Technische
Hochschule
Rosenheim
Technical University of Applied Sciences

Fakultät für Informatik

Studiengang Informatik

Entwurf und Implementierung einer automatischen Selbstkalibrierung für einen Navigation Constellation Simulator

Bachelor Thesis

von

Christian Pritzl

Datum der Abgabe: 12.08.2019
Erstprüfer: Prof. Florian Künzner
Zweitprüfer: Prof. Dr. Gerd Beneken

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, den 12.08.2019

Christian Pritzl

Kurzfassung

Die Simulation von Global Navigation Satellite Systems (GNSS)-Signalen findet vielfältige Anwendung bei Entwicklung und Tests von entsprechenden Empfängern, womit wesentlicher Einfluss auf den Alltag vieler Menschen ausgeübt wird.
Das Ziel der vorliegenden Arbeit ist der Entwurf und die Implementierung einer automatischen Selbstkalibrierung für einen modularen GNSS-Signalgenerator, der Teil eines Navigation Constellation Simulators ist.
Um den Genauigkeitsverlust des Signalgenerators über die Zeit, bedingt durch Umwelteinflüsse und Bauteilalterung, auszugleichen, ist es notwendig, nach der initialen Werkskalibrierung regelmäßig nachzukalibrieren. Der Fokus lag dabei auf der Kalibrierung der Signalphasen der Signalpfade eines Signalmoduls, um eine Abweichung unter 0.5 Grad voneinander zu erreichen.
Das aktuell angewendete, manuelle Konzept kalibriert einen Simulator als Ganzes unter Zuhilfenahme externer Messgeräte. Im Rahmen dieser Bachelorarbeit wurde ein neues Konzept ausgearbeitet, welches die Grundlage schafft, die Einzelmodule individuell und automatisch zu kalibrieren. Dazu wurde erstmalig auf einen neuen, internen Detektor zurückgegriffen.
Um die Modularität des Navigation Constellation Simulators zu unterstützen, werden die Kalibrierdaten pro Modul abgelegt. Hierzu wurde ein neues Konzept zur Datenspeicherung, welches Kompatibilität zu zukünftigen Softwareversionen ermöglicht und für unkomplizierte Erweiterungen offen ist, entworfen. Die Treiber und Firmware der verwendeten Module wurde angepasst und erweitert sowie neue Schnittstellen geschaffen.
Zur Durchführung der konkreten Kalibrierung wurde ein Algorithmus entwickelt und in Python implementiert. Dazu wurde das eingesetzte Embedded-Linux aktualisiert. Um die Ausfallzeit des Generators gering zu halten, wurde auf eine zeitliche Optimierung des Kalibrierprozesses geachtet. Einen wichtigen Aspekt der Arbeit stellte die Verifikation der Ergebnisse gegen die Resultate der Werkskalibrierung dar.
Im Rahmen der Arbeit wurden Dokumentationen erstellt, um zukünftig den Einstieg in die Arbeit mit dem verwendeten Navigation Constellation Simulator zu erleichtern.

Schlagworte: Satellitennavigation, Embedded-Linux, Mikrocontroller, Autocalibration, Python, C

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

API	Advanced Programming Interface
BSON	Binary JavaScript Object Notation
CD	Continous Delivery
CI	Continous Integration
CLI	Command Line Interface
CNC	Computerized Numerical Control
CSV	Comma Separated Value
DAC	Digital-Analog-Converter
EEPROM	Electrically Erasable Programmable Read-Only Memory
EU	Europäische Union
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GHz	Gigahertz
GLONASS	Global'naya Navigatsionnaya Sputnikovaya Sistema
GNSS	Global Navigation Satellite Systems
GPS	Navigational Satellite Timing and Ranging - Global Positioning System
I2C	Inter Integrated Circuit
IDE	Integrated Development Environment
ID	Identifier
IP	Internet Protocol

