Abstract

The purpose of the Ferris IOT security system is to create a modifiable, scalable, all-in-one cloud security package for indoor surveillance, monitoring and security. The system consists of a modifiable base station that can hold up to three modules that allow for different functions, such as remote viewing, weather environment sensing, or an alarm. You can mix and match up to three modules on the base station at once, and even have duplicates. Users can install multiplatform management and viewing software on virtually any desktop or laptop computer, and use these devices anywhere in the world with our homebrew cloud solution that prevents the need for difficult setup and configuration. The entire system is plug and play, and our cloud service manages authentication, data delivery and data integrity so you don’t have to.

Table of Contents

Base Station Design

Base Station Background

The first thing that was done in regards to the Base Station was preliminary drawings by hand to determine what design we should follow. Our original preliminary design had the option to have up to three modules connected to the Base Station at a time. The Base Station would house the Raspberry Pi. It was determined that 3-D printing the Base Station was the best option to get the custom components we wanted. The 3-D rendering software that was used was Solid Edge.

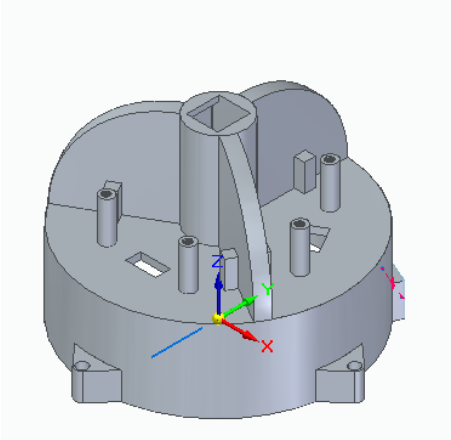
Base Station Preliminary Drawings

The preliminary drawings of the Base Station were created to determine as a group what design for the camera would work the best. All the original designs featured a Base Station that was circular and had a dome shape. The original design was circular with four individual modules that could be connected to the Base Station. One of the original designs was quickly ruled out because this particular design featured the camera as the center of the Base Station, with the modules connected around it. The reason why this design was ruled out was that the camera module was a different shape than the other modules. This aspect of this design violated our project's scope because we needed each module to be able to be placed anywhere on the Base Station where a module could be placed. This Base Station would then have holes where screws would go to secure the modules to the base station. .For the final preliminary design, the Base Station could fit three different modules. The Base Station has a center pillar with dividers to separate the different modules. These dividers and center pillar allow the modules to have sturdy support to rest against. There will be pillars underneath the Base Station where the Raspberry Pi will be secured. There will also be slots where the USB-C connectors attach to the Base Station where the modules are plugged in to communicate with the Raspberry Pi.

Design Stage

Designing the Base Station in Solid Edge

The hardware team researched different 3-D design software to determine which one was best for our application. A Ferris State University faculty was contacted to help in this decision-making. The 3-D software that was used to design the Base Station was a program called Solid Edge. This program was very helpful in designing the Base Station it was very similar to other 3-D design software that was familiar in the past to the team. However, some research was needed to help with some aspects of the project. This included how to make threads in the screw holes and how to create an assembly file to determine how the pieces fit together.



Final 3-D Modeling of Base Station

Base Station Design Changes

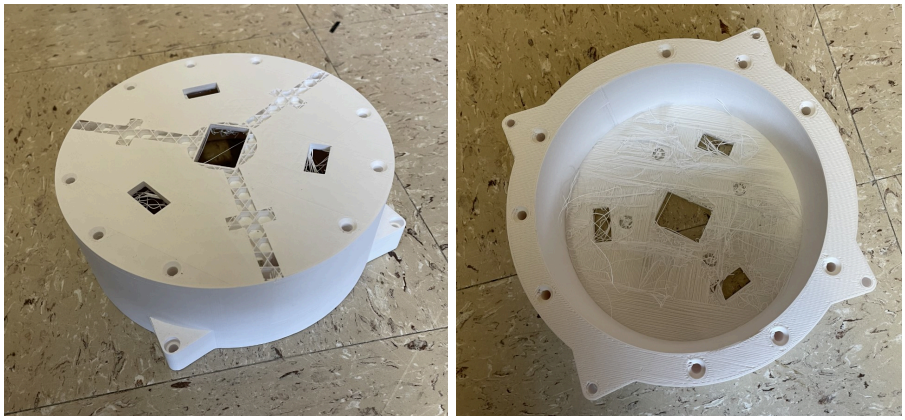
When designing the Base Station, there was a design change that was implemented. This design change was that the PIR sensor would be permanently attached to the Base Station instead of being in its module. This change did not require a whole lot of time changing the design. It was easily implemented to where the PIR sensor would be located at the top of the center pillar of the Base Station. The wires from the PIR sensor would run down the inside of the center pillar and attach to the Raspberry Pi. Another change that was implemented was there were four external mounts added to the Base Station so the device can be mounted. Another change that was implemented was the thickness of the mounts that would hold the Raspberry Pi. The original size of these mounts was too thin and was prone to break. The size of the mounts was made thicker and shorter to create a stronger and more stable base. When checking over a design, it was realized that it would be beneficial if the Base Station and the

modules have slots to allow for the holes to line up easier when securing the modules to the Base Stations with the screws. This also provided an easier method of holding the modules in place.

Printing

3-D Printing the Base Station

When 3-D printing the Base Station the Makerspace at Ferris State University. They allowed us to 3-D print the Base Station for free. The type of program that was used to 3-D print the Base Station was the program Ultimaker Cura to slice the 3-D file of the Base Station. The 3-D printers that were available were Ultimaker and Prusa. For the final Base Station the total print time for the part of two days, thirteen hours, and forty-seven minutes. The material that was used to print the Base Station was called PLA which stands for Poly Latic Acid. This is a generic material that is common in 3-D printing. Some challenges were experienced when trying to 3-D print the Base Station. The first problem that was encountered was the prints came out smaller than what was designed. They were 25.4 times smaller than originally designed. This issue took about two weeks to figure out. It was determined that even though the file the Base Station was in a metric file when the file was opened in the Ultimaker Cura software to print it was in imperial. This problem was solved by multiplying the dimensions by a factor of 25.4. Another issue that occurred during the printing of the Base Station was the printing process failure. The Base Station came out incomplete. This print became unusable because the dividers and the center pillar were not printed. The print overall just did not turn out as nice as it could have. There was no definitive solution to this problem. Fortunately, this issue did not arise again. It was believed that some external factors contributed to this failed print because all the other prints that were trying to be completed that weekend failed. A mistake was made during the design stage of the Base Station which caused the Base Station to be printed again. The issue was that the slots and the inserts of the Base Station and the modules were made the same size so they would not slide in. Sanding down the holes and the inserts were considered, but for simplicity's sake, a new Base Station was printed.



Images of the failed print

Assembly

Assembling with the Base Station

When assembling the different components to the Base Station some problems occurred. One issue that was presented was the ethernet cable did not fit in the slot that was originally printed. To solve this issue some filing was done to ensure the plug fit in the slot provided. Drilling was also required to create new screw holes to secure the plug. Another issue that was presented was the screw holes did not print very clean so there was some material blocking the hole. This issue was solved by drilling out these holes to clean up that extra material. Upon further expedition, it was discovered that these holes did not have clean threads to accommodate a machine screw, so a screw with bigger threads was used to allow the screw to grip the inside of the hole. The holes for mounting the Raspberry Pi became an issue because for some reason the printer did not print them. This problem was determined to be caused by the size of the holes and the supports that were needed to print the Base Station due to it being hollow caused them not to print. This issue was resolved by drilling out these holes with a drill. The supports themselves were oriented wrong to where the snap-in USB-C mounts would get in the way of the Raspberry Pi. This meant another Base Station had to be printed to correct the issue.

Image showing the incorrect orientation of supports for Raspberry Pi

During the assembly stage of the Base Station general sanding and filing occurred to ensure that the PIR sensor fit. Also, sanding was required because some rough spots occurred during the printing process. This allowed the Base Station to have a clean and even appearance.

Conclusion

Conclusion for Base Station

Overall, the Base Station came out well and it met all the criteria that were required based on the scope of the project. Some aspects that would be changed in the future would be that the Base Station would have more ventilation to help disperse the heat created by the Raspberry Pi. Another change would be that find a better method and material to produce the Base Station other than 3-D printing because it would allow the threads of the screws to be cleaner as well as make the Base Station more durable.

