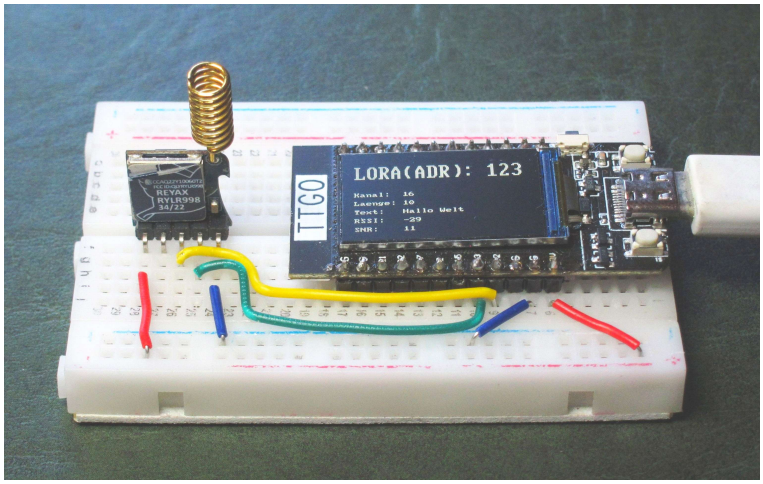
LoRaWAN

Eine praktische Einführung mit Micropython sowie dem RYLR998-Modul und dem ESP32-Board TTGO T-Display



von

G. Heinrichs

Mönchengladbach, 12.07.2024

Einführung

Mit dem WLAN steht eine häufig eingesetzte Funkübertragungstechnik zur Verfügung. In dem Skript ***WLAN-Experimente mit dem ESP32-Board TTGO T-Display*** [WLN] habe ich dargestellt, wie man mit dieser Technologie nicht nur einfache Daten wie z. B. einen Temperaturwert oder eine Zeitangabe, sondern auch umfangreiche Dateien wie z. B. ganze Webseiten übertragen kann. Ähnliches gilt auch für **Bluetooth** [BLE]..

Neben dem WLAN und Bluetooth ist uns noch das ***HANDY-Netz (GSM/LTE/5G)*** geläufig. Auch dieses kann nicht nur kleine Datenmengen (wie z. B. SMS), sondern auch große Dateien wie Videos übertragen.

Wozu jetzt also LoRaWAN als weitere Funkübertragungstechnik?

Die oben genannten Techniken haben unterschiedliche Nachteile: WLAN und Bluetooth haben nur eine geringe Reichweite. Im Freien kommen wir beim WLAN auf bis zu 100 m; in Gebäuden werden meist nur ein bis zwei Mauern durchdrungen. Beim Handy-Netz ist die Reichweite deutlich höher; hier werden (im Freien) Strecken von mehreren Kilometern überwunden. Allerdings ist hier der Energieaufwand deutlich höher als beim WLAN. Außerdem gibt es (neben der Anschaffung) Kosten für die Nutzung des Netzes. LoRaWAN hat nun folgende Vorteile:

- **Geringer Energieeinsatz**: Im Leerlauf brauchen diese Geräte praktisch keine Energie, wenn man die Geräte so programmiert, dass sie nur wenn erforderlich aktiv werden. Dies kann man z. B. durch die Festlegung bestimmter Zeitschlitze erreichen.

- **Hohe Reichweite**: Ein Grund für die höhere Reichweite ist die niedrigere Frequenz: Grob gesagt ist die Reichweite um so größer, je kleiner die Frequenz ist. Hinzu kommt aber auch eine spezielle Modulationstechnik. Aufgrund dieser beiden Faktoren können nun deutlich größere Reichweiten erzielt werden als bei WLAN oder Bluetooth: In Gebäuden können häufig mehrere Mauern und Decken überwunden werden. Im Freien sind Reichweiten von mehreren Kilometern möglich. Die Abkürzung LoRa beschreibt genau diese Eigenschaft: ***Lo***ng ***Ra***nge (große Reichweite).

- **Keine zusätzlichen Kosten**: Die Benutzung des 868MHz-Bandes (SRD-Band - **S**hort **R**ange **D**evice) ist frei; insbesondere ist keine Anmeldung erforderlich. Jedoch gibt es einige Vorgaben: So soll man z. B. (bei der in diesem Skript benutzten Frequenz von 868,5 MHz) nicht mehr als 1% der Zeit senden, vgl. Kapitel 6.

All diese Vorteile werden allerdings erkauft durch eine geringe Datenrate. LoRaWAN ist prädestiniert für das **IoT** (**I**nternet **o**f **T**hings); hier werden meist nur kleine Datensätze übertragen, bei denen die Datenübertragungsrate nicht im Vordergrund steht. Damit können z. B. Füllstände, Temperaturwerte oder auch der Zählerstand von "Stromzählern" übermittelt werden.

Häufig werden die Abkürzungen **LoRa** und **LoRaWAN** verwechselt oder auch synonym verwendet. Dies ist eigentlich nicht korrekt: LoRa beschreibt die von SEMTECH entwickelte *physische Schicht*, d.h. die Modulationstechnik, mit der Kommunikationsverbindungen über Langstrecken erreicht werden. (Ihr wesentliches Merkmal ist die Benutzung von so genannten **Chirps**; hierbei handelt es sich um Signale mit einer ansteigenden Frequenz. Solche Chirps tauchen übrigens in akustischer Form auch in der Natur auf; sie werden z. B. von Delfinen und Fledermäusen benutzt.) LoRaWAN hingegen definiert das Kommunikationsprotokoll und die Systemarchitektur für das Netzwerk, welche die LoRa-Modulation in der physischen Schicht verwendet.

In diesem Skript werde ich im Wesentlichen nur auf **Peer-to-Peer-Übertragungen** (d. h. die Übertragung von einem LoRa-Modul zu einem anderen) eingehen. Es gibt auch die Möglichkeit, Daten per LoRaWAN an **LoRa-Gateways** zu übermitteln (vgl. Abb. 1). Diese senden dann die empfangenen Daten mittels TCP/IP an Server weiter, wo sie dann verarbeitet werden oder anderen Nutzern zur Verfügung gestellt werden.

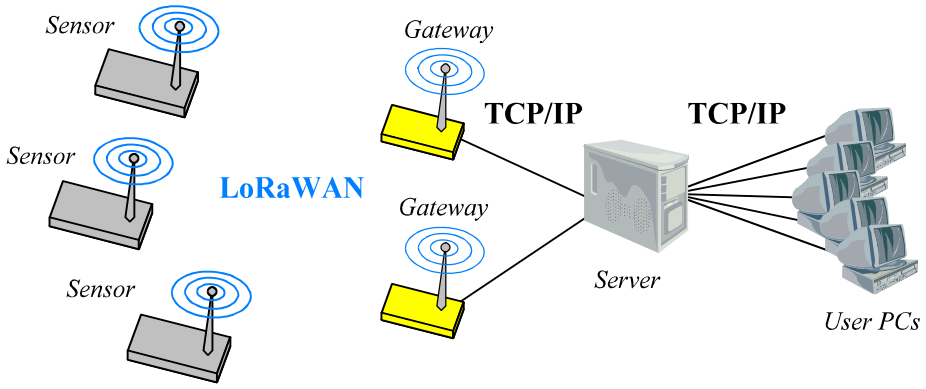


Abb. 1

Das LoRa-Modul REYAX RYLR998

Das LoRa-Modul RYLR998 der Firma REYAX (Abb. 2) kann als Sender und auch als Empfänger benutzt werden. Gesteuert wird es durch so genannte **AT-Commands**. Solche AT-Commands sind schon vor Jahrzehnten - insbesondere bei Telefonen und Modems - zum Einsatz gekommen. Auch das weit verbreitete GSM-Modul SIM800 bedient sich solcher AT-Commands (s. [GSM]).

Abb. 2

Mit diesen AT-Commands können sowohl Einstellungen am LoRa-Modul (z. B. das benutzte Frequenz-Band) vorgenommen als auch Daten übertragen werden. Die AT-Commands werden über eine UART-Schnittstelle an das LoRa-Modul übertragen. Über diese Schnittstelle kann das Modul auch empfangene Botschaften ausgeben.

Das LoRa-Modul RYLR998 ist gerade für einen Einstieg in unser Thema gut geeignet: Es müssen keine zusätzlichen Installationen vorgenommen werden, die AT-Commands sind einfach strukturiert und ihre Anzahl ist gut überschaubar. Für unsere Experimente werden wir mit etwa einem Dutzend davon auskommen.

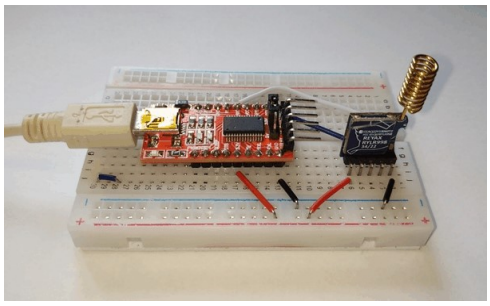


Abb. 3

An dieser Stelle soll aber nicht verschwiegen werden, dass diese Vorteile mit gewissen Einschränkungen verbunden sind: So kann man z. B. nicht alle LoRa-Parameter unabhängig von einander einstellen.

Um mit dem LoRa-Modul zu kommunizieren, wollen wir in diesem Skript zwei Wege benutzen.

1. Weg: Wir benutzen einen USB-UART-Konverter **mit 3,3-V-Signalen(!)** wie z. B. den FTDI232RL (s. Abb. 3). Durch diesen Konverter wird der LoRa-Baustein mit einem PC verbunden. Über diese Verbindung können wir nun mit Hilfe eines Terminal-Programms (z. B. *Hterm*) AT-Commands senden und auch Antworten vom LoRa-Baustein erhalten.

2. Weg: Wir können das LoRa-Modul direkt an einen Mikrocontroller (z. B. unseren TTGO T-Display) anschließen. Der Mikrocontroller kann dann über eine seiner UARTs durch entsprechende Programmierung AT-Commands an das LoRa-Modul senden und damit Daten senden und empfangen.

Bei diesem 2. Weg werden wir **Micropython** als Programmiesprache verwenden. Hierbei handelt es sich um eine Variante der Programmiersprache **Python**, die speziell für Mikrocontroller konzipiert wurde: Micropython benutzt dieselbe Syntax wie Python, aber im Kern besitzt sie nur eine Teilmenge der Python-Befehle; hinzu kommt allerdings noch eine Reihe von Mikrocontroller-spezifischen Befehlen.

Sie, lieber Leser, sollten einige Vorkenntnisse hinsichtlich der Programmierung mit Mikropython besitzen. In meinen Skripten "WLAN-Experimente mit dem ESP32-Board TTGO T-Display" [WLN]und "Bluetooth Low Energy: Eine praktische Einführung mit Micropython und dem ESP32-Board TTGO T-Display" [BLE] schildere ich diese genauer und biete dort auch neben einer Installationsanweisung für die Entwicklungsumgebung *Thonny* einen kleinen Micropython-Einführungskurs.

Die einzelnen Kapitel dieses LoRa-Skripts bauen aufeinander auf. Es bietet sich - insbesondere für Anfänger - deswegen an, diese Kapitel der Reihe nach durchzugehen. Ganz bewusst habe ich mich entschieden, das Thema nicht an einem einzigen Projekt abzuhandeln. Vielmehr lernen Sie Schritt für Schritt neue Eigenschaften des LoRa-Moduls und zugehörige Anwendungen kennen. Ganz wichtig war es für mich dabei, ab und zu einen *behutsamen Blick hinter die Kulissen von LoRa* zu gewähren: So lernen Sie z. B. wichtige Begriffe wie Modulation, Chirps, Bandbreite oder Spreizfaktor kennen. Damit können Sie dann eine Vorstellung davon erhalten, wie LoRa funktioniert.

Inhaltsverzeichnis

1 Ein erster Kontakt

Wir beschreiten hier den Weg 1: Dazu schließen wir an eine USB-Buchse unseres PC einen USB-UART-Wandler (z. B. FTDI 232RL) an. Die Pins 3.3 V (oft einfach nur mit 3V bezeichnet), GND (Masse), RxD und TxD dieses Wandlers schließen wir an die entsprechenden Pins unseres LoRa-Moduls **RYLR998** an (vgl. Abb. 2 der Einführung). Dabei muss natürlich der RxD-Anschluss [TxD-Anschluss] des Wandlers mit dem TxD-Anschluss [RxD-Anschluss] des LoRa-Moduls verbunden werden.

Achtung: Beim LoRa-Modul RYLR998 dürfen sowohl die **Versorgungsspannung** als auch die Signalspannungen **maximal 3,6 V** sein. Manche USB-UART-Wandler arbeiten mit höheren Spannungen (z. B. 5 V); diese würden unser LoRa-Modul **zerstören**. Beim FTDI 232RL können wir mit Hilfe eines Jumpers (in der Abb. 3 zwischen dem IC und der Stiftleiste) die Spannung 3,3 V bzw. 5,0 V wählen. In der Abb. 3 ist der Jumper für die Spannung 3,3 V eingesteckt worden.

Als Terminal benutze ich das Programm ***HTerm***, welches Sie im Materialien-Ordner finden können. Eine aktuelle Version können Sie auch von der Webseite

https://www.der-hammer.info/pages/terminal.html

downloaden. Nach dem Entpacken kann das Programm *HTerm.exe* (ohne Installation) direkt gestartet werden.

Bevor wir unser erstes AT-Command an das LoRa-Modul senden können, müssen wir bei *HTerm* einige Einstellungen vornehmen. Dazu

- geben wir die Portnummer (ggf. Mit Hilfe des Gerätemanegers ermitteln) ein,
- wählen wir die Baudrate 115200 (Standardwert bzw. Default bei unserem LoRa-Modul) aus,
- betätigen wir die Connect-Schaltfläche,
- wählen wir im Receive-Bereich *Newline at* die Option **LF** (**L**ine **F**eed = **N**ew **L**ine) und aktivieren (nur) das Ascii-Feld,
- aktivieren wir im Transmit-Bereich (nur) das Ascii-Feld, wählen bei *Send on* die Option CR-LF (**CR** steht für **C**arriage **R**eturn = Wagenrücklauf) und bei *Type* die Option ASC.

In das nebenstehende Eingabefeld geben wir zur ersten Kontaktaufnahme das Kommando **AT** ein und schließen es mit der Enter-Taste ab. (Benutzen Sie ***nicht*** die AutoSend-Schaltfläche!) In dem Feld darunter sehen wir, dass neben der Zeichenkette AT auch die Steuerzeichen \r für Carriage **R**eturn und \n für **N**ew Line an das LoRa-Modul gesendet wurden. Im Bereich Received Data sollte jetzt die Quittung +OK\r\n zu lesen sein. Ansonsten sollten die Verbindungskabel sowie die Einstellungen beim Terminal-Programm überprüft werden. Bei der Fehlersuche kann auch die erhaltene Quittung hilfreich sein: Im Datenblatt des LoRa-Moduls (https://reyax.com/products/rylr896/ bzw. [LD1]) findet man dafür eine Tabelle mit möglichen Fehlerquellen. So deutet z. B. die Fehlermeldung +ERR=1 darauf hin, dass das LoRa-Modul kein CR-LF-Signal (Hexcodes: 0x0D und 0x0A) erhalten hat.

Das Kommando AT dient hier nicht zur Initialisierung, sondern nur zur Kontrolle der Verbindung.

Einstellung der Adresse

Damit dem LoRa-Modul Botschaften zugesandt werden können, muss es eine Adresse besitzen. Mit dem Kommando

AT+ADDRESS=16

wird dem Modul die Adresse 16 zugewiesen. Im Receive-Bereich erhalten wir dabei eine entsprechende Quittung. Die Adresse wird dauerhaft im Flash abgespeichert.

Hat man die Adresse vergessen, kann man sie mit dem Kommando AT+ADDRESS? anzeigen lassen. Geben Sie dazu ein:

AT+ADDRESS?

Einstellung der Frequenz

Nun müssen wir die voreingestellte Frequenz ändern: In Europa sind Frequenzen im Bereich zwischen 863 MHz und 870 MHz (SRD-Band Europa) nutzbar (Mehr dazu im Kap. 6). Wir geben hier die Frequenz 868,5 MHz ein:

AT+BAND=868500000

Bei den RYLR998-Modulen mit der Versionsnummer 1.2.0 (oder höher) kann der Frequenzwert dauerhaft im Flash gespeichert werden, wenn man das Kommando

AT+BAND=868500000,M

benutzt. Ansonsten muss man bei jedem Neustart des Moduls die Frequenz erneut eingeben. Die aktuelle Frequenz können Sie mit dem Kommando AT+BAND? abfragen.

Die Versionsnummer können sie übrigens mit dem Kommando AT+VER? in Erfahrung bringen.

Beachten Sie: Die Anzahl der Lösch/Schreibvorgänge beim Flash-Speicher unseres LoRa-Moduls ist nach Angaben des Manuals [LD2] auf ca. 200 K, d. h. 200 000, begrenzt.

Einstellung der Sendeleistung

Der Standardwert für die Sendeleistung ist bei unserem LoRa-Modul 22 dBm. Gemäß der **CE-Zertifikation** (vgl. [LD1]) für den Betrieb unseres LoRa-Moduls muss die **Sendeleistung auf maximal 14 dBm begrenzt** werden. Die Sendeleistung wird mit dem CRFOP-Command (**C**ontrol **RF O**utput **P**ower) eingestellt. **Der maximal zulässige Wert ist hiernach also 14**. Das entsprechende Kommando lautet:

AT+CRFOP=14

Dieser Wert wird dauerhaft im Flash abgespeichert.

Einstellung der Baudrate

Manche Mikrocontroller kommen mit der voreingestellten Baudrate von 115200 Baud an ihre Grenzen. Dann empfiehlt es sich, eine niedrigere Baudrate zu wählen. Ich habe mich bei meinen Experimenten für 57600 Baud entschieden. Damit ist die serielle Übertragung zwischen Mikrocontroller bzw. Terminal und LoRa-Modul immer noch deutlich schneller als die eigentliche Funkübertragung zwischen den beiden LoRa-Modulen.

Um 57600 als neuen Wert für die Baudrate festzulegen, benutzt man das AT-Kommando

AT+IPR=57600

Die neue Baudrate wird dauerhaft im Flash gespeichert. Von jetzt ab kommuniziert das LoRa-Modul nur noch mit dieser neuen Baudrate. Das bedeutet insbesondere: Wenn man nach der Ausführung des Kommandos weiter mit dem LoRa-Modul arbeiten möchte, muss man die auf S. 2 durchgeführte Festlegung der Baudrate beim Terminal entsprechend ändern.

Zu den gerade vorgestellten AT-Commands gibt es weiterführende Hinweise in dem oben bereits erwähnten Datenblatt. Hier findet man eine Reihe von weiteren AT-Commands, die zum Senden und Empfangen von Daten erforderlich sind. Einige davon werden wir in den nächsten Kapiteln ausführlich studieren. Dabei werden wir an Stelle der Kombination von PC und USB-UART-Wandler auch unseren TTGO einsetzen (2. Weg). Wer kein Freund von Texten ist, der sollte sich vielleicht jetzt erst einmal das Video [VID] aus dem Anhang anschauen. Es zeigt in konzentrierter Form, wie man mit unserem LoRa-Modul ***umgeht***.

Eigentlich könnten wir hier schon loslegen und mit 2 Kombinationen von LoRa-Modul, USB-UART-Wandler und PC (mit *Hterm* -Programm) erste Nachrichten von einem LoRa-Modul (z. B. mit der Adresse 16) zu einem anderen (z. B. mit der Adresse 123) senden (vgl. Abb. 1.1). Natürlich müssen dabei der Sender als auch der Empfänger mit derselben Frequenz arbeiten. Wir wollen dies aber auf das nächste Kapitel verschieben; dort werden wir gleich einen Schritt weiter gehen und den Sender mit unserem TTGO ansteuern.

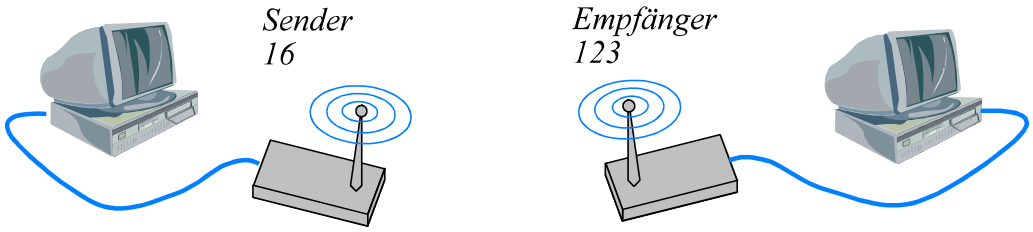


Abb. 1.1

2 AT-Commands mit Micropython

In diesem Kapitel wollen wir die AT-Commands nicht mehr über das bislang benutzte Terminal-Programm *HTerm* an unser LoRa-Modul senden, sondern mit Hilfe unseres TTGO-Microcontrollers. Die Idee ist folgende: Ein Micropython-Programm nimmt Ihre Kommandos (als Zeichenkette) mittels der `input`-Funktion über das *Thonny*-Terminal entgegen und sendet dann diese Zeichenkette über eine eigene serielle Verbindung des TTGO an das LoRa-Modul. Anschließend nimmt das Programm die Antwort des LoRa-Moduls entgegen und gibt sie mit Hilfe eines `print`-Befehls im Terminal-Bereich von *Thonny* aus.

Hinweis: Damit man die Vorgänge beim Ablauf des Programms kontrollieren kann, habe ich eine Reihe von `print`-Befehlen eingefügt. Sie können diese bei Bedarf natürlich entfernen. Dies gilt auch für die Programme, die in den folgenden Kapiteln vorgestellt werden.

Die Verbindung zwischen dem Microcontroller TTGO und dem LoRa-Modul stellen wir gemäß folgender Tabelle her:

TTGO T-Display		LoRa-Modul RYLR998
Bedeutung	Pin	Pin/Bedeutung
GND	G	GND
3.3 V	3V	VDD
TxD	12	RxD
RxD	13	TxD

Kommen wir jetzt zum Micropython-Programm! Zunächst müssen wir die `sleep`-Funktion aus dem time-Modul sowie die UART-Klasse UART aus dem Modul `machine` importieren:

```
from time import sleep
from machine import UART
```

Mit der Anweisung

```
uart1 = UART(1, baudrate = 57600, tx = 12, rx = 13)
```

stellen wir eine Instanz `uart1` dieser Klasse her. Diese stellt Methoden zur Verfügung, mit deren Hilfe Zeichenketten (genauer: einen Byte-String) über die Anschlüsse TxD und RxD mit 57600 Baud zum LoRa-Modul gesendet und umgekehrt auch von diesem empfangen werden kann.

Befehl	Bedeutung
`uart1.write(c)`	sendet den Byte-String `c`; Rückgabewert ist die Anzahl der gesendeten Bytes
`c = uart1.read()`	empfängt einen Byte-String und speichert ihn in `c` ab.
`c = uart1.readline()`	empfängt einen Zeile (d. h. einen Byte-String bis zu einem LF-Zeichen (0x0A)) und speichert sie in `c` ab.
`uart1.any()`	liefert den Rückgabewert `True`, wenn sich (mindestens) ein Byte im Lesepuffer befindet, ansonsten `False`

Damit definieren wir die folgenden Funktionen; es wird sich später als nützlich erweisen, auch die UART-Instanz als Parameter an die Funktionen zu übergeben:

```
def send_command(uart, cmd):   # cmd ist vom Typ string
    c = bytes(cmd, 'UTF-8') + b'\r\n'    # c ist vom Typ Byte-String
    anzahl_der_gesendeten_bytes = uart.write(c)
    print(anzahl_der_gesendeten_bytes, 'Bytes gesendet:', c)

def read_response(uart):
    print('auf Rückmeldung warten...')
    while not uart.any(): # warten, bis ein Byte im Lesepuffer ist
        sleep(0.1) # pass führt zu Problemen beim Threading (vgl. Kap. 11)
    return uart.readline()
```

Die erste Funktion sendet ein AT-Kommando über die serielle Schnittstelle `uart` an das LoRa-Modul. Dabei fügt es an das Kommando automatisch die Steuerzeichen `\c\n` an - genauso wie es auch unser Terminal-Programm *HTerm* macht. Zu Kontrollzwecken wird auch noch die Anzahl der gesendeten Bytes ausgegeben.

