Olivia Liebler

8 April 2022

CSC 400 – SapSense

# Preface

## Background and Project Goals

The idea behind this project was to solve an agricultural application problem: How can we make it easier to monitor maple sap collection? Maple syrup is a thriving industry in the northeastern United States and Canada. According to [a 2019 report](https://agriculture.vermont.gov/sites/agriculture/files/documents/AgDevReports/Maple%20Syrup%20Market%20Research%20Report.pdf), over 9,000 separate maple syrup makers are in business in the United States alone. Many of these maple syrup makers are small businesses or hobbyists. During the 'sugaring season' in early spring, they collect maple sap using traditional taps and buckets, the way sap has been collected for hundreds of years. For big industrialized farms, expensive vacuum tube and remote pressure monitoring systems like the [TapTrack system developed in 2016](https://www.farmingmagazine.com/maple/sugarmaking-technology-past-present/) increase yield and monitor sap collection conditions 24/7. But for smaller farms that use buckets and not vacuum tubes, a different solution is needed. The goal of this project was to build a proof-of-concept that LoRa-connected wireless sensors could be used to monitor maple sap buckets. A system like this would be relatively inexpensive to set up and keep connected to the internet and reduce the amount of work and worry that goes into tracking maple sap collection during the season.

The number one functional requirement behind this system is to let the sap farmer know when a sap collection bucket is full and needs to be emptied. Additionally, the farmer should be able to login to the system to check on the status of all the buckets at the same time.

Diagram

Description automatically generated

Figure 1: Use Case diagram representing the required functions of the sensor system, created during project development.

Diagram, text

Description automatically generated

Figure 2: Sequence diagram representing the notification function, created during project development.

To accomplish this functional goal, the challenge was to link multiple layers of technology together. First, the sensor itself needed to be built and programmed to transmit data. Next, it needed to be connected to the internet. For this step, long range connectivity was needed because maple sap farms can often cover multiple acres. The sensor data needed to be decoded and repackaged on the server end to make it understandable for the end user. Finally, a way to send notifications to the user needed to be connected.

## Conclusions and Next Steps

The major challenge of this project overall was the breadth of different tools and skills required to accomplish the end goal. Building and programming the sensor, connecting it to the Helium network, parsing the collected data, and adding integrations to make that data useful all required different technical knowledge and skill sets. Put together, all these tools create a unique "stack". Considering this challenge, the product of this project is a 'user manual' which explains in detail how this “stack” works. Hopefully, other students working on similar projects will be able to use the manual as a guide and build on the ideas presented here.

One next step for the SapSensor project is modifying the power supply. The hardware design currently must run on the 5V input power provided by the USB connector (this works when connected to a computer or other USB port). However, if deployed in a real maple farm, this sensor would need to run off of rechargeable solar battery. This would require rewriting the on-board software to minimize power consumption and testing the sensor with the battery module.

Another next step that would improve the sensor is adding a thermometer component. The temperature data could maple syrup makers understand how changes in overnight temperature affect sap output. It would also let the user know when a sap bucket has gone below freezing, which can be a desirable time to empty it as the floating ice can be separated from the remaining sap.

One issue that would arise as this project is taken further is scalability. The more sensors required, the more work would need to be done manually to add each sensor to the network, hardcode each device's OTAA keys in the software, and add devices to the dashboard. If this project was expanded to hundreds of devices, the amount of manual work required for setup would grow to be difficult to manage and would need to be automated.

Over the course of this project, I learned to appreciate the complexity of trying to solve a real-world problem using digital technology. This sensor prototype shows how a liquid sensor, the Helium network, and TagoIO integration could be used together to do something valuable. The end product has some significant limitations which would be difficult to overcome. More than anything, I hope that the outcome of this project and this manual will help to inform further exploration into developing Internet of Things devices here at Southern Connecticut State University.

# Manual

The SapSensor is designed to detect whether a bucket of maple sap is full or empty. It runs on a RAKwireless WisBlock kit hardware and the Helium network for internet connectivity.