ITU	International Telecommunications Union
JSON	JavaScript Object Notation
KHz	Kilohertz
KB	Kilobyte
LTS	Long Term Support
MB	Megabyte
MEO	Medium Earth Orbit
MHz	Megahertz
NCS	Navigation Constellation Simulator
OCXO	Oven Controlled Crystal Oscillator
PCB	Printed circuit board
PCIe	Peripheral Component Interconnect Express
PEP	Python Enhancement Proposal
PLL	Phase-Locked Loop
PPS	Precise Positioning Service
RAM	Random Access Memory
SMD	Surface-mounted device
SMS	Short Messaging Service
SPI	Serial Peripheral Interface
SPS	Standard Positioning Service
SQL	Structured Query Language
SSH	secure shell
SoM	System on Module
TCP	Transmission Control Protocol

TCXO	Temperature Compensated Crystal Oscillator
UDP	User Datagram Protocol
UML	Unified Modeling Language
UTC	Universal Time, Coordinated
VCS	Version Control System
XML	eXtensible Markup Language
dB	Dezibel
u-boot	universal bootloader
dBm	Dezibel Milliwatt

1 Einleitung

Die WORK Microwave GmbH, gegründet 1986 in Holzkirchen, beschäftigt circa 100 Mitarbeiter in den vier Geschäftsbereichen „Satellite Technologies“, „Defence Electronics“, „Navigation Simulators“ und „Sensors and Measurement“. Das ursprüngliche Portfolio an Produkten im Bereich Oszillatoren, Synthesizern sowie Zwischen- und Hochfrequenzkonvertermodulen wurde in den zurückliegenden 30 Jahren kontinuierlich um neue Produkte in den Bereichen der digitalen Hochfrequenzlösungen, Entwicklungen von Geräten im Video- und Netzwerkbereich und Navigation Constellation Simulator (NCS)s erweitert. Heute ist WORK Microwave bekannt für seine Expertise in der Konzeptionierung und Entwicklung produktionsfertiger Prototypen, Hardwaredesign und Softwareentwicklung, in den Bereichen Satelliten und drahtlose Kommunikation, TV & Media, Radar & Militärische Kommunikation, Hochfrequenzelektronik, Hochfrequenzmesstechnik & Digitale Signalverarbeitung, Hochgeschwindigkeitsdatenerfassung und Industriesensoren [WOR].
Außerdem finden am Standort in Holzkirchen Surface-mounted device (SMD)-Bestückung, Gehäusefertigung mittels Computerized Numerical Control (CNC), sowie die Montage und Endtests der Produkte statt. Dadurch können Klein- und Kleinstserien nach höchsten Standards an Qualität, Zuverlässigkeit und Leistung aus verschiedensten Branchen effizient gefertigt werden [Deu].
Aufgrund des starken Wachstums der vergangenen Jahre – alleine 2017 wuchs der Umsatz um 50 % auf 15,3 Millionen Euro – ist WORK Microwave seit 2019 der erste Mieter im neu entstehenden Technologiepark im Herzen des Holzkirchner Gewerbegebietes [Pro].
Ende 2016 wurde in Zusammenarbeit mit der Ifen GmbH aus Poing der NCS TITAN GNSS Simulator, siehe Abbildung 1.1, am Markt veröffentlicht. Dank 256 Kanälen, beziehungsweise 1024 Multipfadkanälen wird der parallele Einsatz mehrerer Simulatoren im Testzyklus von Satellitenempfängern überflüssig und der Kunde erhält durch den modularen Aufbau die Möglichkeit vor Ort Änderungen vorzunehmen [Ife].
Aktuell arbeitet WORK Microwave in Eigenregie an der nächsten Generation von NCS basierend auf dem TITAN.