Die zweite Funktion liefert die Antwort vom LoRa-Modul: Zunächst durchläuft das Programm eine `while`-Schleife so lange, bis sich ein Byte im Lesepuffer befindet. Ist dies der Fall, wird eine Zeile aus dem Puffer gelesen und zurückgegeben.

Das Hauptprogramm ist jetzt sehr einfach:

```
print('AT-Commands ausführen lassen...')
print('Abbruch der Schleife mit Strg-C')
print()

while True:
    # Ein AT-Command ausführen lassen...
```

```
at_command = input('AT-Command (ohne CR-LF):')    # ohne \r\n
send_command(uart1, at_command)
print('gesendetes Kommando: ', at_command)
print('Rückmeldung:', read_response(uart1))
print()
```

In einer Endlos-Schleife wird zunächst ein AT-Command eingegeben. Durch die `send_command`-Funktion werden die nötigen Steuerzeichen für CR (**C**arriage **R**eturn) und NL (**N**ew **L**ine) angefügt und das Ergebnis davon als Byte-String an das LoRa-Modul gesendet. Die Rückmeldung vom LoRa-Modul wird nun mit Hilfe der Funktion `read_response` gelesen und durch die anschließende print-Funktion ausgegeben. Das gesamte Programm befindet sich auch in der Datei `AT_Commands_1b.py (s. [MPD])`. Wenn Sie es ausführen, sollten Sie nicht vergessen, die Sendeleistung mit AT+CRFOP=14 zu begrenzen.

In Abb. 2.1 sind zwei Eingaben bei unserem AT-Command-Programm dokumentiert. Bei der zweiten Eingabe wurde eine Botschaft an ein **anderes** LoRa-Modul mit der Adresse 123 (**muss zuvor mit AT+ADDRESS=123 eingestellt worden sein, vgl. Kapitel 1)** gesendet. Wenn man gleichzeitig dieses LoRa-Modul (mit der Adresse 123) wie im Kap. 1 beschrieben mit *HTerm* betreibt, dann empfängt es (ohne Eingabe weiterer Kommandos!) diese Botschaft und leitet sie an *HTerm* weiter, wo sie unverzüglich im Bereich *Received Data* angezeigt wird (Abb. 2.1). Dabei ist es egal, ob *Thonny* und *HTerm* auf demselben Rechner laufen oder auf verschiedenen Rechnern.

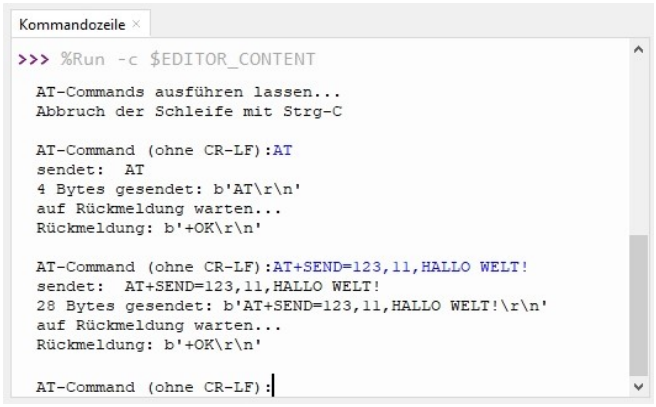


Abb. 2.1

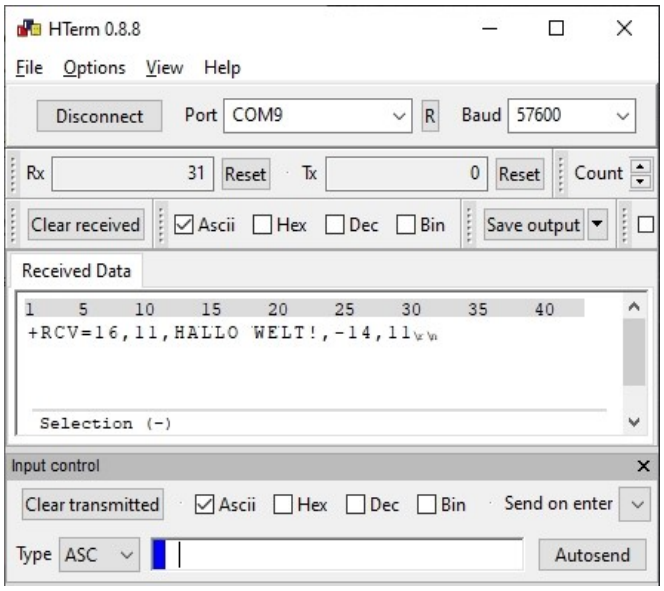


Abb. 2.2

Damit haben wir schon unsere erste Übertragung per LoRaWAN durchgeführt. Wer will, kann versuchen, auch andere Botschaften zu übertragen. Dabei kann Ihnen ggf. das Datenblatt zum LoRa-Modul RYLR998 [LD1] dabei helfen, die Bedeutung des 2. Parameters beim SEND-Command herauszufinden.

Wer zwei Rechner zur Verfügung, kann auch einmal untersuchen, wie groß die Reichweite ist. In einem späteren Kapitel werden wir zeigen, wie man solche Reichweitenuntersuchungen auch mit einem einzigen Rechner durchführen kann.

3 Empfänger

Am Ende des letzten Kapitels hatte unser TTGO T-Display per LoRaWAN eine Botschaft an das Terminal-Programm *HTerm* gesendet. Nun wollen wir den umgekehrten Weg beschreiten: Von *HTerm* aus soll eine Botschaft an den TTGO gesendet werden; der TTGO soll diese sowohl auf dem *Thonny*-Terminal als auch auf dem Display des TTGO anzeigen.

Beginnen wir mit den Instanziierungen für die UART und das Display:

```
#  Import und Instanziierungen
from time import sleep
from machine import UART, SPI, Pin
import vga1_bold_16x32 as font2
import vga1_8x8 as font1
import st7789
import sys

# Display:
spi = SPI(1, baudrate = 20_000_000, polarity = 1, sck = Pin(18), mosi =
                                                                  Pin(19))
display = st7789.ST7789(spi, 135, 240, reset=Pin(23, Pin.OUT), cs=Pin(5,
    Pin.OUT), dc=Pin(16, Pin.OUT), backlight=Pin(4, Pin.OUT), rotation=3)
display.init()
display.fill(0) # Display löschen

# UART:
uart1 = UART(1, baudrate = 57600, tx = 12, rx = 13)
```

Es folgen die bekannten Funktionen zur Kommunikation mit dem LoRa-Modul:

```
# Funktionen zur Kommunikation mit dem LORA-Modul
def send_command(uart, cmd):     # cmd ist vom Typ String
    c = bytes(cmd, 'UTF-8') + b'\r\n'    # c ist vom Typ Byte-String (Bytes)
    anzahl_der_gesendeten_bytes = uart.write(c)
    print(anzahl_der_gesendeten_bytes, 'Bytes gesendet:', c)

def read_response(uart):
    print('auf Nachricht warten...')
    while not uart.any(): # warten, bis (mindestens) ein Byte im Lesepuffer
        sleep(0.1) # pass führt zu Problemen beim Threading (vgl. Kap. 11)
    return uart.readline()
```

Mit der folgenden Funktion `lora_init` wird zunächst kontrolliert, ob die Verbindung zum LoRa-Modul korrekt funktioniert; gleichzeitig wird auch die Adresse des angeschlossenen LoRa-Moduls ermittelt und als Rückgabe-Wert an das Hauptprogramm zurückgegeben. Falls keine Verbindung zustande gekommen ist, wird die Zeichenkette '???' zurückgegeben. Dies kann manchmal ge-

schehen, wenn der TTGO stand-alone - also ohne *Thonny* - betrieben wird. Mehr Informationen zum stand-alone-Betrieb finden Sie in der Textbox am Ende dieses Kapitels.

```
def lora_init(uart):
    # Prüfe nach, ob LoRa-Modul antwortet...
    lora_address = '???'
    send_command(uart, 'AT')
    r = read_response(uart)
    print(r)
    if r == b'+OK\r\n': # Wenn nicht r=OK, dann...
        # LoRa-Adresse ermitteln und anzeigen
        send_command(uart, 'AT+ADDRESS?')
        lora_address = read_response(uart) # Rückgabewert ist vom Typ Bytes
        lora_address = str(lora_address, 'UTF-8') # umwandeln in Typ String
        lora_address = lora_address[9:]
        print('Eigene LORA-Adresse:', lora_address)
        print()
    return lora_address
```

Die Abbildung 2.2 aus dem letzten Kapitel zeigt, dass die empfangenen Botschaften ein Byte-String von der Form

```
b'+RCV=123,5,Hallo,-13,10\r\n'
```

sind: Hinter dem Gleichheitszeichen stehen die Parameter

123	Das ist die Adresse des Absenders.
5	Diese Zahl gibt die Anzahl der Zeichen von der gesendeten Botschaft an (ohne die Steuerzeichen \r\n).
Hallo	Dies ist die gesendete Botschaft (Payload).
-13	Hierbei handelt es sich um den *RSSI* -Wert (Dazu mehr in Kapitel 4 u. 10)
10	Hierbei handelt es sich um den *SNR* -Wert (Dazu mehr in Kapitel 10)

Um diese Parameter einzeln auf dem Display anzeigen zu können, benutzen wir die folgende Funktion:

```
def response_to_list(r): # Liste von Zeichenketten
    r = str(r, 'UTF-8')
    r = r.split(',')
    r0 = r[0].split('=') # Adresse isolieren
    r[0] = r0[1]
    r4 = r[4].split('\r') # CR-LF abtrennen
    r[4] = r4[0]
    return r
```

Diese Funktion zerlegt den empfangenen Byte-String in eine Liste, deren Elemente gerade aus den uns interessierenden Parametern besteht. In unserem Fall wäre der Rückgabewert dann

`['123', '5', 'Hallo', '-23', '11']`.

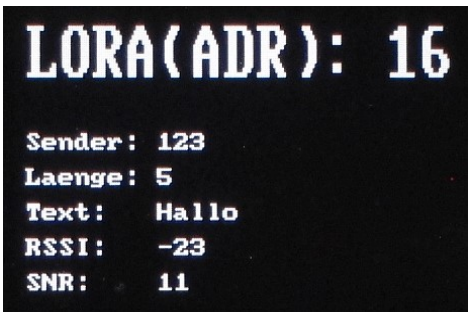


Abb. 3.1

Diese Zerlegung erfolgt schrittweise mit Hilfe der `split`-Methode, die jedes String-Objekt besitzt. Wie funktioniert diese Methode? Die Methode hängt man wie üblich durch einen Punkt getrennt an das zugehörige Objekt an; das Trennzeichen übergibt man der Methode als Parameter. Bei dem folgenden Beispiel benutzen wir als Trennzeichen einen Stern (*):

```
zk = 'Dies ist * ein Test fuer * das Zerlegen + vo*n Zeichenketten'
```

liefert `zk.split('*')` z. B. als Ergebnis

```
['Dies ist ', ' ein Test fuer ', ' das Zerlegen + vo', 'n Zeichenketten']
```

Bei der obigen Funktion `response_to_list` wird der Empfangsstring zunächst gemäß dem Komma-Trennzeichen aufgespalten. Im Fall der Zeichenkette

```
'+RCV=123,5,Hallo,-23,11\r\n'
```

erhält man zunächst die Liste

`r = ['+RCV=123', '5', 'Hallo', '-23', '11\r\n']`.

Um an die Zahl 123 zu gelangen, spalten wir das erste Element `r[0]` noch ein weiteres Mal auf, diesmal mit dem Trennzeichen '='. Die nächsten drei Elemente müssen wir nicht weiter behandeln; bei dem letzten Element müssen wir auf ähnliche Weise wie beim ersten Element die Steuerzeichen `\r\n` entfernen.

Durch die oben eingeführten Funktionen wird das Hauptprogramm nun recht einfach; seine Funktionsweise ergibt sich aus den Kommentaren im Programm.

```
# Hauptprogramm
lora_address = lora_init(uart1) # Rückgabewert ist '???', wenn keinen Antwort
                                                              vom LoRa-Modul
display.text(font2, 'LORA-ADR: ' + lora_address, 5, 10)

if lora_address == '???':
    display.text(font2, 'Resetten!', 5, 80, st7789.RED)
    print('Keine Verbindung zum LORA-Modul!')
    print('RESET-Knopf am TTGO betätigen!')
    sys.exit() # Programm beenden, wenn keine Antwort vom LoRa-Modul
# Das Lora-Modul empfängt automatisch die an ihn adressierte Botschaften
print('Empfangsschleife kann mit Strg-C abgebrochen werden.')
print()
```

```
while True: # Schleife mit Str-C abbrechen
    r = read_response(uart1) # empfangene Botschaft lesen
    # Ausgabe auf dem Thonny-Terminal
    print(r) # Rohdaten
    r_list = reponse_to_list(r) # Listenform
    print(r_list)
    print()
    # Ausgabe auf dem Display
    display.fill(0)
    display.text(font2, 'LORA(ADR): ' + lora_address, 5, 10)
    display.text(font1, 'Sender: ' + r_list[0], 5, 60)
    display.text(font1, 'Laenge: ' + r_list[1], 5, 75)
    display.text(font1, 'Text:   ' + r_list[2], 5, 90)
    display.text(font1, 'RSSI:   ' + r_list[3], 5, 105)
    display.text(font1, 'SNR:    ' + r_list[4], 5, 120)
```

Das gesamte Programm finden Sie in der Datei `Empfaenger_3a.py` (s. [MPD]). Testen Sie es ausgiebig, auch mit längeren Zeichenketten. Ab einer bestimmten Länge, wird der Text auf dem Display allerdings nicht mehr vollständig angezeigt. Wenn Sie mögen, können Sie das Programm einmal so abändern, dass es bis zu drei Zeilen Text auf dem Display anzeigen kann; Sie können dafür die Angaben von *RSSI* und *SNR* entfallen lassen.

Den TTGO stand-alone betreiben

Für den Betrieb ohne *Thonny* speichern Sie zunächst das Empfangsprogramm mit *Datei - Kopie speichern - (Where to save?) Micropython Device* unter dem Namen `main.py` im Flash-Speicher des ESP32 ab. Versorgen Sie nun den TTGO samt LoRa-Modul mit einer Powerbank. Manchmal wird nach dem Anschließen auf dem Display zunächst "LORA ???" angezeigt, ein Hinweis darauf, dass kein Kontakt mit dem LoRa-Modul hergestellt werden konnte. Betätigen Sie dann den Reset-Button am TTGO; jetzt sollte auf dem Display "LORA" mitsamt der Adresse des angeschlossenen LoRa-Moduls angezeigt werden (s. Abb. 3.1).
Warum klappt es hier mit der Verbindung erst im zweiten Versuch? Nun, eine Überprüfung mit einem Digital Analyzer zeigt: Kurz nach dem Anschließen an die Powerbank geht der TxD-Ausgang des TTGO für eine gewisse Zeit auf Low; in dieser Zeit arbeitet das LoRa-Modul schon, kann aber mit diesem (Schmutz-) Signal nichts anfangen und gibt entsprechend eine Fehler-Meldung (und nicht +OK) zurück. Nach dem Reset wird das Programm nun gestartet, ohne dass dieses Schmutz-Signal erzeugt wird.

4 Sender

Nachdem wir im letzten Kapitel ein Programm kennen gelernt haben, welches Botschaften empfangen kann, wollen wir uns nun um das Senden von Botschaften kümmern. Diese Botschaften könnten z. B. Messwerte sein, die mit einem an den TTGO angeschlossenen Sensor aufgenommen werden. Versehen wir die Kombination aus TTGO, LoRa-Modul und Sensor noch mit einer elektrischen Quelle (z. B. einer Powerbank oder einem Akku), dann haben wir einen Funksensor vor uns.

Um zunächst die wesentlichen Bestandteile des LoRa-Sende-Programms kennen zu lernen, begnügen wir uns in diesem Kapitel mit dem folgenden Ziel: Unser Sender soll die Zahlen 100 bis 999 über das LoRaWAN senden. Als Empfänger benutzen wir wieder ein LoRa-Modul, welches über einen USB-UART-Wandler an einen PC angeschlossen ist (vgl. Kap. 2).

Aus dem letzten Kapitel können wir direkt übernehmen:

- die Instanziierungen für die UART und das Display
- die Funktionen `send_command`, `read_response` und `lora_init`

Zusätzlich werden wir noch die folgende Funktion zum Senden einer Botschaft benutzen:

```
def send_message(uart, addr, msg): # addr: Adresse; msg: Botschaft
    payload_length = len(msg)
    cmd = 'AT+SEND=' + str(addr) + ',' + str(payload_length) + ',' + msg
    print('send_message:', cmd)
    send_command(uart, cmd)
```

Dabei muss die Adresse vom Typ `Integer (int)` und die Botschaft vom Typ `String (str)` sein. Die Funktionen `send_command`, `read_response`, `response_to_list`, `lora_init` und `send_message` finden Sie auch in der Datei `lora.py` (s. [MPD]). Von diesem Grundgerüst werden wir in Zukunft immer wieder Gebrauch machen.

Nach den nun hinlänglich bekannten Importen und Instanziierungen folgt das Hauptprogramm, das aus nur wenigen Zeilen besteht:

```
lora_address = lora_init(uart1)
display.text(font2, 'LORA-ADR: ' + lora_address, 5, 10)
if lora_address == '???':
    display.text(font2, 'Resetten!', 5, 80, st7789.RED)
    print('Keine Verbindung zum LORA-Modul!')
    print('RESET-Knopf am TTGO betätigen!')
    sys.exit() # Programm beenden, wenn keine Antwort vom LoRa-Modul
print('Die Sende-Schleife kann mit Strg-C beendet werden')
print()

for zaehler in range(100, 1000):
```

```
print('jetzt senden: ', zaehler)
z_str = str(zaehler)
display.text(font2, 'senden: '+ z_str, 10, 60)
send_message(uart1, 123, z_str)  # Zahl an Addr 123 senden
print(read_response(uart1))
sleep(10)  # Wartezeit zwischen zwei Botschaften (zu klein???!!!)
print()
```

Beachten Sie, dass der Wert von `zaehler` in einen String konvertiert werden muss, um ihn auf dem Display ausgeben und als Botschaft senden zu können. Das vollständige Programm finden Sie auch in der Datei `Zahlen_Sender3c.py` (s. [MPD]).

Nachdem wir unser zweites LoRaWAN-Modul mit *HTerm* aktiviert haben, starten wir unseren Zahlen-Sender. Auf dem Monitor des Rechner sehen wir dann die empfangenen Zahlen (s. Abb. 4.1).

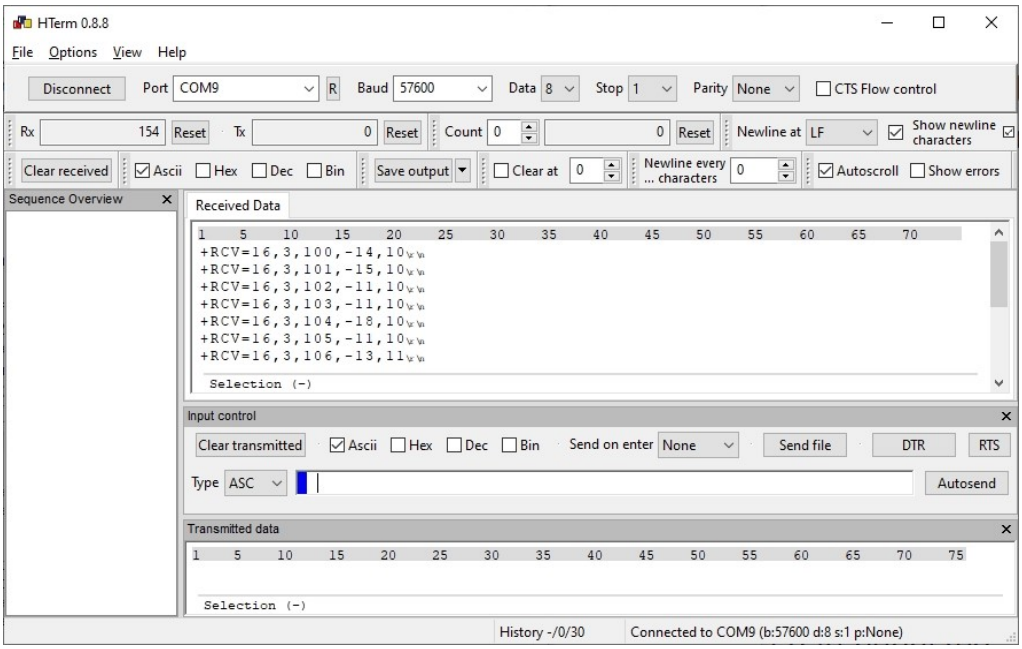


Abb. 4.1

Mit unserem Programm können wir nun einen einfachen Reichweiten-Test machen. Dazu speichern wir das Sende-Programm als `main.py` im Flash des TTGO ab. Während *HTerm* aktiv ist, schließen wir den TTGO an eine elektrische Quelle an und gehen damit durch verschiedene Räume des Hauses oder machen vielleicht auch einen kurzen Spaziergang.

```
+RCV=16,3,100,-17,10
+RCV=16,3,101,-36,10
+RCV=16,3,102,-45,11
+RCV=16,3,103,-59,10
+RCV=16,3,104,-70,10
+RCV=16,3,105,-78,10
+RCV=16,3,106,-75,10
+RCV=16,3,107,-66,10
+RCV=16,3,108,-53,10
+RCV=16,3,109,-35,10
+RCV=16,3,110,-16,10
```
Abb. 4.2

Bei meinem Gang vom Dachgeschoss (in der 2. Etage) in den Keller erhielt ich die Empfangswerte aus der Abb. 4.2. Dabei befand ich mich beim Zählerstand von 105 in der hintersten Ecke des Kellers (bei geschlossener Kellertür). Deutlich erkennt man, dass die *RSSI*-Werte auf dem Weg dorthin abnehmen und auf dem Rückweg wieder zuneh-

men. Tatsächlich steht die Abkürzung ***RSSI*** für **R**eceived **S**ignal **S**trength **I**ndicator; der *RSSI*-Wert ist also ein Maß für die Stärke des empfangenen Signals: Je niedriger dieser Wert ist, desto schwächer ist das empfangene Signal. Mit dem *RSSI*-Wert werden wir uns in Kapitel 10 noch eingehender beschäftigen.

Bei einem kleinen Spaziergang mit dem Sender in der Hand habe ich die *RSSI*-Werte von meinem Empfänger im Dachgeschoss aufzeichnen lassen: Der Weg war nicht ganz geradlinig. Es gab meist auch keine direkte Sichtverbindung zwischen Sender und Empfänger: U. A. standen einige Bäume mit dichtem Blattwerk zwischen Sender und Empfänger. Die Messung begann in einer Entfernung von ca. 520 m zum Empfänger und endete etwa 10 m vor dem Empfänger. Abb. 4.3 gibt den *RSSI*-Wert in Abhängigkeit von der Entfernung wieder; dabei ist nur jeder sechste Messwert in das Diagramm eingetragen worden.

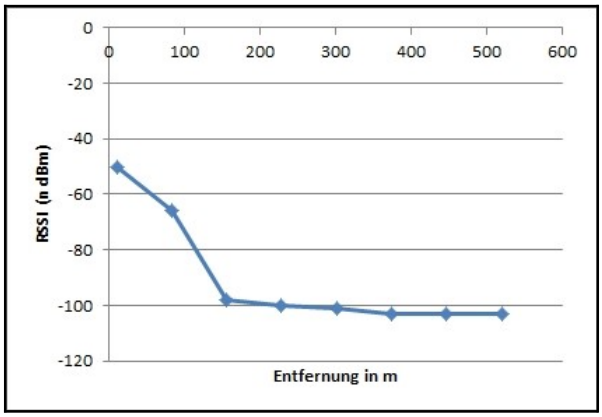


Abb. 4.3

5 Funkthermometer

Bislang hatten wir nur recht uninteressante Daten gesendet. Jetzt wollen wir den TTGO dazu bringen, dass er Temperaturwerte misst und diese dann per LoRa an unseren PC mit dem Terminal-Programm *HTerm* sendet.

