

Ein Feinstaub-Messknoten für das LoRaWAN IOT-Netzwerk „The Things Network“

Bernd Laquai, 04.11.2018

Gemessen daran, wie lange ein Mensch ohne ein gewisses „Lebensmittel“ überleben kann, ist die Atemluft sicher die Nummer 1 in der Rangliste. Aus diesem Grund wird auch verständlich, warum vielen Menschen sehr daran liegt, dass die Luft sauber bleibt. In wohlhabenden, demokratischen Hochtechnologie-Ländern kann man dieses Bedürfnis deutlich an Citizen Science Bewegungen bemerken, wo Bürger selbst versuchen über den Einsatz von Low-Cost Technologie Luftqualität messbar zu machen um sich einer von wirtschaftlichen Interessen allzu sehr geprägten Politik entgegenzustellen. Ein Beispiel ist das Feinstaub-Messnetz von Luftdaten.info mit über 1000 Messknoten, welche von Bürgern selbst hergestellt wurden und noch werden und deren Daten vom Open Knowledge Lab (OK Lab) in Stuttgart gesammelt werden.

Die Luftqualität wird von vielen Parametern bestimmt, dazu zählen Schadgase wie Stickoxide, Ozon, Kohlenmonoxid usw. aber auch die Anzahl- oder Massenkonzentration von Partikeln (Particulate Matter, PM). Für diese Parameter hat auch die EU Richt- und Grenzwerte vorgegeben, die von ihren Mitgliedstaaten eingehalten werden müssen, was in großen Städten oft nicht der Fall ist und zu heftigen politischen Diskussionen und gerichtlichen Auseinandersetzungen führt. Solche Parameter sind aber äußerst aufwändig zu messen, sofern man genau messen will. Professionelles Equipment liegt im Kostenbereich von mehreren zehntausend Euro und erfordert Einiges an Kenntnissen zur korrekten Bedienung und Interpretation. Deswegen wird selbst von den Behörden, welche mit der Überprüfung der Luftqualitäts-Parameter beauftragt sind, nur an verhältnismäßig wenig Stellen gemessen und es kommt selbst dort immer wieder zu Messfehlern.

Nun wurden aber in den letzten Jahren, vor allem wegen den massiven Problemen mit der Luftverschmutzung in China, sehr kleine und kostengünstige Low-Cost Sensoren für Feinstaub entwickelt, welche in Heimgeräten, ähnlich einem Barometer, oder auch in Klimaanlage eingesetzt werden. Diese kleinen Sensoren arbeiten nach dem Laser-Streulichtverfahren und können meist nur Partikelgrößen bis ca. 5µm zuverlässig messen und sind für die Anwendung in Innenräumen gedacht. Nun kamen Citizen-Scientists auf den Gedanken auch an verkehrsbelasteten Straßen in der Außenluft flächendeckend Feinstaub zu messen und setzten dabei ebenfalls solche billigen Sensoren ein um selbst ein Gefühl für die Luftverschmutzung mit Partikeln in ganzen Stadtteilen und Wohngebieten zu bekommen, vor allem dort wo die Behörden normalerweise nicht messen.

Dabei zeigte sich jedoch zunächst, dass das im Prinzip zwar möglich ist, dass aber teilweise erhebliche Messfehler entstehen, von denen man Kenntnis haben muss um eine korrekte Bewertung vornehmen zu können. Mit amtlichen, gerichtsfesten Messungen lassen sich die so erhobenen Messdaten nicht ohne Weiteres vergleichen. Dafür gibt es im Wesentlichen zwei Gründe. Zu einen schreiben die EU-Richtlinien vor, die Massenkonzentration von trockenem Feinstaub zu messen, d.h. ohne Feuchteinfluss. Daher muss auch das Aerosol (Luft-Partikel-Gemisch) vor der Messung getrocknet werden, so dass keine Nebeltröpfchen mehr enthalten sind und salzhaltige Partikel, welche durch ihre hygroskopischen Eigenschaften Feuchte aus der Luft aufgenommen haben, diese wieder abgeben, bevor die Masse bestimmt wird. Macht man das nicht, so führt das insbesondere in den feinstaubträchtigen Wintermonaten mit Nebel und hohen Luftfeuchten zu massiven Messfehlern. Dabei werden dann Nebeltröpfchen als Partikel gewertet und salzhaltige Partikel in der Masse völlig überschätzt. Ein anderer Messfehler rührt daher, dass die Low-Cost Sensoren bauartbedingt keine Partikel mit Durchmesser größer etwa 5µm wahrnehmen können und diese deswegen nur aus den

Anzahlkonzentrationen kleinerer Partikel abschätzen. Wenn daher mit Low-Cost Sensoren ermittelte Massenkonzentrationen von Partikeln bis etwa 10µm (PM10) gegen den EU-Grenzwert verglichen werden, dann kommt es auch dabei zu stärkeren Abweichungen, weil die Masse der großen Partikel vor allem im straßennahen Bereich unterschätzt wird.

Dennoch kann man erkennen, dass insbesondere die Partikelfraktion bis etwa 2.5µm (PM2.5) bei Luftfeuchten unter 70% in der Regel durchaus brauchbar genau gemessen wird, die Schätzung für PM10 bei Luftfeuchten unter 70% abseits der großen Straßen nicht völlig daneben sind und für flächige Messungen mit vielen Netzknoten derzeit kaum andere ökonomisch sinnvollen Alternativen bestehen. Daher machen solche Messnetze bei korrekter Bewertung der Ergebnisse durchaus noch Sinn.

Das Kommunikations-Konzept des Feinstaub-Sensorknotens des OK Lab besteht darin, dass der PM-Sensor mit einem Mikrocontroller mit WLAN-Funkmodul verbunden ist, der die Daten über den WLAN-Router z.B. in einem Wohnhaus in das Internet überträgt. Nun gibt es aber doch zahlreiche Orte, welche für eine Messung interessant wären, die nicht durch ein WLAN abgedeckt sind, oder wo ein WLAN Funknetz z.B. nachts nicht vorhanden ist. In solchen Fällen ist die Übertragung der Daten ins Internet nur mit Funknetzen größerer Reichweite möglich. Die Mobilfunknetze sind für solche Zwecke weniger geeignet, da einerseits nur geringe Datenraten übertragen werden müssen und daher nur eine geringe Bandbreite notwendig ist und andererseits die Registrierung einer Mobilfunk SIM-Karte immer aufwändiger wird. Genau für diesen Anwendungsfall bietet sich aber der Einsatz eines Low-Power Wide Area Network (LP-WAN) an, welches auch unter dem Namen IoT-Netz (Internet-of-Things-Netz) bekannt ist.