1 `https://insidegnss.com/ifen-launches-ncs-titan-gnss-simulator`, Datum: 18.07.2019

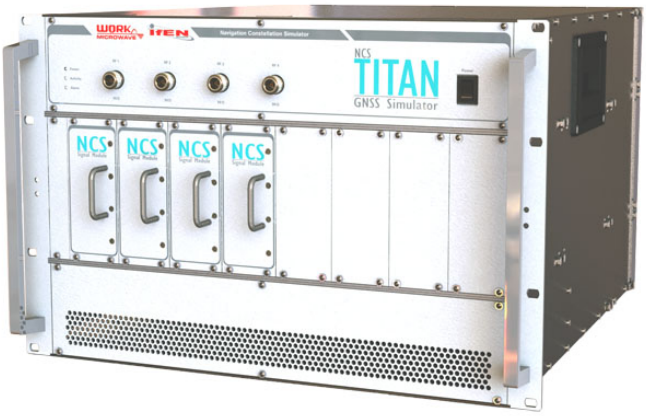

Abbildung 1.1 Produktbild NCS TITAN GNSS [1]

1.1 Motivation

Mit der zunehmenden Verbreitung von Smartphones und anderen mobilen Satellitenempfängern über die letzten Jahre [Ten], ist das Testen dieser Empfänger immer stärker in den Fokus der Unternehmen gerückt. Die Anzahl verwendeter GNSS-Empfänger soll von circa vier Milliarden im Jahr 2017 auf bis zu neun Milliarden im Jahr 2023 ansteigen [DK17, S. 11]. Entsprechend wird der Bedarf an Simulatoren zunehmen [DK17, S. 916]. Diese Schätzung deckt jedoch nur zivile Bereiche wie die Navigation von Fahrzeugen zu Wasser, Land und Luft, in der Landwirtschaft, ortsbasierte Spiele, der Nachverfolgung bei sportlicher Aktivität oder in sozialen Netzwerken sowie die Überwachung von Tieren ab [DK17, S. 924 ff]. Eine exakte Abschätzung der Marktkapitalisierung lässt sich nur schwer treffen, da der Preisbereich der GNSS-Empfänger von wenigen Dollar bis zu mehreren Millionen Dollar für atomgetriebene Navigation-Sets in U-Booten reicht. Anzunehmen ist eine Kategorie im Bereich von etwa 100 Milliarden Dollar für das Jahre 2021 [DK17, S. 917].
Einer der Schlüsselfaktoren beim Testen von Hardware ist die Kontrolle über die Testbedingungen und -umgebungen und die Möglichkeit diese Testumstände zu reproduzieren. An dieser Stelle kommen NCSs zum Tragen, welche das Erzeugen einer eigenen geschlossenen Welt mit Satellitenbewegung, Empfängerbewegung, Signalcharakteristiken, Atmosphäre, Umwelteinflüssen, et cetera ermöglichen [Spi, S. 3]. Des Weiteren bieten sie die Möglichkeit bislang nicht verfügbare GNSS-Systeme oder Funktionen vorab zu testen. Beim typischen Testablauf eines Produkttestes wird zunächst mithilfe eines Simulators im Labor gearbeitet. Später werden Aufzeichnungen eingespielt und erst am Ende des Testzyklus werden Feldtests durchgeführt [Spi, S. 24].

Unter anderem bedingt durch Umwelteinflüsse und Bauteilalterung verlieren Messgeräte an Präzision, welche durch regelmäßige Wartung und Kalibrierung wieder hergestellt werden muss. Aktuell werden die Module vom Kunden eingeschickt, anschließend von Hand vermessen und neu kalibriert. Im Hinblick auf zukünftige Einsatzmöglichkeiten, die sich aus dem modularen Aufbau der NCS ergeben, ist dies ein teurer und langsamer Prozess, der nicht zukunftsfähig ist. Aus diesen Gründen beschäftigt sich die vorliegende Arbeit mit dem Entwurf und der Implementierung einer automatischen Selbstkalibrierung.

