KARLSRUHE INSTITUTE OF TECHNOLOGY
Communications Engineering Lab
Prof. Dr.rer.nat. Friedrich K. Jondral

Anwendung von Reed-Muller Codes bei OFDM-basierter Datenübertragung

Bachelorarbeit

Nico Otterbach

Hauptreferent	:	Prof. Dr.rer.nat. Friedrich Jondral
Betreuer	:	Dipl.-Ing. Martin Braun

Beginn	:	01.11.2010
Abgabe	:	29.04.2011

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Literaturquellen sind im Literaturverzeichnis vollständig zitiert.

Karlsruhe, den 29.04.2011

Nico Otterbach

Zusammenfassung

In dieser Arbeit werden überwiegend binäre Reed-Muller Codes behandelt, wobei insbesondere auf die Vorteile bei deren Verwendung für eine OFDM-basierte Datenübertragung eingegangen wird.

Hierzu wird zunächst das OFDM-Verfahren vorgestellt, bei dem ein serieller Datenstrom parallel auf mehreren Subträgern übertragen wird. Durch die Parallelisierung ergibt sich eine, im Vergleich zum korrespondierenden Einträgerverfahren, größere Symboldauer, wodurch der Einfluss von Inter-Symbol-Interferenzen abgeschwächt wird. Die Verwendung eines OFDM-basierten Verfahrens bringt somit bei Kanälen mit Mehrwegeausbreitung Vorteile mit sich. Der prinzipielle Aufbau eines OFDM-Systems ist in Abb. 0.1 dargestellt.

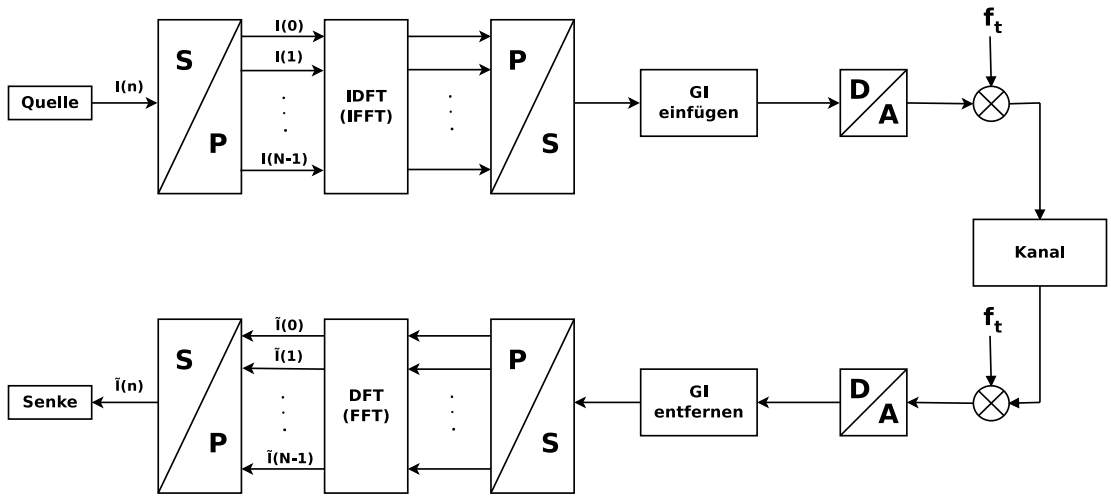

Abbildung 0.1.: Prinzipieller Aufbau einer OFDM-Übertragungsstrecke

Einen wesentlichen Nachteil bei der Verwendung des OFDM-Verfahrens stellt das Peak Power Problem dar. Es entsteht dadurch, dass durch die OFDM-Operation ein Sendesignal erzeugt wird, dessen komplexe Einhüllende nicht konstant ist. Hierbei treten, im Vergleich zur mittleren Leistung der komplexen Einhüllenden, Spitzenwerte für die momentane Leistung der komplexen Einhüllenden auf, welche den Einsatz teurer Verstärker mit großem Dynamikbereich erfordern, da sonst nichtlineare Verzerrungen des Sendesignals auftreten können.

In Abb. 0.2 ist das OFDM-Zeitsignal für die Einsfolge der Länge 8 dargestellt. Die Einsfolge stellt hierbei eine der vier binären Sequenzen dar, für die das PMEPR seinen maximalen Wert annimmt. Der maximale Wert des PMEPR, der in einem OFDM-basierten System ohne entsprechende Maßnahmen zur Spitzenwertreduktion erreicht wird, entspricht gerade der verwendeten Anzahl an Subträgern. Somit ist mit steigender Anzahl an Subträgern des verwendeten Systems der Einsatz von Verstärkern mit immer größerem Dynamikbereich erforderlich.

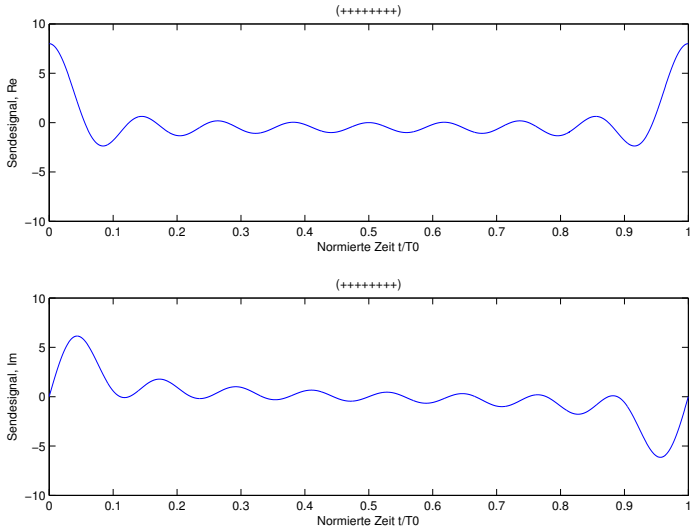

Abbildung 0.2.: OFDM-Zeitsignal der Einsfolge mit Länge 8

Wie gezeigt wird, kann auf die mittlere Leistung der komplexen Einhüllenden für eine feste Anzahl an Subträgern keinen Einfluss genommen werden. Dadurch folgt für die Lösung des Peak Power Problems, dass die auftretenden Spitzenwerte der zu übertragenden Sequenzen reduziert werden müssen. Für beliebige Sequenzen lässt sich hierbei eine Verbindung zwischen den im Zeitsignal auftretenden Spitzenwerten und der aperiodischen Autokorrelationsfunktion für Verschiebungen ungleich null herstellen.

In diesem Zusammenhang werden zur Lösung des Peak Power Problems binäre Golay–Sequenzen eingeführt, da ihre aperiodische AKF günstige Eigenschaften in Bezug auf die Spitzenwertreduktion besitzen. Es wird weiter gezeigt, dass sich bei ausschließlicher Verwendung von Golay-Sequenzen für die Übertragung das PMEPR (und damit auch das PAPR) auf 3 dB begrenzen lässt.

Ein Beispiel für das OFDM-Zeitsignal einer Golay-Sequenz ist in 0.3 dargestellt. Hierbei lässt sich im Vergleich zur Einsfolge in Abb. 0.2 ein deutlich geringeres PMEPR erkennen.

In [DJ97] wurde erstmals eine Verbindung zwischen Golay-Sequenzen und Reed-Muller Codes zweiter Ordnung aufgezeigt. Mit Hilfe dieses Zusammenhangs lässt sich eine Co-

dierung für OFDM-basierte Systeme realisieren, wodurch zusätzlich zur Lösung des Peak Power Problems noch eine Möglichkeit zur Fehlerkorrektur gegeben ist.

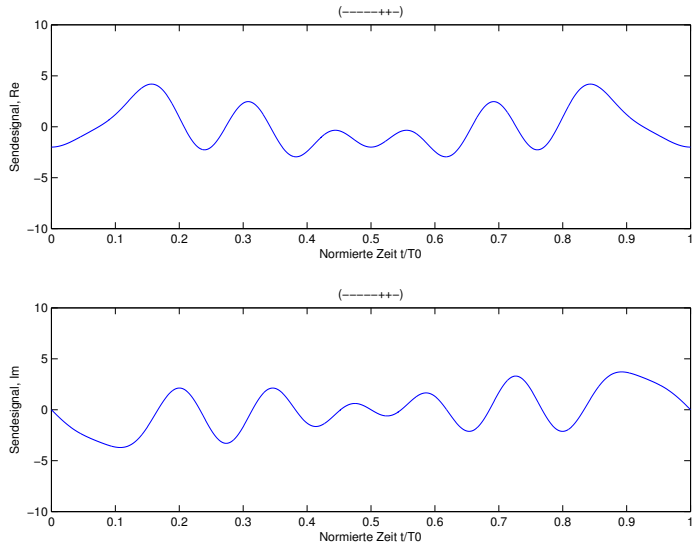

Abbildung 0.3.: OFDM-Zeitsignal der Golay-Sequenz (——++–)

In diesem Zusammenhang werden die, zur Klasse der linearen Blockcodes gehörenden, binären Reed-Muller Codes eingeführt. Hierbei werden zunächst die binären Booleschen Funktionen vorgestellt, aus denen die Generatormatrix eines binären Reed-Muller Codes aufgebaut ist.

Nachdem auf die Definition und Eigenschaften, beispielsweise die Korrekturfähigkeit, binärer Reed-Muller Codes eingegangen wurde, folgt eine Beschreibung des Algorithmus zur HD-Decodierung binärer Reed-Muller Codes nach Reed [Ree54].

Nachdem eine Einführung in die klassischen Reed-Muller Codes und einem Hinweis auf die Möglichkeit zu deren Verallgemeinerung erfolgt ist, lässt sich nun in 3.5 die Idee zur Spitzenwertreduktion mittels Golay-Sequenzen aus Kapitel 2 erneut aufgreifen.

Um mittels einer Reed-Muller Codierung ausschließlich Golay-Sequenzen erzeugen zu können, muss zunächst der $RM(2,m)$-Code in die Nebenklassen des $RM(1,m)$-Codes zerlegt werden. Nach [DJ97] beinhalten nun genau $\frac{m!}{2}$ dieser Nebenklassen ausschließlich Golay-Sequenzen.

Damit folgt, dass $\frac{m!}{2}$ Möglichkeiten einer $RM(1,m)$-Codierung, welche durch Addition eines Nebenklassenanführers einer bestimmten Nebenklasse zugeordnet wird, existieren, bei der ausschließlich Golay-Sequenzen als Codewörter verwendet werden (Im folgenden als RMG-Codierung bezeichnet). Wie bereits erwähnt, eignet sich diese Form der Codierung besonders für OFDM-basierte Systeme, da mit ihrer Hilfe das Peak Power Problem gelöst werden kann.

Ein Nachteil stellt hierbei die geringe erzielbare Coderate dar, welche für unterschiedliche Anzahlen verwendeter Golay-Nebenklassen in Abb. 0.4 gegeben ist. Hierbei stellt die rote Kurve eine obere Schranke für die, mittels einer RMG-Codierung erreichbare, Coderate dar. Die blaue Kurve hingegen, bei der nur eine der $\frac{m!}{2}$ Golay–Nebenklassen zum Einsatz kam, kann als untere Schranke für die Coderate bei Verwendung einer RMG–Codierung interpretiert werden.

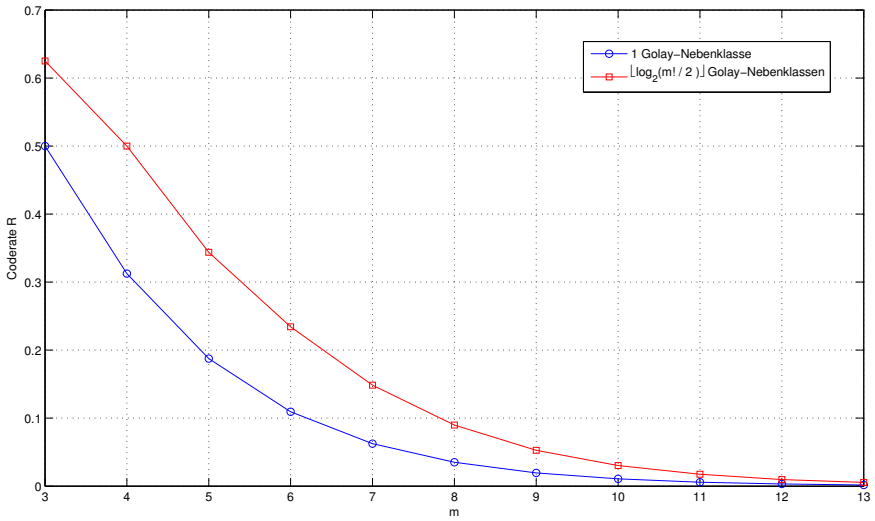

Abbildung 0.4.: Coderate für unterschiedliche Anzahl von Golay-Nebenklassen

Schließlich wird auch für die RMG-Codierung eine Möglichkeit zur Encodierung vorgestellt. Hierbei wird eine bestimmte Anzahl an Informationsbits zur Auswahl einer der Golay-Nebenklassen verwendet.

Zur Decodierung bei Verwendung der RMG-Codierung wird außerdem noch auf einen, auf der Fast-Hadamard-Transformation basierenden, Algorithmus eingegangen, welcher sowohl zur Hard- als auch zur Soft-Decision-Decodierung verwendet werden kann.

Die vorgestellten Algorithmen zur RM- und RMG-Codierung wurden im Rahmen dieser Arbeit in GNU Radio implementiert. Hierbei wurde ein effizientes Speicherkonzept verwendet, welches durch die bitweise Speicherung aller auftretenden Vektoren und Matrizen einen geringen Overhead erzeugt und effiziente Berechnungen ermöglicht. In Kapitel 4 werden neben dem Speicherkonzept weitere, bei der Implementierung verwendete, Methoden näher betrachtet.

Abb. 0.5 zeigt ein Screenshot des GNU Radio Companion, wobei sich die implementierten Blöcke rechts im Ordner „RM-Coding“ innerhalb der „Channel Coding“-Toolbox befinden. Das geöffnete Fenster im Zentrum von 0.5 zeigt die Parameter des HD-Decoders bei Verwendung einer RMG-Codierung.

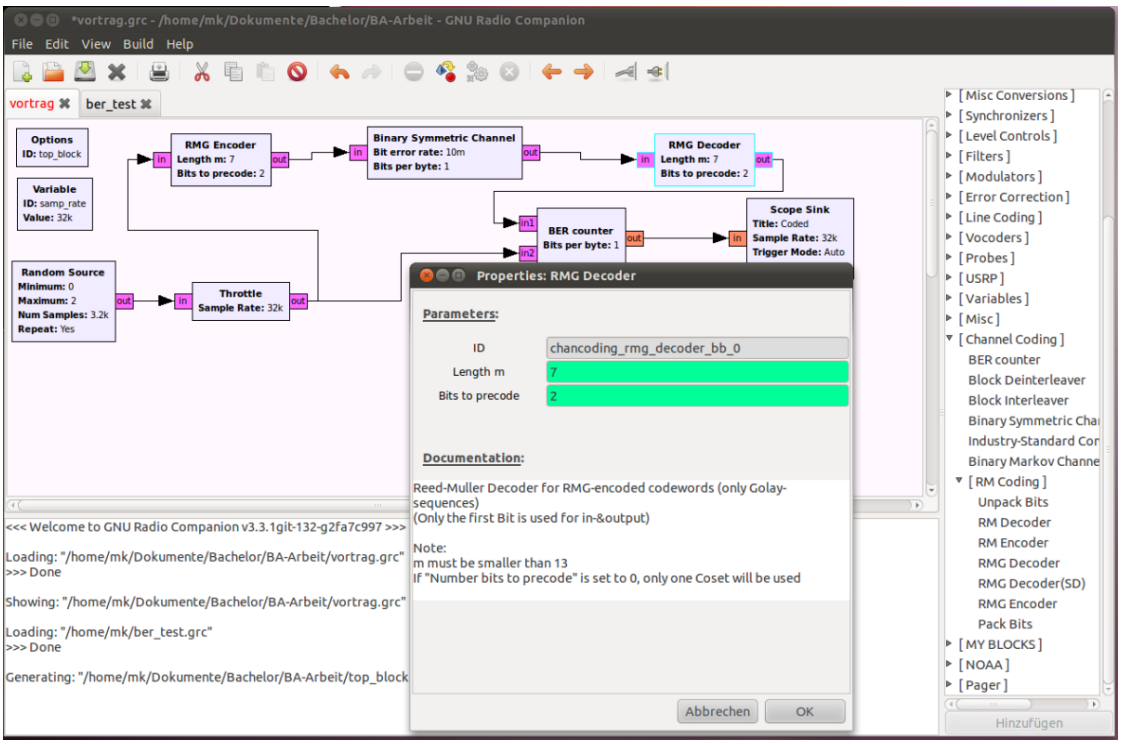

Abbildung 0.5.: Screenshot des GNU Radio Companion mit implementierter RM-Codierung

In Kapitel 5 erfolgen abschließend Betrachtungen zu der, mittels der Implementierung erreichbaren, Performance. In diesem Zusammenhang wird gezeigt, dass die Decodierung die kritische Komponente im Hinblick auf die erreichbare Geschwindigkeit bzw. Datenrate darstellt, sowie Möglichkeiten zur Performance-Steigerung aufgezeigt.

Inhaltsverzeichnis

1. Einleitung

In den letzten beiden Jahrzehnten schritt die Entwicklung digitaler Funk- und Mobilkommunikationssysteme rasant voran, wodurch hoch-entwickelte Systeme heute einer breiten Masse für relativ geringe Kosten zur Verfügung stehen.

Um der Nachfrage immer höherer Datenraten gerecht werden zu können, erhielten viele Methoden und Techniken Einzug in praktische Systeme, welche zwar schon seit langem bekannt waren, aber aus verschiedenen Gründen bis zu diesem Zeitpunkt nicht angewandt wurden.

Eine dieser Techniken, die heute in vielen Systemen zur Anwendung kommt, stellt das Orthogonal Frequency-Division Multiplexing, kurz OFDM, dar. Das OFDM-Verfahren bringt vor allem bei Kanälen mit Mehrwegeausbreitung Vorteile mit sich, da hierbei weniger Interferenzen als beim korrespondierenden Einträgerverfahren auftreten. Für den Einsatz des OFDM-Verfahrens in Kanälen mit Mehrwegeausbreitung spricht zusätzlich die Tatsache, dass sich die Parameter eines OFDM-Systems für gegebene Randbedingungen sehr flexibel gestalten lassen.

Der entscheidende Nachteil bei der Verwendung eines OFDM-basierten Systems stellt das Peak Power Problem dar. Durch Anwenden der OFDM-Operation entsteht ein Sendesignal dessen komplexe Einhüllende nicht konstant ist. Dadurch können für die momentane Leistung der komplexen Einhüllenden des Sendesignals mitunter sehr hohe Spitzenwerte im Vergleich zur mittleren Leistung der komplexen Einhüllenden auftreten.

Treten im Vergleich zur mittleren Leistung der komplexen Einhüllenden hohe Spitzenwerte der momentanen Leistung der komplexen Einhüllenden auf, so werden teure Verstärker mit großem Dynamikbereich benötigt um nichtlineare Verzerrungen zu vermeiden. Da der Einsatz eines Verstärkers mit großem Dynamikbereich aus Kostengründen häufig keine Option für die Realisierung darstellt, gilt es Methoden zu finden, mit denen sich das Peak Power Problem lösen lässt. Hierzu lassen sich in der Literatur vielfältige Ansätze finden.

In dieser Arbeit wird der Ansatz verfolgt, das Peak Power Problem mittels geeigneter Codierung zu lösen. Hierzu sei an dieser Stelle insbesondere die Arbeit von James A. Davis und Jonathan Jedwab [DJ97] erwähnt, in der erstmals eine Verbindung zwischen Golay-Sequenzen und Reed-Muller Codes zweiter Ordnung hergestellt wurde.

Mit Hilfe der in [DJ97] dargestellten Zusammenhänge lässt sich eine Codierung für OFDM-Systeme realisieren, bei der ausschließlich Golay-Sequenzen als Codewörter auftreten.

Golay-Sequenzen bieten im Zusammenhang mit OFDM-basierten Systemen den Vorteil reduzierter Spitzenwerte im Vergleich zu beliebigen Sequenzen. Außerdem ist bei der vorgestellten Methode zur Codierung zusätzlich zur Spitzenwertreduktion noch eine Möglichkeit zur Fehlerkorrektur gegeben.

Um mit Hilfe des Reed-Muller Codes zweiter Ordnung ausschließlich Golay-Sequenzen als Codewörter zu erzeugen, bedarf es einer Nebenklassenzerlegung des selbigen in die Nebenklassen des Reed-Muller Codes erster Ordnung. Einige der so entstandenen Nebenklassen enthalten ausschließlich Golay-Sequenzen und können somit zur Codierung verwendet werden.

Im Folgenden wird in Kapitel 2 zunächst auf das OFDM-Verfahren sowie die Spitzenwertreduktion mittels Golay-Sequenzen eingegangen. Anschließend folgt in Kapitel 3 eine Einführung in das Themengebiet der Reed-Muller Codes. Hierbei wird insbesondere auch der Zusammenhang zwischen den Reed-Muller Codes zweiter Ordnung und Golay-Sequenzen erläutert. Kapitel 4 behandelt die Implementierung der Reed-Muller Codes in GNU Radio, bevor in Kapitel 5 schließlich die erreichbare Performance der selbigen untersucht wird.

2. OFDM - Ein Mehrträgerverfahren

Bei OFDM (*Orthogonal Frequency-Division Multiplexing*) handelt es sich um ein Mehrträgerverfahren mit orthogonalen Subträgern, welches bereits 1957 im „Collins Kineplex System“ Anwendung fand [HDM57]. Allerdings brachten erst weitere Schritte der Entwicklung die notwendigen Vereinfachungen um das System effizient einsetzen zu können. Das Mischen auf die einzelnen Trägerfrequenzen mittels der *diskreten Fourier Transformation* (DFT), vorgestellt in [WE71], erwies sich dabei als sehr effizient im Vergleich zur rein analogen Methode mittels Filter- und Oszillatorbänken. Die Einführung der *schnellen Fourier Transformation* (FFT) nach [CT65] in Kombination mit den erweiterten Möglichkeiten der digitalen Signalverarbeitung führten letztendlich dazu, dass das Verfahren für praktische Systeme attraktiv wurde.

Heute kommt OFDM in den verschiedensten Systemen zum Einsatz. So wird es in kabelgebundenen Systemen, wie zum Beispiel ADSL (*Asymmetric Digital Subscriber Line*), ebenso verwendet wie in Funknetzen, zum Beispiel WLAN (*Wireless Local Area Network*). Auch in digitalen Rundfunksystemen wie DAB (*Digital Audio Broadcasting*) oder DVB-T (*Digital Video Broadcasting - Terrestrial*) kommt OFDM zum Einsatz.

Aktuell wird eine Implementierung des OFDM-Verfahrens für die vierte Generation Mobilfunk (*4G*) in Betracht gezogen, so soll das Verfahren beispielsweise bei LTE (*3GPP Long Term Evolution*) Anwendung finden.

2.1. Funktionsweise

Der Grundgedanke bei der Mehrträgerübertragung besteht darin, dass ein serieller Symbolstrom in mehrere parallel zu übertragende Symbolströme aufgeteilt wird (siehe Abb. 2.1). Die einzelnen Ströme werden hierbei derart auf einzelne Träger moduliert, dass möglichst keine *Inter-Channel-Interference* (ICI) zwischen den einzelnen Subträgern entsteht. Bei OFDM wird dies implizit durch Erfüllen der Orthogonalitätsbedingung gelöst.

Man definiere zunächst die *N Basisfunktionen* $\Psi_n(t)$ als die, mit einem *Rechteckfenster der Länge* T_0 gewichteten, komplexen Trägerschwingungen der *Frequenz* f_n:

$$\Psi_n(t) = e^{j2\pi f_n t} \cdot \frac{1}{\sqrt{T_0}} \cdot \mathrm{rect}\left(\frac{t}{T_0}\right), \qquad n = 0, 1, ..., N-1 \tag{2.1}$$

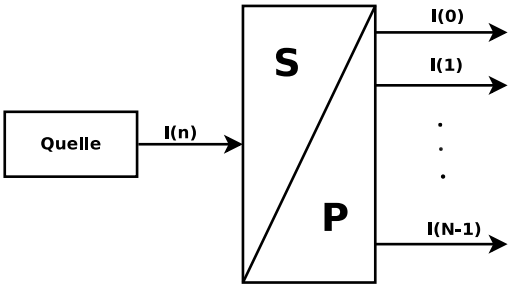

Abbildung 2.1.: Seriell-/Parallelwandlung der Symbole

Die *OFDM-Symboldauer* T_0 bezeichnet hierbei die Dauer eines *OFDM-Nutzsymbols*, welches sich aus der Überlagerung der parallel vorliegenden Symbole $I(0), ..., I(N-1)$ ergibt.

Die Orthogonalitätsbedingung ergibt sich nun zu

$$\int_{-\infty}^{\infty} \Psi_n(t) \cdot \Psi_m^*(t)dt = \int_0^{T_0} \Psi_n(t) \cdot \Psi_m^*(t)dt = \begin{cases} 1 : n = m \\ 0 : n \neq m \end{cases} . \tag{2.2}$$

Diese ist genau dann erfüllt, wenn die Subträgerfrequenzen im Abstand von

$$\triangle f = \frac{1}{T_0} \tag{2.3}$$

liegen. Damit folgt für die einzelnen Subträgerfrequenzen:

$$f_n = f_{min} + n \cdot \triangle f, \qquad n = 0, 1, ..., N-1. \tag{2.4}$$

Insgesamt lässt sich ein OFDM-Nutzsymbol im Zeitbereich durch

$$u(t) = \sum_{n=0}^{N-1} I(n) \cdot \Psi_n(t) = \sum_{n=0}^{N-1} I(n) \cdot e^{j2\pi f_n t} \cdot \frac{1}{\sqrt{T_0}} \cdot \text{rect}\left(\frac{t}{T_0}\right) \tag{2.5}$$

beschreiben. Da die Informationssymbole innerhalb einer Symbolperiode als konstant angenommen werden können, wird hier o.B.d.A. nur ein OFDM-Nutzsymbol betrachtet. Will man zusätzlich die Abfolge der OFDM-Nutzsymbole miteinbeziehen, so müssen die einzelnen Informationssymbole $I(n)$ mit einem Index versehen werden, welcher sich auf das aktuelle OFDM-Nutzsymbol bezieht.