Solche IoT-Netze schießen derzeit aus dem Boden wie Pilze, und in der Tat findet eine rasante Entwicklung in der Elektronik und im Netzaufbau statt um diese Netzwerk-Technologie in der Praxis auch tatsächlich anwendbar zu machen. Meist sind es Funknetzwerke für die Datenübertragung, die mit sehr niedriger Leistung auf der Teilnehmerseite arbeiten und nur geringe Daten-Bandbreiten zur Verfügung stellen (Bytes pro Minute), dafür aber Reichweiten im Kilometerbereich erreichen. Architektonisch sind sie so aufgebaut, dass viele Sensorknoten ihre Daten zu vergleichsweise wenigen Gateways (ganz ähnlich einem WLAN-Router) funken, welche die empfangenen Daten ins Internet an die Server des IoT-Netzwerk-Providers weiterleiten. Im Netzwerk-Backend des Providers werden mehrfach empfangene Pakete aussortiert und geordnet. Danach werden die Daten je nach Wunsch des Sensorknoten-Betreibers entweder an „normale“ Internet-Server weitergereicht oder an weiterverarbeitende Dienste z.B. zur Darstellung auf Webseiten oder zum Versand von Benachrichtigungen (z.B. per E-Mail) übergeben.

Die Tatsache, dass diese neuen Funk-Technologien mit sehr niedrigem Leistungsverbrauch hohe Reichweiten erreichen, ermöglicht vor allem den Aufbau von dezentralen Netzen mit vielen kleinen IoT-Netzknoten, auch an Stellen in der Umwelt, wo ein WLAN Router nicht so ohne weiteres hinreicht. Von daher sind diese Funknetze vor allem für eine datensparsame Umweltüberwachung fast ideal geeignet, insbesondere dann, wenn flächige verteilte Sensoren ihre Messdaten autonom erfassen und komprimiert über ihre Funkanbindung durch das Internet an eine Datenbank zur Visualisierung und zur weiteren Nachbearbeitung übertragen sollen.

Diese neue IoT-Technologie soll hier am Beispiel eines Sensorknotens für die Erfassung der Feinstaub-Massenkonzentration in der Luft (gemessen in µg/m³ Luft) und der Datenübertragung über das LoRaWAN (Long Range Wide Area Network) zu dem Netzwerk-Provider „The Things Network (TTN)“ dargestellt werden. Zur Visualisierung und Auswertung der Daten soll die IoT-Plattform „Cayenne“ von der Firma Mydevices verwendet werden. Sowohl die Dienste des TTN, wie auch in gewissem Umfang die Dienste von Mydevices sind kostenfrei, sofern die Anwendung keinen kommerziellen Hintergrund hat. Im Gegensatz dazu sind die übrigen bereits nutzbaren IoT-Netzwerke mit nennenswerter

Verbreitung in Europa, wie beispielsweise Sigfox oder LTE IoT-NB (Telekom), ähnlich wie der normale Mobilfunk, immer mit gewissen Kosten verbunden.

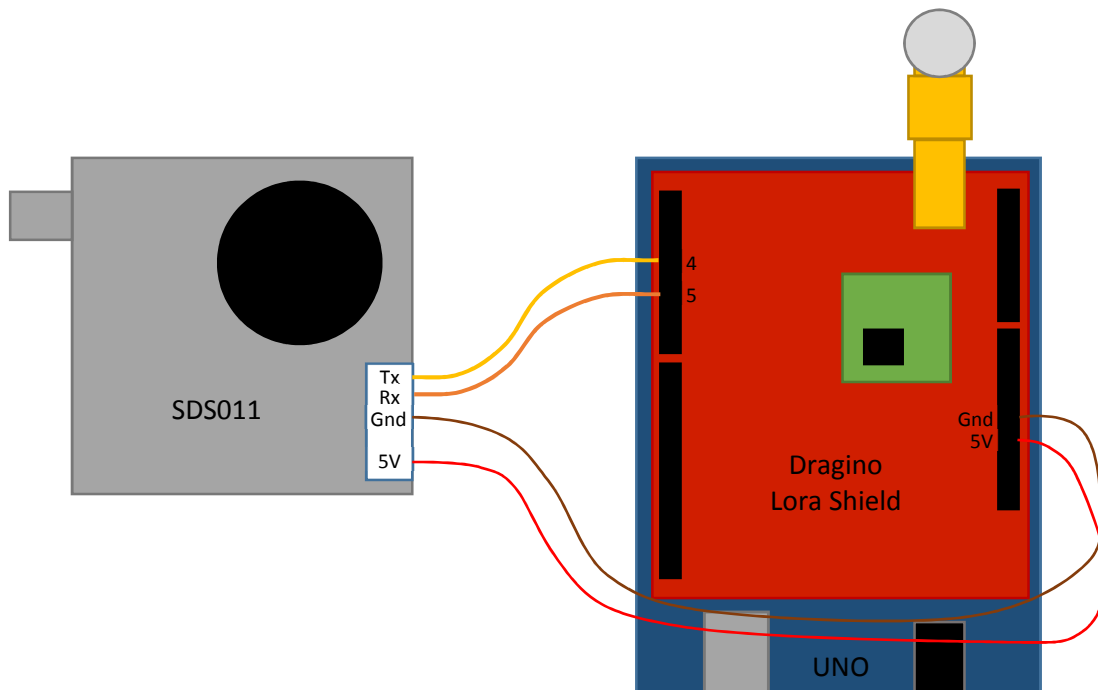


Abb. 1: Schematischer Aufbau des LoRaWAN Feinstaub-Messknotens

Der hier vorgestellte Feinstaub-Netzwerkknoten besteht aus einem Low-Cost PM-Sensor SDS011 der chinesischen Firma Nova-Fitness und einem Arduino Mikrocontroller mit einem kostengünstigen LoRa-Funknetz-Shield der chinesischen Firma Dragino. Das Dragino LoRa Shield verwendet ein LoRa Funkmodul des chinesischen Herstellers HopeRF, welcher offensichtlich eine Lizenz zur Implementierung des von Semtech patentierten LoRa Chirp-Modulationsverfahrens erworben hat. Sehr wichtig ist, dass dieses Funkmodul im Gegensatz zu anderen, nicht über eine UART Schnittstelle, sondern über eine SPI Schnittstelle mit den Signalen MISO, MOSI, SCLK und SS gesteuert wird (siehe auch Wikipedia, Serial Peripheral Interface), welche auf Arduino Pins abgebildet werden. Deswegen ist eine spezielle Bibliothek für den Betrieb des Funkmoduls nötig.

Dagegen muss die Verbindung des SDS011 Sensors zum Arduino durch eine serielle UART Schnittstelle erfolgen. Wenn die Serielle Schnittstelle des Arduino für die Ausgabe von Debug-Information mit der Serial.print() Methode weiterhin zur Verfügung stehen soll, muss daher eine zusätzliche Software-Serielle Schnittstelle auf zwei noch freien Pins des Arduino mit Hilfe der SoftwareSerial-Bibliothek erzeugt werden. Die Pins, die hier dafür genutzt werden sind die Digital-Pins 4 und 5.

