Kommunikation durch Pulsdauermodulation in Software Defined Radio

Author: Varban Metodiev
Betreuer: Dr. Hristomir Yordanov

Technische Universität Sofia, FDIBA

2019
Oktober

Inhaltsverzeichnis

1 Einführung und Überlegungen

Seit der letzten zehn Jahren wurden interessante Entwicklungen im Feld der mobilen technologien beobachtet. Die erste Generation (die so genannte 1G) führte die Mobilität auf dem Markt. Die mobile Telefonie war nicht nur ein technologisches Phänomen, sondern auch eine soziale Notwendigkeit. Dann kommte die 2G und 3G Netze, die am Anfang des 21-Jahrhundert diente als die Brücke zwischen dem packetvermittellten Internet und den kanalkommutierten Telefonnetzen.

Heutzutage erfahren wir die Vorteilen der LTE-Technologie, die alle netze als Packetvermittlelte definiert hat. Netzwerkdurchsätze über 100Mbps sind etwas normales, die nicht so langsamer im Vergleich mit der Kabel-Internetanbieter sind. Mobiltelefone werden nicht mehr nur als Kommunikationsgeräte benutzt, sondern als komplette Computersysteme – menschen tauschen Dokumenten aus, Finanztransaktionen, Audio und Video mit hoher Auflösung.

In der Zukunft werden neue Herausforderungen für die mobile Industrie erwartet im Bereich Sensornetze, Roboter und die verschiedene Aspekte der Prozessautomatisierung (z.B. für die Ziele der Herstellung und der Smart-Haushalt). Die Verbindung für diese Netze soll nicht nur hohe Datenraten erreichen kann, sondern muss es stabil und energieeffizient langfristig bleiben. Für die ziele der Kontrolle verschiedenen Geräten in Echtzeit, die Latenz spielt eine maßgebliche rolle – deswegen soll man diesen zusätzlichen Faktor beachten für die Entwicklung der Kommunikationsschnittstellen.

Einer Rückblick in der Geschichte der 1G, 2G, 3G und LTE gibt es die Tendenz der Evolution der Mobilnetzen aus. Die Industrie fängt mit einfachen Modulationsverfahren an - wie AM (Amplitudenmodulation) und PSK (Phasenmodulation) an. Als die Bauteilen der Netzwerkgeräten billigen werden, der Ansatz wird komplexer. Modulationen wie QPSK, 16-QAM und QAM von höheren Reihen werden weit verbreiten. Fast alle TV-Empfänger und WiFi-Router haben solche Modems verwendet. Und wenn es über die Kapazität des Kanals geht, OFDM (Orthogonal Frequency-Division Multiplexing) ist der De-facto-Standard für Sammlung und Anordnung verschiedenen Frequenzen in einem bestimmten Kanal.

Obwolh diese komplizierte Verfahren für Signalverarbeitung sehr effektiv um höhe Datenraten zu erreichen sind, sie kann nicht immer praktsicht sein, besonders für Sensornetze. Das Konzept der IoT (Internet der Dinge) behauptet eine Verwendung von größen Anzahl von Sensoren, die miteinander kommunizieren. In vielen Fällen können die Paketgröße zwischen diesen Sensoren weniger als 1000 Bytes sein. Für Datenerfassungsgeräte gibt es andere Aspekte, die wichtiger als die Datenrate sind - wie Stabilität der Datenübertragung und Einfachkeit der Realisierung, die auch den Preis und Strommverbrauch beeinflusst.

2 Meine Lösung - PDM für Datenübertragung

In diesem Arbeit schlage ich eine solche Lösung vor - Pulsedauermodulation mithilfe der SDR (Software Defined Radio) Technologie. Es kann effizienter als die 4-QAM Modulation sein und braucht eine sehr einfache Logik im FPGA für Codieren und Decodiereren.

Für den Begriff "Software Defined Radio" hat keine offizielle Definition, aber die Bedeutung beschreibt ein System, dass dedizierte Hardware nur für die Sendung und Empfang des RF-Signals benutzt und Mehrzweckcomputer für die Signalverarbeitung von allen anderen verwandten Operationen.

"Radio in which some or all of the physical layer functions are software defined" - eine Definition von der Electrical and Electronic Engineers (IEEE) P1900.1 Gruppe[3].

An der Seite des Senders, das SDR System generiert das Signal im Basisband, dann schiebt es nach eine Zwischenfrequenz. Am Ausgang wird diese mit den Trägerfrequenz multipliziert - so wird das RF-Signal generiert. Der SDR Empfänger abtastet das RF-Signal, generiert die Zwischenfrequenz, dann wird die Zwischenfrequenz zum Basisband konvertiert und am Ende wird die Information nach der Regeln der Symbol-Bit-Mapping dekodiert.

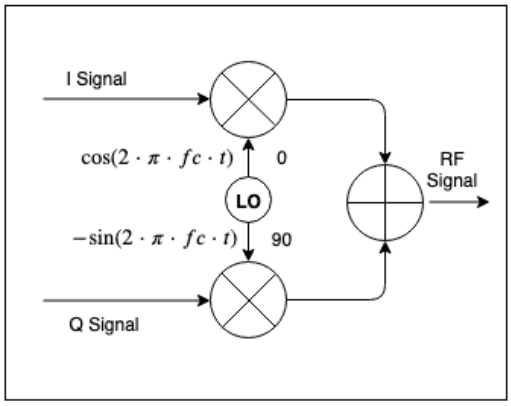
In meisten praktischen Umsetzungen des SDRs, nur das Basisband wird vom Mehrzweckcomputer generiert und verarbeitet. In diesem Arbeit werde ich auch eine FPGA benutzen, um den Decodierer für Pulsdauermodulation einzubauen.

3 Bekannte Verfahren für Datenübertragung

3.1 Modulationsverfahren

Heutigen Kommunikationsgeräte, in meisten Fällen, fassen den IQ-Modulator um. Das Konzept der Quadratursignale liegt an der Theorie der komplexen Zahlen. Quadrdatursignale sind zweidimensional und ihre Werte können in bestimmter Zeit als zwei Teile beschrieben werden – realen und imaginären Teil. An der Sprache der Kommunikationstechnik kann man auch die Begriffe In-phase und Quadrature-phase benutzen.

Abbildung 1: IQ Modulator



Normalerweise ist das I-Signal eine Kosinusfunktion und das Q-Signal ist eine Sinusfunktion. Mit einer Trägerschwingung *fc*, das Signal am Sender wird mit der folgenden Gleichung beschrieben:

$$S(t) = Si(t) * cos(2 * pi * fc * t) - Sq(t) * sin(2 * pi * fc * t)$$

Am Empfänger ist die Trägerscwingung *2fc* zwei mal höher. Die Abhängigkeiten zwischen Sinus und Kosinus gibt die folgenden Transformationen aus (für die reale und die imaginäre Komponente):

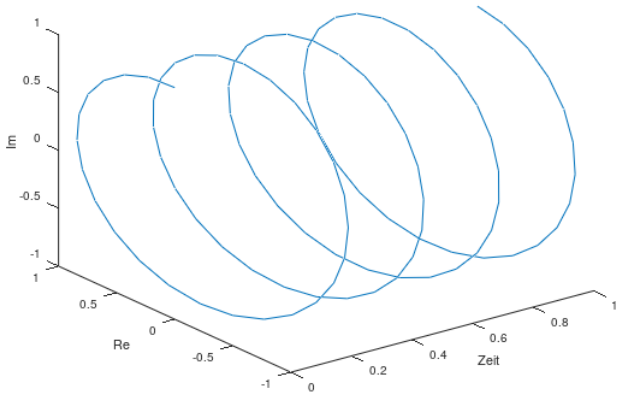
$$rI(t) = [Si(t) * cos(2 * pi * fc * t) - Sq(t) * sin(2 * pi * fc * t)] * 2cos(2 * pi * fc * t)$$
$$rI(t) = Si(t) * [1 + cos(4 * pi * fc * t)] - Sq(t) * sin(4 * pi * fc * t)$$
$$rQ(t) = -Si(t) * sin(4 * pi * fc * t) - Sq(t) * [1 - cos(4 * pi * fc * t)]$$

Die Gleichungen stellen vor, dass die I und Q Kanäle unabhänging nach der Summierung bleiben. An der Seite des Empfängers sind sie rekonstruierbare. Durch solche Summierung der I und Q Signalen können Amplitudenmodulation, Phasenmodulation und Frequenzmodulation umsetzbar werden. Nach dem Ausgang des IQ-Modulators, das komplexe Signal folgt die Forme der Euler-Formel, die eine komplexe Sinusoide beschreibt. Die Abhängigkeit zwischen der komplexen Exponentialfunktion und den trigonometrischen Funktionen (besonders Sinus und Kosinus) wird mit der Exponentialfunktion gegeben:

$$e^{j(2*pi*f*t)} = \cos(2 * pi * f * t) + j \sin(2 * pi * f * t)$$

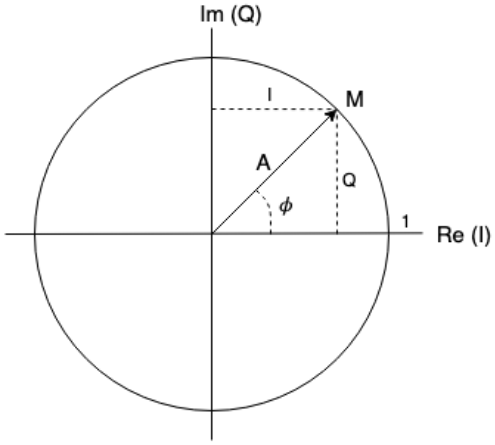
Im dreidimensionalen Raum, die Grafik dieser Funktion hat die folgende Darstellung:

Abbildung 2: Exponentialfunktion



Ein Verfahren für den praktischen Aufbau eines IQ Modulators liegt in den folgenden Abhängigkeiten zwischen der Amplitude und Phase der Konstellationspunkte. Ein solcher Punkt is *M*, der an der *Abbildung 3* dargestellt wird:

Abbildung 3: Abhängigkeiten der Phase und Amplitude



$$I = A \cdot \cos\phi$$
$$Q = A \cdot \sin\phi$$

Aus der dargestellten Formel kann man zusammenfassen, dass es zwei wichtige Parameter für die technische Implementierung gibt: Länge des A-Vektors und den Wert der Winkel φ.

Die einfachste Quadraturmodulation im Kontext des IQ-Modulators ist BPSK-Binary Phase Shift Keying. Es wird nur ein der Kanäle benutzt - entweder I oder Q. In diesem Arbeit legen wir eine Simualtion (im GNU Octave) als Beispiel vor. Wir werden dieses BPSK-Model benutzen um die Nachteile dieser standardisierten IQ-Modulationen nachzudenken und die Vorteile der SDR-basierten Pulsedauermodulation zu bestimmen.

3.2 Beispiel von BPSK

Der Vorgang der BPSK-Modulation wird in den folgenden Schritten verteilt:

- Die Konstellation der IQ Symbole wird definiert
- Das Signal im Basisband wird als Rechteckimpulse erzeugt
- Das Signal im Basisband wird nach der Konstellation nach zwischen den I und Q Kanälen aufgeteilt und kodiert
- Das Signal wird als Rechteckimpulse gefiltert - sodass das Signal verengt bekommt

- Die gefilterte Signale werden an der I und Q Kanälen geliefert (für BPSK wird nur I benutzt, an Q werden neutrale Werte festgelegt). Am Ende werden I und Q summiert und als eine Radiowelle geschickt.

Nachunter folgen die komplette Beschreibungen jedes Schrittes.

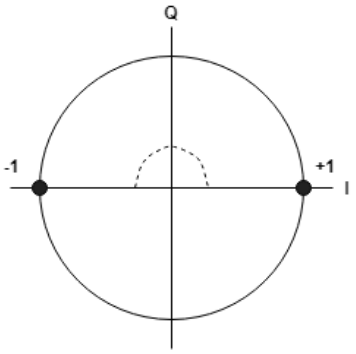
3.2.1 Definition der Konstellation für die IQ Symbole

Die Konstellation dient als eine Tabelle für Kodierung der Bit-Reihe. Sie gibt die Transformation von den Bits nach komplexen Zahlen an dem kartesischen Koordinatensystem aus (mit IQ Achsen). Für die BPSK (Binary Phase Shift Keying) Modulation, diese Zustände sind zwei:

Bits	I-Phase	Q-Phase
0	-1	0
1	1	0

Nach der Definition von der Tabelle, die Konstellation sieht wie so aus (ganz einfach):

Abbildung 4: BSPK Symbolen am IQ-Koordinatensystem

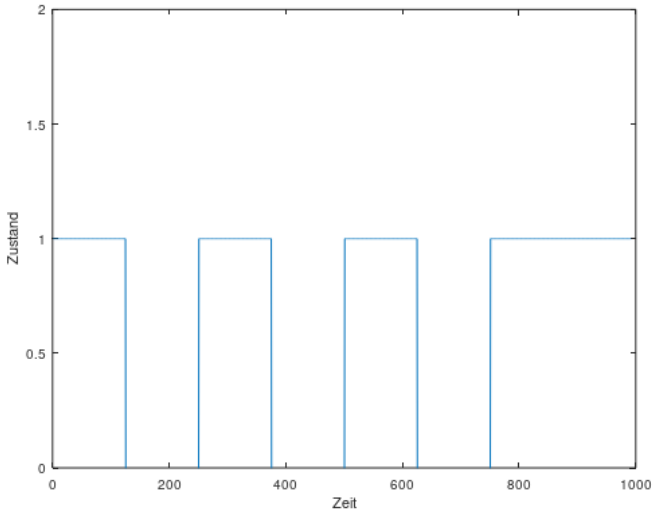


3.2.2 Signal im Basisband als Rechteckimpulse

Diese Impulse sind die direkte Vorstellung von den Bit-Reihen im Form eines Signals. Die BPSK dient als ein Modulationsverfahren in der digitalen Umfeld. Die Quelle des Signals vor dem Filter und dem Kodier hat eine quadratische Form. Für dieses Beispiel, die folgenden Bit-Reihe wird ausgewählt:

$$Bits = \{1, 0, 1, 0, 1, 0, 1, 1\}$$

Abbildung 5: Signal im Basisband - als Rechteckimpulse



3.2.3 Kodierung nach der BPSK-Konstellation

Alle Bits, die von den quadratischen Impulsen beschreibt werden, sollen nach Symbolen aus der Konstellation transformiert werden. Der Kodierungsprozess folgt zwei grundsätzliche Schritte: zuerst werden die Bits aus der Reihenfolge getrennt nach den entsprechenden I und Q Achsen (nur im Fall, wenn ein Symbol trägt mehr als ein Bit); danach werden die verteilte Bit-Reihen als neue Werte transformiert (aus dem IQ-Koordinatensystem). Für das Signal aus Abbildung 5, die trigonometrische Abhängigkeiten werden angewendet:

Die Bitfolge Bits = {1, 0, 1, 0, 1, 0, 1, 1} bekommt {1, -1, 1, -1, 1, -1, 1, 1}, nach der folgenden Transformation, die nach dem I-Kanal entspricht:

Für Bitwert 1:

A = 1 , $\phi = 0$
I = 1 * $\cos 0$
I = 1

Für Bitwert 0:

A = 1 , $\phi = \pi$
I = 1 * $\cos \pi$
I = -1

Für die BPSK-Modulation in diesem Beispiel wird nur das I-Kanal benutzt. Auf dieser Grund wird der Q-Kanal mit *0* (als neutrales Wert) ausgefüllt.

3.2.4 Signal im Basisband filtern

Jedes signal im Basisband soll gefiltert werden, um enger Frequenzkanal zu besetzen. Es gibt viele Verfahren und Algorithmen für die Implementierung eines Filters. Einer der bekanntesten sind Bessel Filter und Raised Cosine Filter.

Raised Cosine Filter (Kosinus-Roll-off-Filter) gehört zu der Kategorie der Nyquist-Tiefpassfiltern. Dieser Filter vermindert oder völlig beseitigt die Intersymbolinterferenz. Einer der Eigenschaften diese Filters is das so genannte Roll-off Faktor, dessen Wert zwischen 0 und 1 liegt. Das Roll-off Faktor kontrolliert wie viel Bandbreite für die Impulse konsumiert werden kann. Für die beste Effizienz zwischen SNR und Bandbreite, das Roll-off Faktor soll einen Wert von 0.22 zu 0.33[1] besetzen. Die Übertragungsfunktion des Filters in diesem BPSK-Beispiel, die im Octave modelliert wurde, hat die folgende Darstellung:

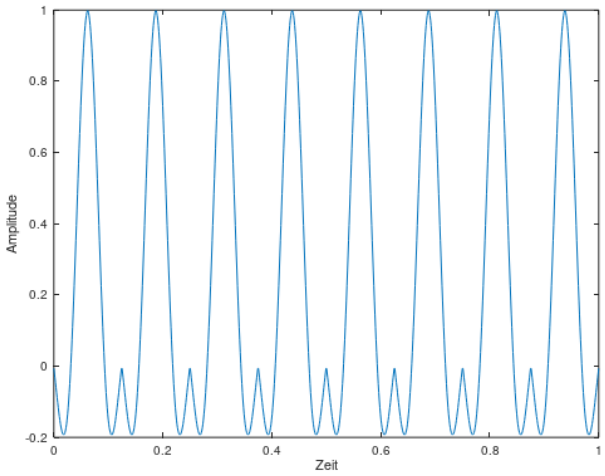
$$y(t) = \frac{\sin(\pi * \frac{t}{T})}{\pi * \frac{t}{T}} * \frac{\cos(\pi * \alpha * \frac{t}{T})}{1 - 4 * \alpha^2 * \frac{t^2}{T^2}}$$

In unserem Model der gewählter Wert für ist 0.25. Der komplette Code der Implementierung im Octave sieht wie so aus:

```
# Kosinus-Roll-off-Filter
function retval = raised_cosine(alpha, period)
  T = period/2;
  for t=1:period
    y(t)=(sin(pi*t/T)/(pi*t/T))*(cos(pi*alpha*t/T)/(1-4*alpha^2*t^2/T^2));
  endfor
  right = y;
  left = fliplr(right);
  retval = [ left right ];
endfunction
```

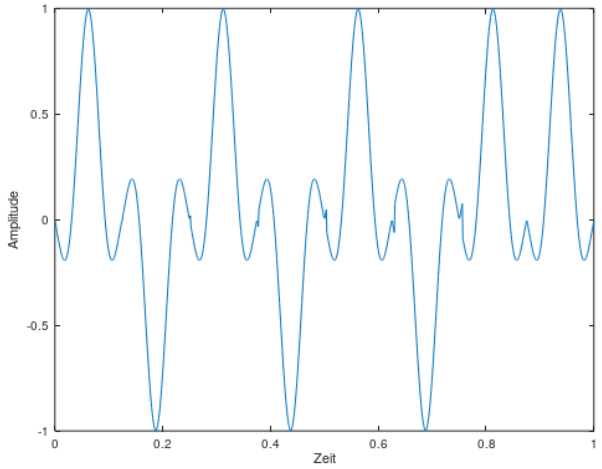
Darunter ist die Grafik von der Ausgabe dieser Funktion (wenn es als eine Welle dargestellt wird):

Abbildung 6: Kosinus-Roll-off-Filter



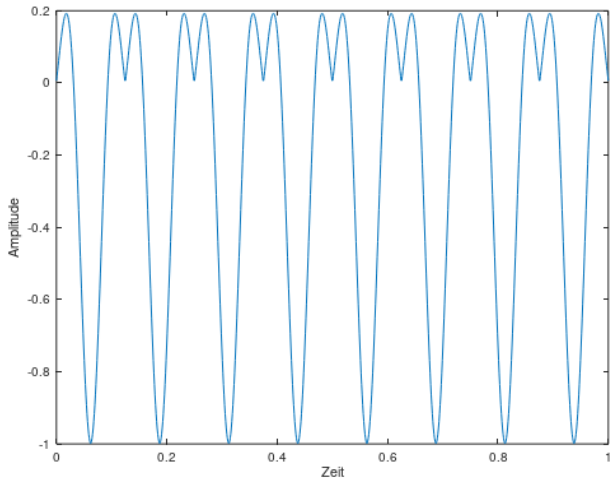
Nach Multiplikation der Bitfolge { 1, -1, 1, -1, 1, -1, 1, 1 } und der Welle vom Kosinus-Roll-of-Filter, das Signal hat die folgende Form:

