HOCHSCHULE FÜR
TECHNIK UND WIRTSCHAFT
DRESDEN
UNIVERSITY OF APPLIED SCIENCES

Fakultät
Informatik/Mathematik

Bachelorarbeit

über das Thema

Entwicklung eines DECT-Monitors mittels Software Defined Radio

Autor: Felix Werner
s72924@htw-dresden.de
Matrikelnummer: 37540

Betreuer: Prof. Dr. Jörg Vogt
Zweitgutachter: Prof. Dr.-Ing. Robert Baumgartl

Abgabedatum: 24.07.2017

Danksagung

Besonderer Dank gebührt Prof. Dr.-Ing. Jörg Vogt, der diese Arbeit betreut und durch Ratschläge und Hardware unterstützt hat.

Weiterhin bin ich den Korrekturlesern Ron Luft, Evelyn Werner und Philipp Wenskus zu Dank verpflichtet, durch die diese Arbeit orthografisch und grammatikalisch wesentlich verbessert wurde.

Inhaltsverzeichnis

I Abbildungsverzeichnis

II Tabellenverzeichnis

III Listingverzeichnis

IV Abkürzungsverzeichnis

ADU	Analog-Digital-Umsetzer
CGRAN	Comprehensive GNU Radio Archive Network
DAU	Digital-Analog-Umsetzer
DECT	Digital Enhanced Cordless Telecommunications
DVB-T	Digital Video Broadcasting – Terrestrial
ETSI	European Telecommunications Standards Institute
FP	(Radio) Fixed Part
GRC	GNU Radio Companion
GUI	Graphical User Interface
NG-DECT	New Generation DECT
OSI	Open Systems Interconnection
Osmocom	Open Source Mobile Communications
PCIe	Peripheral Component Interconnect Express
PP	(Radio) Portable Part
RFFE	Radio Frequency Front-End
SDR	Software Defined Radio
SR	Software Radio
TDMA	Time Division Multiple Access
ULE	DECT Ultra Low Energy
USB	Universal Serial Bus
WLAN	Wireless Local Area Network

1 Einleitung

Heutzutage sind Funkverbindungen nicht mehr aus dem Leben wegzudenken: Das Smartphone, mit dem jeder dauerhaft und so gut wie überall erreichbar ist, der Laptop, der sich nach dem Aufklappen automatisch per WLAN mit dem Internet verbindet, und sogar die Uhr, die sich ganz von allein richtig einstellt. Ein Leben ohne Funkverbindungen wäre mit großen Einschränkungen verbunden. Während dies im Heimbereich durchaus noch zu bewältigen wäre, müsste unterwegs wieder auf Telefonzellen zurückgegriffen werden.

Durch immer neue Anforderungen müssen Funkstandards häufig weiter- oder gar komplett neu entwickelt werden. Bisher ist es üblich, Funkstandards nahezu komplett in Hardware zu realisieren, jedoch können so im Nachhinein keine Änderungen mehr am Standard gemacht werden. Im Mobilfunkbereich beispielsweise gibt es sehr oft neue Mobilfunkstandards, für deren Nutzung immer wieder neue Hardware nötig ist. Diese ist oftmals teuer und nur wegen eines neuen Standards ist der Umstieg für den Endnutzer meist nicht rentabel. Bei diesem Ansatz ist die Entwicklung eines Standards sehr aufwändig: Da am fertigen Produkt keine Änderungen mehr gemacht werden können, sollte der Standard fehlerfrei und auch in Zukunft ausreichend sicher sein. Das Thema Sicherheit ist ein wichtiger Punkt, da oft erst beim praktischen Einsatz Sicherheitslücken auffallen.

Das Konzept von Software Defined Radio (SDR) soll diese Probleme lösen: Die Intelligenz einer Funkverbindung, wie vor allem das Protokoll, aber auch die Verschlüsselung sollen in die Software ausgelagert werden. So wäre es möglich, vorhandener Hardware einen neuen Standard per Softwareupdate nachzurüsten und Sicherheitslücken zu beheben, wenn sich das Produkt bereits beim Kunden befindet. Diese Möglichkeiten machen SDR zu einem überaus interessanten Thema.

In dieser Arbeit soll anhand des Festnetzfunkstandards Digital Enhanced Cordless Telecommunications (DECT) die Funktionsweise eines SDR gezeigt und mittels GNU Radio, einem freien Entwicklerwerkzeug zur Implementierung von SDR, eine Anwendung entwickelt werden, die DECT-Stationen aufspüren und Daten empfangen kann.

Dazu werden in Kapitel 3 die nötigen Grundlagen zu SDR, sowie der Aufbau und die nötige Hardware geklärt. In Kapitel 4 wird GNU Radio als Entwicklerwerkzeug für SDR behandelt. Der Aufbau und die Funktionsweise der sogenannten Blöcke und die für diese Arbeit notwendigen Erweiterungen werden erklärt. Kapitel 5 handelt vom Festnetzfunkstandard DECT. Der Test der Hardware und die Umsetzung und Erweiterung des DECT-Empfängers mittels GNU Radio werden in Kapitel 6 dokumentiert. In Kapitel 7 wird die Umsetzung eines Praktikumsversuchs diskutiert und Aufgaben ausgearbeitet. Schließlich findet in Kapitel 8 eine Betrachtung der Ergebnisse und ein Ausblick statt.

2 Zielstellung

In dieser Arbeit soll mittels der SDR-Plattform GNU Radio und einem USB-Dongle ein Empfänger entwickelt werden, der DECT untersützt. Dieser soll DECT-Geräte im Empfangsbereich erkennen und, wenn unverschlüsselte Sprachdaten verfügbar sind, diese über die Soundkarte wiedergeben. Auf Basis der Ergebnisse soll ein Praktikumsversuch ausgearbeitet werden.

2.1 Aufgabe

Folgende Fragen ergeben sich:

- Lässt sich ein DECT-Empfänger mittels RTLSDR, einem Treiber der die Nutzung eines Digital Video Broadcasting – Terrestrial (DVB-T)-Sticks als SDR-Hardware ermöglicht, und der GNU-Radio-Plattform realisieren?
- Lässt sich dieser Empfänger erweitern, sodass die DECT-Kanäle automatisch nach Geräten abgesucht werden?
- Ist es möglich, durch weitere Modifikationen, DECT-Geräte bei einem selbstständigen Kanalwechsel zu verfolgen?

2.2 Lösungsweg

1. Aufsetzen einer funktionierenden GNU-Radio-Plattform unter Ubuntu
2. Installation und Test der Treiber für den Empfänger
3. Test des Empfängers auf Eignung zum Empfangen von DECT
4. Finden und Testen einer DECT-Implementierung für GNU Radio als Arbeitsgrundlage
5. Testen mehrerer Geräte auf unverschlüsselte Sprachübertragung
6. Erweiterung des Empfängers um eine Scan-Funktion, die alle Kanäle selbständig durchsucht
7. Erweiterung des Empfängers, sodass ein ausgewähltes Gerät bei einem Frequenzwechsel verfolgt werden kann
8. Ausarbeiten des Praktikumsversuches

3 Software Defined Radio (SDR)

In diesem Kapitel soll der Begriff Software Defined Radio (SDR) erklärt, sowie die Hardware vorgestellt werden. Dazu werden Grundprinzipien und Aufbau beschrieben.

3.1 Grundprinzip

Joseph Mitola prägte 1991 als Erster den Begriff Software Defined Radio (SDR). Damals war noch nicht klar, wie weit Funkhardware digitalisiert werden würde, doch wurde das Potential des Konzeptes bereits deutlich [1, Seite 2].

Klassische Funkhardware ist, wenn es um den Einsatzbereich geht, sehr eingeschränkt, da ein Gerät nur über einen Funkstandard kommunizieren kann. Diese spezifische Hardware, die nur für einen Einsatzbereich entwickelt ist, hat den Vorteil, dass sie einfach auf Leistung, Größe und Stromverbrauch zu optimieren ist [2, Seite 5].

Das Wireless Innovation Forum definiert SDR folgendermaßen: „Radio in which some or all of the physical layer functions are software defined.“[3]
Mit „physical layer“ sind die unteren vier Schichten des Open Systems Interconnection (OSI)-Modells gemeint, welches die Design-Grundlage für alle Kommunikationsprotokolle darstellt (Abb. 1). Diese Schichten sind für die eigentliche Datenübertragung, die Zuverlässigkeit, Vermittlung und Segmentierung zuständig.

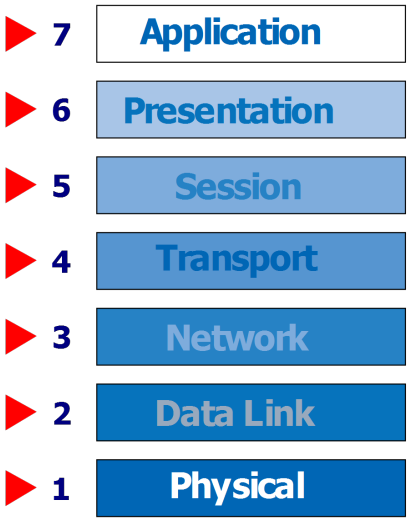


Abbildung 1: OSI-Modell [1]

Bei einem SDR sollen also möglichst viele dieser Schichten „software defined“, das heißt auf Software basierend, sein, damit ohne Änderungen der Hardware Änderungen der Funkparameter möglich sind. Die Abgrenzung von SDR ist mit dieser Definition leider nicht sehr genau möglich. Die meisten modernen Geräte sind eher „software controlled“ als „software defined“. Ein Smartphone zum Beispiel unterstützt verschiedene Kommunikationsstandards zwischen denen, gesteuert von Software auf

[1] Quelle: `https://openclipart.org/detail/190147/osi-network-model`

dem Smartphone, gewechselt wird. Der Hauptunterschied ist, dass ein „software controlled radio“ auf die Funktionalität beschränkt ist, die der Designer vorgesehen hat, während ein SDR umprogrammiert werden kann, sodass es Aufgaben erledigt, die nie geplant waren [2, Seite 6,7].

3.2 Aufbau

Das ideale SDR, auch Software Radio (SR) genannt, besteht nur aus einer Antenne, einem Analog-Digital-Umsetzer (ADU) und einem Digital-Analog-Umsetzer (DAU) (Abb. 2). Ausgangssignale werden durch den DAU umgewandelt und gesendet, während Eingangssignale direkt nach der Antenne digitalisiert und danach in Software ausgewertet werden.

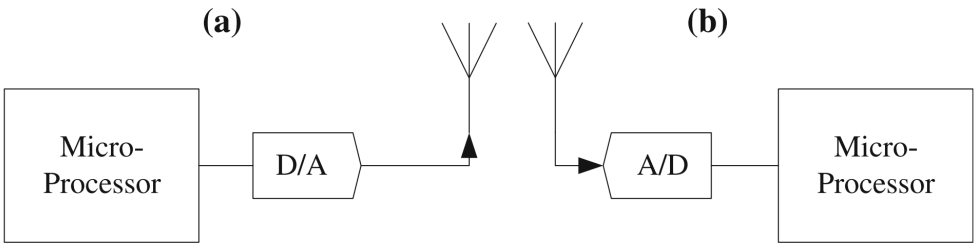


Abbildung 2: Ideales SDR: (a) Sender, (b) Empfänger [2]

Bei diesem Aufbau stößt man jedoch auf einige Probleme:
Das SR sollte in der Lage sein, einen sehr großen Frequenzbereich zu empfangen (<1 MHz bis >60 GHz), jedoch müssen Antennen für einen bestimmten, kleinen Frequenzbereich designed werden, um optimale Empfangseigenschaften aufzuweisen.

Radio Frequency Front-End (RFFE) bezeichnet die Komponenten, die Signale direkt nach der Antenne verarbeiten. Ohne ein solches RFFE sind die Auswahl des gewünschten Signals und die Unterdrückung von Störsignalen nicht möglich.

Wenn das gewünschte Frequenzband nicht selektiert empfangen werden kann, muss das gesamte Frequenzband digitalisiert werden. Laut dem Nyquist-Shannon-Theorem muss die Abtastrate mindestens doppelt so hoch sein wie die höchste im Signal enthaltene Frequenz [4]. In diesem Beispiel bedeutet das, dass ADU und DAU mit 120 GHz (2x 60 GHz) abtasten müssten, jedoch sind aktuelle Umwandler nicht mal annähernd dazu in der Lage.

Um die Vielzahl an Störsignalen vom eigentlichen Signal zu unterscheiden, muss der Umwandler sehr hoch auflösen. Weiterhin muss er sehr linear arbeiten, da sonst unerwünschte Intermodulationen auftreten.

Das letzte große Problem stellt die extreme Datenmenge dar, die anfallen würde: Bei einer Auflösung des ADU von 24 bit und der nötigen Abtastrate von 120 GSamples/s fällt eine Datenmenge von 360 GigaByte/s an. Derzeit kann kein Rechner eine solche Datenmenge in Echtzeit verarbeiten.

[2]Quelle: [2, Seite 7]

Letztendlich kommt man zu dem Schluss, dass es ein ideales SR nicht geben kann. Ein solches Gerät würde sehr viel Platz einnehmen und hätte einen enormen Stromverbrauch, sodass es nur für Forschungszwecke zu gebrauchen wäre [2, Seite 7,8]. Um ein SDR zu realisieren, muss zwangsweise ein Kompromiss eingegangen werden, sodass vor allem der Energieverbrauch in einem annehmbaren Rahmen bleibt. Deshalb enthält handelsübliche SDR-Hardware mindestens einen Hochfrequenzverstärker, der einzelne Frequenzbereiche selektiv verstärkt und Filter, die Störsignale separieren.

3.3 Hardware

Die Preise für Hardware, mit der man ein SDR experimentell betreiben kann, reichen von mehreren tausend Euro teuren Entwicklergeräten bis kostenlos, da selbst mit dem Mikrofoneingang der Soundkarte eines Computers Signale empfangen werden können. Die Geräte unterscheiden sich hauptsächlich im Frequenzbereich, der maximalen Bandbreite und der Abtastrate. Als Schnittstelle zum Computer kommt meist Universal Serial Bus (USB), Ethernet oder Peripheral Component Interconnect Express (PCIe) zum Einsatz. Günstige Hardware ist meist nur in der Lage, Signale zu empfangen - Transceiver, die empfangen und senden können, sind oft wesentlich teurer.

Als Beispiel sei die HackRF One Platine genannt (Abb. 3). Sie ist ein Open Source Hardware Transceiver, der ursprünglich als Kickstarter-Kampagne finanziert wurde und im Hobbybereich aufgrund des vergleichsweise geringen Preises von rund 300€ sehr beliebt ist.

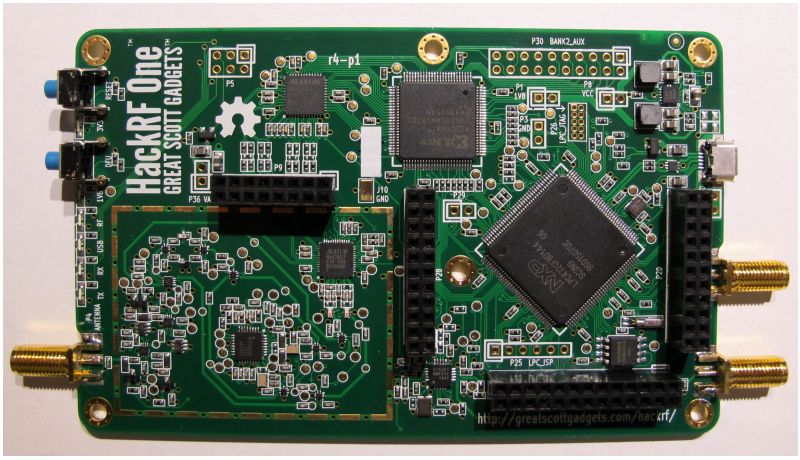


Abbildung 3: HackRF One Platine [3]

Wenn nur Signale empfangen werden sollen, stellt ein DVB-T-Stick (Abb. 4) eine sehr günstige Option dar. Mithilfe eines veränderten Treibers kann der Stick mit der SDR-Plattform GNU Radio genutzt werden. Für diese Arbeit wurde ein Empfänger der Marke *Terratec TStick+* verwendet.

[3] Quelle: `https://upload.wikimedia.org/wikipedia/commons/0/0b/SDR_HackRF_one_PCB.jpg`

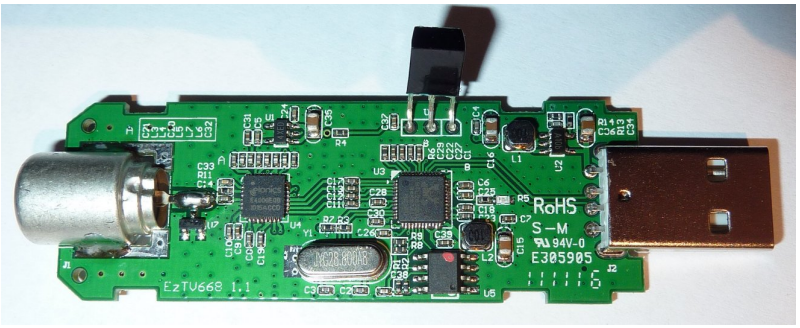


Abbildung 4: ezcap USB 2.0 DVB-T/DAB/FM dongle [4]

[4]Quelle: `https://osmocom.org/attachments/download/2243/ezcap_top.jpg`

4 GNU Radio

In diesem Kapitel werden die GNU Radio Plattform, sowie die Erweiterungen, die für die Integration eines DVB-T-Sticks als Empfänger nötig sind, vorgestellt.

4.1 Allgemeines

GNU Radio ist ein freies Open Source Entwicklerwerkzeug zur Implementierung von Software Defined Radio (SDR) mittels Signalverarbeitungsblöcken. Es kann mit externer Funkhardware oder als Simulation ohne Hardware genutzt werden. GNU Radio ist im Hobbybereich weit verbreitet, wird aber auch in der Wissenschaft und im kommerziellen Bereich genutzt [5]. GNU Radio Programme sind hauptsächlich in der Programmiersprache Python geschrieben, wobei echtzeitkritische Signalverarbeitungen in C++ implementiert sind.

4.2 Funktionsweise

GNU Radio verarbeitet den Datenstrom, der vom Empfänger kommt, und bereitet Daten vor, um diese zu senden. Dabei ist die gesamte Datenverarbeitung in Blöcke eingeteilt, die beliebig zu einem sogenannten *flowgraph* zusammengestellt werden können. GNU Radio liefert eine Vielzahl von Filtern, Kanal-Codes, Synchronisationselementen, Equalizern, Demodulatoren, Vocodern, Dekodierern und vieles mehr. Diese Blöcke stellen die Hardwarekomponenten klassischer Funkhardware dar [5]. Falls nötig können auch selbst Blöcke hinzugefügt werden. Dabei sollte darauf geachtet werden, dass ein Block genau eine Aufgabe erledigen soll, damit er so vielseitig wie möglich eingesetzt werden kann. GNU Radio bietet die Möglichkeit, diese Blöcke miteinander zu verbinden und regelt den Datenfluss zwischen diesen.

Für die digitale Verarbeitung von Daten kommen in GNU Radio verschiedene Datentypen zum Einsatz: Der Datenstrom des Empfängers besteht in den meisten Fällen aus komplexen Zahlen, also einem Datentyp, bei dem jedes Datum aus zwei Float-Werten, dem Imaginärteil und dem Realteil, besteht. Zwischen Blöcken können jedoch beliebige Datentypen zum Einsatz kommen, üblicherweise Short, Byte, Integer und Float. Der Datentyp am Eingang und Ausgang eines Blockes muss dabei nicht gleich sein.

Um mit GNU Radio eine SDR-Anwendung zu bauen, muss man Blöcke, die die einzelnen Funktionseinheiten darstellen, miteinander zu einem *flowgraph* kombinieren. Sollte ein benötigter Block nicht verfügbar sein, kann man diesen mit dem *gr_modtool* selbst hinzufügen. Beim Starten des *flowgraphs* ruft GNU Radio nun jeden Block nacheinander auf und stellt sicher, dass Daten von einem Block zum nächsten weitergereicht werden [6]. Ergebnisse können durch Graphical User Interface (GUI)-Blöcke dargestellt werden. Diese nutzen die freien und plattformübergreifenden GUI-Toolkits *wxWidgets* und *Qt*.