Pin Mapping For LoRa

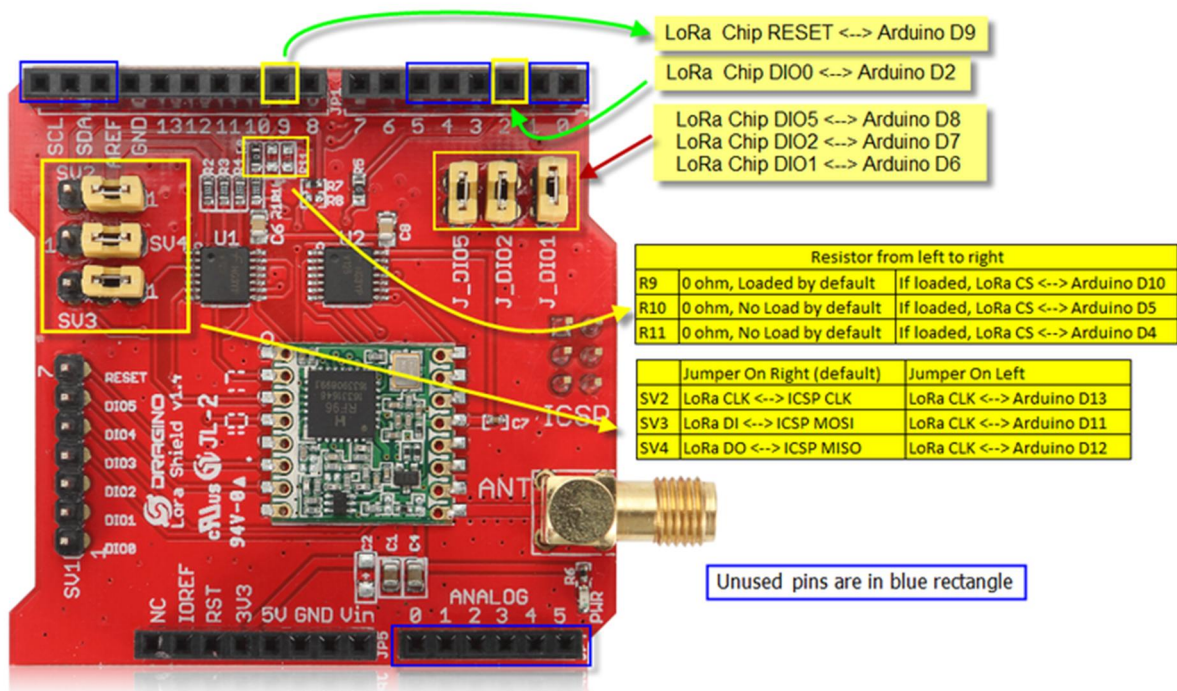


Abb. 2: Pin Belegungen des Dragino Lora Shield (Quelle: wiki.dragino.com)

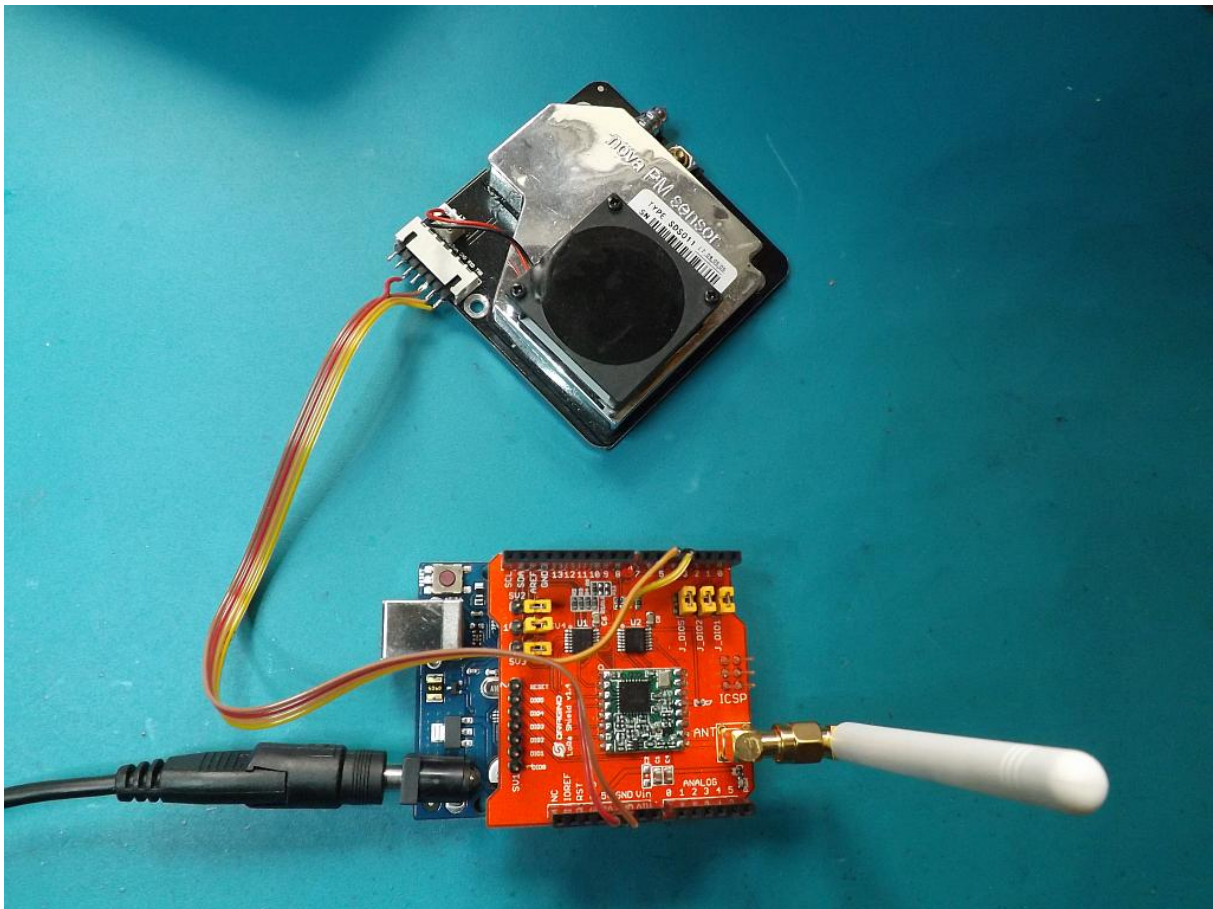


Abb. 3: Realer Aufbau des Feinstaub-Messknotens

Bevor man sich jedoch entscheidet, einen LoRaWAN Netzknoten zu entwickeln, empfiehlt es sich die Netzwerkabdeckung in der Gegend, in welcher der Knoten aufgestellt werden soll, zu prüfen. Da die LoRaWAN Gateways in der Regel von ambitionierten Bastlern und Sponsoren der Open Source Community oder von Hochschulen aufgestellt werden und der Netzwerkaufbau erst begonnen hat, deckt das Netzwerk zumeist nur die größeren oder besonders innovativen Städte ab. Die beste Möglichkeit der Prüfung bietet das TTN-Mapper Portal <https://ttnmapper.org/>. Hier werden die von Freiwilligen erfassten Daten zur Netzabdeckung dargestellt, welche die Empfangssignalstärke mit Hilfe von mobilen IoT-Netzknoten vermessen. Die auf dem Portal dargestellten Graphen zeigen mit farblich codierten Strahlen an, wie gut das Signal der Netzknoten an unterschiedlichen Orten von den verschiedenen Gateways empfangen wurde.

