# Scout IoT Development Kit

Sponsor: Mark Easley
Team Members: Emily Pigeon, Jacob Sparks, Caleb Snow, Qiang Yang

December 11, 2019

# Table of Contents



# Table of Figures

# Background

As the demand for interconnectivity between devices grows, more and more developers are entering the IoT (Internet of Things) market. These developers are often looking for an IoT kit that can allow them to enter in to this market as fast as possible, as cheaply as possible, and with less and less technical knowledge. This design offers developers a relatively inexpensive and simple way to develop IoT-capable devices with only some embedded C programming skills required.

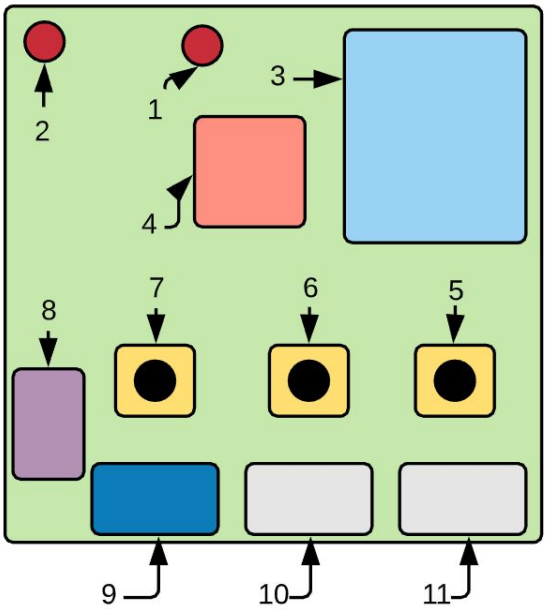
For developers that have some experience using either TI BoosterPacks or Seeed Grove modules, this kit will stand out as it can accept sensors in either format. This kit can utilize analog Grove sensors, digital Grove sensors, and TI BoosterPacks simultaneously. This gives the developer the ability to choose from several hundred different sensors and a vast amount of combinations. Accepting both the BoosterPack and Grove module formats gives this design more versatility, without requiring any more technical experience.

# Requirements

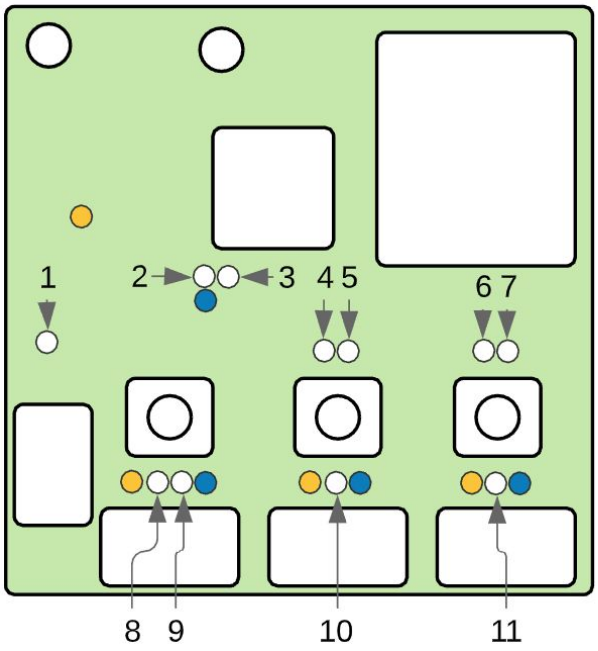
**version 3.1**

1. **General Functional and Mechanical Requirements**
   - 1.1. Product MUST have an open-source PCB layout and bill of materials
   - 1.2. Product MUST have a Quick Start Guide paper insertion with a pinout diagram, instructions on how to obtain software demos and PCB layout, and a list of materials.
   - 1.3. Product MUST transmit sensor data via Wi-Fi to cloud service at no less than 1Mbps
   - 1.4. Product MUST transmit sensor data via BLE, at least 1Mbps, to the mobile phone app
   - 1.5. Product MUST cost under $50 to manufacture
   - 1.6. Product MUST include a plastic chassis with tool-less assembly and disassembly
2. **User Interface Requirements**
   - 2.1. Product MUST have a user-programmable LED and push button
   - 2.2. Product MUST have an LED to display Wi-Fi connection status
   - 2.3. Product MUST have power status LED
3. **Hardware Requirements**
   - 3.1. Product MUST use micro-USB connector for power
   - 3.2. Product MUST use TI MCU
   - 3.3. Product MUST have clear silkscreen labeling of important board markings for users
   - 3.4. Product MUST have headers for hardware debugger (MSP-FET)
   - 3.5. Product MUST include 3 Seeed Grove connectors, 2 designed for analog signals and 1 for digital I²C signals
4. **Software Requirements**
   - 4.1. Product MUST have at least 5 code examples with documentation
   - 4.2. Code examples MUST be available for Code Composer Studio
   - 4.3. Code examples MUST include blinking LED, button input, and wireless communication initialization
   - 4.4. Cloud Service MUST show data from all sensors and define which sensor ports are not in use.
   - 4.5. Mobile Application MUST show real-time sensor readings

# System Drawing




1. LED 1 (p1.0)
2. LED 2 (p4.0)
3. ESP32
4. MSP430
5. Button 1 (p4.2)
6. Button 2 (p4.1)
7. Button 3 (RST)
8. USB Port
9. I2C Digital Sensor Port
   SDA(p7.0)/SCL(p7.1)
10. Analog Port 1
    (p1.4)
11. Analog Port 2
    (p1.3)

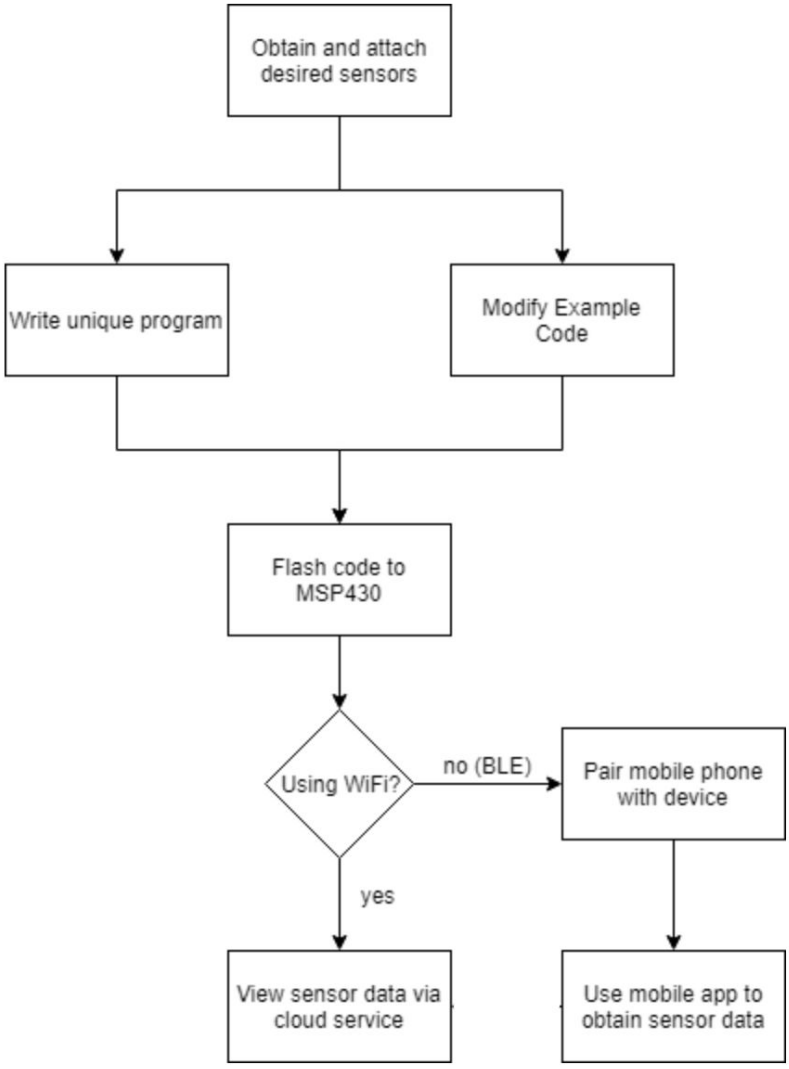



● = 3.3V

● = Ground

1. 5V
2. TCK
3. TDIO
4. UCA1 TX
5. UCA1 RX
6. MSP430 -> ESP32
7. ESP32 -> MSP430
8. SDA (p7.0)
9. SCL (p7.1)
10. P1.4
11. P1.3

