How to use the GSS

Version 1.0

# Overview

This readme file goes over how to setup the GSS in the incarnation that was used for the Expo presentation.

This involves the GSU and sensing vehicles, sending the status over LoRa to the GSG (Dragino) then sending over the internet to the MQTT server and then updating the simulation software with the status of the parking spot. The simulation can then be viewed on a webpage by going to the web server ip address.

Additionally, information about the other programs is also provided.

The readme assumes basic knowledge of using Arduino, Arduino IDE and networking.

Note: All of these sketches have serial print statements. If the Arduino is to be run “headless” without serial monitor output, the serial prints need to be commented out, otherwise the program may not get past the setup.

# Adafruit Feather M0

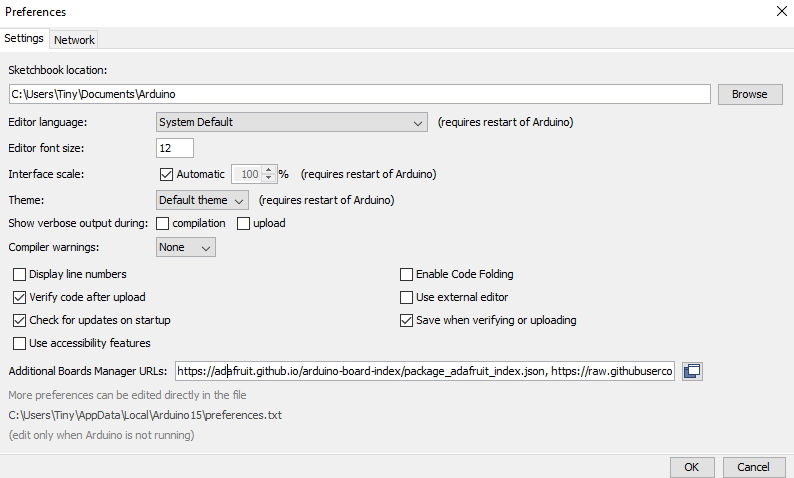
The Feather M0 is programmed using the Arduino IDE located at <https://www.arduino.cc/en/Main/Software>

The Feather board will need to be added via the board manager by doing the following:

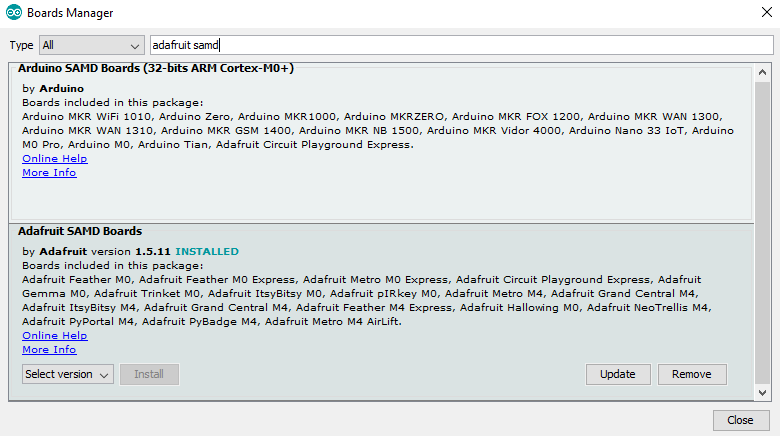
In the Arduino IDE, go to File -> Preferences

Add the following link to Additional Board Manager URLs:

<https://adafruit.github.io/arduino-board-index/package_adafruit_index.json>



Then, go to Tools -> Boards –> Board Manager, search for Adafruit SAMD and install the Adafruit SAMD Boards package.



Sketches will now be able to be compiled and uploaded to the Feather M0.

# The GSU

Zane created an enclosure which houses GSU ID 2. It contains the Adafruit Feather M0, DS3231 RTC, Antenna, LED, Ultrasonic sensor and PIR sensor. The Antenna, Ultrasonic, PIR sensors and LED can be seen on the outside.

The GSU is ceiling mounted 8-10 ft above ground with the sensors facing down. There is a micro USB port on the outside of the GSU which will need to be attached to mains power, powerbank, computer or other means of providing power.

The GSU is running the sketch called Park\_-\_it\_-\_Cda.ino

The GSU sensing can be tested mounted or unmounted. If there is no obstruction within 100 cm (3.28 ft), the LED will be green. Placing a hand or driving a vehicle under the sensors will cause the sensor readings to change. If the obstruction is within 100 cm, the GSS considers the parking spot occupied.

# Networking - GSU, GSG - Dragino

The GSG is a Dragino LG02. Both the GSG and GSU are configured to listen on the 900 MHz frequency using the default modem configuration settings.

Bandwidth = 125 kHz, Coding rate = 4/5, Spreading factor = 128chips/symbol

The Dragino connects to a WiFi network and has been successfully connected to Tyrel's, Nikolai's and Zane's home networks. To connect to a new network will require connecting a computer to the LAN port using a network cable.

Open a web browser and go to 10.130.1.1. The default login is root/dragino. Go to network -> wireless -> scan. Search for the network to join and enter its credentials. Click on save and apply and the Dragino will connect to the new network. The full user manual is located in the Manuals and Datasheets folder.

The Dragino can be checked to see if it is receiving messages by going to Service -> Logread -> RxTxJson

Messages received over LoRa appear here and the sequence being looked for begins with 3C313E which is <1> in ascii hexadecimal encoding. These are first 3 bytes of the message being sent from the GSU to the GSG and let it know to forward it onto the MQTT server.

# MQTT Server and Simulation

Ensure that you have already downloaded all of the libraries that are included in this code.