Module Design

Module Background

For the modules that will connect to the Base Station, the main idea is that the modules are similar enough to where they can be placed on the Base Station anywhere in an available slot. Also, the main idea for these modules is that they can be screwed to the Base Station securely. Each module would house a different capability that could be added to the Base Station. The different module capabilities would be a camera, PIR sensor, and speaker. These modules would then be 3-D printed.

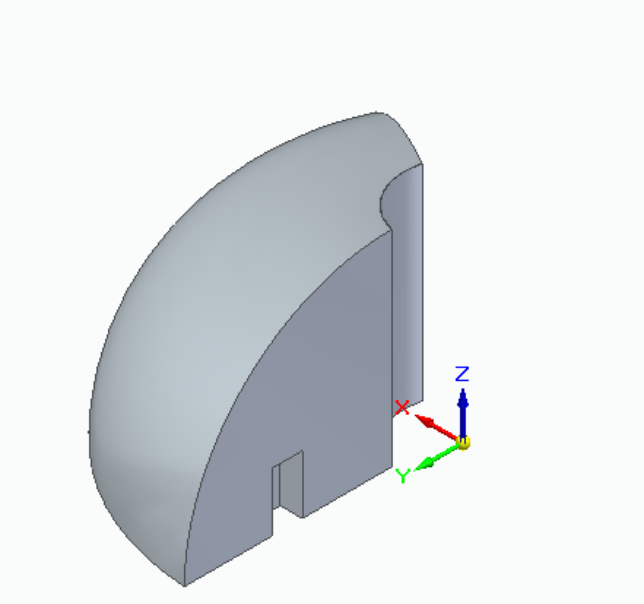
Modules Preliminary Drawings

Some of the preliminary drawings showed that the modules would be triangular-shaped with an arc bottom to match the curve of the Base Station. The modules would then have curved surfaces to provide an angle for the camera. When the modules are together the overall shape of the Base Station and the attached modules would be a dome shape. For the final module preliminary design, the modules would be a triangular shape with a circular arc at the base. However, a pillar was added to the center of the Base Station, so there is a curved indent to the top of the module this allows the module more points of contact to get a secure attachment to the Base Station.

Design Stage

Designing the Modules in Solid Edge

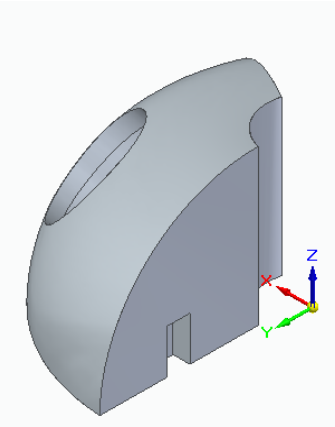
When designing the modules the 3-D software that was used was called Solid Edge, which was used to design the Base Station. What was key in designing the different modules was that there had to be a general shape the modules had to adhere to have the ability to be placed anywhere in the Base Station slots. This was achieved by designing a blank module. Which to then each blank module was given its unique characteristics.



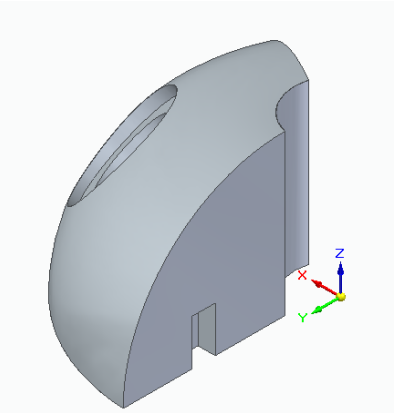
Blank Module

Modules Design Changes

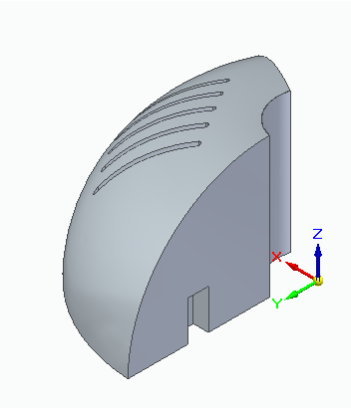
There was a design change with the PIR sensor module. The PIR sensor module will be not needed because the PIR sensor will be permanently attached to the center pillar of the Base Station. However, there was a module added to the list of available modules. A module that can measure weather aspects like temperature, humidity, and pressure. This module was simple to design due to the blank module already having been created. Also, another design change that was mentioned before in the Base Station section was there were curves added to where the module will slide into with rails on the Base Station to provide a nice secure attachment.



Camera Module



Speaker Module



Weather Module

Printing

3-D Printing the Modules

When 3-D printing the Modules in the Makerspace at Ferris State University. They allowed us to 3-D print the Modules for free. The type of program that was used to 3-D print the Modules was the program Ultimaker Cura to slice the 3-D file of the Modules. The 3-D printers that were available were Ultimaker and Prusa. For the final Modules, the total print time for the part was seven hours and 5 minutes. There were some of the challenges that were faced when trying to 3-D print the modules. The first challenge was the camera hole for the camera to be placed in was too small for the camera to fit into. The hole was too small to be filed or sanded to be effective. The solution to this problem was to print another camera module with a bigger camera hole. Another challenge that was faced was during the printing of the weather sensor. When trying to print the weather sensor it was discovered that the Makerspace did not have any printer with black-colored material. This would mean that the weather module would be a different color from the rest. The original color would have been clear. To remedy this problem the module was painted a matte black finish the same as the other modules.

Assembly

Assembling with the Modules

When assembling the modules the circuit boards for the speaker and weather module were attached to the different modules using double-sided tape. There was some filing and sanding done to the speaker module because the hole was not big enough to accommodate the wiring for the speaker. A groove was filed in the hole for the speaker to allow the wires to fit and make the speaker lay flat against the module. The camera module also needed some sanding and filing done to it because the hole for the camera developed an edge that did not allow the camera to fit once this edge was sanded down the camera was put into place. To attach the speaker and camera to their respective modules double-sided tape was also used. The holes in the modules lined up perfectly with the holes in the Base Station.

Conclusion

Conclusion for the Modules

Overall, the modules fulfilled all the requirements that were outlined in the scope. Some things that would be changed in the future would be that the camera module would allow the camera to be positioned. Another change would be the speaker module. The aspect that would be changed would be the placement of the speaker so that it is not exposed as much. This will protect the speaker from the elements.

USB Interfacing and Modules

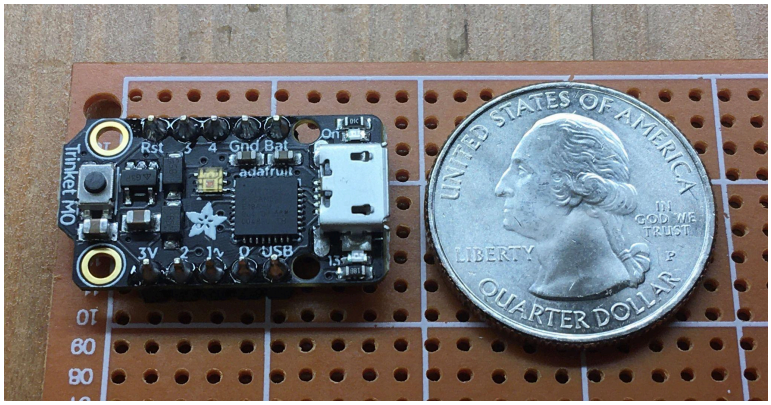
Introduction

The concept of the modules being user-swappable was central to the project's design, and it required a universal interface that all three modules could use interchangeably to communicate with the base station. Any module could be installed in any slot and would need to be automatically recognized by the base station and begin functioning immediately. This functionality was achieved using USB.

The camera module required little development in this regard, because it used an off-the-shelf USB webcam as a turnkey solution. The weather sensor and alarm modules, however, would require the development of their own USB interface.