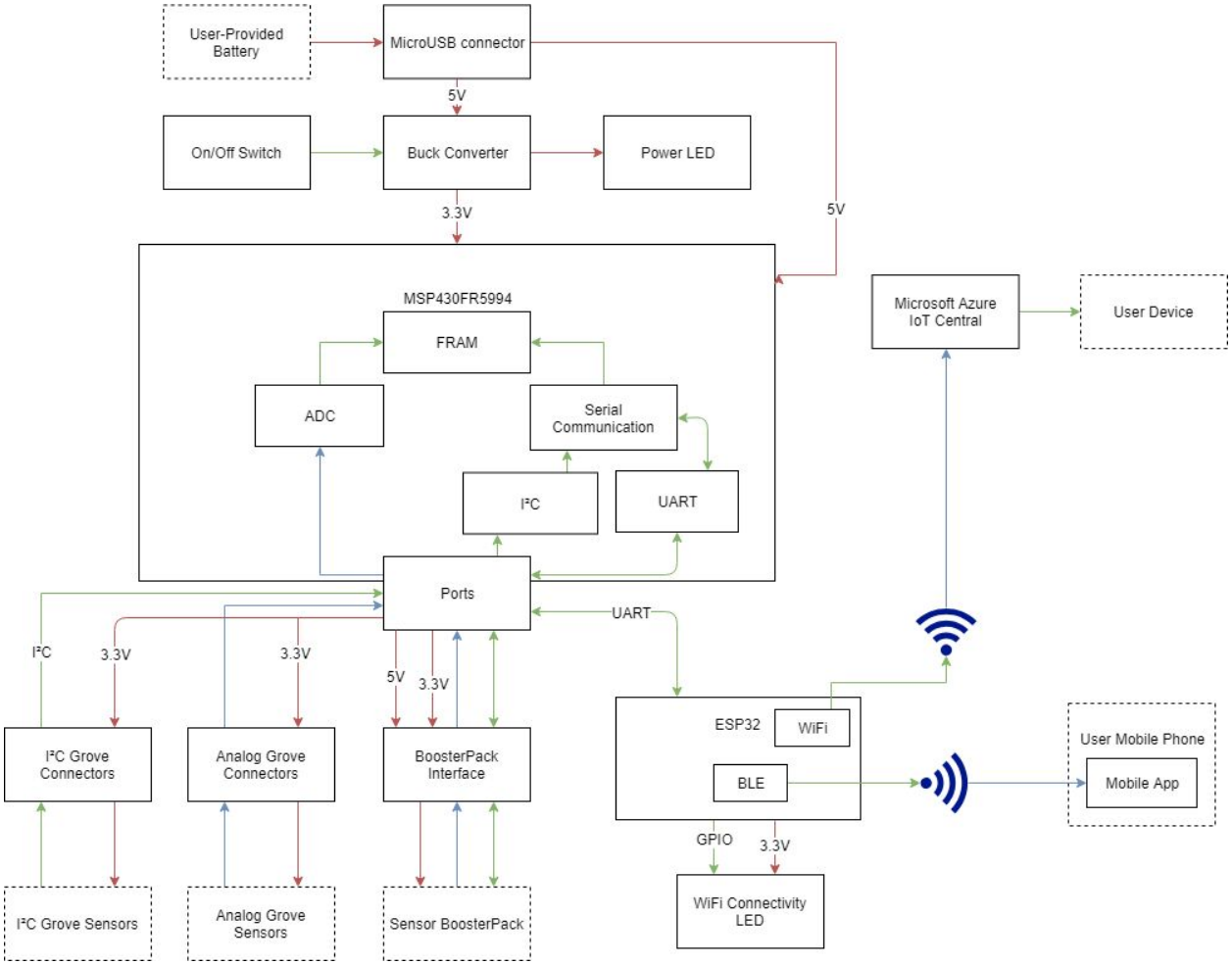
# Operational Diagram




Since this kit is aimed at aspiring developers, the user operation will require a limited knowledge of embedded systems coding. We will be providing example code with functions that connect to a Wi-Fi network and send data over either connection, but for specific applications beyond these simple starting points, some expertise will be required.

The user will view and analyze sensor data via our chosen cloud service, Microsoft Azure IoT Hub. The sensor data could also be analyzed on the chip, and actions related to the specific application could be taken without direct user intervention, e.g. using a general-purpose I/O to activate an actuator.

# Detailed System Block Diagrams




The two key blocks in our architecture are the MSP430 microcontroller and the ESP32 network processor. Part of the reason we decided on the ESP32 can be seen in our diagram - ease of connection. Only a single UART connection will be necessary for the MSP430 to communicate with the ESP32 and, therefore, the cloud service and user. As can be seen in the legend, green arrows are digital information, either in the $I^2C$ format or UART on our device, and in other forms on the user devices. Red arrows represent power that our power system will provide for and blue arrows show analog signals, either sensor data to be sampled by the MSP430's analog-to-digital converter (ADC) or wireless signals.

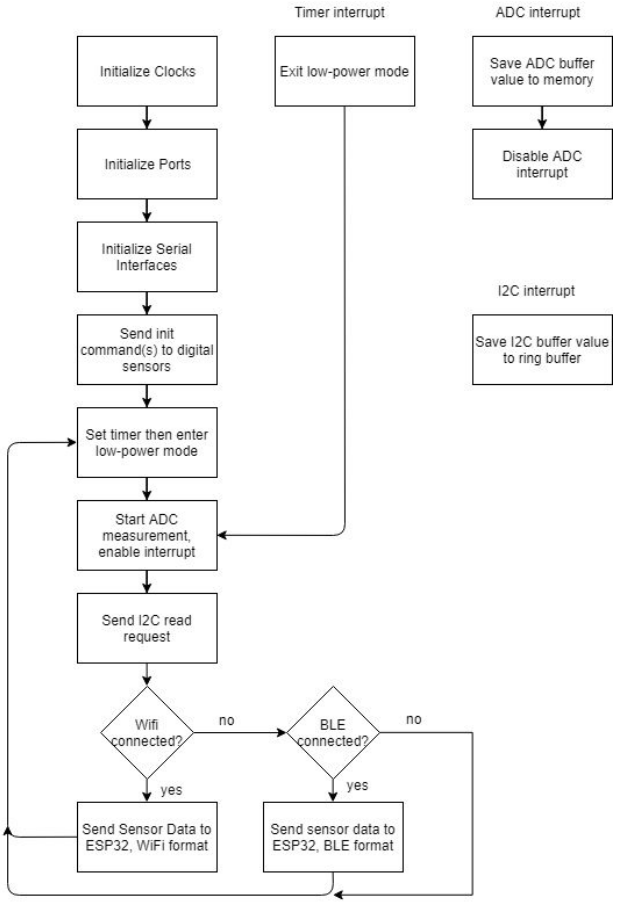
Blocks shown with a dotted border are subsystems that are critical to the full function of our device but will be provided by the user. Any power source that conforms to the MicroUSB standard could be used in place of a battery, provided it can fully supply the power required. This means that a user could instead use an AC/DC converter to power the device from the wall if they were so inclined.

The sensor choice is up to the user, and we provide two methods of connection: Grove sensors and BoosterPacks. The sensors will communicate with the MSP430 either through an analog signal or a serial digital communications protocol. Two of the Grove connectors will use the I²C protocol and the other two will send an analog signal to be processed by the MSP430's ADC.

The ESP32 will forward sensor data to Microsoft Azure where it will be processed and visible to the user through any device. The forwarding will either happen through Wi-Fi, where it will connect directly to Microsoft Azure through the Internet, or through Bluetooth Low-Energy (BLE) via a mobile phone application.

The overall format of the software provided as an example is a loop wherein the device stays in a low-power mode for as long as possible. After initialization, the main loop consists of enabling interrupts, getting sensor data through those interrupts, forwarding the data to the ESP32, then reentering low-power mode.

The messages sent to the ESP32 will depend on whether the device is connected to a wireless network with access to Microsoft Azure or paired via Bluetooth to the mobile app.



# System Design and Component Selection Alternatives

Microcontroller

Alternatives to the MSP430FR5994 include the MSP432P401Y, a more expensive 32-bit MCU with an ARM Cortex CPU and a clock of up to 48MHz, and the less expensive MSP430FR2355. We chose the MSP430FR5994 due to its value and ability to perform the functions necessary for an IoT development kit. The larger number of ADC and I²C channels and larger memory made it the ideal choice when compared to the lower-priced FR2355. The MSP432401Y's higher-powered CPU would not be fully utilized by our applications, and since it had almost equivalent I/O capabilities with the FR5994, it was ruled out.

| Microcontroller | Price | I²C | ADC channels |
|---|---|---|---|
| MSP430FR5994 | $3.60 | 3 | 20 |
| MSP430FR2355 | $2.40 | 2 | 12 |
| MSP432P401Y | $5.40 | 3 | 12 |

Network Processor

Alternatives to the ESP32 included the TI CC3100MOD, a Wi-Fi only network processor made with cooperation with MSP430 microcontrollers in mind. However, its higher price, lack of Bluetooth capability, and more complicated proprietary API made the ESP32 the better choice. Espressif also makes cheaper network processors such as the ESP8285. These would have served well for our application if Wi-Fi had been our only method of communication, but the added benefit of Bluetooth made the ESP32 worth the extra cost.

| Network Processor | Cost | Capability | Ease of use |
|---|---|---|---|
| Espressif ESP32 | $3.80 | Wi-Fi + Bluetooth | AT Commands |
| TI CC3100MOD | $6.99 | Wi-Fi | TI API |
| Espressif ESP8285 | $1.80 | Wi-Fi | AT Commands |

Cloud Service

| Cloud Service | Pros | Cons |
|---|---|---|
| Microsoft Azure IoT Hub | • Cost: Free to sign up<br>• Made to be simple and easy to use for people with little cloud experience<br>• Provides a GUI with charts for incoming data<br>• Many online resources on how to set up | • No specific online documents relating to ESP32 from Microsoft<br>• Less code samples |
| Amazon Web Service FreeRTOS | • Partners with Espressif<br>• Good online documentation concerning the ESP32<br>• Includes support for BLE Mobile<br>• Many code examples from Amazon | • Price: Charges for data transfer, otherwise free to use<br>• Code intensive |
| Google Cloud IoT Core | • Partners with Espressif<br>• Thorough overall documentation<br>• Many code examples from Google | • Price: Charges by volume of data, up to 250MB free<br>• No online documentation about the ESP32 from Google<br>• Code intensive |

Amazon Web Service's FreeRTOS is a contender because Amazon is the most popular company on the market providing cloud services, which means it is well documented and will likely have many available resources for us to use in development, as well as the consumer. This particular service is directed specifically at IoT developers working with MCUs. A major benefit of this service is the BLE mobile app, but this service is still in beta testing. There is an online guide to set up service on an ESP32 Development Kit, and it is very thorough, but it also shows how labor intensive it is. The price could also pose an issue, as most of what we will be using the service for is data transfer from the device to the cloud, which means this service could be costly

to us. The total cost includes many different aspects of how we use the service, so price estimation would be very difficult. In addition, during the research of this service, we could not find any examples of how the data would be displayed on the PC client, which implies that is something we'd set up ourselves. Overall, while the service is very popular and well documented, it is meant for people with a lot of experience setting up a cloud service, and we do not have enough time or resources to dedicate to using this software.