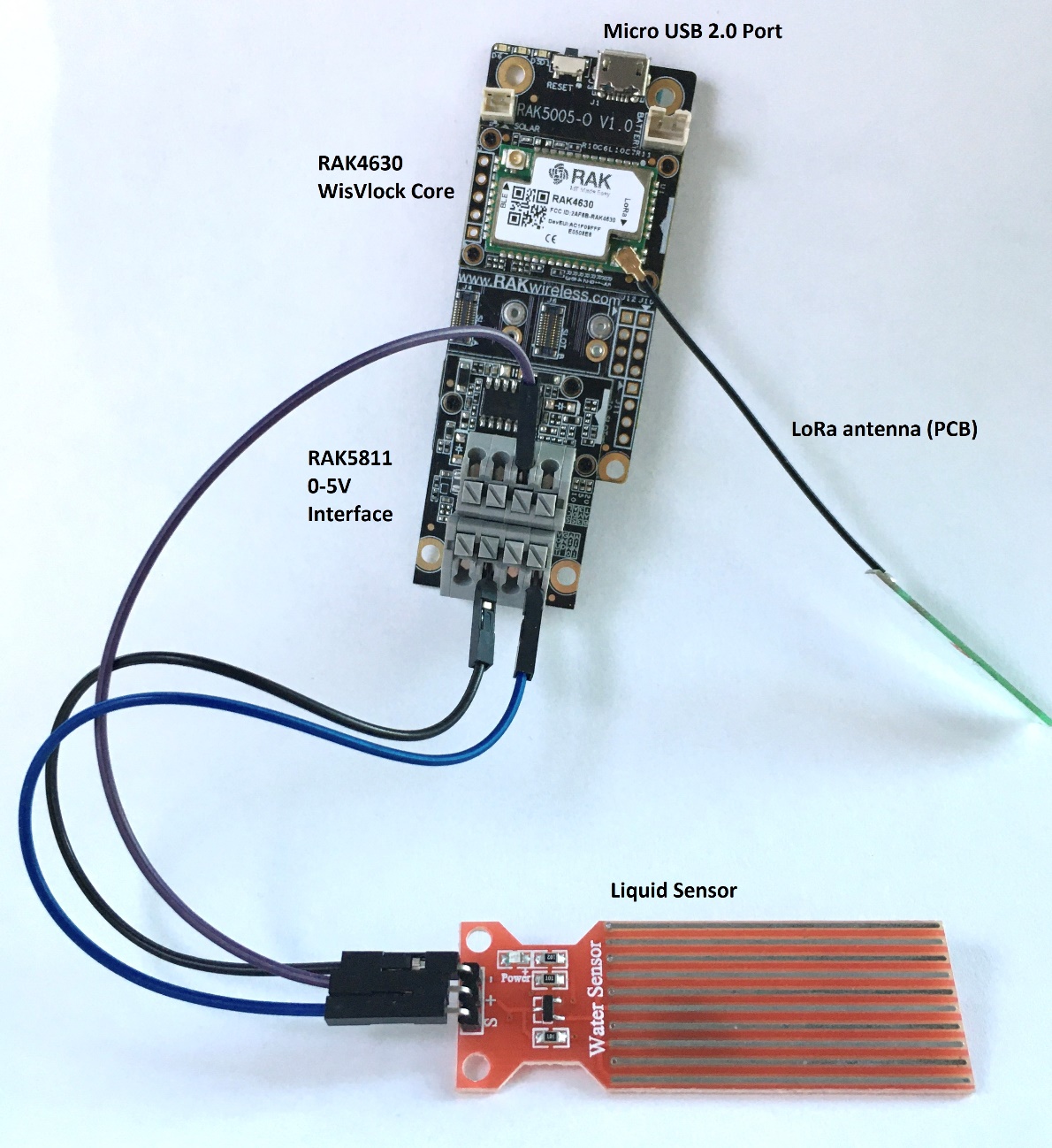


Figure 3: The SapSensor hardware, including liquid sensor, LoRa antenna, and WisBlock base.

The SapSensor uses a liquid sensor, RAKWireless WisBlock base and core modules, RAK5811 I/O interface, and a LoRa antenna.

## Background on Setting up the SapSensor

RAKWireless WisBlock kit devices are programmed similarly to an Arduino. The device has a USB 2.0 micro port on the board that can be connected to a USB standard port on a computer to write new programs to it (“flash” it) and to power it. The WisBlock can also be powered by a battery, but only if specially configured for lower power consumption. Currently, the SapSensor program is set up to run on computer power, not battery power.

To initially set up the device, I followed the RAKWireless [WisBlock Quick Start Guide](https://docs.rakwireless.com/Product-Categories/WisBlock/Quickstart/#key-features). Additionally, [the LoRaWAN communications example walkthrough](https://github.com/RAKWireless/WisBlock/tree/master/examples/RAK4630/communications/LoRa/LoRaWAN) was helpful in getting familiar with the device’s workflow. Much of the LoRaWAN enabling code in the SapSensor program is based on the introductory code in this module. Aside from building new code, this example is also a good test to make sure the device and hotspot are working before trying to add additional functions to the sensor.

