TECHNISCHE UNIVERSITÄT DRESDEN
FAKULTÄT ELEKTROTECHNIK UND INFORMATIONSTECHNIK
Institut für Nachrichtentechnik

DIPLOMARBEIT

"Passive Detektion konfigurierbarer Kanalmasken in adaptiven Frequenzsprungverfahren am Beispiel von Bluetooth"

von

Hannes Ellinger
geboren am 30.09.1989 in Jena

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

(Dipl.-Ing.)

Tag der Einreichung:	08.09.2015
Betreuer der Diplomarbeit:	Dipl.-Ing. Axel Schmidt
	Dipl.-Ing. Ulf Wetzker
Verantwortlicher Hochschullehrer:	Prof. Dr.-Ing. Eduard A. Jorswieck

Eidesstattliche Erklärung

Hiermit erkläre ich, Hannes Ellinger, geboren am 30. September 1989 in Jena, dass die vorgelegte Diplomarbeit mit dem Titel "Passive Detektion konfigurierbarer Kanalmasken in adaptiven Frequenzsprungverfahren am Beispiel von Bluetooth" durch mich vollkommen selbstständig verfasst wurde. Ich habe keine anderen als die angegebenen Quellen sowie Hilfsmittel benutzt und die Diplomarbeit nicht bereits in derselben oder einer ähnlichen Fassung an einer anderen Fakultät oder einem anderen Fachbereich zur Erlangung eines akademischen Grades eingereicht.

Inhaltsverzeichnis

1 Motivation und Problemstellung **1**
1.1 Stand der Technik . . . 3
1.2 Präzisierung der Aufgabenstellung . . . 5

2 Grundlagen **7**
2.1 Grundlagen des Bluetooth-Standards . . . 7
2.1.1 Physikalische Ebene . . . 8
2.1.2 Logische Ebene . . . 10
2.1.3 Sprungschema . . . 11
2.1.4 Aufbau einer Verbindung . . . 12
2.1.5 Änderungen von Verbindungsparametern . . . 13
2.1.6 Aktivieren von Adaptive Frequency Hopping . . . 14
2.1.7 Relevanz für diese Arbeit . . . 14
2.2 Projekt Ubertooth . . . 15
2.2.1 Mikrocontroller . . . 17
2.2.2 Hostprogramme . . . 18
2.2.3 libbtbb . . . 19

3 Festlegung zu betrachtender Rahmenbedingungen **23**

4 Vorbetrachtung: Untersuchung von Einflüssen verschiedener Störquellen auf die Kanalmaske **25**
4.1 Messaufbau . . . 25
4.1.1 Einrichtung der WLAN-Knoten . . . 29
4.1.2 Traffic-Generator . . . 32
4.1.2.1 Sender . . . 32
4.1.2.2 Galton-Watson-Prozess . . . 33
4.1.2.3 Empfänger . . . 34
4.1.2.4 Konfigurator . . . 34
4.1.3 Bestimmung und Ausgabe der Kanalmaske . . . 35
4.2 Durchgeführte Messungen . . . 35
4.2.1 WLAN-Störer . . . 36
4.2.2 Bluetooth-Störer . . . 45

4.2.3 IEEE 802.15.4 Störer 48
4.3 Ergebnisse des Vorversuchs 49

5 Entwicklung eines Algorithmus zur passiven Kanalmaskendetektion 53
5.1 Entwurf eines Detektionsalgorithmus 53
5.1.1 Erkennung genutzter Kanäle anhand der beobachteten Pakete . 53
5.1.2 Erkennen der nicht mehr genutzten Kanäle 54
5.1.3 Optimierung zur schnelleren Erkennung von Veränderungen der Kanalmaske 56
5.1.4 Optimierung zur schnelleren initialen Erkennung der Kanalmaske 59
5.1.5 Abfangen eines LMP_set_AFH Pakets 59
5.1.6 Gesamter Detektionsalgorithmus 60
5.1.7 Fehlerbetrachtung 60
5.2 Implementierungen im Ubertooth Projekt 63
5.2.1 Firmware des Mikrocontrollers 63
5.2.2 libbtbb 64
5.2.3 Hostcode 64
5.2.4 libubertooth 65
5.3 Messung der Leistungsfähigkeit des Detektionsalgorithmus 67
5.3.1 Erkennung genutzter Kanäle 67
5.3.2 Erkennung nicht mehr genutzter Kanäle 68
5.3.3 Bewertung des Detektionsalgorithmus 70

6 Auswertung 75

7 Ausblick 77

8 Einsatz der aktuellen Implementierung 81

Abbildungsverzeichnis

2.1 Format eines Bluetooth-Pakets bei verwendung von Basic Rate und Enhanced Data Rate . . . 11
2.2 Handshake beim Page Request . . . 13
2.3 Format eines LMP Pakets . . . 13
2.4 Format des LMP Pakets für die Aktivierung von Adaptive Frequency Hopping . . . 14
2.5 Komponenten des Ubertooth . . . 16
2.6 Datenverarbeitungspfad des Ubertooth . . . 17
2.7 Im Bitstrom befindet sich ein Bluetoothpaket, dessen Präambel mit dem sechsten Bit des Bitstroms startet. . . . 19
2.8 Zwei mögliche Werte für CLK27–0 für eine Folge von empfangenen Paketen. Der Wert 6 scheidet nach Empfang des zweiten Pakets auf Kanal 56 aus. . . . 21

4.1 Im gemessenen Funkspektrum sind keine Funkstörer sichtbar. . . . 26
4.2 Im funkabgeschirmten Zelt bilden sich durch starke Reflexionen Stehwellen aus, die eine Bluetooth-Verbindung stören. . . . 27
4.3 Der gesamte Testaufbau besteht aus drei WLAN-Knoten, zwei Raspberry Pis, Ethernet-Switch und Netzteil. . . . 28
4.4 Belegung des Wannensteckers am WLAN-Knoten . . . 30
4.5 Hauptfenster des Konfigurators . . . 34
4.6 AFH Channel Map bei einer Störung durch WLAN mit einer Senderate von 200 Paketen pro Sekunde für eine Dauer von 100 Sekunden bei 0 dB Dämpfung . . . 37
4.7 AFH Channel Map bei einer Störung durch WLAN mit einer Senderate von 200 Paketen pro Sekunde für eine Dauer von 100 Sekunden bei 31,5 dB Dämpfung . . . 38
4.8 Bei einer Senderate von 50 Paketen pro Sekunde ist keine Anpassung der AFH Channel Map sichtbar. . . . 41
4.9 Erst bei einer Senderate von 150 Paketen pro Sekunde wird das WLAN als Störer erkannt. . . . 42

4.10 AFH Channel Map bei einer Störung durch WLAN bei maximal möglicher Senderate. Deutlich erkennbar ist der Anstieg der WLAN Senderate, sobald die entsprechenden Bluetooth-Kanäle nicht mehr benutzt werden. 44
4.11 Eine BNEP Verbindung als Störer hat keinen Einfluss auf die Kanalmaske. 47
4.12 Auch eine Audioverbindung, die kein Adaptive Frequency Hopping nutzt, stört eine Bluetooth-Verbindung nicht. 48
4.13 Ein Störsignal mit einer Breite von 1 MHz führt dazu, dass mehr als ein Bluetooth-Kanal nicht benutzt wird. 50

5.1 Erfolgswahrscheinlichkeiten für eine korrekte Erkennung von CLK27–0 bei einem bis fünf Kanalfehlern in der Kanalmaske. 58
5.2 Ablauf der Kanalmaskendetektion 62
5.3 Gegenüberstellung der gemessenen Dauer ohne Optimierung und mit Optimierung für verschiedene Kanalfehlerzahlen 69
5.4 Die Detektionsgeschwindigkeit bei Änderungen in der Kanalmaske ist stark abhängig von der gewählten Schwellwahrscheinlichkeit, ab der ein Kanal vom Sniffer als unbenutzt angesehen wird. 71
5.5 Bei starken, wechselnden Störquellen ändert sich die Kanalmaske häufig. Die Geschwindigkeit des Detektionsalgorithmus genügt nicht, um Änderungen der Kanalmaske zu erkennen, bevor sich diese erneut ändert. 73

Abkürzungsverzeichnis

A2DP Advanced Audio Distribution Profile

AC Access Code

AFH Adaptive Frequency Hopping

BLE Bluetooth Low Energy

BNEP Bluetooth Network Encapsulation Protocol

BR Basic Rate

CLK27–0 Taktzählerregister eines Bluetooth-Gerätes, mit dem Frequenzsprünge und Datenübertragungen koordiniert werden

CLKN 3200 Hz Referenztakt eines Bluetooth-Gerätes

DMA Direct Memory Access

EDR Enhanced Data Rate

FHSS Frequency Hopping Spread Spectrum

GFSK Gaussian Frequency Shift Keying

GPIO General-Purpose Input/Output

HID Human Interface Device

ISM Industrial, Scientific and Medical (Frequenzbänder)

JFFS Journaling Flash File System

LAP Lower Address Part

LE Little Endian

LMP Link Manager Protocol

LSB Least Significant Bit

MAC Media Access Control

NAP Nonsignificant Address Part

PANU Personal Area Network User

PSK Phase Shift Keying

PTP Precision Time Protocol

RF Radio Frequency

RP-SMA Reverse Polarity SubMiniature version A

RSSI Received Signal Strength Indicator

SMA SubMiniature version A

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

UAP Upper Address Part

USB Universal Serial Bus

WLAN Wireless Local Area Network

1 Motivation und Problemstellung

Industrielle Fertigung ist mehr und mehr von Kommunikation abhängig. Sei es die Kommunikation zwischen zwei Produktionsstandorten in der Lieferkette oder die Kommunikation zwischen mehreren Fertigungsprozessen oder Maschinen oder allein die Übertragung von Protokolldaten zu einer zentralen Qualitätssicherungsstelle, überall soll die Kommunikation zuverlässig, schnell und günstig sein. Den minimalen Einrichtungsaufwand bieten dabei drahtlose Lösungen, die gegenüber drahtgebundenen Lösungen wertvolle Zeit bei der Einrichtung und ein aufwendiges Kabelmanagement sparen und die Anbindung mobiler Funkknoten ermöglichen.

Um den hohen Ansprüchen an Zuverlässigkeit, Ausfallsicherheit und Echtzeitfähigkeit gerecht zu werden, empfiehlt der *Zentralverband Elektrotechnik- und Elektronikindustrie (ZVEI)*, bei der Ersteinrichtung von Funksystemen eine Frequenzbandplanung durchzuführen [44]. Jedem der konkurrierenden Funksysteme soll dabei ein eigener Bereich im Frequenzband zugeteilt werden. So soll verhindert werden, dass sich nebeneinander arbeitende Funksysteme gegenseitig störend beeinflussen.

Aufgrund der durch die enormen Stückzahlen geringen Fertigungspreise haben sich auch in der Industrie Standards aus dem Consumerbereich durchgesetzt: WLAN nach dem Standard IEEE 801.11 [13], Bluetooth [29] und diverse Unterarten das IEEE 802.14.5-Standards [15] wie ZigBee [1] oder WirelessHART [5]. All diese Standards arbeiten im Lizenzfreien 2,4 GHz ISM Band. Lizenzfrei bedeutet einerseits, dass in diesem Band Sender betrieben werden dürfen, ohne dass eine Sendelizenz erworben werden muss, andererseits wird in diesem Band nicht für eine Übertragungsqualität garantiert.

Im 2,4 GHz Band senden nicht nur die Geräte, die in der Industrieautomation eingesetzt werden, sondern auch WLAN- und Bluetooth-Geräte wie zum Beispiel die Mobiltelefone der Mitarbeiter oder die Drahtlosfunksysteme von Industriebetrieben in der Umgebung. Auch unzureichend geschirmte Mikrowellenöfen stören diesen Frequenzbereich. Hier versagt die Frequenzbandplanung, da nicht ausgeschlossen werden kann, dass solche Störer in Reichweite betrieben werden.

Um gegenseitige Störungen zu minimieren, implementieren alle Standards in diesem Frequenzband Mechanismen, die eine Koexistenz mit anderen Funksystemen des gleichen oder eines anderen Standards gewährleisten. WLAN-Netze benutzen *carrier sensing* und *listen-before-talk* [13], sodass sie den Kanal nur belegen, wenn dieser gerade frei ist. Außerdem arbeiten sie auf festen Kanälen und halten so das restliche Frequenzband frei. Bluetooth hingegen benutzt das Frequenzsprungverfahren, bei dem in rascher Folge der Kanal gewechselt wird. So wird erreicht, dass eine Kollision mit anderen Funksysteme auf einem Kanal nur von kurzer Dauer ist. In der aktuellen Bluetooth-Revision wird die in Kapitel 2.1 beschriebene Technik *Adaptive Frequency Hopping (AFH)* verwendet, bei der das Bluetooth-Gerät Kanalmasken benutzt, um die Kanäle konkurrierender Funksysteme zu meiden. Diese Mechanismen senken die Wahrscheinlichkeit eines Totalausfalls der Funkverbindung, allerdings ist die Qualität der Verbindung dadurch schlechter vorhersagbar. Wird beispielsweise ein WLAN-Netz gestört, bricht seine Datenrate massiv ein und die Latenz steigt.

Störungen dieser Art führen leicht zu Einschränkungen oder Ausfällen der Kommunikationsverbindung. Da Ausfälle im industriellen Umfeld häufig mit hohen Kosten verbunden sind, ist hier eine effektive und schnelle Problemanalyse gewünscht. Die Ursache von auftretenden Verbindungsproblemen liegt häufig nicht allein bei einem einzelnen Funksystem. Daher bedarf es bei Problemen einer umfassenden Analyse der Situation. Das Fraunhofer Institut für integrierte Schaltungen in Dresden entwickelt schnelle und einfach zu bedienende Lösungen zur Diagnose von Funksystemen in der Industrieautomation. Dabei werden unter anderem Sniffer eingesetzt, um den übertragenen Datenstrom abzuhören, zu analysieren und darin aufgetretene Probleme zu erkennen.

Das passive Abhören von Funkverbindungen ist ein häufig thematisiertes Problem. Bei WLAN-Netzen gestaltet es sich als weitgehend einfach. WLAN benutzt feste Kanäle, sodass eine WLAN-Übertragung immer einem festen Frequenzbereich zuzuordnen ist. Außerdem besitzen einige WLAN-Treiber einen so genannten *Monitor Mode*, der es ihnen erlaubt, sämtliche Pakete in der Umgebung zu empfangen, nicht nur jene, die explizit an die eigene Netzwerkadresse adressiert sind.

Bluetooth hingegen besitzt weder einen Monitor Mode, noch benutzt es einen festen Kanal für eine Datenübertragung. Bluetooth-Geräte nutzen im Zuge des Frequency-Hoppings das gesamte 2,4 GHz Band und verweilen nur für eine kurze Periode auf ein und dem selben Kanal. Danach wird der Kanal nach einem pseudozufälligen Sprungschema gewechselt. So ist es nicht möglich, von der aktiven Funkfrequenz direkt auf eine Bluetooth-Übertragung zu schließen. Weiterhin haben passive Sniffer häufig nur ein schmalbandiges Funk-Frontend. Sie können also zu jeder Zeit nur einen Kanal beobachten. Das macht es notwendig, dem Sprungschema der beobachteten Verbindung

synchron zu folgen. Das wiederum setzt Kenntnis des Sprungschemas voraus. Allerdings ist das Sprungschema nur den Geräten einer Bluetooth-Verbindung bekannt, nicht aber einem passiven Sniffer. Seit Revision Bluetooth v1.2 wurde das adaptive Kanalsprungverfahren *Adaptive Frequency Hopping (AFH)* eingeführt. Dabei wird eine Kanalmaske genutzt, um von konkurrierenden Funksystemen gestörte Bluetooth-Kanäle nicht in das Sprungschema einzubeziehen. Die aktuelle Kanalmaske beeinflusst hierbei also das Sprungschema. Die Kanalmaske kann sich bei veränderten Störbedingungen jederzeit ändern und wird nach außen nicht bekannt gegeben. All diese Mechanismen machen es sehr kompliziert, eine Bluetooth-Verbindung mit einem Sniffer abzuhören [33].

Ziel dieser Arbeit ist es, ein Teilproblem des Abhörens einer Bluetooth-Verbindung zu lösen. Es soll eine Methode entwickelt werden, mit der die Kanalmaske einer Bluetooth-Verbindung passiv detektiert werden kann. Wie in Kapitel 8 ausgeführt wird, kann schon allein die Kenntnis der Kanalmaske wertvolle Informationen für die Diagnose von Verbindungsproblemen liefern.

1.1 Stand der Technik

Auf das Abhören von Bluetooth-Verbindungen haben sich einige Hersteller von kommerziellen Bluetooth-Sniffern spezialisiert. Es gibt beispielsweise aktive Bluetooth-Sniffer, die eine Verbindung zum beobachteten Bluetooth-Netz herstellen. Dadurch erhalten sie alle Informationen zur Berechnung des Sprungschemas und erhalten auch Kenntnis der verwendete Kanalmaske. Das setzt jedoch voraus, dass der Sniffer Zugriff auf die beobachtete Bluetooth-Verbindung erhalten kann, was im praktischen Einsatz aus Sicherheitsgründen ausgeschlossen ist. Diese Einschränkung macht aktive Sniffer für die Anwendung im industriellen Umfeld nutzlos.

Eine Alternative ist der passive Sniffer *Frontline FTS4BT* [11]. Er basiert auf einem herkömmlichen Bluetooth-Dongel, dessen Firmware von der Firma Frontline modifiziert wurde, sodass er als Sniffer arbeitet. Der FTS4BT kann das Sprungschema aus der Adresse des Bluetooth-Netzes berechnen. Die Adresse muss dafür allerdings bekannt sein. Es ist nicht möglich, eine beliebige unbekannte Bluetooth-Verbindung in der Umgebung zu analysieren. Außerdem ist der Frontline FTS4BT und seine Software auf das Abhören von Bluetooth-Paketen optimiert, nicht auf die angestrebte Diagnose von Verbindungsproblemen. Da die Firmware nicht quelloffen ist, sind keine Änderungen an ihr möglich, die entsprechende Funktionen nachrüsten würden. Hinzu kommt ein sehr hoher Preis von etwa 10000 $.

Flexibler sind breitbandige Sniffer wie der *Frontline ComProbe Sodera Wideband* [10], der alle Kanäle gleichzeitig beobachten kann und somit kein Frequency-Hopping betreiben muss. Mit diesem Ansatz können sogar mehrere Bluetooth-Übertragungen gleichzeitig analysiert werden. Der Frontline ComProbe Sodera Wideband nutzt intern ein Software-Defined-Radio Modul, um das Empfangssignal abzutasten. Die Datenverarbeitung selbst geschieht in einem angeschlossenen Computer. Dieser Ansatz ist sehr rechenaufwendig, da die Demodulation des Empfangssignals in Software gelöst ist. Der Energieverbrauch ist vergleichsweise hoch, Computer und Empfangsgerät nehmen viel Platz ein und das Gerät ist durch seine Komplexität in der Anschaffung teuer.

Günstiger ist der Ansatz, ein handelsübliches Software-Defined-Radio einzusetzen, wie zum Beispiel das *USRP B200* von Ettus Research [25]. Mit der von Michael Ossmann und Dominic Spill entwickelten Software *gr-bluetooth* [35] existiert ein Modul für GNU Radio [20], das das Bluetooth-Basisband implementiert und als Basis für einen Bluetooth-Sniffer genutzt werden kann. Das Projekt gr-bluetooth ist quelloffen, lässt sich also einfach um Funktionen zur Funkdiagnose erweitern. Allerdings hat dieser Ansatz die gleichen Nachteile wie der Frontline ComProbe Sodera Wideband: Die Notwendigkeit eines leistungsfähigen Computers, hohen Energieverbrauch, dadurch eine hohe Wärmeentwicklung, große Abmessungen des Messaufbaus und weiterhin einen hohen Anschaffungspreis.

Ein Nischenprodukt ist Ubertooth [38], ein Bluetooth-Sniffer im Format eines USB-Sticks. Ubertooth ist ein Projekt, das im Zuge einer Crowdfunding Kampagne entstand und wie gr-bluetooth ebenfalls von Michael Ossmann und Dominic Spill entwickelt wird. Der Sniffer kann unbekannte und sogar versteckte Bluetooth-Geräte in der Umgebung finden, ihre Adressen und ihren Takt ermitteln, daraus das Hopping Muster errechnen und der Verbindung folgen. Hardware und Software sind quelloffen, sodass der Sniffer problemlos und flexibel für den Einsatz in einem Diagnosesystem angepasst werden kann. Durch diese flexible Anpassbarkeit, den kleinen Formfaktor, den geringen Energieverbrauch und den niedrigen Anschaffungspreis eignet sich der Ubertooth-Sniffer sehr gut für den Einsatz in einem Fehlerdiagnosesystem für Bluetooth-Verbindungen im industriellen Umfeld. Bisher hat Ubertooth jedoch keine Unterstützung für Adaptive Frequency Hopping. Einer Verbindung die dieses Sprungverfahren nutzt, kann er daher nicht folgen. Da diese Technik mittlerweile weit verbreitet ist, schränkt das die Einsatzmöglichkeiten des Sniffers stark ein.

1.2 Präzisierung der Aufgabenstellung

Ziel dieser Arbeit ist es, einen Algorithmus zu entwickeln und auf dem quelloffenen Bluetooth-Sniffer Ubertooth zu implementieren, mit dem die Kanalmaske einer Bluetooth-Verbindung detektiert werden kann. Das fertige Programm soll in den Quelltext des Sniffers einfließen und ihn in seiner Funktion erweitern.

Um dieses Ziel zu erreichen, wird zunächst in Kapitel 2 das genaue Verhalten von Bluetooth-Verbindungen untersucht, sowie die Fähigkeiten des Ubertooth Sniffers vorgestellt.

Aus den Möglichkeiten einer passiven Kanalmaskendetektion, die der Bluetooth-Standard bietet und den Fähigkeiten des Ubertooth-Sniffers werden in Kapitel 3 Annahmen für Rahmenbedingungen abgeleitet, unter denen die Kanalmaskendetektion entwickelt werden soll.

Ein Vorversuch, der das Verhalten von Adaptive Frequency Hopping unter möglichst vielen verschiedenen Störszenarien ermitteln soll, wird in Kapitel 4 durchgeführt.

In Kapitel 5 werden verschiedene Algorithmen zur passiven Detektion der Kanalmaske entwickelt und vorgestellt. Schließlich wird die Leistungsfähigkeit des Detektionsalgorithmus experimentell ermittelt und bewertet. Eine Zusammenfassung der erreichten Ergebnisse und ein Ausblick auf weiterführende Arbeiten schließen die Arbeit ab.

2 Grundlagen

Im folgenden Kapitel werden die für die Aufgabenstellung relevanten Komponenten und Prozeduren des Bluetooth-Standards erklärt und die Funktionen und Fähigkeiten des Ubertooth-Sniffers vorgestellt.

2.1 Grundlagen des Bluetooth-Standards

Bluetooth ist ein vielseitiges drahtloses Übertragungsprotokoll, das im 2,4 GHz ISM-Band arbeitet. Das Hauptaugenmerk liegt auf einem breiten Einsatzfeld und auf einer guten Koexistenz mit anderen Funkstandards im selben Frequenzband. Da der Haupteinsatzbereich mobile Geräte sind, spielt auch niedriger Energieverbrauch eine Rolle. Datendurchsatz, Zuverlässigkeit der Verbindung und Quality of Service sind zweitrangige Ziele.

Ursprünglich wurde die Entwicklung von Bluetooth von der Firma Ericsson vorangetrieben. Der Standard sollte einen drahtlosen Ersatz für die serielle Schnittstelle RS-232 in Mobiltelefonen bieten und so beispielsweise eine Synchronisation von Telefonbuchkontakten und Terminkalendern über verschiedene Geräte hinweg ermöglichen. Nach einer noch nicht ausgereiften ersten Version wurde *Bluetooth v1.1* im Jahr 2002 als IEEE Standard 802.15.1 ratifiziert [14]. Aktuell wird der Standard von der *Bluetooth Special Interest Group (Bluetooth SIG)* und nicht mehr von IEEE gepflegt.

Mit *Bluetooth v1.2* [27] wurde das adaptive Frequenzsprungverfahren *Adaptive Frequency Hopping (AFH)* eingeführt, das die Koexistenz mit anderen Funksystemen, die feste Kanäle belegen, verbessert und Störungen vermeidet [29]. Beim Adaptive Frequency Hopping erkennen die Bluetooth-Geräte Kanäle, die von Störungen oder Interferenzen beeinträchtigt werden und verwendet eine Kanalmaske, um diese Kanäle nicht für Datenübertragungen zu nutzen. Diese Kanäle heißen *ungenutzte Kanäle*, ungestörte Kanäle, die für Datenübertragungen genutzt werden, heißen *genutzte Kanäle*.

Bluetooth v2.0 [28] erweitert den Standard um komplexere Modulationen, mit denen höhere Datenraten von bis zu 3 Megabit pro Sekunde erreicht werden können [29].

Bei Einsatz dieser Modulationen spricht man von *Enhanced Data Rate (EDR)*. Wird die Standardmodulation benutzt, ist eine Bruttodatenrate von einem Megabit pro Sekunde möglich und man spricht von *Basic Rate (BR)*.

Bluetooth v3.0 [30] erhöht die erreichbare Datenrate auf 24 Megabit pro Sekunde. Hierbei bauen, wie in vorhergehenden Standards, zwei Geräte über Bluetooth eine Verbindung auf. Die Nutzdaten jedoch werden über eine WLAN-Verbindung übertragen. Diese Übertragungsart ist laut Standard optional und nur dann möglich, wenn beide Geräte sowohl Bluetooth- als auch WLAN-Funktechnik integriert haben.

Die aktuelle Entwicklung *Bluetooth v4.0* [31] definiert einen nicht abwärtskompatiblen Protokollstapel *Bluetooth Low Energy (BLE)*, der durch kürzere Verbindungsprozeduren und weniger Kanalbenutzungen den Energiebedarf drastisch reduziert.

Im folgenden Abschnitt soll die Revision *Bluetooth v2.1* betrachtet werden. Diese Version war vor Bluetooth v4.0 am weitesten verbreitet und enthält alle wichtigen Mechanismen, die für eine passive Kanalmaskendetektion betrachtet werden müssen. Die in den folgenden Abschnitten dieses Kapitels erklärten Sachverhalte sind in der Spezifikation von Bluetooth v2.1 [29] definiert.

2.1.1 Physikalische Ebene

Bluetooth sendet im 2,4 GHz ISM Band. Es teilt dieses Frequenzband in 79 Kanäle, nummeriert von 0 bis 78, ein. Jeder Kanal ist 1 MHz breit. Der unterste Kanal 0 liegt bei 2402 MHz, der oberste Kanal 78 bei 2480 MHz. Da sich in diesem Frequenzbereich mehrere Bluetooth-Verbindungen, WLAN Netze, Sensornetze und Störer wie Mikrowellenherde befinden können, benutzt Bluetooth das Frequenzsprungverfahren *Frequency Hopping Spread Spectrum (FHSS)*, um den Einfluss gegenseitiger Störungen zu minimieren. Dabei verbleiben die Kommunikationspartner während der Verbindung nur für einen kurze Zeit auf einem Kanal. Danach springen sie gemeinsam auf einen anderen. So werden andere Funkgeräte wie WLAN Teilnehmer, die immer auf einem festen Kanal senden, nur für diese kurze Zeit gestört und die Wahrscheinlichkeit einer Paketkollision sinkt. Weil jede Bluetooth-Verbindung ein individuelles Sprungschema verwendet, ist die Wahrscheinlichkeit, dass sich zwei Bluetooth-Verbindungen stören, sehr gering.