Für das Mapping wird in der Regel ein Mobiltelefon benutzt, auf welcher die ttnmapper App läuft. Die App verwendet die Standortdaten des Telefons und übermittelt diese zusammen mit der Kennung für den mobilen IoT-Netzknoten an einen Server des ttnmapper Portals. Gleichzeitig sendet der IoT-Netzknoten eine beliebige Nachricht an das TTN Netzwerk, welches die empfangenden Gateways und deren Position ebenfalls an das ttnmapper Portal weitergibt. Nun geht man davon aus, dass sich der mobile IoT-Netzknoten und das Mobiltelefon an ein und derselben Stelle befinden und kann so die Empfangssignalstärke am jeweiligen Gateway dem Ort des Mobiltelefons zuordnen.

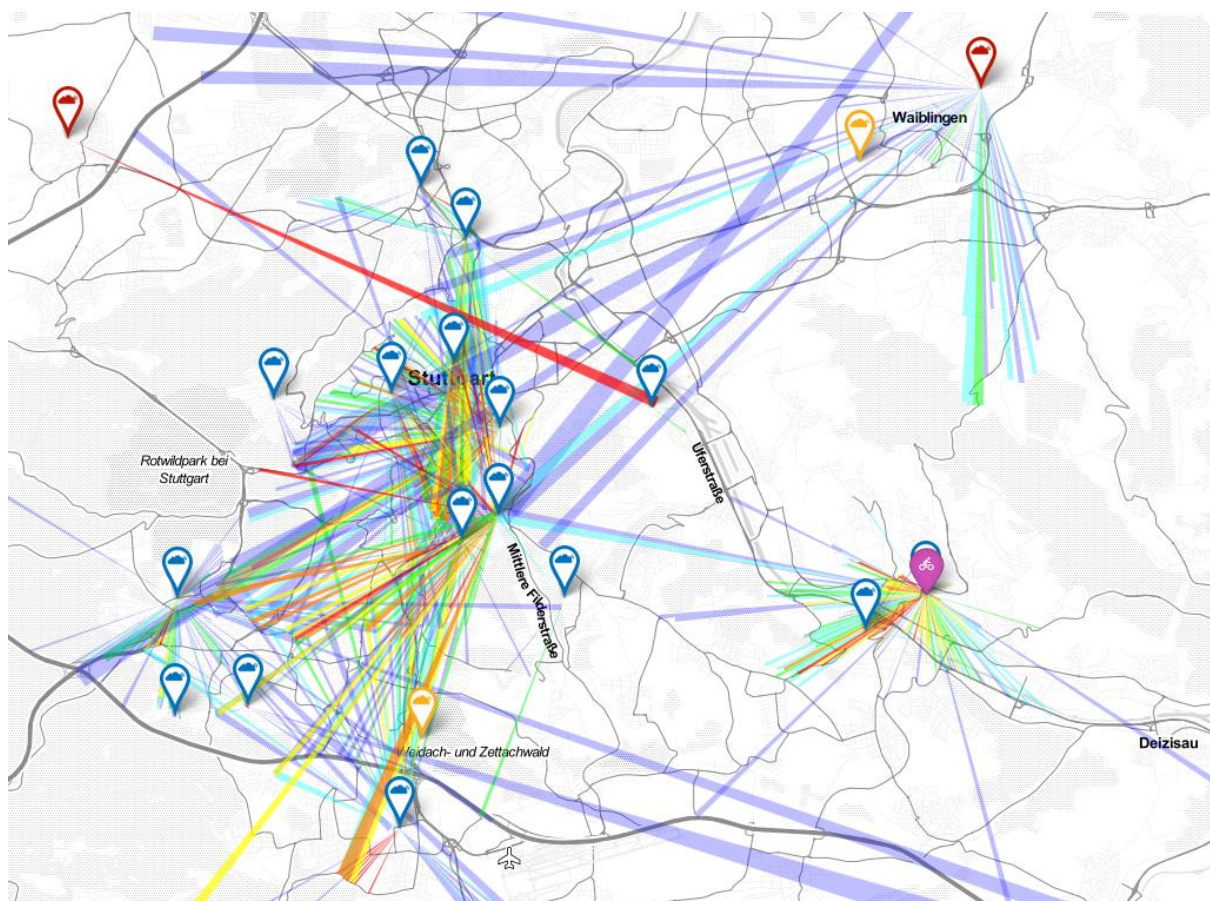


Abb. 4: Der derzeitige Status der TTN-LoRaWAN Netzabdeckung in der Gegend um Stuttgart

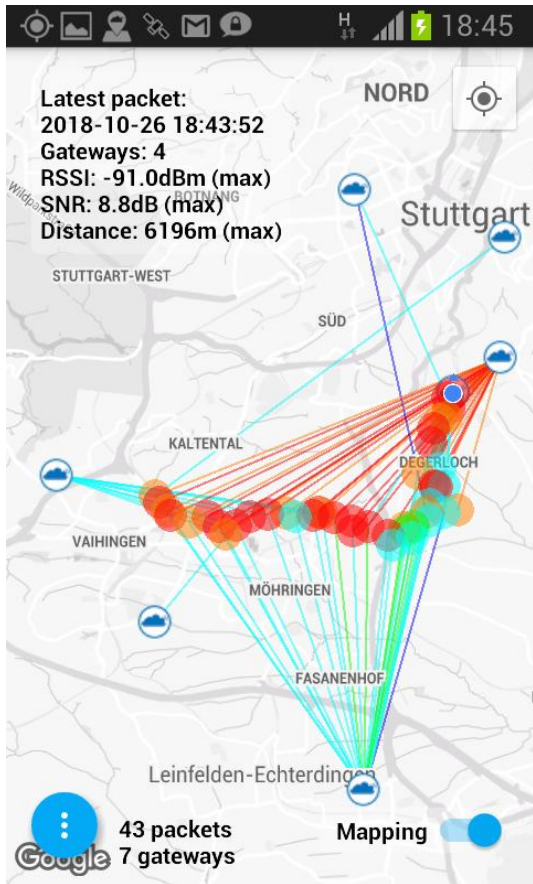


Abb. 5: Mapping Ergebnis eines mobilen TTN LoRaWAN Netzknotens im Süden Stuttgarts. Die blauen Wolken stellen die Position der Gateways und deren Empfangssignalstärke in farbigen Strahlen dar.

Wenn nun für das Dragino Shield mit dem HopeRF Funkmodul als IoT-Netzwerk das „The Things Network“ (TTN) benutzt werden soll, bleibt kaum eine andere Alternative übrig, als die ursprünglich von IBM entwickelte LMIC Bibliothek zur Kommunikation einzusetzen, da die von TTN selbst zur Verfügung gestellte Bibliothek nur Funkmodule mit UART unterstützt, welche in der Regel deutlich teurer sind. Der Nachteil der LMIC ist leider, dass sie äußerst trickreich implementiert ist (event-basiert), schwer zu verstehen ist und viel Speicherplatz benötigt. Dennoch lohnt sich der Einsatz derzeit schon rein aus Gründen der Kosten für das günstigere Funkmodul.