A key step in setting up this device was editing the OTAA keys and setting the correct region. Without the correct region selected, the device will use the wrong frequency. US915 is the correct region to use in Connecticut. Example code for the WisBlock kit often comes with other regions selected, like EU868, so when working with example programs it is important to save a copy of the code and modify the region parameter before using it.

Over-The-Air-Activation (OTAA) Keys are used by IoT networks to securely activate LoRaWAN devices. The Device EUI, App EUI, and App Key written in the software on the device must match the device profile in Helium for the device to join the network. In the SapSensor implementation, these keys are generated by Helium when setting up the device in the console and then copied by hand into the device software. One important note is that keys can be expressed either most significant bit first (MSB) or least significant bit first (LSB). Within the SapSensor code, keys are MSB first, and if setting up a new device the keys would need to be retrieved from Helium as MSB. Reversing the keys would prevent the device from joining the network.

Graphical user interface, text, application

Description automatically generated

Figure 4: Device EUI, App EUI, and App Key generated in Helium console. The Device EUI is expanded using the diagonal double arrow button and shows the 'msb' label.

Text

Description automatically generated

Figure 5: Matching OTAA keys in the SapSensor program.

## Hardware Assembly

The SapSensor uses 5 main components: the RAK5005-O base, RAK4630 core, RAK5811 IO interface, the MH Water Level sensor, and a PCB LoRa antenna. The 5811 interface was selected because it digitally registers changes in an input voltage between 0 and 5 volts. The signal pin on the liquid level sensor outputs different voltage when the sensor is dry and when it is immersed in liquid. The sensor is connected to the 3.3 V power pin, the ground pin, and the input signal pin AIN1. The WisBlock can use multiple kinds of IO Interfaces like the RAK5801, which is different from the RAK5811 in that it changes a current signal into a voltage signal. For different sensor projects, the interface should be selected appropriately based on the kind of signal the sensor component generates.

The WisBlock kit also includes a larger LoRa antenna which can be used to increase range. For indoor/proof-of-concept use, the PCB (printed circuit board) antenna can usually reach hotspots without difficulty.

## Using the Arduino IDE

The SapSensor code was written and edited in the Arduino IDE, available for free [here](https://www.arduino.cc/en/software). For those unfamiliar with Arduino IDE, the [documentation](https://docs.arduino.cc/software/ide-v1/tutorials/Environment) is a good resource for getting used to the most important functions like *Verify, Upload*, and *Serial Monitor*. The *Verify* button is used to compile and check the code. The *Upload* button will compile and flash the code to the connected SapSensor device. The *Serial Monitor* displays text output from the device over the USB connection while the program is running like how the console displays output from a program written in Python or Java. In the SapSensor code, the Serial Monitor displays the voltage signal of the water sensor, as well as confirmation/error messages when packets are sent to the network.

A picture containing application

Description automatically generated

Figure 6: From left to right: Verify button, upload button, new sketch button, open, save.

There are some common issues when working with the WisBlock on the Arduino IDE. One is that it sometimes takes multiple tries to flash code to the device successfully. This might be because while a program is running on the device, there are limited windows to interrupt it and write a new program. The fix for this issue is to reattempt the flash, optionally disconnecting and reconnecting the device to the USB port between attempts.

An additional issue is that sometimes as the device is setting up, its Port (COM4, COM5, COM6) will change unexpectedly. This can cause problems with flashing, as well as using the Serial Monitor, since the Serial Monitor only monitors one port at a time. Sometimes while flashing, an error message will state that “The board at port COM4 is unavailable” or a similar message. This indicates that the port has switched. To fix problems with flashing or not seeing any output on the Serial Monitor, go to Tools > Port and then select the one with the device name next to it. This will also change the Serial Monitor to the active port.

Graphical user interface, application

Description automatically generated

Figure 7: Selecting the correct Serial Port using the Tools menu.

Graphical user interface, text, application

Description automatically generated

Figure : Another example of selecting the correct Serial Port using the Tools menu, on MacOS

## Installing the Board Support Package and Additional Libraries

Before using the RAKWireless WisBlock either for building a SapSensor or another project, it is necessary to first install the Board Support Package (BSP) that matches in the Arduino IDE. Complete instructions from RAKWireless on how to install the BSP can be found [in their documentation](https://docs.rakwireless.com/Knowledge-Hub/Learn/Installation-of-Board-Support-Package-in-Arduino-IDE/). As an overview: first, in File > Preferences, add the [URL](https://raw.githubusercontent.com/RAKwireless/RAKwireless-Arduino-BSP-Index/main/package_rakwireless_index.json) for the WisBlock (in this case the 4630) to the field for Additional Boards Manager URLs.