Abbildung 7: I Signal, gefiltert



Obwohl die Q-Linie des IQ-Modulators nicht benutzt wird, wir transformieren die 0-Werte nach -1, indem das Q-Signal auch gefiltert nach der Multiplikation bekommt:

Abbildung 8: Q Signal, gefiltert



3.2.5 Ausgangssignal am IQ-Modulator

Die Beide I und Q-Kanäle sollen mit der Träger-Welle multipliziert werden (die nach 90 Grad verschoben sind), und dann werden diese modulierte Wellen addiert. Wie die nächste Abbildungen illustrieren, die Ausgabe des IQ-Modulators ist ein gemischtes (summiertes), komplexes Signal, dessen Komponente die I und Q sind. Das Folgende Grafiken stellen das modulierte Signal im 2D *Abbildung 9* und 3D *Abbildung 10* Koordinatensystemen dar:

Abbildung 9: Ausgangssignal, 2D

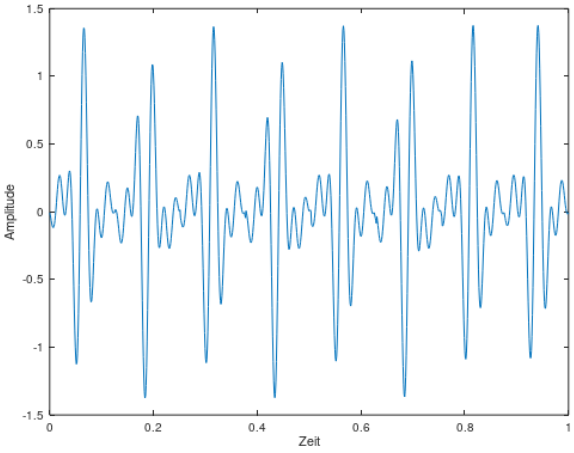
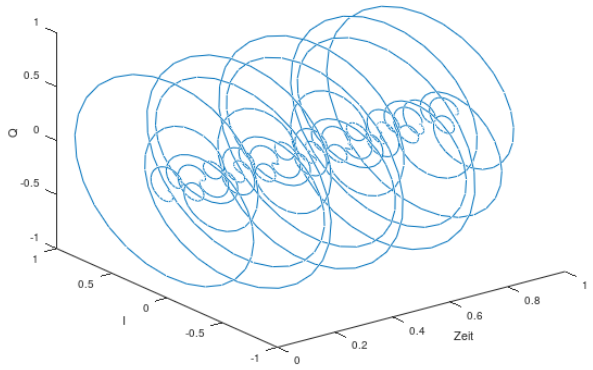


Abbildung 10: Ausgangssignal, 3D

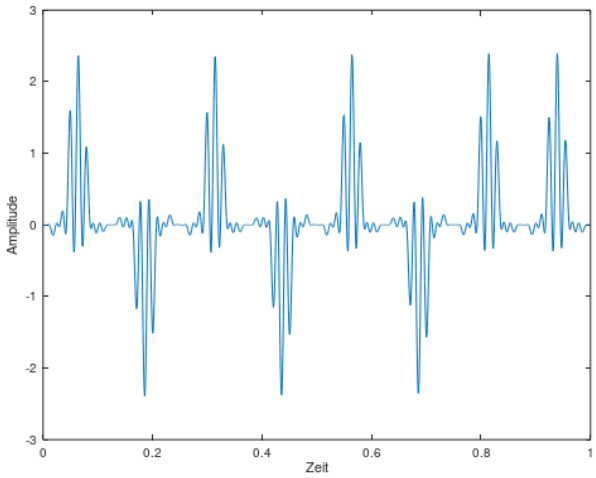


3.3 An der Empfängerseite

An der Empfängerseite, das Signal wird mit zwei mal höhere Frequenz als der Senderfrequenz multipliziert. Als Ergebnis, unerwünschte Bauteile erscheinen. Um das Quellensignal richtig rekonstruiert zu werden, man soll noch das Empfängersignal filtern. Die Bilder darunter schauen eines Verfahren, dass die Fourier-Transformationen verwendet.

Diese Methode ist auch für Modulation wie QPSK, 16QAM, etc anwendbar - nicht nur für unser Beispiel mit BSPK spezifisch.

Abbildung 11: I-Signal am Empfänger, nicht gefiltert



Die Fourier-Transformation schaut die unerwünschte Komponenten, die sollen gefiltert werden. Man kann diese Komponente wegwerfen. Die folgende Bilder illustrieren das Ergbennis:

Abbildung 12: Octave, abs(FFT) des I-Signals

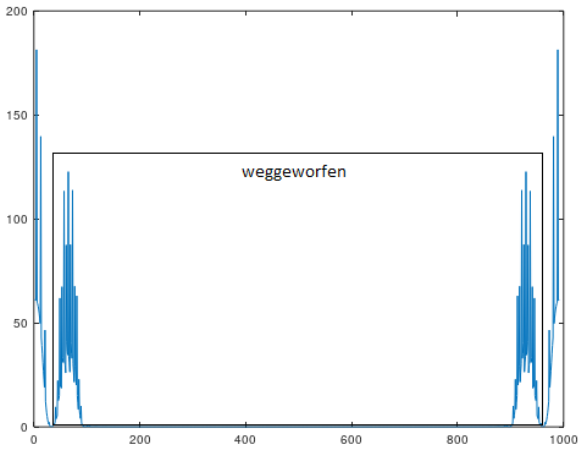


Abbildung 13: I-Signal am Empfänger, gefiltert (nachgestellt)

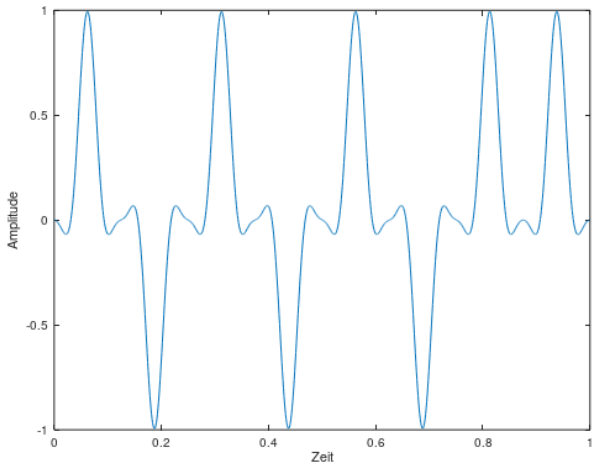


Abbildung 13 stellt die nachgestellte Form des I-Signals dar. Dieses Empfängersignal ist nach dem Quellensignal (Sendersignal) ähnlich. Nach Dekodierung, eine Bitfolge kann erzeugt werden.

3.4 Nachteile bestimmen

Obwohl der IQ-Modulator aus dem zwanzigsten Jahrhundert stammt und ist heutzutage noch immer ein Bauelement jedes Radio- Sender und Empfänger, IQ-basierte Modulationen - wie BSPK, 4-QAM (QPSK) und die QAM aus höherer Ordnung haben irgendwelche konzeptionelle Nachteile.

Alle von diesen IQ-Modulationen wandeln das Basisbandsignal von digitalen Impulsen in analoge Wellen nach der bestimmter Konstellation. Filter wie der Kosinus-Roll-off-Filter machen die Verarbeitung des Signals immer schwieriger für Realisierung mit digitalen Rechnern. Diese Transformationen bedeuten komplexe digitale Schaltungen und logische Blöcke, die trigonometrische Operationen führen können. Im SDR (Software Defined Radio) Aspekt, es geht immer schlimmer - Signalverarbeitung in Echtzeit bedeutet intensive Berechnungen an der Instruktionsebene, die viele Prozessor- und RAM-Ressourcen konsumieren.

Wenn es um den Abtastraten geht, jede Modulation heutzutage folgt das Nyquist-Shannon-Abtasttheorem, die sagt, dass die Abtastraten soll mindestens zweimal größer als die maximale Signalfrequenz sein. Die Kombination zwischen der mathematischen Analyse und dieses Theorem liegt in Prinzipen von allen klassischen Modulationen.

4 Pulsdauermodulation als eine Alternative

4.1 Beschreibung des Konzepts

In diesem Material versuche ich eine Alternative, die an den Prinzipen der Pulsdauernmodulation (PDM) liegt. Heutzutage wird die PDM häufig für Steuerung von Leuchtdioden und Monotoren verwendet. In Kombination mit der FPGAs und der Analog-Digital und Digital-Analog Wandler mit höhen Abtastraten, wir ändern die Anwendung der Pulsedauermodulation von Steuerung nach Datenübertragung. Dieser Verfahren basiert auf den folgengen Aspekten:

- Abtastrate - viel höher als die genuge Raten aus der Abtasttheorem
- Information innherhalb einer einzelnen Periode der Sinusschwingung codieren (im Basisband)
- Einfache Logik der Codier und Decodierer (ohne trigonometrische Berechnungen)
- Keine Notwendigkeit zum Filtern (das gespiegelte Signal wird auch gelasst)
- Keine Primitive für Synchronisierung

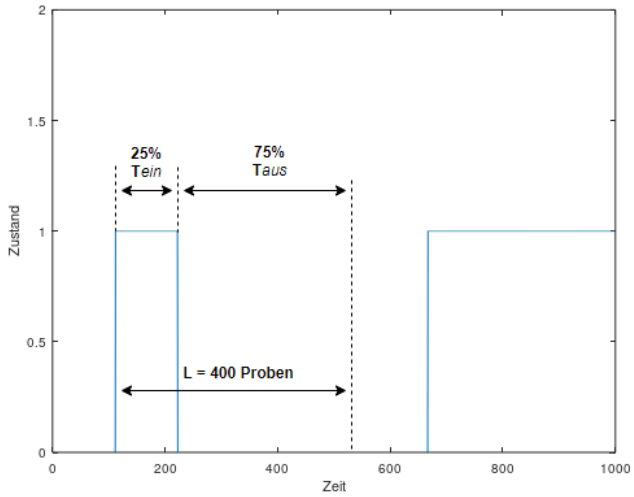
Für die Realisierung von Pulsedauermodulation soll die Periode einer quadratischen Welle oder Sinusschwingung mit bestimmter Länge verarbeitet werden. Im Kontext der Analog-Digital und Digital-Analog Wandler, die Länge der Periode soll in Proben gemesst werden. Die Formel, die die Abhängigkeit zwischen der Proben und Frequenz beschreibt, sieht wie so aus:

$$L = \frac{Abr}{Fb}$$

Mit L wird die Länge der Periode (in Proben) bezeichnet, Abr ist die übersamplierte Abtastrate und Fb ist die Frequenz im Basisband.

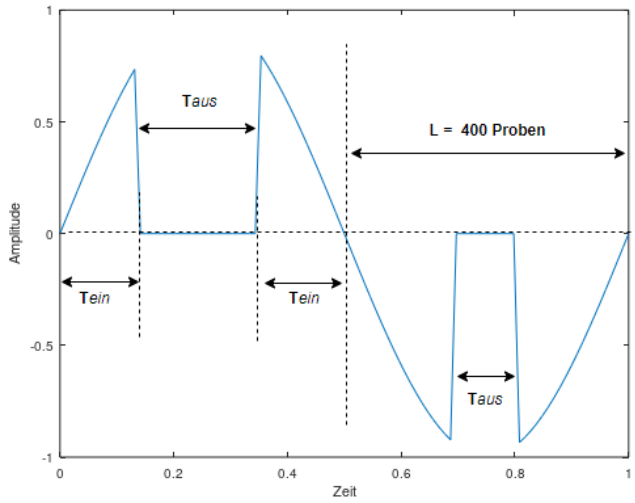
Die nächste Abbildungen schauen, wie die Periode verarbeitet werden kann, um die PDM zu realisiren. Für die quadratische Wellen, der *25% Tein* Teil wird gelasst und *75% Taus* wird nach Multiplikation der Perioder der quadratischen Wellen mit *0* erzeugt.

Abbildung 14: Pulsdauermodulation - quadratische Welle



Für die Sinusschwingung, *Taus* kann in der Mittel der Halbeperiode gelegt werden, zwischen zwei *Taus* Teilen:

Abbildung 15: Pulsdauermodulation - Sinusschwingung



Ich werde die Länge der *Taus* benutzen, um die Information durch PDM zu kodieren.

Weil das modulierte PDM Basisbandsignal als ein Radiosignal geschwinkt werden muss, ich berücksichtige zwei Richtugen meiner Forschung. Zuerst, ich werde das Spektrum von den quaradischen und den sinusförmigen Wellen im Basisband vergleichen. Je enger die Bandbreite, desto effektiver die Form des Signals. Und zweitens, man soll den maximal Anzahl von Symbolen bestimmen, die stabil übertragt werden kann - beziehungsweise die Menge von PDM-Impuls Längen.

4.2 Werkzeuge für die Umsetzung

4.2.1 Hardware

Das Modul *USRP B205-Mini* ist ein SDR-Entwicklungsboard von *Ettus Research*. Dieses Modul liefert die Funktionalität eines SDR in dimensionen einer Kreditkarte an. Der Frequenzbereich der Radio IC ist von 70MHz bis 6GHz. Die Verbindung mit einem Computer ist durch die USB3.0 Interface möglich.

Jetzt folgt die völlige Liste der Eigenschaften:

- Frequenzbereich: 70 MHz – 6 GHz
- Bis 56 MHz Kanal-Bandbreite
- Vollduplex Operation

- Programmierbare Xilinx Spartan-6 XC6SLX150 FPGA
- USB 3.0 Verbindung
- Synchronisierung mit 10 MHz Taktreferenz oder PPS Zeitreferenz
- GPIO und JTAG - für Kontrolle und Debug
- Dimensionen 83.3 x 50.8 x 8.4 mm
- USRP Hardware Driver^{TM} (UHD)
- GNU Radio Unterstützung und Integration

Abbildung 16: Block-Diagram von USRP B205-Mini

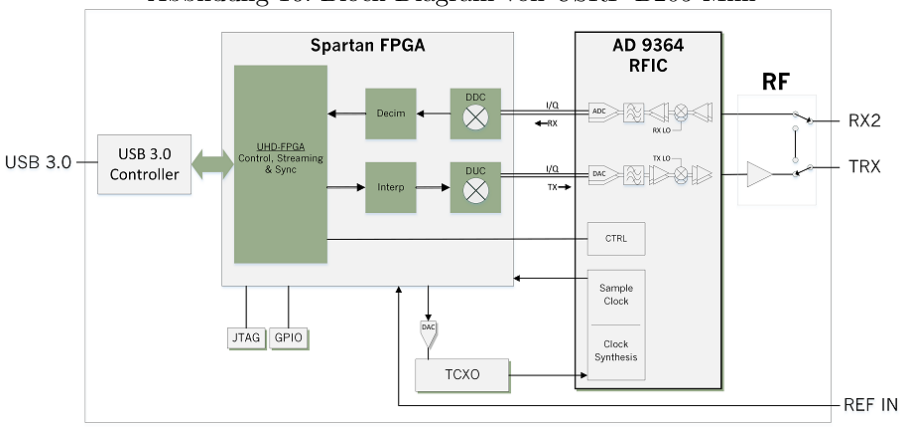
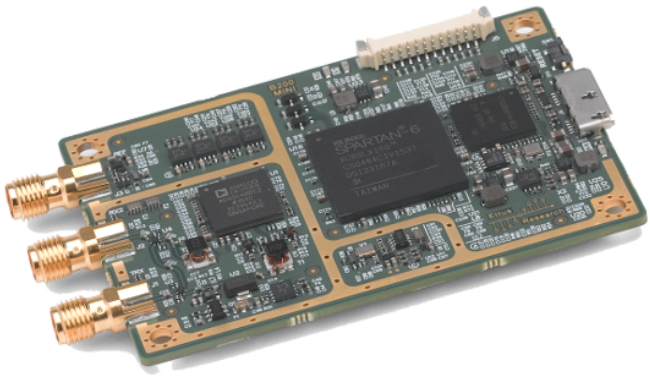


Abbildung 17: USRP B205-Mini[4]



Ich wähle dieses Modul aus, weil es ein *Spartan-6 XC6SLX150* FPGA enthält. In diesem FPGA werde ich meinen Decodierer integrieren.

4.2.2 Software

Für den konzeptionellen Beweiß wählen wir die GNURadio Anwendung. GNURadio ist einen Open-Source Softwarewerkzeug für Signalverarbeitung und Implementierung von Software Defined Radios. Es hat Modulstrukur und untestütz alle populäre SDR Boards auf dem Markt. Bei der Installation kommt GNURadio mit fertigen Blöcken, mit denen der Benutzer verschiedene Modulation, Filtern und Bündler anwenden kann.

GNURadio wird breit für verschiedene Forschungen von vielen Ingenieuren und Radiohobbyisten benutzt. Die Kombination zwischen SDR und GNURadio bekommt immer interessanter mit der aufkommenden Forschung im bereich 5G - man kann diese Technologie für Simulation und Experimenten mit Phasenrasterantennen, Design von Filtern und neue Methoden für Signalverarbeitung benutzten.

Dieser Werkzeug läuft als Software auf Linux-Betriebssystem - deswegen kann

man nicht nur an der x86-64 Architektur installieren, sondern an ARM-basierte Entwicklungsboards wie Raspberry PI. GNURadio wird (in meisten Fällen) nur für Steuerung des Signals im Basisband angewendet.

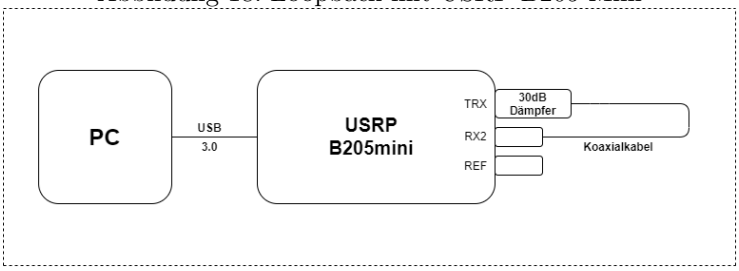
Nach der Experimenten mit GNURadio, ich mache die stabile Implementierung der PDM Kommunikation mit der nativen *USRP Hardware Driver (UHD)* Schnittstelle.

4.3 Konzeptioneller Beweiß

4.3.1 Topologie

Die nächste Abbildung stellt meine Einstellung dar. In Kürze, es wurde ein physischer Loopback mit einem einzigen USRP Gerät verwendet. Ein Dämpfer für die direkte Verbindung ist obligatorisch.

Abbildung 18: Loopback mit USRP B205-Mini



4.3.2 GNURadio Experiment

Für den Versuch wurden die folgenden Parameter im GNURadio eingestellt:

- Frequenz im Basisband: 10kHz
- Abtastungsrate: 5MSps (Megasamples pro Sekunde)
- Trägerfrequenz: 433MHz
- Bandbreite: 2000kHz

Ziel des Experiments ist erfolgreiche Übertragung von minimum 8 Symbolen in einer Halbperiode mit keinen Filterelementen und nur einem Kanal des IQ-Modulators (I-Kanal).

Aus der beschriebenen Parametern soll man die Länge einer Periode bestimmen. Um dieses zu berechnen, man benutzt die Formel:

$$P = \frac{Abr}{Fb} = 5 * \frac{10^6}{10^4} = 500S$$

Bemerkung: Die Werte von Abr und Fb wurden nach vielen empirischen Versuchen als eine relevante Abhängigkeit bestimmt. Das GNURadio Software läufte auf einer virtuellen Maschine mit 4GB RAM und Core i5-3320M Prozessor. Höhere Abtastraten und Basisbandfrequenzen haben das System mehrmals abgestürzt. Diese Werte wurden als zutreffend (und adäquat) für die Forschung zugelasst.

