# Abstract

# System Description/Introduction

Include a high level system diagram and Description

# Hardware/Software Architecture

## WIFI - ESP8266

The team chose to use an ESP8266 as the wifi module as it is a cheap UART wifi module, that already impliments all of TCP, UDP, and SSL. This allows the device to essentially act as a stream based device rather than a packet solution. Additionally, the ESP8266 uses the AT command set which allows for easy computer based experimentation to ensure commands work as intended.

### AT Command Set

The AT command set is a subset of the Hayes command set. These are string based commands originally designed to control dialup modem. The team only needed a few AT commands primarily: Access Point Hosting, Server Hosting, Open Connection, Close Connection, Send Data.

All commands except for receiving data from a connection are essentially blocking. The command and the arguments are sent via UART, and then the ESP8266 response with either Error, the command repeated, or Ok.

As long as an AT command is not running, the ESP8266 can place “IPD:ConnectionInfo:Data” onto the UART. Inside the ConnectionInfo is the number of bytes, connection number, IP, and port in the received message. Typically, this will be an entire TCP stream or UDP packet unless the stream exceeds the ESP8266 buffer size. There is no end of message indicator. Similarly a “Connection Received” message can be sent via UART whenever the wifi device is idle.

### TM ESP8266 Library

The TM ESP8266 Libraries provided a call back based ESP8266 driver. The library required us to implement low level UART initiation, device reset, UART send data, and UART Receive data interrupt handler. These low level functions allow the library to initialize and communicate with the ESP8266.

The library provided weak functions that had to be overridden to handle the callbacks in the case of ESP8266 informing the STM32F4 that it has received a new connection, or it has received a new complete message.

Data requests including HTTP, and temperature as well as the corresponding parsing is entirely done by the “data received” callback function.

## Temperature - DS18B20

### TMOneWire Library

# System Design

## Wifi Connection

### Physical

* One as access point
* UART

### Application Layer

* HTTP GET Parsing for web browsers request.
* Temperature Request response
  + String Based parsing, for code reuse of the HTTP parser
  + Prevents the need of packing and unpacking the floats into the datapacket.

### Call Back Design

* The used library was designed as a hardware independent callback based design.
  + All resources are shared. There are no separate threads waiting for each connection. The callback is called for a given connection for a given request
  + Periodic “Update Function” Call
* Bugs in the library
  + Messages between 28 and 128 bytes were invalidly parsed. This bug was fixed as there was an invalid check in the code
  + Invalid Parsing of messages longer than 128 bytes.
  + Occasionally it misses the receiving of UART data. This causes a stalled connection
    - Implemented a timeout for data receive mode
    - Could not detect the cause of this failure, may have been because of the Print not being interruptable?
    - Could be from semihosting

## Temperature

* One Wire overview,
  + Reset Line
  + Send Connection to Specific device
  + Perform a byte read twice
  + Recover the float value
  + Reset
  + Send a Request to get new temperature
  + Wait 750ms
* So it is periodically checked

# Results

Show screen shots of web pages

# Conclusion

The entire project could have most likely been implemented directly on an ESP8266.