Das Comprehensive GNU Radio Archive Network (CGRAN) ist eine Plattform, auf der GNU Radio Programme geteilt werden können.

4.3 Arten von Blöcken

Die verschiedenen Funktionseinheiten in GNU Radio, auch Blöcke genannt, haben unterschiedliche Merkmale und lassen sich nach diesen einteilen.

4.3.1 Einteilung nach IO

Jeder Block hat eine gewisse Anzahl an Eingängen und Ausgängen, diese werden *Ports* genannt. Jeder dieser Ports hat einen oder mehrere Datentypen, die er akzeptiert bzw. ausgibt. Die Anzahl der Ports eines Blockes ist beliebig.
Einen Block ohne Eingänge nennt man *Source*, also Quelle. Ein solcher Quellblock kann Daten erzeugen, aus einer Datei wiedergeben oder von einem externen Empfänger, wie in Kapitel 3.3 vorgestellt, lesen.
Ein Block ohne Ausgänge nennt man *Sink*. Ein Ausgabeblock gibt typischerweise Daten an einen externen Sender weiter. Er kann die Daten aber auch in eine Datei schreiben oder auf dem GUI, zum Beispiel in einem Diagramm, darstellen.

4.3.2 Einteilung nach Datenfluss

Blöcke können auch nach dem Verhältnis von Ein- und Ausgangsdaten unterschieden werden.
Ein Block, der gleich viele Daten konsumiert wie er produziert (Verhältnis 1:1), wird Synchron-Block genannt. Ein Synchron-Block, der keinen Eingang hat, ist ein Quell-Block. Analog dazu ist ein solcher Block ohne Ausgang ein Ausgabe-Block. Ist das Verhältnis von Ein- und Ausgangsdaten N:1, wird von einem Dezimierer-Block gesprochen. Ein Block mit Verhältnis 1:M heißt Interpolations-Block. Gibt es kein festes Verhältnis, wird von einem Basisblock gesprochen. Alle oben genannten Blöcke sind Vereinfachungen des Basisblocks [7].

4.4 Kommunikation zwischen Blöcken

Oft ist es nötig, zusätzlich zum Datenstrom Informationen an einen Block weiterzugeben. Dafür gibt es in GNU Radio zwei Möglichkeiten: Stream Tags und Message Passing. Diese Mechanismen werden im Folgenden erklärt.

4.4.1 Stream Tags

Mit *Stream Tags* können Metadaten synchron zum Datenstrom transportiert werden. Ein solches *Stream Tag* besteht aus einem *offset*, der angibt, zu welchem Datum im Datenstrom der Tag gehört. Weiterhin enthält er einen *key* der zur Identifikation notwendig ist, da einem Datum mehrere Tags zugewiesen werden können. Zu jedem Tag gehört schließlich ein Wert, der die eigentlichen Metadaten enthält [7].

4.4.2 Message Passing

Das *Message Passing Interface* arbeitet komplett anders als *Stream Tags*. Nachrichten haben keinen *offset* und normalerweise auch keinen *key*, auch wenn es möglich ist, Paare aus Schlüsseln und Werten zu senden. Nachrichten können nicht über

einen normalen Ausgang eines Blocks gesendet werden, es muss ein spezieller *message port* angelegt werden [8].
Da Nachrichten asynchron zu den Daten übertragen werden, eigenen sie sich hauptsächlich dafür, den nachfolgenden Block zu steuern.

4.5 Kommunikation mit Blöcken

Vorallem in GNU-Radio-Anwendungen mit GUI ist es nützlich, einem Block Parameter übergeben zu können. So möchte man zum Beispiel über ein Dropdown-Menü den Frequenz-Parameter des Empfängers ändern. Um das zu ermöglichen, können Parameter für Blöcke definiert werden. In diesem Beispiel ist der Wert des Dropdown-Menüs der Parameter des Blockes. Sobald nun der Wert des Menüs verändert wird, startet eine Callback-Funktion, die Parameter an den Block weitergibt. Durch diesen Mechanismus kann während der Laufzeit auf Nutzereingaben reagiert werden.

4.6 GNU Radio Companion

Der GNU Radio Companion (GRC) ist ein Softwaretool mit grafischer Oberfläche (Abb. 5), mit dem sich GNU Radio Blöcke zusammenstellen und verbinden lassen. Solange nur Standardblöcke benutzt werden, kann ganz ohne Programmierkenntnisse ein *flowgraph* generiert und ausgeführt werden. Verschiedenste Blöcke lassen sich komfortabel per Drag&Drop anordnen [5]. Ports können miteinander verbunden und Parameter von Blöcken per Menü festgelegt werden.
Dieses Tool macht die komplette GNU Radio Plattform sehr zugänglich und lädt zum Experimentieren ein. Der GNU Radio Companion wurde auch für diese Arbeit verwendet.

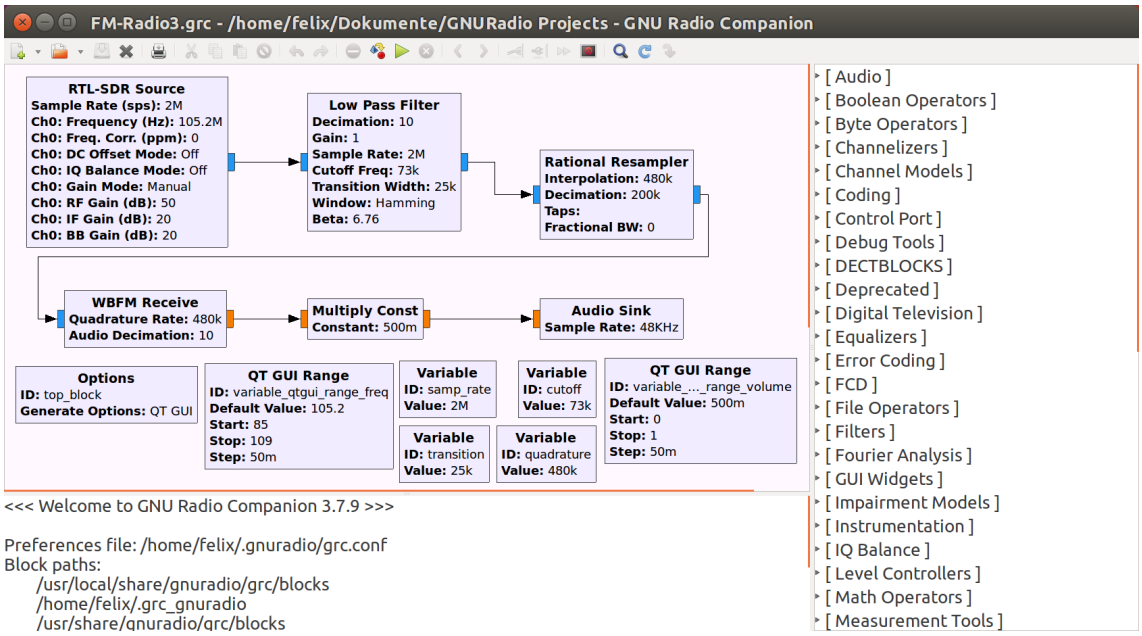


Abbildung 5: Nutzeroberfläche des GNU Radio Companion

4.7 Osmocom RTLSDR

Open Source Mobile Communications (Osmocom) ist ein Projekt, das sich mit der Entwicklung von Software und Werkzeugen für eine Vielzahl von Mobilfunkstandards beschäftigt [9]. Osmocom hat *OsmoSDR* entwickelt, eine günstige SDR Hardware, die durch das Paket *GrOsmoSDR* in GNU Radio eingebunden werden kann [10].

RTLSDR, eine weitere Entwicklung von Osmocom, ist ein Treiber, der es ermöglicht, DVB-T-Sticks, die auf dem Realtek RTL2832U Chip basieren, als günstigen SDR-Empfänger zu nutzen. Der Realtek-Chip ermöglicht es, Rohdaten an den Host-PC zu senden, um diese Zweckentfremdung zu ermöglichen [11]. Eine Liste mit kompatiblen Geräten[5] ist online verfügbar.

Der Realtek RTL2832U gibt 8-bit Komplex-Samples mit einer theoretisch maximalen Abtastrate von 3,2 MSamples/s aus [11]. Die tatsächliche Abtastrate ist niedriger und von Gerät zu Gerät verschieden. Der Frequenzbereich hängt vom verbauten Tuner ab, wobei der Elonics E4000 Tuner den größten Frequenzbereich ermöglicht (siehe Tabelle 1) [11].

Tuner	**Frequenzbereich**
Elonics E4000	52 - 2200 MHz
Rafael Micro R820T	24 - 1766 MHz
Rafael Micro R828D	24 - 1766 MHz
Fitipower FC0013	22 - 1100 MHz
Fitipower FC0012	22 - 948.6 MHz
FCI FC2580	146 - 308 MHz and 438 - 924 MHz

Tabelle 1: Tuner mit jeweiligen Frequenzbereichen [6]

[5] `https://www.reddit.com/r/RTLSDR/comments/s6ddo/rtlsdr_compatibility_list_v2_work_in_progress/`

[6] Quelle: `https://osmocom.org/projects/sdr/wiki/rtl-sdr`

5 Digital Enhanced Cordless Telecommunications

Digital Enhanced Cordless Telecommunications (DECT) ist ein sehr weit verbreiteter Standard zur kabellosen Sprachübertragung. In diesem Kapitel soll ein Überblick über den Standard gegeben und das Protokoll grob erklärt werden.

5.1 Allgemeines

DECT ist ein in Europa entwickelter Standard und stand ursprünglich für Digital European Cordless Telephone. Da später auch eine Variante für die USA entwickelt wurde, benannte man es in Digital Enhanced Cordless Telecommunications um [12]. Nachdem 1992 die erste Version von DECT verfügbar war, konzentrierte sich die Arbeit auf die Einführung von *Access Profiles*, die die reibungslose Interoperation mit verschiedenen Telefonnetzen ermöglichten [12, 13]. Ende 1995 wurde die zweite Version von DECT fertiggestellt [13]. Der DECT-Standard wurde vom European Telecommunications Standards Institute (ETSI) festgeschrieben.

DECT bietet folgende Eigenschaften:

- Betrieb in einem exklusiven Frequenzband (EU)
- Einsparung von Frequenzen
- Optimierung auf minimale Verzögerung und gleichbleibende Verbindungsqualität
- geringer Stromverbrauch bei kleiner Bauweise und geringen Kosten für die Endgeräte
- hörbar bessere Tonqualität als vorhergehende Standards
- hohe Abhörsicherheit
- universeller Einsatz
- Telefonie-Leistungsmerkmale
- gleichzeitiger Betrieb mehrerer Handteile
- gebührenfreie interne Gespräche
- Mobilteile sind an mehreren Basisstationen nutzbar
- herstellerunabhängige Nutzung von Mobilteilen an den Basisstationen
- Handover (automatischer Wechsel der Basisstation) [12]

DECT wird hauptsächlich für Festnetztelefone und Babyphones eingesetzt. Es gab auch Hardware, die DECT nutzte, um Computer zu vernetzen und diese mit dem Internet zu verbinden [14]. Dies setzte sich jedoch wegen der damals schon geringen Übertragungsrate nicht durch.

Der Standard wird stetig weiterentwickelt, um mit technologischen Entwicklungen Schritt zu halten. So wird seit 2006 an New Generation DECT (NG-DECT) gearbeitet. NG-DECT fügt viele Weiterentwicklungen zum Basisstandard hinzu, wobei trotzdem eine Abwärts-kompatibilität zu älteren Geräten geboten wird. Unter anderem werden eine höhere Sprachqualität, Videotelefonie, verbesserte Sicherheit und Software Updates unterstützt [15].

Eine weitere Entwicklung ist DECT Ultra Low Energy (ULE), was für Heim- und Industrieautomatisierung optimiert wurde. Sie reiht sich mit einem geringeren Ener-

gierverbrauch als Wireless Local Area Network (WLAN) (IEEE 802.11) und einer wesentlich höheren Reichweite als Bluetooth Low Energie (IEEE 802.15) zwischen diesen beiden Standards ein [15].

5.2 Der Standard

Der DECT Standard ist in Schichten unterteilt und im Standard ETS 300 175, Teil 1 bis 8, festgeschrieben (Tab. 2).

Teil	Titel	Beschreibung
1	Overview	Allgemeine Vorstellung und Überblick über die anderen Teile des ETSI-EN-300-175-Standards
2	Physical Layer (PHL)	Funkparameter von DECT, z.B. Frequenzbereich, Timing, Sendeleistung, Synchronisation von Sender und Empfänger, Modulationsverfahren
3	Medium Access Control (MAC) layer	Beschreibung von Prozeduren, Nachrichten und Protokollen für Verbindungsaufbau, Kanalauswahl, Kanalwechsel, Verbindungsabbau, Qualitätssicherung u.a.
4	Data Link Control (DLC) layer	Vorschriften, um eine zuverlässige Datenverbindung zum Network layer sicherzustellen
5	Network (NWK) layer	Unter Anderem Beschreibung von Funktionen zur Verbindungs- und Anrufkontrolle
6	Identities and addressing	Identifizierung und Adressierung von (Radio) Fixed Part (FP), (Radio) Portable Part (PP), Verbindungen und Hardware
7	Security features	Sicherheit von Authentifikation und Verschlüsselung
8	Speech and audio coding and transmission	Anforderungen an Geräte, um Echtzeitkommunikation in beide Richtungen bereitzustellen. Festlegung der Sprachcodierungsalgorithmen.

Tabelle 2: DECT-Layer mit Beschreibung [7]

5.2.1 Eigenschaften

In Europa nutzt DECT das Frequenzband von 1 880 MHz bis 1 980 MHz, welches in 10 Kanäle mit jeweils 1 728 kHz Abstand unterteilt ist. Geräte dürfen mit einer maximalen Leistung von 250 mW senden. Die rohe Datenrate liegt, je nach Modulationsverfahren, zwischen 1 152 und 6 912 kbit/s. Die Kommunikation zwischen den Geräten wird durch ein Time Division Multiple Access (TDMA), ein sogenanntes Zeitslot-Verfahren, geregelt [16, Seite 27]. In allen folgenden Ausführungen dieser Arbeit wird von einer zweistufigen Modulation ausgegangen, das heißt, ein Symbol ist gleich ein Bit.

[7] Quelle: [13] [16, Seite 24, 25]

5.2.2 Fixed Part und Portable Part

DECT-Geräte lassen sich in zwei Gruppen einteilen: (Radio) Fixed Part (FP) und (Radio) Portable Part (PP). Als FP wird die Basistation bezeichnet - diese verändert ihren Standort in der Regel nicht. Der PP ist bei einem Festnetztelefon das Handteil. Der FP sendet ständig ein sogenanntes *beacon*-Signal. Es enthält ID, Funktionsumfang und Status der Basistation, sowie Informationen über eingehende Anrufe. Ein PP analysiert diese Signale und erkennt, ob es Senderechte hat, welche Fähigkeiten der Basisstation mit den Anforderungen des Handteils zusammen passen und, falls eine Kommunikation nötig ist, ob die Station freie Kapazitäten hat [13].

Ein PP befindet sich zu jeder Zeit in einem der in Abbildung 6 zu sehenden Zustände.

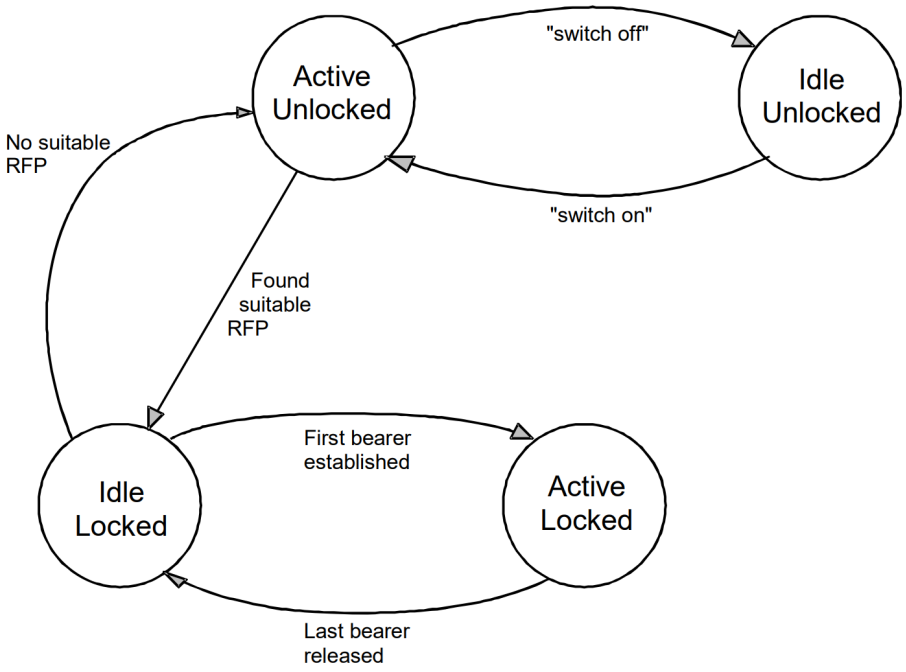


Abbildung 6: Zustände eines (Radio) Portable Part (PP) [8]

- **Active Locked:** Das PP ist mit mindestens einem FP synchronisiert und es läuft mindestens eine Übertragung.
- **Idle Locked:** Das PP ist mit mindestens einem FP synchronisiert und in der Lage, eine Übertragung zu starten oder entgegenzunehmen.
- **Active Unlocked:** Das PP ist nicht synchronisiert und kann dementsprechend auch keine Übertragung starten oder entgegennehmen. Es versucht, ein passendes FP zu finden und den Idle Locked Status zu erreichen.
- **Idle Unlocked:** Das PP ist nicht synchronisiert und versucht nicht, sich mit einem FP zu verbinden. In den meisten Fällen ist das Gerät ausgeschaltet [17, Seite 26].

[8] Quelle: [17, Seite 26]

Die Zustände des FP, bis auf **Inactive**, in dem er ausgeschaltet ist, sind in Abbildung 7 zu sehen.

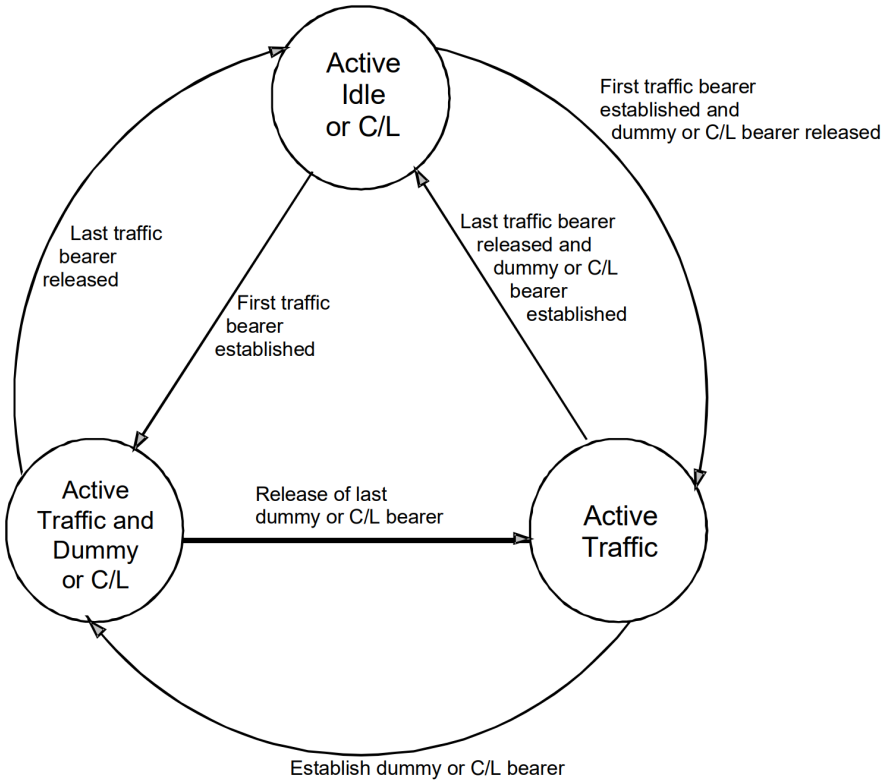


Abbildung 7: Zustände eines (Radio) Fixed Part (FP) [9]

- **Active Idle or C/L:** Der FP hat entweder mindestens eine Dummy-Übertragung (sendet das Beacon-Signal) oder eine verbindungslose Übertragung und einen Empfänger, der die Kanäle in einer bekannten Sequenz absucht.
- **Active Traffic:** Der FP hat mindestens eine aktive Übertragung, aber keine Dummy- oder verbindungslose Übertragung.
- **Active Traffic and Dummy or C/L:** Der FP hält eine aktive Übertragung aufrecht und hat eine Dummy- oder verbindungslose Übertragung [17, Seite 27].