Ab Revision Bluetooth v1.2 können Bluetooth-Verbindungen das adaptive Frequenzsprungverfahren *Adaptive Frequency Hopping (AFH)* verwenden. Bei Adaptive Frequency Hopping werden Kanäle, die von Störungen beeinträchtigt sind, nicht für das Hopping genutzt. Diese Kanäle heißen *ungenutzte Kanäle*. Alle Kanäle, die nicht von

einer anhaltenden Störung beeinträchtigt sind und für das Hopping genutzt werden, heißen *genutzte Kanäle*. Immer dann, wenn beide Kommunikationsteilnehmer normalerweise gemeinsam auf einen ungenutzten Kanal springen würden, springen sie beim Adaptive Frequency Hopping stattdessen auf einen genutzten Kanal. Die Kriterien dafür, wann ein Kanal genutzt wird und wann nicht, ist nicht im Standard festgelegt und somit abhängig von der jeweiligen Bluetooth-Implementierung. Die Information, welche der 79 Kanäle genutzt sind und welche ungenutzt, speichern Bluetooth-Geräte in der *Kanalmaske*. Die Kanalmaske ist ein 79 Elemente fassendes Bitfeld, das für jeden Kanal den jeweiligen Zustand speichert: Ist der Kanal genutzt, so ist das entsprechende Bit in der Kanalmaske gesetzt. Während eine Bluetooth-Verbindung besteht, haben alle Verbindungsteilnehmer eine gemeinsame, synchronisierte Kanalmaske.

Im Allgemeinen entspricht der Aufbau einer Bluetooth-Verbindung einer Sterntopologie, dem so genannten *Piconet*. Ein Piconet besteht aus einem Master und einer beliebigen Anzahl von Slaves. Sehr häufig ist jedoch nur ein Slave verbunden, so dass die Verbindung als Punkt-zu-Punkt Verbindung betrachtet werden kann. Verschiedene Piconets werden durch ihren *Access Code (AC)* unterschieden, der sich aus der Adresse des Masters ableitet. Zu jeder Zeit findet höchstens eine Kommunikation zwischen dem Master und einem Slave statt. Untereinander können die Slaves nicht direkt kommunizieren.

Bei der Verwendung der Basic Rate wird der Bitstrom mittels *Gaussian Frequency Shift Keying (GFSK)* bei einer Symbolrate von 1 Megasymbol pro Sekunde moduliert. Da bei GFSK pro Symbol ein Bit kodiert werden kann, ergibt sich eine Bruttobitrate von 1 Megabit pro Sekunde. Beim Einsatz von Enhanced Data Rate wird mit der gleichen Symbolrate gesendet, allerdings mit den Modulationen DQPSK oder 8-PSK mit 2, respektive 3 Bit pro Symbol. So ergibt sich eine maximale Übertragungsrate von 3 Megabit pro Sekunde.

Während der Verbindung verwendet Bluetooth eine Sprungfrequenz von 1600 Sprüngen pro Sekunde, verbleibt also immer für eine Slotdauer von 625 µs auf einem Kanal, respektive einer Frequenz. Innerhalb eines jeden Slots kann ein Bluetooth-Paket gesendet werden.

Für eine korrekte zeitliche Abstimmung der Sprünge hat jedes Bluetooth-Gerät seinen eigenen Referenztakt. Dieser Takt wird CLKN genannt und läuft mit einer Frequenz von 3200 Hz, der doppelten Sprungfrequenz. CLKN entspringt einem frei laufenden Oszillator, der eine maximale Frequenzabweichung von +/- 20 ppm nicht überschreiten darf [29, s. 231].

Weiterhin pflegt jedes Gerät eine Zählerstandsregister, das 28 Bit lange *CLK27–0*, das mit jeder Periode von *CLKN* inkrementiert wird. Die einzelnen Bits des Registers werden als Taktteiler eingesetzt. Das *Least Significant Bit (LSB) CLK0* schwingt mit 3200 Hz,

wechselt seinen Wert also alle 312,5 µs. Das nächste Bit *CLK1* wechselt alle 625 µs, hält also für die Dauer eines Zeitslots seinen Wert. Mit jedem Wechsel von CLK1 wird ein Sprung durchgeführt. Alle weiteren Bits wechseln ihren Wert mit der halben Frequenz ihres Vorgängerbits. Auf diese Weise werden Sprünge, Sende- und Empfangszeitpunkt und weitere Steuerungsaufgaben koordiniert.

Damit alle miteinander verbundenen Geräte zur gleichen Zeit springen, wird beim Verbindungsaufbau CLKN und der Zählerstand CLK27–0 zwischen allen Geräten eines Piconets synchronisiert.

2.1.2 Logische Ebene

Jedes Bluetooth-Gerät besitzt eine eigene 6 Byte lange *Adresse* ähnlich den MAC Adressen von Ethernet-Netzwerkgeräten. Sie gliedert sich in drei Teile: Den 3 Byte langen *Lower Address Part (LAP)*, den 1 Byte langen *Upper Address Part (UAP)* und den 2 Byte langen *Nonsignificant Address Part (NAP)*, der von sämtlichen Implementierungen ignoriert wird und somit für die Adressierung keine Rolle spielt. Die Adresse des Masters eines Piconets wird als die *Adresse des Piconets* bezeichnet.

Die Datenübertragung erfolgt bei Bluetooth paketorientiert. Die für die Übertragung verwendete Bitreihenfolge entspricht dem *Little Endian (LE)* Format. Das LSB wird stets zuerst übertragen. Abbildung 2.1a zeigt den typischen Aufbau eines Bluetooth-Pakets. Es beginnt mit einer 4 Bit langen Präambel, gefolgt vom 68 oder 72 Bit langen *Access Code (AC)*. Die Präambel ist abhängig vom ersten Bit (LSB) des Access Code. Hat dieses Bit den Wert 0, so lautet die Präambel 0101, andernfalls 1010. So beginnt jedes Bluetooth-Paket mit einer Folge von fünf wechselnden Bits. Der AC dient zur Adressierung des Pakets und wird aus dem LAP des Piconets gebildet. Auf den Access Code folgen der 54 Bit lange *Header* und schließlich ein *Payload* von variabler Länge (0 Bit bis 2745 Bit), der die Nutzdaten des Pakets enthält. Der Header dient zur Adressierung innerhalb des Piconets. Er beinhaltet ein 3 Bit breites Adressfeld, ein 4 Bit breites Feld für den Pakettyp, drei Flusskontrollbits und eine 8 Bit breite Prüfsumme.

Wird EDR verwendet, werden AC und Header weiterhin GFSK moduliert, danach erfolgt ein Wechsel zur PSK Modulation. Deshalb befinden sich, wie in Abbildung 2.1b gezeigt, zwischen Header und Payload zusätzlich ein Guard Intervall von 5 µs und eine Synchronisationssequenz, um durch den Modulationswechsel keine Daten zu verlieren. Zusätzlich befindet sich hinter dem Payload ein Trailer. Syncword, Payload und Trailer sind PSK moduliert.

Präambel	Access Code	Header	Payload
4 Bits	68 oder 72 Bits	54 Bits	0 – 2745 Bits

(a) Basic Rate

Präambel	Access Code	Header	Guard	Syncword	Payload	Trailer
4 Bits	68 oder 72 Bits	54 Bits	5 µs	11 µs	0 – 2745 Bits	2 µs

(b) Enhanced Data Rate

Abbildung 2.1: Format eines Bluetooth-Pakets bei verwendung von Basic Rate und Enhanced Data Rate

2.1.3 Sprungschema

Die Kanalsprünge erfolgen nach einem festen Sprungschema, dem *Hopping-Pattern*. Das Hopping-Pattern ist eine geordnete Liste der Kanälen, die nacheinander genutzt werden sollen. Es müssen vor allem zwei Anforderungen erfüllen:

- Es muss während der Verbindung sowohl dem Master als auch dem Slave bekannt sein, damit sich beide auf dem gleichen Kanal befinden und kommunizieren können.
- Es muss für verschiedene gleichzeitige Verbindungen unterschiedlich sein, sodass nebeneinander stattfindende Übertragungen auf verschiedenen Kanälen stattfinden und sich nicht gegenseitig stören.

Deshalb ist das Hopping-Pattern eine deterministische Pseudozufallszahlenfolge, die aus der Adresse des Masters berechnet wird. Da diese Information sowohl Master als auch Slaves einer Verbindung bekannt ist, können beide Partner unabhängig voneinander das gleiche Hopping-Pattern ausrechnen [29]. Da verschiedene Verbindungen immer eine unterschiedliche Adresse haben, ergeben sich hier immer unterschiedliche Hopping-Patterns. Der aktuelle Kanal, auf dem zu einem bestimmten Zeitpunkt gesendet wird, ist das CLK27–0. Folgenelement des Hopping-Patterns. Die Länge der Folge ist durch die Breite des CLK27–0 Registers bestimmt und beträgt 2^{28} [34]. Die Kanäle im Hopping Pattern sind gleichverteilt, sodass jeder Kanal mit etwa der gleichen Häufigkeit vorkommt [29].

Die Berechnung des nächsten zu benutzenden Kanals übernimmt der so genannte *Hop Selection Kernel*. Der Hop Selection Kernel ist ein in jedem Bluetooth-Gerät vorhandenes Rechenmodul, der mit verschiedenen Abschnitten aus UAP, LAP und CLK27–0

Additionen, XOR-Operationen, eine 5-Bit-Permutation und mehrere Modulo Operationen durchführt. So wird erreicht, dass sich schon bei einer kleinen Änderung der Eingangsparametern der berechnete Kanal vollkommen ändert [29]. Das so berechnete Sprungschema heißt *Basic Hopping-Pattern*. Es kommt bei Bluetooth-Verbindungen zum Einsatz, die kein Adaptive Frequency Hopping verwenden.

Wird hingegen AFH verwendet, ist der Hop Selection Kernel um einen Remapper erweitert [29]. Enthält das Basic Hopping-Pattern einen Kanal, der in der AFH Channel Map als unbenutzt markiert ist, bildet der Remapper ihn auf einen Kanal aus der Menge der benutzten Kanäle ab. Die Abbildung ist gleichmäßig über alle benutzten Kanäle verteilt, sodass auch hier jeder Kanal mit der gleichen Häufigkeit vorkommt. Sie hängt vom UAP des Piconets, sowie der Anzahl der genutzten Kanäle ab. So entsteht das *Adaptive Hopping-Pattern*.

Basic Hopping-Pattern und Adaptive Hopping-Pattern haben die gleiche Länge, da die unbenutzten Kanäle des Basic Hopping Patters nicht entfernt, sondern neu verteilt, also durch genutzte Kanäle ersetzt werden. Außerdem stimmen die genutzten Kanäle des Basic Hopping-Patterns mit denen des Adaptive Hopping-Patterns überein [29, s. 254].

2.1.4 Aufbau einer Verbindung

Bevor eine Verbindung besteht und Daten übertragen werden können, müssen sich beide Verbindungsteilnehmer zunächst finden und verbinden. Das geschieht in mehreren Schritten. Für jeden Schritt wird ein eigens dafür optimierter physikalischer Kanal genutzt [29].

Für das Auffinden von potentiellen, bisher unbekannten Bluetooth-Geräten in der Umgebung wird ein *Inquiry Scan* durchgeführt. Der dabei verwendete *Inquiry Scan Channel* beschreibt einen physikalischen Kanal mit einem für alle Geräte gemeinsamen langsamen Hopping-Pattern, das 32 Kanäle umfasst [29, s. 247]. Der Master durchläuft nacheinander alle gültigen Kanäle. Auf jeder Frequenz sendet er eine Anfrage, den *Inquiry Request*, und lauscht auf eine Antwort, die *Inquiry Response*. Der Kanal entspricht einer Punkt-zu-Punkt-Verbindung [29].

Der eigentliche Verbindungsaufbau findet beim *Page Request* statt. Wie auch der *Inquiry Scan Channel* beschreibt der *Page Request Channel* eine Punkt-zu-Punkt Verbindung. Allerdings ermöglichen die äußerst kurzen Nachrichten hier ein schnelleres Hopping-Schema mit 3200 Hops pro Sekunde [29]. Der *Page Request* läuft in Form eines Drei-Wege-Handshakes ab, der in Abbildung 2.2 dargestellt ist: Der Master sendet eine Anfrage, den *Page Request*. Der Slave antwortet mit der *Slave Page Response*. Sobald

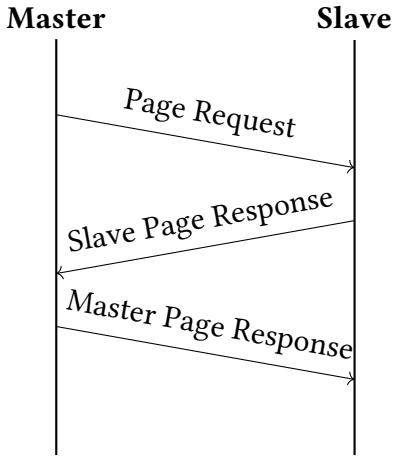


Abbildung 2.2: Handshake beim Page Request

Master diese empfängt, schickt er die *Master Page Response*. Danach ist die Verbindung aufgebaut und es können Verbindungsparameter ausgehandelt werden.

2.1.5 Änderungen von Verbindungsparametern

Alle Signalisierungen, die die Steuerung einer Verbindung betreffen, werden über das *Link Manager Protocol (LMP)* übertragen. Dabei handelt es sich um eine logische Protokollschicht, über die eine spezielle Art von Bluetooth-Paketen gesendet wird. Am Empfänger werden diese Pakete interpretiert, jedoch nicht an höhere Protokollschichten weitergereicht [29, s. 373].

LMP Pakete werden in DM1 Paketen gekapselt gesendet. Diese bestehen aus einem *OpCode*, in dem das durchzuführende Kommando kodiert ist, und einem Payload, der Parameter und zusätzliche Informationen zum Kommando enthält (s. Abbildung 2.3).

TID	OPC	Payload
1 Bit	7 Bits	0 Bytes bis 16 Bytes

Abbildung 2.3: Format eines LMP Pakets

2.1.6 Aktivieren von Adaptive Frequency Hopping

Auch die Steuerung von Adaptive Frequency Hopping erfolgt über LMP. Nach dem erfolgten Verbindungsaufbau ist AFH zunächst deaktiviert. Die Verbindungspartner folgen einem generischen Hopping-Pattern. Die Aktivierung von AFH geht in jedem Fall vom Master aus. Über LMP sendet der Master das in Abbildung 2.4 gezeigte LMP-Paket mit dem Kommando *LMP_set_AFH* (OpCode `0x3c`) und folgenden Parametern:

- **AFH_INSTANT (4 Bytes):**
  Der CLK27–0 Wert, ab dem AFH aktiviert sein soll.

- **AFHMODE (1 Byte)**:
  `0x01` für AFH aktivieren

- **AFH_CHANNEL_MAP (10 Bytes)**:
  Die AFH Channel Map, die ab der Aktivierung von AFH benutzt werden soll.

TID	OPC	AFH_INSTANT	AFHMODE	AFH_CHANNEL_MAP
1/0	0x2C	CLK27-0 bei Aktivierung	0x01: AFH aktivieren	AFH Channel Map
1 Bit	7 Bits	4 Bytes	1 Byte	10 Bytes

Abbildung 2.4: Format des LMP Pakets für die Aktivierung von Adaptive Frequency Hopping

Soll die AFH Channel Map geändert werden, wird das gleiche Paket mit dem aktualisierten Wert für AFH_CHANNEL_MAP noch einmal gesendet. Auch zum Deaktivieren von Adaptive Frequency Hopping kommt das gleiche Paket zum Einsatz, diesmal allerdings mit dem Parameter AFHMODE = `0x00`.

2.1.7 Relevanz für diese Arbeit

Im Hinblick auf das Ziel dieser Arbeit, die passive Detektion der AFH Channel Map, ergeben sich nach der Betrachtung des Bluetooth-Standards einige interessante Möglichkeiten und gewisse Einschränkungen.

Zunächst zeigt sich, dass Inquiry Scans und Page Requests uninteressant für die Kanalmaskenerkennung sind. Sie haben jeweils ein eigenes Hopping Schema, das sich

sowohl vom Basic Hopping-Pattern als auch vom Adaptive Hopping-Pattern unterscheidet. Es reicht also aus, den Fall zu betrachten, dass sich die beobachteten Geräte bereits miteinander verbunden haben.

Weiterhin ist das Hopping Schema deterministisch und unterscheidet sich unter gleichen Bedingungen zwischen verschiedenen Geräten nicht, da der Hop Selection Kernel in der Bluetooth-Spezifikation beschrieben und somit in allen Bluetooth-Geräten gleich aufgebaut ist. Er kann also auch in einem Sniffer implementiert werden. Das Hopping-Pattern ist nur von UAP und LAP abhängig. Diese Adressteile sind entweder bekannt oder man kann sie durch Scans herausfinden. Der aktuelle Kanal wird durch CLK27–0 gewählt. Dieser Taktzählerstand muss in Erfahrung gebracht werden.

Die Hopping-Patterns mit AFH und ohne AFH stimmen in den genutzten Kanälen überein. Außerdem treten sowohl im Basic Hopping Pattern als auch im Adaptive Hopping Pattern alle genutzten Kanäle mit der gleichen Häufigkeit auf.

Bei Änderung der AFH Channel Map wird diese per LMP versendet, ist also von einem Sniffer abfangbar. Das Paketformat ist bekannt. Ist es also möglich einen Großteil der gesendeten Pakete erfolgreich abzufangen, wird der Sniffer bei einer Aktualisierung der AFH Map mit großer Wahrscheinlichkeit die neue Map empfangen.

2.2 Projekt Ubertooth

Für eine passive Detektion der Kanalmaske wurde für diese Arbeit der Bluetooth-Sniffer *Ubertooth* verwendet. Ubertooth [38] ist eine Entwicklungsplattform für Funkprojekte im 2,4 GHz ISM Band, speziell Bluetooth. Es besteht aus einem Gerät im USB-Stick-Format und einer Software. Das Projekt entsprang einem Crowdfunding Projekt und sowohl Hardware als auch Software sind quelloffen. Die Software ist auf Github veröffentlicht [37].

Diese Arbeit basiert auf der aktuellen Hardware-Revision *Ubertooth One*. Abbildung 2.5 zeigt den schematischen Aufbau seiner Komponenten. Auf ihm übernimmt der Transceiverchip CC2400 [41] zusammen mit dem RF-Frontend CC2591 [42] die Funkverbindung. Beide Chips werden von Texas Instruments hergestellt. Zum Senden und Empfangen wird über eine RP-SMA Buchse eine passende Antenne, etwa eine WLAN-Antenne, angeschlossen. Die Steuerung übernimmt der ARM Cortex-M3 Mikrocontroller LPC175x [26] von NXP Semiconductors. Er verfügt über eine Full-Speed USB-Schnittstelle, über die er mit dem Hostrechner verbunden werden kann.

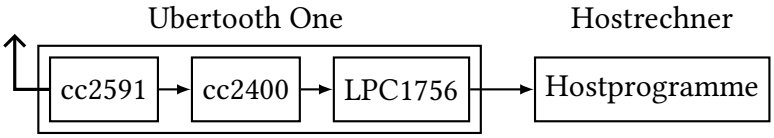


Abbildung 2.5: Komponenten des Ubertooth

Nach Angaben der Hauptentwickler Dominic Spill und Michael Ossmann [23] entstand das Projekt aus dem Bedürfnis, einen kostengünstigen Bluetooth-Sniffer zu entwickeln, der trotzdem über alle wichtigen Funktionen verfügt, wie das Entdecken von versteckten Geräten, das Bestimmen der Adresse des Piconets, das Erkennen des Taktzählerstandsregisters CLK27–0 und das Folgen der Verbindung entsprechend dem Sprungschema [34]. Wie in Abschnitt 1.1 erwähnt wurde, gibt es bereits mehrere Bluetooth-Sniffer auf dem Markt. Die meisten von ihnen verfügen allerdings nicht über alle der genannten Features. Ubertooth bietet eine kleine Abmessung, einen geringen Energieverbrauch, quelloffene und damit anpassbare Hard- und Software und einen niedriger Anschaffungspreis. Zurzeit ist er als Bausatz für 96 € zu kaufen.

Die Datenverarbeitung des Projekts ist dreigeteilt. Auf dem Mikrocontroller übernimmt die Firmware die Ansteuerung des Funkchips. Im Snifferbetrieb veranlasst sie die Kanalsprünge und überträgt empfangene Sequenzen per USB an den Hostrechner.

Auf dem Host-Computer sind je nach Einsatzzweck verschiedene Programme für die Kommunikation mit dem Ubertooth verantwortlich. Sie definieren im Allgemeinen eine Hauptschleife mit Handlerfunktion, die auf empfangene Daten reagieren. Gleichzeitig können Steuerkommandos an den Ubertooth gesendet werden, um beispielsweise seine Betriebsart zu ändern, Kanalsprünge zu veranlassen, Parameter zu setzen und Statusinformationen auszulesen.

Die eigentliche Interpretation von empfangenen Daten übernimmt die ebenfalls von den Ubertooth Entwicklern bereitgestellte *libbtbb (Bluetooth Baseband Library)* [36]. Sie erkennt Bluetooth-Pakete im Datenstrom, berechnet das Hopping-Pattern und stellt Datenstrukturen für Pakete und Piconets zur Verfügung.

Abbildung 2.6 zeigt die Funktionsblöcke des Datenverarbeitungspfades im Ubertooth Projekt. In den folgenden Abschnitten sollen kurz die für die Arbeit relevanten Funktionen der im Bild gezeigten Abschnitte erklärt werden.

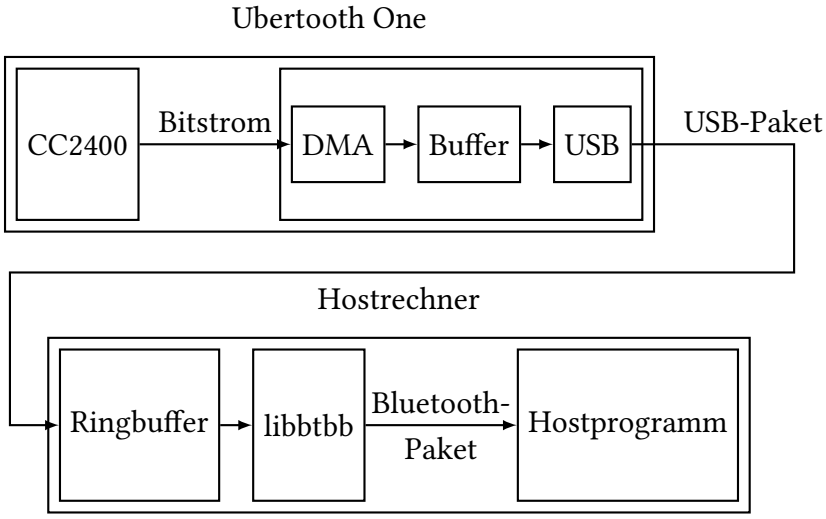


Abbildung 2.6: Datenverarbeitungspfad des Ubertooth

2.2.1 Mikrocontroller

Der Mikrocontroller kann in verschiedenen Betriebsarten arbeiten. Beispielsweise kann er als Sniffer für Bluetooth und Bluetooth Low Energy Bluetooth-Pakete empfangen, als Spectrum Analyzer die Signalstärke für verschiedene Frequenzbereiche messen oder ein Testsignal senden. Die für diese Arbeit interessante Betriebsart ist das Mitschneiden von Bluetooth-Verkehr. Dies kann auf zwei verschiedene Arten geschehen: Entweder es werden alle möglichen Kanäle willkürlich durchlaufen (*Sweep*) oder es werden direkt die Kanäle verwendet, die dem Hopping-Schema der Bluetooth-Verbindung entsprechen (*Follow*).

In beiden Modi versetzt der Mikrocontroller den cc2400 Transceiver auf den entsprechenden Kanal und wartet auf Ereignisse. Sobald der Transceiver eine Trägerfrequenz erkennt, beginnt er mit der Demodulation und überträgt die demodulierten Bits direkt per *Serial Peripheral Interface (SPI)* in einen Puffer im Speicher des Mikrocontrollers (Direct Memory Access — DMA).

Der Puffer ist zweiteilig gestaltet und funktioniert nach dem Double-Buffering-Prinzip [2]. In eine Hälfte wird per SPI ausschließlich geschrieben (*aktiver Teil*), während zur gleichen Zeit aus dem anderen vom Mikrocontroller ausschließlich gelesen werden kann (*passiver Teil*).

Jedes mal, wenn der aktive Teil des Puffers voll ist, wird per Interrupt umgeschaltet, die aktuelle Zeit im Zeitstempel des Puffers gespeichert und ein Flag gesetzt, das

das Hauptprogramm anweist, auf neue Daten zu reagieren. In einer entsprechenden Routine werden die Daten mit ihrem Zeitstempel und dem aktuellen Kanal in ein USB-Paket gekapselt und in den Sendepuffer des USB-Subsystems eingereiht, um an den Hostrechner gesendet zu werden.

Um Frequenzsprünge und Zeitstempel zeitgenau zu setzen, betreibt der Mikrocontroller einen Timer. Der Timer läuft mit einer Frequenz von 10 MHz, inkrementiert also sein *Timer Counter Register T0TC* alle 100 ns. Erreicht T0TC den Wert 3125, wird es sofort auf 0 zurückgesetzt und es wird ein Interrupt ausgelöst.

Dieses Interrupt findet folglich alle 312,5 µs statt, also der Periode von CLKN aus der Bluetooth-Spezifikation [29]. Mit jedem Interrupt wird ein 28 Bit langes Register inkrementiert. Dieses Register entspricht dem in Abschnitt 2.1 Zählerregister CLK27–0 der Bluetooth-Spezifikation. Die Interruptroutine löst ebenfalls Sprünge aus, abhängig von Betriebsart und Zählerregisterwert.

Um feingranularere Zeitstempel zu erhalten, wird ein zweites Register vorgehalten, das Register CLK100NS. Es speichert einen aktuellen Zeitstempel, gemessen in Einheiten von 100 ns. CLK100NS wird für die Zeitstempel der Bluetooth-Pakete genutzt. Sein Wert wird zusammengesetzt aus dem Timer Counter Register T0TC, sowie den unteren 20 Bit von CLK27–0, multipliziert mit 3125:

$$CLK100NS = (\text{CLK19–0}\ ) * 3125 + T0TC, \tag{2.1}$$

$$CLK100NS = (\text{CLK27–0}\ \wedge \mathtt{0xFFFFF}) * 3125 + T0TC. \tag{2.2}$$

2.2.2 Hostprogramme

Der Programmteil, der auf dem Hostrechner läuft, nutzt libusb [21], eine freie Softwarebibliothek zur Abstaktion von USB-Verbindungen, zur Kommunikation mit dem Ubertooth. Dabei kommen zwei logische USB-Kanäle zum Einsatz: Der *Bulk Transfer Channel* zum Übertragen der Bluetooth-Paketdaten vom Stick an der Rechner und der *Control Transfer Channel* zum Senden von Kommandos an den Ubertooth.

Zunächst wird der Ubertooth über einen entsprechenden Control Transfer initialisiert. Danach prüft die Hauptschleife periodisch, ob neue USB-Pakete im USB-Empfangspuffer vorliegen. Wird ein USB-Paket empfangen, wird es zunächst in einen 10 Pakete fassenden Ringbuffer eingeordnet. Die Pakete bestehen aus einem 50 Byte langen Bitstrom, den der Funkchip aus dem Empfangssignal demoduliert hat.

Wie in Abbildung 2.7 gezeigt, kann sich im Bitstrom an einer unbekannten Stelle ein Bluetooth-Paket befinden. Da weder Funkchip noch Mikrocontroller eine Überprüfung der Empfangsdaten durchführen, kann es sich dabei um Bluetooth-Pakete des gewünschten Piconets, eines anderen Piconets in der Umgebung oder sogar um Funksignale handeln, die nicht von Bluetooth herrühren sondern aus Versehen vom cc2400 als Funkträger erkannt wurden. Deshalb muss zunächst ein Bluetooth-Paket im Bitstrom gesucht werden.

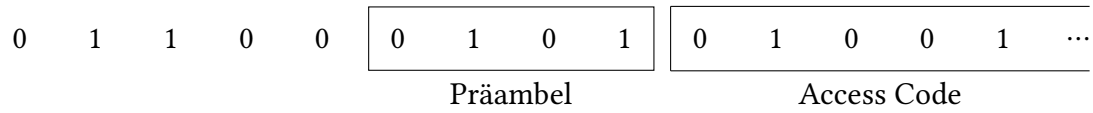


Abbildung 2.7: Im Bitstrom befindet sich ein Bluetoothpaket, dessen Präambel mit dem sechsten Bit des Bitstroms startet.

