Technische Hochschule Mittelhessen

Fachbereich Informationstechnik - Elektrotechnik - Mechatronik

LoRa based IoT endpoint and gateway

Bachelorarbeit

von

Silvere Sacker Ngoufack

Betreuer: John Madieu

Erster Prüfer: Prof. Dr.-Ing. Hartmut Weber

Zweiter Prüfer: Prof. Dr.-Ing. Martin Gräfe

Friedberg, den 24. Juli 2020

Danksagung

Ich möchte mich an dieser Stelle bei den Jenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Zuerst bedanke ich mich herzlich bei John Martial Madieu, meinem Chef. für seine hilfreichen Anregungen und seine konstruktiv Kritik sowie seine interessanten Tipps bei der Entwicklung dieser Arbeit.

Herrn Prof. Dr. Ing. Hartmut Weber und Herrn Prof. Dr. Ing. Martin Gräfe danke ich für die Übernahme und die Betreuung sowohl für diese Arbeit als auch für mein Bachelorstudium.

Ich bedanke mich bei meinen Eltern, Raphaël und Elvire Kenfack dafür, dass sie mich immer unterstützt haben und meine bisherige Ausbildung, welche nach dieser Bachelorarbeit in Form eines Masterstudiums weitergeht, ermöglicht haben.

Ein besonderer Dank gilt sowohl meiner Freundin, meinen Freunden, die durch ihre Fragen und Anmerkungen mein Wissen erweitert haben, als auch Justin Neumann für das Korrekturlesen dieses Dokuments.

Eidesstattliche Erklärung

Hiermit versichere ich, Silvere Sacker Ngoufack, die vorliegende Arbeit selbstständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Inhalte dieser Arbeit, die wörtlich oder sinngemäß aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit oder Teile daraus wurden in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegt und auch nicht veröffentlicht.

Fiedberg, den 03. August 2020

SILVERE SACKER NGOUFACK

Abstract

„Die Ziele des IoT sind die IT-Vernetzung von Gegenständen und die Bereitstellung von funktionalitäten beziehungsweise Dienstleistungen, wie es so noch nie gab.“ Das Internet der Dinge ist eine zentrale Bedeutung für das Privatleben und die Wirtschaft geworden, da ein riesiger Anteil an vernetzten Objekten miteinander kommuniziert. Es entstehen neue Einsatzgebiete und Anwendungen mit dem Einsatz hochmoderner Technologien.

Durch den sofortigen Zugriff auf Informationen über die Umwelt und die Objekte erhöhen sich Effizienz und Produktivität, wodurch sich große Chancen für die Wirtschaft und das Privatleben eröffnen. Der Einsatz des IoT ermöglicht es, umfangreiche Echtzeitinformationen aus der Umwelt oder vom beweglichen und bewegungslose Objekte zu berücksichtigen. Die Vorteile sind unter anderen, die zeitliche Verfolgung von Gegenstände.

Diese Arbeit beschäftig sich mit der Entwicklung eines IoT-Endgeräts, das auf der LoRa-Technologie und einem STM32L4-Mikrocontroller basiert ist. Dieses IoT-Endgerät erfasst sowohl Umwelt-Daten, wie Temperatur und Feuchtigkeit, als auch die Beschleunigung. Diese Daten werden über das LoRa-WAN-Protokol an ein Embedded-Linux basiertes Gateway gesendet, entweder lokal in diesem Gateway verarbeitet oder über das Internet an einen Anwendungsserver weitergeleitet.

Inhaltsverzeichnis

Danksagung i

Eidesstattliche Erklärung ii

Abstract iii

Abkürzungsverzeichnis vi

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Aufgabenstellung und Zielsetzung . . . 1
- 1.3 Gliederung der Arbeit . . . 4

2 Hardware Komponenten eines Endgeräts 5
- 2.1 STM32L4 Discovery Kit . . . 5
  - 2.1.1 HTS221 Temperatursensor- und Feuchtigkeitssensor . . . 6
  - 2.1.2 LSM6DSL 3D Gyroskope und 3D Beschleunigungssensor 11
- 2.2 LoRa Endgerät: i-nucleo-lrwan1 . . . 14
  - 2.2.1 LoRa und LoRaWAN-Protokol . . . 17
  - 2.2.2 Sicherung der Daten . . . 22
  - 2.2.3 Aktivierung des Endgeräts . . . 24
  - 2.2.4 AT Kommandos . . . 27

3 Gateway und LoRaWAN-Server 30
- 3.1 Gateway . . . 30
- 3.2 Einstellung des LoRaWAN-Servers . . . 32
- 3.3 MQTT Protokoll . . . 39

4 Software Implimmentierung 43
- 4.1 Entwicklungsumgebung . . . 43
  - 4.1.1 Eclipse . . . 43

4.1.2 Libopencm3 Bibliotek installieren 47
4.2 Sensoren Auslesen 48
4.3 AT-Kommandos senden 53
4.4 Downlinks Behandlung 58

5 Zusammenfassung **62**
5.1 Ausblick 62
5.2 Fazit 63

Abkürzungsverzeichnis

ABP Activation By Personalization 24
ADC Analog Digital Converter 6
ADR Adaptive Data Rate 34
AppSKey Application Session Key 23
BLE Bluetooth Low Energie 5
CSV Comma-Separated Values 62
dBm (Decibel miliwatt 38
dps Degrees per second 11
FSK Frequency Shifting Keying 17
GPS Global Positioning System 3
GSM Global System for Mobile Communications 1
HTTP Hypertext Transfer Protocol 39
I2C Inter-Integrated Circuit 4
IoT Internet of Things 1
ISR Interrupt Service Routine 59
Kbps kilobit per second 3
LPUART Low Power UART 15
LSB Less Significant Bit viii
MAC Media Access Control address 23
MIC Message Integrity Code 23
MQTT Message Queuing Telemetry Transport 39
NwkSKey Network Session Key 23
ODR Output Data Rate 12
OTAA Over-The-Air Activation 24
RH Relative Humidity viii
RSSI Received Signal Strength Indication 38
SF Speading Factor 18
SMD Surface-Mounted Device 11
SPI Serial Peripheral Interface 5

TCP Transmission Control Protocol 39
UART Universal Asynchronous Receiver Transmitter 4

Abbildungsverzeichnis

1.1 Allgemeine LoRaWAN Netzwerkarchitektur [2] 2
1.2 Labcsmart IoT-Netzwerk 3

2.1 B-L475E-IOT01A Discovery kit [15] 6
2.2 Humidity sensor analog-to-digital flow [13] 7
2.3 Linear interpolation to convert Less Significant Bit (LSB) to %Relative Humidity (RH) [13] 9
2.4 Linear interpolation to convert LSB to °C [13] 11
2.5 Flußdiagram zur Datenermitlung 13
2.6 I-Nucleo-LRWAN1 [17] 14
2.7 I-Nucleo-LRWAN1 Architektur [17] 16
2.8 LABCSMART LoRa End-Node physisches Aussehen (a) und Verbindung (b) 16
2.9 Reichweite anhängig vom SF (Geändert von [3]) 18
2.10 Vergleich zwischen LoRa und andere IoT Kommunikationstechnologien [2] 19
2.11 Klassen von LoRaWAN [2] 21
2.12 Klasse A 21
2.13 Klasse B 22
2.14 Klasse C 22
2.15 LoRaWAN-Nachricht Verschlüsselung [3] 23
2.16 Join-Request Verfahren (Verändert von [3]) 26

3.1 LABCSMART LoRaWAN Gateway 31
3.2 Einstellung des Gateways 33
3.3 Einstellung des Netzwerks 33
3.4 Einstellung des Profils 34
3.5 Einstellung der Gruppe 35
3.6 ABP Registrierung 36
3.7 Einstellung OTAA 37

3.8 Uplinks/Downlinks 3.8(a) und Frames 3.8(b) 38
3.9 MQTT-Protokoll 40
3.10 MQTT Connector vom Server 41

4.1 Build Targets einstellen 45
4.2 Build Targets fertig 45
4.3 Libopencm3 Doxygen 48
4.4 I2C Kommunikation 50
4.5 Verbindungstest mit Saleae 58

Tabellenverzeichnis

2.1 Kalibrierregister für relative Feuchtigkeit 8
2.2 Kalibrierregister zur Temperaturermitllung 10
2.3 Rechweite abhängig der Umgebung 19

3.1 Downlink Nachrichten 42

1 Einleitung

Dieses Kapitel soll den Leser auf den Inhalt der Arbeit aufmerksam machen, ihn mit der Aufgabestellung vertraut machen und über die Strukturierung und Zielsetzung der Arbeit Auskunft geben.

1.1 Motivation

Objekte werden in der heutigen Zeit immer mehr mit Elektronik und Intelligenz versehen. Die Leute wollen aufgrund dieser Entwicklung, dass Prozesse oder bestimmte Aufgaben ohne menschliches Eingreifen erledigt und miteinander vernetzt werden. Das System soll lediglich überwacht und die Ergebnisse zu bestimmten Zwecken benutzt werden.

Das Internet der Dinge (Internet of Things (IoT)) wird dazu genutzt, um die Interaktion zwischen Menschen und vernetzten elektronischen Geräten zu vereinfachen.

1.2 Aufgabenstellung und Zielsetzung

Man möchte Daten wie, den Energieverbrauch eines Hauses, die Bewegung eines Objekts oder die Temperatur eines Raums kennen und über lange Strecken (20 km) übertagen ohne hohe Kosten mit geringem Energieverbrauch. Es gibt heutzutage Technologien, wie Global System for Mobile Communications (GSM), Bluetooth oder Wi-Fi, die diese Arbeit erledigen können. Das Problem dabei ist, dass beim Nutzen von GSM hohe Lizenzkosten fallen können. Was Bluetooth und Wifi betrifft, ist ihre Reichweite sehr begrenzt. Dieses Ziel kann mithilfe der LoRa-Technologie erreicht werden, da sie diese Nachteile beseitigt.

In dieser Abschlussarbeit soll ein Prototyp gebaut werden, der mithilfe der

LoRa-Technologie Daten an einen Anwendungsserver sendet und von diesem Server Daten empfängt. Anders gesagt, diese Bachelorarbeit beschäftigt sich mit der Entwicklung eines vernetzten Systems bestehend aus einem 3D-Beschleunigungssensor, einem 3D-Gyroskop sowie einem Temperatur- und Feuchtigkeitssensor. Die Sensoren messen Daten und übergeben diese an den STM32L475 Mikrocontroller.

Der Mikrocontroller soll die Daten verarbeiten und mithilfe eines LoRa-Moduls [18] drahtlos an einen Server übertragen. Bevor die Übertragung erfolgt, muss das LoRa-Endgerät Zugang zu dem Netzwerk durch den Server bekommen. Nachdem das LoRa-Endgerät dem Netzwerk hinzugefügt wurde, können nun Informationen zwischen dem LoRa-Endgerät und dem Netzwerk-Server bis zu einem Anwendungsserver ausgetauscht werden. Abbildungen 1.1 und 1.2 geben einen Überblick über den Aufbau des gesamten Systems.

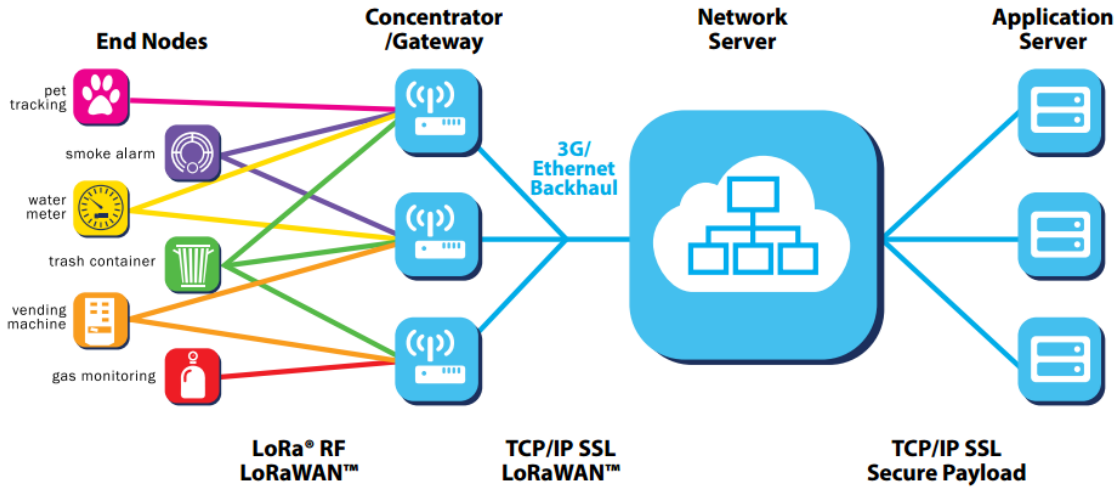


Abbildung 1.1: Allgemeine LoRaWAN Netzwerkarchitektur [2]

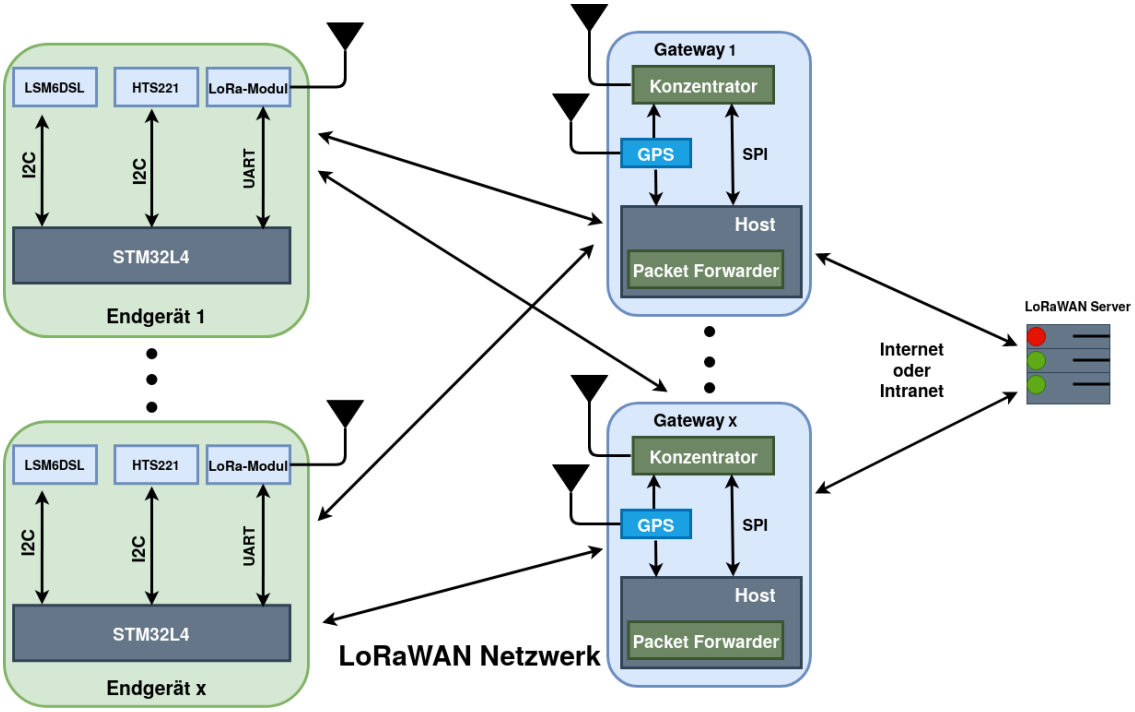


Abbildung 1.2: Labcsmart IoT-Netzwerk

LoRa: ist eine Abkürzung für ***Long Range*** und ist eine drahtlose Technologie, welche geringe Sendeleistung verbraucht, um kleine Datenpakete (0,3 kilobit per second (Kbps) bis 5,5 Kbps) über eine lange Strecke zu senden oder zu empfangen.

End Node Endgerät: ist ein Gerät, das aus zwei Teilen besteht, ein Funkmodul mit Antenne und ein Mikrocontroller zur Verarbeitung der Daten wie Sensordaten. Diese Daten können entweder an ein anderen LoRa-Endgerät per Point-To-Point-Verbindung oder an einen LoRaWAN-Server versandt werden.

LoRaWAN: steht für ***Long Range Wide Area Network*** und ist das Kommunikationsprotokoll für das Netzwerk.

Gateway: ist ein Gerät, das aus mindestens einem Funkkonzentrator, einem Host und einer Netzverbindung zum Internet oder ein privates Netzwerk (Ethernet, 3G, Wi-Fi), möglicherweise einem Global Positioning System (GPS)-Empfänger besteht.

LoRaWAN Server: ist ein abstrakter Computer, der die von dem Gateway empfangene RF-Pakete verarbeitet und übersendet die RF-Pakete als Antwort an das Gateway zurück.

Application Server: ist eine Anwendung, womit der Benutzer die von den Sensoren gemessenen Daten entweder tabellarisch oder grafisch auswerten kann.

Uplink: ist die Kommunikation von einem Endgerät zu einem Gateway.

Downlink: ist die Kommunikation von einem Gateway zu einem Endgerät.

1.3 Gliederung der Arbeit

Das Kapitel 2 gibt einen detaillierten Überblick über allen Hardware-Komponenten, die bei der Entwicklung eines Endgerät verwendet werden. Als Erstes wird auf die Eigenschaften von dem benutzten STM32-Nucleo Board eingegangen. Diesem Kapitel ist auch zu entnehmen, warum genau dieses Board gewählt wurde.

Als nächstes wird auf das LoRa-Modul eingegangen. Dieses LoRa-Modul wird dazu verwendet, um die erfassten Daten dem Server drahtlos zu übertragen. Dieses Kapitel berichtet über das Funkprotokol, das zur Übertragung der Daten eingesetzt wurde und wie diese Daten gesichert werden.

Das Kapitel 3 beschreibt den LoRaWAN-Server und das Gateway, zwei wichtige Teile dieser Thesis. Die Funktionsweise wird erklärt und die Servereinstellung wird gezeigt. Diese Einstellung sind notwendig, da diese dem endgerät den Zutritt in das Netzwerk gewähren.

Als nächstes wird die Softwareentwicklung behandelt. Hier geht es zu Beginn um die Entwicklungsumgebung des gesamten Projekts (Eclipse).Anschließend daran werden die angewandten Bibliotheken dargestellt, ihre Installation und Nutzung erklärt. Es wurde für diese Arbeit zwei bekannte Kommunikationsschnittstellen verwendet (Inter-Integrated Circuit (I2C) und Universal Asynchronous Receiver Transmitter (UART)). Sie erfahren ebenfalls wie diese Schnittstellen mit der Programmiersprache-C angesteuert werden und welche Software-Tricks eingesetzt wurden, um AT-Befehle zu senden.

Anschließend wird im Kapitel 5 eine Zusammenfassung und eine kleine Ausblick der Arbeit gegeben.

2 Hardware Komponenten eines Endgeräts

In diesem Kapitel werden die für das Gesamtsystem benutzten Hardware Komponenten hinsichtlich ihrer Funktionsweise und Ansteuerung in Einzelnen erläutert.

2.1 STM32L4 Discovery Kit

Das STM32L4 Discovery Kit ist ein IoT Knoten, womit ein Benutzer Anwendungen mit direkter Verbindung zu einem oder mehreren Cloud-Servern entwickeln kann. Dieses Discovery Kit ermöglicht eine Vielzahl von Anwendungen, indem es eine Multilink-Kommunikation (Bluetooth Low Energie (BLE)) mit geringem Stromverbrauch, Multiway-Erkennung der Umwelt liefert (Siehe Abbildung 2.1).

