Praktikum
Digitale Übertragungstechnik

SoSe 2025

Prof. Matthias Kronauge

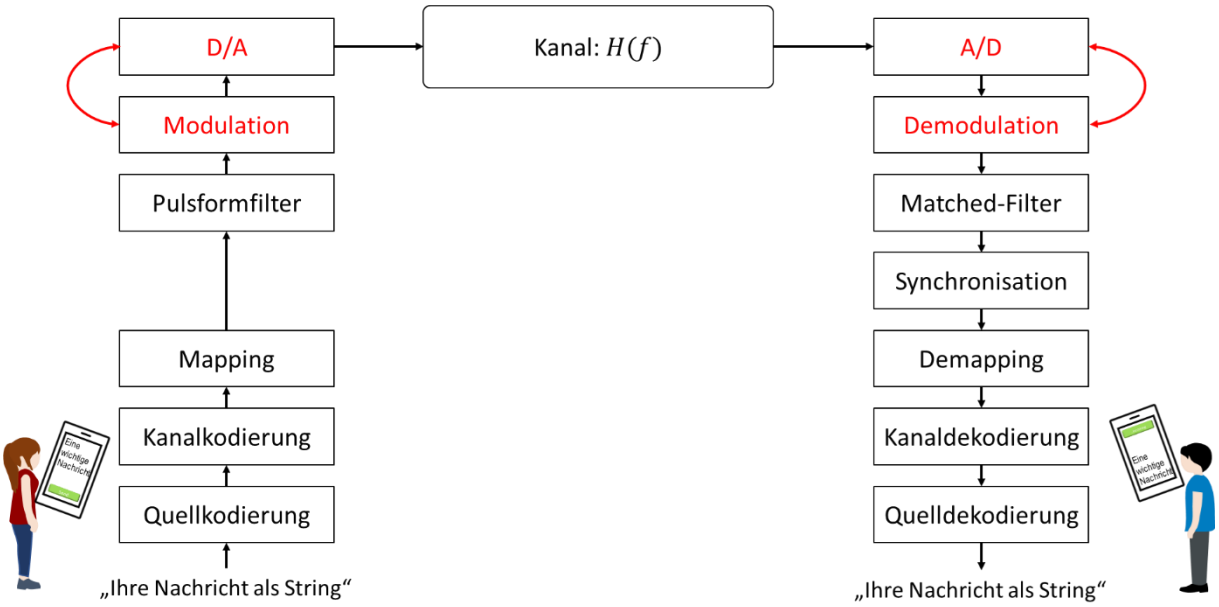
Aufbau eines digitalen Kommunikationssystems

In diesem Praktikum zur Vorlesung Digitale Übertragungstechnik soll nach und nach ein digitales Übertragungssystem realisiert werden.

Die Methoden, die Sie in der Vorlesung dazu kennenlernen basieren auf denen, die Sie bereits aus den Modulen Signale und Systeme 1 und 2 kennen. Zur Implementierung verwenden wir MATLAB. Das hat zwei Vorteile:

- Es ist sehr einfach die verwendeten Signale im Zeit- und Frequenzbereich zu Plotten und zu analysieren
- Die Implementierung ist einfacher als z.B. in C, da viele Funktionen wie die Faltung oder die FFT direkt verfügbar sind und die Handhabung von Arrays und Datentypen vereinfacht wird.

Außerdem soll die verwendete Hardware möglichst einfach gehalten werden. Daher verwenden wir als Digital-Analog-Wandler im Sender und Analog-Digital-Wandler im Empfänger jeweils eine USB-Soundkarte mit einer Abtastrate von $f_{sa} = 48\,kHz$. Um darüber hinaus keine weitere Hardware in Form z.B. eines Modulators zu benötigen, vertauschen wir die Reihenfolge von D/A-Wandler und Modulator im Sender und A/D-Wandler und Demodulator im Empfänger und führen die Modulation und Demodulation digital durch – wie im folgenden Bild dargestellt.



Als Kanal dient zunächst ein Audiokabel, das später mit Lautsprecher und Mikrophon durch die Luft ersetzt werden kann.

