({
  componentDidMount: function() {
    if (this.props.name.length === 1) {
      window.addEventListener('keydown', this._onKeyDown);
    }
  },

  _onKeyDown: function(e) {
    var char = String.fromCharCode(e.keyCode).toLowerCase();
    if (char === this.props.name) {
      this.props.onClick(e);
    }
  },

  componentWillUnmount: function() {
    if (this.props.name.length === 1) {
      window.removeEventListener('keydown', this._onKeyDown);
    }
  } });
window.removeEventListener('keydown', this._onKeyDown);
},

render: function() {
  return (
    <span>
      <img className="icon" src="images/toggleButton.jpg" width="16"/>
      <label>{this.props.name}</label>
      <input 
        type="button" 
        value={this.props.value} 
        onClick={this.props.onClick} />
    </span>
  );
}
});

var MomentaryButton = React.createClass({
  componentDidMount: function() {
    if (this.props.name.length === 1) {
      window.addEventListener('keydown', this._onKeyDown);
      window.addEventListener('keyup', this._onKeyUp);
    }
  },
  componentWillUnmount: function() {
    if (this.props.name.length === 1) {
      window.removeEventListener('onKeyDown', this._onKeyDown);
      window.removeEventListener('onKeyUp', this._onKeyUp);
    }
  },
  _onKeyDown: function(e) {
    var char = String.fromCharCode(e.keyCode).toLowerCase();
    if (char === this.props.name) {
      this.props.onMouseDown(e);
    }
  },
  _onKeyUp: function(e) {
    var char = String.fromCharCode(e.keyCode).toLowerCase();

if (char === this.props.name) {
    this.props.onMouseUp(e);
}

render: function() {
    return (
        <span>
            <img className="icon" src="images/momentaryButton.jpg" width="16"/>
            <label>{this.props.name}</label>
            <input
                type="button"
                value={this.props.value}
                onMouseDown={this.props.onMouseDown}
                onMouseUp={this.props.onMouseUp}
            />
        </span>
    );
};
});

var Dashboard = React.createClass({
    mixins: [ReactFireMixin],

    getInitialState: function() {
        return {
            data: {}
        },

    },

    componentWillMount: function() {
        var firebaseRef = new Firebase(fbUrl + 'data/');
        this.bindAsObject(firebaseRef, 'data');
    },

    _onChange: function(type, name, e) {
        var value;
        var elem = e.target;

        if (type === 'slider') {
            value = elem.value * 1;
        } else if (type === 'text') {
            value = elem.value;
        }

        this.firebaseRefs['data'].child(name).set(elem.value);
_onClick: function(type, name, e) {
    var elem = e.target;
    this.firebaseRefs['data'].child(name).set(!this.state.data[name]);
},

_onMouseDown: function(type, name, e) {
    var elem = e.target;
    this.firebaseRefs['data'].child(name).set(true);
},

_onMouseUp: function(type, name, e) {
    var elem = e.target;
    this.firebaseRefs['data'].child(name).set(false);
},

_getControlElem: function(control) {
    var type = control.type;
    var name = control.name;
    var value = this.state.data[name] || false;
    if (type === 'toggleButton') {
        return <ToggleButton
            name={name}
            value={value}
            onClick={this._onClick.bind(null, type, name)}
        />
    } else if (type === 'momentaryButton') {
        return <MomentaryButton
            name={name}
            value={value}
            onMouseDown={this._onMouseDown.bind(null, type, name)}
        
            onMouseUp={this._onMouseUp.bind(null, type, name)}
        />
    } else if (type === 'slider') {
        return (
            <span>
                <img className="icon" src="images/slider.jpg" width="16"/>
                
                <input
                    value={value}
                    type="range"
                    min="0"
                
