

Technische Hochschule Mittelhessen

Fachbereich Informationstechnik - Elektrotechnik - Mechatronik

LoRa based IoT endpoint and gateway

Bachelorarbeit

von

Silvere Sacker Ngoufack

Betreuer:

Erster Prüfer Prof. Dr.-Ing. Hartmut Weber

Zweiter Prüfer Prof. Dr.-Ing. Martin Gräfe

Friedberg, den 10. Juli 2020

Inhaltsverzeichnis

Danksagung	iii
Selbständigkeitserklärung	iv
Abstract	viii
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-Protokoll	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 Emplimentierung	41
4.1 Entwicklungsumgebung	41
4.1.1 Eclipse	41
4.1.2 Libopencm3 Bibliothek installieren	44
4.2 Sensoren Auslesen	46

4.3	AT-Kommandos senden	50
4.3.1	Downlinks Behandlung	55
5	Fazit	56
5.1	Zusammenfassung	56
5.2	Ausblick	56

Danksagung

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

Zuerst bedanke ich mich herzlich bei John Martial Madieu mein Chef. Für seine hilfreichen Anregungen und seine konstruktive Kritiken bei der Entwicklung dieser Arbeit möchte ich mich bedanken.

Ich bedanke mich bei meiner Eltern, dafür das sie mich immer unterstützt haben.

Ein besonderer Danke gilt meiner Freundin und meinen Freunden, die durch ihre Fragen und Anmerkungen mein Wissen erweitert haben.

Selbständigkeitserklärung

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

Abbildungsverzeichnis

1.1	Allgemeine LoRaWAN Netzwerkarchitektur [2]	2
1.2	Labcsmart LoRaWAN 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 LSB to %RH [13]	9
2.4	Linear interpolation to convert LSB to °C [13]	11
2.5	Flußdiagramm zur Datenermittlung	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 Ver- bindung (b)	16
2.9	Vergleich zwischen LoRa und andere IoT Kommunikationstech- nologien [2]	19
2.10	Klassen von LoRaWAN [2]	20
2.11	Klasse A	21
2.12	Klasse B	22
2.13	Klasse C	22
2.14	LoRaWAN-Nachricht Verschlüsselung [3]	23
2.15	Join-Request Verfahren	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 Verbindung	36
3.7	Einstellung OTAA	37
3.8	Zwischenergebnis	38

4.1	Build Targets einstellen	43
4.2	Build Targets fertig	43
4.3	Libopencm3 Doxygen	45
4.4	I2C Kommunikation	47
4.5	Saleae Output	55

Tabellenverzeichnis

2.1	Kalibrierregister für relative Feuchtigkeit	8
2.2	Kalibrierregister zur Temperaturermittlung	10
2.3	Reichweite abhängig der Umgebung	18

Abstract

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 Tagen 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 nur überwacht werden und die Ergebnisse zu bestimmten Zwecken benutzt werden.

Das Internet der Dinge (in Englisch *internet of things*, Kurzform **IoT**) wird dazu benutzt, um die Interaktion zwischen Menschen und vernetzten elektronischen Geräten zu vereinfachen.

1.2 Aufgabenstellung und Zielsetzung

Man möchte Daten wie Energieverbrauch eines Hauses, Bewegung eines Objekts oder die Temperatur eines Raums kennen und über lange Strecken (20Km) übertragen ohne hohe Kosten mit aber hoher Batterielaufzeit. Es gibt heutzutage Technologien wie GSM, Bluetooth oder Wifi, die diese Arbeit erledigen können. Das Problem dabei ist, dass es beim Nutzen von GSM hohe Lizenzkosten fallen müssen. Was Bluetooth und Wifi betrifft, sind ihre Reichweite sehr kurz. Dieser Ziel kann mit Hilfe der LoRa-Technologie erreicht werden, da sie diese Nachteile beseitigt.

In dieser Abschlussarbeit soll ein Prototyp gebaut werden, der mit Hilfe der LoRa-Technologie Daten an einem Server sendet und von diesem Server Daten

Empfangen. 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 mit Hilfe eines LoRa-Moduls[18] drahtlos an einem Server übertragen. Bevor die Übertragung erfolgt, muss das LoRa-Endgerät Zugang zu dem Netzwerk durch dem Server bekommen. Nach dem das LoRa-Endgerät dem Netzwerk hinzugefügt wurde, können nun Informationen zwischen dem LoRa-Endgerät und dem Netzwerk-Server bis zu einem Anwendungsserver vertauscht werden. Abbildungen 1.1 und 1.2 geben einen Überblick über den Aufbau des gesamten Systems.

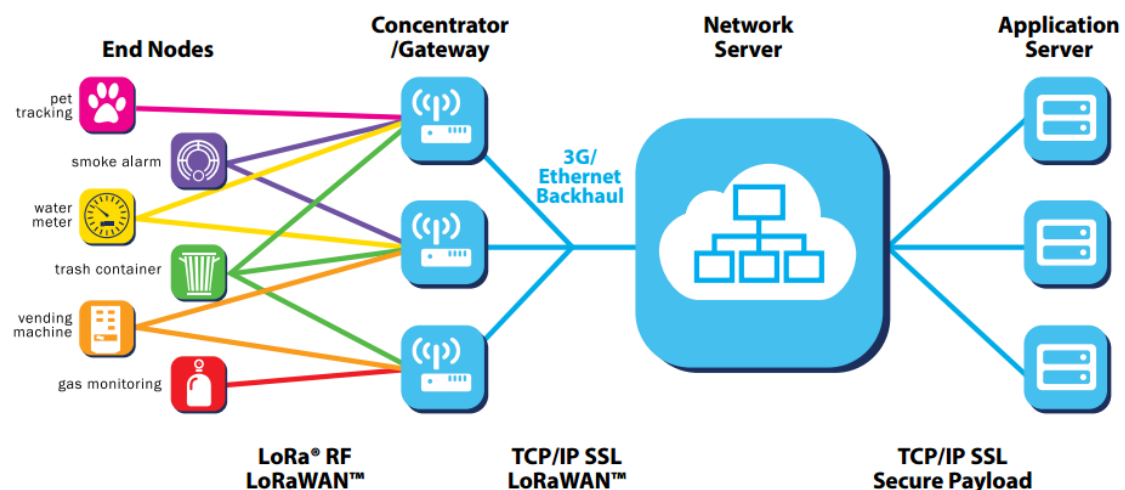


Abbildung 1.1: Allgemeine LoRaWAN Netzwerkarchitektur [2]

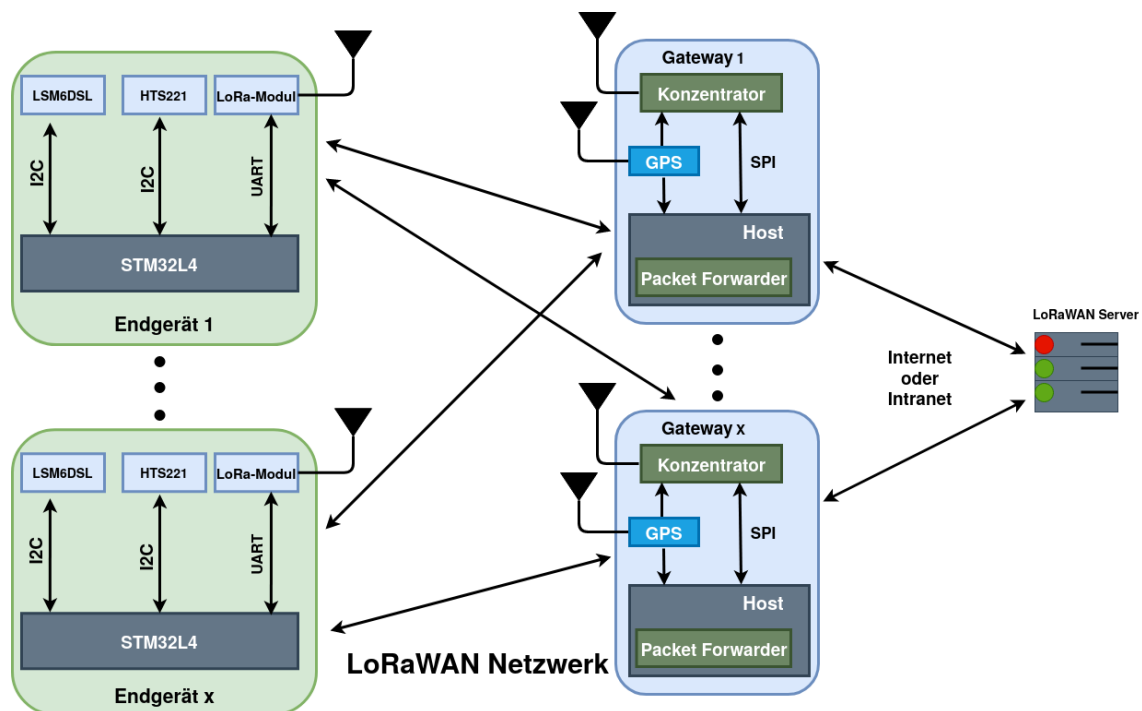


Abbildung 1.2: Labcsmart LoRaWAN Netzwerk

LoRa: ist eine abkürzung für *Long Range* und es ist eine drahtlose Technologie, das geringe Sendeleistung verbraucht wird, um kleine Datenpakete (0,3 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 einem Mikrocontroller zur Verarbeitung der Daten wie Sensordaten. Diese Daten können entweder an einem anderen LoRa-Node per Point-To-Point-Verbindung oder an einem LoRaWAN-Netzwerk versandt werden.

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

Gateway: ist ein Gerät, das aus mindestens einem Funkkonzentrator, einem Host und einer Netzverbindung zum Internet oder einen privaten Netzwerk (Ethernet, 3G, Wifi), möglicherweise einem GPS-Empfänger besteht.

LoRaWAN Server: ist ein abstrakter Computer, der die von dem Gateway empfangene RF-Pakete verarbeitet und sendet RF-Pakete als Antwort, dass das Gateway zurücksenden muss.

Application Server: ist eine Anwendung, womit der Benutzer die von den Sensoren gemessenen Daten entweder Tabellarisch oder Grafisch ansehen kann.

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

Downlink: ist die Kommunikations 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 des benutzten STM32-Nucleo Board eingegangen. Diesem Kapitel ist auch zu entnehmen, warum genau dieses Board gewählt wurde.

Als nächstes wird das LoRa-Modul. Dieses LoRa-Modul wird dazu verwendet, um die gesammelten Daten dem Server drahtlos zu übertragen. Dieser Kapitel berichtet über das Funkprotokoll, das zur Übertragung der Daten eingesetzt wurde, und wie diese Daten gerichtet 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, weil sie erlauben einem Endgerät den Netzwerk einzutreten.

Als nächstes wird die Softwareentwicklung behandelt. Hier geht es erstmal um die Entwicklungsumgebung des gesamten Projekts (Eclipse). Anschließend daran werden die angewante Bibliotheken dargestellt, ihre Installation und Nutzung erklärt. Es wurde für diese Arbeit zwei bekannte Kommunikationsschnittstellen verwendet (I2C und UART) Sie erfahren auch 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 einen kleinen 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 Multikink-Kommunikation (Bluetooth Low Energie) mit geringem Stromverbrauch Multiway-Erkennung der Umwelt ermöglicht (Siehe Abbildung 2.1).

Das STM32L4 hat einen eingebetteten ST-LINK Debugger/Programmierer, eingebettete Sensoren und viele andere Features, die in dem Datenblatt [15] erläutert zu finden sind. Genau wegen der Vielfalt an Eigenschaften wurde dieses Board ausgewählt. Man braucht kein Breadbord im Vergleich zu dem Arduino oder dem Raspberry-Pi, um die Sensoren mit den Schnittstellen (USART, SPI oder I2C) des Mikrocontrollers zu verbinden. Noch dazu eignet sich dieses Discovery Kit für ein Endgerät von STMicroelectronics, da Arduino-Verbinder vorhanden sind. Dazu ist nur das Endgerät in dieser Verbinder stecken.

