IoT-Based Smart Bathroom Fan Controller

Stewart Schuler
George Mason University
sschule@gmu.edu
ECE508 - Team #01 Spring 2024

I. Abstract

This paper presents an affordable design for a smart bathroom fan controller. The proposed design can automatically control the on/off state of a bathroom fan based on the humidity of the room. The design offers three key features, being automated, being safe – by using third party verification for actions – and, being user-friendly as it retrains the same interface as the traditional manual control switch. Leveraging the explosive growth in Internet of Things technologies allows for the design to be built in a cost efficient manor using readily available parts. Each of the modules included in the design wirelessly transmit lightly processed sensor reading to a central control server which processes the input and adjusts the physical state of the fan by transmitting MQTT packets to the fan control module.

II. Use Case Description

When a person takes a shower the steam produced by the hot water will linger in the bathroom for a period of time after the person has turned off the water. As the room begins to cool once the water source is turned off the humidity in the room produced will eventually condensate onto every surface of the room. If left dried, the moisture will lead to mold which can pose health risks as well as being costly and time consuming to remove.

The traditional solution to the moisture condensation problem has been to include an exhaust fan in every bathroom which contains a shower. These fans are nearly always controlled by a physical switch somewhere in the room. Standard operation involves the user flipping the switch on to turn the fan on at the start of their shower, and after some period of time has passed and the humidity in the room has dissipated to an acceptable level the user will manually flip the switch off.

On the surface the aforementioned traditional approach may not seem onerous, however if we consider the time period between the completion of the shower and when the humidity decays to an acceptable level to turn off the fan we see it can impose a burden. Figure 1 show the decay of humidity as a function of time after a typical shower is completed. It can be seen that the time taken before humidity r pre-shower levels is more than an hour and a half.

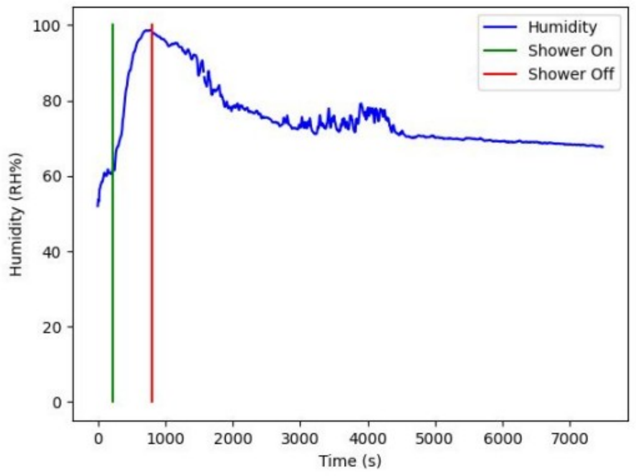
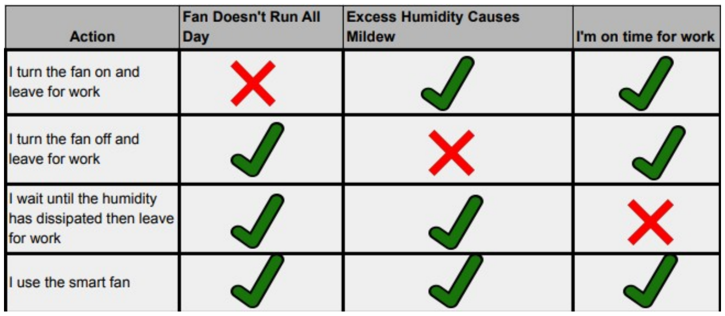


Figure 1. Humidity vs Time of a typical shower

When we consider that a large number of people shower before leaving the home (for work, or other social occasions), and are therefore likely not around to manually turn the fan off and thus the fan runs until they return causing unnecessary wear on the fan motor. The need for an automated system can be seen. Given the following problem there are three desirable outcomes, 1. The fan doesn't run for an excessively long duration after the humidity dissipates. 2. The fan runs for some period of time to properly dissipate humidity. 3. The user doesn't need to stay around for a long period of time after showering to turn the fan off. Figure 2 illustrates these cases and indicates how the smart fan can address all three desires without compromise.



