Bisquare IoT Project

ESP8266 Based IoT Solutions

With the rise in internet-enabled devices in the market, there are only few companies in India doing R&D and providing end-to-end solutions in the IoT space. Most budget options are simply Chinese rebranded products, which are, one, unreliable, two, Bisquare wished to develop their propriety IoT solutions to address this void in the market. The intent was to create an end-to-end IoT platform, starting from developing their propriety hardware, based on ESP8266 (shown below) to building a full-stack backend, enabling an array of services

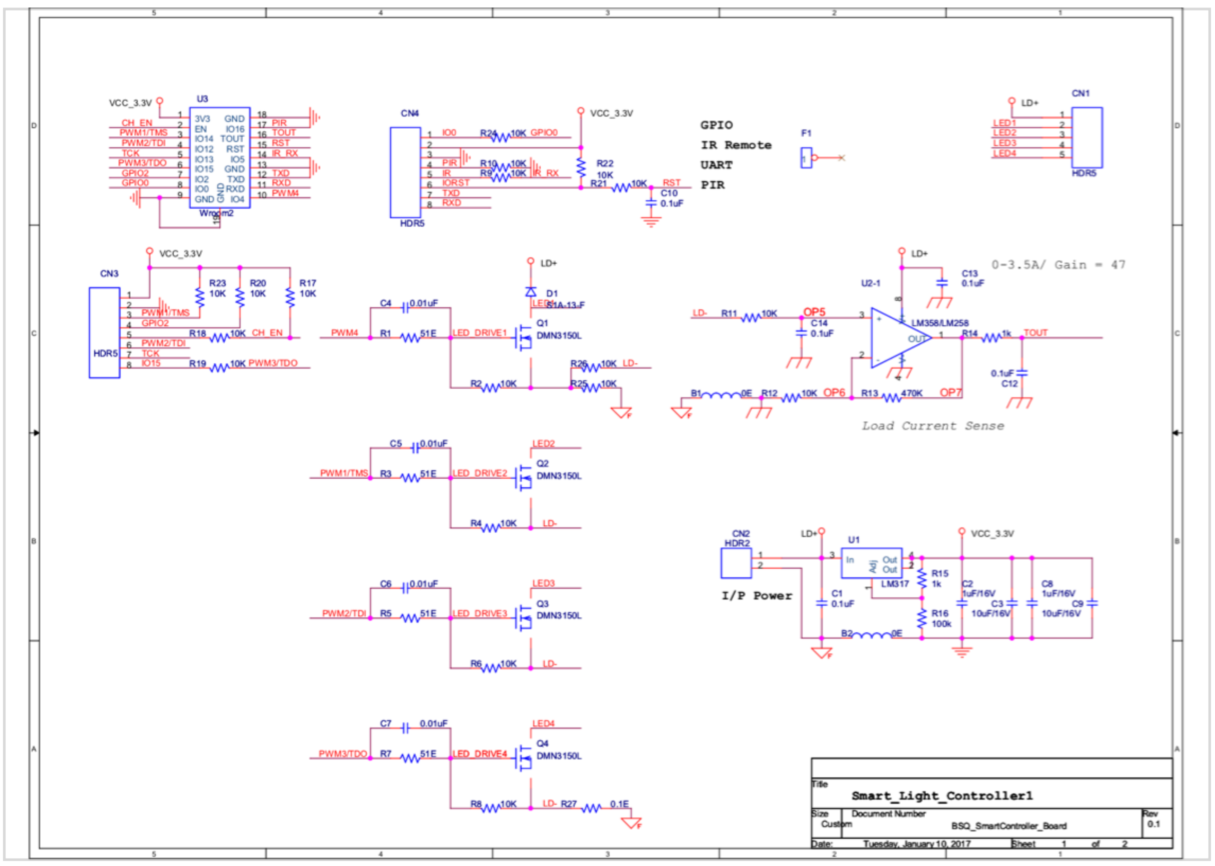
several IoT-based solutions using a proprietary ESP8266 based module (as shown below), and create an IoT platform with an array of services for home automation (with mobile apps and Voice Control).