Aus der Formel kann man berechnen, dass die Länge der Halbperiode gleich 250 Proben ist. Die quadratische Welle besteht aus eine Halbperiode, die Niveau von *1* hat, und die andere Halbperiode ein Niveau von Null hat. Nach Multiplikation mit einem Vektor, der aus *0s* inzwischen eine gleiche Bitfolge von *1s* von links und rechts besteht, werden wir die Länge von dem modulierten Impuls verändern (oder *Taus* generieren). Ein Beispiel von einer Halbperiode mit 9 Elemente (Proben) wird gegeben:

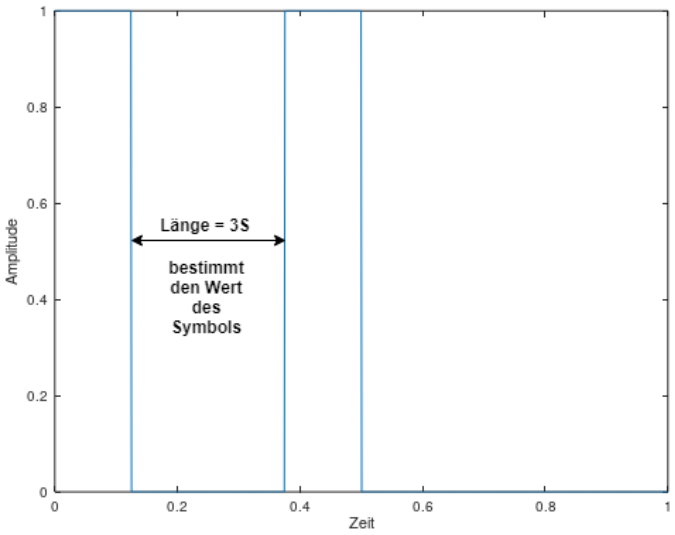
$$BitfolgeHalbPeriode = \{1, 1, 1, 1, 1, 1, 1, 1, 1\}$$

$$BitfolgeMultVektor = \{1, 1, 1, 0, 0, 0, 1, 1, 1\}$$

$$ModuliertenImpuls = BitfolgeHalbPeriode \cdot BitfolgeMultVektor$$

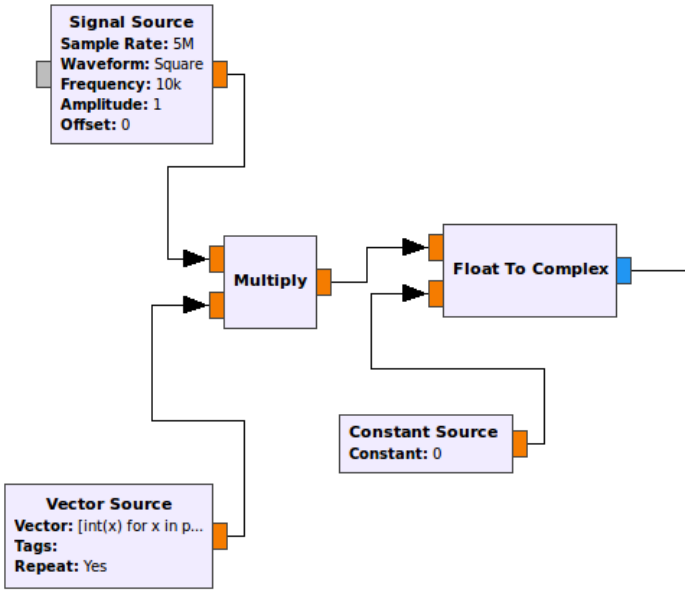
$$ModuliertenImpuls = \{1, 1, 1, 0, 0, 0, 1, 1, 1\}$$

Abbildung 19: Modulierten Impuls {1, 1, 1, 0, 0, 0, 1, 1, 1}



Die Implementierung im GNU Radio sieht wie so aus:

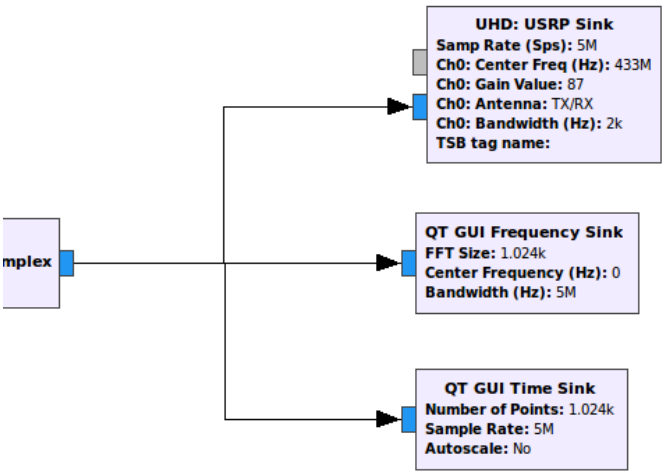
Abbildung 20: GNURadio: Multiplikation eines Quellensignals und Vektors



Der *Signal Source* Block generiert eine quadratische Welle, die mit der Ausgabe des *Vector Source* Blocks multipliziert wird. Der Block *Float To Complex* setzt die Generation der komplexen Zahl um. An diesem Diagramm wird nur den I-Kanal benutzt. Um die Neutralität des Q-Kanals zu garantieren, erfülle ich den zweiten Eingang des *Float To Complex* Blocks mit *Constant Source* Block, der die Konstante 0 generiert.

Der Ausgang des *Float To Complex* Blocks schickt die digitate Beschreibung des Signals nach dem UHD: *USRP Sink* Block.

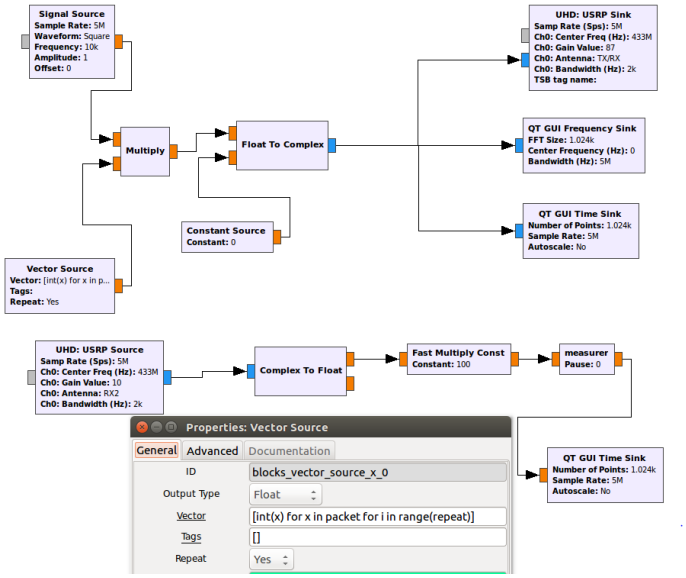
Abbildung 21: GNU Radio, USRP Sink Block



An der Seite des Empfängers, das Signal folgt eine einfache Blockkette, die das Niveau mit Faktor 100 multipliziert - *Fast Multiply Const.* Die Ausgabe dieses Blocks wird mit einem benutzerdefinierten Block *Measurer* verbundet. Der *Measurer* sammelt die erste 4096 Proben in Echtzeit und dann schreibt die Information in einer Datei.

Die Multiplikationsvektoren, die im Experiment benutzt wurden, haben eine Länge von 250 Elementen. Anstatt die ganze Bitfolge manuell einzugeben, ich definiere nur 25 Bits und dann ein Ausdruck (im Python) wiederholt jedes Bit zehnmal.

Abbildung 22: Vector Source Block wiederholt jedes Bit zehnmal



4.3.3 Definition der Symbolen

Für den PDM-Versuch werden wir insgesamt 9 Symbole definieren. Die folgende Tabelle beschreibt die Mapping von den Symbolwerten und den Bitfolgen:

Symbol	Bit-Reihenfolge (jedes Element wird zehnmal wiederholt)
1	11111111111111111111111111 11111111111100011111111111
2	11111111111111111111111111 11111111110000011111111111
3	11111111111111111111111111 11111111100000011111111111
4	11111111111111111111111111 11111111000000011111111111
5	11111111111111111111111111 11111110000000011111111111
6	11111111111111111111111111 11111100000000011111111111
7	11111111111111111111111111 11111000000000011111111111
8	11111111111111111111111111 11110000000000011111111111
9	11111111111111111111111111 11100000000000011111111111

4.3.4 Ergebnisse mit rechtecken Impulsen

Als Ergebnisse wurden drei Grafiken generiert: Signal an der Seite des Empfängers über die Laufzeit von 4096 Elementen (Proben), Signalspektrum und dasselbe Signal von 4096 nach 1000 Proben gezoomt. Weil es keinen Filter verwendet wurde, das Spektrum des Signals soll breiter als die sinuosoidale Impulse sein.