Action	Fan Doesn't Run All Day	Excess Humidity Causes Mildew	I'm on time for work
I turn the fan on and leave for work	✗	✓	✓
I turn the fan off and leave for work	✓	✗	✓
I wait until the humidity has dissipated then leave for work	✓	✓	✗
I use the smart fan	✓	✓	✓

Figure 2. Use case for humidity dissipation solutions

III. Technical Solution

The smart bathroom controller is a modular design consistent with standard practices for IoT system [1]. The design is broken up into four components existing in the various standard layers. The design has two modules on the physical layer, the *fan controller* and the *fan state sensor.* Both are implemented with a single microcontroller and various sensors physically wired to the microcontroller. They interact with the physical world and send/receive MQTT [2] messages containing those physical measurements or states to the *broker* module. The *broker* module is a MQTT server that exists on a publicly accessible cloud hosted by *mosquitto.org [3]*. This module is used as a wireless medium between the modules of the design, it is a free COTS product. The final module of the design is the *server* modules which exists in the data processing layer. It is responsible for converting sensor data messages into a control state for the physical fan and sending that message to the *fan controller* module to enact the requested state. Each of the three custom module designs (excluding the *broker* because it's a 3rd party product) will be detailed in the following subsections. The high level architecture diagram containing modules and connections is depicted in Figure 3.

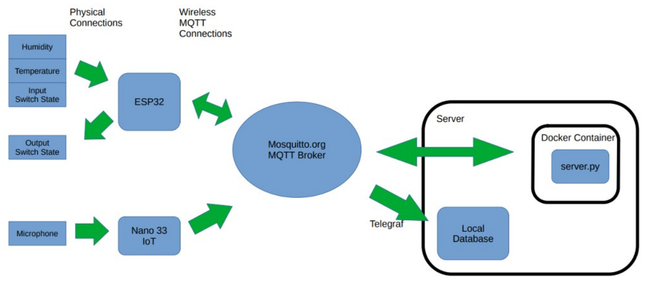


Figure 3. Smart Bathroom Fan Architecture Diagram

III.A Fan Controller

The *fan controller* module is responsible for interacting with both the user and fan control mechanism. Additionally the *fan controller* reported the current humidity state to the server.

User inputs a read by a physical switch connected to one of the many GPIO pins present on the ESP32-C3 microcontroller. In order to properly read a digital 0 value when the switch is open, a 10KOhm pull-down resistor is placed between the input GPIO pin and a ground connection. This large resistance value enforces the input GPIO pin to be shorted to ground when the switch is open, and causes a minimal amount of current flow when the switch is closed. The full wiring diagram for the module can be seen in Figure 4.

The module controls the fan by an external relay [4] which can be controlled by the 3.3V GPIO output from the microcontroller. The relay would be spliced into the power wire of the fan, the same way a traditional physical control switch would be attached. The chosen relay can support 110V AC power which is standard power for all in-home bathroom fans. For testing purposes the fan power connection was replaced by a simple LED to indicate whether the device under test (DUT) was receiving power.

Lastly, the microcontroller also reads in the current humidity of the room using a DHT22 temperature and humidity sensor. Wired to a GPIO input pin. The full wiring diagram for the *fan controller* module can be seen in Figure 4.

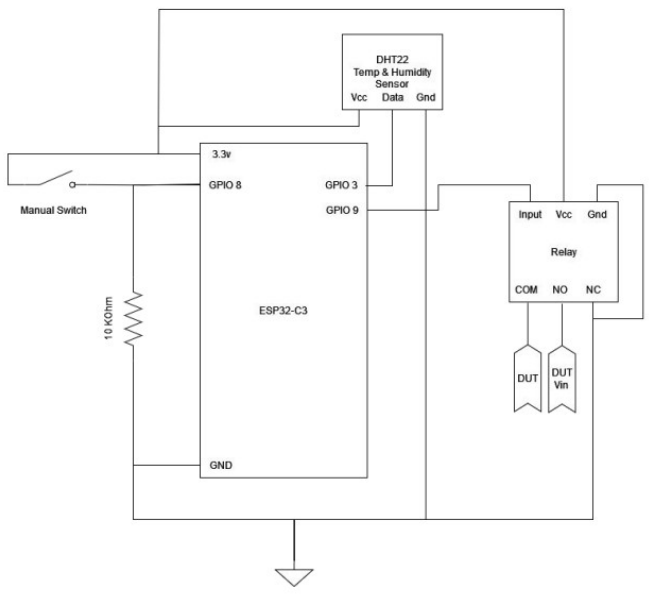


Figure 4. *Fan Controller* Wiring Diagram

The microcontroller software polls the temperature and humidity readings and publishes them to the MQTT broker at a rate of once per 5 seconds. This message is a JSON packet containing the timestamp, unique identifier for the module, the temperature, and the humidity. The format is dumped version of the JSON document shown in Figure 5.

```
{
  "timestamp": <UTC Integer>,
  "source": <Device ID String>,
  "temp_f": <Temperature (F) Float>,
  "humidity": <Humidity (RH%) Float>
}
```

Figure 5. *Fan Controller S*ensor Packet JSON Structure

At the same time it sends data it is also constantly listening for incoming messages from the server containing output fan state change commands. These will be discussed further in the *server* section. Lastly the software is constantly monitoring the state of the manual switch, when the switch **changes** value it means a user is present and requesting a specific state. When this happens the output state of the fan is updated in accordance with the request switch state. If the switch remains unchanged than the fan state is static until a server command is received. This way the fan can be controlled in the exact same was as a traditional switch, but in the absence of user input the smart aspect of the controller takes over an automatically determines the fan state.

