Report: Reading and Storing Data from a Weighing Machine via USB to Serial Interface

1. Introduction

This report details the steps and code required to read data from a weighing machine via a USB to serial interface and store the received readings. The implementation uses Python with the pyserial library and JavaScript for an alternative approach.

2. Requirements

- **Weighing Machine**: Equipped with an RS-232 interface.
- **USB to RS-232 Adapter**: Converts RS-232 signals to USB.
- **Computer**: With appropriate software installed.
- **Programming Language**: Python or JavaScript with relevant libraries.

3. Setup

3.1 Physical Connection

1. **Connect the Weighing Machine**: Connect the RS-232 port of the weighing machine to the USB to RS-232 adapter.
2. **Connect to Computer**: Plug the USB end of the adapter into the computer's USB port.

3.2 Software Setup

1. **Install Drivers**: If using a USB to RS-232 adapter, install necessary drivers for the adapter.
2. **Identify COM Port**: Check the Device Manager (Windows) or /dev/ directory (Unix-based systems) to find the assigned COM port.

4. Serial Communication Settings

- **Baud Rate**: 9600 (example, check the weighing machine manual)
- **Data Bits**: 8
- **Parity**: None
- **Stop Bits**: 1
- **Flow Control**: None

5. Python Implementation

The following Python code demonstrates how to read data from the weighing machine:

python
Copy code

```
import serial
import re

# Open the serial port
ser = serial.Serial('COM4', 9600, timeout=1)

# Send the command to initiate the weight measurement
ser.write(b'P\n')

# Read the response from the weighing machine
response = ser.readline().decode('utf-8')

# Parse the response to extract the weight measurement
weight = re.search(r'(\d+\.\d+)', response).group(0)
print(f'Weight: {weight} kg')

# Close the serial port
ser.close()
```

Explanation

1. **Open Serial Port**: The serial.Serial method opens the specified COM port with the required settings.
2. **Send Command**: The ser.write method sends a command to initiate the measurement.
3. **Read Response**: The ser.readline method reads the response from the weighing machine.
4. **Parse Data**: The re.search method extracts the weight value from the response string.
5. **Close Port**: The ser.close method closes the serial port.

6. JavaScript Implementation

The following JavaScript code demonstrates the same process using Node.js and the serialport library:

javascript
Copy code

```
const SerialPort = require('serialport');
const Readline = require('@serialport/parser-readline');
```

```
const port = new SerialPort('/dev/ttyUSB0', {
 baudRate: 9600,
 parity: 'none',
 dataBits: 8,
 stopBits: 1,
 flowControl: false
});

const parser = port.pipe(new Readline({ delimiter: '\n' }));

port.on('open', () => {
 console.log('Serial port opened');
 port.write('P\n', (err) => {
   if (err) {
     return console.log('Error on write: ', err.message);
   }
   console.log('Command sent');
 });
});

parser.on('data', (data) => {
 console.log('Received data: ', data);
 const weight = data.toString().split(':')[1].trim();
 console.log('Weight: ' + weight + ' kg');
});

port.on('error', (err) => {
 console.log('Error: ', err.message);
});
```

Explanation

1. **Open Serial Port**: The SerialPort method opens the specified port with the required settings.
2. **Pipe Data**: The pipe method connects the port to a parser that reads lines of data.
3. **Send Command**: The port.write method sends the command to initiate the measurement.
4. **Read Response**: The parser.on method handles incoming data and extracts the weight value.
5. **Error Handling**: The port.on('error') method captures any errors.

Data Collection from IoT Sensors

How we implemented a weather station with KNIME - Part 1

The manufacturing industry suffered some severe blows in 2020 with shutdowns impacting supply chains and production levels. Manufacturers are looking to future proof their businesses and agility could be key. Those companies who are adopting digital technologies are better able to react swiftly to changes in the market. IoT initiatives for example, can help manufacturers improve their supply chain management or enable predictive part maintenance.