Das übernimmt eine Funktion der libbtbb [36]. Da Bluetooth-Pakete, wie in Abschnitt 2.1 beschrieben, immer mit einer Präambel und dem Access Code beginnen, kann der Bitstrom auf dieses Muster durchsucht werden. Libbtbb ist dabei sogar imstande, bis zu drei Bitfehler zu korrigieren. War die Suche erfolgreich, wird die Position des Paketanfangs im Bitstrom zurückgegeben.

Alle bisher empfangenen Daten im Ringbuffer werden vom Hostprogramm ab dem Paketanfang in eine Datenstruktur geschrieben, die ein Bluetooth-Paket repräsentiert. Diese Datenstruktur wird ebenfalls durch die libbtbb bereitgestellt.

2.2.3 libbtbb

Mit diesem Bluetooth-Paket kann nun auf verschiedene Art und Weise verfahren werden. Zunächst bietet libbtbb eine Funktion zum Dekodieren des Pakets. Dabei wird der Pakettyp festgestellt und versucht, seinen Payload zu extrahieren und auszugeben. Diese Funktion kann genutzt werden, um direkt Informationen aus Paketheader oder -inhalt auszugeben.

Die zweite interessante Methode, die libbtbb bereitstellt, kann das Hopping-Pattern des Piconets bestimmen und somit Sprünge vorhersagen. Dazu ist der gesamte, in Abschnitt 2.1.3 beschriebene Hop Selection Kernel aus der Bluetooth-Spezifikation in der Bibliothek implementiert. Anhand des UAP und LAP wird für jeden möglichen Wert von CLK27–0 zunächst die Folge aller 2^{28} Sprünge berechnet und gespeichert. Diese Folge ist das Hopping-Pattern des Piconets, das ebenfalls in Abschnitt 2.1.3 vorgestellt

wurde. Der Kanal des ersten verarbeiteten Pakets liefert mögliche Startpunkte: Alle Positionen im berechneten Hopping-Pattern, deren Kanal mit dem des Pakets übereinstimmen, kommen als Startpunkt in Frage. Von jedem weiteren empfangenen Paket wird der zeitliche Abstand vom ersten Paket berechnet. Zusammen mit dem Kanal des Pakets dient der zeitliche Abstand dazu, weitere mögliche Startpunkte im Hopping-Pattern auszusortieren. Somit bleiben mit jedem verarbeiteten Paket immer weniger mögliche Startpunkte übrig, bis letztendlich nur noch ein einziger in Frage kommt. Die Position des Startpunktes im berechneten Hopping-Pattern entspricht dem tatsächlichen Wert von CLK27–0 des ersten Pakets. Die Differenz zwischen diesem tatsächlichen Wert und dem Zeitstempel des empfangenen Pakets liefert das Taktoffset, die Abweichung zwischen CLK27–0 des Ubertooths und des Bluetooth-Piconets. Ist dieses Taktoffset berechnet, kann das Hostprogramm den Ubertooth anweisen, dem Piconet zu folgen, also das gleiche Sprungschema zu verwenden, wie das beobachtete Piconet. Das geschieht über den USB-Control-Channel unter Übergabe des Taktoffsets zwischen Sniffer und Piconet und der Adresse des Piconets.

Abbildung 2.8 verdeutlicht diese Prozedur anhand eines Beispiels. Das Hopping-Pattern wurde bereits für den UAP und LAP des Piconets berechnet. Jedem möglichen Wert für CLK27–0 ist ein Kanal zugeordnet. Die Abbildung zeigt einen Ausschnitt des Hopping-Patterns unter den jeweiligen CLK27–0 Werten in den oberen Zeilen. Die dritte Zeile zeigt die empfangene Paketfolge. Zum Zeitpunkt t_0 empfängt der Sniffer zunächst ein Paket auf Kanal 1. Mögliche Werte von CLK27–0 für dieses Paket sind in diesem Ausschnitt sowohl 0 als auch 6. Nun wird das zweite Paket zum Zeitpunkt $t_0 + 4$, also 4 Slots nach dem ersten Paket, auf Kanal 56 empfangen. Als möglicher CLK27–0 Wert für das erste Paket scheidet 6 nun aus, da zum Zeitpunkt $6 + 4$ laut Hopping-Pattern ein Paket auf Kanal 41 empfangen werden müsste. Zum Zeitpunkt $0 + 4$ hingegen stimmt der Kanal des berechneten Hopping-Patterns mit dem des empfangenen Pakets überein. Somit ist 0 weiterhin ein gültiger Kandidat für den CLK27–0 Wert des ersten empfangenen Pakets. Da die Abbildung nur einen kleinen Ausschnitt des Hopping-Patterns betrachtet, passt die bisher empfangene Paketfolge wahrscheinlich auf weitere Werte von CLK27–0. Diese müssen durch die Beobachtung von weiteren empfangenen Paketen sukzessiv aussortiert werden.

CLK27-0 offset	0	1	2	3	4	5	6	7	8	9	10
Hopping Pattern	1	27	5	34	56	13	1	78	63	9	41
Empfangene Pakete	1	x	x	x	56		1	x	x	x	56

Abbildung 2.8: Zwei mögliche Werte für CLK27–0 für eine Folge von empfangenen Paketen. Der Wert 6 scheidet nach Empfang des zweiten Pakets auf Kanal 56 aus.

3 Festlegung zu betrachtender Rahmenbedingungen

Ziel der Arbeit ist es, Vorgehensweisen zu entwickeln und zu untersuchen, mit denen ein passiver Bluetooth-Sniffer Kenntnis der Kanalmaske einer Bluetooth-Verbindung erlangen soll. In diesem Kapitel sollen Rahmenbedingungen festgelegt und begründet werden, unter denen die Möglichkeiten einer Kanalmaskendetektion untersucht werden sollen.

Adaptive Frequency Hopping wurde mit Bluetooth v1.2 eingeführt und besteht seitdem weitgehend unverändert. Diese Version ist mittlerweile von der Bluetooth SIG als veraltet (deprecated) eingestuft. Daher soll in dieser Arbeit die aktive Version Bluetooth v2.1 betrachtet werden.

Weiterhin wird ausschließlich die Übertragung mittels Basic Rate betrachtet, nicht Enhanced Data Rate. Das hat den Grund, dass der Ubertooth-Sniffer nur die Modulation GFSK beherrscht, nicht die bei EDR eingesetzten Verfahren DQPSK und 8PSK. Die Übertragungsverfahren unterscheiden sich vor allem durch die im Paketpayload verwendete Modulation; auf den Hoppingvorgang hat EDR keinen Einfluss. Alle Erkenntnisse dieser Arbeit lassen sich auch auf spätere Bluetooth-Revisionen anwenden, in denen AFH benutzt wird. Letztendlich wird mit dem entwickelten Programm auch die Kanalmaske von EDR Verbindungen detektierbar sein, weil dafür nur der Paketheader benötigt wird, der unabhängig von EDR immer mit GFSK moduliert übertragen wird.

Auch *Bluetooth Low Energy (BLE)* benutzt AFH und geht dabei prinzipiell auf die gleiche Art vor wie frühere Bluetooth-Revisiones. Da eine Kanalmaskendetektion von Bluetooth Low Energy nicht Ziel dieser Arbeit ist, soll BLE hier nicht betrachtet werden. Gleichwohl lassen sich die Erkenntnisse dieser Arbeit auf BLE anwenden, sodass ein entsprechender Algorithmus leicht zu implementieren sein sollte. Ein dahingehender Ansatz für weiterführende Arbeiten wird in Kapitel 7 angesprochen.

Die in dieser Arbeit verwendeten Geräte verwenden die Bluetooth-Implementierung BlueZ [24], den Bluetooth-Stack des Linux-Kernels. Dieser ist einerseits aufgrund der

weiten Verbreitung von Linux im Embedded-Bereich und auf Mobiltelefonen (Android) häufig anzutreffen. Zum anderen bietet er eine einfache Möglichkeit, per Software die aktuelle AFH Channel Map abzufragen. So lässt sich die passiv detektierte AFH Channel Map leicht mit der tatsächlich genutzten vergleichen.

Für den Algorithmus zur Detektion und Beobachtung der AFH Channel Map soll angenommen werden, dass die beobachteten Verbindungsteilnehmer zu jeder betrachteten Zeit verbunden sind. Verbindungsaufbau, -trennung und -unterbrechungen sollen nicht betrachtet werden. Diese Einschränkung kann problemlos in Kauf genommen werden, wenn der angestrebte Einsatzbereich des fertigen Algorithmus betrachtet wird. Das fertige Gerät soll eingesetzt werden, um einen bestehenden Funknetzaufbau im Betrieb auf Verbindungsprobleme zu untersuchen. Zum Zeitpunkt der Funkanalyse bestehen die Verbindungen also bereits und werden bis zum Ende der Messung im Regelfall nicht unterbrochen. Selbst wenn eine Unterbrechung auftritt, sind die Übergangszustände nur von kurzer Dauer.

Die Kanalmaskendetektion soll passiv durchgeführt werden. Der Sniffer soll also keine Verbindung zu dem beobachteten Piconet eingehen und dieses auch nicht manipulieren. Da die Kanalmaske nur innerhalb des Piconets bekanntgegeben wird, muss der Sniffer sie also aus Beobachtungen schätzen.

Um die AFH Channel Map allein aus Beobachtungen erfolgreich zu detektieren, muss vorausgesetzt werden, dass diese über einen hinreichend großen Zeitraum konstant bleibt. Wie groß dieser Zeitraum sein muss und unter welchen Umgebungsbedingungen das erfüllt ist, wird im Vorversuch in Kapitel 4 untersucht. Unabhängig davon kommen in der Industriekommunikation häufig Geräte mit konstanten, konfigurierbaren Kanalmasken zum Einsatz. Hier ändert sich die Kanalmaske unabhängig von äußeren Störungen nicht, sodass unvorhersehbare Kanalbelegungen ausgeschlossen werden können. Als Beispiel für ein solches Gerät sei der Bluetooth-Ethernet-Umsetzer *FL BT EPA* [19] von Phoenix Contact genannt, der eine Ethernet-Verbindung über Bluetooth tunneln kann.

4 Vorbetrachtung: Untersuchung von Einflüssen verschiedener Störquellen auf die Kanalmaske

In einem Vorversuch soll zunächst untersucht werden, inwiefern die Kanalmaske unterschiedlich auf den Einfluss verschiedener konkurrierender Funksysteme reagiert. Das folgende Kapitel schildert zunächst den Messaufbau. Nach einer Beschreibung der durchgeführten Messungen und deren Ergebnissen werden in einer Auswertung Erkenntnisse gesammelt, die Aufschluss darüber geben, unter welchen Bedingungen eine passive Detektion der Kanalmaske möglich ist und welche Strategien dafür sinnvoll sind.

4.1 Messaufbau

Da die AFH Channel Map ständig aktualisiert wird, ist sie stark abhängig von der weiteren Belegung des 2,4 GHz Bandes. In mehreren Messungen soll ermittelt werden, wie die Kanalmaske auf verschiedene Störer reagiert. Um möglichst definierte Messbedingungen zu haben, werden alle zusätzlichen vorhandenen Störer beseitigt, indem der gesamte Messaufbau im Keller des Instituts aufgestellt wird. Wie in Abbildung 4.1 eine Messung mit einem Spectrum Analyzer[1] zeigt, sind durch die Stahlbetonwände die Signale sämtlicher Funkteilnehmer im Gebäude hinreichend stark gedämpft, sodass sie nicht mehr vom Rauschen unterscheidbar sind.

Ursprünglich war geplant, die Messung in einem 2×3 Meter großen, funkabgeschirmten Zelt durchzuführen. Dessen Zeltwände reflektieren jedoch so stark, dass sich im Inneren Stehwellen bilden. Abbildung 4.2 zeigt das Ergebnis einer Messung mithilfe eines Ettus Research B200 Software Defined Radio. Zum Zeitpunkt 0 im Bild wurde ein Impuls gesendet. Auf ihn folgen gut sichtbare Stehwellen im Abstand von etwa 60

[1] Agilent N9342c Spectrum Analyzer

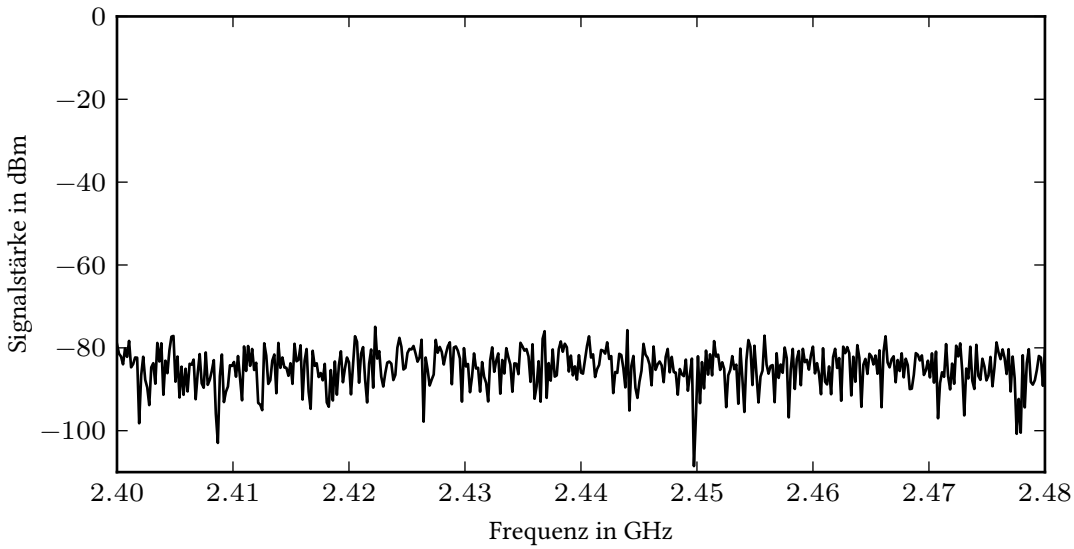


Abbildung 4.1: Im gemessenen Funkspektrum sind keine Funkstörer sichtbar.

µs. Diese Reflexionen verursachen, dass Bluetooth-Geräte ihre eigenen Übertragungen als Störung wahrnehmen. Das macht eine Untersuchung von den Einflüssen äußerer Störer auf die Bluetooth-Verbindung unmöglich. Auch ein Aufstellen von Dämpferelementen brachte keine hinreichende Verbesserung, sodass das Zelt als Messumgebung ausscheidet.

Als Störer soll eine WLAN-Verbindung gemäß 802.11n [13] (2,4 GHz, 20 MHz Bandbreite), mehrere Bluetooth-Verbindungen, sowohl mit als auch ohne Adaptive Frequency Hopping, sowie ein IEEE 802.15.4-Sender mit 1 MHz Bandbreite zum Einsatz kommen.

Als WLAN-Störer fungieren drei *WLAN-Knoten* vom Typ TP Link MR3020. Weiterhin befinden sich zwei *Raspberry Pi* Kleincomputer als Bluetooth-Störer im Messaufbau. Beide verfügen über einen Bluetooth-Dongel[2], über den sie miteinander eine Netzwerkverbindung über das *Bluetooth Network Encapsulation Protocol (BNEP)* [8] aufbauen. Ein weiterer Bluetooth-Störer ist das *Tastaturmodul* Bluefruit EZ Key [17] zum Einsatz. Dabei handelt es sich um ein Modul für Bastler, mit dem ein Bluetooth-*Human Interface Device (HID)* kompatibles Gerät gebaut werden kann. Es nimmt unter anderem Befehle über eine serielle Schnittstelle entgegen und sendet daraufhin HID-Kommandos per Bluetooth an einen verbundenen Rechner. Ein dritter Bluetooth-Störer ist ein *Bluetooth-Audio-Modul*, das über das *Advanced Audio Distribution Profile (A2DP)* [12] per Bluetooth Audio-Übertragungen entgegen nimmt, decodiert und über

[2]Broadcom Corp. BCM20702A0 Bluetooth 4.0

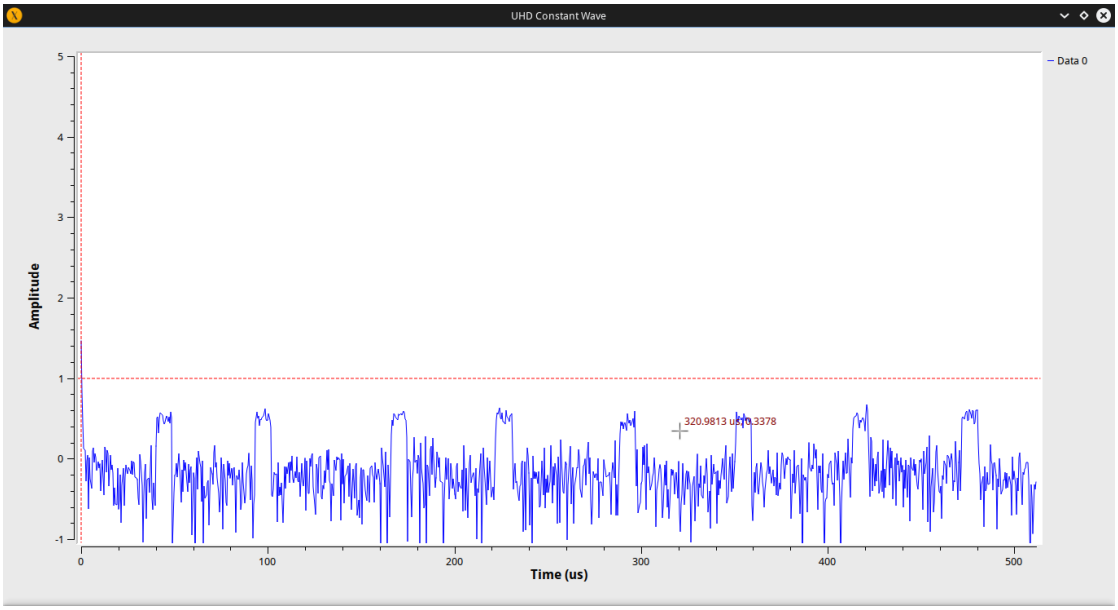


Abbildung 4.2: Im funkabgeschirmten Zelt bilden sich durch starke Reflexionen Stehwellen aus, die eine Bluetooth-Verbindung stören.

eine Klinkenbuchse ausgibt. Das Audio-Modul unterstützt als einziger Bluetooth-Störer kein Adaptive Frequency Hopping. Der *IEEE 802.15.4-Störer* wird mit einem ein Bluetooth-Dongel mit einem CSR BlueCore02 Chipsatz nachgebildet. Dieser Chipsatz kann im Debugmodus ein konstantes Testsignal auf einem Bluetooth-Kanal mit 1 MHz Bandbreite senden.

Die drei WLAN-Knoten und die beiden Raspberry Pis sind per Ethernet über einen Switch verbunden. So können sie über das Netzwerk kontrolliert werden, sodass Messungen bequem von einem anderen Raum aus durchgeführt werden können. Über eine Kabelpeitsche und ein Industrienetzteil werden alle Geräte gleichzeitig über das Netzwerkkabel mit Energie versorgt, um Kabel einzusparen und eine gewisse Flexibilität im Versuchsaufbau zu ermöglichen. Alle fünf Geräte befinden sich in einem eigens dafür eingerichteten VLAN des Institutsnetzes und haben zur besseren Identifikation statische IP Adressen. Abbildung 4.3 zeigt den gesamten Messaufbau.

Der erste WLAN-Knoten arbeitet als Access Point und spannt ein offenes WLAN auf, mit dem sich die anderen verbinden. Dabei sorgt ein MAC-Filter dafür, dass sich nur die vorgesehenen Testgeräte mit dem Netz verbinden können. Die WLAN-Verbindung wird mit einem für diesen Zweck entwickelten Traffic-Generator belastet.

Die Raspberry Pis bauen unter Verwendung des *Bluetooth Network Encapsulation Protocol (BNEP)* eine Bluetooth-Verbindung auf, durch die eine Ethernet-Verbindung ge-

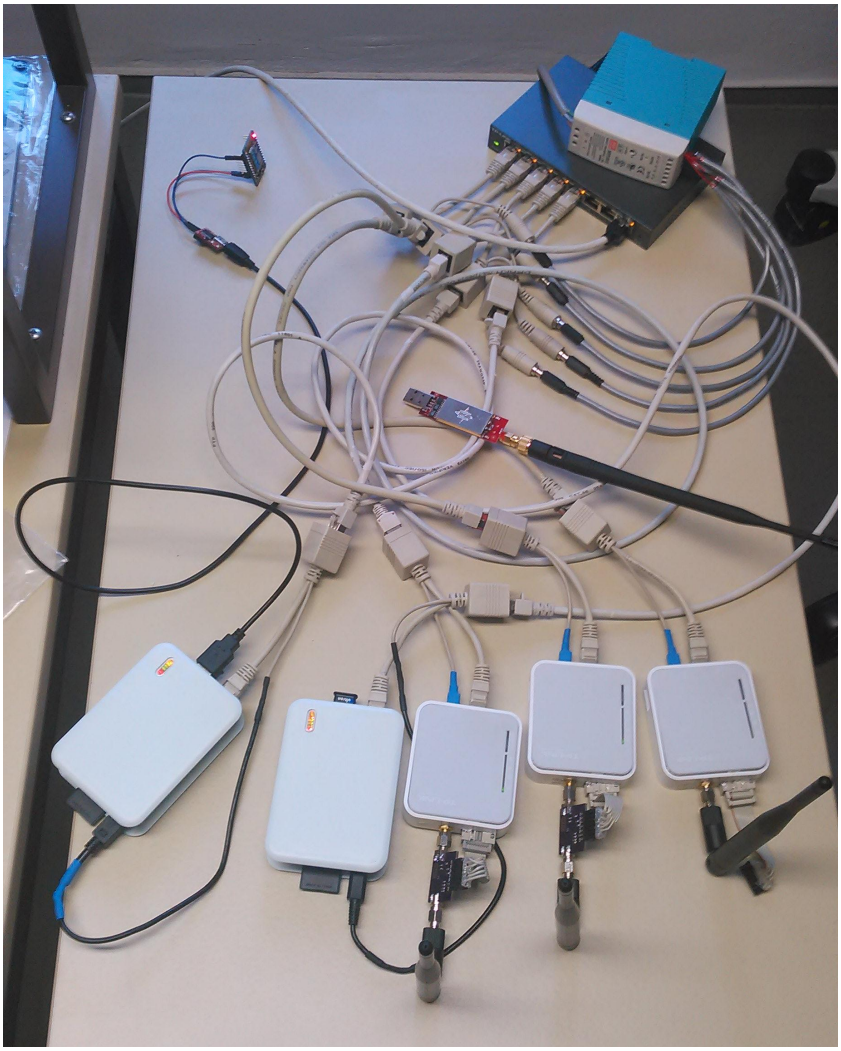


Abbildung 4.3: Der gesamte Testaufbau besteht aus drei WLAN-Knoten, zwei Raspberry Pis, Ethernet-Switch und Netzteil.

tunnelt wird. Das erste Raspberry Pi übernimmt dabei die Rolle des *Network Access Point (NAP)*, die vergleichbar mit der Rolle des Access Points in WLAN-Netzen ist. Das zweite agiert als *Personal Area Network User (PANU)* und übernimmt die Clientrolle. Auch diese virtuelle Netzwerkverbindung wird mit dem *Traffic-Generator* ausgelastet.

Mit der seriellen Schnittstelle des ersten Raspberry Pis ist das Tastaturmodul verbunden. So können über die serielle Schnittstelle zeitgesteuert Tastendrücke ausgelöst werden.

Das Bluetooth-Audio-Modul verbindet sich über das Advanced Audio Distribution Profile mit dem ersten Raspberry Pi und empfängt von ihm Audiodaten.

Der IEEE 802.15.4-Störer ist ebenfalls mit dem ersten Raspberry Pi verbunden und wird über USB gesteuert.

4.1.1 Einrichtung der WLAN-Knoten

Um die Messungen zu erleichtern, wurden an den WLAN-Knoten einige Modifikationen vorgenommen. Da auf ihnen ein Traffic-Generator laufen soll, um automatisiert Störfälle ablaufen zu lassen, wurde die ursprüngliche Firmware durch OpenWRT [39] ersetzt, eine Linux Distribution speziell für Router. Ferner wurde der Flashspeicher, der das Betriebssystem enthält, ausgewechselt, sodass nun 16 MB statt ursprünglich 4 MB an Speicherplatz bereitstehen.

Diese Modifikation macht den Einsatz einer modifizierten Version des Bootloaders U-Boot notwendig, da die ursprüngliche Version den größeren Flash nicht erkennt [6]. Die modifizierte Version stellt sogar ein Webinterface zur Verfügung, über das ein Firmware Image in den Flash geschrieben werden kann [7]. U-Boot selbst musste mit einem Flash Programmer geschrieben werden. Hierfür kam ein Programmer mit FT232H [22] Chip zum Einsatz. Das Flashen funktionierte besonders bei großen Images nur unzuverlässig. Deshalb wurde nur der Bootloader mit dem Programmer in der Flash geschrieben und die Firmware über dessen Webinterface geflasht.

Um auch im kleinen Versuchsaufbau zu simulieren, dass die Geräte entfernt voneinander stehen, wurden die WLAN Antennen durch ein vorgeschaltetes Dämpfungsglied ergänzt. Dazu mussten zunächst die Microstrip Antennen von der Platine abgesägt und stattdessen eine SMA Buchse aufgelötet werden. So kann eine externe Antenne wahlweise direkt oder mit dem Dämpfungsglied angeschlossen werden. Das Dämpfungsglied basiert auf dem Step Attenuator SKY12347 [32] und dämpft im Bereich von DC bis 4 GHz um 0 dB bis zu 31,5 dB, jeweils einstellbar in Schritten von 0,5 dB. Die

Dämpfungseinstellung wird vom angeschlossenen WLAN-Knoten per SPI vorgenommen. Der WLAN-Knoten agiert hierbei als SPI Master, der Dämpfer als Slave. Aufgrund der langen Anschlusskabel wurde für den Bittakt eine moderate Frequenz von 1 kHz gewählt.

Der Dämpferchip SKY12347 nutzt eine Wortlänge von 6 Bit. Dabei wird das höchstwertige Bit zuerst übertragen. Das Datenwort entspricht einer invertierten 5,1-Festkommazahl. Somit können Werte von 0,0 dB bis 31,5 dB dargestellt werden.

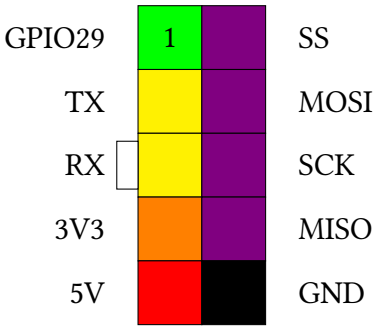


Abbildung 4.4: Belegung des Wannensteckers am WLAN-Knoten

Um das Dämpfungsglied per SPI mit dem WLAN-Knoten zu verbinden, wurden dazu nötige Schnittstellen am WLAN-Knoten nach außen geführt. LEDs auf der Platine, die softwareseitig als *General-Purpose Input/Output (GPIO)* angesteuert werden, wurden entfernt. Die so frei gewordenen GPIOs wurden mit einem angelöteten Kabel und einem Wannenstecker nach außen gelegt. Abbildung 4.4 zeigt die Pinbelegung des Wannensteckers. Anschlüsse, die für SPI verantwortlich sind, wurden violett eingezeichnet. GPIO29 ist ein bisher nicht genutzter Anschluss, der zukünftig zum Ansteuern von etwaigen zusätzlichen Geräten genutzt werden kann. Zusätzlich wurden neben Spannungsversorgung und Masse auch Sendepin (TX) und Empfangspin (RX) der seriellen RS-232-Schnittstelledes Routers nach außen geführt. Dämpfungsglied und WLAN-Knoten werden mit einem Flachkabel verbunden.