Für diese Arbeit werden wir uns auf zwei Sensoren beschränken. Erstmal den HTS221[13] Temperatur- und Feuchtigkeitssensor und dann den LSM6DSK 3D-Gyroscope und 3D-Beschleunigungssensor [14]. Diese 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

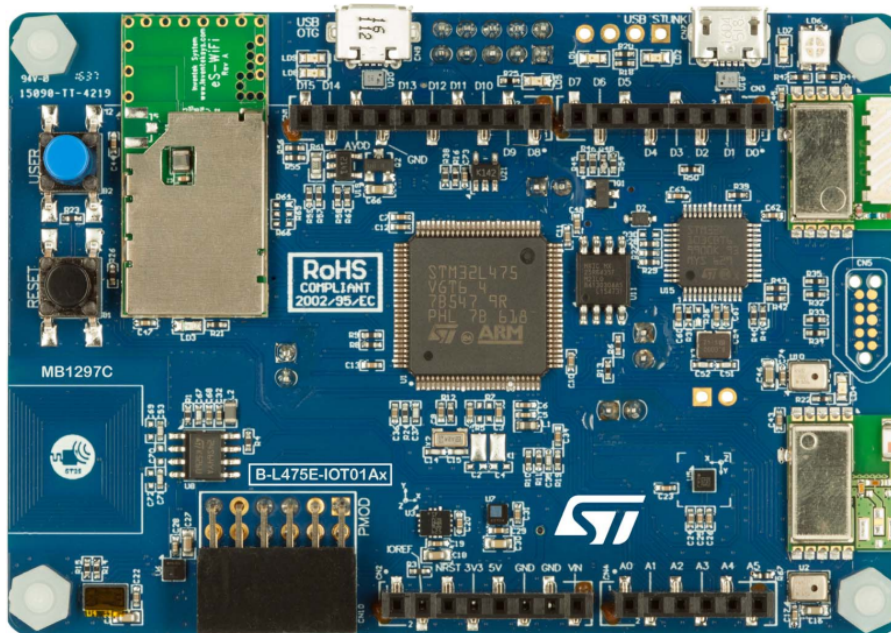


Abbildung 2.1: B-L475E-IOT01A Discovery kit [15]

Realität entsprechen. Laut dem Datenblatt ist mit einer Temperaturgenauigkeit von $\pm 0.5^{\circ}\text{C}$ und einer Feuchtigkeitsgenauigkeit von $\pm 3.5\%$ zu rechnen.

2.1.1 HTS221 Temperatursensor- und Feuchtigkeitssensor

In diesem Unterkapitel wird der HTS221 Temperatur- und Feuchtigkeitssensor beschrieben und erklärt wie die Temperatur und 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 gelesen werden. Die gespeicherte Daten sind Rohdaten, die am Ausgang von dem Analog-Digital-Converter (ADC) verfügbar sind (Siehe Abbildung 2.2). Um die Temperatur in $^{\circ}\text{C}$ und die relative Feuchtigkeit in % zu bekommen, muss man die Daten aus den Registern auslesen und mit Hilfe der Formel 2.1.1 und 2.1.1 die richtigen Werten herausfinden.

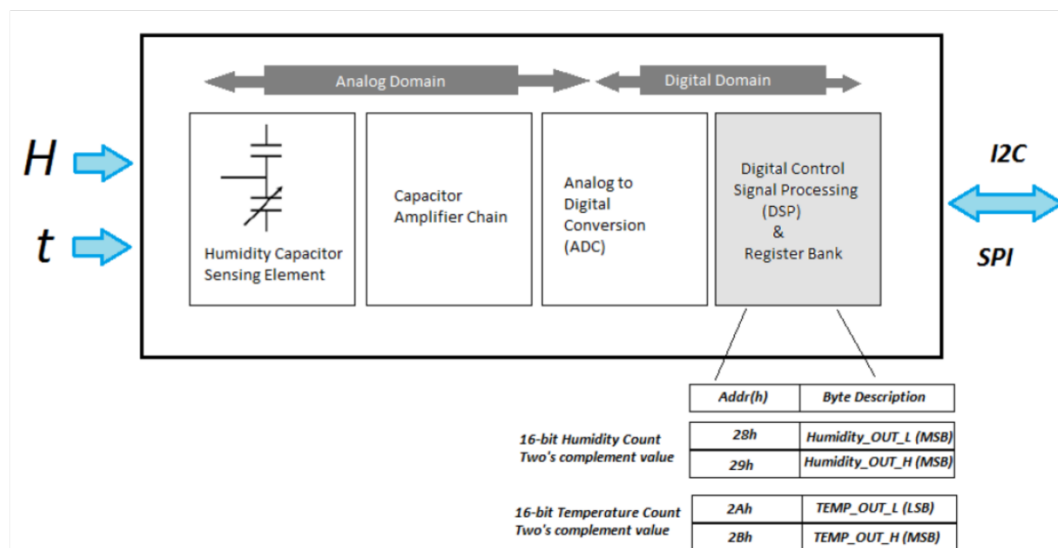


Abbildung 2.2: Humidity sensor analog-to-digital flow [13]

Feuchtigkeit ermitteln

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

- H_OUT_H (0x29) (Höchstwertige Bits)
- H_OUT_L (0x28) (Niedrigwertige Bits)

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

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

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

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

Variable	Adresse	Format ¹
<i>H0_rH_x2</i>	0x30	u(8)
<i>H1_rH_x2</i>	0x31	u(8)
<i>H0_TO_OUT_H</i>	0x36	s(16)
<i>H0_TO_OUT_L</i>	0x37	s(16)
<i>H1_TO_OUT_H</i>	0x3A	s(16)
<i>H1_TO_OUT_L</i>	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 wird. Die folgenden Schritten müssen vor der Berechnung durchgeführt werden:

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

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

Aus Abbildung 2.3 bekommt man mit Interpolation die folgende Formel [13]:

$$RH\% = \frac{((H1_rH - H0_rH) \cdot (H_T_OUT - H0_T0_OUT))}{(H1_T0_OUT - H0_T0_OUT)} + H0_rH$$

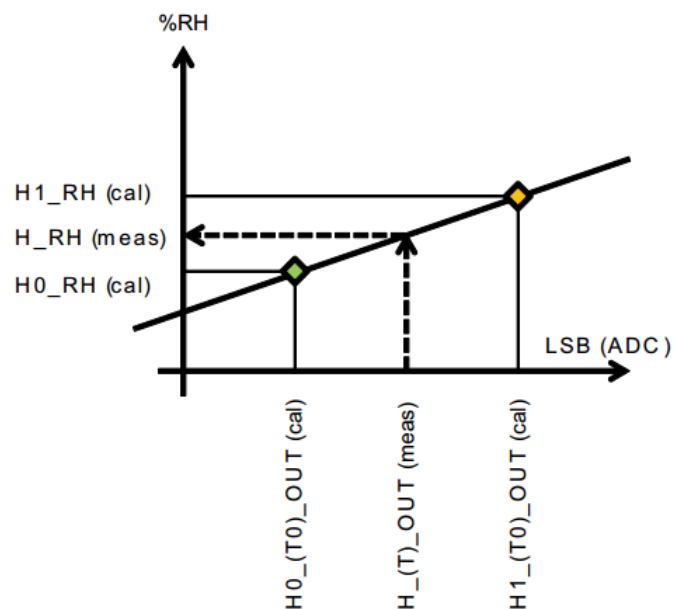


Abbildung 2.3: Linear interpolation to convert LSB to %RH [13]

Temperatur ermitteln

Der HTS221 Sensor speichert die Temperaturwerten in Rohzählungen in zwei 8-Bit-Registern:

- T_OUT_H (0x2A) (Höchstwertige Bits)
- T_OUT_L (0x2B) (Niedrigwertige Bits)

Die zwei Bytes werden verkettet, um ein Zweierkomplement dargestelltes 16-Bit Wort zu bilden. Die Polarität wird durch den höchstwertigen Bit vom T_OUT_H Register bekannt gegeben.

- Ist dieser Bit 0, die gelesene Temperatur ist positiv.
- Ist dieser Bit 1, die gelesene temperatur ist negativ. In diesem fall ist der Zweierkomplement des gesamten Wort zu bilden, um den richtigen Wert zu bekommen.

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

Die Tabelle 2.2 stellt diese Kalibrierregister dar.

Registern	Adresse	Format
<i>T0_degC_x8</i>	0x32	u(8)
<i>T1_degC_x8</i>	0x33	u(8)
<i>T1/T0msb</i>	0x35	(u2),(u2)
<i>T0_OUT_H</i>	0x3D	s(16)
<i>T0_OUT_L</i>	0x3C	s(16)
<i>T1_OUT_H</i>	0x3F	s(16)
<i>T1_OUT_L</i>	0x3E	s(16)

Tabelle 2.2: Kalibrierregister zur Temperaturermittlung

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

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

Nachdem diese Kalibrierregister gelesen wurden, kann man mittels linearer Interpolation die Temperatur in °C berechnen.

Abbildung 2.4 zeigt der Graph woraus die lineare Interpolation stammt. Die folgende Formel wurde sogar daraus hergeleitet.

$$T[c] = \frac{((T1_degC - T0_degC) \cdot (T_OUT - T0_OUT))}{(T1_OUT - T0_OUT)} + T0_degC$$

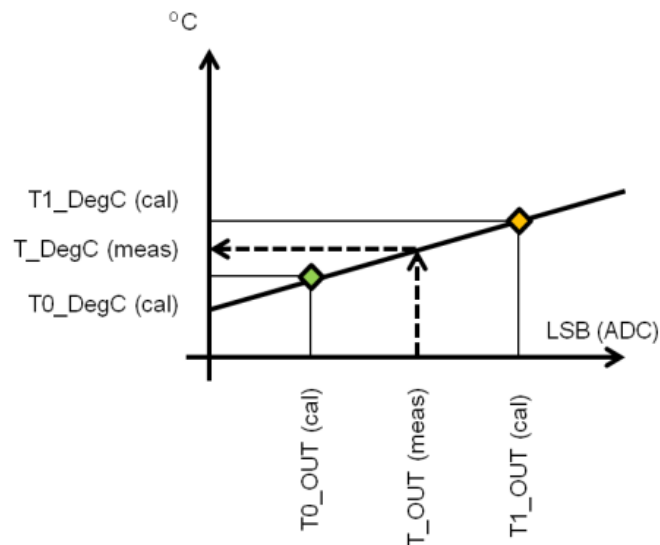


Abbildung 2.4: Linear interpolation to convert LSB to °C [13]

Da die Kalibrierwerten 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 des 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 $\pm 2 \mid \pm 4 \mid \pm 8 \mid \pm 16g$ (g ist gleich 9,81m/s) und einen Winkelgeschwindigkeitsbereich von $\pm 125 \mid \pm 250 \mid \pm 500 \mid \pm 1000 \mid \pm 2000dps$ (Degrees per second).

Das extrem geringe Größe und das geringe Gewicht des SMD-Packets machen den LSM6DSL zu einer idealen Wahl für tragbare Anwendungen wie Smart-

phones, 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 der 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 der Gyroskop in einem Ruhestand gesetzt werden.

Sobald das Gerät mit Strom versorgt wird, werden die Kalibrierkoeffizienten vom eingebetteten Flash-Speicher zu den internen Registern. Dieser Vorgang dauert ungefähr 15 Millisekunden. Nach dieser Zeit fallen der Beschleunigungsmesser und der Gyroskop in den Power-Down Modus. Durch die *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 demnächst) ausgelöst, wenn der entsprechende Byte vom Beschleunigungsmesser bzw. vom Gyroskop in dem *INT1_CTRL*-Register geschrieben wurde. Das Vorhandensein der Daten kann nun mit Hilfe 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 Gyroskop-Ausgang ein neuer Datensatz verfügbar ist.

Die Abbildung 2.5 ist das Flußdiagramm zur Ermittlung der Achsen- und Winkelveränderungen des Beschleunigungsmessers und des Gyroskops.