Legen Sie sich für dieses Praktikum einen Ordner namens „*CommunicationSystem*“ auf dem PC an. Legen Sie außerdem einen Unterordner namens „*functions*“ an, in dem Sie dann alle Funktionen als MATLAB Dateien abspeichern können.

In Ihrem Hauptordner „*CommunicationSystem*“ erstellen Sie nun eine Datei namens „*SingleCarrierCommunication.m*“, in der Sie das Sende- und Empfangssystem implementieren können.

Schreiben Sie als erstes einen Header in dieser Datei, der Informationen über den Inhalt, Ihre Namen etc. enthält.

Es empfiehlt sich ganz am Anfang die Kommandozeile zu löschen, den Workspace (alle angelegten Variablen) zu löschen und alle geöffneten Fenster zu schließen.

Schließlich müssen Sie noch Ihren Unterordner einbinden um später auf Funktionen, die Sie darin speichern verwenden zu können.

Dann kann es losgehen und Sie definieren eine Nachricht, die Sie versenden wollen als Text-String.

Die ersten Zeilen Ihrer Hauptdatei können dann so aussehen:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Single Carrier Communication
% Name: …
% Datum: …
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc
clear
close all
addpath('functions\');

%% Definieren Sie eine Nachricht,
% die versendet werden soll:
msg = 'Hallo hier ist eine wichtige Nachricht!';

% Diese Nachricht stellt die Quelle für unsere Kommunikation dar.
```

Aufgabe 1 - Quellkodierung

Zunächst muss über die Quellkodierung der Text in einen Bitstrom übersetzt werden. Um es einfach zu halten, wird ein Code fester Länge verwendet. Groß- und Kleinbuchstaben, Ziffern sowie Satzzeichen und die meisten Sonderzeichen befinden sich im ASCII-Code im Bereich 0…127 – können also mit nur 7 Bit einfach abgebildet werden. Bei dieser 7 Bit Kodierung verzichten wir auf Umlaute und auf das ‚ß'. Damit bleibt ein Bit in jedem Byte übrig, das später zur Kanalkodierung genutzt werden kann.

Legen Sie in Ihrem Unterordner ‚*functions*' folgende Dateien an:

- ‚*sourceCoding.m*'
- ‚*sourceDecoding.m*'
- ‚*decToBinVec.m*'
- ‚*binVecToDec.m*'

In MATLAB kommen Sie sehr leicht von einem character z.B. ‚a' zum entsprechenden ASCII-Code als Dezimalwert. Eine Binärzahl gib MATLAB mit der Funktion ‚dec2bin' allerdings nur als String zurück. Wir benötigen aber einen Vektor mit ‚1' und ‚0' als Zahlen, um damit rechnen zu können. Daher die unteren beiden Hilfsfunktionen.

Die erste Hilfsfunktion könnte folgendermaßen aussehen:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Name: **** ****
%  Date: **.**.****
%
%  decToBinVec: converts decimal number to binary vector
%
%  Input:   - dec:  decimal number (in the range 0...2^N-1)
%           - N:    length of the output vector (number of bits)
%  Output:  - bin:  vector of bit values (0 or 1) MSB at index 1
%  Example: decToBinVec(12,7) yields: [0 0 0 1 1 0 0]
%
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function bin = decToBinVec(dec,N)
bin = zeros(1,N);    % prepare output vector with zeros
% for loop to calculate each bit
for i = 1:N
    % calculate each single bit
    bin(i) = bitand(dec,2^(N-i))>0;
end
```