Das Firmwareimage für OpenWRT wurde selbst erstellt, um feinere Einstellmöglichkeiten zu haben. Dabei kam die zurzeit stabile Version *Barrier Braker* zum Einsatz. Zunächst musste im entsprechenden Makefile die Größe des Flash Speichers von 4 MB auf 16 MB angepasst werden, damit das Erstellen des Images nicht mit einer Fehlermeldung abbricht, die sich über eine zu große Imagegröße beschwert. Das geschieht in `./target/linux/ar71xx/image/Makefile`, wo in der entsprechenden Zeile für den TLMR3020 mtdsize auf 16M gesetzt wird.

Der Linux Kernel, der auf dem WLAN-Knoten zu Einsatz kommt, hatte ursprünglich den Treiber für LEDs fest eingebaut. Das hat zur Folge, dass alle GPIOs beim System-

start als LED konfiguriert werden und nicht mehr für andere Ansteuerungen zur Verfügung stehen. Da die GPIOs aber als SPI Port genutzt werden sollen, wurde der Treiber `leds-gpio` als Modul kompiliert, das jederzeit entladen werden kann. Gleichzeitig wurde der Treiber `spi-gpio` und `spi-gpio-custom`, der vorher nicht im Kernel enthalten war, als Modul übersetzt. Das Modul wird bei jedem Systemstart mit folgendem Befehl geladen:

```
modprobe spi-gpio-custom bus0=1,0,17,27,0,10000,7.
```

Die Parameter des Befehls erklären sich wie folgt: Der erste Parameten mit dem Wert 1 ist die ID des Geräts. Darauf folgen die GPIO-Pinnummern der SPI-Pins SCK (0), MOSI (17) und MISO (27). Der nächste Parameter legt den SPI Mode fest. Der Wert 0 Konfiguriert die Schnittstelle mit positiver Polarität und ohne Phasenversatz. Es folgen die Taktfrequenz von 10000 Hz und schließlich die GPIO-Pinnummer des CS Pins.

Zum komfortablen Einstellung der Dämpfung wurde das Programm `libattenuate` geschrieben. Es nimmt als Parameter die gewünschte Dämpfung, sowie das SPI Gerät, an das der Dämpfer angeschlossen ist, entgegen, berechnet das der Dämpfung entsprechende Datenwort und sendet es per SPI an den Dämpfer.

Das Firmwareimage soll das Dateisystem *Journaling Flash File System (JFFS)* nutzen. Dies ist ein Dateisystem, das speziell für SPI Flash konzipiert ist und Lese- und Schreibzugriff unterstützt. Normalerweise baut OpenWRT sein Firmwareimage zweiteilig: Eine readonly Partition enthält den Kernel und alle Programme, eine zweite read/write Partition enthält die Konfigurationsdateien. Letztere kann gelöscht werden, um den Router in den Werkszustand zurückzusetzen. Erstere ist schreibgeschützt, sodass immer ein funktionierendes Betriebssystem bereitsteht. Das erhöht die Ausfallsicherheit im Betrieb, hat aber einen Nachteil: Sämtliche Programme auf der Systempartition sind nicht überschreibbar. Soll eine neue Version installiert werden, muss sie auf die schreibbare Partition geschrieben werden, während die alte Version weiterhin auf der nur lesbaren Partition existiert. Das verschwendet wertvollen Speicherplatz. Weiterhin ist für die Messzwecke ein Zurücksetzen auf einen Werkszustand nicht notwendig. Daher wurde beim Erstellen des Images der Imagetyp manuell auf JFFS gesetzt, sodass nur eine read/write Partition angelegt wird, die sowohl Betriebssystem als auch Konfigurationsdateien enthält.

Um Stör- und Messprozesse auf verschiedenen Geräten synchron zu starten, wurde die Systemzeit der WLAN-Knoten und Raspberry Pis über das *Precision Time Protocol (PTP)* synchronisiert. Dieses Protokoll ermöglicht eine Zeitsynchronisation über das Netzwerk mit einer Genauigkeit von etwa einer Mikrosekunde. Die dafür nötige Software `ptpd2` ist nicht in den Quellen von OpenWRT enthalten und musste selbst gebaut werden.

Zum Erstellen von Softwarepaketen stellt OpenWRT viele Hilfsmittel bereit, die den Aufwand für den Entwickler minimieren. Letztendlich muss man ein Makefile für das Paket erstellen, das die Paketquellen herunterlädt, die Toolchain für die Zielplattform konfiguriert, dann das Makefile des Programms selbst ausführt und die übersetzten Programmdateien ins Paketverzeichnis installiert.

Insgesamt wurden vier Pakete erstellt: ptpd2, der Traffic-Generator und seine Abhängigkeit libtins, eine Packet Crafting Bibliothek, sowie die libattenuate.

4.1.2 Traffic-Generator

Der Traffic-Generator ist ein C++ Programm zum Erzeugen von synthetischem Netzwerkverkehr. Er wird bei den Messungen dazu benutzt, die Auslastung von drahtlosen Netzwerkverbindungen zwischen den Funkstörern präzise einzustellen und so verschiedene Störintensitäten zu realisieren. Zu diesem Zweck unterstützt er verschiedene Betriebsmodi, die verschiedene Netzwerkverkehrsmuster verursachen. Der Traffic-Generator benutzt die libtins [9] zum Erzeugen, Versenden, Empfangen und Aufzeichnen von Netzwerkpaketen. Er besteht aus drei Komponenten — Sender, Empfänger und Konfigurator — sowie drei zusätzlichen Programmen für Testzwecke.

4.1.2.1 Sender

Die Senderkomponente versendet TCP Pakete an Port 8088 des Zielrechners. Zusätzlich wird jedes versendete Paket in eine PCAP Datei nach `/tmp/sender.pcap` geschrieben. Die Senderkomponente kennt drei Modi:

- **Burst:** Sende Pakete so schnell es geht für eine festgelegte Dauer
- **Flood:** Sende Pakete mit einer festgelegten Rate für eine festgelegte Dauer
- **Stochastic:** Sende Pakete mit einer Rate, die durch einen modifizierten Galton-Watson-Prozess [43] bestimmt wird.

4.1.2.2 Galton-Watson-Prozess

Im *Stochastic* Modus sendet der Paketgenerator mit einer Paketrate, die von einem modifizierten Galton-Watson-Prozess gesteuert wird. Der Galton-Watson-Prozess ist ein getakteter Markow-Prozess [16]. Das bedeutet, dass die Senderate eines Taktes eine Zufallsgröße ist, die von der Senderate des vorhergehenden Taktes abhängt. Die hier verwendete Taktdauer beträgt 1 ms. Der Prozess wird mit folgenden Parametern konfiguriert:

- Startwert N $(N > 0)$,
- Reproduktionsrate P_{rep} $(0 < P_{rep} < 1)$,
- Generationsrate P_{gen} $(0 < P_{gen} < 1)$.

Der Prozess startet im ersten Takt mit einer Senderate von N Paketen pro Sekunde. In jedem neuen Takt hat jedes einzelne Paket die Möglichkeit, sich zu reproduzieren. Dabei wird für jedes Paket die Anzahl der Pakete, die es im nächsten Takt generiert, mit einem Poisson-Prozess [16] mit Erwartungswert P_{rep} generiert. Ist die Reproduktionsrate $P_{rep} < 1$, so wird der Paketstrom nach einigen Takten zum Erliegen kommen, andernfalls wächst er ins unermessliche. Deshalb ist stets eine Reproduktionsrate $P_{rep} < 1$ zu wählen.

Als Erweiterung zu einem reinen Galton-Watson-Prozess besteht in jedem Takt die Möglichkeit, dass weitere Übertragungen initiiert werden. Die Anzahl der neuen Übertragungen wird aus einem Poisson-Prozess mit dem Erwartungswert P_{gen} generiert. Jede neue Übertragung fügt dem nächsten Takt N Pakete hinzu. Dadurch kann auch ein zum Erliegen gekommener Paketstrom neu gestartet werden.

Mit verschiedenen Parametern können unterschiedliche stochastische Netzwerkauslastungen simuliert werden.

Beispiele:

- **Bursts mit Pausen:** $N = 20$ $P_{rep} = 0,999$ $P_{gen} = 0,01$,
- **Kontinuierlicher Verkehr:** $N = 20$ $P_{rep} = 0,9$ $P_{gen} = 0,7$,
- **Wechselnder Verkehr mit Spitzen:** $N = 20$ $P_{rep} = 0,7$ $P_{gen} = 0,7$.

4.1.2.3 Empfänger

Die Empfängerkomponente lauscht auf Port 8088. Jedes empfangene Paket in eine PCAP Datei nach `/tmp/receiver.pcap` geloggt. Außerdem erfolgt eine Terminalausgabe mit dem entsprechenden Zeitstempel.

4.1.2.4 Konfigurator

Der Konfigurator ist ein in C++ und mit dem Framework QT5 geschriebenes grafisches Programm zur Konfiguration des Paketgenerators. Er dient zur komfortablen Einstellung und Verteilung von Ablaufplänen. Abbildung 4.5 zeigt sein Programmfenster. Am oberen Rand des Fensters befinden sich Buttons zum Hinzufügen von Ablaufblöcken. Diese werden dem Fenster dynamisch hinzugefügt. Weiterhin besteht die Möglichkeit, erstellte Konfigurationen zu speichern und zu laden. Am rechten Rand befinden sich ein Button zum Versenden der Konfiguration an das entsprechende Gerät. Ein weiterer Button startet den Ablauf auf allen Geräten. Dabei wird der Ablauf zu einem Zeitpunkt gestartet, der drei Sekunden nach dem Knopfdruck liegt. So wird sicher gestellt, dass trotz etwaiger Übertragungsverzögerungen der Ablauf auf allen Geräten gleichzeitig gestartet wird.

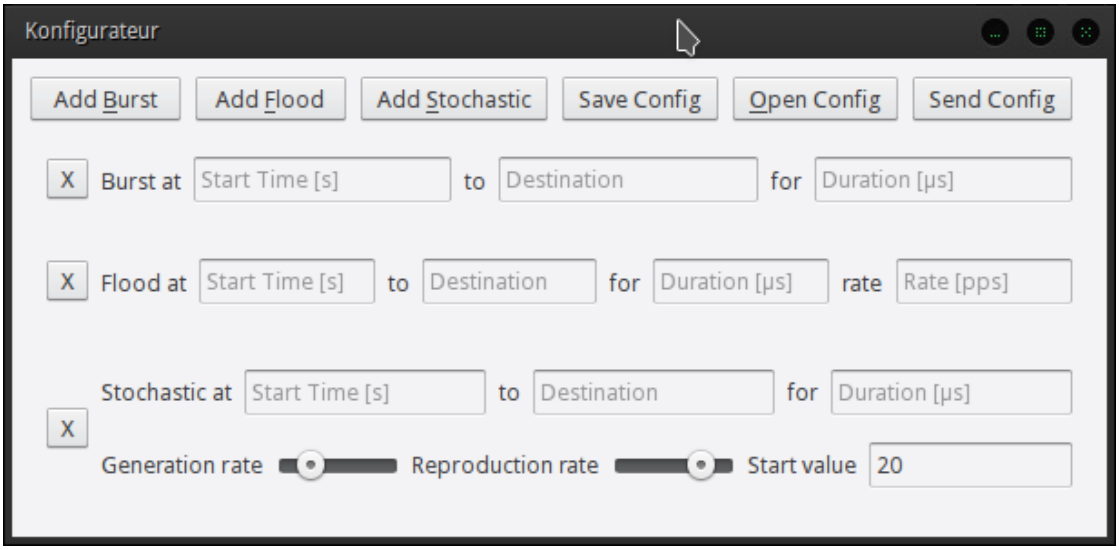


Abbildung 4.5: Hauptfenster des Konfigurators

4.1.3 Bestimmung und Ausgabe der Kanalmaske

Anders als beim passiven Sniffen besteht im Messaufbau Zugriff auf die zu beobachtenden Bluetooth-Geräte. Das ermöglicht es, die Kanalmaske direkt über Werkzeuge des Bluetooth-Stacks BlueZ auszulesen. Das für diese Arbeit geschriebene Programm `readafh` überwacht die Kanalmaske und gibt sie bei jeder Änderungen im Big Endian Format zusammen mit einem Zeitstempel aus.

Alle grafischen Plots der Kanalmaske wurden mit Matplotlib [40] gezeichnet. Matplotlib ist eine Sammlung von Werkzeugen zum Erstellen von Plots mit der Programmiersprache Python. Es bietet Funktionen, mit denen verschiedenste Daten flexibel und übersichtlich dargestellt werden können. Allerdings existiert bisher keine Funktion zum Plotten von Änderungen in von Bitfeldern, wie zum Beispiel der Kanalmaske. Eine solche Funktion wurde im Zuge der Arbeit nachgerüstet. Sie erlaubt es, sowohl die Kanalmaske über die Zeit darzustellen, als auch sie mit einer Referenzkanalmaske zu vergleichen und Unterschiede hervorzuheben.

In den Kanalmaskendarstellungen sind die benutzten und unbenutzten Kanäle über der Zeit aufgetragen. Jede Zeile im Bild repräsentiert einen Kanal. Die Untere Zeile entspricht Kanal 0, die obere entspricht Kanal 78. Benutzte Kanäle erscheinen dunkel, ungenutzte Kanäle sind hell dargestellt. Diese Darstellungsweise ist beispielsweise in Abbildung 4.6 zu sehen. In Kapitel 5 werden in ähnlichen Plots die tatsächliche Kanalmaske mit der vom Sniffer detektierten Kanalmaske verglichen und Unterschiede hervorgehoben. Auch hier werden vom Piconet benutzte Kanäle dunkel, unbenutzte hell dargestellt. Unterscheidet sich die Kanalmaske des Piconets von der des Sniffers, sind die entsprechenden Kanäle mit einem Rotton überlagert. Kanäle die vom Piconet genutzt werden, vom Sniffer aber als ungenutzt erkannt wurden, sind dunkelrot dargestellt. Vom Piconet ungenutzte Kanäle, die jedoch vom Sniffer als genutzt erkannt wurden, sind hellrot dargestellt. Ein Beispiel dafür ist Abbildung 5.4.

4.2 Durchgeführte Messungen

Die Kanalmaske kann nur durch passive Detektion bestimmt werden, wenn sie während der Bestimmung für eine hinreichend lange Zeit konstant bleibt. Ziel der Messung ist es daher, Aussagen darüber treffen zu können, wie veränderlich die AFH Channel Map einer Bluetooth-Verbindung unter verschiedenen Umgebungsbedingungen ist.

Für die Messungen wurde eine Bluetooth-Verbindung unter den Einfluss verschiedener Störungen mit verschiedener Intensität gesetzt. Gleichzeitig wurde über eine

Schnittstelle im BlueZ Bluetooth-Stack die Kanalmaske der Verbindung ausgelesen und auf Änderungen überwacht. Als Bluetooth-Verbindung kam bei allen Messungen eine Bluetooth-Audio-Verbindung über das *Advanced Audio Distribution Profile (A2DP)* [12] zwischen einem Computer und einem Mobiltelefon zum Einsatz, mit der über den gesamten Messzeitraum Musik abgespielt wurde. Als Störer kamen nacheinander die in Abschnitt 4.1 genannten Geräte zum Einsatz: Die WLAN-Knoten, verschiedene Bluetooth-Verbindungen und ein IEEE 802.15.4-Störer.

4.2.1 WLAN-Störer

Als WLAN-Störer kommen die in Abschnitt 4.1.1 vorbereiteten WLAN-Knoten zum Einsatz. Sie sind über ein WLAN-Netz miteinander verbunden, das auf dem WLAN-Kanal 3 betrieben wird. Dieser Kanal erstreckt sich zwischen 2,412 GHz und 1,432 GHz und hat sein Zentrum bei 2,422 GHz [13]. Das Sendesignal wird mit den Dämpfungsgliedern gedämpft. Die Auslastung der Verbindung wird mit dem in Abschnitt 4.1.2 beschriebenen Traffic-Generator gesteuert.

Zunächst sollte untersucht werden, welchen Einfluss die Signalstärke auf die Kanalmaske hat. Dafür wurde für eine feste Senderate von 200 Paketen pro Sekunde und eine feste Sendezeit von 100 Sekunden die eingestellte Dämpfung des Dämpfungsglieds variiert. Folgende Messungen wurden durchgeführt:

- 0 db Dämpfung,
- 20 db Dämpfung,
- 22,5 db Dämpfung,
- 25,5 db Dämpfung,
- 28,5 db Dämpfung,
- 31,5 db Dämpfung.

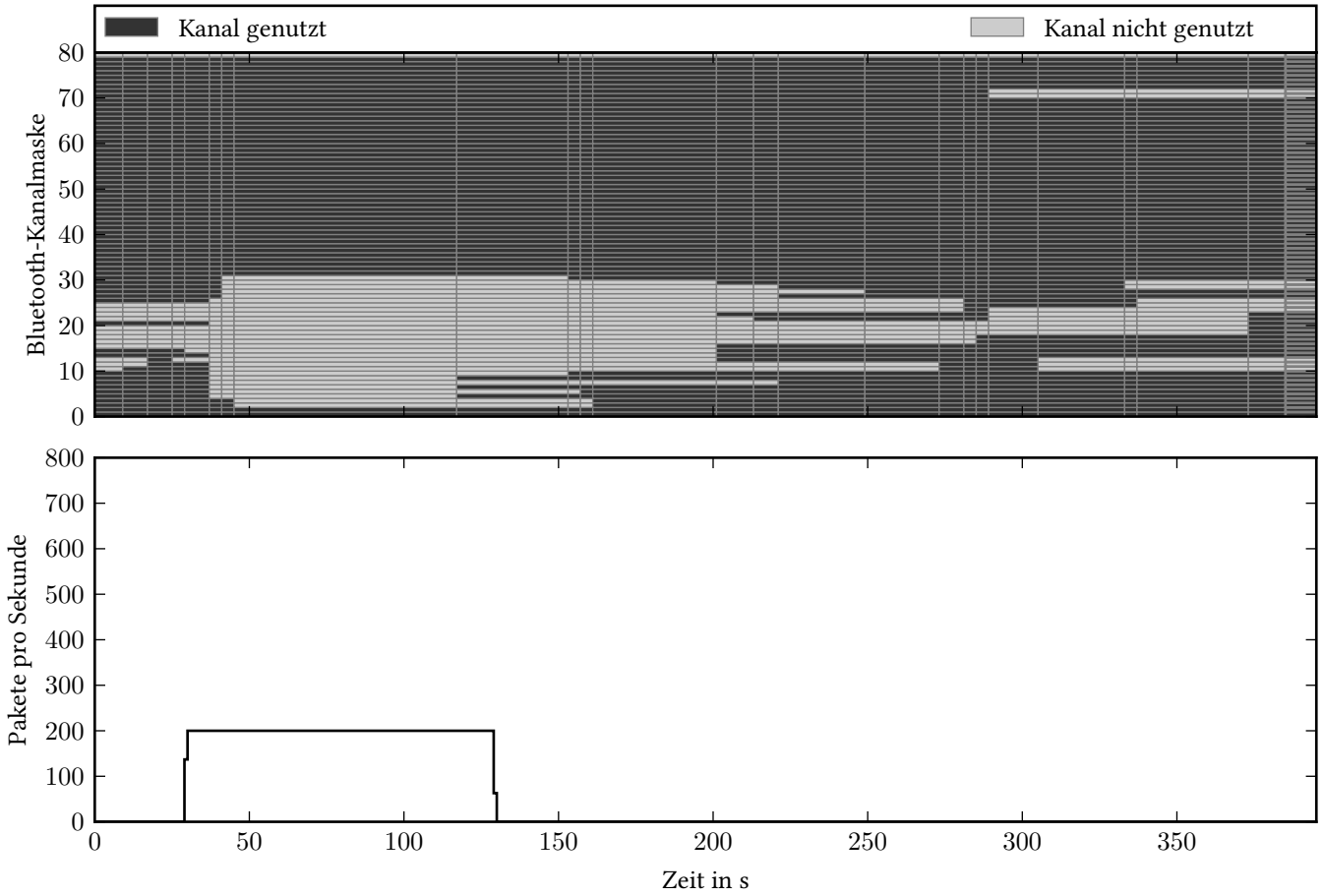


Abbildung 4.6: AFH Channel Map bei einer Störung durch WLAN mit einer Senderate von 200 Paketen pro Sekunde für eine Dauer von 100 Sekunden bei 0 dB Dämpfung

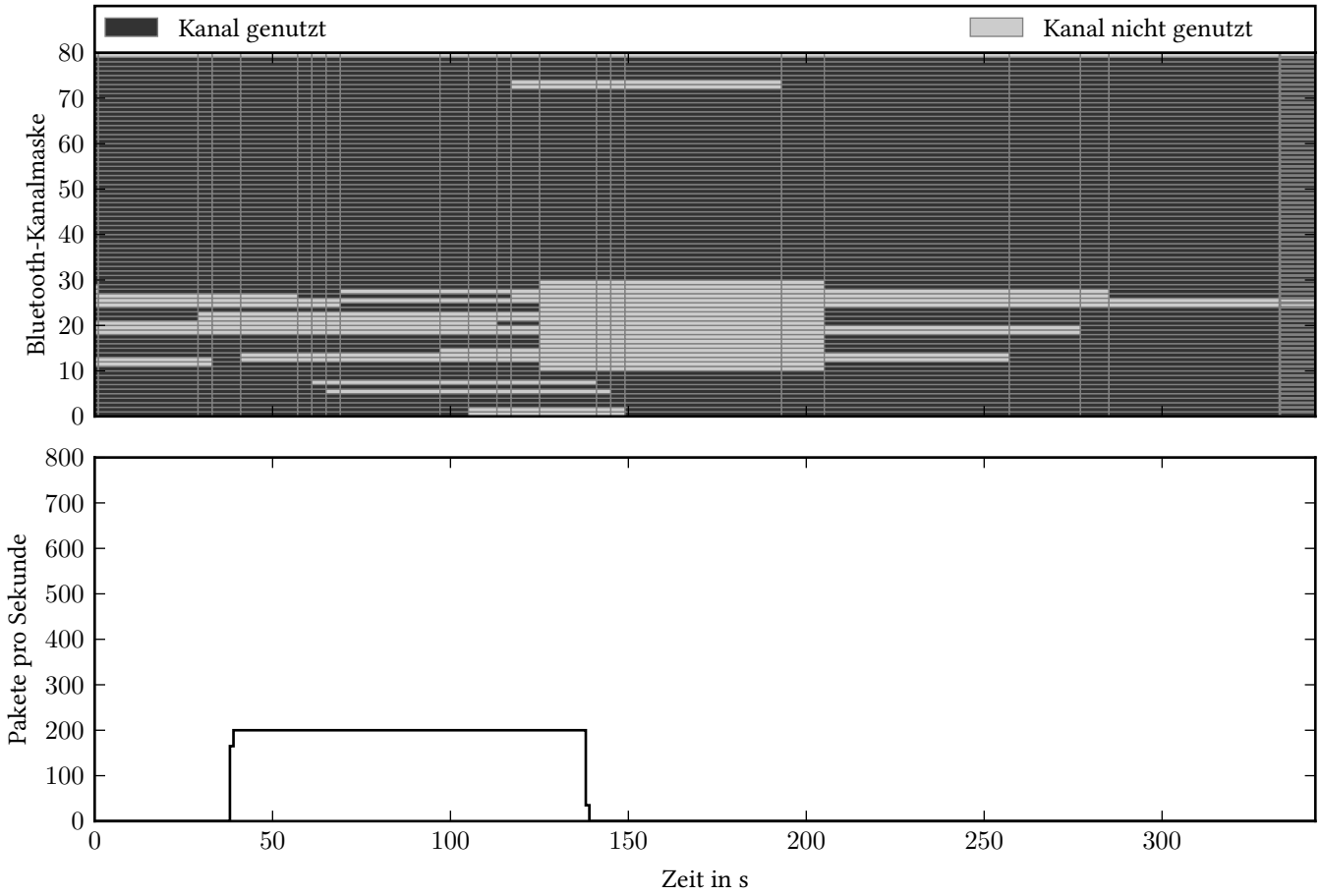


Abbildung 4.7: AFH Channel Map bei einer Störung durch WLAN mit einer Senderate von 200 Paketen pro Sekunde für eine Dauer von 100 Sekunden bei 31,5 dB Dämpfung

Die Abbildungen 4.6 und 4.7 zeigen die zeitliche Änderung der Kanalmaske jeweils für eine Dämpfung von 0 dB und 31,5 dB. Benutzte Kanäle sind dunkel eingezeichnet, unbenutzte Kanäle erscheinen hell. Zunächst ist zu erkennen, dass in beiden Fällen bereits Kanäle im Bereich des WLAN-Netzes ungenutzt sind, selbst wenn keine Datenübertragung stattfindet. Das Muster ist jedoch äußerst unbeständig. Grund dafür ist, dass WLAN Access Points in größeren Zeitabständen Beacon-Frames versenden. Diese Frames werden periodisch versendet, um das WLAN öffentlich sichtbar zu machen und enthalten Informationen zur Konfiguration des Dahtlosnetzes. Diese Signale können von Bluetooth als Störer erkannt werden. Da dieser Effekt auch in der Praxis auftritt, ist er eine hinnehmbare Verfälschung der Messergebnisse.

Die zweite Auffälligkeit ist, dass ein zeitlicher Versatz zwischen dem Beginn der WLAN-Datenübertragung und der Anpassung der Bluetooth-Kanalmaske existiert. Dieser Versatz ist umso größer, je höher die Dämpfung des Störsignals ist.

Auch zwischen dem Ende der WLAN-Datenübertragung und der Wiederbenutzung der zuvor gestörten Bluetooth-Kanäle existiert ein zeitlicher Versatz. Dieser beträgt unabhängig von der Dämpfung des Störsignals etwa eine Minute.

Schlussendlich ist zu erkennen, dass bei einer höheren Störsignaldämpfung mehr Bluetooth-Kanäle benutzt bleiben. Das betrifft vor allem jene Bluetooth-Kanäle, die sich an den Rändern des WLAN Kanals befinden. Hier ist die Signalstärke des WLANs anscheinend so weit abgefallen, dass das Signal nicht mehr als Störer erkannt wird.

In der Praxis werden aufgrund der unterschiedlichen Standorte der störenden WLAN-Geräte verschiedene Dämpfungsfälle auftreten. Eine hohe Dämpfung tritt auf, wenn der Störer weit entfernt ist oder sich dämpfende Elemente wie zum Beispiel Wände, Metallteile, Fensterscheiben oder wasserhaltige Körper in der Umgebung befinden.

Als nächstes wurde der Einfluss der Senderate auf die Kanalmaske untersucht. Dabei wurde bei einer fest eingestellten Dämpfung von 0 dB die Paketrate variiert und ein 100 Sekunden lang andauerndes Störsignal gesendet. Folgende Messungen wurden durchgeführt:

- 50 Pakete pro Sekunde,
- 100 Pakete pro Sekunde,
- 150 Pakete pro Sekunde,
- 180 Pakete pro Sekunde,
- 200 Pakete pro Sekunde,
- 300 Pakete pro Sekunde,
- Maximaler Durchsatz.

Es zeigt sich, dass geringere Senderaten genau wie eine starke Signaldämpfung zu einer verzögerten Anpassung der Kanalmaske führen. Ebenfalls ist deutlich feststellbar, dass bei einer geringen Senderate weniger Kanäle als unbenutzt markiert werden, als bei einer hohen Senderate. Allerdings treten die Unterschiede hier nicht nur an den Rändern des WLAN-Kanals auf, sondern betreffen auch einzelne Bluetooth-Kanäle im Bereich des WLAN-Kanals.