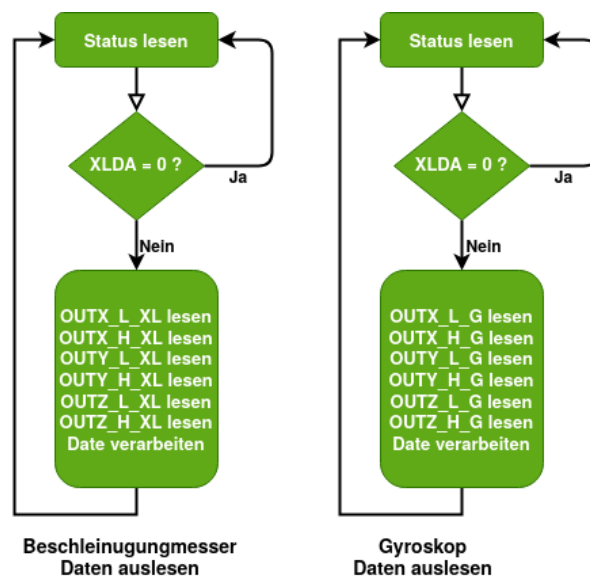


Abbildung 2.5: Flußdiagramm zur Datenermittlung

Wie oben schon erläutert, kann das Gerät so eingestellt werden, dass ein neuer Satz von Messdaten durch ein Signal erkennbar wird. Das *XLDA*-Bit von dem *STATUS_REG*-Register wird auf 1 gesetzt, wenn Daten aus dem Beschleunigungsmesser zum Lesen verfügbar sind. 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 *GDA*-Bit auf 1 gesetzt, wenn die Daten zum lesen verfügbar sind. Das Signal kann durch den *INT1*-Pin angesteuert werden, indem das *INT1_DRDY_G*-Bit vom *INT1_CTRL*-Register auf 1 gesetzt wird. Die gemessenen Beschleunigungsdaten werden in *OUTX_H_XL*-, *OUTX_L_XL*-, *OUTY_H_XL*-, *OUTY_L_XL*-, *OUTZ_H_XL*-, *OUTZ_L_XL*-Register gesendet. Die gemessenen Winkelgeschwindigkeitsdaten werden dagegen in *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 sind.

Mit dem LSM6DSL kann man den Inhalt des unteren und oberen Teils der Ausgangsdatenregister vertauschen, 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 Standardmäßig) oder auf 1 (für Big-

Endian) Big-Endian bedeutet, dass der höchstwertige Byte des Datensatzes in der niedrigsten Speicherstelle gespeichert wird. Little-Endian bedeutet, dass der niedrigstwertige 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 Protokoll an einem Server versandt werden. Um diese Daten drahtlos und auf eine lange Strecke zu übertragen, haben wir uns auf das LoRaWAN-Protokoll entschieden. Die Gründe warum genau dieses Protokoll 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.

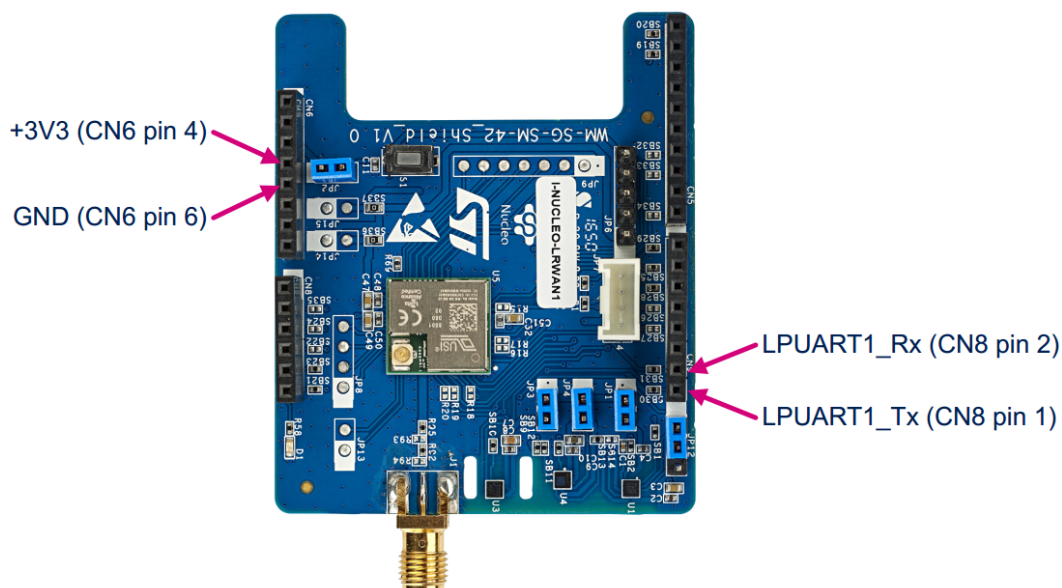


Abbildung 2.6: I-Nucleo-LRWAN1 [17]

Abbildung 2.6 zeigt das Endgerät, das zur Datenübertragung verwendet ist. 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 den USI® LoRaWAN™ Technologiemodul für kostengünstiges und stromsparendes Weitverkehrsnetz (LPWAN), welches mit einem eingebetteten Stapel von AT-Befehlen (AT-Befehl beschreiben) mitgeliefert wird. Dieses Board wurde ausgewählt, weil es durch ein externes Board wie das Nucleo-L053 oder user B-L475E-IOT01A Discovery kit 2.1 über mehrere Schnittstellen wie 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 relative Feuchtigkeits- und Temperatursensor (HTS221)
- ST Drucksensor (LPS22HB)

Im Vergleich zu anderen Endgeräten, worauf keine Sensoren vorhanden sind, braucht man keine zusätzliche Sensoren kaufen. Die Kommunikation mit einem anderen Mikrocontroller erfolgt einfach durch UART, man braucht nicht auf das integrierte Radio-Modul zum Senden oder Empfangen von Daten und Befehlen zu kümmern. 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. Das SX1272-Chip ist das eigentliche LoRa-Radio-Modul, das die Daten oder die AT-Befehle per Funk durch die Antenne an entweder einem Gateway (Router) oder einem anderen Endgeräten sendet.

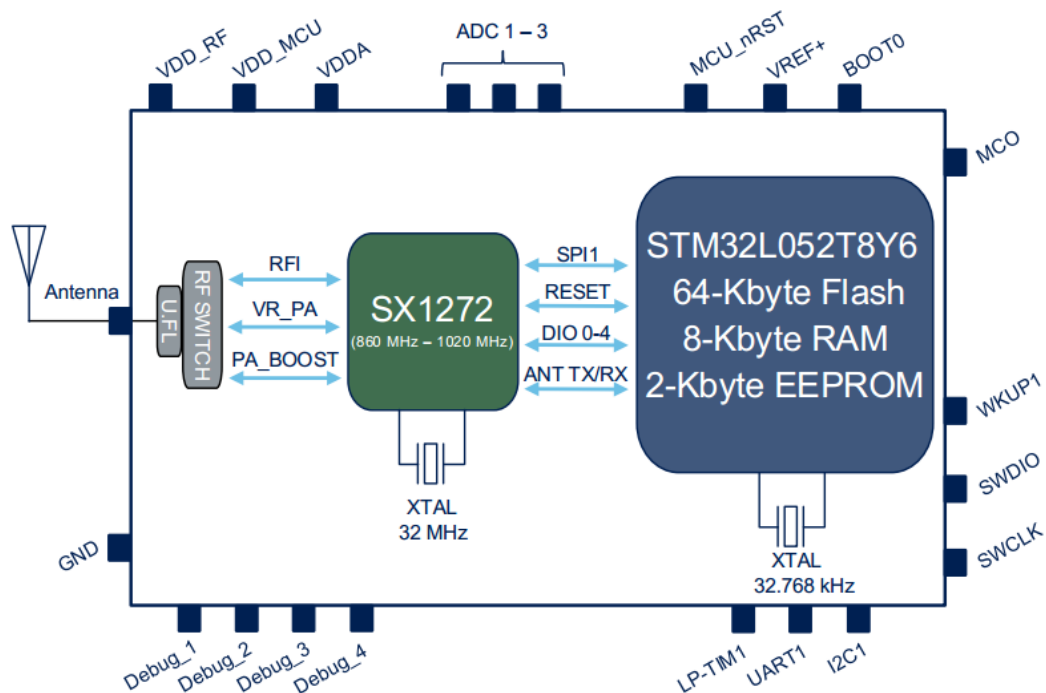


Abbildung 2.7: I-Nucleo-LRWAN1 Architektur [17]

Das I-Nucleo-LRWAN1 wird mit Hilfe 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)

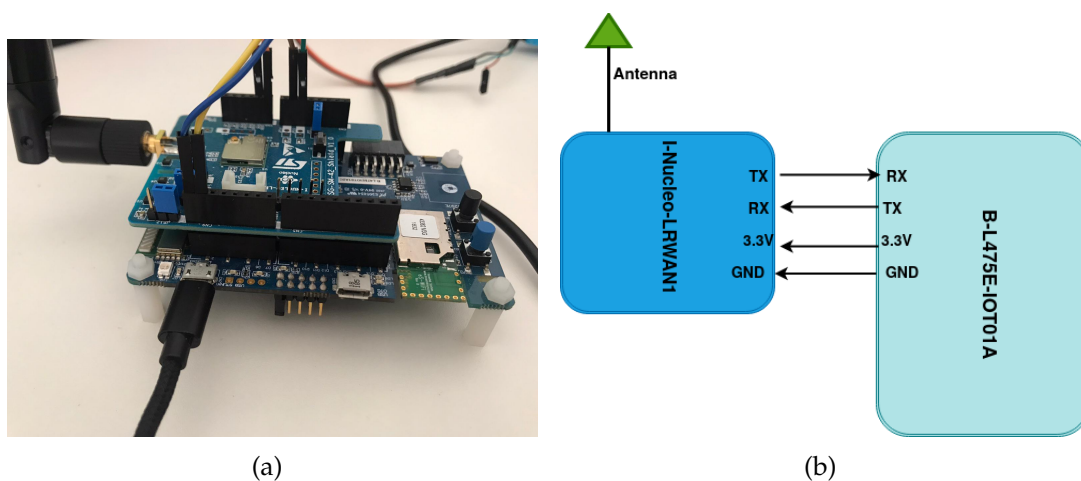


Abbildung 2.8: LABCSMART LoRa End-Node physisches Aussehen (a) und Verbindung (b)

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 Str-om 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 erhaltene Sensordaten mit Hilfe des LoRa-WAN-Protokol gesendet werden.

2.2.1 LoRa und LoRaWAN-Protokol

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

LoRa: Die Physikalische Schicht

Eine einzige Technologie kann nicht alle Anwendungen des IoT decken. Technologien wie Wifi und Bluetooth Low Energie (BLE) sind wiet verbreitete Standart und decken die Kommunikation persönlicher Geräte recht gut. Die Mobilfunktechnologie passt hervorragend zur Anwendung, die einem hohen Datendurchsatz benötigen.

Diese Technologien sind zwar gut aber haben es bestehen einige Nachteile, wie dem hohen Energieverbrauch und die Strecke von einem Sender zu einem Empfänger ist relativ kurz. LoRa bieten Lösungen zu diesen Nachteilen an, nämlich eine mehrjährige Batterielebensdauer, die Übertragung von kleinen Datenmengen über große Entfernungen. LoRa ist die physikalishce Schicht oder die verwendete drahtlose Modulation, um eine lange Bereichskommunikationsverbindung zu schaffen.

Viele ältere drahtlose Systeme verwenden die Frequenzumtastungenin (Englisch *Frequency Shifting Keying*, Kurzform **FSK**) als physikalishce Schicht, weil es eine sehr effiziente Modulkation zur Erzielung geringer Leistung ist. LoRa basiert auf die Chirp-Spreizspektrum-Modulation (Englisch *Schirp Spread Spectrum modulation*). Diese Modulation behaält die gleiche Eigenschaft der geringer Leistung wie FSK-Modulation bei und erhöht aber deutlich die Kommunikationsreichweite. Das Chirp-Spreizspektrum wird seit Jahrzehnten auf Grund seiner Kommunikationsreichweite und seiner Robustheit gegenüber Störungen in der Militär-und Weltraumkommunikation eingesetzt. LoRa ist derzeit die erste kostengünstige Implementierung für dien 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 existiert konkurrierende Technologien zu LoRa wie Narrowband IoT (NB-IoT) und Sigfox. Das LoRa kann Video- und Audio-Nachrichten nicht übertragen, man kann nur sehr kleine Datenpakete wie Sensordaten übertragen. 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 Spreizfaktor (in Englisch *Spreading Factor*, Kurzform **SF**) bestimmt. Durch die Steuerung des Signals wird seine Reichweite zum nachteil der Bitrate erhöht, da es über einen längeren Zeitraum übertragen wird. Das verbraucht auch zu viel Energie und spielt nachteilhaft zur Autonomie des Geräts. Das heisst, je größer der Spreizfaktor ist, umso kleiner wird die Bitrate und die Reichweite wird dagegen größer. Ein LoRaWAN-Netzwerk bietet sechs mögliche Spreizfaktoren (**SF7, SF8, SF9, SF10, SF11, SF12**).

