**W.A.S.T.E.**

**Wi-Fi Alarm System Trusted Environment**

**Sarah Devitt, Thomas Krumenacker, Mary Kate Nawalaniec, Samuel Proulx**

# 

# 

# 

**Contents**

[**1 Introduction**](#_9jn4bdmhqy6) **4**

[**2 Detailed System Requirements**](#_tmmxrupf1w18) **6**

[**3 Detailed Project Description**](#_d6qsrylx4wsz) **9**

3[.1 System Theory of Operation](#_vnu6sanw49qz) 9

3[.2 System Block Diagram](#_lz7s6mg7alzb) 9

3[.3 Detailed Operation of Hardware](#_xptt74omaphs) 10

3[.4 Detailed Operation of Software](#_twa4dtabauks) 12

3[.5 Detailed Operation of Mobile App](#_jrcutfsg82sm) 15

3[.6 Interfaces](#_m1l1riefwhab) 18

[**4 System Integration Testing**](#_wd7jjcl2v8h1) **20**

4[.1 Description of subsystem testing](#_cx5hvlqo3jbs) 20

4[.2 Testing and Requirements](#_oyoywd7z1vwy) 21

[**5 User Manual/Installation Manual**](#_pln0nq2tal9a) **23**

5[.1 How to install your product](#_mzc6fkxeumos) 23

5[.2 How to setup your product](#_mnxthspxtl4) 23

5[.3 How the user can tell if the product is working](#_jumluyi6o7ff) 29

5[.4 How the user can troubleshoot the product](#_6ekdnjp5r65) 29

[**6 To-Market Design Changes**](#_7fyapsm8mg5p) **31**

6[.1 Hardware/Software Enhancements](#_7fgxpem6ct5g) 31

6[.2 Mobile App Future Enhancements](#_16m7hkl88i1i) 32

[**7 Conclusions**](#_96lzs0j3zqld) **33**

[**8 Appendices**](#_qqvp30od4s1l) **34**

# 

# **1 Introduction**

Every year, there are over 2 million burglaries, which is the greatest threat to homeowners according to the FBI.[[1]](#footnote-0) Home security systems are the best way to prevent this. Several studies have shown that a burglar will first look for a home security system, and avoid such homes.[[2]](#footnote-1) For the security and peace of mind of the homeowner and their family, pets, and belongings, installing a home security system is the best way to prevent intrusion. The project W.A.S.T.E. (Wi-Fi Alarm System Trusted Environment) is a solution to these burglaries and a way to feel safe at home.

W.A.S.T.E. is a Wi-Fi based home system that works differently than traditional security systems, where a phone line can be cut, disrupting any signals that notify of a break in. Rather, the W.A.S.T.E. system uses an internet connection to directly notify the homeowner when one of the sensors installed in their home is triggered. The system uses a network of sensors that, when triggered, either by someone moving around a room or opening a window or door, sends a notification to their mobile device.

Our board is designed around the ESP8266 module, a low-power Wi-Fi chip; with a battery charging module, MCP73831; a DC-DC converter for voltage stability, TPS61200; a one-shot chip to easily reset the ESP, 74LVC1G123; and associated logic to use various sensors. A slide switch is used to turn the board on/off, and a pushbutton is used to hard reset the Wi-Fi network. Figure 1 shows the final schematic of our board.

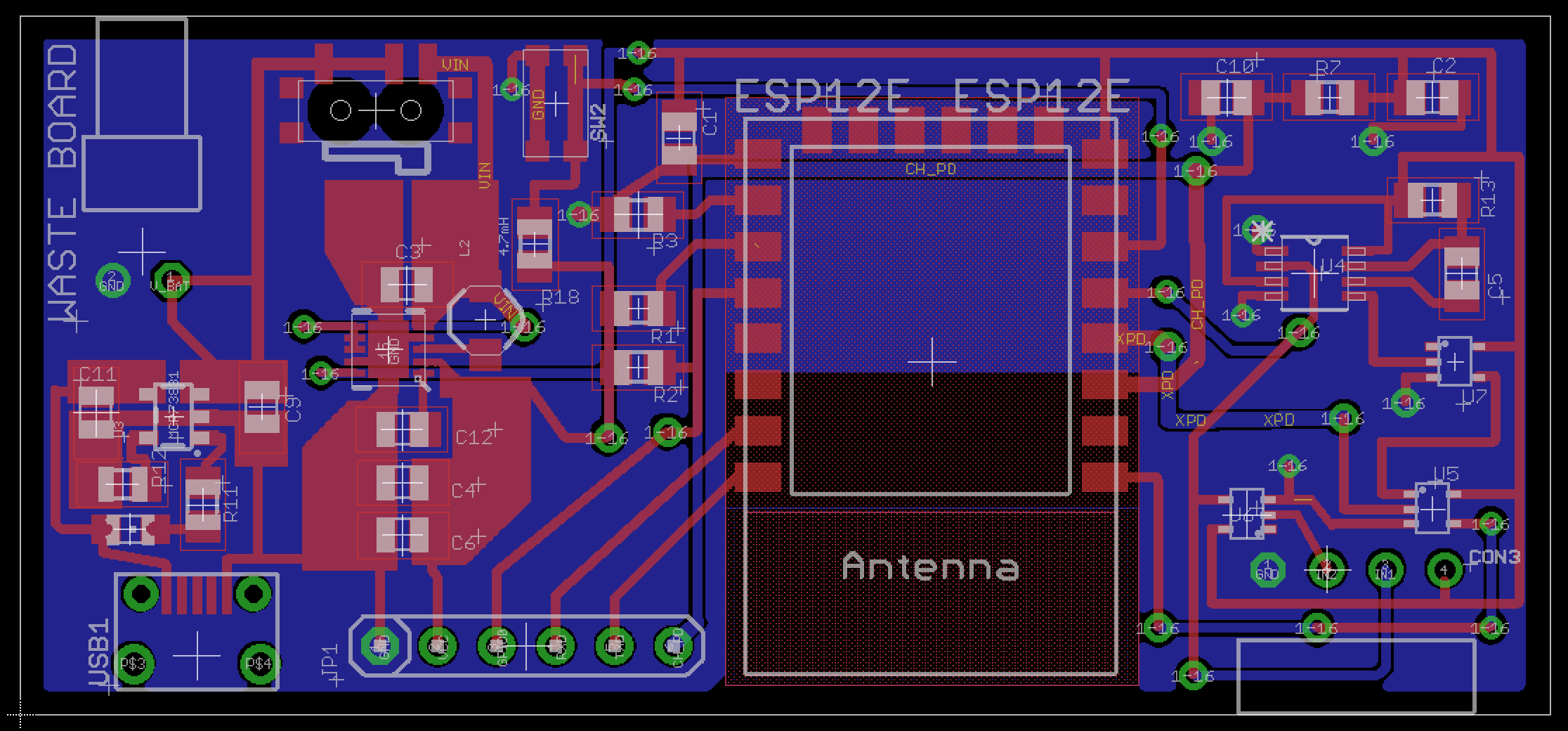


Figure 1. Final Board Schematic

The board we designed was flawed in several ways, but these were corrected easily by hand. Issues included missing connections, incorrect logic, and several floating values, each of which is corrected in our final schematic. The board that was printed and used for the demo is shown in Figure 2.

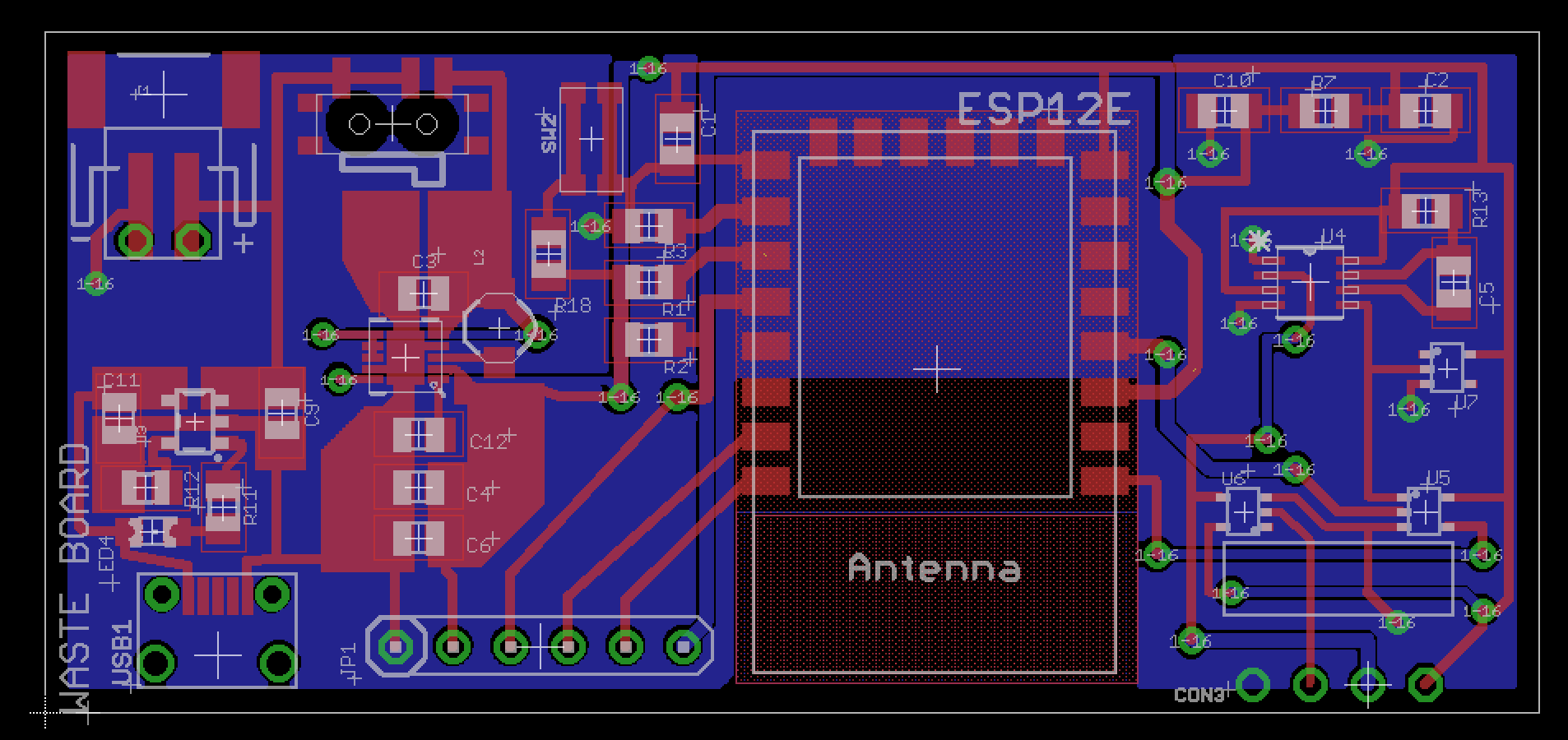


Figure 2. Final Board Comparison

The implementation of the sensors was met with several problems, such as debouncing the entry sensor, and making the motion sensor reliable. These problems lead to reset issues with the board, that can be avoiding in testing by using a wire or switch to simulate the sensor acting perfectly. The system is as accurate as possible using these sensor inaccuracies, WiFi connection delay, and logic timing, however most of our hardware has software redundancy to ensure precise performance.