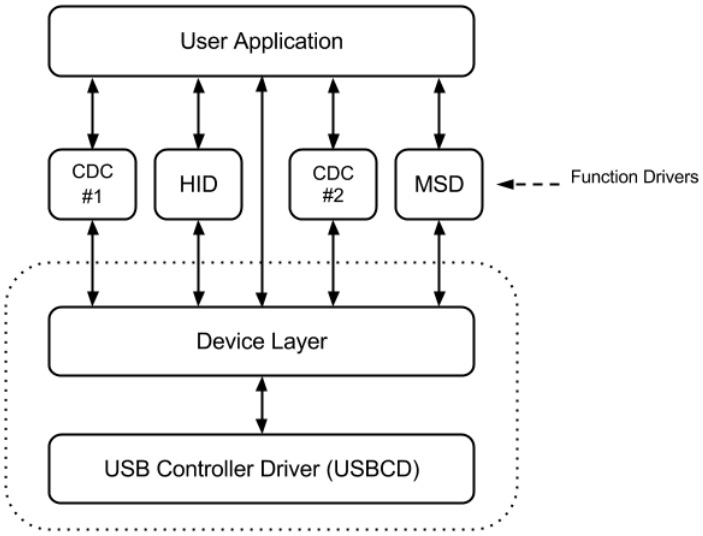
Implementation

The hardware chosen to implement this interface was the Adafruit Trinket M0 microcontroller board. It was an ideal choice for this task because it's tiny, containing only what is needed for this application and not much else. It has a USB micro B connector, an Atmel ATSAMD21E18A microcontroller (which contains an on-chip USB hardware peripheral), and access to a handful of the microcontroller's I/O pins. It has the ability to facilitate communication between each module's electronics and the base station, while supplying power to the electronics via the USB connector.



Adafruit Trinket M0 Microcontroller Board

Software for the Trinket was programmed in C and developed with the aid of Microchip's MPLAB Harmony framework, which includes a USB stack capable of implementing various USB device classes. This allowed the Trinket to be programmed to appear to a host as a USB serial device without the need to program USB device drivers from scratch. The architecture of the Harmony USB Stack is shown below.



MPLAB Harmony USB Stack Architecture

The function driver implements the chosen device class (CDC #1 in this case) and provides an abstracted interface for the application to access the USB device functionality. The function driver accesses the USB peripheral through the device layer, which is also responsible for responding to enumeration and control requests issued by the host, and is the only software layer with direct access to the USB controller driver. The controller driver manages the hardware state of the USB peripheral itself and provides the device layer with a structured method of accessing the USB hardware.

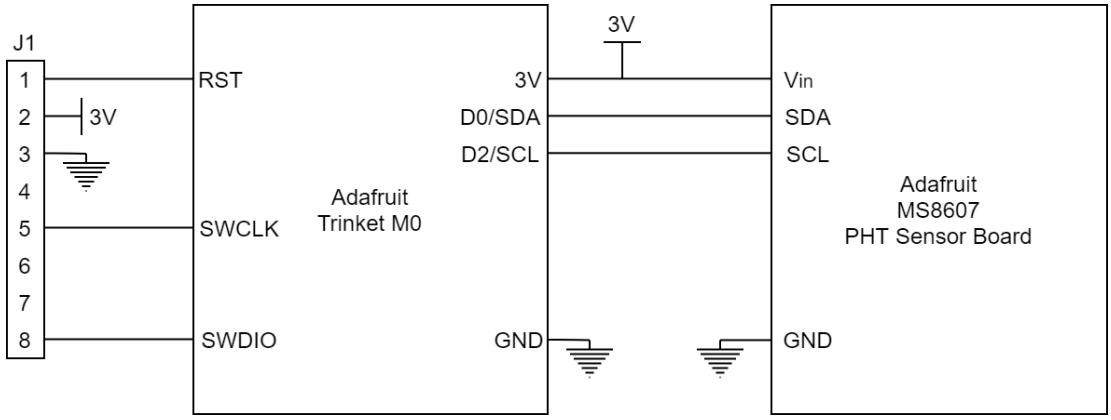
An application implementing the basics of the USB CDC device on the Trinket was created by starting with a USB example program provided by Microchip and modifying it to run on this board. The example program was simplified considerably, and some hardware-specific changes needed to be made, such as redefining I/O pins and USB endpoint addressing.

Once the application was running properly, the Trinket appeared as a COM port when plugged into a host PC, and serial data could be sent and received from the Trinket via a serial terminal. This provided a foundation for the USB interface to be used by the weather sensor and alarm modules.

Weather Sensor Module Design

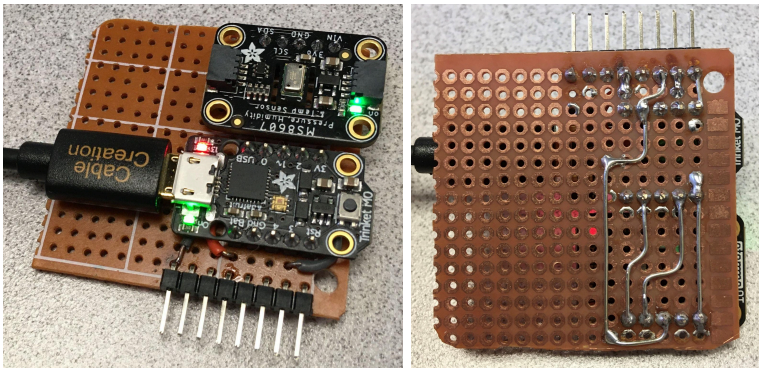
Weather Sensor Module Electronic Assembly

The weather sensor module needed the ability to regularly report pressure, temperature, and humidity values to the base station. The MS8607 PHT sensor from TE Connectivity provided a suitable solution for measuring these values, and Adafruit offers a breakout board containing the sensor and its supporting components, making it easy to add to the project. The electronics used in the weather sensor module are shown in the schematic diagram below. The J1 header is used to connect the MPLAB Snap programmer/debugger for programming the Trinket, and the MS8607 connects to the Trinket via I^2C. Pull-up resistors for the I^2C bus are included on the MS8607 board.



Weather Sensor Module Schematic

After testing the circuit on a breadboard, the components were soldered onto a piece of prototyping board, shown in the photos below.

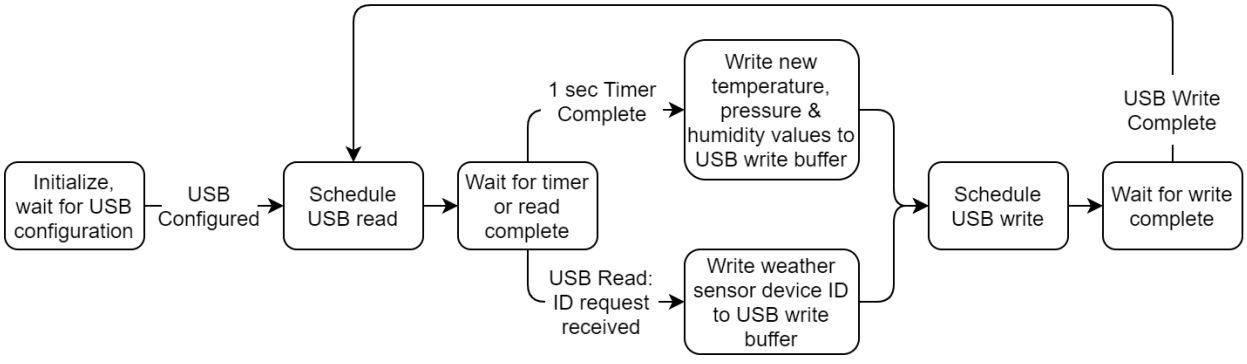


Weather Sensor Module Electronic Assembly

Weather Sensor Module Software

A generic C driver for the MS8607 sensor was obtained from TE Connectivity's GitHub page. The driver included several unimplemented I2C functions that needed to be defined to call the correct functions to invoke the I2C communication of the system. Once the I2C functions were correctly implemented, the driver provided a variety of useful functions for using the sensor. One of these functions was used to retrieve new temperature, pressure and humidity values from the sensor. The sensor includes factory-set calibration values stored in an internal EEPROM, which are automatically retrieved and used by the driver in the calculation of pressure, temperature, and humidity values.

The MS8607 sensor driver was combined with the MPLAB Harmony Framework's USB stack to create an application to run on the Trinket M0 that would regularly report updated pressure, temperature and humidity values to the base station. The application is shown in the state machine diagram below.