There will need to be an MQTT broker setup first and foremost. I used mosquitto on my raspberry pi and it worked perfectly. Put that username and password into the code where required – garageStatus.py, garage.py, webpage.ino. Ensure there is a mosquitto broker service setup in systemctl. Choose a topic name and put it in the code.

There will also need to be a mysql database setup with at minimum a ID [int] primary key auto\_increment, spotId [int], status [int], and tStamp [datetime] columns. Put that username and password into the code in garageStatus.py. Setup a systemctl service to start garageStatus.py when the system starts (I had this running on my PI). That is the code to record all updates on the topic chosen.

Once the code is all updated with usernames, passwords, wifi info, and topics then you should be able to upload the sketch to an esp32 and then access the page. Meanwhile you can run the garage.py program the pi or another computer that has access to the network the broker is on in order to simulate all of the spots. The code has default values that it uses, but it will also accept input from the command line for varying the number of cars and spots and the speed in which the simulation runs. You should be able to see the spots change color on any device that has access to the network with the esp32 on it.

NOTE: I used port forwarding on my router to grant access for everyone to see the website. You will need to configure this or get somewhere to host the site.

# Other Sketches

## Client and Server

These two programs are provided by Adafruit.

<https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/rfm9x-test>

There are multiple versions of these programs on the site for different modules and boards and they may work also.

The main thing to get correct is wiring pins and pinouts in the sketch.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | CS | RST | INT / G0 | MOSI | MISO | SCLK |
| Arduino Uno | 4 | 2 | 3 | 11 | 12 | 13 |
| Arduino Mega | 4 | 2 | 3 | 51 | 50 | 52 |
| Adafruit Feather M0 | 8 | 4 | 3 | N/A | N/A | N/A |

Unos tested were the official Arduino Uno and an Elegoo Uno. The Mega was an Elegoo Mega 2560. These pins may vary depending on the board used. MOSI, MISO and SCLK are I2C pins so look for those pins on the board being used.

Upload the client sketch to one Arduino, server to the other. Open two Arduino IDE instances and set the COM ports to different Arduinos for each one. Open the Serial Monitor on both and there will be messages being sent and received. The client will send a message periodically. The server will listen for messages and if it receives one, will send a reply back. The client will wait for a reply within 5 seconds and display on console output if one was received.

If the message “LoRa radio init failed” appears, the pins were wired or defined incorrectly.

## ObtainID

This program retrieves and prints out the Unique ID associated with the Arduino to the serial monitor. This ID is used to statically define simpler IDs for use in the GSS sketch and its derivatives.

## setTime

This program requires the DS3231 RTC module connected to the Arduino. This was tested with the Adafruit Feather M0. The pins wired up are VCC, GND, SCL and SDA. SQW is the square wave generator and is used for generating interrupts from sleep but this wasn’t successfully tested. The time is set in SS:mm:HH:dd:DD:MM:YY format in the setup.

SS = Seconds

mm = Minutes

HH = Hours

dd = Day of week

DD = Day of month

MM = Month

YY = Year

Upon startup, the time will be set. If the Arduino is powered off with this sketch still on it, it will be reset to the same time upon powering on again. Got around this by uploading a different sketch e.g. blink before powering it off. Look for a better way of doing this. Synchronizing multiple DS3231s requires uploading the same sketch with the same time set; set to ahead of time, then uploaded to each one, and reset at the same time.

## GSS

This was to be the main program that the GSS would run on and is incomplete. It uses the Feather M0 with the DS3231 attached. It does not have Zane’s sensor portion added to it.

The program uses the Unique ID of the Feather to generate a simpler ID from 1-7 based on the first byte since they are unique for the batch that we had. This was done statically. ID 1 was defined as the GSM role and doesn’t change. Every other ID uses the GSU role. This is done in loop by branching based on the ID. The built in RTC is used to set an alarm every 30 seconds for the GSM and every 10 for the GSU.

The GSM goes to sleep and wakes up every 30 seconds and listens for 10. If a message is received and addressed to it, it sends a reply. It also encapsulates the message with the GSG ID of 99 and sends that out as well.

The GSU will go to sleep and wake up every 10 seconds and sends a message. If a reply is not heard within 5 seconds, it will keep trying until one is received.

The addressing was only partially implemented here, with messages from the GSU being sent to the GSM. The replies had not been implemented yet, so if two GSUs send at about the same time and the GSM replies to both sequentially. The GSU that sent second may receive the first reply from the GSM and assume it got a reply. Another is that the first GSU may assume that the second GSU sending a message was the reply from the GSM.

## ZSS

This was the sketch used after Spring break and COVID-19 derailed the GSS development. It is a mix of sending and receiving branches based on the ID. This was due to driving around the neighborhood with a laptop and feather listening, mounting them on flagpoles/PVC piping, having a sender, passive listener and so on.

## TyrelGSS

This was a sketch created for Tyrel to test the GSS. It send a message already fully formatted for the Dragino to receive and process.

## Encrypt

This README file describes how to use and implement the encryption class that was created for data security across our mesh network. It provides data concealment of data and also data authorization checks for each data packet being recieved.

The class can be found under the following files:

- duhCrypto.h

- duhCrypto.cpp

To create an object type:

duhCrypto() <object name>

There are many function that can be used on this object including a crypIt() function that encrypts or decrypts a message depending on it's argument. It also either matches or adds the authorization phrase at the beginning of each data packet.

For more details on the functions and how to implement the class, read the comments in the files.

An example of the class being implemented on a "server" and a "client" Arduino board can be found under the following files:

- server.ino

- client.ino

Load the "serverCryp.ino" file on the server Arduino board and the "clientCryp.ino" on the client Arduino board. Both boards must have LoRa feather antennas attached. Run both boards and notice how the encryption class is implemented.