Die Hardware Schnittstelle wird für die LMIC über eine Pin Mapping Anweisung passend gemacht. Hier werden die Pins beschrieben, welche für die SPI-Kommunikation mit dem Funkmodul nötig sind und zu welchen Arduino Digital-Pins sie auf dem Dragino Shield verbunden sind.

Das LMIC Pin mapping für das Dragino LoRa Shield (ohne GPS) sieht wie folgt aus:

```
const lmic_pinmap lmic_pins = {
  .nss = 10,
  .rxtx = LMIC_UNUSED_PIN,
  .rst = 9,
  .dio = {2, 6, 7},
};
```

Dabei ist die Bedeutung der Pins für des HopeRF RFM95 Funk Modul folgende:

```
.nss = NSS (SPI select)
.rxtx = RXTX/RFMOD (nicht auf HopeRF, Umschaltung von Tx auf Rx Mode läuft über DIO Pins)
.rst = NRESET
.dio {DIO0, DIO1, DIO2}
```

Dieses PIN-Mapping muss stimmen, insbesondere deswegen, weil es auf dem Dragino Shield auch gegenüber diesem Default geändert werden kann.

Darüber hinaus muss dem LMIC Code noch die Routine `getSdsData()` für das Auslesen des SDS011 Sensors hinzugefügt werden. Sie folgt den Angaben des Herstellers Nova Fitness für die UART Schnittstelle des SDS011. Die LMIC selbst wird in der `setup()`-Routine des Arduino zunächst initialisiert und dann mit `do_send()` als regelmäßiger Event so installiert, dass sie in einem festen Zeitintervall wiederholt aufgerufen wird (`const unsigned TX_INTERVAL = 60;`). Deswegen bleibt die `loop()` Routine des Arduino auch leer und wird nie erreicht. Da das Auslesen des SDS011 von der Verfügbarkeit dieser Daten abhängt, die ebenfalls asynchron entstehen, muss das Auslesen auch in einer Schleife laufen, die erst dann von einer `return` Anweisung beendet wird, wenn die Daten vollständig sind. Danach werden die Daten für das Weiterleiten vom TTN-Netzwerk Backend an die IoT-Visualisierungsplattform Cayenne unter Verwendung der Cayenne-LPP Bibliothek (Cayenne Low Power Payload, LPP) formatiert und danach mit `LMIC_setTxData2(1, lpp.getBuffer(), lpp.getSize(), 0);` an TTN versandt.

Innerhalb der TTN Plattform wird der IoT-Netzknoten innerhalb des Benutzerkontos angemeldet und festgelegt, was das TTN Netzwerk-Backend mit den empfangenen Zählraten-Daten tun soll. Prinzipiell können die vom TTN empfangenen Zählratenwerte bereits über ein Browserfenster als Zahlen im Hex-Format beobachtet werden. Da es für Luftqualitätsmessungen, wie auch für andere Daten sinnvoll ist, den zeitlichen Verlauf des gemessenen Sensorsignals anzuzeigen, empfiehlt es sich diese noch einmal an einen weiteren internetbasierten Dienst weiterzuschicken, der die Aufgabe der grafischen Darstellung im Browserfenster übernimmt. Alternativ dazu kann man die Daten natürlich auch an einen eigenen Webserver weiterschicken, auf dem ein Skript (z.B. ein in PHP geschriebenes) die Daten entgegennimmt und in einer eigenen Datenbank einträgt (z.B. mittels SQL). Mit einem weiteren Skript könnten diese Daten dann ebenfalls, auf Anfrage eines Browsers hin, in eine Grafik gezeichnet oder textuell ausgegeben werden. Die serverseitige Programmierung und das Bereithalten der Datenbank werden einem aber vollständig abgenommen, wenn man den Weg über einen Dienst wie Cayenne wählt. Dafür ist aber die verfügbare Flexibilität bei der Darstellung etwas stärker eingeschränkt.

Die Firma Mydevices (<https://mydevices.com/>) bietet die Visualisierung von Daten aus IoT-Netzen mit dem Cayenne-Tool als Dienstleistung an, die für private Kleinanwendungen bisher noch kostenlos und zeitlich nicht beschränkt ist. Gleichzeitig hat auch der TTN-Netzprovider eine Schnittstelle zu der Visualisierungsoberfläche Cayenne als eine mögliche Variante der Datenweitergabe (die sogenannte Integration) implementiert. Wählt man diese im Benutzerbereich für eine Klasse von Netzknoten (Application) aus, und hat man im Netzknoten die entsprechenden Aufrufe an die Cayenne LPP Bibliotheks-Schnittstelle (API) eingebaut, dann erzeugt dies nach einer einfachen Konfiguration der Oberfläche gleich die gewünschte Grafik. Außerdem bietet Cayenne noch den Export der Daten nach Excel an. Schließlich kann man noch Trigger-Punkte setzen, so dass Cayenne automatisch, z.B. nach Überschreiten eines Schwellwerts, eine E-Mail zur Warnung verschickt.

Sobald alle Verbindungen zwischen Sensor und Arduino hergestellt sind, muss der Uno noch in der Arduino Entwicklungsumgebung programmiert werden. Der dafür erforderliche Programm-Sketch findet sich im Anhang. Das Programm benötigt etwa 72 % des verfügbaren Speicherplatzes des Controllers auf dem Arduino Uno und 52% des dynamischen Speichers. Damit könnten also nicht mehr allzu viele zusätzliche Aktionen von dem Mikrocontroller wahrgenommen werden.

Allerdings müssen in dem Programm-Sketch vorher noch die erforderlichen Schlüssel, die bei der Anmeldung des Netzknotens beim TTN vergeben werden, eingetragen werden. Für die hier verwendete Einwahlmethode „OTAA“ (Over the Air Activation) benötigt man die App EUI, die Device

EUI und den App Key welche im Sketch eingetragen werden müssen. Dazu muss zunächst ein Account eingerichtet werden und eine Application hinzufügen werden. Dazu begibt man sich nach der Anmeldung auf die Seite „Console“. Unter Applications kann man nun verschiedene Applications hinzufügen. Unter einer Application versteht man eine Anwendung für mehrere Netzwerk-Knoten. Wenn man also eine oder mehrere Messstationen als Netzwerk-Knoten betreiben will, dann würde man die Application vielleicht mit „Feinstaub Messnetz“ bezeichnen und als Application Id den Namen „pmnet“ vergeben. Damit ergibt sich bereits die App EUI.

ADD APPLICATION

Application ID
The unique identifier of your application on the network

pmnet

Description
A human readable description of your new app

Feinstaub Messnetz

Application EUI
An application EUI will be issued for The Things Network block for convenience, you can add your own in the application settings page.

EUI issued by The Things Network

Handler registration
Select the handler you want to register this application to

ttn-handler-eu

Cancel Add application

Abb. 6: Registrierung einer Application bei TTN

Die Application EUI ist ein Schlüssel für die Application und wird beim Hinzufügen automatisch vergeben. Danach muss man den Netzwerk-Knoten registrieren. Dazu vergibt man dem Netzknoten unter Device Id noch einen Namen z.B. „pmnode1“. Da das HopeRF Modul selbst keine Device EUI gespeichert hat, muss man bei Device EUI auf das Wechselsymbol klicken, so dass die Registrierungssoftware von TTN selbst eine Device EUI generiert (es erscheint: „This field will be generated“). Der App Key wird ebenfalls von der TTN Registrierungssoftware generiert. Mit dem Klick auf „Register“ erhält man dann alle erforderlichen Keys und kann sie in verschiedenen Formaten kopieren (Kopiersymbol ganz rechts im jeweiligen Feld).