Google Cloud's IoT Core is also a contender. Google is another reputable company but is not as established as Microsoft or Amazon when it comes to IoT applications. The service itself is extremely well documented, with many code samples in many different languages. However, there is very little documentation to be found concerning the ESP32 specifically anywhere online, and none was found from Google. Pricing is also a concern. Google charges for every message sent after a limit of 250 MB is exceeded. The IoT Core from Google would also need to use Google's IoT Pub/Sub service, which we would also be charged for. Overall, similarly to AWS FreeRTOS, this service would be difficult to set up and could be expensive.

Our final contender is Microsoft Azure's IoT Hub, which is Microsoft Azure's IoT cloud service that is comparable to the previous two choices. IoT Hub can handle connection to the device, neatly displaying sensor data in graphs, and creating rules and actions concerning sensor data, such as sending an email if a sensor is tripped if used with Hub Solutions. Pricing is also very simple, as it is free to sign up with $200 credit. Considering that we will only be using one device, and likely will be under the limit, we will likely be able to use this service either free or very cheaply. Unfortunately, there isn't documentation from Microsoft concerning the ESP32 specifically, however there are general guides to aid set up. There are thorough online guides about how to set up the software itself as well. Another potential issue with this software is that it is more limited than the previous examples, so there is not much we can control. After comparing these cloud services, we selected Azure's IoT Hub for its apparent ease of use and simple pricing.

Mobile App Builder

To provide clients with examples of setting up Bluetooth from the ESP32, we must select a simple mobile application builder to debug our software. To do this, we must use a quick and easy mobile application builder that can communicate through Bluetooth and accurately read back sensor data. There are many options on the market for mobile app builders, but it is important that we find one with many available online resources and support.

| Mobile App Builder | Pros | Cons |
|---|---|---|
| Thunkable | • No coding required<br>• Online tutorials with the ESP32<br>• Free for public projects, offers special education prices<br>• Works with both iOS and Android<br>• Public library of projects<br>• Supports BLE | • Reports of bugs<br>• ESP32 example uses Arduino IDE |
| Evothings Studio | • Supports BLE<br>• Works with Android and iOS<br>• Completely free and open source<br>• Extensive BLE documentation | • Coding in HTML5<br>• No specific ESP32 examples |
| Android Studio | • Reliable company<br>• Extensive BLE documentation<br>• Coding in Java or C/C++<br>• Completely free | • No specific ESP32 examples<br>• Only supports Android |

We chose Thunkable as our sample mobile application. Because this software requires no coding, it will be much quicker to get up and running when our ESP32 is ready to be debugged. Android Studio and Evothings Studio both require the application to be built from code, which will make debugging more difficult because bugs could exist in either our mobile application or our development kit. Also, Evothings Studio requires software to be done in HTML5, which none of us have experience with. Thunkable also has a public library of apps that other developers that use the software have created. Within this library are examples of a BLE application that connects specifically to the ESP32. While both Evothings and Android have very thorough documentation of how BLE can be implemented on their platforms, neither has examples that relate to the ESP32 specifically. Finally, Thunkable, as well as Evothings, creates applications that can be used on both iOS and Android, which will allow our code to be more

thoroughly tested and easier to develop. Android Studio, while being a reliable company, is not available on iOS which means we will have to have an Android phone to test.

# Hardware Design

**PCB Design/CAD Tool Selection**

The CAD tool chosen for the schematic and PCB design of this project is Eagle. Eagle was chosen from the selection of tools including: Altium, KiCad, Eagle, OrCad, and DipTrace. A major selection criteria in this process was design files/footprint/symbols accessibility. Creating these from scratch would be much less time-efficient than finding them online and importing them into our design tool. Eagle offered the best online support of the tools mentioned and can import files directly from the UltraLibrarian tool on Texas Instruments' website. Any other components can be sourced from the many websites that offer libraries for Eagle.

Another consideration was the user interface. The more intuitive the interface is, the less time consuming it will be to begin using the CAD tool. Eagle's user interface is as simple and intuitive as any of the other choices. The Eagle schematic editor includes:

- Links to a library
- Electrical rule check
- Netlist generation
- Forward/Back annotation between schematic and PCB (keeps things in sync when making changes in one or the other)
- Schematic hierarchy for the design organization
- User access to define many things like nets, wire width, clearance, etc.