Further, the data from the IoT modules would be used for analytics for learning usage patterns and develop smarter services like suggestive control, ambient temperature based on local weather etc.

This project report lays out the various reference material I found pertinent to the project, as well as my learnings from my time spent at Bisquare Systems.

Embedded Hardware Design

This part of the the various developmental stages of embedded hardware excites me the most. During the first couple of weeks at the internship, our goal was to design the WiFi microcontroller which would be deployed for various IoT solutions. The key here was to develop a module with using the ESP8266 microcontroller, with LED drivers, as well as support for IR interfacing through the PWM port. After a few weeks, with some help from Mr. Himendra, we created the board schematics for the module, which was sent out to be printed.



(Layout for the ESP8266-based module)

Introduction to ESP8266

The ESP8266 is a microcontroller-cum-WiFI platform, which has gain a lot of momentum in the IoT forums and communities all across the globe. This is due to the aggressive pricing of the Chinese manufacturer (these modules come at a starting cost of <$2) along with

The idea was to make a single board/module which could be purposed for all the various solutions we ideated to make (a few of them being smart street lights, AC companion using IR, motion/presence detection using PIR and remote control compatibility). It was thus essential to research and pick out the right model/specifications of the WiFi module that goes it the board. We initially settled for the ESP8266 12F, primarily because it allowed us to access many features of the ESP8266: 11 GPIO pins, one analog-to-digital converter (ADC) with a 10 bit resolution and up to 5 PWM ports. *[this particular model was discontinued later, and we swapped it out for the ESP-32, which was similarly spec'd]*

Programming the ESP8266

Using FTDI

The basic way to upload code to the ESP8266-12e in standalone mode is by connecting it to an FTDI module.

The connections are simple, but when searching on Google for “ESP8266-12E FTDI” I got many results that were either not written clearly, lacked a schematic diagram or were missing details. So I decided to clear things up a bit by making a quick lookup table for the various modes.

Booting Modes

The ESP12e module has 3 booting modes, to enter these modes the following pins need to be connected as shown:

mode	gpio0	gpio2	gpio15
Flash startup (normal)	1	1	0
UART download (programming)	0	1	0
SD-card boot	0	0	1

Programming Mode

In order to program the ESP module using an FTDI module we need to set it to boot in programming mode, and set the rest of the connections:

VCC (ESP)	3.3v (external regulator connected to the common GND)
GND (ESP)	GND (FTDI)
GPIO15 (ESP)	GND
GPIO0 (ESP)	GND
GPIO2	VCC (ESP)
TX (ESP)	RX (FTDI)
RX (ESP)	TX (FTDI)
EN (ESP)	VCC (ESP)

ESP8266 board pack for Arduino IDE

I decided to program the ESP8266 using the Arduino IDE, mostly because I was more familiar with it, and I was able to find good libraries and community support with the same.

In order to do so, I installed the ESP8266 board pack.

I found some online references that said to choose the 'ESP Generic module' and play around with the settings, but that didn't really work for me. Instead, I chose the 'NodeMCU 1.0' option, didn't touch the settings and it worked on the first try.

Connect the ESP module to the power and the FTDI module to your computer using a USB cable, open the Arduino IDE, choose your board and COM port and click upload.

Interestingly, in this configuration, the board does not reset by itself, so in order to successfully upload the code, I had to reset the moment the IDE had finished uploading the code.

To reset the board, I had to put the ESP EN pin to GND and back to VCC, this will boot the module in programming mode. We later built a dedicated reset switch jumper which could be connected to the final board/module as well.