III.B Fan State Sensor

The *fan state sensor* module acts as a third party verification method for the *fan controller*. It is a conceivable scenario that a control state of on or off could be sent to the controller, which erroneously doesn't change the state as requested. This could be because the module is damaged or for any other reason. In such a scenario the design should have some way of error checking itself and reporting that it is an error state. This processes is achieved by way of a second mircocontroller and a cheap analog microphone place in close proximity to the fan. The microcontroller will sample the microphones analog output using on of its ADC channels. And using standard signal processing techniques it will determine the frequencies of the noisiest signal in the environment. If this signal matches the expected RPMs of the fan, and persists at a consistent power and frequency for some period of time the *state sensor* will consider the fan to be on. This state is computed and reported back to the *server* at a rate of once per 5 seconds. The module is built using the *Arduino* Nano 33 IoT [5] as the microcontroller with the only external connection being for the power analog microphone, the amplifiers in the microphone are powered by on of the external 3.3V pins of the microcontroller. The wiring diagram for the module is shown in Figure 6.

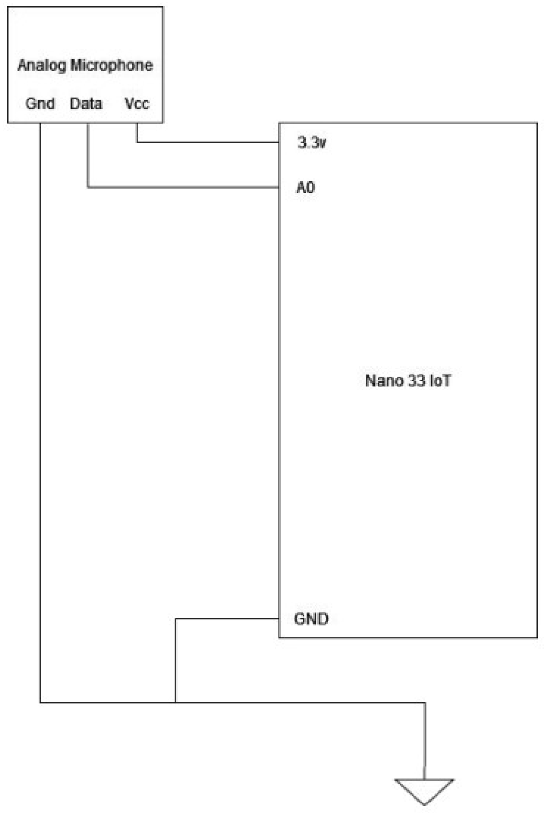


Figure 6. *Fan State Sensor Wiring Diagram*

The fan state is determined by the forthcoming algorithm. The microcontroller is programmed to execute the following sequence every five seconds. First the module samples the incoming analog signal using its ADC at a rate of 1KHz, 1KHz would be a low rate for sampling audio signals of speech, but for this application the sampler is attempting to detect a tone caused by the fan, this tone will be at the frequency corresponding to the RPMs of the fan motor. A 1KHz sample rate will allow the module to accurately capture a tone generated without violating the Nyquist-Shannon sampling theorem assuming the RPMs are less than 500 rotations per second. The modules samples and stores a block of 128 samples, each sample is an unsigned 10-bit value. Because the ADC has voltage range of 0V to 3.3V, a sample containing no audio energy would have a digitally sampled value of 512. To correct for this offset, after being sampled the value is converted to a signed 16-bit integer data type and has the 512 offset subtracted from it. Creating a signed signal centered around zero, which is the standard form for digital signal processing. Once captured the adjusted 128 samples are transformed into the frequency domain using the *Fast Fourier Transform* algorithm. This was accomplished by leveraging the open-source *Arduino FFT [6]* library. Once transformed, the magnitude of the complex valued frequency bins is computed and converted to a decibel scale.

The process of sampling and converting to frequency magnitudes is repeated ten times, with each magnitude being averaged with all the previous values. In total 1280 samples are taken, giving the average frequency over a 1.28 second window of time. This was done negate the effect of any short duration high powered sound noise that may trip the power detector. Because 128 samples captured at a rate of 1KHz are used per window the width of the resulting frequency bins can be computed using the following formula – which is computed to be 7.8125Hz per bin.

$$bin\,width_{hz} = \frac{sample\,rate_{hz}}{number\ of\ samples} \qquad (1)$$

Examples of this averaged frequency domain representation can be see in the following Figures. Figure 7 shows the frequency domain in the presence of only the ambient noise. This will serve as a power floor for detection. If the power at our bin of interest (where the fan is expected to be) is less than the ambient power the module can conclude that the fan is not active.

