Seminar Radio Packet Network - Universität Basel
AFSK Modem

Yaowen Rui, Beatriz Balboa Ruiz

17. Februar 2025

1 Abstract

1.1 Projekt Überblick

Das Ziel dieses Projekts ist die Entwicklung eines softwaredefinierten AFSK-Modems, das für digitale Datenübertragung über analoge Kommunikationskanäle eingesetzt werden kann. Dieses Modem nutzt die Standard-Audioeingänge und -ausgänge eines Computers und ermöglicht das Senden und Empfangen von Nachrichten ohne zusätzliche Hardware.

1.2 Arbeitsteilung

Wir haben beide an allen Teilen des Projekts gearbeitet, den Code gegenseitig korrigiert und optimiert. Alle wichtigen Entscheidungen und Implementationen wurden gemeinsam getroffen.

2 Systembeschreibung

2.1 Bestandteile

Das Modem besteht aus vier Hauptkomponenten:

- **Sender**: kodiert die Nachrichten und spielt sie entweder direkt über den Audioausgang des Computers ab oder speichert sie in einer `.wav`-Datei.
- **Empfänger**: dekodiert die empfangenen Audiosignale. Er kann die Signale direkt aufnehmen oder sie aus einer `.wav`-Datei lesen.
- **Wave-Generator**: erzeugt drei Sinus-Töne: einen Space-Ton (1200,Hz) für Bit 0, einen Mark-Ton (2200,Hz) für Bit 1 und einen Synchronisationston (3000,Hz) zu Beginn.
- **Byte-Bit-Konverter**: wandelt Bytes in Bitstrings um und umgekehrt.