IoT applications are, however, complex, normally involving multiple tools and technologies to connect to sensors, set up a data storage and collect the data, and optionally integrate some form of analysis - time series analysis - to predict the future. An open software platform that is able to integrate with different tools and blend different types of data makes this easier. We therefore challenged ourselves to build a solution that would forecast temperature, using the open KNIME software.

The data generated by an IoT device needs to be collected and managed before it can be used. This article describes how we used a web service to collect and transfer the data from a sensor board to a data storage.

A second article will demonstrate how we tackled data forecasting based on a sARIMA model.

You can find all applications used to build this weather station on the KNIME Hub in KNIME Weather Station.

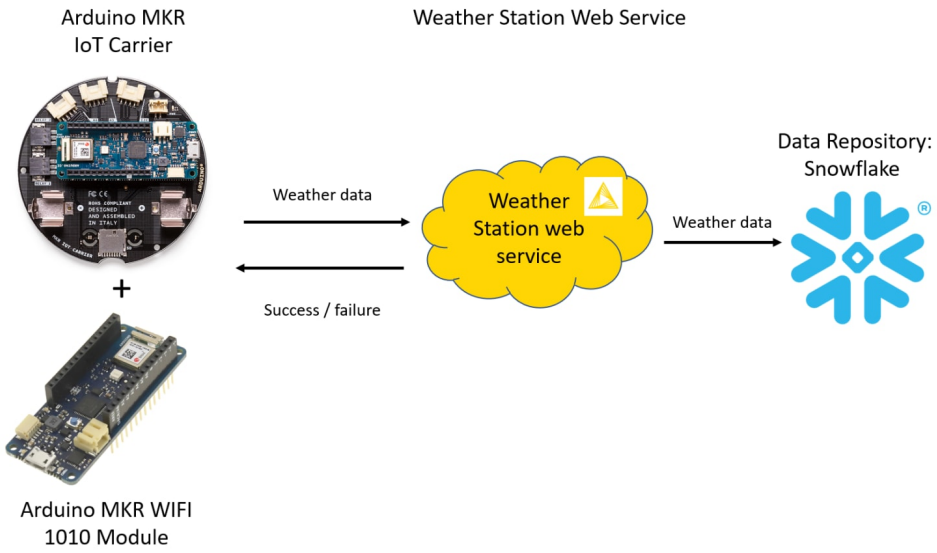


Fig. 1: Data collection architecture of the KNIME Weather Station: sensor board, web service, and data repository.

Data Collection from an Arduino Sensor Board

Around mid of May 2021, we installed the Arduino MKR IoT Carrier sensor board out of our office window.

This sensor board is embedded with different sensors, relays, LEDs, sound Buzzers, etc. For this project, we focused only on the sensors for temperature, humidity, and pressure. The sensor board has been extended by mounting a Wi-Fi module and powered using a micro USB 2.0 cable. This Wi-Fi extension was needed to connect to the REST service that collects the data.

For instructions on how to set the Arduino sensor board as a weather forecast station we got inspiration from this post Making an Arduino Oplà IoT Weather Station with Cloud Dashboard in the DIY Life blog. This post also teaches how to use the Arduino IoT Cloud to configure the sensors' variables and the Arduino's web IDE to write sketch code. A sketch is essentially a piece of code where you tell the Arduino board what to do, upon start-up and during execution, like for example attempting to connect to the internet upon startup, configuring display screen on board, etc. The sketch code, in this case resides on the Arduino IoT Cloud and it is written in a variation of C++.

Below, in figure 2, you can see the piece of the sketch code used to get the board started. It calls two functions -

and

.

is called when the board starts. It will only run once after power is switched on or a reset event occurs.

, on the other hand, does what the name suggests, i.e. loops continuously to control the Arduino board.

This Arduino MKR board is capable of recording values every 0.1 second, but a sample value every minute was judged sufficient and sampling frequency was set to one sample every minute through the

delay() function (highlighted in Fig. 2) in the C++ sketch.