Als Temperatursensor setzen wir hier den Baustein LM75BD ein. Diesen erhält man für wenige Euro schon fertig auf einer Platine montiert. Als I2C-Baustein besitzt er neben den Anschlüssen zur elektrischen Versorgung zwei Anschlüsse mit den Bezeichnungen **SDA** und **SCL** (s. u.). Über diese kann er mit unserem ESP32 Daten austauschen. Wie dies genau funktioniert, kann z. B. unter

Abb. 5.1

`www.g-heinrichs.de/attiny/I2C_Grundlagen_Attiny.pdf`

nachgelesen werden. Wir müssen hier nur wissen:

- Standardmäßig sind beim TTGO-Modul die Pins 21 und 22 für die I2C-Kommunikation vorgesehen. Wir schließen den LM75BD folgendermaßen an:

LM75BD	-	ESP32
SCL	-	22
SDA	-	21
GND	-	G
Vcc	-	3V

- An diese beiden Pins können mehrere I2C-Bausteine angeschlossen werden. Alle I2C-Bausteine besitzen eine so genannte **I2C-Adresse**. Diese besteht aus einer 7-Bit-Zahl.

- Der ESP32 steuert die I2C-Kommunikation: Er ist es, der über die I2C-Adresse den gewünschten I2C-Baustein aussucht; und er ist es auch, welcher bei dem adressierten Baustein Daten anfordert. In dieser Funktion bezeichnet man den ESP32 als **Master**. Der LM75BD wird hier als so genannter **Slave** eingesetzt.

- Alle Slaves, welche an einen Master angeschlossen werden, müssen unterschiedliche Adressen haben; nur so können sie vom Master gezielt angesprochen werden. Bei dem LM75BD-Baustein aus Abb. 1 kann man mit Hilfe von Lötbrücken die Adresse verändern; auf dieses Weise kann der Master auch mehrere I2C-Bausteine des gleichen Typs adressieren.

- Die Übertragung der Daten erfolgt seriell. Dabei steuert der Master mit einem **Taktsignal** an seinem SCL-Ausgang, wie rasch die einzelnen Daten-Bits über die SDA-Leitung wandern. Je höher die Taktfrequenz ist, desto schneller ist die Übertragung.

Micropython stellt im `machine`-Modul eine Klasse `I2C` zur Verfügung . Wir importieren sie, erzeugen damit ein I2C-Objekt und weisen es der Variablen `i2c` zu:

```
from machine import Pin, I2C
i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
i2c_addr = 0x48 # 7-Bit-Adresse im HEX-Format
```

Im Rahmen der Instanziierung haben wir auch die Pin-Zuordnung sowie die Taktfrequenz (in Hz) festgelegt. In der letzten Zeile haben wir die I2C-Adresse unseres I2C-Moduls in der Variablen `i2c_addr` gespeichert. In den Datenblättern von I2C-Bausteinen werden diese meist in hexadezimaler Form angegeben. Deswegen haben wir auch hier diese Form benutzt.

Nachdem der LM75BD eine Temperaturmessung durchgeführt hat, hält er das Messergebnis in seinem Speicher in Form von 2 Bytes (beginnend mit der Speicheradresse 0) bereit. Wir können diese beiden Bytes mittels der Methode `readfrom_mem(addr, memaddr, nbytes)` von `i2c` auslesen:

```
raw_val = i2c.readfrom_mem(i2c_addr, 0, 2)
```

Hierbei stellt `raw_val` einen Byte-String für den Rohwert dar, bestehend aus zwei Bytes. Das erste Byte (mit dem Index 0) gibt den ganzzahligen Anteil des Temperaturwertes an. Das zweite Byte (mit dem Index 1) gibt die "Nachkommastellen" des Temperaturwertes an, allerdings nicht in dezimaler Form. Vielmehr gibt es an, wie viele 256-tel noch zu dem ganzzahligen Wert hinzukommen. In dezimaler Form ist der Temperaturwert dann `raw_val[0] + raw_val[1]/256`. (Hinweis: Das Ergebnis von raw_val[...] ist vom Typ `int`!)

Die Funktion

```
def get_temp():
    raw_val = i2c.readfrom_mem(adr, 0, 2)
    temp_val = raw_val[0] + raw_val[1]/256
    return str(temp_val)
```

liefert uns damit als Rückgabewert eine Zeichenkette mit dem gemessenen Temperaturwert. Eigentlich müssen wir jetzt nur noch in dem Programm die angegebenen Programmzeilen einfügen und statt des Zähler-Wertes den Temperatur-Wert senden.

Bevor wir dies durchführen, wollen wir aber noch eine Vereinfachung einführen: Statt alle LoRa-spezifischen Funktionen immer wieder neu zu schreiben (oder aus der Datei `lora.py` des letzten Kapitels zu kopieren), speichern wir die Datei `lora.py` im Flash des TTGO.

Auf die Funktionen dieses `lora`-Moduls können wir dann zugreifen, wenn wir sie mit dem Befehl

```
from lora import send_command, read_response, response_to_list, lora_init,
                                                                send_message
```

importieren. Damit erhalten wir für das Funkthermometer folgendes Programm:

```
# Import und Instanziierungen:
from time import sleep
from machine import UART, SPI, Pin, I2C
import vga1_bold_16x32 as font2
import st7789
import sys
from lora import send_command, read_response, response_to_list, lora_init,
                                                                send_message

# Display:
spi = SPI(1, baudrate = 20_000_000, polarity = 1, sck = Pin(18),
                                                            mosi = Pin(19))
display = st7789.ST7789(spi, 135, 240, reset=Pin(23, Pin.OUT), cs=Pin(5,
Pin.OUT), dc=Pin(16, Pin.OUT), backlight=Pin(4, Pin.OUT), rotation=3)
display.init()
display.fill(0) # Display löschen

# UART:
uart1 = UART(1, baudrate = 57600, tx = 12, rx = 13)

# I2C:
i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)

# LM75BA
def get_temp():
    i2c_addr = 0x48  # HEX-Adresse (7-Bit, also ohne Read/Write-Bit)
    raw_val = i2c.readfrom_mem(i2c_addr, 0, 2)
    temp_val = raw_val[0] + raw_val[1]/256
    return str(temp_val)

# Hauptprogramm
lora_address = lora_init(uart1)
display.text(font2, 'LORA-ADR: ' + lora_address, 5, 10)
if lora_address == '???':
    display.text(font2, 'Resetten!', 5, 80, st7789.RED)
    print('Keine Verbindung zum LORA-Modul!')
    print('RESET-Knopf am TTGO betätigen!')
    sys.exit() # Programm beenden, wenn keine Antwort vom LoRa-Modul

print('Die Sende-Schleife kann mit Strg-C beendet werden')
print()
while True:
    t = get_temp()
    print('jetzt senden: ', t)
    display.text(font2, 'senden: ' + t + '  ', 10, 60)
    send_message(uart1, 123, t)  # Zahl an Addr 123 senden
    print(read_response(uart1))
```

```
    sleep(20)  # Wartezeit zwischen zwei Botschaften
    print()
```

Dieses Programm finden Sie auch in der Datei `Temperatur_Sender_1.py` (s. [MPD]). Wenn es stand-alone ausgeführt werden soll, muss es (wie immer) unter dem Namen `main.py` im Flash gespeichert werden.

Wer mag, kann jetzt das Programm `Empfaenger_3a.py` so abändern, dass es unsere Temperatur-Werte empfangen kann. Es ist sinnvoll, dabei wieder das Modul `lora.py` zu benutzen. Mit einem zweiten TTGO können die gesendeten Temperaturwerte dann auch stand-alone angezeigt werden.

6 Trägerfrequenzen, Kanäle, Bandbreiten

Rechtliches

Unser LoRa998-Modul kann mit verschiedenen Frequenzen arbeiten (vgl.[LD2]). Auf S. 5 finden wir die Angaben:

Minimum: 820 MHz
Typisch: 868-915 MHz
Maximum: 960 MHz

In der **ERC-Empfehlung 70-03** [ERC] finden wir auf S. 32 in der Rubrik *Non Specific Devices* (NSD) als zulässigen Frequenzbereich für Deutschland (D) unter Annex 1G die **Vorgabe 863 - 870 MHz**.

Der folgende Ausschnitt aus der Verfügung *Vfg 133/2019* (geändert durch die Verfügung *Vfg 12/2020)* der **Bundesnetzagentur** zeigt, dass für die verschiedenen Frequenzbereiche unterschiedliche Anforderungen gelten. So sehen wir z. B.: Wenn wir mit Frequenzen zwischen 865 MHz und 868 MHz arbeiten, brauchen wir uns um Frequenzzugangs- und Störungsminderungstechniken nicht zu kümmern, wenn wir einen Arbeitszyklus (Duty Cycle, s. u.) von maximal 1% verwenden.

Daneben gibt es auch noch weitere Einschränkungen für die EU: **Im Allgemeinen darf nur eine Bandbreite (s. folgenden Abschnitt!) von 125 kHz benutzt werden**. Lediglich für den Spreizfaktor 7 (vgl. Kap. 8) gibt es eine Ausnahme: Hier darf auch eine Bandweite von 250 kHz benutzt werden.

Außerdem darf die Sendeleistung maximal 14 dBm betragen (Vgl. Kap. 1).

868,0 - 868,6	25 mW	Es gelten Anforderungen an Frequenzzugangs- und Störungsminderungstechniken[7)] Alternativ kann ein maximaler Arbeitszyklus[2)] von 1% verwendet werden.
868,7 - 869,2	25 mW	Es gelten Anforderungen an Frequenzzugangs- und Störungsminderungstechniken[7)] Alternativ kann ein maximaler Arbeitszyklus[2)] von 0,1% verwendet werden.
869,40 - 869,65	500 mW	Es gelten Anforderungen an Frequenzzugangs- und Störungsminderungstechniken[7)] Alternativ kann ein maximaler Arbeitszyklus[2)] von 10% verwendet werden.

Abb. 6.1: Ausschnitt aus der Verfügung 133/2019 bzw. 12/2020

Arbeitszyklus (Duty Cycle)

Kümmern wir uns nun um den Arbeitszyklus. Um die gesetzlichen Bedingungen erfüllen zu können, müssen wir wissen, wie viel Zeit das Funksignal mit unserer Botschaft unterwegs ist; diese Zeit wird auch **Time on Air (ToA)** genannt. Nun gibt es Tabellen, in denen man die Datenübertragungsrate (in Bit pro s) in Abhängigkeit von der jeweiligen Konfigurationen des LoRa-Moduls erfahren kann. In unserem Fall wäre das 1760 bit/s. Diese Information hilft uns aber nicht weiter, weil wir

nicht wissen, wie viele Bytes tatsächlich von unserem Modul übertragen werden. Sicherlich sind es deutlich mehr als die Bytes vom Payload. Denn neben diesem Payload werden zahlreiche weitere Daten gesendet, z. B. die Adresse vom Empfänger und vom Sender. Abb. 6.2 zeigt das LoRa-Frame-Format. Ohne ins Detail zu gehen, sehen wir auf einen Blick, dass zahlreiche weitere Daten übertragen werden. Die eigentliche Nachricht, welche wir in diesem Skript fortan häufig als **User- oder Nutzer-Payload** bezeichnen, finden wir hier übrigens im Application Layer unter der Bezeichnung **Frame Payload**.

Deswegen beschreite ich hier bei der Abschätzung der Sendezeit zunächst einen experimentellen Weg. Hierzu benutze ich die gleiche Konfiguration wie in Kap. 3: Als Sender benutze ich ein LoRa-Modul, welches seine Kommandos vom Programm *HTerm* erhält; ich bezeichne es im Folgenden als Modul A. Als Empfänger wird ein LoRa-Modul zusammen mit dem TTGO betrieben (im Folgenden bezeichne ich es als Modul B); als Empfangs-Programm benutze ich `Empfaenger_3b.py` .

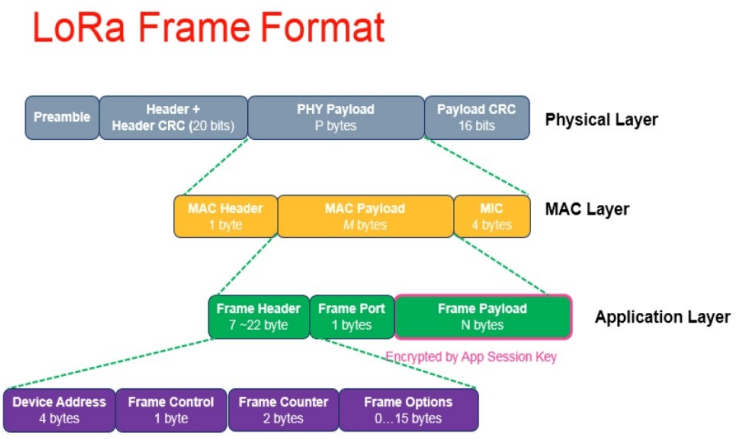


Abb. 6.2: Das LoRa Frame Format (nach [LFF])

Die Idee ist jetzt: Ich **messe** die Zeitspanne zwischen folgenden Ereignissen:

1. Ereignis: Das LoRa-Modul A hat gerade das **letzte Bit** des AT-Commands (z. B. AT+SEND=16,5,HALLO) erhalten.

2. Ereignis: Das LoRa-Modul B gibt gerade das **erste Bit** der Botschaft an den TTGO ab.

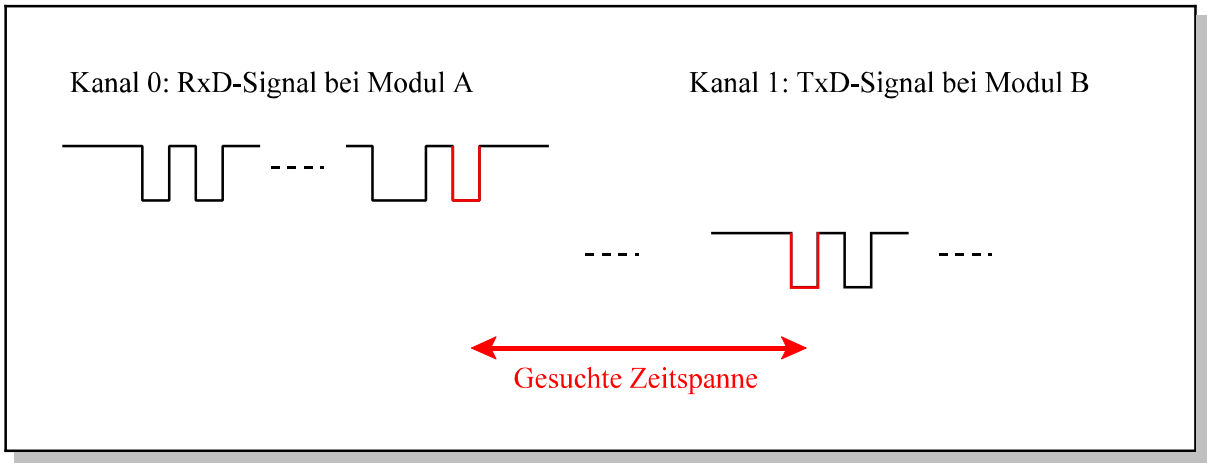


Abb. 6.3

Die so ermittelte Zeitspanne ist größer als die ToA, wenn wir davon ausgehen, dass unser LoRa-Modul neben dem Senden bzw. Empfangen weitere Aufgaben (wie z. B. die Demodulation, vgl. Kap. 8) nicht schon während des Sendens bzw. Empfangens ausführt. Wenn wir die von uns gemessene Zeitspanne dann als Schätzwert für die tatsächliche ToA benutzen, können wir damit den Duty-Cycle näherungsweise berechnen. (Mehr zur Bestimmung der ToA finden Sie in Kap. 9.)

Wie geht man nun bei dieser Messung konkret vor? Den RxD-Anschluss von Modul A verbinde ich zusätzlich mit dem Kanal 0 eines Logic-Analyzers (z. B. Saleae Logic), den TxD-Anschluss mit dem Kanal 1. Die Massen von beiden Modulen schließe ich dann noch an die Masse des Analyzers an. Getriggert wird der Analyzer über Kanal 0.

Zunächst wird das Empfangsprogramm auf dem TTGO (LoRa-Modul B) gestartet und dann die Aufzeichnung der Signale beim Analyzer-Programm. Direkt im Anschluss geben wir ein SEND-Kommando eingegeben, z. B.

AT+SEND=16,5,HALLO

In diesem Fall stelle ich fest: Das letzte Bit wird vom Modul A ca. 3 ms nach dem Triggern empfangen, und vom Modul B wird das erste Bit ca. 172 ms nach dem Triggern an den TTGO gesendet. Die Zeitspanne beträgt damit ungefähr 170 ms. Weitere Messungen führen zu dem Diagramm in Abb. 6.4. Es zeigt, dass hier in etwa ein linearer Zusammenhang vorliegt. Aus der angegebenen Geradengleichung können wir schließen: Allein bei einem Nutzer- oder User-Payload von 0 Zeichen kämen wir auf eine ToA von etwa. 140 ms. Pro Payload-Byte kommen dann etwa 5 ms dazu.

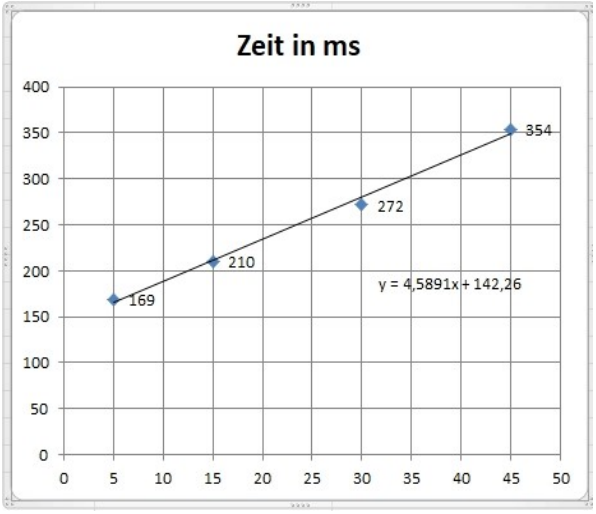


Abb. 6.4

Was bedeutet das z. B. konkret für unser Funkthermometer-Programm aus Kap. 5? Dort hatten wir zwischen zwei Messungen 20 Sekunden vergehen lassen. Die Zeichenketten für die Temperaturwerte sind durchschnittlich etwa 5 Byte lang. Die Dauer des Funksignals für eine einzige Übertragung beträgt ungefähr 170 ms. Der Arbeitszyklus ist 100 mal so lang, also ca. 17 Sekunden. Demnach müssen wir für die nächste Übertragung dann **höchstens** 100 mal so lange warten, das entspricht etwa 17 s.

Bandbreite, Kanäle, Mittenfrequenz

Stellen wir uns vor: Wir betreiben unseren LoRa-Sender wie immer mit einer Frequenz f_1 = 868,50 MHz. Mein Nachbar hat sich ebenfalls ein LoRa-Modul zugelegt. Um Interferenzen zu vermeiden, möchte er sein Modul nicht mit derselben Frequenz betreiben. So wählt er für seinen Sender eine andere Frequenz, z. B. f_2 = 868,45 MHz. Obwohl unsere beiden Frequenzen unterschiedlich sind, kommt es zu Störungen. Woran liegt das?

Tatsächlich werden neben den Wellen mit der eingestellten **Trägerfrequenz** auch Wellen mit etwas größerer oder kleinerer Frequenz ausgesendet. Es handelt sich hier nicht etwa um eine Unzulänglichkeit der Module. Der Grund ist vielmehr: Um Nachrichten mit dem Trägersignal zu übermitteln, müssen wir dieses modulieren. Dies kann z. B. erfolgen, indem wir den Sender ein- und ausschalten (Morsezeichen). Das Signal entspricht in diesem Fall nicht einer endlosen Sinus-Schwingung (Abb. 6.5 A), sondern besteht aus endlich vielen Schwingungen (Abb. 6.5 B). Hier sehen wir in dem eingeschalteten Zustand 4 Schwingungen; im Fall unserer LoRa-Übertragung sind es in Wirklichkeit viele Tausende von Schwingungen.

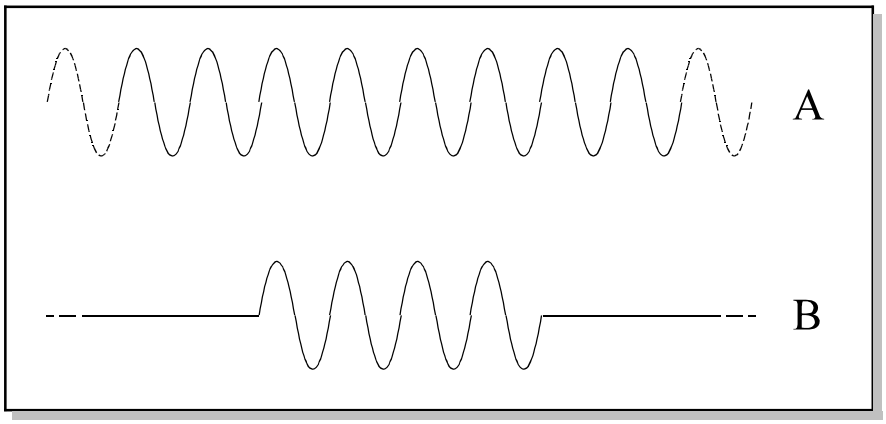


Abb. 6.5

Mit Hilfe einer so genannten **Fouriertransformation** kann man ausrechnen, welche zusätzlichen Frequenzen mit welcher Intensität im Fall des Ein-und Ausschaltens auftauchen; wer mag, kann sich die zugehörige Rechnung in der Fourier-Box am Ende dieses Kapitels anschauen. Das Ergebnis ist in Abb. 6.6 wiedergegeben. Die horizontale Achse gibt die Frequenz (in Hz) an, und auf der vertikalen Achse können wir die **Strahlungsleistung** pro Frequenz-Intervall ablesen; man spricht hier auch von einem **Spektrum**. Wir erkennen sofort, dass das Signal nicht eine einzige Frequenz besitzt, sondern ein Gemisch (Überlagerung) von Schwingungen mit unterschiedlicher Frequenz darstellt. Dabei sind die Wellen/Schwingungen mit Frequenzen nahe der Trägerfrequenz dominant. Wir sehen aber auch, dass es deutlich sichtbare Beiträge gibt durch Wellen/Schwingungen, deren Frequenz um mehr als ca. 50 kHz von der Trägerfrequenz abweicht. Genau solche Beiträge sind es, die dazu führen, dass es zu Interferenzen zwischen den Signalen von meinem LoRa-Modul und dem meines Nachbarn führen (Abb. 6.7).

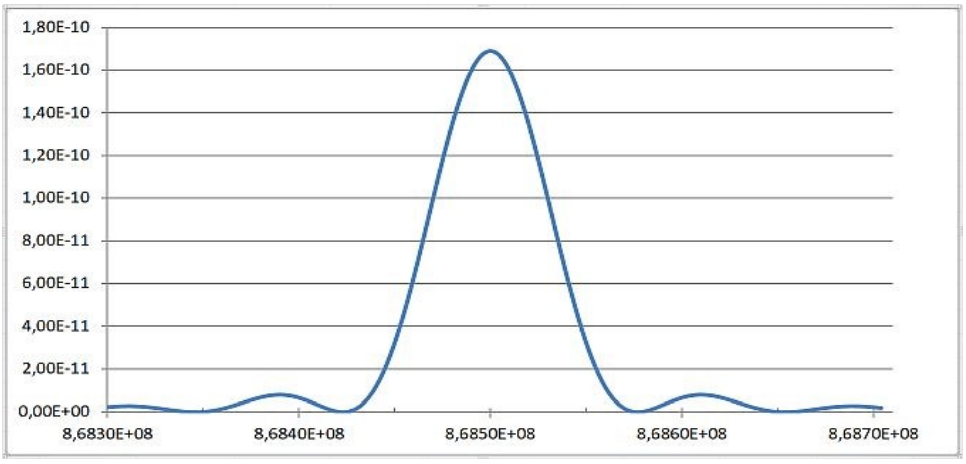


Abb. 6.6: Spektrum von "meinem" Lora-Modul

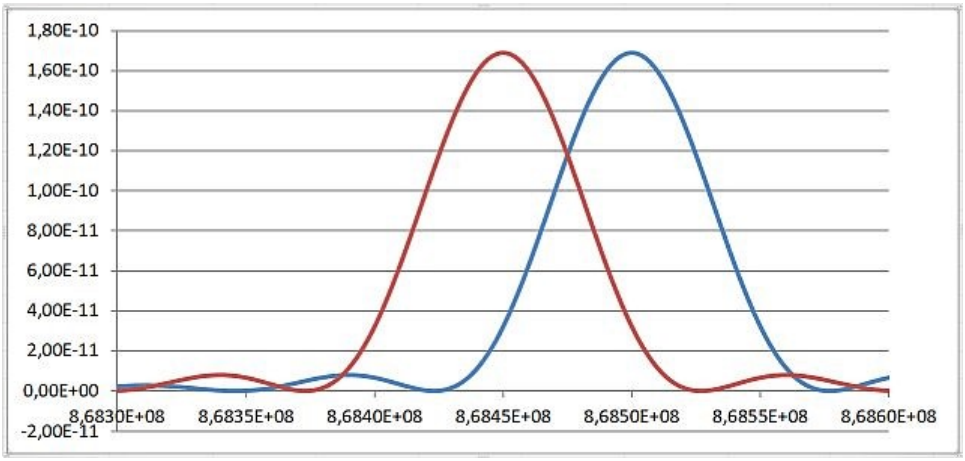


Abb. 6.7: Spektrum des "Nachbarns" (rot) und meins (blau)

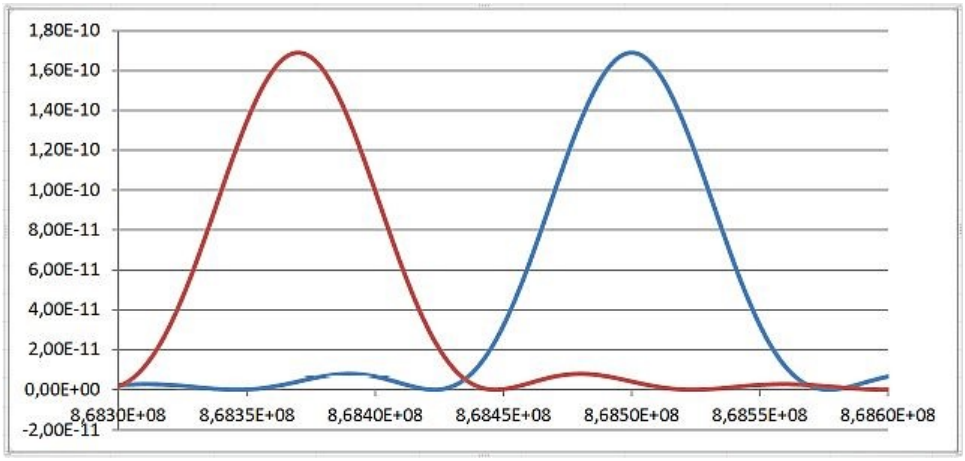


Abb. 6.8: Beide Spektren. Die Trägerfrequenzen unterscheiden sich um 125 kHz.