2.2 Vollständiger Prozess

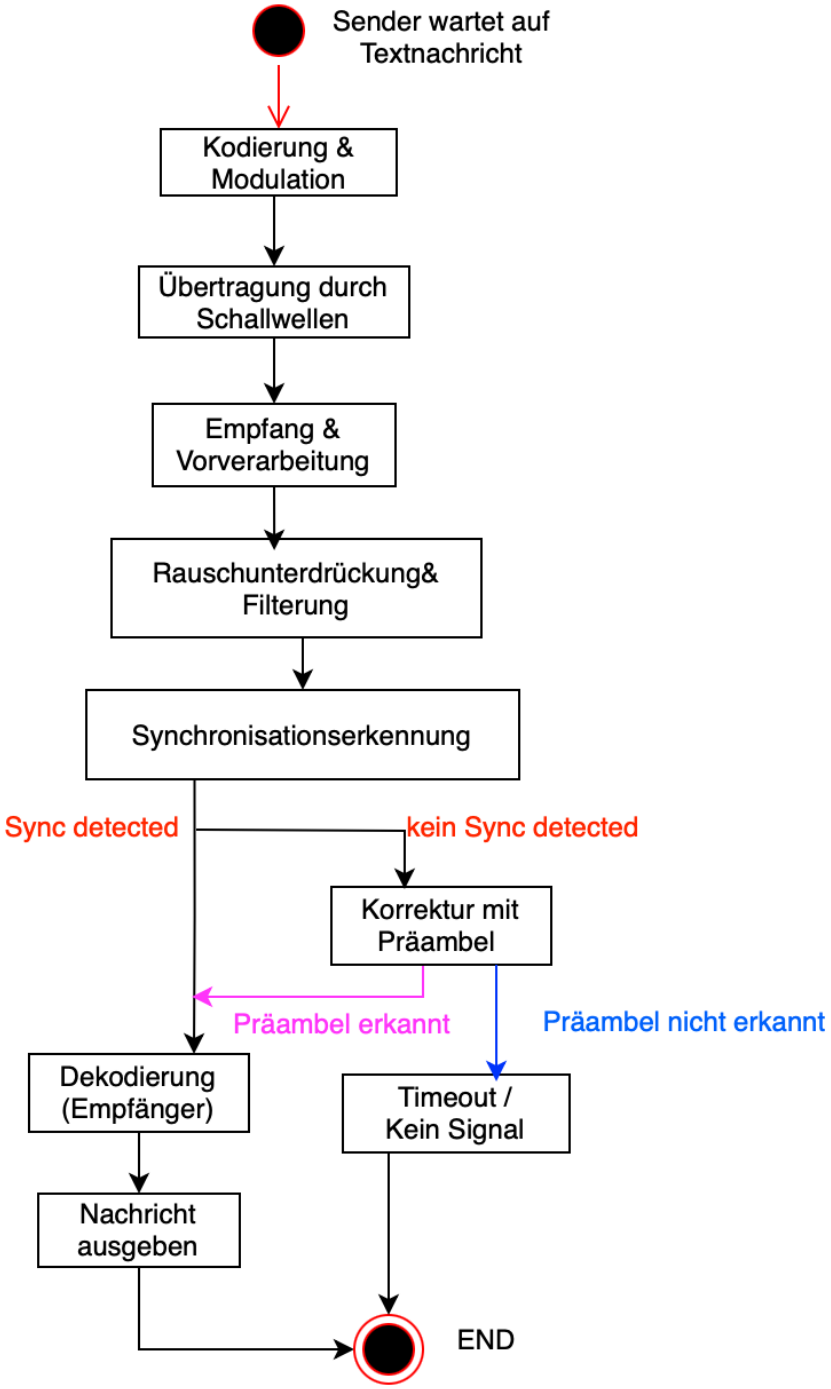

Abbildung 1: Prozess-Zustandsdiagram

3 Implementation

Wir verwenden eine Baudrate von 300 Bits/s bei einer Abtastrate von 48 kHz. Im Sender wird eine Kombination aus Space-/Mark-Tönen (1200/2200 Hz) erzeugt, gefolgt vom Synchronisationston (3000 Hz) für 3 s.

3.1 Sender

Der Sender wandelt die Nachricht in ein AFSK-Signal um und kann es entweder direkt über den Lautsprecher ausgeben oder als `.wav` Datei speichern.

- **Text zu Bitstring**: Die Nachricht wird in UTF-8 kodiert und in einen Bitstring umgewandelt.
- **AFSK-Modulation**: Bits werden in 1200-Hz- (0) oder 2200-Hz-Töne (1) umgewandelt.
- **Synchronisation**: Ein 3000-Hz-Ton wird am Anfang für 3 Sekunden gesendet.
- **Speicherung oder Wiedergabe**: Das Signal kann abgespielt oder gespeichert werden.

3.2 Empfänger

Der Empfänger dekodiert das empfangene Signal in mehreren Schritten:

- **Bandpass-Filter**: Entfernt Frequenzen ausserhalb des 1000-2500 Hz Bereichs.
- **Spektrale Subtraktion**: Entfernt Hintergrundgeräusche basierend auf einem vorab aufgenommenen Rauschprofil.
- **Synchronisationserkennung**:
  - Falls der 3000-Hz-Ton erkannt wird, beginnt die Dekodierung danach.
  - Falls nicht, wird nach der Prämbel gesucht, um den Start zu bestimmen.
- **FFT-Analyse mit Hamming-Fenster**: Identifiziert dominierende Frequenzen pro Bit-Frame.
- **Bitstring-Rekonstruktion**: Bestimmt den Bitstring basierend auf den Spektraldaten.
- **Live-Dekodierung**: Das empfangene Signal wird in einem temporären Buffer gespeichert und anschliessend offline verarbeitet.
- **Dekodierung aus Datei**: Der Empfänger kann das Signal aus einer `.wav` Datei lesen und dekodieren.

3.3 Methodologie

Hauptproblem:
Da die Signalübertragung über Schallwellen zwischen zwei Geräten erfolgt, kann das empfangene Signal durch verschiedene Störungen stark beeinträchtigt werden, wie z.B. durch Umgebungsgeräusche oder Überlagerung von Echo-Signalen. Diese erschweren die Dekodierung und können die Genauigkeit verringern.