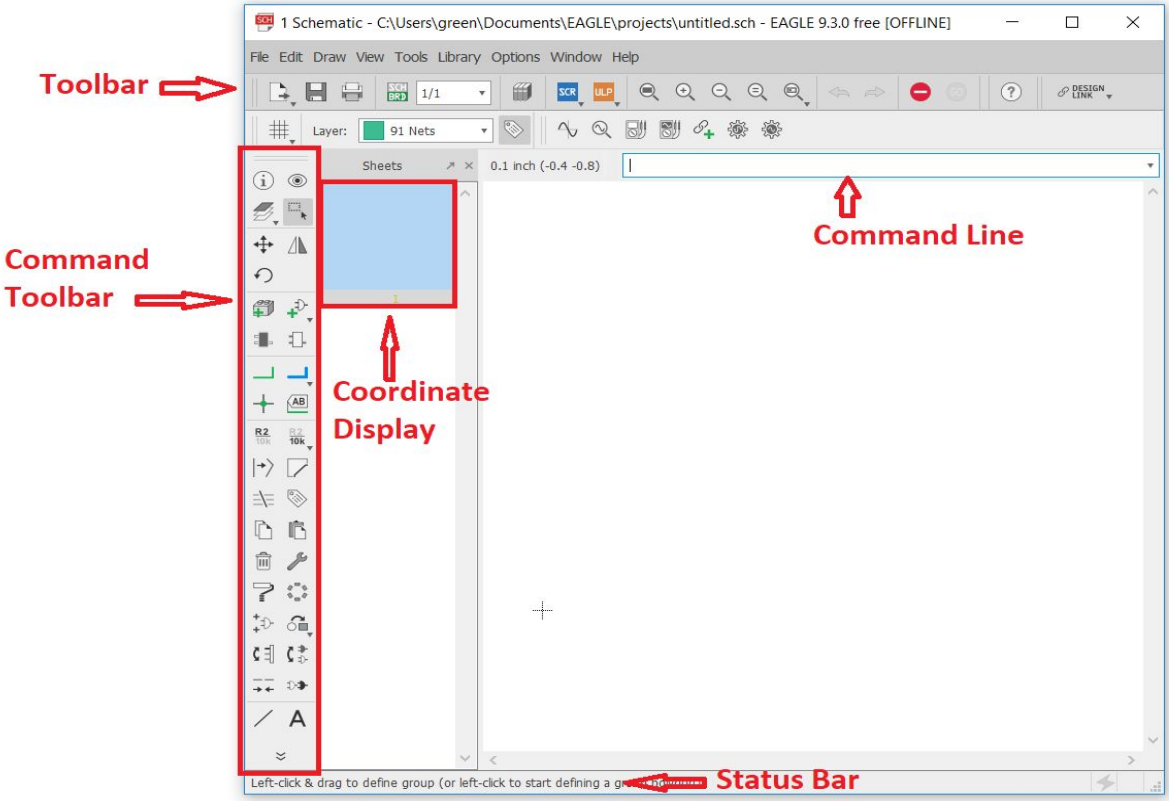



*Figure 8 - Eagle User Interface for Schematic Design Window*

## Basic Circuit Design

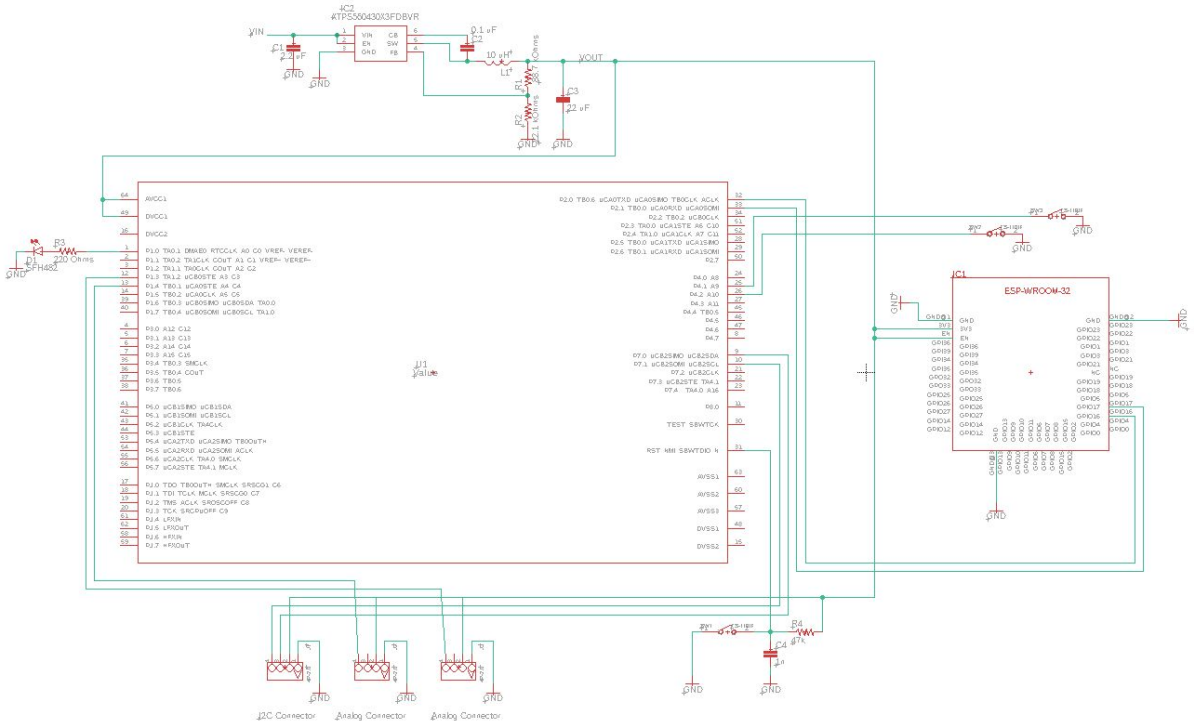



*Figure 9 - Schematic Design*

Depicted above is the first iteration of the circuit schematic. This schematic includes 2 analog grove sensor connectors, 1 digital grove sensor connector, 2 I/O push buttons, a reset button circuit, a programmable LED circuit, the power supply circuit, the ESP32 surface-mounted chip, and the MSP430 64-pin surface-mounted chip. This schematic is identical to the final design except that the power supply circuit was replaced with a through-hole buck converter module to save space and for fast prototyping. The intention was to replace the less efficient and more expensive power supply module with an improved power supply circuit and buck converter IC on the board, however this was not accomplished. The microUSB port did not have a schematic element and was therefore not included in the schematic. However it can be seen on the final PCB as illustrated in figure 10.

**Final PCB**

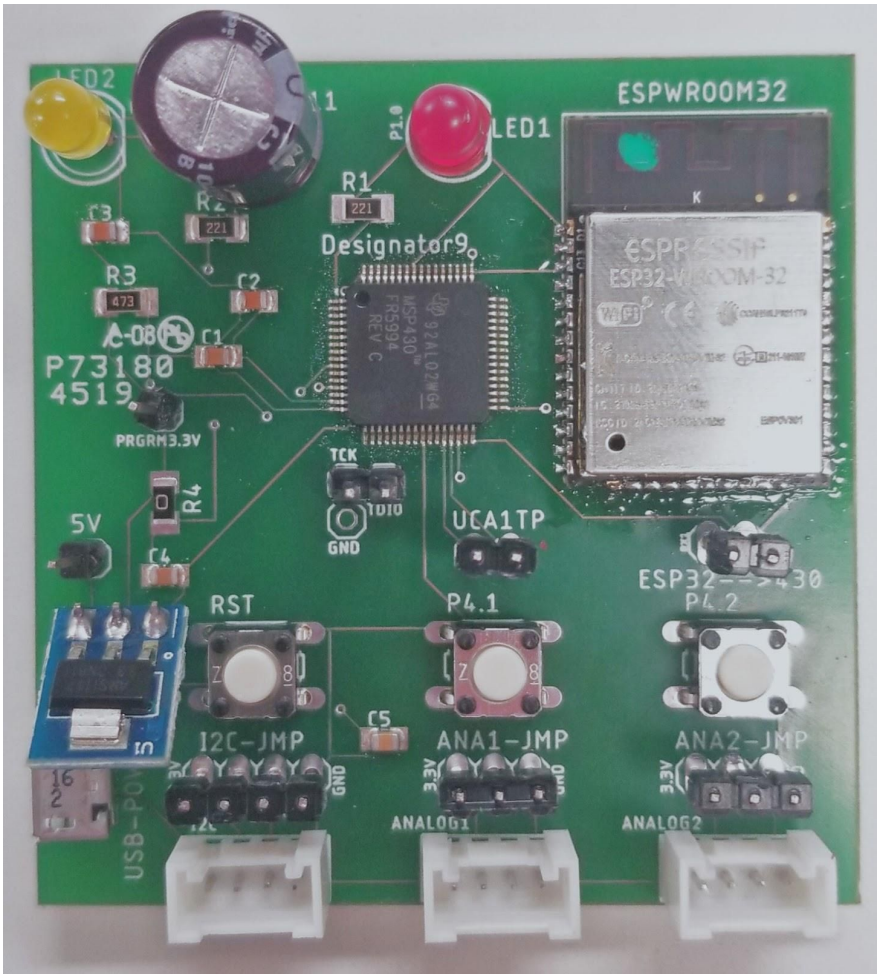



*Figure 10 - Final PCB*

The final PCB is illustrated in figure 10. The dimensions were 2.35” by 2.3”. The substrate for the PCB was FR-4. Anywhere possible, surface mounted components were favored over through-hole to save space and costs. The labeling of components and headers was very important for this design, as its target audience will include novices. Keeping not only the components labeled but also labeling what pins on the MCU they are connected too will help users when flashing the board, intercepting communication between elements, or wiring boosterpacks and other non-grove sensors. The PCB was manufactured by Advanced Circuits and its price was $99 for 3 boards. This equates to one board costing $33. This price would be much less when manufacturing in bulk.

# Subsystem Block Diagrams

**Microcontroller**

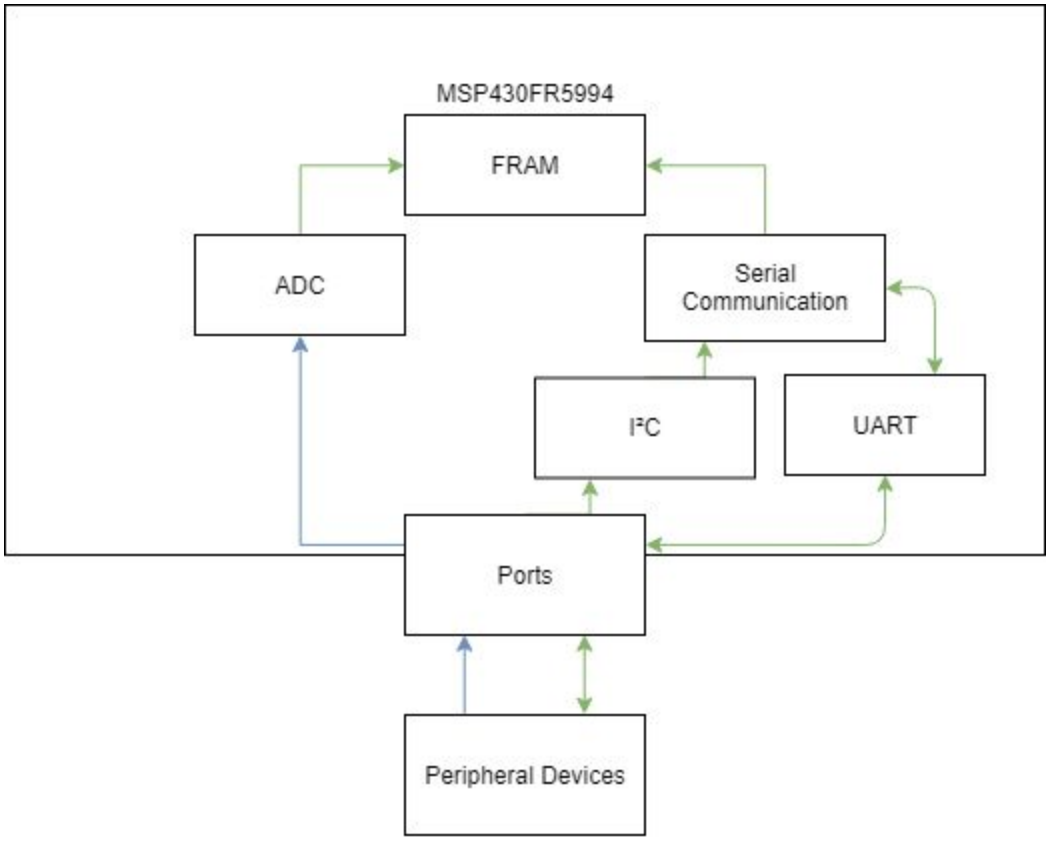



*Figure 9 - MSP430 Simplified Block Diagram*

The microcontroller we have chosen for the device is the MSP430FR5994. It has 256kB of Ferroelectric RAM (FRAM), which is faster and more resilient than traditional flash memory, and unlike DRAM, is non-volatile, meaning it retains its values even without power. Having a healthy amount of memory will be important for storing sensor data if an Internet or Bluetooth connection is temporarily unavailable.

The MSP430 we have chosen has up to 20 ADC channels and 4 I²C channels, although the package we are using will only have 3 I²C ports available. A more detailed view of the full capabilities of the MSP430 can be seen below.