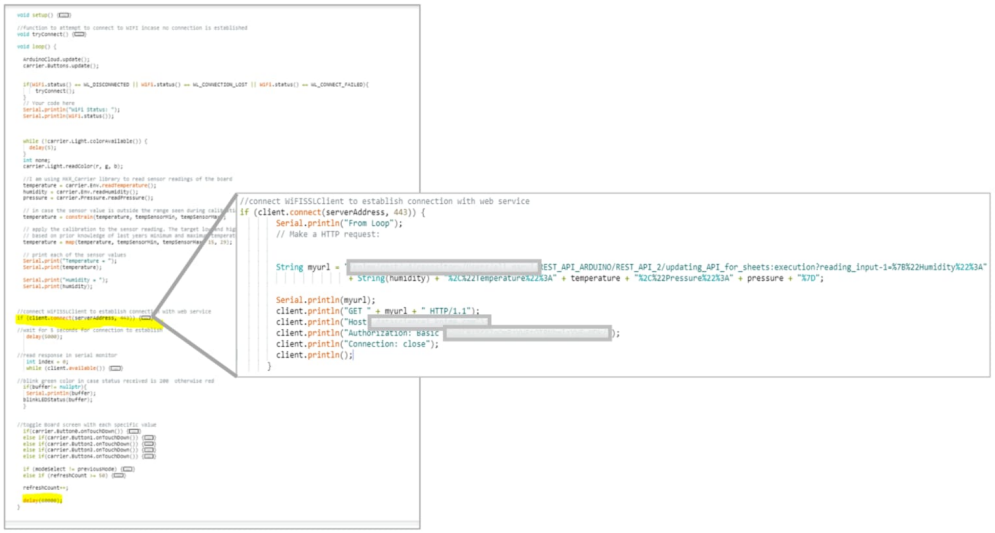


Fig. 2: Arduino Sketch Layout

Data Storage

Now what good is it to sample a signal, if the readings are not stored anywhere? As a data repository, we chose a Snowflake database.

Notice that KNIME Analytics Platform has connectors to many data repositories. You can see a subset of those in the last published cheat sheet on KNIME Connectors. We could have used a big data platform or a local installation of a classic SQL database. We chose to use a Snowflake database as a remote repository solution. We also used a Google Sheet as a backup storage.

In order to get the data from the sensor board into the data repository, the board script calls a web service, named Weather Station and residing on the KNIME server, every time that new data is available.

Data Transfer from Sensors via Web Service to Snowflake

The Wi-Fi module that we installed together with the sensor board can connect to an external Wi-Fi and therefore can consume any web service. The data is transferred first from the sensors in the Arduino board onto the Wi-Fi module via an https connection; then from the Wi-Fi module to the Snowflake database via the Weather Station web service.

A call to the Weather Station web service on the KNIME Server has been embedded in the sketch code (Fig. 2). In the image (you might need to zoom in), there is an IF block that incorporates a call to the data collection web service. The call to the web service is a GET request, requiring a path to the web service and a header with host name and basic authorization token.

**Just a tip, add the hostname of your server when updating the SSL certificates in the Wi-Fi module (Fig. 3). This task requires a Desktop version of Arduino IDE which can be downloaded from Arduino's website. One advantage of using Arduino Desktop is that it comes

loaded with examples and library manager along with other features like managing firmware in the Wi-Fi module. Also please remember to update Wi-Fi firmware before updating the sketch script.