Um derartigen Problemen aus dem Wege zu gehen, hat man bestimmten Trägerfrequenzen sogenannte **Kanäle** (channel) zugeordnet. Hierbei handelt es sich um "Schutzbereiche". Die Größe dieses Schutzbereichs wird durch die so genannte **Bandbreite** (band width, kurz: **BW**) beschrieben. In diesem Fall beträgt sie 125 kHz. Mein LoRa-Modul hat den Frequenz-Bereich $[f_1 - BW/2;\ f_1 + BW/2]$ zur Verfügung, geht also von 868,438 MHz bis 868,562 MHz. Dieser hat übrigens die Kanalnummer 2. Die Frequenz f_1 wird in diesem Zusammenhang auch als **Mittenfrequenz** bezeichnet. Mein Nachbar muss dann einen Frequenz-Bereich $[f_2 - BW/2;\ f_2 + BW/2]$ benutzen, der meinen nicht schneidet (Abb. 6.8). **Seine Trägerfrequenz bzw. Mittenfrequenz muss also mindestens 125 kHz von unserer entfernt sein.** Eine Frequenz von 868,45 MHz oder selbst von 868,4 MHz würde also nicht ausreichen! Am besten würde unser Nachbar gleich die Trägerfrequenz von 868,3 MHz wählen; diese ist übrigens dem Kanal 1 zugeordnet.

Standardmäßig arbeitet unser LoRa-Modul mit einer Bandbreite von 125 kHz. In späteren Kapiteln werden wir noch sehen, dass es in manchen Situationen sinnvoll sein kann, mit anderen Bandbreiten zu arbeiten; unser LoRa-Modul kann dazu dann passend konfiguriert werden.

Dem Diagramm aus Abb. 6.3 liegt ein "Morsesignal" mit der Dauer $\Delta t = 130\ \mu s$ zugrunde. Verdoppelt man diese Zeit, so wird das zugehörige Spektrum schmaler: In horizontaler wird es um den Faktor 2 gestaucht. Das bedeutet insbesondere, dass sich dann der Abstand Δf zwischen den beiden Minimalstellen links und rechts von der Maximalstelle halbiert. Ganz allgemein gilt also:

$$\Delta f \sim \frac{1}{\Delta t}$$

Wer will, kann dies an weiteren Beispielen selbst an Hand der EXCEL-Tabelle "Fouriertransformation.xlsx" (s. [FTS]) überprüfen. Umgekehrt kann man nun schließen: Je kürzer das "Morsesignal" ist, desto größer ist die benötigte Bandbreite. Multiplizieren wir $\Delta f = 125$ kHz mit $\Delta t = 130$ ms, so erhalten wir einen Wert von ungefähr 1 : $\Delta f \cdot \Delta t \approx 1$. Dies gilt wegen der oben festgestellten Proportionalität allgemein.

Wir merken uns: Um eine Struktur von der zeitliche Länge Δt durch eine Überlagerung von Sinusschwingungen darzustellen, ist eine Bandbreite von

$$\Delta f \approx \frac{1}{\Delta t}$$

erforderlich. Dieser Zusammenhang besteht auch, wenn wir es mit komplexeren Signalen zu tun haben.

Sparen, sparen, sparen...

... das ist eine wichtige LoRa-Devise. Der Standardwert für die Bandbreite bei unserem LoRa-Modul ist gleichzeitig auch der kleinste Wert, mit dem das Modul arbeiten kann. Ein kleiner Bandbreitenwert sorgt dafür, dass in dem zulässigen Frequenzbereich viele Kanäle zur Verfügung stehen. Gleichzeitig sorgt der Duty Cycle von 1% dafür, dass Sendezeit gespart wird: Auf diese Weise steht ein und derselbe Kanal möglichst vielen Anwendern zur Verfügung.

Fourier-Box: Unsere Signal-Funktion aus Abb. 6.2B beschreiben wir mit der komplexwertigen Funktion

$$s(t) = \begin{cases} A\,e^{i\,2\pi\,f_0\,t} & \text{für } -\frac{t_0}{2} \le t \le \frac{t_0}{2} \\ 0 & \text{sonst} \end{cases}$$

Dabei sind:

A die Amplitude
f_0 die Trägerfrequenz
t_0 die Signaldauer

Die Fouriertransformierte $\hat{s}(f)$ berechnet sich dann so:

$$\begin{aligned} \hat{s}(f) &= \int_{-\infty}^{+\infty} s(t)\; e^{-i\,2\pi\,f\,t}\,dt \\ &= \int_{-t_0/2}^{+t_0/2} A\; e^{i\,2\pi\,f_0\,t}\; e^{-i\,2\pi\,f\,t}\,dt \\ &= A \int_{-t_0/2}^{+t_0/2} e^{i\,2\pi\,(f_0 - f)\,t}\,dt \\ &= A \left[\frac{e^{i\,2\pi\,(f_0 - f)\,t}}{i\,2\pi\,(f_0 - f)} \right]_{-t_0/2}^{+t_0/2} \\ &= A\; \frac{\sin(\pi\,(f_0 - f)\,t_0)}{\pi\,(f_0 - f)} \end{aligned}$$

Die Strahlungsleistung pro Frequenz-Intervall ist dann $(\hat{s}(f))^2$. In Abb. 3 ist der Graph von $(\hat{s}(f))^2$ für $A = 1$, $t_0 = 2 \cdot 10^{-5}$ s, $f_0 = 868{,}5$ MHz dargestellt.

Hinweis: Auf den ersten Blick scheint $\hat{s}(f)$ eine Polstelle bei $f = f_0$ zu besitzen. Tatsächlich kann die Funktion $\hat{s}(f)$ aber mit Hilfe der Regel von de l' Hospital stetig nach f_0 fortgesetzt werden:

$$\lim_{f \to f_0} \hat{s}(f) = \lim_{f \to f_0} \frac{\sin(\pi\,(f_0 - f)\,t_0)}{\pi\,(f_0 - f)} = \lim_{x \to 0} \frac{\sin(x \cdot t_0)}{x} = \lim_{x \to 0} \frac{\cos(x \cdot t_0) \cdot t_0}{1} = t_0$$

7 Netzwerke und Verschlüsselung

Wenn wir Daten mit dem LoRa-Modul senden, müssen wir uns dessen bewusst sein, dass diese im Prinzip von jedem LoRa-Empfänger gelesen werden können, der gerade in der Nähe ist und mit demselben Kanal arbeitet. Den Kreis der Empfänger können wir auf zwei Weisen einschränken: Wir können das Standardnetzwerk mit der ID 18 ändern, z. B. auf den Wert 7. Nun können nur noch mit solchen LoRa-Modulen über unser Modul kommunizieren, welche ebenfalls diese Netzwerk-ID besitzen. Eine deutlich größere Einschränkung (und damit auch eine erheblich größere Sicherheit) erreichen wir, indem wir die "Botschaft" mit einem Passwort verschlüsseln.

Netzwerke

Neben dem standardmäßigen Wert 18 stehen als Netzwerk-ID noch die Zahlen 3 bis 15 (einschließlich) zur Verfügung. Um die Netzwerk-ID z. B. auf den Wert 7 einzustellen, benutzt man das Kommando:

AT+NETWORKID=7

Das Modul quittiert mit der Meldung **+OK**. Mit dem Kommando AT+NETWORKID? kann man die aktuelle Netzwerk-ID abfragen.

Von nun an kann das LoRa-Modul nur noch mit solchen Modulen kommunizieren, welche auch diese Netzwerk-ID haben.

Verschlüsselung mit einem Passwort

Als Passwörter werden HEX-Zahlen zwischen 0x00000001 und 0xffffffff benutzt. Dies entspricht PINs zwischen 1 und 4.294.967.296. Achten Sie darauf, dass Sie das Passwort hexadezimal mit großen Buchstaben eingeben. Für die Hexadezimalzahl 0xeb0a5cc2 sieht das dann so aus:

AT+CPIN=EB0A5CC2

Damit eine Kommunikation mit einem anderen LoRa-Modul zustande kommen kann, muss dieses auch dasselbe Passwort benutzen. Aus Sicherheitsgründen sollte dieses Passwort ***unbedingt*** auf einem anderen Weg ausgetauscht werden!

Standardmäßig arbeitet das LoRa-Modul nicht mit einem Passwort. Das Passwort können wir mit

AT+CPIN?

abfragen. Ist kein Passwort festgelegt worden, antwortet das LoRa-Modul mit

+CPIN=No Password!

Das Passwort 00000000 ist nicht zulässig; es ist auch **nicht** gleichbedeutend damit, dass kein Passwort festgelegt wurde. Dementsprechend lässt sich die Verschlüsselung mit einem Passwort auch nicht abstellen, indem man einfach das Kommando AT+CPIN=00000000 eingibt. Um den Passwortschutz wieder auszuschalten, muss vielmehr ein Reset durchgeführt werden. Dies geschieht mit dem Kommando:

AT+RESET

Das Modul antwortet dann mit:

+READY

8 Chirps und Chips

*"Endgeräte und Gateways im LoRaWAN nutzen ein proprietäres und patentiertes Übertragungsverfahren, basierend auf einer **C**hirp-**S**pread-**S**pectrum-Modulationstechnik mit der Bezeichnung „LoRa" der Semtech Corporation."* So beginnt der Wikipedia-Artikel zum Thema LoRa [LRW]. Und wie es für solche Beiträge typisch ist, tauchen die wichtigsten Informationen direkt im ersten Satz auf:

- Es geht um ein von der Semtech Corporation patentiertes Übertragungsverfahren.
- Dieses Verfahren beruht auf einer Modulationstechnik, bei der so genannte Chirps benutzt werden.
- Dabei spielt ein Spreizfaktor (spread factor) eine wichtige Rolle.

Diese Modulationstechnik, die aus der Radartechnik stammt, werden wir nun studieren. Für eine eingehende Behandlung wären gründliche Kenntnisse aus der Signalübertragungstechnik erforderlich. Hier möchte ich einen Weg beschreiten, der einerseits die Zusammenhänge möglichst anschaulich darstellt, allerdings aber auch wichtige quantitative Aussagen ermöglicht.

Amplituden- und Frequenz-Modulation

Ein Radio empfängt elektrische Schwingungen mit Trägerfrequenzen zwischen einigen hundert kHz und etwa 100 MHz. Diese Schwingungen werden vom Sender mit den Tonsignalen "moduliert". Bei der **Amplituden-Modulation (kurz: AM)** wird die Amplitude der hochfrequenten elektrischen Schwingungen durch das niederfrequenteTonsignal verändert, die Frequenz der elektrischen Schwingung bleibt dabei gleich (vgl. Abb. 8.1 und [AMO]). Diese Modulationstechnik wird hauptsächlich bei Mittelwellen- und Kurzwellenradios eingesetzt; in Deutschland wird seit einigen Jahren im Mittelwellenbereich praktisch nicht mehr gesendet. Unsere Morsezeichen-Modulation aus dem Kapitel 5 stellt eine spezielle, sehr einfache Form dieser Modulationsart dar: Hier gibt es nur die Signalwerte 1 und 0.

Bei der **Frequenz-Modulation (kurz: FM)** ist es genau umgekehrt wie bei der Amplituden-Modulation: Hier bleibt die Amplitude der hochfrequenten Schwingung gleich, während ihre Frequenz gemäß dem Tonsignal variiert wird. Diese Modulationsart finden wir auch heute noch bei Ultrakurzwellen-Radios, kurz UKW-Radios. (Mehr dazu unter [FMO])

Damit wir die ursprünglichen Töne hören können, muss eine so genannte **Demodulation** stattfinden. Im Fall der Amplituden-Modulation gelingt das mit Hilfe einer Diode: Diese lässt den Strom nur in eine Richtung hindurchfließen. Nach dieser Gleichrichtung sieht das Signal dann wie in Abb. 8.1 links unten (blauer Graph) aus. Betreibt man mit diesem Strom (ggf. nach Verstärkung) einen Kopfhörer oder einen Lautsprecher, so wird dessen Membran wie in Abb. 8.1 links unten (roter Graph) schwingen; die Membran ist wegen ihrer Trägheit nämlich nicht in der Lage, die hochfrequenten Schwingungen auszuführen. Damit schwingt die Membran genauso wie das Ton-Signal, mit welchem die Modulation durchgeführt worden ist. (Mehr dazu unter [AMO])

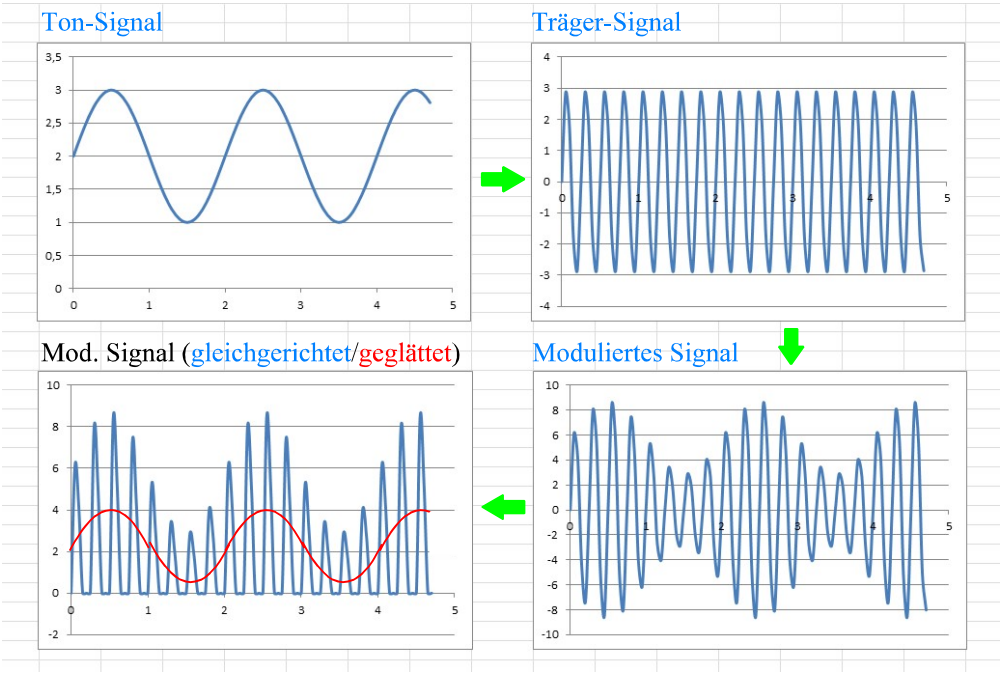


Abb. 8.1: Amplituden-Modulation und Demodulation

Chirps

Unter einem **Chirp** versteht man ein kurzes Signal, welches seine Frequenz gleichmäßig erhöht (vgl. auch [ACF]). In der Abb. 8.2 beginnt das Signal z. B. mit einer Frequenz von 2 Hz; sie wird dann innerhalb von 1 s gleichmäßig auf 10 Hz erhöht. Die Amplitude bleibt dabei konstant. Ihr Wert spielt für die folgenden Betrachtungen keine Rolle; deswegen können wir von einem Amplitudenwert 1 ausgehen. Die Formel für dieses Signal lautet:

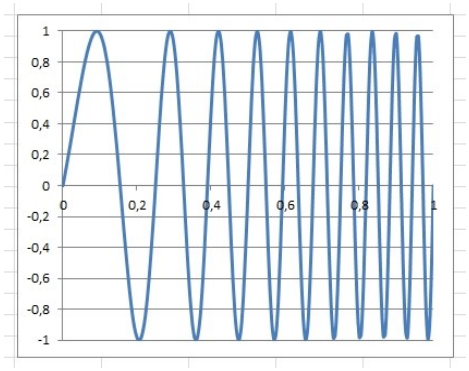


Abb. 8.2

$s(t) = \sin(2\,\pi\,(2\ \text{Hz} + 8\ \text{Hz/s} \cdot t) \cdot t)$.

Das Argument der Sinus-Funktion wird auch als Phasen winkel φ bezeichnet; in diesem Fall ist

$\varphi(t) = 2\,\pi\,(2\ \text{Hz} + 8\ \text{Hz/s} \cdot t) \cdot t$.

Deutlich einfacher lässt sich unser Chirp aus der Abb. 2 durch sein *f*-*t*-Diagramm beschreiben: Es handelt sich um ein Rampensignal, welches in der Zeit von $T = 1$ s von $f_{min} = 2$ Hz auf $f_{max} = 10$ Hz ansteigt.

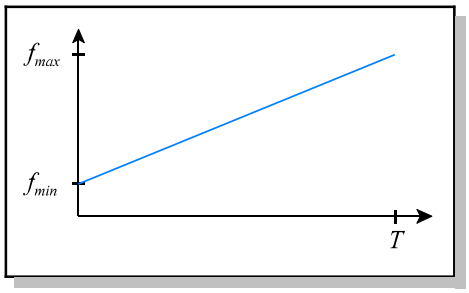


Abb. 8.3

Bei unserem Modell-Chirp aus Abb. 8.2 bzw. 8.3 ändert sich die Frequenz relativ stark: Bezogen auf die mittlere Frequenz von f_m = 5 Hz beträgt die relative Änderung $\frac{\Delta f}{f_m} = \frac{8\ \text{Hz}}{5\ \text{Hz}} = 160\ \%$.

Bei einem LoRa-Chirp fällt diese Änderung wesentlich kleiner aus. Hier gilt nämlich:

$$f_{min} = f_M - \frac{BW}{2}$$

$$f_{max} = f_M + \frac{BW}{2}$$

Dabei bezeichnet f_M wieder die Mitten- (bzw. Träger-) Frequenz und BW die Bandbreite (vgl. Kap. 6). In diesem Fall ist die relative Änderung der Frequenz $\frac{BW}{f_M}$; für die von uns benutzten Frequenz von 868,5 MHz und eine Bandbreite von 125 kHz beträgt sie etwa 0,014 %. In einer Darstellung des Chirps wie in Abb. 8.2 würden wir die Frequenzänderung mit dem Auge nicht wahrnehmen können.

Chirp-Modulation

Ein LoRa-Chirp muss nicht unbedingt bei der niedrigsten Frequenz f_{min} beginnen. In Abbildung 8.3 ist eine Folge von Chirps zu sehen: Sie beginnen mit unterschiedlichen Startfrequenzen, besitzen aber alle die gleiche Änderungsrate bei der Frequenz. Wenn die Obergrenze f_{max} erreicht ist, gibt es einen Sprung zu f_{min} und die Frequenz wird von da an wieder mit derselben Änderungsrate erhöht. Die Dauer T_M der einzelnen Chirps ist immer gleich groß.

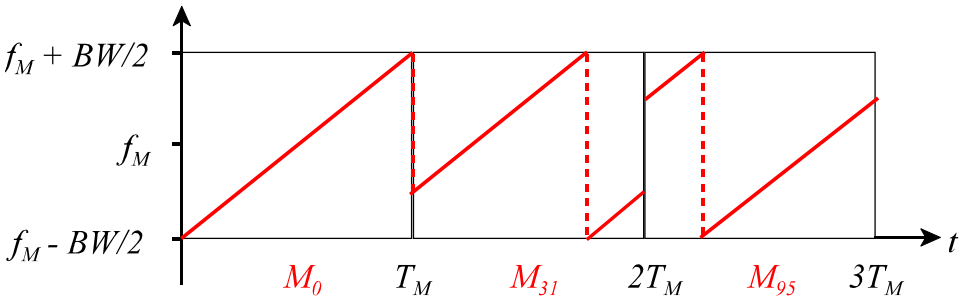


Abb. 8.3: Folge von 3 Chirps mit unterschiedlichen Startfrequenzen (SF = 7)

Die Graphen der verschiedenen Chirps gehen im Prinzip durch eine horizontale Verschiebung auseinander hervor; dabei wird das links fehlende Graphenstück rechts von der Sprungstelle angefügt.

Mit Hilfe der Anfangswerte der Frequenz werden nun Informationen kodiert. Die einzelnen Informationen werden als **Symbole** bezeichnet. Sie können z. B. für Zahlen, Buchstaben oder auch Wörter stehen. Wie viele solcher Anfangswerte unser LoRa-Modul unterscheiden kann, hängt von dem so genannten Spreizfaktor SF ab. Dieser Spreizfaktor kann (in bestimmten Grenzen) durch AT-Commands festgelegt werden (s. Kap. 9).

Die Anzahl M der möglichen Anfangswerte ergibt sich aus der Formel:

$$M = 2^{SF}$$

Dies folgt direkt aus der **Definition des Spreizfaktors**: $SF = \log_2(M)$

Bei dem Spreizfaktor von 7 können also 128 verschiedene Symbole M_0, M_1, M_2, ..., M_{127} übertragen werden. In der Abb. 8.3 sind die Chirps für die Symbole M_0, M_{31} und M_{95} zu sehen. Der Wert der Startfrequenz f_i für das Symbol M_i ergibt sich aus der Formel:

$$f_i = f_M - \frac{f_{BW}}{2} + i \cdot \frac{f_{BW}}{M} \qquad \text{für } i = 0, 1, 2, \ldots M-1$$

Benachbarte Symbole unterscheiden sich in der Startfrequenz also jeweils um den Wert f_{BW}/M.

Chips

Jedes Chirp kann in M gleich große **Chips** unterteilt werden. Jedes Chirp besteht dann aus einer Kette von Chips; erreicht die Kette den Frequenzwert $f_M + BW/2$, bricht sie ab und beginnt dann neu mit dem Frequenzwert $f_M - BW/2$, vgl. Abb. 8.4.

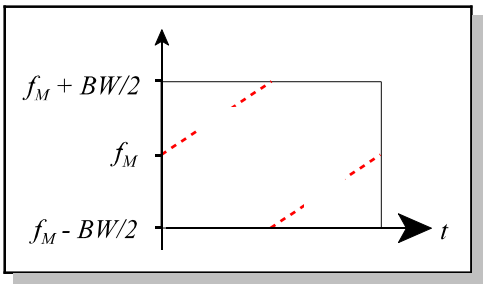


Abb. 8.4: Chips (nicht maßstäblich)

Die Chip-Dauer T_C kann nicht beliebig klein gewählt werden. In dieser Zeit muss das LoRa-Modul nämlich die Frequenz um $\Delta f = BW$ ändern können. Nach den Überlegungen aus Kapitel 6 muss dafür eine Zeit von ungefähr $\frac{1}{\Delta f} = \frac{1}{BW}$ zur Verfügung stehen. Und tatsächlich benutzt LoRa genau diesen Wert $1/BW$ für die Chip-Dauer T_C[1].