Weather Sensor Application State Machine

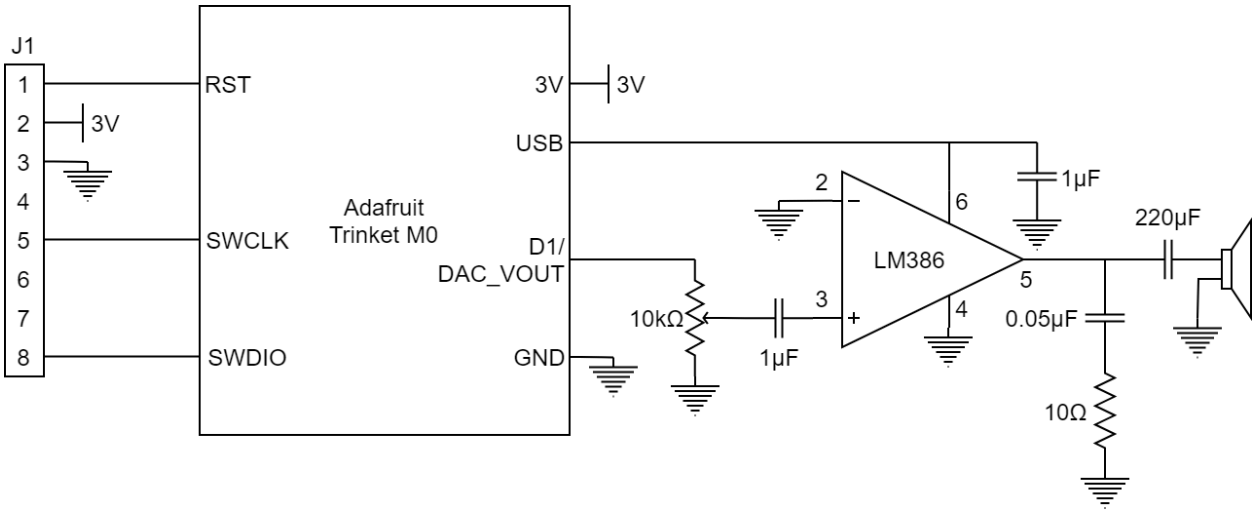
On startup, initializations are performed and the USB connection to the host is configured. When this is complete, the application schedules a USB read and then begins waiting for either the read to complete or for a 1-second timer to expire. The reason for scheduling the USB read is so the module can receive and respond to an ID request from the base station. The base station sends a ‘t’ to request for the module to identify itself, and when this request is received, the module responds with a ‘$1’ to identify itself as a weather sensor module. The application is structured so that a USB read is scheduled at all times, so the module is always able to respond to an ID request immediately. Typically the ID request is only issued once, when the system is powered up.

The rest of the time, the weather sensor module application just waits for the 1-second timer to expire, at which time it retrieves new values of pressure, temperature, and humidity from the sensor and writes them via USB to the base station. The values are sent with the following format: “[pressure in mBar] | [temperature in °F] | [relative humidity in %] \r\n”. The values are separated with the ‘|’ character so the string can be parsed and the values can be separated from each other.

Alarm Module Design

Alarm Module Electronics

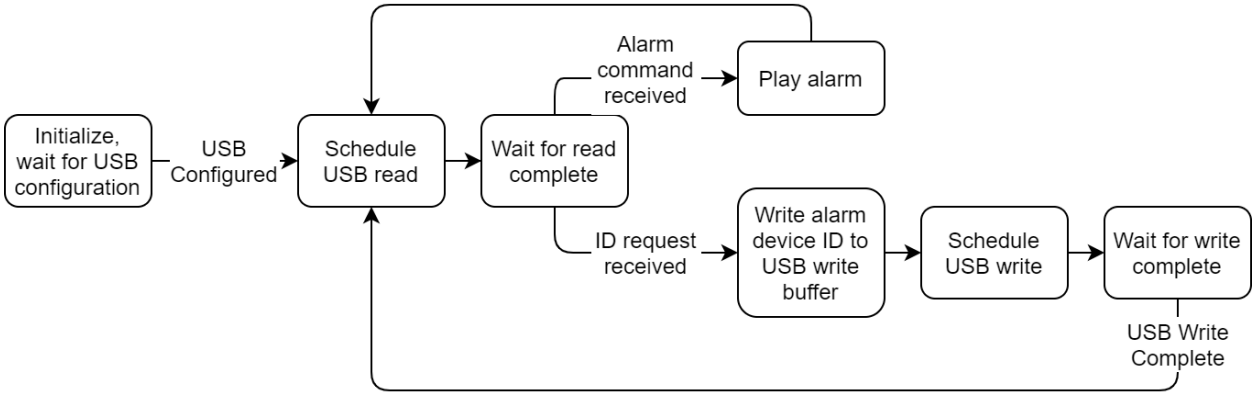
The electronics hardware in the alarm module were built around a Trinket M0 microcontroller board to provide the USB communication interface, similar to the weather sensor module. The ATSAMD21 microcontroller on the Trinket features an on-chip digital to analog converter, which allows it to output an analog waveform directly. This analog signal was connected to the input of an amplifier circuit which was used to drive a speaker and create an audible alarm. The schematic of the alarm module electronics is shown below.



Alarm Module Schematic

Alarm Module Software

The application that runs on the Trinket M0 in the alarm module is similar to that of the weather sensor module. It uses the MPLAB Harmony USB stack to handle USB communications. A state machine diagram depicting the operation of the alarm module application is shown below.



Alarm Module Application State Machine

Once the USB connection to the host is configured and a USB read is scheduled, the application waits for the read to complete. This occurs when the base station sends an ID request or a command to sound the alarm. The device ID request works the same as in the weather sensor module, except this time the module responds with a ‘$2’ to indicate it is an alarm module.

When the base station sends an ‘A’, the module interprets it as a command to sound the alarm. The alarm waveform is generated by writing a sequence of values from a lookup table to the digital to analog converter. A periodic timer interrupt occurs every 10μs to write the next value to the DAC. The waveform generated in this application is a sawtooth wave, which produces a harsh and attention-getting sound. The lookup table contains 100 values which make up one cycle of the waveform, so writing them to the DAC every 10μs produces a tone with a frequency of 1000Hz. The tone is turned on and off by toggling the status of a boolean variable, and this variable is checked by the interrupt routine to determine whether or not to write the next lookup table value to the DAC. When the alarm command is received, the tone is turned on and off seven times with delays in between, to create a beeping alarm tone.

Audio Amplifier Design

When originally designing this audio amplifier it was intended to use an LM741 to build the amplifier. The reason for using an IC amplifier instead of creating a multistage amplifier with transistors is due to simplicity and space within the module. However, during the testing of this amplifier, there were some design changes. The LM386 was used in place of the LM386 because this audio amplifier allowed the circuit to require fewer components as well as requiring a lower voltage than the LM741. The design itself was based on an example circuit that was in the Texas Instruments datasheet of the LM386. The only change that was made was a decoupling capacitor was added to the voltage input. The reasoning is mentioned in the Troubleshooting section.

Amplifier Testing

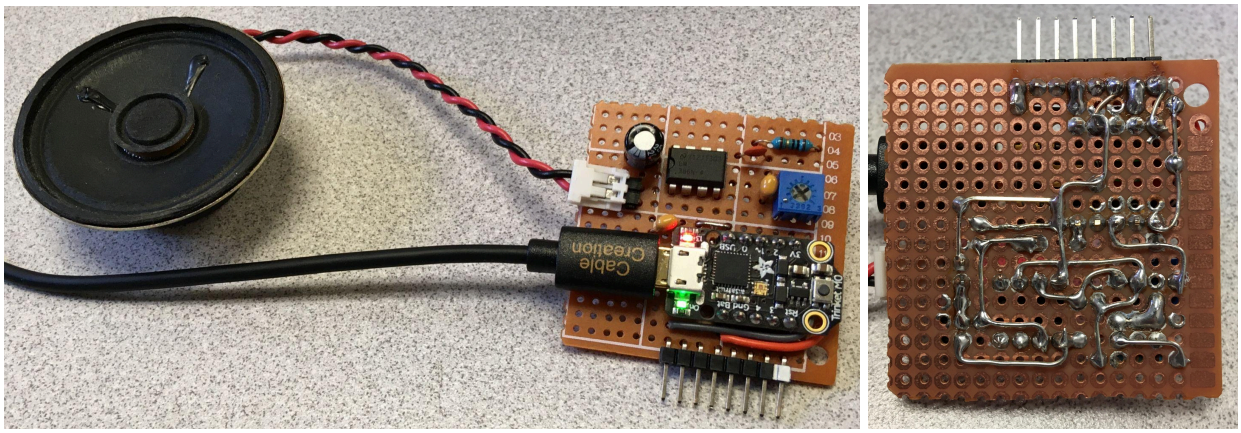
When testing this audio amplifier as stated before the LM741 was replaced with the LM386. This is due to the LM741 producing a very noisy output. It was believed that with the low voltage power the amplifier was not operating properly. Another factor is that the LM741 is not the best option when it comes to creating an audio amplifier. When using the LM386 it only required a supply voltage of 4 Volts, which was perfect for it to be powered by the Trinket M0. Also, the LM386 is designed to be an audio amplifier, so it produced a clearer output.