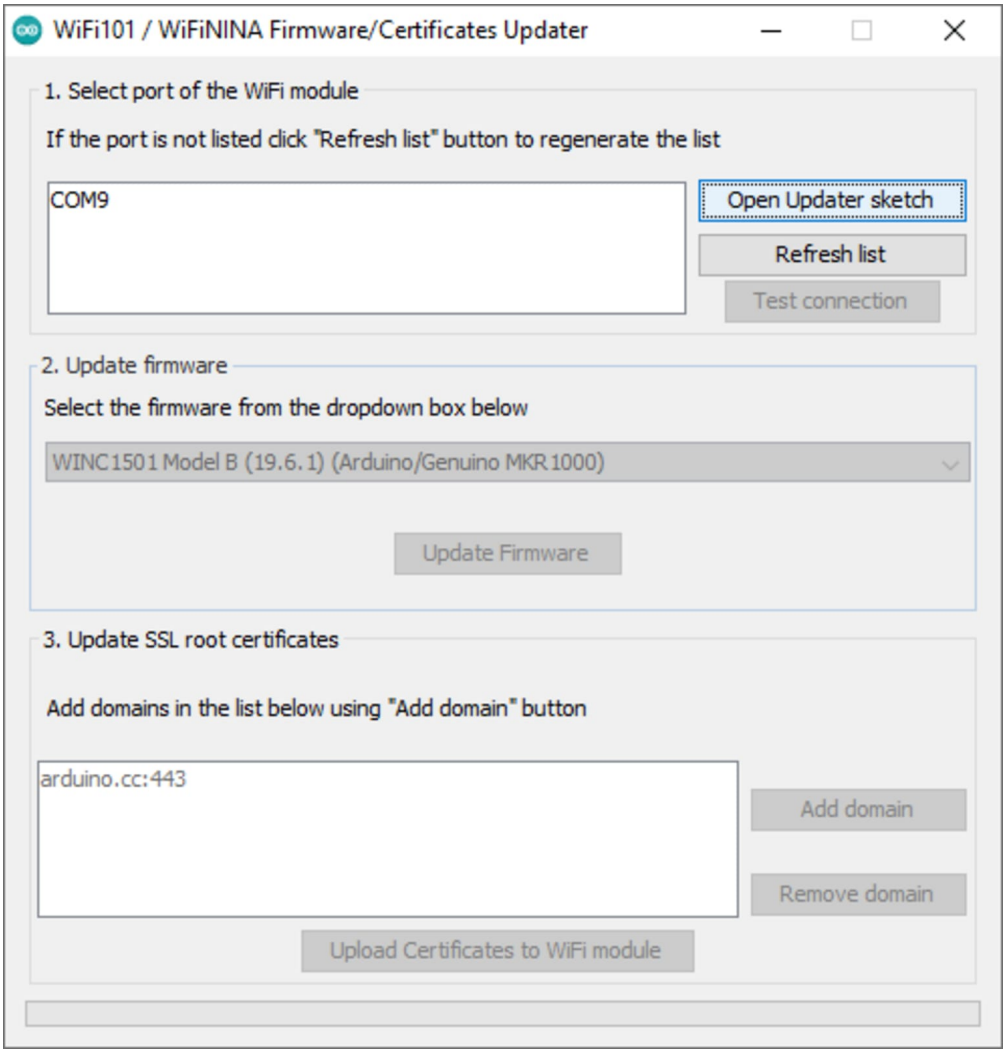


Fig. 3: Updating Arduino device firmware. On Arduino Desktop IDE, from menu bar, select Tools → WiFi101/WiFiNINA Firmware Updater

The Weather Station Web Service - REST API

We have reached the heart of the data collection part: the web service that transfers the data from the sensor board to the data repository. For that, we built our own Weather Station web service, a REST API, without programming but using KNIME software instead. No hassle of CROS, tokens or anything else; just basic oAuth authorization, JSON data structure, development on KNIME Analytics Platform, and deployment on a KNIME Server.

KNIME Server lets you easily deploy applications as web services, reach workflow REST endpoints, schedule executions, create web applications, and collaborate with other users. Among the many things the KNIME Server can do, let's focus here on deploying applications as REST API services.

Let's move step by step in the creation of this web service.

1. Transform a KNIME workflow into a web service on the KNIME server

That is easy! Any KNIME workflow on a KNIME Server can be called as

a REST service. More details on that, can be found in this blog article, The KNIME Server REST API.

2. Pass the sensor data to the REST service
To complete our Weather Station web service though, we need it to accept Requests and produce Responses, for example with JSON formatted data. In the workflow on the KNIME server we added a Container Input (JSON) node to define the JSON structure of the Request and a Container Output (JSON) node to shape the structure of the data in the Response (Fig. 5). The workflow implementing the REST web service then accepts Request data through the Container Input (JSON) node.

3. Add a timestamp
Every data operation must be documented at the very least with a timestamp documenting the time of execution. We generate the timestamp via a Create Date&Time Range node. The timestamp is then joined with the incoming data.