Die Reichweite zwischen LoRa-Sender und -Empfänger hängt 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 abhängig der Umgebung.

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

Tabelle 2.3: Reichweite abhängig der Umgebung

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

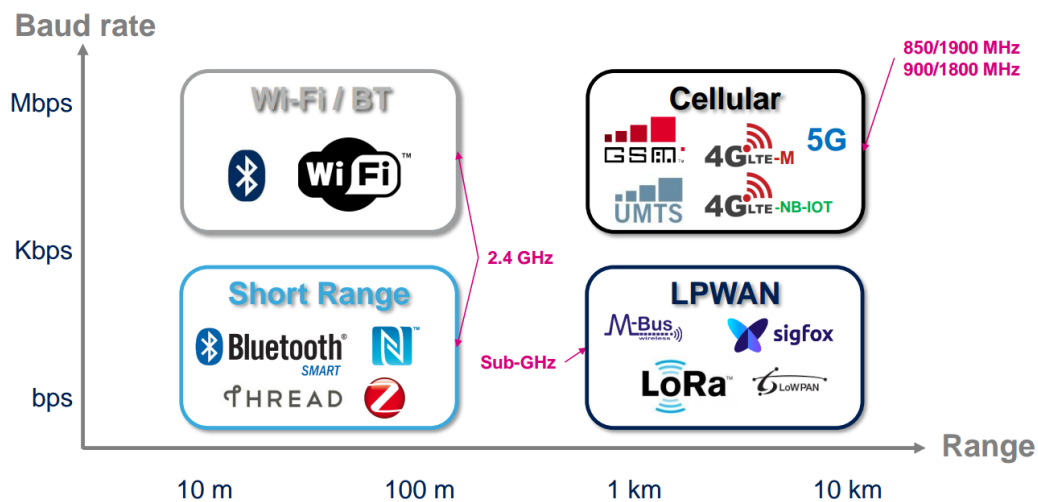


Abbildung 2.9: Vergleich zwischen LoRa und andere IoT Kommunikationstechnologien [2]

Abbildung 2.9 ist zu entnehmen, dass LoRa im Vergleich zu andere Technologien wie Wifi oder 4G eine kleine Baudrate hat. Aber seine Reichweite ist deutlich größer als weit bekannte Technologien wie Bluetooth oder Wifi.

Die LoRa-Technologie kann man in vielen Gebiete Einsetzen. Die folgende Auflistung gibt einen groben Überblick auf 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, die die Fernkommunikationsverbindung ermöglicht. Das Protokoll und die Netzwerkarchitektur haben den größten Einfluss auf die Bestimmung der Batterielebensdauer eines Endgeräts, die Netzwerkkapazität, die Servicequalität, die Sicherheit und die Vielzahl der vom Netzwerk Bereitgestellten Anwendungen.

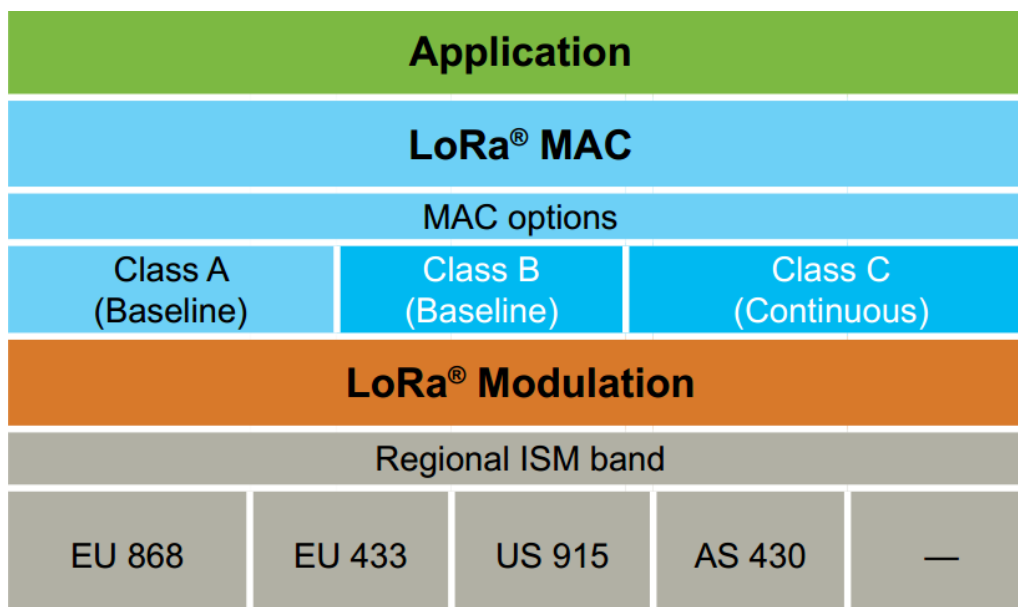


Abbildung 2.10: Klassen von LoRaWAN [2]

Wie es auf Abbildung 2.10 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 Dateübertragung 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 Anwendung der LoRa-Technologie.

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

Klasse A (All end-devices) : Ein Endgerät der Klasse A ermöglicht eine bidirektionelle Kommunikation, wobei nach jedem Uplink eines Endgeräts für kurze Zeit zwei kurzen Downlink-Empfangsfenstern folgen. Diese Empfangsfenster 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 in dem Server gespeichert. Verglichen zu den anderen Klassen, sind Endgeräte der Klasse A die niedrigste Leistungsfresser.

Nachdem die zwei Downlink-Empfangsfenster geschlossen sind, kann das Gateway keine weitere Downlinks mehr senden. Die nächsten Downlinks werden erst berücksichtigt, wenn das Endgerät ein Uplink gesendet hat. Das unten stehende Bild zieht dieses Verhalten recht gut.

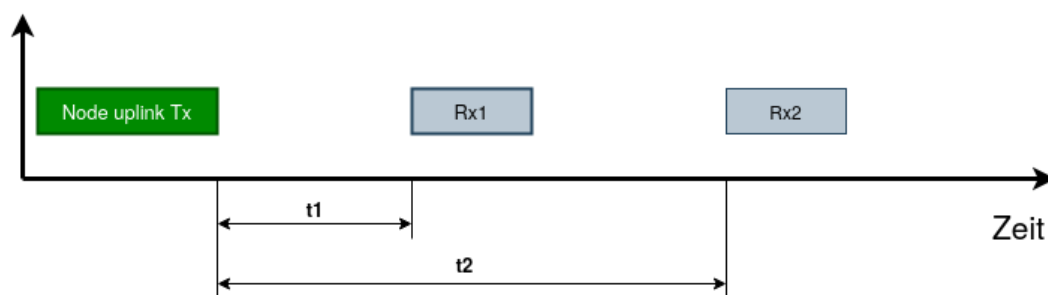


Abbildung 2.11: Klasse A

Klasse B (Beacon) : Zusätzlich zu den zufälligen Empfangsfenstern der Klasse A, Geräte der Klasse B öffnen zusätzliche Empfangsfenster zu geplanten 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.

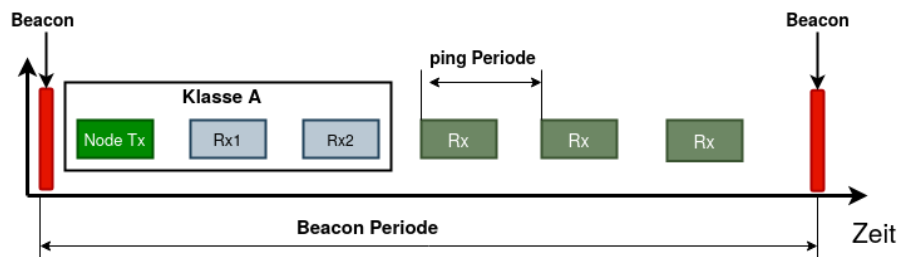


Abbildung 2.12: Klasse B

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

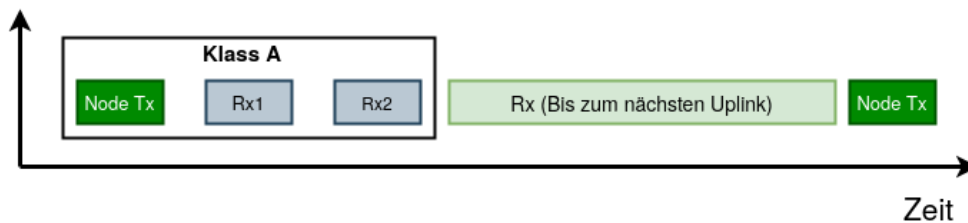


Abbildung 2.13: Klasse C

Im Rahmen dieser Thesis wird nur die Klasse A berücksichtigt, 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 ein Dritter ohne Zugriffsrechte auf das Netzwerk die Informationen lesen kann. 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* Kurzform **NwkSKey**) stellt die Authentizität der Geräte im Netzwerk sicher. Der zweite ist der Anwendungssitzungsschlüssel (Englisch *Application Session Key* Kurzform **AppSKey**). Der NwkSKey wird von dem Endgerät und dem Server benutzt, um den Nachricht-

tenintegritätscode (Englisch *Message Integrity Code* Kurzform **MIC**) zu berechnen und die Integrität aller Daten zu prüfen. Es wird weiterhin verwendet, um das Nutzdatenfeld von MAC-Daten zu verschlüsseln und zu entschlüsseln.

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

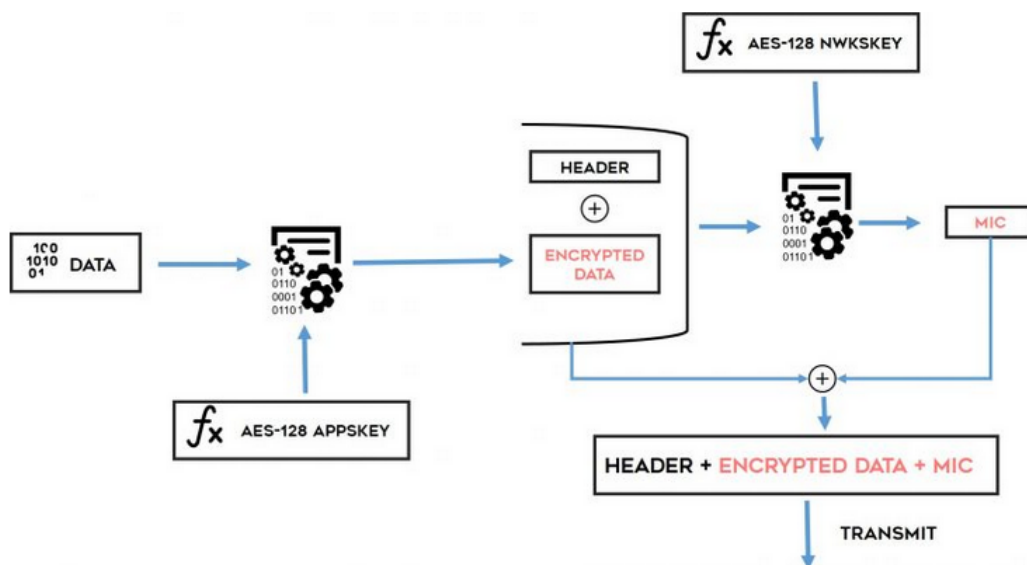


Abbildung 2.14: LoRaWAN-Nachricht Verschlüsselung [3]

Laut Abbildung 2.14 werden die zu sendende Daten erst mit dem AppSKey verschlüsselt. Ein Header, der andere Adressen des Endgerät enthält, wird an den verschlüsselten Daten hinzugefügt. Nach dieser Verknüpfung wird der 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 mit Hilfe des MICs geprüft werden. Die Daten werden nur berücksichtigt, wenn der 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 Endgeräts kann auf zwei Arten erfolgen, entweder per Over-The-Air-Aktivierung (Englisch *Over-The-Air Activation* kurzform **OTAA**), oder per Aktivierung durch Personalisierung (Englisch *Activation By Personalization* kurzform **ABP**), wobei die zwei Schritte der Personalisierung und Aktivierung in einem Schritt erfolgt.