1.2 Aufgabenstellung und Zielsetzung

Unter Verwendung der vorhandenen Ansteuerungen der Systemkomponenten des NCSs über Transmission Control Protocol (TCP)/Internet Protocol (IP) und Serial Peripheral Interface (SPI), beziehungsweise Neuimplementierungen wo notwendig, soll die vorhandene konzeptuelle Idee eines Algorithmus auf Basis von Curve-Fitting zur Kalibrierung umgesetzt werden. Entscheidend wird sein, die Annahmen und Abschätzungen bezüglich Leistungsfähigkeit und Genauigkeit der Hardware zu verifizieren sowie das verstreute Wissen über die einzelnen Komponenten der Hard- und Software zu bündeln und zu dokumentieren.
Hierzu werden im ersten Schritt die Schnittstellen der einzelnen Komponenten erweitert und ein neues Format zur Speicherung der Kalibrierdaten mit einem Fokus auf Erweiterbarkeit und Lesbarkeit entworfen. Anschließend soll unter Berücksichtigung zeitlicher Optimierung des Prozesses und der Kommunikation zwischen den Komponenten die Kalibrierung entworfen und implementiert werden. Abschließend werden die Ergebnisse der neuen Kalibrierungsfunktionalität mit den Werten der bestehenden Werkskalibrierung verglichen.
Als optional wird die Implementierung eines Webinterfaces zur einfachen Steuerung und Überwachung der Kalibrierung sowie die Anzeige und Analyse der ermittelten Werte eingestuft.
Ein entscheidender Faktor ist die Dokumentation der Prozesse und Ergebnisse, um den Einstieg in die Arbeit mit dem NCS zu erleichtern und die Weiterentwicklung der Software zu ermöglichen.

1.3 Projektmanagement

Zu Beginn des Projektes wurde, abgeleitet von Kapitel 1.2, eine Grobspezifikation angefertigt, welche anschließend in einen Zeitplan, siehe Tabelle 1.1, überführt wurde. Hierzu

wurden die drei Meilensteine „Entwurfs- und Vorbereitungsphase“, „Implementierungs- und Integrationsphase“ sowie „Dokumentationsphase“ definiert und die verfügbare Zeit von fünf Monaten, entsprechend dem geschätzten Aufwand, aufgeteilt. Dabei bauten die nachfolgenden Komponenten, entsprechend dem Wasserfallmodell, auf den Vorhergehenden auf, beziehungsweise wurden über Rückkopplungen, basierend auf neu gewonnenen Erkenntnissen, einer Überarbeitung unterzogen. Innerhalb der Meilensteine wurde darauf geachtet, dass in kurzen Zeitabständen einsatzbereite Komponenten für die weitere Entwicklung verfügbar waren. Beispielsweise wurden die gemessenen Werte sofort in der neuen Datenstruktur abgelegt, um das Risiko einer falschen oder fehlerhaften Architekturentscheidung gering zu halten und Fehler bei der Implementierung möglichst früh zu entdecken.
Ebenfalls hilfreich hierzu sind Tests, die ein essentieller Bestandteil moderner Softwareentwicklung sind und so früh wie möglich eingeplant werden sollen [Spi14, S. 14]. Aus diesem Grund wurden für die ersten beiden Phasen dedizierte Zeitblöcke zum Testen der Zwischenstände eingeplant und die erwarteten Sollzustände vorab in Grundzügen definiert. Getestet wurden die Funktionalitäten der Software anschließend direkt auf der Zielplattform, sodass auf eine dedizierte Testumgebung verzichtet werden konnte.
Eine große Herausforderung bei der Umsetzung genau spezifizierter Zeitpläne für ein sequentielles Vorgehen sind unvorhergesehene Ereignisse, beispielsweise in Form geänderter Anforderungen, wie sie in Kapitel 4.6 beschrieben werden [All18, S. 42]. Hier erwies sich die durch tägliche Abstimmungen gewonnene Flexibilität innerhalb einer Projektphase als vorteilhaft. So konnte schnell auf geänderte Anforderungen reagiert und das sequentielle Vorgehensmodell um einen iterativen Aspekt erweitert werden.
Um die stetige Weiterentwicklung der Softwarekomponenten nachvollziehbar zu dokumentieren sowie überprüfbar und unabhängig von paralleler Entwicklung zu gestalten, wurde sie mittels des Version Control System (VCS) „Git“ versioniert [Pop, S. 11ff]. Das Konfigurationsmanagement der Abteilung Navigation Simulators setzt hierzu auf eigenständige Repositories für die Komponenten, welche gegebenenfalls über Submodule weiter aufgeteilt sind. Zum Bauen der Komponenten steht ein Debian-basierter Server zur Verfügung. So kann sichergestellt werden, dass keine Fehler beim Kompilieren durch lokale Konfigurationsunterschiede der Entwicklercomputer verursacht werden.
Entsprechend dem Aufbau des Gesamtsystems als Sammlung von Modulen, wurde auch das Modul zur Phasenkalibrierung als separates Modul entlang seiner Verantwortlichkeit entwickelt. Dies ermöglicht eine lose Bindung an das Gesamtsystem und somit einfacheres Testen und Weiterentwickeln. Es ist möglich, das Autocalibrationmodul unabhängig vom restlichen System auszuführen [Ana18, S. 287].
Um Konsistenz mit dem vorhandenen Programmcode zu gewährleisten, wurde von Be-