Lösungsansätze:
Ein Bandpass-Filter wird verwendet, um Rauschen ausserhalb des Frequenzbereichs (1000 - 2500 Hz) zu eliminieren. Zusätzlich wird die Spektrale Subtraktion einge-

setzt, um Störgeräusche weiter zu minimieren. Diese Methode setzt voraus, dass zuvor ein Rauschprofil aufgenommen wurde, das als Referenz zur Rauschreduktion dient.

Anschliessend wird das Signal in einzelne Bit-Blöcke unterteilt, wobei unterschiedliche Bit-Offsets getestet werden, um die bestmögliche Dekodierung zu erreichen. Zusätzlich wird eine Korrelation mit der Präambel-Sequenz verwendet, um den exakten Startpunkt der Datenbits zu bestimmen und die Synchronisation zu verbessern.

Das dekodierte Signal wird schliesslich auf gültige Zeichen überprüft, indem der Anteil an lesbaren Zeichen berechnet wird. Der Text mit der höchsten Bewertung wird als empfangene Nachricht ausgewählt.

4 Code Überblick

Methode	Beschreibung
prepare_for_sending	Kodiert einen Text in eine Bitfolge, wandelt diese in Audiosignale um (Mark-/Space-Töne) und fügt eine Synchronisationssequenz sowie eine Pause am Ende hinzu.
send_msg	Sendet Audiosignal über den Lautsprecher.
write_to_file	Speichert Audiosignal als WAV-Datei mit angegebenen Filename.

Abbildung 2: Methoden des Senders

Methode	Beschreibung
bandpass_filter	Entfernt unerwünschte Frequenzen aus dem Audiosignal
measure_noise_profile	Misst das Hintergrundrauschen, um es später bei der spektralen Subtraktion zu berücksichtigen
spectral_subtraction	Reduziert Störgeräusche aus dem Audiosignal
compute_band_power	Berechnet die Leistung eines bestimmten Frequenzbereichs im FFT-Spektrum
detect_bit_adaptive	Bestimmt anhand einer spektralen Analyse, ob ein Bit als 0 oder 1 interpretiert wird
_reform_bit_string_adaptive	Zerlegt das Audiosignal in Bit-Blöcke und dekodiert es schrittweise
decode_with_offset	Dekodiert das Audiosignal mit einem bestimmten Offset, um eine genauere Bitfolge zu erhalten
score_text	Bewertet einen dekodierten Text basierend auf der Anzahl gültiger Zeichen, hilft festzustellen, ob der dekodierte Text sinnvoll ist.
find_preamble_offset	Sucht nach der besten Übereinstimmung zwischen dem Präambel-Bitmuster und der Bitfolge
align_and_decode	Versucht verschiedene Bit-Offsets, um die beste mögliche Dekodierung zu finden
stop_at_end_marker	Schneidet den dekodierten Text an der Endmarkierung ab
offline_decode	Dekodiert ein gespeichertes Audiosignal unter Anwendung von Filtern und Synchronisationserkennung
record_fixed_time	Nimmt über eine feste Zeitspanne Audio auf und gibt die Rohdaten zurück
read_from_file	dekodiert direkt enthaltene Nachricht von der WAV-Datei

Abbildung 3: Methoden des Empfängers

5 Ergebnisse

- **Dekodierung aus Datei**: Der Empfänger dekodiert fehlerfrei, da kaum Störungen vorhanden sind.
- **Live-Dekodierung**: Funktioniert gut bei geringer Umgebungsläutstärke, kann jedoch durch externe Geräusche beeinträchtigt werden. Die Qualität kann schwanken, insbesondere wenn die Aufnahme verrauscht ist oder Echos vorhanden sind.

5.1 Einschränkungen

- **Echounterdrückung**: In macOS kann die automatische Echo-Kompensation die eigentliche AFSK-Signalstärke herabsetzen.
- **Rauschen**: Hohe Umgebungsgeräusche können zu Bitfehlern führen.
- **Latenz**: Die Speicherung des Signals im Buffer führt zu einer geringen Verzögerung bei der Echtzeit-Dekodierung, da das gesamte Signal erst erfasst werden muss, bevor es verarbeitet wird.