Operation Modes

- **AP:**
  An access point (AP) is a device that provides access to Wi-Fi network to other devices (stations) and connects them further to a wired network. ESP8266 can provide similar functionality except it does not have interface to a wired network. Such mode of operation is called soft access point (soft-AP). The maximum number of stations connected to the soft-AP is five.

```
#include <ESP8266WiFi.h>        // Include the Wi-Fi library

const char* ssid     = "SSID";         // The SSID (name) of the Wi-Fi network
you want to connect to
const char* password = "PASSWORD";     // The password of the Wi-Fi network

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send
messages to the computer
  delay(10);
  Serial.println('\n');
  WiFi.begin(ssid, password);             // Connect to the network
  Serial.print("Connecting to ");
  Serial.print(ssid); Serial.println(" ...");

  int i = 0;
  while (WiFi.status() != WL_CONNECTED) { // Wait for the Wi-Fi to connect
    delay(1000);
    Serial.print(++i); Serial.print(' ');
  }

  Serial.println('\n');
  Serial.println("Connection established!");
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());         // Send the IP address of the ESP8266
to the computer
}

void loop() { }
```

- **Station Mode:**
  Station (STA) mode is used to get ESP module connected to a Wi-Fi network established by an access point.

  Station class has several features to facilitate management of Wi-Fi connection. In case the connection is lost, ESP8266 will automatically reconnect to the last used access point, once it is again available. The same happens on module reboot. This is possible since ESP is saving credentials to last used access point in flash (non-volatile) memory. Using the saved data ESP will also reconnect if sketch has been changed but code does not alter the Wi-Fi mode or credentials.

  Switching the module to Station mode is done with begin function. Typical parameters passed to begin include SSID and password, so module can connect to specific Access Point.

```

#include <ESP8266WiFi.h>        // Include the Wi-Fi library


const char *ssid = "ESP8266 Access Point"; // The name of the Wi-Fi network
```

```
that will be created
const char *password = "thereisnospoon";   // The password required to connect
to it, leave blank for an open network

void setup() {
  Serial.begin(115200);
  delay(10);
  Serial.println('\n');

  WiFi.softAP(ssid, password);             // Start the access point
  Serial.print("Access Point \"");
  Serial.print(ssid);
  Serial.println("\" started");

  Serial.print("IP address:\t");
  Serial.println(WiFi.softAPIP());         // Send the IP address of the
ESP8266 to the computer
}

void loop() { }
```

Smart Connect

At this point, we needed a better way to program the ESP module to connect to the WiFi, as on the final product, we couldn't connect to an AP by changing the SSID and password fields in the code each time. This is where the smart connect method comes to play. It is an API provided by the manufacturers (Espressif) themselves: SmartConfig – ESP-IDF Programming Guide v4.0-dev-1437-g39f090a4f documentation

The way this works is that, through the android app API, it broadcasts multiple UDP packets over WiFI to all the ESP devices configured in the smart connect mode. These packets contain the ssid and the password to the network you wish to connect the ESP device to. So even though UDP incurs high packet loss, it is made up for by sending multiple of these packets to all the devices.

This was an important step towards building a production-ready implementation, as we couldn't possibly hard-coded the ssid and password for the devices beforehand. Fortunately, implementing wasn't as complicated as it may seem. The SmartConnect API from ESP is well documented and was easy to set up on the Arduino IDE.

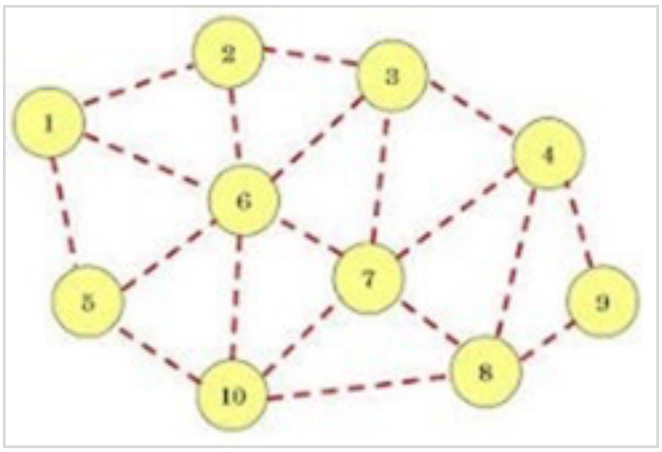
UART

I used UART Initially used to connect to WiFi AP. This allowed me a chance to kearn about I2C and synchronous and asynchronous data transfer
etc.