Amplifier Troubleshooting

When testing the audio amplifier even with the LM386 the output still produced a very noisy waveform. This problem was solved by placing a coupling capacitor at the positive voltage supply input of the LM386. This filtered out any noise from the power supply which would cause noise at the output.

Amplifier Assembly

The final assembly of the amplifier was soldered and placed in the speaker module. The assembled alarm module circuitry is shown below. The overall gain of this amplifier was around 21. The overall power the amplifier was using was 350 mW of power. During final testing, the amplifier's trim pot was adjusted to produce the loudest output possible without the amplifier clipping. To produce a louder output from the amplifier, a higher-voltage power supply may be needed. However, this could incorporate noise into the system, especially with the use of a switching power supply.



Alarm Module Electronic Assembly

Amplifier Conclusion

Overall, the audio amplifier met all the requirements that were outlined in the scope of the project. The only thing that would be changed in the future is to increase the volume of the speaker. The way this would be done is to add an external power supply to this module to increase the power of the amplifier. Another way to increase the output of the speaker would be to add more stages to the amplifier.

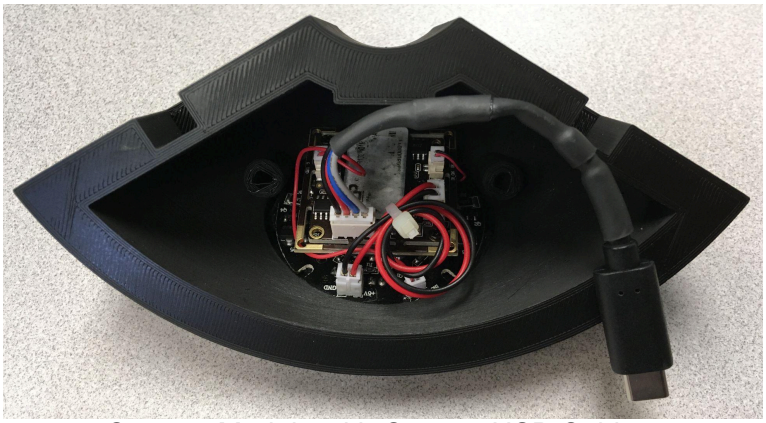
Camera Module Design

Camera Module Electronic Assembly

As previously mentioned, the camera module required very little hardware or software development, because it utilized an off-the-shelf USB webcam. The only major task necessary to integrate it into the project was to create a custom cable to connect it to the base station. The USB connection on the camera is provided by a 4-pin JST connector, which is not a standard USB connector. The camera shipped with a 3-foot JST to USB-A cable, which allowed it to be plugged into a PC, but it needed to connect to the base station's USB-C connectors, so a custom cable needed to be built.

The correct type of JST connectors were found on Digi-Key, and a USB-C connector was cut from the end of a USB-A to USB-C adapter. Getting the USB-C connector from this adapter was helpful, because its cable only contained the four conductors of a typical USB cable, instead of all the additional wires a USB-C cable would normally have. This made it easier to determine which wire was used for which pin of the USB connection.

The JST to USB-C cable was soldered together with a length of ~4 inches, with heat shrink tubing covering the connections. This cable worked correctly at first, but using it to plug the camera module into the base station forced the cable to bend sharply in the area where the wires were soldered, and soon the cable stopped working because one of its solder joints broke. The cable was disassembled and rebuilt, this time using a Western Union splice to join the wires, which involves twisting the wires together to form a mechanical connection before the joint is soldered, resulting in a stronger connection. The assembled camera module with its custom cable is shown in the photo below.



Camera Module with Custom USB Cable

Base Station Software

Introduction

The base station runs a custom-solution software package in order to communicate with the main server. The base station software handles USB sensors, cameras and alarms, and is responsible for taking a headcount of connected devices. The system is designed so that a variable amount of components can be initialized at runtime, therefore allowing people to leave certain module slots void, or install duplicate modules with minimal interference.

Design

Preliminary Considerations

When our group first started working on concepts, it was immediately apparent that we needed software to dynamically control components and allow people to access those resources. The server implementation at the time was in its infancy, so these platforms were developed in tandem and in close synchrony. In order to fulfill the scope of the project for what the base station can do, we had to complete the following:

1. Allow the user to view a live video stream from the base station via the internet
2. Supply weather data from the base station to the user over the internet
3. Allow the user to set off an alarm over the internet
4. Provide the ability for the base to send messages to users upon detecting motion

As well as these main functions, all actions should be authenticated such that rogue users of outside origin should not be allowed to modify or view these facets contained within the system.

Initial Design and Changes

Initially, there were versions of certain modules of this software that were considered for finalization by the software team, but were changed later. Those modules that were changed are included here in this section

1. PIR Sensor interfacing method
    a. Initially, the design was for the PIR sensor to be a separate module, able to be slotted in the base as a USB device.
    b. The PIR sensor is now embedded in the base station at the apex of the system, part of every system. The Adafruit trinket was removed in favor of using the GPIO pins on the embedded pi
    c. The software changed, where instead of preparing to detect another USB device and interface with that, the application now was responsible for reading voltage levels present on pin 24 of the GPIO stack present on the embedded pi

2. Video streaming
    a. Initially, we wanted to use an application to handle streaming from the device called Motion. Motion would launch a web server on ports 80 and 443, and used this along with a simple javascript applet to provide an HLS stream. This also handled all the complicated camera streaming dynamics, allowing the main server to avoid excess data ingress and egress from the cameras.
    b. This system was changed on the base, server, and client, as the ability to obtain an available connection to a specific port on a device behind uncertain network conditions. The following iterations of changes were tried and rejected:
        i. NAT-PMP (not always available, see networking section 1.0)
        ii. UPnP (not always available, see networking section 1.0)
        iii. PCP (not always available, see networking section 1.0)
        iv. TCP-Hole punching (common firewall/security renders impossible, see networking section 1.1)
        v. UDP-Hole punching (only works on some hardware, implementation out of time scope, networking section 1.2)
    c. The finalized form of video streaming utilizes RTSP (Real time streaming protocol) with FFMPEG running on the base station to gather, encode, buffer and publish stream data to the main server. FFMPEG is called via a java wrapper

interface that assembles an instance of FFMPEG with specific arguments, is then started, and is placed in a separate thread pool.

3. Base station authentication and traversal
    a. Initially, there was no consideration for authentication and security. This addition affects the main server more than the base.
    b. During preliminary testing and development, HTTP was used to traverse data from base to server, and there was no way to be sure that the base station was authentic to our records.
    c. The finalized form of authentication traversal includes the usage of HTTPS/TLS1.3 to encrypt our data end to end, and each base station would come with a serial number and authenticator, where the authenticator used as a ‘password’ for the base station, thus preventing base station spoofing.

4. USB Device detection
    a. Initially, USB device detection was done with this library, and parts of this library are still used, but its reliance has been decreased drastically.
    b. During the stage where we needed a way to detect USB devices and interact with them, we made a registry of valid devices based on vendor ID and product ID.
    c. Device detection was changed to using v4l2-ctl, a command line tool available in most linux distributions.

Implementation

Functionality

Described below are all of the systems onboard, providing a service or function to users or the main server

Status reporting:

During the lifecycle of the camera, a status code will be sent to the server via HTTP request, along with a few parameters. The status code is formatted in this way:

<num_cameras><num_weather-sensors><num_alarms><storage_status_code>

An example code would look like: 021M where

1. The first 0 indicates no cameras are connected
2. The 2 indicates 2 weather sensors are connected
3. The 1 indicates 1 alarm connected
4. The M is the approximate storage used. M is the 13th letter of the alphabet, therefore (13/26 * 100 = 50) percent of the total memory is used.

This status is compiled every 5 minutes and sent to the server. If the server sees no status for more than 10 minutes, it will assume the base station is offline.

PIR Sensing

During normal operation, the base station will be looking for movement, and report this data via MQTT. When motion is detected, the base publishes to the main server under the topic “<base_serial>-M”. The QOS for publishing is 0 due to how clients currently handle and request this data. The PIR sensor is powered by the GPIO pins on the pi, and communicates on pin 24. In Java, we use a library called pi4j in order to interface with the Raspberry Pi.

Weather Reporting

When a weather sensor is installed, the base station communicates with the device to determine what device class it is. Weather sensors are USB Serial devices, so we send a ‘t’ character to the device and wait for output. After we identify the weather sensor, the weather

sensor provides constant data, outputting weather in a formatted string of <pressure_millibar>|<temperature_farenheight>|<humidity_percentage>. A timer task has been set up to read this and upload the value to the server after authentication where the value can be retrieved by the user later.