In OFDM-basierten Systemen werden keine Schutzbänder benötigt, da aufgrund der Orthogonalität der Subträger keine ICI auftritt. Dies zeigt sich anschaulich im Frequenzbereich, wozu man die *Fourier-Transformierte* $U(f)$ des Zeitsignals betrachte:

$$F\{u(t)\} = U(f) = \sum_{n=0}^{N-1} I(n) \cdot \sqrt{T_0} \cdot si(\pi(f - f_n)T_0) \tag{2.6}$$

Die Einzelspektren für $N = 8$ Subträger sind in Abb. 2.2 dargestellt[1]. Am absoluten Maximum eines Subträgers folgt für alle verbleibenden Subträger eine Nullstelle. Sie stören sich also nicht gegenseitig.

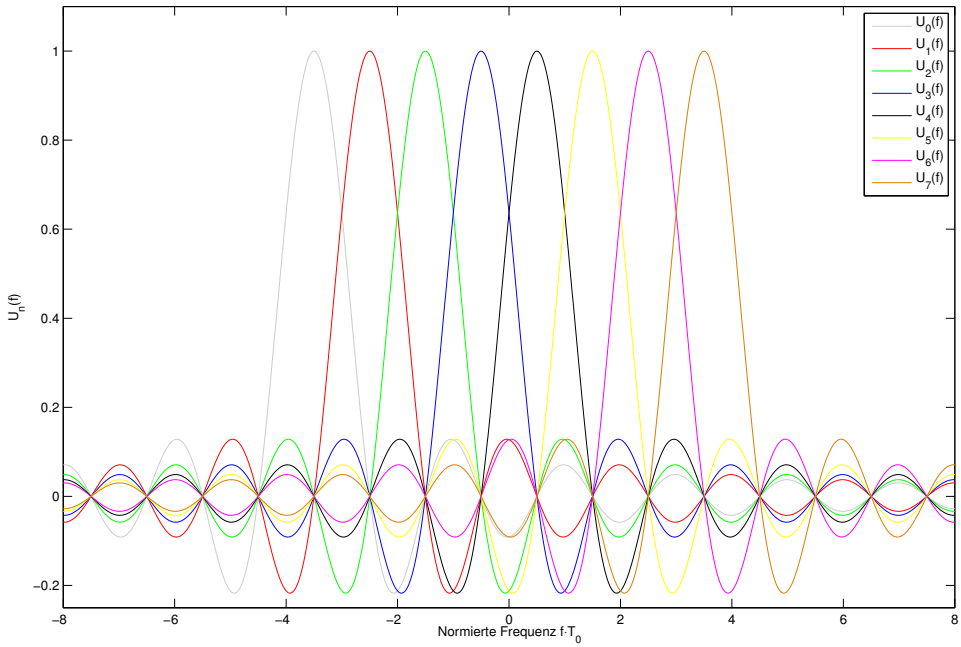

Abbildung 2.2.: Einzelspektren für $N = 8$ Subträger

Betrachtet man nun sowohl die Bandbreite als auch die Symboldauer des Mehrträgerverfahrens im Vergleich zum Einträgerverfahren, so ergeben sich durch die Seriell-/Parallelwandlung die Symboldauer T_0 und Bandbreite B_t eines OFDM-Nutzsymbols zu

$$T_0 = N \cdot T_S, \tag{2.7}$$

$$B_t = \frac{B_S}{N}, \tag{2.8}$$

wobei B_S der Bandbreite eines Informationssymbols bei Übertragung mit dem entsprechenden Einträgerverfahren und T_S der Symboldauer vor der Seriell-/Parallelwandlung entspricht.

Anhand von (2.7) lässt sich erkennen, dass ein OFDM-Nutzsymbol im Vergleich zu einem Symbol, welches mit dem entsprechenden Einträgerverfahren übertragen wird, eine um den Faktor N höhere Symboldauer besitzt. Dies bringt insbesondere bei Kanälen, in denen *Inter-Symbol-Interferenzen* (ISI) auftreten, Vorteile mit sich. ISI tritt vor allem bei Kanälen

[1] Hierbei wurde eine um $+\frac{1}{2}$ verschobene Rechteckfensterung zur vereinfachten Darstellung des Spektrums gewählt.

mit Mehrwegeausbreitung auf, da sich hierbei Echos vergangener Symbole mit dem aktuell zu empfangenden Symbol additiv überlagern und somit Störungen in diesem Symbol verursachen. Die *Kanalimpulsantwort eines Kanals mit Mehrwegeausbreitung* auf N_P Pfaden ergibt sich zu

$$h(\tau, t) = \sum_{n=0}^{N_P-1} a_n(t) \cdot \delta(\tau - \tau_n) \qquad , \tag{2.9}$$

wobei $a_n(t)$ für die zeitvarianten, komplexen Gewichte und τ_n für die Verzögerung der einzelnen Pfade stehen. Abb. 2.3 verdeutlicht den Zusammenhang zwischen den unterschiedlichen Symboldauern und der Kanalimpulsantwort bei vorhandener Mehrwegeausbreitung.

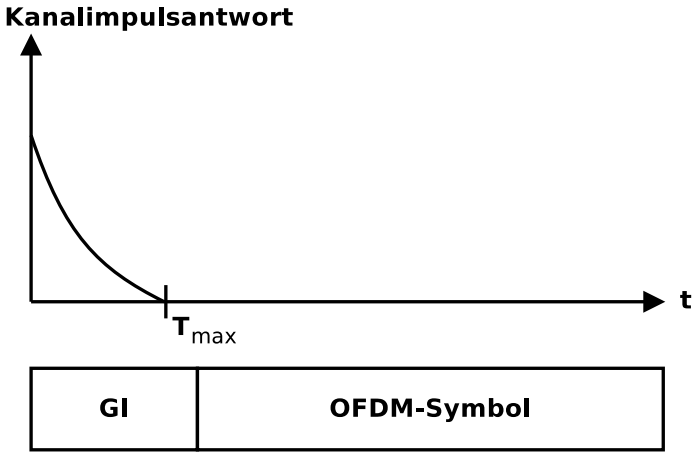

Abbildung 2.3.: Kanalimpulsantwort und OFDM-Symboldauer nach [Kam08, S.583]

Aufgrund der, im Vergleich zur maximalen Verzögerungszeit des Kanals, längeren Symboldauer bei OFDM-basierten Übertragungsverfahren ist der Einfluss der ISI geringer als beim korrespondierenden Einträgerverfahren. Dennoch tritt auch bei OFDM-basierten Systemen ISI auf, weshalb ein Schutz-Intervall (*Guard Interval*, GI) eingefügt wird. Das Schutzintervall wird so groß gewählt, dass es mindestens der maximalen Verzögerungszeit des Kanals entspricht, siehe Abb. 2.3. Es ergibt sich damit die Gesamtsymboldauer eines OFDM-Symbols zu

$$T_{Ges} = T_0 + T_{GI}. \tag{2.10}$$

Im Vergleich zum korrespondierenden Einträgerverfahren bietet OFDM also den Vorteil einer reduzierten ISI. Die Bandbreiteneffizienz des Einträgerverfahrens fällt bei OFDM aufgrund des eingefügten Schutzintervalls zwar etwas geringer aus, da jedoch aufgrund der Orthogonalitätsbedingung keine Schutzbänder zwischen den Subträgern benötigt werden, ist OFDM dennoch sehr effizient im Spektralbereich (siehe Abb. 2.4). Betrachtet man

ein Einträgerverfahren, so ergibt sich bei gegebener Bandbreite und Modulationsart (hier *binary phase shift keying*, BPSK) die Symboldauer zu

$$T_S = \frac{1}{R_S} \quad \text{mit} \quad R_S = B_S \cdot 1bit\,, \tag{2.11}$$

wobei R_S die Datenrate und B_S die Bandbreite für die Übertragung beschreiben. Aufgrund der antiproportionalen Beziehung zwischen der Datenrate R_S und der Symboldauer T_S macht sich der Vorteil der reduzierten ISI vor allem bei hohen Datenraten bemerkbar.

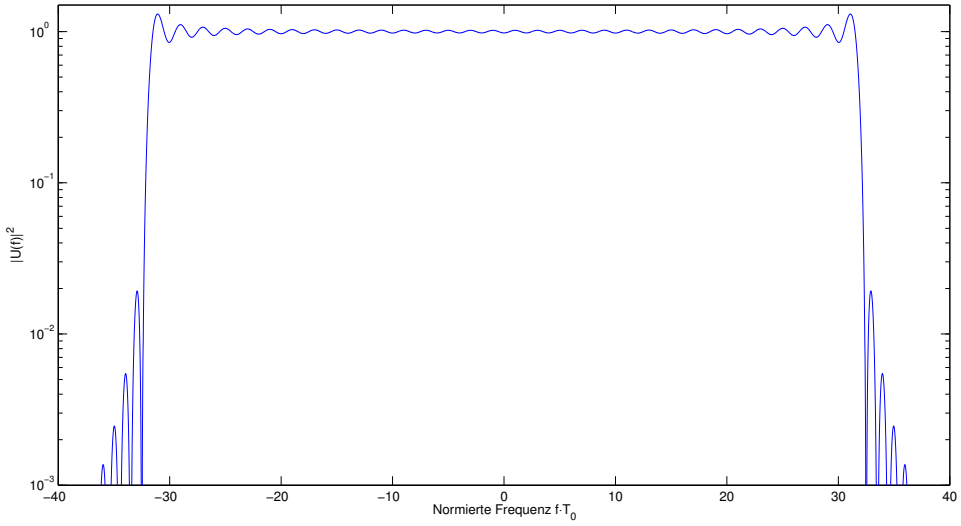

Abbildung 2.4.: Summenspektrum eines OFDM-Nutzsymbols für $N = 64$

Da der Einfluss der ISI mit kürzer werdender Symboldauer, aufgrund größerer Überlappung der einzelnen Symbole zunimmt, ist der Einsatz des OFDM-Verfahrens bei steigender Datenrate, und damit kürzer werdender Symboldauer, immer rentabler.

2.1.1. Prinzipieller Systemaufbau

Ein OFDM-Nutzsymbol entsteht indem die parallel vorliegenden Informationssymbole auf die einzelnen Trägerfrequenzen moduliert werden. Die analoge Umsetzung dieser Modulation mittels Filter- und Oszillatorbänken lässt sich, vor allem bei einer hohen Anzahl an Subträgern, nur durch einen sehr hohen Hardware-Aufwand realisieren.

Da die Informationssymbole auf viele Subträger im Frequenzbereich gemischt werden müssen liegt die Verwendung der DFT für die eigentliche OFDM-Operation nahe. Die Verlagerung der Modulation in den digitalen Bereich bringt außerdem eine erhebliche

Reduktion des Hardware-Aufwands mit sich, weshalb diese Art der Realisierung in den meisten praktischen Systemen Anwendung findet.

Vergleicht man die Definition der *inversen diskreten Fourier Transformation* (IDFT)

$$IDFT\{X(p)\} = x(q) = \frac{1}{N} \sum_{p=0}^{N-1} X(p) \cdot e^{j\frac{2\pi}{N} \cdot p \cdot q}, \qquad q = 0, 1, ..., N-1 \tag{2.12}$$

mit der eines abgetasteten[2] OFDM-Nutzsymbols im Zeitbereich

$$u\left(t = q \cdot \frac{T_0}{N_t}\right) = u(q) = \sum_{p=0}^{N_t-1} I(p) \cdot e^{j\frac{2\pi}{N_t} \cdot p \cdot q}, \qquad q = 0, 1, ..., N_t - 1, \tag{2.13}$$

so erkennt man, dass sich beide lediglich durch den Vorfaktor $\frac{1}{N}$ unterscheiden.

Das diskrete OFDM-Zeitsignal lässt sich also, abgesehen von der Multiplikation mit einem Skalar, direkt aus der IDFT der Informationssymbole gewinnen. Eine anschließende Digital-/Analogwandlung und Tiefpass-Filterung liefert schließlich das kontinuierliche Zeitsignal.

In praktischen Systemen wird aufgrund der schnelleren und effizienteren Berechnung der FFT- bzw. *(inverse schnelle Fourier Transformation)* IFFT-Algorithmus nach [CT65] verwendet. Auf diese Weise lässt sich die gesamte OFDM-Operation im Vergleich zur analogen Methode sehr kostengünstig, platzsparend und effizient realisieren.

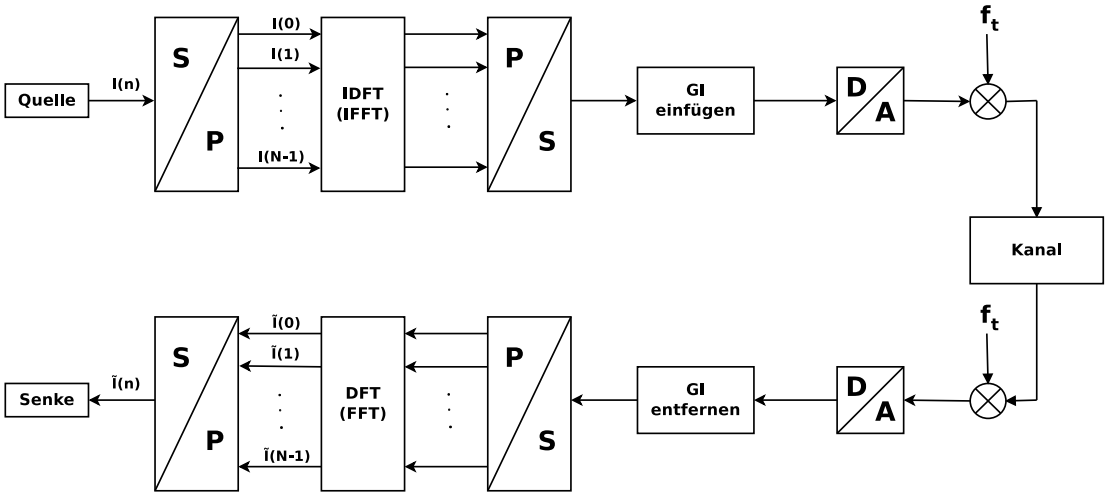

Abbildung 2.5.: Prinzipieller Aufbau einer OFDM-Übertragungsstrecke

Abbildung 2.5 zeigt den prinzipiellen Aufbau einer OFDM-Übertragungsstrecke. Nach der Seriell-/Parallelwandlung und der IDFT/IFFT folgt eine Parallel-/Seriellwandlung,

[2]Hier wurde eine Abtastung mit der Nyquistfrequenz f_N angenommen.

an deren Ende man die zeitdiskreten Abtastpunkte des kontinuierlichen Zeitsignals erhält. Schließlich wird noch das Schutzintervall vor der Digital-/Analogwandlung eingefügt, bevor das entstandene kontinuierliche Zeitsignal auf die Träger-/Mittenfrequenz heraufgemischt wird. Die Empfangsseite ist reziprok zur Sendeseite aufgebaut. Um die Informationen im Empfänger schätzen zu können muss hier jedoch das abgetastete Zeitsignal nach Entfernen des Schutzintervalls mittels der DFT/FFT erst wieder in den Frequenzbereich zurücktransformiert werden.

2.2. Das Peak Power Problem

Ein großer Nachteil bei OFDM-basierten Übertragungssystemen folgt aus der Tatsache, dass das Sendesignal im Zeitbereich keine konstante komplexe Einhüllende besitzt. Da das Sendesignal aus unterschiedlichen additiven Überlagerungen komplexer Schwingungen[3] entsteht, kann es zu sehr hohen Spitzenwerten im Vergleich zur durchschnittlichen Leistung des Sendesignals kommen. Mathematisch lässt sich dieser Sachverhalt für Bandpasssignale durch das *Peak-to-Average Power Ratio* (PAPR) beschreiben, welches bei OFDM-basierten Systemen sehr hohe Werte annehmen kann.

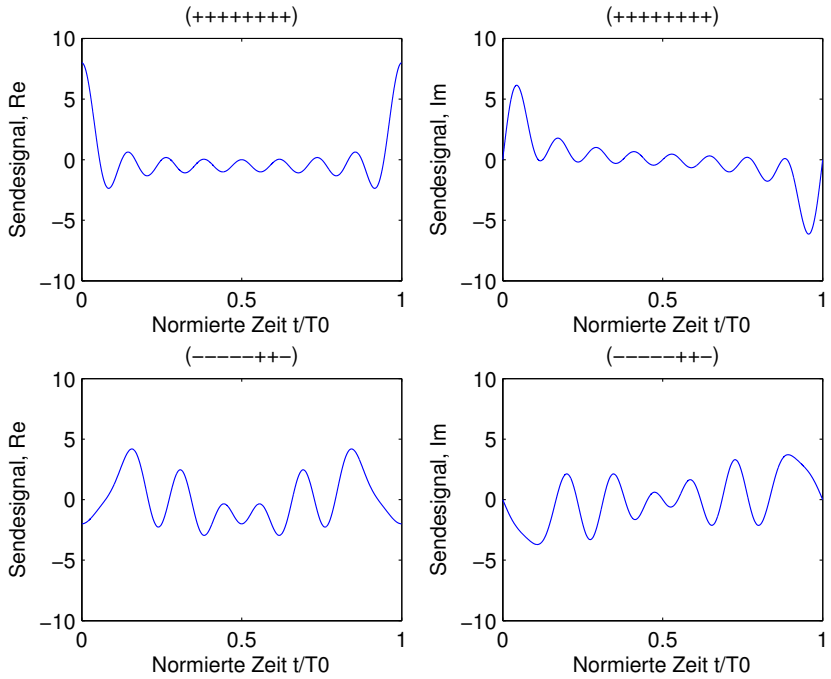

Abbildung 2.6.: OFDM-Zeitsignale für $N = 8$

[3]Die einzelnen komplexen Schwingungen werden durch die OFDM-Operation beim BPSK-Verfahren beispielsweise mit $+1/-1$ gewichtet und aufsummiert.

Abbildung 2.6 zeigt exemplarisch die Zeitsignale zweier unterschiedlicher OFDM-Symbole für $N = 8$. Es lässt sich leicht erkennen, dass das Signal im oberen Teil der Abbildung einen höheren Spitzenwert im Realteil als das Signal im unteren Abschnitt besitzt.

In praktischen Systemen erfordern hohe PAPR-Werte teure Verstärker, da ein großer Dynamikbereich möglichst linear verstärkt werden soll. Kommt es zu Nichtlinearitäten bei der Verstärkung, so treten weitere negative Effekte zu Tage. Unterliegt das Sendesignal nichtlinearen Verzerrungen, so treten beispielsweise verstärkt Außerbandstrahlungen *(out-of-band emissions)* auf, bei denen im Spektralbereich auch Leistungsanteile außerhalb des gewünschten Bandes vorhanden sind. Aufgrund gesetzlicher Bestimmungen und zur Vermeidung von Interferenzen in benachbarten Systemen gilt es jedoch die Leistung außerhalb des eigenen Bandes so gering wie möglich zu halten.

2.2.1. PAPR und PMEPR

In der Literatur finden sich je nach Autor und behandeltem Thema oft unterschiedliche Definitionen für das PAPR. In dieser Arbeit bezieht sich das PAPR im Zusammenhang mit OFDM-basierten Systemen auf das reelle Bandpasssignal. Diese Art der Beschreibung ist deshalb sinnvoll, weil es letztendlich die Bandpasssignale sind, die dem Verstärker zugeführt werden und somit bei der Systemauslegung eine entscheidende Rolle spielen. Das PAPR gibt das Verhältnis der Spitzenleistung zur Durchschnittsleistung des *reellen Sendesignals* $s(t) = Re\{u(t) \cdot e^{j2\pi f_T t}\}$ nach dem Mischen auf die *Trägerfrequenz* f_T wieder und ist nach [RL06] wie folgt definiert:

$$\text{PAPR} = \frac{\max\{|s(t)|^2\}}{E\{|s(t)|^2\}} = \frac{\max\{|Re\{u(t) \cdot e^{j2\pi f_T t}\}|^2\}}{E\{|Re\{u(t) \cdot e^{j2\pi f_T t}\}|^2\}}. \tag{2.14}$$

Zur analytischen Betrachtung hinsichtlich der Spitzenwertreduktion ist das *Peak-to-Mean Envelope Power Ratio* (PMEPR) jedoch geeigneter. Es bezieht sich auf das Signal im Basisbandbereich und damit auf die *komplexe Einhüllende* $u(t)$ des Signals. Das PMEPR ist aufgrund der Beschränkung auf den Basisbandbereich und nicht erforderlicher Realteilbildung mathematisch einfacher zu handhaben und Zusammenhänge lassen sich leichter herleiten. Nach [RL06] gilt für das PMEPR:

$$\text{PMEPR} = \frac{\max\{|u(t)|^2\}}{E\{|u(t)|^2\}}. \tag{2.15}$$

Insgesamt lässt sich zwischen dem PAPR und dem PMEPR im Falle einer OFDM-basierten Übertragung mit Phasenmodulation der einzelnen Informationssymbole der Zusammenhang

$$\text{PAPR} \leq \text{PMEPR} \tag{2.16}$$

finden [Lit07, S. 7]. Damit ist gezeigt, dass eine Beschränkung des PMEPR eine Beschränkung des PAPR impliziert. Aufgrund der einfacheren mathematischen Verhältnisse wird im weiteren Verlauf dieser Arbeit nur noch das PMEPR betrachtet.

2.2.2. Möglichkeiten zur Reduktion des PMEPR

Die einfachste Methode zur Spitzenwertreduktion[4] in einem OFDM-System stellt das *clipping* dar; hierbei wird das Signal ab einem bestimmten Wert abrupt abgeschnitten. Clipping kann beispielsweise innerhalb eines Verstärkers auftreten, wenn die Amplitude des Eingangssignal so groß ist, dass der Verstärker bis in den Sättigungsbereich ausgesteuert wird und die Amplitude am Ausgang nicht weiter ansteigt. Das abrupte Abschneiden der Signalanteile höherer Amplituden stellt eine nichtlineare Verzerrung des Eingangssignals dar und bringt daher die bereits erwähnten negativen Effekte mit sich.

Es existieren neben dem Clipping noch weitere Verfahren, die das zu sendende Signal erst nach der OFDM-Operation manipulieren. Diese Verfahren können der störungsbehafteten Spitzenwertreduktion zugeordnet werden, da sie entweder zu Verzerrungen oder zur Leistungsreduktion des Sendesignals führen. Auf sie wird daher im weiteren Verlauf dieser Arbeit nicht näher eingegangen.

Der störungsbehafteten Spitzenwertreduktion steht die störungsfreie Spitzenwertreduktion gegenüber, von der insbesondere das Verfahren der Spitzenwertreduktion mittels geeigneter Codierung vor der OFDM-Operation betrachtet werden soll. Für weitere Verfahren zur Spitzenwertreduktion sei beispielsweise auf [HL05] verwiesen.

Im Folgenden wird das Signal o.B.d.A. nur über die Dauer einer Symbolperiode, d.h. $0 \leq t \leq T_0$, betrachtet. Dies hat zur Folge, dass die Informationssymbole als konstant angenommen werden können. Für weitere Betrachtungen werden die Informationssymbole außerdem durch ihre komplex modulierten Werte dargestellt:

$$I(n) = \xi^{a_n} \qquad \text{für } n = 0, 1, \ldots, N-1. \tag{2.17}$$

Wählt man $\xi = e^{j\pi}$,so stellt (2.17) eine BPSK-Modulation für die binäre Sequenz $a = \{a_0, a_1, \ldots, a_{N-1}\}$ mit $a_n \in \{0, 1\}$ dar. Durch Einsetzen von (2.17) in (2.5) ergibt sich die komplexe Einhüllende eines OFDM-Signals zu

$$u(t) = \sum_{n=0}^{N-1} \xi^{a_n} \cdot e^{j2\pi f_n t} = \sum_{n=0}^{N-1} \xi^{a_n + 2 f_n t} \qquad \text{für } 0 \leq t \leq T_0. \tag{2.18}$$

Nach Einsetzen von (2.4) in (2.18) und unter Berücksichtigung von $P(t) = |u(t)|^2$ folgt nach [DJ99] für die *momentane Leistung der komplexen Einhüllenden* eines OFDM-Signals:

$$P(t) = \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \xi^{a_n(t) - a_m(t) + 2(n-m)\Delta f t}. \tag{2.19}$$

[4] Wie in diesem Abschnitt noch gezeigt wird, lässt sich eine Reduktion des PMEPR letztendlich auf die Reduktion des Spitzenwertes zurückführen. Im weiteren Verlauf ist daher die Spitzenwertreduktion mit der Reduktion des PMEPR gleichzusetzen.