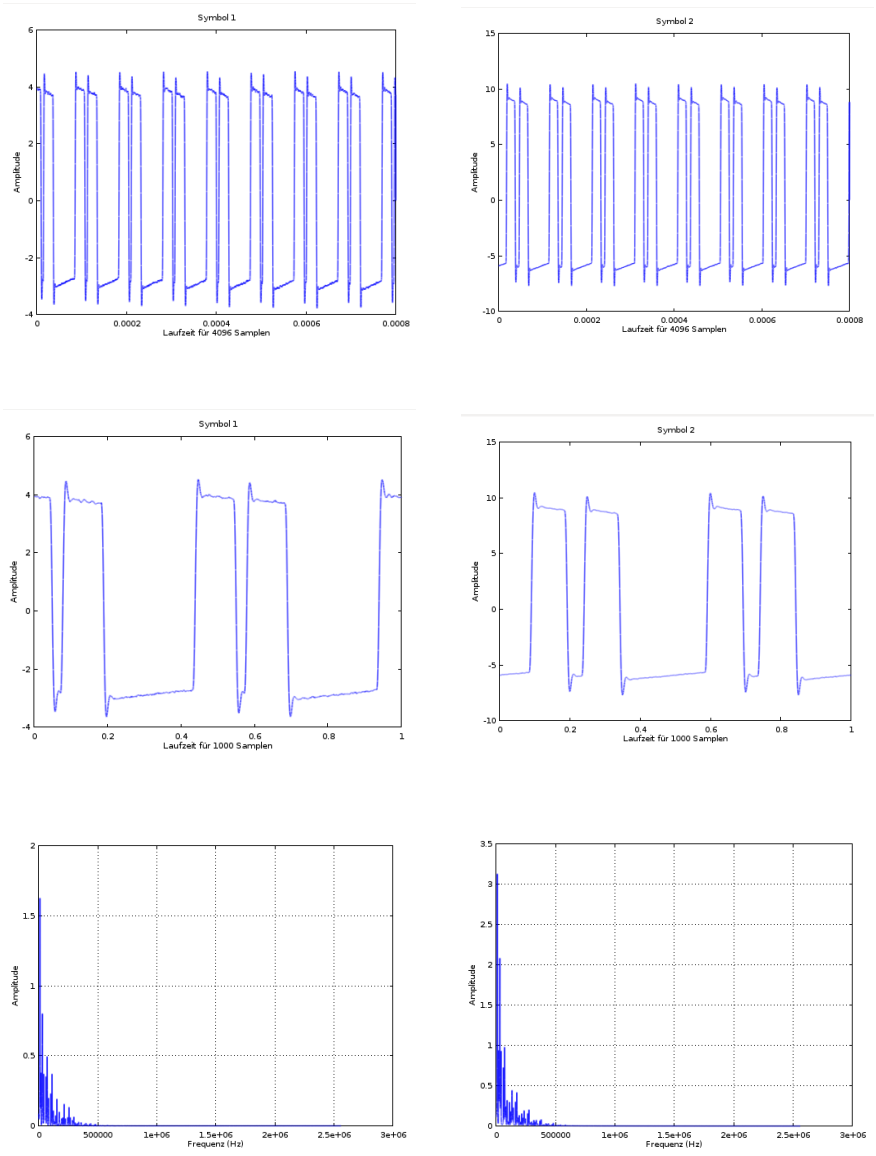


Abbildung 23: Symbol 1 (links) und Symbol 2 (rechts) - Form und Spektrum

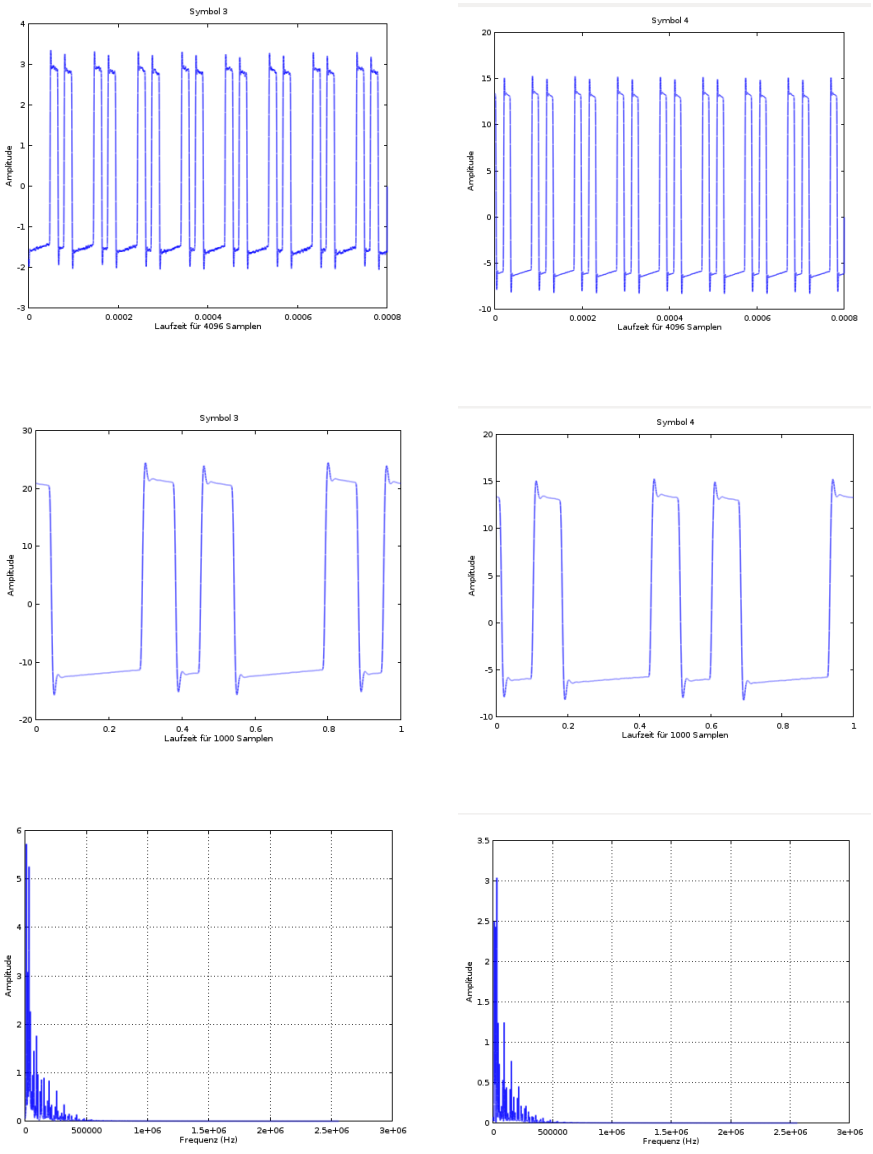


Abbildung 24: Symbol 3 (links) und Symbol 4 (rechts) - Form und Spektrum

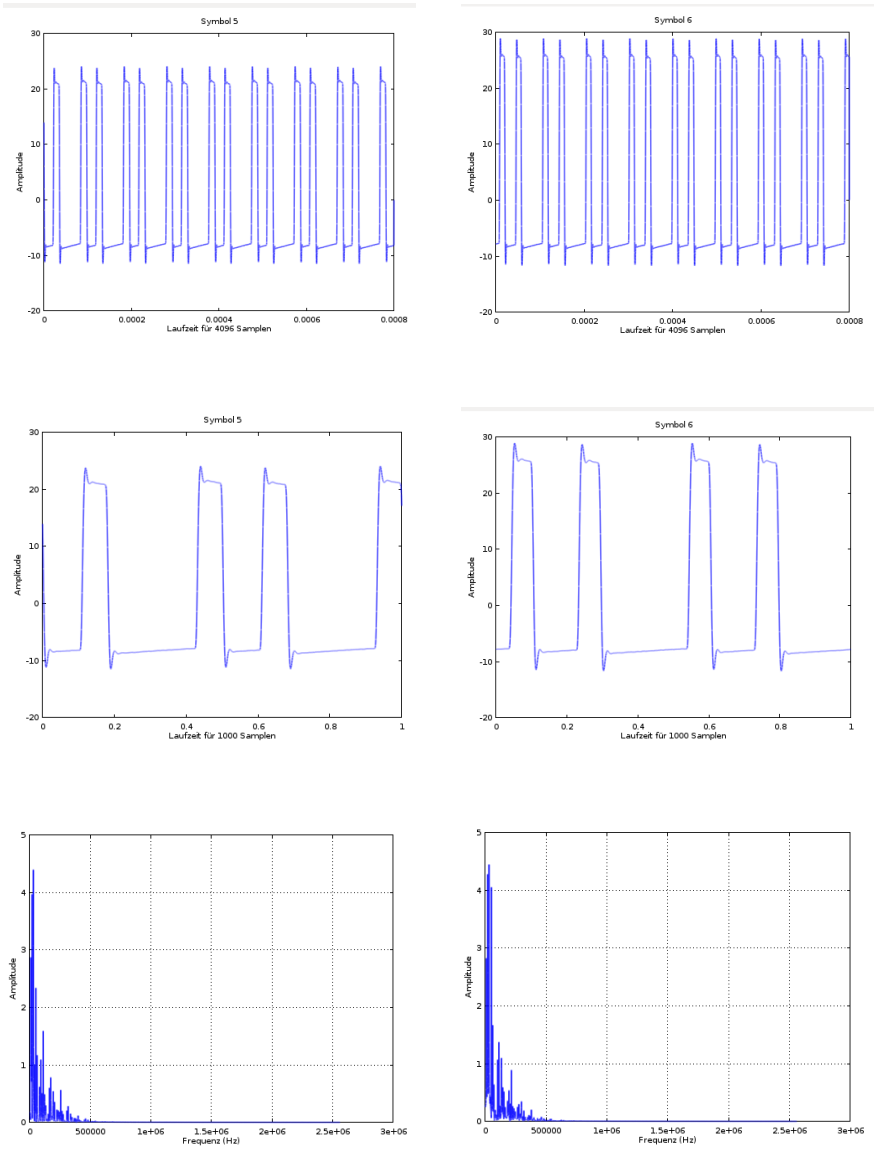


Abbildung 25: Symbol 5 (links) und Symbol 6 (rechts) - Form und Spektrum

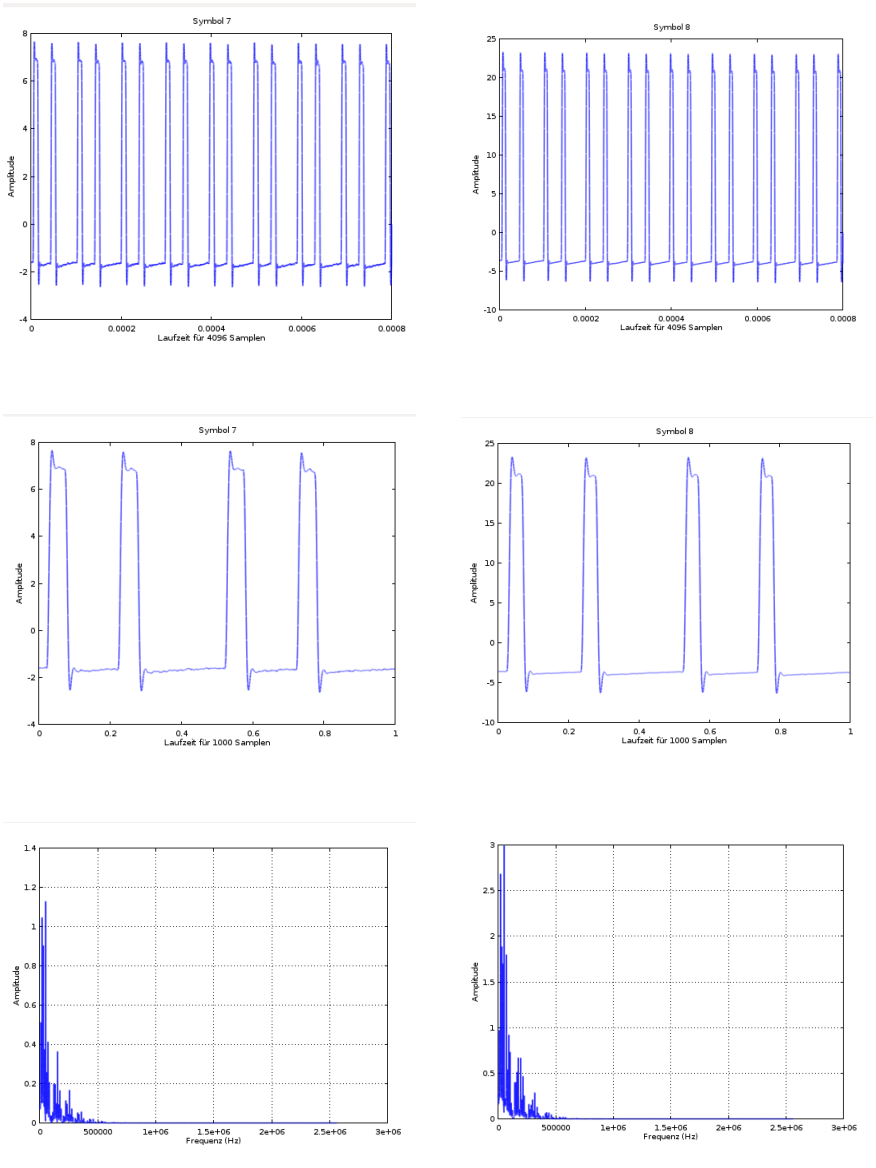


Abbildung 26: Symbol 7 (links) und Symbol 8 (rechts) - Form und Spektrum

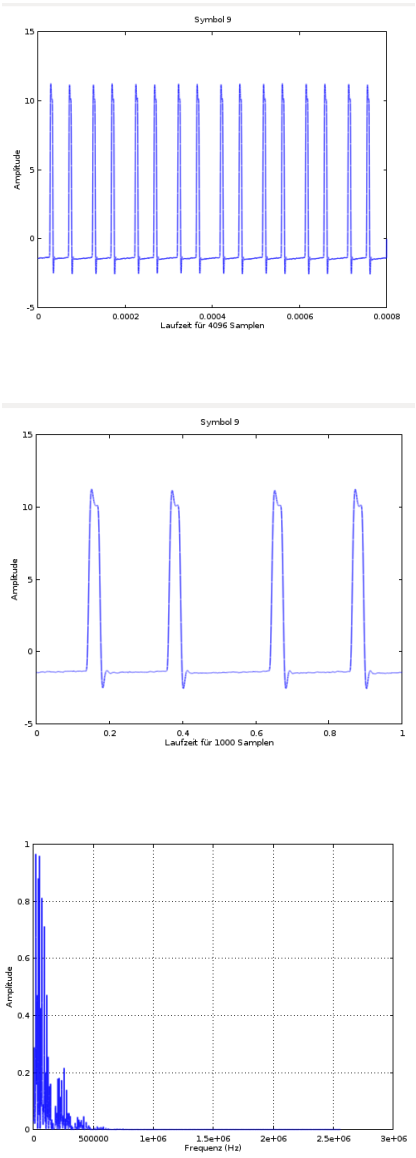


Abbildung 27: Symbol 9 - Form und Spektrum

Wie die Figuren schauen, alle neun Symbole wurden erfolgreich gesendet und emfängt. Wichtig hier ist, dass wir nur die Halbperiode des Impulses modulieren, die im Trägersignal auf *1* stehen. Die andere Halbeperide, die normalerweise auf *0* sheht, kann auch moduliert werden - um mehrere Symbole zu übertragen. Interessant ist auch das Spektrum - obwohl es viele Komponente von 0 bis 750kHz hat, die eingestellte Bandbreite im GNURadio von 200kHz ist für die verwendbare Teile des Signals genug - man kann klar die Dauer des modulierten Impulses bestimmen.

Darunter wurde dasselber Versuch mit Sinusschwingungen anstatt Rechteckimpulsen dargestellt. Als Ergebnisse wurden noch einmal drei Grafiken generiert für jedes Symbol: Signal an der Seite des Empfängers über die Laufzeit von 4096 Elementen (Proben), Signalspektrum und dasselbe Signal von 4096 nach 1000 Proben gezoomt.

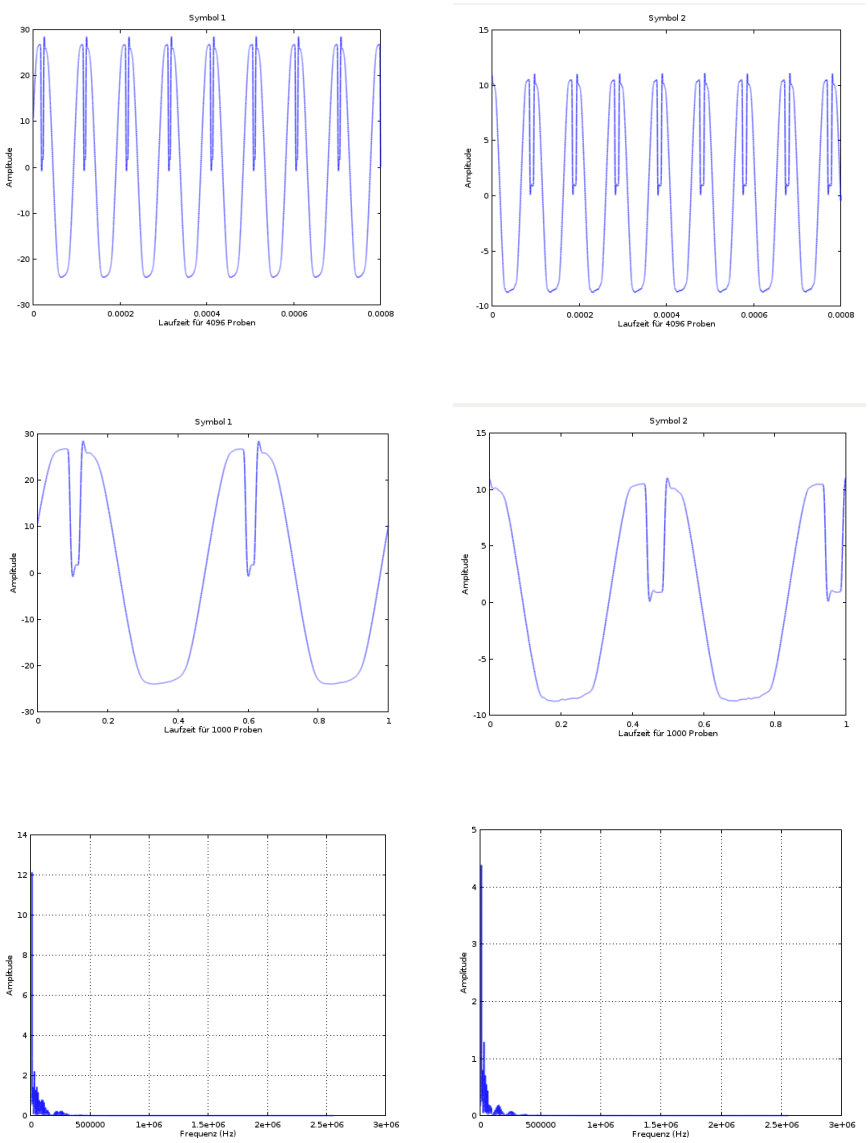


Abbildung 28: Symbol 1 (links) und Symbol 2 (rechts) - Form und Spektrum

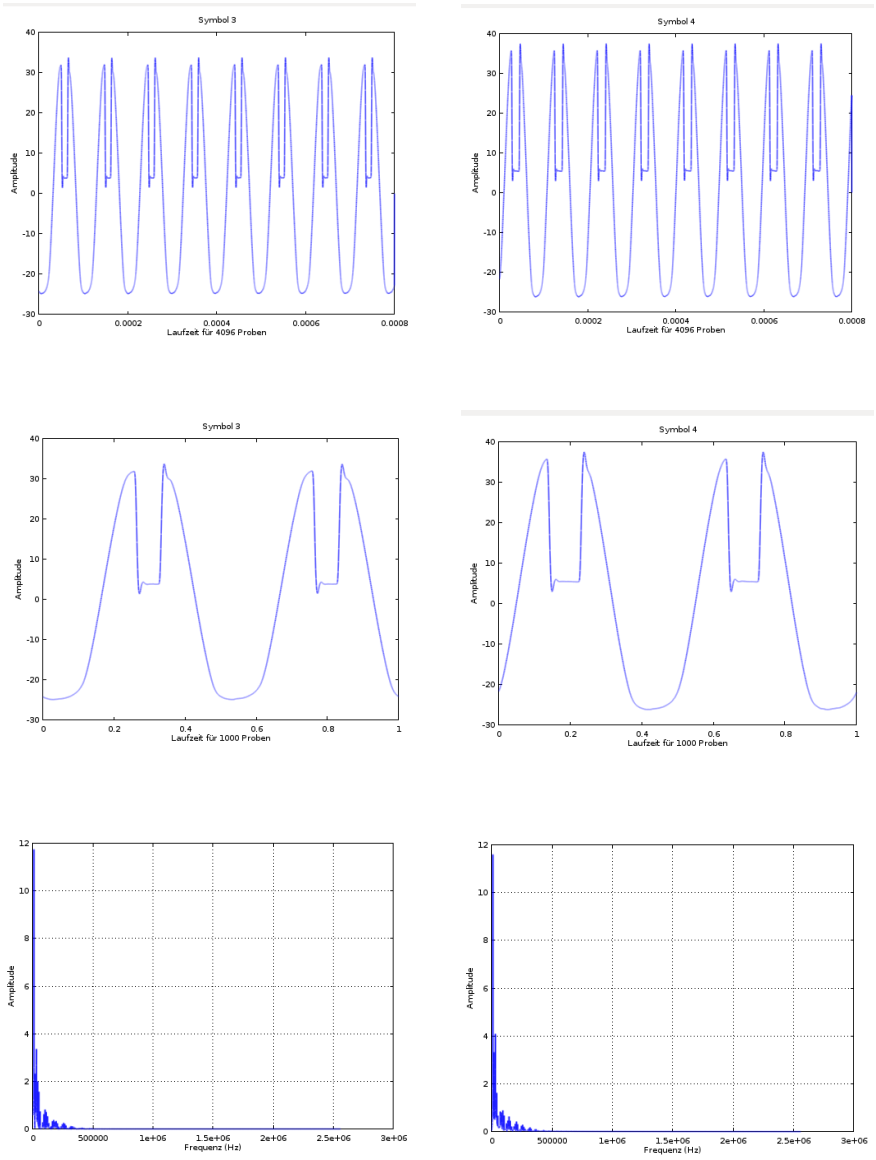


Abbildung 29: Symbol 3 (links) und Symbol 4 (rechts) - Form und Spektrum

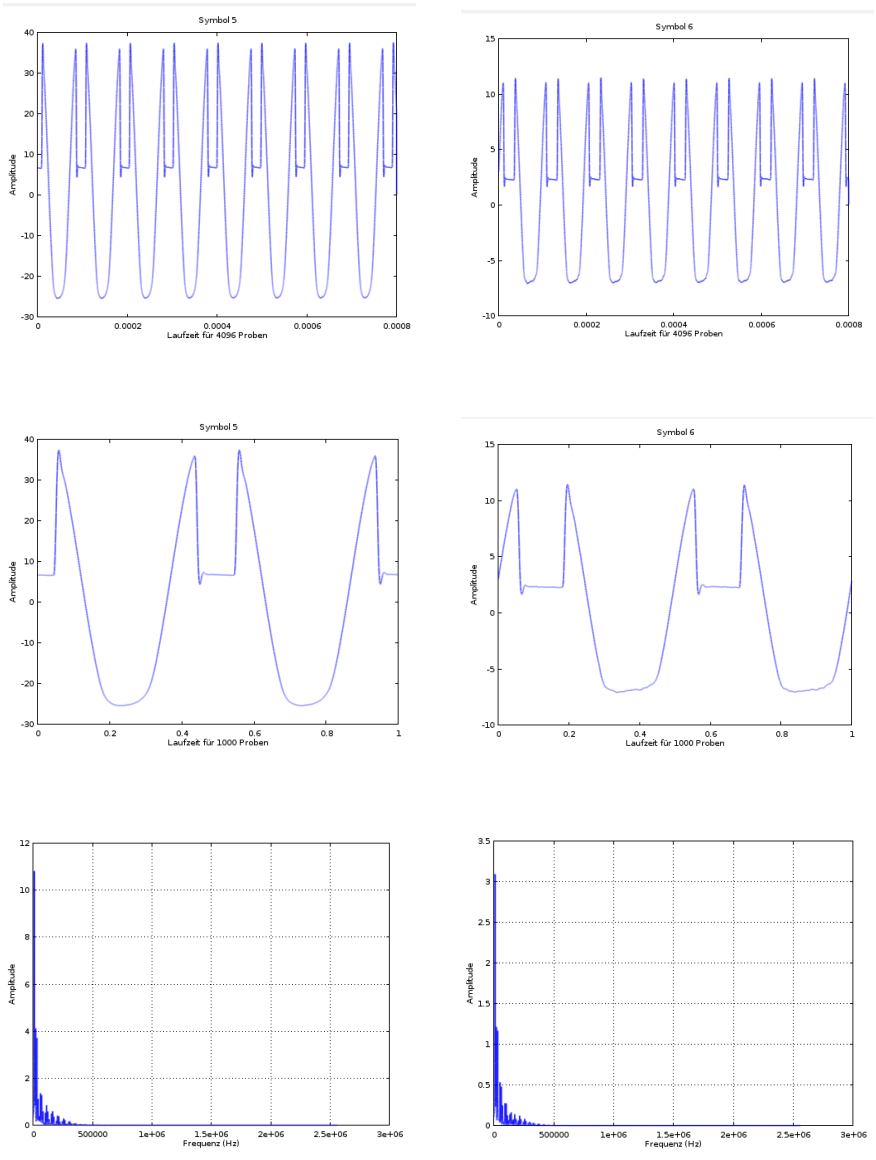


Abbildung 30: Symbol 5 (links) und Symbol 6 (rechts) - Form und Spektrum

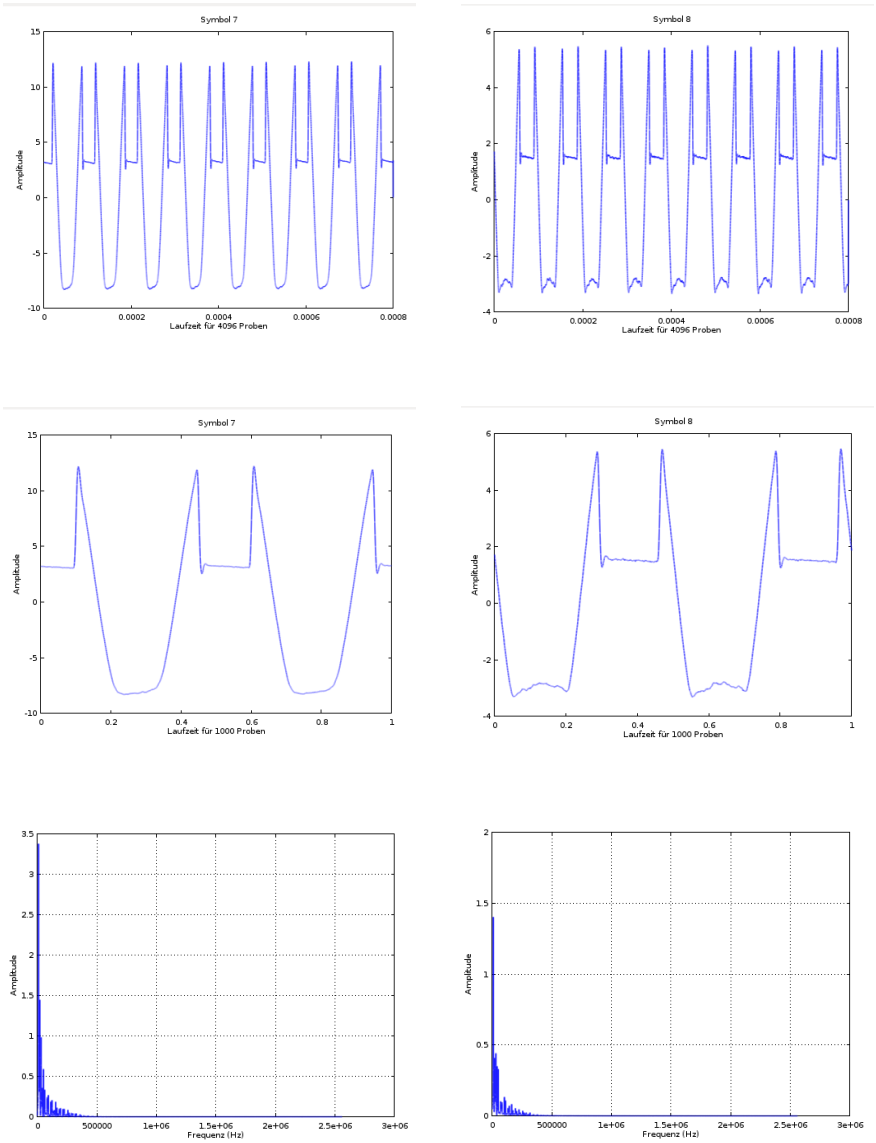


Abbildung 31: Symbol 7 (links) und Symbol 8 (rechts) - Form und Spektrum

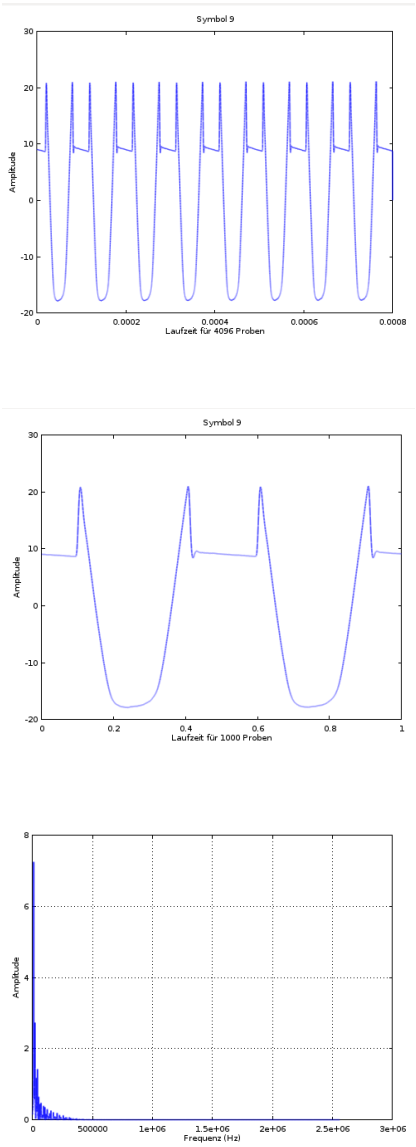


Abbildung 32: Symbol 9 - Form und Spektrum

In diesem Fall die sinusförmige Impulse beschränken das Spektrum. Der integrierte Generator für Sinusschwingungen dient als ein Filter im Basisband. Die eingestellte Bandbreite im GNU Radio von 200kHz ist auch für diesen Versuch adäquat. Alle Symbolen wurden erfolgreich empfängt und lesbar.