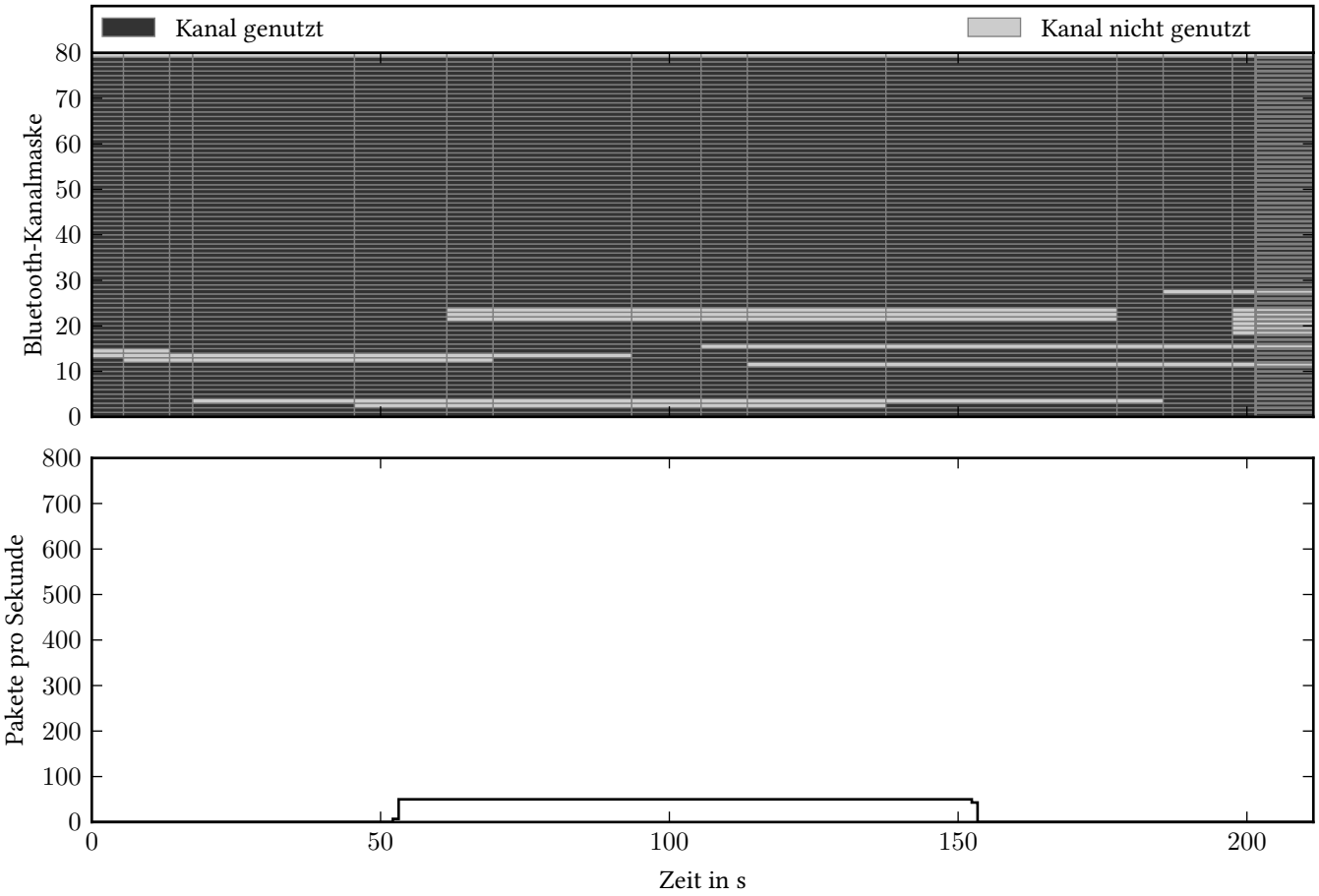


Abbildung 4.8: Bei einer Senderate von 50 Paketen pro Sekunde ist keine Anpassung der AFH Channel Map sichtbar.

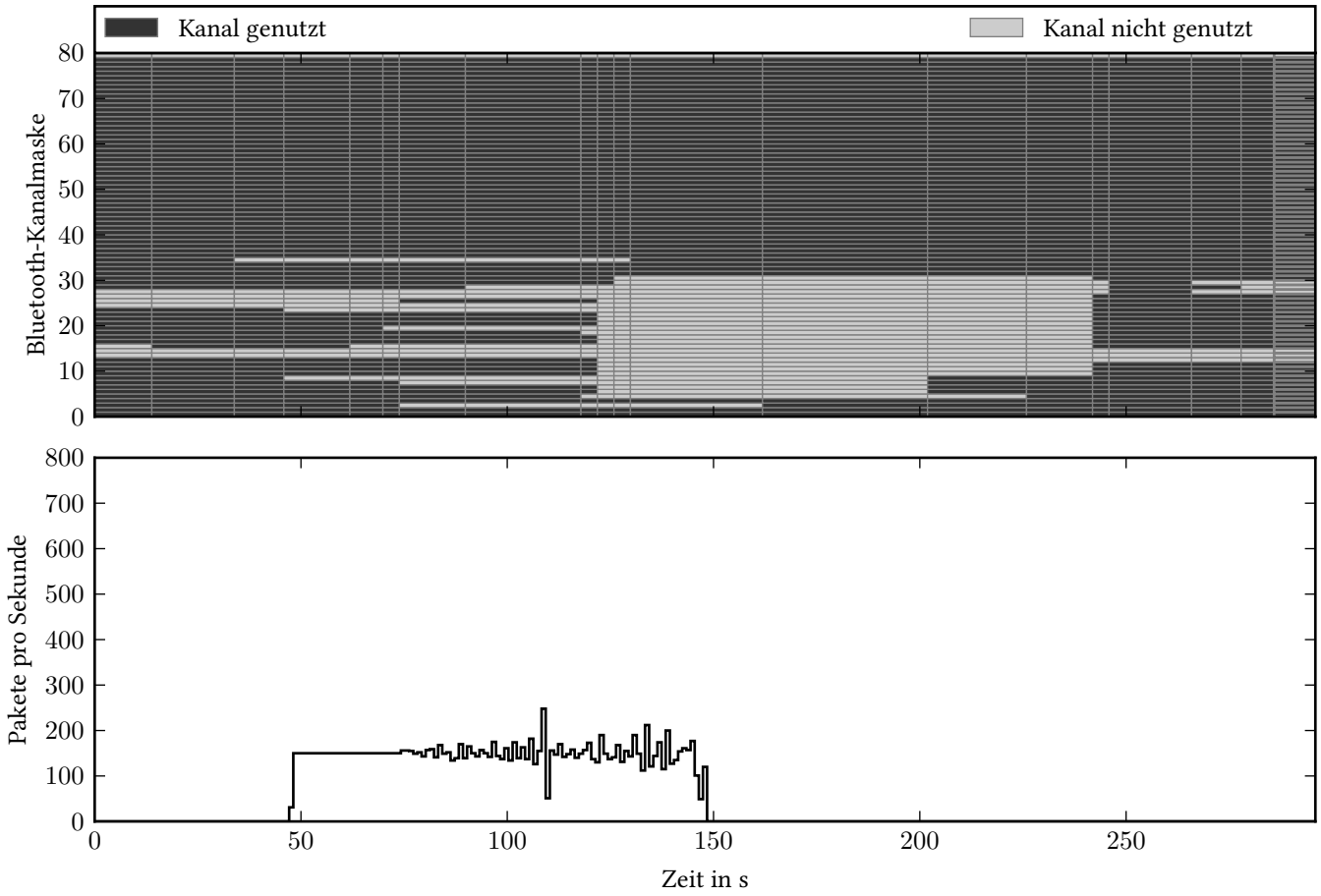


Abbildung 4.9: Erst bei einer Senderate von 150 Paketen pro Sekunde wird das WLAN als Störer erkannt.

Bei einer Senderate von 50 Paketen pro Sekunde ist keine Anpassung der Kanalmaske sichtbar, die nicht auch bei einem unbenutzten WLAN auftreten würde (s. Abbildung 4.8). Die Störungen sind also anscheinend nicht langanhaltend und häufig genug, als dass das Bluetooth-Piconet sie als permanente Störung klassifizieren würde. Erst ab einer Paketrate von 150 Paketen pro Sekunde wird ein wesentlicher und zusammenhängender Teil der Bluetooth-Kanäle im Bereich des WLAN Kanals aus der Kanalmaske entfernt (s. Abbildung 4.9).

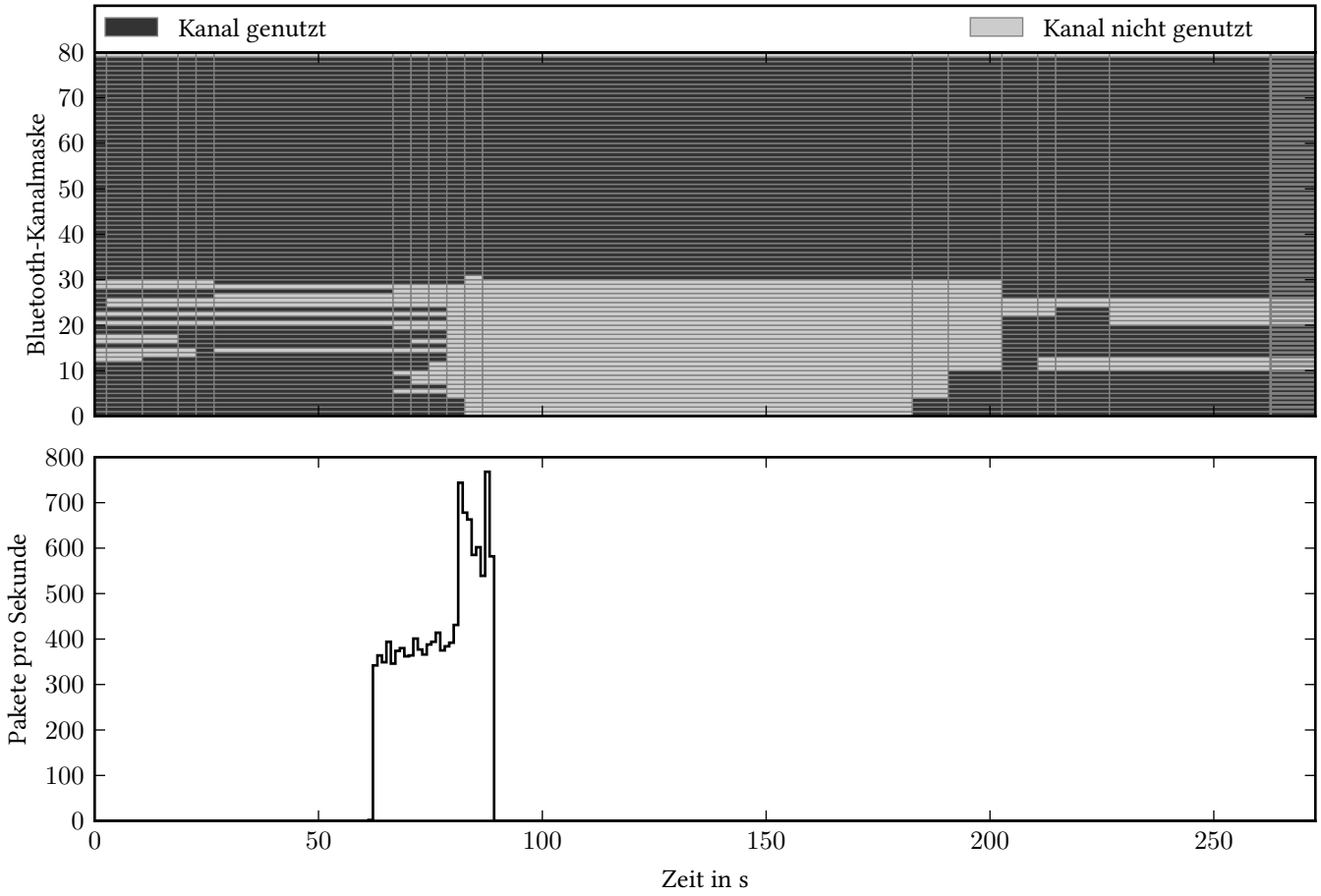


Abbildung 4.10: AFH Channel Map bei einer Störung durch WLAN bei maximal möglicher Senderate. Deutlich erkennbar ist der Anstieg der WLAN Senderate, sobald die entsprechenden Bluetooth-Kanäle nicht mehr benutzt werden.

Interessant ist der in Abbildung 4.10 dargestellte Fall, bei dem der WLAN-Knoten mit maximal möglicher Datenrate senden soll. Hier ist deutlich zu erkennen, dass die Datenrate des Störers sofort deutlich ansteigt, sobald die im selben Frequenzbereich liegenden Bluetooth-Kanäle nicht mehr genutzt werden. Der Grund hierfür ist, dass WLAN selbst Koexistenzmechanismen besitzt, die eine Störung anderer Funktechnologien verringern soll. Durch das Listen-Before-Talk Verfahren erkennen die WLAN-Geräte die Bluetooth-Verbindung als Störer und verzögern ihre Datenübertragung, bis der Kanal frei ist. Das geschieht zu Lasten der erreichbaren Datenrate. Trotzdem wird die Bluetooth-Verbindung hinreichend stark gestört, sodass die entsprechenden Bluetooth-Kanäle als ungenutzt markiert werden. Sobald das geschehen ist, entfällt die Wartezeit bei der WLAN Übertragung, sodass sich die Paketrate beinahe verdoppeln kann.

In der Praxis treten in WLAN-Netzen verschiedene Paketdichten auf. Maximaler Durchsatz lässt sich etwa während eines Downloads beobachten. Die Dauer ist von der Größe der heruntergeladenen Datei abhängig.

Eine niedrige Paketdichte entspricht zum Beispiel dem Versand von Keepalive-Paketen eines Dienstes oder dem Versand einer Instant Message. Auch alle Verkehrsmuster dazwischen mit verschiedener Paketdichte und verschiedener Dauer können aufgrund der großen Vielfalt von Netzwerkdiensten jederzeit auftreten.

Die Messergebnisse zeigen, dass ein WLAN-Netz eine Bluetooth-Verbindung stark stört. Diese Störung ist auf etwa 20 Bluetooth-Kanäle begrenzt. Diese Zahl korrespondiert mit der Bandbreite von IEEE 802.11n Netzen, die 20 MHz beträgt. Hier hilft Adaptive Frequency Hopping dabei, Kollisionen zwischen Bluetooth und WLAN zu vermeiden, indem die von der WLAN-Störung betroffenen Bluetooth-Kanäle nicht benutzt werden. Kritisch wird die Situation, sobald beispielsweise drei WLAN-Netze nebeneinander betrieben werden, die das gesamte 2,4 GHz Band und somit alle Bluetooth-Kanäle stören. In solchen Situationen treten trotz Adaptive Frequency Hopping Paketkollisionen und damit Datenverlust auf.

4.2.2 Bluetooth-Störer

Als Bluetooth-Störer kommen die drei Verbindungen zum Einsatz, die in Abschnitt 4.1 beschrieben wurden. Zunächst eine Ethernet-Verbindung über Bluetooth über das Bluetooth Network Encapsulation Protocol, deren Auslastung mit dem Traffic-Generator eingestellt wird. Mit dieser Strecke kann eine hohe Paketdichte und damit eine hohe Zahl an Kanalbenutzungen erreicht werden.

Der zweite Störer ist der Bluetooth-Tastaturcontroller *Bluefruit EZ Key*, der drahtlos Tastenanschläge sendet. Dabei kann jeder Tastenanschlag über seine serielle Schnittstelle zeitgenau ausgelöst werden. Auf dieser Strecke wird nur eine äußerst geringe Paketdichte erreicht.

Der dritte Störer ist eine Audioverbindung zwischen einem Raspberry Pi und einem Bluetooth-Audio-Stick, der kein Adaptive Frequency Hopping unterstützt. Der Stick empfängt Musik mit einer festen Bitrate.

Folgende Messungen wurden durchgeführt:

- Messung der Kanalmaske eines Geräts bei voller Auslastung einer BNEP Verbindung
- Messung der Kanalmaske eines Geräts bei einer Tastaturauslastung von einem Zeichen pro Sekunde
- Messung der Kanalmaske mit dem Hausnetz als WLAN Störer und einer Bluetooth-Audioverbindung im Hintergrund

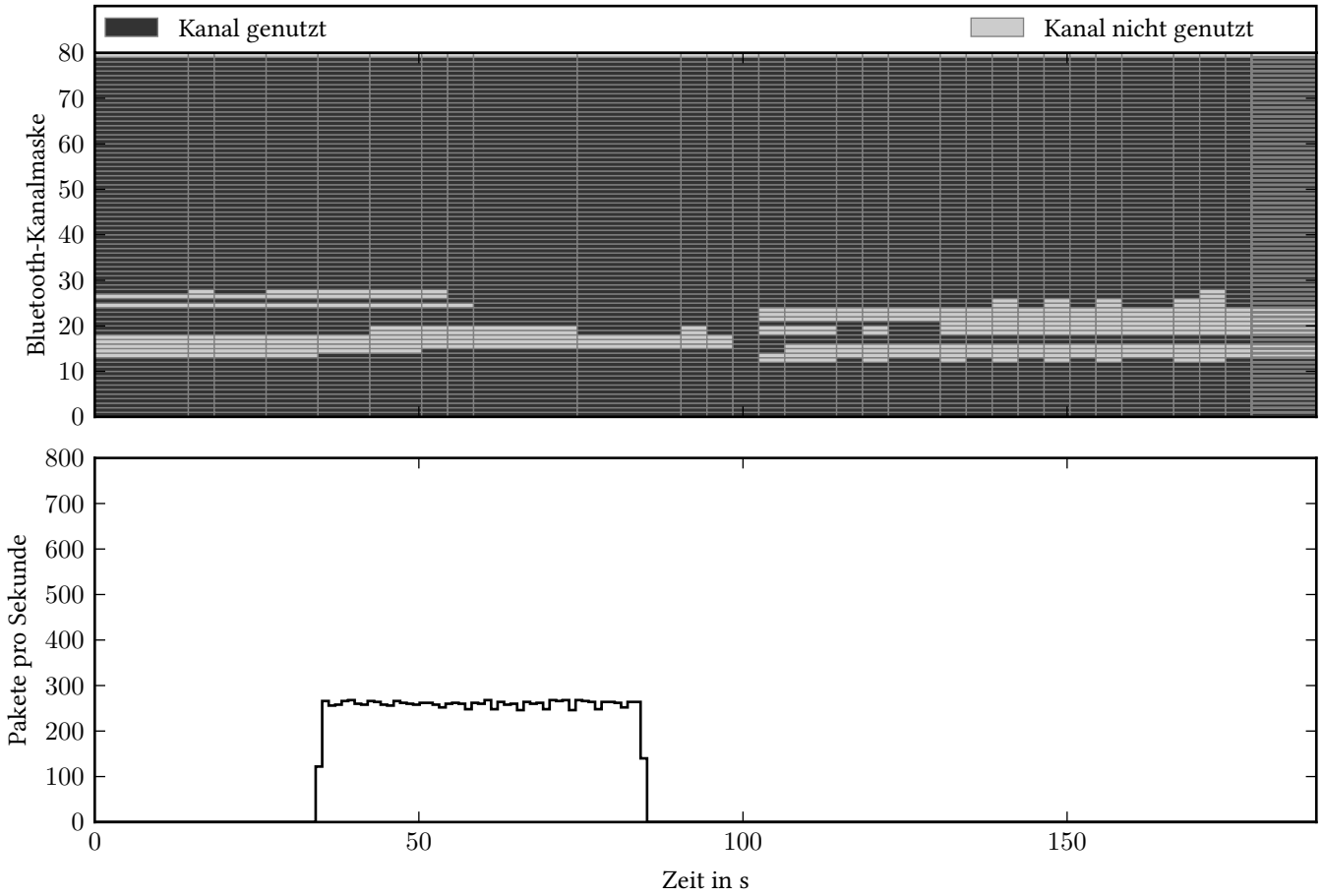


Abbildung 4.11: Eine BNEP Verbindung als Störer hat keinen Einfluss auf die Kanalmaske.

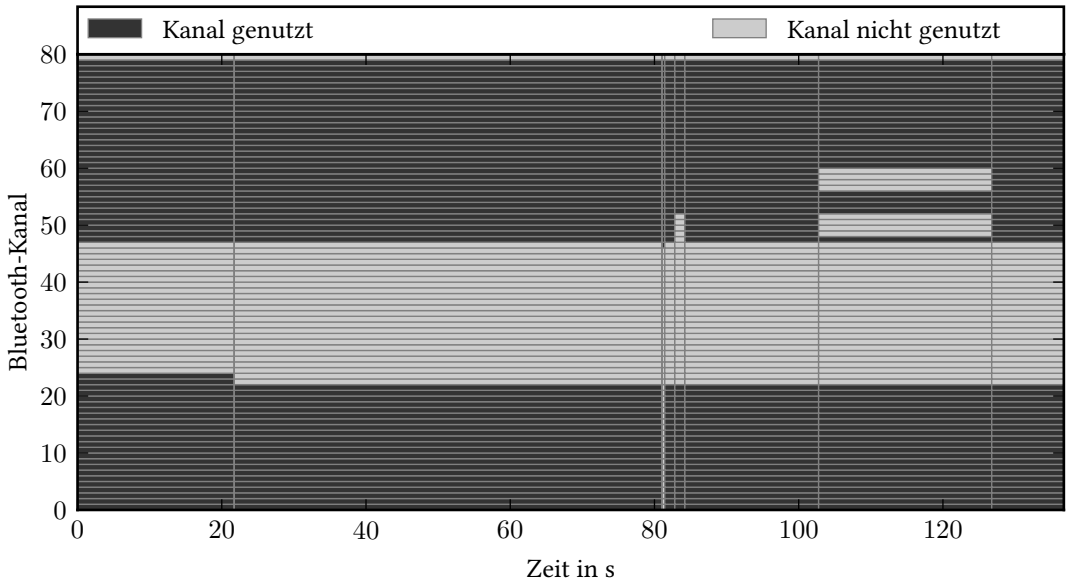


Abbildung 4.12: Auch eine Audioverbindung, die kein Adaptive Frequency Hopping nutzt, stört eine Bluetooth-Verbindung nicht.

Abbildungen 4.11 und 4.12 zeigt mehrere Bluetooth-Verbindungen, die durch gleichzeitig parallel bestehende Bluetooth-Verbindungen gestört werden. In Abbildung 4.11 sendet eine voll ausgelastete BNEP Verbindung für 20 Sekunden. Die in Abbildung 4.12 gezeigte Kanalmaske gehört zu einer Bluetooth-Verbindung, die die gesamte Beobachtungszeit über vom Institutsnetz auf WLAN-Kanal 6, sowie einer Audioverbindung mit einem Audio-Stick gestört wurde. Der Audio-Stick unterstützt kein Adaptive Frequency Hopping. In allen Fällen ist keine Änderung der Kanalmaske zu erkennen, die auf eine Störung durch eine Bluetooth-Übertragung zurückzuführen ist.

Bei zwei nebeneinander laufenden Bluetooth-Verbindungen verhindert das Frequency-Hopping sehr zuverlässig eine anhaltende Störung einzelner Kanäle. Frequency Hopping ist also eine gute Möglichkeit, Störungen zu vermeiden. Allerdings werden Paketkollisionen häufiger auftreten, wenn mehrere parallele Verbindungen nebeneinander bestehen.

4.2.3 IEEE 802.15.4 Störer

Auch schmalbandige Störungen können im industriellen Umfeld auftreten. Drahtlose Sensoren senden häufig auf genau einem Kanal. Beispiele hierfür sind Implementie-

rungen des Standards IEEE 802.15.4 [15], wie Beispielsweise das Protokoll WirelessHART [5], sowie der Standard ISA 100.11a [18]. Als ein entsprechender Störer wurde ein weiterer Bluetooth-Dongel benutzt, der im Debug-Modus genau einen Kanal mit einer festen Sendeleistung stört. Zusätzlich wurde ihm ein Dämpfungsglied vorgeschaltet, um verschiedene Intensitäten zu untersuchen. Gemessen wurde mit einer Dämpfung von 0 dB und einer Dämpfung von 31,5 dB.

Abbildung 4.13 zeigt die jeweilige Kanalmaske. Hier zeigt sich, dass bei einer Störung, die auf nur einem Kanal auftritt, auch die Nachbarkanäle des gestörten Kanals als ungenutzt markiert werden.

4.3 Ergebnisse des Vorversuchs

Der Vorversuch zeigt, dass die Kanalmaske von Bluetooth-Verbindungen unter dem Einfluss verschiedener Störer sehr unterschiedlich reagiert.

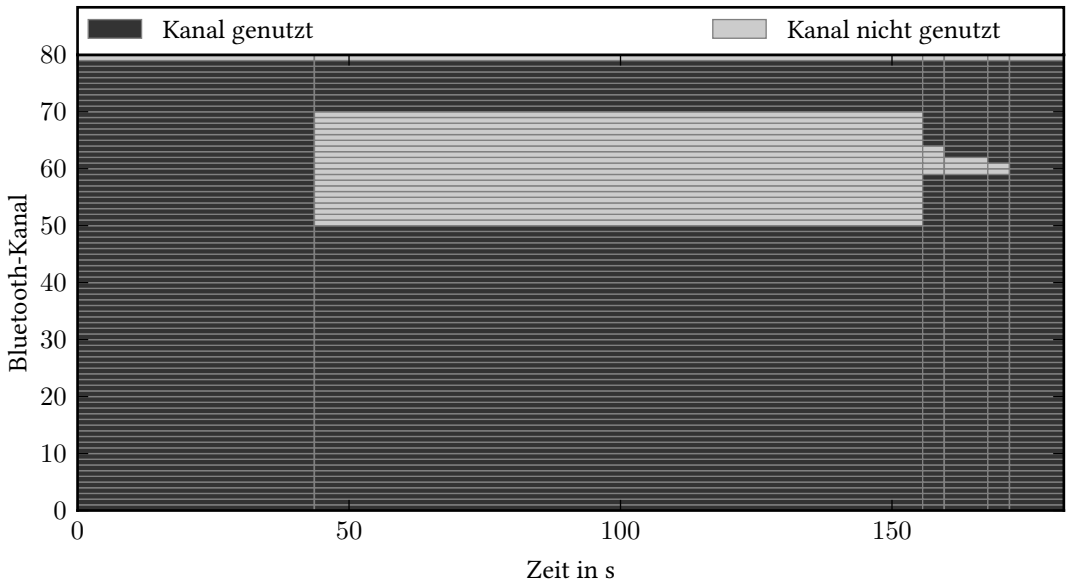
Eine Änderung der Kanalmaske aufgrund veränderter Störbedingungen tritt immer etwas verzögert auf. Das kann von wenigen Sekunden bis zu einer Minute reichen. In jedem betrachteten Fall hat sich die AFH Channel Map höchstens einmal alle vier Sekunden geändert.

Die Verzögerung der Reaktion ist stark abhängig von der Signalstärke und der Senderate des Störers. Je stärker das Signal und je häufiger die Kanalbenutzungen, desto schneller reagiert die AFH Channel Map. Die Wiederbenutzung einzelner ungenutzter Kanäle nachdem diese wieder frei sind, ist ebenfalls verzögert. Ein nicht benutzter Kanal wird erst nach über einer Minute wieder genutzt.

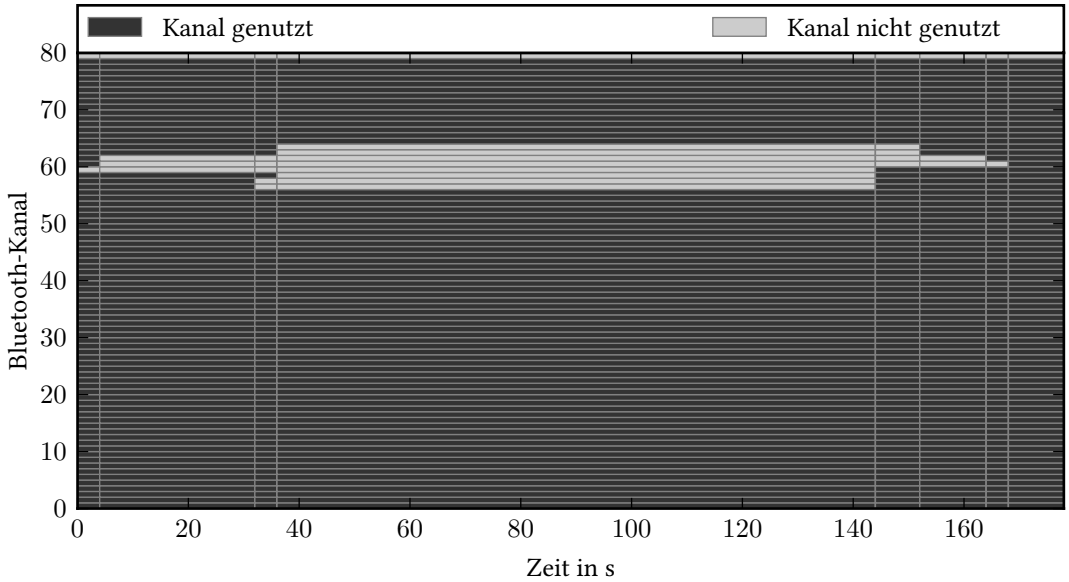
Zwei gleichzeitig stattfindende Bluetooth Verbindungen stören sich gegenseitig nicht. Dabei spielt es keine Rolle, ob ein Bluetooth-Störer Adaptive Frequency Hopping unterstützt oder nicht.