In a realistic setting, additional security measures would have to be taken, particularly with publishing and monitoring the MQTT server. Also, the board could theoretically be continuously reset by tampering with the sensor, causing the board to constantly try to connect to WiFi but be reset before sending the triggered message. Some of these complications could be avoided by altering our design, such as physical metal contacts instead of a reed switch, or never going to sleep and constantly be connected to WiFi, but these changes bring additional complications. No design is perfect, but we are confident our approach solves this engineering problem reliably and efficiently.

# 2 Detailed System Requirements

The overall system requires a robust board that will be able to accommodate several different sensors. Assuming the system will be set up in an apartment style home, like one of the off campus houses, the maximum range would be 50 feet, from board to the home router.

To start, the system will include two main types of sensors: entry and motion. Entry sensors use a magnetic reed switch, seen in Figure 3. One half of the assembly is set on a window or door frame and the other attached to the window or door itself. When the switch set is separated from each other the contact is broken and triggers the board.



Figure 3. Entry (Door/Window) Sensor

The motion sensor, a passive infrared sensor (PIR), seen in Figure 4, is used to monitor a general space for occupancy. The device takes power and ground, and outputs a data line that sends a logic high when motion is detected. The sensor used can be adjusted to detect motion from three to seven meters, and with a time delay of five seconds to five minutes.

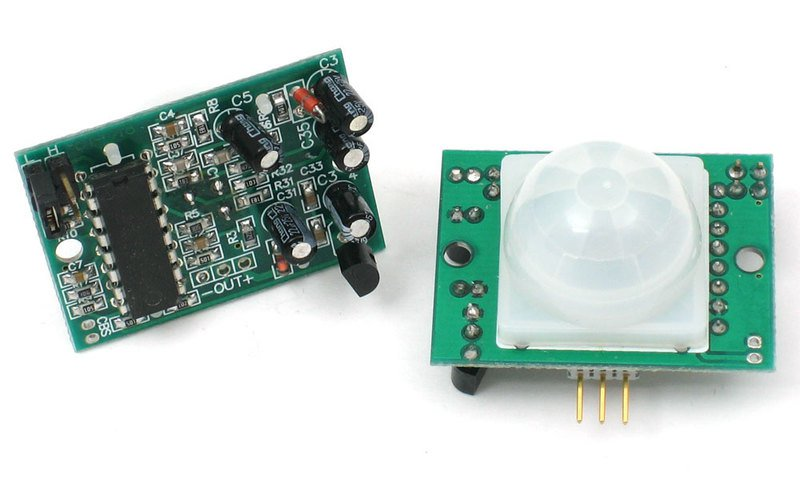


Figure 4. Infrared Occupancy Sensor

The user interfaces with the system through a mobile application. The app displays the status of the sensor and the time the last update was received. They will also have the ability to add or remove sensors from the application. Shown below in Figure 5 are some screenshots of the mobile application.



Figure 5. Mobile Application Screens

The app allows the system to be adaptive to different user settings. The user can enter their MQTT server information and Wi-Fi network SSID and password.

System installation will be very straightforward. A one-time install is all that is necessary. Each sensor will need to be connected to the Wi-Fi network individually. Then each sensor will need to be installed in the desired location. Once the initial installation is complete, the system should run smoothly on the Wi-Fi network. The only maintenance would be to recharge the batteries in the sensor as they die.

None of the voltages or currents are in excess of normal household products. Any and all safety precautions are those associated with common products--such as making sure that the product is plugged in properly, not put it near water or outside where it might be exposed to the elements and malfunction in such a way.

The mechanical requirements are minimal. Once the system is installed so that the door and window sensor have strong magnetic connections, and the motion sensor is in a desired location, installation is complete. Mechanical requirements are not a major concern of this project.

# 

# 

# **3 Detailed Project Description**

## **3.1 System Theory of Operation**

Once installation is complete, each sensor should be set up at entry points or monitoring rooms so that any unwanted movement or action will be detected. For example, an entry sensor on a back window. If this window is opened or broken, the one-shot chip on the board will register a state change, and send a reset pulse to the ESP chip to wake it from sleep. Once awake, the ESP will check the state of a pin monitoring the sensor. If the sensor is triggered, in this example an open circuit, then it will write a ‘triggered’ value to memory and attempt to connect to WiFi. This memory writing is done so that if the chip is reset while attempting to connect to WiFi, for instance if the entry sensor was on a door and it was closed seconds later, a ‘triggered’ message will still be sent when the ESP is able to connect.

After connecting to WiFi, the ESP publishes the ‘triggered’ message to a specific MQTT topic designated to that sensor. The user names the topic when setting up the system, and is used to differentiate between sensors. Once the message is sent, the board will clear the triggered state from memory and return to sleep. One can easily monitor if ‘triggered’ messages have been sent from any of the sensors using the mobile application. If, as in this scenario with a broken window, the sensor remains open after sending the initial ‘triggered’ message, it will wake up from sleep every 30 seconds and send another ‘triggered’ message. The value was set at this mainly for testing purposes and can be increased if used in a home. This sleep is used to conserve battery power to allow longer periods of use without charging. When the battery is low, a wall-charger can be plugged into the charging port, and a red LED indicates that the battery is charging. When the LED is turned off, the battery is fully charged.

## **3.2 System Block Diagram**

Figure 6 represents the overall system broken into subsystems. The subsystems will be described in the next section.



Figure 6. Overall System Block Diagram

The team designed their own board in Eagle that acts as the controller for the sensor, which is shown in Figure 1. This board translates inputs from the sensors, converts them to trigger messages to sent to the app via MQTT. The board uses an ESP8266 E-12 chip as its microcontroller. The standard Arduino interface is used to program the board after manufacturing. The programmer uses a USB to UART converter to connect to the computer, while the board has header pins on its edge. Further details about the capabilities of the sensor boards will be described in the next section.

## **3.3 Detailed Operation of Hardware**

A full board schematic is included in Figure 7, with the board itself shown in Figure 1, and each subsystem is examined here. Succinctly, the battery provides power, that is regulated by the TPS converter, managed by by MCP charger, that runs the ESP-12E chip, that is controlled and reset by the LVC one-shot through logic chips.

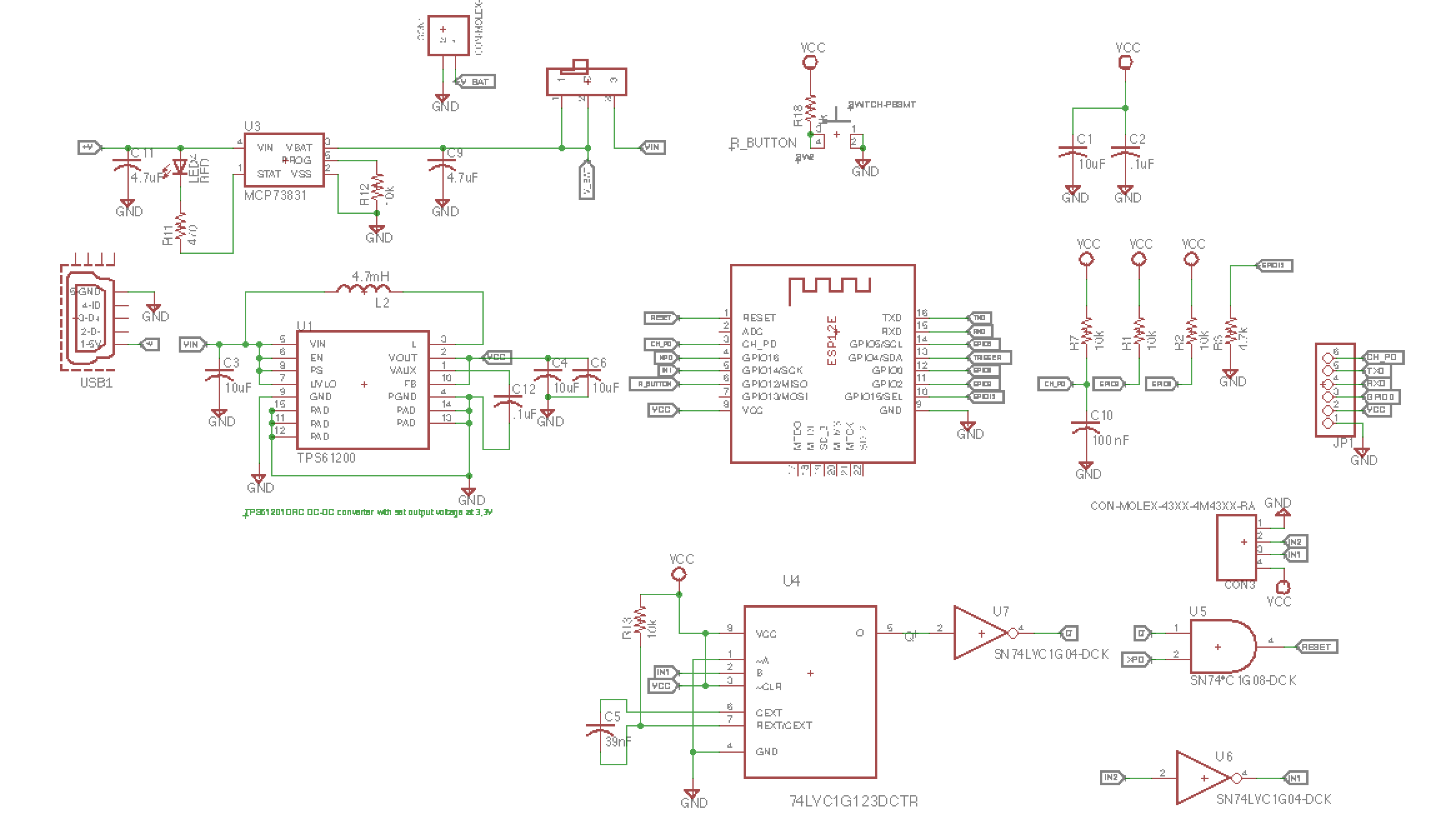


Figure 7. W.A.S.T.E. Board Schematic

The boards can use a series of 3.7 volt Lithium-ion batteries rated for 1200-2000 mAh for power. The battery can be affixed to the back of the board, and is the sole power source for the device. The batteries were tested to be closer to 4.1-4.3V maximum, decreasing to 3.8-3.9V after several hours of constant operation.