4.3.5 Stabilität des Empfangs

Von den Abbildungen ist zu entnehmen, dass die sinusoidale Impulse verwenden weniger Bandbreite. Wie oben erwähnt, das Signal ist an der Empfänger lesbar mit Frequenz von 433MHz und Bandbreite von 200kHz, mit keinen Filtern im Basisband. Die Länge des Impulses bleibt wiedererkennbar in den beiden Fällen. Im Bezug auf die Stabilität, die Experimente demonstrieren die folgende Ergebnisse als Erfolgsrate von zehn Messungen:

Symbol	Rechtecke Impulse	Sinusoidale Impulse
1	8/10	6/10
2	6/10	8/10
3	7/10	7/10
4	7/10	8/10
5	6/10	7/10
6	7/10	6/10
7	6/10	6/10
8	6/10	7/10
9	7/10	6/10

Alle diese Werte können verbessert werden, wenn eine Synchronisierung implementiert wird. In diesem bestimmten Beispeil, die Sender und Empfänger Blöcke arbeiten im Kontext der Prozessen des Betriebssystems - die GNURadio Anwendung im Linux. Es gibt keine Synchronisierungsprimitiven am Software-Ebene.

Mit den beschriebenen Nachteilen, man kann zusammefassen, dass die Ergebnisse nicht stabil sind, aber das Konzept ist gültig. Für die tatsächliche Implementierung der PDM soll man die übermäßige Abstraktionen im Software eliminieren.

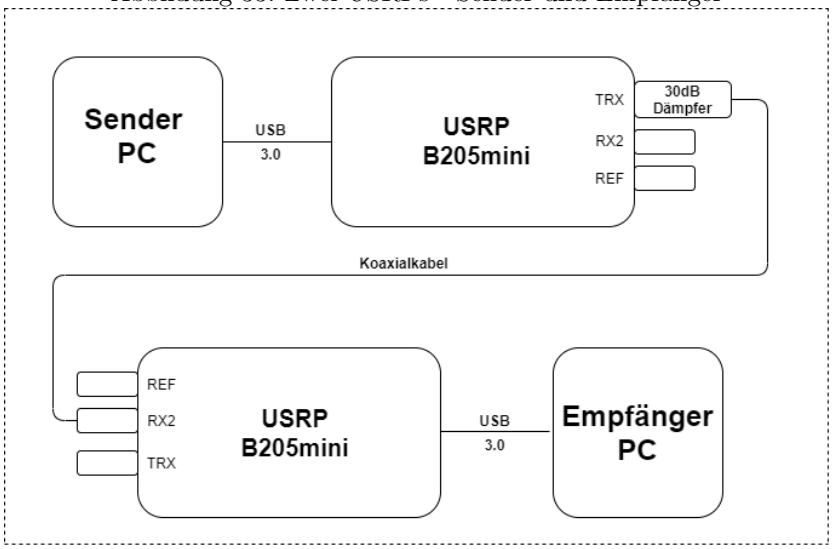
4.4 Tatsächliche Umsetzung

Für die Realisierung der PDM werde ich die native Schnittstelle verwenden. Obwohl die GNURadio Anwendung breit einsetzbar für fast alle existierende SDR Geräte ist, braucht man für die PDM kodierung mehr spezifische Kontrolle auf Hardware und Driver-Ebene. Auf dieser Grund sollte ich die Schnittstelle von USRP benutzen - *UHD (USRP Hardware Driver)*. Die GNURadio Module wie *USRP Sink* und *USRP Source* werden mithilfe dieser Bibliothek realisiert. Die UHD Schnittstelle kommt mit irgendwelchen Beispielen, die benutzt und modifiziert werden können, um die PDM Impulse zu kodieren. Zuerst soll ich das Konzept in Software überprüfen. Danach werde ich versuchen, die Decodiererlogik (am Empfänger) in der FPGA zu implementieren.

4.4.1 Topologie

In diesem Fall wurden zwei USRP B205mini miteinander verbunden in der Topologie, die das Diagramm darstellt:

Abbildung 33: Zwei USRPs - Sender und Empfänger



Der Sender USRP wurde an den folgenden PC ausgeschlossen:

- Prozessor: Intel Core i5-3320M @ 2.60GHz
- RAM: DDR3 @ 1.60GHz, 8GB
- Betriebssystem (Host): Windows 10
- Betriebssystem (VM): Ubuntu 19.10 an VMware Workstation 15

Und der Empfänger USPR wurde an den PC ausgeschlossen:

- Prozessor: AMD Athlon 200GE @ 3.20GHz (Ryzen)
- RAM: DDR4 @ 2.40GHz, 16GB
- Betriebssystem (Host): Windows 10
- Betriebssystem (VM): Ubuntu 19.10 an VMware Workstation Pro 15

4.4.2 UHD - Installierung von Quellcode

Die UHD Schittstelle soll aus der Git herunterladet und kompiliert werden. Darauf folge ich die Instruktionen aus der Dokumentation:

Vorbereitung der *cmake* Umwelt:

```
cd <uhd-repo-path>/host
mkdir build
cd build
cmake ../
```

Jetzt kompilieren und installieren:

```
make
make test
sudo make install
sudo ldconfig
```

4.4.3 Einstellungen von Sender und Empfänger

ich führe die Versuche mit Frequenzen und Abtastraten durch, die für meine Computerhardwareressourcen adäquat sind und können stabil in Lange der Zeit bleiben. Nach vielen Experimenten wurde zusammengefasst, dass die Sinusoide im Basisband soll 10kHz sein. Die Abtastraten sollen Werten von 1MSps (am Sender) und 2MSps am Empfänger eingestellt werden.

Der Wert von 1MSps für den Sender wurde gewählt, um eine Länge von 100 Elementen des Pulses zu definieren. Das bedeutet, dass die Periode der Sinuswelle gleich 100 Proben ist (oder 50 für jede Halbperiode). Die PDM-modulierte Symbole werden dadurch dasselbe einfache Methode generiert, wie in den Beispielen mit GNURadio - die Elemente im Mittel der Halbperiode mit *0* multiplizieren.

4.4.4 Sinusschwingung mit I/Q und nur I Komponenten

Ich benutze die folgenden UHD-Anwendungen, zuerst ohne Änderungen. Die Frequenz von 433MHz wurde benutzt, weil sie zu dem ISM-Bereich gehört:

Empfängerseite:

```
sudo ./rx_samples_to_file --freq 433e6 --rate 2e6 --bw 200e3
--gain 0 --nsamps 0 --spb 1000
```

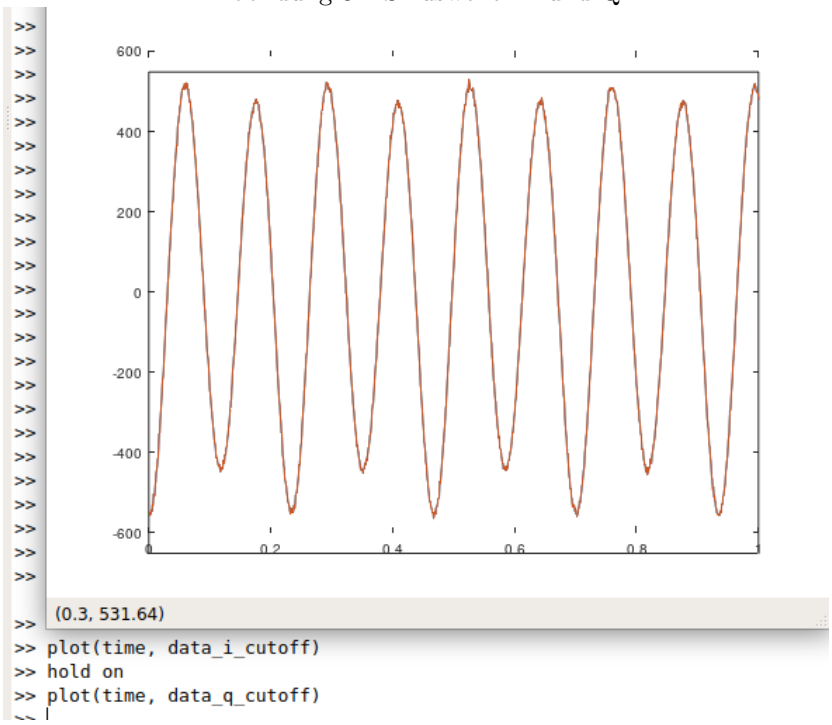
Senderseite:

```
sudo ./tx_waveforms --rate 1e6 --freq 433e6 --ampl 0.6
--gain 70 --bw 1e6 --wave-type SINE
--wave-freq 10e3 --spb 100 --otw sc16
```

Das *–spb* Argument bedeutet *samples per buffer*. Innerhalb dem Code, *tx_waveforms* ruft den UHD *Stream* an, der einen Puffer mit N Elementen schickt. Um ganz genau zu sein, ich stelle die Länge der Impulsperiode. Mit *–rate* wird den Abtastsrate eingestellt.

Die erwartete Sinuswelle sieht wie so aus:

Abbildung 34: Sinuswelle - I und Q

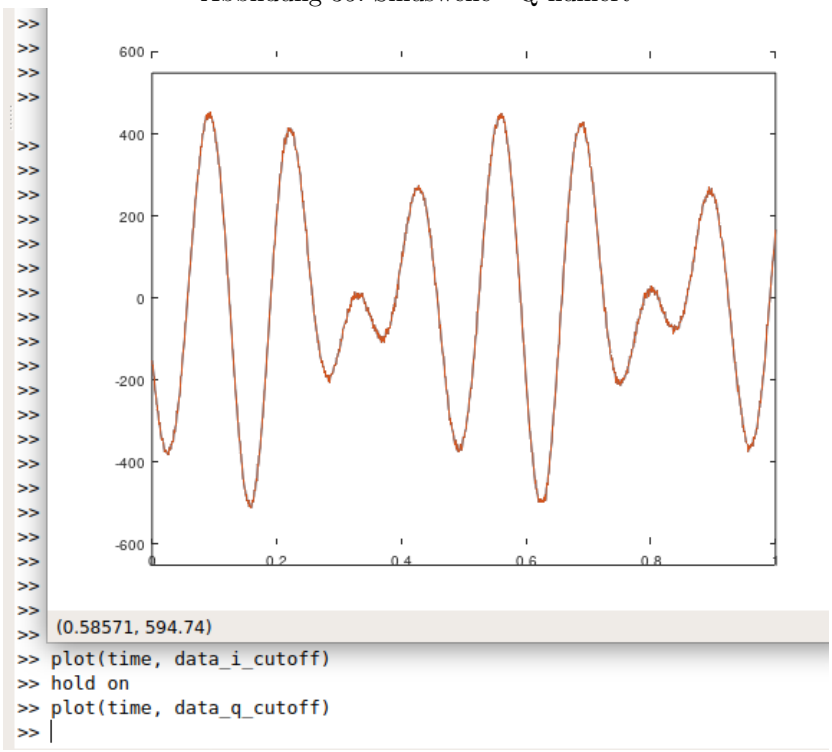


Als nächstes Schritt, das Q-Komponent wurde nulliert, sodass nur das I-Komponent benutzt wird - ich habe empirisch überprüft, dass die Zustand der PDM-codierten Impulse bleiben stabil, nur wenn der Q-Kanal auf Null gesetzt wird. Auf dieser Grund wurden die folgenden Zeilen in der Code addiert:

```
// Zero the Q component
for (size_t n = 0; n < buff.size(); n++) {
        buff[n] = std::complex<float>(buff[n].real(), 0);
}
```

Die Grafik der Sinuswelle am I-Kanal ist das folgende:

Abbildung 35: Sinuswelle - Q nulliert

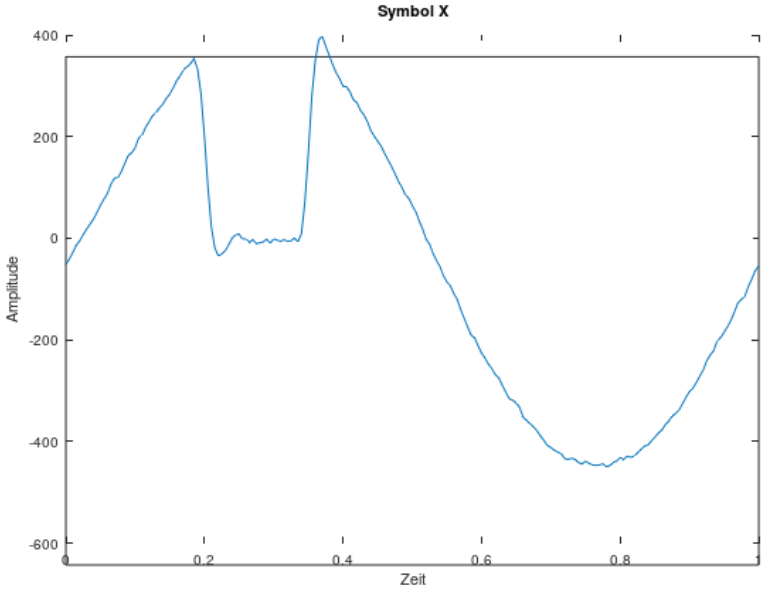


Aus dem Bild ist es sichtbar, dass es zwei Periode mit maximaler Amplitude gibt, die für die PDM-Codierung geeignet sind. Dies wurde ohne Kalibrierung oder Synchronisierung mit externer Frequenzquelle zwischen den SDR Geräten erreicht. Ich vermeide weitere Einstellungen, weil mein Ziel ist, die optimale Ergebnisse *ohne Filterung, komplizierte Kalibrierung oder zentralisierte Synchronisierung* vorzustellen.

4.4.5 Kodierung und Empfang der Symbolen

Wie in den GNURadio Experimenten, die Symbolen haben die Forme aus dem Bild:

Abbildung 36: Forme des PDM-Symbols



Ich habe die folgenden PDM-codierten Impulse mit verschiedenen Längen erfolgreich generiert:

Abbildung 37: Tabelle der Symbolen

Symbol	Sample links = 0	Sample rechts = 0	Länge (am Sender)	Länge (am Emfänger)	Spektrogramm (Name des Bildes)
S1	18	31	13	25	Spektrum S1
S2	18	32	14	27	Spektrum S2
S3	17	32	15	29	Spektrum S3
S4	17	33	16	31	Spektrum S4
S5	16	33	17	33	Spektrum S5
S6	16	34	18	35	Spektrum S6
S7	15	34	19	37	Spektrum S7
S8	15	35	20	39	Spektrum S8
S9	14	35	21	41	Spektrum S9
S10	14	36	22	43	Spektrum S10
S11	13	36	23	45	Spektrum S11
S12	13	37	24	47	Spektrum S12
S13	12	37	25	49	Spektrum S13

Einer der Beispielanwendungen von der UHD Schnittstelle ist einen Software-FFT Spektrumanalysator, der die *ncurses* Bibliothek für die Visualisierung der Spektrogramm benutzt. Für jedes Symbol aus der Tabelle wurde eine Spektralanalyse durchgeführt. Das sind die Ergebnisse:

Abbildung 38: Spektrum von S1

Abbildung 39: Spektrum von S2

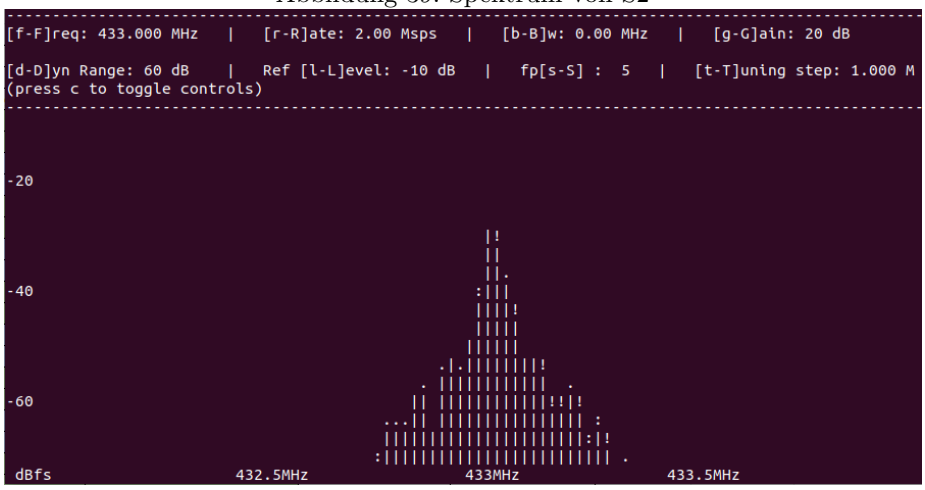


Abbildung 40: Spektrum von S3

Abbildung 41: Spektrum von S4

Abbildung 42: Spektrum von S5

Abbildung 43: Spektrum von S6

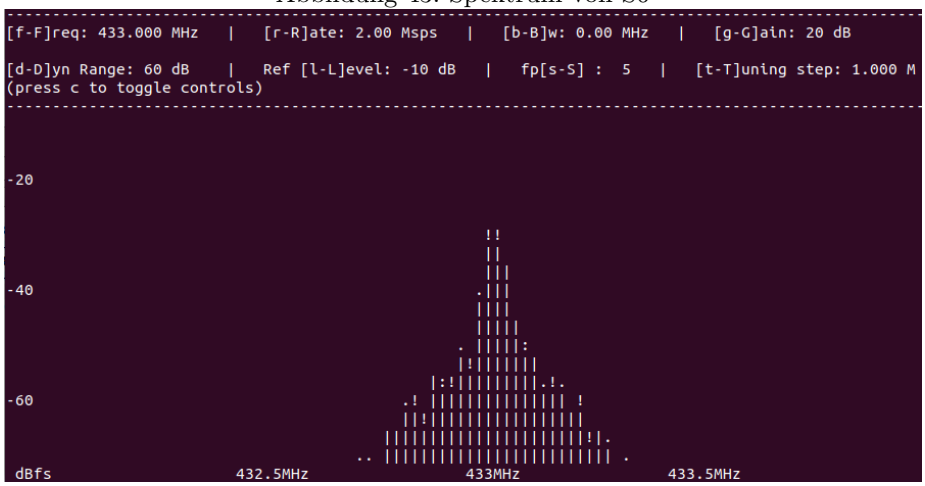


Abbildung 44: Spektrum von S7

Abbildung 45: Spektrum von S8

Abbildung 46: Spektrum von S9

Abbildung 47: Spektrum von S10

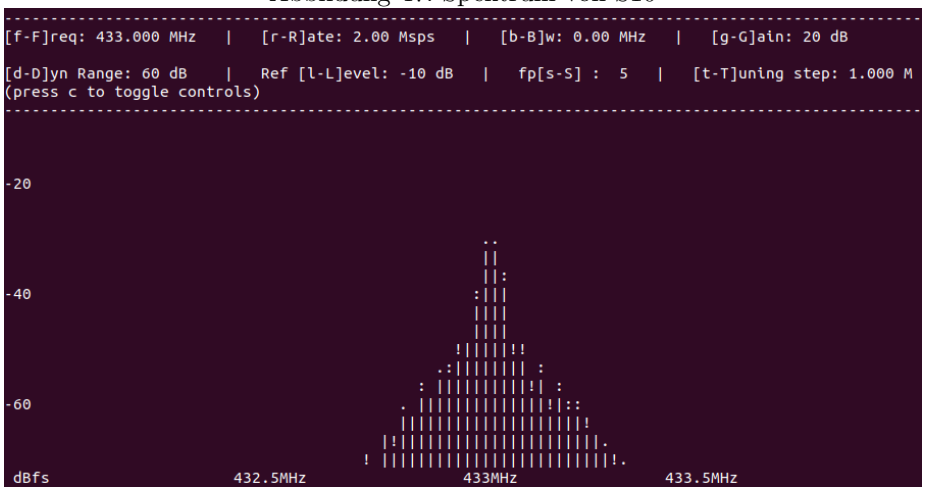
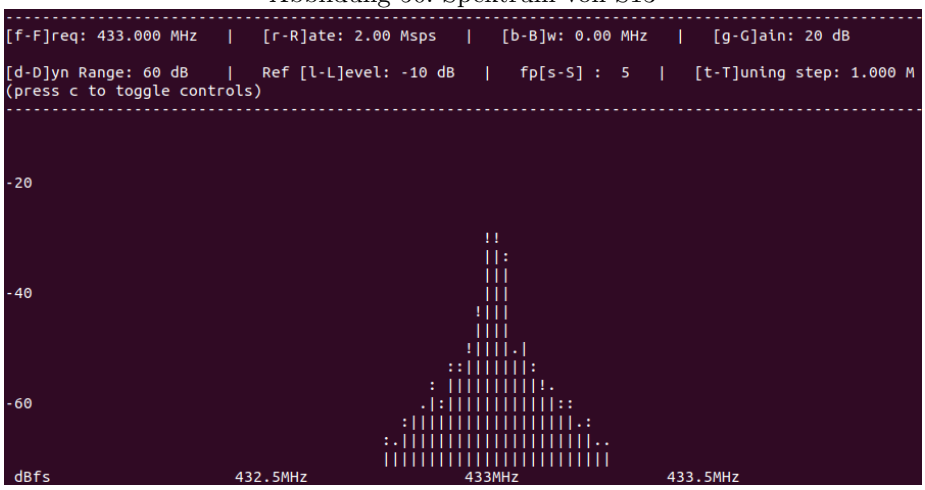