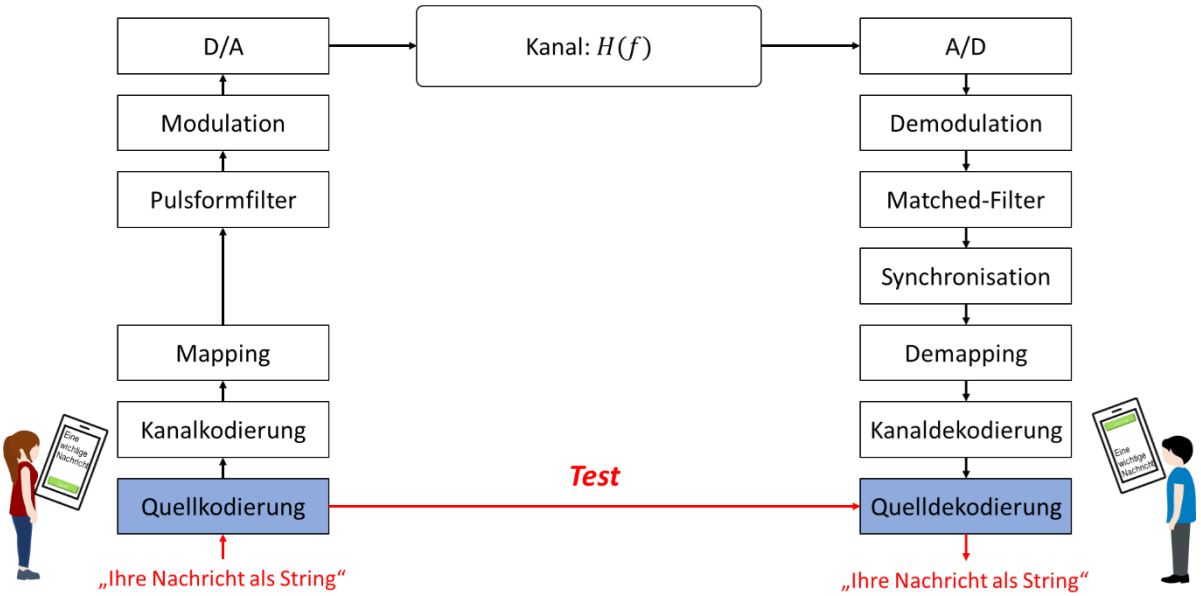
Beachten Sie wie Funktionen in MATLAB in Dateien definiert werden:

```
function [output1, output2] = functionName(input1, input2)
```

Dabei muss der Funktionsname exakt dem Dateinamen entsprechen.

Programmieren Sie so die Inhalte der vier Funktionen. Die Funktion „*sourceCoding*" bekommt den gesamten String der zu sendenden Nachricht und gibt die entsprechenden Bits als Vektor zurück. Diesen Vektor können Sie dann (wie in folgendem Bild dargestellt) zum Testen in die Funktion „*sourceDecoding*" eingeben und müssen Ihre Nachricht wieder als String zurück erhalten.

Diesen Test schreiben Sie in Ihre Hauptfunktion „*SingleCarrierCommunication.m*".


D/A
Kanal: $H(f)$
A/D
Modulation
Demodulation
Pulsformfilter
Matched-Filter
Synchronisation
Mapping
Demapping
Kanalkodierung
Kanaldekodierung
Quellkodierung
Test
Quelldekodierung
„Ihre Nachricht als String“
„Ihre Nachricht als String“

Aufgabe 2 - Kanalkodierung

Als Kanalkodierung soll eine einfache Parity-Matrix – wie aus der Vorlesung bekannt – verwendet werden.

Legen Sie dazu folgende Dateien und Funktionen in Ihrem Unterordner „*functions*“ an:

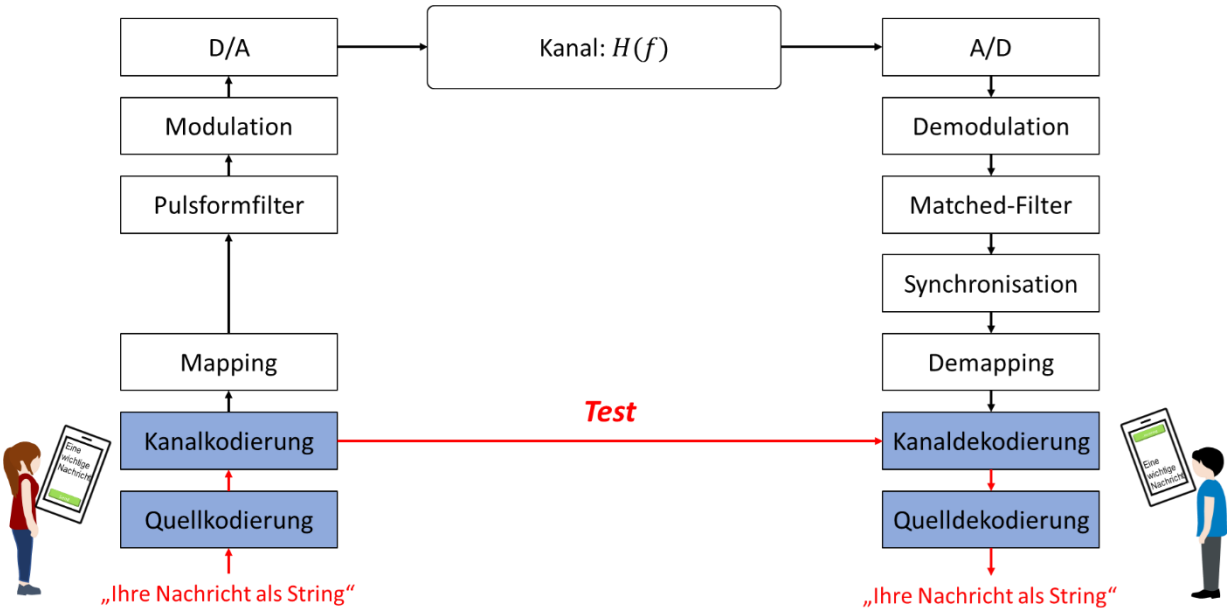
- „*channelCoding.m*“
- „*channelDecoding.m*“

In die Funktion *channelCoding* wird der Bit-Vektor mit der gesamten Quell-kodierten Nachricht eingegeben. Fassen Sie dann in der Funktion immer 7x7 Bit zu einer Matrix zusammen, sodass inklusive Parity-Bits eine 8x8 Bits große Matrix entsteht. 64 Bit lassen sich später gut auf Symbole mappen.

Wenn die Länge der eingegangenen Nachricht nicht einem Vielfachen von 7 entspricht, füllen Sie sie der Einfachheit halber schon nach der Definition des Texts in der Hauptdatei mit einem beliebigen Zeichen wie z.B. ‚*‘ in entsprechender Anzahl auf.

Geben Sie als Rückgabewert bei der Dekodierung neben den dekodierten Bits auch die Anzahl der detektierten Fehler aus und – wenn nur ein Fehler aufgetreten ist – ob ein Fehler korrigiert wurde.

Fügen Sie dem Test der Quellkodierung in Ihrer Hauptfunktion „*SingleCarrierCommunication.m*“ nun auch die Kanalkodierung hinzu – wie im Bild dargestellt:



Aufgabe 3 - Mapping

Definieren Sie zunächst ein Alphabet an Sendesymbolen. Im ersten Schritt soll das Alphabet rein reell sein – z.B.:

```
alphabet = [-3; -1; +1; +3;];
```

Legen Sie dazu folgende Dateien und Funktionen in Ihrem Unterordner „*functions*“ an:

- „*symbolMapping.m*“
- „*symbolDemapping.m*“

Als Eingang für die Funktion werden die Kanal-kodierten Bits eingegeben sowie das zuvor definierte Alphabet. Berechnen Sie aus der Mächtigkeit des Alphabets (Anzahl Symbole) zunächst wie viele Bits Sie pro Symbol übertragen werden können. Damit und mit der Anzahl der eingegangen Bits können Sie einen Output-Vektor der richtigen Länge anlegen.

Ebenfalls aus der Mächtigkeit des Alphabets kann direkt die Generatormatrix (und für das Demapping die inverse dazu) erstellt werden. Sie können entweder zu Beginn ein Codebuch (Lookup-Table) erstellen mit allen möglichen Bitkombinationen und den entsprechenden Symbolen oder Sie berechnen die Kodierung jedes Mal direkt mit den Matrizen.

Erweitern Sie das Mapping auch auf komplexe Symbole z.B. für eine 16 QAM. Dazu muss immer je eine Hälfte der Bits für den Realteil und die andere Hälfte für den Imaginärteil Gray-kodiert werden.

Ausgabewerte sind jeweils reelle oder komplexe Symbole d.h. ein Vektor, der nur Werte aus dem vorgegebenen Alphabet enthält.