Da hier nur eine Symbolperiode betrachtet werden soll, können die Werte $a_n(t)$ und $a_m(t)$ als zeitunabhängig betrachtet werden. Aufgrund dieser Annahme und nach der Substitution[5] von $m = n + u$ ergibt sich die momentane Leistung der komplexen Einhüllenden einer Sequenz a für $0 \leq t \leq T_0$ zu:

$$\begin{aligned} P_a(t) &= \sum_u \sum_n \xi^{a_n - a_{n+u} - 2u\Delta f t} \\ &= \sum_{n=0}^{N-1} \xi^{a_n - a_n} + \sum_{u=1}^{N-1} \sum_{n=0}^{N-1-u} \xi^{a_n - a_{n+u} - 2u\Delta f t} + \sum_{u=-N+1}^{-1} \sum_{n=-u}^{N-1} \xi^{a_n - a_{n+u} - 2u\Delta f t} \\ &= \sum_{n=0}^{N-1} 1 + \sum_{u \neq 0} \sum_n \xi^{a_n - a_{n+u} - 2u\Delta f t} \\ &= N + \sum_{u \neq 0} \sum_n \xi^{a_n - a_{n+u} - 2u\Delta f t}. \end{aligned} \tag{2.20}$$

Vergleicht man nun (2.20) mit der Definition der *aperiodischen Autokorrelationsfunktion (AKF)* für die Sequenz a bei einer Verschiebung u,

$$C_a(u) = \sum_n \xi^{a_n - a_{n+u}} = \begin{cases} \sum_{n=-u}^{N-1} \xi^{a_n - a_{n+u}} & \text{für } -N < u < 0 \\ \sum_{n=0}^{N-1-u} \xi^{a_n - a_{n+u}} & \text{für } 0 \leq u < N \\ 0 & \text{sonst} \end{cases}, \tag{2.21}$$

so lässt sich leicht folgender Zusammenhang herstellen:

$$P_a(t) = N + \sum_{u \neq 0} C_a(u) \xi^{-2u\Delta f t} \qquad \text{für } 0 \leq t \leq T_0. \tag{2.22}$$

Die aperiodische Autokorrelationsfunktion nach (2.21) stellt hierbei ein Maß für die Ähnlichkeit von a mit seinen um u verschobenen Kopien dar, wobei im Gegensatz zur periodischen AKF die Sequenz nicht zyklisch wiederholt, sondern nur eine Periode berücksichtigt wird.

Unter Verwendung von (2.22) kann nun für die innerhalb einer Symbolperiode auftretende Spitzenleistung der komplexen Einhüllenden die obere Schranke

$$P_a(t) \leq N + \sum_{u \neq 0} |C_a(u)| \cdot 1 \leq N + 2 \sum_{u=1}^{N-1} N - u = N + 2 \cdot \frac{N}{2} \cdot (N-1) = N^2 \tag{2.23}$$

angegeben werden.

Die maximal auftretende momentane Leistung der komplexen Einhüllenden wird in der Literatur häufig mit *Peak Envelope Power* (PEP) bezeichnet. Betrachtet man nun zusätzlich,

[5] Bei Anwendung der Substitution muss innerhalb der Summen darauf geachtet werden, dass sowohl die Werte für n als auch die Werte für $n + u$ im Bereich { 0, 1, …, N-1 } liegen.

dass sich die mittlere Leistung der komplexen Einhüllenden über einer Symbolperiode unabhängig von der Sequenz a nach [DJ99] zu

$$E\{P_a(t)\} = N \tag{2.24}$$

ergibt, so folgt schließlich mit der Definition des PMEPR

$$\text{PMEPR} = \frac{\max\{P_a(t)\}}{E\{P_a(t)\}} = \frac{\max\{P_a(t)\}}{N} = \frac{PEP}{N}. \tag{2.25}$$

Aufgrund der oberen Schranke der PEP nach (2.23) ergibt sich damit direkt die obere Schranke des PMEPR:

$$\text{PMEPR} \leq \frac{N^2}{N} = N. \tag{2.26}$$

Dieser maximale Wert wird beispielsweise für die Null-/Einsfolge oder die alternierende Null-Einsfolge erreicht. Insgesamt verursachen jedoch nur wenige Sequenzen, deren Anzahl auch nicht mit N ansteigt, ein maximales PMEPR, weshalb das Auftreten des maximalen Wertes mit steigendem N immer unwahrscheinlicher wird.

In diesem Abschnitt wurde gezeigt, dass das PMEPR nur vom Spitzenwert der komplexen Einhüllenden abhängt, welcher wiederum direkt vom maximalen Wert der aperiodischen AKF der Sequenz a abhängt (siehe (2.23)). Zur Reduktion des PMEPR gilt es also Sequenzen zu finden, deren aperiodische AKF für alle Verschiebungen ungleich Null möglichst geringe Werte annimmt. Vertreter dieser Sequenzen sind beispielsweise die Barker-Sequenzen, bei denen der Betrag der aperiodischen AKF für Verschiebungen ungleich Null nur die Werte Null oder Eins annehmen kann [Jed08]. Von ihnen sind jedoch nur sehr wenige bekannt und die Existenz von Barker-Sequenzen mit einer Länge größer als 13 gilt als sehr unwahrscheinlich [Jed08].

2.2.3. Reduktion des PMEPR mittels Golay-Sequenzen

Nach [DJ99] ist das Paar der Sequenzen

$$a = (a_0, a_1, \ldots, a_{N-1}) \qquad \text{mit } a_i \in \{0, 1\} \tag{2.27}$$

und

$$b = (b_0, b_1, \ldots, b_{N-1}) \qquad \text{mit } b_i \in \{0, 1\} \tag{2.28}$$

als *komplementäres Golay-Paar über $\mathbb{Z}_2$ der Länge N* definiert, sofern für die Summe ihrer aperiodischen Autokorrelationsfunktionen[6]

$$C_a(u) + C_b(u) = \sum_n a_n \cdot a_{n+u} + \sum_n b_n \cdot b_{n+u} = 0 \qquad \text{für } u \neq 0 \tag{2.29}$$

[6] An dieser Stelle sei nochmals darauf hingewiesen, dass bei der Summation über n sowohl n als auch $n + u$ im Bereich $\{0, 1, \ldots, N-1\}$ liegen müssen.

gilt. Des weiteren wird jede Sequenz, die Teil eines komplementären Golay Paares ist, als *Golay-Sequenz* bezeichnet [Gol61].

Die Existenz komplementärer Golay-Paare über $\mathbb{Z}_2$ ist für Längen von $N = 2^{\alpha}10^{\beta}26^{\gamma}$, mit $\alpha, \beta, \gamma \geq 0$ nachgewiesen [DJ99]. Daraus folgt insbesondere, dass für Sequenzlängen, welche einer Zweierpotenz entsprechen, immer Golay-Sequenzen existieren.

Setzt man (2.29) in (2.22) ein, so ergibt sich die Summe der momentanen Leistungen der komplexen Einhüllenden für komplementäre Golay-Paare zu:

$$P_a(t) + P_b(t) = N + \sum_{u \neq 0} C_a(u)\xi^{-2u\Delta ft} + N + \sum_{u \neq 0} C_b(u)\xi^{-2u\Delta ft} \overset{(2.29)}{=} 2N. \tag{2.30}$$

Da die Leistung der Sequenz b eine positive Größe ist, gilt

$$P_b(t) = |u_b(t)|^2 \geq 0. \tag{2.31}$$

Nach Umformen von (2.30) und Einsetzen von (2.31) ergibt sich schließlich die obere Schranke für die Leistung der komplexen Einhüllenden einer Golay-Sequenz der Länge N nach [Pop91]:

$$\begin{aligned} P_a(t) + P_b(t) &= 2N \\ P_b(t) &= 2N - P_a(t) \\ 0 &\leq 2N - P_a(t) \\ P_a(t) &\leq 2N. \end{aligned} \tag{2.32}$$

Setzt man nun (2.32) in (2.25) ein, so ergibt sich das PMEPR für Golay-Sequenz zu

$$\text{PMEPR} = \frac{\max\{P_a(t)\}}{N} \leq \frac{2N}{N} = 2. \tag{2.33}$$

Damit ist gezeigt, dass das PMEPR, und damit auch das PAPR, für Golay-Sequenzen auf 2 bzw. 3 dB begrenzt ist.

In diesem Kapitel wurde das Mehrträgerverfahren OFDM eingeführt. Es wurde auf eines der gravierendsten Probleme bei diesem Verfahren, das Peak Power Problem, hingewiesen und eine Möglichkeit zur Milderung der negativen Einflüsse des selbigen erläutert. Im nächsten Kapitel wird der Lösungsansatz zur PMEPR-Reduktion mittels Golay-Sequenzen erneut aufgegriffen und im Hinblick auf die Möglichkeit einer praktischen Implementierung weiterentwickelt.

3. Reed-Muller Codes

Die in diesem Kapitel betrachteten Reed-Muller Codes (*RM*-Codes) können der Klasse der fehlerkorrigierenden linearen Blockcodes zugeordnet werden. Sie stellen ein Verfahren zur Kanalcodierung dar und wurden für den binären Fall von I.S. Reed im Jahre 1954 vorgestellt [Ree54]. An der Entwicklung beteiligt war neben Irving S. Reed noch David E. Muller, woraus direkt die Namensgebung folgt.

Eines der bekanntesten Beispiele für die Anwendung von *RM*-Codes stellt die Mariner 9 Mission der NASA im Jahre 1971 dar [Fri95, S. 404]. Hierbei wurden Graustufen-Bilder von der Marsoberfläche zur Erde übertragen. Aufgrund der großen Distanz und der, im Vergleich zu heute, eingeschränkten technischen Möglichkeiten hätte eine Übertragung ohne den Einsatz einer geeigneten Codierung sehr hohe Bit-Fehlerwahrscheinlichkeiten mit sich gebracht. Mit dem Einsatz der *RM*-Codierung konnte diese erheblich reduziert werden.

In [DJ97] konnten James A. Davis und Jonathan Jedwab erstmals eine Verbindung zwischen Reed-Muller Codes zweiter Ordnung und Golay-Sequenzen nachweisen. Dies stellt die Grundlage für die praktische Implementierung, welche Ziel dieser Arbeit ist, dar.

In diesem Kapitel werden zunächst einige wichtige Grundbegriffe der Kanalcodierung eingeführt. Anschließend folgt die Definition binärer boolescher Funktionen, welche zur Beschreibung von *RM*-Codes zwingend erforderlich sind. Schließlich werden in Abschnitt 3.3. die binären Reed-Muller Codes vorgestellt, wobei direkt auf Methoden zur En-/Decodierung eingegangen wird. Nach einer kurzen Erläuterung zur Verallgemeinerung binärer *RM*-Codes wird im letzten Teil schließlich der Zusammenhang zwischen binären *RM*-Codes und binären Golay-Sequenzen hergestellt und erläutert.

3.1. Grundlagen der Kanalcodierung

Kanäle beeinflussen Signale auf die verschiedensten Arten. Einige davon üben einen negativen Einfluss auf das Signal aus, wodurch Fehler am Empfänger entstehen können. Die Kanalcodierung hat zum Ziel diese Fehler erkennen oder gar korrigieren zu können. Dies wird durch gezieltes Hinzufügen von Redundanz zur eigentlichen Information erreicht. Im Folgenden werden nun einige Grundbegriffe der Kanalcodierung eingeführt, welche für die späteren Betrachtungen von Bedeutung sind.

Kanalcodierungstheorem

In [Sha48] führte C.E. Shannon, der als der Begründer der heutigen Informationstheorie gilt, erstmals das *Kanalcodierungstheorem* ein. Es besagt, dass für einen diskreten, gedächtnislosen Kanal mit der *Kanalkapazität*

$$C = W \log_2(1 + \mathrm{SNR}), \tag{3.1}$$

wobei W für die *Systembandbreite* und SNR für das *Signal-zu-Rauschverhältnis* steht, bei optimaler Decodierung[1] eine Codierung existiert, mit der die *Wort-Fehlerwahrscheinlichkeit* beliebig klein gemacht werden kann, sofern die *Bitübertragungsrate* R kleiner als die Kanalkapazität C ist.

Das Kanalcodierungstheorem geht lediglich auf die Existenz einer solchen Codierung ein, nicht jedoch auf deren Aufbau. Im Umkehrschluss folgt aus dem Kanalcodierungstheorem, dass für eine Bitübertragungsrate R, welche größer als die Kanalkapazität C ist, eine bestimmte untere Schranke für die Wort-Fehlerwahrscheinlichkeit existiert, die durch Codierung nicht weiter verbessert werden kann.

Mit Hilfe des Kanalcodierungstheorems lässt sich außerdem die *Shannon-Grenze* herleiten [Jon08, S. 117ff.]. Sie besagt, dass für das Verhältnis der *pro Bit empfangenen Energie* E_b zur *Rauschleistungsdichte* N_0 bei etwa $-1,59$ dB eine untere Grenze existiert, ab der fehlerfreie Übertragung überhaupt erst möglich ist.

Coderate

Die *Coderate* R gibt das Verhältnis von k *Informationsbits* zu n *Codebits* an:

$$R = \frac{k}{n} \leq 1. \tag{3.2}$$

Sie macht eine Aussage darüber, welchen Anteil das Informationswort an der Gesamtlänge eines Codewortes besitzt und ist damit indirekt ein Maß für die hinzugefügte Redundanz.

Hammingdistanz und Hamminggewicht

Die *Hammingdistanz* gibt an, an wie vielen Stellen sich zwei Codewörter unterscheiden.

Beispiel: Die Codewörter 00101110 und 10101011 unterscheiden sich in 3 Stellen und weisen somit eine Hammingdistanz von 3 auf.

Neben der Hammingdistanz ist außerdem das *Hamminggewicht* wt_H von Bedeutung. Als Hamminggewicht bezeichnet man die Anzahl der Stellen im Codewort, welche ungleich Null sind.

[1] Die Maximum-Likelihood Decodierung gilt als optimale Decodierung.

Beispiel: Das Codewort 00101110 hat 4 von Null verschiedene Stellen und besitzt somit ein Hamminggewicht von 4.

Korrekturfähigkeit

Um Aussagen über die Leistungsfähigkeit eines Codes im Sinne der Fehlererkennung bzw. -korrektur machen zu können, betrachtet man die *minimale Hammingdistanz* d_{min}. Sie ist definiert als die minimale Hammingdistanz, die bei Betrachtung aller möglichen Paare von Codewörtern auftreten kann. Bei linearen Blockcodes entspricht die minimalen Hammingdistanz gerade dem kleinsten auftretenden Hamminggewicht der Codewörter. Insgesamt lassen sich in Abhängigkeit der minimalen Hammingdistanz maximal

$$t' = d_{min} - 1 \tag{3.3}$$

Fehler erkennen. Um Fehler korrigieren zu können, müssen, zur eindeutigen Zuordnung eines Empfangsvektors zu einem Codewort, die einzelnen Codewörter eine größere Hammingdistanz aufweisen. So lassen sich bei linearen Blockcodes maximal

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor \tag{3.4}$$

Fehler korrigieren [Fri95, S. 73ff.].

Codierungsgewinn

Da bei den Verfahren zur Kanalcodierung Rückschlüsse auf den Sendevektor durch Hinzufügen von Redundanz ermöglicht werden, entsteht ein gewisser Overhead. Es müssen also mehr Bits als bei der uncodierten Variante übertragen werden. Um nun eine gleich bleibende Datenrate für die Informationsbits zu erreichen muss aufgrund des Overheads die Bitdauer verkürzt werden, was zu einer geringeren *Sendeleistung pro Bit* E_b führt. Daraus ergibt sich direkt ein geringeres, auf ein Bit bezogenes, SNR $\frac{E_b}{N_0}$. Aufgrund dessen werden die einzelnen Bits bei der codierten Übertragung im Vergleich zur uncodierten Übertragung bei gleich bleibender Sendeleistung stärker verfälscht. Der Code muss nun im Stande sein so viele Fehler zu korrigieren, dass dieser Effekt wieder ausgeglichen wird. Nur wenn der Code auch fähig ist weitere Fehler korrigieren zu können ergibt sich ein Gewinn im Vergleich zur uncodierten Übertragung.

Der *Codierungsgewinn* G_c ist definiert als die Differenz der Sendeleistung pro Bit E_b der uncodierten und codierten Übertragung, bezogen auf die jeweilige Rauschleistungsdichte N_0, bei der für beide Übertragungen die identische *Bitfehlerwahrscheinlichkeit* P_b erreicht wird. Damit folgt:

$$G_c(P_b) = \left[\frac{E_b}{N_0}\right]_{P_b,uncodiet} - \left[\frac{E_b}{N_0}\right]_{P_b,codiet}. \tag{3.5}$$

Betrachtet man den Codierungsgewinn beim Grenzübergang $\frac{E_b}{N_0} \to \infty$, so ergibt sich der *asymptotische Codierungsgewinn* G_a. Er stellt eine obere Schranke für den Codierungsge-

winn G_c dar. Aufgrund der unterschiedlichen Leistungsfähigkeiten der *Hard-Decision*- (HD) und *Soft-Decision-Decodierung* (SD) wird hierbei zwischen beiden unterschieden.

Nach [Fri95, S. 29] lässt sich der asymptotische Codierungsgewinn für HD-Decodierung mit

$$G_{a,hard} = 10 \cdot \log_{10}(t+1) \quad \text{[dB]} \tag{3.6}$$

berechnen. Für die SD-Decodierung folgt

$$G_{a,soft} = 10 \cdot \log_{10}(d_{min}) \quad \text{[dB]}. \tag{3.7}$$

Die SD-Decodierung ist prinzipiell leistungsfähiger als die HD-Decodierung, weshalb der asymptotische Codierungsgewinn für SD-Decodierung bis zu 3 dB höher liegen kann als bei HD-Decodierung [Fri95, S. 28]. In praktischen Systemen werden jedoch meist nur 2 dB erreicht.

3.2. Binäre Boolesche Funktionen

Für die weiteren Betrachtungen, insbesondere bei den binären Reed-Muller Codes, werden zunächst die Grundlagen der binären Booleschen Funktionen eingeführt, die im weiteren Verlauf nur noch als Boolsche Funktionen bezeichnet werden. Boolesche Funktionen gehen zurück auf George Boole, welcher sich bereits im 19. Jahrhundert mit der mathematischen Beschreibung der Logik befasste.

Man definiere zunächst die *Boolesche Variable* x; Eine Boolesche Variable, oder auch binäre Variable, ist eine Variable, die nur die Werte 0 und 1 annehmen kann:

$$x \in \{0,1\}. \tag{3.8}$$

Eine *binäre Boolesche Funktion* ist definiert als eine Funktion, welche nur von Booleschen Variablen abhängt und deren Ergebnis nur 0 oder 1 sein kann. Auch Boolesche Variablen stellen hierbei Boolesche Funktionen dar. Mathematisch kann eine Boolesche Funktion als eine Abbildung von $\mathbb{Z}_2^m$ nach $\mathbb{Z}_2$ beschrieben werden:

$$f : \mathbb{Z}_2^m = \{(x_1, x_2, \ldots, x_m) \text{ mit } x_i \in \{0,1\}\} \rightarrow \mathbb{Z}_2. \tag{3.9}$$

Damit ergeben sich nun 2^m Monome. Das leere Monom 1 entspricht der Booleschen Funktion nullter Ordnung und enthält 2^m Einsen.

Die m Booleschen Funktionen erster Ordnung entsprechen den Monomen $x_i = x_1, x_2, \ldots, x_m$. Jedes dieser Monome ist eine Sequenz der Länge 2^m, wobei alternierend 2^{m-i} Nullen bzw. Einsen, beginnend mit null, die Elemente dieser Sequenz bilden.

Die Booleschen Funktionen höherer Ordnungen ergeben sich aus Linearkombinationen der Booleschen Funktionen erster Ordnung, wobei die Koeffizienten wiederum Elemente von $\mathbb{Z}_2$ sind. Da alle Berechnungen im *Galois Feld(2)* (GF(2)) erfolgen, entspricht die

Und-Verknüpfung der Multiplikation und es kann für die Linearkombinationen, zweier Monome beispielsweise, die Schreibweise $x_i x_j$ eingeführt werden.

Die Boolesche Funktion der Ordnung m ergibt sich aus der Und-Verknüpfung aller Boolescher Funktionen der ersten Ordnung. Das folgende Beispiel zeigt die Monome für $m = 3$:

Beispiel: Boolesche Funktionen für $m = 3$:

Die Boolesche Funktion nullter Ordnung lautet:

$$1 \quad = \quad (11111111).$$

Die Booleschen Funktionen der ersten Ordnung lauten:

$$\begin{aligned} x_1 &= (00001111) \\ x_2 &= (00110011) \\ x_3 &= (01010101). \end{aligned}$$

Die Booleschen Funktionen zweiter Ordnung ergeben sich zu:

$$\begin{aligned} x_1 x_2 &= (00000011) \\ x_1 x_3 &= (00000101) \\ x_2 x_3 &= (00010001). \end{aligned}$$

Die Boolesche Funktion dritter Ordnung lautet:

$$x_1 x_2 x_3 \quad = \quad (00000001).$$

3.3. Binäre Reed-Muller Codes

In diesem Abschnitt werden die *binären Reed-Muller Codes* zusammen mit der Methode zur Hard-Decision Decodierung nach Reed vorgestellt. Hierbei wird, im Hinblick auf die spätere Implementierung, nur auf den binären Fall eingegangen. Wie im nächsten Abschnitt gezeigt wird, existiert die Möglichkeit zur Verallgemeinerung der Reed-Muller Codes, so dass Definitionen für höherwertige Alphabete, wie z.B. $\mathbb{Z}_4$ oder $\mathbb{Z}_8$, gefunden werden können.

Da sich die folgenden Betrachtungen auf den binären Fall beziehen, ist in diesem und allen folgenden Kapiteln die Bezeichnung $RM(r, m)$ mit der Bezeichnung $RM_2(r, m)$ gleichzusetzen. Auf höherwertige Alphabete wird ausschließlich in Abschnitt 3.4 eingegangen. Diese tragen generell die Bezeichnung $RM_q(r, m)$, wobei q für die Mächtigkeit des zugrunde gelegten Alphabets steht.

3.3.1. Definition und Eigenschaften

Der *binäre Reed-Muller Code* $RM(r,m)$ *der Ordnung* r ist gegeben durch die Menge aller Boolescher Funktionen der Länge

$$n = 2^m \tag{3.10}$$

und einer Ordnung kleiner oder gleich r.

Die Booleschen Funktionen der Ordnung i ergeben sich aus Linearkombinationen der m Booleschen Funktionen erster Ordnung, wobei i die Anzahl der Monome beschreibt, welche am Aufbau beteiligt sind. Es werden also i Monome aus m ausgewählt und miteinander verknüpft. Da der $RM(r,m)$-Code aus den Booleschen Funktionen bis zur Ordnung r aufgebaut ist, folgt für die Dimension dieses Codes:

$$k = 1 + \binom{m}{1} + \binom{m}{2} + \ldots + \binom{m}{r} = \sum_{i=0}^{r} \binom{m}{i}. \tag{3.11}$$

Da es sich bei den binären Reed-Muller Codes um lineare Blockcodes handelt, werden k Informationsbits auf n Codebits codiert. Daraus ergibt sich direkt die Coderate:

$$R = \frac{k}{n} = \frac{\sum_{i=0}^{r} \binom{m}{i}}{2^m}. \tag{3.12}$$

Wählt man verschiedene Kombinationen für r und m, so ergeben sich unterschiedliche Coderaten, wovon hier zwei Sonderfälle insbesondere erwähnt werden sollen; Wählt man $r = 0$, so ergibt sich ein Wiederholungscode der Länge $n = 2^m$ mit einer Coderate $R = \frac{1}{2^m}$, bei dem das zu codierende Bit lediglich 2^m-mal wiederholt wird. Wählt man $r = m$, so ergibt sich die Coderate $R = 1$, was zur Folge hat, dass keinerlei Fehler mehr korrigiert werden können.

Insgesamt besitzt der $RM(r,m)$-Code

$$2^k = 2^{\sum_{i=0}^{r} \binom{m}{i}} \tag{3.13}$$

verschiedene gültige Codewörter, die zueinander eine minimale Hammingdistanz von

$$d_{min} = 2^{m-r} \tag{3.14}$$

aufweisen. Damit folgt unter Berücksichtigung von (3.4), dass mit diesem Code

$$t = \left\lfloor 2^{m-r-1} - 1 \right\rfloor \tag{3.15}$$

Fehler korrigiert werden können.

3.3.2. Encodierung

Da es sich bei den binären Reed-Muller Codes um lineare Blockcodes handelt, wird für die Encodierung eine *Generatormatrix* $\boldsymbol{G}$ verwendet. Die Booleschen Funktionen $1, x_1, x_2, \ldots, x_m, x_1x_2, \ldots, x_1x_2 \ldots x_m$ stellen hierbei direkt die Zeilen der Generatormatrix dar[2], wobei lediglich Monome bis zur Ordnung r berücksichtigt werden. Der Aufbau der Generatormatrix lässt sich leicht an einem Beispiel zeigen:

Für $r = 0$ tritt nur das leere Monom als Zeile der Generatormatrix auf, da nur die Boolesche Funktion nullter Ordnung berücksichtigt wird.

Beispiel: Generatormatrix für $RM(0,3)$:

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Für $r = 1$ werden neben der nullten Ordnung außerdem die m Booleschen Funktionen der ersten Ordnung berücksichtigt.

Beispiel: Generatormatrix für $RM(1,3)$:

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ \boldsymbol{x_2} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_3} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Für $r = 2$ treten nun auch Boolesche Funktionen der zweiten Ordnung auf, d.h. alle möglichen Und-Verknüpfung von zwei der m Booleschen Funktionen erster Ordnung.

Beispiel: Generatormatrix für $RM(2,3)$:

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ \boldsymbol{x_2} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_3} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_1x_2} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_1x_3} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_2x_3} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Es werden also sukzessive genau so lange $\binom{m}{i}$ Reihen zur Generatormatrix von $RM(i-1, m)$ hinzugefügt bis $i = r$ erreicht ist.

[2] Die höchste vorkommende Ordnung, d.h. die maximale Anzahl der Faktoren, wird durch r bestimmt.

Somit ergibt sich für $r = m$ die letzte Zeile der Generatormatrix von $RM(m, m)$ durch die Boolesche Funktion m-ter Ordnung, welche wiederum durch die Und-Verknüpfung aller Monome $x_1, x_2, \ldots, x_m$ entsteht.