Abbildung 48: Spektrum von S11

Abbildung 49: Spektrum von S12

Abbildung 50: Spektrum von S13



Der Quellcode, der die PDM Impulse generiert, enthält eine einfache Substitution in dem Probenpuffer:

```
// // //
//

    size_t offset_left  = 12;   // value determined empirically
    size_t offset_right = 37;   // value determined empirically
    const float pdm_null = 0.0;

    size_t half_period = spb/2;
    bool encode_negative_halfperiod = false; // by default

    // pre-fill the buffer with the waveform (sine wave)
    for (size_t n = 0; n < buff.size(); n++) {
        buff[n] = wave_table(index += step);
    }

    // Zero the Q channel
    for (size_t n = 0; n < buff.size(); n++) {
        buff[n] = std::complex<float>(buff[n].real(), pdm_null);
    }

    // Really simple PDM pulse encoding

    // The positive half period of the sine
    for (size_t n = offset_left; n < offset_right; n++) {
        buff[n] = std::complex<float>(pdm_null, pdm_null);
    }
```

```
if (encode_negative_halfperiod) {
    // The negative half period of the sine
    for (size_t n = half_period + offset_left;
    n < half_period + offset_right;
    n++) {
            buff[n] = std::complex<float>(pdm_null, pdm_null);
    }
}
//
// // //
```

Alle diese Wellenformen enthalten das Symbol nur in der positiven Halbperiode codiert. Danach wurde es überprüft, dass die Symbolen können auch im negativen Teil der Impulsen codiert werden. Darunter stelle ich drei weitere PDM-Symbole (W1, W2, W3), die die folgende Konfiguration haben:

Abbildung 51: Symbole in der positiven und negativen Halbperiode

Symbol	Positive Halbperiode PDM-Länge	Negative Halbperiode PDM-Länge	Spektrogramm (Name des Bildes)
W1	L(S1)	L(S1)	Spektrum W1
W2	L(S13)	L(S13)	Spektrum W2
W3	L(S1)	L(S13)	Spektrum W3

Das interessanteste Symbol in dieser Menge sei W3 - es enthaltet die engste und breiteste Pulslänge (die Länge von Symbol S1 in der positiven Halbperiode und die Länge von S13 in der negativen), deshalb soll er das breiteste Spektrum haben.

Die Ergebnisse sind die folgenden:

Abbildung 52: Symbol W1

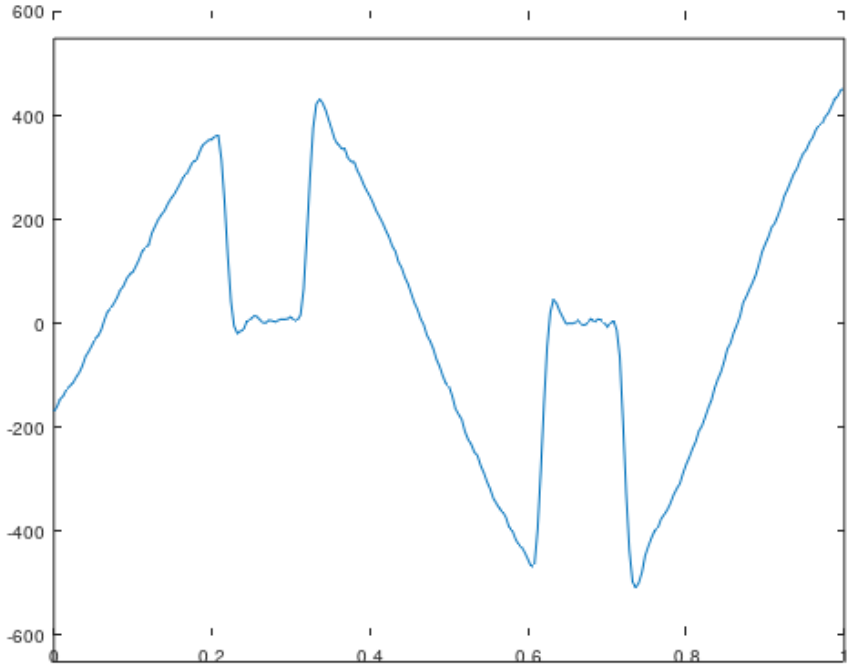


Abbildung 53: Spektrum von W1

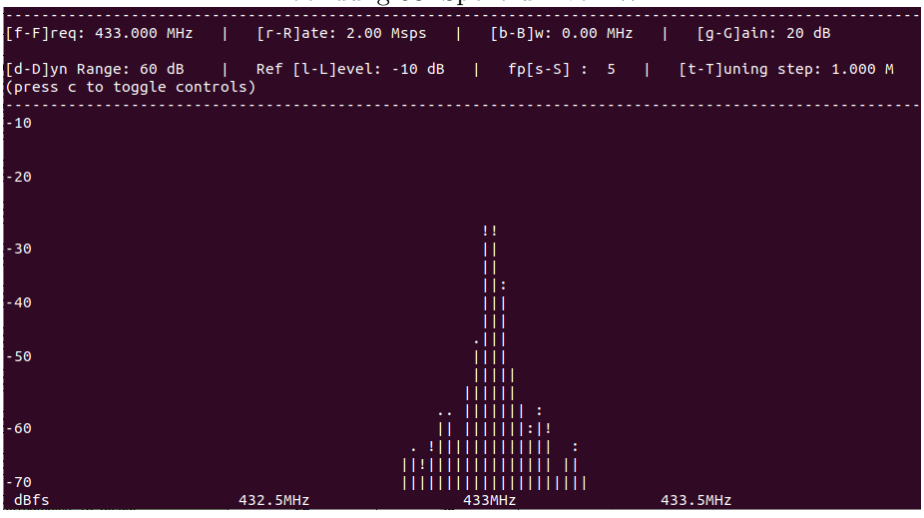


Abbildung 54: Symbol W2

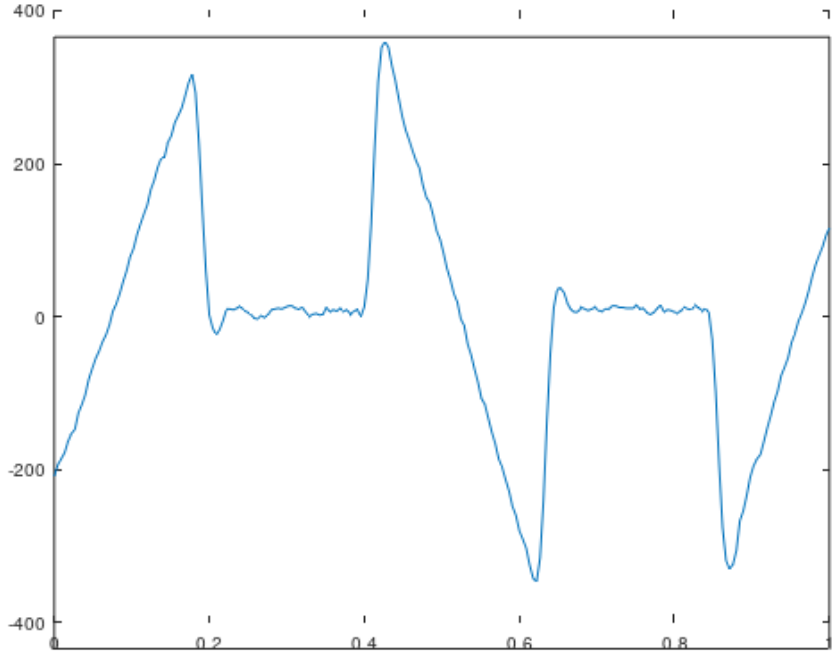


Abbildung 55: Spektrum von W2

Abbildung 56: Symbol W3

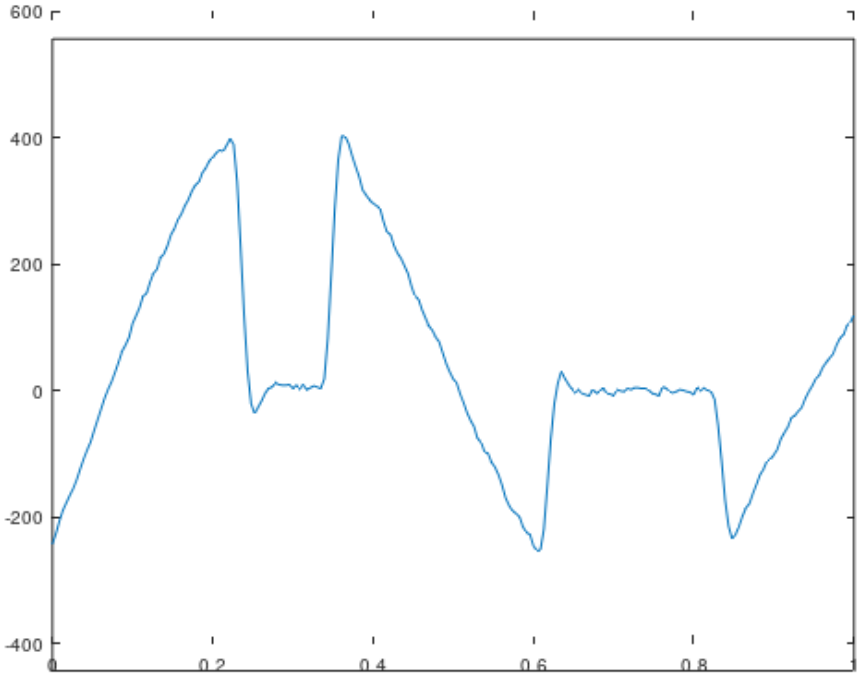
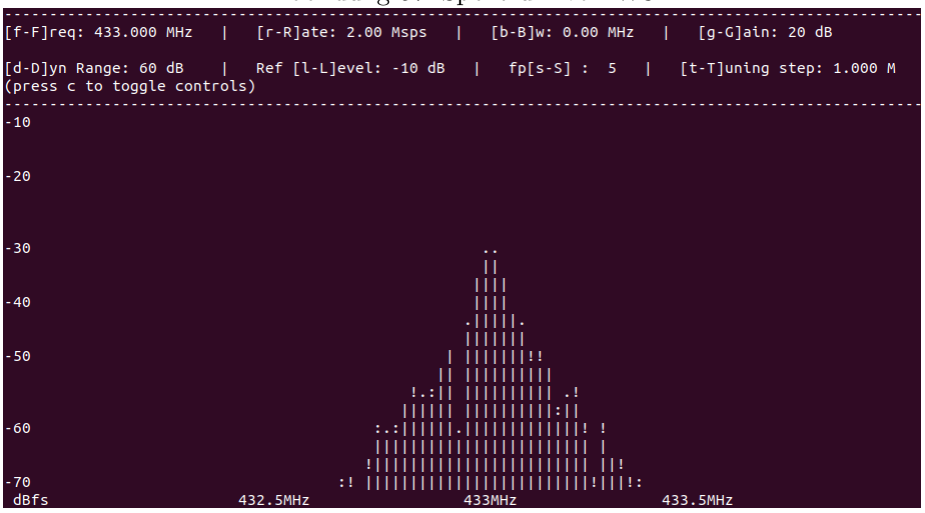


Abbildung 57: Spektrum von W3



Alle Spektraldiagramme sehen normal aus und entsprechen der eingestellten Bandbreite von 200kHz (aus dem *–bw 200e3* Argument in den Befehlen). Das ist die minimale einstellbare Bandbreite für das USRP B205-Mini Modul.

4.4.6 Der Q-Kanal

Es wurde schon beschrieben, dass für einen stabilen Empfang soll den Q-Kanal auf Null gesetzt werden. Dies wurde empirisch überprüft. Obwohl der Sender kodiert nur an dem I-Kanal, der nulierte Q-Kanal hat einen Einfluss auf die Bildung des komplexen Signals nach dem IQ-Modulator. Deshalb ist es interessant wie der Zustand dieser Q-Kanal an der Seite des Empfängers aussieht. Die folgenden Bilder stellen die Antwort vor:

Abbildung 58: I-Kanal

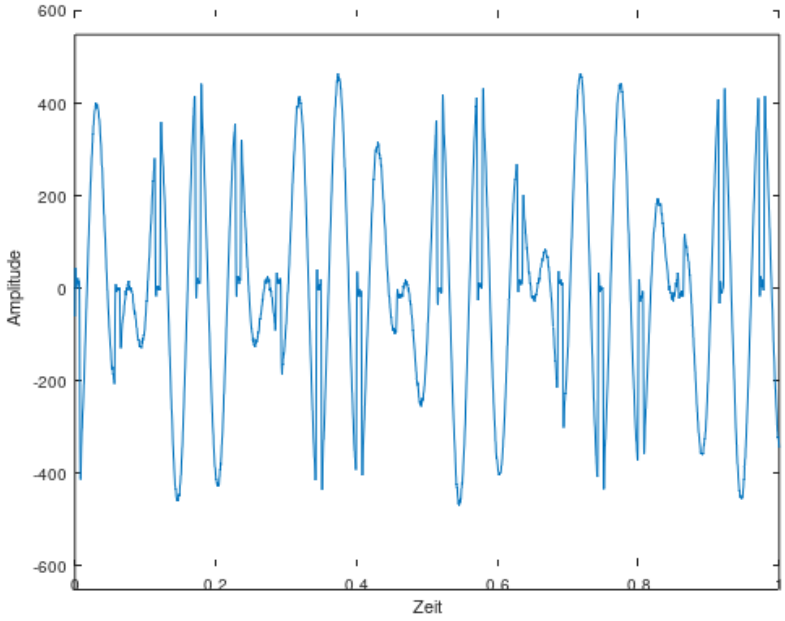


Abbildung 59: Q-Kanal

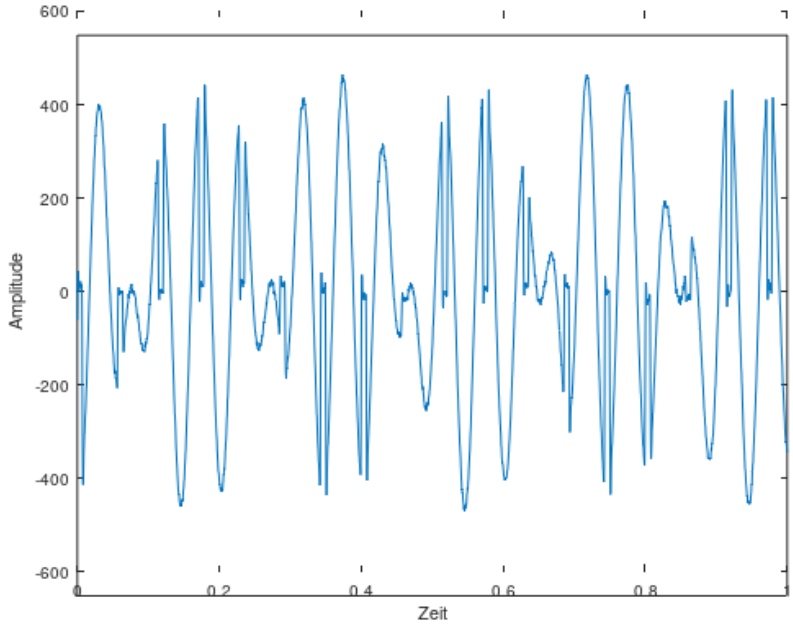
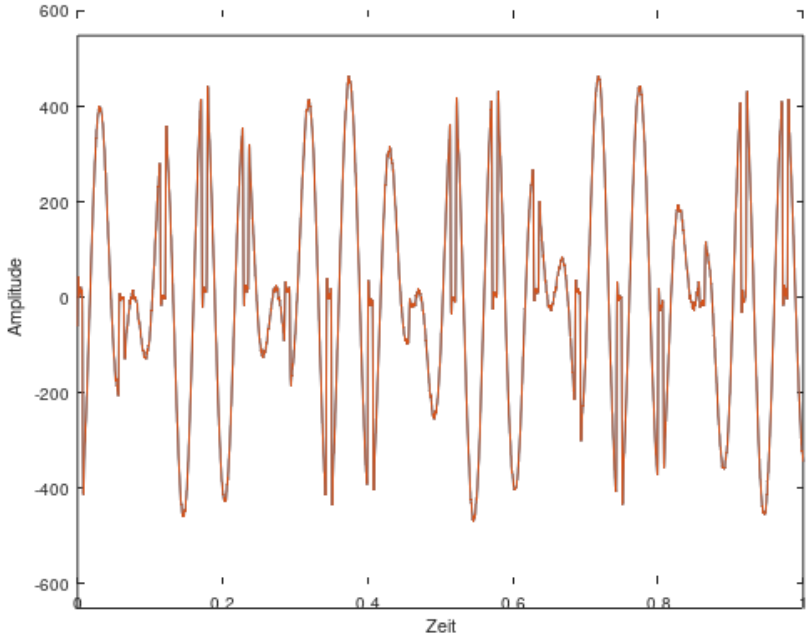


Abbildung 60: I und Q Kanäle - übereinander geplottet

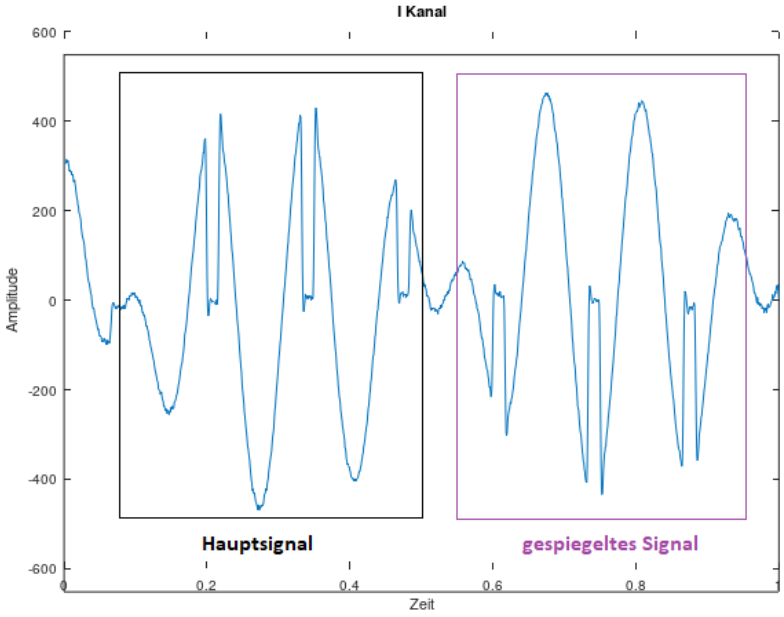


Man kann daraus schließen, dass die beide I und Q Kanäle gleich sind. Das beweist, dass sie abhängig voneinander bleiben und der IQ-Modulator kann nicht als Multiplexor für eine zusätzliche Übertragungsleitung funktionieren.