Power from the battery is regulated by a TPS61201, a low voltage synchronous boost converter optimized for 3.3V. The datasheet is included in [Section 8](#_qqvp30od4s1l), but the application circuit we based our design off of is shown below in Figure 8. We had little trouble with the TPS, its robust design was easy to solder and produced a reliable 3.3V consistently.

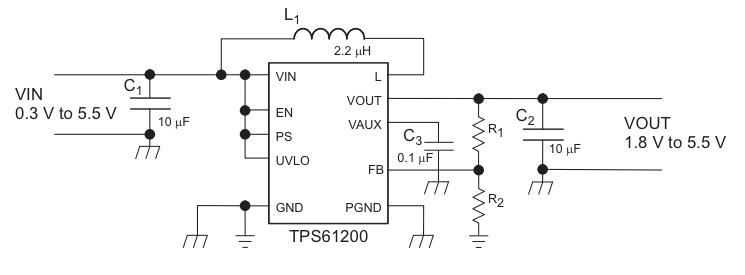


Figure 8. TPS61201 Typical Application

The MCP73831 Li-ion battery charge management controller controls the charging of the battery. Figure 9 depicts the layout of the MCP charging system. An external micro-USB plug allows the board to accept 5V externally, which is fed to the MCP. When the battery is charging, the red LED indicator will turn on, showing that the battery voltage is less than 4V. When charged, STAT on the MCP is pulled high, turning the LED off.

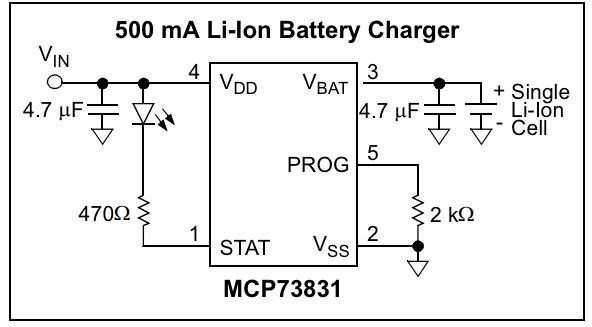


Figure 9. MCP73831 Typical Application

The sensor boards utilize the ESP8266-12E microcontroller, which has a PCB antenna, 5MB of memory, and built-in Wi-Fi capabilities, seen below in Figure 10. This chip is the core and brain of our board, and is designed completely around it. Detailed schematics and pinouts can be found in [Section 8](#_qqvp30od4s1l). This device was programmed using an external FTDI chip and the Arduino IDE. This package in particular was chosen for its PCB antenna, which is difficult to reproduce accurately. The RST pin was used to wake the device from ‘Deep Sleep’, and several I/O pins were used to monitor external triggers. In essence, this device is a compact, intelligent microcontroller, that was incredibly easy to understand and work with.

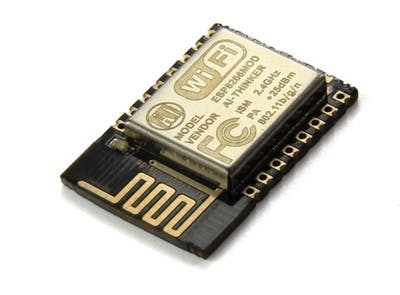


Figure 10. ESP8266-12E

The 74LVC1G123 single retriggerable monostable multivibrator is a ‘one-shot’ that takes a state change at its input, and outputs a pulse of a programmable length, depending on an external RC time constant. This chip was necessary to produce the logic low pulse for 10-20ms to reset the ESP chip. Initial design included a 10kΩ resistor and 100pF capacitor, though it was determined through testing with the logic analyzer that this pulse was not enough to reset the ESP12 chip. The 100pF capacitor was exchanged for a 39nF capacitor. Figure 11 shows the logic inputs/outputs of the one-shot chip. The final board design used an input on B, while !A was pulled low to ground and !CLR was pulled high to VCC.

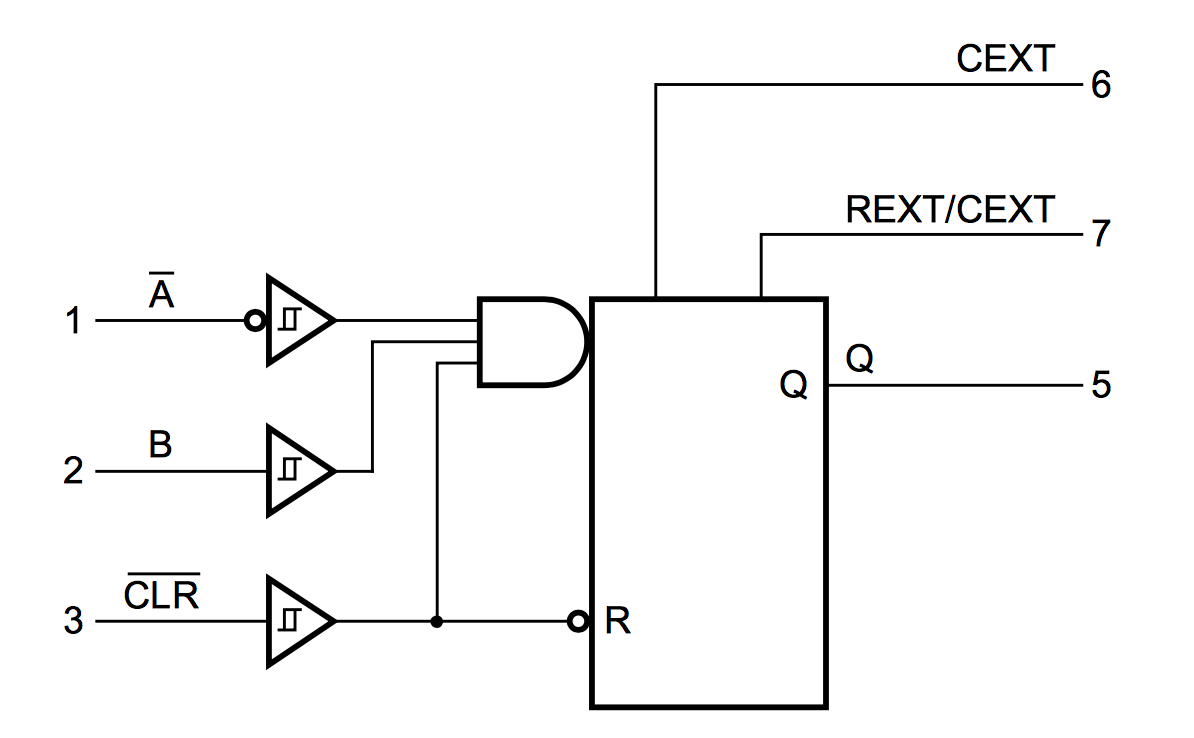
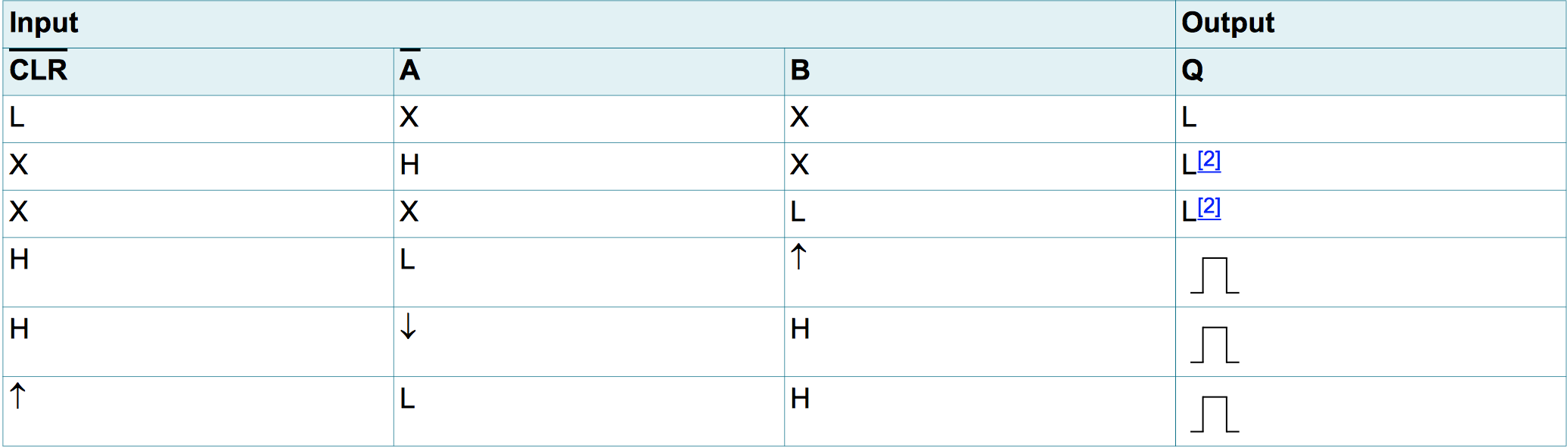


Figure 11. Function Diagram of One-Shot

Other logic was used, inverters and an AND gate, to accommodate various sensors. The inclusion of an inverter allowed for a single board design for multiple sensors. The two sensors we used, door and motion, had different voltage inputs. The door sensor, seen in Figure 3, was a simple switch, but we found most success when we held the switch high when it was closed and pulled it low when the circuit was open. The motion sensor, shown in Figure 4, was connected to both power at 5V and ground. In order to use the motion sensor with the designed board, where the max voltage was 3.3V, we removed the voltage regulator on the sensor and hard wired the voltage input to what should have been the output of the voltage regulator, as the sensor only uses 3.3V to run, though it was designed to accept higher voltages. When the motion sensor detected movement, the voltage was pulled high.