Beispiel: Generatormatrix für $RM(3, 3)$:

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_2} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_3} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_1x_2} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_1x_3} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_2x_3} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \boldsymbol{x_1x_2x_3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Für $m = 4$ und $r = 2$ kann äquivalent vorgegangen werden; hier existieren $m = 4$ Boolesche Funktionen erster Ordnung. Dadurch ergeben sich, aufgrund der Beziehung $\binom{m}{i}$ für $i = r = 2$ nun 6 Boolesche Funktionen zweiter Ordnung.

Beispiel: Generatormatrix für $RM(2, 4)$:

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_2} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_3} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_4} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_1x_2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1x_3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_1x_4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_2x_3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_2x_4} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \boldsymbol{x_3x_4} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Die eigentliche Encodierung eines *Datenvektors $\vec{d}$ der Länge k* zu einem *Codevektor $\vec{c}$ der Länge n* lässt sich zusammen mit der Generatormatrix $\boldsymbol{G}$ sehr einfach ausdrücken:

$$\vec{c}^T = \vec{d}^T \boldsymbol{G} \tag{3.16}$$

Führt man die Matrixmultiplikation nach (3.16) aus, so erkennt man, dass die Komponenten des Datenvektors $d_0, d_1, \ldots, d_{k-1}$ als Koeffizienten für die gewichtete Addition[3] der

[3] Bei der gewichteten Addition der Zeilen können nur die Koeffizienten 0 und 1 auftreten. Die Koeffizienten bestimmen also, ob eine Zeile am Aufbau des Codevektors beteiligt ist oder nicht. Weiter gilt es zu beachten, dass alle Berechnungen in $GF(2)$ ausgeführt werden. Die Addition entspricht also einer XOR-Verknüpfung.

Zeilen der Generatormatrix interpretiert werden können. Dies soll anhand des folgenden Beispiels verdeutlicht werden.

Beispiel: Encodierung von $\vec{d}^T = (0111)$ für den $RM(1,3)$-Code:

$$\vec{c}^T = \vec{d}^T \boldsymbol{G} = (0111) \cdot \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_2} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_3} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} = (01101001)$$

3.3.3. Hard-Decision Decodierung nach Reed

Zur HD-Decodierung eines *Empfangsvektors* $\vec{r}$ wird an dieser Stelle ein Algorithmus vorgestellt, welcher auf dem von Reed in [Ree54] vorgeschlagenen Algorithmus zur Decodierung basiert. Dabei wird das *Prinzip der Mehrheitsentscheidung* (majority logic, „threshold" decoding) angewandt, welches in diesem Fall (für binäre Reed-Muller Codes) eine optimale HD-Decodierung[4] darstellt [Wic95, S. 155].

Charakteristische Vektoren

Bei dem folgenden Algorithmus werden die *charakteristischen Vektoren* der Zeilen der Generatormatrix zur Decodierung verwendet. Sie besitzen die Eigenschaft, dass das Skalarprodukt eines charakteristischen Vektors mit einer Zeile der Generatormatrix nur für die zugehörige Zeile ungleich null ist; sie sind also orthogonal zu allen anderen Zeilen.

Zur Bildung der charakteristischen Vektoren der Zeile i in $\boldsymbol{G}$ betrachte man die Booleschen Funktionen erster Ordnung, welche am Aufbau dieser Zeile beteiligt sind[5]. Die 2^{m-i} charakteristischen Vektoren der Zeile i ergeben sich nun aus Linearkombinationen der Booleschen Funktionen erster Ordnung, welche nicht am Aufbau beteiligt sind. Diese $m - i$ Monome können sowohl in ihrer eigentlichen als auch in ihrer negierten Form am Aufbau der charakteristischen Vektoren beteiligt sein, wobei das Vorhandensein beider Formen innerhalb eines charakteristischen Vektors jedoch ausgeschlossen ist. Der Aufbau der charakteristischen Vektoren lässt sich leicht am folgenden Beispiel nachvollziehen:

[4] Maximum Likelihood Decodierung

[5] Die Zeilen in $\boldsymbol{G}$ stellen, abgesehen von der ersten Zeile, Boolesche Funktionen erster Ordnung oder Linearkombinationen der selbigen dar.

Beispiel: Charakteristische Vektoren für $RM(2,4)$:

Am Aufbau der letzten Zeile der Generatormatrix für $RM(2,4)$ sind die Booleschen Funktionen erster Ordnung x_3 und x_4 beteiligt (s.o.). Daraus folgt, dass x_1 und x_2 sowohl in eigentlicher als auch in negierter Form am Aufbau der charakteristischen Vektoren beteiligt sein können:

$$x_1, x_2, \bar{x_1}, \bar{x_2}.$$

Die charakteristischen Vektoren sind nun alle Linearkombinationen dieser Monome, wobei jedes Monom nur einmal, bezogen auch auf seine negierte Form, vorkommen darf:

$$x_1x_2, \bar{x_1}x_2, x_1\bar{x_2}, \bar{x_1}\bar{x_2}.$$

Algorithmus zur HD-Decodierung

Das Skalarprodukt eines Empfangsvektors $\vec{r}$ mit einem charakteristischen Vektor einer Zeile der Generatormatrix kann als eine Schätzung dafür interpretiert werden, ob diese Zeile am Aufbau des gesendeten Codewortes $\vec{c}$ beteiligt war oder nicht. Dies gilt für alle charakteristischen Vektoren, wobei sie bei der Abschätzung jeweils unterschiedliche Bits des Empfangsvektors miteinbeziehen[6].

Die Mehrheit der Ergebnisse stellt eine Schätzung für den Koeffizienten der zu den charakteristischen Vektoren gehörenden Zeile dar, wobei alle relevanten Bits des Empfangsvektors $\vec{r}$ berücksichtigt wurden. Treten die Ergebnisse 0 und 1 gleich häufig auf, so deutet dies auf eine fehlerhafte Übertragung hin, bei der der Fehler jedoch nicht korrigiert werden kann.

Insgesamt führen diese Zusammenhänge auf die folgende, von Reed vorgestellte, Methode zur HD-Decodierung eines beliebigen $RM(r,m)$-Codes:

(1) Setze $\vec{r^0} = \vec{r}$, $i = r$, $\hat{d} = \vec{0}$ und beginne mit der letzten Zeile der Generatormatrix $\boldsymbol{G}$.

(2) Bilde die 2^{m-i} charakteristischen Vektoren der aktuellen Zeile.

(3) Bilde das Skalarprodukt von $r^{(n)}$ mit jedem der charakteristischen Vektoren.

(4) Die Mehrheit der Ergebnisse aus (3) ergibt den Koeffizienten und damit das geschätzte Informationsbit $\hat{d}$ für die aktuelle Zeile.

[6] Jedes der n Bits im Codewort wird nach einem bestimmten Muster aus den k Bits des Informationswortes erzeugt. Diese Verknüpfung ist durch die Generatormatrix vorgegeben und lässt sich entsprechend umkehren, so dass sich aus den Bits des Codewortes Rückschlüsse auf die Informationsbits schließen lassen.

(5) Wiederhole (2) bis (4) für alle Zeilen, welche aus den Booleschen Funktionen der Ordnung i aufgebaut sind.

(6) Setze $r^{\vec{(n)}} = r^{(\vec{n-1})} - \hat{d}\boldsymbol{G}$ und verringere i um 1.

(7) Wiederhole (2) bis (6) bis $i = 0$ gilt.

(8) Die Mehrheit der Werte von $r^{\vec{(n)}}$ ergibt den Koeffizienten der ersten Zeile und damit $\hat{d}_0$.

Zur Veranschaulichung sei im Folgenden noch ein Beispiel zur HD-Decodierung nach Reed gegeben:

Beispiel: HD-Decodierung für $RM(1,3)$:

Die Generatormatrix für den $RM(1,3)$-Code lautet:

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \boldsymbol{x_2} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \boldsymbol{x_3} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

Bei Verwendung eines $RM(1,3)$-Codes lässt sich nach (3.15) $\left\lfloor 2^{3-1-1} - 1 \right\rfloor = 1$ Fehler korrigieren.

Man nehme nun eine gestörte Übertragung des im Beispiel zur Encodierung verwendeten Codewortes $\vec{c} = (01101001)$ an, bei dem genau ein Bit verfälscht wurde:

$$\vec{r} = (01111001).$$

Entsprechend dem Algorithmus setzen wir $\vec{r^0} = \vec{r} = (01111001)$, $i = r = 1$ und beginnen mit der letzten Zeile der Generatormatrix.

Die charakteristischen Vektoren dieser Zeile ergeben sich zu: $x_1x_2, \bar{x_1}x_2, x_1\bar{x_2}, \bar{x_1}\bar{x_2}$. Damit folgt für die Skalarprodukte der charakteristischen Vektoren mit $\vec{r}$:

$$(01111001) \cdot (00000011) = 1 \qquad (01111001) \cdot (00110000) = 0$$
$$(01111001) \cdot (00001100) = 1 \qquad (01111001) \cdot (11000000) = 1.$$

Die Mehrheit der Werte bestimmt den Koeffizienten der letzten Zeile: $\hat{d}_3 = 1$.

Für die dritte Zeile ergeben sich die charakteristischen Vektoren zu: $x_1x_3, \bar{x_1}x_3, x_1\bar{x_3}, \bar{x_1}\bar{x_3}$. Damit folgt für die Skalarprodukte:

$$(01111001) \cdot (00000101) = 1 \qquad (01111001) \cdot (01010000) = 0$$
$$(01111001) \cdot (00001010) = 1 \qquad (01111001) \cdot (10100000) = 1.$$

Die Mehrheit bestimmt den Koeffizienten der dritten Zeile: $\hat{d_2} = 1$.

Für die zweite Zeile lauten die charakteristischen Vektoren: $x_2x_3, \bar{x_2}x_3, x_2\bar{x_3}, \bar{x_2}\bar{x_3}$. Damit ergeben sich die Skalarprodukte zu:

$$(01111001) \cdot (00010001) = 1 \qquad (01111001) \cdot (01000100) = 1$$
$$(01111001) \cdot (00100010) = 1 \qquad (01111001) \cdot (10001000) = 1.$$

Die Ergebnisse der Skalarprodukte lauten alle 1, somit ergibt sich der Koeffizient der zweiten Zeile zu: $\hat{d_1} = 1$.

Da nun alle Monome der Ordnung $i = 1$ abgearbeitet wurden, setzt man $\vec{r^{(n)}}$ zu:

$$\vec{r^{(n)}}^T = \vec{r^{(n-1)}}^T - \vec{\hat{d}}^T \boldsymbol{G} = (01111001) - (01101001) = (00010000)$$

Der Koeffizient für die erste Zeile ergibt sich nun aus der Mehrheit der Werte in $\vec{r^{(n)}}$, in diesem Fall also $\hat{d_0} = 0$.

Damit ergibt sich das decodierte Datenwort zu

$$\hat{d} = (0111).$$

Der Fehler in der Übertragung wurde also erkannt und korrigiert.

3.4. Verallgemeinerte Reed-Muller Codes

In [DJ99] werden neben den binären Booleschen Funktionen außerdem noch die *verallgemeinerten Booleschen Funktionen* eingeführt. Sie sind definiert als eine Abbildung der Form

$$f : \mathbb{Z}_2^m \to \mathbb{Z}_{2^h}. \tag{3.17}$$

Mit den verallgemeinerten Booleschen Funktionen lässt sich die Definition der binären Reed-Muller Codes verallgemeinern, so dass ebenfalls RM-Codes für den Raum $\mathbb{Z}_{2^h}$ angegeben werden können. In diesem Fall tragen sie, wie bereits erwähnt, die Bezeichnung $RM_{2^h}(r, m)$ bzw. mit $q = 2^h$ die Bezeichnung $RM_q(r, m)$.

Auch für die binären komplementären Golay-Paare lässt sich eine Verallgemeinerung für den Raum $\mathbb{Z}_{2^h}$ finden, so dass sich die folgenden Zusammenhänge auch für größere Räume als $\mathbb{Z}_2$ herleiten lassen.

Bedeutung hat diese Verallgemeinerung für eine praktische Implementierung in dem Sinne, als das damit eine höherwertige PSK-Modulationen für die Subträger eines OFDM-Systems bei reduziertem PAPR ermöglicht wird. Im Hinblick auf die in dieser Arbeit angestrebte Implementierung wird jedoch nur der binäre Fall betrachtet. Für weitere Ausführungen bezüglich der möglichen Verallgemeinerung sei beispielsweise auf [DJ99] oder [Pat00] verwiesen.

3.5. Reed-Muller Codes und Golay-Sequenzen

In [DJ97] wurde erstmals eine Verbindung zwischen den Reed-Muller Codes der zweiten Ordnung und Golay-Sequenzen hergestellt. Im Folgenden werden nun die Zusammenhänge erläutert und jeweils eine Möglichkeit zur En- und Decodierung nach [DJ97] vorgestellt.

3.5.1. Nebenklassenzerlegung des $RM(2,m)$-Codes

Nach [Fri95, S.401] gilt für binäre Reed-Muller Codes die *Inklusionsbeziehung*

$$RM(r,m) \subset RM(r+1,m). \tag{3.18}$$

Der binäre Reed-Muller Code der Ordnung r ist also als eine Untermenge des binären Reed-Muller Codes der Ordnung $r+1$.

Von besonderem Interesse sind hier die $RM(2,m)$-Codes welche sich in Nebenklassen des $RM(1,m)$-Codes unterteilen lassen. Der $RM(2,m)$-Code beinhaltet nach (3.13) gerade $2^{1+m+\binom{m}{2}}$ verschiedene Codewörter, von denen sich 2^{m+1} dem $RM(1,m)$-Code zuordnen lassen. Folglich existieren

$$2^{\binom{m}{2}} = 2^{\frac{m(m-1)}{2}} \tag{3.19}$$

Nebenklassen des $RM(1,m)$-Codes innerhalb des $RM(2,m)$-Codes, welche jeweils 2^{m+1} Codewörter enthalten.

Abb. 3.1 verdeutlicht diesen Zusammenhang für $m = 3$. Nach (3.19) existieren für $m = 3$ genau 8 Nebenklassen des $RM(1,3)$-Codes innerhalb des $RM(2,3)$-Codes. In Abb. 3.1 ist dies dadurch angedeutet, dass die Nebenklassen des $RM(1,3)$-Codes durch gleichgroße Quadrate innerhalb des Rechtecks, welches für den $RM(2,3)$-Code steht, dargestellt sind.

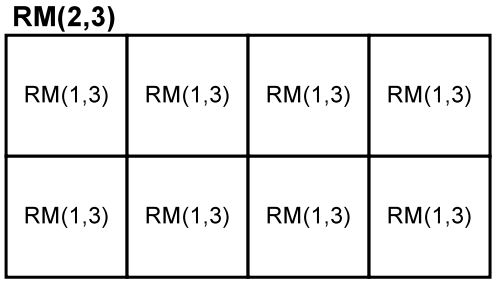

Abbildung 3.1.: Nebenklassen für $RM(2,3)$

Die Nebenklassenzerlegung lässt sich anschaulich erklären, indem man das Zustandekommen der Codewörter des $RM(2,m)$-Codes betrachtet. Der $RM(1,m)$-Code ergibt sich aus dem $RM(2,m)$-Code, indem man bei der Encodierung jene Informationsbits, welche innerhalb der Generatormatrix den Booleschen Funktionen der zweiten Ordnung zugeordnet sind, zu null setzt und die restlichen Bits variiert. Analog dazu ergeben sich die restlichen Nebenklassen durch die feste Wahl der Informationsbits, welche den Booleschen Funktionen der zweiten Ordnung zugeordnet sind, und variieren der restlichen Bits.

Tabelle 3.1 zeigt die Möglichkeiten für $m = 3$, wobei zusätzlich die *Nebenklassenanführer* angegeben sind[7].

$d_4d_5d_6$	Nebenklassenanführer
000	00000000
001	00010001
010	00000101
011	00010100
100	00000011
101	00010010
110	00000110
111	00010111

Tabelle 3.1.: Nebenklassenanführer für $RM(2,3)$

Die Nebenklassenanführer ergeben sich, indem man alle Informationsbits des $RM(2,3)$-Codes, außer jenen, welche gerade den Booleschen Funktionen der zweiten Ordnung zugeordnet sind, zu null setzt und die restlichen Bits variiert. Sie sind repräsentativ für die jeweilige Nebenklasse.

[7] Für $m = 3$ besitzt die Generatormatrix genau 3 Zeilen, welche aus den Booleschen Funktionen der zweiten Ordnung aufgebaut sind. Bei der Encodierung bilden somit die letzten 3 Bits die Koeffizienten für diese Zeilen.

3.5.2. Identifikation der Golay-Nebenklassen

Nach [DJ99, Corollary 6] existieren nun genau $\frac{m!}{2}$ Nebenklassen[8] von $RM(1,m)$ in $RM(2,m)$ der Form

$$\sum_{k=1}^{m-1} x_{\Pi(k)} x_{\Pi(k+1)}, \tag{3.20}$$

welche ausschließlich Golay-Sequenzen der Länge 2^m beinhalten. Π bezeichnet hierbei die Permutation der Symbole $\{1,2,\ldots,m\}$.

Beispiel: Golay-Nebenklassen in $RM(2,3)$:

Die eigentliche Reihenfolge der Elemente ist gegeben durch $\{1,2,3\}$. Dies führt nach (3.20) auf:

$$x_1x_2 + x_2x_3.$$

Die erste Permutation ist gegeben durch $\{1,3,2\}$. Damit folgt für den Nebenklassenanführer:

$$x_1x_3 + x_3x_2.$$

Für die zweite Permutation, $\{2,1,3\}$, folgt analog:

$$x_2x_1 + x_1x_3.$$

Es existieren 3 weitere Permutationen, welche jedoch auf die selben Ergebnisse führen. Diese lauten:

$$\{3,2,1\},\ \{2,3,1\},\ \{3,1,2\}.$$

Die resultierenden Golay-Nebenklassen sind in Tabelle 3.2 gegeben.

$d_4d_5d_6$	Nebenklassenanführer
011	00010100
101	00010010
110	00000110

Tabelle 3.2.: Anführer der Golay-Nebenklassen für $RM(2,3)$

Es ist also mit Hilfe der Reed-Muller Codes zweiter Ordnung möglich, ausschließlich Golay-Sequenzen zu erzeugen, wobei zusätzlich noch eine Möglichkeit zur Fehlerkorrektur gegeben ist.

[8] Der Faktor $\frac{1}{2}$ in $\frac{m!}{2}$ ergibt sich dadurch, dass eine Vertauschung der Reihenfolge der Elemente einer Permutationen auf das selbe Ergebnis führt wie die Permutation in der ursprünglichen Reihenfolge. Es werden also nicht alle $m!$ möglichen Permutationen, sondern gerade die Hälfte berücksichtigt.

3.5.3. Encodierung durch Mapping

Im Folgenden wird die in [DJ99] vorgeschlagene Methode zur Encodierung erläutert, bei der ausschließlich Golay-Sequenzen als Codewörter entstehen. Wie bereits in den beiden vorhergehenden Abschnitten gezeigt wurde, beinhaltet jede der $\frac{m!}{2}$ Golay-Nebenklassen 2^{m+1} Codewörter des $RM(1,m)$-Codes.

Es lassen sich somit $m+1$ Informationsbits direkt mit der Generatormatrix des $RM(1,m)$-Codes und der Methode nach Abschnitt 3.3.2 encodieren.

Da es nun noch $\frac{m!}{2}$ Möglichkeiten zur Auswahl der Nebenklasse gibt, bezieht man weitere Informationsbits in den Prozess der Encodierung mit ein. Es erscheint sinnvoll gerade $\left\lfloor \log_2\left(\frac{m!}{2}\right) \right\rfloor$ Informationsbits zusätzlich zu verwenden, um die Nebenklasse auszuwählen. Als *Mapping* wird in diesem Zusammenhang die Zuordnung der zusätzlichen Informationsbits zu den Nebenklassen bzw. deren Anführern bezeichnet.

Um das bereits $RM(1,m)$-codierte Codewort nun noch einer bestimmten Nebenklasse eindeutig zuzuordnen, wird der aufgrund des Mappings bestimmte Nebenklassenanführer hinzu addiert. Dadurch erfolgt implizit die eindeutige Zuordnung zu einer der Nebenklassen.

Es lässt sich damit die maximale Coderate für die Encodierung von ausschließlich Golay-Sequenzen erreichen. Dies ist in Abb. 3.2 rot dargestellt.

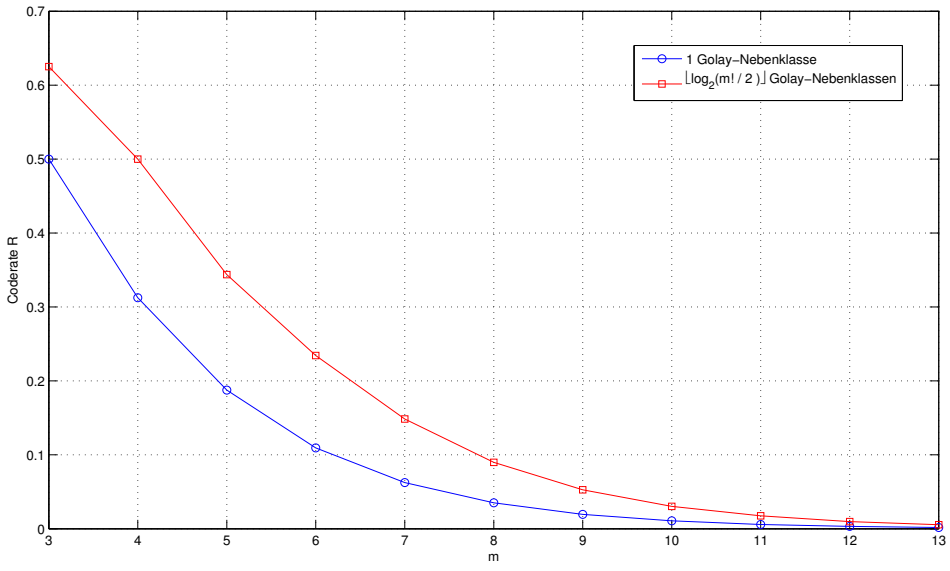

Abbildung 3.2.: Coderate für unterschiedliche Anzahl von Golay-Nebenklassen

Die blaue Kurve in Abb. 3.2 stellt hierbei die Coderate bei Verwendung nur einer Golay-Nebenklasse dar, was als untere Schranke interpretiert werden kann.

Da bei Verwendung nur einer Golay-Nebenklasse die Codewörter untereinander eine minimale Hammingdistanz gleich dem entsprechenden $RM(1,m)$-Code besitzen, lassen sich in diesem Fall $2^{m-1-1}-1 = 2^{m-2}-1$ Fehler korrigieren. Die blaue Kurve in Abb. 3.3 stellt dies dar. Die rote Kurve beschreibt die Anzahl an korrigierbaren Fehlern bei Verwendung von mehr als einer Golay-Nebenklasse. In diesem Fall besitzen die Codewörter nur noch die halbe minimale Hammingdistanz, da mehrere Nebenklassen des $RM(1,m)$-Codes in $RM(2,m)$ beteiligt sind und es sich somit prinzipiell um einen Reed-Muller Code zweiter Ordnung handelt.

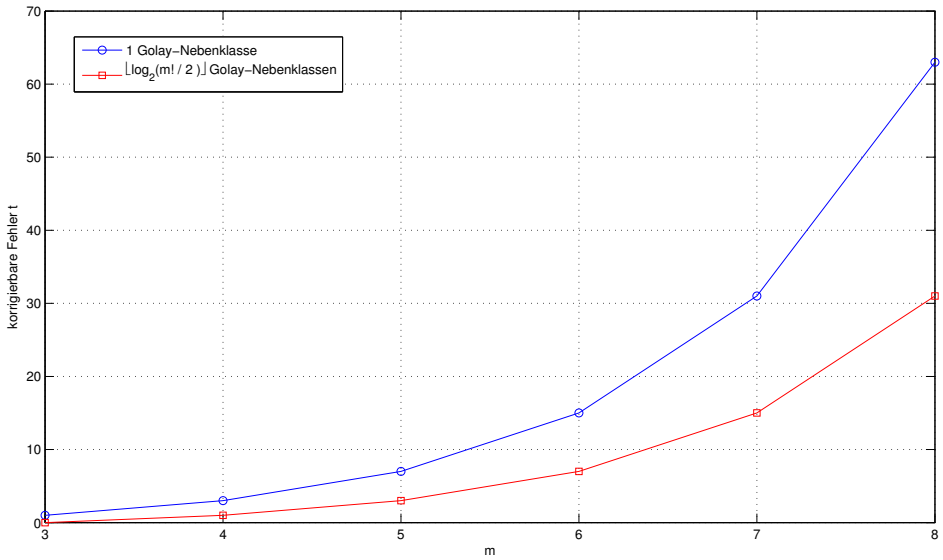

Abbildung 3.3.: Korrigierbare Fehler für unterschiedliche Anzahl von Golay-Nebenklassen

Die eigentliche Encodierung kann also flexibel bezogen auf die Anzahl der verwendeten Golay-Nebenklassen gestaltet werden. Die Kurven in Abb. 3.2 und 3.3 stellen somit die Grenzen dar, zwischen denen sich die Werte für die Encodierung bewegen können.

3.5.4. Hard- und Soft-Decision Decodierung

Im Folgenden wird eine Methode zur Hard-Decision Decodierung vorgestellt, welche sich leicht dahin gehend modifizieren lässt, als dass damit auch eine SD-Decodierung realisiert werden kann. Der vorgestellte Algorithmus basiert auf dem von Davis und Jedwab vorgeschlagenen Algorithmus [DJ99, Algorithm 17] zur Decodierung eines $RM(2,m)$-Codes, bei dem ausschließlich Golay-Nebenklassen als Codewörter erzeugt wurden. Der hier vorgestellte Algorithmus unterscheidet sich in soweit von dem in [DJ99] vorgestellten, als das hier nur der binäre Fall betrachtet wird. Für diesen Fall stellt der Algorithmus eine Methode zur Supercode Decodierung dar, welcher sich der *Fast-Hadamard-Transformation* (FHT) bedient.