Das STM32L4 hat einen eingebetteten ST-LINK Debugger/Programmierer, eingebettete Sensoren und viele andere Features, die in dem Datenblatt erläutert zu finden sind. Auf Grung der Vielfalt an Eingenschaften wurde dieses Board ausgewählt. Man braucht kein Breadboard im Vergleich zu dem Arduino oder dem Raspberry-Pi, um Sensoren durch Schnittstellen (UART, Serial Peripheral Interface (SPI) oder I2C) mit dem Mikrocontroller zu verbinden. Noch dazu eignet sich dieses Discovery Kit für das LoRa-Modul von STMicroelectronics, da Arduino-Verbinder vorhanden sind. Dazu ist lediglich das LoRa-Modul in diesen Verbindern zu stecken.

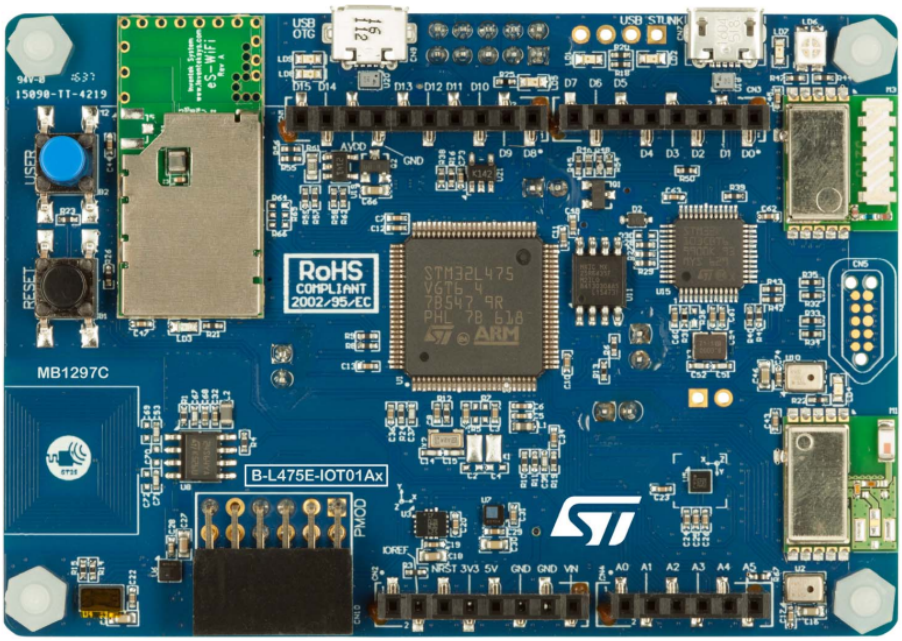


Abbildung 2.1: B-L475E-IOT01A Discovery kit [15]

Für diese Bachelorarbeit beschränken wir uns auf zwei Sensoren. Zum einen den HTS221[13] Temperatur- und Feuchtigkeitssensor zum anderen LSM6DSK 3D-Gyroscope und 3D-Beschleunigungssensor [14]. Daten werden erfasst und drahtlos an den LoRaWAN-Server übertragen. Die folgenden Unterkapitel beschreiben, wie diese Sensoren funktionieren und erklären, wie sie anzusteuern sind, damit die erhaltenen Daten im Rahmen der geforderten Toleranzen der Realität entsprechen. Laut dem Datenblatt ist mit einer Temperaturgenauigkeit von ±0.5°C und einer Feuchtigkeitsgenauigkeit von ±3.5% zu rechnen.

2.1.1 HTS221 Temperatursensor- und Feuchtigkeitssensor

In diesem Unterkapitel wird der HTS221 Temperatur- und Feuchtigkeitssensor beschrieben und erklärt wie die Temperatur alsauch die Feuchtigkeit zu ermitteln sind.

Der HTS221 Sensor misst die relative Feuchtigkeit (H) und die Temperatur (T) und speichert die Daten (16-Bits von Datentyp Integer) als Zweierkomplement. Diese Daten können über I2C- oder SPI-Schnittstelle ausgelesen werden. Die gespeicherten Daten sind Rohdaten, die am Ausgang von dem Analog Digital Converter (ADC) zur Verfügung gestellt werden (Siehe Abbildung 2.2). Um die Temperatur in °C und die relative Feuchtigkeit in % zu erhalten, müssen die

Daten aus den Registern ausgelesen und mit Hilfe der Formel 2.1.1 und 2.1.1 die richtigen Werten ermittelt werden.

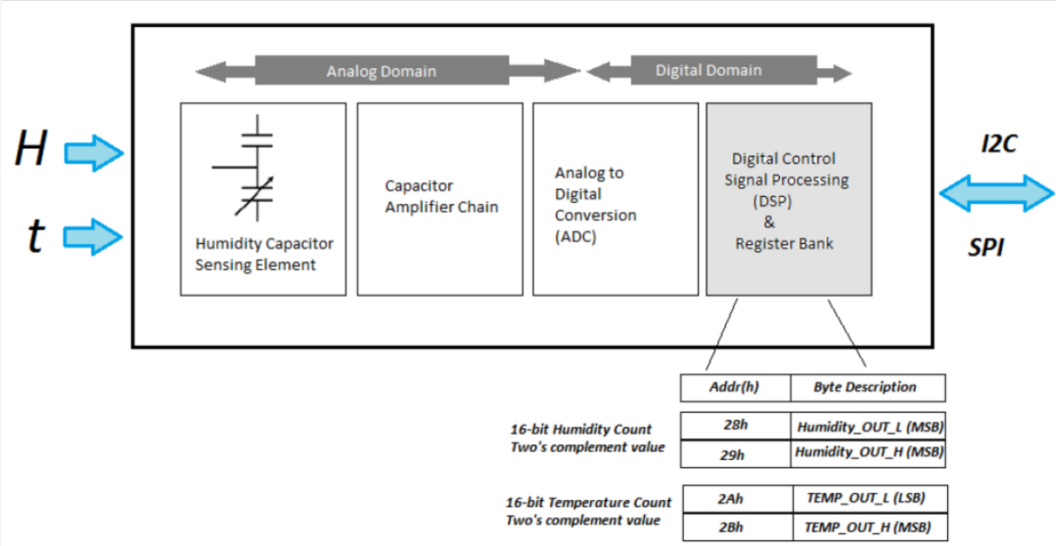


Abbildung 2.2: Humidity sensor analog-to-digital flow [13]

Feuchtigkeit ermitteln

An dieser Stelle wird erklärt wie die Feuchtigkeit von dem Sensor ermittelt wird. Der HTS221 Sensor speichert den Feuchtigkeitswert in Rohzählungen in zwei 8-Bit-Registern:

- *H_OUT_H* (0x29) (Höchstwertiges Byte)
- *H_OUT_L* (0x28) (Niedrigswertiges Byte)

Die zwei Bytes werden verkettet, um ein Zweierkomplement dargestelltes 16-Bit Wort zu bilden. Der relative Feuchtigkeitswert muss durch lineare Interpolation der Register (*HUMIDITY_OUT_H* & *HUMIDITY_OUT_L*) mit den Kalibrierregistern berechnet werden.

Der HTS221 Sensor ist bei der Herstellung bereits kalibriert und die erforderlichen Koeffizienten sind ADC 16-Bit-Werte, die in den Registern des Sensors zu lesen sind. Eine weitere Kalibrierung durch den Benutzer ist nicht erforderlich.

Die Tabelle 2.1 stellt die Register dar, in denen die Kalibrierwerten zur Ermittlung der relativen Feuchtigkeit gespeichert sind.

Variable	Adresse	Format[1]
H0_rH_x2	0x30	u(8)
H1_rH_x2	0x31	u(8)
H0_TO_OUT_H	0x36	s(16)
H0_TO_OUT_L	0x37	s(16)
H1_TO_OUT_H	0x3A	s(16)
H1_TO_OUT_L	0x3B	s(16)

Tabelle 2.1: Kalibrierregister für relative Feuchtigkeit

Nun wissen wir welche Register zu lesen sind, damit die relative Feuchtigkeit mithilfe der Interpolation berechnet werden kann. Die folgenden Schritten müssen vor der Berechnung durchgeführt werden:

- Werte von *H0_rH_x2* und *H1_rH_x2* aus Registern 0x30 und 0x31 auslesen
- *H0_rH_x2* und *H1_rH_x2* durch zwei teilen
- Werte von *H0_TO_OUT* aus Registern 0x36 und 0x37 auslesen
- Werte von *H1_TO_OUT* aus Registern 0x3A und 0x3B auslesen
- Rohdaten von *H_T_OUT* aus Registern 0x28 und 0x29 auslesen

Nachdem diese Register gelesen wurden, kann nun die Berechnung der relativen Feuchtigkeit erfolgen.

Aus Abbildung 2.3 resultiert nach linearer Interpolation folgende Formel [13]:

$$RH\% = \frac{((H1\_rH \quad H0\_rH):(H\_T\_OUT \quad H0\_T0\_OUT))}{(H1\_T0\_OUT \quad H1\_T0\_OUT)} + H0\_rH$$

[1](u8) 16Bit-Wert ohne Vorzeichen, (s16) 16Bit-Wert mit Vorzeichen

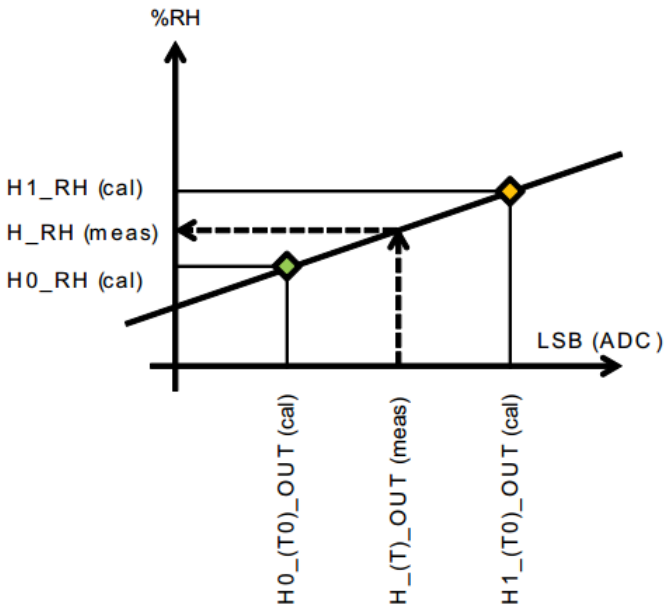


Abbildung 2.3: Linear interpolation to convert LSB to %RH [13]

Temperatur ermitteln

Der HTS221 Sensor speichert den Temperaturwert in Rohzählungen in zwei 8-Bit-Registern:

- *T_OUT_H* (0x2A) (Höchstwertiges Byte)
- *T_OUT_L* (0x2B) (Niedrigswertiges Byte)

Die zwei Bytes werden verkettet, um ein Zweierkomplement dargestelltes 16-Bit Wort zu bilden. Die Polarität wird duch das höchstwertigste Bit vom *T_OUT_H* Register bekannt gegeben.

- Ist dieses Bit 0, ist die gelesene Temperatur positiv.
- Ist dieses Bit 1, ist die gelesene Temperatur negativ. In diesem Fall ist das Zweierkomplement des gesamten Wort zu bilden, um den richtigen Wert zu erhalten.

Auch hier ist die Temperatur durch lineare Interpolation von den Kalibrierregistern und den Registern *T_OUT_H* und *T_OUT_H* in Zweierkomplement zu errechnen.

Die Tabelle 2.2 stellt diese Kalibrierregister dar.

Registern	Adresse	Format
T0_degC_x8	0x32	u(8)
T1_degC_x8	0x33	u(8)
T1/TOmsb	0x35	(u2),(u2)
T0_OUT_H	0x3D	s(16)
T0_OUT_L	0x3C	s(16)
T1_OUT_H	0x3F	s(16)
T1_OUT_L	0x3E	s(16)

Tabelle 2.2: Kalibrierregister zur Temperaturermitllung

Da die Kalibrierregister vom Hersteller mit den korrekten Werten versehen werden, werden wir nun diese Register auslesen und mithilfe der gelesenen Werten die Temperatur ermitteln. Bevor die Temperatur mit linearer Interpolation berechnet werden kann, sind folgende Schritten erstmal erforderlich.

- Die Koeffizienten *T0_degC_x8* und *T1_degC_x8* aus den Registern 0x32 und 0x33 auslesen
- Die Werte von *T0_degC_x8* und *T1_degC_x8* durch 8 dividieren, um die Koeffizienten *T0_degC* und *T1_degC* zu erhalten.
- Die höchstwertigste Bits von T1*_degC*(T1.9 und T1.8) und *T0_degC*(T0.9 und T0.8) aus dem Register 0x35 auslesen. Diese Werte mit den im Schritt 2 ermittelten Werten verketten, damit *T0_degC* und *T1_degC* vollständig werden.
- Der Wert von *T0_OUT* aus den Registern 0x3C und 0x3D auslesen.
- Der Wert von *T1_OUT* aus den Registern 0x3E und 0x3F auslesen.
- Der Wert von *T_OUT* aus den Registern 0x2A und 0x2B auslesen.

Nachdem diese Kallibrierregister ausgelesen wurden, kann mittels linearer Interpolation die Temperatur in °C berechnet werden.

Abbildung 2.4 zeigt den Graph, aus dem die lineare Interpolation stammt. Die folgende Formel wurde daraus hergeleitet.

$$T[c] = \frac{((T1\_degC \quad T0\_degC):(T\_OUT \quad T0\_OUT))}{(T1\_OUT \quad T0\_OUT)} + T0\_degC$$

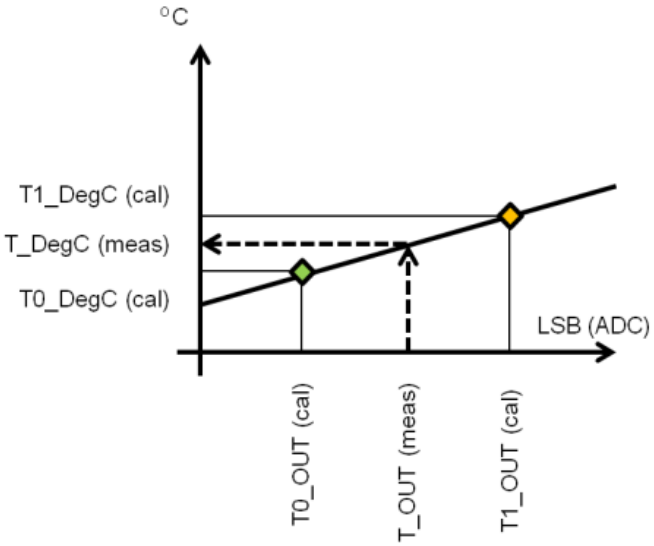


Abbildung 2.4: Linear interpolation to convert LSB to °C [13]

Da die Kalibrierwerte zur Berechnung der Temperatur und der relativen Feuchtigkeit bei der Herstellung des Bausteins vorab festgesetzt sind, soll man die Kalibrierregister bei der Programmierung nur ein mal auslesen. Dies erspart den Rechenaufwand des Mikrocontrollers.

2.1.2 LSM6DSL 3D Gyroskope und 3D Beschleunigungssensor

Dieses Unterkapitel berichtet von dem LSM6DSL 3D-Gyroskope und 3D-Beschleunigungssensor. Hier ist zu entnehmen, wie die X-,Y-, und Z-Koordinaten der Sensoren zu ermitteln sind und wie der Sensor, abhängig vom Zweck, skaliert werden kann.

Der LSM6DSL ist ein digitaler 3D-Beschleunigungsmesser und ein 3D-Gyroskopsystem mit einer digitalen seriellen I2C/SPI Schnittstelle mit einer Leistung von 0.65mA im kombinierten Hochleistungsmodus. Das Gerät verfügt über einen von Benutzer wählbaren dynamischen Beschleunigungsbereich von ±2 | ±4 | ±8 | ±16g (g is gleich 9,81m/s) und einen Winkelgeschwindigkeitsbereich von ±125 | ±250 | ±500 | ±1000 | ±2000Degrees per second (dps).

Das extrem geringe Größe und das geringe Gewicht des Surface-Mounted Device (SMD)-Packets machen den LSM6DSL zu einer idealen Wahl für tragbare

Anwendungen wie Smartphones, IoT-verbundene Geräte und andere Anwendungen, bei der reduzierte Paketgröße und -gewicht erforderlich sind.

Der LSM6DSL bietet drei mögliche Betriebskonfiguration:

- nur Beschleunigungsmesser aktiv und Gyroskope inaktiv
- nur Gyroskope aktiv und Beschleunigungsmesser inaktiv
- beide aktiv mit unabhängigem Output Data Rate (ODR)

Der Beschleunigungsmesser und das Gyroskop können unabhängig voneinander konfiguriert werden unter anderem: Power-down, Low-Power, Normal- und High-Performance Modus. Um den Stromverbrauch des Sensors zu reduzieren, kann das Gyroskop in eines Ruhestand versetzt werden.

Sobald das Gerät mit Strom versorgt wird, werden die Kalibrierkoeffizienten vom eingebettetem Flash-Speicher in den Registern geladen. Dieser Vorgang dauert ungefähr 15 ms. Nach dieser Zeit fallen der Beschleunigungsmesser und das Gyroskop in den Power-Down Modus. Durch der *CTRL1_XL* bzw. *CTRL2_G*-Register können die Geräte geweckt werden, indem man den Betriebsmodus auswählt.

Wenn die Daten verfügbar sind, wird eine Unterbrechung (Interrupt) ausgelöst, wenn das entsprechende Byte vom Beschleunigungsmesser bzw. vom Gyroskop in das *INT1_CTRL-* Register geschrieben wurde. Das Vorhandensein der Daten kann nun mithilfe des Statusregisters abgefragt werden. Das *XLDA*-Bit wird auf 1 gesetzt, wenn am Ausgang des Beschleunigungsmessers ein neuer Datensatz verfügbar ist. Das *GDS*-Bit wird auf 1 gesetzt, wenn am Gyroskopausgang ein neuer Datensatz verfügbar ist.

Die Abbildung 2.5 stellt das Flussdiagram zur Ermittlung der Achsen- und Winkelveränderungen des Beschleunigungsmessers und des Gyroskops dar.

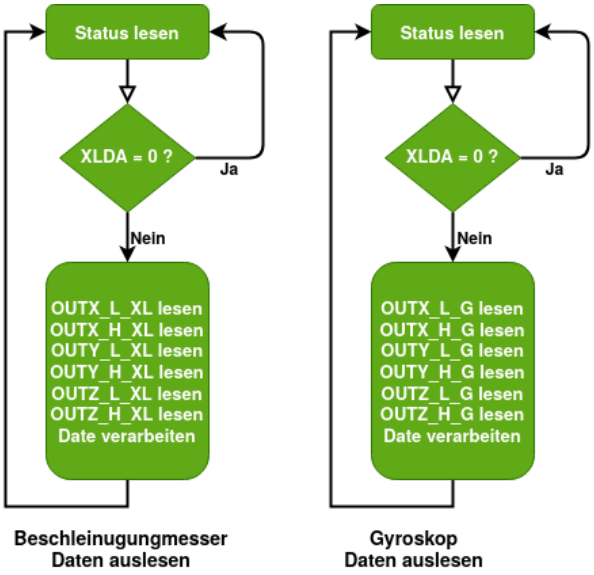


Abbildung 2.5: Flußdiagram zur Datenermitlung

Wie oben bereits erwähnt, kann das Gerät so eingestellt werden, dass ein neuer Satz von Messdaten durch ein Signal erkennbar wird. Das *XLDA*-Bit des *STATUS_REG*-Registers stellt das Signal des Vorhandenseins der Beschleunigungsdaten dar. Das Signal kann durch den *INT1*-Pin angesteuert werden, indem das *INT1_DRDY_XL*-Bit vom *INT1_CTRL*-Register auf 1 gesetzt wird.