Graphical user interface, text, application, email

Description automatically generated

Figure : Example of adding the additional board manager URL in the Preferences panel

Then, restart Arduino IDE to implement those changes. After restarting, open the Boards Manager from the Tools menu.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure : Opening the Boards Manager

Within the Boards Manager, searching for “RAK” will bring up the different BSPs for different RAKWireless boards. For the SapSensor, the RAKWireless nRF Boards packet should be installed, which supports the 4630/4631 core.

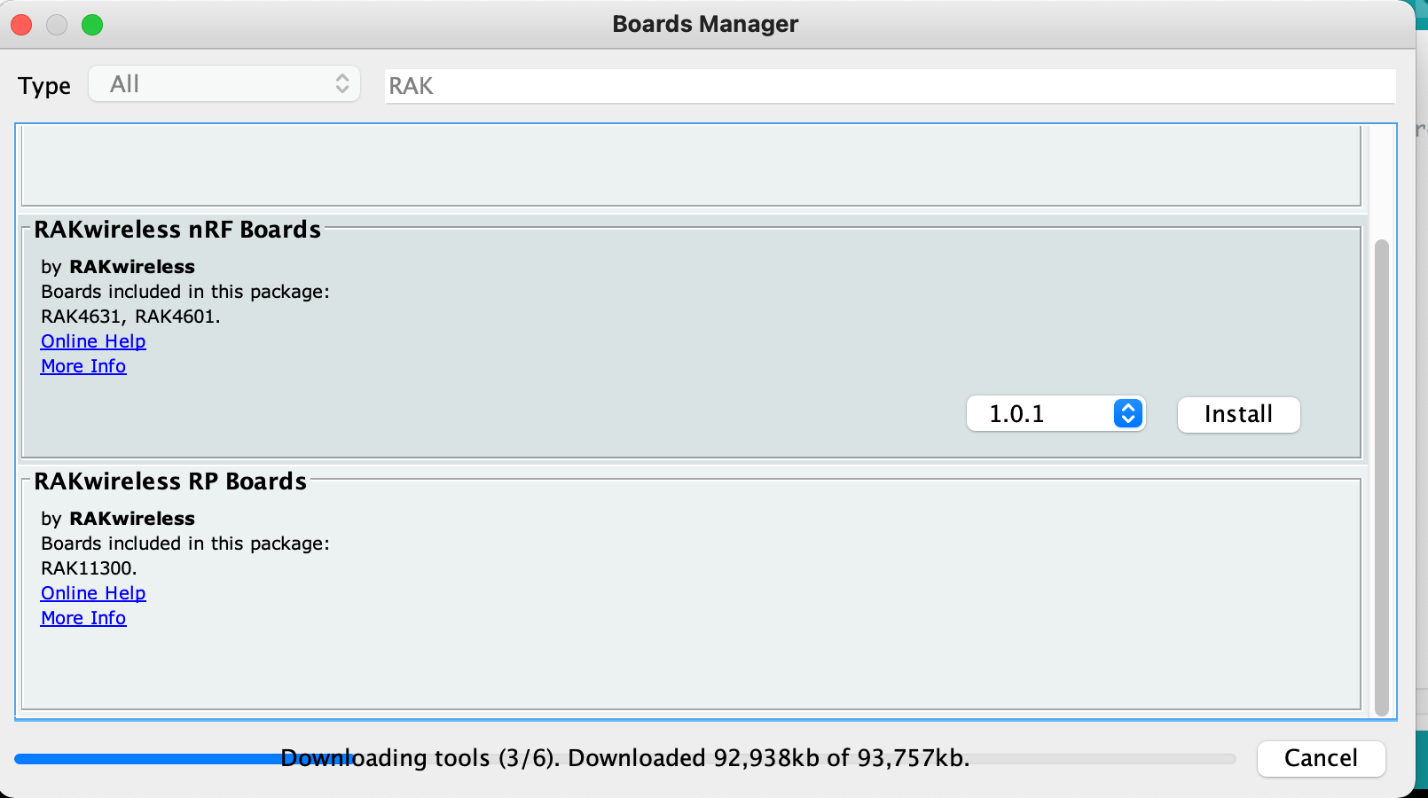


Figure : Inside the Boards Manager

After installing this BSP, in the Tools > Board menu, the WisBlock 4631 and 4601 should appear:

Graphical user interface, text, application

Description automatically generated

Figure : WisBlock RAK4631 option selected in Tools > Board menu

Selecting the board model that matches the one you are working with will update the system configuration to use the right compiler and settings for the module.