Die Fast-Hadamard-Transformation

Die FHT stellt eine lineare, orthogonale Transformation dar, deren Ergebnis sich als eine Art Korrelation der Eingangssequenz mit den Zeilen der *Hadamard-Matrix* interpretieren lässt[9]. Es wird wird zunächst die Konstruktion einer Hadamard-Matrix nach Sylvester betrachtet.

Für die Hadamard-Matrix der nullten Ordnung gilt:

$$H(0) = [1]. \tag{3.21}$$

Die Hadamard-Matrix der Ordnung $n+1$ wird nach folgendem Schema rekursiv aus der Hadamard-Matrix der Ordnung n gewonnen:

$$H(n+1) = \begin{bmatrix} H(n) & H(n) \\ H(n) & -H(n) \end{bmatrix}. \tag{3.22}$$

Bei Verwendung dieser Konstruktion enthält die quadratische Hadamard-Matrix der Ordnung n genau 2^n Zeilen- bzw. Spaltenvektoren, welche den *Walsh-Funktionen* entsprechen. Aus diesem Grund wird die Transformation in diesem Fall auch als *Fast-Walsh-Hadamard-Transformation* (FWHT) bezeichnet. Neben der FWHT existieren noch weitere Varianten der Hadamard-Transformation, bei denen lediglich die Zeilen- bzw. Spaltenvektoren in ihrer Reihenfolge vertauscht sind. Das folgende Beispiel zeigt die Konstruktion der Hadamard-Matrizen der ersten und zweiten Ordnung nach Sylvester:

Beispiel: Die Hadamard-Matrix der ersten Ordnung ergibt sich zu:

$$H(1) = \begin{bmatrix} H(0) & H(0) \\ H(0) & -H(0) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Damit folgt für die Hadamard-Matrix der zweiten Ordnung:

$$H(2) = \begin{bmatrix} H(1) & H(1) \\ H(1) & -H(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Die Hadamard-Transformierte $\hat{y}$ der Eingangssequenz y ergibt sich zu:

$$\hat{y} = y \cdot H_n \qquad \text{mit} \quad H_n = \tfrac{1}{2^{\frac{m}{2}}} H(n) \ . \tag{3.23}$$

[9] Es findet keine Korrelation im Sinne der Definition statt. Die Elemente des Ergebnisvektors stellen Koeffizienten für die Gewichtung der einzelnen Zeilen dar, so dass die Summe aller Zeilen, gewichtet mit ihren Koeffizienten, der Eingangssequenz entspricht. Hat die Eingangssequenz große Ähnlichkeit mit einer Zeile, so wird auch der Wert für den entsprechenden Koeffizienten groß.

Die Hadamard-Transformation nach (3.23) benötigt zur Berechnung genau N^2 Schritte. Da sich die Hadamard-Matrizen der Ordnung n aber rekursiv aus Hadamard-Matrizen geringerer Ordnung ergibt, kann die Berechnung auf die Verknüpfung mehrerer Hadamard-Transformationen geringerer Ordnung, die weniger Schritte zur Berechnung benötigen, zurückgeführt werden. Somit ergibt sich, ähnlich zur Vereinfachung der DFT bei der FFT, eine Verringerung des Rechenaufwands, so dass bei der FHT im Vergleich zur Hadamard-Transformation letztendlich nur noch $N \cdot \mathrm{ld}(N)$ anstatt N^2 Schritte zur Berechnung nötig sind.

In Abb. 3.4 ist, wieder in Anlehnung an die Beziehung der FFT zur DFT, der *Butterfly-Graph* zur Berechnung der FHT dargestellt. Die schwarzen bzw. roten Pfeile stellen hierbei, entsprechend den Elementen der jeweiligen Hadamard-Matrizen, eine Multiplikation mit 1 bzw. 0 dar.

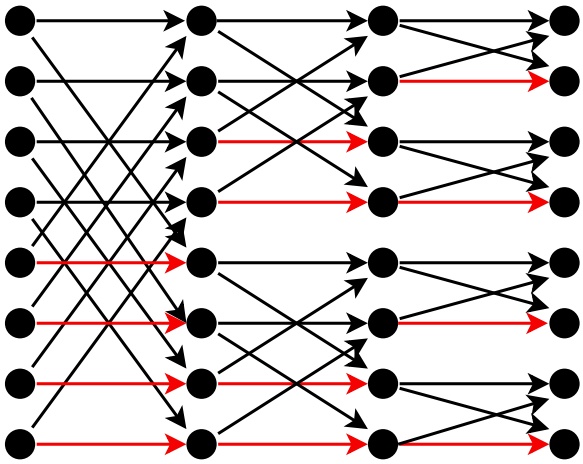

Abbildung 3.4.: Butterfly-Graph zur Berechnung der FHT

Algorithmus zur HD-Decodierung

Der im folgenden vorgestellte Algorithmus basiert, wie bereits erwähnt, auf der Supercode Decodiermethode nach [CS86]. Dabei wird durch das Abziehen der einzelnen Nebenklassenanführer vom Empfangsvektor implizit ein Vektor pro Nebenklasse erzeugt, welcher anschließend durch die FHT für den $RM(1, m)$-Codes decodiert wird.

Die FHT stellt, wie bereits erwähnt, eine Art Korrelation der Eingangssequenz mit den Zeilen der Hadamard-Matrix dar. Diese Zeilen stellen, aufgrund der Ähnlichkeit in der Konstruktion der Generatormatrix bei Reed-Muller Codes und den Hadamard-Matrizen, alle möglichen gültigen Codewörter des $RM(1, m)$-Codes dar[10]. Da die Elemente der Fast-Hadamard-Transformierten $\hat{y}$ eindeutig den Zeilen der zugehörigen Hadamard-Matrix, und damit auch allen gültigen Codewörtern des $RM(1, m)$-Codes zugeordnet werden

[10] Da die Elemente der Hadamard-Matrix nur die Werte $+/-1$ annehmen können, ist ein vorausgehendes Mapping der Bits des Empfangsvektors auf $+/-1$ erforderlich.

können, ergibt sich das am wahrscheinlichsten gesendete Codewort als jene Zeile der Hadamard-Matrix, welche dem Element der Fast-Hadamard-Transformierten mit dem größten Wert zugeordnet ist.

Dies gilt insbesondere auch für den Fall, dass ein $RM(2,m)$-Code vorliegt, von dem der jeweilige Nebenklassenanführer abgezogen wird, da dadurch implizit ein Codewort des $RM(1,m)$-Codes entsteht. Das am wahrscheinlichsten gesendete Codewort bei Betrachtung aller Nebenklassen, und damit aller modifizierten Empfangsvektoren, ergibt sich wiederum, indem man das Element mit maximalem Wert innerhalb aller Fast-Hadamard-Transformierten sucht, und der jeweiligen Zeilen innerhalb der Hadamard-Matrix zuordnet.

Es wird also jenes Codewort als das gesendete Codewort angenommen, welches die größte Ähnlichkeit mit dem Empfangsvektor aufweist. Dies entspricht gerade der Maximum-Likelihood Decodierung und ist damit optimal.

Im Folgenden ist nun der Algorithmus nach [DJ99, Algorithm 17] für den binären Fall gegeben:

(1) Sei r das empfangene Codewort der Länge 2^m mit Elementen aus $\mathbb{Z}_2$ und $\boldsymbol{G} = \{g\}$ die Menge aller Nebenklassenanführer von $RM(1,m)$. Setze $k = 0$, $r_0 = r$, $l = 1$ und $Y = 0$.

(2) Bilde die Sequenz y nach

$$(y)_i = \frac{1}{2} - wt_H(r_0 - g_l), \quad \text{für } i = 0, 1, \ldots, 2^m - 1,$$

wobei wt_H für das Hamminggewicht steht.

(3) Sei $\hat{y}$ die FHT von y. Finde jenes j, für welches der Betrag von $(\hat{y})_j$ den größten Wert annimmt.

(4) Wenn $|(\hat{y})_j| > |Y|$, dann setze $Y = (\hat{y})_j$, $J = j$ und $L = l$.

(5) Wenn (2) bis (4) für alle Nebenklassen ausgeführt wurde, gehe zu Schritt (6). Sonst erhöhe l um eins und gehe zu Schritt (2)

(6) Das erste Bit des decodierten Informationswortes ergibt sich zu $\text{sign}(Y)$, die folgenden Bits entsprechen der Binärdarstellung von J und L, wobei L über die letzten Bits entscheidet.

Das Zustandekommen des decodierten Informationswortes ist in Abb. 3.5 schematisch dargestellt. Das erste Bit ergibt sich in Abhängigkeit davon, ob der Maximalwert der FHT positiv oder negativ ist. Dies folgt anschaulich daraus, dass die erste Zeile der Generatormatrix ausschließlich Einsen enthält und damit eine Addition dieser Zeile zu einem

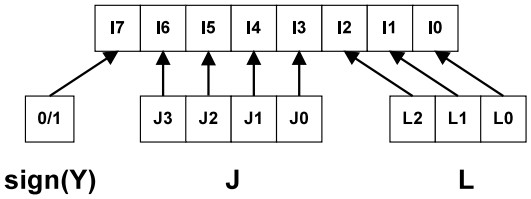

Abbildung 3.5.: Zusammensetzung des decodierten Informationswortes für $RM(2,3)$

Vektor in GF(2) auf den Vektor in negierter Form führt. Dies hat für den Maximalwert der FHT, da sie eine Art Korrelation darstellt, zur Folge, dass er ebenfalls negiert wird.

Die folgenden Bits, die dem Index des Elements maximalen Wertes der FHT in Binärdarstellung entsprechen, ergeben sich aufgrund des Aufbaus der Hadamard-Matrix gerade zu den gesuchten Informationsbits.

Die Binärdarstellung von L entspricht gerade deshalb den letzten Bits, weil bei der Encodierung reziprok vorgegangen wurde. Hier wurde die Dezimaldarstellung der letzten Bits zur Auswahl der Nebenklasse genutzt. Im Decoder entspricht L gerade der Nebenklasse, für die der Maximalwert der FHT auftrat. Somit ergibt die Binärdarstellung von L gerade die gesuchten Informationsbits.

SD-Decodierung

Wie bereits erwähnt lässt sich der vorhergehende Algorithmus leicht derart modifizieren, dass er zur SD-Decodierung verwendet werden kann. Hierzu wählt man die Eingangssequenz in Schritt (1) als die auf das Intervall $[0,1]$ projizierte Phase. Anschließend verwendet man in Schritt (2) ein geändertes Mapping für die Eingangssequenz r_0:

$$y = -1^{\frac{r_0 - g_l}{2}} \tag{3.24}$$

Damit erhält man für die Elemente von y komplexe Werte, welche anschließend wiederum der FHT zugeführt werden.

Der vorgestellte Algorithmus stellt eine optimale Methode zur Decodierung dar, benötigt jedoch bei einer großen Anzahl verwendeter Nebenklassen sehr viele Schritte für die Berechnung[11]. Abhängig von den Systemanforderungen kann in diesem Fall auf suboptimale Methoden zur Decodierung zurückgegriffen werden. Hierzu sei an dieser Stelle auf die Arbeit von Kenneth G. Paterson und Alan E. Jones [PJ00], sowie auf [SF05] verwiesen.

In diesem Kapitel wurden die binären Reed-Muller Codes eingeführt. Es wurde auf den Zusammenhang zwischen Reed-Muller Codes zweiter Ordnung und Golay-Sequenzen eingegangen, sowie Möglichkeiten zur En- und Decodierung vorgestellt.

[11] Für jede Nebenklasse folgt die Berechnung einer FHT.

Im folgenden Kapitel wird auf die vorgestellten Algorithmen zurückgegriffen, um eine praktische Implementierung der Reed-Muller Codes, sowohl allgemein als auch in Bezug auf Golay-Sequenzen, zu realisieren.

4. Implementierung in GNU Radio

Das Ergebnis dieser Arbeit stellt eine Implementierung der binären Reed-Muller Codes in GNU Radio dar. In diesem Kapitel wird auf einige grundlegende Ideen und Methoden eingegangen, welche bei der Realisierung verwendet wurden. Für eine vollständige Auflistung der entstandenen Dateien sowie der enthaltenen Funktionen sei auf A.2.1 verwiesen.

4.1. Speicherkonzept für Matrizen und Vektoren

Im letzten Kapitel wurde im Zusammenhang mit der Encodierung auf den Aufbau der Generatormatrizen eingegangen. Diese können für hohe Werte von m und r mitunter sehr groß werden, so dass eine Möglichkeit zur effizienten Speicherung gefunden werden muss.

Alle Elemente der Matrizen und Vektoren werden in der vorliegenden Implementierung bitweise im Speicher abgelegt. Dies hat den Vorteil, dass kein Speicherplatz verschwendet wird und ermöglicht außerdem eine sehr effiziente Berechnung bei der Verknüpfung von Vektoren.

Um die optimale Anpassung für verschiedenartige Systeme zu gewährleisten, wird der Datenyp `unsigned integer` für die Speicherung verwendet, da die Länge dieses Datentyps an das verwendete System angepasst ist[1]. Damit stellen Verknüpfungen sowie Manipulationen einzelner Vektoren bitweise Operationen dar, welche aufgrund der Anpassung von der CPU sehr effizient ausgeführt werden.

Da die Vektoren bzw. Zeilen der Generatormatrix durchaus größer als die zur Verfügung stehende Länge des Datentyps `integer` werden können, ist es unter Umständen notwendig Arrays zu verwenden. Für Matrizen liegt hierbei die Verwendung mehrdimensionaler Arrays nahe, deren Übergabe zwischen Funktionen für variable Dimensionen jedoch nicht ohne weiteres möglich ist. Aus diesem Grund wird in der vorliegenden Implementierung ein eigenes Konzept verwendet, welches implizit den internen Abläufen bei Verwendung mehrdimensionaler Arrays entspricht.

[1] Bei 64-Bit Systemen ist der Datentyp `integer` häufig nur 32 bit lang, da hier ein zusätzlicher Datentyp eingeführt wurde. Für 16-/32-Bit Systeme ist die Länge des Datentyps `integer` aber immer an die Bitbreite der CPU angepasst.

Zur Speicherung aller Matrizen und Vektoren werden eindimensionale Arrays verwendet, da sich diese leicht zwischen den einzelnen Funktionen übergeben lassen. Eine eindeutige Zuweisung der Größe erfolgt hierbei während der Laufzeit.

Die Größe der einzelnen Vektoren lässt sich leicht mit Hilfe der im letzten Kapitel eingeführten Formeln für beispielsweise Codewortlänge oder Anzahl codierter Informationsbits berechnen. Übersteigt die Anzahl der Elemente eines Vektors nun die in einem `integer` speicherbare Anzahl an Bits, so wird für den Vektor ein Array mit der entsprechenden Anzahl an Elementen initialisiert. Ist die Anzahl der Elemente eines Vektors kleiner als die in einem `integer` speicherbaren Bits, so werden lediglich die benötigten Bits genutzt und der Rest zu null gesetzt. Dadurch entsteht ein gewisser Overhead, welcher jedoch stark begrenzt ist, da die Gesamtzahl der zu speichernden Vektoren bzw. Zeilen der Generatormatrix für kleine Werte von m ohnehin sehr gering ist.

Die Anzahl der benötigten Array-Elemente n_{vec} ergibt sich für einen Vektor der Länge q in Bits somit zu:

$$n_{vec} = \left\lceil \frac{q}{\texttt{sizeof(integer)} \cdot 8} \right\rceil, \tag{4.1}$$

wobei mit dem Faktor 8 im Nenner multipliziert werden muss, da `sizeof()` die Länge eines Datentyps in Bytes zurück gibt.

Diese Art der Speicherung lässt sich leicht auch für Matrizen anwenden; hierbei wird eine Matrix als die Aneinanderreihung vieler Vektoren interpretiert, wobei jeder Vektor eine Zeile der Matrix darstellt. Damit folgt, unter Berücksichtigung des letzten Abschnitts, die Anzahl benötigter Elemente eines Arrays, welches eine pxq-Matrix beinhaltet zu:

$$n_{mat} = p \cdot n_{vec} = p \cdot \left\lceil \frac{q}{\texttt{sizeof(integer)} \cdot 8} \right\rceil. \tag{4.2}$$

Damit lässt sich die zweidimensionale pxq-Matrix in einem eindimensionalen Array mit n_{mat} Elementen speichern. Eine Zeile umfasst dabei immer n_{vec} aufeinander folgende Elemente.

4.1.1. Speicherzugriff und -manipulation

In C++ ist die kleinste adressierbare Speichereinheit ein Byte. Um nun auf einzelne Bits innerhalb eines Bytes oder `integer` zugreifen zu können, bedarf es also spezieller Methoden. In der vorliegenden Implementierung wurde dies durch Vergleiche und Verknüpfungen mit den entsprechenden Datentypen realisiert, bei denen lediglich einzelne Bits mit Werten besetzt sind, wobei die restlichen Bits zu Null gesetzt sind.

Um also den Wert eines einzelnen Bits innerhalb eines `integer` auslesen zu können, vergleicht man den `integer` mit einem, um die entsprechende Anzahl an Bits verschobenen Wert bitweise. Zur Manipulation wird analog dazu vorgegangen, indem der `integer` mit

einem um die entsprechende Anzahl an Bits verschobenen Wert verknüpft wird. Das Verschieben eines einzelnen Wertes lässt sich hierbei sehr effizient durch die `shift`-Operation realisieren.

Zur Änderung eines einzelnen Bits innerhalb eines Vektors oder einer Matrix muss nun nur noch jenes Element innerhalb des Arrays gefunden werden, indem sich das jeweilige Bit befindet. Hierzu muss lediglich durch die Anzahl der speicherbaren Bits pro `integer` geteilt werden und ggf. $n_{vec} \cdot i$ zur Auswahl der i-ten Zeile addiert werden.

In A.2.2 ist die Funktion zum Ändern eines Elements innerhalb einer Matrix gegeben.

Aufgrund des gewählten Speicherkonzepts lassen sich einzelne Vektoren sehr effizient miteinander verknüpfen. Hierbei werden die einzelnen `integer` eines Arrays (Vektors) elementweise durchgegangen, wobei die einzelnen Elemente der beiden zu verknüpfenden Vektoren bitweise verknüpft werden. In der CPU wird dieser Vorgang für alle Bits eines `integer` parallel innerhalb eines Taktschrittes ausgeführt.

Für die Verknüpfung zweier Vektoren sei beispielsweise die Funktion zur XOR-Verknüpfung in A.2.2 gegeben.

4.2. Binäre Reed-Muller Codes

Nachdem im letzten Abschnitt das Speicherkonzept, sowie Methoden zur Verknüpfung von Vektoren vorgestellt wurden, kann nun auf die eigentliche Implementierung der binären Reed-Muller Codes eingegangen werden. Hierzu wird zunächst die verwendete Methode zur Erzeugung der Generatormatrix vorgestellt.

4.2.1. Bilden der Generatormatrix

Die erste Zeile der Generatormatrix eines $RM(r, m)$-Codes ergibt sich direkt als die Einsfolge der Länge 2^m. Die folgenden m Zeilen stellen Boolesche Funktionen der ersten Ordnung dar und entstehen prinzipiell dadurch, dass man für die Zeile i alternierend 1- bzw. 0-Folgen, beginnend mit der 1-Folge, der Länge 2^{m-i} solange aneinander reiht, bis schließlich 2^m Elemente erzeugt wurden. In der Implementierung wurde dies derart realisiert, dass die i-te Zeile der Generatormatrix zunächst in 2^{i-1} gleich große Abschnitte unterteilt wird. Anschließend werden die entstandenen Abschnitte mittels einer Schleife der Reihe nach durchgegangen und, sofern die Modulo-2-Operation des Index des aktuellen Abschnitts einen Wert ungleich null liefert, alle zugehörigen Elemente zu eins gesetzt.

Zur Erzeugung der folgenden Zeilen werden diese zunächst zu eins gesetzt, da sie durch UND-Verknüpfungen aus den m Monomen der ersten Ordnung entstehen. Um nun die Muster der Linearkombinationen zu erhalten bedient man sich der Binärdarstellung der

Zahlen $0 \dots 2^m - 1$, da genau 2^m Möglichkeiten für die Linearkombination von m Elementen existieren. Hierbei wird jeder Stelle innerhalb der Binärdarstellung ein Monom der ersten Ordnung zugewiesen. In Tabelle 4.1 wurde das Zustandekommen der Muster zur Erzeugung der Linearkombinationen für $m = 4$ veranschaulicht.

Dezimal	Binär	entspr. Monom
0	0000	$\mathbf{1}$
1	0001	x_1
2	0010	x_2
3	0011	x_1x_2
4	0100	x_3
5	0101	x_1x_3
6	0110	x_2x_3
7	0111	$x_1x_2x_3$
8	1000	x_4
9	1001	x_1x_4
10	1010	x_2x_4
11	1011	$x_1x_2x_4$
12	1100	x_3x_4
13	1101	$x_1x_3x_4$
14	1110	$x_2x_3x_4$
15	1111	$x_1x_2x_3x_4$

Tabelle 4.1.: Linearkombinationen der Monome erster Ordnung für $m = 4$

Es lässt sich leicht erkennen, dass die Anzahl an Einsen der Ordnung entspricht. Um nun die Generatormatrix des $RM(r, m)$-Codes zu erzeugen, müssen die Ordnungen, und damit die Anzahl an Einsen innerhalb der Muster, beginnend mit 2 der Reihe nach durchgegangen werden bis schließlich r erreicht wurde[2].

Da die Zeilen innerhalb einer Ordnung allerdings immer noch in ihrer Reihenfolge vertauscht sind, bedarf es einer Maßnahme zur Umsortierung. Hierfür führt man ein Abbruchkriterium ein, welches sicherstellt, dass innerhalb einer Ordnung zuerst alle Monome, bei denen x_1 beteiligt ist, berücksichtigt werden. Danach werden alle Monome betrachtet, an denen x_2 beteiligt ist, wobei x_1 jedoch nicht beteiligt sein darf. Dies lässt sich anhand der Binärdarstellung dadurch beschreiben, dass alle Linearkombinationen mit der entsprechenden Anzahl an Einsen und einer Eins an der Stelle i betrachtet werden, wobei die Stellen $i + 1$ bis m alle den Wert 0 aufweisen müssen. Als Startwert für i wird die letzte Stelle, also m gewählt, welche dem Monom x_1 zugeordnet ist[3].

[2] Die ersten $m + 1$ Zeilen der Generatormatrix wurden bereits erzeugt.

[3] Nach dieser Vorgehensweise treten auch die restlichen, am Aufbau beteiligten, Monome der ersten Ordnung in der richtigen Reihenfolge auf. Dies folgt daraus, dass die Koeffizienten der Monome höherer Indizes weiter links stehen und somit, aufgrund der binären Zählweise, an den ihnen zugeordneten Stellen erst später eine 1 auftritt als bei Monomen geringerer Indizes.

Damit, und unter Berücksichtigung der verschiedenen Ordnungen, ergibt sich die Reihenfolge nach Tabelle 4.2.

Dezimal	Binär	entspr. Monom
0	0000	$\mathbf{1}$
1	0001	x_1
2	0010	x_2
4	0100	x_3
8	1000	x_4
3	0011	x_1x_2
5	0101	x_1x_3
9	1001	x_1x_4
6	0110	x_2x_3
10	1010	x_2x_4
12	1100	x_3x_4
7	0111	$x_1x_2x_3$
11	1011	$x_1x_2x_4$
13	1101	$x_1x_3x_4$
14	1110	$x_2x_3x_4$
15	1111	$x_1x_2x_3x_4$

Tabelle 4.2.: Geordnete Linearkombinationen der Monome erster Ordnung für $m = 4$

Stehen die Muster der Linearkombinationen in der richtigen Reihenfolge zur Verfügung, so lassen sich die restlichen Zeilen der Generatormatrix einfach durch UND-Verknüpfungen der entsprechenden Zeilen der ersten Ordnung erzeugen.

Die Funktion zur Erzeugung der Generatormatrix ist in A.2.2 gegeben.

4.2.2. Encodierung

Die Encodierung gestaltet sich mit vorhandener Generatormatrix sehr einfach. Hierzu wird das Datenwort elementweise durchgegangen, wobei die einzelnen Stellen die Koeffizienten für die jeweiligen Zeilen der Generatormatrix darstellen. Tritt eine Eins an der Stelle i im Datenwort auf, so wird die i-te Zeile der Generatormatrix mit dem temporären Codewort XOR-verknüpft und tritt an dessen Stelle.

In A.2.2 ist die Funktion zur Encodierung gegeben.

4.2.3. Decodierung mittels Mehrheitsentscheidung

Zur Decodierung wird der in 3.3.3 vorgestellte Algorithmus nach Reed verwendet. Da hierbei Wert auf die Reihenfolge der Zeilen der Generatormatrix gelegt werden muss,

wird eine Sortierung der Zeilen analog zu 4.2.1 vorgenommen. Dies hat außerdem den Vorteil, dass für jede Zeile die am Aufbau beteiligten Monome der ersten Ordnung direkt zur Verfügung stehen. Diese werden für die Erzeugung der charakteristischen Vektoren zwingend benötigt, da sie aus den Linearkombinationen der nicht beteiligten Monome entstehen. Um nun alle möglichen Linearkombinationen der (negierten) Monome nach 3.3.3 zu erhalten, bedarf es wiederum einer Binärdarstellung ähnlich dem Vorgehen in 4.2.1. Die Anzahl an Stellen in der Binärdarstellung wird hierbei durch die Anzahl der am Aufbau der jeweiligen Zeil nicht beteiligten Monome bestimmt. Die Muster der Linearkombinationen ergeben sich nun dadurch, dass eine dem Monom i zugeordnete 0 derart interpretiert wird, dass das Monom in seiner negierten Version am Aufbau des charakteristischen Vektors beteiligt ist. Analog dazu steht eine 1 dafür, dass das Monom in seiner eigentlichen Form am Aufbau beteiligt ist.