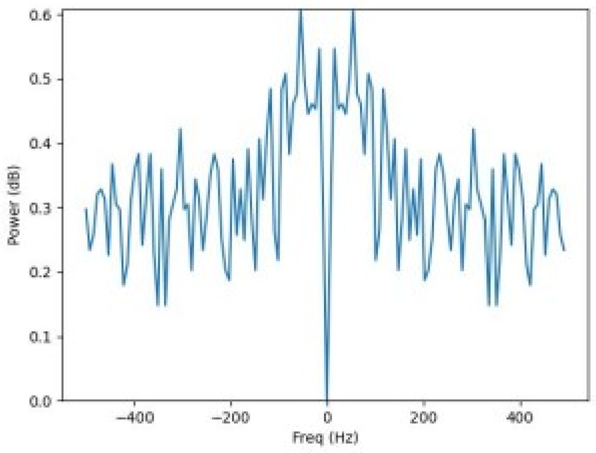


Figure 7. Averaged Frequency Domain of Ambient Noise

For testing purposes Figure 7 shows the averaged frequency domain representation when an audible tone of 300Hz is played at a moderately high value next to the microphone. We would expect to see two peaks at + and – 300Hz, there will be two peaks because the ADC produces a single real value per sample, thus positive and negative frequency cannot be differentiated as can be done with complex sampled signals. Clearly from Figure 8 it can be concluded that frequency domain is as expected.

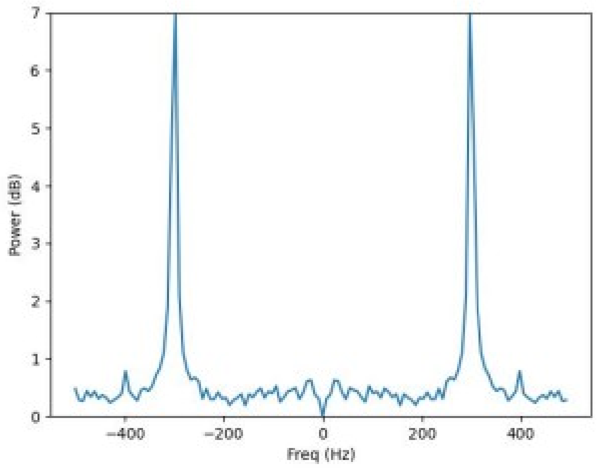


Figure 8. Averaged Frequency Domain of 300Hz Tone

Finally we consider the case of the audio signal produced by the fan being in an active state. Such a case can be seen in Figure 9 where the microphone is placed next to the spinning fan. It can be see that there are again two peaks at + and – 120Hz. 120Hz corresponds to the fans RPMs as expected. It can be seen that the peak bins are well above the noise floor of ~0.6 as derived from Figure 7. The information from Figure 9 can be used to compute a threshold value to determine when a bin has enough energy to be considered “on”. In the case of the bathroom fan used for testing, a decibel value of 1.0 is a sufficient cut off. However, this threshold value can change based upon the ambient noise level of the environment or the ratio between the fan noise and noise floor (the signal to noise ratio). For that reason the make the design generic, the threshold level is input into the system by a server configuration file. Allow the user to use the design in different environments without needing to reprogram the microcontrollers. Likewise the expected fan frequency bin is also a contained in the *server* configuration file, so not only will it work in environments with varying noise levels, but it will also work with fans that produce audio signals at different amplitudes.

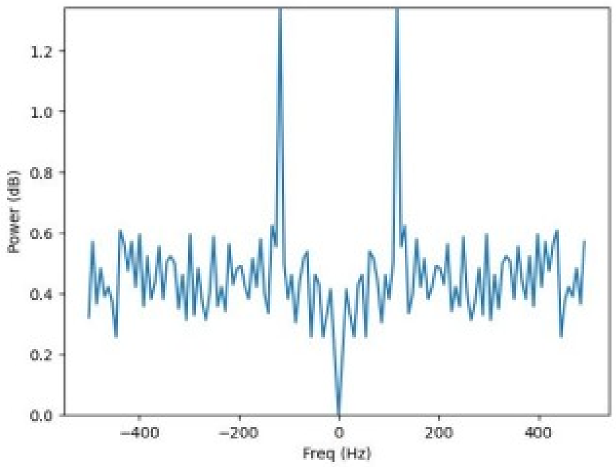
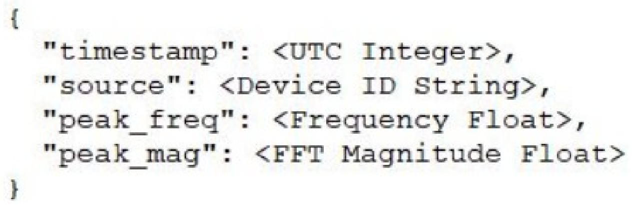