Für den Gyroskopsensor wird das datenbereite Signal durch das *GDA*-Bit des *STATUS_REG*-Registers dargestellt. Das Signal kann durch den *INT1*-Pin angesteuert werden, indem das *INT1_DRDY_G*-Bit vom *INT1_CTRL*-Register auf 1 gestezt wird. Die gemessenen Beschleunigungsdaten werden an *OUTX_H_XL*-, *OUTX_L_XL*-, *OU-TY_H_XL*-, *OUTY_L_-XL*-, *OUTZ_H_XL*-, *OUTZ_L_XL*-Register gesendet. Die gemessenen Winkelgeschwindigkeitsdaten werden dagegen an *OUTX_H_G*-, *OUTX_L_G*-, *OUTY_H_G*-, *OUTY_L_G*-, *OUTZ_H_G*-, *OUTZ_L_G*-Register gesendet. Die vollständigen Ausgangsdaten für die X-,Y- und Z-Achsen sind durch die Verkettung von *OUTX_H_XL(G)* und *OUTX_L_-XL(G)*, *OUTY_H_XL(G)* und *OUTY_L_XL(G)*, *OUTZ_H_XL(G)* und *OUTZ_L_XL* zu erhalten, wobei die Beschleunigungsdaten und die Winkelgeschwindigkeitsdaten als 16-Bit Werte dargestellt werden.

Mit dem LSM6DSL kann der Inhalt des unteren und oberen Teils der Ausgangsdatenregister vertauscht werden, sodass die Darstellung entweder Big-Endian oder Little-Endian entspricht. Dies ist möglich, sofern man das BLE-Bit von dem CTRL3_C-Register auf 0 (Little-Endian standartmäßig) oder auf 1 (für Big-Endian). Big-Endian bedeutet, dass das höchstwertige Byte des Datensat-

zes in der niedrigsten Speicherstelle gespeichert wird. Little-Endian bedeutet, dass das niedrigswertige Byte des Datensatzes in der niedrigsten Speicherstelle gespeichert wird.

Im Unterkapitel 4.2 werden die Funktionen zur Datenermittlung in der Programmiersprache-C sowohl für den HTS221 (Temperatur- und Feuchtigkeitssensor) als auch für den LSM6DSL (3D-Beschleunigungssensor und 3D-Gyroskop) dargestellt und erklärt wie die Kommunikationsschnittstelle (hier I2C) zu benutzen ist.

2.2 LoRa Endgerät: i-nucleo-lrwan1

Die im Kapitel 2.1.1 ermittelten Sensordaten sollen laut der Aufgabestellung mit Hilfe eines drahtlosen Protokolls an einen Server gesendet werden. Um diese Daten drahtlos und über eine lange Strecke zu übertragen, haben wir uns für das LoRaWAN-Protokol entschieden. Die Gründe warum genau dieses Protokol ausgewählt wurde, werden in diesem Kapitel genannt. Noch dazu wird nicht nur auf die Eigenschaften des benutzten Endgeräts eingegangen sondern auch auf den Unterschied von diesem Modul gegenüber anderen Modulen, die auf dem Markt zu finden sind.

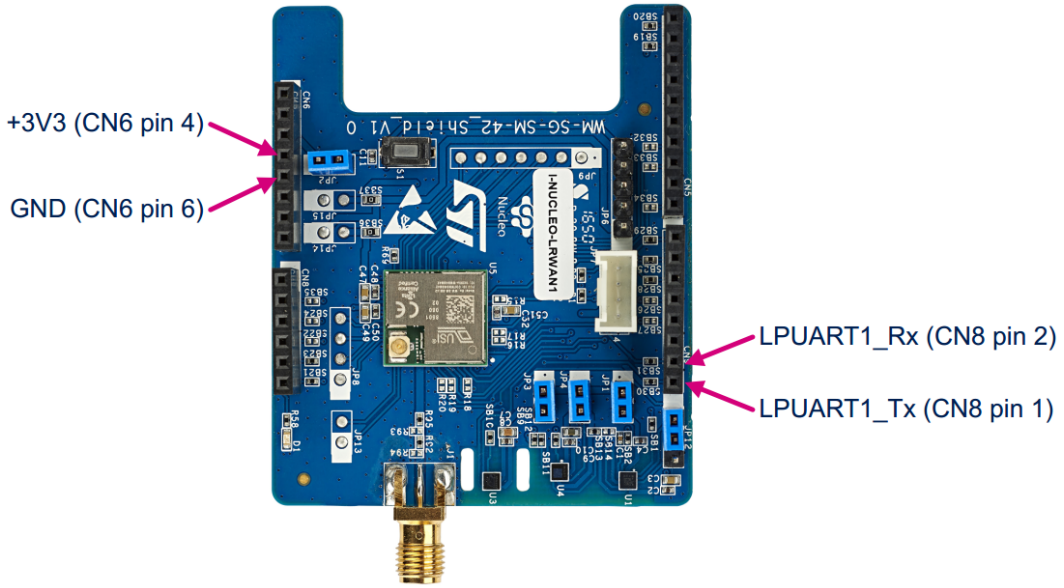


Abbildung 2.6: I-Nucleo-LRWAN1 [17]

Abbildung 2.6 zeigt das Endgerät, das zur Datenübertragung verwendet wird.

Diese Platine mit Arduino-Connectoren und mehr ist eine integrierte Lösung, die jedem ermöglicht Anwendungen mit der LoRa-Technologie zu entwickeln. Das I-Nucleo-LRWAN1 verfügt über das USI® LoRaWAN™ Technologiemodul für kostengünstiges und stromsparendes Weitverkehrsnetz (LPWAN), welches mit einem eingebetteten Stapel von AT-Befehle mitgeliefert wird. Dieses Board wurde ausgewählt, weil es durch ein externes Board, wie das Nucleo-L053 oder das B-L475E-IOT01A Discovery Kit 2.1 über mehrere Schnittstellen, wie Low Power UART (LPUART), SPI oder I2C angesteuert werden kann. Noch dazu verfügt das I-Nucleo-LRWAN1 über die folgenden eingebetteten Sensoren.

- ST Beschleunigungs- und Magnetosensor (LSM303AGR)
- ST Feuchtigkeits- und Temperatursensor (HTS221)
- ST Drucksensor (LPS22HB)

Im Vergleich zu anderen Endgeräten, worauf keine Sensoren vorhanden sind, müssen keine weiteren Sensoren erworben werden. Die Kommunikation mit einem anderen Mikrocontroller erfolgt einfach durch UART, man braucht nicht auf das integrierte Radio-Modul ansprechen, um Daten Befehle zu senden oder empfangen. Das Bild 2.7 zeigt, dass das I-Nucleo-LRWAN1 mit einem STM32L0-Mikrocontroller versehen ist, der dazu zuständig ist, die Kommunikation zwischen dem I-Nucleo-LRWAN1 und einem externen Mikrocontroller zu vereinfachen. Der SX1272-Chip ist das eigentliche LoRa-Radio-Modul, welcher die Daten oder die AT-Befehle per Funk durch die Antenne an entweder ein Gateway oder ein anders Endgerät sendet.

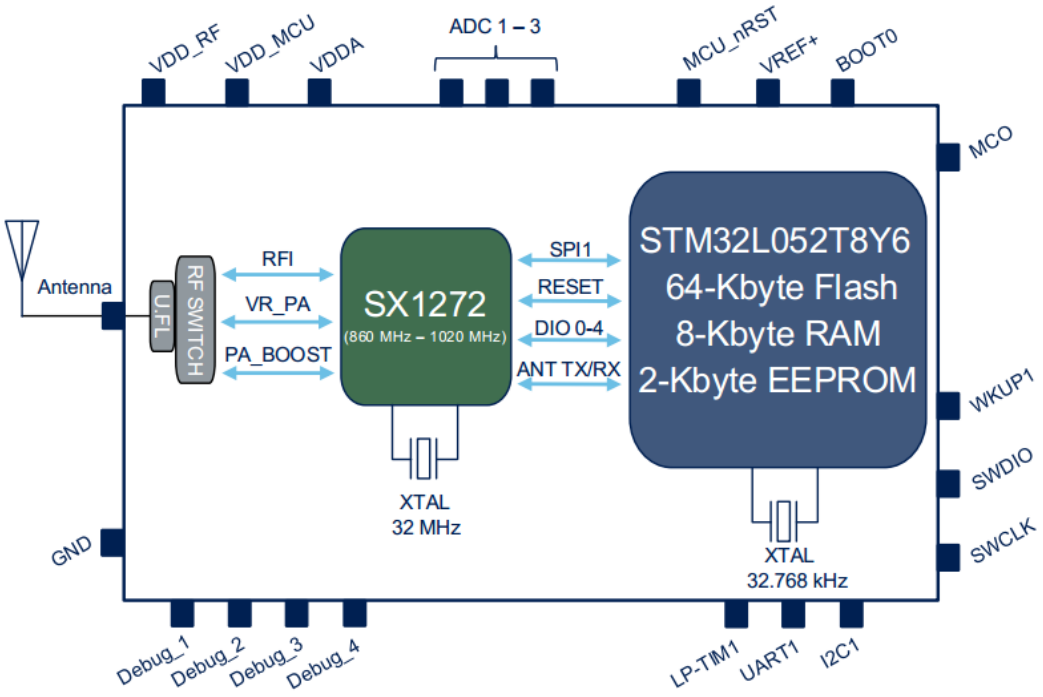
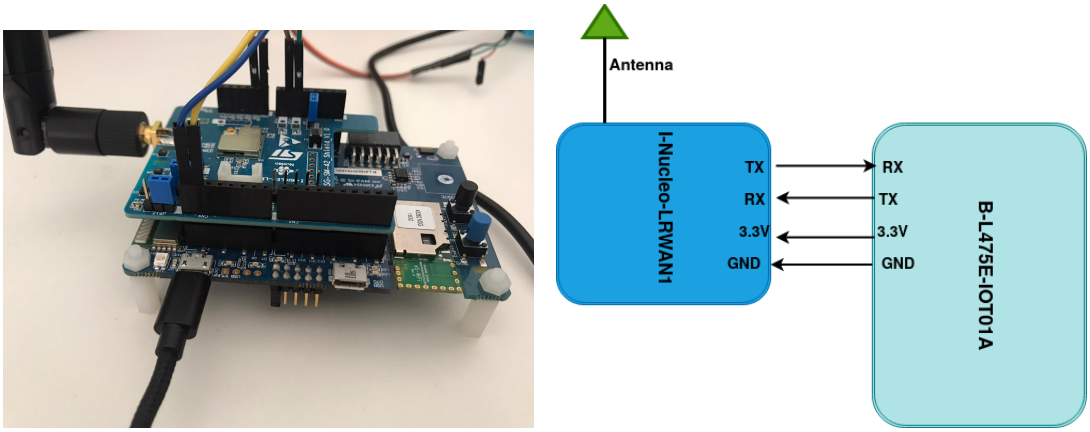


Abbildung 2.7: I-Nucleo-LRWAN1 Architektur [17]

Das I-Nucleo-LRWAN1 wird mithilfe seiner Arduino-Connectoren mit einem externen Board verbunden. Für diese Abschlussarbeit wird dieses Endgerät an den Arduino-Connectoren des B-L475E-IOT01A Discovery Kit verbunden (Siehe Abbildung 2.8)



(a) (b)

Abbildung 2.8: LABCSMART LoRa End-Node physisches Aussehen (a) und Verbindung (b)

Dem Bild 2.8(b) ist zu entnehmen, dass beide Komponenten durch eine UART-Schnittstelle kommunizieren. Das I-Nucleo-LRWAN1 wird von dem B-L475E-

IOT01A mit Strom versorgt. Die Aufgabe des B-L475E-IOT01A besteht darin, erstmal die Sensordaten zu verarbeiten, als nächsten sendet es durch die UART-Schnittstelle AT-Befehle zur Konfiguration des I-Nucleo-LRWAN1, sodass die erhaltenen Sensordaten mithilfe des LoRaWAN-Protokolls versendet werden können.

2.2.1 LoRa und LoRaWAN-Protokol

In diesem Teil der Thesis erfahren Sie sowohl, was LoRa und das LoRaWAN-Protokoll sind, als auch wie das Protokoll implementiert wird, damit ein Endgerät in das LoRaWAN-Netzwerk hinzugefügt werden kann.

LoRa: Die Physikalische Schicht

Eine einzige Technologie kann nicht alle Anwendungen des IoT decken. Technologien wie Wi-Fi und BLE sind weit verbreitete Standards und decken die Kommunikation persönlicher Geräte recht gut. Die Mobilfunktechnologie passt hervorragend zu Anwendungen, die einen hohen Datendurchsatz benötigen.

Diese Technologien sind zwar gut, aber weisen einige Nachteile auf, wie dem hohen Energieverbrauch und eine kleine Reichweite. LoRa bieten Lösungen zu diesen Nachteilen an, nämlich eine mehr järige Batterielebensdauer, ermöglicht die Übertragung von kleinen Datenmengen über große Entfernungen. LoRa ist die physikalische Schicht oder die verwendete drahtlose Modulation, um eine lange Bereichskommunikationsverbindung zu schaffen.

Viele ältere drahtlose Systeme verwenden die Frequenzumtastungen (Englisch ***Frequency Shifting Keying (FSK)***) als physikalische Schicht, weil es eine sehr effiziente Modulation zur Erzielung geringer Leistung ist. LoRa basiert auf die Chirp-Spreizspektrum-Modulation (Englisch ***Chirp Spread Spectrum Modulation***). Diese Modulation behält die gleiche Eigenschaft der geringen Leistung wie FSK-Modulation bei und erhöht deutlich die Kommunikationsreichweite. Das Chirp-Spreizspektrum wird seit Jahrzehnten aufgrund seiner Kommunikationsreichweite und seiner Robustheit gegenüber Störungen in der Militär-und Weltraumkommunikation eingesetzt. LoRa ist derzeit die erste kostengünstige Implementierung für den kommerziellen Einsatz.

Die LoRa-Technologie wurde von einem kleinen französischen Start-Up namens Cycleo entwickelt. In 2012 wurde Cycleo von der Firma Semtech gekauft.

Es existieren konkurrierende Technologien zu LoRa wie Narrowband IoT (ND-IoT) und Sigfox. Das LoRa kann keine Video- und Audio-Nachrichten übertragen, lediglich sehr kleine Datenpakete wie Sensordaten. Der Hauptpunkt von LoRa ist die Kommunikation über lange Strecken und die Verwendung einer sehr geringen Sendeleistung von ungefähr 20mW.

Die Reichweite einer LoRA-Kommunikation wird durch ihre Bandbreite, die Signalausgangsleistung sowie den verwendeten Spreisfaktor (Englisch ***Speading Factor (SF)***) bestimmt. Durch die Steuerung des Signals wird seine Reichweite zum Nachteil der Bitrate erhöht, da es über eine längere Strecke übertragen wird. Das verbraucht zu viel Energie und spielt Nachteilhaft zur Autonomie des Geräts. Das heißt, je größer der Spreisfaktor ist, umso kleiner wird die Bitrate und die Reichweite wird dagegen größer (Siehe Abbildung 2.9). Ein LoRaWAN-Netzwerk bietet sechs mögliche Spreizfaktoren (**SF7, SF8, SF9, SF10, SF11, SF12**).

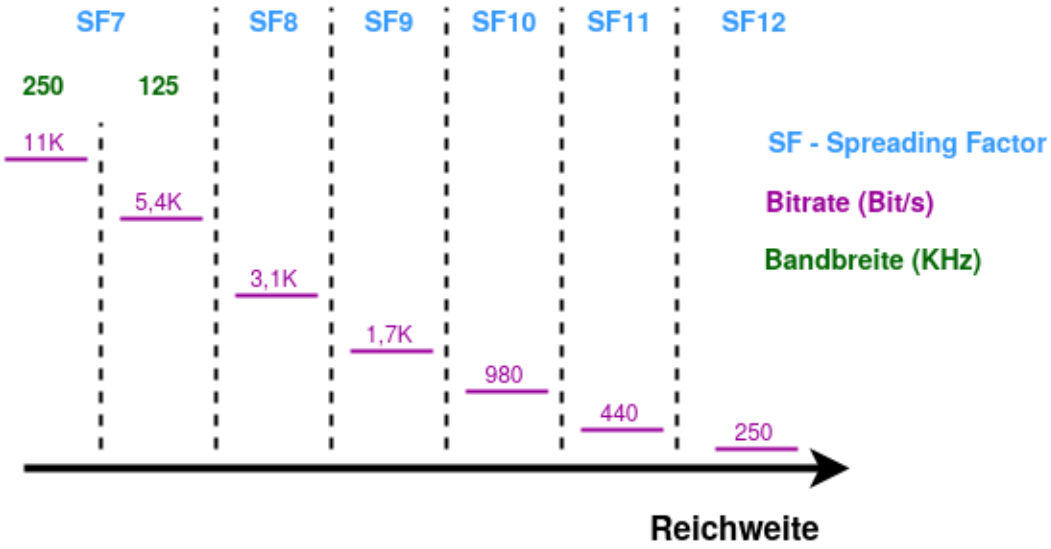


Abbildung 2.9: Reichweite anhängig vom SF (Geändert von [3])

Die Reichweite zwischen LoRa-Sender und -Empfänger hängt auch von der Umgebung ab, in der das Gerät betrieben wird. Die Abdeckung von Innenräumen hängt weitgehend von der Art des verwendeten Baumaterials ab. Die Tabelle 2.3 zeigt die Reichweite der LoRa-Technologie in Abhängigkeit zur Umgebung.

Umgebung	Reichweite in km
Städtische Gebiete	2 bis 5
andische Gebiete	5 bis 15
Direkte Sichtlinie	>15

Tabelle 2.3: Rechweite abhängig der Umgebung

Es gibt Wissenschaftler, die dazu gekommen sind ein Weltrekord zu stellen, indem sie eine LoRa-Verbindung bis auf 200 km geschafft haben. Ein Beispiel ist Herr Andreas Spiess [1].

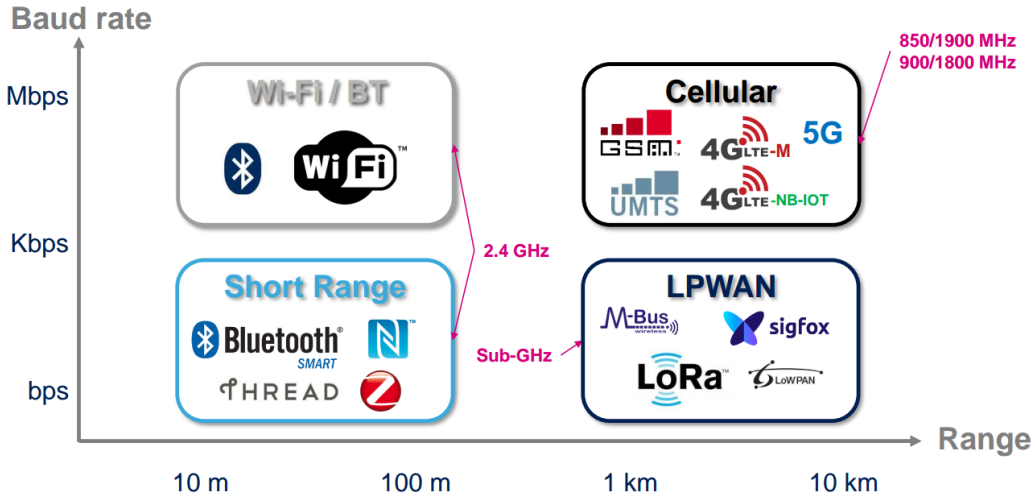


Abbildung 2.10: Vergleich zwischen LoRa und andere IoT Kommunikationstechnologien [2]

Abbildung 2.10 ist zu entnehmen, dass LoRa im Vergleich zu andere Technologien wie Wi-Fi oder 4G eine kleine Baudrate hat. Aber seine Reichweite ist deutlich größer als weit bekannte Technologien wie Bluetooth oder Wi-Fi.