5.2.3 Übertragungstechnik

DECT nutzt Time Division Multiple Access (TDMA) als Zugangsverfahren, um den Zugriff auf einen gemeinsam genutzten Funkkanal zu regeln [16, Seite 27]. TDMA ist ein Zeitmultiplex-Verfahren, bei dem Nutzer Daten nacheinander in eigenen Zeitslots übertragen. Dabei müssen alle Teilnehmer vollständig synchronisiert sein, sonst überlagern sich diese [18]. Alle Zeitslots gemeinsam werden *Frame* genannt. Bei DECT ist einer dieser Frames 10 ms lang und enhält 24 Slots. In den ersten 12 Slots darf nur der FP und in den letzten 12 Slots nur der PP senden (siehe Abb. 8). Somit sind pro Kanal 12 simultane Sprachverbindungen möglich [13].

[9] Quelle: [17, Seite 27]

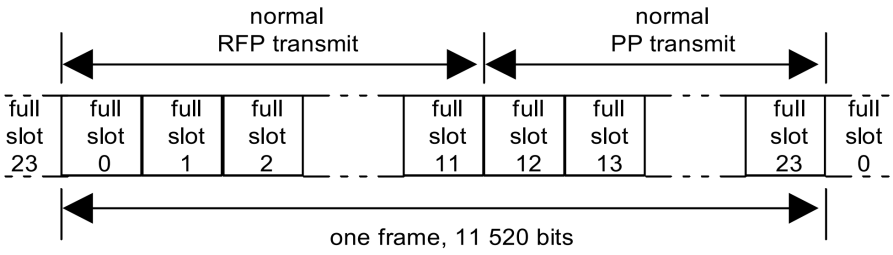


Abbildung 8: Time Division Multiple Access (TDMA) Frame [10]

Ein Full-Slot enthält 480 Bits, was bei einer Framelänge von 24 Slots 11 520 Bits pro Frame ergibt. Durch die Framezeit von 10 ms beträgt die rohe Gesamtdatenrate 1 152 kBits/s. Da dieses Verfahren auf allen 10 Kanälen von DECT zum Einsatz kommt, können theoretisch bis zu 120 Verbindungen gleichzeitig an einem Ort aufgebaut werden, ohne sich zu gegenseitig zu beeinträchtigen.

Es gibt aber auch die Möglichkeit, halbe Slots oder doppelte Slots zu senden [19, Seite 14]. Weiterhin können bis zu 23 Full-Slots für eine Senderichtung gebündelt werden, wobei mindestens ein Slot für die Empfangsrichtung reserviert sein muss. Diese Bündelung kommt bei Datenübertragungen zum Einsatz [12].

5.2.4 Aufbau eines Paketes

In einem Slot wird ein Paket übertragen. Das 420 bzw. 424 Bit lange P32-Paket wird bei den meisten Verbindungstypen benutzt [19, Seite 18] und soll hier als Beispiel dienen. Jedes Paket besteht aus einem Synchronisations-, Daten- und einem optionalen Z-Feld (siehe Abb. 9).

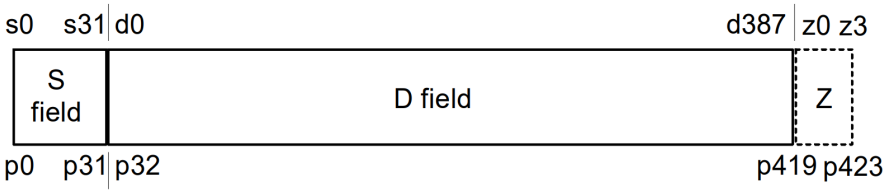


Abbildung 9: Aufbau eines P32-Paketes [11]

Das Synchronisationsfeld wird, wie der Name schon vermuten lässt, vom Empfänger für die Takt- und Paketsynchronisation genutzt. Es ist 32 Bit lang, wobei die ersten 16 Bit eine Präamble und die letzten 16 Bit das Synchronisationswort sind. Die S-Felder für FP und PP sind dabei wie folgt definiert:
FP: 10101010 10101010 11101001 10001010.
PP: 01010101 01010101 00010110 01110101.
Die Bitfolgen sind das Inverse voneinander [19, Seite 22].

Das optionale Z-Feld kann zur frühzeitigen Erkennung eines asynchronen Störsignals, das in das Ende des Paketes hineinläuft, genutzt werden. Es ist vier Bit lang und

[10] Quelle: [19, Seite 14]
[11] Quelle: [19, Seite 23]

enthält die letzten vier Bit des Datenfeldes. Sind diese Bitfolgen nicht gleich, wurde ein Störsignal erkannt [19, Seite 24].

Das 388 Bit lange Datenfeld besteht, wie in Abbildung 10 zu sehen, erneut aus drei Feldern: A-, B- und X-Feld.

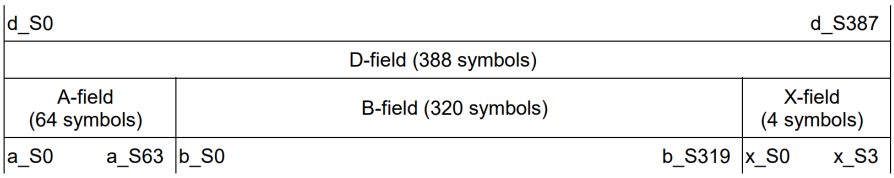


Abbildung 10: Aufbau eines P32-D-Feldes [12]

Das X-Feld enthält analog zu dem Z-Feld die letzten vier Bit des B-Feldes und wird erneut zur Fehlererkennung genutzt [17, Seite 102]. Die Sprachdaten eines Pakets liegen im B-Feld. Dieses ist 320 Bit lang. Das 64 Bit lange A-Feld lässt sich wieder unterteilen: In Header-, Tail- und Redundanz-Bits (siehe Abb. 11).

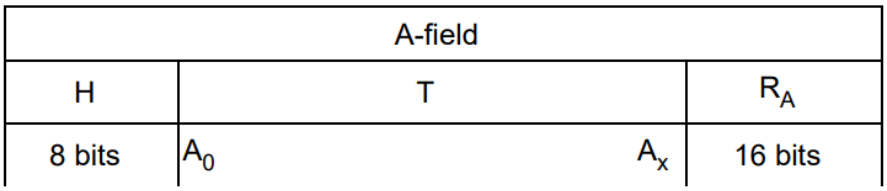


Abbildung 11: Aufbau eines P32-A-Feldes [13]

Der Header enthält Informationen über den Inhalt der Tail-Bits und des B-Feldes. Der Tail ist bei einem P32 Paket 40 Bit lang und kann Identitäts-, System-, Anruf- oder Kontrollinformationen enthalten [17, Seite 108-151]. Die 16 Redundanz-Bits ermöglichen eine Fehlerkontrolle für alle Daten im A-Feld [17, Seite 69].

5.3 Sicherheit

Der DECT-Standard sieht generell eine Verschlüsselung der Sprachdaten vor. Obwohl die meisten Dokumentationen zum Standard öffentlich zugänglich sind, wird gerade der Verschlüsselungsalgorithmus geheim gehalten und ist nur manchen Herstellern von DECT-Hardware zugänglich [20]. Einige, hauptsächlich alte, Geräte verzichten jedoch komplett auf diese Verschlüsselung. Eine Liste dieser Geräte ist online verfügbar[14]. Diese Telefone können mit sehr wenig Aufwand abgehört werden.

Doch selbst, wenn die Verschlüsselung in die Geräte eingebaut wurde, ist die Übertragung nicht sicher. Wie bereits Ende 2008 auf dem 25. Chaos Communication Congress, einem jährlichen Treffen der Hackerszene, das vom Chaos Computer Club organisiert wird, vorgeführt wurde, lässt sich die Verschlüsselung erstens leicht komplett umgehen und zweitens, mit nur wenig Mehraufwand, sogar knacken. Der Schutz

[12] Quelle: [17, Seite 67]
[13] Quelle: [17, Seite 69]
[14] `https://dedected.org/trac/wiki/ListOfPhones`

kann umgangen werden, da eine Basistation die Verschlüsselung für alle Geräte ausschaltet, sobald ein angemeldetes Gerät diese nicht unterstützt [21]. Ein Jahr später gelang es dem Projekt deDECTed, die Verschlüsselung tatsächlich zu knacken. Dazu wurde das A- und B-Feld einer Aufzeichnung ausgewertet. Um durch Analyse des A-Feldes an den Schlüssel zu gelangen, waren 24 Stunden an aufgezeichneten Daten nötig. Bei einer Auswertung des B-Teils gelang es der Gruppe mit nur 3 Stunden aufgezeichneter Stille den Schlüssel zu berechnen. [22].

Das DECT-Forum hat die Sicherheitsschwächen anerkannt und begonnen an einer neuen, offenen Verschlüsselung zu arbeiten [23].

6 DECT-Monitor in GNU Radio

In diesem Kapitel wird die Vorgehensweise für die Entwicklung des DECT-Monitors erläutert.

6.1 Vorarbeiten

Als Arbeitsgrundlage wurde zuerst eine geeignete Implementation von DECT in GNU Radio gesucht. Das Modul *gr-dect2* von Pavel Yazev[15] stellte sich dabei als geeignet heraus. Entwickelt wurde dieses Modul mit Hardware von Ettus Research als Empfänger und einem Babyphone von Motorola als Sender. In Abbildung 12 ist der komplette *flowgraph* zu sehen.

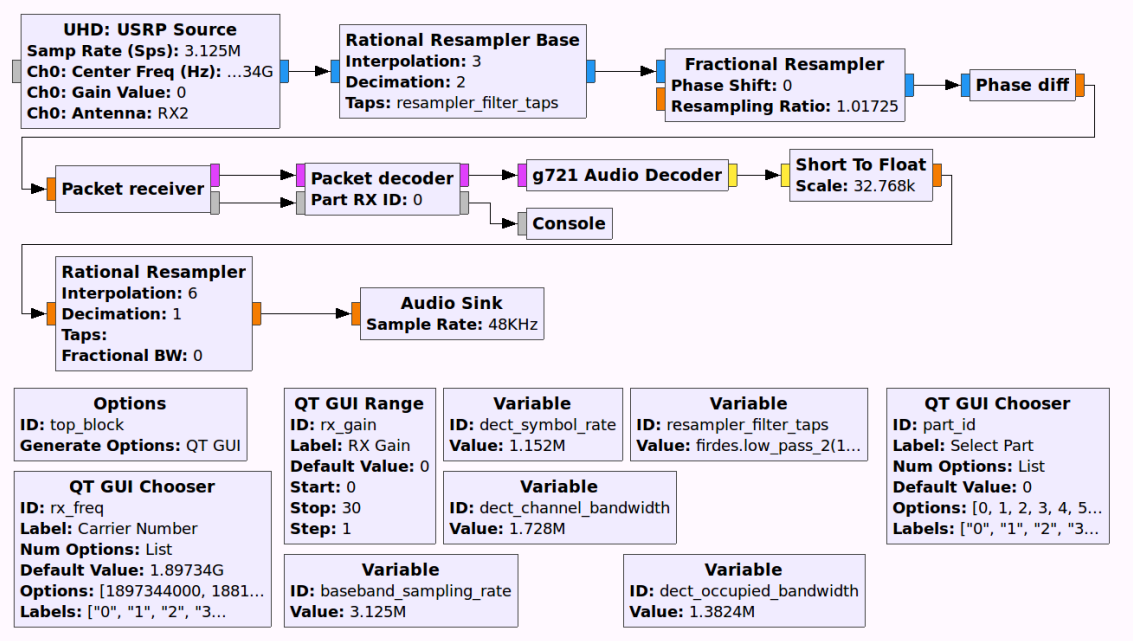


Abbildung 12: Flowgraph von gr-dect2 im GRC

Wie in Abbildung 13 zu sehen bietet die Implementation ein Dropdown-Menü, mit dem der Kanal eingestellt werden kann, eine Konsole, die vorhandene Geräte auflistet, und ein weiteres Dropdown-Menü, mit dem das zu empfangende Gerät ausgewählt wird. Sendet das ausgewählte DECT-Gerät unverschlüsselte Sprachdaten, werden diese über die Lautsprecher wiedergegeben.

[15] https://github.com/pavelyazev/gr-dect2

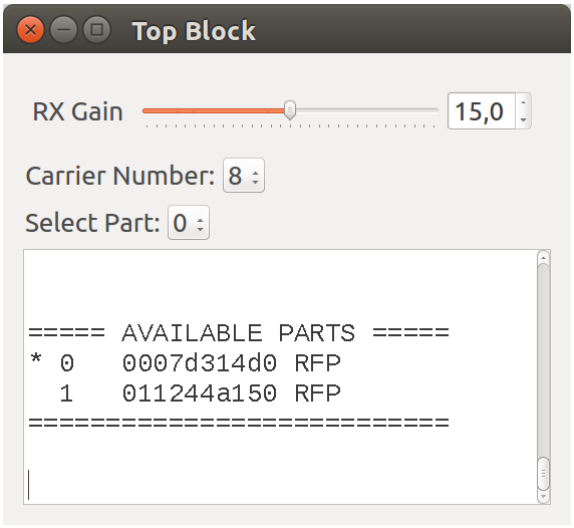


Abbildung 13: GUI von gr-dect2

Das Modul besteht aus vier Blöcken:

- Phase diff
- Packet Receiver
- Packet Decoder
- Console

Der Block *Phase diff* verrechnet die Phasendifferenz des Eingangssignals vom Typ complex und gibt float-Werte aus.

Der Block *Packet-Receiver* liest diese Daten und sucht nach dem Sync-Feld. Dieser Zustand wird als *WAIT_BEGIN* bezeichnet(siehe Abb. 14). Wurde ein Sync-Feld gefunden, wechselt der Block in den *WAIT_END*-Zustand und das Gerät wird in einer Liste registriert. War dies erfolgreich, geht der Block in den *POST_WAIT*-Zustand über und das D-Feld wird an den *Packet-Decoder*-Block weitergereicht. Weiterhin prüft der Block, ob ein Gerät nicht mehr erreichbar ist, indem für jedes Gerät durch Zählen von Samples die inaktive Zeit gemessen wird. Ist eine gewisse Zeit überschritten, zählt das Gerät als verschwunden. Der Eintrag in der Liste wird gelöscht.

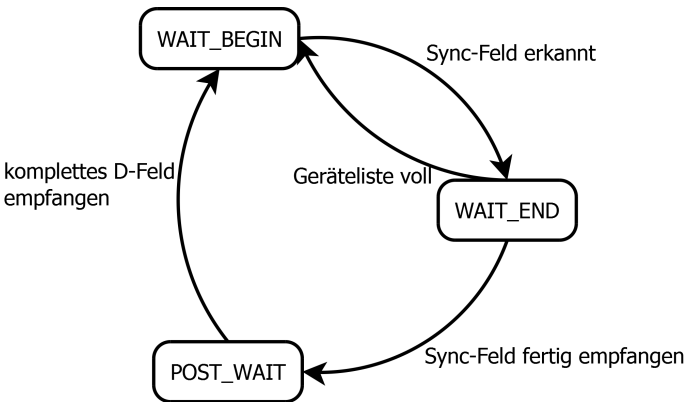


Abbildung 14: Zustandsdiagramm des Packet-Receiver-Blocks

Der *Packet-Decoder* erhält die D-Felder vom *Packet-Receiver*-Block und entnimmt zuerst den A-Teil. Die ID des Gerätes, sowie die Information, ob ein B-Teil vorhanden ist, werden entnommen und in eine Liste eingetragen. Geräte mit gleicher ID werden gesucht und als Paar in der Liste notiert. Die Liste der erkannten Geräte wird an den *Console*-Block weitergegeben. Wurde per Dropdown-Menü ein Gerät ausgewählt, werden falls vorhanden die Sprachdaten dieses Gerätes wiedergegeben.

Der *Console*-Block erzeugt das Textfeld auf dem GUI, in dem Informationen über erkannte Geräte angezeigt werden.

Die Kommunikation zwischen *Packet-Receiver* und *Packet-Decoder* erfolgt über zwei Ports: Auf einem werden die Daten byteweise und mit Stream Tags versehen übertragen. Der andere ist ein Message Port. Die Stream Tags enthalten die ID, unter welcher das Gerät in der Liste gespeichert ist, eine Sequenznummer und den Typen des Gerätes (FP oder PP). Über den Message Port wird eine Nachricht versendet, wenn ein Gerät nicht mehr erreichbar ist. So wird die interne Liste der Geräte stets konsistent gehalten.

6.2 Test der Hardware

Um die Umsetzbarkeit der Aufgabe zu prüfen, mussten zunächst Tests mit der Hardware durchgeführt werden. Diese sind nachfolgend beschrieben.

6.2.1 Empfänger

Der Empfänger selbst und die erfolgreiche Installation der Treiber wurden zunächst mit einem einfachen *flowgraph* zum Empfang von Radio getestet. Der Test war erfolgreich.

Weiter sollte festgestellt werden, ob der vorhandene DVB-T-Stick für den Empfang von DECT geeignet ist. Der Empfänger nutzt den Elonics E4000 Tuner, der laut Tabelle 1 aus Abschnitt 4.7 einen Frequenzbereich von 52 - 2 200 MHz unterstützt. Mit dem Tool *rtl_test* von RTLSDR wurde dies bestätigt (siehe Abb. 15).

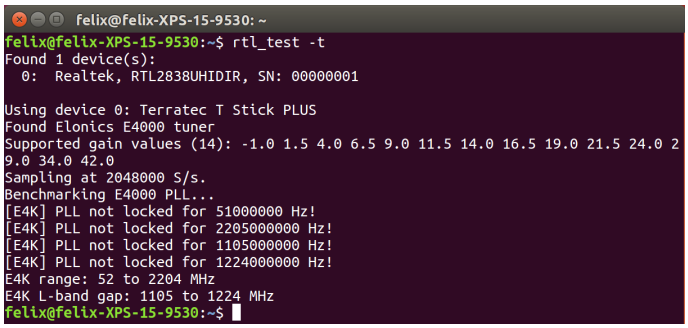


Abbildung 15: Test des Frequenzbereiches des DVB-T-Sticks

Zusätzlich musste getestet werden, ob die Samplingrate des Empfängers ausreichend hoch ist. DECT hat eine Symbolrate von 1 152 000 Symbolen/s. Durch das Nyquist-

Shannon-Theorem ergibt sich folgende Formel:

$$f_{abtast} > 2 \cdot f_S$$

Daraus lässt sich eine Abtastrate von 2 304 000 Samples/s errechnen, die der Empfänger mindestens unterstützen muss. Dies kann erneut mit *rtl_test* festgestellt werden. Dabei muss dem Programm beim Aufruf eine Samplingrate als Argument vorgegeben werden. Es testet dann wie viele Samples verloren gehen und gibt die Anzahl der verlorenen Samples pro Millionen Samples aus (siehe Abb. 16).