Tabelle 1.1 Zeitplanung

Datum	Meilenstein	Veranschlagte Tätigkeit
1. Woche	Entwurfs- und Vorbereitungsphase	SPI Treiber und CLI zur Auslesung und Ansteuerung des Detektors
2. Woche		
3. Woche		Erstellung der Skripte zur Ansteuerung des Signal-Generators
4. Woche		
5. Woche		Testen der Hardware, gegebenfalls Debugging
6. Woche		Test-Reihen zur Bestimmung des optimalen Arbeitsbereiches des Lokaloszillators
7. Woche		
8. Woche		Entwurf der Datenstruktur zur Speicherung der Messwerte und der abgeleiteten Kalibrierwerte
9. Woche		
10. Woche	Implementierungs- und Integrationsphase	Implementierung des Autocalibrationmoduls
11. Woche		
12. Woche		Debugging und Optimierung des Autocalibrationmoduls
13. Woche		Vergleichsmessung mit bestehender Werkskalibrierung
14. Woche		Integration der neuen Datenstruktur in die bestehende Applikation
15. Woche		Verifikationsmessung gegenüber Werkskalibrierung
16. Woche		Puffer / Integration in Gesamtsystem / Webinterface zum Vewalten
17. Woche	Dokumentationsphase	Dokumentation
18. Woche		
19. Woche		
20. Woche		
21. Woche		
22. Woche		

ginn an darauf geachtet, Konformität mit dem Python Enhancement Proposal (PEP)-8 Standard[2] herzustellen. Dies unterstützt andere Entwickler dabei, den Code und seine Funktionalität zu verstehen, beziehungsweise ihn für die Weiterentwicklung zu erweitern oder zu bearbeiten [All18, S. 98ff]. Für die weitere Dokumentation wurde auf Basis der Unified Modeling Language (UML) ein Klassendiagramm zur Modellierung der Systemarchitektur erstellt. Ein solches Diagramm ermöglicht die einfache Darstellung der einzelnen Systemkomponenten und ihrer internen Beziehungen [All18, S. 21ff].
Zur Überwachung und Analyse der Kommunikation zwischen den einzelnen Komponenten wurde auf Wireshark zurückgegriffen, welcher um ein Dissector-Plugin für das verwendete navx-gen2 Protokoll erweitert wurde. Für den späteren Vergleich der Kalibrierergebnisse mit der Werkskalibrierung wurden die Messergebnisse in einer log-Datei protokolliert und abgelegt.

2 `https://www.python.org/dev/peps/pep-0008/`

2 Theoretische Grundlagen

2.1 Definition Kalibrierung

Entsprechend der Definitionen von „Kalibrierung“ [Sch97, S. 11] und „Justierung“ [Sch97, S. 10] wird zuerst eine Kalibrierung, das heißt eine Feststellung der Maßabweichung vom fixen Wert der Phasendrehung vorgenommen und anschließend beim Einstellen der Phase eine Justierung. Im Rahmen der vorliegenden Arbeit werden diese Prozesse unter dem Begriff „Kalibrierung“ zusammengefasst. Dieser zusammengefasste Prozess stellt somit keine Kalibrierung im eigentlich Sinn der ursprünglichen Definition dar, sondern entspricht der umgangssprachlichen Verwendung im Entwicklungsprozess.
Die Gründe für regelmäßiges Kalibrieren von Systemen sind vielfältig. Umwelteinflüsse, wie Temperatur, Feuchtigkeit oder Erschütterungen, sowie die, dadurch beschleunigte, Bauteilalterung sind hierbei zu nennen. Da minimale Schwankungen, wie zum Beispiel Phasenverschiebungen um wenige Grad, im Bereich der Hochfrequenz große Änderungen im Ausgabesignal hervorrufen können, ist regelmäßiges Kalibrieren für ein verlässliches System unabdingbar.