Die LoRa-Technologie kann in vielen Gebiete einsetzt werden. Die folgende Auflistung gibt einen groben Überblick über einige Einsatzgebiete.

- **Intelligente Dienstprogramme**
  - Überwachung eines Leistungstransformators
  - Wasserstandsüberwachung
  - Kraftstoffüberwachung

- **Gesundheit und Hygiene**
  - Temperatur- und Feuchtigkeitsüberwachung
  - Umweltüberwachung
- **Sicherheit**
  - Radioaktivitätsüberwachung
  - Intelligenter Geschwindigkeitsblitzer
- **Landwirtschaft**
  - Überwachung des Tierschutzes
  - Überwachung der Pflanzenwachstumsbedingungen
- **Effizienz**
  - Asset Management (Tracking von Containern, Paletten)
  - Deichmanagement (Verfolgung von Autos, Lieferwagen, Lastwagen)

LoRaWAN: Das Kommunikationsprotokoll

LoRaWAN beschreibt das Kommunikationsprotokoll und die Systemarchitektur des Netzwerks, während LoRa die physikalische Schicht beschreibt, die die Fernkommunikationsverbindung ermöglicht. Das Protokoll und die Netzwerkarchitektur haben den großen Einfluss auf die Bestimmung der Batterielebensdauer, die Netzwerkkapazität, die Servicequalität, die Sicherheit eines Endgeräts und die Vielzahl der vom Netzwerk bereitgesellten Anwendungen.

Wie Abbildung 2.11 zu entnehmen ist, ist das LoRaWAN mit verschiedenen Schichten aufgebaut. Die erste Schicht ist die regionale Schicht, hier geht es um die Frequenzbereiche, die abhängig von der Region zur Datenübertragung verwendet werden können. Die ISM-Bandfrequenzen von Europa liegen zwischen 863 MHz und 870 MHz. Als nächstes kommt die LoRa-Modulation als physikalische Schicht des Netzwerks. LoRaWAN verfügt über viele Klassen, nämlich die Klasse A, B und C. Diese Klassen werden später in diesem Unterkapitel im Einzelnen erklärt. Am Ende kommt die Anwendungsebene der LoRa-Technologie.

Alle Endgeräte funktionieren nicht gleich, aufgrund der von dem Entwickler implementierte Klasse.

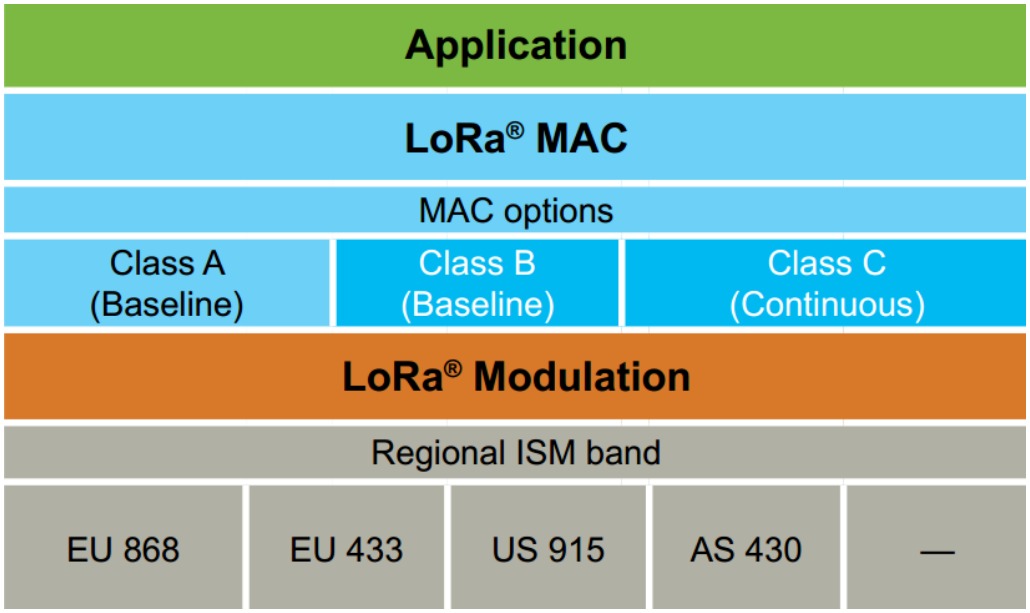


Abbildung 2.11: Klassen von LoRaWAN [2]

Klasse A (All end-devices): Ein Endgerät der Klasse A ermöglich eine bidirektionale Kommunikation, wobei nach jedem Uplink eines Endgeräts für kurze Zeit zwei kurze Downlink-Empfangsfenster folgen. Diese Empfangs Fenster werden jeweils für eine Zeit *RECEIVE_DELAY1*(für das erste Fenster) und *RECEIVE_DELAY2* geöffnet. Die Dauer dieser Zeiten werden sowohl in dem Endgerät, als auch auf dem Server gespeichert. Verglichen zu den anderen Klassen, verbrauchen Endgeräte der Klasse A am niedrigsten Leistung.

Nachdem die zwei Downlink-Empfangsfenster geschloßen sind, kann das Gateway keine weitere Downliks mehr senden. Die nächsten Downlinks werden erst berücksichtig, wenn das Endgerät ein Uplink gesendet hat. Das unten stehende Bild erläutert dieses Verhalten.

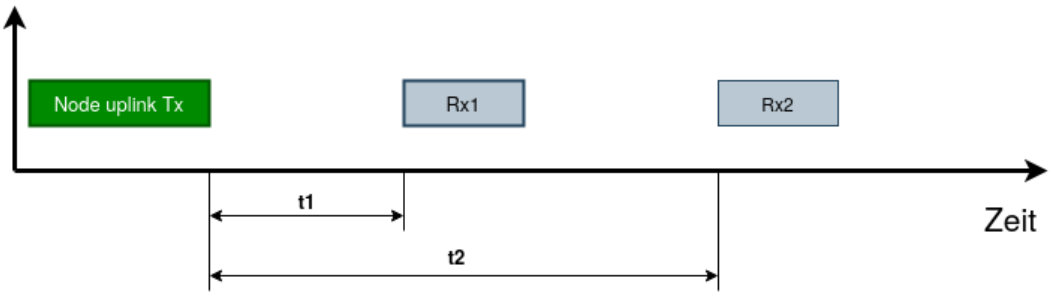


Abbildung 2.12: Klasse A

Klasse B (Beacon): Zusätzlich zu den zufälligen Empfängsfenster der Klasse A, Geräte der Klasse B öffnen zusätzliche Empfangsfenster zu geplannte Zeiten. Damit das Endgerät seine Empfangsfenster an den geplanten Zeiten öffnen kann, bekommt es ein synchronisiertes Beacon von dem Gateway. Dies ermöglicht dem Gateway zu wissen, wann das Endgerät auf Downlinks wartet. Diese Klasse verbraucht mehr Leistung im Vergleich zur Klasse A.

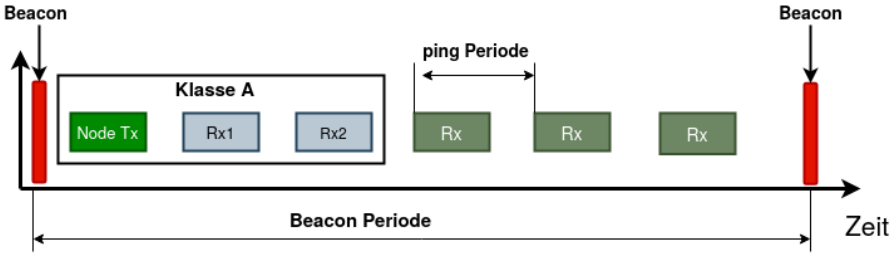


Abbildung 2.13: Klasse B

Klasse C (Continuously listening): Endgeräte der Klasse C haben fast immer geöffnete Empfangsfenster, die sich nur beim Senden schließen. Diese Klasse verbraucht am meisten Energie.

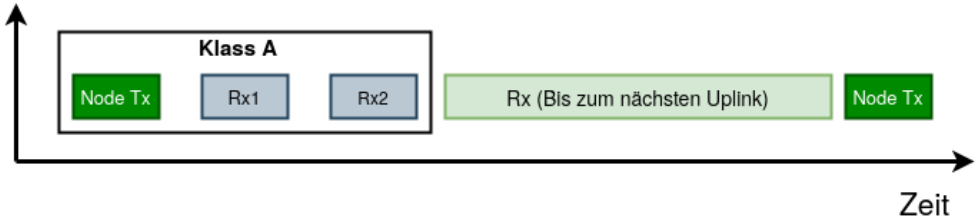


Abbildung 2.14: Klasse C

Im Rahmen dieser Thesis wird nur die Klasse A berücksichtig, weil das ausgewählte Endgerät die Klasse B nicht unterstützt und die Klasse C zu viel Energie verbraucht.

2.2.2 Sicherung der Daten

Wir wollen nicht, dass die gesendeten Informationen durch einen Dritten ohne Zugriffsrechte in dem Netzwerk gelesen werden können. Unabhängig davon, ob die Netzwerksicherheit oder die Vertraulichkeit und Sicherheit der Daten gewährleistet werden soll, ist das Thema Sicherheit äußerst wichtig. Eine Frage, die übrigens das Internet der Dinge als ganzes betrifft.

Um die Netzwerk- und Datensicherheit zu gewährleisten, verwendet das LoRaWAN-Netzwerk zwei AES-128-Verschlüsselung. Der erste ist der Netzwerksitzungsschlüssel (Englisch ***Network Session Key (NwkSKey)***) und stellt die Authentizität der Geräte im Netzwerk sicher. Der zweite ist der Anwendungssitzungsschlüssel (Englisch ***Application Session Key (AppSKey)***). Der NwkSkey wird von dem Endgerät und dem Server benutzt, um den Nachrichtenintegritätscode (Englisch ***Message Integrity Code (MIC)***) zu berechnen und die Intergität aller Daten zu prüfen. Es wird weiterhin verwendet, um das Nutzdatenfeld von Media Access Control address (MAC)-Daten zu verschlüsseln und zu entschlüsseln.

Der AppSKey wird vom Endgerät und Server verwendet diesmal, um das Nutzdatenfeld von anwendungsspezifischen Daten zu verschlüsseln und zu entschlüsseln. Die Anwendungsnutzdaten werden zwischen dem Endgerät und dem Anwendungsserver Ende-zu-Ende verschlüsselt. Das heißt der Netzwerkserver kann möglicherweise den Inhalt der übertragene Daten ändern.

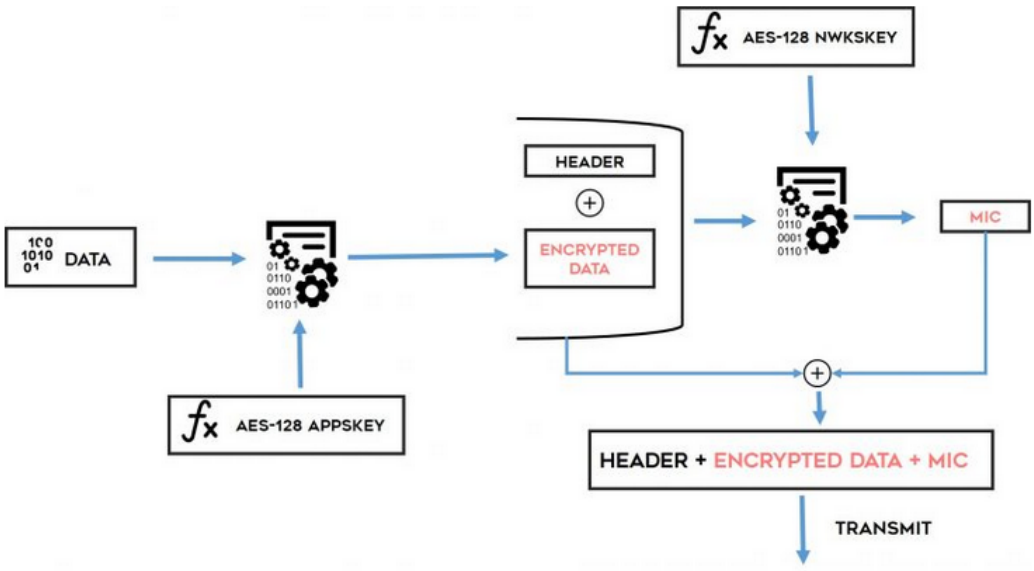


Abbildung 2.15: LoRaWAN-Nachricht Verschlüsselung [3]

Laut Abbildung 2.15 werden die zu sendende Daten erst mit dem AppSKey verschlüsselt. Ein Header, der andere Adressen des Endgerät enthält, wird den verschlüsselten Daten hinzugefügt. Nach dieser Verknüpfung wird das MIC berechnet, der nach der Berechnung an den verschlüsselten Daten und dem Header hinzugefügt wird. Nun können die Daten versandt werden.

Nachdem der Server die Daten empfangen hat, kann die Integrität dieser Daten

vom Server mithilfe des MICs geprüft werden. Die Daten werden nur berücksichtigt, wenn das MIC stimmt, ansonsten werden sie verworfen.

2.2.3 Aktivierung des Endgeräts

Damit ein Endgerät dem LoRaWAN-Netzwerk hinzugefügt werden kann, muss es erst spezifiziert und aktiviert werden. Die Aktivierung eines Endgerät kann auf zwei Arten erfolgen, entweder per Over-The-Air-Aktivierung (Englisch ***Over-The-Air Activation (OTAA)***), oder per Aktivierung durch Personalisierung (Englisch ***Activation By Personalization (ABP)***), wobei die zwei Schritte der Personalisierung und Aktivierung in einem Schritt erfolgen.

Aktivierung durch OTAA

Damit die Over-The-Air-Aktivierung vollständig wird, müssen Endgeräte zwecks Datenaustausch mit einem Server einem Join-Verfahren folgen. Dieses Verfahren wird durchgeführt, wenn ein Endgerät die Sitzungsinformationen verloren hat. Bevor ein Endgerät das Join-Verfahren startet, muss er folgende Informationen haben: eine eindeutige globale Endgerätekennung (**DevEUI**), eine Anwendungskennung (**AppEUI**) und ein AES-128-Schlüssel (**AppKey**).

AppEUI: ist ein 8-Byte-Wert, codiert in Hexadezimalformat und bezeichnet eine Kennung des Anwendungsanbieter.

DevEUI: ist ein 8-Byte-Wert mit hexadezimaler Codierung und bezeichnet die eindeutige Kennung eines Endgeräts. Manche LoRa-Radiomodule erhalten bereits bei der Herstellung eine DevEUI zugeteilt. Sofern nicht bereits vorhanden, kann diese vom Anwendungsanbieter gesetzt werden.

AppKey: ist ein 16-Byte-Wert in Hexadezimalformat. Wenn ein Endgeät mit OTAA das Netzwerk beitritt, wird dieser Schlüssel zur Herstellung des NwkSKey und des AppSKey verwendet, um die Netzwerkkommunikation und die Anwendungsdaten zu verschlüsseln und zu prüfen.

Sobald das Endgerät mit diesen Informationen versehen ist, kann eine Join-Abfrage (**Join request**) an den Server gesendet werden. Der Server antwortet mit einem Join-Zustimmung (**Join accept**), wenn das Endgerät dem Netzwerk beitreten darf. Die Join-Accept-Nachricht wird wie ein normaler Downlink gesendet, benutzt jedoch zwei unterschiedliche Verzögerungen vergli-

chen mit *RECEIVE_DE-LAY1* und *RECEIVE_DELAY2* wie in Abschnitt 2.2.1 beschrieben ist. Diese Verzögerungen sind *JOIN_ACCEPT-_DELAY1* und *JOIN-_ACCEPT_DELAY2*. Dem Endgerät wird keine Antwort geschickt, wenn die Join-Abfrage abgelehnt wurde.

Die Join-Accept-Nachricht enthält eine 3-Byte-Anwendung-Nonce (*AppNonce*), eine Netzwerkkennung (*NetID*), eine Endgerätadresse (*DevAddr*), eine Verzögerung zwischen TX und RX (*RxDelay*) und eine optionale Liste der Kanalfrequenz (*CFList*). Die DevAddr und die AppNonce sind die wichtigste Informationen bei einer Join-Accept-Nachricht.

DevAddr: ist eine 4-Byte-Adresse, womit der Server und das Endgerät nach Aktivierung kommunizieren.

AppNonce: ist ein zufälliger Wert, die vom Netzwekserver bereitgestellt und vom Endgerät verwendet wird, um den NwkSkey und den AppSKey abzuleiten. Der NwkSkey und der AppSKey werden mit der internen Funktion ***aes128_encrypt*** (im LoRa-Radiomodule vom Hersteller zur Verfügung gestellt) bestimmt und wird wie folgt bestimmt [2]:

NwkSKey = aes128_encrypt(AppKey, 0x01 | AppNonce | NetID | DevNonce | pad)

NwkSKey = aes128_encrypt(AppKey, 0x02 | AppNonce | NetID | DevNonce | pad)

Nun können Endgeräte, die dem Neztwerk beigetreten sind, Informationen mit dem Netzwerkserver austauschen (Uplinks und Downlinks). Abbildung 2.16 beschreibt das oben erläuterte Verfahren.

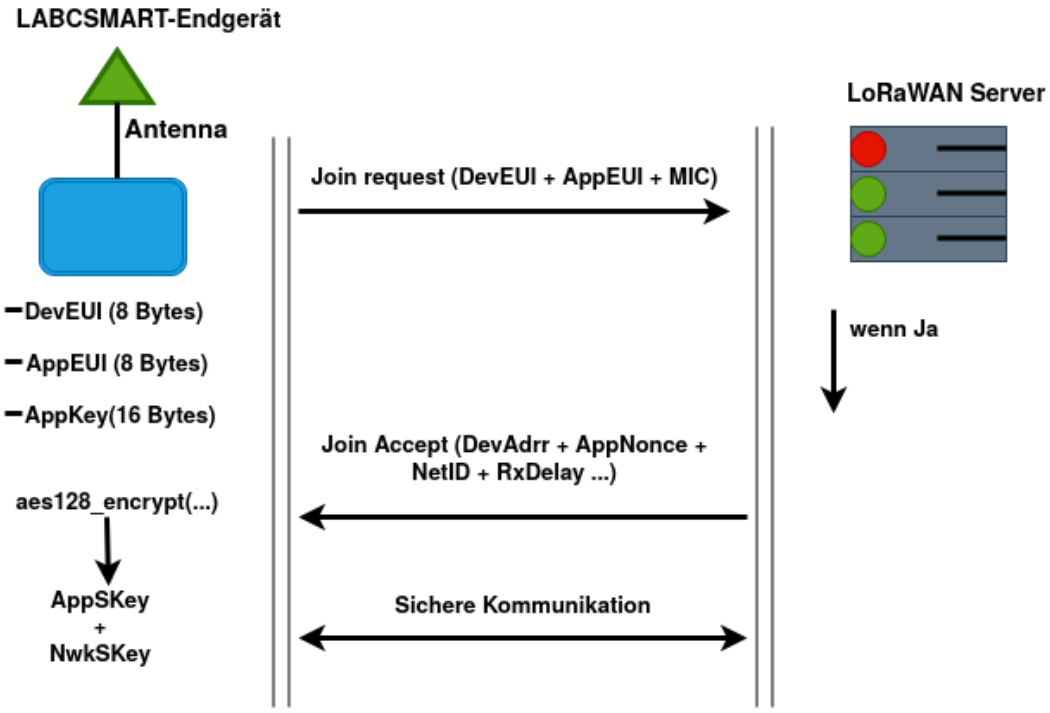


Abbildung 2.16: Join-Request Verfahren (Verändert von [3])

Aktivierung durch ABP

Bei dieser Aktivierungsart, muss das Endgerät keine Join-Abfrage senden, hier geht es um eine direkte Bindung eines Endgerät zu einem bestimmten Netzwerk. Das bedeutet, dass die *DevAddr* und die zwei Sitzungsschlüssel (*AppSKey* und *NwkSKey*) an Stelle der *DevEUI*, *AppEUI* und *AppKey*, im Endgerät gespeichert werden. Jedes Endgerät soll einen eindeutigen Satz von *AppSKey* und *NwkSKey* haben. Das Kompromittieren der Schlüssel eines Geräts sollte die Sicherheit der Kommunikation anderer Geräte nicht beeinträchtigen.