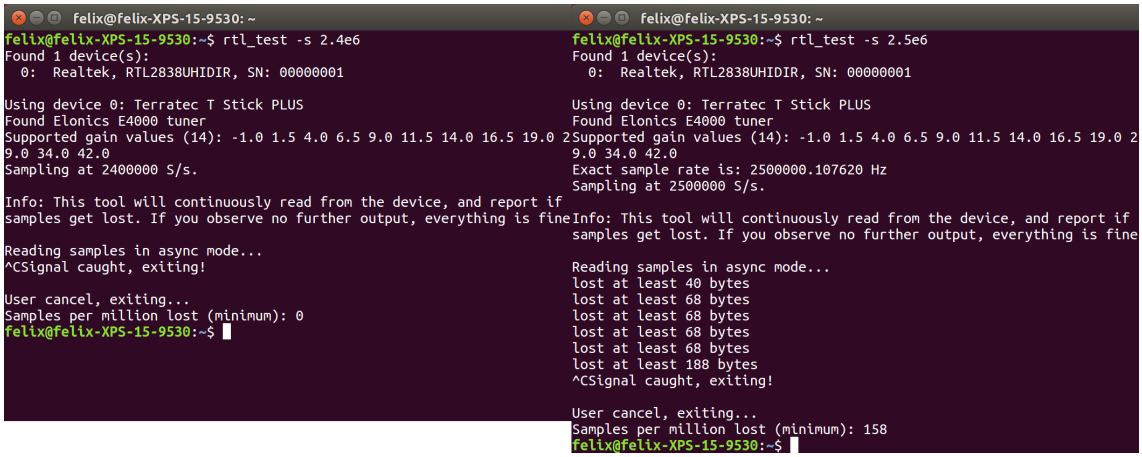


Abbildung 16: Test der Samplingrate des DVB-T-Sticks links: keine Fehler mit 2,4 MS/s, rechts: mit 2,5 MS/s treten Fehler auf

So wurde festgestellt, dass der Empfänger bis zu 2,4 Megasamples/s fehlerfrei abtasten kann. Er ist dadurch vollkommen für den Empfang von DECT geeignet. Um ihn mit der gr-dect2 Software zu nutzen, muss lediglich der vorhandene Source Block gegen den RTLSDR-Source Block getauscht und die eingestellte Samplingrate angepasst werden.

6.2.2 DECT-Telefone

Für einen vollständigen Test der Software sollte ein DECT-Gerät gefunden werden, das die Sprachdaten nicht verschlüsselt. Dafür wurden zwei Festnetz-Telefone und ein Babyphone mittels gr-dect2 getestet. Der Test lief für alle Geräte gleich ab: Die Testgeräte wurden eingeschaltet und danach wurde mit der Software nach dem *Beacon*-Signal des FP gesucht, indem alle Kanäle durchgeschaltet und die Ausgabekonsole beobachtet wurde. Nachdem ein interner Anruf gestartet wurde, tauchten auch die PPs in der Geräteliste auf. Nun konnte ein Gerät ausgewählt und Sprachdaten abgespielt werden, falls diese unverschlüsselt gesendet werden.

Dabei wurde festgestellt, dass das Festnetztelefon Siemens Gigaset A265, das 2007 auf den Markt kam, bereits die Sprachdaten verschlüsselt. Beim Sprechen in die Handteile war keine Sprache zu hören.
Da laut der Dokumentation von gr-dect2 Babyphones Sprachdaten nicht verschlüsseln, wurde als nächstes ein Babyphone der Marke Philips Avent SCD530 getestet. Dieses Gerät kam 2009 auf den Markt und auch hier war es nicht möglich, die Sprachdaten

wiederzugeben.
Schließlich war der Test mit dem Siemens Gigaset 2015 von 1998 erfolgreich. Da zu diesem Telefon jedoch nur ein Handteil verfügbar ist, wurde ein Handteil der Marke Telekom Sinus 44 mit der Basisstation verbunden, um interne Anrufe tätigen zu können. Die Sprache wird leicht verzögert, aber gut verständlich am Computer wiedergegeben.

6.3 Erweiterungen

Da nun eine funktionierende DECT-Implementation in GNU Radio als Grundlage, ein geeigneter Empfänger und Geräte, die Sprache unverschlüsselt senden, zur Verfügung stehen, kann die Software erweitert werden. Dazu wurde das komplette Modul in *gr-dectblocks* umbenannt und ein Projekt bei GitLab, einem Webservice zur Softwareversionsverwaltung mittels git, angelegt. Nachfolgend werden die einzelnen Erweiterungen erläutert.

6.3.1 Ausgabe des A-Teils

Zu Debug-Zwecken kann es sehr sinnvoll sein, den A-Teil zu analysieren. Deshalb soll dieser mit in der Geräteliste ausgegeben werden. Weiterhin soll er, falls er *static system information* enthält, ausgewertet und die enthaltenen Daten dargestellt werden. Um dies zu erreichen, muss nur der *Packet-Decoder* Block verändert werden. Um den A-Teil und die Systeminformationen zu speichern, müssen zunächst in der Headerdatei entsprechende Variablen angelegt werden:

```
1    uint8_t  a_field_data[8];
2
3    //static information
4    bool     staticinf_rcvd;
5    bool     nr_bit;         //normal-reverse bit
6    uint8_t  sn_bits;        //slot number bits
7    uint8_t  sp_bits;        //start position bits
8    bool     esc_bit;        //wheter QT escape is broadcast
9    uint8_t  nrtrans_bits;   //number of transceivers in RFP
10   bool     extinf_bit;     //Extended carrier information available
11   uint16_t carrier_bits;   //which carriers are available
12   uint8_t  carrier_number;//carrier number of transmission
13   uint8_t  nextcarrier_number;//carrier on which one receiver will
        be listening on the next frame
```

Listing 1: Variablen zur Analyse des A-Feldes

Die Methode *decode_afield()* wurde um folgende Schleife erweitert, um das A-Feld zu speichern:

```
1  for(int i=0; i<8; i++)
2    d_cur_part->a_field_data[i]=field_data[i];
```

Listing 2: Speichern des A-Feldes im Packet-Decoder

Das Extrahieren der *static system information* erfolgt auch in dieser Methode mittels Bitschiebeoperationen, allerdings in einem Zweig der Fallunterscheidung. Der Codeausschnitt dafür ist in Anhang B.2.3 (Zeilen 156 - 219) zu finden.

Schließlich wurde auch die *print_parts()*-Methode, die die verfügbaren Geräte im Textfeld ausgibt, um eine Schleife zur Ausgabe des A-Feldes und um Anweisungen zur Ausgabe der System-Informationen erweitert (siehe Anhang B.2.3 Zeilen 398 - 434).

Die Änderungen der Ausgabe in das Textfeld sind in Abbildung 17 zu sehen. Unter *Slot Pair* sind die Slots angegeben, die vom Gerät für das TDMA genutzt werden (siehe Abb. 8 in Abschnitt 5.2.3). *Nr of Transceivers* bezeichnet die Anzahl der am FP aktuell registrierten Geräte. Auf welchen Kanälen das Gerät senden und empfangen kann, ist unter *Supported RF carriers* zu sehen. Diese Information ist wichtig, da nicht jedes Gerät auf allen Kanälen kommunizieren kann. *Current RF carrier* gibt den aktuellen und *Next RF carrier* den nächsten Kanal an. Weiterhin ist zu sehen, dass das Gerät zuerst ohne Systeminformationen aufgelistet wurde. Nachdem die Informationen empfangen und dekodiert werden, wird die Liste aktualisiert.

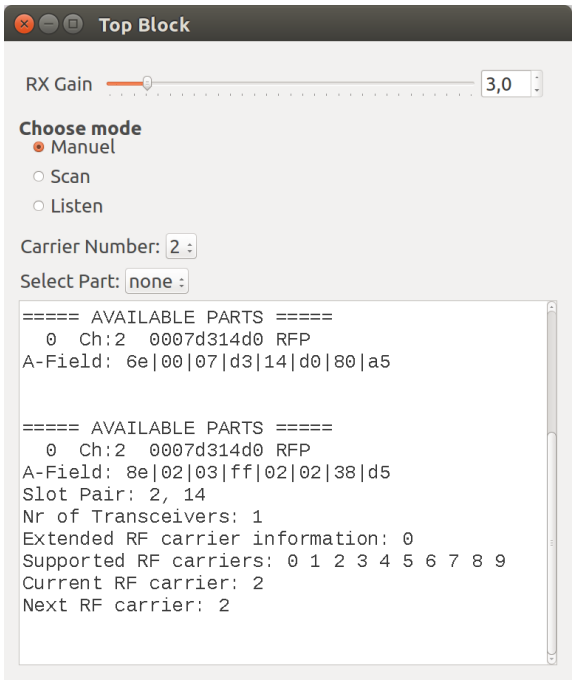


Abbildung 17: Statusausgaben nach Erweiterung

6.3.2 Automatischer Frequenz-Scan

Mit dieser Erweiterung soll das Programm in der Lage sein, selbstständig alle DECT-Kanäle nach Geräten zu durchsuchen. Dieser Modus soll per Radio-Buttons auf dem GUI ausgewählt werden können.

Die Erweiterung stellt eine ganz neue Anforderung an das Programm: ein Block selbst muss in der Lage sein, einen Parameter eines anderen Blockes zu ändern. Bisher wurde der Kanal per Dropdown-Menü vom Nutzer ausgewählt und somit

direkt der Frequenz-Parameter des Source-Blockes manipuliert. Nun soll jedoch der *Packet-Receiver*-Block diesen Parameter ändern können.

Dafür muss der *Packet-Receiver*-Block zunächst einen Parameter erhalten. Über diesen erhält der Block die Information, welcher Modus eingestellt wurde. Wird nun ein Radio-Button angeklickt, startet im Hintergrund eine Callback-Funktion, die den Wert an den Block weiter gibt. Da dieser Block bisher noch keine Parameter hat, muss die Callback-Funktion erst angelegt werden. Dazu muss diese zuerst in der Blockbeschreibungsdatei *dectblocks_packet_receiver.xml* eingetragen werden:

```
1 <callback>select_rx_mode($rx_mode)</callback>
2 <param>
3   <name>RX Mode</name>
4   <key>rx_mode</key>
5   <value>0</value>
6   <type>int</type>
7 </param>
```

Listing 3: Definition eines Block-Parameters

In den Header-Dateien *packet_receiver.h* und *packet_receiver_impl.h* muss die Funktion ebenfalls definiert werden:

```
1 virtual void select_rx_mode(uint32_t rx_mode);
```

Die Implementation der Funktion ist sehr simpel. Der Modus wird in eine Variable gespeichert. Weiterhin wird ein Flag gesetzt, das die Änderung dieser Variable anzeigt.

```
1 void packet_receiver_impl::select_rx_mode(uint32_t rx_mode)
2 {
3   if(d_selected_rx_mode != rx_mode)
4   {
5     d_selected_rx_mode_changed=true;
6     d_selected_rx_mode = rx_mode;
7   }
8 }
```

Listing 4: Callback-Funktion für Block-Parameter

Es werden zwei neue Nachrichtentypen benötigt, die über den Message Port zwischen *Receiver*- und *Decoder*-Block gesendet werden können. Bisher gibt es nur *lost_part*. Die beiden neuen Nachrichten sind *change_mode* zur Änderung des Modus und *change_freq* zum Ändern des Kanals. Diese müssen dem *Message-Event-Handler* des *Package-Decoders* hinzugefügt werden:

```
1 void packet_decoder_impl::msg_event_handler(pmt::pmt_t msg)
2 {
3   if(pmt::dict_has_key( msg, pmt::mp("rcvr_msg_id")))
4   {
5     pmt::pmt_t msg_id = pmt::dict_ref( msg, pmt::mp("rcvr_msg_id"),
          pmt::PMT_NIL);
6     if(pmt::eq(msg_id, pmt::mp("lost_part"))){
7       //Remove Part from List
```

```
8         (...)}
9       else if(pmt::eq(msg_id, pmt::mp("change_freq"))){
10        //Change Frequency
11        (...)}
12      else if(pmt::eq(msg_id, pmt::mp("change_mode"))){
13        //Change Mode
14        (...)}
15    }
16  }
```

Listing 5: Message-Event-Handler eines Blockes mit Message-Port

Die Nachrichten werden an den entsprechenden Stellen gesendet. Damit die Befehle im *Packet-Decoder*-Block komplett verarbeitet werden, wird außerdem eine leere Nachricht auf den Datenausgang geschickt:

```
1  if(d_selected_rx_mode_changed)
2  {
3    d_selected_rx_mode_changed=false;
4    (...)
5    //send message to change mode
6    pmt::pmt_t msg = pmt::make_dict();
7    msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt::mp("
         change_mode"));
8    msg = pmt::dict_add(msg, pmt::mp("rx_mode"), pmt::mp((uint64_t)
         d_selected_rx_mode));
9    message_port_pub(pmt::mp("rcvr_msg_out"), msg);
10
11   //send data with void tag to refresh next block
12   *out++ = (char)0;
13   add_item_tag(0, nitems_written(0) + oo, pmt::mp("packet_len"),
         pmt::mp(1));
14   add_item_tag(0, nitems_written(0) + oo, pmt::mp("packet_type"),
         pmt::mp("void"));
15   oo++;
16   noutput_items++;
17 }
```

Listing 6: Senden einer Nachricht über den Message-Port

Wenn der *Packet-Decoder*-Block nun eine *change_freq*-Nachricht bekommt, muss er mit dem Source-Block kommunizieren. Das funktioniert mittels *Signal Probe* und *Probe Function*. *Signal Probe* wandelt einen Datenstrom in eine Variable um. Durch einen Aufruf der *level()*-Funktion des Blockes, wird der Wert abgefragt. Das erfolgt durch den *Probe-Function*-Block. Dieser ruft eine Funktion eines anderen Blockes in einem einstellbaren Intervall auf, und stellt den Rückgabewert in einer GNU Radio Variable bereit. Diese kann nun anderen Blöcken als dynamischer Parameter übergeben werden [24].

Um diese Funktionalität zu nutzen, muss zunächst ein neuer Ausgang am *Packet-Decoder*-Block angelegt werden (siehe Lst. 7).

```
1 <source>
2   <name>out_freq</name>
```

```
3     <type>int</type>
4     <optional>1</optional>
5   </source>
```

Listing 7: Definition eines Block-Ausgangs

Dieser Ausgang wird auch in den Konstruktor der *packet_decoder_impl.cc*-Datei übernommen:

```
1  packet_decoder_impl::packet_decoder_impl():
       gr::tagged_stream_block("packet_decoder",
       gr::io_signature::make(1, 1, sizeof(unsigned char)),
       gr::io_signature::make2(2, 2, sizeof(unsigned char), sizeof(
       uint32_t)), std::string("packet_len"))
2  {
3     (...)
4  }
```

Listing 8: Definition eines Block-Ausgangs in der Implementationsdatei

Als nächstes können in GNU Radio *Probe Signal* und *Function Probe* angelegt werden. Der *Signal-Probe*-Block wird auf den Variablentypen des neuen Ausgangs eingestellt und mit diesem verbunden. In den *Probe-Function*-Block wird die ID und der Name der *Signal Probe* eingegeben. Schließlich erhält der Source-Block den Variablennamen als Frequenz-Parameter (siehe Abb. 18).

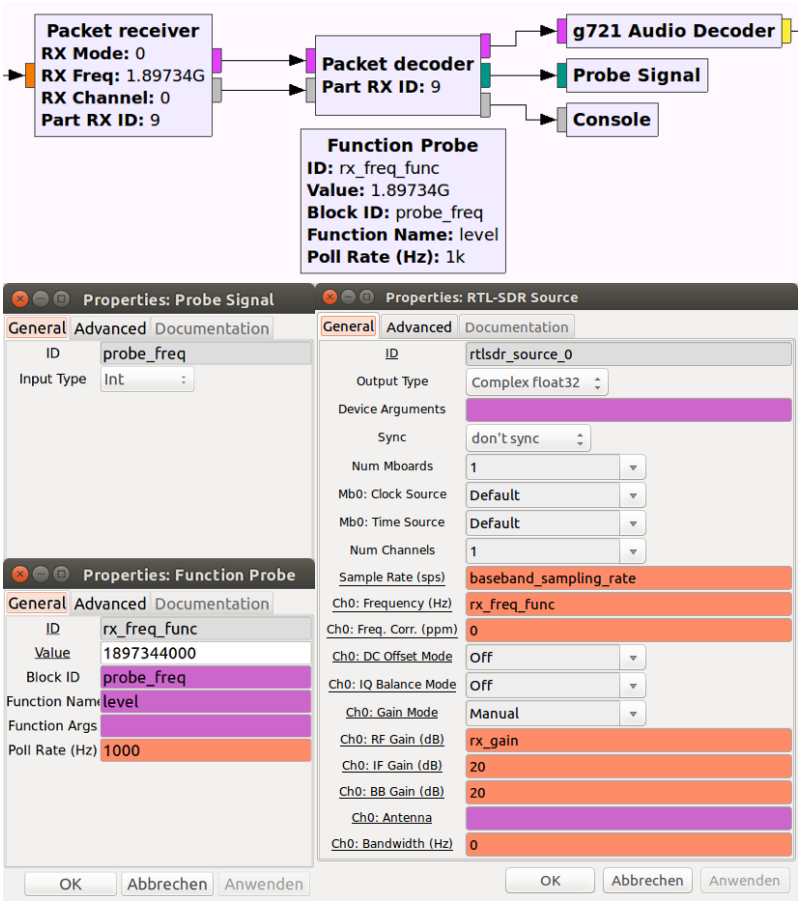


Abbildung 18: Probe Signal und Function Probe im Flowgraph mit Details

Nun kann im Block die gewünschte Frequenz auf den *out_freq*-Ausgang gelegt und in den Source-Block übernommen werden.

```
1 *out_freq++ = chan_to_freq[d_change_to_rx_channel];
```

6.3.3 Kanalverfolgung der Geräte

DECT-Geräte sind in der Lage während einer Datenübertragung den Slot im aktuellen Kanal oder sogar den ganzen Kanal zu wechseln. So kann auf plötzlich auftretende Störungen reagiert werden, ohne die Übertragung zu unterbrechen. Der DECT-Monitor soll erweitert werden, sodass er einen anstehenden Kanalwechsel erkennt und dem Gerät folgt. Diese Erweiterung wird aus später genannten Gründen nur theoretisch abgehandelt.

Um die gewünschte Funktionalität zu erreichen müssen zusätzlich zu der *system information*-Nachricht, die in Kapitel 6.3.1 hinzugefügt wurde, die *MAC layer control*-Nachrichten implementiert werden. Genauer sind nur die *quality control*-Nachrichten von Interesse (siehe Abb. 19).

0 0 1 1	command	param_1	param_2	0000 1111	0000 1111
a_8	a_{15}	a_{16} a_{23}	a_{24} a_{31}	a_{32}	a_{47}

Abbildung 19: Aufbau einer Quality Control Nachricht [16]

Der PP sendet eine Anfrage zum Wechsel des Kanals an den FP. Diese Anfrage enthält neben dem Kanal auch das Slot-Paar, das auf dem neuen Kanal benutzt werden wird. Der FP kann diese Anfrage nun ignorieren oder bestätigen. Als Bestätigung sendet der FP die gleiche Nachricht wieder an das PP zurück. Der genaue Aufbau des *frequency-replace*-Befehls ist in Abbildung 20 zu sehen.

command	param_1	param_2	Meaning
1000	0000 SN	0000 CN	frequency replacement to carrier CN on slot pair SN request PT -> FT confirm FT -> PT

Abbildung 20: Frequency-Replace-Befehl [17]

Diese Nachricht würde erneut in der Switch-Case-Anweisung der *decode_afield()*-Methode im *Package-Decoder* analysiert werden. Jedoch muss unterschieden werden, ob die Nachricht von einem PP oder FP kommt. Es reicht aus, den *frequency-replace*-Befehl vom FP zu betrachten, da dieser den Wechsel bestätigt und erneut Kanal und Slot-Paar enthält. Wird dieser Befehl von einem FP empfangen, kann sofort auf den neuen Kanal gewechselt werden. Dazu wird die entsprechende Frequenz auf den *out_freq*-Ausgang gelegt.

6.3.4 Probleme

Leider traten mit fortschreitender Arbeit an der Software des DECT-Monitors immer mehr Probleme auf: Zunehmend wurden Nachrichten auf dem Message Port von

[16] Quelle: [17, Seite 141]
[17] Quelle: [17, Seite 141]

Packet-Receiver zu *Packet-Decoder* nicht registriert. Dies führte dazu, dass Beispielsweise die Gerätelisten der beiden Blöcke nicht synchron gehalten wurden. Weiterhin sorgten unerklärliche Abstürze der Blöcke für ein erschwertes Arbeiten.