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 auch durchgeführt, wenn es vorkommen würde, dass ein Endgerät die Sitzungsinformationen verloren hat. Bevor ein Endgerät das Join-Verfahren startet, muss es folgende Informationen haben: Eine global eindeutige Endgeräteerkennung (**DevEUI**), eine Anwendungskennung (**AppEUI**) und ein AES-128-Schlüssel (**AppKey**).

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

DevEUI: ist ein 8-Byte-Wert mit hexadezimaler Codierung und bezeichnet die eindeutige Kennung eines Endgeräts. Manche LoRa-Radiomodule beinhalten eine DevEUI vom Hersteller her. Wenn nicht schon vorhanden, diese kann vom Anwendungsanbieter gesetzt werden.

AppKey: ist ein 16-Byte-Wert in Hexadezimalformat. Wenn ein Endgerät mit OTAA das Netzwerk beitrifft, 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 es eine Join-Abfrage (**Join request**) am Server schicken. Der Server antwortet mit einer Join-Zustimmung (**Join accept**), wenn das Endgerät dem Netzwerk beitreten darf. Die Join-Accept-Nachricht wird wie ein normales Downlink gesendet, aber benutzt zwei unterschiedliche Verzögerungen verglichen mit *RECEIVE_DELAY1* und *RECEIVE_DELAY2* der im Abschnitt 2.2.1 beschriebenen Verzögerungen. 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 ein 3-Byte-Anwendung Nonce (*AppNonce*), eine Netzwerkennung (*NetID*), eine Endgerätadresse (*DevAddr*), eine Verzögerung zwischen TX und RX (*RxDelay*) und eine optionale Liste der Kanalfrequenz (*CFList*). Die *DevAddr* und das *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 oder eine Form einer eindeutige ID, die vom Netzwekserver bereitgestellt wird und vom Endgerät verwendet, 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) und wird wie folgt bestimmt [2]:

$$\begin{aligned} \text{NwkSKey} &= \text{aes128_encrypt}(\text{AppKey}, 0x01 \parallel \text{AppNonce} \parallel \text{NetID} \parallel \text{Dev-} \\ &\quad \text{Nonce} \parallel \text{pad}) \quad \text{NwkSKey} = \text{aes128_encrypt}(\text{AppKey}, 0x02 \parallel \text{AppNonce} \parallel \text{NetID} \parallel \text{DevNonce} \parallel \text{pad}) \end{aligned}$$

Nun können Endgeräte, die dem Netzwerk beigetreten sind, Informationen mit dem Netzwerkeserver austauschen (Uplinks und Downlinks). Abbildung 2.15 beschreibt das oben erläuterte Verfahren.

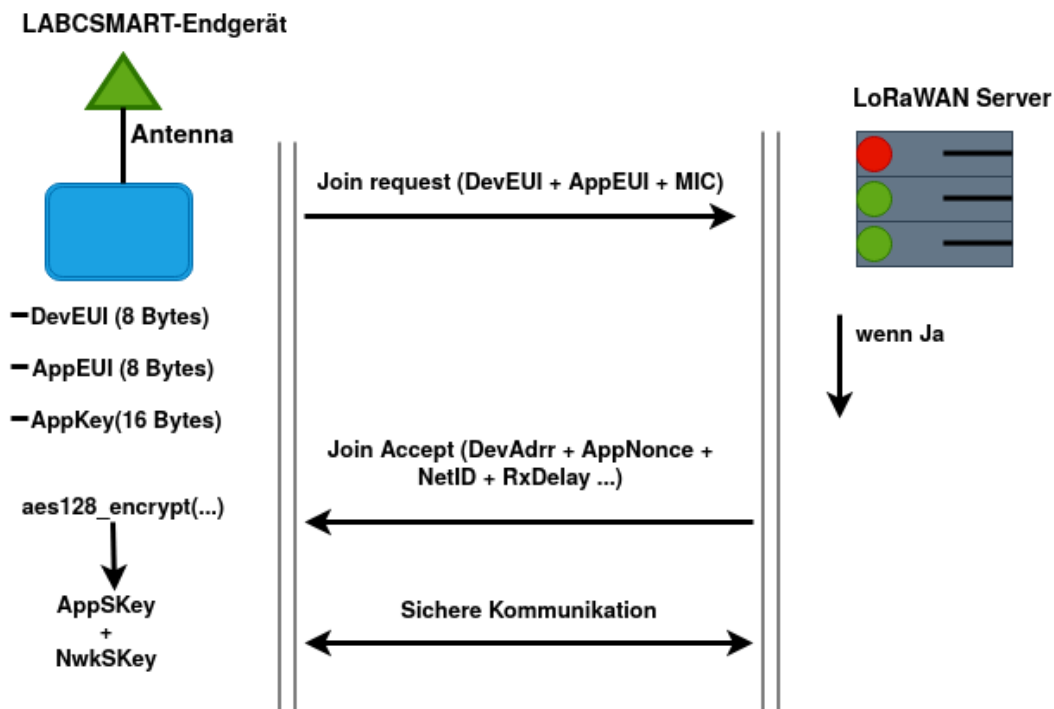


Abbildung 2.15: Join-Request Verfahren

Aktivierung durch ABP

Bei dieser Aktivierungsart, braucht 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 werde. Jedes Endgerät soll einen eindeutigen Satz von *AppSKey* und *NwkSKey*. 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 ABD genug. Wenn es um ein größeres Netzwerk geht, ist 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 Informationen (Appkey, NwkKey und mehr) dem Endgerät zugewiesen werden als auch wie die Daten am Server gesendet werden. Im Kapitel 2.2 wurde das Wort "AT-Befehl" 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, das bewirkt, das andere Kommandos nur einmal ausgeführt werden. Hier ist es benutzt, um das i-nucleo-lrwan1 einzustellen. Da das LoRa-Radiomodul sich nicht selbst einstellen kann, ist auch die Aufgabe des B-L475E-IOT01A 2.1 die Einstellung durchzuführen. Das i-nucleo-lrwan1 verfügt über eine UART-Schnittstelle (Siehe 2.6), um mit dem B-L475E-IOT01A zu kommunizieren. Diese UART-Schnittstelle hat folgende Konsolekonfiguration:

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

Die Syntax des Kommandos ist wie folgt:

- Allgemeine Kommandos:
 - **AT**: Prüft ob die UART Schnittstelle benutzbar ist
 - **ATE [=<enable>]**: Aktivieren oder Deaktivieren des lokalen Echos
 - **ATZ**: Modul zurücksetzen
 - **AT+VERB [=<enable>]**: Ausfö¼hrliche 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 Zeichen [] bedeuten, dass das Parameter optional ist. Mit Parametern ist ein AT-Befehl wie ein Set-Befehl, und ohne ist es ein Get-Befehl.

Da wir nun wissen, wie diese Kommandos zu nutzen sind, können wir Beispiele für OTAA und ABP 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 TX-Datenrate (LoRa SF12/125KHz 250 Bit/s)
- **AT+RX1DT=1000:** Setzt die Verzögerung des ersten Empfangsfenster (in ms)
- **AT+RX2DT=2000:** Setzt die Verzögerung des zweiten Empfangsfensters (in ms)
- **AT+JRX1DT=5000:** Setzt die Verzögerung des zweiten Empfangsfensters (in ms)
- **AT+JRX2DT=6000:** Setzt die Verzögerung des zweiten Empfangsfensters (in ms)
- **AT+RF=14,8671000000,12,0,1:** Konfiguriert das LoRa-Radiomodule. Ausgangsleistung: 14dBm, Frequenz: 867.1MHz, Spreading factor: SF12, Bandbreite: 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 Geräteadresse
- **AT+NSK=1122334455663EAB546829CB361CAB7D**: Setzt den NwkS-Key
- **AT+ASK=887766554433BCFACDE52476CA4598BA**: Setzt den AppSKey
- **AT+JOIN=0**: Sendet eine Join-Abfrage als ABP

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

Im AT-Befehlsreferenzhabdbuch 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 einem LoRaWAN-Server zu senden und Downlink vom Server zu bekommen. Aber was ist das 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, dass es billiger als fertige Gateway ist, sich ein eigenes zu bauen. Noch dazu steckt eine wissenschaftliche Idee dahinter. Zu wissen, wie ein Gateway gebaut wird, welche Komponenten und welche Software im Spiel kommen.

Ein Gateway ist ein Gerät, das aus mindestens einem Konzentrador, einem Host und einer Netzverbindung zum Internet oder einem privaten Netzwerk (Ethernet, 3G, Wifi), möglicherweise einem GPS-Empfänger besteht. Der Konzentrador ist ein Board, das Funkpakete senden und empfangen kann. Ein Konzentrador 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 ausgeführt wird. Der Host steuert den Konzentrador 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, aber kann auch Daten vom Gateway 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 gesendete Daten werden von allen Gateways des Netzwerks bekommen. Das Gateway leitet das Datenpaket an dem 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 den Netzwerkserver eine Antwort vom Anwendungsserver, bestimmt der Netzwerkserver welche Gateway benutzt wird, um dem Endgerät die Antwort zu senden (Downlink).

Der Konzentrator kann Funkpakete zwar empfangen und senden, er ist aber nur eine elektronische Komponente und braucht eigentlich 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 weiterleitet und sendet die vom Servergesendete Funkpakete weiter. Der Packet-Forwarder kann auch ein netzwerkweites synchrones GPS-Signal senden, das zur Koordination aller Endgeräte des Netzwerks verwendet wird.

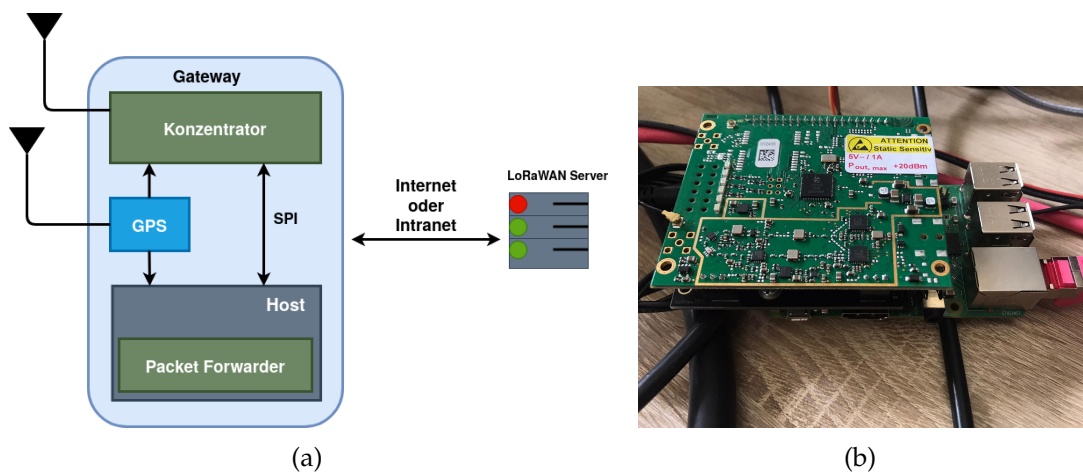


Abbildung 3.1: LABCSMART LoRaWAN Gateway

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 einem einzigen Modul. Das heisst, der Konzentrator, das Gateway und der Server zusammen

in einem Block eingebaut sind. Der Raspberry-Pi ist gleichzeitig der Host und der Server. **(Referenzen zur Installation des Paket-forwarder im Ausblick)**

3.2 Einstellung des LoRaWAN-Servers

An dieser Stelle ist die Arbeit fast fertig, da unsere Anwendung theoretisch in der Lage ist, Uplinks an dem Gateway zu schicken. Nun interessieren 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, dazu ist ein Server. 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 Anwendungssever. Man kann damit alle Ereignisse, und alle Daten, die entweder vom Endgerät oder vom Packet-Forwarder kommt. Der Server wurde in 79% in der Programmiersprache Erlang [10] geschrieben.

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 Gateway verbunden werden (Nur ein in unserem Fall). Der Server bekommt alle Uplinks, die von den Gateways weitergeleitet werden, betrachtet nicht welches Endgerät welchem Netzwerk gehört.