Because the pulses from each input was different, an inverter was included to fix this problem. We inverted the input from the magnetic door switch so that it was a low to high pulse and was the same input into the one shot. Table 1 is an taken from the one-shot data sheet. It explains how to create a pulse from the chip, based on different input variables. With only a single one-shot on the board, it was necessary that the input from the sensors be the same.

Table 1: One Shot Inputs to Outputs

## **3.4 Detailed Operation of Software**

The code implemented on the sensor boards is identical regardless of the type of sensor. This is because the way the boards were designed, in order to make one universal board for all sensor types.

In order to preserve battery, the ESP8266 is in a deep sleep for the majority of the time. Only a hardware signal can wake up the chip out of this state. The chip is set up in such a way to wake up for every alarm, and also wake up every 30 seconds. Upon waking up, the system runs from start up every time, regardless of the type of pulse that woke it up.

To account for this, the first thing the program does is check to see if the sensor has been tripped. It does this first to minimize the chance of the sensor being tripped but not being read fast enough. If the sensor was tripped, it saves this information in non-volatile memory as enabled by the EEPROM library. This allows it to retain information through multiple resets.

The next check it performs is a check to see if the reset button is pushed. If it is, the chip waits 10 seconds to ensure the reset is desired, then checks the button again. If it is still pushed, then the chip runs through a routine to clear all of the non-volatile memory it has. This performs a factory reset of the sensor board.

The program reads the non-volatile memory to see if there is Wi-Fi and MQTT information stored. If so, then the ESP8266 connects to the Wi-Fi network using the stored information. If not, it sets up an AP server with automatic redirect. The user connects to the network set up by the chip, named “WASTESensor,” and inputs their Wi-Fi and MQTT information. When this is submitted, the chip saves it to specific places in the non-volatile memory, and connects to the specified Wi-Fi network.

Once connected, the program reads the EEPROM to see if the triggered flag is set to true. If it is, it connects as a client to the MQTT server and publishes a message to the topic defined by the user that says it has been triggered. It then resets the triggered flag to zero, or logical false, and saves it to memory. If it has not been triggered, it connects and publishes a message that it is secure.

After it publishes a message, the program gracefully disconnects from both the MQTT server and the Wi-Fi network. The chip then sets a background timer for 30 seconds and goes into deep sleep mode to conserve battery.

This system flow is shown in Figure 12.

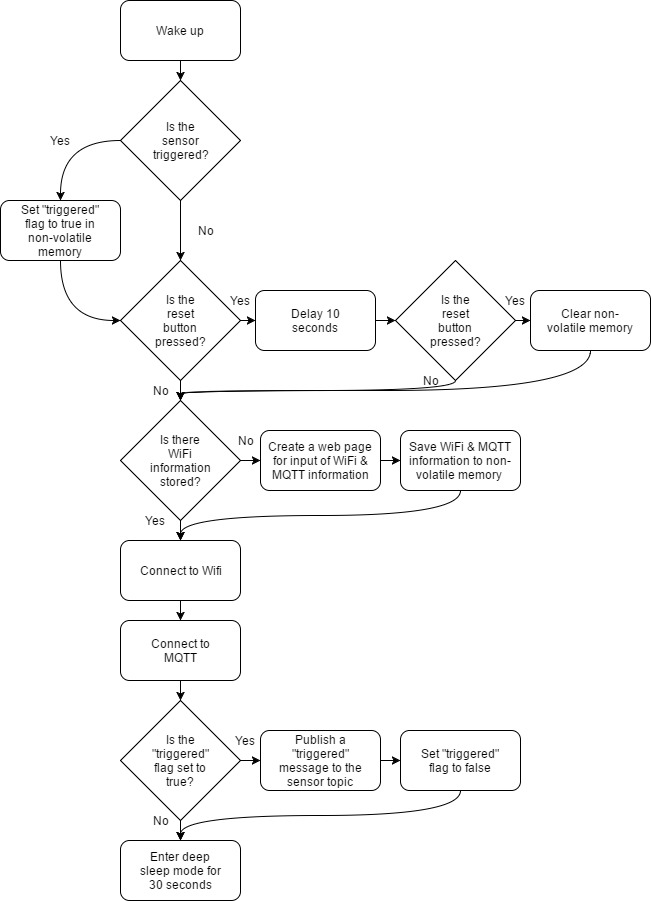


Figure 12: Sensor Board Code Flowchart

## **3.5 Detailed Operation of Mobile App**

The mobile application is written in Swift using Xcode and is only compatible with iPhone 6 and newer models. The code for the app can be accessed on [GitHub](https://github.com/mnawalan/WASTE_SD). Simply download the files from the ‘master’ branch and run the ‘3-20-client.xcworkspace’ file from Xcode on a Mac computer. Figure 13 shows the steps required to download the app from the internet.



Figure 13. GitHub Screenshot for Code Download

The main reason the app is only available for the iPhone 6 and newer is because the app is secured using fingerprint login. Older phone models do not have fingerprint login capability. The app will not crash if it is used on an older model. An error will be presented to notify the user that the device does not have fingerprint login functionality.

The app uses a pre-existing MQTT client written in swift called [Moscapsule](https://github.com/flightonary/Moscapsule). The framework is implemented as a wrapper of Mosquitto library, an open source message broker that implements MQTT. Moscapsule can be installed as a ‘CocoaPod’, a dependency manager for Swift and Objective-C. The following process is used to install Moscapsule as a CooaPod:

* In terminal:
  + touch ~/.bash\_profile; open ~/.bash\_profile
  + Edit to:

export GEM\_HOME=$HOME/.gem

export PATH=$GEM\_HOME/ruby/2.0.0/bin:$PATH

* + source ~/.bash\_profile
  + gem install cocoapods --user-install
  + pod setup --verbose
  + Cd to project directory in terminal
  + Pod init
  + Pod install
  + Open -a Xcode Podfile

Example Podfile:

# Uncomment the next line to define a global platform for your project

# platform :ios, '9.0'

use\_frameworks!

target 'WasteTiles' do

# Comment the next line if you're not using Swift and don't want to use dynamic frameworks

use\_frameworks!

# Pods for WasteTiles

pod 'Moscapsule', :git => 'https://github.com/flightonary/Moscapsule.git', :branch => 'swift2'

pod 'OpenSSL-Universal', '~> 1.0.1.18'

target 'WasteTilesTests' do

inherit! :search\_paths

# Pods for testing

end

target 'WasteTilesUITests' do

inherit! :search\_paths

# Pods for testing

end

end

The app utilizes ‘NSUserDefaults’ to save information unique to each user. ‘NSUserDefaults’ is a built in tool provided by Apple. Unless the app is completely deleted from the phone, it will save information even after the app is ‘killed’. The app will save the user’s sensors and MQTT server information. Each time the user edits the list of sensors or the MQTT information, the User Defaults are updated.

Each sensor object has the following properties: name, status, last updated time, and image. The status is displayed as ‘Waiting’ when first created until a message is received. When a user adds a sensor, the sensor object subscribes to the topic matching its name. When the user attempts to save a new sensor, the app checks to make sure the name of the new sensor does not match the name of already existing sensor. If it does, an alert will display to indicate to the user that the name needs to be changed.

Notice in the product setup that the user is told that it is very important that the sensor is named the same thing in the app and the Wi-Fi manager. When a message is successfully received, the app checks which topic the message came from. Once the specific sensor is determined, the app changes the sensor’s status property to the received message and the last updated time property to the time the message was received. This information is updated in User Defaults and then updated in the table on the app. The user can also remove sensors from their list in the app. Once removed, the app unsubscribes from that topic. The user can change the MQTT server information at any time from the settings button in the app. Once updated, the app makes a new connection to that server and saves the information in User Defaults.

For the project demo, an [AWS server](http://senior-mqtt.esc.nd.edu) was used to store MQTT messages. The benefit of using a server is that the app and the sensors don’t have to be on the same Wi-Fi network to receive status updates.

This system flow is shown in Figure 13.

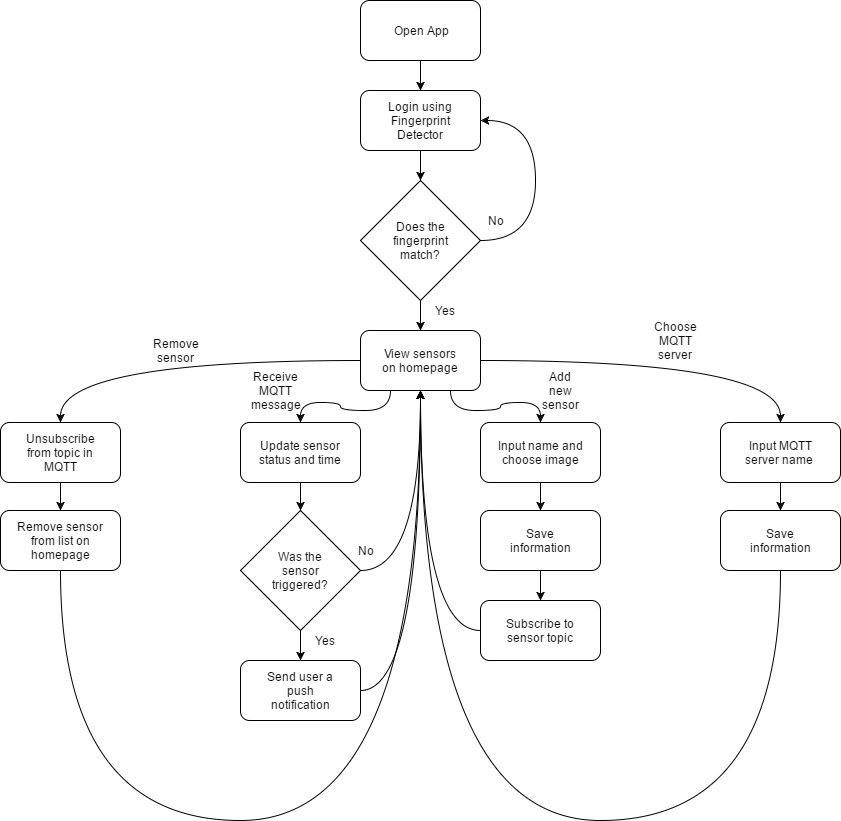


Figure 13: Mobile App Flowchart

## **3.6 Interfaces**

The product mainly uses Message Queue Telemetry Transport (MQTT) protocol to communicate between the sensor board and the mobile application. MQTT was designed to be a lightweight, reliable protocol in situations with limited bandwidth and spotty connections. The crux of the protocol is the message broker, a service that is running all of the time on a server. For W.A.S.T.E., this broker lives on an Amazon Web Services server. By housing the broker on an AWS server, it allows the user to receive updates on the status of their house regardless of what network they are on, even on cellular networks. The broker determines what messages are sent to what users. The way it does this is by the topic system.