Figure 9. Averaged Frequency Domain of the Fan

Once the averaged magnitude frequency domain has been computed the module reports to the server the frequency value corresponding to the bin with the most energy, and the energy of that bin. Like with the *fan controller* module the messages are sent wirelessly over MQTT, as dumped JSON strings of the JSON document structure shown in Figure 10.



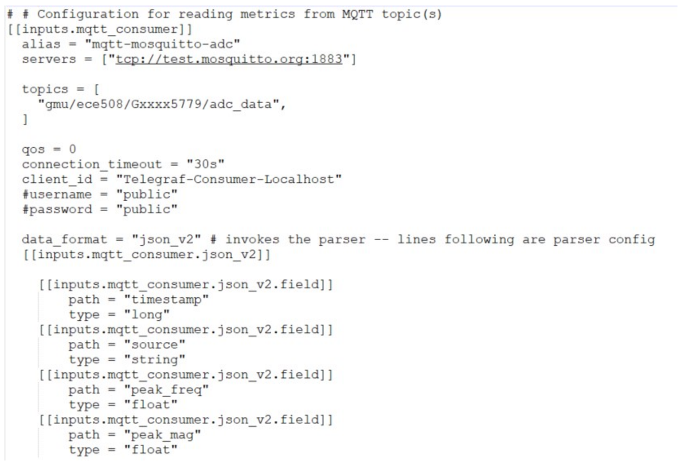
```
{
  "timestamp": <UTC Integer>,
  "source": <Device ID String>,
  "peak_freq": <Frequency Float>,
  "peak_mag": <FFT Magnitude Float>
}
```

Figure 10. *Fan State Sensor* Packet JSON Structure

Since the module acts as an intelligent sensor it doesn't need to receive commands from the *server* module. That is when powered, and connected to an acceptable wireless network the module should always be producing messages at a fixed period. The *server* can detect a fail in this module if the packets are not being received at the expected time windows. The *timestamp* values in the message can help the server verify this.

III.C SERVER

The final custom module in the design is the *server*. The *server* itself consists of two modules, the server logic – which is a python script running in an isolated Docker [7] container connected to the same WiFi network as the microcontrollers – and a logger which is using the COTS product *Telegraf* [8], which logs and stores all MQTT network traffic in a database. The *Telegraf* logger is a mature public available product and can be setup by launching it with a configuration file pointing it towards the specific MQTT server and topics used by the design. An example *Telegraf* configuration module is shown in Figure 11, this block is responsible for logging messages sent by the *fan state sensor* module to the MQTT topic *gmu/ece508/Gxxxx5779/adc_data.* A similar module is create for each message type (which each have their own topic) by modifying the topic name, and JSON parsing format.



```
# # Configuration for reading metrics from MQTT topic(s)
[[inputs.mqtt_consumer]]
  alias = "mqtt-mosquitto-adc"
  servers = ["tcp://test.mosquitto.org:1883"]

  topics = [
    "gmu/ece508/Gxxxx5779/adc_data",
  ]

  qos = 0
  connection_timeout = "30s"
  client_id = "Telegraf-Consumer-Localhost"
  #username = "public"
  #password = "public"

  data_format = "json_v2" # invokes the parser -- lines following are parser config
  [[inputs.mqtt_consumer.json_v2]]

    [[inputs.mqtt_consumer.json_v2.field]]
        path = "timestamp"
        type = "long"
    [[inputs.mqtt_consumer.json_v2.field]]
        path = "source"
        type = "string"
    [[inputs.mqtt_consumer.json_v2.field]]
        path = "peak_freq"
        type = "float"
    [[inputs.mqtt_consumer.json_v2.field]]
        path = "peak_mag"
        type = "float"
```

Figure 11. *Telegraf* Logger for MQTT ADC Data

The server logic module is a python script responsible for reading in the MQTT messages coming from the microcontrollers. It's message processing logic to determin the output fan state is as depicted by the flow chart in Figure 12.