Die Umsetzung der Kanalverfolgung hätte für noch mehr Fehler gesorgt, da diese die Aufgabentrennung der Blöcke verletzt. Der *Packet-Receiver*-Block soll nur Daten empfangen und diese weitergeben, ohne sie zu analysieren. Der *Packet-Receiver*-Block würde mit der Erweiterung Frequenzwechsel veranlassen, von denen *Receiver*-Block nichts weiß. Somit können die Gerätelisten erneut nicht synchron bleiben.

7 Praktikumsversuch

In diesem Kapitel wird die Ausarbeitung eines Praktikumsversuchs zum Thema SDR und GNU Radio beschrieben. Dabei werden Umfang und Bearbeitungsmöglichkeiten auf Basis der in dieser Arbeit gemachten Erfahrungen diskutiert.

7.1 Entwurf

Das Praktikum soll dem Studenten einen Einblick in das Thema SDR und den Umgang mit GNU Radio geben. Speziell sollen die Nutzung des GRC, sowie das Modifizieren von Signalverarbeitungsblöcken, erklärt werden. Im Rahmen des Praktikums sollen verschiedene Aufgaben am PC gelöst werden.

Der Praktikumsversuch soll eine Bearbeitungszeit von 90 Minuten haben, dementsprechend ist der Aufgabenumfang zu wählen. Folgendes soll dem Studenten vermittelt werden:

- Funktionsweise, Vorteile und Nachteile von SDR
- Grundlegende Funktionsweise von GNU Radio
- Erstellung und Ausführung eines Flussgraphen mittels GRC
- Anpassen eines vorhandenen Signalverarbeitungsblocks am Beispiel von DECT

Die beste Voraussetzung für das Praktikum wäre es, wenn jeder Student einen eigenen Empfänger zur Verfügung hätte. Als Empfänger könnte, wie in dieser Arbeit, ein DVB-T-Stick dienen. Als Sender könnte für eine erste Demonstration der GNU Radio Plattform ein Radiosender genutzt werden. Für einen Flussgraphen, der DECT empfängt, könnte ein Festnetztelefon im Raum platziert werden. Dieses würden alle Studenten empfangen. Leider kann nicht vorausgesetzt werden, dass für jeden Studenten ein Empfänger bereitgestellt wird. Auch wenn eine Gruppenarbeit diese Situation entschärfen würde, sollen in diesem Praktikum Signal-Mitschnitte, in Dateien bereitgestellt, als Signalquellen dienen. Statt des RTLSDR-Source-Blocks wird ein File-Source-Block, in dem eine Datei geladen wird, verwendet. Somit kann größtenteils auf Hardware verzichtet werden.

Es kann angenommen werden, dass jedem Studenten ein Labor-PC zur Bearbeitung der Aufgaben zu Verfügung steht. Aufgrund der fehlenden Root-Rechte der Studenten muss GNU Radio bereits vorinstalliert sein. Weiterhin wird vorausgesetzt, dass der Student durch die Vorlesung *Programmierung II*, aus dem zweiten Semester, Programmierkenntnisse in C++ besitzt.

Unter den gegebenen Voraussetzungen ergeben sich folgende Aufgabenbereiche:

1. Einstiegsaufgabe zu SDR
2. Kennenlernen der GNU Radio Plattform
3. Starten und Untersuchen des GRC
4. Erstellung eines Flussgraphen zum Empfang von FM-Radio mittels GRC
5. Öffnen und Erweitern des Flussgraphen zum Empfang von DECT

7.2 Aufgaben

Die Aufgaben wurden mit den in Kapitel 7.1 erfassten Voraussetzungen erarbeitet. Alle dieser Aufgaben sind ohne zusätzliche Hardware lösbar. Nach dem Bearbeiten der Aufgaben soll der Student grundlegende Kenntnisse zu SDR, der Arbeit mit GNU Radio und der Erweiterung von vorhandenen Signalblöcken erworben haben. Die fertigen Aufgabenstellungen und Lösungen sind im Anhang A zu finden.

Einstiegsaufgabe

Die Einstiegsaufgabe soll grundlegende Fragen zu SDR klären. Dazu soll der Student Informationen aus dem Vorlesungsskript und aus Internetquellen zusammentragen. Ein paar Quellen werden als Link zu Verfügung gestellt, für weitere Informationen kann eine Suchmaschine genutzt werden. Dabei sollen folgende Fragen beantwortet werden:

- Was unterscheidet ein SDR von anderen Funkgeräten?
- Aus welchen Bauteilen besteht dieses Idealerweise?
- Welche Probleme lassen sich mit einem SDR lösen?
- Welche Nachteile hat es?

Kennenlernen der GNU Radio Plattform

Der Student soll sich selbstständig über GNU Radio und dessen Funktionsweise informieren. Dazu werden Internetquellen bereitgestellt, weitere Informationen sind erneut per Suchmaschine zusammenzutragen. Folgende Fragen sind zu beantworten:

- Wie ist ein GNU Radio Block aufgebaut und welche Eigenschaften hat er?
- Welche Möglichkeiten zum Datenaustausch zwischen Blöcken gibt es?
- Wie kann der Nutzer mit Blöcken interagieren?

Erstellen eines Radio-Empfängers mittels GRC

Der GRC soll gestartet werden. Der Student soll sich einen groben Überblick über die Funktionen der Software machen. Danach wird, mit einem Screenshot als Vorlage, ein Flowgraph zum Empfang von FM-Radio erstellt. Da kein tatsächlicher Empfänger zur Verfügung steht, wird eine der bereitgestellten Dateien geladen. Über Lautsprecher oder Kopfhörer sollte nun Radio zu hören sein.

- Bewerten Sie die Qualität!
- Erläutern Sie grob die Funktion der einzelnen Blöcke!

Erweitern eines GNU Radio Blocks am Beispiel von DECT

Die in einem Verzeichnis bereitgestellten Blöcke sollen installiert und der enthaltene Flowgraph geöffnet werden. Das Programm soll ausgeführt und getestet werden. Da kein echter Empfänger genutzt wird, ist es nicht möglich den Kanal interaktiv zu wählen. Eine Datei mit Beacon-Signal wird bereitgestellt. Nach dem Lesen eines Einleitungstextes zu DECT sind folgende Aufgaben sind zu lösen:

- Im GUI werden verschiedene Daten angezeigt. Erklären Sie die Bedeutung dieser!
- Erweitern Sie den *Packet-Decoder*-Block dahingehend, dass das A-Feld als Hexdaten angezeigt wird!
- Erweitern Sie den Block erneut, und werten Sie die *Static-System-Info* aus! Welche Bedeutung haben die enthaltenen Daten?

8 Ergebnisse und Ausblick

8.1 Ergebnisse

Das Ziel dieser Bachelorarbeit war es, Software Defined Radio (SDR) anhand von Digital Enhanced Cordless Telecommunications (DECT) zu demonstrieren.

Dazu wurde die GNU-Radio-Plattform unter Ubuntu eingerichtet. Als Empfänger kam ein Digital Video Broadcasting – Terrestrial (DVB-T)-Stick mit RTL2832U-Chip zum Einsatz. Durch den veränderten Treiber *RTLSDR* konnte dieser in GNU Radio eingebunden werden.

Es wurde sich mit dem sehr umfangreichen DECT-Standard auseinandergesetzt. Der USB-Empfänger wurde erfolgreich auf die Eignung für diese Aufgabe geprüft. Als Arbeitsgrundlage in GNU Radio kam die DECT-Implementation *gr-dect2* von Pavel Yazev zum Einsatz. Diese wurde zunächst angepasst, sodass sie mit *RTLSDR* funktioniert. Durch diese Anpassung konnten nun verschiedene Geräte auf unverschlüsselte Sprachübertragung getestet werden. Schließlich wurde festgestellt, dass das *Siemens Gigaset 2015* unverschlüsselt sendet. Dieses Gerät wurde fortan für die Weiterentwicklung und den Test des Empfängers genutzt.

Die erste Erweiterung des Empfängers ermöglichte eine genauere Auswertung des A-Feldes einer DECT-Nachricht. So konnten Verbindungsparameter wie der verwendete Kanal und das Slot-Paar ausgelesen werden. Für diese Erweiterung musste lediglich der *Packet-Decoder*-Block verändert werden. Als nächste Erweiterung sollte ein automatischer Frequenzscan implementiert werden. Dafür waren zusätzliche Kommunikationsstrukturen und Parameter anzulegen. Der *Packet-Receiver*- und der *Packet-Decoder*-Block mussten wesentlich verändert werden, um die gewünschte Funktionalität zu erhalten. Dies funktionierte relativ gut, allerdings traten erste Probleme auf. Die Nachrichten auf dem *Message-Port* wurden teilweise nicht mehr vom *Packet-Decoder* registriert. So kam es dazu, dass die Gerätelisten in der beiden Blöcke nicht mehr synchron waren. Die Verfolgung eines Gerätes bei automatischem Kanalwechsel war als dritte Erweiterung geplant, konnte jedoch aufgrund der Probleme der Software nicht umgesetzt werden. Die Vorgehensweise wurde theoretisch beschrieben.

Es wurde ein Praktikumsversuch ausgearbeitet, der den Studenten einen Einblick in das Thema SDR und GNU Radio geben soll.

8.2 Schlussbemerkung und Ausblick

Folgende Verbesserungen wären in zukünftigen Versionen des Empfängers:

- Komplette Umstrukturierung der Geräteliste: Da *Packet-Receiver* und *Packet-Decoder* zwei eigene Gerätelisten führen, die synchron gehalten werden, traten Probleme auf. Diese könnte durch die Kombination der beiden Blöcke in einen Block beseitigt werden.
- Ausbau der Protokollunterstützung im Empfänger: DECT ist ein sehr großer und komplexer Standard. Der Empfänger war selbst am Ende der Arbeit nur in der Lage, die wichtigsten Befehle des kompletten Standards zu unterstützen.

Die Zukunft von SDR bleibt spannend, es wird jedoch noch einige Zeit dauern, bis diese Technik ihren Weg zum privaten Endnutzer findet. In der Forschung und beim Militär findet sie bereits Einsatz. So erhielt z.B. vor kurzem der Münchner Elektronikkonzern Rohde & Schwarz den Auftrag, 50 Führungsfahrzeuge der Bundeswehr mit *Streitkräftegemeinsamer Verbundfähiger Funkgeräteausstattung*, basieren auf SDR, auszurüsten [25].

9 Quellenverzeichnis

[1] Elettra Venosa, Fredric J. Harris und Francesco A. N. Palmieri. *Software Radio: Sampling Rate Selection, Design and Synchronization*. Springer New York, 2012. ISBN: 978-1-4614-0113-1.

[2] Eugene Grayver. *Implementing Software Defined Radio*. Springer New York, 2013. ISBN: 978-1-4419-9332-8.

[3] *What is Software Defined Radio?* http://www.wirelessinnovation.org/index.php?option=com_content&view=article&id=63:Introduction_to_SDR&catid=19:site-content&Itemid=77
Zugriff: 27.06.2017.

[4] *Abtasttheorem*. https://www.mikrocontroller.net/articles/Abtasttheorem
Zugriff: 28.06.2017.

[5] *What is GNU Radio?* https://wiki.gnuradio.org/index.php/What_is_GNU_Radio%3F
Zugriff: 30.06.2017.

[6] *TutorialsCoreConcepts*. https://wiki.gnuradio.org/index.php/TutorialsCoreConcepts
Zugriff: 30.06.2017.

[7] *BlocksCodingGuide*. https://wiki.gnuradio.org/index.php/BlocksCodingGuide
Zugriff: 30.06.2017.

[8] *Guided Tutorial Programming Topics*. https://wiki.gnuradio.org/index.php/Guided_Tutorial_Programming_Topics
Zugriff: 30.06.2017.

[9] *Open Source Mobile Communications*. https://osmocom.org/
Zugriff: 30.06.2017.

[10] *Welcome to OsmoSDR*. http://osmocom.org/projects/osmosdr/wiki
Zugriff: 30.06.2017.

[11] *rtl-sdr*. https://osmocom.org/projects/sdr/wiki/rtl-sdr
Zugriff: 30.06.2017.

[12] *DECT - Digital Enhanced Cordless Telecommunications*. https://www.elektronik-kompendium.de/sites/kom/0505231.htm
Zugriff: 01.07.2017.

[13] *The DECT Standard*. http://einstein.informatik.uni-oldenburg.de/rechnernetze/seite24.htm
Zugriff: 01.07.2017.

[14] *DMAP - DECT Multimedia Access Profile*. https://www.elektronik-kompendium.de/sites/kom/0909201.htm
Zugriff: 01.07.2017.

[15] *DECT*. http://www.etsi.org/technologies-clusters/technologies/dect
Zugriff: 01.07.2017.

[16] *Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 1: Overview*. Standard. http://www.etsi.org/deliver/etsi_

en/300100_300199/30017501/02.06.01_60/en_30017501v020601p.pdf. European Telecommunications Standards Institute, Juli 2015.

[17] *Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 3: Medium Access Control (MAC) layer*. Standard. http://www.etsi.org/deliver/etsi_en/300100_300199/30017503/02.06.01_60/en_30017503v020601p.pdf. European Telecommunications Standards Institute, Juli 2015.

[18] *TDMA (time division multiple access)*. http://www.itwissen.info/TDMA-time-division-multiple-access-TDMA-Verfahren.html
Zugriff: 04.07.2017.

[19] *Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 2: Physical Layer (PHL)*. Standard. http://www.etsi.org/deliver/etsi_en/300100_300199/30017502/02.06.01_60/en_30017502v020601p.pdf. European Telecommunications Standards Institute, Juli 2015.

[20] *Welcome to deDECTed.org Project*. https://dedected.org/trac
Zugriff: 05.07.2017.

[21] *CRE102 Der DECT Hack*. https://cre.fm/cre102-der-dect-hack
Zugriff: 05.07.2017.

[22] *26C3: Verschlüsselungscode für DECT-Funktelefone geknackt*. https://www.heise.de/security/meldung/26C3-Verschluesselungscode-fuer-DECT-Funktelefone-geknackt-893693.html
Zugriff: 05.07.2017.

[23] *DECT Forum Announces DECT and CAT-iq Security Certification*. https://www.dect.org/userfiles/file/Press%20releases/DF_Press%20Information_Security%20Certification_12232009.pdf
Zugriff: 23.07.2017.

[24] *Developing interactive projects with GNU Radio Companion – Part 1*. https://www.nutaq.com/blog/developing-interactive-projects-gnu-radio-companion-%E2%80%93-part-1
Zugriff: 07.07.2017.

[25] *Taktische Kommunikation mit SVFuA: Rohde & Schwarz stattet Bundeswehr mit modernen softwarebasierten Funksystemen aus*. http://www.presseportal.de/pm/75527/3682408
Zugriff: 23.07.2017.

Anhang

A Praktikum

A.1 Aufgaben

Dieses Praktikum stellt eine Einführung zum Thema Software Defined Radio (SDR) dar. Als Plattform kommt GNU Radio zum Einsatz.

Ziele:

- Kennenlernen von Funktionsweise, Vorteilen und Nachteilen von SDR
- Nutzung von GNU Radio als Plattform für SDR
- Erstellen von Flussgraphen mittels GNU Radio Companion (GRC)
- Erweitern von vorhandenen GNU Radio Blöcken

Hinweise:

- Alle benötigten Dateien finden Sie im Zip-Archiv *Prakt_SDR.zip* unter ... . Entpacken Sie dieses in Ihr Home-Verzeichnis.
- Im Ordner *sources* finden Sie verschiedene Signalmitschnitte. Diese werden in den Aufgaben 3 und 4 benötigt.

Aufgabe 1 - Grundlagen zu SDR

Nutzen Sie das Vorlesungsskript sowie die angegebenen Quellen, um sich über das Thema SDR zu informieren. Folgende Fragen sollen beantwortet werden:

- Was unterscheidet ein SDR von anderen Funkgeräten?
- Aus welchen Bauteilen besteht dieses Idealerweise?
- Welche Probleme lassen sich mit einem SDR lösen?
- Welche Nachteile hat es?

Weblinks:
`http://www.etsi.org/website/document/Workshop/SoftwareDefinedRadio/SDRworkshop1-7Chadwick-Zarlink.pdf`
`http://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio.pdf`
`http://www.itwissen.info/SDR-software-defined-radio-SDR-Radio.html`

Aufgabe 2 - Kennenlernen der GNU Radio Plattform

Lesen Sie die gegebenen Quellen grob, machen Sie sich mit den Begriffen vertraut und beantworten Sie folgende Fragen:

- Wie ist ein GNU Radio Block aufgebaut und welche Eigenschaften hat er?
- Welche Möglichkeiten zum Datenaustausch zwischen Blöcken gibt es?
- Wie kann der Nutzer mit Blöcken interagieren?

Weblinks:
`https://wiki.gnuradio.org/index.php/Main_Page`

https://wiki.gnuradio.org/index.php/TutorialsCoreConcepts
https://wiki.gnuradio.org/index.php/BlocksCodingGuide

Aufgabe 3 - Erstellen eines Radio-Empfängers mittels GRC

Öffnen Sie das Terminal und starten Sie GRC mit dem Befehl *gnuradio-companion*. Dieser dient dazu, verschiedenste Signalverarbeitungsblöcke miteinander zu kombinieren. Machen Sie sich mit der Bedienung der Software vertraut. Auf der rechten Seite sind, in verschiedene Kategorien eingeteilt, alle installierten GNU Radio Blöcke zu sehen. Diese können per Drag and Drop auf die freie Fläche gezogen werden. Mit jeweils einem Klick auf den Port zweier verschiedener Blöcke, können diese verbunden werden.

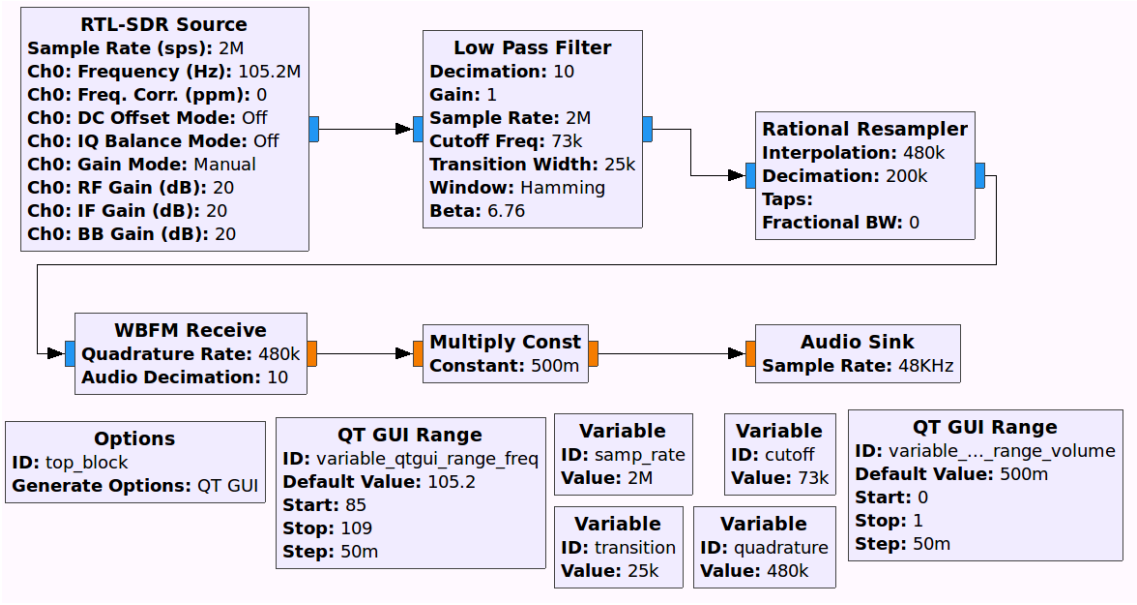


Abbildung 21: Flowgraph eines FM-Empfängers im GRC

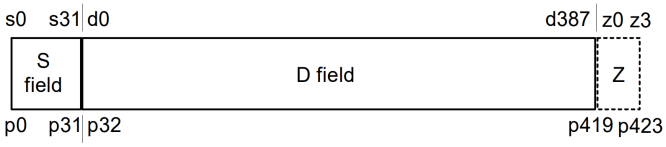
Bauen Sie den im Screenshot zu sehenden Flowgraph nach. Da in diesem Praktikum ohne Hardware gearbeitet wird, muss eine Datei als Signalquelle genutzt werden. Dadurch ist es nicht möglich, die Frequenz dynamisch einzustellen. Im Ordner *sources* sind verschiedene Radiosender als aufgezeichnete Daten verfügbar. Laden Sie eine dieser Dateien in den *File Source*-Block. Nutzen Sie Lautsprecher oder Kopfhörer, um sich das Ergebnis anzuhören.