A given client can connect to the broker, and subscribe to the topic, e.g. “Front Door,” and receive all messages sent to the topic “Front Door.” Additionally, it can publish to whatever topic it wants. The namesake of the MQTT protocol is message queueing, where if a given client is subscribed to a topic, but unexpectedly drops connection, then the broker will retain all of the messages sent to that topic, and forward them to the client when it reconnects. In the product, MQTT is utilized at quality-of-service 1, which means that every message will be delivered at least once.

# 

# 

# **4 System Integration Testing**

## **4.1 Description of Subsystem Testing**

To test the system, the sensors were manually triggered (i.e. magnetic sensors closed/open, motion) while monitoring MQTT activity on [Windows MQTT client TT3](https://github.com/francoisvdm/TT3) and the mobile app. By using the reliable Windows client, the functionality of the sensors and the app could both be confirmed. The messages that were published by the sensors should be visible on TT3 and the app if the system is working correctly.

Using a serial interface while connecting the sensors to the Wi-Fi Network confirmed connection or failure. More simply, if the sensor connected successfully, the sensor should no longer appear as a network to phones or computers.

To test the capability to expand the system, sensors without network information were connected to the network via the Wi-Fi Manager and named with a unique identifier. The same identifier was used to add a sensor to the app. The sensor’s topic matches their given name. TT3 was once again used to confirm that the sensor and app were functioning correctly. This confirms the design requirement to allow the user to expand their security system by adding sensors. New sensors were added on the app with the same name as an existing sensor. An alert appears on the app to ask the user to change the name of the new sensor. This prevents a sensor from publishing to the same topic as another sensor. The user also has the ability to edit the name of an already existing sensor. To test this, the name was changed in the app. The sensor device itself was reset to reconfigure the name. Once reconfigured, the sensor was triggered. TT3 and the app both displayed the updated status of the sensor.

Furthermore, on reset, the sensors should reappear as an available network on a computer or phone. Upon connecting to that sensor, the Wi-Fi Manager screen would reappear. In the event that the user’s Wi-Fi network or password changes, the same sensors could be reused with a simple reset and reconfiguration via the Wi-Fi network. This was tested by resetting the sensor, connecting to SDNet, and confirming functionality by receiving status updates on the app.

The app itself serves as a test for the timed MQTT publish on the sensor. The user’s sensors in the app each have a text field that displays when the last status message was received. If the magnetic sensor is held open, a status message will be published every 30 seconds. If this is not the case, the magnetic sensor is not working correctly.

The serial monitor and TT3 were used to verify performance of the system. Additionally, the blue LED on the ESP12 board would blink when the sensor was reset. When a sensor was triggered to wake up from sleep, the group would look for the LED to blink, check the serial monitor for print statements indicating the sensor was awake and attempting to connect to the internet, and monitoring TT3 to make sure the sensor published a message to the server. Once the app was developed, the app could act as a check for correct performance by looking for sensor updates. The same process was used to verify the sensor could wake up via a timer. TT3 publishes the time of each message received. Therefore, the group could verify the time between messages was consistent with the timer interval decided in the software.

Lastly, the device could be powered solely from the battery. This was confirmed by removing the usb power and only running the system on battery power. The charging functionality was tested by using a charging indicator LED. When charging, the LED would light up.

## **4.2 Testing and Requirements**

The design requirements are very straightforward. The goal was to provide a Wi-Fi home alarm system with a mobile app functioning as the user interface. The testing of the mobile app showed that the user could adequately monitor the state of their system. The user would be able to view each sensor’s status and the time of the last update. The design was also meant to accommodate changes in the system. Testing showed the user has the ability to add and remove sensors from monitoring and change the MQTT server information. Each sensor’s reset and Wi-Fi configuration functionality was tested and confirmed to be working. This allows the user to use their system on any Wi-Fi network and purchase additional sensors to expand their security. Additionally, an important component of the design is the battery. Through testing, the group confirmed that each board could be powered solely using the battery and that the circuit supported battery charging.

# **5 User Manual/Installation** M**anual**

## **5.1 How to install your product**

Download the app to your mobile device (only compatible with iOS mobile devices, excluding iPads).

In addition, the sensor boards will have to be installed close to the location to be monitored, with the sensors themselves securely mounted to the door, window, or wall.

* For the door and window sensors, secure the wired side of the sensor to the frame of the aperture, with the screw holes facing away from the opening. The unwired piece of the sensor should be mounted to the window or door, with the screw holes facing away from the opposite side of the sensor. Make sure that the two sides of the sensor are no more than 5 centimeters apart.
* For the occupancy (infrared) sensor, place the sensor in the upper corner of the room to be monitored. The dome should be facing outward to see as much of the room as possible. The sensor should be mounted to the wall as securely as possible.

## **5.2 How to setup your product**

1. Use the Molex connector to attach each sensor to its respective board and microcontroller.
2. Plug-in one sensor board to the battery. Confirm the Blue LED blinks. If not, see troubleshooting.
3. On any Wi-Fi enabled device, go to available networks and look for the network named “WASTESensor.”
   1. For example, on an iPhone navigate to Settings->Wi-Fi. Look for the sensor as an available network, as seen in Figure 14.



Figure 14. WiFi Setup for Board

1. Choose to connect to the sensor’s network
2. Upon connection, the Wi-Fi Manager, shown in Figure 15 will be displayed. If it is not, refer to troubleshooting.

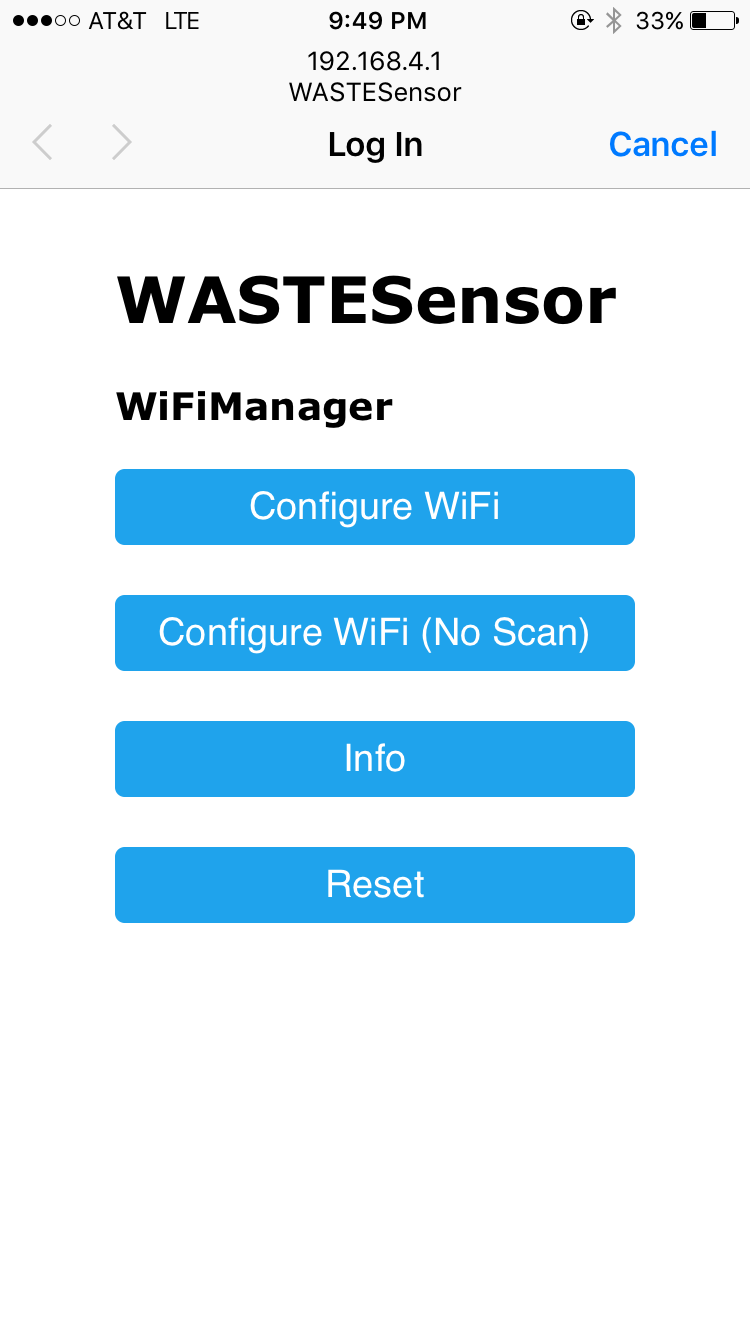


Figure 15. WiFi Manager

1. Enter your Network SSID name and password into the appropriate fields. Also input the MQTT server and sensor name. It is very important that the sensor name you choose matches the sensor name you add to the mobile app later in the setup. It is space and case-sensitive. Figure 16 shows an example of the inputs in the WiFi Manager page.