Damit stehen nun die charakteristischen Vektoren aller Zeilen zur Verfügung und es kann direkt das Skalarprodukt mit dem (temporären) Empfangsvektor gebildet werden. Nach der anschließenden Mehrheitsentscheidung, die die Koeffizienten und damit die decodierten Informationsbits liefert, folgt, nachdem alle Zeilen einer Ordnung decodiert wurden, die Addition zum (temporären) Empfangsvektor. Anschließend der Vorgang wird für die nächstkleinere Ordnung wiederholt.

Der Koeffizient, welcher der ersten Zeile der Generatormatrix zugeordnet ist, ergibt sich direkt aus der Mehrheit der Bits des temporären Empfangsvektors, nachdem der Algorithmus für alle anderen Zeilen durchlaufen wurde.

In A.2.2 ist die Realisierung des Decoders nach Reed gegeben.

4.3. Reed-Muller Codes zur Erzeugung von Golay-Sequenzen

Im Hinblick auf die praktische Anwendung der Implementierung in OFDM-basierten Systemen wurden die in Kapitel 3 eingeführten Methoden zur En- und Decodierung bei ausschließlicher Verwendung von Golay-Sequenzen als Codewörter ebenfalls realisiert.

Im Folgenden wird zunächst auf die Identifikation der Golay-Nebenklassen und deren Speicherung in einer Precodematrix eingegangen, bevor schließlich die verwendeten Methoden zur En- und Decodierung vorgestellt werden.

4.3.1. Identifikation der Golay-Nebenklassen

Wie im letzten Kapitel gezeigt wurde, beinhalten die $\frac{m!}{2}$ Nebenklassen des $RM(1,m)$-Codes innerhalb des $RM(2,m)$-Codes der Form

$$\sum_{k=1}^{m-1} x_{\Pi(k)} x_{\Pi(k+1)} \tag{4.3}$$

ausschließlich Golay-Sequenzen, wobei Π für die Permutationen der Symbole $\{1, 2, \ldots, m\}$ steht.

Wie bereits erwähnt, tritt der Faktor $\frac{1}{2}$ in $\frac{m!}{2}$ deshalb auf, da eine Vertauschung der Symbol-Reihenfolge innerhalb einer Permutation das selbe Ergebnis liefert. Es gilt also jene Permutationen auszuschließen, die in der umgekehrten Reihenfolge bereits berücksichtigt wurden.

In der Implementierung wurde dies dadurch gelöst, dass für gültige, d.h. zu berücksichtigende, Permutationen die Bedingung

$$\Pi(m) > \Pi(1) \tag{4.4}$$

erfüllt sein muss. Das Permutationen mit vertauschter Reihenfolge durch die Bedingung (4.4) ausgeschlossen werden, lässt sich leicht anhand des folgenden Beispiels nachvollziehen.

Beispiel: Auswahl der Permutationen für $m = 4$:

Für $m = 4$ steht für die Permutationen die Menge der Symbole $\{1, 2, 3, 4\}$ zur Verfügung.

Man betrachte nun zunächst alle Permutationen mit dem Element 1 an der ersten Stelle. Für sie ist die Bedingung (4.4) immer erfüllt, da 1 das kleinste Element darstellt. Damit ergeben sich die folgenden Permutationen:

$$\{1, 2, 3, 4\}, \{1, 2, 4, 3\}, \{1, 3, 2, 4\}, \{1, 3, 4, 2\}, \{1, 4, 2, 3\}, \{1, 4, 3, 2\}.$$

Damit diese Permutationen nicht in vertauschter Reihenfolge auftreten, ist es hinreichend das Symbol 1 als letztes Element auszuschließen.

Es werden nun die Permutation mit dem Element 2 an erster Stelle betrachtet, wobei keine 1 als letztes Element auftreten darf. Damit ergeben sich die folgenden Permutationen:

$$\{2, 1, 3, 4\}, \{2, 1, 4, 3\}, \{2, 3, 1, 4\}, \{2, 4, 1, 3\}.$$

Es ergibt sich wiederum, dass für alle folgenden Permutationen das letzte Element weder eine 1 noch eine 2 sein darf.

Für die Permutationen mit einer 3 an erster Stelle folgt damit:

$$\{3, 1, 2, 4\}, \{3, 2, 1, 4\}.$$

Damit ergibt sich, dass für eventuelle weitere Permutationen auch keine 3 mehr als letztes Element auftreten dürfte. Da aber das Symbol 4 nicht gleichzeitig am Anfang und am Ende auftreten kann, können keine weiteren Permutationen gefunden werden.

Aus der Bedingung (4.4) folgt analog dazu, dass bei der Wahl des Symbols 4 als erstes Element keine gültigen Permutationen gefunden werden können, da das Symbol 4 bereits das größte Element darstellt.

Wurden alle Permutationen gefunden, so werden diese derart ausgewertet, dass immer zwei aufeinander folgende Symbole als eine Boolesche Funktion der Ordnung 2 interpretiert werden. Sie lassen sich damit den Zeilen der zweiten Ordnung innerhalb der Generatormatrix zuordnen.

Zur Speicherung der Golay-Nebenklassen in der Precodematrix werden pro Nebenklasse $\binom{m}{2}$ Bits verwendet. Dies entspricht genau der Anzahl an Zeilen innerhalb der Generatormatrix, welche der zweiten Ordnung entsprechen. Ist eine Zeile am Aufbau des Nebenklassenanführers beteiligt[4], so wird an der entsprechenden Stelle eine 1 gespeichert, andernfalls eine 0.

In A.2.2 ist der Quelltext zur Erzeugung der Precode-Matrix gegeben.

4.3.2. Encodierung

Die verwendete Methode zur Encodierung gestaltet sich bei vorhandener Precodematrix und vorhandener Reed-Muller Encodierung sehr einfach. Hierzu wählt man nach 3.5.3 bis zu $\left\lfloor \log_2 \left(\frac{m!}{2}\right) \right\rfloor$ Informationsbits zusätzlich zu den, mit dem $RM(1,m)$-Code zu codierenden Informationsbits aus, welche zur Auswahl der Golay-Nebenklasse dienen.

Die Auswahl erfolgt dabei derart, dass die entsprechende Anzahl an Informationsbits am Ende des Informationswortes als Binärdarstellung einer Dezimalzahl interpretiert werden. Diese Dezimalzahl gibt die zu verwendende Golay-Nebenklasse an, indem sie als Zeilenangabe für die in der Precodematrix gespeicherten Bitmuster verwendet wird.

Die in der Precodematrix gespeicherten Bitmuster entsprechen gerade den Koeffizienten für jene Zeilen der Generatormatrix, welche der Ordnung 2 zugeordnet sind. Durch die Verknüpfung dieser Zeilen mit den, in der Precodematrix gespeicherten, Koeffizienten ergeben sich direkt die Anführer der Golay-Nebenklassen. Durch die Addition eines Nebenklassenanführers wird nach 3.5.3 ein in $RM(1,m)$-codiertes Codewort einer bestimmten Nebenklasse eindeutig zugeordnet[5].

[4] Eine Zeile der zweiten Ordnung ist am Aufbau des Nebenklassenanführers beteiligt, sofern zwei aufeinander folgende Symbole innerhalb einer Permutation den, am Aufbau der jeweiligen Zeile beteiligten Monomen erster Ordnung, entsprechen.

[5] Eine zusätzliche Encodierung der Bitmuster für den $RM(2,m)$-Code impliziert die Addition des entsprechenden Nebenklassenanführers.

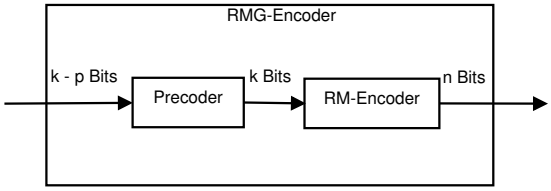

Abbildung 4.1.: RM-Encoder zur Erzeugung von Golay-Sequenzen

Damit ergibt sich der gesamte Prozess der Encodierung wie in Abb. 4.1 dargestellt als eine einfache $RM(2, m)$-Codierung der bereits precodierten Informationsbits. Der Precoder führt hierbei ein Mapping der entsprechenden Anzahl Informationsbits am Ende des Informationswortes auf die, in der Precodematrix gespeicherten, Bitmuster aus.

Die Funktion zur Encodierung ist ebenfalls in A.2.2 gegeben, wobei hier aus Performance-Gründen noch die Möglichkeit zur Wahl der Anzahl zu verwendender Golay-Nebenklassen implementiert wurde. Zur Encodierung lassen sich somit zwischen einer und $2^{\lfloor \log_2(\frac{m!}{2}) \rfloor}$ der insgesamt $\frac{m!}{2}$ verfügbaren Golay-Nebenklassen verwenden, wobei die Anzahl verwendeter Nebenklassen immer einer Potenz von 2 entsprechen muss[6].

4.3.3. Decodierung mittels FHT

Bei der Realisierung der Decodierung mittels der Fast-Hadamard-Transformation gilt es zu beachten, dass die in 4.3.1 eingeführte Precodematrix nicht die eigentlichen Nebenklassenanführer, sondern nur entsprechende Bitmuster zu deren Erzeugung mittels des $RM(2, m)$-Encoders enthält.

Die Nebenklassenanführer müssen somit für jede der möglichen Nebenklassen berechnet werden, bevor sie schließlich vom Empfangsvektor subtrahiert werden können.

Des weiteren müssen die Elemente der an die FHT zu übergebende Sequenz y vom Datentyp `float` sein, da die einzelnen, ganzzahligen Werte durch die FHT in den Raum den reellen Zahlen, $\mathbb{R}$, überführt werden. Für den Fall der SD-Decodierung entstehen für die Elemente der Sequenz y unter anderem auch komplexe Werte, weshalb hier der Datentyp `complex<float>` eingeführt werden muss.

Die FHT selbst lässt sich für beide Fälle relativ einfach durch drei verschachtelte `for`-Schleifen realisieren. Die Funktion zur Berechnung der FHT bei der HD-Decodierung ist in A.2.2 gegeben.

Wurde der Maximalwert des Betrags der FHT, sowie die Nebenklasse, für die der Wert auftrat, gefunden, so lässt sich nach 3.5.4 leicht das decodierte Informationswort bilden.

[6] Da zur Auswahl einer Golay-Nebenklasse Bits des Informationswortes als Binärdarstellung einer Dezimalzahl interpretiert werden, sind nur Potenzen von 2 für die Anzahl der zu verwendenden Golay-Nebenklassen zulässig.

Da bei der Encodierung die Möglichkeit zur Wahl der Anzahl zu verwendender Golay-Nebenklassen implementiert wurde, muss dies auch bei der Decodierung berücksichtigt werden. Da sowohl die Subtraktion des Nebenklassenanführers vom Empfangsvektor als auch die FHT für jede der verwendeten Nebenklassen einzeln berechnet werden muss, ergibt sich für eine große Anzahl verwendeter Golay-Nebenklassen mitunter ein sehr hoher Rechenaufwand. Dieser lässt sich durch die Beschränkung der zu verwendenden Nebenklassen stark reduzieren lässt.

In A.2.2 ist die Funktion zur HD-Decodierung bei Verwendung von Golay-Sequenzen gegeben.

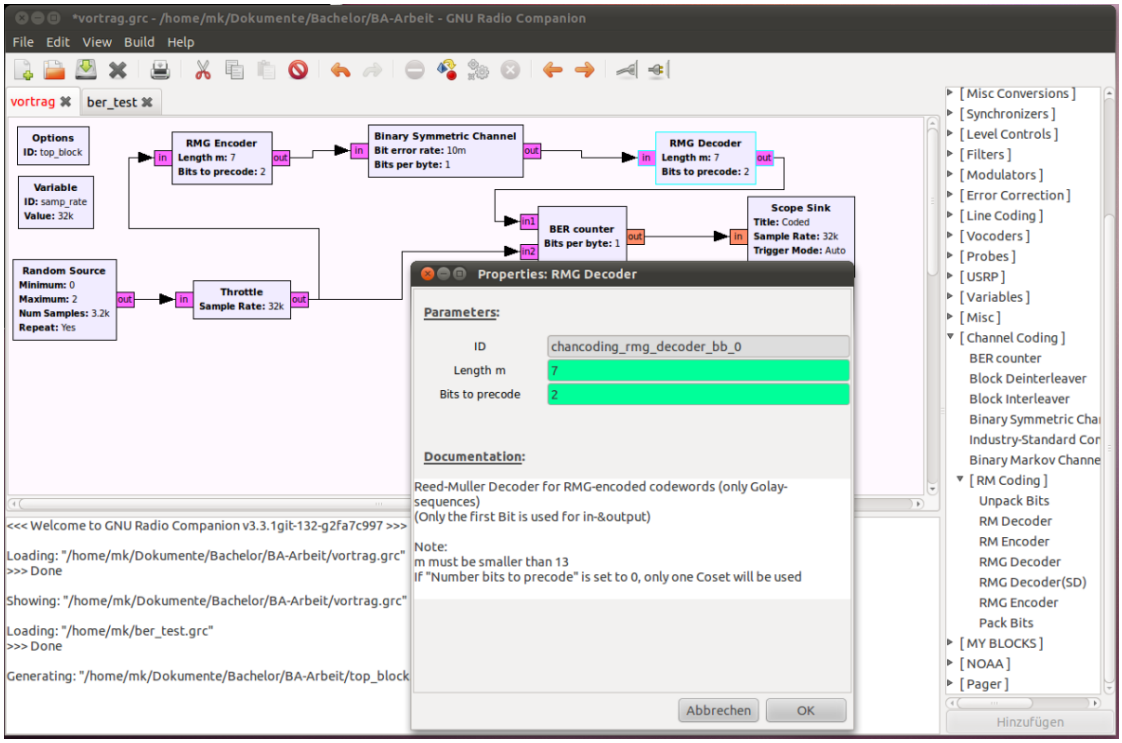

Abbildung 4.2.: Screenshot des GNU Radio Companion mit implementierter RM-Codierung

Abb. 4.2 zeigt rechts unter dem Punkt „RM-Coding“ die implementierten Blöcke innerhalb der „Channel Coding Toolbox“, welche mit GRC-Bindings versehenen wurden.

Etwa im Zentrum der Abbildung ist ein Fenster zur Bearbeitung der Parameter des RM-Decoders bei Verwendung von Golay-Sequenzen (RMG-Decoder) gegeben. Im Hintergrund lässt sich ein Aufbau zur BER-Messung einer Übertragung über einen *Binary Symmetric Channel* (BSC) erkennen, bei dem ein $RM(2,7)$-Code unter Verwendung von $2^2 = 4$ Golay-Nebenklassen verwendet wird.

In diesem Kapitel wurden einige, in der Implementierung verwendete, Methoden zur Realisierung vorgestellt. Im folgenden Kapitel wird die Performance der Implementierung untersucht, sowie auf vorhandene Beschränkungen eingegangen.

5. Performance-Analyse und praktische Anwendung

In diesem Kapitel wird zunächst die Leistungsfähigkeit der Implementierung hinsichtlich erzielbarer Geschwindigkeit und erforderlichem Speicher untersucht. In diesem Zusammenhang wird auf die Grenzen der Implementierung eingegangen, wobei eine Begründungen für deren Existenz gegeben wird.

Abschließend erfolgt eine kurze Betrachtung der Möglichkeiten zur praktischen Anwendung, wobei besonderer Wert auf die Anwendung in OFDM-basierten Systemen gelegt wird.

5.1. Performance der Implementierung

Im folgenden wird die erzielbare Performance der Implementierung betrachtet. Hierbei wird zunächst auf die Initialisierung eingegangen. Der Prozess der Initialisierung ist für die Performance der eigentlichen Übertragung nicht von Bedeutung, da sie vor deren Beginn ausgeführt werden kann.

Wie gezeigt wird, stellt die Decodierung die kritische Komponente hinsichtlich der erreichbaren Performance dar, weshalb in diesem Zusammenhang auch auf die erzielbare Datenraten eingegangen wird.

Insgesamt gilt für die folgenden Betrachtungen hinsichtlich der Performance, dass sie stark vom verwendeten System abhängig sind, weshalb die gewonnen Werte als Anhaltspunkte zur Abschätzung der, mittels der Implementierung, erzielbaren Leistungsfähigkeit interpretiert werden sollten.

Für alle Messungen wurde das folgende System verwendet:

- Dell Inspiron 1720
- Intel Core2 Duo T7500 @2,2 GHz
- 2 GB RAM
- Ubuntu 10.10 (64 bit)
- GNU Radio 3.3

5.1.1. Initialisierung

Bei der Initialisierung werden, je nach verwendetem Block, die Generator- und die Precodematrix gebildet, sowie im Speicher abgelegt. Bei den verwendeten Methoden zum Bilden dieser Matrizen gilt es die Grenzen der Implementierung zu beachten, welche im folgenden vorgestellt werden.

Für die Erzeugung der Bitmuster zum Bilden der Linearkombinationen wird jeweils ein Bit des Datentyps `integer` einem Monom der ersten Ordnung zugeordnet (siehe Abb. 5.1). Dadurch ist die mögliche Anzahl der Monome erster Ordnung ($= m$) auf die Länge des Datentyps `integer` in Bit beschränkt. Für praktische Systeme stellt diese Beschränkung jedoch kein Problem dar, da für sehr große Werte von m die erzeugten Codewörter sehr lange werden, was weitere Nachteile zur Folge hat.

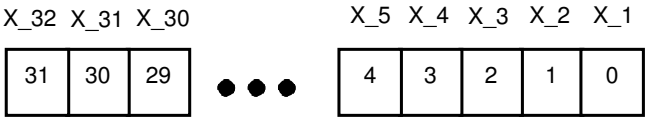

Abbildung 5.1.: Zuordnung der einzelnen Bits zu den Monomen erster Ordnung

Für die Verwendung von RM-Codes zur Erzeugung von Golay-Sequenzen gilt es eine weitere Beschränkung zu beachten; hierbei müssen alle zu verwendenden Golay- Nebenklassen in der Precodematrix gespeichert werden. Die Anzahl der möglichen Nebenklassen $\frac{m!}{2}$ steigt jedoch mit wachsendem m stark an. Da jede der Nebenklassen eine Zeile der Precodematrix, und damit ein Element eines Arrays, darstellt, muss deren Anzahl beschränkt sein, da jedem Element ein Index (welcher wiederum ein Datentyp darstellt) zugeordnet werden muss. In der vorliegenden Implementierung ist daher die Anzahl verwendbarer Golay-Nebenklassen auf den mit dem Datentyp `unsigned long` maximal erreichbaren Wert beschränkt.

In den GRC-Bindings wurde diese Beschränkung derart berücksichtigt, dass m auf 13 beschränkt wurde[1]. Dies ermöglicht für jede zulässige Wahl von m die Verwendung von $\left\lfloor \log_2 \left(\frac{m!}{2}\right) \right\rfloor$ Bits zur Auswahl der Nebenklasse.

Nachdem auf die Beschränkungen der Implementierung hingewiesen wurde, können nun die Betrachtungen zum Speicherbedarf der Matrizen folgen.

Für die klassischen $RM(r, m)$-Codes folgt ein Speicherbedarf von

$$\sum_{i=0}^{r} \binom{m}{i} \cdot 2^m \text{ Bits} \tag{5.1}$$

[1] Werden weniger als $2^{\left\lfloor \log_2 \left(\frac{m!}{2}\right) \right\rfloor}$ Golay-Nebenklassen verwendet, so lassen sich mit Hilfe der nichthierarchischen Blöcke auch Systeme mit $m > 13$ realisieren. Die Anzahl der zu verwendenden Nebenklassen muss jedoch immer kleiner als der größte, mit dem Datentyp `unsigned long` erreichbare, Wert bleiben.

für die Generatormatrix. Die Summe der Binomialkoeffizienten stellt hierbei die Anzahl der Zeilen dar, die schließlich noch mit der Anzahl an Bits pro Zeile (2^m) multipliziert werden muss.

In Abb. 5.2 ist unter anderem der Speicherbedarf eines $RM(1,m)$-Codes dargestellt. Er kann als untere Schranke für den Speicherbedarf eines $RM(r,m)$-Codes interpretiert werden. Analog dazu kann der in Abb 5.2 dargestellte Speicherbedarf eines $RM(m,m)$-Codes als obere Schranke für einen $RM(r,m)$-Code interpretiert werden, da die Generatormatrix hierbei die maximale Anzahl an Zeilen enthält.

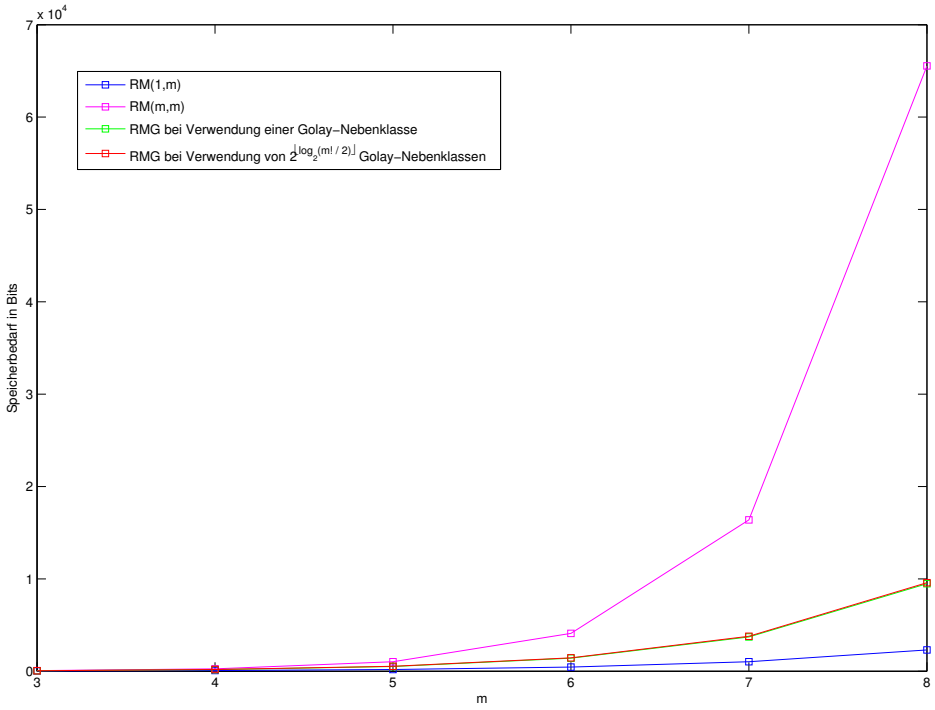

Abbildung 5.2.: Speicherbedarf für Generator- und Precodematrix

Bei Verwendung eines RM-Codes zur Erzeugung von Golay-Sequenzen muss neben der Generatormatrix der Ordnung $r = 2$ noch die Precodematrix gespeichert werden. Damit ergibt sich in diesem Fall ein Speicherbedarf von

$$\sum_{i=0}^{2} \binom{m}{i} \cdot 2^m + m \cdot n_{NK} \text{ Bits}, \tag{5.2}$$

wobei n_{NK} die Anzahl der zu verwendenden Golay-Nebenklassen beschreibt. Auch in diesem Fall sind in Abb. 5.2 zwei Kurven dargestellt, die wiederum als obere und untere Schranke für den Speicherbedarf interpretiert werden können. Wie sich erkennen lässt, beeinflusst die Anzahl der zu verwendenden Nebenklassen den gesamten Speicherbedarf nur sehr geringfügig.

5.1.2. Encodierung

Zur Ermittlung der Performance wurde der in Abb. 5.3 dargestellte Aufbau in GNU Radio Companion verwendet. Die gewonnenen Messwerte für Dauer und CPU-Auslastung wurden mit Hilfe des Befehls `top` ermittelt.

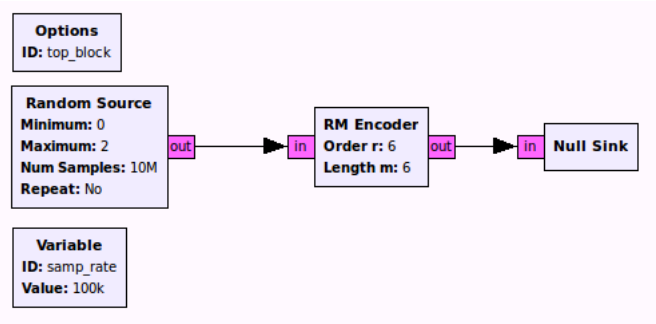

Abbildung 5.3.: Aufbau zur Ermittlung der Performance

Hinweis: Die verwendete Methode ist einigen Einflüssen unterworfen, die es bei der Interpretation der Ergebnisse zu berücksichtigen gilt.

- Die Bits werden zufällig erzeugt: Die Anzahl benötigter Taktzyklen zur Berechnung ist von den auftretenden Bits abhängig.
- Zusätzliche Blöcke: Auch die zusätzlich verwendeten Blöcke belegen Ressourcen des Systems.
- Aktivität der zweiten CPU: Die zweite CPU wird nur teilweise zur Berechnung verwendet.
- Prozesse im Hintergrund: Auch das Betriebssystem und laufende Anwendungen belegen System-Ressourcen.
- Hierarchische Blöcke: Es wurden hierarchische Blöcke verwendet, dadurch erfolgt am Ein- und Ausgang zusätzlich zur eigentlichen Codierung ein Bit-(Un-)Packing.

Da die Einflüsse teilweise aber auch bei der praktischen Anwendung auftreten, können die ermittelten Werte durchaus zu einer realistischen Einschätzung der Leistungsfähigkeit der Implementierung dienen.

Die hier erwähnten Einflüsse gilt es bei allen folgenden Betrachtungen zu berücksichtigen.