4.4.7 Das gespiegelte Signal

Es wurde mehrmals beobachtet, dass es in den Aufnahmen am Empfänger existiert. Das nächste Bild beweist dies:

Abbildung 61: gespiegeltes Signal



Normalerweise ist das gespiegelte Signal nur eine Verursache für Energieverlust. Für die Datenübertragung ist nur das Hauptsignal wichtig. Deswegen soll man einen Filter stellen, der die Komponenten des gespiegelten Signals entfernen.

Dies wurde jedoch nicht getan, weil einer Filter bedeutet komplexere Logik entweder im FPGA oder im Software. Und diese gespiegelte Proben kann auch Bestätigung des Hauptsignals dienen, wenn der Decodierer die PDM Pulsen in der negativen Halbperiode verarbeiten kann.

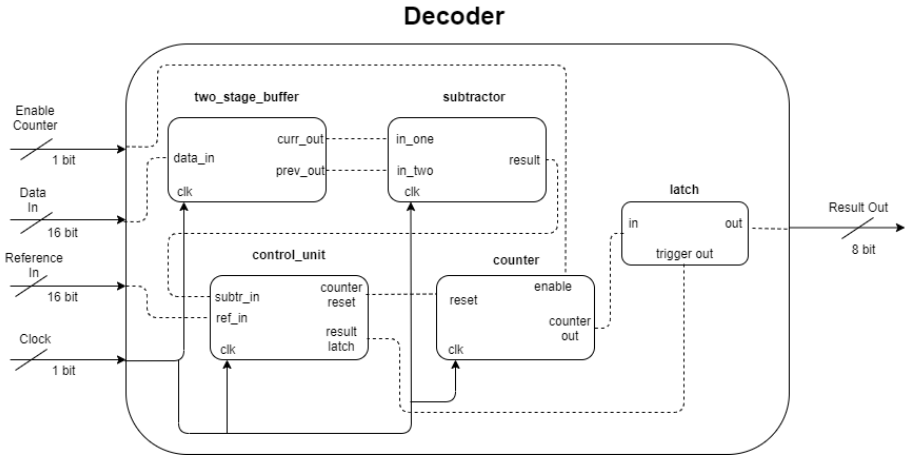
4.4.8 Realisierung des Decoders

Nach dem erfolgreichen Empfang der Symbolen, man soll ein Decodierer implementieren. Natürlich ist es möglich die Logik als C++ Code realisieren, aber aus dem Experiment mit GNURadio habe ich schon überprüft, dass das Betriebssystem kann keine Echtzeitausführung des Programms garantieren. Als die USRP B205-Mini Modul fasst ein *Spartan 6* FPGA um, ich werde meine Decodiererlogik im Verilog schreiben und in der Empfangskette integrieren.

Der Decodierer wurde aus den folgenden Komponenten aufgebaut:

- Two Stage Buffer: Es dient als ein Puffer, das den aktuellen und den vorherigen Wert der Eingangsdaten speichert.
- Subtractor: Führt eine Subtraktion von dem aktuellen und vorherigen Wert aus.
- Control Unit: Bestimmt die Flanken des Pulses - an fallender Flanke setzt den Zähler auf Null; an steigender Flanke verriegelt das Ergebnis.
- Counter: Trivial 8-bit Zähler.
- Latch: Ein 8-bit Flip-Flop von D-Typ.

Abbildung 62: Diagramm der Bauteilen des Decoders



Der Decoder hat die folgende Eingänge:

- 1-Bit Enable Counter : Signal für Aktivierung des Zählers
- 16-Bit Data In: Eingang für den Probenwert
- 16-Bit Reference In: Größe der Flanke bestimmen
- Clock

Darunter beschreibe ich die Implementierung im Detail.

Die digitalisierte Proben gehen in *Data In* ein. Diese Proben werden in dem *Two Stage Buffer* gespeichert. Dieses Puffer ist ein Schieberegister, der aus drei 16-Bit Elemente besteht. Seine Realisierung im Verilog ist wie folgt:

```
module two_stage_buffer(
                clock ,       // The clock
                data_in ,     // 16-bit data input
                current_out,
                prev_out
         );

        //-------- Registers --------
        reg signed [15:0] data_in_reg ;
        reg signed [15:0] one_step_delay ;

        //-------- Input ports --------
        input wire clock ;
        input wire signed [15:0] data_in ;

        //-------- Output ports --------
        output reg signed [15:0] current_out ;
        output reg signed [15:0] prev_out ;

        //-------- Behaviour --------
        always @ (posedge clock)
        begin : TWO_STAGE_BUFFER // module name
                current_out <= data_in ;
                one_step_delay <= current_out ;
                prev_out <= one_step_delay ;
        end

endmodule
```

Das Modul gibt den ersten und den dritten Abtastwert aus. Der zweite wird von dem internen Register *one_step_delay* gespeichert und auf dieser Weise verspätet. Die beide Werte werden an das nächste Modul (*Subtractor*) geliefert.

Das *Subtractor* macht nur eine Subtraktion zwischen zwei Variablen:

```
module subtractor(
                clock ,
                input_one ,
                input_two ,
                result
    );

    //-------- Registers --------

    // ...

    //-------- Input ports --------
    input wire clock ;
    input wire signed [15:0] input_one ;
    input wire signed [15:0] input_two ;

    //-------- Output ports --------
    output reg signed [15:0] result ;

    //-------- Behaviour --------
    always @ (posedge clock)
    begin : SUBTRACTOR
            result <= input_one - input_two ;
    end

endmodule
```

Das Ergebnis der Subtraktion wird an den Eingang des *Control Unit* Moduls geliefert. Das *Control Unit* vergleicht dies mit dem eingestellten *Ref In* Wert. Wenn der absolute Wert nach der Subtraktion größer als dem *Ref In* ist, das bedeutet, dass eine Flanke von PDM Impuls wurde entdeckt. Die Logik prüft ob die Flanke steigende oder fallende ist. An der fallenden Flanke wird den Zähler (*Counter*) nulliert. An der steigenden Flanke - das Ergebnis des *Counter* wird ausgegeben und der Zähler wird nulliert. Weil wir den PDM Impuls nur in der positiven Halbperiode der Sinusswingung kodieren, die fallende Flanke bezeichet den Anfang des PDM-Impulses und die steigende bezeichet den Ende.

Die Quellcode des *Control Unit* Moduls ist wie folgt:

```
module control_unit(
                clock ,
                subtracted_in ,
                ref_in ,
                counter_reset ,
                result_latch
    );

    //------- Input ports --------
    input wire clock ;
    input wire signed [15:0] subtracted_in ;
    input wire signed [15:0] ref_in ;

    //------- Output ports/regs --------
    output reg counter_reset ;
    output reg result_latch ;

    //------- Output initialization --------
    initial counter_reset = 1'b0 ;
    initial result_latch = 1'b0 ;

    //------- Registers --------
    reg signed [15:0] subtracted_in_reg ;
    reg signed [15:0] ref_in_reg ;

    //-------- Behaviour --------
    always @ (posedge clock)
    begin : CONTROL_UNIT
       if ( result_latch == 1'b1 ) // check if the latch up,
                                   // set it to down
           begin
               result_latch <= 1'b0 ;
            end

       if ( counter_reset == 1'b1 ) // check if the reset flag up,
                                    // set it to down
          begin
                  counter_reset <= 1'b0 ;
          end

       subtracted_in_reg <= subtracted_in ;
       ref_in_reg <= ref_in ;
```

```
        if ( subtracted_in_reg < 0 )
        begin
            if ( subtracted_in_reg + ref_in_reg < 0 )
            begin
                // A falling slope, start counting
                counter_reset <= 1'b1 ;
             end
         end

         else // if ( subtracted_in_reg > 0 )
           begin
             if ( subtracted_in_reg - ref_in_reg > 0 )
             begin
                 // A rising slope, trigger the latch to show the result
                 result_latch <= 1'b1 ;
             end
           end
         end
endmodule
```

Das *Counter* Modul ist ein einfacher Zähler. Die Implementierung im Verilog sieht wie so aus:

```
module simple_counter(
                clock , // Clock input of the design
                reset , // active high, synch. reset input
                enable , // active high, enable signal for counter
                counter_out // 7-bit vector output of the design
    );

        //-------- Input ports --------
        input clock ;
        input reset ;
        input enable ;

        //-------- Output ports -------
        output [7:0] counter_out ;

        //-------- Input ports Data Type --------
        // By rule all the input ports should be wires
        wire clock ;
        wire reset ;
        wire enable ;

        //-------- Output ports Data Type --------
        reg [7:0] counter_out ;

        // Code Starts here
        always @ (posedge clock)
        begin : COUNTER // block name is 'COUNTER'
                // At every rising edge we check if reset is active
                // If active, we load the counter output with 4'b0000
                if ( reset == 1'b1 ) begin
                        counter_out <= 8'b00000000;
                end
                else if ( enable == 1'b1 ) begin
                        counter_out <= counter_out + 1;
                end
        end
endmodule
```

Nach der Ermittlung einer steigengen Flanke, das Ergebnis aus dem Zähler beschreibt die Länge des PDM-Impulses. Dieses Ergebnis soll an den Ausgang des Decodierers geliefert werden. Das *Latch* Modul führt diese einfache Operation durch:

```
module latch(
                trigger_out ,
                data_in ,
                data_out
    );

        //-------- Input ports --------
        input wire trigger_out ;
        input wire signed [7:0] data_in ;

        //-------- Output ports/regs --------
        output reg signed [7:0] data_out ;

        //-------- Behaviour --------
        always @ (posedge trigger_out)
        begin : LATCH
          data_out <= data_in ;
        end

endmodule
```

4.4.9 Testbenches

Jedes Schaltung soll getestet werden. In der FPGA Welt, die Testbenches sind analog zu den Unit und Integrationstests in der Softwareentwicklung. Das Verhältnis des Decoders soll mit allen Wellenformen in der Menge von Symbolen überprüft werden. Nur für den Fall, diese Wellenformen können durch Hinzufügen einer zusätzlichen Sinuswellenperiode kompliziert werden. Auf diese Weise werde ich realistische Umwelt für die Simulation erzeugen, mit normalen Sinuskurven inzwischen den PDM-modulierten Impulsen.

Am Sender führe ich die *tx_waveforms* Anwendung mit 200 Proben - die erste 100 sind noch das PDM Symbole von den Beispielen und die nächste 100 Proben sind die nicht modulierte zusätzliche sinusoidale Perioden. Der Befehl in diesem Fall ist:

```
sudo ./tx_waveforms --rate 1e6 --freq 433e6 --ampl 0.6
--gain 70 --bw 200e3 --wave-type SINE
--wave-freq 10e3
--spb 200 --otw sc16
```

Für jedes Symbol wurden zwei Aufnahmen der Übertragungsproben ausgewählt. Die erste überprüft ob der Decoder von eine undefinierte Zustand stabil bleiben kann und aus der zweiten soll der Decoder das Symbol erkennen. Ein Beispiel mit Symbol S4 sieht wie so aus:

Abbildung 63: Beispiel für undefinierten Zustand im S4-Symbol

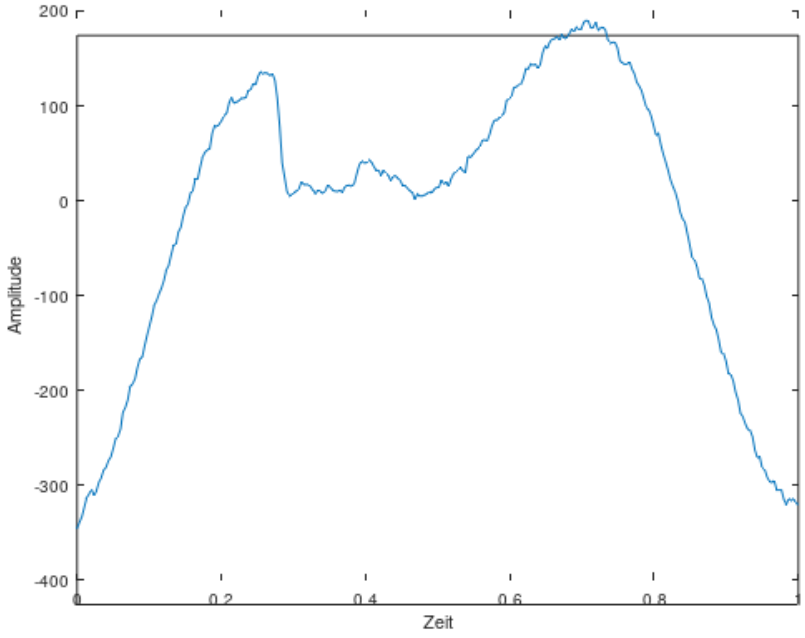
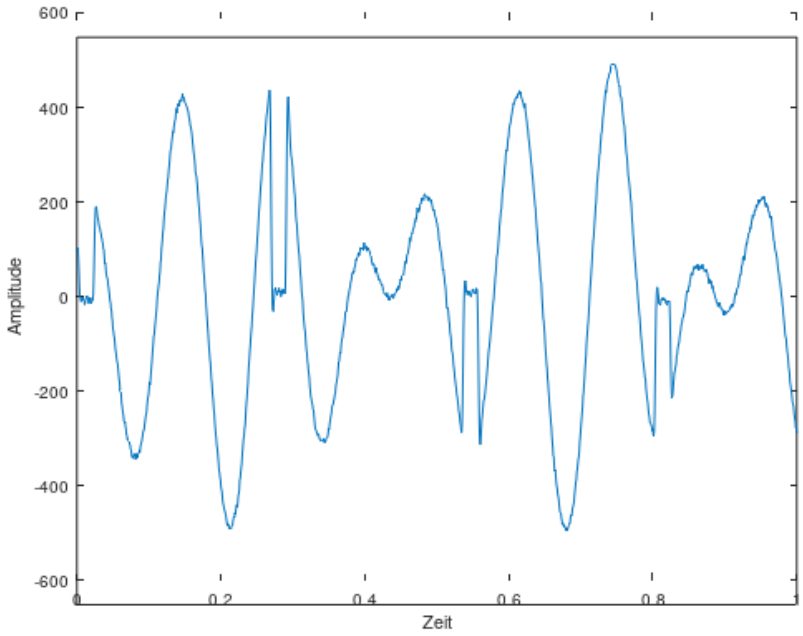
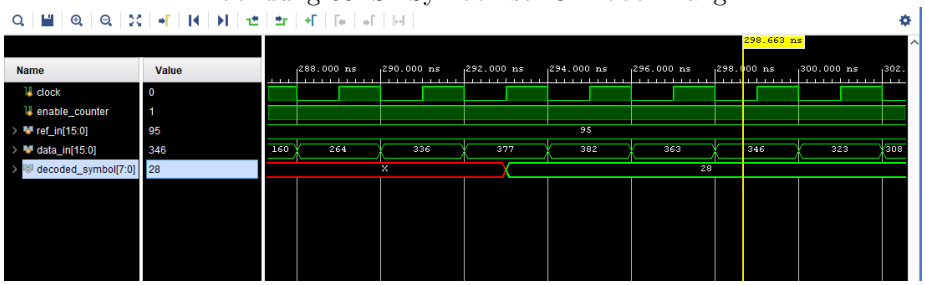


Abbildung 64: Übertragung von S4-Symbol gefolgt von einer Sinusswingung



Für die Simulation der Verilog-Testbenches wurde die *Vivado* Anwendung von *Xilinx* verwendet. Alle Eingangswerte waren die realistische Daten, die von den UHD-Anwendung *rx_samples_to_file* versammelt wurden. Ein Beispiel von Vivado für Symbol S4 sieht wie so aus:

Abbildung 65: S4-Symbol ist 28 Proben Lang



Ein Python-Skript wurde verwendet, um die gespeicherte Daten von GNUOctave (beziehungsweise UHD) in Eingangswerte für eine Verilog Simulation umzuwandeln. Die folgende Tabelle zeigt die Erfolgsrate des Decoders für jedes Symbol:

Abbildung 66: Testbench Ergebnisse

Symbol	Ausgangswert	Richtig decodiert	Ref_in Wert
1	22	Ja	95
2	24	Ja	95
3	27	Ja	95
4	28	Ja	95
5	31	Ja	95
6	33	Ja	95
7	36	Nein	95
8	36	Nein	95
9	38	Ja	95
10	40	Ja	95
11	45	Nein	95
12	45	Nein	95
13	48	Ja	95

Aus der Tabelle kann man zusammenfassen, dass es eine Kollision zwischen den S7-S8 und S11-S12 Symbolen gibt. Als eine Auflösung soll die Abtastrate am Empfänger erhöht werden. Für meine Hardware-Implementierung werde ich mit dieser Konfiguration arbeiten und die S7, S8, S11 und S12 Kollisionen ignorieren. Ich folge das einfachste Realisierung. Alle andere Einstellungen und Eichwesen sind eine Aufgabe für zukünftige Optimierung.

4.4.10 Decoder als Coprozessor im USRP B205-Mini

Da die Decoder-Implementierung fertig und getestet ist, muss die PDM Logik jetzt in der FPGA Image des USRPs integriert werden. Die Register müssen über die USB-Schnittstelle freigelegt werden. Es ist auch notwendig an den richtigen Leitungen im existierenden Quellcode des Images zu finden und den

Decoder mit ihnen zu verbinden.