            </span>
        )
max="10"
onChange={this._onChange.bind(null, type,
name)} />
</span>
);
} else if (type === 'text') {
return (  
<span>
<input
value={value}
type="text"
onChange={this._onChange.bind(null, type,
name)} />
</span>
);
}
,
_getReadableTypeName: function(key) {
var type = types.filter(function(type) {
  return type.key === key;
});
return type.length ? type[0].readableValue : null;
},

render: function() {
  var _this = this;
  var createItem = function(item, index) {
    return (  
      <li key={ index } className="data-item">
        <span onClick={ _this.props.removeItem.bind(null,
item['.key']) } className="removeButton">
          X
        </span>
        { _this._getControlElem(item) }
      </li>
    );
    return <ul className="data-items">
      { this.props.items.map(createItem) }
    </ul>;
  };
  return <ul className="data-items">
    { this.props.items.map(createItem) }
  </ul>;
});
var DashboardEditor = React.createClass({

  getInitialState: function() {
    return {
      name: '',
      type: ''
    }
  },

  handleSubmit: function(e) {
    e.preventDefault();
    var name = this.state.name;
    var type = this.state.type;

    if (!name || name.trim().length === 0 ||
        types.filter(function(filter) { return filter.key ===
        type; }).length === 0)
    {
      alert('Please choose a name and a type');
      return false;
    }

    this.props.addItem({
      name: name.trim(),
      type: type
    });
    this.setState({name: ''});
  },

  _makeTypeRadioElem: function(type) {
    return (</div>
      <input
        type="radio"
        name="type"
        id={'selector-for-' + type.key}
        value={type.key}
        onChange={this._onTypeChange} />
      <label
        htmlFor={'selector-for-' + type.key}>
        {type.readableValue}
      </label>
    </div>);
_onKeyChange: function(e) {
    this.setState({name: e.target.value});
},

_onTypeChange: function(e) {
    this.setState({type: e.target.value});
},

render: function() {
    return (
        <form onSubmit={ this.handleSubmit }>
            <label for="type">Type: </label><br />
            { types.map(this._makeTypeRadioElem) }
            <br />

            <label for="key">Key: </label>
            <input id="key" onChange={ this._onKeyChange } value={ this.state.name } />
            <br />

            <button>Add Control Element</button>
        </form>
    );
});

var DashboardApp = React.createClass({
    mixins: [ReactFireMixin],

    getInitialState: function() {
        return {
            items: [],
            name: ''
        };
    },

    componentWillMount: function() {
        var firebaseRef = new Firebase(fbUrl + 'items/');
        this.bindAsArray(firebaseRef, 'items');
    },

    removeItem: function(key) {
        var firebaseRef = new Firebase(fbUrl + 'items/');
        firebaseRef.child(key).remove();
    },
addItem: function(item) {
  this.firebaseRefs['items'].push(item);
  this.setState({
    name: ''
  });
},
render: function() {
  return (
    <div onKeyDown={this._onKeyDown}>
      <Dashboard items={this.state.items} removeItem={this.removeItem} />
      <hr />
      <DashboardEditor items={this.state.items} addItem={this.addItem} />
    </div>
  );
};

React.render(<DashboardApp />, document.getElementById('dashboard-app'))
C2: Web App HTML

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <!-- React JS -->
  <script src="https://fb.me/react-0.13.3.min.js"></script>
  <script src="https://fb.me/JSXTransformer-0.13.3.js"></script>

  <!-- Firebase -->
  <script src="https://cdn.firebase.com/js/client/2.3.0/firebase.js"></script>

  <!-- ReactFire -->
  <script src="https://cdn.firebase.com/libs/reactfire/0.5.1/reactfire.min.js"></script>

  <!-- Libraries -->
  <script src="./node_modules/keymaster/keymaster.js"></script>

  <script type="text/jsx" src="app.js"></script>

  <style>
    .icon {
      margin-right: 5px;
    }

    .removeButton {
      color: red;
      margin-right: 5px;
      cursor: pointer;
      opacity: 0;
      transition: opacity 0.3s;
    }

    .data-items {
      list-style-type: none;
      margin: 0;
      padding: 0;
    }
  </style>
.removeButton:hover {
  opacity: 1;
}
</style>
</head>
<body>
  <div id="dashboard-app"></div>
</body>
</html>
C3: Arduino code for RC car