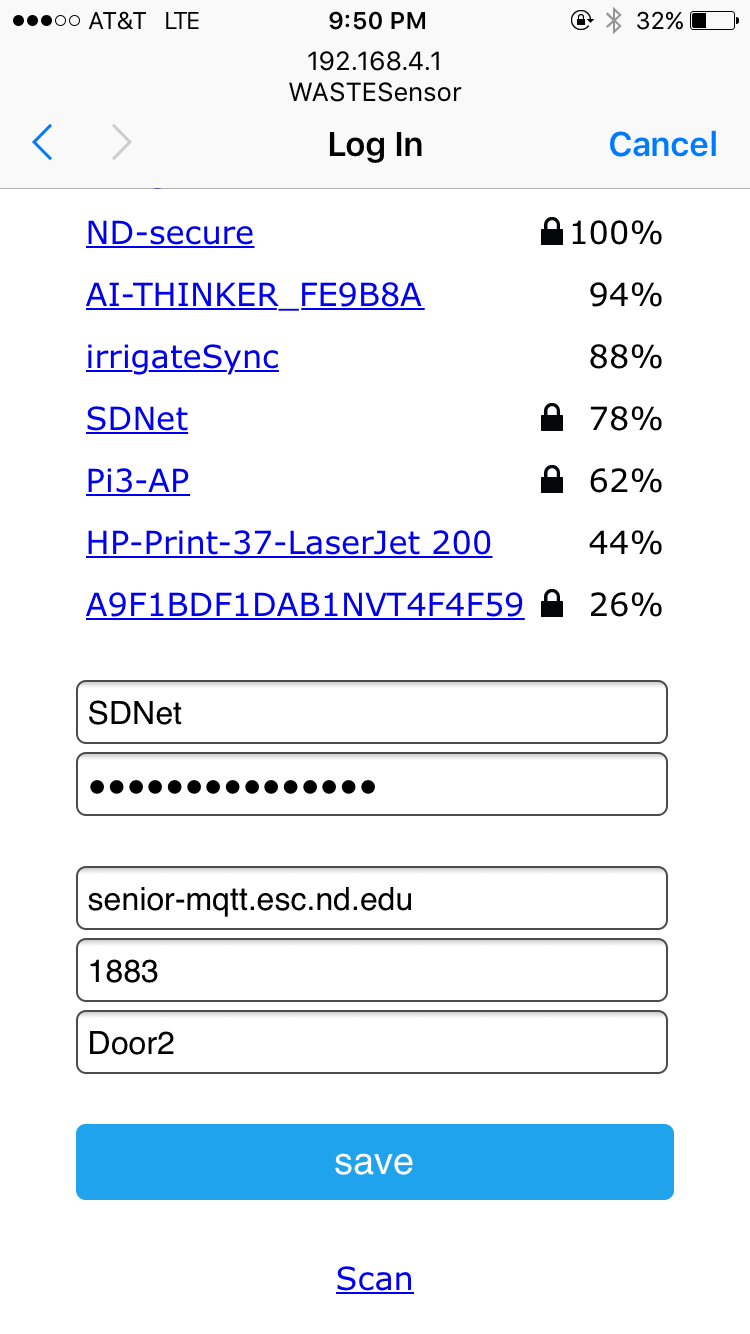


Figure 16. Inputs of WiFi Manager Page

1. Verify that the sensor connects to the network by confirming that the sensor no longer appears as an available network.
2. Open and login into the app via fingerprint login, seen in Figure 17.

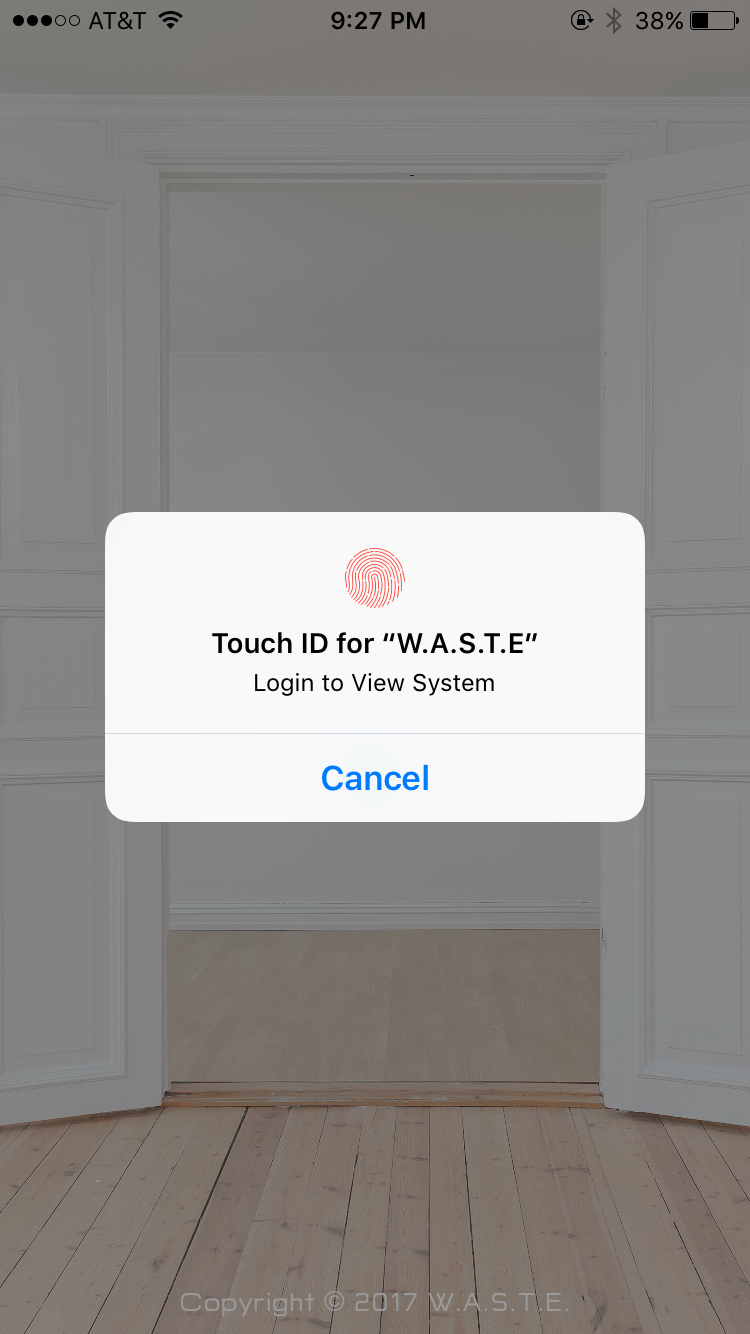
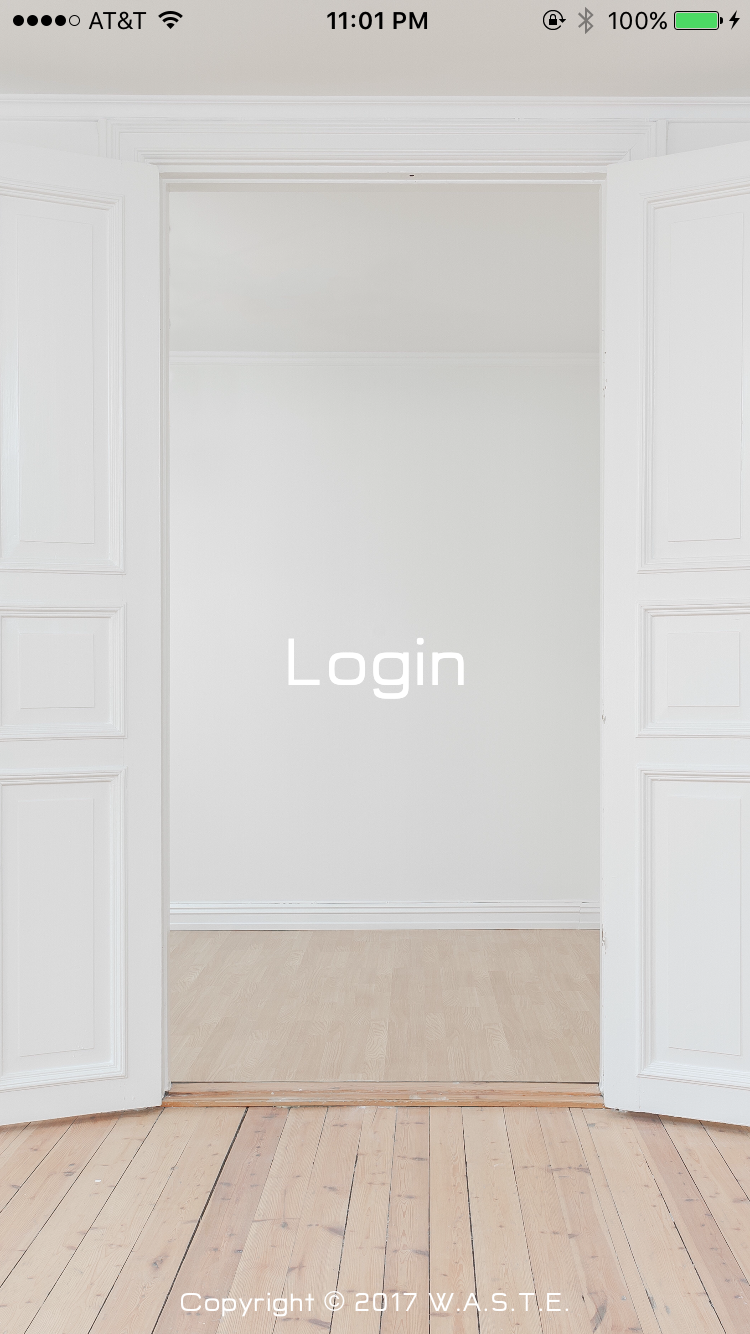