Camera streaming

A standard USB camera can be installed into the system to allow for remote video streaming. When the base station starts, it scans for available cameras to connect to and adds them to a list, then dispatches threads that use FFMPEG to begin and work with streaming. Streams are separated by designation, where camera 1 would show as S0, camera 2 shows as S1, and so forth. FFMPEG is configured to publish to the main domain, triagecore.com, with port 8554.

Alarm Systems

When an alarm is installed, it is initially found to be a USB serial device. After the base station connects to the main server on port 1883, alarm devices are queried for by sending them a ‘t’ over serial and waiting for the output it gives back. After device registration, sending a capital A to the device over serial activates an alarm for 6 seconds. The alarm module allows the base station to use a service called MQTT, or Message queueing transport telemetry. The base station dynamically subscribes to a topic consisting of its serial number followed by the constant “-A” for Alarm. Upon receiving any message on that topic, the alarm will activate. The reasoning for using MQTT relies on the fact that it was a simple way to allow for a device to receive messages from another device, not requiring the end user to have a direct route to the device. Instead, the base station forms a link to the server and listens for data. Given MQTT’s open existence, it would have been futile to reinvent the wheel and not use it.

Communication Protocols

Described here are all the libraries or modules used to provide outward communication with the cloud service.

HTTPS, TLS1.3

- Used for authentication information traversal, status reporting, and weather reporting
- Communicates via port 8000 at the triagecore.com domain
- Utilizes a self-signed certificate to verify authenticity of the domain
- Uses TLS for end-to-end encryption

RTSP

- Used for publishing camera stream data to the main server if a camera is installed
- Uses port 8554

MQTT

- Used to read and write simple data to the server, leveraging an event based system such that users can interact with devices, even without a direct route to the device
- Communicates on port 1883
- Used for PIR sensors, and alarm triggering

Other Software Used

Operating system:

- Ubuntu ARM 20.01 LTS

Camera streaming software

- FFMPEG

Java Runtime

- OpenJDK 16

Conclusions

Base Station Software Conclusion

After finalizing all components, the software that was deployed works well and covers all needed areas of function. The program is fairly lightweight and poses no stress on the hardware or network. All functions work autonomously and require no user input, and therefore the software package meets all requirements of the project.

Security Server

Introduction

The server platform is a cloud based service where data is remotely handled and dispersed to the appropriate clients and devices. Its entire purpose is to tie in the user experience such that connections are seamless, data is available constantly, and most critical data is handled *for* users rather than *by* users. The server platform should provide all methods of data egress and ingress, to and from the base station(s) to the user(s). The server package consists of three main components, being the custom software by the name of SecurityServer.jar, a docker installation hosting a container running RTSP-Simple-Server, and a standard installation of the Mosquitto MQTT broker server.

Design

Preliminary Considerations

When deciding what the server should do, there are 5 main functions and datatypes users and base stations should be able to pull from and push to the server.

1. Camera streams
2. Alarm events
3. Motion events
4. Weather data
5. Authentication services

Weather data and authentication should be user request based. Users will not consistently need this data and can ask for it when it is needed. Alarm and motion should be listener based. We do not know when the user may receive this data, as this random nature requires us to form a listening channel to the server. Camera streams should follow a subscribe/publish format, where the server holds the buffer for the stream and users can ‘tune in’ to the stream, or essentially a ‘one to many’ distribution schema. We do not know when users are going to tune in, but we do

know when cameras are going to publish, so this system requires us to have highly available data.

Initial Design and Changes

Listed here are some changes made along the path of development. Some things were tried for finalization, but never made it through due to the reasons below.

1. Connection binder service
    a. Originally, we created an API where users would ask the server for a connection, and base stations would listen for these requests and prepare for NAT hole punches. The server would provide addresses and ports of the users and bases to each other, and a connection would be attempted.
    b. After finding how routers may or may not mistreat flows after a socket closes, UPnP or any other constituent port openers may or may not be available, and how most restricted networks would simply filter the hole punch efforts regardless of the method, the system changed to a request system instead of this binder system.
    c. The usage of RTSP and MQTT were made paramount to the project after this change was made.
2. HTTP and data traversal
    a. Initially, there was no consideration for authentication and security. This addition affects the main server more than the base.
    b. During preliminary testing, HTTP was used without encryption, leaving login data susceptible to people intercepting the traffic and gathering information easily.
    c. The finalized form of authentication traversal includes the usage of HTTPS/TLS1.3 to encrypt our data end to end, and each base station would come with a serial number and authenticator, where the authenticator used as a ‘password’ for the base station, thus preventing base station spoofing.

Implementation

Functionality

Described below are all of the systems onboard, providing a service or function to users or the base stations.

Authentication

During normal operation, the server must readily accept requests to authenticate from any source, and discern if the request is valid or not. The chain of events are described below:

1. User requests authentication via HTTP post with header key “request” and value “authentication”, and body format as <username>|<password>
2. Server sanitizes both inputs and gauges if the correct format is used. If the correct format is not used, discard the request and fail to respond.
3. If the format is correct, proceed to parse the information and perform a database lookup of the matching credentials.
4. If no match, respond with status code %INVALID and close the connection
5. If there is a match, generate a new session key with the current time and user ID, store it in a map, and respond with HTTP code 200, and body filled with the session key. The session key has 5 minutes before being deleted, unless it is kept alive via subsequent requests.

Status Management

While base stations are present, a status will be uploaded to the server every 5 minutes, and when the base initially boots. The server stores this value so long as the request to do so has the correct known base station authenticator included. The program stores this in the standard table containing all relevant info about the base stations. The status is stored as a string, at most 4 characters long.

Weather Management

If a base station has a weather module installed, it will consistently report the weather in the form of HTTP POST requests. The last best weather value is stored in the same database that contains all relevant information about the base station as a string. These values are also cached in a map object on the server, where, in the event of a request by the user, they are pulled from to avoid excessive usage of the database.

Camera Stream Management

The server platform does not natively handle streams within java, as this would be like reinventing the wheel. Instead, this program relies on a companion program that works async from the main server platform. The program runs in a container for convenience of management and deployment, that being under Docker. The application image running is called RTSP-Simple-server. Standard settings are used, such as port configuration, however a special configuration for authentication was created, where a simple command override was made, where the path to the authentication server was pointed to the host machine on port 8000. This allows the server to ‘ask itself’ for authentication and therefore saves an immense amount of dev time. The RTSP server, upon receiving a request to watch a publishing stream, will send the path, username, and password to the main server’s primary program. Upon receiving a 200 OK, RTSP Simple server begins relaying data to the requester.
Base stations can indiscriminately publish to any path they desire on the server, however they are only programmed to publish to a dynamic path, that being a function of their unique identifier and the camera number, signified by “S#” with # being a number from 0-2. 0 Would be camera 1, 1 would be camera 2, etc.

PIR Motion Events

The main server does not have a closely integrated MQTT server, and instead relies on a service called Mosquitto for the processing of MQTT information and events. In the case of PIR events, the client will receive QOS-0 tier messages, generated by the base every time a new motion event is created. Events are sent as messages with dynamic topics, and no message body. A user would subscribe to a topic automatically, with the topic being in this format: “<UUID_of_base>-M”.

Alarm Events

This is a system of hybrid interaction, where HTTP and MQTT are used. MQTT is not closely integrated, so this instead relies on the Mosquitto MQTT server installed alongside this program. Users can solicit the server with an alarm request, consisting of an HTTP POST request with header key “solicit” and value “alarm” with body consisting of the current session key, user id and base station UUID. If the authentication system validates that this user is congruent with its session key, that the base station UUID is owned by the user, and the session key has not expired, then an internal MQTT call is made. This message consists of the topic <UUID_of_base>-A, a QOS of 0 and no message body. If the base is connected, the base receives a call to activate its alarm.

Setup

Networking and Environment

The main server must be open to user and base station solicitation due to its nature as ‘the arbitrator of all that is’. In cloud network environments, the openness of the network and hardware is proportional to the functionality, and so a server in the cloud should have special qualities that we can otherwise not expect from users or the conditions their base stations exist in.

Some fundamental rules apply:

1. The server must be able to either, while behind a nat, be port forwarded, or otherwise be open to the public internet.
2. The server should be reachable from an unchanging alias, such as a domain, and if the address of the network that is used to reach the server is privy to changing, the network should have the ability to update it's domain dynamically
3. The server should be highly available and should not be on a computer that constantly shuts off at night or is privy to constant restarts.