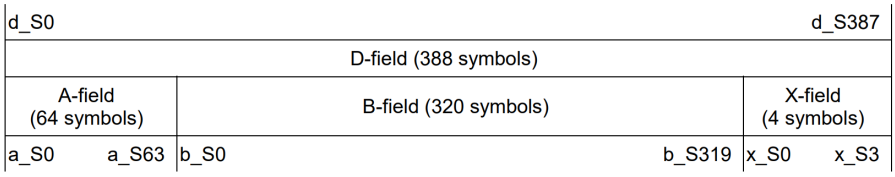
- Bewerten Sie die Qualität!
- Erläutern Sie grob die Funktion der einzelnen Blöcke!

Aufgabe 4 - Erweitern eines GNU Radio Blocks am Beispiel von DECT

Digital Enhanced Cordless Telecommunications (DECT) ist ein internationaler Standard für Schnurlostelefone. Der Standard nutzt Time Division Multiple Access (TDMA), um die Kommunikation zwischen mehreren Geräten zu ermöglichen. Jeder der bis zu 10 Kanäle wird in 24 Zeitslots eingeteilt: 12 für das Senden und 12

für das Empfangen von Daten. In einem Slot wird üblicherweise ein Paket gesendet, dessen Aufbau nachfolgend zu sehen ist.





Öffnen Sie den bereitgestellten Ordner *gr-dectblocks* im Terminal. Durch Eingabe folgender Befehle installieren Sie die Blöcke:

```
1 mkdir build
2 cd build
3 cmake ../
4 make
5 sudo make install
6 sudo ldconfig
```

Nun sind die Blöcke unter dem Menü *Dectblocks* im GRC verfügbar. Laden Sie den Flowgraph *dectblocks* aus dem Unterverzeichnis *grc*. Laden die Datei *dect_beacon* aus dem Verzeichnis *sources* in den File-Source-Block und starten Sie den Flowgraph. Was sehen Sie?

Laden Sie nun die Datei *dect_call* in den File-Source-Block. In diese Datei wurde ein interner Anruf, dass heißt eine Verbindung zwischen den beiden Handteilen der DECT-Station, aufgezeichnet. Wählen Sie im Dropdown-Menü *Select Part* eines der unter *Available Parts* aufgelisteten Geräte. Der aufgezeichnete Ton sollte nun zu hören sein.

Der *Packet-Receiver*-Block empfängt Daten und leitet das D-Feld an den *Packet-Decoder*-Block weiter. Dieser analysiert den A-Teil und behandelt die Daten im B-Feld entsprechend.

Erweitern Sie den Decoder-Block dahingehend, dass das A-Feld zu jedem Gerät hexadezimal ausgegeben wird. Dazu sind Änderungen in den Dateien *packet_decoder_impl.cc* und *packet_decoder_impl.h* nötig. Diese sind im Ordner *gr-dectblocks/lib/* zu finden. Nachdem die Dateien verändert wurden, müssen sie erneut installiert werden.

Tipp: Legen Sie zuerst in der Headerdatei für jedes Gerät eine neue Variable an. Diese muss groß genug sein, um das 64 bit lange A-Feld zu speichern. Die Methode *decode_afield()* bekommt das komplette A-Feld übergeben. Es empfiehlt sich es in dieser Methode zu speichern. Die Methode *print_parts()* ist für die Ausgabe der Daten in die Konsole zuständig.

Zusatzaufgabe

Erweitern Sie den *Packet-Decoder*-Block aus Aufgabe 4 erneut, sodass die Nachricht *Static-System-Information* ausgewertet wird. Diese gibt unter anderem Auskunft über das aktuell verwendete Zeitslot-Paar und die vom Gerät unterstützten DECT-Kanäle. Welche Informationen können noch ausgelesen werden?

Alle nötigen Informationen sind im Standard EN 300 175-3 in Kapitel 7.1 und 7.2.3 festgelegt.

`http://www.etsi.org/deliver/etsi_en/300100_300199/30017503/02.06.01_60/en_30017503v020601p.pdf`

A.2 Lösungen

Aufgabe 1 - Grundlagen zu SDR

- Es wird so wenig wie möglich Hardware verwendet. Die meisten Signalverarbeitungen finden in Software statt.
- Antenne, Analog-Digital-Umsetzer (ADU), Digital-Analog-Umsetzer (DAU)
- Anwendungsgebiet ist unabhängig von Hardware. Per Softwareupdate kann ein Gerät einen neuen Funkstandard nutzen oder sogar eine komplett andere Aufgabe übernehmen.
- Bisher aufgrund der vergleichsweise schlechten Energieeffizienz nicht für mobile Geräte geeignet.

Aufgabe 2 - Kennenlernen der GNU Radio Plattform

- Ports (Eingänge und Ausgänge), Parameter, Verhältnis von Eingangs- und Ausgangsdaten, work-Funktion
- Datenstrom mit verschiedenen Datentypen (complex, float, int, ...), Stream-Tags, Message Ports
- GUI-Elemente, die Werte über Parameter übergeben

Aufgabe 3 - Erstellen eines FM-Empfängers mittels GRC

- Im Ordner *Lösungen/A3/* ist der fertige Flowgraph zu finden
- Qualität ist relativ gut, da DVB-T Frequenzbereich nicht weit von FM-Radio-Bereich entfernt ist (vorallem Antenne ähnlich).

Aufgabe 4 - Erweitern eines GNU Radio Blocks am Beispiel von DECT

- Lösungen sind im Ordner *Lösungen/A4/* zu finden.
- *decode_afield()* und *print_parts()* wurden verändert.

Zusatzaufgabe

- Lösung im Ordner *Lösungen/Zusatz/*
- Erneut wurden *decode_afield()* und *print_parts()* verändert
- Informationen: Slot Paar, Anzahl der registrierten Geräte im (Radio) Fixed Part (FP), unterstützte Kanäle, aktueller Kanal, nächster Kanal

B Quellcode

B.1 Packet-Receiver

B.1.1 dectblocks_packet_receiver.xml

```
1  <?xml version="1.0"?>
2  <block>
3    <name>Packet receiver</name>
4    <key>dectblocks_packet_receiver</key>
5    <category>DECTBLOCKS</category>
6    <import>import dectblocks</import>
7    <make>dectblocks.packet_receiver()</make>
8    <callback>select_rx_mode($rx_mode, $rx_freq, $rx_chan, $rx_id)</
         callback>
9    <param>
10     <name>RX Mode</name>
11     <key>rx_mode</key>
12     <value>0</value>
13     <type>int</type>
14   </param>
15   <param>
16     <name>RX Freq</name>
17     <key>rx_freq</key>
18     <value>0</value>
19     <type>int</type>
20   </param>
21   <param>
22     <name>RX Channel</name>
23     <key>rx_chan</key>
24     <value>0</value>
25     <type>int</type>
26   </param>
27   <param>
28     <name>Part RX ID</name>
29     <key>rx_id</key>
30     <value>9</value>
31     <type>int</type>
32   </param>
33   <sink>
34     <name>in</name>
35     <type>float</type>
36   </sink>
37   <source>
38     <name>out</name>
39     <type>byte</type>
40   </source>
41   <source>
42     <name>rcvr_msg_out</name>
43     <type>message</type>
44     <optional>1</optional>
45   </source>
46 </block>
```

B.1.2 packet_receiver_impl.h

```
1  #ifndef INCLUDED_DECTBLOCKS_PACKET_RECEIVER_IMPL_H
2  #define INCLUDED_DECTBLOCKS_PACKET_RECEIVER_IMPL_H
3
4  #include <dectblocks/packet_receiver.h>
5
6  #define MAX_PARTS           8  // Maximum number of DECT parts
7  #define SMPL_BUF_LEN        (32 * 4)
8  #define TIME_TOL            10 // Time tolerance
9  #define INTER_SLOT_TIME    (480 * 4)
10 #define INTER_FRAME_TIME   (INTER_SLOT_TIME * 24)
11 #define S_FIELD_BITS        32
12 #define P32_D_FIELD_BITS    388
13 #define RFP_SYNC_FIELD      0xAAAAE98A
14
15 namespace gr {
16   namespace dectblocks {
17     class packet_receiver_impl : public packet_receiver
18     {
19       private:
20         typedef enum {_RFP_, _PP_} part_type;
21
22         part_type d_part_type;
23
24         uint32_t  d_rx_bits_buf[4];
25         // Buffer to save demodulated bits. Input signal has four
               samples per bits.
26         // We save bits related to null sample in null element,
               bits related to first sample in firts element
27         // and so on. Each element in this array should be
               considered as circular buffer.
28         unsigned  d_rx_bits_buf_index;
29         enum  {_WAIT_BEGIN_, _WAIT_END_, _POST_WAIT_} d_sync_state;
30
31         uint32_t  d_begin_pos;
32         uint32_t  d_end_pos;
33         uint32_t  d_post_wait_cnt;
34
35         float     d_smpl_buf[SMPL_BUF_LEN];
36         uint32_t  d_smpl_buf_index;
37
38         uint32_t  d_smpl_cnt;
39         uint32_t  d_out_bit_cnt;
40         uint64_t  d_inc_smpl_cnt; // Incomming samples counter
41
42         uint64_t  d_part_time[MAX_PARTS];
43         uint32_t  d_part_seq[MAX_PARTS];
44         uint64_t  d_part_channel[MAX_PARTS];
45         uint32_t  d_part_activity;
46
47         int32_t   d_cur_part_rx_id;
48
49         uint32_t d_selected_rx_mode; //selected mode (radio buttons
               )
50         bool     d_selected_rx_mode_changed;
```

```
51          uint32_t d_cur_rx_freq;   //rx frequency from probe
52          uint32_t d_selected_rx_chan; //selected chan from dropdown
53          bool     d_rx_chan_changed;
54          uint64_t d_last_freq_switch_time;
55          uint32_t d_channel_incr;
56          uint32_t d_part_rx_id;
57          bool     d_part_rx_id_changed;
58
59          int d_decimation;
60          int  decimation () const { return d_decimation; }
61          void set_decimation (int decimation)
62          {
63            d_decimation = decimation;
64            set_relative_rate (1.0 / decimation);
65          }
66
67          int fixed_rate_ninput_to_noutput(int ninput);
68          int fixed_rate_noutput_to_ninput(int noutput);
69
70          int check_part_activity(int time);
71          int register_part(int multi);
72          int find_best_smpl_point(void);
73
74        public:
75          packet_receiver_impl();
76          ~packet_receiver_impl();
77
78          virtual void select_rx_mode(uint32_t rx_mode, uint32_t
                rx_freq, uint32_t rx_chan, uint32_t rx_id);
79
80          // Where all the action really happens
81          void forecast (int noutput_items, gr_vector_int &
                ninput_items_required);
82
83          int general_work(int noutput_items,
84            gr_vector_int &ninput_items,
85            gr_vector_const_void_star &input_items,
86            gr_vector_void_star &output_items);
87      };
88    } // namespace dectblocks
89  } // namespace gr
90  #endif /* INCLUDED_DECTBLOCKS_PACKET_RECEIVER_IMPL_H */
```