Bei einem schmalbandigen Störer, der nur einen Bluetooth-Kanal belegt, werden auch die benachbarte Kanäle aus der Kanalmaske gelöscht. So kommt es nur äußerst selten vor, dass ein einzelner Kanal ungenutzt ist.

Ohne Störung ist die Kanalmaske instabil. Es werden Kanäle als unbenutzt gekennzeichnet, obwohl keiner der Störquellen im Versuchsaufbau funkt. Dabei ändert sich die AFH Channel Map alle vier bis zwölf Sekunden. Am stabilsten blieb die Kanalmaske bei Messungen mit einer beständiges, leistungsstarkes, scharf begrenztes Störung.



(a) Dämpfung 0 dB



(b) Dämpfung 31.5 dB

Abbildung 4.13: Ein Störsignal mit einer Breite von 1 MHz führt dazu, dass mehr als ein Bluetooth-Kanal nicht benutzt wird.

Die Auslastung der gestörten Bluetooth-Verbindung selbst hat jedoch keinen spürbaren Einfluss auf ihre Kanalmaske. Das betrifft sowohl das Löschen von gestörten Kanälen aus der Kanalmaske, als auch das Wiederbenutzen von nicht mehr gestörten Kanälen.

Bluetooth reagiert empfindlich auf Reflexionen. Treten starke Reflexionen auf, erkennt BlueZ offensichtlich das eigene Signal als Störer. Das hat zur Folge, dass unter solchen Bedingungen nahezu alle Kanäle in der AFH Channel Map alle vier Sekunden genau umschalten. Zum Beispiel werden erst alle oberen Kanäle verwendet, dann alle unteren.

Zu beachten ist, dass sämtliche Geräte im Messaufbau die Bluetooth-Implementierung BlueZ [24] nutzen. Die beiden Raspberry Pis verwenden den Bluetooth-Dongel BCM20702A0 von Broadcom. Da die Klassifikation von Bluetooth-Kanälen nicht im Bluetooth-Standard festgelegt ist, ist die Kanalmaske und ihre Reaktion auf Störungen abhängig von der jeweiligen Bluetooth-Implementierung. Alle Ergebnisse des Vorversuchs sind also nicht notwendigerweise auf andere Bluetooth-Implementierungen oder Bluetooth-Chipsätze übertragbar. Im Ausblick in Kapitel 7 wird daher empfohlen, entsprechende Versuche auch mit anderen Bluetooth-Implementierungen durchzuführen.

Für eine passive Detektion der Kanalmaske eröffnen die Ergebnisse des Vorversuchs einige Möglichkeiten, die eine erfolgreiche Erkennung der AFH Channel Map vereinfachen. Im Allgemeinen muss ein entsprechender Detektionsalgorithmus in der Lage sein, die Kanalmaske korrekt zu erkennen oder Änderungen an ihr festzustellen, bevor sie sich erneut ändert. Wie sich gezeigt hat, kann durch eine starke Störung die Kanalmaske über einen längeren Zeitraum konstant gehalten werden. In mehreren Messungen konnten Kanalmasken beobachtet werden, die über Stunden hinweg konstant bleiben. Bei einigen im industriellen Umfeld eingesetzte Bluetooth-Geräte, wie zum Beispiel dem Bluetooth-Ethernet-Umsetzer *FL BT EPA* [19] von Phoenix Contact, ist die Kanalmaske fest konfigurierbar, bleibt also im Betrieb unabhängig von äußeren Störungen konstant. In all diesen Fällen bleibt also genügend Zeit für eine passive Detektion.

5 Entwicklung eines Algorithmus zur passiven Kanalmaskendetektion

Im folgenden Kapitel wird ein Algorithmus zur passiven Detektion der Kanalmaske vorgestellt. Zunächst werden allgemeine Lösungen von Teilproblemen der Kanalmaskendetektion entwickelt, darauf folgt eine Beschreibung der konkreten Implementierung im Quelltext des Ubertooth Projekts. Schließlich wird die Leistungsfähigkeit des implementierten Detektionsalgorithmus experimentell geprüft und bewertet.

5.1 Entwurf eines Detektionsalgorithmus

In diesem Abschnitt wird der Prozess der Kanalmaskendetektion in Teilprobleme zerlegt. Zunächst wird die Erkennung von genutzten Kanälen gelöst. Im Anschluss wird eine Möglichkeit zur Erkennung von ehemals genutzten Kanälen gezeigt, die nach einer Änderung der Kanalmaske nicht mehr genutzt werden. Zuletzt mehrere Maßnahmen zur Erhöhung der Detektionsgeschwindigkeit gezeigt.

In Kapitel 2 wurden Teile der Bluetooth-Spezifikation vorgestellt, die für eine passive Detektion der Kanalmaske relevant sind. Die in Abschnitt 2.1.7 zusammengefassten Erkenntnisse werden für die Entwicklung des Detektionsalgorithmus genutzt.

5.1.1 Erkennung genutzter Kanäle anhand der beobachteten Pakete

Die einfachste Möglichkeit in Erfahrung zu bringen, welche Kanäle genutzt werden und welche nicht, ist, mit dem Ubertooth eine ausreichend lange Zeit auf einem Kanal zu lauschen und auf Pakete vom zu untersuchenden Piconet zu warten. Wird ein Paket

empfangen, wird der Kanal offensichtlich genutzt, ist also zurzeit in der Kanalmaske des Piconets gesetzt. Somit kann auf den nächste Kanal gewechselt und dort gelauscht werden. Wird in der Zeit jedoch kein Paket empfangen, besteht die Möglichkeit, dass der Kanal ungenutzt ist und es muss auf dem nächsten Kanal gelauscht werden.

Mit diesem Vorgehen ist die Erkennung der Kanalmaske möglich, ohne dass der Sniffer das Sprungschema des beobachteten Piconets kennt. Das ist notwendig, denn wie in Abschnitt 2.1.3 gezeigt wurde, ist das Sprungschema abhängig von der Kanalmaske. Kenntnis des Sprungschemas setzt also Kenntnis der Kanalmaske voraus.

Daher wird diese Herangehensweise in der initialen Kanalmaskendetektion aufgegriffen. Im entsprechenden Algorithmus verbleibt der Ubertooth-Sniffer maximal 158 CLKN-Perioden oder 98,75 ms auf einem Kanal und lauscht auf ankommende Pakete. Diese Dauer entspricht der Zeit, die ein Bluetooth Piconet benötigt, um zweimal 79 Kanalsprünge durchzuführen (vgl. Abschnitt 2.1). In dieser Zeit wird der Kanal, auf dem gelauscht wird, also durchschnittlich zweimal von der Bluetooth-Verbindung genutzt. Auf diese Weise konnte in einem ersten Test die AFH Channel Map einer Bluetooth Audioverbindung in den meisten Fällen innerhalb von 30 Sekunden bestimmt werden. Einzelne Versuche mit längeren Wartezeiten pro Kanal führten zu einer langsameren Entdeckung. Eine weitere Variation der Wartezeit wurde nicht vorgenommen, weil diese Arbeit die prinzipielle Entwicklung eines Algorithmus zur Kanalmaskendetektion zum Ziel hat. Eine dahingehende Optimierung der Detektionszeit kann in einer weiterführenden Arbeit untersucht werden.

5.1.2 Erkennen der nicht mehr genutzten Kanäle

Der zweite Schritt ist die Bestimmung von ehemals genutzten Kanälen, die aufgrund von zwischenzeitlich aufgetretenen Störungen aus der Kanalmaske des Piconets gelöscht wurden. Es besteht die Möglichkeit, Kanäle, auf denen lange Zeit keine Übertragung mehr stattgefunden hat, als nicht benutzt zu betrachten. Doch wie unterscheidet man zwischen den tatsächlich vom Piconet nicht genutzten Kanälen und solchen, in deren Zeitslot zufällig keine Daten zur Übertragung vorlagen? Dieses Problem kann nicht mit Sicherheit gelöst, wohl aber statistisch betrachtet werden.

Wie in Abschnitt 2.1.3 gezeigt wurde, werden Frequency-Hopping alle genutzten Kanäle mit der gleichen Häufigkeit verwendet [29]. Angenommen, die Verteilung der versendeten Pakete über die Zeit sei gleichverteilt, so ist der Empfang eines Paketes auf jedem Kanal gleich wahrscheinlich. Wenn ein Paket empfangen wird, so ist die

Wahrscheinlichkeit, dass es auf dem Kanal k empfangen wurde für jeden Kanal gleich, nämlich:

$$P_k = \frac{1}{79} \qquad \forall k \in [0; 78]. \tag{5.1}$$

Der Paketempfang auf einem Kanal k entspricht also einem *Bernoulli-Prozess* [16] mit einer Erfolgswahrscheinlichkeit von P_k. Werden n Pakete insgesamt empfangen, so ist die Anzahl der auf Kanal k empfangenen Pakete N_k binominalverteilt [16]:

$$p(N_k = x) = B\left(x \middle| \frac{1}{79}, n\right). \tag{5.2}$$

Ist Kanal k jedoch ungenutzt, so ist die Wahrscheinlichkeit P_k eines Paketempfangs in jedem Fall:

$$P_k = 0. \tag{5.3}$$

Die Entscheidung, ob ein Kanal benutzt ist oder nicht, lässt sich auf eine Schätzung von P_k zurückführen. Angenommen, seit dem letzten Paketempfang auf Kanal k wurden n Pakete auf anderen Kanälen empfangen. Keines davon wurde auf Kanal k empfangen, es gilt $N_k = 0$. Somit kann eine *Punktschätzung* [16] durchgeführt werden. Die geschätzte Empfangswahrscheinlichkeit ist:

$$\hat{P}_k = \frac{N_k}{n} = \frac{0}{n} = 0. \tag{5.4}$$

Sie kann jedoch von diesem Wert abweichen und deutlich größer sein. Sicher ist allein, dass die tatsächliche Empfangswahrscheinlichkeit zwischen 0 und 1 liegt. Eine Punktschätzung allein liefert also keine Erkenntnisse.

Interessant ist das Intervall, in dem nach einem Empfang von n Paketen auf einem anderen Kanal als k die tatsächliche Empfangswahrscheinlichkeit P_k mit der hohen Wahrscheinlichkeit γ liegt. Eine solche Schätzung nennt man *Bereichsschätzung* [16]. Das entsprechende Intervall heißt *Konfidenzintervall zum Konfidenzniveau* γ.

Da N_k binominalverteilt ist, ergibt sich laut [3] die obere Grenze des Konfidenzintervalls $\max(P_k)$ aus dem unteren γ-Quantil der Betaverteilung:

$$B(\alpha = N_k + 1, \beta = n - N_k). \tag{5.5}$$

Da unter den n empfangenen Paketen keines auf Kanal k empfangen wurde, ist $N_k = 0$. Die entsprechende Betaverteilung ergibt sich also zu:

$$B(\alpha = 1, \beta = n). \tag{5.6}$$

Die untere Grenze $\min(P_k)$ ist immer 0.

Entscheidend ist nun, ab welcher Zahl n das γ-Quantil der Betaverteilung auf einen Wert unterhalb der Empfangswahrscheinlichkeit $P_k = \frac{1}{79}$ für einen benutzten Kanal fällt. Für ein Konfidenzniveau von $\gamma = 95\% = 0,95$ müssen beispielsweise $n = 236$ Pakete auf einem anderen Kanal als k empfangen werden. Erst dann kann behauptet werden dass Kanal k mit einer Wahrscheinlichkeit von $\gamma = 95\%$ ungenutzt ist. Für eine Sicherheit von nur 75% reichen $n = 109$ Pakete aus.

Die C++ Bibliothek Boost [4] besitzt Datenstrukturen, die eine Betaverteilung darstellen, und zugehörige Methoden, um den wichtigen Parameter β, der hier der Paketanzahl n entspricht, zu berechnen. Um unterschiedliche Signifikanzniveaus und damit unterschiedliche Sicherheiten bei der Kennzeichnung ungenutzter Kanäle zu ermöglichen, wurde eine Bibliothek geschrieben, die zu einem gegebenen Konfidenzniveau γ die notwendige Anzahl an zu empfangenden Paketen n berechnet.

5.1.3 Optimierung zur schnelleren Erkennung von Veränderungen der Kanalmaske

Im bisherigen Ansatz zur Detektion der Kanalmaske kommt ein Sprungschema zum Einsatz, bei dem alle Kanäle nacheinander durchlaufen werden. Es ist vollkommen unabhängig vom eigentlichen Hopping-Pattern des Bluetooth-Piconets. So kann nur ein Bruchteil der tatsächlich gesendeten Pakete empfangen werden, weil der Ubertooth die meiste Zeit auf einem Kanal lauscht, während Pakete auf einem anderen gesendet werden. Besser wäre es, wenn der Sniffer das Hopping-Pattern des Piconets in Erfahrung bringen und dem Piconet unter Verwendung des Hopping-Patterns folgen könnte.

Wie in Kapitel 2 beschrieben, ist das Adaptive Hopping-Pattern von LAP, UAP und der Kanalmaske abhängig. In [34] wurde gezeigt, wie sich alle Parameter bis auf die Kanalmaske in Erfahrung bringen lassen. Die Kanalmaske selbst kann mit dem in Abschnitt 5.1.1 beschriebenen Algorithmus anhand von beobachteten Paketen initial detektiert werden. Danach kann das Hopping-Pattern mit dem in Abschnitt 2.1.3 beschriebenen Hop Selection Kernel des Sniffers berechnet werden. Daraufhin kann der Ubertooth-Sniffer den aktuellen Wert von CLK27–0 des Piconets bestimmen (vgl. Abschnitt 2.2.2). Nun kann der Sniffer dem Piconet folgen.

Um sowohl genutzte als auch nicht mehr genutzte Kanäle detektieren zu können, muss der Sniffer dabei das Basic Hopping Pattern verwenden. Das Basic Hopping-Pattern stimmt in den vom Piconet benutzten Kanälen mit dem Adaptive Hopping-Pattern

überein [29]. Vom Piconet nicht genutzte Kanäle werden auf genutzte Kanäle abgebildet (vgl.Abschnitt 2.1.3). Folgt der Sniffer einem Piconet, das Adaptive Frequency Hopping verwendet, aber unter Verwendung des Basic Hopping-Patterns, wird er also alle Pakete empfangen, die auf den vom Piconet genutzten Kanälen gesendet werden. Alle anderen Pakete lassen sich nicht beobachten. Auf der anderen Seite wird der Sniffer auf den unbenutzten Kanälen nie ein Paket des Piconets empfangen. Das Verfahren erhöht also die Rate der vom Sniffer empfangenen Pakete, was sowohl die Erkennung von Kanälen, die das Piconet nicht mehr nutzt, als auch die Erkennung von neuerdings Kanälen, die vom Piconet neuerdings genutzt werden, bei Änderung der Kanalmaske erheblich beschleunigt.

Da das die initiale der Kanalmaske, die für die Berechnung des Sprungschemas nötig ist, mit der in Abschnitt 5.1.1 beschriebenen Methode fehleranfällig ist, kann es zu *Kanalfehlern* in der erkannten Kanalmaske kommen. Bei einem Kanalfehler unterscheidet sich die erkannte Kanalmaske in einem Kanal von der tatsächlich vom Piconet genutzten Kanalmaske. Zum einen kann es sein, das auf einem vom Piconet genutzten Kanal kein Paket beobachtet wurde und der Kanal deshalb vom Sniffer fälschlicherweise als ungenutzt gekennzeichnet wird. Zum anderen kann es passieren, dass sich die Kanalmaske während der initialen Detektion ändert und vormals vom Piconet genutzte Kanäle nicht mehr genutzt werden, vom Sniffer aber noch als benutzt erkannt sind. Solche Kanalfehler können zu Fehlern in der Erkennung von CLK27–0 führen. Das passiert folgendermaßen: Wie in Abschnitt 2.2.3 beschrieben, vergleicht die libbtbb beobachtete Pakete mit dem aus Kanalmaske, UAP und LAP berechneten Hopping-Pattern und berechnet CLK27–0 als Offset zwischen den beiden Sequenzen [34]. Durch die Kanalfehler in der geschätzten Kanalmaske hat das errechnete Hopping-Pattern ebenfalls Kanalfehler. Immer dann, wenn ein Paket entdeckt wird, das zu einer Position im Hopping-Pattern passen würde, bei dem ein Kanalfehler auftritt, wird diese Folge als falsch erkannt. Schlimmer noch: wenn die Beobachtung durch die Kanalfehler auf ein anderes Offset passt, wird dieses Offset fälschlicherweise als CLK27–0 erkannt. Wenn allerdings nur Pakete zu Zeitpunkten empfangen werden, an denen kein Kanalfehler im Hopping-Pattern existiert, wird das Offset korrekt berechnet.

Die Wahrscheinlichkeit für eine korrekte Berechnung ist abhängig von der Anzahl der Kanalfehler und von der für die Berechnung von CLK27–0 notwendigen Anzahl der beobachteten Pakete. Angenommen, die Kanäle sind gleichmäßig über das Hopping-Pattern verteilt, so sind auch durch eine fehlerhafte Kanalmaske verursachte Kanalfehler gleichmäßig verteilt. Durch die gleichmäßige Verteilung sind die Kanalfehler unabhängig voneinander verteilt. Bei f Kanalfehlern in der Kanalmaske ergibt sich die Wahrscheinlichkeit, dass ein Paket zu einem Zeitpunkt empfangen wurde, zu dem

kein Kanalfehler im Hopping-Pattern existiert zu:

$$P_f = \frac{79 - f}{79}. \tag{5.7}$$

CLK27–0 wird korrekt berechnet, wenn für alle beobachteten Pakete kein Kanalfehler an der entsprechenden Position im Hopping-Pattern auftritt. Durch die Unabhängigkeit der Kanalfehler ist die Wahrscheinlichkeit hierfür:

$$P_{Erfolg} = {P_f}^n \tag{5.8}$$

$$P_{Erfolg} = \left(\frac{79 - f}{79}\right)^n. \tag{5.9}$$

Die Wahrscheinlichkeiten sind für verschiedene Kanalfehlerzahlen über der Anzahl notwendiger Pakete in Abbildung 5.1 dargestellt. Dort ist erkennbar, dass es selbst bei drei Kanalfehlern in der detektierten Kanalmaske in den meisten Fällen erreicht werden kann, dem Piconet zu folgen. Die dazu notwendige Berechnung von CLK27–0 benötigt in der Regel 10 korrekt empfangene Pakete, nur in seltenen Fällen werden über 20 Pakete benötigt. Selbst wenn die vom Sniffer erkannte Kanalmaske eine kleine Zahl von Kanalfehlern enthält, kann der Sniffer mit hoher Wahrscheinlichkeit dem beobachteten Piconet folgen.

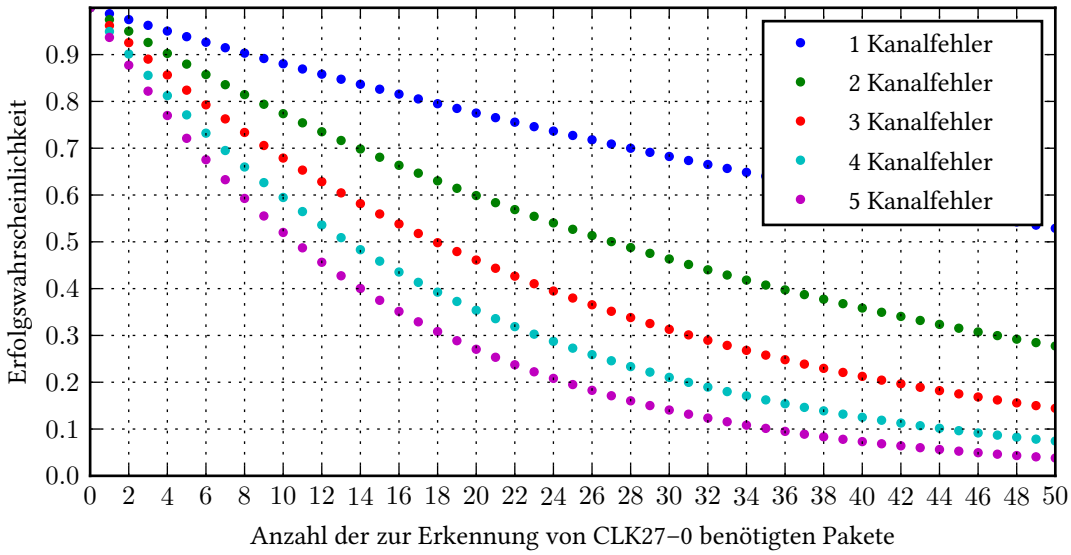


Abbildung 5.1: Erfolgswahrscheinlichkeiten für eine korrekte Erkennung von CLK27–0 bei einem bis fünf Kanalfehlern in der Kanalmaske.

5.1.4 Optimierung zur schnelleren initialen Erkennung der Kanalmaske

Da eine initiale Erkennung der AFH Channel Map ohne Kenntnis des Sprungmusters des Piconets nicht vermieden werden kann, lohnt es sich, auch hier zu optimieren. Beim Programmstart, wenn der Sniffer noch kein Paket vom beobachteten Piconet empfangen hat, ist jeder Kanal als ungenutzt markiert. Die Ergebnisse des Vorversuchs in Abschnitt 4.3 haben gezeigt, dass sich die Kanalmaske des Piconets abhängig von Störeinflüssen schnell ändern kann. Daher ist es wünschenswert, die Liste der genutzten Kanäle im Sniffer schnellstmöglich zu füllen.

Dafür kann man auf die Entdeckung von nicht mehr genutzten Kanälen verzichten und nur auf Kanälen lauschen, auf denen man bisher noch kein Paket empfangen hat. Das steigert die Wahrscheinlichkeit, hier ein Paket zu empfangen.

Da in realen Bluetooth-Verbindungen selten die ungenutzten Kanäle einzeln auftreten (vgl. Messergemnisse des Vorversuchs in Abschnitt 4.3), kann nach jedem neu erkannten genutzten Kanal die AFH Channel Map auf ungenutzte Kanäle geprüft werden, deren Nachbarkanäle beide genutzt sind. Diese Kanäle werden dann ebenfalls als genutzt markiert.

5.1.5 Abfangen eines LMP_set_AFH Pakets

Sobald die AFH Channel Map und damit das Adaptive Hopping-Pattern mit hinreichend wenigen Kanalfehlern bestimmt ist, kann mit geringem Paketverlust dem Piconet gefolgt werden. Somit kann auch mit hoher Wahrscheinlichkeit das bei Änderung der Kanalmaske gesendete LMP_set_AFH Paket abgefangen werden, das die neue AFH Channel Map enthält (s. Abschnitt 2.1.6). Sobald das passiert ist, kennt der Ubertooth die tatsächliche AFH Channel Map und kann dem Piconet ohne Paketverlust folgen. Dieses Verfahren unterscheidet sich insofern von allen bisher erklärten Verfahren, dass es nicht eine potentiell fehlerhafte Schätzung der Kanalmaske des Piconets liefert, sondern die tatsächlich vom Piconet verwendete Kanalmaske.

Ein LMP_set_AFH Paket wird vom Master eines Bluetooth-Piconets gesendet, sobald sich die Kanalmaske ändert oder sobald Adaptive Frequency Hopping aktiviert oder deaktiviert wird. Der Zeitpunkt eines solchen Ereignisses ist vom Sniffer nicht vorhersehbar. Ein erfolgreicher Empfang dieses Pakets setzt deshalb voraus, dass der Sniffer dem Piconet folgt und somit dessen Pakete mit einem möglichst geringen Paketverlust empfangen kann.

Damit dieses Verfahren genutzt werden kann, müssen vorher alle bisher beschriebenen Verfahren angewendet werden, um möglichst genaue Kenntnis der Kanalmaske zu erhalten und so die Wahrscheinlichkeit zu senken, das LMP_set_AFH Paket zu verpassen.

5.1.6 Gesamter Detektionsalgorithmus

Alle bis hierher beschriebenen Lösungen der Teilprobleme auf dem Weg zur Detektion der Kanalmaske werden nun in einem Detektionsalgorithmus zusammengeführt. Der gesamte Algorithmus besteht aus drei Phasen. Zunächst erfolgt die notwendige initiale Detektion der noch unbekannten Kanalmaske. Dabei kommen die beiden in Abschnitt 5.1.4 beschriebenen Optimierungen für eine möglichst schnelle initiale Erkennung zum Einsatz.

Nach einer einstellbaren Zeit wird in der zweiten Phase versucht, dem Piconet zu folgen. Da die erkannte AFH Channel Map Kanalfehler enthalten kann, sind dafür eventuell mehrere Versuche notwendig (vgl. Abschnitt 5.1.3). Sobald CLK27–0 des Piconets erkannt ist, wird der Ubertooth angewiesen, dem Piconet unter Verwendung des Basic Hopping Patterns zu folgen.

Phase drei überwacht die Kanalmaske auf Veränderungen. Kanäle, auf denen lange Zeit kein Paket empfangen wurde, werden wie in Abschnitt 5.1.2 als unbenutzt gekennzeichnet. Jedes empfangene Paket wird dekodiert und überprüft. Ist es ein LMP Paket mit dem Befehl LMP_set_AFH, wird die aktualisierte Kanalmaske des Piconets ausgelesen.

5.1.7 Fehlerbetrachtung

Bei der Implementierung des beschriebenen Detektionsalgorithmus hat sich gezeigt, dass das Ubertooth Projekt einen Fehler hat, der verursacht, dass das Folgen unzuverlässig arbeitet. Die Erkennung des Wertes für CLK27–0 und die Berechnung des Hopping Patterns erfolgt zwar korrekt und die Software meldet, dass der Ubertooth dem Piconet folgt. Ab diesem Zeitpunkt werden jedoch keine Pakete mehr empfangen.

Wird das Programm beendet, der Ubertooth jedoch nicht zurückgesetzt, inkrementiert dieser seinen Registerwert CLK27–0 weiterhin mit seinem internen Takt CLKN. Wird nun ein zweites mal versucht, dem selben Piconet zu folgen, sollte CLK27–0 des Sniffers

mit CLK27−0 des Piconets schon übereinstimmen, bevor libbtbb versucht, CLK27−0 des Piconets zu erkennen.

Bei der Erkennung des Wertes von CLK27−0 des Piconets durch die libbtbb gibt Software das Offset zwischen dem erkannten CLK27−0 Wert des Piconets und dem des Ubertooth aus. Dieses Offset ist häufig nicht wie erwartet 0 sondern eine andere Zahl zwischen -10 und 10. Wie diese Abweichung zustande kommt, konnte bis zum Abgabetermin nicht geklärt werden.

Da die Berechnung von CLK27−0 erfolgreich abgeschlossen wird, kann davon ausgegangen werden, dass das Hopping-Pattern durch den in Abschnitt 2.2.3 beschriebenen Algorithmus der libbtbb richtig berechnet wurde. Möglicherweise liefert jedoch die Berechnung von CLK27−0 in der libbtbb ein fehlerhaftes Ergebnis. Das ließe sich durch einen Vergleich des berechneten CLK27−0-Wertes mit dem tatsächlichen CLK27−0-Wert des Piconets überprüfen. Über eine Schnittstelle in BlueZ kann der aktuelle CLK27−0-Wert abgefragt werden. Falls sich herausstellen sollte, dass dieser Wert korrekt berechnet wird, ist es möglich, dass Verzögerungen bei der Übertragung des Offsets zum Ubertooth und dessen Verarbeitung in der Firmware den Fehler verursachen. Auch ein Fehler bei der Steuerung des Timers, der in der Ubertooth-Firmware Kanalsprünge koordiniert, kommt als Fehlerquelle in Frage.