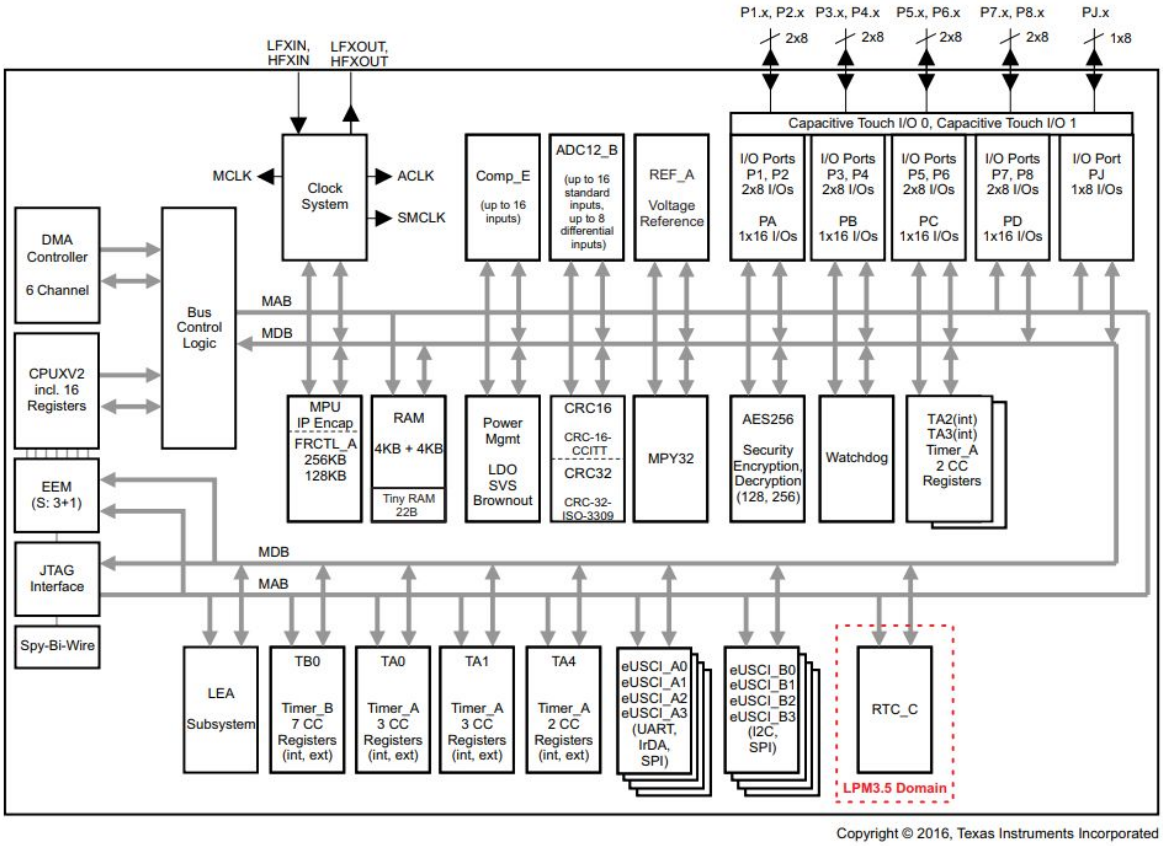



*Figure 10 - MSP430 Detailed Block Diagram*

**Sensors**

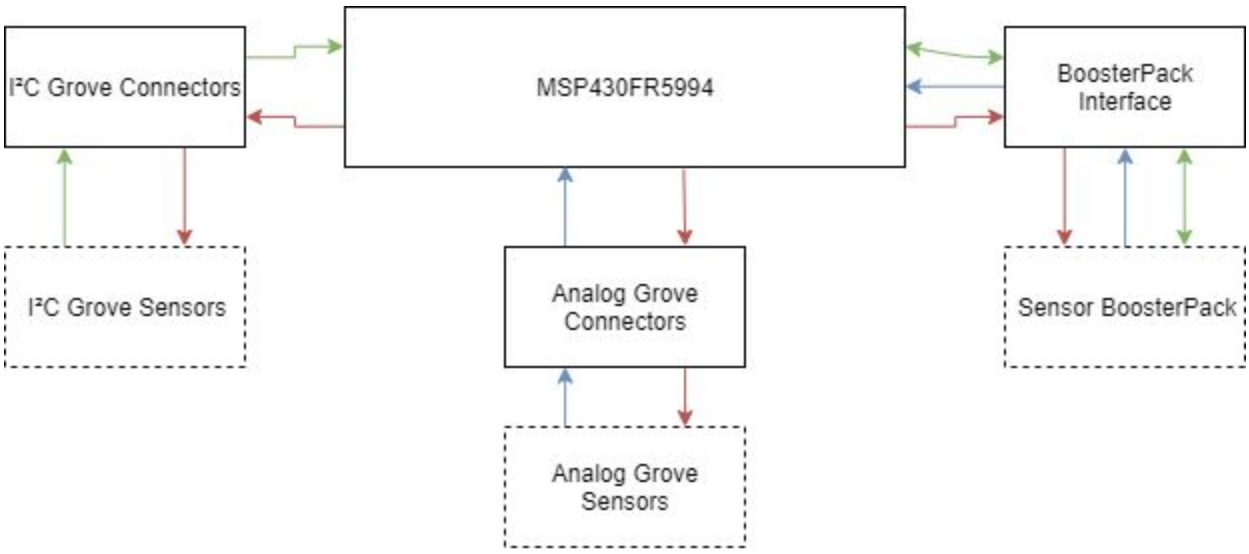



*Figure 11 - Sensor Subsystem Block Diagram*

The device will include four Seeed Grove connectors. We chose this sensor ecosystem due to its popularity and simplicity. Each connector only has four pins, making the hardware design straightforward. Two of our connectors will be analog and two will use the I²C digital protocol.

| Pin | Function |
|---|---|
| pin1 | Primary Analog In / I²C clock |
| pin2 | Secondary Analog In / I²C data |
| pin3 | VCC |
| pin4 | GND |

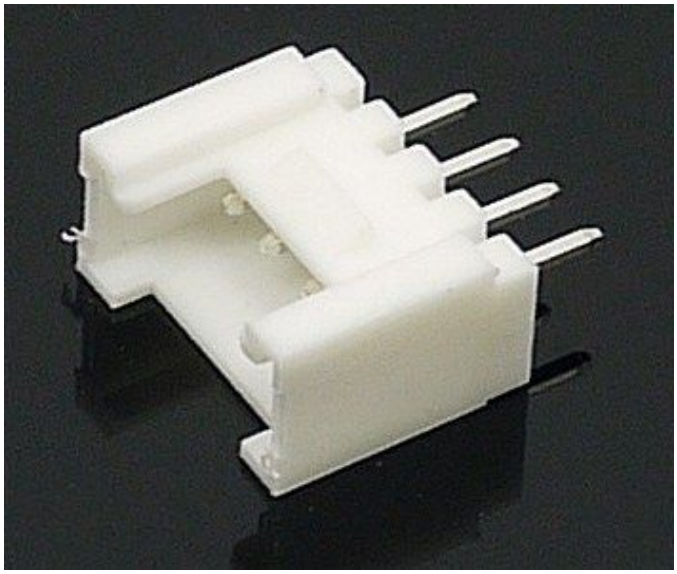


*Figure 12 - Grove Universal 4-pin connector*

The second method a user will have to connect sensors will be the BoosterPack interface on the board. These pins will be capable of accepting any TI BoosterPack compatible daughterboard or any jumpers the user might use instead. We decided on the BoosterPack ecosystem due to its maturity, availability of documentation, and ease of use in our CAD tool.