Zur Ermittlung der durchschnittlichen Dauer zur Encodierung eines Informationsbits wurden 100 Millionen Bits zufällig erzeugt und dem entsprechenden Encoder zugeführt. Die Ergebnisse sind in Abb. 5.4 gegeben.

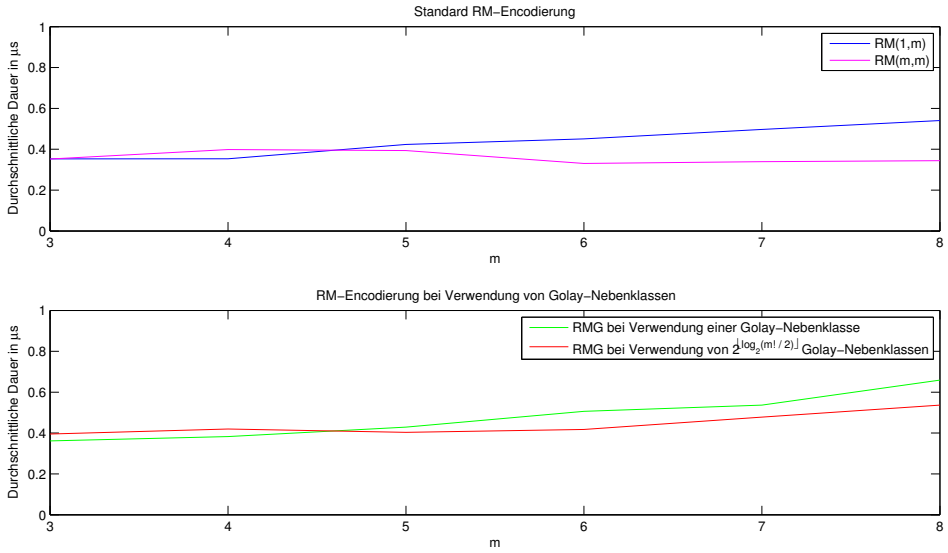

Abbildung 5.4.: Durchschnittliche Dauer zur Encodierung eines Informationsbits

Es lässt sich erkennen, dass alle implementierten Encoder ähnlich schnell bei der Encodierung eines Informationsbits sind, wobei die Dauer mit steigendem m geringfügig zunimmt. Da sich die Decodierung wesentlich aufwendiger als die Encodierung gestaltet und somit die kritische Komponente bei der praktischen Anwendung darstellt, wird hier nicht näher auf die erzielbaren Datenraten eingegangen, da diese deutlich über den bei der Decodierung erreichbaren Datenraten liegen.

Neben der erzielbaren Datenrate ist außerdem noch die durch die Encodierung verursachte CPU-Auslastung von Interesse, da bei der praktischen Anwendung neben der Kanalcodierung noch weitere Komponenten, beispielsweise zur Modulation, verwendet werden.

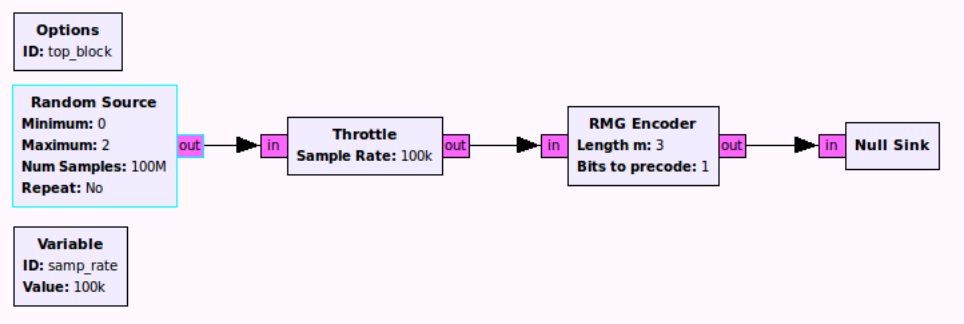

Abbildung 5.5.: Aufbau zur Ermittlung der CPU-Auslastung

Die ermittelten Werte für die CPU-Auslastung sind in Abb. 5.6 dargestellt. Hierbei wurde ein Aufbau nach Abb. 5.5 mit einer Netto-Datenrate von 100 kbit/s verwendet.

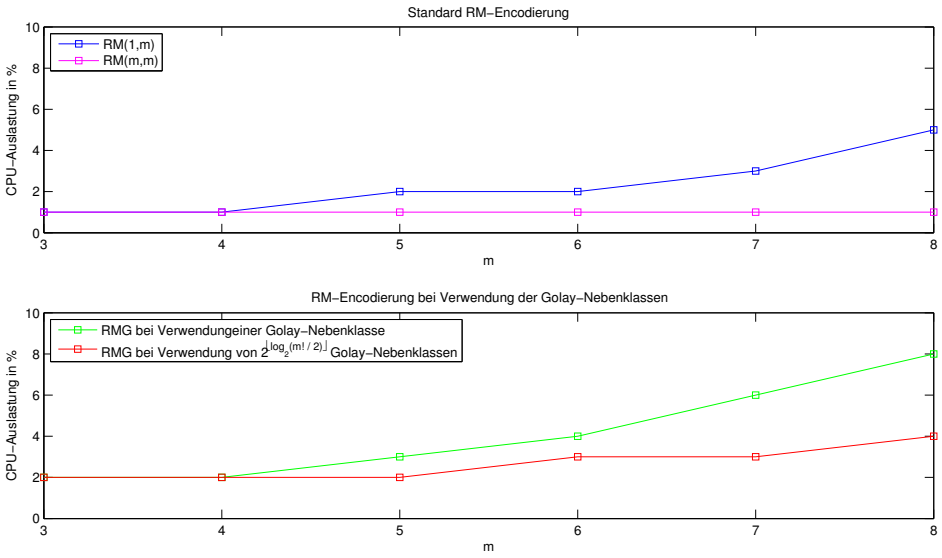

Abbildung 5.6.: Durch die Encodierung verursachte CPU-Auslastung bei einer Netto-Datenrate von 100 kbit/s

Wie sich erkennen lässt, liegt bei der verwendeten Datenrate die CPU-Auslastung für alle getesteten Encoder unter 10%. Für diese Datenrate sind also noch genügend Kapazitäten für weitere Komponenten vorhanden.

Die Performance der Encodierung bei Verwendung von Golay-Sequenzen lässt sich noch geringfügig steigern, indem in der Precodematrix direkt die Nebenklassenanführer anstatt Bitmuster zu deren Erzeugung gespeichert werden. Dadurch verringert sich der Aufwand, da der jeweilige Nebenklassenanführer nicht während der Encodierung berechnet werden muss.

5.1.3. Decodierung

Wie bereits erwähnt, stellt die Decodierung die kritische Komponente bei der praktischen Anwendung dar. Auch hier wurden Daten zur durchschnittlichen Dauer der Decodierung ermittelt. Diese beziehen sich hierbei allerdings auf die Brutto-Datenrate bzw. die durchschnittliche Dauer zur Decodierung eines Codebits.

Analog zum vorherigen Abschnitt ist in Abb. 5.7 die durchschnittliche Dauer zur Decodierung eines Codebits für die verschiedenen Decodervarianten gegeben.

Hierbei gilt es zu beachten, dass zur Ermittlung der Werte für die SD-Decodierung ein Block zur Typen-Konvertierung eingefügt werden muss. Damit ergibt sich für diesen Fall der in Abb. 5.8 dargestellte Aufbau in GNU Radio Companion. Für alle anderen Varianten wurde ein Aufbau analog zu Abb. 5.3 verwendet.

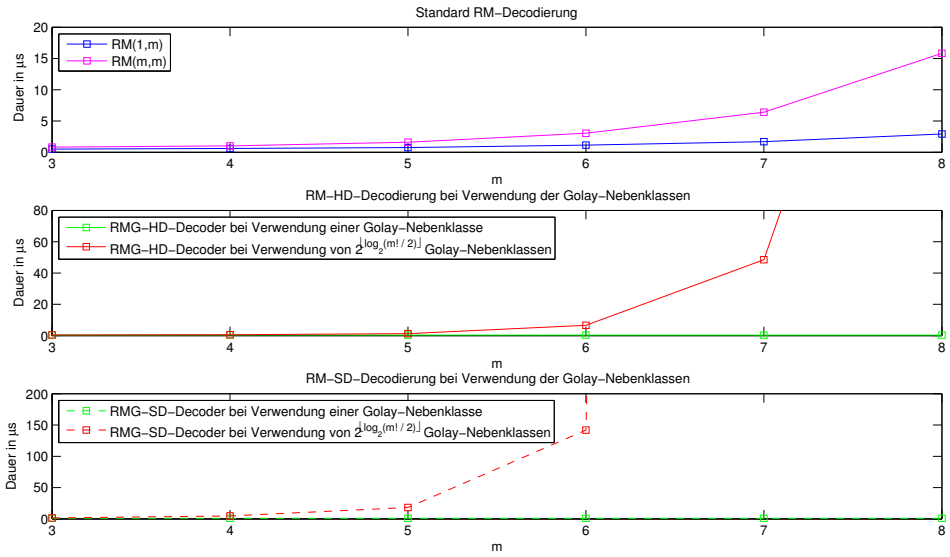

Abbildung 5.7.: Durchschnittliche Dauer zur Decodierung eines Codebits

Für die RM-Decodierung lässt sich erkennen, dass die Dauer stark von der verwendeten Ordnung abhängt. Die Kurve für $RM(m,m)$ stellt hierbei eine obere und die Kurve für $RM(1,m)$ eine untere Schranke für die durchschnittliche Dauer zur Decodierung eines Codebits dar.

Mit wachsendem m nimmt die Dauer für beide zu, für $RM(m,m)$ jedoch wesentlich stärker. Dies hängt damit zusammen, dass für hohe Ordnungen wesentlich mehr charakteristische Vektoren gebildet und verknüpft werden müssen.

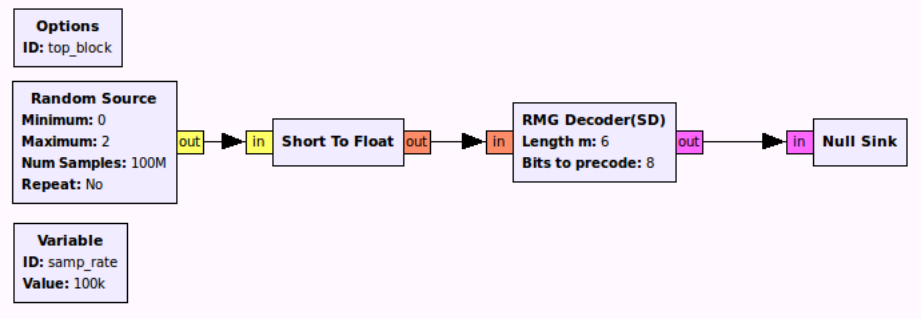

Abbildung 5.8.: Aufbau zur Ermittlung der Werte im Falle der SD-Decodierung

Bei Verwendung von Golay-Sequenzen bei der RM-Decodierung lässt sich erkennen, dass diese bei der Verwendung von nur einer Golay-Nebenklasse sehr effizient ausgeführt wird. Werden hingegen $2^{\lfloor \log_2(\frac{m!}{2}) \rfloor}$ Golay-Nebenklassen bei der En- und Decodierung verwendet, so steigt die zur Decodierung eines Codebits benötigte Dauer für $m > 5$ stark an.

Dies lässt sich zum einen dadurch erklären, dass mit jedem weiteren Anstieg von m die doppelte Anzahl an `integer` verarbeitet werden muss. Außerdem muss die FHT für jede zu verwendenden Nebenklassen separat berechnet werden. Daraus folgt, dass der Rechenaufwand mit der Anzahl zu verwendender Nebenklassen stark ansteigt.

Für die SD-Decodierung bei Verwendung von Golay-Sequenzen ergibt sich die durchschnittliche Dauer zur Decodierung eines Codebits noch höher als bei der HD-Decodierung. Dies ist auf ein geändertes Mapping, sowie auf den gesteigerten Rechenaufwand durch die nun komplexe FHT zurück zu führen.

In Abb. 5.9 sind, ausgehend von den Werten in Abb. 5.7, die, mit den unterschiedlichen Decodervarianten, erzielbaren Brutto-Datenraten dargestellt. Die entsprechenden Netto-Datenraten ergeben sich leicht mit Hilfe der in Abb. 3.2 gegebenen Coderaten.

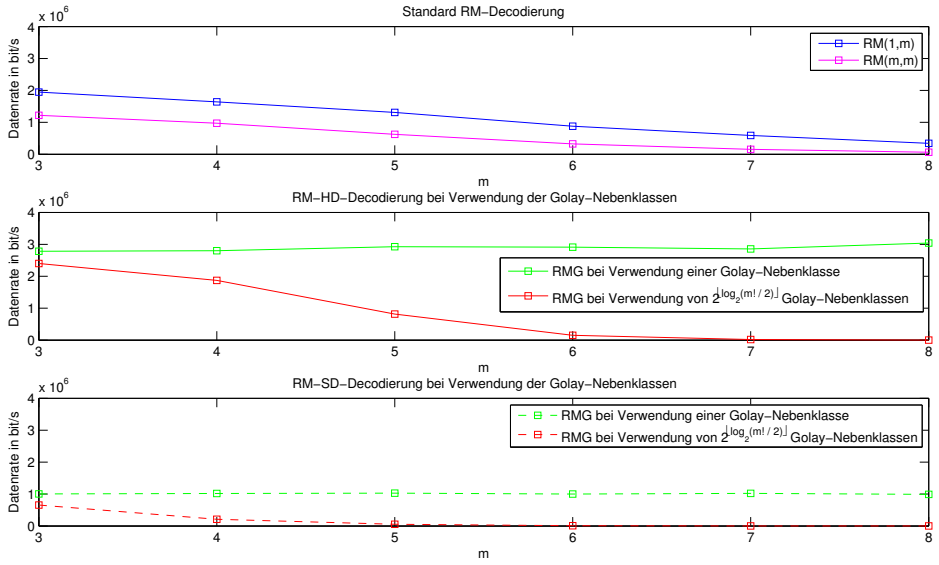

Abbildung 5.9.: Erzielbare Brutto-Datenraten für die Decodierung

Da im Decoder häufig zusätzliche Komponenten, beispielsweise zur Kanalschätzung und -entzerrung, benötigt werden, ist die von der Decodierung verursachte CPU-Auslastung hier von besonderem Interesse. In Abb. 5.10 ist die von der Decodierung verursachte CPU-Auslastung für eine Brutto-Datenrate von 100 kbit/s dargestellt[2].

Die fehlenden Punkte bei einigen Decodervarianten sind hierbei derart zu interpretieren, dass in diesen Fällen eine Brutto-Datenrate von 100 kbit/s nicht realisiert werden konnte. Die Begründungen für die unterschiedlichen Verläufe wurden bereits bei den Betrachtungen zur Dauer bzw. erzielbaren Datenrate gegeben. Auch hier zeigt sich, dass die Decodierung mittels FHT eine sehr effiziente Variante darstellt, sofern die Anzahl der Golay-Nebenklassen klein gewählt wird.

[2]Hierbei wurde für die Berechnungen nur einer der beiden verfügbaren Prozessorkerne genutzt.

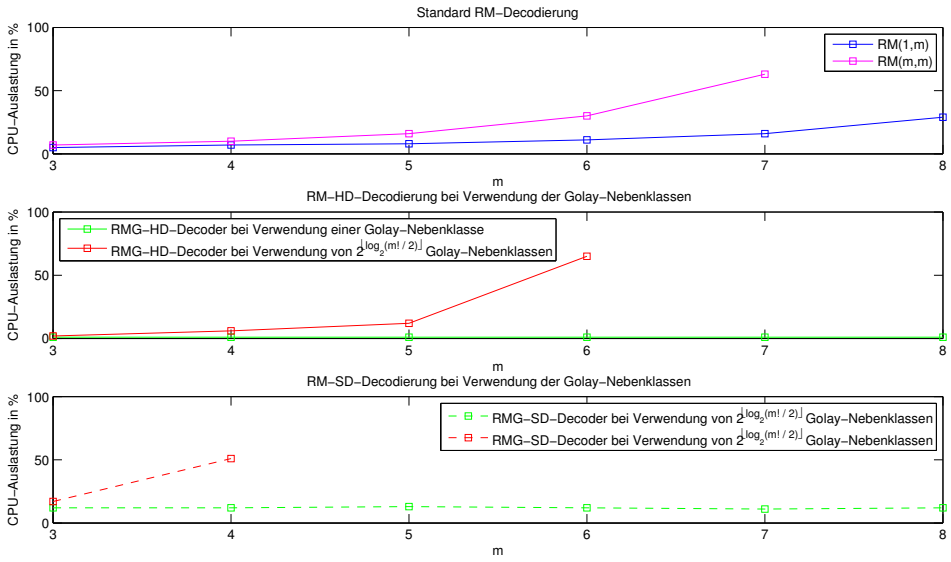

Abbildung 5.10.: Durch die Decodierung verursachte CPU-Auslastung bei einer Brutto-Datenrate von 100 kbit/s

Auch die Performance der RM-Decodierung lässt sich auf Kosten des Speicherbedarfs verbessern; speichert man alle charakteristische Vektoren bereits bei der Initialisierung, so fällt der zur eigentlichen Decodierung benötigte Rechenaufwand wesentlich geringer aus.

Analog zur Encodierung kann die Performance der Decodierung bei Verwendung von Golay-Sequenzen dadurch verbessert werden, dass in der Precodematrix direkt die Nebenklassenanführer anstatt Bitmuster zu deren Erzeugung gespeichert werden. Durch die Einführung der verallgemeinerten Reed-Muller Codes lässt sich die Performance weiter verbessern, da hierbei mehrere Informationsbits zu PSK-modulierten Symbolen zusammengefasst werden können, deren Decodierung sich mittels der FHT ähnlich performant wie im binären Fall realisieren.

5.2. Möglichkeiten zur praktischen Anwendung

Die in dieser Arbeit entstandene Implementierung in GNU Radio eignet sich besonders für die Verwendung in OFDM-basierten Systemen. Wie in Kapitel 2 gezeigt wurde, lässt sich mit Hilfe von Golay-Sequenzen das Peak Power Problem lösen.

Für eine Datenübertragung mit Hilfe der entstandenen Implementierung, lässt sich eine sehr flexible Anpassung an die, durch den Übertragungskanal vorgegebenen, Systemparameter, realisieren.

Die Anzahl zu verwendender Subträger ergibt sich hierbei direkt aus den Eigenschaften des Kanals, sowie aus den an die Leistungsfähigkeit des Systems gestellten Forderungen. Für die Realisierung eines praktischen Systems wählt man die Anzahl der Subträger zu einer Potenz von zwei, was die effiziente (De-)Modulation mittels der (I)FFT ermöglicht. Entsprechend dazu wählt man die Länge der Codewörter (2^m) derart, dass jedem Subträger pro erzeugtem Codewort ein Codebit zugeordnet werden kann.

Die Korrekturfähigkeit der verwendeten Codierung lässt sich hierbei durch die Anzahl zu verwendender Golay-Nebenklassen beeinflussen, wodurch eine weitere Anpassung an den Kanal ermöglicht wird. Auch die Wahl einer HD- bzw. SD-Decodierung kann abhängig vom Kanal und der Leistungsfähigkeit des verwendeten Systems erfolgen.

Für Übertragungen mit nicht-OFDM-basierten Systemen lassen sich wahlweise die klassische RM-Codierung oder die RM-Codierung bei Verwendung von Golay-Sequenzen einsetzen[3]. Wählt man die klassischen RM-Codes, so lassen sich für unterschiedliche Kombinationen von m und r sehr unterschiedliche Coderaten mit unterschiedlichen Korrekturfähigkeiten realisieren, womit eine sehr flexible Anpassung an den verwendeten Kanal möglich ist.

In diesem Kapitel wurde die Leistungsfähigkeit der Implementierung aufgezeigt. Es wurde in diesem Zusammenhang auf Möglichkeiten zur Steigerung der Performance hingewiesen. Darüber hinaus wurden Möglichkeiten zur Anpassung an einen Übertragungskanal für eine praktische Anwendung in einem realen System vorgestellt.

[3] Der Einsatz einer RMG-Codierung ist in nicht-OFDM-basierten Systemen nur bei Verwendung einer einzigen Golay-Nebenklasse sinnvoll, da bei Verwendung mehrerer Nebenklassen ein reiner Overhead entsteht, der keinen weiteren Vorteil liefert.

6. Zusammenfassung

Im Rahmen dieser Arbeit wurden zwei Varianten der binären Reed-Muller Codes in GNU Radio implementiert. Die erste Variante stellt hierbei eine Reed-Muller Codierung im klassischen Sinne dar, welche als ein Verfahren zur Kanalcodierung vor allem bei Datenübertragungen über schwierige Kanäle eingesetzt werden kann.

Die zweite Variante wurde speziell für den Einsatz in OFDM-basierten Übertragungssystemen implementiert. Hierbei tritt das Peak Power Problem auf, da die Sendesignale durch die OFDM-Operation keine konstante Einhüllende besitzen und teilweise sehr hohe Spitzenwerte der momentanen Leistung der komplexen Einhüllenden im Vergleich zur mittleren Leistung der komplexen Einhüllenden auftreten können. Dieses Problem lässt sich durch die in der zweiten Variante verwendete Codierung lösen.

Im ersten Kapitel wurde gezeigt, dass sich bei ausschließlicher Verwendung von Golay-Sequenzen für die Codewörter bei der Übertragung das PMEPR und damit auch das PAPR auf 3 dB begrenzen lassen. Dieser Ansatz zur Lösung des Peak Power Problems wurde im zweiten Kapitel erneut aufgegriffen und nach [DJ99] ein Zusammenhang zwischen Golay-Sequenzen und den Reed-Muller Codes der zweiten Ordnung hergestellt. Damit wurden, ebenfalls in Kapitel 2, Algorithmen zur En- und Decodierung sowohl für klassische Reed-Muller Codes als auch für Reed-Muller Codes bei ausschließlicher Verwendung von Golay-Sequenzen vorgestellt.

Bei der Implementierung der vorgestellten Algorithmen wurde besonderer Wert auf die Flexibilität hinsichtlich möglicher Anpassungen an den Übertragungskanal sowie das bei der zur Übertragung verwendete System gelegt. Damit lassen sich in Abhängigkeit vom verwendeten System Codes mit unterschiedlichen Coderaten und somit auch unterschiedlichen Korrekturfähigkeiten realisieren, deren Decodierung, wiederum in Abhängigkeit von der Leistungsfähigkeit des verwendeten Systems, wahlweise per Hard-Decision oder per Soft-Decision erfolgen kann.

Schließlich wurden in Kapitel 5 noch Möglichkeiten zur Performance-Steigerung der Implementierung aufgezeigt. Hierbei wurde unter anderem die Verwendung verallgemeinerter Reed-Muller Codes zur Codierung im Zusammenhang mit Golay-Sequenzen vorgeschlagen, wodurch sich eine PSK-Modulation und damit eine gesteigerte Performance realisieren ließe.

A. Anhang

A.1. HD-Decodierung für $RM(2, m)$ bei Verwendung von Golay-Nebenklassen

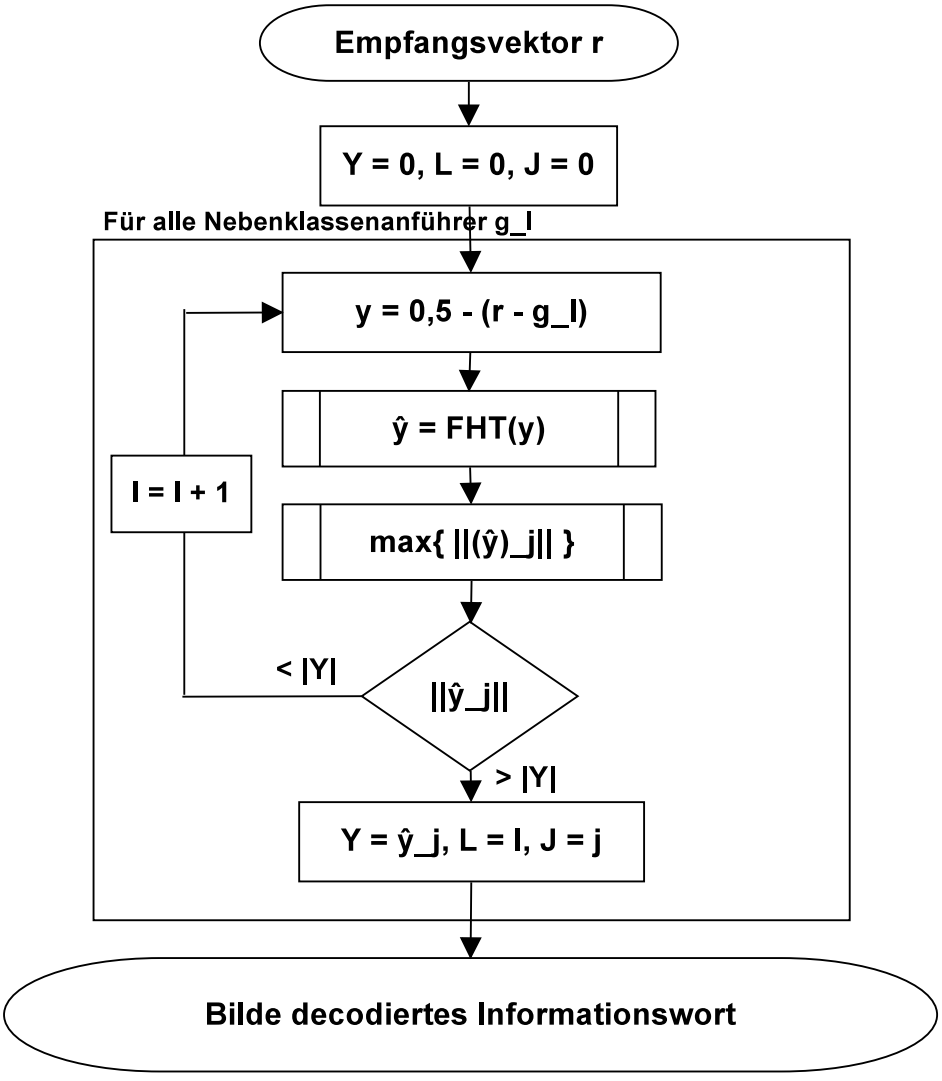

Abbildung A.1.: Decodieralgorithmus nach [DJ99]

A.2. Ergänzungen zur Implementierung

A.2.1. Auflistung der entstandenen Dateien und Funktionen

Die entstandene Implementierung beinhaltet 7 Blöcke, die der Signalverarbeitung zugeordnet werden können. Darüber hinaus entstanden 5 weitere Blöcke, welche als hierarchische Blöcke Verknüpfungen der anderen Blöcke darstellen. Für 7 der 12 Blöcke wurden außerdem Bindings für den GNU Radio Companion hinzugefügt, mit dessen Hilfe sich die Blöcke visuell miteinander verknüpfen lassen.