4. Write to the Snowflake database
To write in the Snowflake database we just need two nodes: The Snowflake Connector and the DB Insert node. The Snowflake Connector node connects to the Snowflake database, throughout credentials if required. The DB Insert node exploits the connection to the database at its input port and implements an INSERT SQL instruction on the database with the data at its other input port. In order to access the database, credentials are required and provided by the Credentials Configuration node. This node accepts credentials, encrypts them, passes them on in the form of a flow variable, and offers some options to store them.

5. Provide a graceful exit in case of failure
Databases are databases and sometimes they might not be available. They are down, credentials have changed, maybe unreachable, and so on. You know it. We need some kind of construct for a graceful exit in case the database is not available. We encased the database operation within a Try/Catch block (Fig. 4). The upper part of the Try/Catch block, i.e. the Try part, covers the database operation and adds an execution code 200 for success. The lower part, i.e. the Catch part, provides the graceful exit. In case the database operation did not succeed, a status code 312 for failure is provided. The component execution thus is always successful, only that in one case produces a success code (200) and in the other case a failure code (312).

6. Log failure or success of web service
The result of this operation is also reported in a log file updated each time by the CSV Writer node.

7. Write to the Google Sheet
A similar construct is used to write the input data on the backup Google Sheet. Here a Google Authentication node, a Google Sheet Connector node, and a Google Sheet Updater node are encased within a Try/Catch block outputting a 200 (success) or a 315 (failure) code. Notice that the Google Authentication node provides the Google authentication service on the Google side and not locally in the workflow. The final Response, including incoming data and Status Code, is sent out at client end and can be viewed back on the Arduino Serial Monitor.

8. Call to the REST API
The final workflow implementing the Weather Station web service is shown in Fig. 5 and is available on the KNIME Hub (KNIME Weather Station). Let's see how we can call it from the Arduino board.

Creating a JSON body for REST calls is a lot of work, if done on the Arduino device. To avoid this extra work, we include the data into the HTTP GET request. The REST Endpoint - in a percent-encoded format - is reported below and was obtained via the Swagger interface available on the KNIME Server.

All we have to do now is to incorporate it in the Arduino sketch script (Fig. 2).

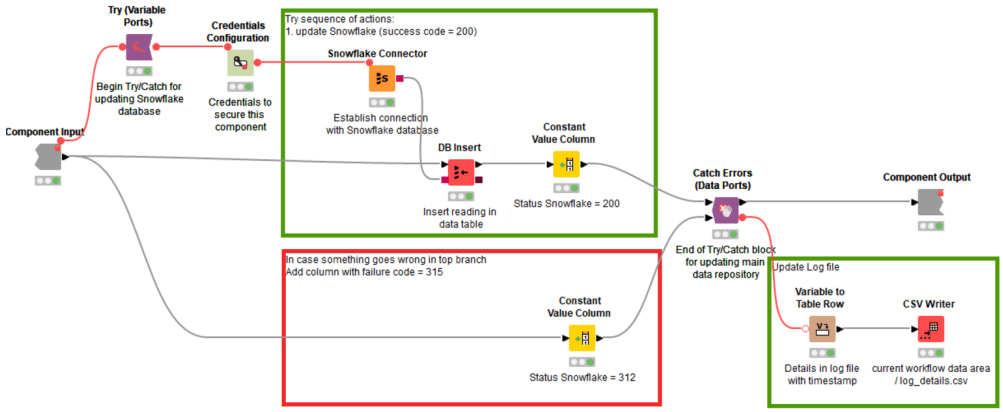


Fig. 4: Try/Catch block for handling error when updating Snowflake database.

9. Send GET Request
In summary, the sensor board sends a GET Request to the Weather Station web service. Thus, the Weather Station web service, that is waiting on the KNIME Server, accepts the data in the Request, adds the time stamp, inserts data in data repositories, and produces a Response back to the sensor board describing the success or failure of the execution.