Mesh

This was a particularly interesting concept I learnt through this project. For variety of reasons (discussed ahead) it makes sense for each ESP8266 node to act both, as an AP to the next sequential ESP node, as well as a station to connect either to yet another ESP node, or to the router, hence creating a mesh of ESP devices. This provides two prominent advantages over a traditional star setup to the router otherwise:

- Only one device requesting DHCP to the router: which is good as many home routers can handle only as many devices on them.
- Improved Range: so if a device is placed much farther from the router, it can connect to neighboring nodes, instead of directly communicating to the router.



There is a library on Arduino IDE for implementing mesh on ESP nodes called Painless Mesh. I was able to demonstrate the concept by connecting 3 ESP devices in tandem and have them send messages to each other. I didn't incorporate this method for my final implementation (could be done for future work).

Rest APIs

This will be discussed later in detail in the AWS stack setup section. But broadly, we are sending/receiving our requests to the ESP modules through RestAPI's. This means we send specific parameters to the device embedded with the url. For ex:

http://192.168.1.172/setLED?led1=TRUE&led2=FALSE

Here we indicate the device to turn on LED 1 and turn off the LED 2.

HTTP GET Requests

Use GET requests **to retrieve resource representation/information only** – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be **safe methods**. Additionally, GET APIs should be **idempotent**, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

For any given HTTP GET API, if the resource is found on the server, then it must return HTTP response code 200 (OK) – along with the response body, which is usually either XML or JSON content (due to their platform-independent nature).'

In case resource is NOT found on server then it must return HTTP response code 404 (NOT FOUND). Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code 400 (BAD REQUEST)

HTTP Post

Use POST APIs **to create new subordinate resources**, e.g., a file is subordinate to a directory containing it or a row is subordinate to a database table. Talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources. Ideally, if a resource has been created on the origin server, the response SHOULD be HTTP response code 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a location header.

Many times, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either HTTP response code 200 (OK) or 204 (No Content) is the appropriate response status.

Responses to this method are **not cacheable**, unless the response includes appropriate Cache-Control or expires header fields.
Please note that POST is **neither safe nor idempotent**, and invoking two identical POST requests will result in two different resources containing the same information (except

resource ids).

There are more APIs, but we'd be primarily be communicating through the two discussed.

Mood Lighting

Introduction

We started off with multi-colored LED mood lighting as our first product. The WiFi module (above) is capable of driving lights up-to 40W using the MOSFETs circuits. This was designed so as to make it compatible for street lights later on. Thus, on the hardware side, the circuit was not much complicated. The three MOSFETs were mapped to 3 PWM ports of the ESP8266 (refer to Fig. 1 for schematics).