Figure 17. App Login Scrren

1. Choose ‘Add New Sensor’ in the Sensors table. See Figure 18.

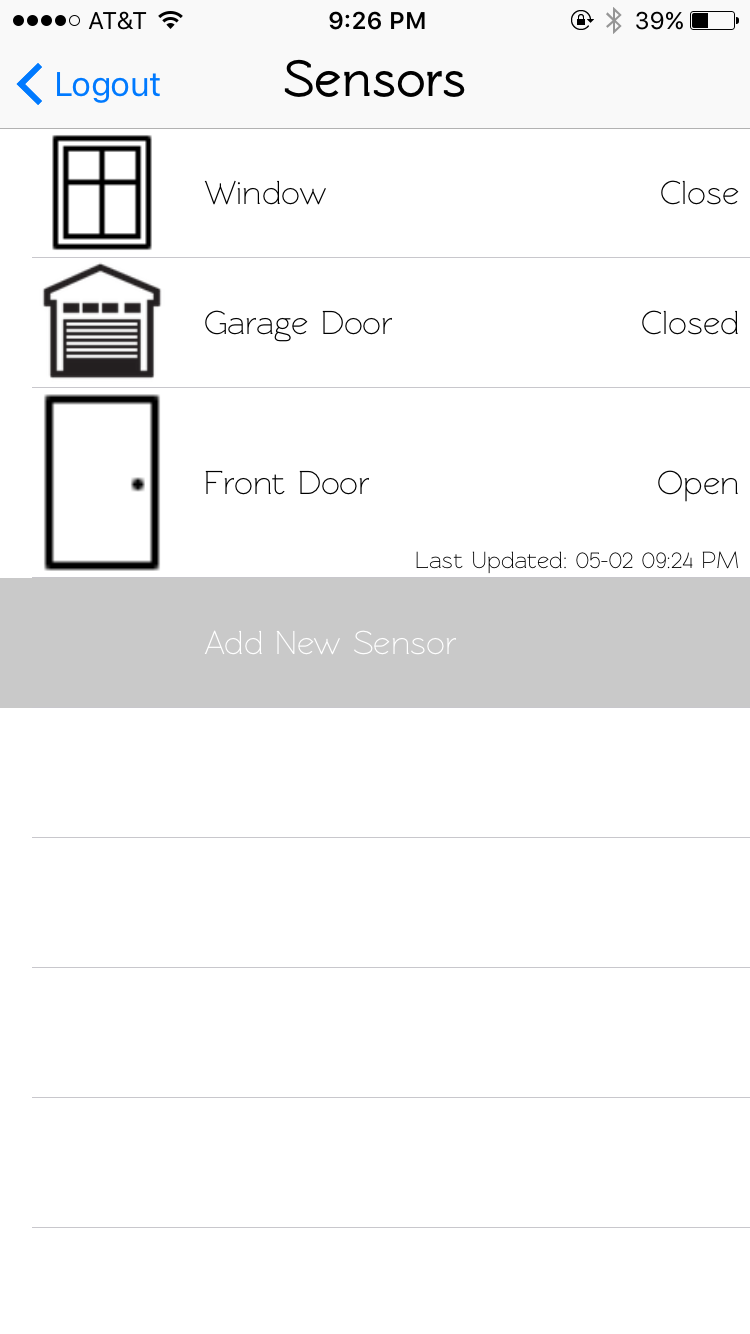


Figure 18. Add New Sensor to App

1. Name the sensor with the same name used in the Wi-Fi manager, in the box shown in Figure 19. Choose a default icon or an icon from your personal library. The app will ask to access your photo library. Confirm if you want to use personal pictures. Save the sensor using the button in the upper right corner.

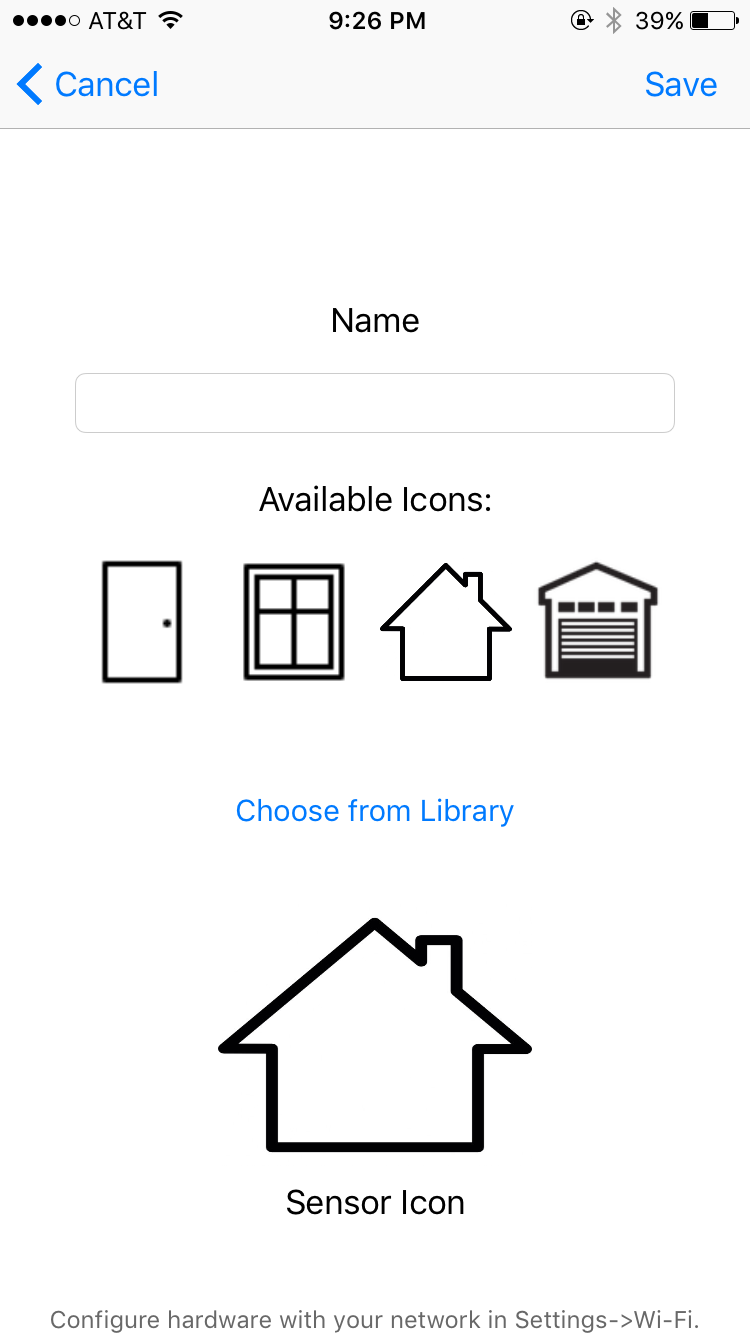


Figure 19. Name New Sensor

1. Manually trigger the sensor and monitor the sensor status in the app to confirm functionality, as shown in Figure 20.



Figure 20. Check Sensor Status

12. Repeat steps 1-11 for each sensor.

## **5.3 How the user can tell if the product is working**

* Test general status updates:
  + Force a trigger on the sensor. The status of the corresponding sensor should update in the app.
* Test timed updates:
  + Leave a window/door open. While held open, the app should update the status of that sensor every 30 seconds (select that sensor row to view updated time).
* Test sensor connectivity:
  + After resetting the sensor or in preparation to connect a new sensor, select the sensor’s network in available Wi-Fi networks on your phone or computer. The network should not be available after successful connection.
* Test power to board:
  + Plug in the sensor board to power. The blue LED blinks if successfully powered.
* Fingerprint login:
  + App should navigate to list of sensors after opening with fingerprint login.

## **5.4 How the user can troubleshoot the product**

* Reset router + sensor connections
  + Unplug device and hold reset for 15 seconds while turning it back on
  + Reconnect to network/connect to new network
* Replace batteries/charge device
* Delete the app
  + This will delete the user’s saved sensors so those will have to be re-added.
  + If the app does not appear to be updating the sensor’s status, the user can download a MQTT client of choice and subscribe to the sensors.
  + If the app is working correctly but the sensor is not publishing, reset the sensor and reconfigure the Wi-Fi information. Confirm that the name given to the sensor in the Wi-Fi Manager matches the name in the app. (Name is editable by swiping left on the sensor row).
* Fingerprint login
  + If the fingerprint login is not functioning correctly, navigate to Settings->Touch ID & Passcode and ensure that at least one fingerprint has been added.
* Sensor not connecting to WiFi
  + Confirm the network name and password are correct.
* Wifi manager web page does not show up when connecting to the network.
  + Connect to the network named WASTESensor
  + Open a web browser and enter “192.168.4.1” into the address bar and hit enter.

# 

# 

# **6 To-Market Design Changes**

Due to budget and time limitations, the presented product lacks features that would ideally be available in the product available for purchase. The following information describes changes to the product before it would be made commercially available.

## **6.1 Hardware/Software Enhancements**

In the future, we could alter the setup function to allow the user to define the Wi-Fi network and password without individually programming each sensor. We could also use Bluetooth to connect the sensors to the base station, providing a more secure connection between the base station and the sensors. The sensor boards could be updated and remanufactured to have no external wires. In addition, the battery performance could be analyzed to determine the average lifetime of the product in the field. More possibilities include a way the sensors can communicate that they have a low battery, and a way to contact law enforcement directly with a time, location, and method of entry. In addition, we could include a sensor to detect if the window was broken. We could add temperature and lighting control via the application, while also making it iPad compatible. Finally, we could create an aesthetically-pleasing and functional casing and packaging for the base system and sensors.

Over the air updates would have to be separate from the app but could be executed via a user’s computer. Over the air updates would allow the user to reset Wi-Fi network configuration without resetting each individual chip. If a user wanted to rename a sensor in the app, an over the air update could be done on the computer to update the sensor information. Ideally, the product would allow USB updates or OTA updates. By providing USB update capability, there would be less limitations on update size and wouldn’t have to rely on availability or strength of WiFi. In addition, many more upgrades could be made in the security of the transmission.

Most importantly, the security of our actual security system would need to be improved before going to market. A third-party would only need to know the MQTT server address and the sensor topics. MQTT has many security features built in, and there could be username or IP address verification to ensure that it is the sensor sending the message, and not an external threat.

## **6.2 Mobile App Future Enhancements**

Apple requires a developer membership at the cost of $99/year to use their Push Notification Service. Due to the timeline of this project, the membership was not purchased. Therefore, push notifications could not be utilized. In the future, the app would make use of push notifications to alert the user when the status of a sensor changed. For a more robust user-interface, the sensor could publish an MQTT message when the battery was low to warn the user. A future app would also display the status of the connection to the MQTT server. In the event that the connection was lost, the user would be notified.

The ideal To-Market Product would allow the user to create a username and password instead of only having the option of fingerprint login. Additionally, the app would have the ability to request information from the sensor for troubleshooting and robustness purposes.

# **7 Conclusions**

The final presentation of W.A.S.T.E. will have a functioning home security system with several different types of sensors to alert a homeowner when one of the sensors is triggered so that they have security of mind in terms of their house. The project will require application development and Wi-Fi signaling from microcontrollers. Once combined, the user should have a comprehensive alert system that has several safeguards and redundancies.

Several different sources were used in designing the system and the features included. Zigbee[[3]](#footnote-2) and SimpliSafe[[4]](#footnote-3) are Wi-Fi home security systems that this system is modeled after. Other sources used were IEEE[[5]](#footnote-4) and Xbee remote sensors[[6]](#footnote-5) in designing this project.

W.A.S.T.E. the Wi-Fi Alarm System Trusted Environment seeks to provide peace of mind and security to users in their daily life.

# 

# 

# **8 Appendices**

Complete hardware schematics

Complete Software listings

Relevant parts or component data sheets (do NOT include the data sheets for the microcontroller or other huge files but give good links to where they may be found.)