Ganz wichtig ist nun, dass die LMIC die App EUI und die Device EUI mit den Bytes in umgekehrter Folge (LSB first) benötigt – im Gegensatz zum App Key. Dazu klickt man zunächst auf das „<>“ Symbol um die Keys in Hex Notation anzuzeigen und dann auf das Wechselsymbol dahinter, was von MSB first auf LSB first umschaltet. Klickt man dann auf das Kopiersymbol „clip to clipboard“ ganz rechts, kann man die Hex Codes in der richtigen Reihenfolge in die Zwischenablage kopieren und von da mit Einfügen in den LMIC Programmcode kopieren. Den App Key kann man durch klicken auf das Augensymbol sichtbar machen, muss ihn dann aber mit MSB first in den LMIC Code übernehmen.

REGISTER DEVICE [bulk import devices](#)

Device ID
This is the unique identifier for the device in this app. The device ID will be immutable.

pmnode1

Device EUI
The device EUI is the unique identifier for this device on the network. You can change the EUI later.

this field will be generated

App Key
The App Key will be used to secure the communication between you device and the network.

this field will be generated

App EUI

70 B3 D5 7E D0 01 41 1D

Cancel Register

Abb. 7: Registrierung eines Feinstaub-IoT-Netzknosens bei TTN.

Das korrekte Kopieren der Codes ist nötig, damit die versendeten Pakete des Netzknosens an der richtigen Stelle im Netzwerk-Backend herauskommen und die Verschlüsselung der Daten richtig funktioniert.

Nun muss das Arduino LMIC Programm mit den richtigen Schlüssel-Nummern versehen noch auf den Arduino Uno hochgeladen werden. Danach beginnt dann der Netzknosens zu arbeiten und man kann die Daten, die empfangen wurden, unter dem Reiter „Data“ im Device Fenster von „pmnode1“ überprüfen. Die Datenbytes werden dabei im hexadezimalen Format als Payload angezeigt.

Bis dahin erreichen die gesendeten Daten also nur das Netzwerk-Backend des TTN-Netzwerks. Eine aussagekräftige Visualisierung der Daten oder ein Export ist aber an dieser Stelle bisher nicht möglich. Um das zu erreichen, kann jetzt zusätzlich eine Integration zur Application hinzugefügt werden, welche sich auf die Daten aller Netzknosens dieser Application auswirkt. Unter dem Menüpunkt Applications->pmnet-> Integrations kann Cayenne als Ziel ausgewählt werden. Um genau zu sein, handelt es sich hier um die Anwendung des Cayenne-LPP Protokolls für das Weiterleiten der Daten an den Dienstleister mydevices.com .

Overview Data Settings

APPLICATION DATA

pause clear

Filters: uplink downlink activation ack error


time	counter	port	
▲ 22:07:01	182	1	payload: 01 65 00 15
▲ 22:06:28	181	1	payload: 01 65 00 13
▲ 22:05:56	180	1	payload: 01 65 00 13
▲ 22:05:23	179	1	payload: 01 65 00 13
▲ 22:04:50	178	1	payload: 01 65 00 13
▲ 22:04:17	177	1	payload: 01 65 00 15
▲ 22:03:43	176	1	payload: 01 65 00 14
▲ 22:03:10	175	1	payload: 01 65 00 14
▲ 22:02:37	174	1	payload: 01 65 00 14

Abb. 8: Beobachtung der vom TTN Netzwerk-Backend empfangenen Daten

Applications > pmnet > Integrations

Overview Devices Payload Formats Integrations Data Settings

ADD INTEGRATION



Cayenne (v2.6.0)
myDevices
Quickly design, prototype and commercialize IoT solutions with myDevices Cayenne
[documentation](#)

Process ID
The unique identifier of the new integration process

pmprocess

Access Key
The access key used for downlink

no selection

Cancel Add integration

Abb. 9: Hinzufügen einer Cayenne Integration

Als nächstes muss man sich unter mydevices.com einen Account anlegen. Dann fügt man unter „Add new...“ Device/Widget -> Lora -> The Things Network -> Dragino Technology Lora Shield den Netzwerk-Knoten hinzu. In dem Fenster, welches nun am rechten Rand des Browsers aufgeht, gibt man ebenfalls die Device EUI als Schlüssel Nummer ein. Entscheidend ist hier, dass über den Eingabefeldern steht: „This device uses Cayenne LPP“ .

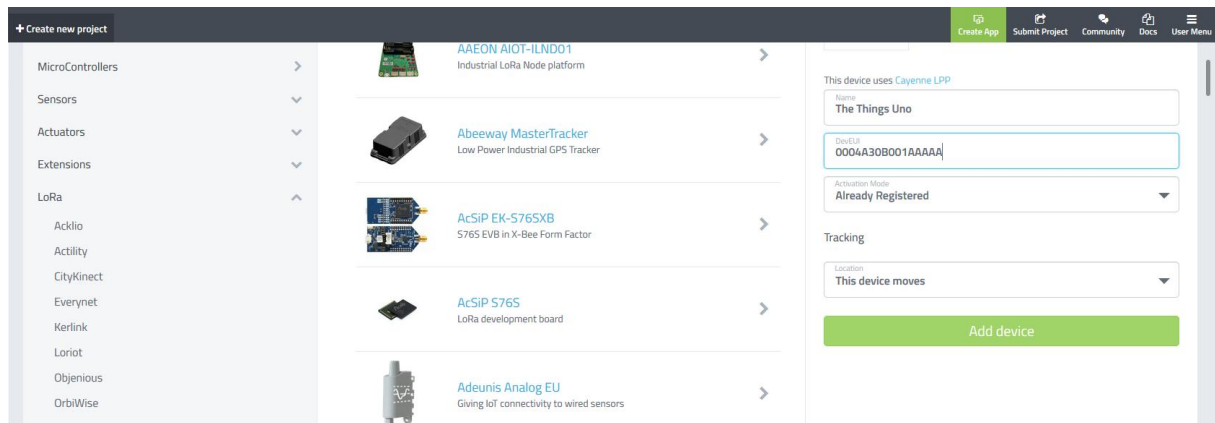


Abb. 8: Auswählen des IoT-Netzknos Typs bei Cayenne

Drückt man nun „Add Device“, dann entsteht ein Fenster mit der Meldung: „Your dashboard appears when Cayenne receives data from this device. Set up your device to transmit more frequently to speed up the process.“ Sobald also der Feinstaub-IoT-Netzknos Daten sendet, erscheint die Anzeige für die Zählrate als Luminosity(1). Das rührt daher, dass die Cayenne Visualisierungs-Plattform bisher keine Feinstaub-Massenkonzentration als Messgröße kennt. Deswegen wurden in den Programm-Sketch in die Routine `do_send` der Aufruf: `lpp.addLuminosity(1, PM2_5val);` platziert. Dieser bewirkt nun, dass der PM2.5 Messwert (in $\mu\text{g}/\text{m}^3$) ersatzweise als Luminosity für Kanal 1 erscheint. Dies kann aber bei der Anzeige im Dashboard von Cayenne mit dem Setting Symbol (Zahnrad) nach Belieben umbenannt werden.