#include <Firebase.h>

// create firebase client.
Firebase fbase = Firebase("sample.firebaseio.com")
    .auth("secret_or_token");
FirebaseStream stream;

int steeringRPin = 4;
int steeringLPin = 5;
int speedFPin = 1;
int speedRPin = 3;
int honkPin = 2;
int headlightsPin = 0;

void setup()
{
    Serial.begin(9600);
    pinMode(steeringLPin, OUTPUT);
    pinMode(steeringRPin, OUTPUT);
    pinMode(speedFPin, OUTPUT);
    pinMode(speedRPin, OUTPUT);
    pinMode(honkPin, OUTPUT);
    pinMode(headlightsPin, OUTPUT);
    // connect to wifi.
    WiFi.begin("SSID", "Password");
    Serial.print("connecting");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print(".");
        delay(500);
    }
    Serial.println();
    Serial.print("connected: ");
    Serial.println(WiFi.localIP());
    stream = fbase.stream("/data");
}

void loop()
{
    if (stream.error()) {
        Serial.println("streaming error");
        Serial.println(stream.error().message());
    }
    if (stream.available()) {
        String event;

auto type = stream.read(event);
//Serial.print("event: ");
//Serial.println(type);
if(event.charAt(10) == '\"')
{
}
else if (type == FirebaseStream::Event::PUT) {
    //Serial.print("data: ");
    Serial.println(event);

    String data = event;
    int i = 0;
    String key = "";
    String val = "";

    while(data[i] != ':')
    {
        i++;
    }
    i += 3;
    while(data.charAt(i) != '\"')
    {
        key += data.charAt(i);
        i++;
    }
    while(data.charAt(i) != ':')
    {
        i++;
    }
    i++;
    while(data.charAt(i) != '}')
    {
        if(data.charAt(i) != '\"')
        {
            val += data.charAt(i);
            if(data.charAt(i) == '}')
            {
                break;
            }
        }
        i++;
    }
    Serial.println(val);
    if(key.equals("w"))
    {

if(val.equals("true"))
{
    digitalWrite(speedFPin, HIGH);
    digitalWrite(speedRPin, LOW);
}
else if(val.equals("false"))
{
    digitalWrite(speedFPin, LOW);
    digitalWrite(speedRPin, LOW);
}
else if(key.equals("s"))
{
    if(val.equals("true"))
    {
        digitalWrite(speedFPin, LOW);
        digitalWrite(speedRPin, HIGH);
    }
    else if(val.equals("false"))
    {
        digitalWrite(speedFPin, LOW);
        digitalWrite(speedRPin, LOW);
    }
}
else if(key.equals("a"))
{
    if(val.equals("true"))
    {
        digitalWrite(steeringRPin, LOW);
        digitalWrite(steeringLPin, HIGH);
    }
    else if(val.equals("false"))
    {
        digitalWrite(steeringRPin, LOW);
        digitalWrite(steeringLPin, LOW);
    }
}
else if(key.equals("d"))
{
    if(val.equals("true"))
    {
        digitalWrite(steeringRPin, HIGH);
        digitalWrite(steeringLPin, LOW);
    }
    else if(val.equals("false"))
    {
    
}}
digitalWrite(steeringRPin, LOW);
digitalWrite(steeringLPin, LOW);
}
}

else if(key.equals("h"))
{
    if(val.equals("true"))
    {
        digitalWrite(honkPin, HIGH);
    }
    else
    {
        digitalWrite(honkPin, LOW);
    }
}

else if(key.equals("Headlights"))
{
    analogWrite(headlightsPin, val.toInt() * 102);
}
}
C4: Arduino code for Garage Door Sensor

```cpp
#include <Firebase.h>
#include <EEPROM.h>

Firebase fbase = Firebase("sample.firebaseio.com")
  .auth("secret_or_token");

void setup()
{
  EEPROM.begin(512);
  pinMode(13, INPUT);

  //read the previous status
  int garageDoorStatus = EEPROM.read(0);

  //read the new status
  int newStatus = digitalRead(13);

  //has the status changed?
  if(newStatus != garageDoorStatus)
  {
    EEPROM.write(0, newStatus);
    EEPROM.commit();

    // connect to wifi.
    WiFi.begin("SSID", "Password");
    while (WiFi.status() != WL_CONNECTED)
    {
      delay(1);
    }
    if(newStatus == 0)
    {
      fbase.set("/data/Window_Status", "\"Open\"");
    }
    else
    {
      fbase.set("/data/Window_Status", "\"Closed\"");
    }
  }

  ESP.deepSleep(1000000, WAKE_RF_DEFAULT);
}

void loop() {
}
```
C5: Latency Test Codes