Code like the one that controls the SapSensor relies on additional libraries that must be installed on the computer before the code can be compiled. The SapSensor code uses the built-in Arduino.h library file, SPI.h library file, LoRaWan-RAK4630.h file for LoRa functions and Adafruit\_TinyUSB.h file for the liquid level sensor. Attempting to compile the SapSensor code before installing the external [SX126x-Arduino](https://github.com/beegee-tokyo/SX126x-Arduino) library causes a compilation error which looks like this:

Graphical user interface, application

Description automatically generated

Figure : Example of compilation error due to missing library files

To install additional library files, the easiest way is to open Arduino IDE and go to Tools > Manage Libraries.

Graphical user interface, text, application

Description automatically generated

Figure 14: Option to Manage Libraries in Tools menu

The LoRaWan-RAK4630.h library is included in the library [SX126x-Arduino](https://github.com/beegee-tokyo/SX126x-Arduino), which can be installed using the library manager.

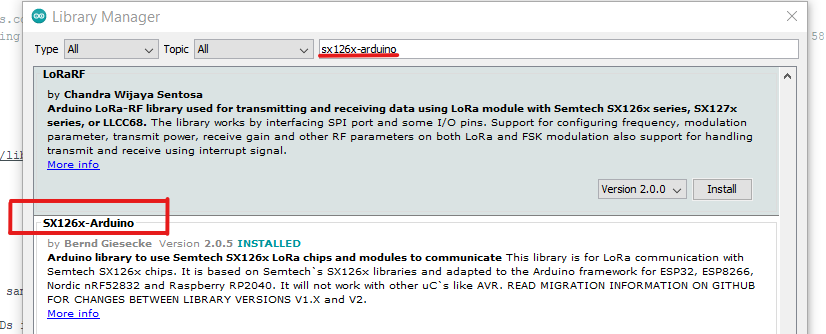


Figure 15: Installing the SX126x-Arduino library

The other library files should be included in the Built-In libraries and BSP, so no other libraries should need to be installed manually for the SapSensor. Each sensor module for the RAKWireless WisBlock requires different libraries, some of which are not built-in, so it is a good idea to check an example tutorial for each additional sensor module to find out which libraries must be installed.

After installing the library files, the code should be able to compile successfully and flash to the device as shown below.

Graphical user interface, text, application

Description automatically generated

Figure : Successfully flashed SapSensor device

## About the SapSensor Device Software

The code running on the SapSensor is based on code provided by RAKWireless for LoRa support, and example code written for the 0-5V sensor interface. It uses the Arduino.h library and LoRaWan-RAK4630.h library for LoRa functions, and the Adafruit\_TinyUSB.h library for the liquid sensor. The code follows the usual Arduino format: it consists of two main functions, setup() and loop(). The function setup() runs a single time when the device is turned on, and loop() runs repeatedly while the device is connected to power.

The program has two objectives. One is to monitor the liquid level sensor and store its value in a variable. The other is to connect to the Helium network and send frames of data containing that water level variable on a user-specified time interval. Both objectives have elements that run during setup() and loop().

In setup(), the LoRa chip is initialized, the Serial Monitor is initialized for debugging output, and then packet transmission is initialized by calling the function timers\_init() . After setting the OTAA keys and initializing LoRaWAN, the function lmh\_join() is called, which starts the network join procedure. The setup() function also initializes the sensor.

Text

Description automatically generated

Figure 17: Block of code where timers\_init() is called, resulting in a chain of calls that eventually sends the LoRa frame

Within loop(), the function analogRead() registers the reading from the liquid sensor which is a 10-bit value between 0 and 1023. To increase accuracy, this is repeated multiple times and the average value of the sensor input is stored as the result. This value is printed to the Serial monitor and stored in the global variable sensor\_reading. The loop() then delays for a preset amount of time before starting over.

For the purposes of testing, the loop() function currently delays for 20 seconds between reading. However, in a real implementation, this delay would be at least an hour, because the amount of liquid in a sap bucket changes more slowly. Additionally, this loop function reports the minimum and maximum value taken from the sensor on each loop to the serial monitor to make it easier to see whether the averaging is working properly.

Text, letter

Description automatically generated

Figure 18: Within the loop function, the function analogRead() registers input from the liquid sensor.

While loop() continuously updates the variable sensor\_value, the function send\_lora\_frame() packages that value into a packet and sends it to the network. The function send\_lora\_frame() is called by tx\_lora\_periodic\_handler(), which is called by timers\_init() in setup. This means that the timeline for sending packets is a separate process from the loop that reads the sensor.