6 Plots und Kommentare

Dieser Abschnitt zeigt die Plots, die generiert werden:

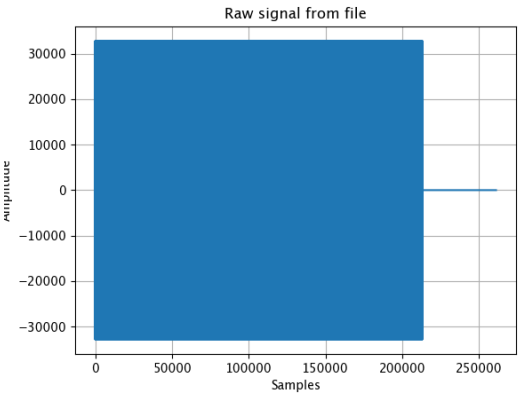

Abbildung 4: Raw signal from file

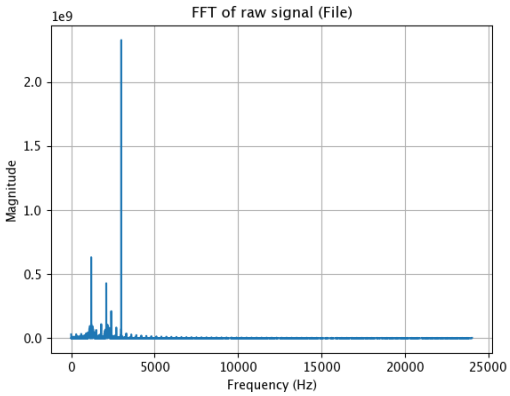

Abbildung 5: FFT of raw signal (File)

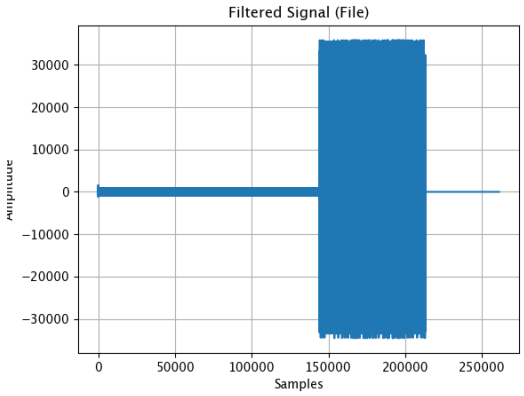

Abbildung 6: Filtered Signal (File)

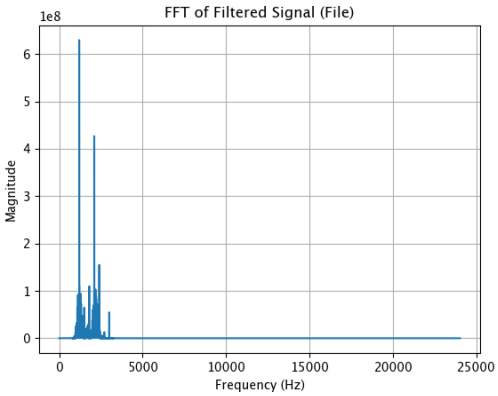

Abbildung 7: FFT of Filtered Signal (File)

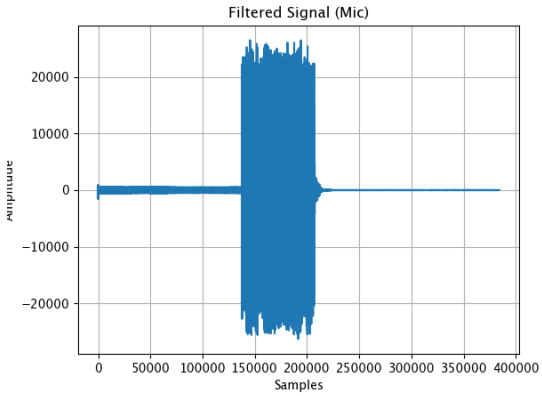

Abbildung 8: Filtered Signal (Mic)