Summary of Our IoT System - Without Coding

We have implemented an IoT system with KNIME Analytics Platform and an Arduino board. Most of the project was carried on without coding, except for the Arduino scripts. This office weather station is a complete IoT system in the sense that it went from collecting the weather data from a sensor board to storing and analyzing the same data for next day predictions.

The architecture of our IoT system consists of: a sensor board to collect the data, a repository system to store the data, a backup repository system, a web service to transfer the data, and a sARIMA model to perform the predictions for the next day.

We collected the temperature values from the sampled signal via the

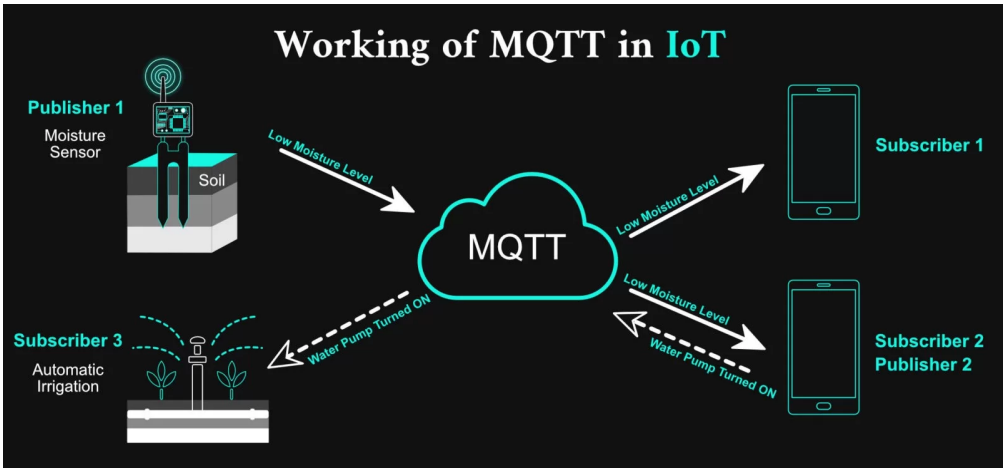
sensor board “MKR IoT Carrier” by Arduino along with a Wi-Fi MKR 1010 module. The sensor board was installed right out of our office window and calibrated as suggested by the Arduino in its example document of Calibration. Please note that this is one of the techniques for calibration, there are more techniques posted by the Arduino community in the project hub.

We adopted an installation of the Snowflake database to store the data. We also used a Google sheet as a backup system in case a recovery was ever needed. Then, we developed a REST service to transfer data from the board onto the data repository: the REST service resides on the KNIME server and waits to be called by the sensor board to transfer the data.

The complicated part was actually to set the Arduino board properly and to set a reasonable calibration of the sensors on it. Thanks to the open nature of KNIME software and the ability to connect and blend with other technologies, the next phase of building a REST service to import the data into Snowflake, aggregating them, training and deploying the sARIMA model, and building a dashboard was easy and - all codeless.

In our next article we will explain how we used the collected data and trained a sARIMA model to forecast the average temperature for the next hour given the average temperature values for the past 24 hours. The trained model was then deployed to produce a dashboard for us to consume the maximum and minimum temperatures of the upcoming day along with hourly forecasts for the next 24 hour from current time.

WHAT IS MQTT PROTOCOL FOR IOT DEVICES?



References

➔ https://www.knime.com/blog/data-collection-from-iot-sensors - > Data Collection from IoT Sensors
➔ https://tech.forums.softwareag.com/t/simulating-csv-data-in-cumulocity-iot-two-approaches/281502 ->
➔ https://psiborg.in/advantages-of-using-mqtt-for-iot-devices/ - >

➔ https://github.com/fdbatista/nodejs-torrey-weighing-machine
➔ https://projecthub.arduino.cc/user334153146/diy-speedometer-using-arduino-and-processing-android-app-ebc20b
➔ https://www.se.com/in/en/faqs/FA221729/
➔ https://stackoverflow.com/questions/66500743/how-to-get-value-from-the-output-of-a-weighing-scale
➔ https://www.codeproject.com/Questions/5291295/Read-weighing-scale-data-from-RS232
➔ https://github.com/fdbatista/nodejs-torrey-weighing-machine/blob/master/src/util/port-handler-util.mjs
➔ https://stackoverflow.com/questions/58568538/read-weighing-scale-data-via-rs232-to-usb-cable-in-python?rq=3
➔ https://www.perplexity.ai/search/Read-data-from-pocm9GNEQlu3vGmbo7HCUQ