Zusammengefasst ist OTAA komplexer als ABP, aber bietet eine höhere Sicherheit. Falls ein Prototyp oder ein kleines Netzwerk erstellt werden soll, ist ABP genug. Wenn es um ein größers Netzwerk geht, wird OTAA empfohlen, weil es sicherer und agiler ist.

2.2.4 AT Kommandos

Nun wissen wir was LoRa und LoRaWAN sind und wie es funktioniert, aber nicht wie, sowohl Informationen (Appkey, NwkKey und mehr) dem Endgerät zugewiesen werden als auch wie die Daten an den Server gesendet werden. Im Kapitel 2.2 wurde das Wort "AT-Befehle" kurz erwähnt, in diesem Abschnitt erfahren Sie was diese Befehle sind und welche gebraucht werden, damit die eine Verbindung per OTAA oder per ABP erfolgreich wird.

Im UNIX-Systemen ist AT ein Kommando, welches bewirkt, dass andere Kommandos nur einmal ausgeführt werden. Hier wird es genutzt, um das i-nucleo-lrwan1 einzustellen. Da das LoRa-Radiomodul sich nicht selbst einstellen kann, ist auch die Aufgabe des B-L475E-IOT01A die Einstellung durchzuführen. Das i-nucleo-lrwan1 verfügt über eine UART-Schnittstelle (Siehe 2.6), um mit dem B-L475E-IOT01A zu komunizieren. Diese UART-Schnittstelle hat folgende Konsolenkonfiguration:

- Baudrate: 115200
- Daten: 8 Bit
- Parität: keine
- Stopbit: 1 Bit

Die Syntax dieser AT-Befehle ist wie folgt:

- Algemeine Kommandos:
  - **AT:** Prüft, ob die UART Schnittstelle benutzbar ist
  - **ATE [=<enable>]:** Aktivieren oder Deaktivieren des lokalen Echos
  - **ATZ:** Modul zurücksetzen
  - **AT+VERB [=<enable>]:** Ausführliche Antwort aktivieren oder deaktivieren
- LoRa MAC-Kommandos: **AT+Kommando [=parameter]** Die MAC-Kommandos werden hier nicht alle dargestellt, da es zu lang wäre sie alle zu erklären. Zur Erklärung aller Kommandos siehe das AT-Befehlsreferenzhandbuch [18]. Die Zeichnen [ ] bedeuten, dass der Parameter optional ist. Mit Parametern ist ein AT-Befehl wie ein Set-Befehl, ohne ist es ein Get-Befehl.

Da wir nun wiessen, wir diese Kommandos zu nutzen sind, können wir Beispiele für OTAA- und ABP-Aktivierung machen (Diese Einstellungen wurden getestet und funktionieren einwandfrei).

OTAA: Die Folgenden AT-Befehle werden von B-L475E-IOT01A nacheinander per UART am i-nucleo-lrwan1 gesendet.

- **AT+BAND=0:** Setzt die Region des Netzwerks (Hier EU868)
- **AT+CLASS=0:** Die Klasse A wird verwendet
- **AT+DC=1:** Deaktiviert den Auslastungsgrad
- **AT+DR=0:** Setzt die TX-Datenrate (LoRa SF12/125KHz 250 Bit/s)
- **AT+RX2DR=0:** Setzt die RX2-Datenrate (LoRa SF12/125KHz 250 Bit/s)
- **AT+RX1DT=1000:** Setzt die Verzögerung des ersten Empfangsfensters (in ms)
- **AT+RX2DT=2000:** Setzt die Verzögerung des zweiten Empfangsfensters (in ms)
- **AT+JRX1DT=5000:** Setzt die Verzögerung des ersten Join-Accept-Empfangsfensters (in ms)
- **AT+JRX2DT=6000:** Setzt die Verzögerung des zweiten Join-Accept-Empfangsfensters (in ms)
- **AT+RF=14,8671000000,12,0,1:** Konfiguriert das LoRa-Radiomodule. Ausgansleistung: 14dBm, Frequenz: 867.1MHz, Spreading factor: SF12, Bandbreit: 125KHz, Cyclic Codingrate: 4/5.
- **AT+APPEUI=ABC123ADF135CBD8:** Setzt die AppEUI
- **AT+AK=00112233445566778899AABBCCDDEEFF:** Setzt den AppKey
- **AT+JOIN=1:** Sendet eine Join-Abfrage als OTAA

ABP: Die Folgenden AT-Befehle werden von B-L475E-IOT01A nacheinander per UART am i-nucleo-lrwan1 gesendet.

- **AT+BAND=0**
- **AT+CLASS=0**
- **AT+DC=1**
- **AT+DR=0**
- **AT+RX2DR=0**

- **AT+RX1DT=1000**
- **AT+RX2DT=2000**
- **AT+JRX1DT=5000**
- **AT+JRX2DT=6000**
- **AT+RF=14,8671000000,12,0,1**
- **AT+ADDR=12345678:** Setzt die Gerätadresse
- **AT+NSK=1122334455663EAB546829CB361CAB7D:** Setzt den NwkS-Key
- **AT+ASK=887766554433BCFACDE52476CA4598BA:** Setzt den App-SKey
- **AT+JOIN=0:** Sendet eine Join-Abfrage als ABP

Daten senden: **AT+SEND=2,Daten,1** Hier werden die Daten durch den Port 2 gesendet. Die Daten mussen in hexadezimalem Format angegeben werden, und sollen nicht größer als 64 Bytes sein. Die 1 am Ende steht für die Bestätigung des Datenempfangs.

Im AT-Befehlsreferenzhandbuch stellt das Appendix 3 Tabellen für die Konfiguration der Datenrate abhängig von der Region zur Verfügung.

3 Gateway und LoRaWAN-Server

Nun ist es möglich ein Endgerät so einzustellen, dass es fähig ist Uplinks an einen LoRaWAN-Server zu senden und Downlink vom Server zu bekommen. Aber was ist der LoRaWAN-Server und wozu wird das Gateway benutzt. Diese Fragen werden in diesem Kapitel beantwortet.

3.1 Gateway

Es gibt fertige Gateways auf dem Markt, die man kaufen und direkt einsetzen kann. Für diese Thesis wird ein selbst gebautes Gateway benutzt. Der Grund dafür ist. Noch dazu steckt eine wissenschaftliche Idee dahinter. Zu wissen, wie ein Gateway gebaut wird, welche Komponenten und welche Software ins Spiel kommen.

Ein Gateway ist ein Gerät, das aus mindestens einem Konzentrator, einem Host und einer Netzverbindung zum Internet oder ein privates Netzwerk (Ethernet, 3G, Wi-Fi), möglicherweise einem GPS-Empfänger, besteht. Der Konzentrator ist ein Board, das Funkpakete senden und empfangen kann. Ein Konzentrator basiert auf einem Semtech-Mehrkanalmodem (***SX130x***), einem Transceiver (***SX135x***) und/oder eigenständige Modems mit geringem Stromverbrauch (***SX127x***).

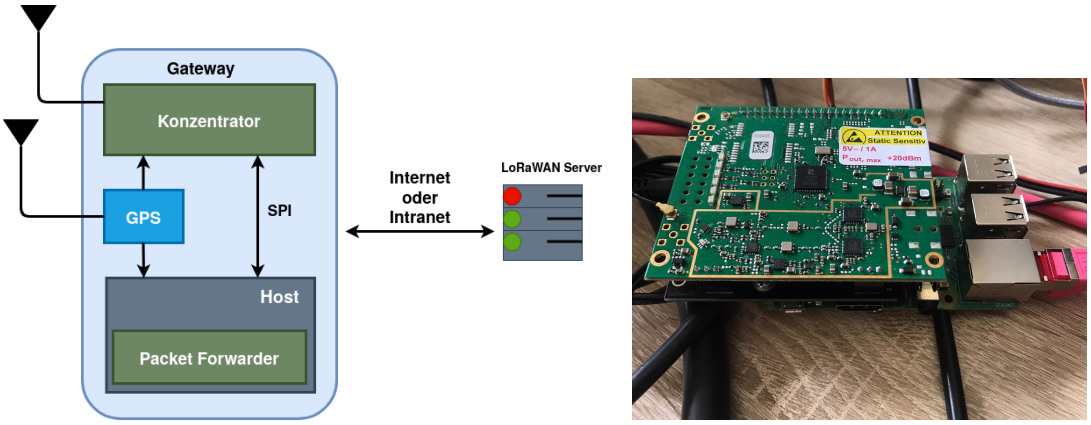
Ein Host ist ein eingebetteter Computer, auf dem die Paketweiterleitung ausgefürt wird. Der Host steuert den Konzentrator über eine SPI-Schnittstelle. Für diese Arbeit ist der Host ein Raspberry-Pi. Ein Gateway kann viele Endgeräte gleichzeitig behandeln. Die Kommunikation zwischen einem Endgerät und einem Gateway ist bidirektional. Das heißt, das Endgerät sendet dem Gateway Daten, kann aber auch von dem Gateway Daten empfangen.

Die Kommunikation von einem Endgerät zum Gateway ist ein Uplink, während die Kommunikation vom Gateway zum Endgerät ein Downlink ist.

Ein Endgerät sendet Uplinks als Broadcast, das heißt die gesendeten Daten werden von allen Gateways des Netzwerks empfangen. Das Gateway leitet das Datenpaket an den Netzwerkserver weiter. Der Netzwerkserver sammelt die Nachrichten aller Gateways, filtert doppelte Daten heraus und bestimmt das Gateway mit der besten Rezeption. Der Netzwerkserver leitet seine Daten zu dem entsprechenden Anwendungsserver, womit der Nutzer die Sensordaten ansehen und/oder verarbeiten kann.

Bekommt der Netzwerkserver eine Antwort vom Anwendungsserver, bestimmt der Netzwerkserver welches Gateway benutzt wird, um dem Endgerät die Antwort zu senden (Downlink).

Der Konzentrator kann Funktpakete zwar empfangen und senden, er ist jedoch lediglich eine elektronische Komponente und benötigt eine Software, um empfangene oder zu sendenden Pakete zu bearbeiten. Diese Software heißt LoRa-Packet-Forwarder [12]. Der Paket-Forwarder ist ein Programm, das auf dem Host ausgeführt wird, um Funkpakete, die vom Konzentrator empfangen werden, über eine IP/UDP- Verbindung an den Server weiterzuleitet und sendet die vom Server gesendeten Funkpakete weiter. Der Packet-Forwarder kann auch ein netzwerkweites synchrones GPS-Signal senden, das zur Koordination aller Endgeräte des Netzwerks verwendet wird.



(a) (b)

Abbildung 3.1: LABCSMART LoRaWAN Gateway

Abbildung 3.1(a) zeigt die Netzwerkkomponenten, beginnend mit dem Gateway und seinen Bestandteilen. Es ist deutlich zu sehen, dass das der Konzentrator und der Host durch einer SPI-Schnittstelle verbunden sind und die Verbindung zwischen dem Gateway und dem Server das Internet ist. Das heißt,

der Konzentrator, das Gateway und der Server sind zusammen in einem Block eingebaut. Der Raspberry-Pi ist gleichzeitig der Host und der Server.

3.2 Einstellung des LoRaWAN-Servers

An dieser Stelle ist die Arbeit fast fertig, da unsere Anwendung theoretisch in der Lage ist, Uplinks an das Gateway zu schicken. Nun konzentrieren wir uns auf die Verarbeitung der empfangenen Daten. Die Daten werden zwar in die Luft gesendet, aber der Benutzer kann diese nicht sehen oder verarbeiten, dafür ist ein Server zuständig. Dieser Server soll in der Lage sein, gesendete Funkpakete zu interpretieren und darzustellen, sodass der Benutzer diese lesen und verstehen kann.

Der verwendete Server heißt **lorawan-server** (Open-source). Er wurde von Herrn **Petr Gotthard** [5] entwickelt und ist ein kompakter Server für private LoRaWAN-Netzwerke. Dieser Server dient nicht nur als Netzwerkserver, sondern auch als Anwendungsserver. Man kann damit alle Ereignisse und alle Daten ansehen, die entweder vom Endgerät oder vom Packet-Forwarder kommen. Der Server wurde in 79% in der Programmiersprache Erlang [10] geschieben.

In diesem Kapitel, erfahren Sie wie dieser Server einzustellen ist, um Endgeräte mit OTAA oder ABP zu verbinden. Bevor ein Endgerät hinzugefügt wird, muss der Server dazu vorbereitet werden. Er muss die MAC-Adresse des Gateways, der Netzwerk, das Profil des Netzwerks und die Gruppe des Endgeräts kennen.

Gateway: Der Server kann mit einem oder mehrere Gateways verbunden werden (Nur eins in unserem Fall). Der Server bekommt alle Uplinks, die von den Gateways weitergeleitet werden, betrachtet nicht welches Endgerät welchem Netzwerk gehört.

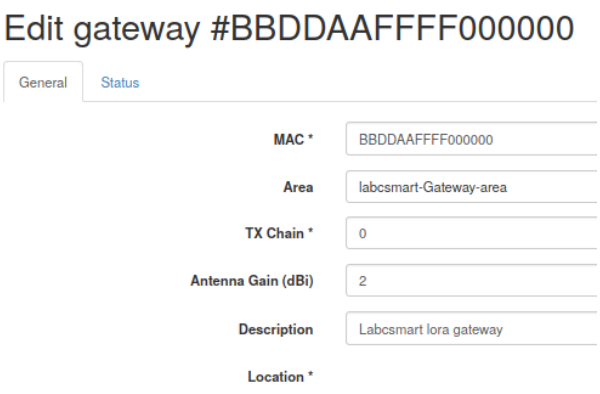
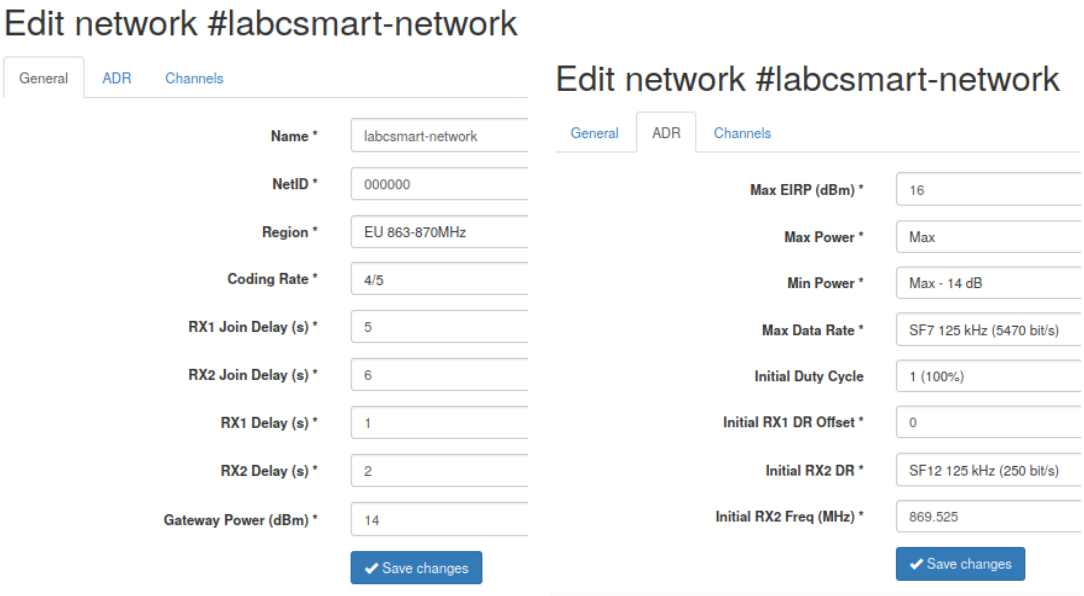


Abbildung 3.2: Einstellung des Gateways

Netzwerk: Der Server kann ein oder mehrere Netzwerke verarbeiten. Jede Netzwerkkonfiguration umfasst:

- Eine Netzwerkkennung, um die Endgerät-Adresse (DevAddr) neu verbundener Endgeräte zu erstellen.
- LoRaWAN-Regionparameter, einschließlich zusätzliche Frequenzen.



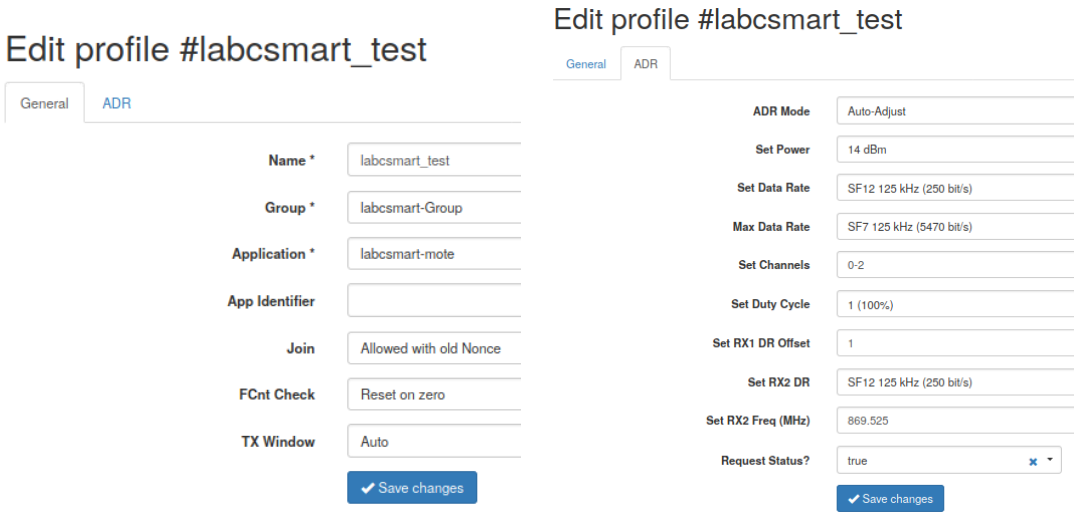
(a) (b)

Abbildung 3.3: Einstellung des Netzwerks

Profil: Das Profil repräsentiert eine bestimmte Hardware und alle statischen Einstellungen in der Firmware, die für eine Gruppe von Geräten gleich sind. Die Konfiguration umfasst:

- Die Referenz zu einem bestimmten Netzwerk.
- Die Fähigkeit des Geräts, Adaptive Data Rate (ADR) durchzuführen oder den Batteriestatus bereitzustellen.