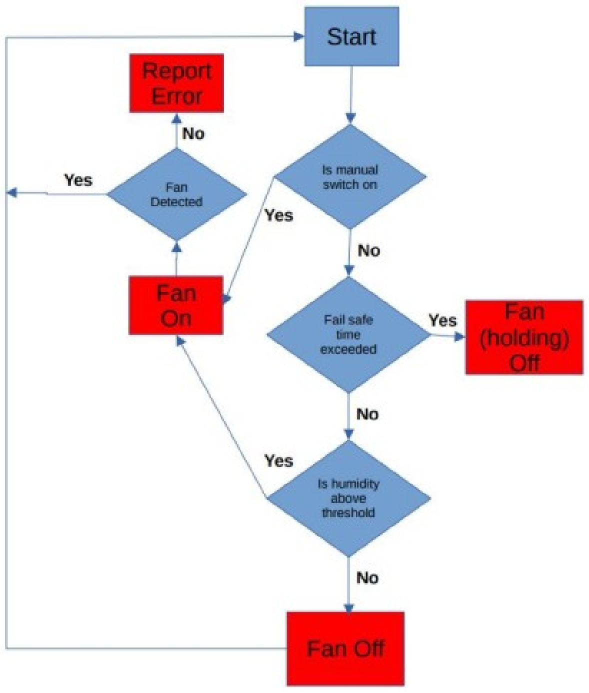


Figure 12. *Server Logic* State Decision Flowchart

In the above logical flowchart the red boxes represent fan states, and the blue diamonds are decisions, implemented by *if* statements in the code. The decisions are decided by the incoming messages from the microcontollers. The two decision blocks which aren't self-evident and require further discussion are the "Is humidity above threshold" and the "fan detected" decisions.

Determining if the humidity is above or below the threshold can be determined by the messages sent by the *fan controller* module containing the instantaneous humidity in the room. However as shown in Figure 1, we observed that the instantaneous humidity measurements can suffer from a wide variance, specifically when comparing to an acceptable humidity level input to the *server* by its configuration file. When approaching the threshold the measurement noise could cause the fan state to be changed by the *server* with each new measurement which would be undesirable. To negate this problem the server keeps a memory of the 12 most recent humidity measurements and keeps a running average of their values. As previously stated, the humidity measurements are reported to the server once every five seconds. That means the humidity used when comparing to the threshold is the average humidity recorded over one minute of data. Enough to minimize the measurement variance caused by the low cost humidity sensor.

The second decision to discuss is the "fan detected" decision. This decision is made using the incoming messages from the *fan state sensor* module. As discussed in that section the module report to the *server* the frequency of the highest magnitude bin present in the averaged FFT and the bin's value in decibels. These measurements are then compared to values defined by the *server* configuration file, in the case of fan used to generate the plots in S*ection III.B*, those values were 120Hz for bin frequency and 1.0 for decibel magnitude.

Once a state leaf has been reached by the server it sends out a message to the *fan controller* module, it does so wireless using the MQTT. The messages contain the same *timestamp* and *source* information as all the other packets and for it's data payload it contains an integer fan state value which the *fan controller* module interprets to a physical fan state and acts accordingly. The MQTT message is a JSON dump of the JSON document shown in Figure 13.

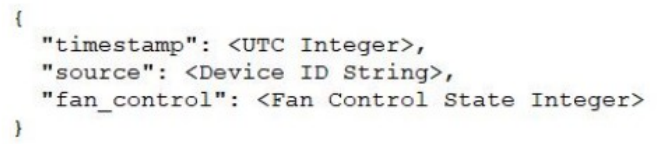

```
{
  "timestamp": <UTC Integer>,
  "source": <Device ID String>,
  "fan_control": <Fan Control State Integer>
}
```

Figure 13. *Server* Packet JSON Structure

It should be noted that upon reaching a state leaf, the server logic restarts and following the flow path again – with potentially new input having been received from the microcontrollers. If the *server* logic reaches the same state conclusion as it's previously sent message there is no need to regenerate the message as the microcontroller would have already acted upon that state request.

IV. Results

To demonstrate functionality of this design as previously mentioned the output relay is connected to LED light to visualize the state of the fan. Figure 14 shows the design with the two microcontrollers wired as described by their respected wiring diagrams. Not pictured is the server console running on the same network receiving and processing their messages. In Figure 14 it can be observed that the LED representing the fan state is off. In Figure 15, the manual switch has been thrown and the LED is in an on state.