Klickt man auf das Data-Symbol links oben im Fenster des Graphen, dann erscheint eine Liste der Werte. Der Zeitraum ist über „Custom“ einstellbar und die Daten können über den Download nach Excel exportiert werden. Cayenne ermöglicht darüber hinaus noch einige weitere nette Optionen für die Datenbankabfrage und die Visualisierung der Daten.

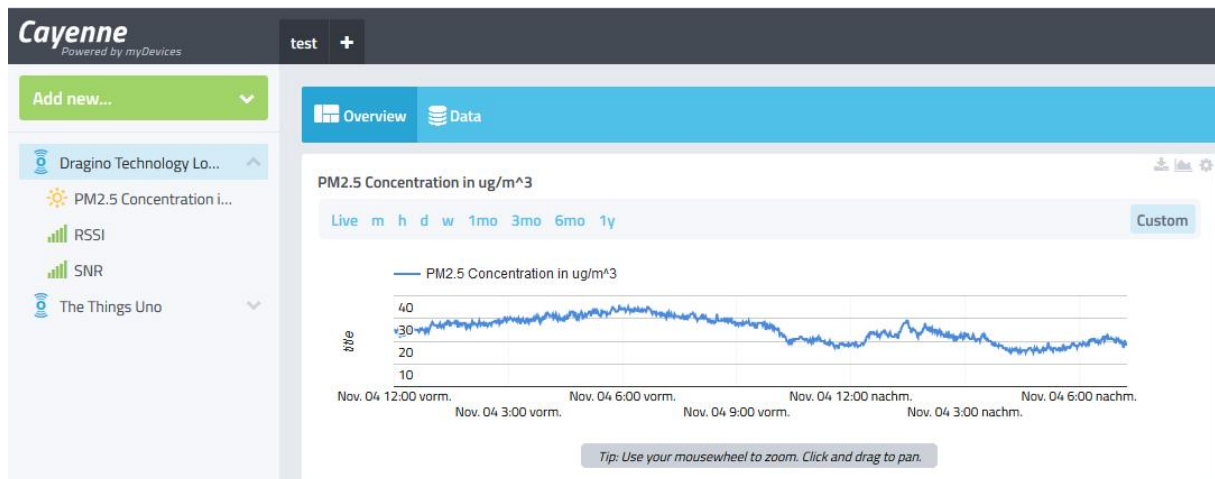


Abb. 10: Anzeige der Feinstaub-Massenkonzentration der Größenfraktion PM2.5 (Partikel kleiner als etwa 2.5 μm)

Anhang

Listing für den Arduino Uno mit Dragino LoRa Shield

```
//SDS011 sensor added Bernd Laquai, Nov. 3,18
//          ttn_otaa          example          taken          from:
https://github.com/lukastheiler/ttn_moteino/blob/master/ttn_moteino_otaa/ttn_moteino_otaa.ino

/*****
Original script is Copyright (c) 2015 Thomas Telkamp and Matthijs Kooijman
1) ttnctl devices register DEEDDEEDDEEDDEED
   INFO Generating random AppKey...
   INFO Registered device AppKey=94F00F5C07C2F536438600A54CAFF740 DevEUI=DEEDDEEDDEEDDEED
2) → ~ ttnctl devices info DEEDDEEDDEEDDEED
   Dynamic device:
   AppEUI: YOUR-OWN-APP-EUI
           {0x__, 0x__, 0x__, 0x__, 0x__, 0x__, 0x__, 0x__}
   DevEUI: DEEDDEEDDEEDDEED
           {0xDE, 0xED, 0xDE, 0xED, 0xDE, 0xED, 0xDE, 0xED}
   AppKey: 94F00F5C77C2F536438600A54CAFF740
           {0x94, 0xF0, 0x0F, 0x5C, 0x77, 0xC2, 0xF5, 0x36, 0x43, 0x86, 0x00, 0xA5, 0x4C, 0xAF,
0xF7, 0x40}
   Not yet activated
3) Run this code & be very patient. You should get the following messages:
   Starting
   155: EV_JOINING
   9276632: EV_JOINED
   Packet queued
4) You can now see that the device is activated, and if you run ttnctl subscribe DEEDDEEDDEEDDEED
   up, you'll see that the message was sent.
   → ttnctl devices info DEEDDEEDDEEDDEED
   Dynamic device:
   AppEUI: YOUR-OWN-APP-EUI
           {0x__, 0x__, 0x__, 0x__, 0x__, 0x__, 0x__, 0x__}
   DevEUI: DEEDDEEDDEEDDEED
           {0xDE, 0xED, 0xDE, 0xED, 0xDE, 0xED, 0xDE, 0xED}
   AppKey: 94F00F5C07C2F536438600A54CAFF740
           {0x94, 0xF0, 0x0F, 0x5C, 0x77, 0xC2, 0xF5, 0x36, 0x43, 0x86, 0x00, 0xA5, 0x4C, 0xAF,
0xF7, 0x40}
   Activated with the following parameters:
   DevAddr: 1C5055EB
           {0x1C, 0x50, 0x55, 0xEB}
   NwkSKey: 0F0D2D38AA050BFBFE5F42C591568963
           {0x0F, 0x0D, 0x2D, 0x38, 0xAA, 0x05, 0x0B, 0xFB, 0xFE, 0x5F, 0x42, 0xC5, 0x91, 0x56,
0x89, 0x63}
   AppSKey: 59E7EEAA7463948FC844A48DE70DD707
           {0x59, 0xE7, 0xEE, 0xAA, 0x74, 0x63, 0x94, 0x8F, 0xC8, 0x44, 0xA4, 0x8D, 0xE7, 0x0D,
0xD7, 0x77}
   FCntUp: 1
   FCntDn: 0
   *****/
#include <SoftwareSerial.h>
#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>
#include <CayenneLPP.h>

SoftwareSerial mySerial(4, 5); // RX, TX
unsigned char incomingByte;
unsigned char buf[10];
int PM2_5val = 0;
int Pm10val = 0;

CayenneLPP lpp(51);

static const u1_t APPEUI[8] = { <fill in your own APPEUI> }; // REVERSE ORDER
static const u1_t DEVEUI[8] = { <fill in your own DEVEUI> }; // REVERSE ORDER
static const u1_t APPKEY[16] = { <fill in your own APPKEY> };

// provide APPEUI (8 bytes, LSBF)
void os_getArtEui (u1_t* buf) {
    memcpy(buf, APPEUI, 8);
}
```

```

// provide DEVEUI (8 bytes, LSBF)
void os_getDevEui (u1_t* buf) {
    memcpy(buf, DEVEUI, 8);
}

// provide APPKEY key (16 bytes)
void os_getDevKey (u1_t* buf) {
    memcpy(buf, APPKEY, 16);
}

static osjob_t sendjob;
static osjob_t initjob;

// Schedule TX every this many seconds (might become longer due to duty
// cycle limitations).
const unsigned TX_INTERVAL = 60;

// Pin mapping adapted to Dragino shield
const lmic_pinmap lmic_pins = {
    .nss = 10,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 9,
    .dio = {2, 6, 7},
};

void onEvent (ev_t ev) {
    Serial.print(os_getTime());
    Serial.print(": ");
    switch (ev) {
        case EV_SCAN_TIMEOUT:
            Serial.println(F("EV_SCAN_TIMEOUT"));
            break;
        case EV_BEACON_FOUND:
            Serial.println(F("EV_BEACON_FOUND"));
            break;
        case EV_BEACON_MISSED:
            Serial.println(F("EV_BEACON_MISSED"));
            break;
        case EV_BEACON_TRACKED:
            Serial.println(F("EV_BEACON_TRACKED"));
            break;
        case EV_JOINING:
            Serial.println(F("EV_JOINING"));
            break;
        case EV_JOINED:
            Serial.println(F("EV_JOINED"));
            do_send(&sendjob);
            break;
        case EV_RFU1:
            Serial.println(F("EV_RFU1"));
            break;
        case EV_JOIN_FAILED:
            Serial.println(F("EV_JOIN_FAILED"));
            break;
        case EV_REJOIN_FAILED:
            Serial.println(F("EV_REJOIN_FAILED"));
            // Re-init
            os_setCallback(&initjob, initfunc);
            break;

        case EV_TXCOMPLETE:
            Serial.println(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            if (LMIC.dataLen) {
                // data received in rx slot after tx
                Serial.print(F("Data Received: "));
                Serial.write(LMIC.frame + LMIC.dataBeg, LMIC.dataLen);
                Serial.println();
            }
            // Schedule next transmission
            os_setTimedCallback(&sendjob, os_getTime() + sec2osticks(TX_INTERVAL), do_send);
            break;

        case EV_LOST_TSYNC:

```

```

        Serial.println(F("EV_LOST_TSYNC"));
        break;
    case EV_RESET:
        Serial.println(F("EV_RESET"));
        break;
    case EV_RXCOMPLETE:
        // data received in ping slot
        Serial.println(F("EV_RXCOMPLETE"));
        break;
    case EV_LINK_DEAD:
        Serial.println(F("EV_LINK_DEAD"));
        break;
    case EV_LINK_ALIVE:
        Serial.println(F("EV_LINK_ALIVE"));
        break;
    default:
        Serial.println(F("Unknown event"));
        break;
    }
}

//read the SDS011 data
void getSdsData() {
    while (1) {
        int i;
        unsigned char checksum = 0;

        if (mySerial.available() > 0) {
            do {
                incomingByte = mySerial.read();
            } while (incomingByte != 0xAA);
            delay(500); //this number is critical, all 9 following bytes must have arrived
            for (i=0; i<9; i++) {
                buf[i] = mySerial.read();
            }
            if ((buf[0] == 0xC0) && (buf[8] == 0xAB)) {
                for (i=1; i<=6; i++) {
                    checksum = checksum + buf[i];
                }
                if (checksum == buf[7]) {
                    PM2_5val = ((buf[2] << 8) + buf[1]) / 10.0;
                    PM10val = ((buf[4] << 8) + buf[3]) / 10.0;
                    Serial.print(" PM2.5: ");
                    Serial.print(PM2_5val);
                    Serial.print(" ug/m3 ");
                    Serial.print("PM10: ");
                    Serial.print(PM10val);
                    Serial.println(" ug/m3");
                    return;
                }
            }
        }
    }
}

void do_send(osjob_t* j) {
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        Serial.println(F("OP_TXRXPEND, not sending"));
    } else {
        getSdsData();
        // Prepare upstream data transmission at the next possible time.
        // Cayenne
        lpp.reset();
        lpp.addLuminosity(1, PM2_5val);
        LMIC.txChnl = 0;
        LMIC_setTxData2(1, lpp.getBuffer(), lpp.getSize(), 0);
        //end Cayenne
        Serial.println(F("Packet queued"));
    }
}

// initial job
static void initfunc (osjob_t* j) {
    // reset MAC state
    LMIC_reset();
}

```

```

    // LMIC_setClockError(MAX_CLOCK_ERROR * 1 / 100);

    // start joining
    LMIC_startJoining();
    // init done - onEvent() callback will be invoked...
}

void setup() {
    mySerial.begin(9600);

    Serial.begin(115200);
    Serial.println(F("Starting"));
    // LMIC init
    os_init();

    // Reset the MAC state. Session and pending data transfers will be discarded.
    os_setCallback(&initjob, initfunc);

    // Set up the channels used by the Things Network, which corresponds
    // to the defaults of most gateways. Without this, only three base
    // channels from the LoRaWAN specification are used, which certainly
    // works, so it is good for debugging, but can overload those
    // frequencies, so be sure to configure the full frequency range of
    // your network here (unless your network autoconfigures them).
    // Setting up channels should happen after LMIC_setSession, as that
    // configures the minimal channel set.

    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI); // g-band
    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK, DR_FSK), BAND_MILLI); // g2-band
    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
    // devices' ping slots. LMIC does not have an easy way to define set this
    // frequency and support for class B is spotty and untested, so this
    // frequency is not configured here.

    // Disable link check validation
    LMIC_setLinkCheckMode(0);

    // required for downlink
    LMIC.dn2Dr = SF9;

    // Set data rate and transmit power (note: txpow seems to be ignored by the library)
    LMIC_setDrTxpow(DR_SF7, 14);

    os_runloop();
    // Never get here
}

void loop() {
    // Never get here
}

```

Links und Literatur

/1/ Dragino Technology Lora Shield:

<http://www.dragino.com/products/module/item/102-lora-shield.html>

http://wiki.dragino.com/index.php?title=Lora_Shield

/2/ Nova Fitness Produktseite zum SDS011

<http://inovafitness.com/en/Laser-PM2-5-Sensor-SDS011-35.html>

/3/ Ein Umwelt-Radioaktivitäts-Messknoten für das LoRaWAN IOT-Netzwerk „The Things Network“

<http://www.opengeiger.de/LoRaGeigerTTN.pdf>

/4/ Ein einfaches Mapping Gerät für das LoRaWAN TTN-Netzwerk auf Basis eines Arduino MKR WAN 1300

<http://www.opengeiger.de/LoRaWAN/MkrWan1300Mapper.pdf>

/5/ Kompensation des Feuchte-Effekts bei Low-Cost Feinstaubsensoren-Sensoren nach dem Streulichtverfahren

<http://opengeiger.de/Feinstaub/FeuchteKompensation.pdf>

/6/ Suitability of the Low-Cost SDS011 Particle Sensor for Urban PM-Monitoring

Matthias Budde, Thomas Müller, Bernd Laquai, Norbert Streibl, Almuth D. Schwarz, Gregor Schindler, Till Riedel, Michael Beigl, Achim Dittler

3rd International Conference on Atmospheric Dust, Bari, Italy, 29.-31. Mai 2018