Main Code: -

```
import SerialPort from 'serialport'
import AxiosUtil from './axios-util.mjs'
import ParserUtil from './parser-util.mjs'
import Readline from '@serialport/parser-readline'

export default class PortHandlerUtil {

  static baudRate = parseInt(process.env.BAUD_RATE)
  static lineDelimiter = ' kg'

  static build() {

    let that = this

    return new Promise(function (resolve, reject) {
      PortHandlerUtil.getActivePort().then((port) => {
        if (!port) {
          throw "ALERT: No valid port detected."
        }

        const serialPort = new SerialPort(port.path, {
          baudRate: that.baudRate
        })

        const parser = serialPort.pipe(
          new Readline({
            delimiter: that.lineDelimiter
```

```
            })
        )

        parser.on("data", function (data) {
            let machineWeight =
ParserUtil.machineReadingToFloat(data)

            if (machineWeight > 0) {
                AxiosUtil.post(machineWeight)
            }

        });

        serialPort.on("open", function () {
            console.log("Port open")
        });

        serialPort.on("error", function (error) {
            console.log(error)
        });

        serialPort.on("close", function () {
            console.log("Port closed")
        });

        resolve(serialPort)
      })
    })

  }

  static getActivePort() {
    return new Promise(function (resolve, reject) {
      return SerialPort.list().then((ports) => {
        let vendorId = process.env.VENDOR_ID
        let productId = process.env.PRODUCT_ID
        let port = ports.find(port => port.vendorId ===
vendorId && port.productId === productId)

        resolve(port)
      })
    })
  }

  static getAvailablePorts() {
```

```
        return new Promise(function (resolve, reject) {
            return SerialPort.list().then((ports) => {
                let availablePorts = ports.filter(port =>
    port.vendorId && port.productId)

                resolve(availablePorts)
            })
        })
    }

}
```

Read weighing scale data from RS232

Summary of CodeProject Post on Reading Weighing Scale Data via RS232 in C#

Question: A user is attempting to read data from a YH-T7E weighing scale via RS232 in a C# application. The user has tried different approaches to read the data but is facing issues with inconsistent data and improper display of weight values.

Details:

- **Device Output:**
  - The weighing scale outputs data at a baud rate of 1200.
  - Example weight on scale: 3.900 kg.
  - Example output string: 009.300.

Initial Code Attempt:
csharp
Copy code

```
private void serialPort1_DataReceived(object sender,
SerialDataReceivedEventArgs e){
  while (_serialPort.BytesToRead > 0) {
     hex += string.Format("{0:X2} ", _serialPort.ReadByte());
  }
  textBox1.Text = String.Empty;
  if (hex != "") {
     byte[] data = FromHex(hex.Trim());
     int lengthOfNumbers = data.Length;
     textBox2.Text = lengthOfNumbers.ToString();
     textBox1.Text = Encoding.ASCII.GetString(data, 0,
data.Length).Trim();
  } else {
     textBox1.Text = "no";
```

```
    }
}

public byte[] FromHex(string aHex) {
    aHex = aHex.Replace(" ", "");
    byte[] raw = new byte[aHex.Length / 2];
    for (int i = 0; i < raw.Length; i++) {
        raw[i] = Convert.ToByte(aHex.Substring(i * 2, 2), 16);
    }
    return raw;
}
```

-

Alternate Code Attempt with Timer:

csharp

Copy code

```
private void timer2_Tick(object sender, EventArgs e) {
    while (this.serialPort1.BytesToRead > 0) {
        var dataLen = this.serialPort1.BytesToRead;
        var byteArray = new byte[dataLen];
        this.serialPort1.Read(byteArray, 0, dataLen);
        var txt = Encoding.UTF8.GetString(byteArray);
        this.textBox4110.Text = txt;
    }
}
```

-

Issues Faced:

- The initial code reads data but displays the same value repeatedly in a concatenated format.
- The timer-based approach shows inconsistent values with each refresh, displaying the weight in different formats (e.g., 50,000 kg as 50,000, 050.00, 00.500).