Zuerst folge ich die offizielle Dokumentation des USRPs für die Arbeit mit den benutzerdefinierten Registern (user registers). Man soll überprüfen, ob eine Loopback-Test verwendet werden kann - Werte schreiben und dann diese Werte zurück korrekt lesen. Um diese Register verfügbar für das UHD-Driver zu stellen, die Variable *USER_SETTINGS* in der *radio_legacy* Datei soll auf *1* gesetzt werden.

```
module radio_legacy
  #(
        parameter RADIO_FIFO_SIZE = 13,
        parameter SAMPLE_FIFO_SIZE = 11,
        parameter FP_GPIO = 0,
        parameter NEW_HB_INTERP = 0,
        parameter NEW_HB_DECIM = 0,
        parameter SOURCE_FLOW_CONTROL = 0,
        parameter USER_SETTINGS = 1,
        parameter DEVICE = "SPARTAN6"
  )
```

Eine Änderung des Parameters *USER_SETTINGS* von *0* nach *1* in der *b205_core* Datei auch notwendig ist:

```
        wire [31:0] fe_gpio_out32;
        wire [9:0] fp_gpio_out10, fp_gpio_ddr10;
        assign fe_gpio_out = fe_gpio_out32[7:0];
        assign fp_gpio_out = fp_gpio_out10[7:0];
        assign fp_gpio_ddr = fp_gpio_ddr10[7:0];

        radio_legacy #(
        .RADIO_FIFO_SIZE(RADIO_FIFO_SIZE),
        .SAMPLE_FIFO_SIZE(SAMPLE_FIFO_SIZE),
        .FP_GPIO(1),
        .NEW_HB_INTERP(1),
        .NEW_HB_DECIM(1),
        .SOURCE_FLOW_CONTROL(0),
        .USER_SETTINGS(1),
        .DEVICE("SPARTAN6")
        )
```

Das Image für die FPGA wurde mit den folgenden Befehlen erzeugt:

```
[ise@localhost b2xxmini]$ pwd
/home/ise/USRP/fpga/usrp3/top/b2xxmini
[ise@localhost b2xxmini]$ make B205mini | tee /tmp/build.log
```

Am Ende der Kompilierung wurde es auch im Log überprüft ob es irgendwelche Fehler aufgetreten sind. Ein “grep ERROR” ist für diese Aufgabe genug.

```
[ise@localhost b2xxmini]$ cat /tmp/build.log | grep ERROR
[ise@localhost b2xxmini]$
```

Das Ergebnis wurde mit der folgenden UHD C++ Zeilen überprüft:

```
auto usrp = uhd::usrp::multi_usrp::make(custom_init_args);
auto regs = usrp->get_user_settings_iface();
regs->poke32(0, 0xCAFE);
regs->poke32(4, 0xBEEF);

std::cout << boost::format("Read value: 0x%016X") % regs->peek64(0)
<< std::endl;
```

Man kann die REAME.txt Instruktionen für die Kompilierung folgen:

```
$ mkdir build/ # Creates a new build directory
$ cd build/
$ cmake ..
$ make
```

Aus der Ausgabe des Programms kann man bestätigen, dass die Werte (0xCAFE und 0xBEEF) aus der Code korrekt geschrieben und gelesen geworden sind:

```
$ sudo ./init_usrp
[INFO] [B200] Detected Device: B205mini
[INFO] [B200] Operating over USB 3.
[INFO] [B200] Initialize CODEC control...
[INFO] [B200] Initialize Radio control...
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Setting master clock rate selection to 'automatic'.
[INFO] [B200] Asking for clock rate 16.000000 MHz...
[INFO] [B200] Actually got clock rate 16.000000 MHz.
Read value: 0x0000BEEF0000CAFE
```

Nächster Schritt ist den bestimmten Leitungen für die Verbindung des Decoders zu finden. Eine Analyse der Quellcode von *radio_legacy.v* führt uns zu den folgenden Zeilen:

```
ddc_chain #(.BASE(SR_RX_DSP), .DSPNO(0), .WIDTH(24),
.NEW_HB_DECIM(NEW_HB_DECIM), .DEVICE(DEVICE)) ddc_chain
        (.clk(radio_clk), .rst(radio_rst), .clr(1'b0),
        .set_stb(set_stb),.set_addr(set_addr),.set_data(set_data),
        .rx_fe_i({rx_fe[31:16],8'd0}),.rx_fe_q({rx_fe[15:0],8'd0}),
        .sample(sample_rx), .run(run_rx), .strobe(strobe_rx),
        .debug(debug_ddc_chain) );
```

Die Ausgang der *DDC (Digital Down Convertor)* ist die *sample_rx*, deren Erneuerung findet an der Flanke der Leitung *strobe_rx* statt. Die UHD-Schnittstelle serialisiert die Ausgang von *sample_rx* mit einer Streamer und liefert es über den USB. Die *sample_rx* Variable is eine 32-bit Leitung (im Zusammenhang mit Verilog).

```
wire [31:0]     sample_rx;
```

Für die C++ Schnittstelle ist diese Variable eine Zahl von Typ *short complex*. Die oberen 16 bits erhalten den I-Wert und die unteren 16 bit erhalten den Q-Wert. Um sicher zu sein, ich werde diesen Wert an den USB darlegen, aus dem UHD lesen und den Amplituden mit den gespeicherten Proben der Symbolen vergleichen.

Die folgende Code realisiert die Idee (komplettes Beispiel im Anhang *radio_legacy_read_samplerx.v*):

```
// // //
// Interconnects for the user defined registers
// // //
wire [31:0] user_reg_0_value, user_reg_1_value, curr_sample_reg_value;
wire [15:0] decoded_symbol_value;
reg [31:0] curr_sample_reg;
// // //
 ddc_chain #(.BASE(SR_RX_DSP), .DSPNO(0), .WIDTH(24),
 .NEW_HB_DECIM(NEW_HB_DECIM), .DEVICE(DEVICE)) ddc_chain
        (.clk(radio_clk), .rst(radio_rst), .clr(1'b0),
        .set_stb(set_stb),.set_addr(set_addr),.set_data(set_data),
        .rx_fe_i({rx_fe[31:16],8'd0}),.rx_fe_q({rx_fe[15:0],8'd0}),
        .sample(sample_rx), .run(run_rx), .strobe(strobe_rx),
        .debug(debug_ddc_chain) );

// PWM/PDM Decoder Instance
   decoder_top decoder_top
        (.clock(strobe_rx),
        .enable_counter(1'b1),
        .ref_in(32'd30),  // Hard-coded now, we will be using the
                          // user_reg_0_value to read the
                          //'sample_rx' via the user registers
        .data_in(sample_rx),
        .decoded_symbol(decoded_symbol_value) );

   always @(posedge strobe_rx) begin
        curr_sample_reg <= sample_rx; // Latch the sample
                                      // at strobe_rx signal, we will
                                      // then read the user register
                                      // via the UHD
   end
// // //
assign curr_sample_reg_value = curr_sample_reg;
// // //
```

Die *rx_samples_to_file* Beispielanwendung wurde mit den folgenden Zeilen modifiziert:

```
std::string userreg_args = "type=b200,enable_user_regs";
uhd::usrp::multi_usrp::sptr usrp = uhd::usrp::multi_usrp::make(userreg_args);

// Read USER REGISTERS here
auto regs = usrp->get_user_settings_iface();
short i_chan = 0;
short q_chan = 0;
auto iq_chans = (regs->peek64(0))
    & 0x00000000FFFFFFFF; // Because we use the lower 32 bits
                          // in the custom FPGA build
                          // for storing the
                          // current sample_rx
i_chan = (iq_chans >> 16) & 0x0000FFFF;
q_chan =  iq_chans       & 0x0000FFFF;

std::cout << boost::format("I: %hi\n") % i_chan;
std::cout << boost::format("Q: %hi\n") % q_chan;
std::cout << boost::format("-------------\n");
```

Ohne Signal aus dem Sender, die I und Q Werte fast auf Null gleich sind (natürlich mit irgendwelchem Geräusch):

```
-------------
I: -1
Q: 0
-------------
I: -3
Q: -2
-------------
I: -10
Q: -2
-------------
I: -3
Q: 1
-------------
I: -5
Q: 4
-------------
I: -4
Q: 2
-------------
```

Bei tatsächlicher Übertragung, die Amplituden von I und Q mit den Daten aus den bekannten Symbolen übereinstimmen. Manche Proben werden übersprungen, weil der FPGA schneller als die C++ Programm ist.

```
----------------
I: 143
Q: 231
----------------
I: -14
Q: 18
----------------
I: 34
Q: -299
----------------
I: -47
Q: -43
----------------
I: -218
Q: -257
----------------
I: -238
Q: -303
----------------
I: -184
Q: -195
----------------
```

Diese Werte sind gut genug, um die Eingliederung des Decoders im FPGA fortzufahren. Als ertens, man soll alle Verilog-Dateien sammeln. Im Pfad *fpga/usrp3/top/b2xxmini* erstelle ich einen Order mit dem Namen "pwm" und die Quellcode des Decoders da kopieren:

```
$ find pwm/
pwm/
pwm/latch.v
pwm/pwm_decoder.v
pwm/two_stage_buffer.v
pwm/simple_counter.v
pwm/subtractor.v
pwm/control_unit.v
```

Die Dateien sollen in dem *Makefile.b205.inc* mit der folgenden Variable addiert werden :

```
CUSTOM_SRCS = \
/home/ise/USRP/fpga/usrp3/top/b2xxmini/pwm/control_unit.v \
/home/ise/USRP/fpga/usrp3/top/b2xxmini/pwm/decoder_top.v \
/home/ise/USRP/fpga/usrp3/top/b2xxmini/pwm/latch.v \
/home/ise/USRP/fpga/usrp3/top/b2xxmini/pwm/simple_counter.v \
/home/ise/USRP/fpga/usrp3/top/b2xxmini/pwm/subtractor.v \
/home/ise/USRP/fpga/usrp3/top/b2xxmini/pwm/two_stage_buffer.v
```

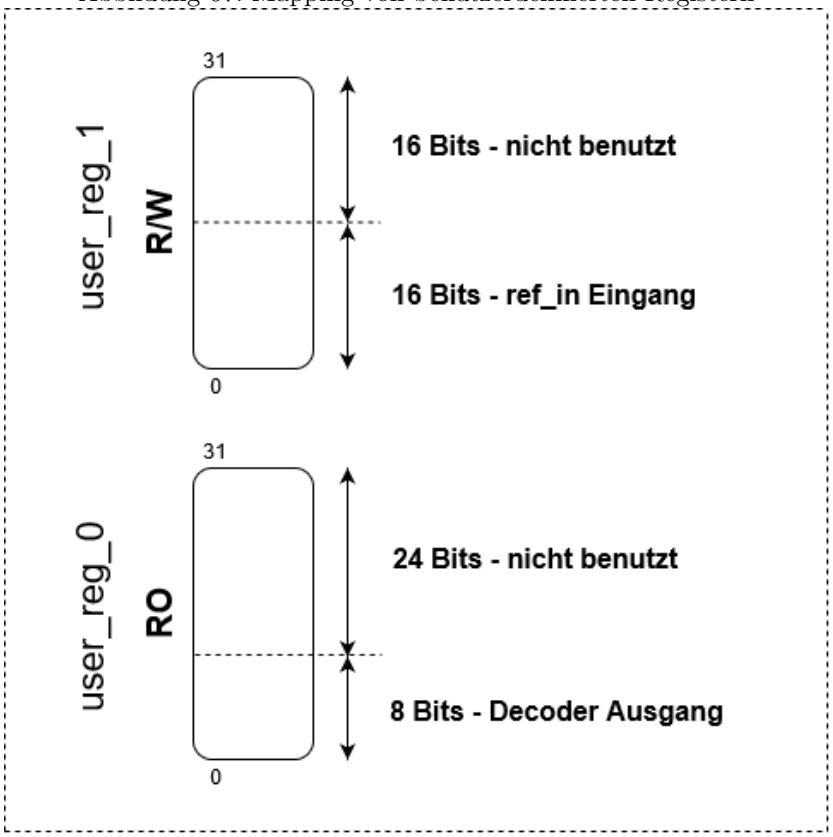
Der PDM-Decoder soll am Ende der RX-Kette im *radio_legacy.v* eingefüllt werden:

```
pwm_decoder pwm_decoder
      (.clock(strobe_rx),
      .enable_counter(1'b1),
      .ref_in(user_reg_1_value[15:0]),
      .data_in(sample_rx[31:16]),
      .decoded_symbol(decoded_symbol_value) );

      always @* begin
      case(rb_addr_user)
              8'd0 : rb_data_user <= {user_reg_1_value,
              24'd0,
              decoded_symbol_value};
              default : rb_data_user <= 64'd0;
      endcase
      end
```

Der Auszug der Quellcode zeigt wie das Modul instanziert und mit dem benutzerdefinierten Registern verbundet wird. Der Mapping der Bits sieht wie so aus:

Abbildung 67: Mapping von benutzerdefinierten Registern



Jetzt soll die *rx_samples_to_file* Anwendung noch einmal modifiziert werden. Ich addiere die folgenden Zeilen:

```
#define CONFIG_REG ( 4 )
#define PWM_EXPECTED_MIN ( 20 )
#define PWM_EXPECTED_MAX ( 50 )

//
// Work with the user defined registers here
//
auto regs = usrp->get_user_settings_iface();
uint32_t ref_in_val = 95 & 0x000FFFF;  // Ensure it is 16-bit long

regs->poke32(CONFIG_REG, ref_in_val);
uint64_t read_reg_val =  (regs->peek64(0));

uint32_t config_reg_value = read_reg_val >> 32;  // Write reg value
                                                 // ref_in value
                                                 // is here

uint16_t pwm_decoder_out = read_reg_val & 0x00FF;// Read reg value
                                                 // decoder value
                                                 // is here

if ( (pwm_decoder_out > PWM_EXPECTED_MIN)
  and (pwm_decoder_out < PWM_EXPECTED_MAX) )     // SW Limit

        std::cout << boost::format("config: 0x%08X, decoded: %hu\n")
        %config_reg_value % pwm_decoder_out;
```

4.4.11 Endergebnisse

Die realistische Erwartungen sind, dass der Empfang nicht so stabil wie in den Testbenches bleiben kann. Um dies zu beweisen, für jedes Symbol sammle ich die Werte, die mit der C++ Anwendung empfängt geworden sind. Für jedes Symbol sind diese keine Konstante, sonder eine Menge:

Abbildung 68: I-Kanal

Symbol	Menge von Werten am Ausgang des Decodiers	
S1	21, 22, 23, 24	[21 - 24]
S2	24, 25, 26	[24 - 26]
S3	25, 26, 27, 28	[25 - 28]
S4	28, 29, 30	[28 - 30]
S5	29, 30, 31	[29 - 31]
S6	32, 33, 34, 35	[32 - 35]
S7	34, 35, 36, 37	[34 - 37]
S8	36, 37, 38, 39	[36 - 39]
S9	38, 39, 40, 41	[38 - 41]
S10	40, 41, 42, 43	[40 - 43]
S11	42, 43, 44, 45	[42 - 45]
S12	44, 45, 46	[44 - 46]
S13	46, 47, 48, 49	[46 - 49]

Für einen 100% stabilen Empfang muss man diese Symbole wählen, deren Wertemengen keine Durchschnitt haben. Solche Symbolen kann S1, S3, S5, S7, S9, S11 und S13 sein. Aber sieben Symbole können jedoch keine Ganzzahl von Bits kodieren. Deswegen reduziere ich die Symbole zu vier. Ich benenne diese Symbole um und versuche noch einmal einen vollständigen Erfolg zu erreichen.

Die ausgewählten vier Symbole sind die folgenden:

Abbildung 69: Die neue Symbole

Symbol	Neuer Name	Menge von Werten
S1	S1	[21 - 24]
S5	S2	[29 - 31]
S9	S3	[38 - 41]
S13	S4	[46 - 49]

Und die folgende Funktion wird in der C++ Code addiert, um die Ergebnisse besser lesbar zu machen. In der Zukunft kann man die einfache Logik der Funktion auch im FPGA realisieren.

```
#define S1_MIN ( 21 )
#define S1_MAX ( 24 )
#define S2_MIN ( 29 )
#define S2_MAX ( 31 )
#define S3_MIN ( 38 )
#define S3_MAX ( 41 )
#define S4_MIN ( 46 )
#define S4_MAX ( 49 )

static uint16_t decode_symbol(unsigned short pwm_value)
{
        if      ( (pwm_value >= S1_MIN) and (pwm_value <= S1_MAX) )
        return 1; // Symbol 1

        else if ( (pwm_value >= S2_MIN) and (pwm_value <= S2_MAX) )
        return 2; // Symbol 2

        else if ( (pwm_value >= S3_MIN) and (pwm_value <= S3_MAX) )
        return 3; // Symbol 3

        else if ( (pwm_value >= S4_MIN) and (pwm_value <= S4_MAX) )
        return 4; // Symbol 4

        // Default
        return 0;
}
```

Die Ausgabe der C++ Anwendung zeigt die folgenden Zeilen an:

Für S1:

```
config: 0x0000005F, Decoded Symbol: S1
config: 0x0000005F, Decoded Symbol: S1
config: 0x0000005F, Decoded Symbol: S1
...
```

Für S2:

```
config: 0x0000005F, Decoded Symbol: S2
config: 0x0000005F, Decoded Symbol: S2
config: 0x0000005F, Decoded Symbol: S2
...
```

Für S3:

```
config: 0x0000005F, Decoded Symbol: S3
config: 0x0000005F, Decoded Symbol: S3
config: 0x0000005F, Decoded Symbol: S3
...
```

Für S4:

```
config: 0x0000005F, Decoded Symbol: S4
config: 0x0000005F, Decoded Symbol: S4
config: 0x0000005F, Decoded Symbol: S4
...
```

Alle Symbole von S1 bis S4 wurden stabil übertragt durch die Pulsedauermodulation.

4.5 Zusammenfassung und zukünftige Forschung

Das Ziel ist erreicht. Ich habe überprüft, dass die Pulsdauermodulation kann für Radiokommunikation ganz realistisch verwendet werden. Mit einer einfachen Decodiererlogik im FPGA, die Effizienz der Übertragung kann gleich der 4-QAM/QPSK Modulation, und immer mehr sein, mit keinen Filtern für das Basisbandsignal.

Das maximale Datenraten kann mit der folgengen Formel beschrieben werden:

$$R = Fb * (N + 1)$$

Mit R bezeiche ich den Datenrate im Bits pro Sekunde, Fb ist die Frequenz der Schwingung im Basisband und N ist der Anzahl der Symbolen. Die *1* im *(N+1)* Ausdruck ist die leere, nicht modulierte Schwingung, die auch als ein zusätzliches Symbol hinzuzählt werden kann.

Die Pulsdauermodulation ergibt eine digitale Aussicht zu den bekannten Verfahren für Radiokommunikation, die mit analogen Methoden für Signalverarbeitung arbeiten. In allen Fällen habe ich beobachtet, dass die Ergebnisse der PDM-Übertragung folgen die Nyquist Formel für maximale Datenrate:

$$Rmax = 2B * \log_2 M$$

Es wäre interessant, ob dieses Theorem immer gültig für die PDM ist, wenn die Abtatsrate am Empfänger viel mehr als zweimal ist. Es gibt zwei Richtungen um dies praktisch zu überprüfen - mit niedriger Bandbreite und hoher Basisbandfrequenz. In meinem SDR-Umwelt konnte ich nicht die Bandbreite weniger als 200kHz einstellen, weil dies der minamale Wert für das *B205* Modul ist. Außerdem waren die Datenübertragung mit höhere als 10kHz Basisbandfrequnzen nicht immer zum Empfang stabil - deswegen war es unmöglich ein solchen Versuch durchzuführen.

Wenn es um den IQ-Modulator geht, wird er überhaupt nicht benönigt. Aber man kann ihn nicht vermeiden, weil er ein Hardware-Bauteil der RF-Frontend von *USRP B205* ist.

Und schließlich kann die Pulsdauermodulation nicht nur für Radioübertragung benutzt werden. Drahtgebundene Übertragung auf Leiterplatten mit hochauflösenden Analog-Digital und Digital-Analog Wandlern, in Kombination mit Logik am Hardware-Ebene (wie FPGA), ist ein weiteres Thema für zukünftige Forschung.

Literatur

[1] G. S. Walia, H. P. Singh, Padma D. *Investigation of Roll off Factor in Pulse Shaping Filter on Maximal Ratio Combining for CDMA 2000 System*, World Academy of Science, Engineering and Technology International Journal of Electronics and Communication Engineering Vol:8, No:6, 2014, https://waset.org/publications/9998394/investigation-of-roll-off-factor-in-pulse-shaping-filter-on-maximal-ratio-combining-for-cdma-2000-system

[2] GNU Radio *Guided Tutorial GNU Radio in C++*, https://wiki.gnuradio.org/index.php/Guided_Tutorial_GNU_Radio_in_C

[3] Chetna Devi Kh1, Dr. Thangadurai N2 *A Review on Recent Trends in Software Defined Radio Design and Applications*, https://www.researchgate.net/publication/320827153

[4] Ettus Research *USRP B205-mini*, https://www.ettus.com/all-products/usrp-b205mini-i-board/

[5] Wilfried Plaßmann, Detlef Schulz *Handbuch Elektrotechnik: Grundlagen und Anwendungen für Elektrotechniker*, SpringerVieweg

[6] Dr.-Ing. Hristomir Yordanov *Vorlesung 8, Basisband Signalübertragung, Grundlagen der Kommunikationstechnik*, TU-Sofia

[7] Richard Lyons *Quarature Signals: Complex, But Not Complicated*, IEEE