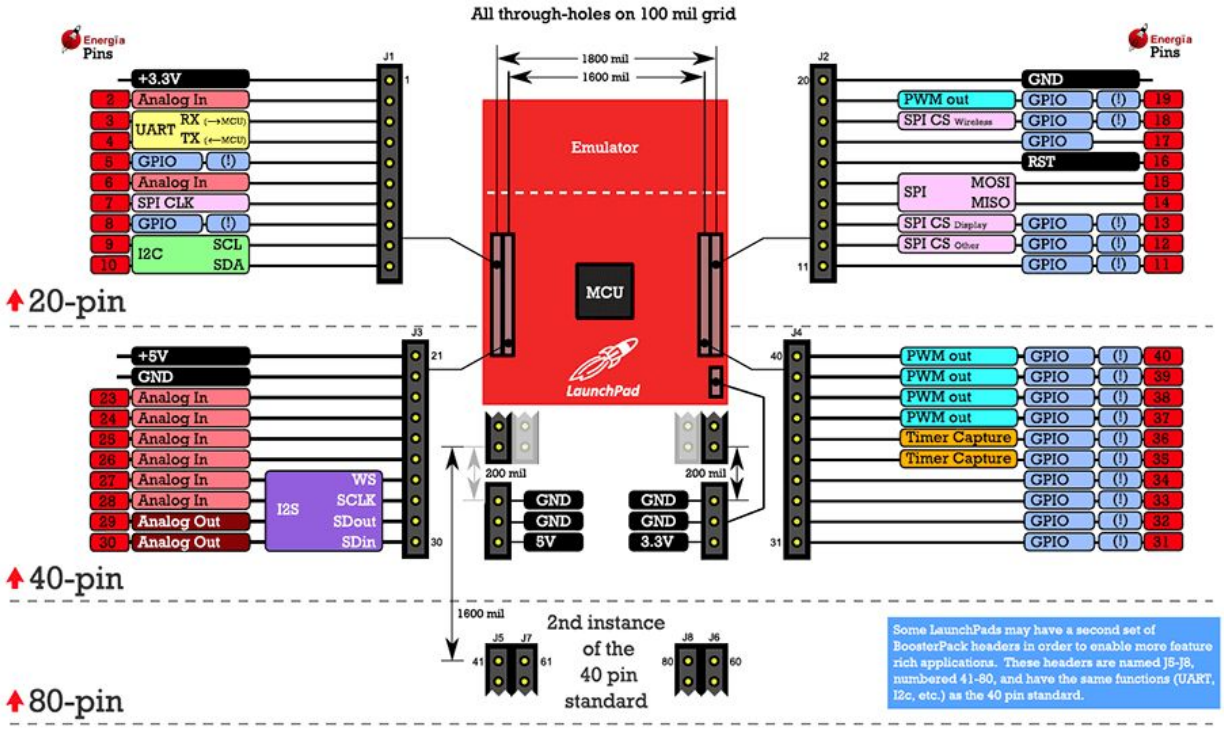
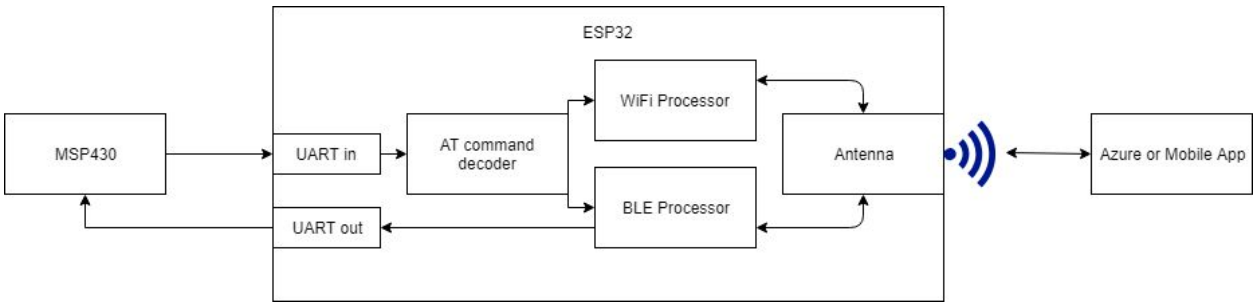



*Figure 13 - BoosterPack pinout*

**ESP32-WROOM-32**




The Wi-Fi and Bluetooth module that we have chosen for this project is the Espressif ESP32-WROOM-32. This module provides Wi-Fi, Bluetooth and Bluetooth MCU capabilities. It includes key components for the printed circuit board design such as a receive amplifier for low-noise, antenna switch, and different filters for different signals.

The ESP32-WROOM-32 provides different power modes and consumes less power than other chips. The sleeping current is less than 5μA and in some applications, the ESP32 can be active only when it is needed. It has two CPU cores which can be controlled individually and the

clock frequency of the CPU ranges from 80 MHz to 240MHz. The duty cycle of the clock is low so that the energy being used will be reduced.


*Figure 14 - ESP32-WROOM-32 SMD package*

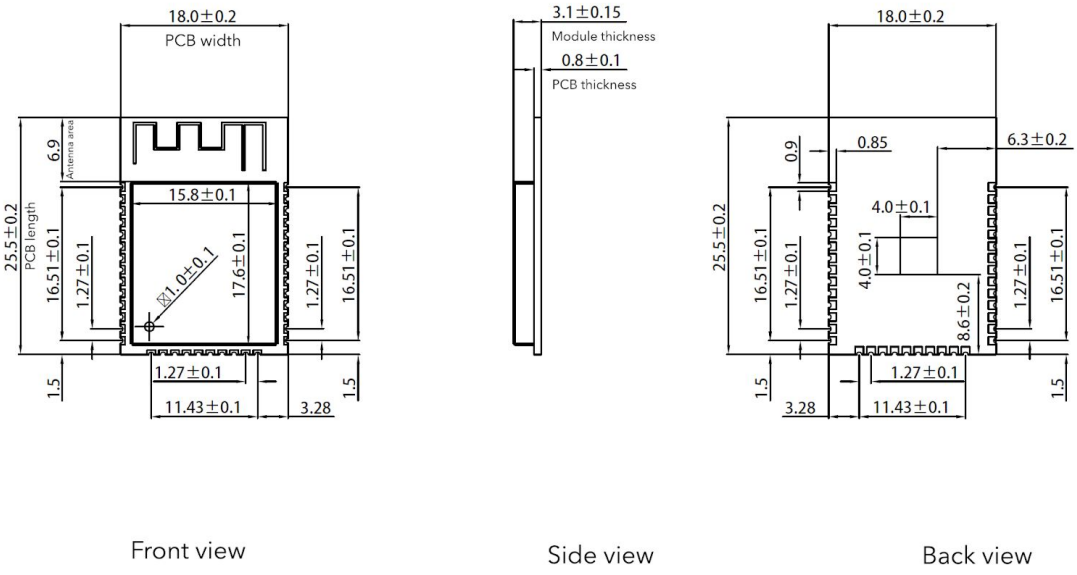



*Figure 15 - Detailed physical dimensions of the ESP32-WROOM-32*

The ESP32-WROOM-32 operates under a voltage between 2.7V to 3.6V under the average current of 80 mA. It has 38 pins including five strapping pins. The five strapping pins are MTDI, GPIO0, GPIO2, MTDO, and GPIO5. Those strapping pins allows the software to read the value from the “GPIO_STRAPPING” register.

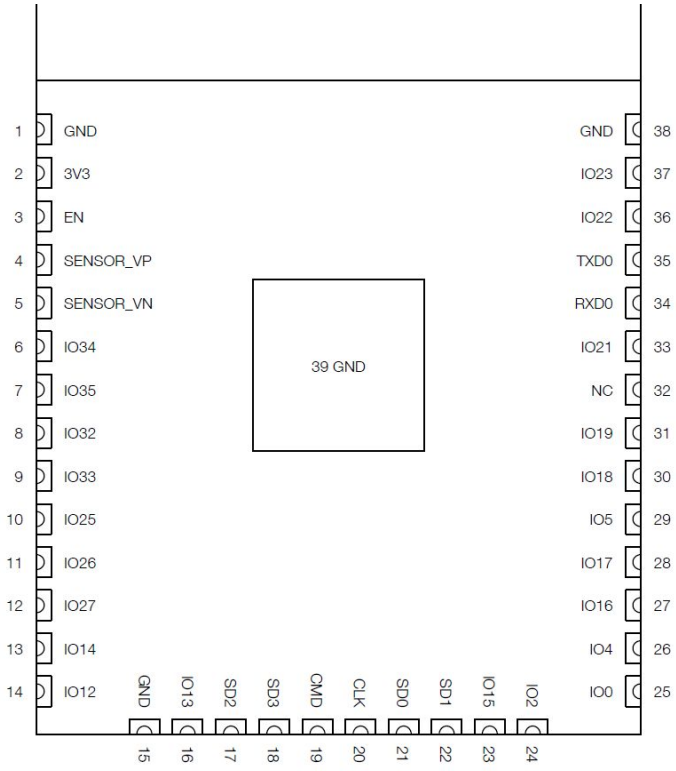



*Figure 16 - ESP32-WROOM-32 pinout*

For more detail, the schematic of the ESP32-WROOM-32 is shown below.