Suggestions and Solutions:

1. **Solution 1 by OriginalGriff:**
    - **RS232 Characteristics:** Emphasized the slow nature of RS232 communication and the need to handle it properly.
    - **Event Handling:** Recommended avoiding direct UI updates from the DataReceived event due to threading issues.
    - **Full Message Assembly:** Suggested using a BackgroundWorker to assemble the full message and

then pass it to the UI.
    - **Start/Stop Characters:** Highlighted the use of boundary characters (like =) to mark the start and end of data sequences.
2. **Solution 2 by Ralf Meier:**
    - **Buffer Handling:** Advised checking if there are at least 6 characters in the buffer before reading to ensure complete data.
    - **Timer Interval:** Suggested adjusting the timer interval to better align with data arrival.
    - **Sequence Indicators:** Recommended using specific characters (e.g., =) to identify the start of new data sequences.

Key Takeaways:

- **Threading:** Avoid direct UI manipulation from non-UI threads.
- **Data Assembly:** Assemble complete data messages before processing.
- **Buffer Management:** Read sufficient characters to ensure complete data.
- **Boundary Characters:** Use start/stop characters to delimit data sequences.

These solutions aim to handle the slow nature of RS232 communication and ensure accurate and consistent data reading from the weighing scale.

Summary of the Stack Overflow Post on Reading Weighing Scale Data via RS232 to USB Cable in Python

Question: A user is attempting to read data from a YH-T7E weighing scale using a Raspberry Pi via an RS232 to USB cable in Python. Despite various attempts, the user is receiving an empty string from the serial read operation. They have tried different methods including using minicom but still receive no data.

Details:

Initial Code Attempt:
python
Copy code
```
import serial
import time
```

```
ser = serial.Serial('/dev/ttyUSB0', baudrate=9600,
parity=serial.PARITY_EVEN,
                    stopbits=serial.STOPBITS_ONE,
bytesize=serial.SEVENBITS, timeout=1)

while True:
  print(ser.read())
```

- The output is an empty byte string (b'').

Alternate Attempt:
python
Copy code

```
while True:
  data = ''
  while ser.inWaiting() > 0:
    data += ser.read()
  if data:
    print("Received: ", data)
```

- This code also waits indefinitely without receiving data.
- **Verification Steps:**
  - Confirmed RS232 connection was detected by the system.
  - Tried using minicom with correct baud rate settings but received no data.

Suggestions and Comments:

- **Configuration Verification:**
  - Ensure the correct baud rate, parity, data bits, and stop bits are set.
  - Try different settings such as parity set to NONE and data bits to 8.
  - Use a serial terminal program to verify data reception.
- **Physical Connection Check:**
  - Verify that the TX from the scale is connected to the RX on the RS232 port and that grounds are connected.
  - Test the USB RS232 adapter on another device to ensure it's working correctly.

Resolution:

- The user later found that the issue was due to improper wiring of the weighing scale. Once the wires were connected correctly, data was received successfully.

Key Takeaways:

1. **Correct Wiring:** Ensure proper physical connections between devices.
2. **Serial Configuration:** Verify and test various serial port settings.
3. **Hardware Testing:** Confirm the functionality of the USB to RS232 adapter with other devices.
4. **Use Serial Terminals:** Utilize tools like minicom to troubleshoot data reception before implementing in code.

This post highlights the importance of thorough verification of both hardware connections and software configurations when dealing with serial communication issues.

7. Conclusion

This report provides a comprehensive guide to setting up and programming a system to read data from a weighing machine via a USB to serial interface using both Python and JavaScript. The implementation details include physical setup, software configuration, and code examples for parsing and storing the weight measurements.

By following these steps and utilizing the provided code, you can successfully capture and process weight data from your weighing machine.