Es ist zu bemerken, dass die Einstellung der Abbildung 3.4(b) genau ist, wie die Einstellung des Endgeräts, die im Abschnitt 2.2.4 erläutert wurde. Weil wir genau dieses Gerät im Netzwerk integrieren wollen, muss auch der Server entsprechend eingestellt werden.



(a) (b)

Abbildung 3.4: Einstellung des Profils

Gruppe: Die Gruppe repräsentiert eine Reihe von Profilen, die zu einem einzelnen Teilnetzwerk gehören. Zu einem einzelnen Kunde zum Beispiel.

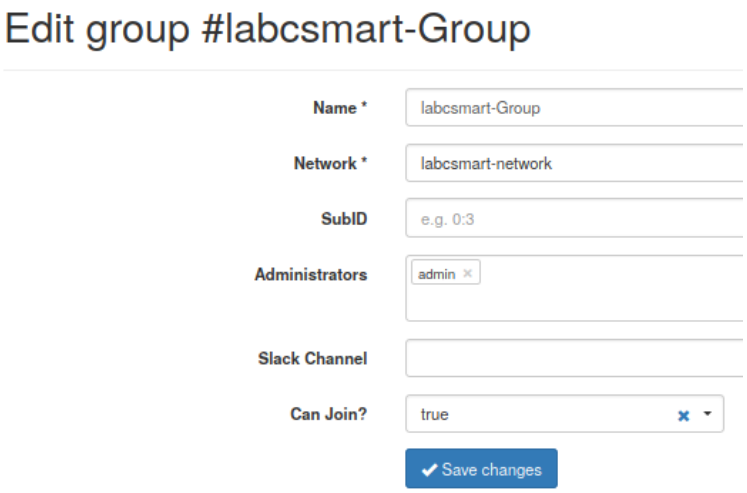
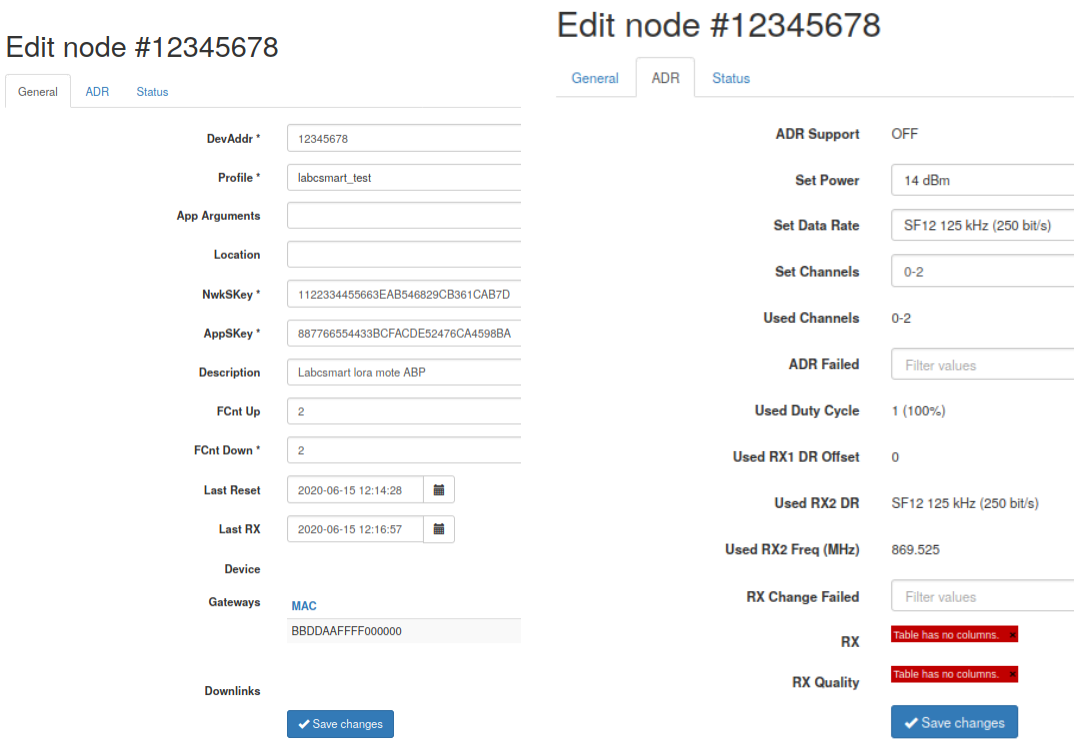


Abbildung 3.5: Einstellung der Gruppe

Nun ist es möglich Endgeräte zum Server hinzuzufügen. Als Erstes versuchen wir es mit ABP, dann mit OTAA.

ABP Verbindung

Geräte die mit ABP verbunden werden sollen, brauchen wir im Abschnitt 2.2.2 DevAddr, NwkSKey und AppSKey. Die Informationen auf Abbildung 3.6(b) werden automatisch erstellt, nachdem ein Endgerät dem Netzwerk hinzugefügt wurde. Man kann die Übertragungsfrequenzen, die Leistung und andere Einstellungen erkennen, die im Abschnitt 2.2.2 erwähnt wurden.



(a) (b)

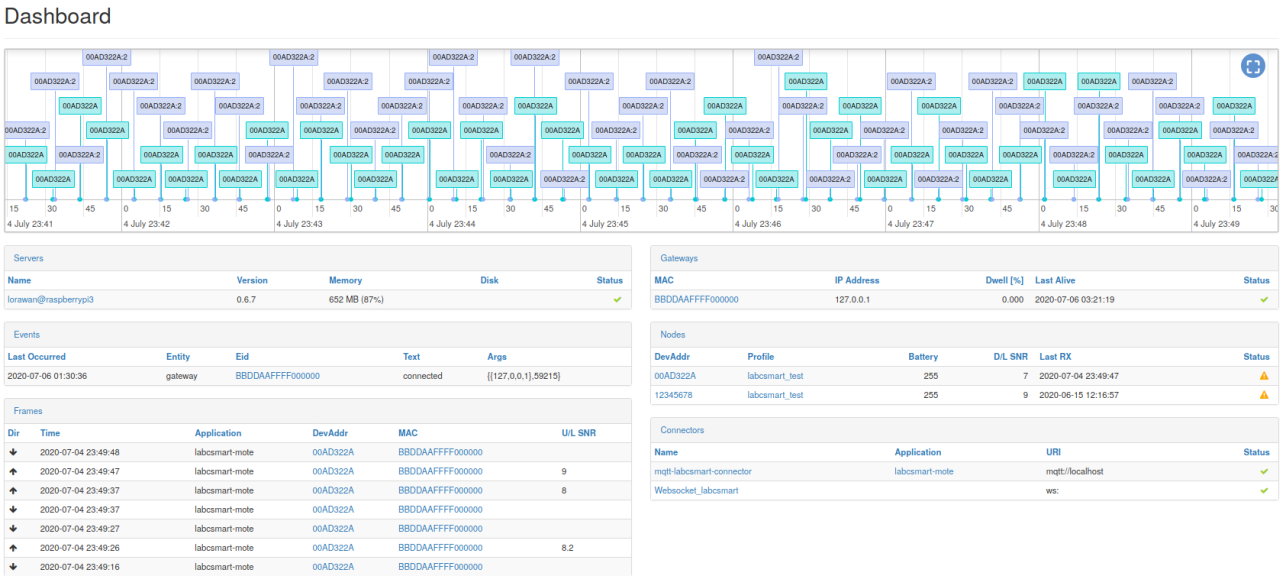
Abbildung 3.6: ABP Registrierung

OTAA Verbindung

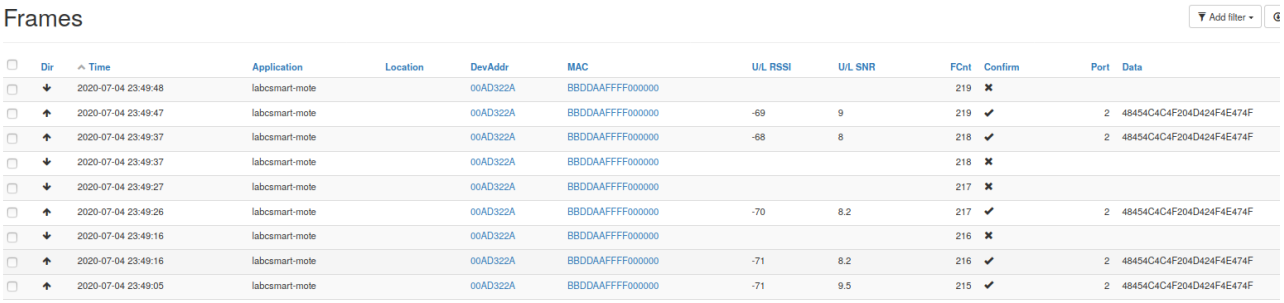
Hier befinden sich Geräte, die dem LoRaWAN-Netzwerk mithilfe von OTAA beitreten dürfen. Man muss dazu aber die DevEUI, AppEUI und AppKey auf jedem Fall kennen und im Server eingeben (Siehe 3.7). Nachdem das Gerät dem Netzwerk beigetreten ist, bekommt es eine DevAddr, einen NwkSKey und einen AppSKey, die mit Hilfe des AppKey berechnet wird zugewiesen (Siehe 2.2.2 für die Erklärung).

Abbildung 3.7: Einstellung OTAA

Nachdem diese Einstellungen fertig sind und die Geräte dem Netzwerk beigetreten sind, können Uplinks im Dashboard gesehen werden. Die übertragenen Daten (In Hexadezimal) befinden sich unter Frames (unten links vom Server).



(a)



(b)

Abbildung 3.8: Uplinks/Downlinks 3.8(a) und Frames 3.8(b)

Grüne Ereignisse im Abbildung 3.8(a) bezeichnen Downlinks, während Ereignisse in lila Uplinks sind. Die Spalte U/L RSSI des Abbildungs 3.8(b) stellt die empfangene Signalatärke (Englisch ***Received Signal Strength Indication (RSSI)***) dar. RSSI ist die empfangene Leistung in milliwatts, wird aber in (Decibel miliwatt (dBm) gemessen. Die RSSI sagt, wie gut ein Empfänger gesendete Signale empfangen kann. RSSI ist ein negativer Wert, je näher 0 dieser Wert ist, desto besser ist das Signal. Der minimale RSSI-Wert für die LoRa-Technologie ist -120dBm.

Zu Ende reicht dieses Ergebnis noch nicht. Was passiert, wenn der Server nicht mehr funktioniert? Soll die Anwendung auch nicht funktionieren oder gibt es eine andere Möglichkeit die Daten irgendwie zu sichern, falls ein Serverausfall vorkommen würde?

Zwischen dem Server und dem Packet-Forwarder gibt es ein Kommunikationsprotokoll namens Message Queuing Telemetry Transport (MQTT). Das nächste Kapitel erklärt was MQTT ist, wie es funktioniert und wie die Daten beim Serverausfall anderswo gesichert werden können.

3.3 MQTT Protokoll

MQTT hat sich in den letzten Jahren zum Standard Protokoll für die Machine-To-Machine- und IoT-Kommunikation von Geräten und Anwendungen entwickelt. MQTT steht für **Message Queue Telemetry Transport** und bietet eine sichere, zuverläsige, performante und wartbare Kommunikation zwischen Anwendungen und Geräten. Dieses Protokoll unterscheidet sich von Request/Response-Protokollen wie Hypertext Transfer Protocol (HTTP) dadurch, dass es alle Kommunikationsteilnehmer entkoppelt. Für den Datenaustausch werden Nachrichten über einen zentralen Verteiler an die Teilnehmer gesendet. Das ist der Grund, warum kein tieferes Wissen über empfangende Anwendungen vorhanden sein muss.

MQTT basiert auf eine 1:N-Kommunikation. Ein Teilnehmer sendet eine Nachricht, worauf einen oder mehreren Abonnenten Zugriff haben können. Teilnehmer können entweder Daten empfangen oder senden. Der MQTT-Broker ist der zentraler Verteiler, über den alle Kommunikationen stattfinden. Der Publisher produziert die Nachricht und der Subscriber abonniert sich an eine bestimmte oder an alle Nachrichten.

Der Begriff Topic, bezeichnet ein Ordner oder Unterordner, an dem Nachrichten veröffentlicht oder empfangen werden können. Möchte ein MQTT-Subscriber Nachrichten für ein Topic empfangen, abonniert er es beim MQTT-Broker. Ein Abonnement kann direkt für ein konkretes Topic erfolgen, oder es können Teilbäume der Topic-Hierarchie abonniert werden. Die abonnierten Subscriber werden durch den Broker vom Vorhandensein neuer Nachrichten benachrichtigt, statt selbst beim Server Änderungen anzufragen. Subscriber und Publisher bleiben über eine Transmission Control Protocol (TCP)-Verbindung mit dem MQTT-Broker verbunden. Diese Verbindung wird von den Clients (Subscriber und Publisher) selbst aufgebaut und benötigen anders als der Broker keine Ip-Adresse.

Kommen wir nun zurück auf den Zusammenhang zwischen MQTT und dieser Thesis. Das Kommunikationsprotokoll zwischen dem Packet-Forwarder und dem LoRaWAN-Server wurde im Kapitel 3.2 nicht erwähnt. Beide benutzen das MQTT-Protokoll, um jeweils Uplinks und Downlinks weiterzuleiten. Es ist derzeit möglich mit einer externen Anwendung an den vom MQTT-Broker veröffentlichten Topics zu abonnieren als auch selbst veröffentlichen.

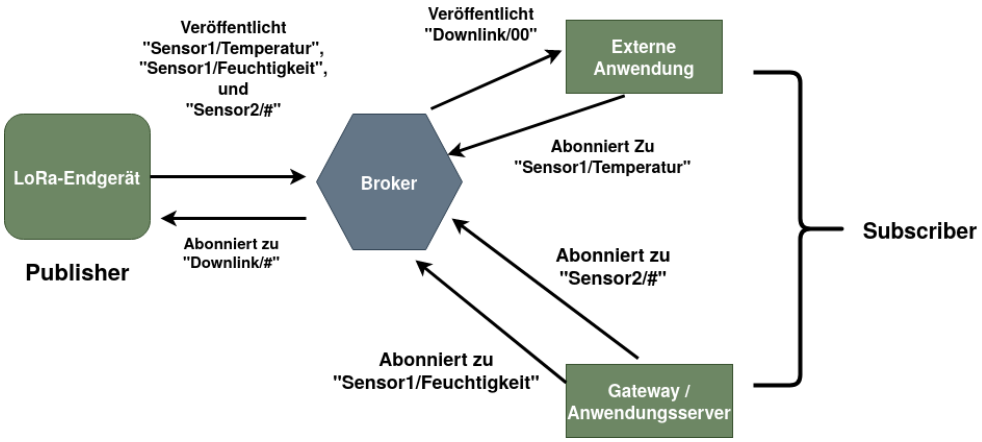


Abbildung 3.9: MQTT-Protokoll

Der MQTT-Broker soll auf den Entwicklungsrechner instaliert werden, sodass es möglich wird, Topics zu abonnieren oder Nachrichten zu veröffentlichen. **Mosquitto** ist die benutzte Software dazu und wird wie folgt installiert.

```
$ sudo apt-get install mosquitto
$ sudo apt-get install mosquitto-clients
```

Nach der Installation kann man schon Topics abonnieren oder eine Nachricht veröffentlichen. Um Topics zu abonnieren, bietet mosquitto das Kommando **Mosquito_sub** und wird wie folgt benutzt.

```
$ Mosquito_sub -h 192.168.0.17 -t out/devaddr
```

Die IP-Adressen gehört dem Server. Wir abonnieren somit alle Nachrichten, die an das Topic **out/devaddr** veröffentlicht werden. Mit dem Kommando **Mosquito_pub** können Nachrichten veröffentlicht werden.

```
$ Mosquito_pub -h 192.168.0.17 -m "Hello Server" -t in/devaddr
```

Falls der LoRaWAN-Server das Topic **in/devaddr** abonniert, würde er die Nachricht **Hallo Server** empfangen.

Achtung: devaddr bezeichnet die Adresse eines Endgeräts, sie ist von einem Endgerät zu einem anderen unterschiedlich.

Zwischen dem Gateway und dem LoRaWAN-Server ist ein MQTT-Broker, der als zentraler Server gilt. Beide Komponenten sind Clients des MQTT-Broker. Hier interessiersn wir uns auf die Konfiguration des Servers als MQTT-Subscriber und -Publisher. Der Server gilt als Subscriber, wenn er Daten vom Endgerät empfängt und als Publisher, wenn downlinks an das versendet Endgerät werden sollen. Es ist möglich mit einer externen Python-Anwendung auf Uplinks zu abonnieren oder Daten als Downlinks an den Server zu senden, da seine IP-Adressen zur Veröffentlichung verwendet wird.

Der Server veröffentlicht dann diese Daten an den MQTT-Broker. Das Gateway abonniert die vom Server veröffentlichte Topics und leitet diese an den Endgerät weiter. Das Endgrät enfängt und verarbeitet diese Daten. Wir können beispielsweise 2 LED's ansteuern, indem wir Downlinks wie im Tabelle 3.1 senden. (Siehe 3.10 Um den Server mit MQTT einzustellen)

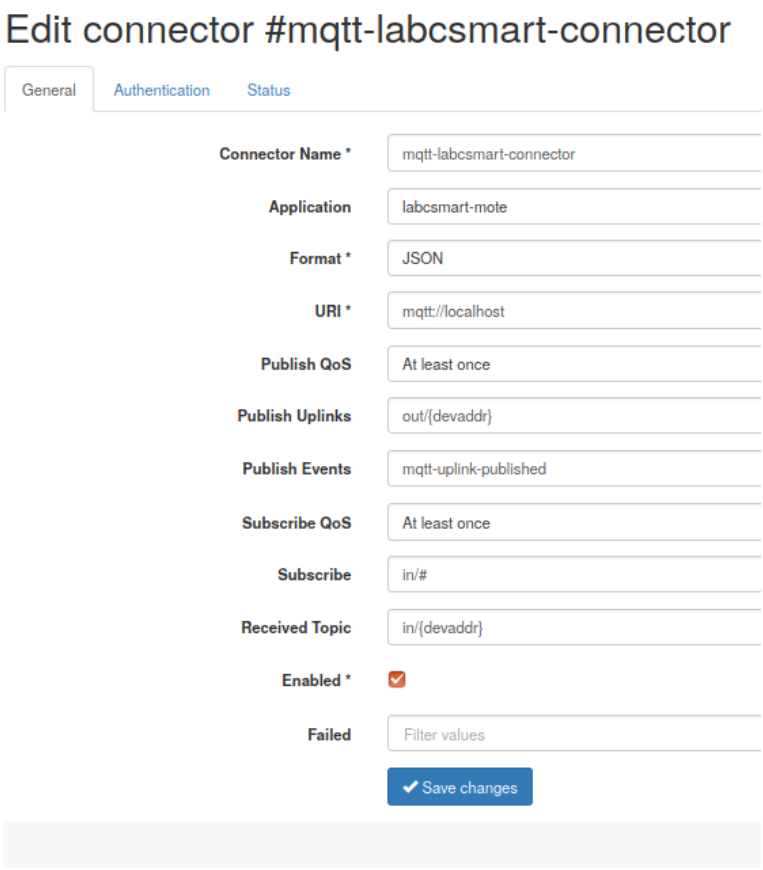


Abbildung 3.10: MQTT Connector vom Server

Es ist auf das Bild 3.10 zu sehen, dass der Server Nachrichten an topic **out/-devaddr** veröffentlicht, **out/devaddr** abonniert und an allen Nachrichten von devaddr interessiert ist.