B.1.3 packet_receiver_impl.cc

```
1  #ifdef HAVE_CONFIG_H
2  #include "config.h"
3  #endif
4
5  #include <gnuradio/io_signature.h>
6  #include "packet_receiver_impl.h"
7
8  namespace gr {
9    namespace dectblocks {
10     packet_receiver::sptr
11     packet_receiver::make()
12     {
13       return gnuradio::get_initial_sptr(new packet_receiver_impl())
           ;
14     }
15
16     packet_receiver_impl::packet_receiver_impl()
17       : gr::block("packet_receiver",
18         gr::io_signature::make(1, 1, sizeof(float)),
19         gr::io_signature::make(1, 1, sizeof(unsigned char)))
20     {
21       set_fixed_rate(true);
22       set_history(4);
23       set_decimation(4);
24
25       message_port_register_out(pmt::mp("rcvr_msg_out"));
26
27       d_rx_bits_buf_index = 0;
28       d_smpl_buf_index    = 0;
29       d_sync_state        = _WAIT_BEGIN_;
30       d_last_freq_switch_time = 0;
31       d_channel_incr = 0;
32
33       d_part_rx_id = 9;
34       d_part_rx_id_changed = false;
35       d_inc_smpl_cnt = 0;
36       d_part_activity = 0;
37       d_selected_rx_mode = 0;
38       d_selected_rx_mode_changed = false;
39       d_cur_rx_freq = 1897344000;
40       d_selected_rx_chan = 0;
41       d_rx_chan_changed = false;
42     }
43
44     packet_receiver_impl::~packet_receiver_impl()
45     {
46     }
47
48     void packet_receiver_impl::forecast (int noutput_items,
         gr_vector_int &ninput_items_required)
49     {
50       unsigned ninputs = ninput_items_required.size ();
51       for (unsigned i = 0; i < ninputs; i++)
```

```
52          ninput_items_required[i] = fixed_rate_noutput_to_ninput(
                noutput_items);
53      }
54
55      int packet_receiver_impl::fixed_rate_noutput_to_ninput(int
            noutput_items)
56      {
57        return noutput_items * decimation() + history() - 1;
58      }
59
60      int packet_receiver_impl::fixed_rate_ninput_to_noutput(int
            ninput_items)
61      {
62        return std::max(0, ninput_items - (int)history() + 1) /
              decimation();
63      }
64
65      // Check for parts activity
66      //        Return:
67      //         Part RX ID - if there is no activity for a part
68      //           -1       - otherwise
69      //
70      int packet_receiver_impl::check_part_activity(int time)
71      {
72        if(d_part_activity)
73        {
74          uint32_t j = 0;
75          uint32_t part_mask = 1;
76          while(part_mask <= d_part_activity)
77          {
78            if(d_part_activity & part_mask)
79            {
80              if(d_inc_smpl_cnt - d_part_time[j] > (time *
                    INTER_FRAME_TIME))
81              { // Release part
82                d_part_activity &= ~part_mask;
83                return j;
84              }
85            }
86            part_mask <<= 1;
87            j++;
88          }
89        }
90        return -1;
91      }
92
93      //  If there are several DECT parts on air we need to keep
            track each in correct way.
94      //  This function does this by taking into account time
            intervals (based on incomming sample counter)
95      //  between received bursts.
96      //  Return:
97      //    Part RX ID - if apropriate part is found or a new one
            assigned
98      //    -1 - otherwise
99      //
```

```
100     int packet_receiver_impl::register_part(int multi)
101     {
102       if(d_part_activity)
103       {
104         uint32_t j = 0;
105         uint32_t part_mask = 1;
106         uint32_t seq;
107
108         while(j < MAX_PARTS)     //update time
109         {
110           if(d_part_activity & part_mask)
111           {
112             uint64_t ltmp = (d_inc_smpl_cnt - d_part_time[j]) %
                  INTER_FRAME_TIME;
113             if(ltmp < TIME_TOL*multi)
114             {
115               seq = (d_inc_smpl_cnt - d_part_time[j])/
                      INTER_FRAME_TIME;
116               break;
117             }
118             else if(INTER_FRAME_TIME - ltmp <= TIME_TOL*multi)
119             {
120               seq = 1 + (d_inc_smpl_cnt - d_part_time[j])/
                      INTER_FRAME_TIME;
121               break;
122             }
123           }
124           part_mask <<= 1;
125           j++;
126         }
127
128         if(j < MAX_PARTS)
129         {
130           d_part_time[j] = d_inc_smpl_cnt;
131           d_part_seq[j]  = (d_part_seq[j] + seq) & 0x1F;
132           return j;
133         }
134         else
135         {
136           // Adding a new active part
137           j = 0;
138           part_mask = 1;
139           while(j < MAX_PARTS)
140             if(d_part_activity & part_mask)
141             {
142               part_mask <<= 1;
143               j++;
144             }
145             else
146             {
147               d_part_activity |= part_mask;
148               break;
149             }
150
151           if(j < MAX_PARTS)
152           {
```

```
153             d_part_time[j] = d_inc_smpl_cnt;
154             d_part_seq[j]  = 0;
155             if(d_cur_rx_freq==1897344000)
156             d_part_channel[j] = 0;
157             else if(d_cur_rx_freq==1895616000)
158             d_part_channel[j] = 1;
159             else if(d_cur_rx_freq==1893888000)
160             d_part_channel[j] = 2;
161             else if(d_cur_rx_freq==1892160000)
162             d_part_channel[j] = 3;
163             else if(d_cur_rx_freq==1890432000)
164             d_part_channel[j] = 4;
165             else if(d_cur_rx_freq==1888704000)
166             d_part_channel[j] = 5;
167             else if(d_cur_rx_freq==1886876000)
168             d_part_channel[j] = 6;
169             else if(d_cur_rx_freq==1885248000)
170             d_part_channel[j] = 7;
171             else if(d_cur_rx_freq==1883520000)
172             d_part_channel[j] = 8;
173             else if(d_cur_rx_freq==1881792000)
174             d_part_channel[j] = 9;
175
176             return j;
177           }
178           else
179             return -1;
180         }
181       }
182       else
183       {
184         // Adding the first active part
185         d_part_time[0]  = d_inc_smpl_cnt;
186         d_part_seq[0]   = 0;
187
188         if(d_cur_rx_freq==1897344000)
189         d_part_channel[0] = 0;
190         else if(d_cur_rx_freq==1895616000)
191         d_part_channel[0] = 1;
192         else if(d_cur_rx_freq==1893888000)
193         d_part_channel[0] = 2;
194         else if(d_cur_rx_freq==1892160000)
195         d_part_channel[0] = 3;
196         else if(d_cur_rx_freq==1890432000)
197         d_part_channel[0] = 4;
198         else if(d_cur_rx_freq==1888704000)
199         d_part_channel[0] = 5;
200         else if(d_cur_rx_freq==1886876000)
201         d_part_channel[0] = 6;
202         else if(d_cur_rx_freq==1885248000)
203         d_part_channel[0] = 7;
204         else if(d_cur_rx_freq==1883520000)
205         d_part_channel[0] = 8;
206         else if(d_cur_rx_freq==1881792000)
207         d_part_channel[0] = 9;
208
```

```
209         d_part_activity = 1;
210         return 0;
211       }
212     }
213
214     int packet_receiver_impl::find_best_smpl_point(void)
215     {
216       if(d_begin_pos != d_end_pos)    // If (d_begin_pos ==
              d_end_pos) we have the only optimal sample point
217       {
218         float max_val   = 0.0;
219         uint32_t max_index = d_begin_pos;
220         while(1)
221         {
222           uint32_t index = d_begin_pos;
223
224           float acc = 0.0;
225           for(uint32_t j = 0; j < 32; j++)
226           {
227             acc += fabs(d_smpl_buf[index]);
228             index = (index - 4) & (SMPL_BUF_LEN - 1);
229           }
230
231           if(acc > max_val)
232           {
233             max_val   = acc;
234             max_index = d_begin_pos;
235           }
236
237           if(d_begin_pos == d_end_pos)
238             break;
239
240           d_begin_pos = (d_begin_pos + 1) & (SMPL_BUF_LEN - 1);
241         }
242         return (d_end_pos - max_index) & (SMPL_BUF_LEN - 1);
243       }
244       return 0; // No Correction
245     }
246
247     void packet_receiver_impl::select_rx_mode(uint32_t rx_mode,
          uint32_t rx_freq, uint32_t rx_chan, uint32_t rx_id)
248     {
249       if(d_selected_rx_mode != rx_mode)
250         d_selected_rx_mode_changed=true;
251       d_selected_rx_mode = rx_mode;
252
253       d_cur_rx_freq = rx_freq;
254
255       if(d_selected_rx_chan!=rx_chan)
256         d_rx_chan_changed=true;
257       d_selected_rx_chan = rx_chan;
258
259       if(d_part_rx_id != rx_id && rx_id != 9)
260         d_part_rx_id_changed=true;
261       d_part_rx_id = rx_id;
262     }
```

```
263
264     int packet_receiver_impl::general_work (int noutput_items,
265       gr_vector_int &ninput_items,
266       gr_vector_const_void_star &input_items,
267       gr_vector_void_star &output_items)
268     {
269       const float *in = (const float *) input_items[0];
270       unsigned char *out = (unsigned char *) output_items[0];
271       unsigned ni = ninput_items[0] - history();
272       uint32_t sync_detected;
273
274       uint32_t ii = 0;
275       uint32_t oo = 0;
276
277       while(ii < ni && oo < noutput_items)
278       {
279         // Detect RX bit
280         uint32_t rx_bit = (*in >= 0)? 0:1;
281         d_rx_bits_buf[d_rx_bits_buf_index] = (d_rx_bits_buf[
              d_rx_bits_buf_index] << 1) | rx_bit;
282         d_smpl_buf[d_smpl_buf_index] = *in++;   // save samples in
              cyrcular buffer to search the best sample point later
283
284         switch(d_sync_state)
285         {
286           case _WAIT_BEGIN_:
287             // Perform SYNC detect.
288             // Because we have four samples per symbol there may be
                   several positions where SYNC can be detected.
289             // So we check interval and then look for the best
                  sample point.
290             //
291             sync_detected = ((d_rx_bits_buf[d_rx_bits_buf_index] ^
                  (uint32_t)RFP_SYNC_FIELD) == 0);
292
293             if(sync_detected)
294               d_part_type = _RFP_;
295             else
296             {
297               sync_detected = ((d_rx_bits_buf[d_rx_bits_buf_index]
                      ^ (~(uint32_t)RFP_SYNC_FIELD)) == 0);
298               if(sync_detected)
299               d_part_type = _PP_;
300             }
301
302             if(sync_detected)
303             {
304               d_begin_pos  = d_smpl_buf_index;
305               d_sync_state = _WAIT_END_;
306             }
307
308             break;
309           case _WAIT_END_:
310             if(d_part_type == _RFP_)
311               sync_detected = ((d_rx_bits_buf[d_rx_bits_buf_index]
                      ^ (uint32_t)RFP_SYNC_FIELD) == 0);
```

```
312             else if(d_part_type == _PP_)
313               sync_detected = ((d_rx_bits_buf[d_rx_bits_buf_index]
                      ^ (~(uint32_t)RFP_SYNC_FIELD)) == 0);
314
315             if(!sync_detected)
316             {
317               d_end_pos  = (d_smpl_buf_index - 1) & (SMPL_BUF_LEN -
                      1);
318
319               // Perform correction to the best sample position
320               d_smpl_cnt = (1 + find_best_smpl_point()) & 3;
321
322               if(d_selected_rx_mode == 0) //Manuell mode
323                 d_cur_part_rx_id = register_part(1);
324               else if(d_selected_rx_mode == 1)    //Scan
325                 d_cur_part_rx_id = register_part(50);
326
327               if(d_cur_part_rx_id < 0)
328               {
329                 d_sync_state = _WAIT_BEGIN_;
330                 break;
331               }
332
333               d_out_bit_cnt = 0;
334               d_sync_state    = _POST_WAIT_;
335             }
336             break;
337
338           case _POST_WAIT_:
339             // Receive packet payload
340             if(d_smpl_cnt == 0)
341             {
342               *out++ = (char)rx_bit;
343
344               if(d_out_bit_cnt == 0)
345               {
346                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        packet_len"), pmt::mp(P32_D_FIELD_BITS));
347                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        part_rx_freq"), pmt::mp((uint64_t)d_part_channel
                        [d_cur_part_rx_id]));
348                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        part_rx_id"), pmt::mp((uint64_t)d_cur_part_rx_id
                        ));
349                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        rx_seq"), pmt::mp((uint64_t)d_part_seq[
                        d_cur_part_rx_id]));
350                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        part_type"), pmt::mp((d_part_type == _RFP_)?"RFP
                        ":"PP"));
351               }
352
353               oo++;
354
355               if(++d_out_bit_cnt == P32_D_FIELD_BITS)
356                 d_sync_state = _WAIT_BEGIN_;
```

```
357             }
358             break;
359         }
360
361         if(d_selected_rx_mode_changed)
362         {
363           d_selected_rx_mode_changed=false;
364
365           //Clear parts list
366           if(d_selected_rx_mode == 0 || d_selected_rx_mode == 1)
367             d_part_activity=0;
368
369           //send message to change mode
370           pmt::pmt_t msg = pmt::make_dict();
371           msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt::mp
                  ("change_mode"));
372           msg = pmt::dict_add(msg, pmt::mp("rx_mode"), pmt::mp((
                  uint64_t)d_selected_rx_mode));
373           message_port_pub(pmt::mp("rcvr_msg_out"), msg);
374
375           //send data with void tag to refresh next block
376           *out++ = (char)0;
377           add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                  packet_len"), pmt::mp(1));
378           add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                  packet_type"), pmt::mp("void"));
379           oo++;
380           noutput_items++;
381         }
382
383         if(d_selected_rx_mode == 0) //Manuell mode
384         {
385           // Check parts activity and inform packet decoder if a
                  part becomes inactive
386           int32_t lost_id = check_part_activity(4);
387           if(lost_id >= 0)
388           {
389             pmt::pmt_t msg = pmt::make_dict();
390             msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt::
                    mp("lost_part"));
391             msg = pmt::dict_add(msg, pmt::mp("part_rx_id"), pmt::mp
                    ((uint64_t)lost_id));
392             message_port_pub(pmt::mp("rcvr_msg_out"), msg);
393
394             //send data with void tag to refresh next block
395             *out++ = (char)0;
396             add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                    packet_len"), pmt::mp(1));
397             add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                    packet_type"), pmt::mp("void"));
398             oo++;
399             noutput_items++;
400           }
401
402           //If channel was changed from dropdown menu
403           if(d_rx_chan_changed)
```

```
404             {
405               d_rx_chan_changed=false;
406
407               //send message to change channel
408               pmt::pmt_t msg = pmt::make_dict();
409               msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt::
                      mp("change_freq"));
410               msg = pmt::dict_add(msg, pmt::mp("rx_channel"), pmt::mp
                      ((uint64_t)d_selected_rx_chan));
411               message_port_pub(pmt::mp("rcvr_msg_out"), msg);
412
413               //send data with void tag to refresh next block
414               *out++ = (char)0;
415               add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                      packet_len"), pmt::mp(1));
416               add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                      packet_type"), pmt::mp("void"));
417               oo++;
418               noutput_items++;
419             }
420           }
421           else if(d_selected_rx_mode == 1) //Scan Mode: change
                  frequency every second
422           {
423             if(d_inc_smpl_cnt - d_last_freq_switch_time > (40 *
                    INTER_FRAME_TIME))     //change channel after 4*
                    INTER_FRAME_TIME
424             {
425               d_last_freq_switch_time = d_inc_smpl_cnt;
426
427               if(d_channel_incr < 9)
428                 d_channel_incr++;
429               else
430                 d_channel_incr = 0;
431
432               //send message to change channel
433               pmt::pmt_t msg = pmt::make_dict();
434               msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt::
                      mp("change_freq"));
435               msg = pmt::dict_add(msg, pmt::mp("rx_channel"), pmt::mp
                      ((uint64_t)d_channel_incr));
436               message_port_pub(pmt::mp("rcvr_msg_out"), msg);
437
438               //send data with void tag to refresh next block
439               *out++ = (char)0;
440               add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                      packet_len"), pmt::mp(1));
441               add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                      packet_type"), pmt::mp("void"));
442               oo++;
443               noutput_items++;
444
445               //update part information (channel)
446               //check if parts are lost
447               int32_t lost_id = check_part_activity(500);
448               if(lost_id >= 0)
```

```
449               {
450                 pmt::pmt_t msg = pmt::make_dict();
451                 msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt
                        ::mp("lost_part"));
452                 msg = pmt::dict_add(msg, pmt::mp("part_rx_id"), pmt::
                        mp((uint64_t)lost_id));
453                 message_port_pub(pmt::mp("rcvr_msg_out"), msg);
454
455                 //send data with void tag to refresh next block
456                 *out++ = (char)0;
457                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        packet_len"), pmt::mp(1));
458                 add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        packet_type"), pmt::mp("void"));
459                 oo++;
460                 noutput_items++;
461               }
462             }
463           }
464           else if(d_selected_rx_mode == 2)  //Follow selected part
465           {
466             if(d_part_rx_id_changed)
467             {
468               d_part_rx_id_changed=false;
469               //send message to change channel
470               pmt::pmt_t msg = pmt::make_dict();
471               msg = pmt::dict_add(msg, pmt::mp("rcvr_msg_id"), pmt::
                      mp("change_freq"));
472               msg = pmt::dict_add(msg, pmt::mp("rx_channel"), pmt::mp
                        ((uint64_t)d_part_channel[d_part_rx_id]));
473               message_port_pub(pmt::mp("rcvr_msg_out"), msg);
474
475               //send data with void tag to refresh next block
476               *out++ = (char)0;
477               add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        packet_len"), pmt::mp(1));
478               add_item_tag(0, nitems_written(0) + oo, pmt::mp("
                        packet_type"), pmt::mp("void"));
479               oo++;
480               noutput_items++;
481             }
482           }
483           d_smpl_buf_index = (d_smpl_buf_index + 1 ) & (SMPL_BUF_LEN
                  - 1);
484           d_rx_bits_buf_index = (d_rx_bits_buf_index + 1) & 3;
485           d_smpl_cnt = (d_smpl_cnt + 1) & 3;
486           d_inc_smpl_cnt++;   // Increase incomming samples counter
487
488           ii++;
489         }
490         consume_each(ii);
491         return oo;
492       }
493     } /* namespace dectblocks */
494 } /* namespace gr */
```

B.2 Packet-Decoder

B.2.1 dectblocks_packet_decoder.xml

```
<block>
  <name>Packet decoder</name>
  <key>dectblocks_packet_decoder</key>
  <category>DECTBLOCKS</category>
  <import>import dectblocks</import>
  <make>dectblocks.packet_decoder()</make>
  <callback>select_rx_part($rx_id)</callback>
  <param>
    <name>Part RX ID</name>
    <key>rx_id</key>
    <value>9</value>
    <type>int</type>
  </param>
  <sink>
    <name>in</name>
    <type>byte</type>
  </sink>
  <sink>
    <name>rcvr_msg_in</name>
    <type>message</type>
    <optional>1</optional>
  </sink>
  <source>
    <name>out</name>
    <type>byte</type>
  </source>
  <source>
    <name>out_freq</name>
    <type>int</type>
    <optional>1</optional>
  </source>
  <source>
    <name>log_out</name>
    <type>message</type>
    <optional>1</optional>
  </source>
</block>
```

B.2.2 packet_decoder_impl.h

```
1  #ifndef INCLUDED_DECTBLOCKS_PACKET_DECODER_IMPL_H
2  #define INCLUDED_DECTBLOCKS_PACKET_DECODER_IMPL_H
3
4  #include <dectblocks/packet_decoder.h>
5
6  #define MAX_PARTS          8
7  #define A_FIELD_BITS      64
8  #define B_FIELD_BITS     320
9
10 namespace gr {
11   namespace dectblocks {
12     class packet_decoder_impl : public packet_decoder
13     {
14       private:
15         typedef enum {_RFP_, _PP_} part_type;
16
17         typedef struct part_descriptor_item
18         {
19           bool     active;
20           bool     voice_present;
21           bool     log_update;
22           bool     part_id_rcvd;
23           bool     qt_rcvd;
24
25           bool     rpf_fn_cor;   // set true if frame number was
                corrected from RFP part
26
27           uint8_t  frame_number;
28           uint64_t rx_seq;
29           uint32_t rx_chan;       //channel
30           uint8_t  part_id[5];
31           part_type type;
32
33           uint64_t packet_cnt;
34           uint64_t afield_bad_crc_cnt;
35
36           uint8_t  a_field_data[8];
37
38           //static informations
39           bool     staticinf_rcvd;
40           bool     nr_bit;    //normal-reverse bit
41           uint8_t  sn_bits;   //slot number bits
42           uint8_t  sp_bits;   //start position bits
43           bool     esc_bit;   //wheter QT escapte is broadcast
44           uint8_t  nrtrans_bits; //number of transceivers in RFP
45           bool     extinf_bit;   //Extended carrier information
                available
46           uint16_t carrier_bits; //which carriers are available
47           uint8_t  carrier_number; //carrier number of transmission
48           uint8_t  nextcarrier_number; //RF carrier on which one
                receiver will be listening on the next frame
49
50           struct part_descriptor_item *pair;
51         } part_descriptor_item;
```

```
52
53          part_descriptor_item d_part_descriptor[MAX_PARTS];
54          part_descriptor_item *d_cur_part;
55          uint32_t d_selected_rx_id;
56          uint32_t d_selected_rx_mode;
57          uint32_t d_change_to_rx_channel;
58
59          uint32_t chan_to_freq[10];
60
61          uint32_t decode_afield(uint8_t *field_data);
62
63          int calculate_output_stream_length(const gr_vector_int &
                ninput_items);
64          void msg_event_handler(pmt::pmt_t msg);
65
66          void print_parts(void);
67          void print_mode(void);
68
69        public:
70          packet_decoder_impl();
71          ~packet_decoder_impl();
72
73          virtual void select_rx_part(uint32_t rx_id);
74
75          int work(int noutput_items,
76            gr_vector_int &ninput_items,
77            gr_vector_const_void_star &input_items,
78            gr_vector_void_star &output_items);
79      };
80    } // namespace dectblocks
81  } // namespace gr
82  #endif /* INCLUDED_DECTBLOCKS_PACKET_DECODER_IMPL_H */
```