Edit gateway #BBDDAAFFFF000000

General Status

MAC * BBDDAAFFFF000000

Area labcsmart-Gateway-area

TX Chain * 0

Antenna Gain (dBi) 2

Description Labcsmart lora gateway

Location *

Abbildung 3.2: Einstellung des Gateways

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

- Eine Netzwerkennung, um die Egerät-Adresse (DevAddr) neu verbundener Endgeräte zu Erstellen.
- LoRaWAN-Regionparameter, einschließlich zusätzliche Frequenzen

Edit network #labcsmart-network

General ADR Channels

Name * labcsmart-network

NetID * 000000

Region * EU 863-870MHz

Coding Rate * 4/5

RX1 Join Delay (s) * 5

RX2 Join Delay (s) * 6

RX1 Delay (s) * 1

RX2 Delay (s) * 2

Gateway Power (dBm) * 14

✓ Save changes

(a)

Edit network #labcsmart-network

General ADR Channels

Max EIRP (dBm) * 16

Max Power * Max

Min Power * Max - 14 dB

Max Data Rate * SF7 125 kHz (5470 bit/s)

Initial Duty Cycle 1 (100%)

Initial RX1 DR Offset * 0

Initial RX2 DR * SF12 125 kHz (250 bit/s)

Initial RX2 Freq (MHz) * 869.525

✓ Save changes

(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, ADR (Adaptive Data Rate) durchzuführen oder den Batteriestatus bereitzustellen.
- Die Anwendung

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 dementsprechend eingestellt werden.

Edit profile #labcsmart_test

General
ADR

Name *

labcsmart_test

Group *

labcsmart-Group

Application *

labcsmart-mote

App Identifier

Join

Allowed with old Nonce

FCnt Check

Reset on zero

TX Window

Auto

✓ Save changes

Edit profile #labcsmart_test

General
ADR

ADR Mode

Auto-Adjust

Set Power

14 dBm

Set Data Rate

SF12 125 kHz (250 bit/s)

Max Data Rate

SF7 125 kHz (5470 bit/s)

Set Channels

0-2

Set Duty Cycle

1 (100%)

Set RX1 DR Offset

1

Set RX2 DR

SF12 125 kHz (250 bit/s)

Set RX2 Freq (MHz)

869.525

Request Status?

☒ true
 ✕

✓ Save changes

(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.

Edit group #labcsmart-Group

The screenshot shows a web form titled "Edit group #labcsmart-Group". The form has the following fields and values:

- Name ***: labcsmart-Group
- Network ***: labcsmart-network
- SubID**: e.g. 0:3
- Administrators**: admin x
- Slack Channel**: (empty)
- Can Join?**: true x

At the bottom of the form is a blue button with a checkmark icon and the text "Save changes".

Abbildung 3.5: Einstellung der Gruppe

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

ABP Verbindung

Geräte die mit ABP verbunden werden sollen, brauchen wie 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.

Edit node #12345678

General ADR Status

DevAddr * 12345678

Profile * labcsmart_test

App Arguments

Location

NwkSKey * 1122334455663EAB546829CB361CAB7D

AppSKey * 887766554433BCFACDE52476CA4598BA

Description Labcsmart lora mote ABP

FCnt Up 2

FCnt Down * 2

Last Reset 2020-06-15 12:14:28

Last RX 2020-06-15 12:16:57

Device

Gateways **MAC**
BBD0AFFFFF000000

Downlinks

☒ Save changes

(a)

Edit node #12345678

General ADR Status

ADR Support OFF

Set Power 14 dBm

Set Data Rate SF12 125 kHz (250 bit/s)

Set Channels 0-2

Used Channels 0-2

ADR Failed

Used Duty Cycle 1 (100%)

Used RX1 DR Offset 0

Used RX2 DR SF12 125 kHz (250 bit/s)

Used RX2 Freq (MHz) 869.525

RX Change Failed

RX

Table has no columns.

RX Quality

Table has no columns.

☒ Save changes

(b)

Abbildung 3.6: ABP Verbindung

OTAA Verbindung

Hier befinden sich Geräte, die dem LoRaWAN-Netzwerk mit Hilfe 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 (Siehe 2.2.2 für die Erklärung).

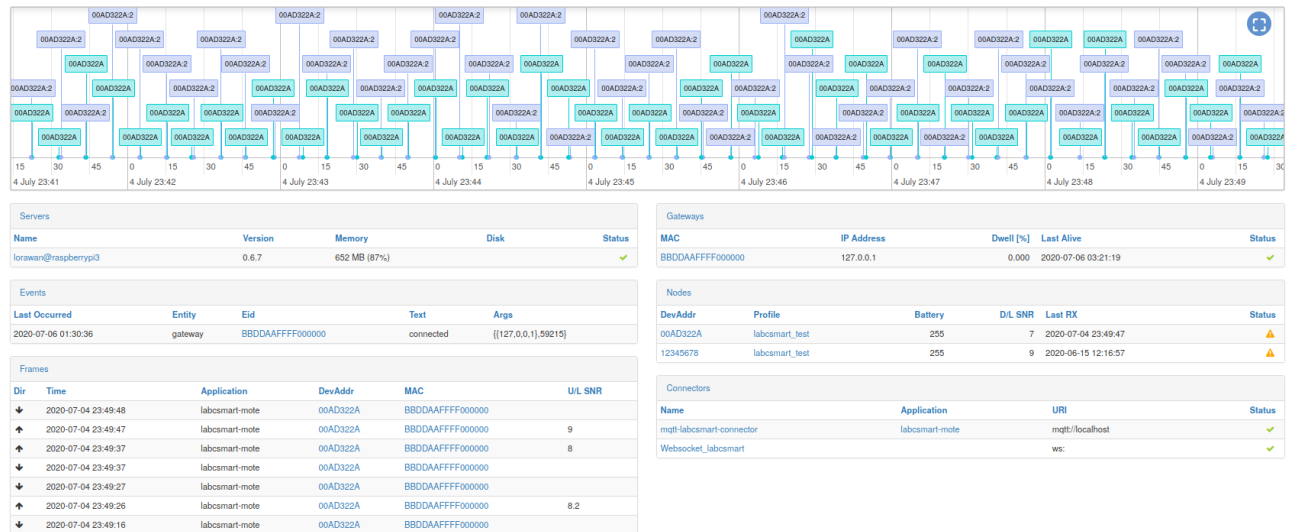
Edit device #E24F43FFFE44C432

DevEUI *	E24F43FFFE44C432						
Profile *	labcsmart_test						
App Arguments							
AppEUI	ABC123ADF135CBD8						
AppKey *	00112233445566778899AABBCCDDEEFF						
Description	Labcsmart lora mote OTAA						
Last Joins	<table><thead><tr><th>Time</th></tr></thead><tbody><tr><td>2020-07-04 23:11:13</td></tr><tr><td>2020-06-24 23:08:30</td></tr><tr><td>2020-06-24 22:59:47</td></tr><tr><td>2020-06-24 22:58:32</td></tr><tr><td>2020-06-24 22:56:44</td></tr></tbody></table>	Time	2020-07-04 23:11:13	2020-06-24 23:08:30	2020-06-24 22:59:47	2020-06-24 22:58:32	2020-06-24 22:56:44
Time							
2020-07-04 23:11:13							
2020-06-24 23:08:30							
2020-06-24 22:59:47							
2020-06-24 22:58:32							
2020-06-24 22:56:44							
Node	00AD322A						
<button>✓ Save changes</button>							

Abbildung 3.7: Einstellung OTAA

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

Dashboard



(a)

Frames

Dir	Time	Application	Location	DevAddr	MAC	U/I RSSI	U/I SNR	FCnt	Confirm	Port	Data
↓	2020-07-04 23:49:48	labcsmart-mote		00AD322A	BBDDAAFFFF000000	-69	9	219	✗	2	48454C4CF204D424F4E474F
↑	2020-07-04 23:49:47	labcsmart-mote		00AD322A	BBDDAAFFFF000000	-68	8	218	✓	2	48454C4CF204D424F4E474F
↑	2020-07-04 23:49:37	labcsmart-mote		00AD322A	BBDDAAFFFF000000			218	✓		
↓	2020-07-04 23:49:37	labcsmart-mote		00AD322A	BBDDAAFFFF000000			217	✗		
↓	2020-07-04 23:49:27	labcsmart-mote		00AD322A	BBDDAAFFFF000000			217	✗		
↑	2020-07-04 23:49:26	labcsmart-mote		00AD322A	BBDDAAFFFF000000	-70	8.2	217	✓	2	48454C4CF204D424F4E474F
↑	2020-07-04 23:49:16	labcsmart-mote		00AD322A	BBDDAAFFFF000000	-71	8.2	216	✗		
↑	2020-07-04 23:49:16	labcsmart-mote		00AD322A	BBDDAAFFFF000000	-71	9.5	215	✓	2	48454C4CF204D424F4E474F
↑	2020-07-04 23:49:05	labcsmart-mote		00AD322A	BBDDAAFFFF000000			215	✓		

(b)

Abbildung 3.8: Zwischenergebnis

Grüne Ereignisse im Abbildung 3.8(a) bezeichnen Downlinks, während Ereignisse in lila sind Uplinks. Die Spalte U/I RSSI des Abbildungs 3.8(b) stellt die empfangene Signalstärke (in Englisch *Received Signal Strength Indication*, Kurzform **RSSI**) dar. RSSI ist die empfangene Leistung in miliwatts, wird aber in dBm(liste der Abkürzung decibel miliwatt) gemessen. Diesen wird 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 diese 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 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 Standardprotokoll für die Machine-To-Machine- und IoT-Kommunikation von Geräte und Anwendungen entwickelt. MQTT steht für **Message Queue Telemetry Transport**, und bietet eine sichere, verlässliche, performante und wartbare Kommunikation zwischen Anwendungen und Geräte. Dieses Protokoll unterscheidet sich von Request/Response-Protokollen wie 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 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 veröf-

fentlichte Topics zu abonnieren als auch selber veröffentlichen. Bei von der externen Anwendung veröffentlichte Topics können Downlinks ersetzen, diesmal werden die Downlinks nicht vom Server gesendet, sondern von der Externen Anwendung.

4 Software Emplimentierung

Bis jetzt wurden die Begriffe theoretisch erwähnt. Man kann zwar schon verstehen, was der Ziel der Arbeit ist, wie die gemessenen Daten gelesen werden und wie die LoRa 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. Die externe Anwendung für das Abonnieren an Topics und das Senden von Downlinks wurde in der Programmiersprache Python geschrieben.

4.1 Entwicklungsumgebung

Das gesamte Projekt wurde unter Ubuntu durchgeführt. Ubuntu ist eine berühmte Linux-Distribution. Da die Kosten der Entwicklung 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 angepasst werden kann. In diesem Teil des Berichts, wird in die Installation und die Konfiguration von Eclipse eingegangen, sodass der Entwickler C-Quelle-Codes durch einen Klick kompilieren und im Mikrocontroller laden kann.

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

```
1 $ sudo apt update
2 $ sudo apt install oracle-java8-installer
3 $ sudo apt install oracle-java8-set-default
```

Man soll nun Eclipse von der offiziellen Website herunterladen. Nach dem die Software heruntergeladen wurde, befindet sie sich unter /Downloads. Die Software soll ungepakt werden bevor die Installation erfolgt.

```
1 $ tar xzf ~/Downloads/eclipse-inst-linux64.tar.gz
2 $ ~/Downloads/eclipse-installer/eclipse-inst
```

Die Software startet, es wird nach dem Paket 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-Quellcodes für normalen Rechner können kompiliert werden, da der GCC-Compiler standardmäßig installiert wird. Wir wollen aber C-Codes für einen ARM-Mikrocontroller compilieren. Aus diesem Grund müssen wir ein externes Tool für ARM-Mikrocontroller in Eclipse einfügen. Damit das erfolgt, soll man das **Eclipse-Marketplace** unter **Help** suchen. Sobald das Eclipse-Marketplace geöffnet ist, soll man nach **GNU MCU Eclipse** suchen und installieren.