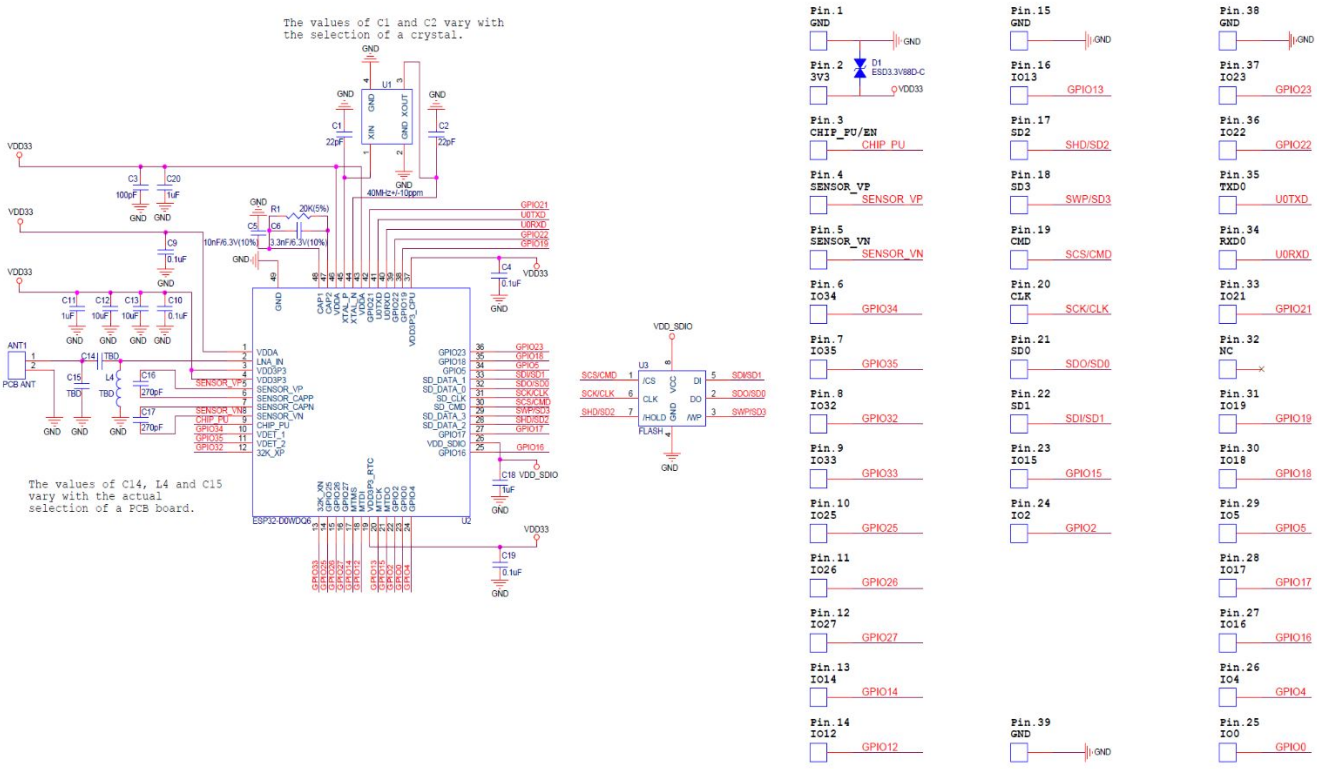



*Figure 17 - ESP32 detailed schematic*

Regarding the Wi-Fi portion of the chip, the standard is 802.11 b/g/n and it supports 2.4 GHz band to 2.5 GHz band. The speed of transporting the data can reach up to 150 Mbps and the output power at antenna can reach up to 20.5 dBm. The security types which are provided by the software of this chip are WPA, WPA2, WPA2-Enterprise, and WPS. Those can help the future developer who use our design to protect their data privacy.

The Bluetooth protocols which is provided by this module are Bluetooth v4.2 BR/EDR and BLE specification. For the radio signal, it provides NZIF receiver with -97 dBm sensitivity and uses transmitter with class-1, class-2, and class-3 power class capabilities. In terms of the audio signal, it provides CVSD and SBC.

Our product requires a cloud service for the client to view data from the sensors on the board. We need a cloud service that is able to receive data from our ESP32 and display it on a PC. We would prefer that our cloud service is simple and quick to set up and operate, as well as offers good online documentation and resources. There are many cloud service providers that are currently available to the IoT market. Three of the most popular companies are Amazon Web Services, Microsoft Azure, and Google Cloud. Each of these companies have multiple solutions on the market directed at IoT developers. Our ideal solution would be cost effective, simple to implement on a PC, and would be well documented to work with the ESP32.

*Azure IoT Hub*

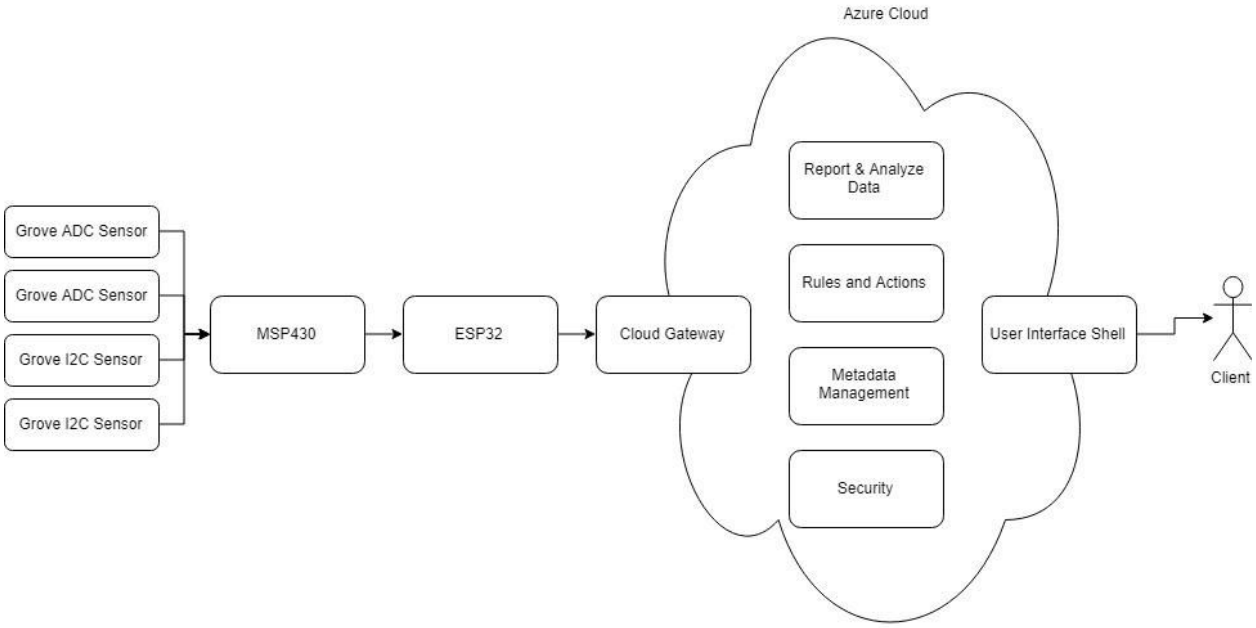
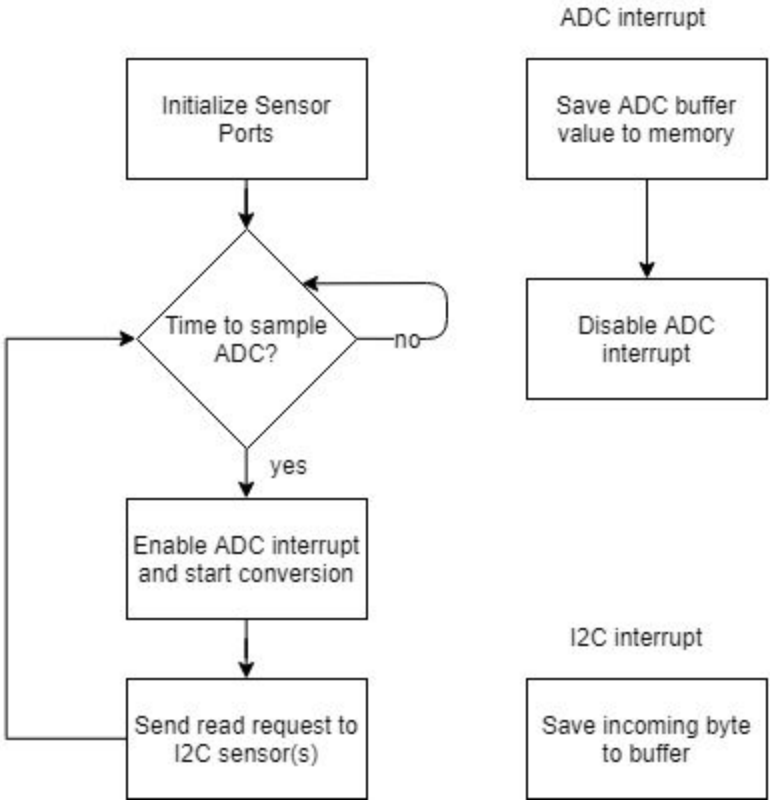



*Figure 20 Azure Cloud and Incoming Data*

As far as how the cloud works, we are most interested in incoming data, so that we can assure the client is receiving accurate data reports with our software. Data will come in from the sensors to the MSP430. The MSP430 will send the data to the ESP32, where it will be transported through WiFi to the cloud gateway. There, IoT Hub will handle reporting and analyzing data, rules and actions regarding data that will be set by the user, as well as metadata management and security. IoT Central's user interface shell is an HTML5 application that allows the user to see graphs and charts relating to the incoming data as well as set different rules and actions.

# Software Design Documentation

**Sensor Software**



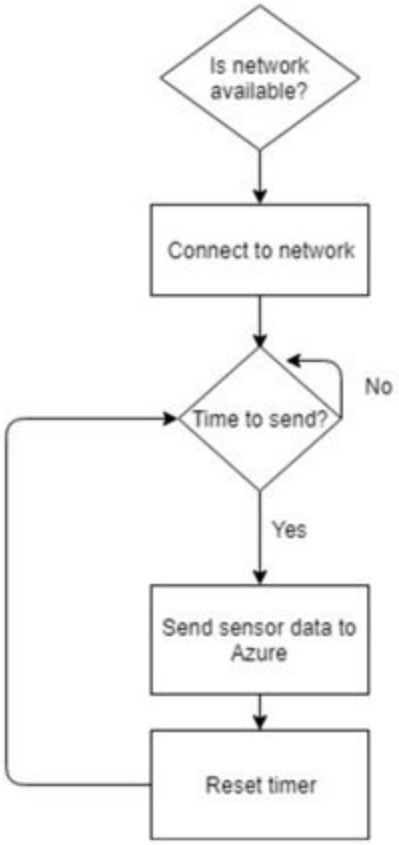

Our sensor software will consist of a simple timer-driven interrupt pattern. When the period given by the specified sampling frequency has elapsed, the MSP430 will exit low-power mode and enable the analog-to-digital conversion and I²C interrupt functions. Within these functions, all the sensor data will be saved to memory (FRAM) before they are ready to be transmitted.

This type of interrupt-based algorithm will allow the device to stay in a low-power mode for as long as possible, saving battery life if a battery is used. In the MSP430's active mode, it draws 120µA/MHz. At the default 8MHz, the current draw is 960µA, making the power draw 3.186mW (P = IV where V = 3.3V). When in low-power mode 1, the current draw is only 40µA at 1MHz, making the power draw only 0.132mW, or 4.14% of the active state power consumption.