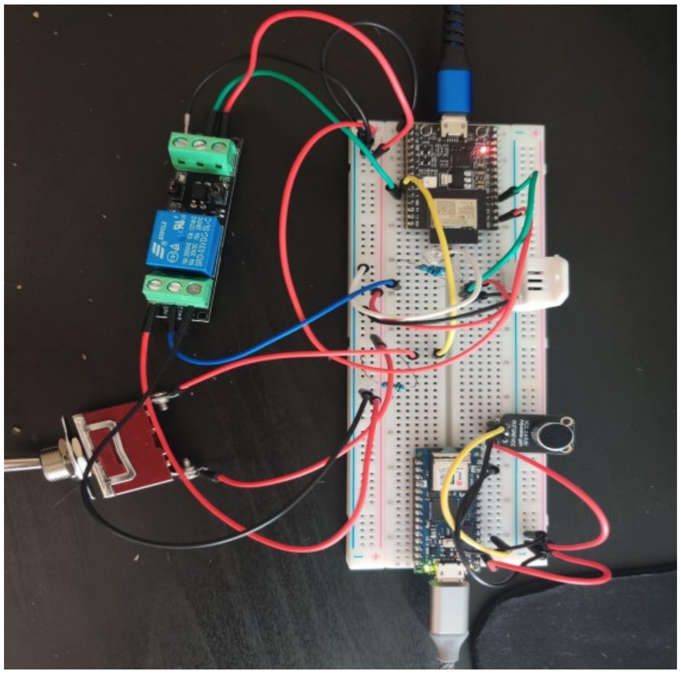


Figure 14. Complete Design Manual Off State

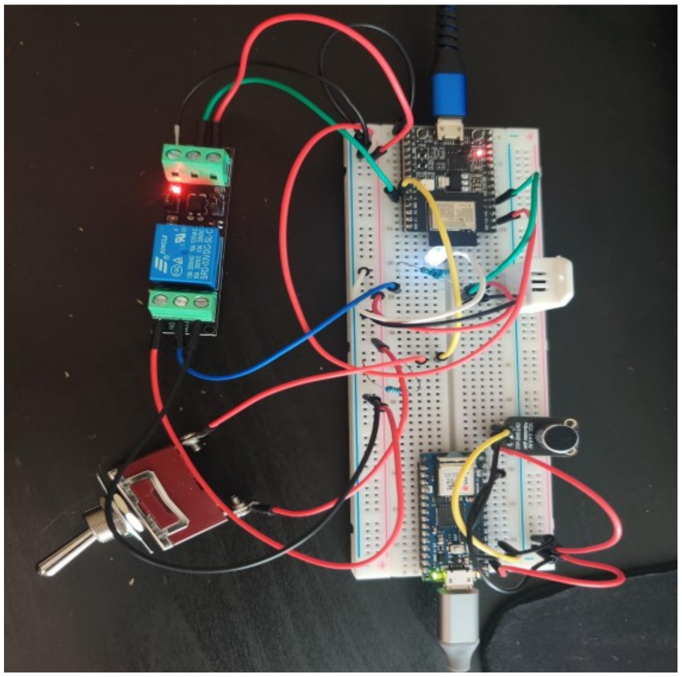


Figure 15. Complete Design Manual On State

Figure 16, depicts the design when being controlled by the server. It can be observed that the manual switch state is set to off, matching the position shown in Figure 14, however the LED representing the fan is on, as it was in Figure 15. This is because the server has detected a high humidity value and turned the fan on remotely without the need for any user input.

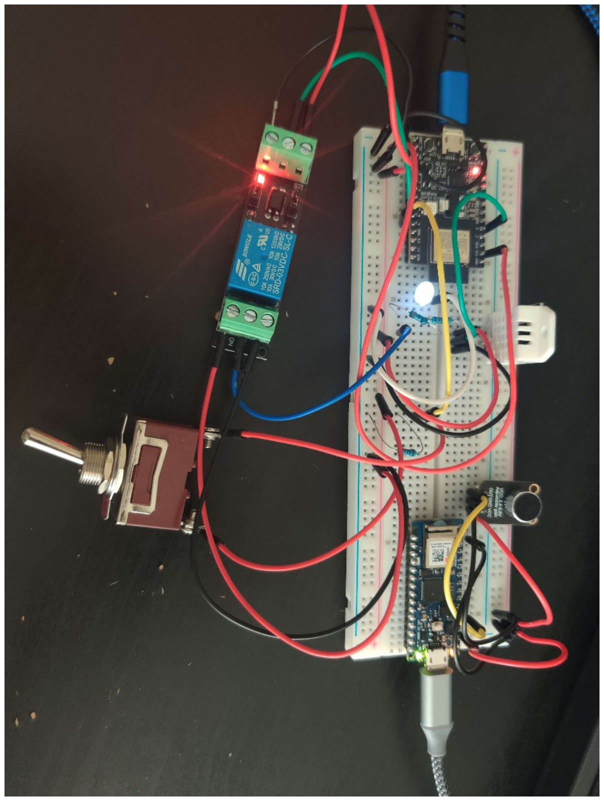
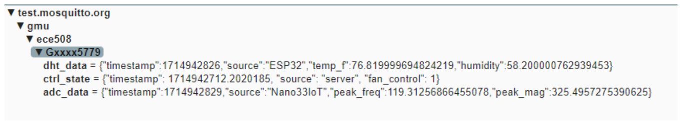