Bei der Bandbreite von 125 kHz ist die Chip-Dauer T_C dann $\frac{1}{BW} = 8\ \mu s$. Damit hat unser Chirp mit dem Spreizfaktor 7 eine Dauer von $M \cdot 8\ \mu s = 128 \cdot 8\ \mu s = 1024\ \mu s \approx 1\ ms$.

Wir merken uns: Die Chirp-Dauer T_{Chirp} ist um so größer, je größer der Spreizfaktor SF ist und je kleiner die Bandbreite ist. Allgemein gilt:

$$T_{Chirp} = 2^{SF} \cdot T_C = 2^{SF} \cdot 1/BW$$

[1] In dem LoRa-Dokument AN 1200.22 [SE1] findet man auf der Seite 10 eine äquivalente Formulierung für die Chip-Rate R_C: $R_C = BW$. Die dort angegebene Herleitung ist m. E. aber unvollständig, weil sie die dabei benutzte Gleichung für die Bit-Rate nicht begründet.

Demodulation

Wie kann der Empfänger nun den empfangenen Chirps die zugehörigen Symbole zuordnen? Anschaulich gesehen kann er dazu deren *f*-*t*-Diagramme der Reihe nach mit solchen vergleichen, die er aufgrund der bekannten Parameter f_M, *SF* und *BW* selbst konstruieren kann.

Wie könnte dieser Vergleich geschehen?

Die Frequenzwerte der einzelnen Chips für das empfangene Symbol *e* und das konstruierte Symbol *k* werden in einem Vektor $e = (e_0, e_1, e_2, ..., e_{M-1})$ und einem Vektor $k = (k_0, k_1, k_2, ..., k_{M-1})$ zusammengefasst. Von diesen wird das Skalarprodukt gebildet; je größer es ist, desto besser passen die beiden Symbole zusammen. In [KOR] finden Sie eine anschauliche Darstellung dazu. Diese Vorgehensweise ist jedoch recht zeitaufwendig. LoRa benutzt ein deutlich schnelleres Verfahren. Dieses näher zu erläutern, würde den Rahmen dieser Einführung sprengen. Wer hierzu mehr erfahren möchte, sei auf [HOW] und [R/P] verwiesen.

Egal wie der Vergleich durchgeführt wird: Damit die Zuordnung korrekt durchgeführt werden kann, müssen Empfangssignal und Vergleichssignal **synchronisiert** sein, d. h. die Startzeiten der beiden Signale müssen übereinstimmen; das bedeutet insbesondere, dass die beiden M_0-Chirps von Sendesignal **und** Vergleichssignal phasengleich sind, d. h. gleichzeitig mit der Frequenz $f_M - BW/2$ beginnen. Um dies zu erreichen, wird vor den Signalen des Payloads eine so genannte **Präambel** gesendet. Diese besteht zunächst aus einer Reihe von M_0-Chirps (meistens 8, bei unserem LoRa-Modul jedoch standardmäßig 12), dann zwei weiteren Chirps[2], gefolgt von 2,25 **Down-Chirps**: Diese Down-Chirps weisen (im Gegensatz zu den bislang betrachteten **Up-Chirps**) eine **fallende Frequenz** auf (vgl. Abb. 8.5); sie markieren das Ende der Präambel. An diese schließen sich unmittelbar die Daten-Chirps an. (Eine anschauliche Animation finden Sie im Video [ANI], s. Anhang.) Man beachte dabei aber, dass zu diesen Daten-Chirps nicht nur unser (Nutzer-) Payload, sondern auch noch weitere Informationen gehören; darauf wurde bereits in Kapitel 6 hingewiesen. Im nächsten Kapitel werden Sie hierzu noch mehr erfahren.

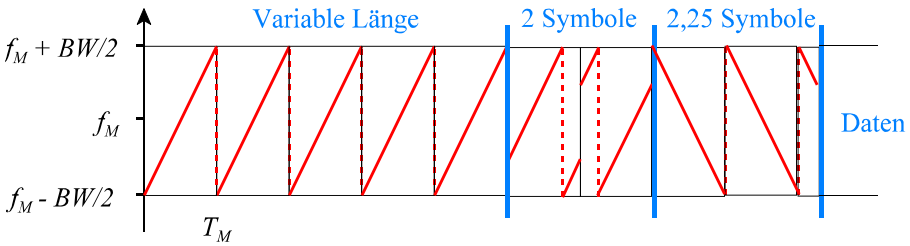


Abb. 8.5: LoRa-Präambel (hier mir 5 Synchronisierungs-Chirps)

Derartige Präambeln werden übrigens sehr häufig zur Synchronisierung benutzt, z. B. bei den Funkmodulen vom Typ rfm12 (siehe z. B.: [RFM]); hier haben sie allerdings eine einfachere Form.

[2] Diese beiden Chirps werden auch als **Sync Word** bezeichnet; sie spielen bei Netzwerken eine Rolle. Wir gehen hier nicht weiter auf sie ein.

9 LoRa-Signale messen und deuten

Signale studieren mit CubicSDR und RTL2832

Um die LoRa-Signale untersuchen zu können, benötigen wir zunächst einen Empfänger für den entsprechenden Frequenzbereich. Hier werden wir dafür das USB-Modul NEW GEN.**RTL2832** (Abb. 9.1) einsetzen. Sodann benötigen wir eine Software, welche die empfangenen Signale aufbereiten und entsprechend darstellen kann; insbesondere möchten wir ein *t-f*-Diagramm ähnlich wie in Abb. 8.5 erhalten. Dazu gibt es eine Reihe von Programmen; ich benutze hier das Programm ***CubicSDR***. Informationen zur Installation dieses Programms und unseres RTL2832-Moduls findet man im Anhang.

Abb. 9.1

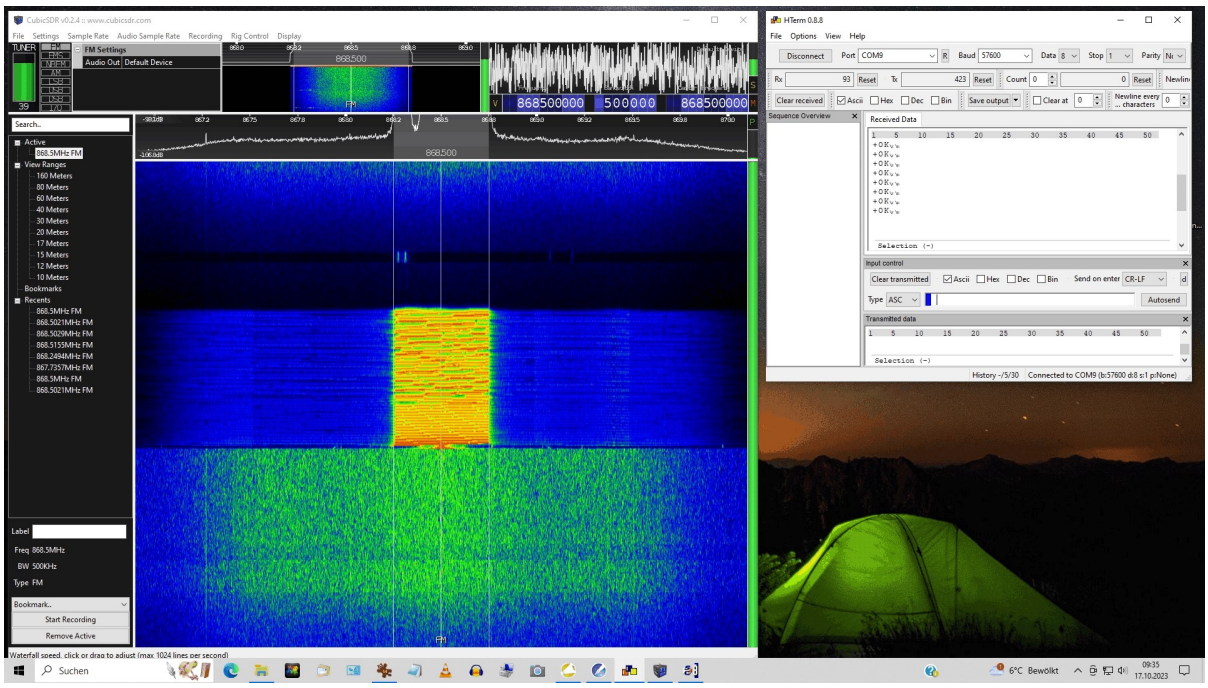


Abb. 9.2

Zum Senden einer Botschaft benutze ich wieder den Weg 1 (vgl. Einführung): LoRa-Modul + USB-Seriell-Wandler + RYLR998-Modul. Auf dem Screenshot des Bildschirms sehen wir rechts das geöffnet *Hterm*-Programm; damit werden wir mit dem Kommando AT+SEND=16,10,Hallo Welt Botschaften aussenden.

Vorher nehmen wir aber noch zwei Einstellungen an unserem LoRa-Modul vor:

- Mit dem Kommando AT+PARAMETER=11,9,1,12 legen wir fest: SF = 11, BW = 500 kHz, CR = 1, Anzahl der Synchronisations-Chirps der Präambel n_0 = 12. Bei unserem LoRa-Modul ist n_0

= 12 der Standardwert; dieser wurde bislang bei allen unseren Anwendungen automatisch benutzt. Für *SF* habe ich den größten Wert gewählt, den unser LoRa-Modul zur Verfügung stellt; damit bei der beschränkten Sample-Rate unsere einfachen Versuchsanordnung überhaupt auswertbare Diagramme angezeigt werden können, sollten die Chirp-Zeiten nämlich möglichst groß sein.

- Mit dem Kommando AT+CRFOP=0 stellen wir die Sendeleistung auf den kleinst möglichen Wert (0 dBm) ein; auf diese Weise begrenzen wir die Reichweite so weit wie möglich. **Achten Sie darauf, dass Sie nach den folgenden Experimenten dieses Kapitels sämtliche Parameter wieder auf die Standardwerte zurücksetzen**.

Nachdem wir das RTL2832-Modul an den Rechner angeschlossen haben, starten wir das *CubicSDR*-Programm. Zunächst öffnet sich automatisch das Fenster aus Abb. 9.3: Wir wählen hier Generic **RTL2832U OEM :: 00000001** aus; im rechten Bereich des Fensters werden dann einige Informationen zu diesem Gerät angezeigt. Danach müssen wir unbedingt die Schaltfläche *Start* betätigen. Das Fenster wird geschlossen und das *CubicSDR*-Programm (linkes Fenster von Abb.9.2) beginnt sofort seine Arbeit. Wir erkennen das an den Signalen, die in den oberen Feldern angezeigt werden; außerdem können wir ein Rauschen hören (falls der PC-Lautsprecher nicht stumm-geschaltet ist).

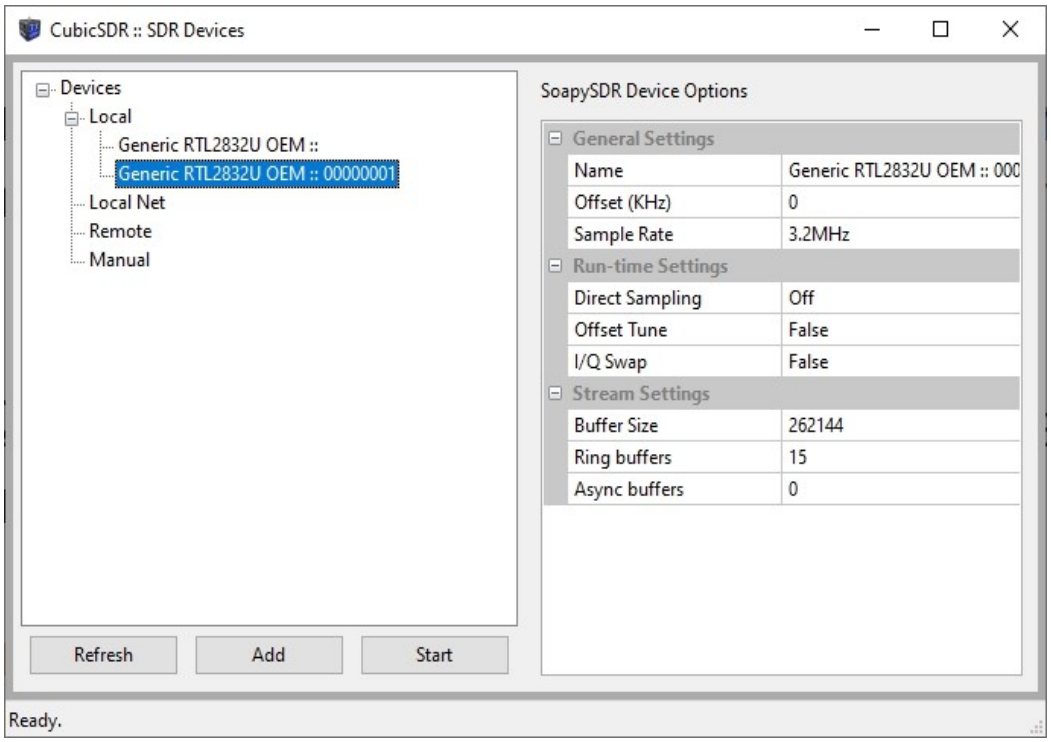


Abb. 9.3

Nun stellen wir im rechten oberen Bereich die Mittenfrequenz f_M = 868 500 000 Hz ein (rechtes Eingabefeld) und links daneben die Bandbreite *BW* = 500 000 Hz. Ferner deaktivieren wir im Menu Settings "Automatic Gain" und stellen dann in dem TUNER-Feld den Gain auf 39. Außerdem wählen wir die Sample Rate 3,2 MHz.

In dem großen blauen Feld des *CubicSDR* -Programms sehen wir jetzt, wie einzelne Punkte oder Flecken wie bei einem **Wasserfall** von oben nach unten rieseln; sie sind auf Rauschsignale zurückzuführen. Am rechten Rand dieses Wasserfall-Feldes ist ein grüner Scrollbalken zu sehen; dieser wird nun mit der Maus ganz nach oben gezogen, damit die Wasserfall-Geschwindigkeit maximal wird.

Nun geben wir im Eingabefeld von *HTerm* das Kommando "AT+SEND=16,10,Hallo Welt" ein. Kaum haben wir es mit der ENTER-Taste abgeschlossen, da "fällt" im Wasserfall-Feld ein rechteckiges Gebilde von oben nach unten; dies ist eine graphische Darstellung des *t-f*-Signals von unserem LoRa-Paket; man bezeichnet sie auch als Wasserfall-Darstellung. Wenn man schnell genug reagiert, dann kann man davon mit Strg-Druck einen Screenshot machen und diesen als Bild mit einem Bildverarbeitungsprogramm speichern. So ist auch Abb. 9.2 entstanden.

Analyse eines "Empfangspakets"

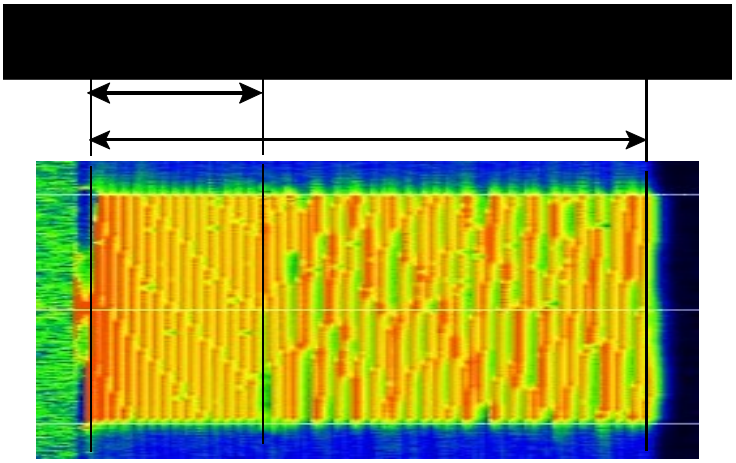


Abb. 9.4

In der Abb. 9.2 liegt die Zeit-Achse vertikal und die Frequenz-Achse horizontal. Auf den ersten Blick erkennen wir schon, dass sich die LoRa-Frequenzen hier wie erwartet in dem Intervall $[f_M - BW/2;\ f_M + BW/2]$ befinden. Um die uns vertraute Darstellungsweise von Abb. 8.5 zu erreichen, wurde nun der relevante Ausschnitt um 90° gegen den Uhrzeigersinn gedreht und horizontal gespiegelt. Das Ergebnis sehen Sie in Abb. 9.4. In dem linken Abschnitt erkennt man hier 14 (stark) ansteigende Strecken; hierbei handelt es sich um die 12 Up-Chirps der Präambel und 2 Chirps für *Sync Word*, jeweils mit der Symbollänge T_M. Nach rechts schließen sich zwei abfallende Strecken an; diese stammen von den 2,25 Down-Chirps. Wenn man genau hinschaut, kann man sogar auch das sich anschließende Viertel Down-Chirp erkennen. Nun folgen nach rechts hin weitere Strecken; diese steigen wie bei den Präambel-Chirps von links nach rechts an, aber sie beginnen nicht unbedingt am unteren Rand und brechen meist auch schon vor dem oberen Rand wieder ab. Hier handelt es sich um Up-Chirps, die zu den verschiedenen Symbolen des Payloads gehören.

Mit diesen Erkenntnissen können wir jetzt auch den *ToA*-Wert bestimmen. Wir kennen nämlich die Zeitspanne der ersten 14 Chirps. Diese ist gerade 14 T_M; dabei gilt für die Symbolzeit T_M:

$$T_M = \frac{2^{FS}}{BW} = 4{,}096\ \text{ms}.$$

Nun messen wir mit einem Lineal die beiden markierten Längen von Abb. 9.4 aus; da es nur auf deren Verhältnis ankommen wird, ist es egal, mit welcher Vergrößerung der Abbildung man

arbeitet. In meinem Fall erhalte ich die Werte 5,4 cm und 17,6 cm. Aus der Beziehung

$$\frac{ToA}{14\,T_M} = \frac{17{,}6}{5{,}4}$$ erhält man dann: $$ToA = \frac{17{,}6}{5{,}4} \cdot 14\,T_M \approx 45{,}6 \cdot 4{,}096\ \text{ms} \approx 187\ \text{ms}.$$

Den *ToA* -Wert kann man auch allein aus der Größe des Payloads und den Parametern des LoRa-Moduls berechnen. Dies schauen wir uns im nächsten Abschnitt genauer an.

Time on Air (*ToA*) berechnen

Das *Semtech* Datasheet [SDS] gibt auf S. 31 an, wie man die *ToA* berechnen kann. Diese setzt sich zusammen aus der Präambel-Zeit und der Payload-Zeit. Die Präambel-Zeit wird mit der Formel

$$T_{preamble} = (n_0 + 4{,}25) \cdot T_M$$

berechnet. Dabei ist n_0 die Anzahl der M_0-Chirps; bei unserem LoRa-Modul ist n_0 = 12 (s. o.). T_M bezeichnet die Symbolzeit (Chirp-Dauer). Die in der Formel zusätzlich auftauchenden 4,25 Symbolzeiten entsprechen dem Sync Word und den Down-Chirps (vgl. Abb. 8.5).

Die Berechnung der Payload-Zeit ist wesentlich aufwendiger. Zunächst bestimmt man die Anzahl der Symbole für den Payload:

$$n_{payload} = 8 + \max\left(\text{ceil}\left[\frac{8\,PL - 4\,SF + 28 + 16\,CRC - 20\,IH}{4\,(SF - 2\,DE)}\right](CR + 4), 0\right)$$

Hinweise:

- Die Funktion max(*a*, *b*) ergibt das Maximum der beiden Werte *a* und *b*.
- Die Funktion ceil(*x*) ergibt den nach oben aufgerundeten Wertes von *x*.

Die Bedeutung der einzelnen Variablen ist folgende:

Bezeichnung	Bedeutung	Reyax 998
PL	*Physikalischer* Payload	maximal 240 (bei *CR* = 1)
SF	Spreizfaktor	5 bis 11, Standardwert: 9
CRC	Prüfwert (**C**yclic **R**edundancy **C**heck)	1 (aktiviert)
IH	Explicit-Header-Aktivierung	0 (Header vorhanden)
DE	LowDataRateOptimize	0 (nicht aktiviert)
CR	Coding Rate	1 bis 4, Standardwert: 1

Der Wert 28 in der Formel gibt die Länge des expliziten Payload-Headers (in Bit) an. Wenn kein solcher Header vorhanden ist (*IH* = 1), dann werden durch den Ausdruck - 20 *IH* von diesen 28 Bit 20 Bits subtrahiert, so dass der Payload-Header jetzt nur noch 8 Bit groß ist (vgl. Abb. 6.2).

Zum **physikalischen Payload** gehören nicht nur die Nutzdaten des Anwenders; bei unserem LoRa-Modul entsprechen diese Nutzdaten gerade der mit dem SEND-Command übertragenen Zeichenkette. Diesen Nutzdaten werden eine Reihe weiterer Daten hinzugefügt, z. B. Informationen über den Absender und den Adressaten; diese Zusatzdaten bestehen aus mindestens 12 Bytes (MHDR[1], DevAddr[4], FCtrl[1], FCnt[2], MIC[4]). Meist kommt noch ein weiteres Bit FPort[1] dazu.

Für unser LoRa-Modul ergibt sich daraus (bei einem Standard *CR* -Wert von 1):

$$n_{payload} = 8 + \max\left(\text{ceil}\left[\frac{8\,PL - 4\,SF + 44}{4 \cdot SF} \right] \cdot 5\,,0 \right)$$

Da bei unserem LoRa-Modul *SF* höchstens 11 sein kann und der *PL* -Wert mindestens 12 ist, ist der Ausdruck in den eckigen Klammern auf jeden Fall positiv. Damit vereinfacht sich die Formel zu:

$$n_{payload} = 8 + \text{ceil}\left[\frac{8\,PL - 4\,SF + 44}{4 \cdot SF} \right] \cdot 5 = 8 + \text{ceil}\left[\frac{2\,PL + 11}{SF} - 1 \right] \cdot 5$$

Auch ohne ins Detail zu gehen, können wir den Formeln Einiges entnehmen:

- Als Grundlage der Berechnung von $n_{payload}$ dient die Anzahl der zu übertragenden Bits; dies wird allein schon durch den Ausdruck $8 \cdot PL$ deutlich.

- Es wird ein Prüfwert (*CRC*) übertragen. Dieser Prüfwert besteht aus 2 Bytes. Das sendende LoRa-Modul bildet diesen Wert nach einer festgelegten Vorschrift und fügt diesen Wert an den Payload an. Das empfangende LoRa-Modul bildet aus den empfangenen Bytes (ohne den *CRC*-Wert) nach derselben Vorschrift wieder den *CRC*-Wert und vergleicht ihn mit dem empfangenen *CRC*-Wert. Stimmen diese beiden *CRC*-Werte nicht überein, weist dies auf einen Übertragungsfehler hin. Das LoRa-Modul zeigt das durch eine entsprechende Fehlermeldung an. Mehr Informationen zu diesem Prüfwert findet man bei [CRC].

- Jeweils 4 Bits werden durch *CR* + 4 Bits kodiert; wenn *CR* = 1 ist, werden also 4 Bits durch 5 Bits dargestellt. Durch die zusätzlichen (Paritäts-)Bits wird der Payload bei der Übertragung auf noch mehr Symbole verteilt, außerdem ermöglichen sie eine Vorwärtsfehlerkorrektur (mehr dazu in Kap. 10).

Die Payload-Zeit ergibt sich damit zu

$$T_{payload} = n_{payload} \cdot T_M \,.$$

Die Summe aus $T_{preamble}$ und $T_{payload}$ bildet schließlich die Time of Air (*ToA*):

$$T_{ToA} = T_{preamble} + T_{payload}.$$

Welcher rechnerische Wert ergibt sich damit nun für die *ToA* des LoRa-Pakets aus dem letzten Abschnitt? Für einen physikalischen Payload von 12 + 10 = 22 Bytes ergibt die Rechnung 181,2 ms; dabei wurden die Einstellungen n_0 = 12, *SF* = 11, *CR* = 1, *BW* = 500 kHz zugrunde gelegt. Wer die Rechnung nicht selbst durchführen möchte, kann auch die EXCEL-Datei (ToA.xlsx) aus dem Materialien-Ordner benutzen [TOA].

Der gerade berechnete *ToA*-Wert stimmt recht gut mit dem im zweiten Abschnitt dieses Kapitels ermittelten Wert überein; die Abweichung beträgt nur etwa 3%!