With that in mind, judging by the nature of advanced users or platform owners only wanting to set up servers, the main server that provides all these features is fine to exist as an entity that requires special care.

In order to set up this server, three ports should be forwarded to the server, that being 8000 for HTTP, 8554 for RTSP and 1883 for MQTT. This provides all the facets of function for the platform at the moment and likely may not need to expand.
The server should be hosted on a standard server that stays up 24/7, ideally with power backup. The ideal operating system is a distribution of linux with a package manager that can install the required software and supports ssh to do so, at least. Networking requirements scale linearly off user interaction, and the basic assumption is that most users will generate spikes of network strain rather than constant strain. Currently the assumption is that for every 100 users, you may need 200MBit/s upload and download.

Installation Procedure

If a user wants to install the program on the server of their choice, certain dependencies must be met before all facets become operational:

- The packages for Mosquitto MQTT and Docker must be available and installed on the distribution of linux the server is running, along with OpenJDK version 16, or equivalent JDK release version.
- The server should be publicly routable.
- Ports 8000, 8554, and 1883 must be forwarded to the server if a NAT exists.

To launch the server, start by opening the SecurityServer.jar with the following command:

```
$ java -jar SecurityServer.jar
```

Leave this process running, either as a background TTY (Ctrl+Z) or via Screen, an optional package that allows for terminal multiplexing on linux.

Start both the docker instance of RTSP Simple server and the Mosquitto server with the commands below:

Mosquitto:

```
$ sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
$ sudo apt-get update
$ sudo apt install mosquitto
$ sudo systemctl start mosquitto
$ sudo systemctl enable mosquitto
```

Mosquitto should now be running as an independent service that starts on boot.

Docker and RTSP Simple Server:

```
$ sudo apt install docker
$ docker run --rm -it -e RTSP_PROTOCOLS=tcp -p 8554:8554 aler9/rtsp-simple-server
```

Docker should now be running in the foreground, where you can use CTRL+Z to send it to the background and the ‘fg’ command to return the process to the foreground. Alternatively run in a separate screen that runs alongside the securityServer application

You should now be able to connect to the server with the user client, as well as base stations.

Conclusion

Server Platform Conclusions

During operation, the server platform performs all needed functions to provide the base station with a place to store remote data, call out to clients, and publish data. The user is provided with a service that can provide authentication, data availability, and a way to interface with the base station and all of its modules. This works without need to worry about networking conditions, as the system is designed to be entirely request based, and each protocol that doesn’t explicitly use requests allows for users or devices to “subscribe” to data which works just as well. The overall implementation of these systems is a success on all fronts.

End User Application

Introduction

The end user application is a custom Java Swing application that is multi-platform and as such can be run on MAC, Windows, or Linux environments without the need for extra downloads or installations. The application communicates with the cloud service to receive information about the base stations associated with a given user and allow that user to connect and interact with any of the connected base stations.

Design

Preliminary Considerations

The objective of the End User Application was to create a graphical user interface(GUI) using Java Swing API to allow users to easily login, access, and manipulate the Ferris IOT Security Device's base station without having to be in physical proximity to the base station or physically connected to the base station. The End User Application was to provide three main areas of user interaction: a login service, a base station selection service, motion detection signaling capabilities, and finally a video and data streaming service from the base station to the End User Application.

Initial Designs and Changes

The initial design of the end user application was built upon the use of HTTPS to handle all communication from the end user application to the base station and cloud service, however one major flaw was found with this design that caused two redesigns.

1. When attempting to create a video stream between the base station and the end user application, a problem was encountered where https could not be used to maintain the video stream without adding unnecessary complexity and leaving the video stream completely exposed, adding a major security risk. The solution to this problem was to use an RTSP video stream having the cloud service host the stream from the base station and the application contacting the cloud service to display the RTSP stream. This solution caused a redesign of the video

streaming service of the end user application and that redesign caused a redesign of the video stream display method.

2. Originally the video stream was to be housed inside of a JPanel, however JPanels do not support the displaying of RTSP streams and so an new display method was required. The solution to the problem was to replace the JPanel used to display the video stream with a JInternalFrame that does support the displaying of RTSP streams, once that was done the JInternalFrame had to be modified to blend with the background of the window and make the JInternalFrame resize with the window if the user changed the window size.

Functionality

Login Page

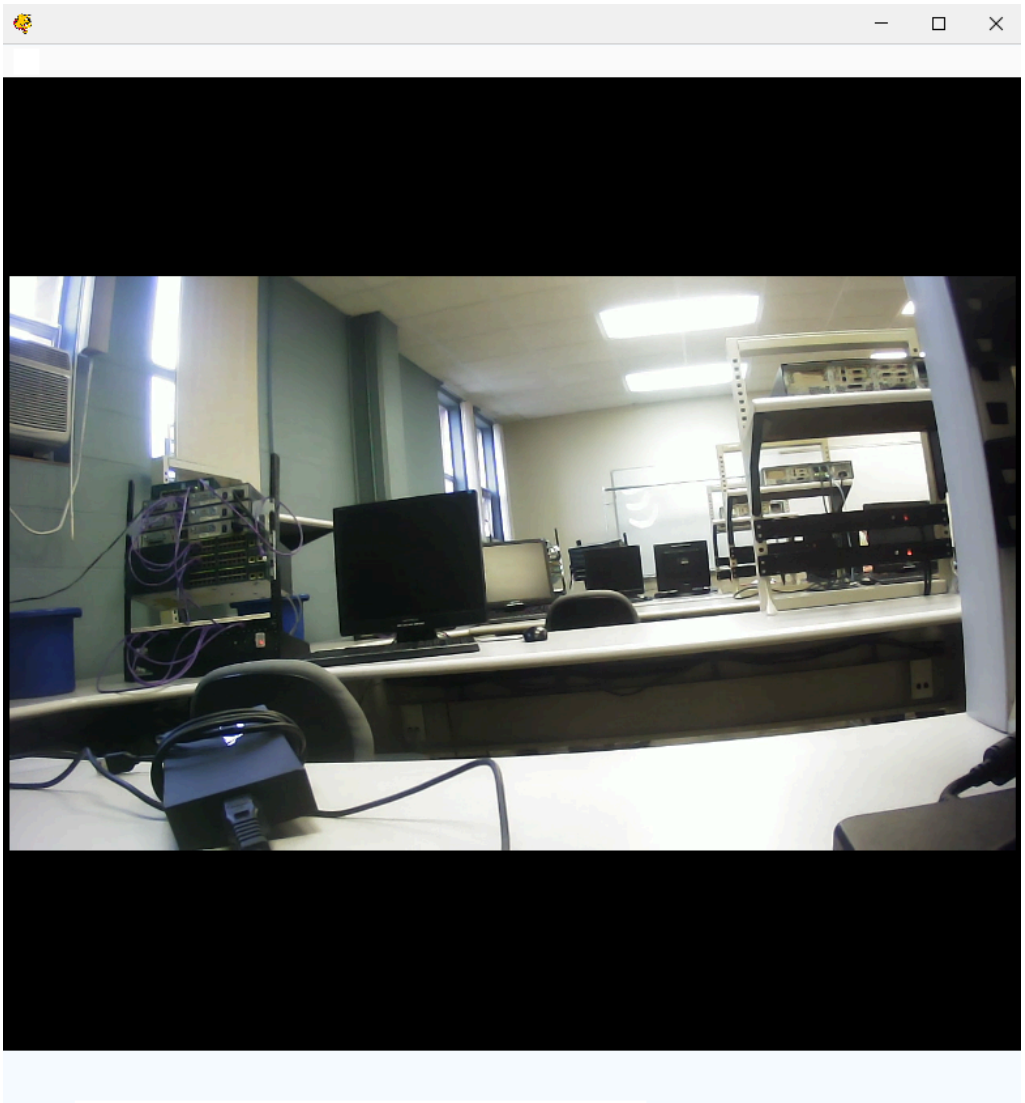
1. Construction: The Login Page consists of three JPanels that hold multiple other components inside them. The First JPanel, named Log, holds a JLabel that displays the company logo and name, the panel also holds a JButton called Enter, the other two JPanels are contained within the first panel as well. The Enter JButton has an action listener attached to it that when the JButton is pressed the application starts Https communications with the cloud service. The second JPanel, named UserSection, holds two components, a JLabel that contains the text ,“Username: ”, and a text field that the user enters their username into. The third Jpanel, named PasswordSection, contains a JLabel that holds the text, “Password: ”, and a JPasswordField that the user enters their password into.
2. Use: The Login Page is the first part component of the End User Application users interact with when the application starts. The page has two fields into which text can be entered, one for the username and one for the password. The page has a button on it that when pressed sends the text entered in the two fields to the cloud service to be checked against a known login credentials. If the login credentials are verified to be known the cloud service sends back a signal to the End User Application and the application generates the Select screen. If the login credentials cannot be verified a dialog box appears that notifies the user that the credentials that were entered are incorrect.
3. Visual:

Base Station Selection Grid

1. Construction: The Base Station Select Grid is built using two JPanels, the first one named choose, that panel holds the second JPanel, a JLabel, and a JButton. The JLabel holds the text, “Please Choose Which Device You Want To Access” and the JButton holds the text “Close” and has an action listener attached to it that closes the window. The second JPanel named Grid has the possibility to contain an infinite number of dynamically created JButtons depending on the number of base stations connected to the account the user logged into, each JButton created is labeled with the name of the base station and an action listener is attached to each button that when pressed a Https communication is sent to the cloud service.
2. Use: The Base Station Selection Grid is the second window the user interacts with, the user has two options when the window opens either to select one of the buttons within the Grid JPanel or press the close JButton that will close the window and send the user back to the login page. When one of the buttons within Grid is pressed an action listener attached to the button will Https communications are sent between the application and the cloud service to retrieve the status of the base station corresponding to the button that was pressed and then a Stream window is opened.
3. Visual:

Stream Window

1. Construction: The stream window is a JInternalFrame and not a JPanel, it is housed inside that frame that also holds the speaker play button, weather sensor info, and motion detection signal JPanels. The stream window holds the RTSP stream that is hosted by the cloud services and displays it for the user. To maintain the look of the frame of the window is customized to blend in with the white background of the JFrame that holds the JInternalFrame.
2. Use: The user can resize the window that holds the stream without disrupting the stream.
3. Visual:



Speaker Play Button

1. Construction: The speaker play button is created inside a JPanel held by the stream window. The button is labeled with the text “Play Sound” and has an action listener attached to it.
2. Use: When the button is pressed an action listener sends an HTTPS post to the cloud service to be relayed to the base station the application is connected to when the button is pressed.
3. Visual:

Weather Sensor Info

1. Construction: The weather sensor info JPanel is constructed dynamically when the end user application connects to the base station, the end user application receives a string of characters from the cloud service that the base station sends to the cloud service. The weather sensor info code parses the string, adds the measurement identifying characters, adds labels in front of the measurements, and then reassembled into another string which is then displayed in a JLabel inside the JPanel for the user to see.
2. Use: The user cannot interact directly with any part of the weather sensor info JPanel or JLabel.
3. Visual:

Motion Detection Signal

1. Construction: The motion detection signal JPanel is constructed as a static size and at a static location, with a JLabel being held in the center of the panel. The JLabel is initialized to hold a green warning triangle image. When the end user application receives a signal through MQTT the JPanel is updated to hold a red warning triangle instead of the normal green warning triangle.
2. Use: The user cannot directly interact with these components, but the user can see when the color changes from green to red meaning that motion was detected by the base station.
3. Visual:

No Motion Detected

Motion Detected

Structure, Governance, And Conclusions

This section will describe the history of our development, including outcomes and final thoughts. The structure of our software team will be discussed, as well as ideas of how we governed ourselves. Statements on all major completion criteria are also provided. A final conclusion on the entire project caps off this section.

Structure

Structure history

During the beginning of this project, the general idea of how to approach developing this system was unorganized. We had ideas, we had things we wanted to accomplish, but no standard structure. In order to organize ourselves, we managed to come up with our own way of directing what we should be working on, when and for how long. The initial idea was to decide who can do what, broadly. This was handled in one swift discussion, where we handled the scope, fundamental breakdown of this scope into focuses (electrical, hardware, software, etc).
It was decided that the software team would consist of Aron and Asher, and the hardware team would be Jacob and Andy. On the software team, roles were decided between backend and frontend, and in backend was designated to Aron, and frontend was designated to Asher. For the hardware team, Electrical and Interfacing was designated to Andy, and Structure hardware and design was designated to Jacob.

Roles

Lead Project Manager - Aron Mantyla

The Lead project manager is responsible for most management and all leadership. The project manager distributes tasks, hosts meetings, enforces policy and quality, and serves as a figurehead for the project.

Frontend Developer - Asher Carpenter

The role of the frontend developer was to handle any and all user interaction with the system, inside the application that would be launched by a normal user. This role was not constrained to creating just the UI, it was also responsible for handing the data that was passed to this application, making requests to the server and dealing with the data that was provided, handling events from the server, and displaying video from the server.

Backend Developer - Aron Mantyla

The role of backend developer was to create an ecosystem of functional integrations that the frontend developer can use. Another responsibility was to make these same systems, but for use with base stations, and also to create the software that would run on the base station. The backend developer was also responsible for the software that ran on the base station in every capacity, regarding how it would talk to the server, how it would start up, and how it would manage its modules.

Electrical and Interfacing - Andy Roggenbuck

The Electrical and Interfacing role was meant to create and maintain all modules that would be installed onto the base station. The base station was to have modules connected via USB to the PI, and In order to make it so everything ran off of USB, the Electrical role was tasked to integrate all of these modules to this protocol. Alongside USB integration, the Interfacing role was to allow for these custom USB devices to communicate with sensor counterparts, such as weather sensors or alarms.

Structural and Hardware - Jacob McClelland

The Structural hardware and design role was responsible for making adequate housing for all components. This role was to work closely with the electrical role, so all components could fit within each housing. This role is responsible for the design as well as the manufacture of these housings.

Task Distribution and Schedule

During normal operation, tasks were automatically distributed to the person working on them, by the person working on them. This autonomous structure works to a degree, but every so often, what is known as subscope confusion occurs, where workers find themselves devoid of work

because of a lack of direction. To avoid this anomaly when possible, weekly meetings would be conducted to casually assess the work needed and set goals.
Goals were determined by dependencies and priority. Many tasks were completable asynchronously from others, such as building the module electronics versus building module housing, however some have dependencies, such as “the module must be built before we can program for it and test it”. There were no weeks where time scheduled for work was ever taken off to relax, and instead all weeks were used in the capacity the schedule was designed for. Tuesdays and Thursdays were from 3PM to EOD, and Wednesday was from 4:30PM to EOD. It was encouraged that weekends be taken off, but if there was work to be done, to work on the weekends.

The tasks that were distributed were determined from a constructional standpoint. This way of working has a goal in mind, and an expert analysis of what can and cannot be worked on given knowledge of what is currently available. For example, you cannot build the decor on a building with only a frame, and as such, you cannot work on interfacing with a temperature sensor via I2C until you have a working USB Serial device. This approach was used for every problem.

Project Change Statement

During the early phases of the project where a proposal was drafted, some ideas within the scope were transmuted, such as including the PIR sensor as a module, allowing for unique audio playback via alarm, and usage of Dart as a programming language. These changes were communicated to each member of the project team and the changes for these decisions were made. These changes did not harm the overall scope of the project, and did not reduce the effective usability of the project. Instead of completely removing major functionality, these ideas were either moved to different areas of the project or replaced with something that worked better.

Criteria Statements

Scope

Regarding the scope and outcomes of the project, and comparing the actual results of the project to the project proposal results section, all desired results were met in full capacity, with extra functionality made available. No part of the scope had to be cut from the project.

Budget

The cost for the project was estimated to be $227, and the budget was designed to be $230. We gave an initial wiggle room of +-10%, and the actual cost was $247, therefore we were within our budget radius. For the end of the project, the final cost will be split between parties, where the person that will inherit the base station will pay 20% more than the rest of the group. The costs fronted by each party will be equalized by everyone, and a common function of input will be reached by each member to equalize the amount of money used across the board of members.

Time

All major development steps have been completed on time. All major functionality with each sector of development was consistently on schedule. The project was functionally finalized a week and a half before presentations.

Final Conclusions

The project was a complete success. We had minimal changes to workflow, no major changes to scope, and no failure to meet deadlines or scope objectives. All tasks were completed on time, and the product worked flawlessly during finalization, presentation, and demonstration and the final cost was within our expected budget range. The team structure of asynchronous work via role designation made an environment where people could focus on their own work and not feel bound to any particular person for work.