Figure 16. Complete Design Server On State

Lastly to demonstrate functionality of the wireless data transfer. The application *MQTT Explorer* [9] can be connected to the MQTT broker and subscribe to the topics being used by the design. An example of the broker traffic is shown in Figure 17. It is observed that the messages – formatted as previously described – are present on their relevant topics.


```
▼ test.mosquitto.org
  ▼ gmu
    ▼ ece508
     ▼ Gxxxx5779
        dht_data = {"timestamp":1714942826,"source":"ESP32","temp_f":76.819999694824219,"humidity":58.200000762939453}
        ctrl_state = {"timestamp": 1714942712.2020185, "source": "server", "fan_control": 1}
        adc_data = {"timestamp":1714942829,"source":"Nano33IoT","peak_freq":119.31256866455078,"peak_mag":325.4957275390625}
```
Figure 17. MQTT Broker Traffic

V. Conclusion

The *Smart Bathroom Fan* design presented in this paper has been demonstrated to be a low cost product to the problem poised in *Section I*. We have demonstrated that it is a robust modular system which utilizes IoT principals for seamlessly automating a manual task. The presented design not only solves the problem but does so it in such a mature fashion which is capable of reporting it's own error state allowing for the user to make replace specific modules that fail without needing to replace the entire system.

VI. Future Work

For future improvements to the *smart bathroom fan* design there are three changes that would be included in a second revision of the design. They are discussed in this section.

The first and simplest change to the design would be to move the DHT temperature and humidity sensor to the *fan state sensor* module. This design was intending to replace a fan switch in the bathroom. However, if the humidity sensor was moved to the *fan state sensor* module, which will be mounted near the fan, the *fan controller* module doesn't need to be in the bathroom (assuming it can still connect to the fan power source). This would give the user more flexibility to move the controller to a more continent location if they desired.

The second improvement would be to include a wireless manual controller to the design. This can be done via an app or software so long as it is able to connect to the broker. The logic is already in place in the design to handle manual inputs. If those inputs where to come over MQTT from a new source they could be implemented with minor changes to the architecture of the design.

The third and final improvement to the design is to solve the problem of having to set a manual threshold for fan noise power detection. They current design of inputting a configurable single value threshold means the design would likely fail in low SNR environments. This could either be caused by an increased ambient noise level near the sensor or for the fan being quieter. It is a reasonable assumption to make that fan designs in the future will be quieter than their present counterparts. In order to future-proof this design are more complex algorithm could be implemented. Two examples of reasonable light-weight algorithms that could be implemented on the *fan state sensor* microcontroller are a constant false alarm rate (CFAR) detector, which can determine the threshold needed to detect a peak based on the power in adjacent ambient noise bins [10]. The other option in very lower SNR environments, where the peak may not even be the highest bin magnitude would be to use matched filter detection, given that we can experimentally determine the expected tone generated by the fan we could pass it through a matched filter of the expected tone to check if it is present. If implemented correctly and enough data is included it is possible to detect signal below the noise floor using this method **CITE MATCHED FILTER**. Either one of those two methods would make the design even more robust to the environment it is used in.

VII. References

[1] Nissaf Fredj, Yessine Hadj Kacem, Sabrine Khriji, Olfa Kanoun, Mohamed Abid, A review on intelligent IoT systems design methodologies, Measurement: Sensors, Volume 18, 2021, 100347, ISSN 2665-9174, https://doi.org/10.1016/j.measen.2021.100347.

[2] *MQTT Version 3.1.1*. Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[3] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *The Journal of Open Source Software*, vol. 2, no. 13, May 2017, DOI: 10.21105/joss.00265

[4] Teyleten Robot DC 1 Channel Optocoupler 3V/3.3V Relay High Level Driver Module Isolated Drive Control Board 3V/3.3V Relay Module for Arduino

[5] https://store.arduino.cc/products/arduino-nano-33-iot

[6] kosme, https://github.com/kosme/arduinoFFT

[7] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

[8] https://www.influxdata.com/time-series-platform/telegraf/

[9] Nordquist, Thomas http://mqtt-explorer.com/

[10] *Scharf, Louis L. Statistical Signal Processing: Detection, Estimation, and Time Series Analysis. Addison Wesley, NY. ISBN 0-201-19038-9.*

[11] *Turin, G. L. (1960). "An introduction to matched filters". IRE Transactions on Information Theory.* ***6*** *(3): 311–329. doi:10.1109/TIT.1960.1057571.*