Text

Description automatically generated

Figure 19: The body of the function send\_lora\_frame().

Within send\_lora\_frame() the payload of the packet is encoded into the array m\_lora\_app\_data.buffer. Each index of the buffer represents a separate byte. To store the variable sensor\_value in the buffer, it is broken apart into two bytes. The first byte, at index i = 0, is sensor\_value with a bit shift right by 8 bits. Shifting right by 8 bits removes the 8 least significant bits from the number. Those 8 bits will be stored in the next index, i = 1. For example, if sensor\_value was 637, or 0000 0010 0111 1101, the byte 0000 0010 would be stored in m\_lora\_app\_data.buffer[0], and the byte 0111 1101 would be stored in m\_lora\_app\_data.buffer[1]. For future development, it’s important to understand this memory storage scheme, because it must be reversed on the payload parser side to extract the value. Finally, after m\_lora\_app\_data.buffer is set, the function lmh\_send() sends this payload to the network within a LoRa frame.

If more functions or sensor modules are required for a future project, the best way to get started modifying this code or writing new code is to check the [RAKWireless GitHub for example code](https://github.com/RAKWireless/WisBlock/tree/master/examples/RAK4630/communications/LoRa/LoRaWAN) that goes with the desired sensor module. Since Arduino code shares a similar structure with the setup() and loop() functions, it is possible to carefully combine sample codes into new code, function by function. One important note is that some combinations of WisBlock modules can cause pin conflicts. RAKWireless provides [an online tool](https://docs.rakwireless.com/Knowledge-Hub/Pin-Mapper/) to check for pin conflicts before physically assembling modules.

## Troubleshooting the Network Connection

Over the course of the SapSensor project, one of the more common problems was the device not successfully joining the network. On the Serial Monitor, this displays as an “OTAA join fail.” There are several things that can keep the device from connecting to the network properly, and each has its own fix. In order from the device to the Helium server, the most common issues are:

1) The LoRa antenna on the device may be damaged or disconnected. This can be checked by unplugging and reattaching it or exchanging it for a different LoRa antenna.

A close-up of a circuit board

Description automatically generated with low confidence

Figure 20: The LoRa antenna snaps onto the RAK 4630 base. There is a similar Bluetooth connector on the other side.

2) There is an error in the program on the device. If running the SapSensor code, one possible problem is that OTAA Keys have not been configured correctly. Keys must be MSB, and the region must be US915. If there is a compilation error or a flashing error, the device will need to be re-flashed with working code and the Arduino IDE will throw an error, so this is not a likely cause of network joining issues.

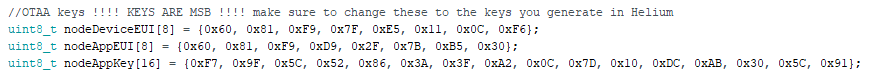


Figure 21: Setting OTAA keys in SapSensor code. The format is separate bytes in hexadecimal.

3) The device can’t transmit a strong signal to a gateway, either because it is too far away or blocked by physical barriers. Outdoor gateways connecting to indoor devices can be blocked by thick walls or by meta mesh like window screens. To eliminate this as a possible problem, the best option is to move closer to the gateway and avoid physical obstructions between the gateway and device.

4) The only hotspot(s) in range may be down. The Helium Explorer is a tool that can be used to check for available hotspots/gateways. It displays the three-word name of each hotspot as well as its status. Sometimes the status of the hotspot in Helium Explorer is not updated, so it is possible that even if a hotspot is not working it still shows as Synced in the Helium Explorer. For hotspots like the Bobcat miner, the most up-to-date status is found by connecting to the same LAN as the hotspot and typing its local IP address.

Graphical user interface, website

Description automatically generated

Figure 22: Displaying a hotspot status on explorer.helium.com. If the status is not Synced, the hotspot may not be working.

5) The Helium API is down. The Helium network relies on its API and Blockchain to run smoothly. At [status.helium.com](file:///C:\Users\Olivia\Documents\00_SapSensorManual\status.helium.com), it’s possible to see whether the API is down or Blockchain is running behind, both of which could cause issues with connecting.

## How Packets Move Through the Network

The SapSensor collects the reading from the liquid sensor and generates a data packet which is sent to the hotspot(s)/gateway(s) over a LoRa signal. The gateways forward the packet to Helium network’s server and appear in the Console. From there, the contents of the packet and its metadata can be repackaged by Helium and sent forward by HTTP integrations. For a Google Sheets integration, the decoder first must parse the payload before it can be sent through the HTTP post to Google Forms. The TagoIO integration can also optionally use a decoder before it is sent. In the SapSensor implementation, a payload decoder is used to parse the data and repackage it in JSON format before forwarding it to TagoIO.