Downlink	**ASCII-Code**	**LED 1**	**LED 2**
11	3131	AN	AN
10	3130	AN	AUS
01	3031	AUS	AN
00	3030	AUS	AUS

Tabelle 3.1: Downlink Nachrichten

Der Abschnitt 4.4 behandelt, wie Das Endgerät Downlinks empfängt und verarbeitet.

4 Software Implimmentierung

Bis jetzt wurden die Begriffe theoretisch erwähnt. Man kann zwar schon verstehen, was das Ziel der Arbeit ist, wie die gemessenen Daten gelesen werden und wie die LoRa-Technologie funktioniert. Das reicht aber noch nicht. In diesem Kapitel geht es um den Host (der Entwicklungsrechner), die Tools, die zur Entwicklung der eingebetteten Software zur Ansteuerung der Sensoren und des LoRa-Moduls eingesetzt wurden. Das gesamte Projekt wurde zum größten Teil in der Programmiersprache C geschrieben.

4.1 Entwicklungsumgebung

Das gesamte Projekt wurde unter Ubuntu durchgeführt. Ubuntu ist eine berümte Linux-Distribution. Da die Kosten der Entwicklung so niedrig wie möglich gehalten werden sollen, wurde eine Linux-Distribution ausgewählt. Der Grund dafür ist die Freiheit von Linux und die Unentgeltlichkeit vieler Entwicklungsumgebung.

4.1.1 Eclipse

Als Entwicklungsumgebung war **Eclipse** vorteilhaft, weil der Entwickler viele externen Tools hinzufügen kann und es sich einfach zum Bedarf angepassen lässt. In diesem Teil des Berichts, wird auf die Installation und die Konfiguration von Eclipse eingegangen, sodass der Entwickler C-Quelle-Codes durch einen Klick compilieren und mithilfe eines Lade-Kommando die Ausführbare Datei im Mikrocontroller laden kann.

Starten wir erstmal mit der Installation der Entwicklungsumgebung. Folgende Kommandos sorgen dafür:

```
$ sudo apt update
$ sudo apt install oracle-java8-installer
$ sudo apt install oracle-java8-set-default
```

Man sollte Eclipse von der offiziellen Website herunterladen. Nachdem die Software heruntergeladen wurde, befindet sie sich unter /Download. Die Software soll entpackt werden bevor die Installation erfolgt.

```
$ tar xfz ~/Downloads/eclipse-inst-linux64.tar.gz
$ ~/Downloads/eclipse-installer/eclipse-inst
```

Die Software startet, es wird nach den Paketen gefragt, die man installieren möchte. Für diese Arbeit brauchen wir das Paket **Eclipse IDE for C/C++ Developers**. Nachdem es ausgewählt wurde, kann man nun die Installation starten.

Nach der Installation kann man Eclipse eigentlich schon benutzen. C-Quelle-Codes für normalen Rechner können kompiliert werden, da der GCC-Compiler standardmäßig installiert wird. Wir wollen jedoch C-Codes für einen ARM-Mikro- controller compilieren. Aus diesem Grund müssen wir ein externes Tool für ARM-Mikrocontroller in Eclipse einfügen. Damit das erfolgt, soll das **Eclipse-Marketplace** unter **Help** gesucht werden. Sobald das Eclipse-Marketplace geöffnet ist, soll man nach **GNU MCU Eclipse** suchen und installieren.

Nun können wir ein Projekt erstellen. Bevor das geschieht, soll man ein Ordner erstellem, in dem das Projekt abgelegt wird. In diesem Ordner muss man ein MAKEFILE erstellen. Folgende Kommandos zeigen wie das geht.

```
$ mkdir projekt1
$ cd projekt1
$ touch Makefile
```

Die Datei ist momentan leer und wird später ausgefüllt. Danach wählt man **file -> new -> project** aus. Nun wird nach dem Typ des Projekts gefragt, wir wählen **Makefile Project with Existing Code** und dann **ARM Cross GCC** aus. Oben gibt man den Namen sowie das Verzeichnis des Projekts ein und drückt auf fertig. Nun ist unser Projekt angelegt, können aber noch keinen ARM-Code compilieren.

Wir wollen mithilfe von **MAKEFILES** unsere Quellen-Codes compilieren. Wir haben dadurch eine volle Kontrolle auf die Kompilierung, und die Fehleraus-

gaben werden schnell verständlich, da wir genau bekommen was wir auch in dem MAKEFILE spezifiziert haben. Wir müssen dazu sogenannte **Build Targets** hinzufügen.

Oben rechts von Eclipse befinden sich diese Build-Targets und müssen wie in Abbildung 4.1 konfiguriert werden.

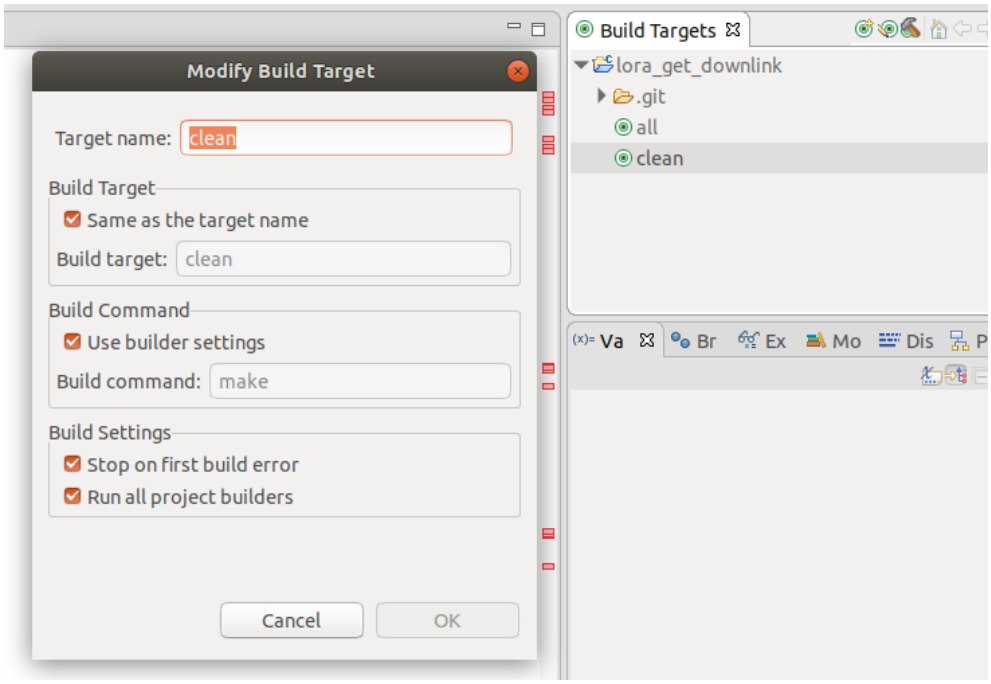


Abbildung 4.1: Build Targets einstellen

Nachdem die Build-Targets konfiguriert sind, erscheinen sie in dem **Project Explorer** wie es der Abbildung 4.2 zu entnehmen ist. Zum Compilieren soll man 2 mal auf **all** klicken, genau so wie auf **clean**, wenn Binär- und ausführbare Dateien gelöscht werden sollen.

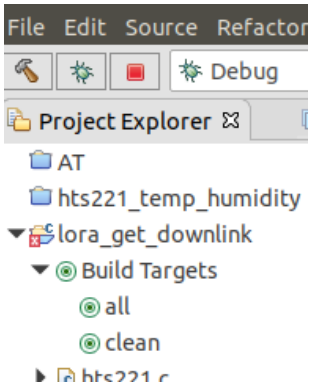


Abbildung 4.2: Build Targets fertig

```
PROJECT = lora_get_downlink //Name des Projekts
BUILD_DIR = bin //Verzeichnis der gebildete Dateien

CFILES = main.c //C-Quelle-Codes die Kompiliert werden muessen
CFILES += lora_modul.c
CFILES += lsm6dsl.c
CFILES += setup.c
CFILES += hts221.c
CFILES += systick.c

DEVICE=stm32l475vg //Der benutzte Mikrocontroller

LDSCRIPT = ./stm32-clicker.ld //Der Linker
LDFLAGS += -u _printf_float //Damit floating points durch UART
    dastellbar werden

VPATH += $(SHARED_DIR)
INCLUDES += $(patsubst %,-I%, . $(SHARED_DIR))
OPENCM3_DIR=../libopencm3 //Das Verzeichnis der benutzte Bibliothek

include $(OPENCM3_DIR)/mk/genlink-config.mk
include ./rules.mk
include $(OPENCM3_DIR)/mk/genlink-rules.mk
```

Der oben stehenden Textblock ist der Inhalt, der dem Makefile gehören muss, damit die Kompilierung erfolgt.

Bemerkung: Man kann das gleiche Makefile bei neuen Projekten benutzen, aber der Name des Projekts, der Mikrocontroller und die zu kompilierenden C-Quelle-Dateien müssen je nach Aufgabe angepasst werden.

Damit der kompilierte Code auf dem Mikrocontroller getestet werden kann, muss er in den Flash-Speicher gespeichert werden. Dafür brauch man ein zusätzliches Werkzeug. Dieses Werkzeug heißt **ST-Link** für STM32-Mikrocontroller. Dieses Werkzeug wird wie folgt installiert.

```
sudo apt-get install libusb-1.0-0-dev git
git clone https://github.com/texane/stlink stlink.git
cd stlink.git
make
```

Nun ist das Programm installiert, aber man kann es nicht einfach und überall ausführen. Damit es möglich wird, das Programm unter irgendwelchen Ordner

und ohne SUDO-Rechte auszuführen, muss es in dem Linux-Filesystem kopiert werden.

```
cd flash
sudo cp st-flash /usr/bin
cd ..
sudo cp *.rules /etc/udev/rules.d
sudo restart udev
```

Nehmen wir an, das der kompilierte Code **stlinkTest.bin** heißt. Das Board mit dem STM32L4-Mikrocontroller soll mit dem Entwicklungsrechner durch ein USB-Kabel verbunden werden. Jetzt kann man den Code mit ST-Link in dem Mikrocontroller speichern.

```
st-flash write stlinkTest.bin 0x8000000
```

4.1.2 Libopencm3 Bibliotek installieren

Sensoren, die für dieses Projekt eingesetzt wurden lassen sich durch eine I2C-Schnittstelle ansteuern, während die UART-Schnittstelle für das Senden von AT-Befehlen und den Empfang von Antworten zu AT-Befehle und Downlinks verwendet wurde. Wir können zwar diese Peripherien durch direkten Zugriff auf die Registern, aber das würde zu viel Zeit kosten, und der Code kann unübersichtlich werden. Damit wir uns diese Arbeit ein wenig sparen können, wurde eine Open-Source-Bibliothek benutzt **(libopencm3)** [7].

Diese Bibliothek stellt fertige Funktionen und Parameter zur Ansteuerung der Peripherien der ganzen ARM-Cortex-M3-Mikrocontroller-Familie zur Verfügung. Es gibt in dem GNU-ARM-Tool von Eclipse eine Bibliotek für ARM-Mikrocontroller. Diese wurde nicht verwenden, weil sie zu dem Makefile nicht passen würde, und die Dokumentation nicht gefunden wurde.

Libopencm3 verfügt über eine Doxygen-Dokumentation, die man mit einem Browser aufrufen kann. Man muss diese Bibliothek erstmal herunterladen bevor die Dokumentation verfügbar wird. Folgende Kommandos zeigen, wie das geht.

```
/* Bibliothek herunterladen und bilden */
$ git clone https://github.com/libopencm3/libopencm3
$ make

/* Beispiele herunterladen */
$ git clone https://github.com/libopencm3/libopencm3-examples
$ make

/* Dokumentation bilden */
$ cd libopencm3/doc/
$ make
```

Nach der Installation kann die Dokumentation mithilfe eines Internetbrowser geöffnet werden (Siehe Abbildung 4.3).

```
$ firefox index.html
```

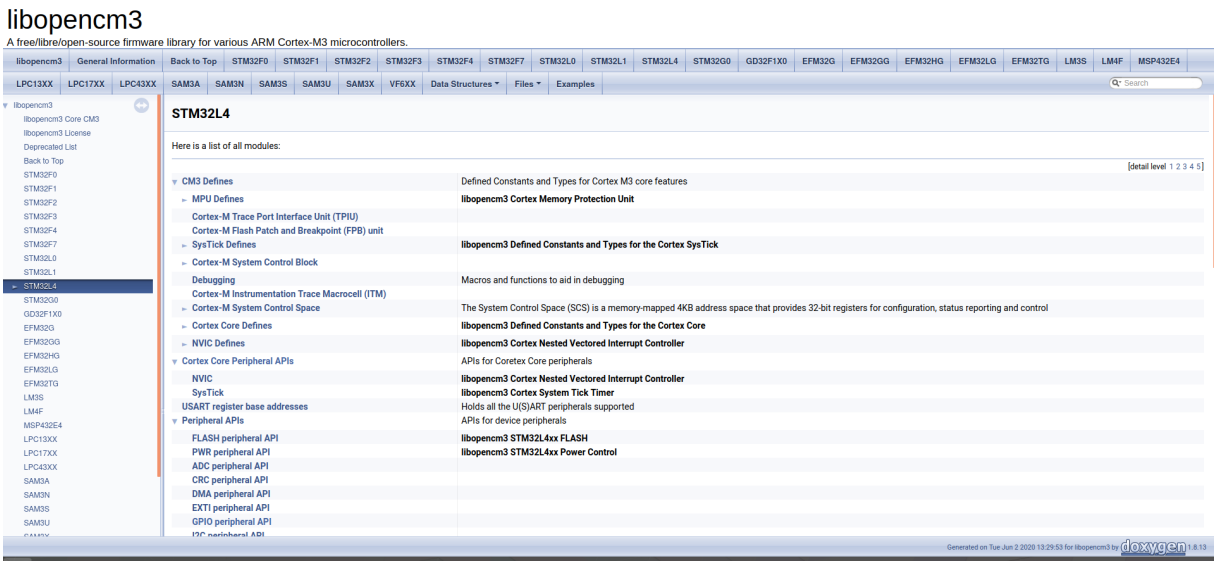


Abbildung 4.3: Libopencm3 Doxygen

4.2 Sensoren Auslesen

Da wir nun eine funktionsfähige Entwicklungsumgebung und eine STM32-kompatible Bibliothek haben, sind wir in der Lage C-Code zur Ansteuerung der Sensoren zu schreiben. In diesem Kapitel werden nur wichtige Teile des gesamten Quellcodes erklärt. Die Sensoren lassen sich per I2C ansteuern, dies bedeutet, dass

die Schnittstelle aktiviert und konfiguriert werden muss, damit wir sie überhaupt nutzen können. Der unten stehende Textblock zeigt die Aktivierung und Konfiguration der Schnittstelle.

```
1  void i2c2_setup(void)
2  {
3    rcc_periph_clock_enable(RCC_I2C2);
4    /* Setup SDA and SLC for I2C communication*/
5    gpio_mode_setup(GPIOB, GPIO_MODE_AF, GPIO_PUPD_NONE, SCL);
6    gpio_mode_setup(GPIOB, GPIO_MODE_AF, GPIO_PUPD_NONE, SDA);
7
8    /* Setup SDA and SCL pin as alternate function. */
9    gpio_set_af(GPIOB, GPIO_AF4, SCL);
10   gpio_set_af(GPIOB, GPIO_AF4, SDA);
11
12   i2c_peripheral_disable(I2C2);
13   i2c_enable_analog_filter(I2C2);
14
15   i2c_set_speed(I2C2,i2c_speed_sm_100k, 8);
16   i2c_enable_stretching(I2C2);
17
18   i2c_set_7bit_addr_mode(I2C2);
19   i2c_peripheral_enable(I2C2);
20 }
```

I2C ist eine Multi-Master-, Multi-Slave-Komminukationsschnittstelle, das heißt einem oder mehrere Master können mit einem oder mehrere Slaves kommunizieren. Für dieses Projekt haben wir einen Master **(STM32L4)** und zwei Slaves **(HTS221 und LSM6DSL)** (Siehe Abbildung 4.4). Jeder Slave hat eine Adresse in Hexadezimal, womit der Master ihn ansprechen kann. Der Master muss diese Adresse kennen, sonst können beide nicht kommunizieren.

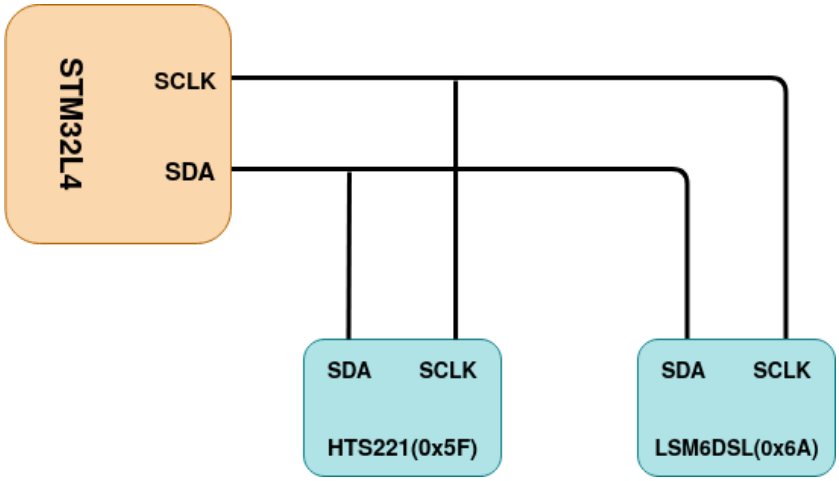


Abbildung 4.4: I2C Kommunikation

Im Kapitel 2.1.1 wurde die Funktionsweise beider Sensoren und die Vorgehensweise zur Ermittlung der gemessenen Daten behandelt. In diesem Abschnitt wird gezeigt wie man Werte in Sensoren-Register schreibt und wie diese Register auszulesen sind. Im Anhang befinden sich alle Quell- und Header-Dateien. In den Header-Dateien **hts221.h** und **lsm6dsl.h** sind alle Register (Als Makro-Definition), Funktionen zur Initialisierung und zum Auslesen der jeweiligen Sensoren definiert. In den Quell-Dateien **hts221.c** und **lsm6dsl.c** sind diese Funktionen implementiert. Der folgende Textblock zeigt, wie Register gelesen werden und wie man in diesen Registern schreiben kann. (Dieses Beispiel bezieht sich auf dem HTS221-Sensor)

```
1  uint8_t cmd[2];
2  cmd[0] = CTRL_REG1;
3  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);
4  cmd[1] |= (HTS221_PD_ON | HTS221_BDU_ON | HTS221_ODR1_ON);
5  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 2, NULL, 0);
```

Die Funktion **i2c_transfer()** hat sechs Parameter:

1. **i2c:** Die Peripherie, die benutzt wird (Hier I2C2)
2. **addr:** Die Slave-Adresse
3. **w:** Buffer der zu schreibenden Daten
4. **wn:** Größe der zu schreibenden Daten (In Byte)
5. **r:** Buffer, in dem die gelesenen Daten geschrieben werden sollen
6. **rn:** Größe der zu lesenden Daten (In Byte)

Achtung: Wenn Daten geschrieben werden sollen, ist **r** gleich dem NULL-Pointer und **rn** gleich null (0), da wir nichts lesen wollen.