Da dieser Fehler ein zuverlässiges Folgen unmöglich macht, werden die Fähigkeiten des Detektionsalgorithmus stark eingeschränkt. Zunächst kann nur ein willkürliches Sprungschema eingesetzt werden, das unabhängig von dem des Piconets ist. Ein hoher Paketverlust ist die Folge. Das LMP Paket mit dem Befehl LMP_set_AFH kann so nicht empfangen werden. Auch die Überwachung der Kanalmaske auf Änderungen muss mit dem Sprungschema der initialen Kanalmaskenerkennung stattfinden und läuft daher deutlich langsamer.

Im bisher nutzbaren Detektionsalgorithmus wird die zweite Phase, das Folgen, also übersprungen. Abbildung 5.2 zeigt den sich so ergebenden Programmablauf, der aus nur zwei Phasen besteht: Die initiale Kanalmaskenerkennung zu Programmstart und die Überwachung auf Kanalmaskenänderungen. Obwohl die Funktion zum Folgen eines Piconets und der Empfang des LMP Pakets nicht genutzt werden, wurden sie bereits im Quelltext implementiert. Die entsprechenden Passagen im Quelltext werden zurzeit im Programmablauf übersprungen. Sie können jedoch leicht aktiviert werden, sobald der Fehler behoben ist.

In Abschnitt 5.3 wird dieser Detektionsalgorithmus untersucht und seine Leistungsfähigkeit bewertet. Trotz aller genannten Nachteile zeigt sich, dass auch dieser eingeschränkte Ansatz zu brauchbaren Ergebnissen führen kann.

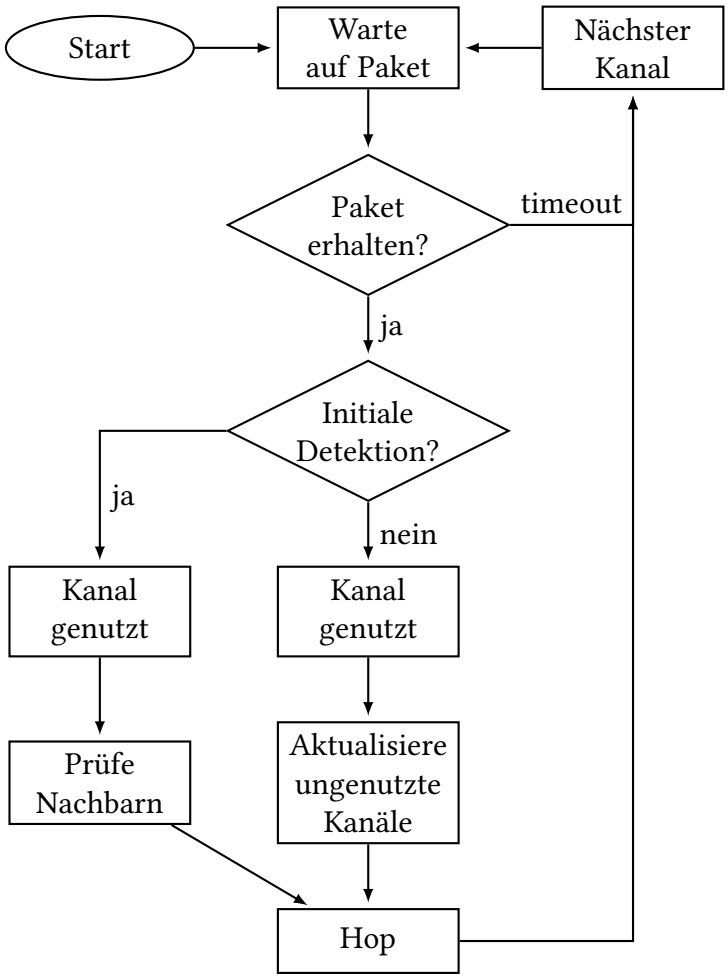


Abbildung 5.2: Ablauf der Kanalmaskendetektion

5.2 Implementierungen im Ubertooth Projekt

Die Implementierung der Algorithmen machen viele Änderungen an Firmware und Hostcode [37],sowie an der libbtbb [36] nötig. Die meisten der in Abschnitt 2.2 vorgestellten Programmteile waren gar nicht, nur unvollständig oder fehlerhaft auf Adaptive Frequency Hopping vorbereitet. Im Folgenden werden konkrete Änderungen vorgestellt.

5.2.1 Firmware des Mikrocontrollers

In der Firmware des Mikrocontrollers waren nur wenige Änderungen nötig. Der in der Firmware implementierte Hopping Kernel unterstützte Adaptive Frequency Hopping bereits. Auch alle interessanten Funktionen zum Aufzeichnen und Weiterleiten von empfangenen Paketen an den Hostrechner waren vorhanden und wurden für andere Betriebsarten genutzt. Neu hinzugekommen ist lediglich das neue Hopping-Schema für die Kanalmaskendetektion und zwei über den USB Control Channel übertragbare Befehle zum Übertragen der Kanalmaske an den Sniffer und zum sofortigen Auslösen eines Kanalsprungs.

Das neue Hopping-Pattern ist auf eine schnelle Erkennung der AFH Channel Map ausgelegt. Es lässt den cc2400, wie in Abschnitt 5.1.1 beschrieben, spätestens nach 158 Perioden von CLKN den Kanal wechseln. Dabei werden alle Kanäle nacheinander durchlaufen. Kanäle, die bereits vom Sniffer als genutzt erkannt wurden, werden allerdings übersprungen.

Ein neuer USB Control Channel Befehl dient dazu, dem Ubertooth-Sniffer eine neue AFH Channel Map mitzuteilen. Dabei werden immer alle 10 Bytes im Little-Endian-Format vom Host als Parameter übergeben. Die aktualisierte Channel Map wird dazu verwendet, um bei der Kanalmaskenerkennung genutzte Kanäle zu überspringen oder beim Folgen das adaptive Hopping-Pattern zu nutzen.

Der zweite neue Befehl löst sofort einen Kanalsprung aus. Er wird verwendet, um während der Detektion der Kanalmaske sofort auf den nächsten Kanal zu springen, sobald ein Paket auf dem aktuellen Kanal empfangen wurde.

5.2.2 libbtbb

Die libbtbb hatte noch keine Unterstützung für Adaptive Frequency Hopping. Hier musste zunächst der Hopping Kernel angepasst werden, um das Adaptive Hopping Pattern richtig zu berechnen. Außerdem wurde eine Gettermethode geschrieben, mit der abgefragt werden kann, ob ein bestimmter Kanal genutzt wird, sowie zwei Settermethoden, um einen bestimmten Kanal als genutzt oder ungenutzt zu klassifizieren.

5.2.3 Hostcode

Die meisten und umfangreichsten Änderungen fielen im Hostcode an. Zunächst wurde das neue Hostprogramm *ubertooth-afh* entwickelt. Es ähnelt im Aufbau dem schon vorhandenen passiven Snifferprogramm *ubertooth-rx*. Im Gegensatz dazu ist jedoch nicht das Ziel, Bluetooth-Pakete zu dekodieren und auszugeben, vielmehr ist nur der Kanal eines Bluetooth-Pakets interessant, das zum beobachteten Piconet gehört.

Das Programm besteht aus drei Abschnitten. Es versetzt den Ubertooth zunächst in den Modus für die initiale Entdeckung der AFH Channel Map, sodass er das in Abschnitt 5.1.1 beschriebene Hopping Pattern verwendet. Wie beim Sniffen wird der empfangene Datenstrom auf Bluetooth-Pakete des gewünschten Piconets durchsucht. Im Erfolgsfall wird der entsprechende Kanal als genutzt markiert und die Kanalmaske angepasst. Dabei werden die Optimierungen aus Abschnitt 5.1.4 genutzt. Die aktualisierte Kanalmaske wird dem Ubertooth über den USB Control Channel übermittelt. Zum Schluss wird über den entsprechenden Befehl ein Kanalsprung im Ubertooth ausgelöst. So wird initial die AFH Channel Map gefüllt. Nach einem einstellbaren Timeout endet der erste Abschnitt.

Der zweite Abschnitt soll versuchen, mit Hilfe der ermittelten AFH Channel Map das Adaptive Hopping-Pattern zu berechnen, CLK27–0 des Piconets zu bestimmen und dem Piconet zu folgen. Dabei soll er allerdings, wie in Abschnitt 5.1.3 beschrieben, das Basic Hopping-Pattern verwenden. Dieser Programmabschnitt wird zurzeit aufgrund des in Abschnitt 5.1.7 beschriebenen Fehlers immer übersprungen.

Nun beginnt der dritte Abschnitt, in dem die Kanalmaske auf Änderungen überwacht wird. Jeder Kanal erhält dafür einen eigenen Zähler. Dieser wird zurückgesetzt, wenn auf diesem Kanal ein Paket empfangen wird. Wenn auf einem anderen Kanal ein Paket empfangen wird, so wird er inkrementiert. Übersteigt der Zähler eines Kanals einen Schwellwert, wird dieser Kanal vom Ubertooth als nicht mehr genutzt angesehen. Der

Schwellwert ergibt sich aus der in Abschnitt 5.1.2 eingeführten Schwellwahrscheinlichkeit. Diese Wahrscheinlichkeit wird zu Programmstart per Kommandozeilenparameter übergeben.

Wie auch im ersten Abschnitt wird der empfangene Bitstrom auf Bluetooth-Pakete durchsucht, die zum beobachteten Piconet gehören. Wird ein Paket empfangen, so wird der entsprechende Kanal als genutzt markiert und sein Zähler zurückgesetzt. Für alle anderen Kanäle wird der jeweilige Zähler inkrementiert. Übersteigt der Zähler eines Kanals den Schwellwert, wird der jeweilige Kanal als unbenutzt markiert. So können Änderungen an der AFH Channel Map festgestellt werden. Nach jeder Änderung wird die AFH Channel Map mit einem aktuellen Zeitstempel ausgegeben.

Die drei Programmabschnitte können unabhängig voneinander übersprungen werden. Ist zum Beispiel die Kanalmaske zu Programmstart bekannt, kann diese dem Programm übergeben werden. Dann wird der erste Abschnitt übersprungen und sofort mit dem nächsten begonnen.

Soll hingegen nur die Kanalmaske möglichst schnell bestimmt, aber nicht auf Änderungen überwacht werden, kann für die initiale Kanalmaskenerkennung ein Timeout von Null übergeben werden. Somit wird nur der erste Abschnitt ausgeführt.

5.2.4 libubertooth

Die Vorgehensweise beim Detektieren der AFH Channel Map unterscheidet sich in einigen Punkten deutlich von allen bisher im Ubertooth Projekt implementieren Funktionen. So mussten neue Funktionen zum Umgang mit den über USB empfangenen Daten implementiert und in den Programmablauf eingebunden werden.

Da der ursprüngliche Hostcode auf ein Ziel hin entwickelt wurde — das Sniffen von Bluetooth-Paketen — war er monolithisch aufgebaut. Die Funktion, die den Empfang von USB-Paketen behandelt, ruft beispielsweise automatisch die Methode zum Decodieren des Bluetooth-Pakets auf. Diese Struktur wurde aufgebrochen. Die Callback-Funktion, die empfange Daten weiter behandelt, wird nun im jeweiligen Hostprogramm definiert. Die Initiierung des USB-Transfers wird ebenfalls im Hauptprogramm aufgerufen.

Eine weitere Änderung ist die neue Unterstützung für mehrere Ubertooth Sticks innerhalb eines Programms. Bisher war der Zustand des angesteuerten Ubertooths in globalen Variablen in der Bibliothek gespeichert. Das hat zur Folge, dass jedes Programm

nur einen globalen Ubertooth Sniffer ansprechen kann. Im Zuge der Änderungen wurde eine neue Datenstruktur erstellt, die einen Ubertooth Stick repräsentiert und seinen Zustand speichert. Sämtliche Funktionen der Bibliothek wurden umgebaut, um mit den Variablen in der Struktur zu arbeiten, statt mit den globalen Variablen.

Auch der Ringbuffer, der empfangene USB-Pakete speichert, war bisher als globale Variable in Form eines Arrays ausgeführt. Verschieden Funktionen griffen lesend und schreibend auf das Array zu, ohne dass ein Lese- und Schreibpointer gespeichert wurde. Das macht das fehlerhafte Überschreiben von Daten bei Programmänderungen sehr wahrscheinlich. Deshalb wurde eine Datenstruktur eines Ringbuffers mit entsprechenden Methoden implementiert. Diese Struktur ist von außen nur über Methoden lesbar und schreibbar, die gleichzeitig die Lese- und Schreibpointer anpassen. Weitere Methoden geben Auskunft, ob der Puffer voll oder leer ist.

Diese Maßnahmen zur Modularisierung der Bibliothek vereinfachen die Entwicklung neuer Funktionen. Die so erreichte Wiederverwendbarkeit von schon bestehenden Bibliotheksteilen verkürzt den neu zu schreibenden Code, was der Lesbarkeit und der Vermeidung von Fehlern zugute kommt.

Bei ersten Versuchen mit dem Ubertooth stellte sich heraus, dass die Erkennung von CLK27–0 eines Piconets nach der in Abschnitt 2.2.3 beschriebenen Methode sehr lange dauert und häufig mit einem Fehler abbricht. Dieser Fehler ist unabhängig von dem in Abschnitt 5.1.7 beschriebenen Problem, das erst nach der Erkennung von CLK27–0 auftritt. Solche Fehler sind ungewöhnlich, da das Hopping-Pattern und die Kanalreihenfolge deterministisch ist. Eine Fehlersuche deckte ein Problem bei der Übergabe vom Wert von CLK27–0 eines USB-Paketes an die Berechnung des Offsetts auf.

Bei der Suche nach Bluetooth-Paketen im USB-Bitstrom wird das Bit-Offset des Bluetooth-Pakets berechnet (s. Abschnitt 2.2.2). Als CLK27–0 wurde allerdings immer der entsprechende Wert übernommen, der zum Zeitpunkt des ersten Bits des USB-Pakets gehört. Bei der bei Bluetooth verwendeten Senderate von 1 MBit/s hat ein Bit eine Dauer von 1 µs. Betrachtet man den Wert von CLK100NS des Pakets, entspricht das 10 Einheiten von 100 ns. Das Bit-Offset lässt sich also in ein zeitliches Offset umwandeln.

Für die Berechnung des Hopping-Patterns ist der Wert von CLK27–0 entscheidend. Dieser Wert wird im Hostprogramm aus CLK100NS des USB-Pakets berechnet. Wenn das zeitliche Offset dabei nicht berücksichtigt wird, passiert es häufig, dass CLK27–0 des Pakets falsch berechnet wird. Damit stimmt die Differenz zwischen CLK27–0 des ersten empfangenen Pakets und des aktuellen Pakets nicht. Das Paket würde in den falschen Slot eingeordnet werden. Der richtige Startpunkt im Hopping-Pattern wird so aussortiert.

Das Berücksichtigen des zeitlichen Offsets der Bits verbessert die Berechnung des Startpunktes enorm. In den meisten Fällen kann CLK27–0 des Piconets schon nach weniger als 10 beobachteten Paketen richtig berechnet werden.

5.3 Messung der Leistungsfähigkeit des Detektionsalgorithmus

In diesem Abschnitt soll die Leistungsfähigkeit des implementierten Detektionsalgorithmus gemessen und bewertet werden. Dazu wird der in Kapitel 4 beschriebene Messaufbau genutzt, der schon im Vorversuch zum Einsatz kam. Der Detektionsalgorithmus bestimmt die Kanalmaske einer Audioverbindung zwischen Mobiltelefon und Computer. Die Audioverbindung nutzt dabei das *Advanced Audio Distribution Profile (A2DP)* [12]. Zum Einsatz kommt auf beiden Geräten die aktuelle Version des Bluetooth-Stacks BlueZ 5, sowie der Soundserver Pulseaudio 6. Der Funkverkehr wird kontinuierlich mit einer WLAN-Verbindung auf Kanal 5 (2,432 GHz) gestört. Das störende WLAN hat sein Zentrum bei Bluetooth-Kanal 30 und erstreckt sich von Kanal 20 bis Kanal 40. Ansonsten befindet sich keine Störer in der Umgebung. Die konstant hohe und scharf begrenzte Störung auf diesem Kanal führt zu einer über den gesamten Messzeitraum konstanten Kanalmaske. Die tatsächliche Kanalmaske der Bluetooth-Verbindung wird, wie in Abschnitt 5.1.5 beschrieben, über eine Schnittstelle in BlueZ überwacht und ausgegeben und graphisch dargestellt, um sie mit der vom Sniffer detektierten Kanalmaske zu vergleichen.

Zunächst wird untersucht, wie lange es dauert, bis alle genutzten Kanäle in der Kanalmaske bestimmt sind. Dabei soll auch der Einfluss der Optimierungen bei der initialen Kanalmaskenerkennung aus Abschnitt 5.1.4 betrachtet werden. Im Anschluss wird die Erkennung von nicht mehr verwendeten Kanälen getestet. Zuletzt wird eine Bewertung und Diskussion der Ergebnisse vorgenommen.

5.3.1 Erkennung genutzter Kanäle

Zunächst wurde bestimmt, wie lange es dauert, bis die AFH Channel Map bestimmt ist. Dazu wurde die initiale Erkennung der Kanalmaske gestartet und nach jedem erkannten Kanal die aktuell vom Sniffer bestimmte AFH Channel Map ausgegeben. Diese nährte sich schrittweise der tatsächlichen AFH Channel Map an. Es wurde die Zeit

gestoppt, bis die vom Sniffer erkannte und die tatsächliche Kanalmaske übereinstimmten. Zusätzlich wurden die Zeiten gemessen, bei denen sich beide Kanalmasken in nur einem, beziehungsweise zwei Bit unterschieden.

Für die Bestimmung wurden die Optimierungen für eine initiale Kanalmaskendetektion gemäß Abschnitt 5.1.2 genutzt. Als Vergleich wurde der Algorithmus ohne die Optimierungen aber mit gleichem Sprungschema herangezogen. Es wurde jeweils in 90 Durchläufen die AFH Channel Map bestimmt und die dafür notwendige Dauer gemessen, um ein aussagekräftiges Ergebnis zu erhalten. Tabelle 5.3.1 stellt die gemessene mittlere Detektionsdauer bei Verwendung des optimierten und des unoptimierten Algorithmus gegenüber.

	Detektionsdauer	
	Ohne Optimierung	Mit Optimierung
0 Kanalfehler	44,44 s	26,16 s
1 Kanalfehler	33,67 s	22,27 s
2 Kanalfehler	26,58 s	18,35 s

Tabelle 5.1: Mittlere Detektionsdauer bei Verwendung des optimierten und des nicht optimierten Algorithmus

Abbildung 5.3 zeigt die Verteilung der Detektionsdauer über allen 90 gemessenen Werten. Dabei werden jeweils die Histogramme der Messungen ohne Optimierung und der Messung mit Optimierung gegenübergestellt. Es zeigt sich, dass die Optimierungen für eine initiale Kanalmaskendetektion die benötigte Zeit um etwa ein Drittel senken können. Die Verkürzung ist besonders stark bei einer fehlerfreien Erkennung der Kanalmaske ausgeprägt. Hier ist die Detektionszeit um über 40% verkürzt. Bei der Erkennung der Kanalmaske bis auf zwei Kanalfehler beträgt die Verkürzung der Detektionsdauer etwa 30%. Somit kann die AFH Channel Map im Mittel nach 26 Sekunden fehlerfrei. Eine detektierte Kanalmaske, die sich in nur zwei Bits von der tatsächlichen unterscheidet, erhält man im Mittel nach 18 Sekunden.

5.3.2 Erkennung nicht mehr genutzter Kanäle

Weiterhin wurde das Verhalten des in Abschnitt 5.1.1 entwickelten Programmabschnitts untersucht, der Änderungen der AFH Channel Map beobachten soll. Dazu wurde eine plötzliche Änderung der AFH Channel Map provoziert, indem das störende WLAN seinen Kanal wechselte. Die Optimierung für die initiale Kanalmaskenerkennung zu Programmstart war für 20 Sekunden aktiviert. Es wurden verschiedene

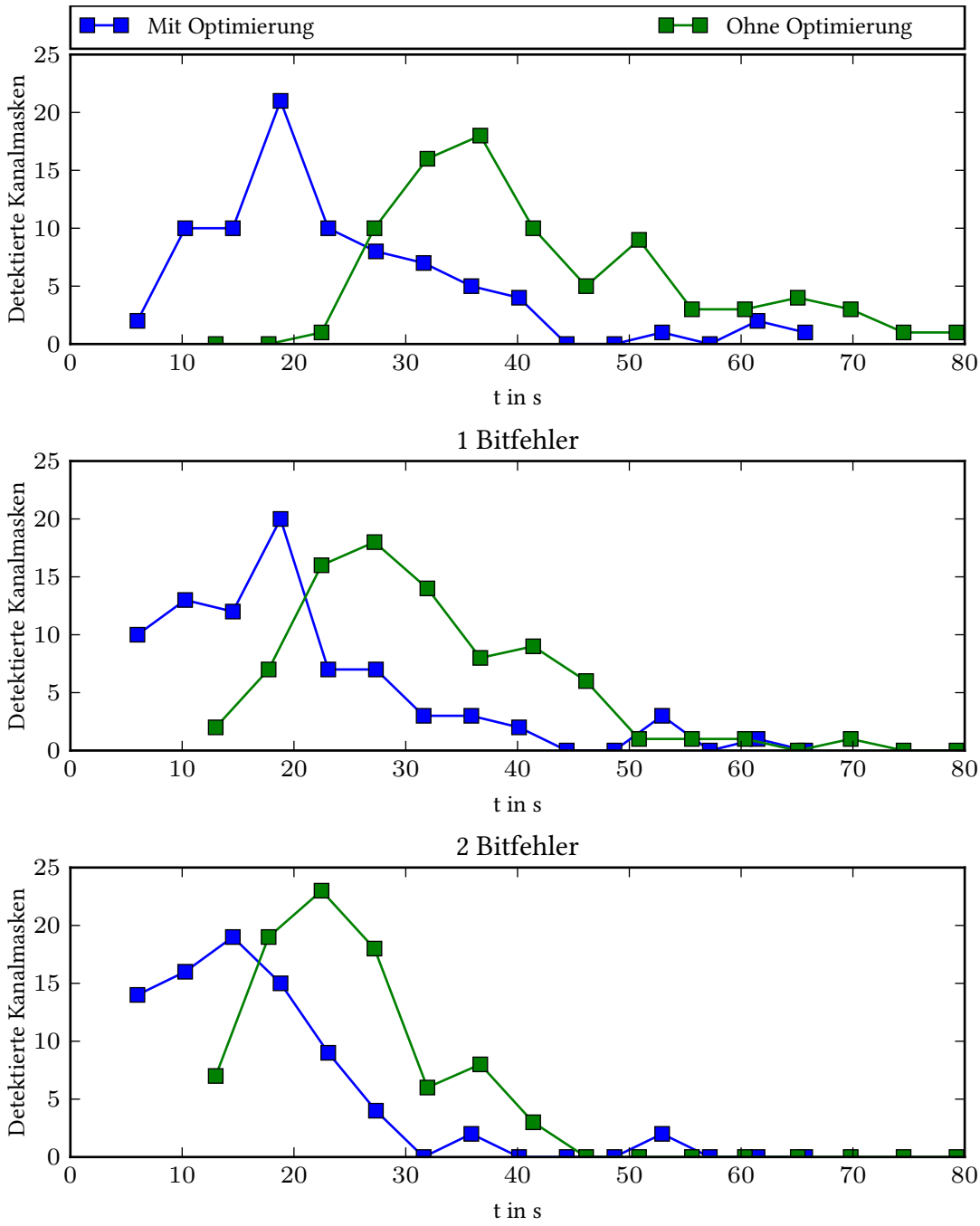


Abbildung 5.3: Gegenüberstellung der gemessenen Dauer ohne Optimierung und mit Optimierung für verschiedene Kanalfehlerzahlen

Messungen durchgeführt, bei der die Schwellwahrscheinlichkeit variiert wurde, ab der der Sniffer einen Kanal als ungenutzt erkennt.

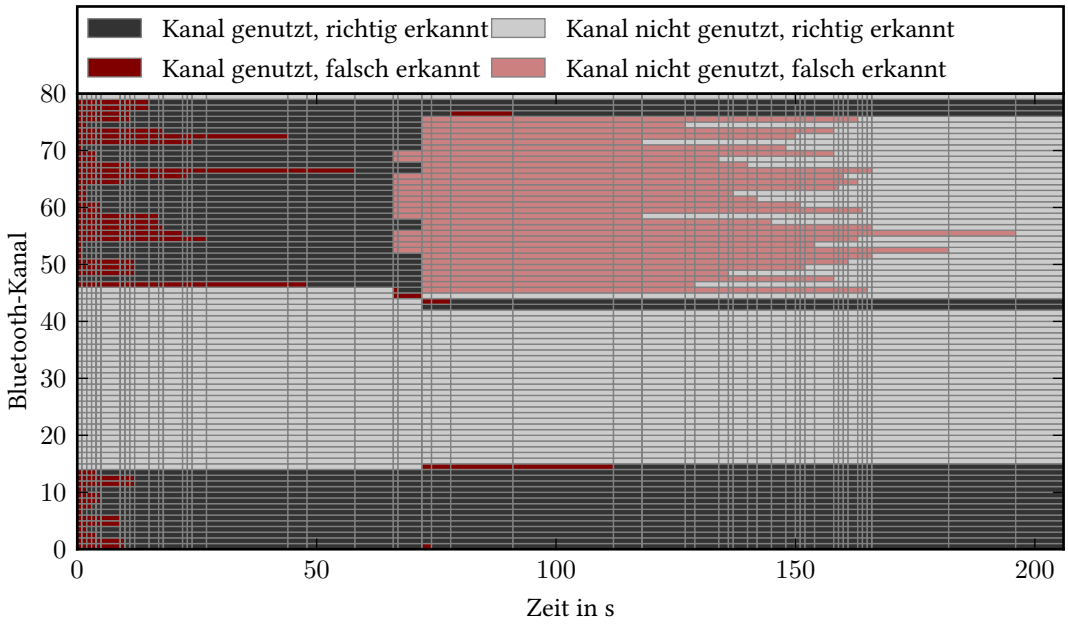
Die Abbildung 5.4 zeigt den Vorgang der passiven Detektion und anschließender Überwachung der AFH Channel Map. Dabei wurde die in Abschnitt 4.1.3 vorgestellte Methode zur graphischen Darstellung verwendet. Kanäle, die vom Piconet genutzt werden, sind dunkel eingefärbt, ungenutzte Kanäle erscheinen hell. Stimmt die Beobachtung des Ubertooths bei einem Kanal nicht mit der tatsächlichen AFH Channel Map überein, so ist der entsprechende Kanal für diesen Zeitraum mit einem Rotton überlagert.

Im oberen Teil von Abbildung 5.4 beträgt die Schwellwahrscheinlichkeit 95%. Hier ist am linken Bildrand die initiale Kanalerkennung zu sehen. Zunächst werden alle Kanäle als unbenutzt betrachtet. Mit der Zeit werden immer mehr Kanäle als benutzt erkannt. Nach etwa 25 Sekunden ist die AFH Channel Map bis auf drei Kanalfehler erkannt worden. Nach etwa einer Minute ist die Detektion vollständig abgeschlossen. Die schlagartige Änderung der AFH Channel Map des Piconets findet etwa 70 Sekunden nach Programmstart statt. Zu diesem Zeitpunkt wechselt das störende WLAN auf Kanal 11 (2.467 GHz, Bluetooth-Kanal 65). Danach korrigiert der Sniffer nacheinander alle Kanäle seiner Kanalmaskenschätzung, die von der tatsächlichen Kanalmaske des Piconets abweichen. Nach etwa zwei Minuten ist die Anpassung abgeschlossen.