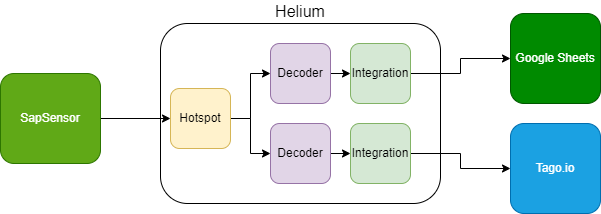


Figure 23: The flow of information from the SapSensor to the Google Sheets or TagoIO endpoint.

Diagram

Description automatically generated with medium confidence

Figure 24: Helium Console 'Flows' Interface showing devices with the tag "sapsense" connected to a TagoIO integration above and a Google Sheets integration below.

In Helium Console, this relationship between Devices and Integrations is represented and managed by the Flows interface. Above, two integrations are used: one TagoIO, and one Google Sheets. The blue label “sapsense” represents all devices that are connected to these integrations. The set of devices that share a label can be connected to multiple integrations simultaneously.

## Decoding a Payload

Once a SapSensor device has successfully connected, and its code is configured to transmit data packets, you will be able to see payloads come in through the Helium console. Records of individual packets are displayed in the Event Log. Graphical user interface, text, application, chat or text message

Description automatically generatedClicking the bug shaped icon on the right-hand side opens the Debugger view, which displays the contents of the packet, including the contents of the payload.

Figure 25: Black Debugger button, above blue manual downlink button.

One important note is that packets in the debugger only show up while the device is actively connected. They disappear when the device is disconnected from the network.

While in the debugger, the entire packet JSON information can be copied to the clipboard. Within this information is a field marked “payload”. This field holds the series of bits specified in the data buffer in the sensor software. In the debugger this series of bits is displayed in Base64 encoding. Plugging the payload into a Base64 decoder will reveal the binary, which can be useful to confirm that the device is sending the data packets you expect without decoding them. The rest of the packet is information needed for transmission to the network.

Text

Description automatically generated

Figure 26: Contents of a packet shown in Debugger Mode in Helium Console.

To turn the payload back into useful data, Helium uses a payload decoder. Most custom decoder functions in the Helium console share a common syntax. The function is called Decoder, and it takes two arguments: bytes, and port. (A third variable, uplink\_info, is available for optional use. ) The function returns a single value, “decoded.” “decoded” can be an Object which has one or many pairs of keys and values. Decoder functions are written in JavaScript syntax. In the example below, the function Decoder reads bytes 0 and 1 of the payload into a new variable called liquid level. It returns the Object decoded which has key “liquid\_level” and an integer value referred to by the variable name liquid\_level.

Graphical user interface, text, application, email

Description automatically generated

Figure 27: A basic decoder function that reads two bytes of payload input and saves the result as a single variable, liquid\_level

The bytes array which contains the payload data is indexed from 0. Within the bytes array, the payload is encoded in binary. Each byte of the payload holds the sensor data in order of its insertion into the buffer in the sensor program. The purpose of the decoder is to read the data byte by byte and reassemble it into a useful format. Many data items take multiple bytes and are reassembled on the receiving side by bitwise operations.

In the example above, the liquid sensor value is stored in bytes 0 and 1, with more significant bits in byte 0. To reassemble the value, first byte 0 is shifted left by 8 bits.

bytes[0] << 8

Then byte 0 is OR-ed with byte 1 to create a 16-bit integer, and the value is stored in liquid\_level.

var liquid\_level = bytes[0] << 8 | bytes[1]

Internally, if byte 0 is 0000 0010 and byte 1 is 0111 1101, your result in “temp” is 0000 0010 0111 1101, or 637.

## Data Port

The data port is set in the code. On an uplink in the debugger, you can see the port value:

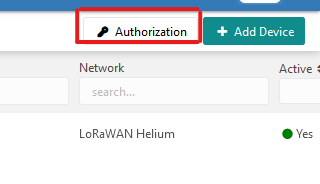
A screenshot of a computer

Description automatically generated with medium confidence

Figure 28: Packet contents showing key "port" and value 2

## Using TagoIO

TagoIO is a powerful tool for visualizing the data sent by IoT devices and adding automatic functionality like email and SMS notifications. Helium has an Integration designed specifically to connect devices to TagoIO. To connect your Helium device to TagoIO:

2) [Add the device to TagoIO .](https://help.tago.io/portal/en/kb/articles/1-getting-started#Mac_or_Linux_Command_Line) After adding the device, Tago prompts the user to generate a Service Authorization token. This token will be needed for the Integration later. Note: Within the General Information tab about the device, there is an option to generate a [device-token](https://help.tago.io/portal/en/kb/articles/4-device-token). This isn’t the same as a Service Authorization token and it won’t work when setting up the integration with Helium.