Diese Vorgehensweise wird (mit unterschiedlichen Registern) mehrmals wiederholt, weil viele Kalibrierregister gelesen und zusammengerechnet werden müssen, um die gemessenen Werten zu bekommen. Die folgenden Textblöcke bezeichnen jeweils, wie die Temperatur vom **HTS221** und die Beschleunigung vom **LSM6DSL** ermittelt werden.

```
/* Temperatur ermitteln wie im Kapitel 2.1.1 beschrieben */

/* Read T0_degc_x8 and T1_degc_x8 */
cmd[0] = T0_DEGC_X8;
i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);
cmd[0] = T1_DEGC_X8;
i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);

/* Read the most significant bit of T1_DEGC and T0_DEGC */
cmd[0] = T1_T0_MSB;
i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+3), 1);

/* Calculate the T0_degc and T1_degc values */
T0_degc_x8 = (((uint16_t)(cmd[3] & 0x02)) << 8) | ((uint16_t)cmd[1])
    ;
T1_degc_x8 = (((uint16_t)(cmd[3] & 0x0C)) << 6) | ((uint16_t)cmd[2])
    ;
T0_degc = T0_degc_x8>>3;
T1_degc = T1_degc_x8>>3;

/* Read T0_OUT less significant bit */
cmd[0] = T0_OUT_L;
i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);

/* Read T0_OUT most significant bit */
cmd[0] = T0_OUT_M;
i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
T0_out = (((int16_t)cmd[2])<<8) | (int16_t)cmd[1];


/* Read T1_OUT less significant bit */
cmd[0] = T1_OUT_L;
i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);

/* Read T1_OUT most significant bit */
```

```
  cmd[0] = T1_OUT_M;
  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
  T1_out = (((int16_t)cmd[2])<<8) | (int16_t)cmd[1];


  /* Read T_OUT less significant bit */
  cmd[0] = T_OUT_L;
  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);

  /* Read T_OUT most significant bit */
  cmd[0] = T_OUT_M;
  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
  T_out = (((int16_t)cmd[2])<<8) | (int16_t)cmd[1];

  /* Calculate the temperature value */
  tmp = ((int32_t)(T_out - T0_out)) * ((int32_t)(T1_degc - T0_degc));
  temperature = ((float)tmp / (float)(T1_out - T0_out)) + (float)(
      T0_degc);

  return temperature;
```

```
  /* Beschleunigung ermitteln wie im Abbildung 2.5 */

  status_cmd[0] = STATUS_REG;
  i2c_transfer7(I2C2, LSM6DSL_ADDR, status_cmd, 1, (status_cmd+1), 1);

  if (status_cmd[1] & GET_XLDA) {
    /* Read X */
    cmd[0] = OUTX_L_XL;
    i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
    outx_l_xl = cmd[1];

    cmd[0] = OUTX_H_XL;
    i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
    outx_h_xl = cmd[1];

    x = ((int16_t)outx_h_xl << 8 | (int16_t)outx_l_xl);

    /* Read Y */

    cmd[0] = OUTY_L_XL;
    i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
    outy_l_xl = cmd[1];

```

```
24      cmd[0] = OUTY_H_XL;
25      i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
26      outy_h_xl = cmd[1];
27
28      y = ((int16_t)outy_h_xl << 8 | (int16_t)outy_l_xl);
29
30      /* Read Z */
31      cmd[0] = OUTZ_L_XL;
32      i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
33      outz_l_xl = cmd[1];
34
35      cmd[0] = OUTZ_H_XL;
36      i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
37      outz_h_xl = cmd[1];
38
39      z = ((int16_t)outz_h_xl << 8 | (int16_t)outz_l_xl);
40
41      acc_xyz[0] = (float)x * FS_XL_4G;
42      acc_xyz[1] = (float)y * FS_XL_4G;
43      acc_xyz[2] = (float)z * FS_XL_4G;
44    }
45  return acc_xyz;
```

Achtung: Die Adresse, die beim Auslesen der Beschleunigung zurückgegeben wird, enthält 3 unterschiedliche Werte. Der Index 0 beinhaltet den Wert der X-Achse, Index 1 beinhaltet den Y-Wert und Index 2 den Z-Wert. (analog für den Gyroskop).

Nachdem die Daten gelesen wurden, kann man sie nun in ASCI konvertieren und mithilfe eines AT-Befehls am Server per LoRaWAN senden. Der folgende Abschnitt zeigt, wie das geht.

4.3 AT-Kommandos senden

In diesem Teil geht es hauptsächlich um UART. Die Peripherie muss aktiviert und konfiguriert werden, bevor ein Transfer startet. Der folgende Quell-Code ist dafür zuständig. Die Konfugiration muss die gleiche sein wie im Abschnitt 2.2.4 beschrieben (Baudrate: 115200, Daten:8 Bit, keine Parität und 1 Stopbit).

```
rcc_periph_clock_enable(RCC_UART4);

void uart_setup(void)
{
  /* Setup GPIO pins for UART4 and USART3 transmit. */
  gpio_mode_setup(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO0);
  gpio_mode_setup(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO1);

  /* Setup UART4 TX and RX pin as alternate function. */
  gpio_set_af(GPIOA, GPIO_AF8, GPIO0);
  gpio_set_af(GPIOA, GPIO_AF8, GPIO1);

  /* UART4 setup */
  usart_set_baudrate(UART4, 115200);
  usart_set_databits(UART4, 8);
  usart_set_stopbits(UART4, USART_STOPBITS_1);
  usart_set_mode(UART4, USART_MODE_TX_RX);
  usart_set_parity(UART4, USART_PARITY_NONE);
  usart_set_flow_control(UART4, USART_FLOWCONTROL_NONE);

  /* Finally enable UART4. */
  usart_enable(UART4);
}
```

Wie im Abschnitt 2.2.4 erwähnt, sollen AT-Befehle an das LoRa-Modul gesendet werden, um das Modul für eine LoRaWAN-Verbindung vorzubereiten. Diese Befehle werden ein Zeichen nach dem anderen per UART an dem LoRa-Modul gesendet. Da viele AT-Befehle Parameter erwarten, wurden mithilfe der Stringification Makros definiert, die dem Benutzer ermöglichen veränderliche Strings in C zu bilden.

Zuerst wurde eine Enumeration aller Befehle erstellt, die gebraucht werden. Danach mithilfe eines zweidimensionalen Feldes und der Stingification können wir Parametern zwischen den Strings setzen. Die Enumeration gibt den Anfangsindex jedes Befehls an. Damit kann man weiter iterieren, um ein kompleten String zu bilden. Die folgenden Zeilen-Code zeigen die Stingification, die Enumeration und das zweidimensionale Feld.

```

/*
* Stringification
* Helpful to convert macro argument into
* string constant
*/
#define AT_COMMAND(cmd, param) "AT+" #cmd "=" #param "\r\n\0"
```

```
/*
* List of possible AT command
*/
enum lora_cmd {
  AT = 0,
  ATZ,
  GET_APPEUI,
  GET_EUI,
  GET_NET_S_KEY,
  GET_APP_S_KEY,
  GET_APP_KEY,
  GET_ADDR,
  GET_JOIN_STATUS,
  AT_RADIO,
  AT_GETDATA,
  AT_RX1_DELAY,
  AT_RX2_DELAY,
  AT_JOIN_ACCEPT1_DELAY,
  AT_JOIN_ACCEPT2_DELAY,
  AT_DUTY_CYCLE,
  AT_BAND,
  AT_DATA_RATE,
  AT_RX2_DATA_RATE,
  AT_CLASS,
  AT_ADDR,
  AT_NET_S_KEY,
  AT_APP_S_KEY,
  AT_APP_KEY,
  AT_APPEUI,
  AT_JOIN_OTAA,
  AT_JOIN_ABP,
  AT_NETWORK_TYP,
  AT_ADAPTIVE_DATA_RATE,
  AT_MAX
};
```

```
/*
* List of possible codes
*/
/* Default network type is public (NTYP = 1, 0 for private network)
* Band = EU868
* DevEUI = e24f43fffe44c432
* Duty Cycle (DC) = is on
* Data Rate (DR) = SF7/125KHz
* Class = Class A
* APPEUI is a 8 bytes hex value
* AppKey (AK) NSK ASP are 16 byte hex value
* Device address (ADDR) is a 4 bytes hex value
* Radio setting: Power: 14dBm
*     Frequency: 867.1MHz
*     Spreading factor: 12
*     Bandwidth: 125KHz
*     Coding rate: 4/5
*
*/

char *code[] = {
  "AT\r\n\0",
  "ATZ\r\n\0",
  "AT+APPEUI\r\n\0",
  "AT+EUI\r\n\0",
  "AT+NSK\r\n\0",
  "AT+ASK\r\n\0",
  "AT+AK\r\n\0",
  "AT+ADDR\r\n\0",
  "AT+JSTA\r\n\0",
  "AT+RF=14,8671000000,12,0,1\r\n\0",
  "AT+RCV\r\n\0",
  AT_COMMAND(RX1DT, 1000),
  AT_COMMAND(RX2DT, 2000),
  AT_COMMAND(JRX1DT, 5000),
  AT_COMMAND(JRX2DT, 6000),
  AT_COMMAND(DC, 0),
  AT_COMMAND(BAND, 0),
  AT_COMMAND(DR, 0), //UE860: SF12-BW125
  AT_COMMAND(RX2DR, 0),//UE860: SF12-BW125
  AT_COMMAND(CLASS, 0),
  AT_COMMAND(ADDR, 12345678),
  AT_COMMAND(NSK, 1122334455663EAB546829CB361CAB7D),
```

```
44    AT_COMMAND(ASK, 887766554433BCFACDE52476CA4598BA),
45    AT_COMMAND(AK, 00112233445566778899AABBCCDDEEFF),
46    AT_COMMAND(APPEUI, ABC123ADF135CBD8),
47    AT_COMMAND(JOIN, 1),
48    AT_COMMAND(JOIN, 0),
49    AT_COMMAND(NTYP, 1),
50    AT_COMMAND(ADR, 1)
51  };
```

Nun ein Beispiel wie das ganze zu verwenden ist. Jedes Zeichen wird einem nach dem anderen durch die UART4-Schnittstelle des Mikrocontrollers gesendet.

```
1  void send_cmd(enum lora_cmd cmd)
2  {
3    char *command = code[cmd];
4    /*Send command*/
5    if(cmd >= AT_MAX)
6      return;
7    printf("cmd: %s\n",code[cmd]);
8    while (*command != '\0') {
9      usart_send_blocking(UART4, *command);
10     command++;
11   }
12 }
```

Nachdem ein Befehl gesendet wurde, bekommt man eine Antwort von dem LoRa-Modul zurück. Diese Antwort ist abhängig von dem Befehl. Geht es um einen Set-Befehl, bekommt man **OK** oder ein Fehler zurück. Ist der Befehle ein Get-Befehl, bekommt man die abgefragte Daten oder ein Fehler zurück. Eine Ausnahme besteht bei Events.

Nehmen wir an ein Endgerät möchte einem LoRaWAM-Netzwerk beitreten. Das Endgerät wird zugelassen, der Server sendet ein Join-Accept. Diese Antwort taucht wie ein Event genau so wie ein normales Downlink auf. Diese Antworten werden dann zur Püfung oder zur Verarbeitung von Downlinks verwendet. Um dieses Verhalten zu visualisieren, wurde ein Logic-Analyser namens Saleae [11] verwendet. Das folgende Bild zeigt das Ergebnis einer Messung. Es wird ein einfaches **AT** gesendet und das LoRa-Modul antwortet mit **OK**.

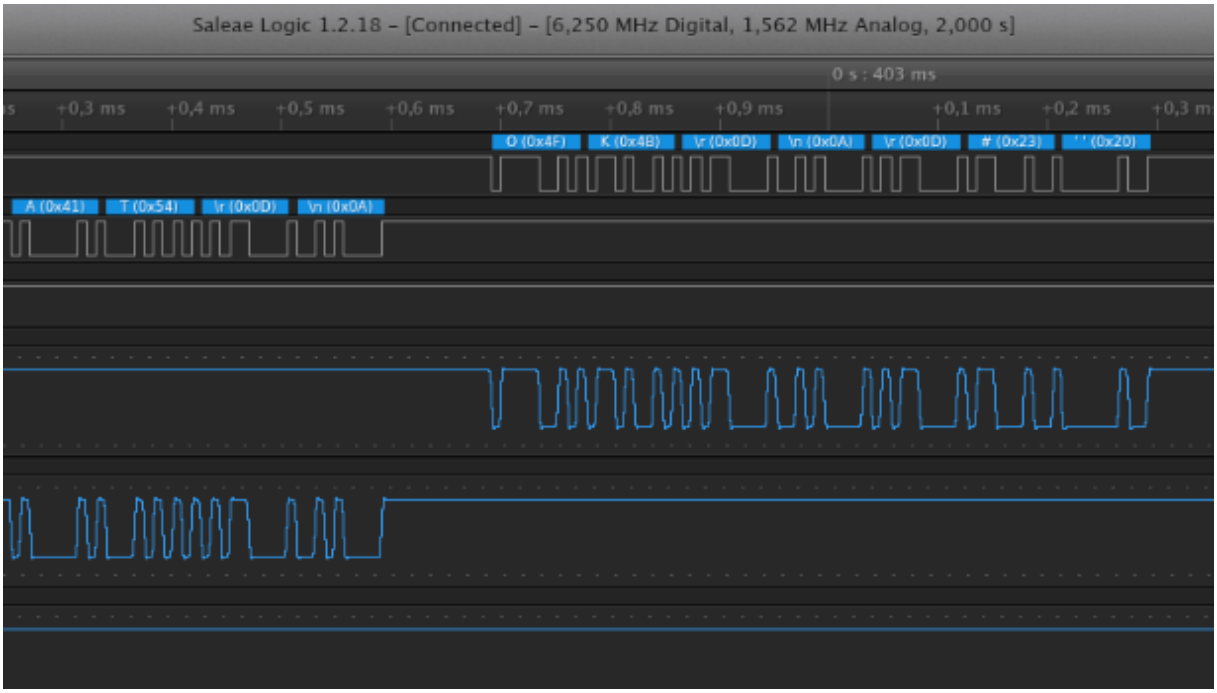


Abbildung 4.5: Verbindungstest mit Saleae

Diese Antwort wird mithilfe eines Interrupts abgefangen und in einem Buffer gespeichert. Um zu prüfen, dass ein Befehle richtig interpretiert wurde, wird der Inhalt des Antwort-Buffers mit dem String **OK** mit hilfe der System-Funktion **strcmp()** verglieschen. So können wir auch prüfen ob ein Endgerät dem Netzwerk eingetreten ist, indem der Antwort-Buffer mit dem Inhalt des Join-Status-Registers. Der Inhalt des Join-Status-Registers ist 1, wenn das Endgerät zugelassen wurde, und 0 wenn nicht. So soll das Endgerät Join-Request solange senden, bis das Endgerät dem Netzwerk zugelassen wird.

Bei Downlinks bekommen wir auch ein Event, das unterschiedlich zu dem Join-Accept oder die Antwort eines Befehls, zu behandeln ist. Das nächte Teil erlütert wie Downlinks abgefangen werden und welche Action kann man abhängig der empfangene Date durchführen.

4.4 Downlinks Behandlung

Mit dem aktuellen stand der Anwendung, ist das Endgrät in der Lage Uplinks zu senden, es kann zwar downlinks-Event empfangen aber nicht verarbeiten. In diesem Abschnitt geht es hauptsächlich um die Behandlung der Downlinks, die vom Server oder mit Mosquitto gesendet werden können.

Der Abbildung 4.5 kann man erkennen, dass das LoRa-Modul auf das Senden (TX) des Kommandos ***AT*** mit ***OK*** antwortet. Die RX-Leitung des Mikrocontrollers empfängt diese Antwort und kann sie mit einer Interrupt Service Routine (ISR) abfangen, um zu prüfen, ob das Senden des Kommandos problemlos abgelaufen ist. Genau wie diese Vorgehensweise können Downlinks abgefangen werden.

Eine Downlinks-Nachricht sieht folgendermassen aus:

```
+RCV=2,2,3130
```

Es bedeutet, dass die daten durch den Port **2** empfangen wurden, **2** Bytes groß sind und die eigentliche Nachricht der ASCII-String ***3130*** ist. Dieser ASCII-String wollen wir benutzen nachdem die komplete Antwort in einem Buffer gespeichert wurden. Da die Nutztdaten ab Index 9 des Empfangsbuffers beginnt, ist es möglich den Empfangsbuffer als **EXTERN** in dem gobalen Header-Datei (**setup.h**) zu deklarieren, sodass es in anderen Dateien verwendbar wird.

```
#ifndef EXTERN
#define EXTERN extern
#endif

EXTERN volatile char payload[30];
```

So kann der Buffer **payload** von dem Empfangsinterrupt der UART Schnittstelle verändert werden (Siehe die Datei **lora_modul.c**).

```
int j = 0;
int joined = 0;

void uart4_isr(void)
{
  if(usart_get_flag(UART4, USART_ISR_RXNE) && joined){

    /* Get downlink
    * EX: +RCV=2,2,3030
    * The gateway send packet to module for APP port 2
    * The packet's payload size is 2 Bytes
    * The payload data in Hex-format string
    */

    payload[j] = usart_recv(UART4);
    j++;

```

```
18    if(j == 13){
19      nvic_disable_irq(NVIC_UART4_IRQ);
20      usart_disable_rx_interrupt(UART4);
21      usart_disable(UART4);
22      j = 0;
23      joined = 0;
24      downlink = 1;
25    }
26  }
27 }
```

Diese ISR wird ausgeführt, wenn jedes zeichen des Downlinkstrings an dem RX-Pin der UART Schnitstelle bereit zum Lesen ist. Nachdem der komplette Downlink in dem Buffer gespeichert wurde, kann es abhängig des Ziels verarbeitet werden. Mehnem wir mal das Beispiel der Tabelle 3.1, wir können LED's abhhängig von den empfangenen Daten an-, ausmachen oder blinken lassen. Die folgenden Zeilen implementieren dieses Beispiel.(Siehe die Datei **main.c**)

```
1  static void downlink_event(char *payload)
2  {
3    if(strcmp(payload, "3131") == 0){ //11
4      led_toggle(1,1);
5    }
6    else if(strcmp(payload, "3130") == 0){ //10
7      led_toggle(1,0);
8    }
9    else if(strcmp(payload, "3031") == 0){ //01
10     led_toggle(0,1);
11   }
12   else if(strcmp(payload, "3030") == 0){ //00
13     led_toggle(0,0);
14   }
15   else{
16     printf("Strings does not match\n");
17   }
18   usart_enable(UART4);
19   nvic_enable_irq(NVIC_UART4_IRQ);
20   usart_enable_rx_interrupt(UART4);
21 }
22
23
24
```

```
25 char temp[20];
26 while(1){
27   temperature = read_temperature();
28   sprintf(temp,"%lX", *(unsigned long*)&temperature); // Temperatur in
         Hex-String umwandeln
29
30   send_data(temp, AT_OK);
31   msleep (5000);
32
33   if(downlink){
34     printf("Payload: %s\n", payload+9);
35     downlink_event(payload+9);
36
37     /* Reset to catch next downlink*/
38     downlink = 0;
39   }
40 }
```

Es ist deutlich zu bemerken, dass das feld **payload** dank der Qualifikation **EXTERN** in zwei unterschiedlichen Dateien verwendet wurde, obwohl es in diesen Dateien nicht deklariert wurden. Zu Testszwecken wurde das Szenario mit LED's ausgedacht, allerdings kann das Szenario unterschiedlich sein, solange die Downlinks zu sinvollen Zwecken benutzt werden. Es wäre zum Beispiel möglich, die Heizung ein bzw. auszuschalten, je nach der gemessenen Temperatur.