In Kapitel 6 hatten wir schon einmal einen *ToA*-Wert experimentell bestimmt. Dort waren wir davon ausgegangen, dass der LoRa-Sender erst dann anfängt zu senden, wenn das letzte Bit über die UART angekommen ist und der Empfänger sein erstes Bit erst dann an die UART abgibt, wenn er das LoRa-Signal vollständig erfasst hat. In dem betrachteten Fall waren wir so auf einen Wert von etwa 170 ms gekommen. Die Rechnung mit der *ToA*-Formel führt zu einem etwas größeren Wert, nämlich 181 ms.

Schauen wir uns zum Schluss dieses Kapitels einmal an, wie sich bei einem physikalischen Payload von 100 Bytes die Anzahl der Symbole und die *ToA* verhalten, wenn man den Spreizfaktor *SF* schrittweise um 1 erhöht. Die Excel-Tabelle ToA.xlsx liefert uns folgende Ergebnisse:

SF	$n_{payload}$	ToA (in ms)
7	158	178,4
8	138	315,9
9	123	570,4
10	113	1058,8
11	103	1953,8

Wenn wir den Spreizfaktor um 1 erhöhen, verdoppelt sich die Anzahl der verschiedenen Symbole, die ein Chirp übertragen kann; jedes Symbol kann nun 1 Bit mehr übertragen. Indem LoRa die einzelnen Bits des physikalischen Payloads zusammen mit den CR-Bits auf die Symbole verteilt, sinkt $n_{payload}$ entsprechend ab. Tatsächlich entspricht das Verhältnis untereinander stehendender $n_{payload}$-Werte in etwa dem umgekehrten Verhältnis der entsprechenden *SF*-Werte.

Viel deutlicher unterscheiden sich die *ToA*-Werte. Hier findet bei jeder Erhöhung von *SF* nahezu eine Verdopplung statt. Dies ist darauf zurückzuführen, dass die Symbolzeit proportional zu 2^{SF} ist; das gleichzeitige Absinken von $n_{payload}$ schwächt dieses exponentielle Wachstum nur wenig ab.

10 Spreizfaktor und Reichweite

Wir hatten am Ende von Kapitel 8 schon festgestellt, dass ein größerer Spreizfaktor *SF* zu einer größeren Chirp-Zeit T_{Chirp} führt: Wird *SF* um 1 erhöht, wird T_{Chirp} verdoppelt. Die daraus resultierende Verringerung der Übertragungsgeschwindigkeit kann aber in Kauf genommen werden, wenn es darum geht, kleine Payloads über größere Strecken hinweg zu übertragen.

Für eine Bandbreite von 125 kHz findet man in [BRO] folgende Angaben für die **Reichweite *d*** (bei 14 dBm Sendeleistung):

SF	7	8	9	10	11
d in km	2	3	5	7	9

Diese Angaben beziehen sich auf "günstige Bedingungen", z. B. eine Übertragung auf Sicht. Auch die Güte der Antennen sowie die dafür benutzten Stecker und Kabel spielen hier eine Rolle.

Messung von Reichweiten

In Kapitel 4 hatte ich schon eine Reichweitenmessung mit der Standardeinstellung (*SF* = 9 und *BW* = 125 kHz) und der Sendeleistung 14 dBm (CRFOP = 14) durchgeführt. Dabei war die freie Sicht an einigen Stellen durch Bäume behindert gewesen.

Um den Zusammenhang zwischen Spreizfaktor und Reichweite genauer zu untersuchen, habe ich ein Gelände mit freier Sicht ausgesucht. Das Senden und Empfangen wurde aus leicht erhöhten Positionen auf einem 1 bis 2 m hohen Damm eines Regenrückhaltebeckens durchgeführt. Als Bandbreite benutzte ich *BW* = 125 kHz, als Spreizfaktoren Werte zwischen 7 und 9.

Bei den verschiedenen Messungen mit Abständen zwischen 300 m und 1400 m fiel auf: Die vom Empfänger-Modul angezeigten *RSSI* -Werte (vgl. auch Kap. 4 u. S. 40f) streuten bei gleicher Messstrecke und gleichem *SF* -Wert um ca. 5 Einheiten. Derartige Veränderungen wurden noch größer, wenn der Empfänger oder Sender seine Position ein wenig änderte, z. B. um 1 m abgesenkt wurde. Insbesondere wurden folgende Beobachtungen gemacht:

- Je größer der Abstand zwischen Sender und Empfänger war, desto kleiner war der *RSSI* -Wert.
- Die *SNR* -Werte (mehr dazu auf S. 40f) streuten um einen Wert von 10 dB und sanken leicht mit der Entfernung.
- Wurde der Empfänger auf den Boden gestellt, dann sanken die *RSSI* - und die *SNR* -Werte um bis zu 10 Einheiten ab.
- War die Sicht durch eine Reihe von Sträuchern und Bäume behindert, brach die Verbindung ab.

Minderung der Reichweite durch die Umgebung

Die Messungen machen deutlich: Befinden sich Gegenstände zwischen Sender und Empfänger, so wird dadurch das Signal gedämpft. Die Dämpfung hängt dabei vom Material und von der "Schichtdicke" ab. In der rechten Tabelle (nach [SMI]) finden Sie einige Beispiele (Erläuterungen zur Einheit dB s. u.).

Material	Dämpfung (in dB)
Glas (13 mm)	2,0
Holz (76 mm)	2,8
Beton (102 mm)	12
Stahlbeton (89 mm)	27

Für einen guten Empfang reicht aber auch eine **direkte Sichtverbindung** (also ohne dazwischen liegendes Material) nicht aus; tatsächlich spielen auch Objekte in der Nähe der **Sichtlinie** eine Rolle. Dieser Nahbereich wird auch als **Fresnel-Zone** bezeichnet. Befinden sich Objekte in dieser Zone (wie z. B. die kleine Bergkuppe in Abb. 10.1), können diese einen negativen Einfluss auf die Wellenausbreitung haben, obwohl eine direkte Sichtverbindung besteht. Jedes Objekt, das sich in der Fresnel-Zone befindet, senkt den Signalpegel und verringert damit die Reichweite (s. Abb. 10.1, nach [SMI]).

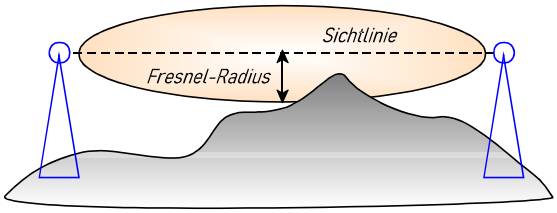


Abb. 10.1

RSSI und *SNR*

Nun ist es an der Zeit, uns genauer um die Bedeutung der Größen *RSSI* und *SNR* zu kümmern. Die Abkürzung *RSSI* steht für **R**eceived **S**ignal **S**trength **I**ndicator. Dieser Wert stellt einen (logarithmischen) Indikator für die Empfangsfeldstärke kabelloser Kommunikationsanwendungen dar. Er wird in **dBm** (Dezibel Milliwatt) gemessen und ist bei unseren Anwendungen immer negativ. Je näher der Wert an 0 liegt, desto stärker ist das Empfangsignal. Genauer: Wenn $P_{Empfangen}$ die Leistung des Empfangssignals bezeichnet, dann ist der *RSSI*-Wert definiert durch

$$RSSI\ (\text{in dBm}) = 10\ \lg\left(\frac{P_{Empfangen}}{1\,\text{mW}}\right).$$

Dabei bezeichnet hier lg den Logarithmus zur Basis 10.

Die Abkürzung *SNR* steht für **S**ignal **N**oise **R**atio, als das **Signal-Rausch-Verhältnis**:

$$SNR = \frac{P_{Signal}}{P_{Rauschen}}$$

Dabei stehen hier P_{Signal} für die **Nutzsignalleistung** und $P_{Rauschen}$ für die **Rauschleistung**. Bei vielen Anwendung bewegt sich dieses Verhältnis in einem Größenbereich, der mehrere Zehnerpotenzen umfasst. Aus diesem Grund benutzt man für *SNR* meist ein logarithmisches Maß:

$$SNR = 10 \cdot \lg\left(\frac{P_{Signal}}{P_{Rauschen}}\right) \text{dB} \qquad \text{(dB steht für Dezibel)}$$

Wir sehen an den Beispielen aus Abb. 10.2: Während die nicht-logarithmischen *SNR*-Werte sich in einer Spanne von etwa 0 bis 2000 bewegen, liegen die logarithmischen Werte etwa zwischen -7 und 33. Insbesondere stellen wir fest: Wenn das Rauschen stärker als das Nutzsignal ist, dann ist der logarithmische *SNR*-Wert negativ.

P_Signal in mW	P_Rauschen in mW	SNR	SNR (dB)
1000	0,5	2000	33,01
50	0,5	100	20,00
20	0,5	40	16,02
2	0,5	4	6,02
0,1	0,5	0,2	-6,99

Abb. 10.2

Im Allgemeinen gilt:

Wenn SNR > 0 dB ist, dann kann der Empfänger das empfangene Signal demodulieren.
Wenn SNR < 0 dB ist, dann kann der Empfänger das empfangene Signal NICHT demodulieren.

In Abb. 4.2 aus Kap. 4 sowie bei den oben beschrieben Messungen ist der vom Empfänger bestimmte *SNR*-Wert (meist) 10 dB. Dass die empfangenen Signale demoduliert werden konnten, ist damit nicht verwunderlich. Wir können für diesen Fall auch rasch ausrechnen, wie sich P_{Signal} und $P_{Rauschen}$ zueinander verhalten:

$$10 = 10 \lg\left(\frac{P_{Signal}}{P_{Rauschen}}\right)$$

$$1 = \lg\left(\frac{P_{Signal}}{P_{Rauschen}}\right)$$

$$10 = \frac{P_{Signal}}{P_{Rauschen}}$$

Demnach ist hier P_{Signal} um den Faktor 10 größer als $P_{Rauschen}$.

Erstaunlich ist nun: LoRa-Demodulation ist auch möglich, wenn der *SNR*-Wert kleiner als 0 dB ist. Auf S. 27 des Semtech-Datenblatts SX1276-7-8-9 (*DS_SX1276-7-8-9_W_APP_V7.pdf*, s. [SDS]) finden wir die folgende Tabelle:

SpreadingFactor (RegModulationCfg)	Spreading Factor (Chips / symbol)	LoRa Demodulator SNR
6	64	-5 dB
7	128	-7.5 dB
8	256	-10 dB
9	512	-12.5 dB
10	1024	-15 dB
11	2048	-17.5 dB
12	4096	-20 dB

Abb. 10.3

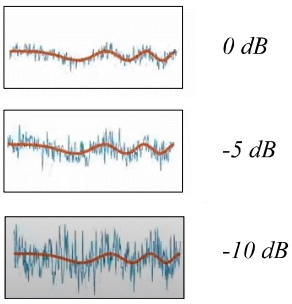


Abb. 10.4 (Quelle: [HOW])

Mit zunehmendem Spreizfaktor *SF* sinkt der *SNR*-Wert, bei dem eine Demodulation noch gewährleistet wird. Bei einem Spreizfaktor *SF* = 12 beträgt dieser *SNR*-Wert -20 dB. Hier ist die Signalleistung 100 mal kleiner als die des Rauschens! In der Abb. 10.4 ist das gleiche Nutzsignal (rot) mit unterschiedlich starken Rauschsignalen (blau) dargestellt. Im unteren Fall sieht man sofort, dass das Rauschsignal erheblich stärker ist als das Nutzsignal; dennoch wäre eine LoRa-Demodulation (bei einem Spreizfaktor 8 oder größer) noch möglich.

Bei der Messung für die Abb. 4.2 hatte ich auch die *SNR*-Werte aufgezeichnet. In der letzten Etappe wurden vom Empfänger *SNR*-Werte von -11 dB angezeigt. In zwei Fällen (etwa bei 480 m) hatte der Empfänger zwei der gesendeten Signale zwar bemerkt, sie aber nicht erfolgreich demodulieren können. Dies hatte er mit der Fehlermeldung +ERR=12 angezeigt, was auf einen CRC-Fehler hinweist (vgl. Kap. 9). Vielleicht war hier der *SNR*-Wert kleiner als -12,5 dB gewesen?

Sind die *RSSI*-Werte (in dBm) sowohl für das Nutzsignal als auch für das Rauschsignal bekannt, kann man den *SNR*-Wert (in dB) ganz einfach über deren Differenz berechnen:

$$
\begin{aligned}
P_{Signal}(dBm) - P_{Rauschen}(dBm) &= 10\,\lg\left(\frac{P_{Signal}}{1\,mW}\right) - 10\,\lg\left(\frac{P_{Rauschen}}{1\,mW}\right) \\
&= 10 \cdot \lg\left(\frac{P_{Signal} \,/\, 1\,mW}{P_{Rauschen} \,/\, 1\,mW}\right) dB \\
&= 10 \cdot \lg\left(\frac{P_{Signal}}{P_{Rauschen}}\right) dB \\
&= SNR
\end{aligned}
$$

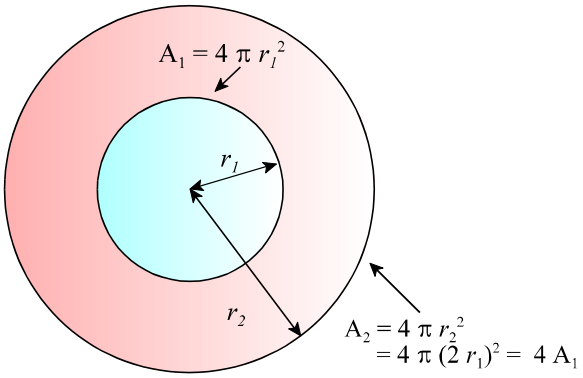
Ist z. B. P_{Signal} = - 102 dBm und $P_{Rauschen}$ = - 89 dBm, dann ist *SNR* = - 102 dBm - (- 89 dBm) = - 13 dB. In diesem Fall wäre eine Demodulation bei einem Spreizfaktor 10 (oder größer) möglich.

Auf ähnliche Weise können wir auch zeigen, dass man die Signalleistung nach Gewinnen oder Verlusten durch **Addition** berechnen kann, z. B.: $P_{Signal,nach} = P_{Signal,vor} + Dämpfung_{Material}$. Beachten Sie: **Verlustwerte müssen dabei immer mit einem Minuszeichen versehen werden!**

Link Budget

Für die Verbindung zwischen dem Sender und dem Empfänger kann eine Bilanz für die Gewinne und Verluste gezogen werden. Dazu gehen wir von einer Sendeleistung von 14 dBm aus. Für den Anschluss an die Antenne tragen wir einen Verlust von 1 dB in Rechnung. Durch die Antenne wird das Signal etwas verstärkt; in der Bilanz veranschlagen wir dies mit einem Gewinn von 2 dBi. Der Index i deutet darauf hin, dass der Wert in Relation zu einer isotropen, d. h. in alle Richtungen gleichmäßig ausstrahlenden Antenne, angegeben wird.

Während der Wellenausbreitung wird das Signal schwächer. Dies ist eine Folge des Energieerhaltungssatzes: Verdoppelt man z. B. den Abstand des Empfängers vom Sender, verteilt sich die Energie auf eine 4 mal so große Fläche. Allgemein gilt: Die Signalleistung nimmt mit dem Quadrat des Abstandes ab.



Der **Pfadverlust** (Path Lost) beschreibt diesen Leistungsverlust durch

$$L = 10 \cdot \lg\left[\left(\frac{\lambda}{4\,\pi\,d}\right)^2\right] = -20\,\lg\left(\frac{4\,\pi\,d}{\lambda}\right)$$

mit der Einheit dB. Für unsere Sendefrequenz f = 868,5 MHz ist λ = 0,345 m; bei einem Abstand von d = 2 km ergibt sich dann ein L-Wert von -97,3 dB, also ein Verlust von 97,3 dB. Verdoppeln wir den Abstand, erhalten wir einen Verlust von 103,3 dB. Allgemein führt eine Verdopplung des Abstandes dazu, dass der L-Wert um ca. 6 dB abnimmt.

Beim Empfänger gehen wir für den Antennengewinn wieder von einem Wert von 2 dB aus und für den Anschluss der Antenne benutzen wir wieder einen Verlust von 1 dB. Bei einer Entfernung von 2 km zum Empfänger erhält man jetzt die aufgenommene Signalstärke (*RSSI*) durch eine Addition der Werte; dabei beachten wir, dass wir die Verluste und Gewinne konsequent mit einem entsprechendes Vorzeichen versehen:

14 dBm - 1 dBm + 2 dBm - 97,3 dB + 2 dB - 1 dB = - 81,3 dBm

Diese Summe bezeichnet man auch als **Link Budget**. Ist dieser Wert über der Empfindlichkeit des Empfängers, dann kann das Signal empfangen bzw. demoduliert werden. Die minimale Empfangsempfindlichkeit liegt bei unserem LoRa-Modul bei -129 dBm, vgl. [LD2]. Sie hängt u. A. vom Spreizfaktor ab (mehr dazu finden Sie unter [LSC]).

Rauschen und Vorwärtsfehlerkorrektur (FEC)

Bislang bin ich davon ausgegangen, dass der Leser eine gewisse Vorstellung von dem Begriff Rauschen besitzt. Nun soll dieser Begriff ein klein wenig genauer erläutert werden.

Unter Rauschen (auch Untergrund genannt) versteht die Physik allgemein eine Störgröße mit breitem unspezifischem Frequenzspektrum. Es kann daher als eine Überlagerung vieler harmonischer Schwingungen oder Wellen mit unterschiedlicher Amplitude und Frequenz beziehungsweise Wellenlänge interpretiert werden. (zitiert nach [RAU]).

Was sind mögliche Ursachen für das Rauschen? Zum Einen können sie durch elektromagnetische Wellen in der Umgebung des Empfängers entstehen. Diese können z. B. durch Funken bei Elektromotoren oder Blitze etc. erzeugt werden. Rauschen entsteht aber auch in der Elektronik von Sendern und Empfängern. So können u. A. schlechte Kabelverbindungen zu einer Vergrößerung des Rauschens und damit auch zu einem schlechteren *SNR*-Wert führen.

Der Empfang kann auch durch fremde Sender beeinflusst werden, wenn sie die gleiche oder eine ähnliche Frequenz benutzen; hier kann es dann zu Interferenzen kommen, die eine Demodulation des LoRa-Signals unmöglich machen. Nach der obigen Definition spricht man hier aber nicht von Rauschen.

Handelt es sich bei diesem anderen Funksender um einen LoRa-Sender, der mit derselben Frequenz sendet wie *unser* LoRa-Sender, dann kann es auch zu solchen störenden Interferenzen kommen. Arbeitet der andere LoRa-Sender dabei allerdings mit einem anderen Spreizfaktor, wird sein Signal allenfalls als (ein schwaches) Rauschen registriert; dies ist eine Folge der Chirp-Modulationstechnik; das bedeutet insbesondere: **Sender und Empfänger müssen immer denselben Spreizfaktor haben**.

Das Rauschen kann bei der Demodulation im Empfänger zu fehlerhaften Bits führen. Diese können bis zu einem gewissen Grad durch eine **Vorwärtsfehlerkorrektur (F**orward **E**rror **C**orrection**)** behoben werden. Dazu werden den eigentlichen Datenbits Redundanzen hinzugefügt; dies geschieht in Form der bereits in Kapitel 9 erwähnten CR-Bits. Mit diesen Zusatz-Bits können die Fehler dann behoben werden, solange die Fehlerrate nicht zu groß ist. In [KRE] wird ein einfaches FEC-Verfahren an Hand eines Beispiels vorgestellt. Hierbei kommen auf 40 Datenbits insgesamt 13 Kontrollbits. Übrigens werden derartige Korrekturverfahren auch bei CDs benutzt!

Beachten Sie: Eine Erhöhung des *CR*-Wertes kann den *SNR*-Wert nicht verbessern, aber sie kann für mehr Zuverlässigkeit sorgen.

Wie kann man sich nun die erstaunliche Leistungsfähigkeit bzw. Empfindlichkeit von LoRa erklären? Dazu betrachten wir den folgenden Fall: Ein physikalischer Payload von 200 Bytes wird bei einem Spreizfaktor von 9 mit *CR* = 0 übertragen. Da es in diesem Fall 512 verschiedene Symbole gibt, könnte man meinen, dass mit einem einzigen Chirp nun 2 Bytes übertragen werden. Dann würden für den Payload nur 100 Chirps benötigt.

Jetzt berechnen wir die Anzahl der **tatsächlich** benutzten Symbole mit Hilfe der Formel aus Kapitel 8. Damit erhalten wir einen Wert von 188 Chirps. Daraus können wir schließen, dass LoRa die einzelnen Bits des Payloads nicht einfach eins zu eins aneinander gereiht überträgt. Vielmehr werden sie gespreizt übertragen; gegebenenfalls kann ein Bit auch auf mehrere Symbole verteilt werden (vgl. [USI]). Auf diese Weise können falsche Bits vermieden und damit der Empfang (auch bei größeren Entfernungen) verbessert werden.

Die Demodulation gelingt um so besser, je stärker das empfangene Signal ist; deswegen ist zur Überbrücken großer Strecken auch eine kleine Bit-Rate günstig. Je kleiner die Bit-Rate ist, desto größer ist nämlich die Energie, die pro Bit eingesetzt bzw. empfangen wird. Kleinere Bit-Raten erhalten wir durch größere Spreizfaktoren.

Spreizfaktor	Chirp-Rate (kHz)	Nutz-Bit-Rate (kHz) bei CR=1
5	3,906	15,625
6	1,953	9,375
7	0,977	5,469
8	0,488	3,125
9	0,244	1,758
10	0,122	0,997
11	0,061	0,537

Eine optimale Konfiguration

Das Verhalten der Sender und Empfänger lässt sich beeinflussen durch die folgenden Parameter:

- Sendeleistung
- Bandbreite *BW*
- Spreizfaktor *SF*
- Coding Rate *CR*

Die optimale Einstellung gibt es nicht; vielmehr wird man die Wahl der Parameter davon abhängig machen, welche Ziele man erreichen will.

Zwei Beispiele sollen dies erläutern.

Beispiel 1: Wir wollen in einer Entfernung von wenigen hundert Metern (bei freier Sichtlinie) jede halbe Minute einen Messwerte im Umfang von ca. 100 Bytes erhalten. Die Messapparatur wird mit einer Batterie betrieben; der Energieeinsatz soll möglichst

gering gehalten werden. Es kann in Kauf genommen werden, wenn zwischendurch einmal ein Datenpaket nicht empfangen werden kann.

In diesem Fall benutzen wir eine kleine Sendeleistung (CRFOP niedrig), eine kleine Bandbreite (125 kHz) und einen kleinen Spreizfaktor, z. B. 7. Als Coding-Rate wird *CR* = 1 gewählt. Der kleine Spreizfaktor führt zu einem relativ geringen Energieverbrauch. Die *ToA* beträgt etwa 200 ms; damit ist der Duty-Cycle 0,7 % also unter 1 %.

Beispiel 2: Wir wollen aus einigen Kilometern Entfernung jede halbe Stunde Daten (z. B. den Wasserstand eines Baches) übertragen. Da der Wasserstand maximal 200 cm beträgt, reichen hier bei unserem LoRa-Modul für die zentimetergenaue Angabe zwei Bytes aus; dazu übertragen wir den Wasserstand hexadezimal. Z. B. senden wir die Wasserhöhe von 142 als Payload in Form der Zeichenkette "8E". Hier ist die Einsparung gegenüber der Zeichenkette "142" natürlich gering. Im Allgemeinen kann aber durch Benutzung einer geschickten "Kodierung" der Payload spürbar reduziert werden.

Messapparatur und Sender werden über das Stromnetz versorgt. Deswegen können wir ohne Energiesorgen eine höhere Sendeleistung wählen. Diese muss mindestens so groß sein, dass beim Empfänger der *RSSI* -Wert die Empfangsempfindlichkeit unterschreitet. Möchte oder kann man aber dabei eine bestimmte Sendeleistung nicht überschreiten, kann man ersatzweise den Spreizfaktor erhöhen. Würde man z. B. den Spreizfaktor von 7 auf 11 erhöhen (was bei der Bandbreite von 125 kHz mit unserem LoRa-Modul allerdings nicht möglich ist), so sinkt die Empfangsempfindlichkeit um 10 dBm. In diesem Fall wäre die *ToA* dann 643 ms; der zugehörige Duty Cycle würde dann etwas mehr als 1 Minute betragen.