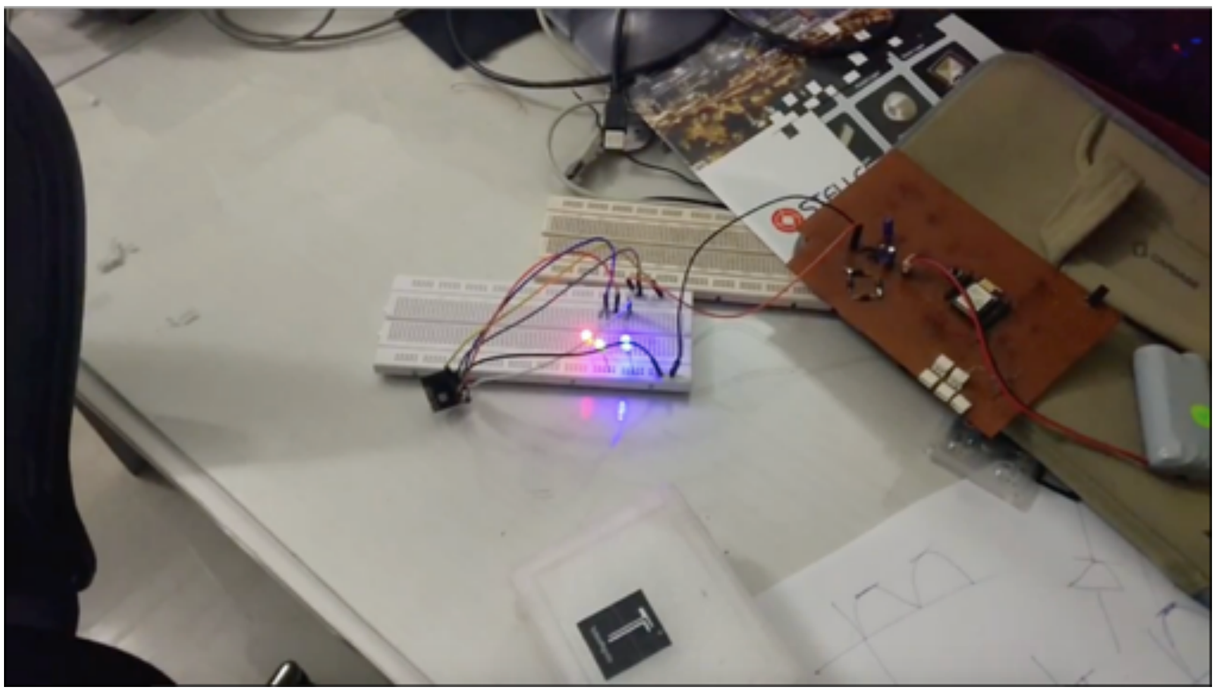
Philips Hue lights is probably the most established product in the market for smart home lights. However, during the course of the project, Philips hasn't established itself in the Indian markets, and thus this was a good opportunity to capitalize on. There are multiple brands trying to get into the market, but they do not provide end-to-end consumer solutions, and are simply rebranding Chinese products in the Indian markets.

We wished to create a complete home-automation suite, which could be controlled and monitored through an app, and would support voice assistants like Alexa and Google Home.

Prototyping

During my initial stages of working on the IoT project and getting familiar with ESP8266, I had built a basic prototype to drive 3 (RGB) LEDs through the module, which was controlled through an HTTP server. The intensities of the LEDs were adjusted according to the color selected.

For the prototype, the PWM GPIOs were directly connected with the LEDs with a series resistor. An android app was programmed to send the corresponding POST request to the local static IP (of the module)