C5.1: Keyboard press with single ESP

C5.1.1: Virtual Keyboard Code

```c
#include <Keyboard.h>

unsigned long timeCount = 0;

void setup() {
    // make pin 2 an input and turn on the
    // pullup resistor so it goes high unless
    // connected to ground:
    pinMode(2, OUTPUT);
    pinMode(12, OUTPUT);
    // initialize control over the keyboard:
    Keyboard.begin();
    Serial.begin(9600);
    delay(3000);
    Serial.println("hi");
    digitalWrite(12, HIGH);
    delay(100);
    digitalWrite(12, LOW);
    delay(1000);

    for(int i = 0; i < 10000; i++)
    {
        Keyboard.press('w');
        digitalWrite(2, HIGH);
        timeCount = micros();
        while(!digitalRead(3));
        //Serial.println(micros() - timeCount);
        Keyboard.releaseAll();
        digitalWrite(2, LOW);
        timeCount = micros();
        while(digitalRead(3));
        //Serial.println(micros() - timeCount);
    }
    digitalWrite(12, HIGH);
    delay(100);
    digitalWrite(12, LOW);
    delay(300);
    digitalWrite(12, HIGH);
```
delay(100);
digitalWrite(12, LOW);
delay(1000);
}

void loop() {}
C5.1.2: ESP Code

#include <Firebase.h>

// create firebase client.
Firebase fbase = Firebase("sample.firebaseio.com")
        .auth("secret_or_token");
FirebaseStream stream;

void setup()
{
    Serial.begin(9600);
    pinMode(steeringLPin, OUTPUT);
    pinMode(steeringRPin, OUTPUT);
    pinMode(speedFPin, OUTPUT);
    pinMode(speedRPin, OUTPUT);
    pinMode(honkPin, OUTPUT);
    pinMode(headlightsPin, OUTPUT);
    // connect to wifi.
    WiFi.begin("SSID", "Password");
    Serial.print("connecting");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print(.);
        delay(500);
    }
    Serial.println();
    Serial.print("connected: ");
    Serial.println(WiFi.localIP());
    stream = fbase.stream("/data");
}

void loop()
{
    if (stream.error()) {
        Serial.println("streaming error");
        Serial.println(stream.error().message());
    }
    if (stream.available()) {
        String event;
        auto type = stream.read(event);
        //Serial.print("event: ");
        //Serial.println(type);
        if(event.charAt(10) == \\
        )
        {

"}
} else if (type == FirebaseStream::Event::PUT) {
//Serial.print("data: ");
Serial.println(event);

String data = event;
int i = 0;
String key = "";
String val = "";

while(data[i] != ':')
{
    i++;
}
i += 3;
while(data.charAt(i) != '\"]")
{
    key += data.charAt(i);
i++;
}
while(data.charAt(i) != ':')
{
    i++;
}
i++;
while(data.charAt(i) != '}')
{
    if(data.charAt(i) != '\"]")
    {
        val += data.charAt(i);
        if(data.charAt(i) == '}')
        {
            break;
        }
    }
i++;
}
Serial.println(val);

if(key.equals("w"))
{
    if(val.equals("true"))
    {
        digitalWrite(2, HIGH);
    }
    else if(val.equals("false"))
    {

digitalWrite(2, LOW);
}
C5.1.3: Arduino Timer Code

```c
void setup()
{
  Serial.begin(57600);
  pinMode(2, INPUT);
  pinMode(3, INPUT);
}

void loop()
{
  while(!digitalRead(2) && !digitalRead(3))
  {
    if(digitalRead(2))
    {
      unsigned long timer = micros();
      while(!digitalRead(3));
      Serial.println(micros() - timer);
      break;
    }
    if(digitalRead(3))
    {
      unsigned long timer = micros();
      while(!digitalRead(2));
      Serial.println(micros() - timer);
      break;
    }
  }
  while(digitalRead(2) && digitalRead(3))
  {
    if(!digitalRead(2))
    {
      unsigned long timer = micros();
      while(digitalRead(3));
      Serial.println(micros() - timer);
      break;
    }
    if(!digitalRead(3))
    {
      unsigned long timer = micros();
      while(digitalRead(2));
      Serial.println(micros() - timer);
      break;
    }
  }
}
```
C5.2: Keyboard delay test

C5.2.1: Virtual Keyboard Code

#include <Keyboard.h>

void setup() {
    Keyboard.begin();
}

void loop() {
    Keyboard.press('w');
    delay(random(0, 100));
    Keyboard.releaseAll();
    delay(random(0, 100));
}