Die letzten Überlegungen kann man mit zwei Personen vergleichen, die sich in einer geräuschvollen Umgebung unterhalten wollen. Wenn sie weit auseinander stehen, dann müssen sie lauter oder langsamer (und damit deutlicher) sprechen. (Nach: [DRS])

11 Eine einfache LoRaWAN-Simulation

In diesem letzten Kapitel wollen wir ein einfaches LoRaWAN simulieren: Endgeräte mit Sensoren (**Nodes**) erfassen Daten und senden sie per LoRa an eine **"LoRaWAN-Station"** (vgl. Abb. 11.1). Die Endgeräte simulieren wir durch eine Kombination aus LoRa-Modul, USB-UART-Wandler und einen PC mit dem Programm *HTerm* . Die "Messwerte" (oder auch andere Daten) werden händisch mit dem AT+SEND-Kommando an die LoRaWAN-Station gesendet. **Unterschiedliche Endgeräte simulieren wir durch Ändern der LoRa-Adresse.** Auf ähnliche Weise könnte man auch Endgeräte mit Aktoren betrachten. Auf solche Endgeräte wollen wir hier aber nicht eingehen.

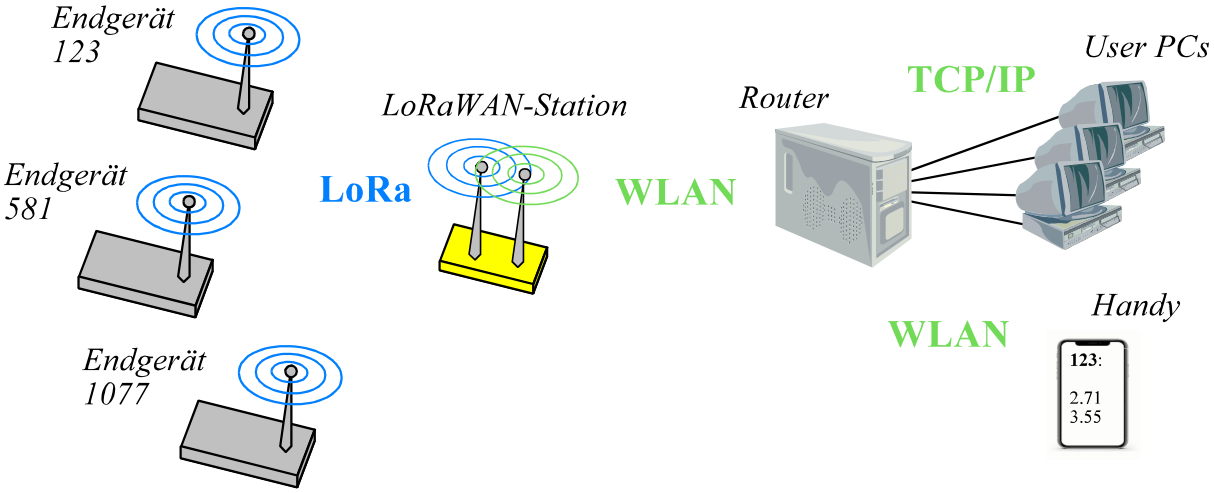


Abb. 11.1

Die Endgeräte senden ihre Daten mitsamt der Endgeräte-Adresse an die LoRaWAN-Station: Diese besteht bei uns aus einem TTGO T-Display und einem LoRa-Modul. Der ESP32 des TTGO **speichert die Daten mitsamt der Adresse des Absenders in seinem Flash-Speicher** . Auf diese Daten kann man nun per WLAN zugreifen: Der ESP32 arbeitet nämlich nebenher auch noch als **Server im Hausnetzwerk** (Station-Mode 'STA-IF', s. u.). Wenn seine IP-Adresse z.B. 198.162.2.105 lautet, kann man auf die Daten eines bestimmten Endgerätes mit Hilfe dessen Adresse zugreifen. Lautet die Adresse z. B. 581, dann gibt man im Browser die URL

http://198.162.2.105/581

ein. Der ESP32 sucht jetzt im Flash-Speicher nach allen Datensätzen mit der Adresse 581 und sendet diese an den Client (Browser).

Die Beschränkung auf die Datensätze einer einzigen Adresse ist für den Anwender recht nützlich; so muss er nicht selbst die für ihn relevanten Daten herausfiltern. Zudem erschwert diese Vorgehensweise auch ein wenig den Zugriff durch andere Parteien. Wird nämlich die Nummer eines nicht existierenden Endgerätes (oder auch keine Nummer) angegeben, so sendet der Server lediglich "Keine Daten vorhanden!".

Natürlich könnte man die Bedienung des Browsers auch etwas anwenderfreundlicher gestalten: So könnte es in der vom Server gesendete Webseite ein Eingabemöglichkeit für die Endgeräte-Adresse geben; die URL würde dann lediglich aus der IP-Adresse (ohne Parameter) bestehen. Auch könnte man weitere Sicherheitsmerkmale einbauen. Das würde aber dem eigentlichen Ziel, nämlich ein ***einfaches*** Programm für das Funktionsprinzip von LoRaWAN vorzustellen, entgegenstehen.

Vom Endgerät zum Speicher der LoRaWAN-Station

In diesem Abschnitt stellen wir zunächst ein Programm für das TTGO T-Display vor, welches Botschaften von einem Endgerät empfängt und diese in einer Datei in seinem Flash speichert. Neben der Botschaft selbst sollen auch die Adresse des Absenders sowie ein Datum-Zeit-Stempel für den Empfang festgehalten werden. Unmittelbar nach dem Empfang soll unser Programm eine entsprechende Quittung an den Absender zurücksenden. Damit kann das Endgerät erkennen, ob seine Botschaft angekommen ist. Gegebenenfalls kann das Endgerät dann - bei einer entsprechenden Programmierung - seine Botschaft erneut absenden; bei unserer Simulation wird man selbst eine solche Kontrolle bei *HTerm* vornehmen und dann ggf. die Botschaft erneut an die LoRaWAN-Station senden.

Das entsprechende Hauptprogramm sieht im Wesentlichen so aus:

```
# LoRa initialisieren...
lora_address = lora_init(uart1)
display.text(font2, 'LORA-ADR: ' + lora_address, 5, 10)
if lora_address == '???':
    display.text(font2, 'Resetten!', 5, 80, st7789.RED)
    print('Keine Verbindung zum LORA-Modul!')
    print('RESET-Knopf am TTGO betätigen!')
    sys.exit() # Programm beenden, wenn keine Antwort vom LoRa-Modul

while True:
    receive_save_confirm(uart1, display, rtc) # empfangen, speichern und
                                                                bestätigen
    print(read_response(uart1)) # Das ist die Antwort b'+OK\r\n'.
    print()
```

Zunächst werden mit der Funktion `lora_init` grundlegende LoRa-Parameter festgelegt und die Adresse des angeschlossenen LoRa-Moduls bestimmt; diese Funktion wird zuvor aus dem Modul `lora.py` importiert (vgl. Kap. 4 und 5). Ist die festgestellte Adresse '???', konnte kein LoRa-Mdul erkannt werden und das Programm wird beendet.

In der folgenden Endlosschleife wird mit der Funktion `receive_save_confirm` eine Botschaft empfangen, im Flash mitsamt der Adresse des Endgerätes und einem Datum-Zeit-Stempel (s. u.) abgespeichert und anschließend eine Quittungs-Botschaft an das Endgerät zurückgesendet. Der Quellcode dieser Funktion lautet:

```
def receive_save_confirm(uart, display, rtc):
    s = read_response(uart) # Byte-String
    item = received_to_list(s, rtc)
    print('item=', item)
    save(data_fname, item, delimiter)
    confirm(item[0], item[2], display, uart)  # Adresse u. Datum
```

Dabei greifen wir auf die Funktion `read_response` aus dem Modul `lora.py` zurück. Sie liefert uns die empfangene Botschaft `s`. Mit dieser Botschaft und dem aktuellen Datum-Zeit-Stempel `rtc` bilden wir eine Liste mit dem Namen `item`:

```
item = received_to_list(s, rtc)
```

Eine solche Liste sieht dann z. B. so aus:

```
['123', 'hallo', '13.11.2023 11:12:30']
```

Das erste Element gibt die Adresse des Absenders (123) an, das zweite Element die Botschaft (User-Payload: hallo), das dritte Element den **Datum-Zeit-Stempel** (13.11.2023 14:12:30). Dieser Datum-Zeit-Stempel wird mit Hilfe eines Objekts `rtc` gebildet, welches eine Instanz der `RTC`-Klasse ist (**R**eal **T**ime **C**lock). Dieses Objekt liefert mit seiner Methode `datetime` in unserem Fall das Tupel (2023, 11, 13, 0, 14, 12, 30, 609003). Die Elemente dieses Tupels haben hier folgende Bedeutung:

2023	Jahreszahl
11	Monat
13	Tag
0	Wochentag (0: Montag, 1: Dienstag, ..., 6: Sonntag)
14	Stunden
12	Minuten
30	Sekunden

Das letzte Element ist für uns irrelevant.

Die Funktion `received_to_list` sieht nun folgendermaßen aus:

```
def received_to_list(s, rtc): # s ist vom Typ Byte-String
    s = str(s, 'UTF-8')
    s = s[5:len(s)]
    print('s =', s)
    liste = s.split(',',2)
    print(liste)
    Addr = liste[0]
    Payload_length = int(liste[1])
    print('PL_Länge: ', Payload_length)
```

```
    Payload = liste[2][0:Payload_length]
    d_t_tuple=rtc.datetime() # now() gibt es nicht!
    Date_time_stamp = str(d_t_tuple[2])+'.'+str(d_t_tuple[1])+'.'
            +str(d_t_tuple[0])+' '+str(d_t_tuple[4])+':'+str(d_t_tuple[5])
            +':'+str(d_t_tuple[6])
    print(Date_time_stamp)
    print(Addr, Payload, Date_time_stamp)
    return [Addr, Payload, Date_time_stamp]
```

Der empfangene Byte-String `s` wird zunächst durch `s = str(s, 'UTF-8')` in einen String (mit demselben Namen) umgewandelt. Mit der Zeile

```
s = s[5:len(s)]
```

entfernen wir dann die Einleitung '+RCV=' aus dieser Zeichenkette (vgl. Kap. 3). Anschließend entnehmen wir dieser (neuen) Zeichenkette `s` nun die für uns relevanten Daten. Dabei gehen wir etwas anders vor als bei der Funktion `read_response` aus unserem Modul `lora.py`. Diese Funktion arbeitet nämlich nur dann korrekt, wenn im User-Payload kein Komma auftaucht. Jetzt wollen wir diesen Fall nicht mehr ausschließen. Dazu zerlegen wir die Zeichenkette `s` zunächst mit `s.split(',',2).` Wenden wir diese Methode z. B. auf die Zeichenkette

```
s = 'qwert,asdfg,yxcvb,1234,5678'
```

an, so erhalten wir die folgende Liste:

```
['qwert', 'asdfg', 'yxcvb,1234,5678']
```

Die ersten 2 Elemente sind die ersten 2 durch Komma getrennten Teile von `s`; das letzte Element besteht dann aus der restlichen Zeichenkette.

Bei unserer Funktion `received_to_list` enthält `liste = s.split(',',2)` als erstes Element (mit dem Index 0) die Adresse des Absenders und als zweites Element (mit dem Index 1) die Länge des Payloads (als Zeichenkette). Letztere wird nun in eine Zahl konvertiert und in der Variablen `Payload_length` gespeichert. Damit können wir schließlich aus dem "Rest-String" `liste[2]` den User-Payload extrahieren:

```
Payload = liste[2][0:Payload_length]
```

Anschließend wird mit `d_t_tuple=rtc.datetime()` ein aktuelles Datum-Zeit-Tupel erzeugt; aus diesem bilden wir schließlich eine entsprechende Zeichenkette `Date_time_stamp`.

Der Rückgabewert `[Addr, Payload, Date_time_stamp]` wird nun in der Funktion `receive_save_confirm` über die Variable `item` an die Funktion `save` übergeben:

`save(data_fname, item, delimiter)`

Diese speichert die in der Liste `item` vorliegenden Daten als **Zeile** in einer Datei ab, deren Namen durch die Variable `data_fname` vorgegeben ist. (Grundlegende Informationen zum Umgang mit Dateien finden Sie in der Textbox auf der nächsten Seite.) Dabei können wir die Liste `item` nicht direkt mit der write-Methode speichern; denn diese kann nur mit Zeichenketten umgehen. Deswegen bilden wir aus den Elementen von `item` eine einzige Zeichenkette; dabei werden die einzelnen Elemente durch ein Abgrenzungszeichen (delimiter) von einander getrennt. Ich benutze hierfür das /-Zeichen.

Mit der Funktion

```
def save(fname, liste, delimiter):
    f = open(fname, 'a') # append mode
    f.write(liste[0] + delimiter)
    f.write(liste[1] + delimiter)
    f.write(liste[2])
    f.write('\n')
    f.close()
```

wird unsere Liste als Zeichenkette mit dem Delimiter '/' gespeichert. Bei der Liste

`['123', 'hallo', '13.11.2023 11:12:30']`

ist die gespeicherte Zeichenkette dann

`'123/hallo/13.11.2023 11:12:30'` (gefolgt von dem NewLine-Steuerzeichen \n)

Nun wollen wir noch die letzte in `receive_save_confirm` auftauchende Funktion `confirm` angeben:

```
def confirm(addr, d_t_stamp, display, uart):
    display.text(font2, 'Bestaetigung:', 5, 45)
    display.text(font2, 'MSG von ' + addr + '     ', 5, 75)
    display.text(font1, d_t_stamp+'      ', 5, 115)
    msg = 'MSG confirmed'
    send_message(uart, int(addr), msg)
```

Durch diese Funktion werden auf dem Display des TTGO zunächst die Adresse des Nodes und der Datum-Zeit-Stempel angezeigt. Anschließend wird an diesen Node die Quittung "MSG confirmed" gesendet.

Exkurs: Daten im Flash speichern oder lesen

Beginnen wir mit einem einfachen Beispiel:

```
file = open('testfile.txt', 'a')
file.write('Hallo Welt!')
file.close()
```

Die Methoden open und write

In der ersten Zeile des Beispiels wird eine Datei mit dem Namen `'testfile.txt'` mit der **open**-Methode geöffnet; existiert noch keine solche Datei, wird sie erzeugt. Dabei wird dieser Datei ein Objekt zugeordnet, welchem wir hier den Namen `file` gegeben haben. Dieses Objekt besitzt nun mehrere Methoden, mit denen wir recht einfach auf die Datei zugreifen können: Mit `file.write('Hallo Welt!')` wird die Zeichenkette 'Hallo Welt!' in die Datei geschrieben. Der Parameter 'a' in der ersten Zeile unseres Beispielprogramms sorgt dafür, dass diese Zeichenkette hinter den bereits bestehenden Inhalt der Datei angefügt wird ('a' steht für append = anhängen). Ist die Datei noch leer, wird sie direkt eingefügt. Benutzt man statt 'a' den Parameter 'w', dann wird der Inhalt der Datei (sofern vorhanden) durch `file.write('Hallo Welt!')` überschrieben ('w' steht für write = schreiben). Alle Daten, die sich in der Datei **vorher** befunden haben, werden also gelöscht! Mehrere Schreibvorgänge können hintereinander durchgeführt. Sollen keine weiteren Daten in der Datei gespeichert werden, muss die Datei unbedingt geschlossen werden; dies geschieht durch die dritte Zeile: `file.close()`

Die Methoden read und readline

Will man eine Datei auslesen, kann dies mit der read-Methode geschehen. Zuvor muss die Datei geöffnet werden; dabei benutzen wir diesmal den Parameter 'r' ('r' steht hier für read = lesen):

```
file = open('testfile.txt', 'r')
s = file.read()
print(s)
file.close()
```

In diesem Beispiel wird die Datei mit dem Namen 'testfile.txt' geöffnet. Dann wir der (gesamte) Inhalt der Datei gelesen und in der Variablen s gespeichert. Mit print(s) wird die Zeichenkette s auf dem Terminal ausgegeben. Zuletzt wird die Datei wieder mit der close-Methode geschlossen.
In manchen Fällen ist es sinnvoll, die einzelnen gespeicherten Datensätze gegeneinander abzugrenzen. Dazu benutzt man das Steuerzeichen '\n' (NewLine). Dieses fügen jeweils an das Ende des Argument vom Schreib-Befehl an, z. B. so: `file.write('Hallo Welt!'+'\n')`. Damit wird der Inhalt der Datei zeilenmäßig strukturiert. Mit der Methode `readline` können diese Zeilen einzeln ausgelesen und verarbeitet werden:

```
file = open('testfile.txt', 'r')
line = file.readline()
while line:
    print(line)
    line = file.readline()
file.close()
```

Nach dem Öffnen der Datei wird die erste Zeile gelesen und in der Variablen line gespeichert. In der `while`-Schleife wird zunächst die zuletzt gelesene Zeile im Terminal angezeigt und dann die nächste Zeile gelesen. Gibt es keine weitere Zeile mehr in der Datei, dann wird eine leere Zeichenkette zurückgegeben. Damit endet die `while`-Schleife; das Leerzeichen hat nämlich für `while` die gleiche Wirkung wie `False`.

Zu guter Letzt geben wir die benötigten Importe und Instanziierungen sowie die benutzten Konstanten an, die am Anfang des Programms stehen:

```
############  Importe und Instanziierungen #############

from time import sleep
from machine import UART, SPI, Pin, RTC
import vga1_bold_16x32 as font2
import vga1_8x8 as font1
import st7789
import sys
from lora import send_command, read_response, response_to_list, lora_init,
                                                              send_message

# Display:
spi = SPI(1, baudrate=20_000_000, polarity=1, sck=Pin(18), mosi=Pin(19))
display = st7789.ST7789(spi, 135, 240, reset=Pin(23, Pin.OUT),
                              cs=Pin(5, Pin.OUT), dc=Pin(16, Pin.OUT),
                              backlight=Pin(4, Pin.OUT), rotation=3)
display.init()
display.fill(0) # Display löschen

# UART:
uart1 = UART(1, baudrate = 57600, tx = 12, rx = 13)

# RTC
rtc = RTC()

#################### Einstellungen ####################

delimiter = '/' # ggf. anderes Zeichen
data_fname = 'lorawan_data.txt'
```

Das vollständige Programm finden Sie in dem Materialien-Ordner (s. [MPD]) unter dem Namen `lora_to_flash.py`.

Führen wir nun einen Test des Programms durch! Zunächst schließen wir den Node (LoRa-Modul plus USB-UART-Wandler) an den PC an, starten das *Hterm*-Programm (oder ein anderes Terminal-Programm) und nehmen die erforderlichen Einstellungen vor.

Dann schließen wir auch unser Breadboard mit dem TTGO T-Display und dem LoRa-Modul an den PC an und starten die Thonny-Entwicklungsumgebung. Nun laden wir das soeben vorgestellte Programm `lora_to_flash.py` und starten es. Auf dem Thonny-Terminal erscheinen die folgenden Zeilen:

```
4 Bytes gesendet: b'AT\r\n'
auf Nachricht warten...
```

```
b'+OK\r\n'
13 Bytes gesendet: b'AT+ADDRESS?\r\n'
auf Nachricht warten...
Eigene LORA-Adresse: 16

auf Nachricht warten...
```

Die angegebene LoRa-Adresse wird auch auf dem Display angezeigt.

Jetzt soll unser Node eine Botschaft an unseren TTGO senden. Dazu geben wir mit *HTerm* das AT-Kommando:

```
AT+SEND=16,10,Hallo TTGO
```

Unmittelbar danach erscheint auf dem Thonny-Terminal die Anzeige:

```
s = 123,10,Hallo TTGO,-17,11

['123', '10', 'Hallo TTGO,-17,11\r\n']
send_message: AT+SEND=123,13,MSG confirmed
30 Bytes gesendet: b'AT+SEND=123,13,MSG confirmed\r\n'
auf Nachricht warten...
b'+OK\r\n'

auf Nachricht warten...
```

Im Bereich Received Data von *Hterm* sehen wir sogleich die Quittung:

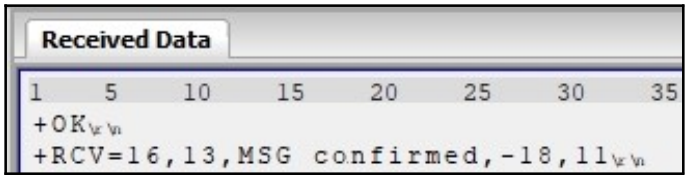


Abb. 11.2

Beenden wir nun unser Micropython-Programm mit der Stop-Schaltfläche, so sehen wir im Thonny-Bereich "MicroPython device" die gerade angelegte Datei "lorawan_data.txt". Wir können sie mit dem Thonny-Editor öffnen, indem wir sie doppelt anklicken (Abb. 11.3).

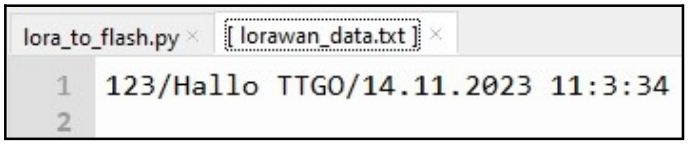


Abb. 11.3

Wir schließen nun wieder das `lorawan_data.txt`-Fenster und senden als nächstes eine Botschaft, welche ein Komma enthält, z. B. "345,7 V". Wenn wir nun die `lorawan_data.txt`-Datei erneut öffnen, wird der Inhalt

123/Hallo TTGO/14.11.2023 11:3:34
123/345,7 V/14.11.2023 11:31:30

angezeigt. In der zweiten Zeile sehen wir: Der User-Payload "345,7 V" wurde korrekt (inklusive Komma!) gespeichert.

Vom Speicher der LoRaWAN-Station per WLAN zum Client

Wir greifen hier auf das **Server**-Programm `sta_name_server_0.py` aus dem Kapitel 10 meines Skripts ***Wlan-Experimente mit dem ESP32-Board TTGO T-Display*** [WLN] zurück. In diesem Programm müssen wir lediglich die Funktion `html` ersetzen. Diese Funktion sah so aus:

```
def html(request):
  getparam = request.split(' ')[1] # Element 1 von ['GET', '/name', ... ]
  myname = getparam[1:] # '/' gehört nicht zum Namen
  return '<html><body><h1>Hallo '+myname+'</h1></body></html>'
```

Zunächst zerlegt sie den Request mit Hilfe der `split`-Methode in seine einzelnen Bestandteile. Wenn im Browser des Clients z. B.

http://198.162.2.105/581

eingegeben worden ist, dann wird `getparam` aus dem Element mit dem Index 1, also aus der Zeichenkette '/581' bestehen. Das liefert uns die Adresse desjenigen Nodes, von dem wir die gespeicherten Daten anschauen wollen. Dazu beseitigen wir nur noch das '/'-Zeichen:

```
lora_addr = getparam[1:]
```

Die Zeichenkette für das HTML-Dokument wird jetzt deutlich komplexer: Sie muss ja alle Datensätze beinhalten, welche zu der Adresse gehören. Dazu wird der BODY-Inhalt sukzessive mitsamt den erforderlichen Formatierungsanweisungen (
 für einen Zeilenvorschub) aus den Datensätzen der Datei `lorawan_data.txt` aufgebaut. Hierzu werden die Daten von `lorawan_data.txt` zunächst mit der Funktion `read_as_list` in Form einer Liste ausgelesen; dabei beinhalt das 1. Listenelement (Index 0) den 1. Datensatz (also die Informationen der ersten Zeile von der Datei `lorawan_data.txt`), das 2. Element (Index 1) den 2. Datensatz (2. Zeile), ... Allerdings bestehen diese Listenelemente nicht aus einer Zeichenkette (z. B. '123/Hallo Welt/14.11.2023 11:3:34'); vielmehr bilden sie ihrerseits eine Liste (in unserem Beispiel ['123', 'Hallo TTGO', '14.11.2023 11:3:34']). Dadurch können wir recht einfach auf die einzelnen Daten zugreifen.