Remote Control

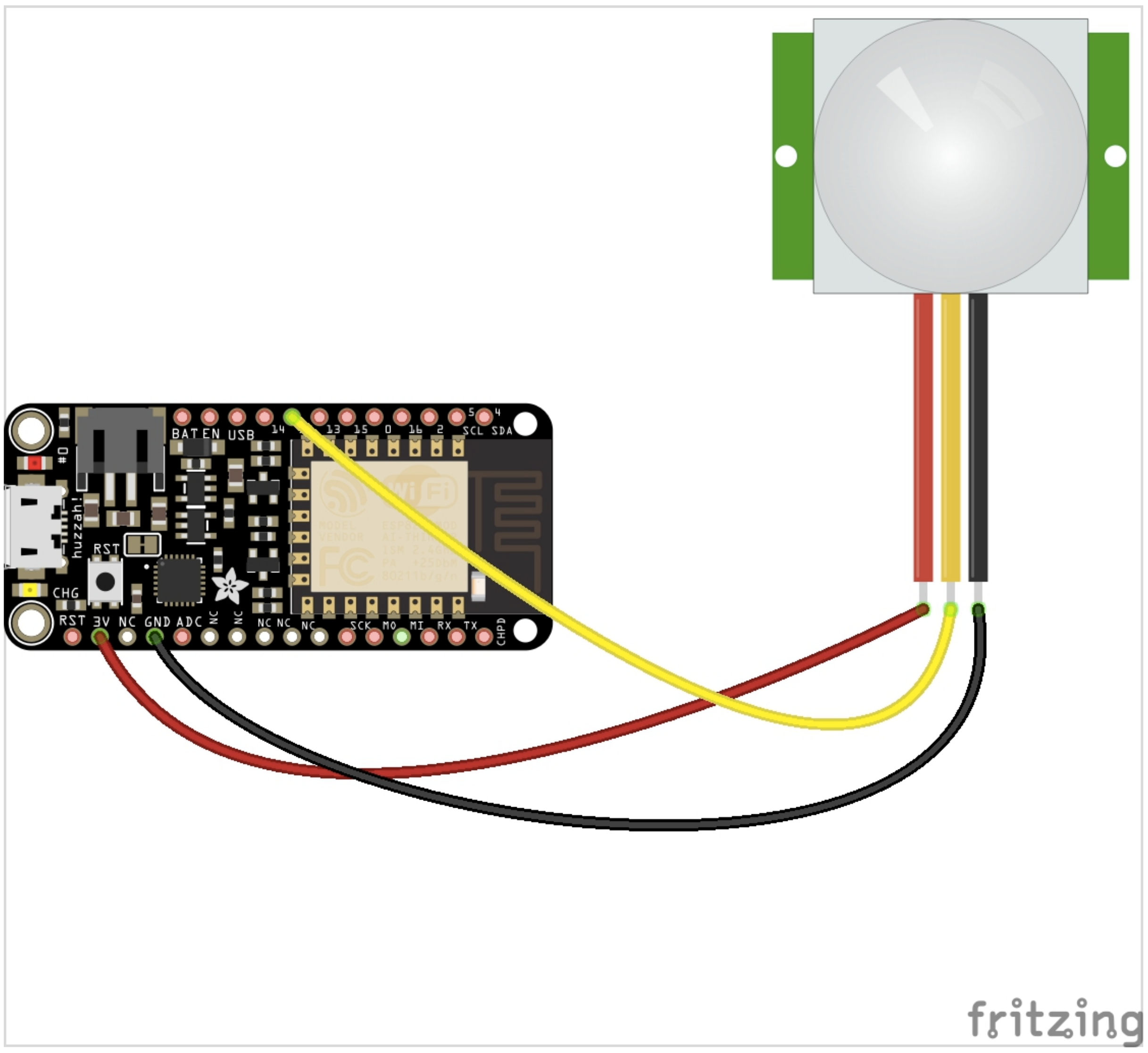
Next, we thought of including support for a physical IR remote. This was a standard IR remote bundled with multi-color LED strips etc. This remote works on the standard NEC packet encoding (discussed later).

Using an external library `IRremoteESP8266`, we were able to trigger a GPIO to receive the packets sent by the remote, and decode it to perform relevant functions (changing colors, increasing/decreasing brightness etc.)

Presence Detection

Connecting PIR sensors to a microcontroller is really simple. The PIR acts as a digital output, it can be high voltage or low voltage, so all I needed to do was listen for the pin to flip high (detected) or low (not detected) by programming the same on the microcontroller.