B.2.3 packet_decoder_impl.cc

```
1  #ifdef HAVE_CONFIG_H
2  #include "config.h"
3  #endif
4
5  #include <gnuradio/io_signature.h>
6  #include "packet_decoder_impl.h"
7  #include <cstdio>
8
9  namespace gr {
10   namespace dectblocks {
11     // scramble table with corrections by Jakub Hruska
12     static const uint8_t scrt[8][31]=
13     {
14       {0x3B, 0xCD, 0x21, 0x5D, 0x88, 0x65, 0xBD, 0x44, 0xEF, 0x34,
             0x85, 0x76, 0x21, 0x96, 0xF5, 0x13, 0xBC, 0xD2, 0x15, 0xD8
             , 0x86, 0x5B, 0xD4, 0x4E, 0xF3, 0x48, 0x57, 0x62, 0x19, 0
             x6F, 0x51},
15       {0x32, 0xDE, 0xA2, 0x77, 0x9A, 0x42, 0xBB, 0x10, 0xCB, 0x7A,
             0x89, 0xDE, 0x69, 0x0A, 0xEC, 0x43, 0x2D, 0xEA, 0x27, 0x79
             , 0xA4, 0x2B, 0xB1, 0x0C, 0xB7, 0xA8, 0x9D, 0xE6, 0x90, 0
             xAE, 0xC4},
16       {0x2D, 0xEA, 0x27, 0x79, 0xA4, 0x2B, 0xB1, 0x0C, 0xB7, 0xA8,
             0x9D, 0xE6, 0x90, 0xAE, 0xC4, 0x32, 0xDE, 0xA2, 0x77, 0x9A
             , 0x42, 0xBB, 0x10, 0xCB, 0x7A, 0x89, 0xDE, 0x69, 0x0A, 0
             xEC, 0x43},
17       {0x27, 0x79, 0xA4, 0x2B, 0xB1, 0x0C, 0xB7, 0xA8, 0x9D, 0xE6,
             0x90, 0xAE, 0xC4, 0x32, 0xDE, 0xA2, 0x77, 0x9A, 0x42, 0xBB
             , 0x10, 0xCB, 0x7A, 0x89, 0xDE, 0x69, 0x0A, 0xEC, 0x43, 0
             x2D, 0xEA},
18       {0x19, 0x6F, 0x51, 0x3B, 0xCD, 0x21, 0x5D, 0x88, 0x65, 0xBD,
             0x44, 0xEF, 0x34, 0x85, 0x76, 0x21, 0x96, 0xF5, 0x13, 0xBC
             , 0xD2, 0x15, 0xD8, 0x86, 0x5B, 0xD4, 0x4E, 0xF3, 0x48, 0
             x57, 0x62},
19       {0x13, 0xBC, 0xD2, 0x15, 0xD8, 0x86, 0x5B, 0xD4, 0x4E, 0xF3,
             0x48, 0x57, 0x62, 0x19, 0x6F, 0x51, 0x3B, 0xCD, 0x21, 0x5D
             , 0x88, 0x65, 0xBD, 0x44, 0xEF, 0x34, 0x85, 0x76, 0x21, 0
             x96, 0xF5},
20       {0x0C, 0xB7, 0xA8, 0x9D, 0xE6, 0x90, 0xAE, 0xC4, 0x32, 0xDE,
             0xA2, 0x77, 0x9A, 0x42, 0xBB, 0x10, 0xCB, 0x7A, 0x89, 0xDE
             , 0x69, 0x0A, 0xEC, 0x43, 0x2D, 0xEA, 0x27, 0x79, 0xA4, 0
             x2B, 0xB1},
21       {0x79, 0xA4, 0x2B, 0xB1, 0x0C, 0xB7, 0xA8, 0x9D, 0xE6, 0x90,
             0xAE, 0xC4, 0x32, 0xDE, 0xA2, 0x77, 0x9A, 0x42, 0xBB, 0x10
             , 0xCB, 0x7A, 0x89, 0xDE, 0x69, 0x0A, 0xEC, 0x43, 0x2D, 0
             xEA, 0x27}
22     };
23
24     static const uint16_t crc_table[16] =
25     {
26       0x0000, 0x0589, 0x0b12, 0x0e9b, 0x1624, 0x13ad, 0x1d36, 0
             x18bf,
27       0x2c48, 0x29c1, 0x275a, 0x22d3, 0x3a6c, 0x3fe5, 0x317e, 0
             x34f7
28     };
```

```
29
30      static uint16_t calc_rcrc(uint8_t *data, unsigned data_len)
31      {
32        uint16_t crc;
33        unsigned tbl_idx;
34
35        crc = 0x0000;
36        while (data_len--)
37        {
38          tbl_idx = (crc >> 12) ^ (*data >> 4);
39          crc = crc_table[tbl_idx & 0x0f] ^ (crc << 4);
40          tbl_idx = (crc >> 12) ^ (*data >> 0);
41          crc = crc_table[tbl_idx & 0x0f] ^ (crc << 4);
42          data++;
43        }
44        return crc ^ 0x0001;
45      }
46
47      static uint8_t calc_xcrc(uint8_t *b_field)
48      {
49        uint8_t rbits[10];
50        uint8_t gp = 0x10;
51        uint8_t crc;
52        uint8_t next;
53        uint32_t i, j;
54        uint32_t bi;
55        uint32_t bw;
56        uint32_t nb;
57        uint8_t  rbyte;
58        uint32_t rbit_cnt, rbyte_cnt;
59
60        // Extract test bits
61        memset(rbits, 0, sizeof(rbits));
62        rbit_cnt = 0;
63        rbyte_cnt = 0;
64        for(i = 0; i <= (83 - 4); i++)
65        {
66          bi = i + 48  * (1 + (i >> 4));
67          nb = bi >> 3;
68          bw = b_field[nb];
69
70          rbyte <<= 1;
71          rbyte |= (bw >> (7 - (bi - (nb << 3)))) & 1;
72
73          if(++rbit_cnt == 8)
74          {
75            rbits[rbyte_cnt++] = rbyte;
76            rbit_cnt = 0;
77          }
78        }
79
80        crc = rbits[0];
81        i = 0;
82        while(i < 10)
83        {
84          if(i < (10 - 1))
```

```
85            next = rbits[i + 1];
86          else
87            next=0;
88          i++;
89          j = 0;
90          while(j < 8)
91          {
92            while(!(crc & 0x80))
93            {
94              crc <<= 1;
95              crc |= !!(next & 0x80);
96              next <<= 1;
97              j++;
98              if(j > 7)
99              break;
100           }
101           if(j > 7)
102             break;
103           crc <<= 1;
104           crc |= !!(next & 0x80);
105           next <<= 1;
106           j++;
107           crc ^= gp;
108         }
109       }
110       return crc >> 4;
111     }
112
113     static bool part_id_cmp(uint8_t *id1, uint8_t *id2)
114     {
115       for(uint32_t i = 0; i < 5; i++)
116       if(id1[i] != id2[i])
117         return false;
118       return true;
119     }
120
121     uint32_t packet_decoder_impl::decode_afield(uint8_t *field_data
        )
122     {
123       for(int i=0;i<8;i++)
124         d_cur_part->a_field_data[i]=field_data[i];
125
126       uint16_t rcrc = (uint16_t)field_data[6] << 8 | field_data[7];
127       uint16_t crc  = calc_rcrc(field_data, 6);
128
129       if(crc != rcrc)
130       {
131         d_cur_part->afield_bad_crc_cnt++;
132         return 0;
133       }
134
135       uint8_t  afield_header = field_data[0];
136       uint8_t  ta_bits = (afield_header >> 5) & 0x07;
137
138       //check TA bits (a0-a2)
139       switch(ta_bits)
```

```
140         {
141           case 0:
142             break;
143
144           case 1:
145             break;
146
147           case 3:
148             d_cur_part->part_id[0] = field_data[1];
149             d_cur_part->part_id[1] = field_data[2];
150             d_cur_part->part_id[2] = field_data[3];
151             d_cur_part->part_id[3] = field_data[4];
152             d_cur_part->part_id[4] = field_data[5];
153             d_cur_part->part_id_rcvd = true;
154             break;
155
156           case 4:  // multiframe synchronization and system
                  information (Qt) - translated every 16 frames in frame
                  number 8
157           {
158             //std::cout << "====== FRAME 8 ======" << std::endl;
159             d_cur_part->frame_number = 8;
160             d_cur_part->qt_rcvd = true;
161
162             //check QH bits
163             uint8_t qh_bits = (field_data[1] >> 5) & 0x07;
164
165             //000X -> static system information (a8-a10)
166             if(qh_bits == 0)
167             {
168               //Normal-Reverse bit (a11)
169               if((field_data[1] >> 4) & 0x01) //reversed half frame
170                 d_cur_part->nr_bit=1;
171               else                           //normal half frame
172                 d_cur_part->nr_bit=0;
173
174               //Slot Number (a12-a15)
175               //0 -> slot-pair{0,12}
176               //1 -> slot-pair{1,13}
177               d_cur_part->sn_bits = field_data[1] & 0x0b;
178
179               //Start Position (a16-a17)
180               d_cur_part->sp_bits = (field_data[2] >> 6) & 0x03;
181
182               //Escape bit (a18)
183               d_cur_part->esc_bit = (field_data[2] >> 5) & 0x01;
184
185               //Number of transceivers in RFP (a19-a20)
186               //0 -> 1 transceiver
187               //1 -> 2 transceiver
188               //2 -> 3 transceiver
189               //3 -> 4 or more
190               d_cur_part->nrtrans_bits = (field_data[2] >> 3) & 0x03;
191
192               //Extended carrier information available (a21)
193               d_cur_part->extinf_bit = (field_data[2] >> 2) & 0x01;
```

```
194
195             //available RF carriers (a22-a31)
196             d_cur_part->carrier_bits = ((field_data[2] & 0x03) <<
                    8) | field_data[3];
197
198             //spare bits (a32-a33)
199             //use is not defined yet
200
201             //carrier number (a34-a39)
202             d_cur_part->carrier_number = field_data[4] & 0x3f;
203
204             //extended static system information available (a40)
205             //not interesting
206
207             //spare bit (a41)
208             //use is not defined yet
209
210             //primary receiver scan carrier number (a42-a47)
211             d_cur_part->nextcarrier_number = field_data[5] & 0x3f;
212
213             if(d_cur_part->staticinf_rcvd==false)
214             {
215               d_cur_part->staticinf_rcvd=true;
216               print_parts();
217             }
218           }
219           break;
220         }
221
222         case 6:
223           //mt_parse(afield.tail);
224           break;
225
226         case 7:
227           //if(pt == _RFP_)
228           //pt_parse(afield.tail);
229           break;
230       }
231
232       //check BA bits
233       if(((afield_header >> 1) & 7) == 0)
234       {
235         if(d_cur_part->voice_present == false)
236         {
237           d_cur_part->voice_present = true;
238           d_cur_part->log_update = true;
239         }
240       }
241       else
242       {
243         if(d_cur_part->voice_present == true)
244         {
245           d_cur_part->voice_present = false;
246           d_cur_part->log_update = true;
247         }
248       }
```

```
249       return 1;
250     }
251
252     packet_decoder::sptr
253     packet_decoder::make()
254     {
255       return gnuradio::get_initial_sptr
256       (new packet_decoder_impl());
257     }
258
259
260     packet_decoder_impl::packet_decoder_impl()
261       : gr::tagged_stream_block("packet_decoder",
262         gr::io_signature::make(1, 1, sizeof(unsigned char)),
263         gr::io_signature::make2(2, 2, sizeof(unsigned char), sizeof
              (uint32_t)), std::string("packet_len"))
264     {
265       set_tag_propagation_policy(TPP_DONT);
266
267       chan_to_freq[0]=1897344000;
268       chan_to_freq[1]=1895616000;
269       chan_to_freq[2]=1893888000;
270       chan_to_freq[3]=1892160000;
271       chan_to_freq[4]=1890432000;
272       chan_to_freq[5]=1888704000;
273       chan_to_freq[6]=1886876000;
274       chan_to_freq[7]=1885248000;
275       chan_to_freq[8]=1883520000;
276       chan_to_freq[9]=1881792000;
277
278       d_selected_rx_id = 9;
279       d_selected_rx_mode = 0;
280       d_change_to_rx_channel = 0;
281
282       message_port_register_in(pmt::mp("rcvr_msg_in"));
283       set_msg_handler(pmt::mp("rcvr_msg_in"), boost::bind(&
              packet_decoder_impl::msg_event_handler, this, _1));
284       message_port_register_out(pmt::mp("log_out"));
285
286       //clear parts list
287       memset(&d_part_descriptor, 0, sizeof(d_part_descriptor));
288     }
289
290     packet_decoder_impl::~packet_decoder_impl()
291     {
292     }
293
294     int packet_decoder_impl::calculate_output_stream_length(const
           gr_vector_int &ninput_items)
295     {
296       int noutput_items = 80;
297       return noutput_items ;
298     }
299
300     void packet_decoder_impl::msg_event_handler(pmt::pmt_t msg)
301     {
```

```
302         if(pmt::dict_has_key( msg, pmt::mp("rcvr_msg_id")))
303         {
304           pmt::pmt_t msg_id = pmt::dict_ref( msg, pmt::mp("
                rcvr_msg_id"), pmt::PMT_NIL);
305           if(pmt::eq(msg_id, pmt::mp("lost_part")))
306           {
307             // Remove active part
308             uint32_t rx_id = (uint32_t)pmt::to_uint64(pmt::dict_ref(
                  msg, pmt::mp("part_rx_id"), pmt::PMT_NIL));
309
310             part_descriptor_item *part_item = &d_part_descriptor[
                  rx_id];
311             part_item->active = false;
312             part_item->voice_present = false;
313             part_item->log_update = false;
314             part_item->qt_rcvd = false;
315             if(part_item->part_id_rcvd == true)
316               print_parts();
317
318             // Cleare part's pair
319             if(part_item->pair != NULL)
320             {
321               if(part_item->type == _PP_)
322                 part_item->pair->pair = NULL;
323               else if(part_item->type == _RFP_)
324               {
325                 part_item->pair->voice_present = false;
326                 part_item->pair->pair = NULL;
327               }
328               part_item->pair = NULL;
329             }
330
331             //delete static information
332             part_item->staticinf_rcvd=false;
333
334           }
335           if(pmt::eq(msg_id, pmt::mp("change_freq")))
336           {
337             d_change_to_rx_channel = (uint32_t)pmt::to_uint64(pmt::
                  dict_ref( msg, pmt::mp("rx_channel"), pmt::PMT_NIL));
338
339             if(d_selected_rx_mode==0)
340             {
341               std::ostringstream os;
342               os << "+++RX Frequency changed to " <<
                    d_change_to_rx_channel << " (" << chan_to_freq[
                    d_change_to_rx_channel] << ")";
343
344               os << "+++" << std::endl;
345               pmt::pmt_t msg = pmt::make_dict();
346               msg = pmt::dict_add(msg, pmt::mp("log_msg"), pmt::mp(os
                    .str()));
347               message_port_pub(pmt::mp("log_out"), msg);
348             }
349           }
350
```

```
351         if(pmt::eq(msg_id, pmt::mp("change_mode")))
352         {
353           d_selected_rx_mode = (uint32_t)pmt::to_uint64(pmt::
                dict_ref( msg, pmt::mp("rx_mode"), pmt::PMT_NIL));
354
355           print_mode();
356
357           if(d_selected_rx_mode == 0 || d_selected_rx_mode == 1)
358           {
359             memset(&d_part_descriptor, 0, sizeof(d_part_descriptor)
                  );
360             print_parts();
361           }
362         }
363       }
364     }
365
366     void packet_decoder_impl::print_parts(void)
367     {
368       std::ostringstream os;
369
370       os << "====== AVAILABLE PARTS ======" << std::endl;
371       for(uint32_t rx_id = 0; rx_id < MAX_PARTS; rx_id++)
372       {
373         if(d_part_descriptor[rx_id].active == true)
374         {
375           part_descriptor_item *part_item = &d_part_descriptor[
                rx_id];
376           if(d_selected_rx_id == rx_id)
377             os << "* ";
378           else
379             os << "  ";
380
381           os << rx_id << "  Ch:" << part_item->rx_chan << "  " <<
                std::hex << std::setfill('0') << std::setw(2) << (
                uint32_t)part_item->part_id[0] << \
382           std::setfill('0') << std::setw(2) << (uint32_t)part_item
                ->part_id[1] << \
383           std::setfill('0') << std::setw(2) << (uint32_t)part_item
                ->part_id[2] << \
384           std::setfill('0') << std::setw(2) << (uint32_t)part_item
                ->part_id[3] << \
385           std::setfill('0') << std::setw(2) << (uint32_t)part_item
                ->part_id[4];
386
387           if(part_item->type == _RFP_)
388             os << " RFP ";
389           else
390             os << " PP  ";
391
392           if(part_item->voice_present)
393             os << "  " << "V";
394           else
395             os << "  ";
396
397           //print a field in binaries
```

```
398             os << "\nA-Field: ";
399             for(int i=0;i<8;i++)
400             {
401               os << std::hex << std::setfill('0') << std::setw(2) <<
                      (uint32_t)part_item->a_field_data[i];
402
403               if(i<7)
404                 os << "|";
405             }
406             os << std::endl;
407
408             //if static information was received
409             if(part_item->staticinf_rcvd)
410             {
411               os << "Slot Pair: " << std::dec << (uint8_t)part_item->
                      sn_bits+0;
412               os << ", " << std::dec << (uint8_t)part_item->sn_bits
                      +12 << std::endl;
413
414               os << "Nr of Transceivers: ";
415               if(part_item->nrtrans_bits < 3)
416                 os << std::dec << part_item->nrtrans_bits+1 << std::
                        endl;
417               else
418                 os << "4 or more" << std::endl;
419
420               os << "Extended RF carrier information: " << part_item
                      ->extinf_bit << std::endl;
421
422               os << "Supported RF carriers: ";
423               for(int i=0; i<10; i++)
424                 if(part_item->carrier_bits >> i)
425                   os << i << " ";
426
427               os << std::endl;
428
429               os << "Current RF carrier: " << std::dec << part_item->
                      carrier_number+0 << std::endl;
430
431               os << "Next RF carrier: " << std::dec << part_item->
                      nextcarrier_number+0 << std::endl;
432             }
433             os << std::endl;
434           }
435         }
436         pmt::pmt_t msg = pmt::make_dict();
437         msg = pmt::dict_add(msg, pmt::mp("log_msg"), pmt::mp(os.str()
                ));
438         message_port_pub(pmt::mp("log_out"), msg);
439       }
440
441       void packet_decoder_impl::print_mode(void)
442       {
443         std::ostringstream os;
444         os << "+++Receiver Mode changed to ";
445
```

```
446         switch(d_selected_rx_mode){
447           case 0: os << "Manuel"; break;
448           case 1: os << "Scan"; break;
449           case 2: os << "Listen"; break;
450         }
451
452         os << "+++";
453         pmt::pmt_t msg = pmt::make_dict();
454         msg = pmt::dict_add(msg, pmt::mp("log_msg"), pmt::mp(os.str()
                ));
455         message_port_pub(pmt::mp("log_out"), msg);
456       }
457
458       void packet_decoder_impl::select_rx_part(uint32_t rx_id)
459       {
460         d_selected_rx_id = rx_id;
461         print_parts();
462       }
463
464       int packet_decoder_impl::work (int noutput_items,
465         gr_vector_int &ninput_items,
466         gr_vector_const_void_star &input_items,
467         gr_vector_void_star &output_items)
468       {
469         const uint8_t *in = (const uint8_t *) input_items[0];
470         uint8_t *out = (uint8_t *) output_items[0];
471         uint32_t *out_freq = (uint32_t *) output_items[1];
472         uint32_t packet_length = ninput_items[0];
473
474         uint32_t rx_freq;
475         uint32_t rx_id;
476         uint64_t rx_seq;
477         part_type ptype = _RFP_;
478
479         noutput_items = 0;
480
481         std::vector<tag_t> tags;
482         get_tags_in_range(tags, 0, nitems_read(0), nitems_read(0) +
              packet_length);
483
484         for (size_t i = 0; i < tags.size(); i++)
485         {
486           if(pmt::eq(tags[i].key, pmt::mp("packet_type")))
487             if(pmt::eq(tags[i].value, pmt::mp("void")))
488             {
489               *in++;
490               *out++ = 0;
491               *out_freq++ = chan_to_freq[d_change_to_rx_channel];
492               noutput_items++;
493               return noutput_items;
494             }
495
496           if(pmt::eq(tags[i].key, pmt::mp("part_rx_freq")))
497             rx_freq = (uint32_t)pmt::to_uint64(tags[i].value);
498           else if(pmt::eq(tags[i].key, pmt::mp("part_rx_id")))
499             rx_id = (uint32_t)pmt::to_uint64(tags[i].value);
```

```
500           else if(pmt::eq(tags[i].key, pmt::mp("rx_seq")))
501             rx_seq = (uint64_t)pmt::to_uint64(tags[i].value);
502           else if(pmt::eq(tags[i].key, pmt::mp("part_type")))
503           {
504             if(pmt::eq(tags[i].value, pmt::mp("RFP")))
505               ptype = _RFP_;
506             else
507               ptype = _PP_;
508           }
509         }
510
511         d_cur_part = &d_part_descriptor[rx_id];
512
513         if(d_cur_part->active == true)
514         {
515           uint64_t seq_diff = (rx_seq - d_cur_part->rx_seq) & 0x1F;
516
517           if(ptype == _RFP_)
518           {
519             d_cur_part->frame_number = (d_cur_part->frame_number +
                  seq_diff) & 0xF;
520
521             // Update frame number for pair if available
522             if(d_cur_part->pair != NULL)
523             {
524               d_cur_part->pair->frame_number = d_cur_part->
                    frame_number;
525               d_cur_part->pair->rpf_fn_cor = true;
526             }
527           }
528           else if(ptype == _PP_)
529           {
530             if(d_cur_part->rpf_fn_cor == true)
531               d_cur_part->rpf_fn_cor =  false;
532             else
533               d_cur_part->frame_number = (d_cur_part->frame_number +
                    seq_diff) & 0xF;
534           }
535
536           d_cur_part->rx_seq = rx_seq;
537           d_cur_part->packet_cnt++;
538         }
539         else
540         {
541           // Register a new part
542           d_cur_part->active = true;
543           d_cur_part->frame_number = 0;
544           d_cur_part->rx_seq = rx_seq;
545           d_cur_part->rx_chan = rx_freq;
546           d_cur_part->voice_present = false;
547           d_cur_part->packet_cnt = 0;
548           d_cur_part->afield_bad_crc_cnt = 0;
549           d_cur_part->log_update = true;
550           d_cur_part->part_id_rcvd =  false;
551           d_cur_part->qt_rcvd = false;
552           d_cur_part->type = ptype;
```

```
553           d_cur_part->pair = NULL;
554         }
555
556         // Try to find pair RFP for PP
557         if(d_cur_part->pair == NULL && d_cur_part->type == _PP_ &&
                d_cur_part->part_id_rcvd == true)
558         {
559           for(uint32_t i = 0; i < MAX_PARTS; i++)
560           {
561             if(i != rx_id)
562               if(d_part_descriptor[i].active == true)
563                 if(part_id_cmp(d_cur_part->part_id, d_part_descriptor
                        [i].part_id) == true)
564                 {
565                   d_cur_part->pair = &d_part_descriptor[i];
566                   d_part_descriptor[i].pair = d_cur_part;
567                 }
568           }
569         }
570
571         uint8_t tmp_byte;
572         uint32_t a_field_byte_cnt = 0;
573         uint8_t a_field[8];
574
575         // Extract A-field
576         for(uint32_t i = 0; i < A_FIELD_BITS; i++)
577         {
578           if(i && ((i & 0x7) == 0))
579             a_field[a_field_byte_cnt++] = tmp_byte;
580           tmp_byte = (tmp_byte << 1) | (*in++ & 0x1);
581         }
582         a_field[a_field_byte_cnt] = tmp_byte;
583
584         decode_afield(a_field);
585
586         if(ptype == _RFP_ && d_cur_part->qt_rcvd && d_cur_part->pair
                != NULL)
587           d_cur_part->pair->qt_rcvd = true;
588
589         if(d_cur_part->log_update && d_cur_part->part_id_rcvd)
590         {
591           print_parts();
592           d_cur_part->log_update = false;
593         }
594
595         if(rx_id == d_selected_rx_id && d_selected_rx_mode !=1)
596         {
597           if(d_cur_part->active && d_cur_part->voice_present &&
                  d_cur_part->qt_rcvd)
598           {
599             uint8_t b_field[40];
600             uint8_t tmp_byte;
601             uint32_t b_field_byte_cnt = 0;
602
603             for(uint32_t i = 0; i < B_FIELD_BITS; i++)
604             {
```

```
605             if(i && ((i & 0x7) == 0))
606               b_field[b_field_byte_cnt++] = tmp_byte;
607             tmp_byte = (tmp_byte << 1) | (*in++ & 0x1);
608           }
609
610           b_field[b_field_byte_cnt] = tmp_byte;
611           uint8_t xcrc = calc_xcrc(b_field);
612           uint8_t x_field = 0;
613           x_field |= ((*in++ & 0x1) << 3);
614           x_field |= ((*in++ & 0x1) << 2);
615           x_field |= ((*in++ & 0x1) << 1);
616           x_field |= (*in & 0x1);
617
618           if(xcrc == x_field)
619           {
620             uint8_t *ptr = b_field;
621             uint32_t whitener_offset=d_cur_part->frame_number % 8;
622             uint8_t descrt_byte;
623
624             for(uint32_t i = 0; i < 40; i++)
625             {
626               descrt_byte = *ptr++ ^ scrt[whitener_offset][i % 31];
627               *out++ = (descrt_byte >> 4) & 0xF;
628               *out++ =  descrt_byte & 0xF;
629               *out_freq++ = chan_to_freq[d_change_to_rx_channel];
630               *out_freq++ = chan_to_freq[d_change_to_rx_channel];
631             }
632             noutput_items += 80;
633           }
634           else
635           {
636             for(uint32_t i = 0; i < 80; i++)
637             {
638               *out++ = 0;
639               *out_freq++ = chan_to_freq[d_change_to_rx_channel];
640             }
641             noutput_items += 80;
642           }
643         }
644         else
645         {
646           for(uint32_t i = 0; i < 80; i++)
647           {
648             *out++ = 0;
649             *out_freq++ = chan_to_freq[d_change_to_rx_channel];
650           }
651           noutput_items += 80;
652         }
653       }
654       return noutput_items;
655     }
656   } /* namespace dectblocks */
657 } /* namespace gr */
```

Erklärung

Ich versichere, dass ich die Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: .....

(Unterschrift)