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

```
1 $ mkdir projekt1
2 $ cd projekt1
3 $ 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, 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, obwohl wir noch keinen ARM-Code compilieren können.

Wir wollen mit Hilfe von **MAKEFILES** unsere Quellen-Codes kompilieren. Wir haben dadurch eine volle Kontrolle auf die Kompilierung, und die Feh-

lerausgaben 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.

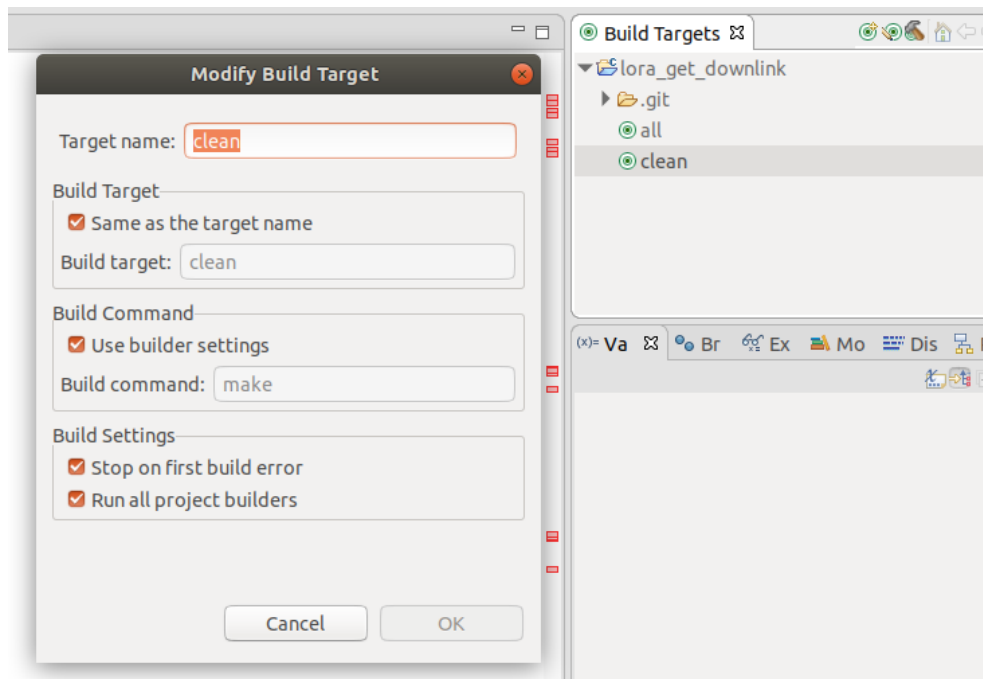


Abbildung 4.1: Build Targets einstellen

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

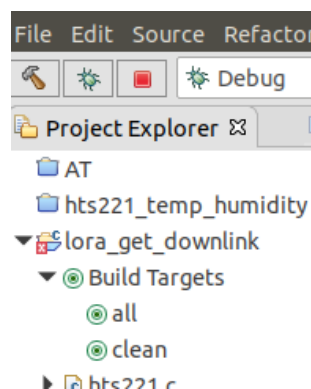


Abbildung 4.2: Build Targets fertig

```
1 PROJECT = lora_get_downlink //Name des Projekts
2 BUILD_DIR = bin //Verzeichnis der gebildete Dateien
3
4 CFILES = main.c //C-Quelle-Codes die Kompiliert werden muessen
5 CFILES += lora_modul.c
6 CFILES += lsm6dsl.c
7 CFILES += setup.c
8 CFILES += hts221.c
9 CFILES += systick.c
10
11 DEVICE=stm321475vg //Der benutzte Mikrocontroller
12
13 LDSCRIPT = ./stm32-clicker.ld //Der Linker
14 LDFLAGS += -u _printf_float //Damit floating points durch UART
    dastellbar werden
15
16 VPATH += $(SHARED_DIR)
17 INCLUDES += $(patsubst %, -I%, . $(SHARED_DIR))
18 OPENCM3_DIR=../libopencm3 //Das Verzeichnis der benutzte Bibliothek
19
20 include $(OPENCM3_DIR)/mk/genlink-config.mk
21 include ./rules.mk
22 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 Projekt, der Mikrocontroller und die zu kompilierenden C-Quelle-Dateien müssen je nach Aufgabe angepasst werden.

4.1.2 Libopencm3 Bibliothek 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 verwendet wurde. Wir können zwar diese Peripherien durch direkter Zugriff auf Registern, aber das würde zu viel Zeit nehmen, und der Code kann unübersichtlich werden. Damit wir uns diese Arbeit ein wenig sparen, 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 den GNU-ARM-Tool von Eclipse eine Bibliothek für ARM-Mikrocontroller. Diese wurde nicht verwendet, 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.

```

1 /* Bibliothek herunterladen und bilden */
2 $ git clone https://github.com/libopencm3/libopencm3
3 $ make
4
5 /* Beispiele */
6 $ git clone https://github.com/libopencm3/libopencm3-examples
7 $ make
8
9 /* Dokumentation bilden */
10 $ cd libopencm3/doc/
11 $ make

```

Nach der Installation kann die Dokumentation mit Hilfe eines Internetbrowser geöffnet werden (Siehe Abbildng 4.3).

```
1 $ firefox index.html
```

libopencm3

A free/libre/open-source firmware library for various ARM Cortex-M3 microcontrollers.

libopencm3	General Information	Back to Top	STM32F0	STM32F1	STM32F2	STM32F3	STM32F4	STM32F7	STM32L0	STM32L1	STM32L4	STM32G0	GD32F1X0	EFM32G	EFM32GG	EFM32HG	EFM32LG	EFM32TG	LM3S	LM4F	MSP432E4
LPC130X	LPC17XX	LPC43XX	SAM3A	SAM3N	SAM3S	SAM3U	SAM3X	VF6XX	Data Structures	Files	Examples										

libopencm3

- libopencm3 Core CMS
- libopencm3 License
- Deprecated List
- Back to Top
- STM32F0
- STM32F1
- STM32F2
- STM32F3
- STM32F4
- STM32F7
- STM32L0
- STM32L1
- STM32L4
- STM32G0
- GD32F1X0
- EFM32G
- EFM32GG
- EFM32HG
- EFM32LG
- EFM32TG
- LM3S
- LM4F
- MSP432E4
- LPC130X
- LPC17XX
- LPC43XX
- SAM3A
- SAM3N
- SAM3S
- SAM3U
- VF6XX

STM32L4

Here is a list of all modules:

Module	Description
CM3 Defines	Defined Constants and Types for Cortex M3 core features
MPU Defines	libopencm3 Cortex Memory Protection Unit
Cortex-M Trace Port Interface Unit (TPIU)	
Cortex-M Flash Patch and Breakpoint (FPB) unit	
SysTick Defines	libopencm3 Defined Constants and Types for the Cortex SysTick
Cortex-M System Control Block	
Debugging	Macros and functions to aid in debugging
Cortex-M Instrumentation Trace Macrocell (ITM)	
Cortex-M System Control Space	The System Control Space (SCS) is a memory-mapped 4KB address space that provides 32-bit registers for configuration, status reporting and control
Cortex Core Defines	libopencm3 Defined Constants and Types for the Cortex Core
NVIC Defines	libopencm3 Cortex Nested Vectored Interrupt Controller
Cortex Core Peripheral APIs	APIs for Cortex Core peripherals
NVIC	libopencm3 Cortex Nested Vectored Interrupt Controller
SysTick	libopencm3 Cortex System Tick Timer
USART register base addresses	Holds all the USART peripherals supported
Peripheral APIs	APIs for device peripherals
FLASH peripheral API	libopencm3 STM32L4xx FLASH
PWR peripheral API	libopencm3 STM32L4xx Power Control
ADC peripheral API	
CRC peripheral API	
DMA peripheral API	
EXTI peripheral API	
GPIO peripheral API	
I2C peripheral API	

Generated on Tue Jun 2 2020 13:29:53 for libopencm3 by doxygen 1.8.19

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 Konfigurations des Schnittstelle.

```
1 void i2c2_setup(void)
2 {
3     rcc_periph_clock_enable(RCC_I2C2);
4     /* Setup SDA and SCL 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-Kommunikationsschnittstelle, 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). Jedes Slave hat eine Adresse in Hexadezimal, womit der Master es ansprechen kann. Der Master muss diese Adresse kennen, sonst können beide nicht kommunizieren.

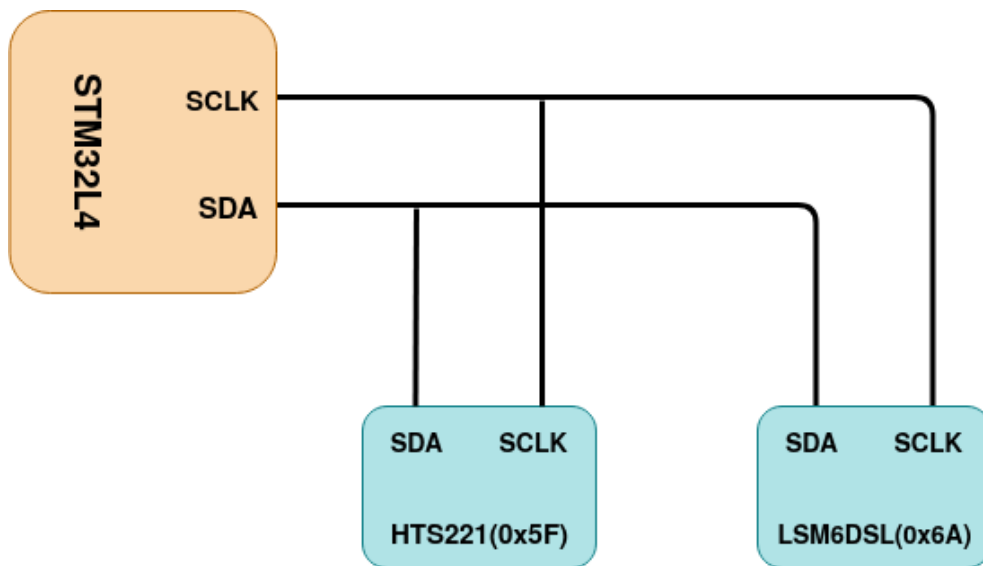


Abbildung 4.4: I2C Kommunikation

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 im Sensoren-Register schreibt und wie diese Register zu lesen sind. Im Anhang befinden sich alle Quellem- 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 Quellen-Dateien **hts221.c** und **lsm6dsl.c** sind diese Funktionen implementiert. Das 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.

```

1  /* Temperatur ermitteln wie im Kapitel 2.1.1 beschrieben */
2
3  /* Read T0_degc_x8 and T1_degc_x8 */
4  cmd[0] = T0_DEGC_X8;
5  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);
6  cmd[0] = T1_DEGC_X8;
7  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
8
9  /* Read the most significant bit of T1_DEGC and T0_DEGC */
10 cmd[0] = T1_T0_MSB;
11 i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+3), 1);
12
13 /* Calculate the T0_degc and T1_degc values */
14 T0_degc_x8 = (((uint16_t) (cmd[3] & 0x02)) << 8) | ((uint16_t) cmd[1])
15             ;
16 T1_degc_x8 = (((uint16_t) (cmd[3] & 0x0C)) << 6) | ((uint16_t) cmd[2])
17             ;
18 T0_degc = T0_degc_x8>>3;
19 T1_degc = T1_degc_x8>>3;
20
21 /* Read T0_OUT less significant bit */
22 cmd[0] = T0_OUT_L;
23 i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);
24
25 /* Read T0_OUT most significant bit */
26 cmd[0] = T0_OUT_M;
27 i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
28 T0_out = (((int16_t) cmd[2])<<8) | (int16_t) cmd[1];
29
30 /* Read T1_OUT less significant bit */
31 cmd[0] = T1_OUT_L;
32 i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);

```

```

32
33  /* Read T1_OUT most significant bit */
34  cmd[0] = T1_OUT_M;
35  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
36  T1_out = (((int16_t)cmd[2])<<8) | (int16_t)cmd[1];
37
38
39  /* Read T_OUT less significant bit */
40  cmd[0] = T_OUT_L;
41  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+1), 1);
42
43  /* Read T_OUT most significant bit */
44  cmd[0] = T_OUT_M;
45  i2c_transfer7(I2C2, HTS221_ADDR, cmd, 1, (cmd+2), 1);
46  T_out = (((int16_t)cmd[2])<<8) | (int16_t)cmd[1];
47
48  /* Calculate the temperature value */
49  tmp = ((int32_t)(T_out - T0_out)) * ((int32_t)(T1_degc - T0_degc));
50  temperature = ((float)tmp / (float)(T1_out - T0_out)) + (float)(
    T0_degc);
51
52  return temperature;

```