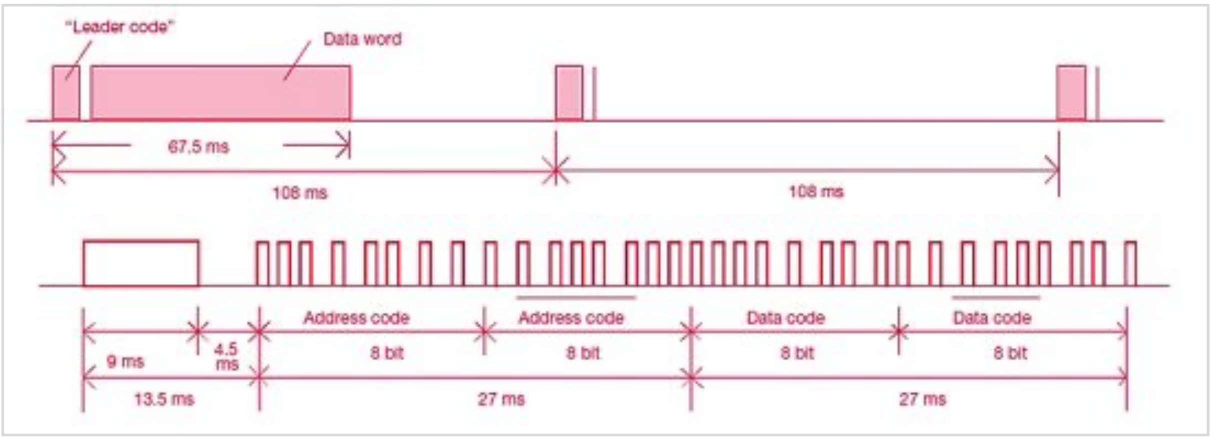
The challenge however lied in timing out; the PIR would trigger the LEDs for a certain duration after timing out unless the sensor is triggered again in that duration. The two potentiometers on the sensor gave us granular control of the sensitivity and the timing of the sensor, but the challenge was to find a setting that worked well.



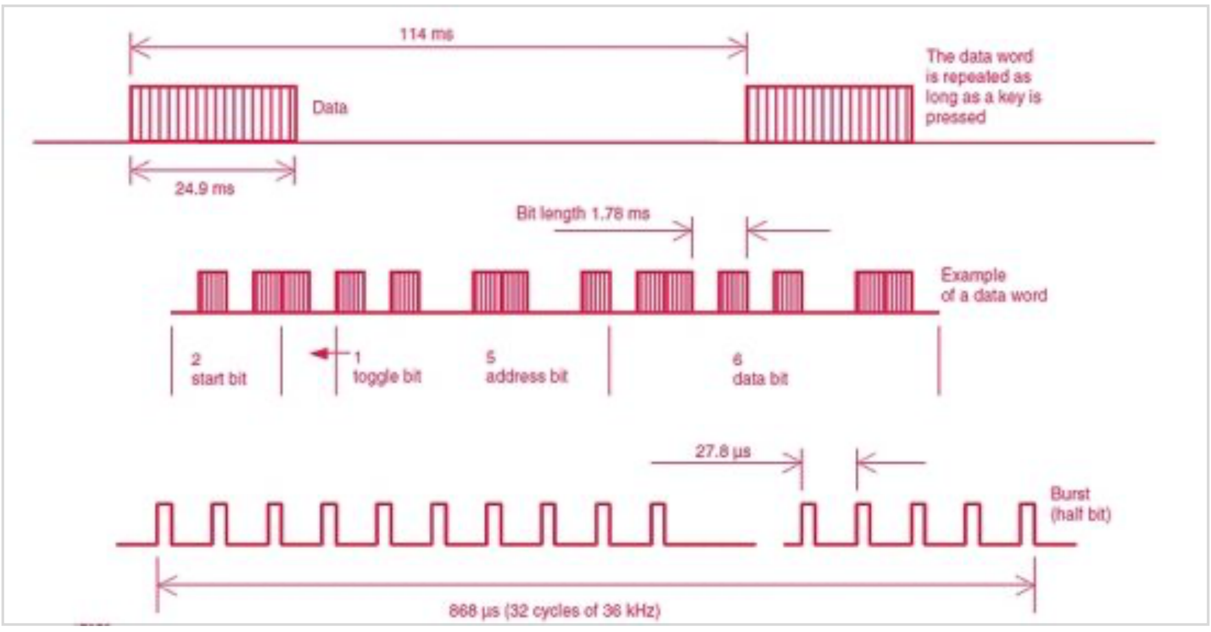
Remote Control Support

Most AC' remotes we tested followed the NEC encoding scheme, which was fortunate as there were several libraries built for ESP8266 for the Arduino IDE that allowed us to receive and send packets similar to that from a remote control. The IRrecv library and IRRemoteESP8266 were two prominent libraries with support for various vendors like Panasonic and Hitachi already built-in. I will discuss the various modulation schemes that I learnt and some challenges involving IR blasting through ESP8266