Name of files: upload to website

Sparkfun ThingDev

[Hardware](https://learn.sparkfun.com/tutorials/esp8266-thing-development-board-hookup-guide/hardware-overview)

[Pinout](https://cdn.sparkfun.com/datasheets/Wireless/WiFi/ESP8266ThingDevV1.pdf)

[ESP Programming Example](http://www.whatimade.today/esp8266-easiest-way-to-program-so-far/)

PIR sensor

[Datasheet](https://www.mpja.com/download/31227sc.pdf)

[How to use](https://learn.adafruit.com/pir-passive-infrared-proximity-motion-sensor/testing-a-pir)

ESP8266

[Datasheet](http://download.arduino.org/products/UNOWIFI/0A-ESP8266-Datasheet-EN-v4.3.pdf)

[12-E](https://mintbox.in/media/esp-12e.pdf)

MQTT

[Installing MQTT on Pi with Websockets](http://blog.thingstud.io/recipes/how-to-make-your-raspberry-pi-the-ultimate-iot-hub/)

[cocoaPod](https://github.com/flightonary/Moscapsule)

[AWS client](http://docs.aws.amazon.com/iot/latest/developerguide/view-mqtt-messages.html)

[AWS with Swift Example](https://github.com/awslabs/aws-sdk-ios-samples/tree/master/IoT-Sample/Swift)

[Examples](http://www.hivemq.com/blog/MQTT-essentials/)

Hardware

[TPS DC-DC Converter](http://www.ti.com/lit/ds/symlink/tps61200.pdf)

[MCP Battery Charger](https://cdn.sparkfun.com/datasheets/Dev/Arduino/Shields/MCP738312.pdf)

[LVC One Shot](http://assets.nexperia.com/documents/data-sheet/74LVC1G123.pdf)

Code Listing: SensorCode.ino

#include <FS.h> //this needs to be first, or it all crashes and burns...

// door + window switch sensor

#include <ESP8266WiFi.h> //ESP8266 Core WiFi Library

#include "PubSubClient.h"

#include <DNSServer.h> //Local DNS Server used for redirecting all requests to the configuration portal

#include <ESP8266WebServer.h> //Local WebServer used to serve the configuration portal

#include "WiFiManager.h" //WiFi Configuration Magic https://github.com/tzapu/WiFiManager

#include <EEPROM.h> //Non-Volatile memory library

char mqtt\_server[60] = "senior-mqtt.esc.nd.edu";

char mqtt\_port[6] = "1883";

char mqtt\_id[40];

void callback(const MQTT::Publish& pub) {

// handle message arrived

Serial.print("Message arrived [");

Serial.print(pub.topic());

Serial.print("] ");

Serial.println(pub.payload\_string());

Serial.println();

}// end of callback function

// Create an ESP8266 WiFiClient class to connect to the MQTT server.

WiFiClient wf\_client; // instantiate wifi client

void setup() {

Serial.begin(115200); //Begin serial communications

EEPROM.begin(512); //Initialize non-volatile memory

Serial.println();

//Initialize values

int val1 = 0;

int val2 = 0;

int triggered = 0;

pinMode(14, INPUT);

pinMode(12, INPUT);

val1 = digitalRead(14);

if (val1) { //if the sensor was triggered

triggered = 1;

Serial.print("\n --------- \n");

Serial.print("TRIGGERED");

Serial.print("\n --------- \n");

EEPROM.write(511, 0x01);

EEPROM.commit();

}

val2 = digitalRead(12);

if (!val2) { //If the reset button was pushed

delay (10000);

if (!val2) { //make sure a reset was desired

for (int i = 0 ; i < 512 ; i++) {

EEPROM.write(i, 0xFF); //clear non-volatile memory

}

EEPROM.commit();

}

}

// id/name placeholder/prompt default length

WiFiManagerParameter custom\_mqtt\_server("mqtt\_server", "mqtt server", mqtt\_server, 60);

WiFiManagerParameter custom\_mqtt\_port("mqtt\_port", "mqtt port", mqtt\_port, 6);

WiFiManagerParameter custom\_mqtt\_id("mqtt\_id", "Sensor Name", mqtt\_id, 40);

WiFiManager wifiManager;

wifiManager.setDebugOutput(false);

wifiManager.resetSettings(); //for testing

//add all your parameters here

wifiManager.addParameter(&custom\_mqtt\_server);

wifiManager.addParameter(&custom\_mqtt\_port);

wifiManager.addParameter(&custom\_mqtt\_id);

//first parameter is name of access point, second is the password

//Read SSID and Password from non-volatile memory

Serial.println("Reading EEPROM ssid");

String esid = "";

for (int i = 0; i < 32; ++i)

{

esid += char(EEPROM.read(i));

if (char(EEPROM.read(i)) == '\0')

break;

}

Serial.print("SSID: ");

Serial.println(esid);

Serial.println("Reading EEPROM pass");

String epass = "";

for (int i = 32; i < 96; ++i)

{

epass += char(EEPROM.read(i));

if (char(EEPROM.read(i)) == '\0')

break;

}

Serial.print("PASS: ");

Serial.println(epass);

Serial.println("Reading EEPROM mqtt\_server");

String mserver = "";

for (int i = 100; i < 160; ++i)

{

mserver += char(EEPROM.read(i));

if (char(EEPROM.read(i)) == '\0')

break;

}

Serial.print("MSERVER: ");

Serial.println(mserver);

Serial.println("Reading EEPROM mqtt\_port");

String mport = "";

for (int i = 162; i < 169; ++i)

{

mport += char(EEPROM.read(i));

if (char(EEPROM.read(i)) == '\0')

break;

}

Serial.print("MPORT: ");

Serial.println(mport);

Serial.println("Reading EEPROM mqtt\_id");

String mid = "";

for (int i = 170; i < 215; ++i)

{

mid += char(EEPROM.read(i));

if (char(EEPROM.read(i)) == '\0')

break;

}

Serial.print("MID: ");

Serial.println(mid);

if ((esid[0] == 0xFF) && (mserver[0] == 0xFF)) //If the SSID and server was not saved

{

wifiManager.autoConnect("WASTESensor"); //Set up web portal to collect WiFi info

strcpy(mqtt\_server, custom\_mqtt\_server.getValue());

strcpy(mqtt\_port, custom\_mqtt\_port.getValue());

strcpy(mqtt\_id, custom\_mqtt\_id.getValue());

Serial.println("Server = ");

Serial.println(mqtt\_server);

Serial.println("port = ");

Serial.println(mqtt\_port);

Serial.println("id = ");

Serial.println(mqtt\_id);

Serial.println(wifiManager.getConfigPortalSSID());

String qsid = wifiManager.\_ssid;

String qpass = wifiManager.\_pass;

Serial.println(qsid);

Serial.println(qpass);

int temp;

Serial.print("Wrote: ");

for (int i = 0; i < qsid.length(); ++i) //write SSID and Password to memory as well as MQTT

{

EEPROM.write(i, qsid[i]);

Serial.print(qsid[i]);

temp = i;

}

++temp;

EEPROM.write(temp, '\0');

Serial.println(" ");

Serial.println("writing eeprom pass:");

Serial.print("Wrote: ");

for (int i = 0; i < qpass.length(); ++i)

{

EEPROM.write(32 + i, qpass[i]);

Serial.print(qpass[i]);

temp = i;

}

++temp;

EEPROM.write(32 + temp, '\0');

Serial.println(" ");

Serial.print("Wrote: ");

for (int i = 0; i < strlen(mqtt\_server); ++i)

{

EEPROM.write(100 + i, mqtt\_server[i]);

Serial.print(mqtt\_server[i]);

temp = i;

}

++temp;

EEPROM.write(100 + temp, '\0');

Serial.println(" ");

Serial.print("Wrote: ");

for (int i = 0; i < strlen(mqtt\_port); ++i)

{

EEPROM.write(162 + i, mqtt\_port[i]);

Serial.print(mqtt\_port[i]);

temp = i;

}

++temp;

EEPROM.write(162 + temp, '\0');

Serial.println(" ");

Serial.print("Wrote: ");

for (int i = 0; i < strlen(mqtt\_id); ++i)

{

EEPROM.write(170 + i, mqtt\_id[i]);

Serial.print(mqtt\_id[i]);

temp = i;

}

++temp;

EEPROM.write(170 + temp, '\0');

Serial.println(" ");

EEPROM.commit();

mserver = String(mqtt\_server);

mport = String(mqtt\_port);

mid = String(mqtt\_id);

}

else

{

const char \* ssid = esid.c\_str();

const char \* pass = epass.c\_str();

Serial.println(ssid);

Serial.println(pass);

wifiManager.connectWifi(ssid, pass); //connect to WiFi

}

Serial.println("successful");

PubSubClient client(wf\_client, mserver, (uint16\_t)mport.toInt()); // pass to pubsub

client.set\_callback(callback);

val1 = digitalRead(14);

if (val1) { //if the sensor was triggered

triggered = 1;

EEPROM.write(511, 0x01);

EEPROM.commit();

}

triggered = EEPROM.read(511); //was the sensor triggered?

if (triggered == 1) {

Serial.println("triggered");

if (client.connect(mid)) { //connect to MQTT

Serial.println("connected");

Serial.println("Message sent to MQTT server.");

String message = "OPEN" + '\0';

client.publish(mid, message); //Publish a triggered message to topic

Serial.println("sent it");

EEPROM.write(511, 0xFF); //clear triggered flag

EEPROM.commit();

}

}

if (client.connect(mid)) { //connect to MQTT

Serial.println("connected");

Serial.println("Message sent to MQTT server.");

String message = "CLOSED" + '\0';

client.publish(mid, message); //Publish a triggered message to topic

Serial.println("sent it");

}

client.disconnect();

Serial.print("\n");

WiFi.disconnect();

Serial.print("Disconnected");

Serial.print("\n");

delay(1000);

Serial.print("Sleeping");

ESP.deepSleep(1000000 \* 30); //Enter deep sleep mode

}

void loop() {

Serial.print("in loop");

}

1. http://www.alarmsystemreport.com/home-burglaries-what-are-the-odds/ [↑](#footnote-ref-0)
2. http://www.alarm.org/HomeSafety/HardFacts.aspx [↑](#footnote-ref-1)
3. <http://www.zigbee.org/what-is-zigbee/494-2/> [↑](#footnote-ref-2)
4. <http://simplisafe.com> [↑](#footnote-ref-3)
5. <http://spectrum.ieee.org/geek-life/hands-on/build-your-own-home-security-system> [↑](#footnote-ref-4)
6. <http://hackaday.com/2011/12/18/xbee-remote-sensors-tell-you-when-someone-enters-your-home/> [↑](#footnote-ref-5)