```

1  /* Beschleunigung ermitteln wie im Abbildung 2.5 */
2
3  status_cmd[0] = STATUS_REG;
4  i2c_transfer7(I2C2, LSM6DSL_ADDR, status_cmd, 1, (status_cmd+1), 1);
5
6  if (status_cmd[1] & GET_XLDA) {
7      /* Read X */
8      cmd[0] = OUTX_L_XL;
9      i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
10     outx_l_xl = cmd[1];
11
12     cmd[0] = OUTX_H_XL;
13     i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);
14     outx_h_xl = cmd[1];
15
16     x = (((int16_t)outx_h_xl << 8) | (int16_t)outx_l_xl);
17
18     /* Read Y */
19
20     cmd[0] = OUTY_L_XL;
21     i2c_transfer7(I2C2, LSM6DSL_ADDR, cmd, 1, (cmd+1), 1);

```

```

22     outy_l_xl = cmd[1];
23
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 Werten. Der Index 0 beinhaltet den Wert der X-Achse, Index 1 beinhaltet den Y-Wert und Index 2 den Y-Wert. (Genau so wie für den Gyroskop)

Nachdem die Daten gelesen worden sind, kann man sie nun in Hexadezimal konvertieren und mit Hilfe 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 einem Transfer startet. Der folgende Quellcode ist dafür zuständig. Die Konfiguration muss die gleiche sein wie im Abschnitt 2.2.4 beschrieben (Baudrate: 115200, Daten:8 Bit, Keine Parität und 1

Stopbit).

```
1  rcc_periph_clock_enable(RCC_UART4);
2
3  void uart_setup(void)
4  {
5      /* Setup GPIO pins for UART4 and USART3 transmit. */
6      gpio_mode_setup(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO0);
7      gpio_mode_setup(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO1);
8
9      /* Setup UART4 TX and RX pin as alternate function. */
10     gpio_set_af(GPIOA, GPIO_AF8, GPIO0);
11     gpio_set_af(GPIOA, GPIO_AF8, GPIO1);
12
13     /* UART4 setup */
14     usart_set_baudrate(UART4, 115200);
15     usart_set_databits(UART4, 8);
16     usart_set_stopbits(UART4, USART_STOPBITS_1);
17     usart_set_mode(UART4, USART_MODE_TX_RX);
18     usart_set_parity(UART4, USART_PARITY_NONE);
19     usart_set_flow_control(UART4, USART_FLOWCONTROL_NONE);
20
21     /* Finally enable UART4. */
22     usart_enable(UART4);
23 }
```

Wie im Abschnitt 2.2.4 erwähnt, sollen AT-Befehle am LoRa-Modul senden, um das Modul für eine LoRaWAN-Verbindung vorzubereiten. Diese Befehle werden ein Zeichen nach dem anderen per UART an dem LoRa-Modul gesendet werden. Da viele AT-Befehle Parameter erwarten, haben wir mit Hilfe der Stringifikation Makros definiert, die dem Benutzer ermöglichen veränderliche Strings in C durch UART zu senden.

Zuerst wurde eine Enumeration aller Befehle herstellt, die gebraucht werden. Danach mit Hilfe eines zweidimensionalen Feldes und des Stingifikation können wird Parametern zwischen den Strings setzen. Die Enumeration gibt den Anfangindex jedes Befehls an. Damit kann man weiteriterieren, um ein komplettes String zu bilden. Die folgenden Zeilen-Code zeigen die Stingifikation, die Enumeration und das zweidimensionale Feld.

```
1  /*
2  *
3  * Stringification
4  * Helpful to convert macro argument into
5  * string constant
6  */
7  #define AT_COMMAND(cmd, param) "AT+" #cmd "=" #param "\r\n\0"
```

```
1  /*
2  * List of possible AT command
3  */
4  enum lora_cmd {
5      AT = 0,
6      ATZ,
7      GET_APPEUI,
8      GET_EUI,
9      GET_NET_S_KEY,
10     GET_APP_S_KEY,
11     GET_APP_KEY,
12     GET_ADDR,
13     GET_JOIN_STATUS,
14     AT_RADIO,
15     AT_GETDATA,
16     AT_RX1_DELAY,
17     AT_RX2_DELAY,
18     AT_JOIN_ACCEPT1_DELAY,
19     AT_JOIN_ACCEPT2_DELAY,
20     AT_DUTY_CYCLE,
21     AT_BAND,
22     AT_DATA_RATE,
23     AT_RX2_DATA_RATE,
24     AT_CLASS,
25     AT_ADDR,
26     AT_NET_S_KEY,
27     AT_APP_S_KEY,
28     AT_APP_KEY,
29     AT_APPEUI,
30     AT_JOIN_OTAA,
31     AT_JOIN_ABP,
32     AT_NETWORK_TYP,
33     AT_ADAPTIVE_DATA_RATE,
34     AT_MAX
35 };
```

```
1  /*
2  * List of possible codes
3  */
4  /* Default network type is public (NTYP = 1, 0 for private network)
5  * Band = EU868
6  * DevEUI = e24f43fffe44c432
7  * Duty Cycle (DC) = is on
8  * Data Rate (DR) = SF7/125KHz
9  * Class = Class A
10 * APPEUI is a 8 bytes hex value
11 * AppKey (AK) NSK ASP are 16 byte hex value
12 * Device address (ADDR) is a 4 bytes hex value
13 * Radio setting: Power: 14dBm
14 *     Frequency: 867.1MHz
15 *     Spreading factor: 12
16 *     Bandwidth: 125KHz
17 *     Coding rate: 4/5
18 *
19 */
20
21 char *code[] = {
22     "AT\r\n\0",
23     "ATZ\r\n\0",
24     "AT+APPEUI\r\n\0",
25     "AT+EUI\r\n\0",
26     "AT+NSK\r\n\0",
27     "AT+ASK\r\n\0",
28     "AT+AK\r\n\0",
29     "AT+ADDR\r\n\0",
30     "AT+JSTA\r\n\0",
31     "AT+RF=14,8671000000,12,0,1\r\n\0",
32     "AT+RCV\r\n\0",
33     AT_COMMAND(RX1DT, 1000),
34     AT_COMMAND(RX2DT, 2000),
35     AT_COMMAND(JRX1DT, 5000),
36     AT_COMMAND(JRX2DT, 6000),
37     AT_COMMAND(DC, 0),
38     AT_COMMAND(BAND, 0),
39     AT_COMMAND(DR, 0), //UE860: SF12-BW125
40     AT_COMMAND(RX2DR, 0), //UE860: SF12-BW125
41     AT_COMMAND(CLASS, 0),
42     AT_COMMAND(ADDR, 12345678),
43     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 per UART4 gessendet.

```

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 desendet 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 dagegen, bekommt man die Abgefragte Daten oder ein Fehler zurück. Eine Ausnahme bestehet bei Events.

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

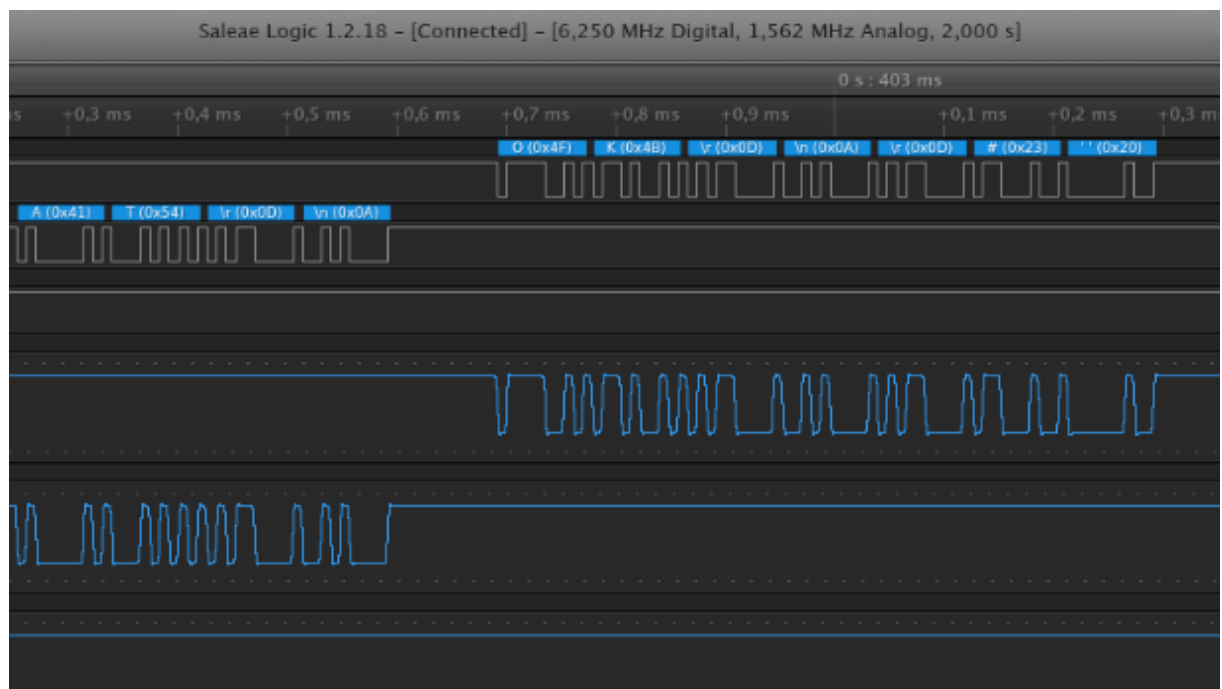


Abbildung 4.5: Saleae Output

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

4.4 Downlinks Behandlung

5 Fazit

5.1 Zusammenfassung

5.2 Ausblick

Literaturverzeichnis

- [1] 120 lora / lorawan range world record attempt. will i succeed? 2017. https://www.youtube.com/watch?v=adhWIo-7gr4&list=PL3XBzmAj53Rkkogh-lti58h_GkhzU1n7U&index=6&t=506s.
- [2] LoRa Alliance. What is lorawan. 2020. <https://loro-alliance.org/resource-hub/what-lorawanr>.
- [3] Ali Benfattoum. De la technologie lora au reseau lorawan. 2016. <https://www.frugalprototype.com/technologie-lora-reseau-lorawan/>.
- [4] Warren Gay. *Beginning STM32. Developing with FreeRTOS, libopencm3 and GCC*. 2018. <https://doi.org/10.1007/978-1-4842-3623-9>.
- [5] Petr Gotthard. Compact server for private lorawan networks. 2016. <https://github.com/gotthardp/lorawan-server/tree/master/doc>.
- [6] Brian Gough. *An Introduction to GCC for the GNU Compilers gcc and g++*. Network Theory Limited, Bristol UK, 2005. <http://www.network-theory.co.uk/gcc/intro/>.
- [7] libopencm3. libopencm3 lowlevel open-source library for arm cortex mcus. 2013. <https://github.com/libopencm3>.
- [8] Muhammad Ali Mazidi, Shujen Chen, and Eshragh Ghaemi. *STM32 ARM Programming for Embedded Systems. Using C language with STM32F4 ARM*. 2014.
- [9] Richard Reese. *Understanding and using C pointers*. O'Reilly Media, Sebastopol, 2013.
- [10] Mickael Remond. *Erlang Programming*. 2003.

- [11] saleae. Debug hardware like the pros with the logic analyzer you will love. 2020. <https://www.saleae.com/>.
- [12] Semtech. Lora network packet forwarder project. 2013. https://github.com/Lora-net/packet_forwarder.
- [13] STMicroelectronics. Capacitive digital sensor for relative humidity and temperature. STMicroelectronics, 2018. <https://www.st.com/en/mems-and-sensors/hts221.html#resource>.
- [14] STMicroelectronics. inemo inertial module: always-on 3d accelerometer and 3d gyroscope. 2018. <https://www.st.com/en/mems-and-sensors/lsm6dsl.html#resource>.
- [15] STMicroelectronics. Discovery kit for iot node, multi-channel communication with stm32l4. 2019. <https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html#resource>.
- [16] STMicroelectronics. dm00083560-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics. 2019. <https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html#resource>.
- [17] STMicroelectronics. Usi lora expansion board for stm32 nucleo. 2019. <https://www.st.com/en/evaluation-tools/i-nucleo-lrwan1.html#resource>.
- [18] USI. Wm-sg-sm-42 at command reference manual. 04 2018. https://github.com/USIWP1Module/USI_I-NUCLEO-LRWAN1/tree/master/preloaded_firmware.