The two most common IR remote control data formats – the NEC code and the RC5 code – are explained in more detail here. The NEC code uses bursts at a carrier frequency of 38 kHz and starts the transmission using a so-called leader code, a burst with a length of 9 ms, followed by a pause of 4.5 ms and then the data word. The original purpose of this leader code (pre-burst) was to let the internal control loops in the IR receiver modules settle. After transmitting the data word, only the leader code and a single bit are transmitted repeatedly for as long as a key is pressed. A special property of this code is a constant word length in combination with pulse-distance modulation. Both the address and the data bits are transmitted twice – first as a normal byte and then followed by an inverted byte. The half-period burst portion of each bit contains 22 pulses, each with a width of 8.77 µs and a period of 26.3 µs. A "0" is represented by a pulse distance of 1.125 ms, and a "1" is represented by a pulse distance of 2.25 ms. Eight address bits are used to identify the device to be controlled. A further eight bits are used for the transmission of the command data. As indicated above, the words are always followed, without a pause, by the inverted words.



The RC 5 standard uses a bi-phase coding the carrier frequency fixed at 36 kHz. The transmission of a data word begins with two start bits followed by a toggle bit. The toggle bit changes its value at each new key press. The five address bits represent the address of the device to be controlled. The six command bits contain the information to be transmitted. Each bit in the data word consists of half a bit period with no transmission and half a bit period with a burst of 32 pulses at 36 kHz.



So receiving packets and decoding them using the NEC packet decoding (for the LED strip remote) was pretty straightforward - I was able to use the remote to change colors and brightness of the LEDs connected to the ESP8266

A huge shortcoming, specifically for sending IR packets for ACs was that ACs require the remote to send the entire state of the AC in every packet, and not simply just the change you wish to make in the AC's settings. For example, if I wish to increase the temperature on the AC by 2 points, instead of sending two packets indicating 1 temperature difference, I would have to send the entire packet - encapsulating the ON/OFF state, mode the AC is running, and the exact temperature. This means, if the packet structure of the AC remote is not known to us beforehand, it is really difficult for us to map the entire packet to what each function corresponds to. So, while I was able to make the IR blaster work for specific AC models (which were part of the libraries), to create a mapping for a new AC would be a completely manual and time-consuming process (unless we ask the manufacturer for the packet encoding schemes)

AWS Backend

The second half of the project was the software stack for the server to collect the data from the various modules and the sensors attached with them, and to do some analytics with it. We decided to use a free-tier Amazon Web Services Elastic Server (EC2), as we thought it would make sense for scalability; i.e. we could expand the capability of the backend as and when the number of users increased.

We had already made provision for REST APIs in the embedded code in the modules, and all we needed at the server was an http listener which would extract the JSON from the POST request and store it in a MongoDB database.

After setting up the MongoDB server on the EC2 server, we send out POST requests from our ESP nodes to the server - including device MAC address, as well as the WiFi configurations etc.

Learning

I feel really fortunate to have worked on really important, cutting edge technologies at Bisquare, despite this being my first formal internship. IoT as a technology is here to stay and through the internship, I got to work and learn all the various stacks involved in the development of an end-to-end IoT solution. Here are a few takeaways for me I learnt at Bisquare:

About Bisquare

Bisquare is an End-to-end product design and Consultancy company whose focus is to design meaningful products for consumers using a combination of cutting edge technology and aesthetic designs. They can proudly boast to be probably one of the few companies in the world that take up the complete responsibility of a product- from 'Conceptualizing' and 'Designing' the product, to making it manufacturable at the appropriate price point and intelligently positioning it in the market for the target customer segment. This is done through an End-to-end design approach of seamlessly combining Industrial Design, Electronics, User Interaction Design and Software along with packaging and branding to deliver the user experience. With an ecosystem of partners like manufacturers/component suppliers & QA vendors, they achieve various aspects of project execution and delivery.