Es folgt eine Auflistung aller entstandener Dateien, sowie eine kurze Beschreibung der realisierten Funktionen. Hierbei wurden in Zusammenhang stehende Dateien nach folgendem Schema zusammengefasst.

Header	Body
QA (Tests)	GRC-Binding

Beschreibung

Es folgen nun zunächst die 5 Blöcke, welche der eigentlichen En- bzw. Decodierung zugeordnet werden können:

lib/chancoding_rm_decoder_vii.h	lib/chancoding_rm_decoder_vii.cc
python/qa_chancoding_rm_decoder_vii.py	

Standard RM-Decoder für `integer` (RM-Decoder)

lib/chancoding_rm_encoder_vii.h	lib/chancoding_rm_encoder_vii.cc
python/qa_chancoding_rm_encoder_vii.py	

Standard RM-Encoder für `integer` (RM-Encoder)

lib/chancoding_rmg_decoder_sd_fi.h	lib/chancoding_rmg_decoder_sd_fi.cc
python/qa_chancoding_rmg_decoder_sd_vfi.py	

SD-RM-Decoder bei Verwendung von Golay-Sequenzen (RMG-SD-Decoder)

lib/chancoding_rmg_encoder_vii.h	lib/chancoding_rmg_encoder_vii.cc
python/qa_chancoding_rmg_encoder_vii.py	

RM-Encoder bei Verwendung von Golay-Sequenzen (RMG-Encoder)

lib/chancoding_rmg_decoder_vii.h	lib/chancoding_rmg_decoder_vii.cc
python/qa_chancoding_rmg_decoder_vii.py	

HD-RM-Decoder bei Verwendung von Golay-Sequenzen (RMG-HD-Decoder)

Es folgen die Hilfsfunktionen, welche von den meisten anderen Blöcken für interne Abläufe verwendet werden:

lib/chancodingi_rm_coding.h	lib/chancodingi_rm_coding.cc
lib/qa_chancodingi_rm_coding.cc	

Mit Hilfe dieser Dateien wurden die folgenden Hilsfunktionen realisiert:

- void rm_encode(unsigned int* gen_mat, int num_rows, int num_int, unsigned int* uncoded, int num_int_uncoded, unsigned int* encoded)
- void rm_fht_sd(complex<float>* y, int m)
- void rm_fht(float* y, int m)
- unsigned int rm_calc_num_bits_precoded(int m)
- void rmg_generate_precode_mat(short m, unsigned int* precode_mat, unsigned int num_rows_precode_mat, unsigned int num_int_precode_mat)
- int rm_calc_dot_product(unsigned int* vec1, unsigned int* vec2, int num_int)
- int rm_count_ones_in_vec(unsigned int* vec, int num_int)
- void rm_generate_gen_mat(short r, short m, unsigned int* gen_mat, int num_rows, int num_int)
- void rm_change_matrix_element(unsigned int* matrix, int num_int, int row, int column, bool value)
- int rm_calc_rows(int r, int m)
- long rm_gcd (long a, long b)
- int rm_binom_coeff(int m, int r)
- void rm_vector_add(unsigned int* matrix1, unsigned int* matrix2, unsigned int* result_matrix, int row1, int row2, int result_row, int num_int)
- void rm_vector_add_neg(unsigned int* matrix1, unsigned int* matrix2, unsigned int* result_matrix, int row1, int row2, int result_row, int num_int)
- void rm_vector_xor(unsigned int* matrix1, unsigned int* matrix2, unsigned int* result_matrix, int row1, int row2, int result_row, int num_int)
- unsigned int rm_calc_num_infobits(int r, int m, bool golay)
- int rm_calc_num_int(int num_bits)

Schließlich wurden noch zwei Blöcke zum Bit-(Un-)Packing implementiert, da Blöcke zur En- bzw. Decodierung alle Vektoren in `integer` speichern. Für beide Blöcke wurden GRC-Bindings hinzugefügt.

lib/chancoding_packed_to_unpacked_vib.h	lib/chancoding_packed_to_unpacked_vib.cc
python/qa_chancoding_packed_to_unpacked_vib.py	grc/chancoding_packed_to_unpacked_vib.xml

Bit-Unpacker: Entpackt eine bestimmte Anzahl Bits aus einem `integer`

lib/chancoding_unpacked_to_packed_bvi.h	lib/chancoding_unpacked_to_packed_bvi.cc
python/qa_chancoding_unpacked_to_packed_bvi.py	grc/chancoding_unpacked_to_packed_bvi.xml

Bit-Packer: Packt eine bestimmte Anzahl Bits in einen `integer`

Abschließend sind die hierarchischen Blöcke gegeben, für die ebenfalls GRC-Bindings hinzugefügt wurden:

lib/chancoding_rmg_encoder_bb.h	lib/chancoding_rmg_encoder_bb.cc
	grc/chancoding_rmg_encoder_bb.xml

Hierarchischer Block, aufgebaut aus: Bit-Packer - RMG-Encoder - Bit-Unpacker

lib/chancoding_rmg_decoder_sd_fb.h	lib/chancoding_rmg_decoder_sd_fb.cc
	grc/chancoding_rmg_decoder_sd_fb.xml

Hierarchischer Block, aufgebaut aus: Stream2Vector - RMG-SD-Decoder - Bit-Unpacker

lib/chancoding_rmg_decoder_bb.h	lib/chancoding_rmg_decoder_bb.cc
	grc/chancoding_rmg_decoder_bb.xml

Hierarchischer Block, aufgebaut aus: Bit-Packer - RMG-HD-Decoder - Bit-Unpacker

lib/chancoding_rm_encoder_bb.h	lib/chancoding_rm_encoder_bb.cc
	grc/chancoding_rm_encoder_bb.xml

Hierarchischer Block, aufgebaut aus: Bit-Packer - RM-Encoder - Bit-Unpacker

lib/chancoding_rm_decoder_bb.h	lib/chancoding_rm_decoder_bb.cc
	grc/chancoding_rm_decoder_bb.xml

Hierarchischer Block, aufgebaut aus: Bit-Packer - RM-Decoder - Bit-Unpacker

A.2.2. Auszüge aus dem Quelltext

Funktion zum Ändern eines Matrix-Elements

```
 void rm_change_matrix_element(unsigned int* matrix, int num_int, int row, int column,
     bool value)
{
   //in which integer is the element?
   int bits_per_int = sizeof(num_int) * 8;
   int current_integer = column / bits_per_int; //results directly to the integer

   //Bit in current integer
   int element_in_int =  column % bits_per_int;

   //Insert the value
   if(!(matrix[row * num_int + current_integer] & (1<<element_in_int)) && value == 1)
      matrix[row * num_int + current_integer] += (1<<element_in_int);

   if((matrix[row * num_int + current_integer] & (1<<element_in_int)) && value == 0)
      matrix[row * num_int + current_integer] -= (1<<element_in_int);
}
```

Funktion zur XOR-Verknüpfung zweier Vektoren

```
void rm_vector_xor(unsigned int* matrix1, unsigned int* matrix2, unsigned int*
    result_matrix, int row1, int row2, int result_row, int num_int)
{
   for(int a = 0; a < num_int; a++)
   {
      result_matrix[result_row * num_int + a] = matrix1[row1 * num_int + a] ^ matrix2[
          row2 * num_int + a];
   }
}
```

Funktion zum Bilden der Generatormatrix

```
void rm_generate_gen_mat(short r, short m, unsigned int* gen_mat, int num_rows, int
    num_int)
{
   memset((void *) gen_mat, 0, num_int * num_rows * sizeof(int) );
   unsigned int i = 1;
   i = (1<<m);

   //Build monoms (order r=1)

   if(r >=1)
   {
      for(int a = 0; a <= ((1<<m) - 1); a++)
         rm_change_matrix_element(gen_mat, num_int, 0, a, 1);
      for(int a = 1; a <= m; a++)
      {
         for(int b = 1; b <= (1<<a); b++)
         {
            for(int c = 1; c <= (1<<(m-a)); c++)
            {
               if(b % 2 != 0)
                  rm_change_matrix_element(gen_mat, num_int, a, ((b-1) * (1<<(m-a)) + c
                      -1), 1);
```

```
            }
        }
    }
}

//Set all rows after the monoms to 1 (prepare for linear combinations)
for(int a = m+1; a < num_rows; a++)
    for(int b = 0; b <= num_int; b++)
        gen_mat[a * num_int + b] = gen_mat[0];

//Rearrange so that the orders are together at the right place
int gen_mat_row_counter = 0;
for(int a = 2; a <= r; a++)
{
    for(int b = m-1; b >= 0; b--)
    {
        for(unsigned int c=1; c <= i; c++)
        {
            bool abort = 0;
            int num_ones = 0;
            for(short k = 0; k < m; k++)
            {
                if(c & (1<<(m-k-1)))
                {
                    num_ones++;
                }
            }
            if(num_ones == a)
            {
                for(int d = b+1; d <= m-1; d++)
                {
                    if(c & (1<<(m-d-1)))
                        abort = 1;
                }
                if(c & (1<<(m-b-1)) && abort == 0)
                {
                    for(int e = 0; e < m; e++)
                    {
                        if(c & (1<<(m-e-1)))
                            rm_vector_add(gen_mat, gen_mat, gen_mat, m+1+gen_mat_row_counter,
                                m-e, m+1+gen_mat_row_counter, num_int);
                    }
                    gen_mat_row_counter++;
                }
            }
        }
    }
}
```

RM-Encoder

```
void rm_encode(unsigned int* gen_mat, int num_rows, int num_int, unsigned int* uncoded,
    int num_int_uncoded, unsigned int* encoded)
{
    memset((void *) encoded, 0, num_int * sizeof(int) );

    int bits_per_int = sizeof(num_rows) * 8;
    int current_int = 0;
    for(int a = 0; a < num_rows; a++)
```

```
    {
        if(a > 0 && a % bits_per_int == 0)
            current_int++;
        if(uncoded[current_int] & (1<<(a-(current_int * bits_per_int))))
            rm_vector_xor(encoded, gen_mat, encoded, 0, a, 0, num_int);
    }
}
```

RM-Decoder

```
int
chancoding_rm_decoder_vii::work (int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items)
{
    const unsigned int *encoded = (const unsigned int *) input_items[0];
    unsigned int *decoded = (unsigned int *) output_items[0];

    for(int output_items_counter = 0; output_items_counter < noutput_items;
        output_items_counter++)
    {
        memset((void *) decoded, 0, d_num_int_decoded * sizeof(int) );
        memset((void *) d_char_vec, 0, d_num_int * sizeof(int) );
        memset((void *) d_order_decoded, 0, d_num_int * sizeof(int) );
        memset((void *) d_zero_columns, 0, d_m * sizeof(int) );

        int decoded_bit_counter = 0;
        unsigned int i = (1<<d_m);
        unsigned int encoded_temp[d_num_int];
        for(int a = 0; a < d_num_int; a++)
            encoded_temp[a] = encoded[a];

        for(int a = d_r; a >=1; a--)
        {
            for(int b = 0; b < d_m; b++)
            {
                for(unsigned int c=i; c >= 1; c--)
                {
                    bool abort = 0;
                    int num_ones = 0;
                    for(short k = 0; k < d_m; k++)
                    {
                        if(c & (1<<(d_m-k-1)))
                        {
                            num_ones++;
                        }
                    }
                    if(num_ones == a)
                    {
                        for(int d = b+1; d <= d_m-1; d++)
                        {
                            if(c & (1<<(d_m-d-1)))
                                abort = 1;
                        }
                        if(c & (1<<(d_m-b-1)) && abort == 0)
                        {
                            int dot_products_sum = 0;
                            int num_zeros_row_c = d_m-num_ones;

                            int zero_columns_counter = 0;
```

```
                        for(int g = 0; g < d_m; g++)
                        {
                            if(!(c & (1<<g)))
                            {
                                d_zero_columns[zero_columns_counter] = g+1;
                                zero_columns_counter++;
                            }
                        }
                        for(int e = 0; e <= ((1<<num_zeros_row_c) -1); e++)
                        {
                            for(int f = 0; f < d_num_int; f++)
                                d_char_vec[f] = d_gen_mat[f];
                            for(int f = 0; f < num_zeros_row_c; f++)
                            {
                                if(e & (1<<f))
                                {
                                    rm_vector_add(d_char_vec, d_gen_mat, d_char_vec, 0,
                                        d_zero_columns[f], 0, d_num_int);
                                }
                                else
                                    rm_vector_add_neg(d_char_vec, d_gen_mat, d_char_vec, 0,
                                        d_zero_columns[f], 0, d_num_int);
                            }
                            dot_products_sum += rm_calc_dot_product(encoded_temp, d_char_vec,
                                d_num_int);
                        }
                        if(dot_products_sum > (1<<(num_zeros_row_c - 1)) && num_zeros_row_c
                            > 0)
                        {
                            rm_change_matrix_element(decoded, d_num_int_decoded, 0,
                                d_num_rows - decoded_bit_counter - 1, 1);
                            rm_vector_xor(d_order_decoded, d_gen_mat, d_order_decoded, 0,
                                d_num_rows - decoded_bit_counter - 1, 0, d_num_int);
                        }
                        decoded_bit_counter++;
                    }
                }
            }
        }
        rm_vector_xor((unsigned int*) encoded, d_order_decoded, encoded_temp, 0, 0, 0,
            d_num_int);
    }
    if(rm_count_ones_in_vec(encoded_temp, d_num_int) > (1<<(d_m-1)))
    {
        rm_change_matrix_element(decoded, d_num_int_decoded, 0, d_num_rows -
            decoded_bit_counter - 1, 1);
    }

    encoded += d_num_int;
    decoded += d_num_int_decoded;
  }
  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

Funktion zum Bilden der Precode-Matrix

```
void rmg_generate_precode_mat(short m, unsigned int* precode_mat, unsigned int
    num_rows_precode_mat, unsigned int num_int_precode_mat)
{
```

```
    memset((void *) precode_mat, 0, num_int_precode_mat * num_rows_precode_mat * sizeof(
        int) );
    int permutation[m];
    memset((void *) permutation, 0, m * sizeof(int) );
    for(int a = 0; a < m; a++)
        permutation[a] = a+1;

    for(unsigned int a = 0; a < num_rows_precode_mat; a++)
    {
        //The last element must be bigger than the first one
        while(permutation[m-1] <= permutation[0])
            next_permutation(permutation, permutation+m);

        //Pass through the patterns of linear combinations and compare it with the current
            permutation
        int counter = 0;
        for(int b = 1; b <= m; b++)
        {
            for(int c = b+1; c <= m; c++)
            {
                //Search for the pattern b,c in current permutation, if it exists -> 1
                for(int d = 0; d < (m-1); d++)
                {
                    if(permutation[d] == b || permutation[d] == c)
                    {
                        if(permutation[d+1] == b || permutation[d+1] == c)
                            rm_change_matrix_element(precode_mat, num_int_precode_mat, a,
                                counter, 1);
                    }
                }
                counter++;
            }
        }

        next_permutation(permutation, permutation+m);
    }
}
```

RM-Encoder bei Verwendung von Golay-Sequenzen

```
int
chancoding_rmg_encoder_vii::work (int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items)
{
    const unsigned int *uncoded = (const unsigned int *) input_items[0];
    unsigned int *encoded = (unsigned int *) output_items[0];

    memset((void *) encoded, 0, noutput_items * d_num_int * sizeof(int) );

    for(int output_items_counter = 0; output_items_counter < noutput_items;
        output_items_counter++)
    {
        int coset_selector = 0;
        int bits_per_int = sizeof(int) * 8;
        int current_int = 0;
        int bit_in_int = 0;

        for(int a = 0; a < d_num_int_uncoded; a++)
            d_uncoded_temp[a] = uncoded[a];
```

```
        // Take the last "d_num_bits_precoded" from uncoded as decimal number to choose the
             coset:
        for(int a = d_m + 1; a <= d_m + d_num_bits_precoded; a++)
        {
           current_int = rm_calc_num_int(a) - 1;
           bit_in_int = a % bits_per_int;
           if(d_uncoded_temp[current_int] & (1<<bit_in_int))
           {
              coset_selector += (1<<(a-d_m-1));
           }
        }

        current_int = 0;
        for(int a = 0; a < d_num_lin_comb_order_2; a++)
        {
           if((a > 0) && (a % bits_per_int == 0))
           {
              current_int++;
           }
           if(d_precode_mat[coset_selector * d_num_int_precode_mat + current_int] & (1<<(a-
                current_int * bits_per_int)))
              rm_change_matrix_element(d_uncoded_temp, d_num_int_uncoded, 0, a+d_m+1, 1);
           else
              rm_change_matrix_element(d_uncoded_temp, d_num_int_uncoded, 0, a+d_m+1, 0);
        }

        current_int = 0;
        for(int a = 0; a < d_num_rows; a++)
        {
           if(a > 0 && a % bits_per_int == 0)
              current_int++;
           if(d_uncoded_temp[current_int] & (1<<(a-(current_int * bits_per_int))))
              rm_vector_xor(encoded, d_gen_mat, encoded, 0, a, 0, d_num_int);
        }

        uncoded += d_num_int_uncoded;
        encoded += d_num_int;
     }

     // Tell runtime system how many output items we produced.
     return noutput_items;
}
```

Realisierung der Fast-Hadamard-Transformation

```
void rm_fht(float* y, int m)
{
   for(int a = m-1; a >= 0; a--)
   {
      for(int b = 0; b < (1<<m); b += (1<<(a+1)))
      {
         for(int c = 0; c < (1<<a); c++)
         {
            float tmp = y[b + c];
            y[b + c] += y[b + c + (1<<a)];
            y[b + c + (1<<a)] = tmp - y[b + c + (1<<a)];
         }
      }
   }
```

```
}
```

RM-Decoder bei Verwendung von Golay-Sequenzen

```
int
chancoding_rmg_decoder_vii::work (int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items)
{
   const unsigned int *encoded = (const unsigned int *) input_items[0];
   unsigned int *decoded = (unsigned int *) output_items[0];

   for(int output_items_counter = 0; output_items_counter < noutput_items;
       output_items_counter++)
   {
      memset((void *) decoded, 0, d_num_int_decoded * sizeof(int) );

      int bits_per_int = sizeof(d_num_int) * 8;
      int current_int = 0;
      int L = 0;
      float Y = 0;
      int J = 0;

      for(int l = 0; l < d_num_rows_precode_mat; l++)
      {
         // Build d_coset:
         memset((void *) d_coset, 0, d_num_int * sizeof(int) );

         current_int = 0;
         for(int a = 0; a < d_num_lin_comb_order_2; a++)
         {
            if(a > 0 && a % bits_per_int == 0)
               current_int++;
            if(d_precode_mat[l * d_num_int_precode_mat + current_int] & (1<<(a-(
                current_int * bits_per_int))))
               rm_vector_xor(d_coset, d_gen_mat, d_coset, 0, a+d_m+1, 0, d_num_int);
         }

         //Build d_y for FHT -> floats
         rm_vector_xor(d_coset, (unsigned int*)encoded, d_coset, 0, 0, 0, d_num_int);
         current_int = 0;
         for(int a = 0; a < d_num_bits; a++)
         {
            if(a > 0 && a % bits_per_int == 0)
               current_int++;
            if(d_coset[current_int] & (1<<(a-(current_int * bits_per_int))))
               d_y[d_num_bits - a - 1] = -0.5;
            if(!(d_coset[current_int] & (1<<(a-(current_int * bits_per_int)))))
               d_y[d_num_bits - a - 1] = 0.5;
         }

         rm_fht(d_y, d_m);

         for(int j = 0; j < d_num_bits; j++)
         {
            if(fabs(d_y[j]) > fabs(Y) )
            {
               Y = d_y[j];
               L = l;
               J = j;
```

```
            }
        }
    }

    // Build decoded Bitstring:
    for(int a = 0; a <= d_m + d_num_bits_precoded; a++)
    {
        if((L & (1<<(d_num_bits_precoded - a - 1))) && (a < d_num_bits_precoded))
            rm_change_matrix_element(decoded, d_num_int_decoded, 0, d_m +
                d_num_bits_precoded - a, 1);
        if((J & (1<<(a-d_num_bits_precoded))) && (d_num_bits_precoded <= a) && (a < d_m
            + d_num_bits_precoded))
            rm_change_matrix_element(decoded, d_num_int_decoded, 0, d_m +
                d_num_bits_precoded - a, 1);
        if((Y < 0) && (a == d_m + d_num_bits_precoded))
            rm_change_matrix_element(decoded, d_num_int_decoded, 0, d_m +
                d_num_bits_precoded - a, 1);
    }

    decoded += d_num_int_decoded;
    encoded += d_num_int;
  }

  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

Literaturverzeichnis

[CS86] CONWAY, J. und N. SLOANE: *Soft decoding techniques for codes and lattices, including the Golay code and the Leech lattice*. Information Theory, IEEE Transactions on, 32(1):41 – 50, Januar 1986.

[CT65] COOLEY, JAMES und JOHN TUKEY: *An Algorithm for the Machine Calculation of Complex Fourier Series*. Mathematics of Computation, 19(90):297–301, 1965.

[DJ97] DAVIS, J.A. und J. JEDWAB: *Peak-to-mean power control and error correction for OFDM transmission using Golay sequences and Reed-Muller codes*. Electronics Letters, 33(4):267 –268, Februar 1997.

[DJ99] DAVIS, J.A. und J. JEDWAB: *Peak-to-mean power control in OFDM, Golay complementary sequences, and Reed-Muller codes*. Information Theory, IEEE Transactions on, 45(7):2397 –2417, November 1999.

[Fri95] FRIEDRICHS, BERND: *Kanalcodierung. Grundlagen und Anwendungen in modernen Kommunikationsanlagen (Information Und Kommunikation)*. Springer, Berlin, 1 Auflage, November 1995.

[Gol61] GOLAY, M.: *Complementary series*. Information Theory, IRE Transactions on, 7(2):82 –87, 1961.

[HDM57] HEALD, M.L., E.T. DOELZ und D.L. MARTIN: *Binary Data Transmission Techniques for Linear Systems*. Proceedings of the IRE, 45(5):656–661, 1957.

[HL05] HAN, SEUNG HEE und JAE HONG LEE: *An overview of peak-to-average power ratio reduction techniques for multicarrier transmission*. Wireless Communications, IEEE, 12(2):56 – 65, April 2005.

[Jed08] JEDWAB, JONATHAN: *What can be used instead of a Barker sequence?* Contemp. Math., 461:153–178, 2008.

[Jon08] JONDRAL, FRIEDRICH: *Nachrichtensysteme: Grundlagen - Verfahren - Anwendungen*. Schlembach Fachverlag, 3. Auflage, April 2008.

[Kam08] KAMMEYER, KARL-DIRK: *Nachrichtenübertragung : mit ... 35 Tabellen*. Studium. Vieweg + Teubner, Wiesbaden, 4. neu bearb. u. erg. Auflage, 2008.

[Lit07] LITSYN, SIMON: *Peak Power Control in Multicarrier Communications: With Applications in OFDM and DMT*. Cambridge University Press, 1 Auflage, Januar 2007.

[Pat00] PATERSON, K.G.: *Generalized Reed-Muller codes and power control in OFDM modulation*. Information Theory, IEEE Transactions on, 46(1):104 –120, Januar 2000.

[PJ00] PATERSON, K.G. und A.E. JONES: *Efficient decoding algorithms for generalized Reed-Muller codes*. Communications, IEEE Transactions on, 48(8):1272 –1285, August 2000.

[Pop91] POPOVIC, B.M.: *Synthesis of power efficient multitone signals with flat amplitude spectrum*. Communications, IEEE Transactions on, 39(7):1031 –1033, Juli 1991.

[Ree54] REED, I.: *A class of multiple-error-correcting codes and the decoding scheme*. Information Theory, IRE Professional Group on, 4(4):38 –49, September 1954.

[RL06] RIHAWI, B. und Y. LOUET: *Peak-to-Average Power Ratio analysis in MIMO systems*. In: *Information and Communication Technologies, 2006. ICTTA '06. 2nd*, Band 2, Seiten 2110 –2114, 2006.

[SF05] SCHMIDT, K.-U. und A. FINGER: *Simple maximum-likelihood decoding of generalized first-order Reed-Muller codes*. Communications Letters, IEEE, 9(10):912 – 914, Oktober 2005.

[Sha48] SHANNON, CLAUDE E.: *A mathematical theory of Communication*. The Bell system technical journal, 27:379–423, Juli 1948.

[WE71] WEINSTEIN, S. und P. EBERT: *Data Transmission by Frequency-Division Multiplexing Using the Discrete Fourier Transform*. IEEE Transactions on Communications, 19(5):628–634, Oktober 1971.

[Wic95] WICKER, STEPHEN B.: *Error control systems for digital communication and storage*. Prentice Hall, 1995.