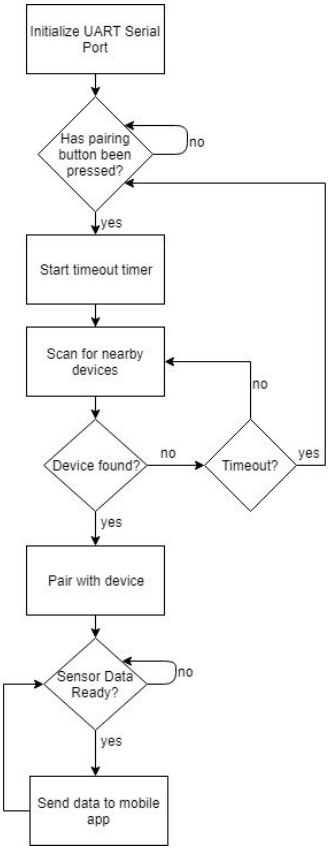
The power savings become even more apparent when comparing the different operating modes of the ESP32. When active, i.e., transmitting, receiving, or listening, the ESP32 will draw up to 240mA typically. That translates to 0.792W of power, which is significantly more than any other device on the board. In the ESP32's light-sleep mode, which it would be kept in unless actively transmitting, it draws 0.8mA of current and thus 2.64mW of power. This is only 0.33% of the active power draw, making the incorporation of switching between states critical.

Networking Software

The following flow chart shows how the Wi-Fi will work in the design cycle. The ESP32 will first detect if the specified network is available. If the network is available, it will connect to it and receive signal from user devices. When it is time to send the data, it will send the data to the cloud service, Microsoft Azure. The following flow chart shows how the Wi-Fi software will work in the design cycle.




In our design, the user needs to press the pairing button and then the chip will have a scan for devices. When the device is detected, it will try to connect to the user's available equipment. If the device is not available at that time and the connection time is out, it will go to the initial state and wait for the button to be pressed. On the other hand, if the device is available, it will pair to the device. After successfully connected to user equipment, it will receive the data from user device or sensor and when it is the time that the data needs to be sent, it will send the data to the phone. The following flow chart shows how the Bluetooth software will work.




*MQTT Protocol*

Messaging Queuing Telemetry Transport, or MQTT protocol, is a very popular IoT protocol that we will be using for both WiFi and Bluetooth communications. It consists of a broker, a server that collects and distributes data, and clients, devices that collect or receive data, in our case the mobile app, cloud service, and sensors. The type of data collected by a sensor is called a topic. Clients like the mobile app and cloud service will ‘subscribe’ to topics, and the sensors will ‘publish’ data. The broker handles where published content goes.

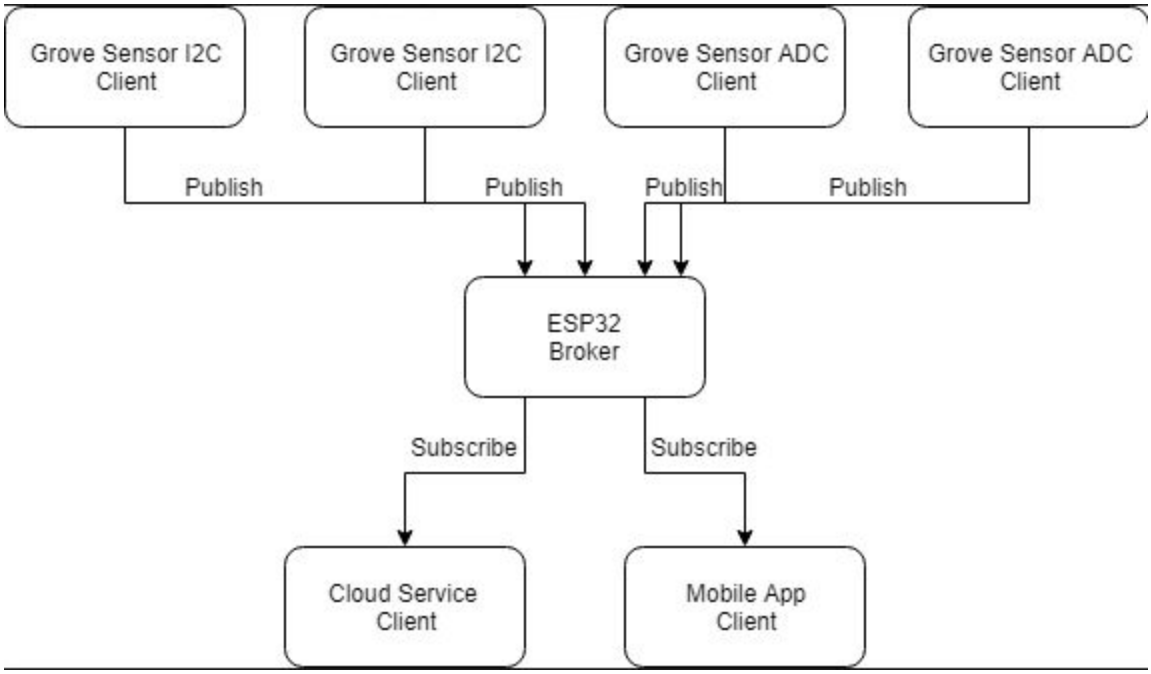
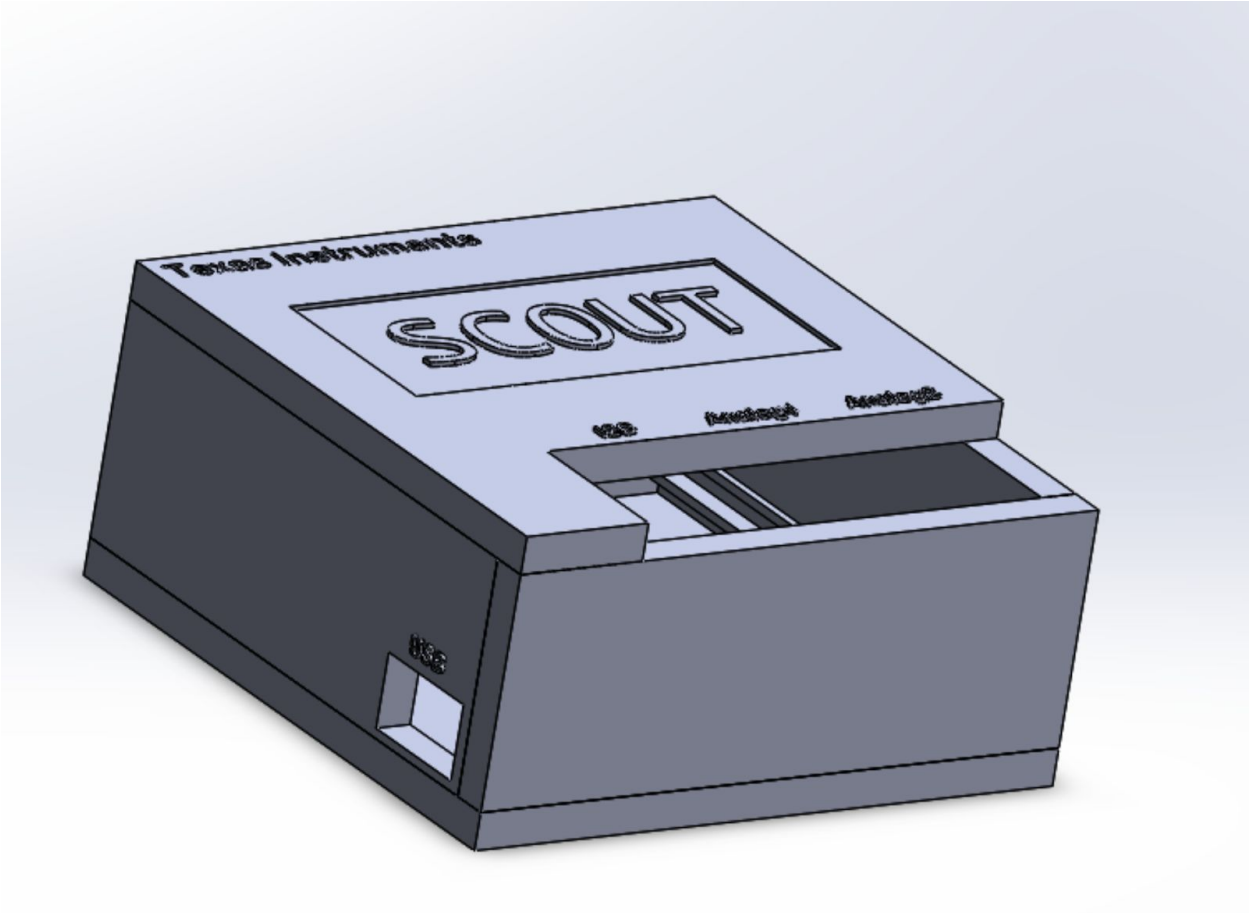



*Figure 21 - MQTT flowchart*

# Enclosure Design




# Conclusion

In this document, we have outlined the main systems that will support the Scout IoT DevKit. This kit is meant to be simple for the user to use and turn into a functional product. Thus, the Scout must prove to be a reliable system that is thoroughly documented and open source. The kit consists of a microcontroller and Wi-Fi/Bluetooth module, 3 Grove sensor connectors and Texas Instruments BoosterPack capabilities. These core features will allow the consumer to easily plug in their own sensors and additional BoosterPacks, and use our code library to easily read sensor data and set up two way communication through Wi-Fi and BLE. The consumer will be able to create their own IoT solutions without having to build from scratch, while still  further accelerating the growth of the IoT market.