Die Funktion `read_as_list` sieht so aus:

```
def read_as_list(fname, delimiter):
    liste = []
    f = open(fname, 'r')
    for x in f: # zeilenweise auswerten; entspricht readlines()
        x = x[0:len(x)-2] # \n\r abtrennen
        listen_element = x.split(delimiter)
        liste.append(listen_element)
    return liste
```

Die `html`-Funktion muss neben `request` auch `data_fname` und `delimiter` als Parameter besitzen. Sie sieht dann so aus:

```
def html(request, data_fname, delimiter):
    getparam = request.split(' ')[1] # Element 1 von ['GET', '/name', ... ]
    lora_addr = getparam[1:] # '/' gehört nicht zum Namen
    lorawan_liste = read_as_list(data_fname, delimiter)
    body_content = 'Daten von Adresse ' + lora_addr + '<BR><BR>'
    count = 0 # zählt die zur eingegebenen Adresse gehörigen Datensätze
    for i in lorawan_liste:
        if i[0] == lora_addr:
        body_content = body_content + i[1] + '   [DT: ' + i[2] + ']<BR>'
            count += 1
    if count != 0:  # wenn Daten vorhanden...
        return '<html><body><h1> '+ body_content + '</h1></body></html>'
    else:
        return '<html><body><h1>' + body_content + 'Keine Daten
                                   vorhanden...'+ '</h1></body></html>'
```

Im Prinzip ist damit unser Server-Programm fertig. Leider senden manche Browser (z. B. Chrome) einen zusätzlichen Request an den Server: Sie fragen über den Parameter `/favicon.ico` nach einem *Favorite Icon* . Einen solchen Request müssen wir (mit einer entsprechenden Verzweigung) abfangen. Ein Programm, welches auch diesen Umstand berücksichtigt, finden Sie in der Datei `flash_to_wlan_4.py` (s. [MPD]).

Eine Abfrage zur Adresse 123 führt bei meinem Handy nun zu folgender Anzeige:

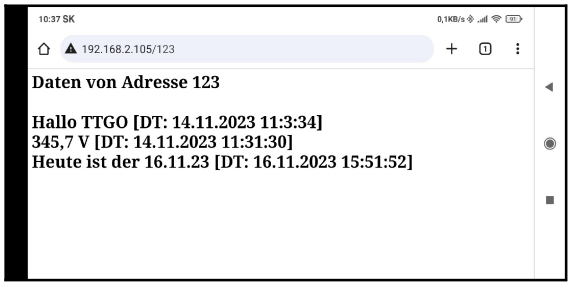


Abb. 11.4

Gesamtprogramm für die LoRaWAN-Station

Unsere LoRaWAN-Station muss die folgenden Aufgaben erfüllen:

1. Sie muss die von einem Endgerät gesendete Botschaften empfangen und im Flash speichern.

2. Sie muss als Server arbeiten: Auf Anfragen (Requests) von Browsern hin muss sie die geforderten Daten in Form eines HTML-Dokuments (per WLAN) an den Client senden.

Im Prinzip müssen dazu nur die beiden Programme `lora_to_flash.py` und `flash_to_wlan_4.py` zu einem einzigen Programm zusammen gefasst werden. Dabei gibt es allerdings zwei Probleme:

Problem 1

Kern des Programms `lora_to_flash.py` ist eine Endlos-Schleife, in der andauernd kontrolliert wird, ob eine LoRa-Nachricht angekommen ist: In der Funktion `receive_save_confirm` wird nämlich die Funktion `read_response` aus dem Modul `lora.py` aufgerufen; und in dieser Funktion sehen wir die folgende Schleife:

```
while not uart.any():
    sleep(0.1)
```

Diese wird erst beendet, wenn sich ein Zeichen im Lesepuffer der UART befindet. Das bedeutet: Solange keine LoRa-Nachricht eintrifft, läuft diese Schleife ohne Unterbrechung weiter.

Eine ähnliche Situation liegt bei dem Programm `flash_to_wlan_4.py` vor: Hier wird in einer Endlos-Schleife gewartet, bis eine Anfrage von einem Client vorliegt:

```
while True:
    connection, addr = s.accept()
    print("Client connected: ", addr)
    ...
```

Beide Schleifen sind in ihrer Wartestellung blind für andere Belange. Dieses Problem lässt sich lösen, indem man die Aufgaben des Programms `lora_to_flash.py` in einem (neuen) **Thread** ausführen lässt. Die Aufgaben des Programms `flash_to_wlan_4.py` verbleiben im Hauptprogramm; dieser kann auch als ein Thread (**Mainthread**) angesehen werden. Die Befehle im Mainthread und im neu erzeugten Thread werden nun **quasi gleichzeitig** ausgeführt, **ohne sich gegenseitig zu beeinflussen**. (Mehr Informationen zum Umgang mit Threads finden Sie in der Textbox auf der nächsten Seite.) Damit kann die Überprüfung des UART-Puffers (aus dem neu erzeugten Thread) und das Warten auf eine Anfrage von einem Client (im Mainthread) unabhängig von einander erfolgen.

Problem 2

Es kann geschehen, dass der neue Thread gerade in dem Augenblick einen Eintrag in unserer Datei `lorawan_data.txt` vornehmen möchte, während dieselbe Datei vom Mainthread ausgelesen wird. Natürlich ist diese Situation recht unwahrscheinlich, aber sie ist möglich und kann gegebenenfalls zu Konflikten führen. Das Problem lässt sich lösen, indem man beim Lesen oder Schreiben einer Datei mit Hilfe eines lock-Objekts weiteren Zugriffsversuchen einen Riegel vorschiebt. Wie man dabei vorgeht, zeigt das Beispielprogramm in der Textbox auf der nächsten Seite. (Tatsächlich wird dieses Problem gerne als Beispiel für das `lock`-Objekt benutzt.)

Wie kann man sich nun die Bearbeitung solcher Threads durch den Mikrocontroller vorstellen? Tatsächlich gibt es zahlreiche Mikrocontroller mit mehreren Prozessor-Kernen: Jeder **Kern** (Core) kann autonom arbeiten. Der ESP32 des TTGO besitzt z. B. 2 Kerne. Allerdings ist hier einer der beiden Kerne (Core 1) für die Bearbeitung ganz bestimmter Aufgaben reserviert. Daher müssen unsere beiden Threads in einem einzigen Kern (Core 0) bearbeitet werden. Dazu benutzt das Micropython-System einen so genannten **Scheduler**; dieser stellt nach einem Zeitplan den einzelnen Threads gewisse Zeitspannen zur Bearbeitung ihrer Aufgaben (im Core 0) zur Verfügung. Das bedeutet: **Die Threads werden genau genommen nicht gleichzeitig bearbeitet**. Vielmehr bearbeitet der Core 0 **nacheinander** für eine kurze Zeitspanne mal die Aufgaben des einen und mal die Aufgaben des anderen Threads (oder ggf. sogar noch die von weitereren Threads). Da der Wechsel von Thread zu Thread sehr rasch erfolgt, erweckt es den Eindruck, als ob die Threads gleichzeitig bearbeitet würden. Dabei muss der Scheduler dafür sorgen, dass bei jedem Verlassen eines Threads der jeweilige Bearbeitungsstand genau festgehalten wird. Nur so kann der Prozessor von Core 0 beim erneuten Wechsel zu diesem Thread korrekt weiterarbeiten.

Dass diese Wechsel sehr rasch erfolgen, kann das Programm `thread_test_4.py` (s. [MPD]) verdeutlichen: Hier existieren neben dem Mainthread gleich zwei weitere Threads. Einer dieser Threads sorgt dafür, dass eine LED mit einer Frequenz von 25 Hertz ein- und ausgeschaltet wird. Wenn der Prozessor mit der Ausgabe von längeren Zeichenketten auf dem Thonny-Terminal beschäftigt war, konnte ich tatsächlich Unregelmäßigkeit beim Blinken feststellen.

Dass dieses Scheduling nicht ganz einfach ist, habe ich beim Programmieren des "Gesamtprogramms" erfahren müssen: Der Wechsel von der einen Endlos-Schleife zur anderen wollte nicht funktionieren. Der Grund lag (etwas versteckt) in der Funktion `read_response` aus dem Modul `lora.py`. Ursprünglich hatte diese Funktion mit der folgenden Schleife auf ein Zeichen im Lese-Puffer gewartet:

```
while not uart.any():
    pass
```

Das funktionierte aber nicht: Vermutlich konnte dieser Prozess vom Scheduler nicht unterbrochen werden; jedenfalls kam es nicht mehr zu einem Wechsel in den Mainthread. Nachdem ich `pass` durch `sleep(0.1)` ersetzt hatte, war das Problem verschwunden…

Arbeiten mit Threads

Bei **Threads** handelt es sich um Prozesse, die parallel und unabhängig von einander arbeiten. Bei dem folgende Beispiel wird im Mainthread (Hauptprogramm) jede 2 Sekunden eine Zahl um 1 erhöht und als Zeichenkette in der Datei `'test.txt'` gespeichert. Zu Beginn des Hauptprogramms starten wir mit `_thread.start_new_thread(test, ())` einen *neuen* Thread, der durch die Funktion `test()` festgelegt ist: Durch diese Funktion wird jede 0,6 Sekunden die gesamte Datei `'test.txt'` gelesen und im Terminal von Thonny ausgegeben. Damit nun ein Lesevorgang in unserem Thread nicht von einem Schreibvorgang des Hauptprogramms gestört wird (oder umgekehrt), **sperren** wir diese Schreib- und Lesevorgänge kurzfristig vor einem "fremden" Zugriff; dies geschieht mit Hilfe des Objekts `lock`.

```
import time
import _thread

lock = _thread.allocate_lock() # lock-Objekt erzeugen
file = open("test.txt", "w") # bestehende Datei löschen...
file.write('0 ') # ... und '0 ' speichern
file.close()

def test():
    while True:
        with lock: # mit Sperren (MUTEX) ausführen...
            with open("test.txt", "r") as file: # automatisches close
                print('Lesen: ', file.read())
        time.sleep(0.6)

# Funktion test() in einem neuen Thread ausführen lassen
_thread.start_new_thread(test, ())

# Hauptprogramm (Mainthread: Sekundenzähler mit Schleife)
count = 0
while True:
    time.sleep(2)
    with lock: # s. o.
        count += 1
        with open("test.txt", "a") as file: # automatisches close
            print('Schreiben: ', count)
            file.write(str(count)+' ')
```

Dieses Programm finden sie in der Datei `thread_test_3.py`.

Bemerkung: Mit Hilfe des 2. Parameters von `_thread.start_new_thread(test, ())` könnten der Thread-Funktion Parameter in Form eines Tupels übergeben werden.

Da die entscheidenden Probleme nun gelöst worden sind, möchte ich darauf verzichten, das Gesamtprogramm für die LoRaWAN-Station hier aufzulisten. Bei diesem Gesamtprogramm habe ich die Ausgabe von Zwischenwerten beibehalten und zusätzlich angegeben, aus welchem Thread sie stammen. Auf diese Weise kann man das Wechselspiel der beiden Threads recht gut nachvollziehen. Wer will, kann die entsprechenden `print`-Anweisungen löschen oder auskommentieren. Sie finden das Programm in der Datei mit dem Namen `lorawan_1.py` (vgl. [MPD]).

Jetzt ist es an der Zeit, das Programm zu testen: Dazu wollen wir zunächst mit einem Handy Daten abfragen, und zwar einmal zur Adresse 123 und einmal zur Adresse 12. Anschließend senden wir von unserem Endgerät mit der Adresse 12 (ggf. erst Adresse mit AT-Command festlegen!) eine neue Botschaft an unsere LoRaWAN-Station. Im nächsten Schritt fragen wir mit dem Handy die Daten zur Adresse 12 erneut ab. Jetzt kontrollieren wir einmal, wie die LoRaWAN-Station reagiert, wenn wir nach Daten von einer Adresse abfragen, zu der es keine Datensätze in der Datei `lorawan_data.txt` gibt. Abschließend senden wir mit unserem Endgerät eine letzte Nachricht.

```
123/Hallo TTGO/14.11.2023 11:3:34
123/345,7 V/14.11.2023 11:31:30
23/Test_lorawan/15.11.2023 12:44:37
23/ABCDE/15.11.2023 13:55:51
123/Heute ist der 16.11.23/16.11.2023 15:51:52
23/Nachricht/16.11.2023 16:42:12
12/Neues Endgeraet/16.11.2023 16:1:46
```

Abb. 11.5

Zu Beginn unseres Experiments ist der Inhalt dieser Datei wie in Abb. 11.5 dargestellt. Der Verlauf der Ereignisse ist in der folgender Tabelle aufgeführt:

Aktion	Mainthread	newThread
Programm starten	LoRa-Modul initialisieren	
		rsc-Schleife starten: auf Antwort warten
	WLAN connected, Server listening	
Daten zur Adresse 123 abfragen	Client connected; zugehörige Daten an Client senden	
Handy empfängt Daten zu Adresse 123		
Daten zur Adresse 12 abfragen	Client connected; zugehörige Daten an Client senden	
Handy empfängt Daten zu Adresse 12		

Endgerät mit der Adresse 12 sendet Nachricht		empfängt neue Nachricht vom Endgerät mit der Adresse 12 und sendet Quittung an das Endgerät
Endgerät erhält Empfangsbestätigung		
Daten zur Adresse 12 abfragen	Client connected; zugehörige Daten an Client senden	
Handy empfängt Daten zu Adresse 12 (inkl. neuer Botschaft)		
Daten zur Adresse 99 abfragen	Client connected; “Keine Daten vorhanden” an Client senden	
Handy empfängt “Keine Daten vorhanden”		
Endgerät mit der Adresse 12 sendet weitere Nachricht (“ENDE”)		empfängt neue Nachricht von Adresse 12 und sendet Quittung
Endgerät erhält Empfangsbestätigung		

Die Datei `lorawan_data.txt` sieht am Ende wie in Abb. 11.5 aus.

```
123/Hallo TTGO/14.11.2023 11:3:34
123/345,7 V/14.11.2023 11:31:30
23/Test_lorawan/15.11.2023 12:44:37
23/ABCDE/15.11.2023 13:55:51
123/Heute ist der 16.11.23/16.11.2023 15:51:52
23/Nachricht/16.11.2023 16:42:12
12/Neues Endgeraet/16.11.2023 16:1:46
12/Hallo von 12/21.11.2023 9:58:25
12/ENDE/21.11.2023 10:11:7
```

Abb. 11.5

Installationen

Zur Webseite von **CubicSDR** gelangen Sie über die URL https://cubicsdr.com/. Hier gibt es auch einen Link zum Download der Installationssoftware:

github.com/cjcliffe/CubicSDR/releases/tag/0.2.4

Wenn Sie mit dem Betriebssystem Windows (64 Bit) arbeiten, dann laden Sie die Datei CubicSDR-0.2.4-win64.exe herunter. Starten Sie das Installationsprogramm und folgen Sie den Anweisungen.

Ein Manual finden Sie unter https://cubicsdr.readthedocs.io/en/latest/.

Bitte beachten Sie: Das *CubicSDR* -Programm benötigt spezielle Funkmodule. Im Folgenden beschreibe ich die Installation des Funkmoduls NEW GEN.RTL2832.

Zur **Installation des NEW-GEN.RTL2832-Moduls** verbinden Sie zunächst die Antenne mit dem Modul. Anschließend stecken Sie das Modul in eine USB-Buchse Ihres Rechners. Nach kurzer Zeit meldet das Betriebssystem, dass das Gerät betriebsbereit ist. Das stimmt aber leider nicht! Es zeigt sich, dass zwar ein Treiber installiert wurde; mit diesem Treiber kann *CubicSDR* aber nicht arbeiten.

Deswegen muss der installierte Treiber durch einen anderen ersetzt werden. Sie brauchen jetzt aber gar nicht den falschen Treiber deaktivieren oder entfernen. Vielmehr setzen wir nunmehr ein Programm ein, welches den falschen Treiber direkt durch einen passenden ersetzt.

Dieses Programm heißt ***Zadig*** . Die neueste Version von Zadig (zadig-2.8.exe; Stand 01.03.2023) kann von der Website zadig.akeo.ie herunter geladen werden. Starten Sie nun das Programm. In dem Programmfenster aus Abb. 1 aktivieren Sie zunächst über das *Options* -Menü *List All Devices*. Nun wählen Sie in dem Auswahlfeld das Gerät *Bulk_in, Interface (Interface 0)* aus. Links vom

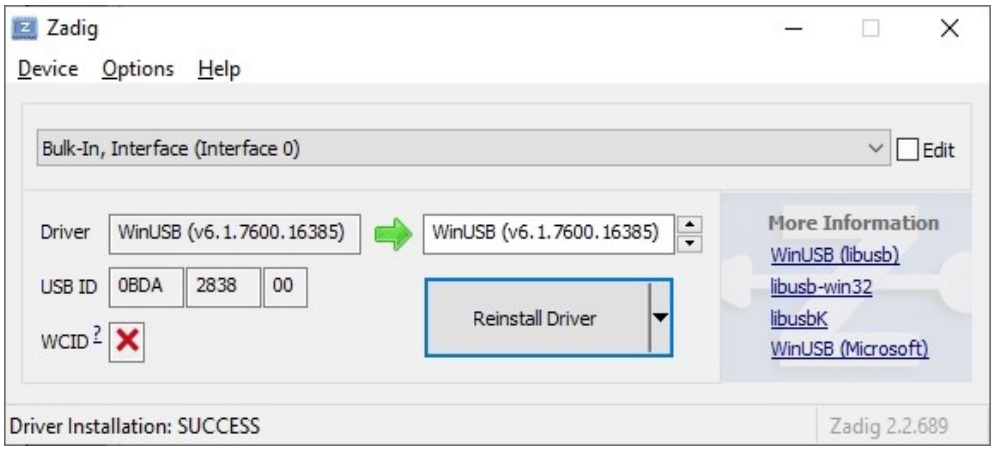


Abb. A1

grünen Pfeil wird bei Ihnen als aktueller Treiber wahrscheinlich irgendetwas anderes stehen als in der Abbildung; das ist aber unwichtig. Wichtig ist lediglich, dass die angezeigte USB ID mit der von Abb. 1 identisch ist.

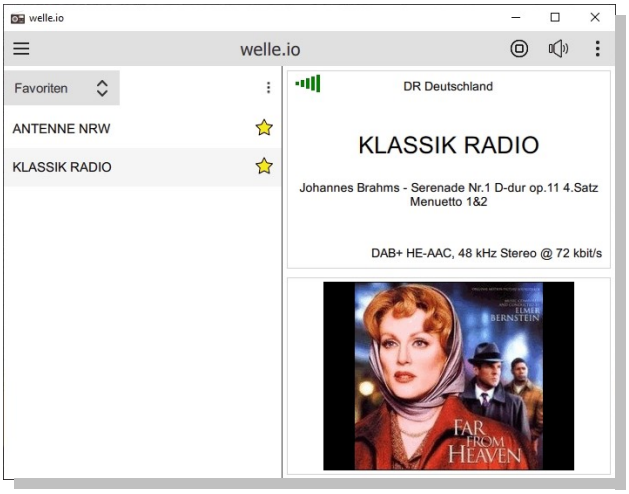
Jetzt brauchen Sie nur noch auf die Schaltfläche *reinstall Driver* zu klicken und schon wird der Austausch der Treiber vorgenommen. Dies kann einige Zeit in Anspruch nehmen. Wenn der Vorgang erfolgreich beendet worden ist, erscheint eine entsprechende Meldung im Programmfenster.

Bemerkung

Das RTL2832-Modul kann auch anderweitig benutzt werden:

- Mit dem *CubicSDR* -Programm kann man UKW-Sender (mit analogem FM-Signal) hören.
- Mit dem Programm ***welle.io*** kann man Sendungen hören, die mit DAB+ ausgestrahlt werden. Information zu diesem Programm findet man unter https://www.welle.io/. Über die Website https://github.com/AlbrechtL/welle.io/releases gelangt man direkt zu den Downloadmöglichkeiten.

Übrigens: Bei mir erfolgte am Ende des Downloads des Installationsprogramms für welle.io der Hinweis, dass der Download nicht bestätigt werden konnte. Tatsächlich wurde die Installationsdatei von meinem PC fälschlich als Medien-Datei gedeutet. Ich habe sie dann einfach in eine exe-Datei umbenannt; und diese lief dann klaglos...



Quellen

[WLN]: https://www.g-heinrichs.de/wordpress/index.php/informatik/ttgo/

[BLE]: https://www.g-heinrichs.de/wordpress/index.php/informatik/ble-mit-dem-esp32/

[BNA]: https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Frequenzen/Allgemeinzuteilungen/FunkanlagenGeringerReichweite/2018_05_SRD_pdf.pdf?__blob=publicationFile&v=7]

[ERC]: https://docdb.cept.org/download/25c41779-cd6e/Rec7003e.pdf

[LFF]: https://www.techplayon.com/lora-long-range-network-architecture-protocol-architecture-and-frame-formats/

[AMO]: https://de.wikipedia.org/wiki/Amplitudenmodulation

[FMO]: https://de.wikipedia.org/wiki/Frequenzmodulation

[SE1]: https://www.frugalprototype.com/wp-content/uploads/2016/08/an1200.22.pdf

[RFM]: https://www.forum.g-heinrichs.de/viewtopic.php?f=12&t=54

[HOW]: https://www.youtube.com/watch?v=jHWepP1ZWTk

[R/P]: Reynders/Pollin: Chirp Spread Spectrum as a Modulation Technique for Long Range Communication
https://www.researchgate.net/publication/311980840_Chirp_spread_spectrum_as_a_modulation_technique_for_long_range_communication

[BRO] Gilles Brocard: LoRa - ein Schweizer Taschenmesser, Elektor, September/Oktober 2023

[SDS] https://www.semtech.cn/products/wireless-rf/lora-connect/sx1276#documentation

[CRC] https://www.epfl.ch/labs/tcl/wp-content/uploads/2020/02/Reverse_Eng_Report.pdf

[RAU] https://de.wikipedia.org/wiki/Rauschen_(Physik)

[KRE] https://de.wikipedia.org/wiki/Kreuzsicherung

[GSM] https://forum.g-heinrichs.de/viewtopic.php?f=12&t=94

[USI] https://unsigned.io/understanding-lora-parameters

[SMI] https://smartmakers.io/lorawan-reichweite-teil-1-die-wichtigsten-faktoren-fuer-eine-gute-lorawan-funkreichweite/

[LSC] https://www.rfwireless-world.com/calculators/LoRa-Sensitivity-Calculator.html

[LD1] RYLR998_AT_Commands.pdf in Materialien-Ordner

[LD2] RYLR998_Datasheet.pdf in Materialien-Ordner

[DRS] https://www.thethingsnetwork.org/docs/lorawan/modulation-data-rate/

[LRW] https://de.wikipedia.org/wiki/Long_Range_Wide_Area_Network#LoRa

Materialien

[FTS] Excel-Datei Fouriertransformation_zeitl_begr_Sinusschwingung.xlsx (in Materialien.zip)

[ACF] Akustische Chirp-Folge in der Audio-Datei chirps.wav (in Materialien.zip)

[KOR] Excel-Datei Korrelation_mit_Rauschen_2.xlsx (in Materialien.zip)

[TOA] Excel-Datei ToA.xlsx (in Materialien.zip)

[MPD] Micropython-Dateien (in Materialien.zip)

[ESP] https://medium.com/gowombat/iot-lora-with-micropython-on-the-esp8266-and-esp32-59d1a4b507ca (*Micropython-Treiber für LoRa-Modul SX127x und ESP32*)

Den Ordner Materialien.zip können Sie über die folgende Webseite herunterladen:

https://www.g-heinrichs.de/wordpress/index.php/informatik/lorawan-mit-esp32-und-rylr998/

Videos

[VID] https://www.youtube.com/watch?v=LiWlPERp1ec (Den RYLR998 mit USB-SERIAL-Konverter bzw. Arduino-Mikrocontroller betreiben)

[ANI] https://www.youtube.com/watch?v=dxYY097QNs0 (LoRa-Chirp von Richard Wenner)

[ANW] https://www.badenova.de/blog/lorawan-einfach-erklaert

[MBF] https://lora.readthedocs.io/en/

Etwas mehr zur Theorie…

[MBF] https://lora.readthedocs.io/en/ (Serie von Video-Tutorials)

[R/P]: Reynders/Pollin: Chirp Spread Spectrum as a Modulation Technique for Long Range Communication
https://www.researchgate.net/publication/311980840_Chirp_spread_spectrum_as_a_modulation_technique_for_long_range_communication

[MOD] https://wirelesspi.com/i-q-signals-101-neither-complex-nor-complicated/ (Verschiedene Modulationsarten)

[IOT] https://www.thethingsnetwork.org/article/how-spreading-factor-affects-lorawan-device-battery-life

[LFB] https://www.univ-smb.fr/lorawan/wp-content/uploads/2022/01/Book-LoRa-LoRaWAN-and-Internet-of-Things.pdf

Stichwortverzeichnis