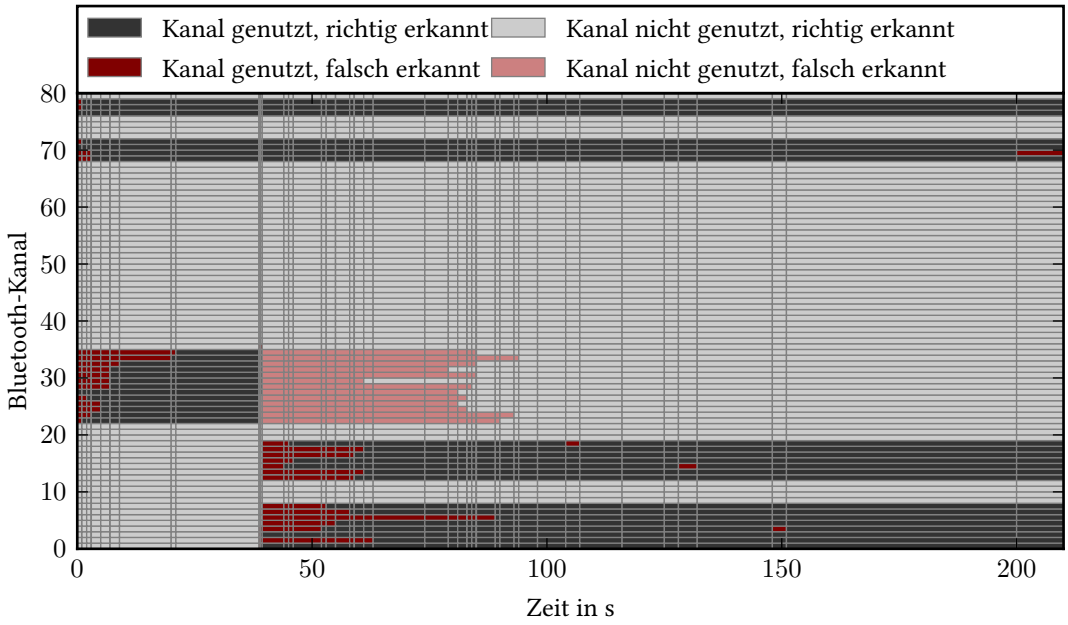
Im unteren Teil von Abbildung 5.4 beträgt die Schwellwahrscheinlichkeit 75%. Hier wechselt der WLAN-Störer etwa 40 Sekunden nach Programmstart von WLAN-Kanal 1 auf WLAN-Kanal 5. Auffällig ist, dass hier schon nach etwa einer Minute sämtliche nicht mehr vom Piconet genutzten Kanäle vom Sniffer als ungenutzt erkannt sind. Bei einer Schwellwahrscheinlichkeit von 75% wird also im Vergleich zu einer Schwellwahrscheinlichkeit von 95% nur die halbe Zeit benötigt. Die zweite Auffälligkeit sind die Kanäle, die der Sniffer 105, 130 und 150 Sekunden nach Programmstart als unbenutzt markiert, obwohl das Piconet sie benutzt. Zu diesen Zeitpunkten wurde auf den Kanälen zu lange kein Paket mehr empfangen. Nur wenige Sekunden später wird dieser Fehler korrigiert. Hier zeigt sich, dass die niedrigere Schwellwahrscheinlichkeit zu kurzzeitigen Fehlern führen kann.

5.3.3 Bewertung des Detektionsalgorithmus

Der implementierte Detektionsalgorithmus ist in der Lage, im Mittel innerhalb von 26 Sekunden nach Programmstart die Kanalmaske eines Piconets fehlerfrei zu erkennen. Treten Änderungen in der Kanalmaske des Piconets auf, sodass einzelne, zuvor



(a) Schwellwahrscheinlichkeit 95%



(b) Schwellwahrscheinlichkeit 75%

Abbildung 5.4: Die Detektionsgeschwindigkeit bei Änderungen in der Kanalmaske ist stark abhängig von der gewählten Schwellwahrscheinlichkeit, ab der ein Kanal vom Sniffer als unbenutzt angesehen wird.

genutzte Kanäle nicht mehr genutzt wenden, passt der Sniffer seine Kanalmaske bei einer Schwellwahrscheinlichkeit von 95% innerhalb von etwa zwei Minuten an.

Wenn einzelne Kanalfehler bei der Kanalmaskendetektion in Kauf genommen werden, kann die benötigte Detektionszeit verkürzt werden. Eine Kanalmaske, die höchstens zwei Kanalfehler enthält, ist im Mittel 18 Sekunden nach Programmstart bestimmt. Bei Änderungen der Kanalmaske des Piconets werden unbenutzte Kanäle bei einer Schwellwahrscheinlichkeit von 75% nach etwa einer Minute vom Sniffer als ungenutzt erkannt.

Die in Abschnitt 5.1.4 entwickelten Optimierungen bei der initialen Kanalmaskenerkennung verkürzen die Detektionszeit um etwa ein Drittel. Bedeutend kürzere Detektionszeiten ließen sich realisieren, wenn der Ubertooth-Sniffer das selbe Hopping Schema verwenden würde, das auch das Piconet benutzt. Ein solcher Ansatz wird zurzeit durch den in Abschnitt 5.1.7 beschriebenen Fehler im Ubertooth-Projekt verhindert. Unter dem Gesichtspunkt, dass der Sniffer ein Hopping-Schema verwendet, das von dem des Piconets abweicht und dem Piconet nicht folgen kann, ist eine derart kurze Detektionszeit beachtlich. Wie in Kapitel 4 aufgezeigt, ändert sich in vielen Fällen die Kanalmaske deutlich langsamer, sodass der Detektionsalgorithmus sie erfolgreich bestimmen kann.

In Umgebungen mit starken, wechselnden Störungen ändert sich die Kanalmaske schneller. Es wurde eine Vergleichsmessung unter solchen Bedingungen durchgeführt. Dazu wurde eine Bluetooth-Audio-Verbindung zwischen Computer und Mobiltelefon in einem Mehrfamilienhaus betrieben, in dem das gesamte 2,4 GHz Band von den WLAN-Netzen der Mieter gestört ist. Der Sniffer wurde angewiesen, die Kanalmaske der Audioverbindung zu detektieren. Dabei wurden die Optimierungen zur initialen Erkennung der Kanalmaske die ersten 20 Sekunden nach Programmstart aktiviert. Um eine schnelle Erkennung der vom Piconet nicht mehr genutzten Kanäle zu gewährleisten, wurde die Schwellwahrscheinlichkeit auf 75% gesetzt. In Abbildung 5.5 zeigen sich die Grenzen des Detektionsalgorithmus. Die Kanalmaske des beobachteten Piconets ändert sich so häufig, dass die Kanalmaske des Ubertooth-Sniffers nie vollkommen mit ihr übereinstimmt.

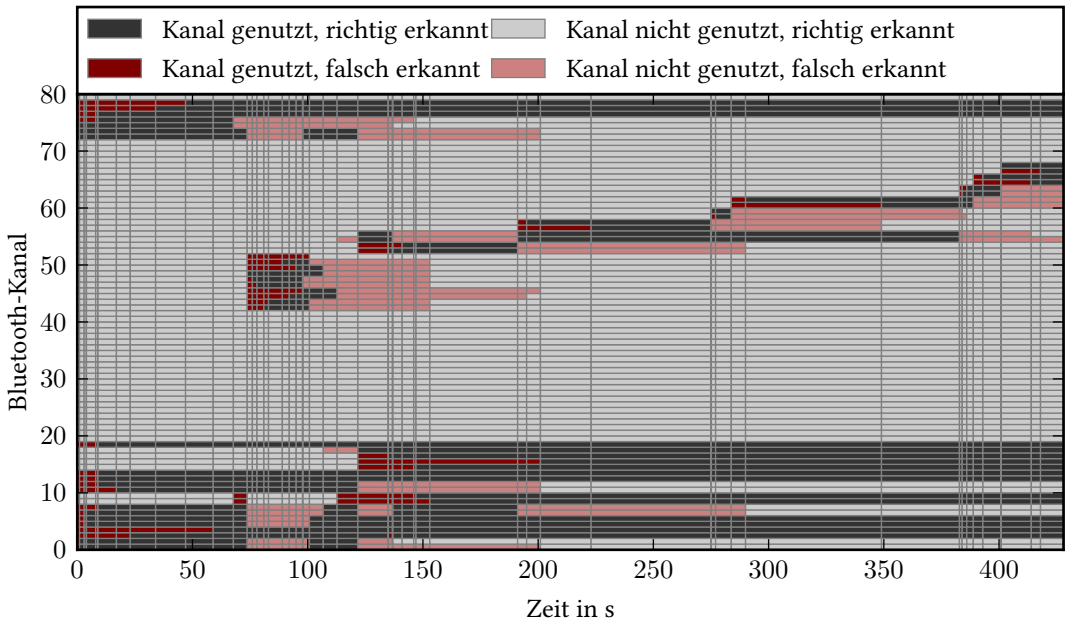


Abbildung 5.5: Bei starken, wechselnden Störquellen ändert sich die Kanalmaske häufig. Die Geschwindigkeit des Detektionsalgorithmus genügt nicht, um Änderungen der Kanalmaske zu erkennen, bevor sich diese erneut ändert.

6 Auswertung

Die Messungen der Funktionalität des implementierten Algorithmus zeigen, dass es prinzipiell möglich ist, die Kanalmaske eines Bluetooth-Piconets mit einem passiven Bluetooth-Sniffer zu detektieren. Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und nennt die dabei erreichten Ziele.

Die in der Bluetooth-Spezifizierung beschriebenen Mechanismen zum Adaptive Frequency Hopping wurden in Kapitel 2 vorgestellt. Sie bieten mehrere Ansatzpunkte für eine passive Kanalmaskendetektion. In einem Vorversuch wurde deutlich, dass die Kanalmaske unterschiedlich auf verschiedene Störquellen reagiert und dass in der Praxis häufig Bedingungen herrschen, die eine passive Detektion der AFH Channel Map ermöglichen.

Unter den in Kapitel 3 festgelegten Voraussetzungen wurde in Kapitel 5 ein Algorithmus entworfen und auf dem quelloffenen Bluetooth-Sniffer Ubertooth implementiert, der in der Lage ist, die Kanalmaske einer beobachteten Bluetooth-Verbindung zu erkennen und auf Veränderungen zu überwachen. Aus den Erkenntnissen des in Kapitel 4 beschriebenen Vorversuchs konnten dabei Optimierungen abgeleitet werden, die die Kanalmaskendetektion deutlich beschleunigen.

Die initiale Detektion der Kanalmaske nach dem Programmstart war in den betrachteten Fällen im Mittel innerhalb von 26 Sekunden abgeschlossen. Die darauf folgende Überwachung auf Veränderung der Kanalmaske benötigt etwa zwei Minuten, um festzustellen, dass ein zuvor genutzter Kanal nicht mehr genutzt wird. Diese kurze Detektionszeit ist beachtlich für eine rein passive Kanalmaskendetektion, reicht aber in manchen Fällen nicht für eine fehlerfreie Detektion aus.

Das implementierte Verfahren eignet sich für Anwendungsfälle, in denen sich die Kanalmaske selten ändert. Das ist einerseits der Fall, wenn die beobachtete Bluetooth-Verbindung von einer starken, stetigen und im Frequenzbereich scharf begrenzten Störquelle beeinträchtigt wird. Andererseits gibt es Bluetooth-Geräte wie den in Kapitel 4.3 genannten Bluetooth-Ethernet-Umsetzer, dessen Kanalmaske fest konfiguriert wird und nicht auf Störungen reagiert. Hier funktioniert der Detektionsalgorithmus

sehr gut und führt schnell zu korrekten Ergebnissen. In Umgebungen mit breitbandiger oder schnell wechselnder Störung kann der Detektionsalgorithmus Änderungen der Kanalmaske des beobachteten Piconets nicht genügend schnell erkennen.

7 Ausblick

In dieser Arbeit wurde das Problem einer passiven Detektion von Bluetooth-Kanalmasken unter schmal umrissenen Bedingungen betrachtet. Damit ist das Thema jedoch keineswegs erschöpfend erforscht. Dieses Kapitel gibt einen Ausblick auf weiterführende Arbeiten, die weitere, auf dieser Arbeit aufbauende Erkenntnisse liefern könnten.

Im Vorversuch dieser Arbeit und bei der Entwicklung und Evaluation des Detektionsalgorithmus kamen ausschließlich Bluetooth-Geräte zum Einsatz, die auf dem Bluetooth-Stack BlueZ aufbauen. Die Klassifikationen und damit die Reaktion der Kanalmaske eines Piconets auf äußere Störquellen ist im Bluetooth-Standard nicht abschließend spezifiziert und hängt von der jeweiligen Implementierung ab. Die Ergebnisse dieser Arbeit sind nicht notwendigerweise auf andere Implementierungen übertragbar. Die interessante Frage, wie die Kanalmaske bei der Verwendung anderer Bluetooth-Implementierungen auf Störungen reagiert, könnte Thema einer weiterführenden Arbeit sein.

Die in dieser Arbeit nicht betrachtete Technik Bluetooth Low Energy verwendet ebenfalls Adaptive Frequency Hopping. Die Abläufe dabei sind prinzipiell ähnlich zu früheren Bluetooth-Revisionen. Somit lassen sich die Erkenntnisse dieser Arbeit auf BLE übertragen. Da der Ubertooth-Sniffer Unterstützung für Bluetooth Low Energy bietet, ist es möglich, den implementierten Detektionsalgorithmus mit wenigen Quelltextänderungen für diese Technik zu verwenden. Bei der Detektion der Kanalmaske einer BLE-Verbindung ist allerdings mit Einschränkungen zu rechnen. Bluetooth Low Energy wurde entwickelt, um den Energiebedarf von Bluetooth-Geräten zu senken. Das wird durch kürzere Paketlängen und schnellere Verbindungsprozeduren erreicht. Geräte, die BLE verwenden, sparen häufig auch dadurch Energie, dass sie nur selten Pakete versenden um längere Zeit im Energiesparmodus zu verweilen. Der Detektionsalgorithmus ist besonders bei der initialen Kanalmaskenerkennung darauf angewiesen, dass er möglichst viele Pakete vom beobachteten Piconet empfangen kann. Werden wenig Pakete gesendet, dauert die Kanalmaskendetektion entsprechend länger.

Auch der implementierte Detektionsalgorithmus kann weiter optimiert werden. Da das Projekt Ubertooth zum aktuellen Zeitpunkt[1] unter dem in Abschnitt 5.1.7 beschriebenen Fehler leidet, der es unmöglich macht, den Kanalsprüngen des beobachteten Piconets zuverlässig zu folgen, nutzt der Detektionsalgorithmus in seiner bisherigen Form ein eigenes langsames Hopping-Pattern, das von dem des Piconets unabhängig ist. Daher werden viele der übertragenen Bluetooth-Pakete vom Sniffer nicht empfangen, was die Erkennung von genutzten und ungenutzten Kanälen stark verlangsamt. Es ist zu erwarten, dass die Erkennung von nicht mehr genutzten Kanälen von aktuell zwei Minuten auf wenige Sekunden verkürzt werden kann, wenn der Sniffer dem Hopping-Pattern des Piconets korrekt folgt. Im Zuge dieser Arbeit wurde der Detektionsalgorithmus schon darauf vorbereitet, einem Piconet folgen zu können. Sobald der Fehler im Ubertooth-Projekt gefunden und behoben wurde, kann es also von dieser Funktion profitieren.

Sobald der Sniffer dem beobachteten Piconet folgt, kann das LMP-Paket abgefangen werden, das bei Änderungen der AFH Channel Map die Slaves über die neue Kanalmaske informiert (s. Abschnitt 5.1.5). Dadurch erhält der Sniffer sofort Kenntnis der tatsächliche Kanalmaske des Piconets. Das vermeidet Kanalfehler, wie sie bei der klassischen passiven Detektion vorkommen können und führt somit zu zuverlässigeren und schnelleren Ergebnissen, als der aktuell genutzte Detektionsalgorithmus. Entsprechende Methoden für das Decodieren von LMP-Paketen sind bereits im Quelltext implementiert. Für die Nutzung dieses Ansatzes ist es jedoch nötig, dass der Sniffer dem beobachteten Piconet korrekt folgen kann.

Deshalb hat auch der bisher genutzte Ansatz seine Berechtigung. Um einem Hopping Pattern des Piconets zu folgen, ist die vollständige Kenntnis der AFH Channel Map notwendig, da das Hopping Pattern von der Kanalmaske abhängig ist. Also muss die Kanalmaskendetektion in jedem Fall mit dem vom beobachteten Piconet unabhängigen Hopping-Pattern funktionieren. Die in Abschnitt 5.1.4 entwickelten Optimierungen beschleunigen dabei die initiale Detektion. Hier ist eine weitere Optimierung möglich: Bei der initialen Kanalmaskendetektion wird aktuell ein Hopping-Pattern verwendet, bei dem sämtliche Bluetooth-Kanäle nacheinander durchlaufen werden, wobei 98,75 ms auf jedem Kanal verweilt wird. Das entspricht der Zeit, in der zweimal 79 Kanalsprünge durchgeführt werden. Der Einfluss einer Variation dieser Wartezeit auf die nötige Detektionszeit wurde in dieser Arbeit nicht untersucht. Es ist denkbar, dass sich bei der initialen Kanalmaskendetektion die mittlere Detektionszeit durch eine längere oder kürzere Kanalwartezeit weiter verkürzen lässt. Da eine kurze initiale Detektionszeit besonders bei der Beobachtungen von Piconets mit einer sich häufig ändernden Kanalmaske sehr wichtig ist, ist eine dahingehende Optimierung ein interessantes Thema für weiterführende Versuche.

[1] Commit 4ff64e4aca1e47bd653a6d6d6e76f4cee8c7b539 im Git Repository des Ubertooth Projekts [37]

Die Bluetooth-Spezifikation ermöglicht es, die Nutzdaten von Bluetooth-Paketen, auch die des LMP-Pakets, mit PSK zu modulieren, um eine höhere Bitrate zu erreichen [29]. Der auf dem Ubertooth-Sniffer verbaute Transceiver unterstützt diese Modulationsart nicht. Um auch Pakete mit dieser Modulation empfangen zu können, muss in einer zukünftigen Hardwareversion des Ubertooth-Sniffers ein Transceiver verbaut werden, der PSK unterstützt. Zum aktuellen Zeitpunkt ist dies nicht in Planung. Da die Hardware des Ubertooth-Projekts ebenfalls quelloffen ist und die Stromlaufpläne frei verfügbar sind, ist es jedoch möglich, in einem weiterführenden Projekt die Hardware zu modifizieren und den Transceiver zu ersetzen.

Neben der Implementierung des Detektionsalgorithmus wurde die Softwarebibliothek zur Kommunikation zwischen Hostrechner und Sniffer stark modularisiert. Das vereinfachte einerseits den Einbau der für die Kanalmaskenerkennung nötigen neuen Programmteile, andererseits vereinfacht es zukünftige Änderungen am Quellcode. Da das Projekt Ubertooth ein quelloffenes Projekt ist, das über Github verwaltet wird, wird ein Großteil der Änderungen am Code in den Hauptentwicklungszweig einfließen.

8 Einsatz der aktuellen Implementierung

Wie in Kapitel 1 beschrieben, entwickelt das Fraunhofer Institut für integrierte Schaltungen Systeme zur Fehlerdiagnose von drahtlosen Funksystemen. Eine Komponente davon ist ein Bluetooth-Sniffer zum Analysieren von Bluetooth-Verbindungen. Dieser wurde um den implementierten Detektionsalgorithmus erweitert. Diese Erweiterungen eröffnen neue Einsatzbereiche:

Der Sniffer kann dazu verwendet werden, eine Bluetooth-Verbindung abzuhören, die Adaptive Frequency Hopping benutzt. Denkbar ist beispielsweise ein Aufbau, in dem zwei Sniffer zum Einsatz kommen. Der erste detektiert dabei die Kanalmaske, der zweite nutzt diese Information, um die Pakete der beobachteten Verbindung abzufangen und zu analysieren.

Schon anhand der erkannten Kanalmaske kann die Qualität einer Bluetooth-Verbindung beurteilt werden und auf vorhandene Funkstörer geschlossen werden. Wenn fast alle Kanäle vom Piconet genutzt werden, spricht das dafür, dass keine Funkstörung vorliegt. Liegt in der Kanalmaske ein Block von etwa 20 Kanälen, die nicht genutzt werden, weist das auf eine Störung durch ein einzelnes WLAN-Netz hin. Wenn nur wenige Kanäle genutzt werden und sich die Kanalmaske häufig ändert, kann der Grund sein, dass das gesamte 2,4 GHz Band gestört ist und die Bluetooth-Verbindung an einem hohen Paketverlust leidet.

Da die Bildung der Kanalmaske nicht im Bluetooth-Standard abschließend definiert ist, sondern von der verwendeten Bluetooth-Implementierung abhängt, kann der implementierte Detektionsalgorithmus dafür verwendet werden, den eingesetzten Bluetooth-Stack zu bestimmen. Die Änderungsrate und die Reaktionsgeschwindigkeit der Kanalmaske kann für eine Bewertung der verwendeten Bluetooth-Implementierung verwendet werden.

Die Ergebnisse dieser Arbeit erweitern den quelloffenen Bluetooth-Sniffer Ubertooth um eine wertvolle Funktion, die viele neue Anwendungsmöglichkeiten eröffnet und eine Grundlage für interessante weiterführende Projekte bietet.

Literatur

[1] ZigBee Alliance. „ZigBee Specification“. In: (2004).

[2] Andrew S. Tanenbaum, *Modern Operating Systems*. Prentice-Hall of India. ISBN: 9788120309746.

[3] Lawrence D Brown, T Tony Cai und Anirban DasGupta. „Interval estimation for a binomial proportion“. In: *Statistical science* (2001), S. 101–117. URL: http://www.jstor.org/stable/2676784;%20http://projecteuclid.org/download/pdf_1/euclid.ss/1009213286.

[4] Beman Dawes, David Abrahams und Rene Rivera. *Boost C++ Libraries*. URL: http://boost.org/.

[5] Deji Chen, Mark Nixon, Aloysius Mok, *WirelessHART: Real-Time Mesh Network for Industrial Automation*. Springer. ISBN: 9781441960467.

[6] Piotr Dymacz. *pepe2k/u-boot_mod*. URL: https://github.com/pepe2k/u-boot_mod.

[7] Piotr Dymacz. *Zmodyfikowany U-Boot dla routerów TP-Link z Atheros AR9331 z trybem aktualizacji oprogramowania przez www i konsolą sieciową NetConsole*. URL: http://www.tech-blog.pl/2013/03/29/zmodyfikowany-u-boot-dla-routerow-tp-link-z-atheros-ar9331-z-trybem-aktualizacji-oprogramowania-przez-www-i-konsola-sieciowa-netconsole.

[8] Kristoffer Fleming. *PAN Bluetooth Network Encapulation Protocol Specification*. Feb. 2003.

[9] Matias Fontanini. *C++ packet sniffing and crafting library*. URL: http://libtins.github.io.

[10] Inc. Frontline Test Equipment. *ComProbe Sodera Wide Band Bluetooth Protocol Analyzer*. URL: http://www.fte.com/products/Sodera.aspx.

[11] Inc. Frontline Test Equipment. *FTS4BT Bluetooth Protocol Analyzer & Packet Sniffer*. URL: http://www.fte.com/products/FTS4BT.aspx.

[12] Bluetooth Audio/Video Working Group. *A2DP - Advanced Audio Distribution Profile*. Juli 2012.

[13] IEEE. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Standard 802.11. Juni 1999.

[14] IEEE. *IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Specific Requirements - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)*. 2002.

[15] IEEE. „IEEE Std 802.15.4-2011, IEEE Standard for Local and metropolitan area networks—Part 15.4: Low-Rate Wireless Personal Area Networks (WPANs)“. In: (Sep. 2011).

[16] Ilja N Bronstein, *Taschenbuch der Mathematik*. ISBN: 9783817120024.

[17] Adafruit Industries. *Bluefruit EZ-Key - 12 Input Bluetooth HID Keyboard Controller - v1.2*. URL: `https://www.adafruit.com/products/1535`.

[18] ISA. „ANSI/ISA-100.11a-2011 Wireless systems for industrial automation: Process control and related applications“. In: (2011).

[19] Phoenix Contact GmbH & Co. KG. *Handbuch zum Bluetooth Ethernet Port Adapter FL BT EPA*. 2013.

[20] Jean-Philippe Lang. *Overview - GNU Radio - gnuradio.org*. URL: `http://gnuradio.org/redmine/projects/gnuradio`.

[21] libusb. *libusb*. URL: `http://libusb.info/`.

[22] S Maryam. *FT232H SINGLE CHANNEL HI-SPEED USB TO MULTIPURPOSE UART/FIFO IC*. Techn. Ber. Future Technology Devices International Limited, 2015.

[23] Michael Ossmann. *mossmann's blog: Project Ubertooth: Building a Better Bluetooth Adapter*. URL: `http://ossmann.blogspot.de/2011/02/project-ubertooth-building-better.html`.

[24] BlueZ Project. *BlueZ*. URL: `http://bluez.org/`.

[25] Ettus Research. *USRP B200 Software Defined Radio (SDR)*. URL: `http://www.ettus.com/product/details/UB200-KIT`.

[26] NXP Semiconductors. *LPC1759/58/56/54/52/51 32-bit ARM Cortex-M3 MCU; up to 512 kB flash and 64 kB SRAM with Ethernet, USB 2.0 Host/Device/OTG, CAN*. Techn. Ber. 2010.

[27] Bluetooth SIG. *Bluetooth Core Specification v1.2*. Techn. Ber. 2003.

[28] Bluetooth SIG. *Bluetooth Core Specification v2.0 + EDR*. Techn. Ber. Nov. 2004.

[29] Bluetooth SIG. *Bluetooth Core Specification v2.1 + EDR*. Techn. Ber. Juli 2007.

[30] Bluetooth SIG. *Bluetooth Core Specification v3.0 + HS*. Techn. Ber. Apr. 2009.

[31] Bluetooth SIG. *Bluetooth Core Specification v4.0*. Techn. Ber. Juni 2010.

[32] Skylabs. *SKY12347-362LF 0.1-3.0 GHz Six-Bit Digital Attenuator with Serial or Parallel Driver (0.5 dB LSB) Data Sheet, document #201371*. Techn. Ber. 2011.

[33] Dominic Spill. *Project Ubertooth: Motivating the Problem*. URL: http://ubertooth.blogspot.de/2013/02/motivating-problem.html.

[34] Dominic Spill und Andrea Bittau. „BlueSniff: Eve Meets Alice and Bluetooth." In: *WOOT*. Hrsg. von Dan Boneh, Tal Garfinkel und Dug Song. USENIX Association, 2007. URL: http://dblp.uni-trier.de/db/conf/woot/woot2007.html#SpillB07.

[35] Dominic Spill und Michael Ossmann. *gr-bluetooth - Bluetooth for GNU Radio*. URL: http://gr-bluetooth.sourceforge.net/.

[36] Dominic Spill und Michael Ossmann. *greatscottgadgets/libbtbb*. URL: https://github.com/greatscottgadgets/libbtbb.

[37] Dominic Spill und Michael Ossmann. *greatscottgadgets/ubertooth*. URL: https://github.com/greatscottgadgets/ubertooth.

[38] Dominic Spill und Michael Ossmann. *Project Ubertooth*. URL: http://ubertooth.sourceforge.net.

[39] The OpenWrt developer team. *OpenWrt*. URL: https://openwrt.org/.

[40] The matplotlib development team. *matplotlib: python plotting — Matplotlib 1.4.3 documentation*. URL: http://matplotlib.org/.

[41] Texas Instruments, Incorporate [SWRS042,A2]. *2.4 GHz Low-Power RF Transceiver (Rev. A*. Techn. Ber. 2008.

[42] Texas Instruments, Incorporated [SWRS070,B]. *CC2591 2.4 GHz RF Front End (Rev. B)*. Techn. Ber. 2015.

[43] Henry William Watson und Francis Galton. „On the probability of the extinction of families". In: *The Journal of the Anthropological Institute of Great Britain and Ireland* 4 (1875), S. 138–144. URL: http://www.jstor.org/stable/2841222.

[44] ZVEI. *Wireless Solutions in Automation*. URL: http://www.zvei.org/Publikationen/Wireless%20Solutions%20in%20Automation.pdf.