According to [TagoIO documentation](https://help.tago.io/portal/en/kb/articles/218-authorization), Authorization is for use with external services such as LoRaWAN or Sigfox. A device-token is used for connecting directly with HTTPS or MQTT. So, a device-token might be used for creating a custom HTTP/MQTT integration in Helium, but not for the prebuilt TagoIO integration.

3) [Set up the TagoIO Integration](https://docs.helium.com/use-the-network/console/integrations/tago/). The only thing that must be added is the Service Authorization that was just created. Also, even though TagoIO supports a payload parser, adding a Helium decoder function upstream of the TagoIO integration in the Flows menu can make it easier to add new data derived from other raw data. For example, a decoder can be used to easily take the value of something like liquid\_level and create a new variable/value pair that is a “normalized” value in a certain range.

Graphical user interface, text, application, email

Description automatically generated

Figure 29: TagoIO Integration setup

Text

Description automatically generated

Figure : A decoder specifically for TagoIO integration that creates a new variable, "normalized", based on the raw "liquid\_level"

## TagoIO Device Emulator

TagoIO devices represent individual IoT devices like each SapSensor. Each device’s data is stored in a Bucket. The Bucket tool can show what variable names have been stored in the bucket, but values are not visible. Also, in the free version, data is only retained in the bucket for 30 days. One very useful tool for TagoIO Devices is the [Emulator](https://help.tago.io/portal/en/kb/articles/95-device-emulator). Within the Emulator tab of the Device, you can write a simulated data packet with pairs of variable names and values, as well as unit or location attributes. When you send data within the Emulator, it will appear in the bucket just as if it was received through the TagoIO integration. With the emulator, it’s possible to add new variables to work with without first changing the payload parser upstream or even building a physical device at all.

Graphical user interface, text, application

Description automatically generated

Figure : Device Emulator showing a simulated JSON payload

This utility can be used to simulate payload data from multiple devices, even if no device prototypes have been built yet. Of course, TagoIO needs a separate Device EUI for each simulated device. You can use Helium to virtually register a device with a Device EUI and set up a Helium integration to connect these virtual devices with TagoIO. Then you can send Emulator packets from these devices, even if they have never sent a real packet before. For someone developing a new system, building out virtual devices and Emulated packets in TagoIO might be a good way to develop a dashboard and device concept before building a physical device.

## Tago Dashboards

Tago Dashboards display information sent from devices in a user-friendly format. They are made up of individual widgets, each of which can be configured separately to display data from one or more devices.

Graphical user interface, application

Description automatically generated

Figure 32: A TagoIO dashboard for SapSensors, showing a tile widget, table widget, solid dial widgets for water level, and a map widget

While dashboards are very flexible, they do have some limitations. Some widgets only work for showing data from a single device. Other widgets like the Static Table widget must be assigned device/variable pairs cell by cell, which means that for applications with hundreds of devices a Static Table might not be a practical structure to build. For projects designed to view hundreds of devices at a time, [a Blueprint Dashboard](https://help.tago.io/portal/en/kb/articles/454-blueprint-dashboard#Blueprint_Devices) might be a better choice, because dashboards can be generated dynamically for each device that shares the same blueprint.

## Tago Actions

Tago Actions are a set of utilities within the TagoIO platform that perform external functions in response to a Trigger. Common Actions are sending email/SMS/Push notifications, triggering a Tago Analysis, or posting data to another endpoint or device. Actions can happen in response to data in device variables, changes in Tago resources, on a time interval, or based on MQTT topics.

Graphical user interface, application

Description automatically generated

Figure 33: An example Tago Action for the SapSensor which sends an email to affected users when the liquid\_level variable passes the "full" threshold at 600.

## Tago Analysis

Tago Analysis is a scripting tool that can be used to process sensor data further. To run Analysis scripts on Tago’s servers, they must be Node.js (although it looks like Python scripts are in beta). Tago has some prewritten classes through @tago-io/sdk, which includes prebuilt definitions for classes Device, Util, and Analysis. Information on this code is available at <https://js.sdk.tago.io/> .

Analysis can be triggered by device events or time intervals. These triggers are defined by a corresponding Action.