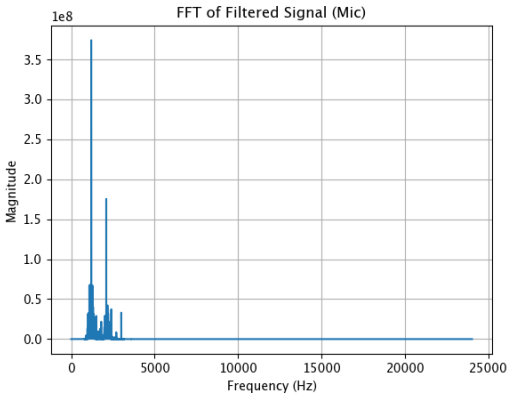

Abbildung 9: FFT of Filtered Signal (Mic)

In Abbildung 4 sieht man die ungefilterte Datei-Ausgangssignatur, wobei der Pegel fast den gesamten Bereich von ±32767 nutzt. In 5 erkennt man Peaks bei ca. 3000 Hz (Synchronisation), 1200 Hz und 2200 Hz. Nach dem Bandpass-Filter (Abbildungen 6 und 7) bleibt nur das Frequenzband 1000–2500 Hz. Die Mikrofonaufnahme (Abbildungen 8 und 9) weist oft mehr Rauschen oder abweichende Amplituden auf, da reale Umgebungsbedingungen und Echo-Kompensation hineinspielen.

7 Appendix: Finaler Code

Im Folgenden der finale Code in Python:

Listing 1: Final Code Snippet

```
import pyaudio
import numpy as np
import wave
import matplotlib.pyplot as plt
from scipy.io.wavfile import write, read
from scipy.signal import butter, sosfiltfilt, correlate
import time
```

```
import string
import threading

DEBUG = True

def plot_signal(signal, title="Signal"):
    ...

def plot_fft(signal, sample_rate, title="FFT"):
    ...

# Global Parameters
SAMPLE_RATE    = 48000
HIGH_AMPLITUDE = 32767
BAUD_RATE      = 300
SYNC_FREQUENCY = 3000
SYNC_DURATION  = 3.0
PREAMBLE_TEXT  = "<<<PREAMBLE:>>>"
END_OF_MSG     = "<<<END>>>"

SPACE_CENTER = 1200
MARK_CENTER  = 2200
BAND_WIDTH   = 100
ALPHA        = 1.2
EPSILON      = 1e-8
EXTRA_MARGIN = 4

class AFSKWaves:
    ...

class ByteBitConverter:
    ...

class Sender:
    ...

class Receiver:
    ...

if __name__=="__main__":
    receiver = Receiver()
    sender   = Sender()

    message  = "Hello, this is a test message."

    def record_task():
        receiver.measure_noise_profile(1.0, chunk_size=256)
        global recorded_data
```

```
56          recorded_data = receiver.record_fixed_time(
                record_secs=8, chunk_size=256)
57
58      rec_thread = threading.Thread(target=record_task)
59      rec_thread.start()
60      time.sleep(1)
61
62      sender.send_msg(message)
63      sender.write_to_file(message,"output.wav")
64
65      rec_thread.join()
66
67      print("Decoding from output.wav...")
68      file_decoded = receiver.read_from_file("output.wav")
69      print("Decoded from output.wav:", file_decoded)
70
71      print("Decoding from mic...")
72      real_decoded = receiver.offline_decode(recorded_data)
73      print("Decoded from mic capture:", real_decoded)
```

7.1 Output bei optimalen Bedingungen

```
Transmitting: Hello, this is a test message.
Transmission complete.
Decoding from output.wav...
Decoded from output.wav: Hello, this is a test message.
Decoding from mic...
Decoded from mic capture: Hello, this is a test message.
```

7.2 Links

Der vollständige Quellcode befindet sich auf GitHub: Code

Video der Demo: Demo