2.2 Curve-Fitting

Bei der Bestimmung der Phasenkorrekturwerte aus den Ergebnissen der jeweiligen Messreihe handelt es sich um Minimumsuchen. Vermutet wurde, dass diese einer Verteilung unterliegen würden, die das simple Ablesen eines Minimums unmöglich macht. In diesem Fall muss eine sogenannte Ausgleichskurve möglichst passend an die vorhandenen Messpunkte angenähert werden [Pap17, S. 312]. Dieser Vorgang wird als „Curve-Fitting“ bezeichnet. Als Ergebnis erhält man eine Funktion, deren Minimum anschließend bestimmt werden kann. Das Prinzip ist in Abbildung 2.1 dargestellt.
Die Programmiersprache Python bietet hierzu in der Bibliothek „scipy“ eine Funktion an, die auf Basis von drei verschiedenen Algorithmen mithilfe des Gauß´schen Prinzips der kleinsten Quadrate, vergleiche

$$S(a;b;c;...) = \sum_{i=1}^{n} v_i^2 = \sum_{i=1}^{n} [y_1 - f(x_i)]^2 \tag{2.1}$$

[Pap17, S. 312] mit den Kurvenparametern *a, b, c, ...*, die Kurve anpassen kann [The].

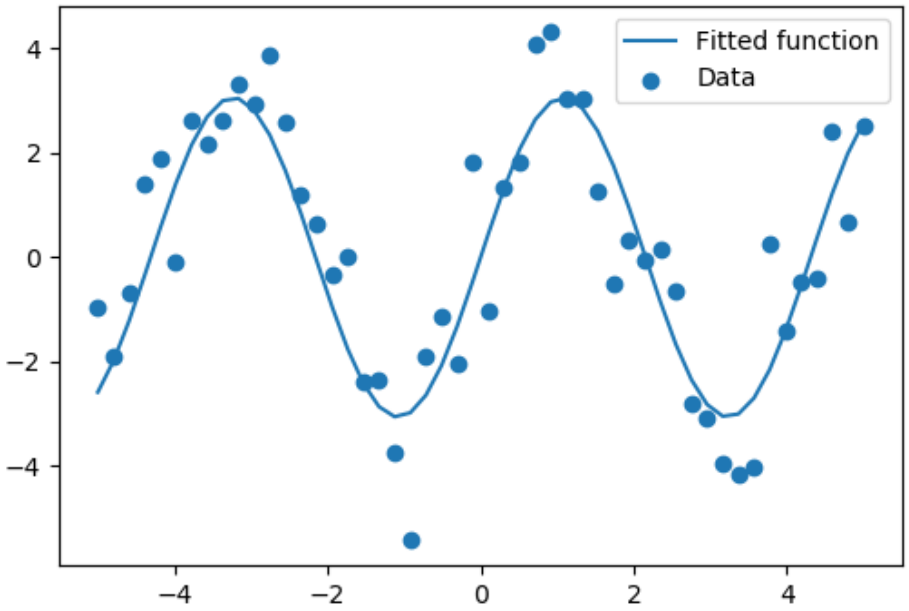

Abbildung 2.1 Ausgleichskurve in einer verteilten Menge von Datenpunkten [1]

2.3 Grundbegriffe der Kommunikationstechnik

2.3.1 Oszillator

Ein Oszillator ist eine elektrische Schaltung, die zur Erzeugung von Schwingungen, welche sich durch eine hohe Genauigkeit und Stabilität auszeichnen, beispielsweise einen Quarzkristall enthält. Um vor allem die starke Temperaturabhängigkeit der Quarzoszillatoren in den Griff zu bekommen, verwendet man Oven Controlled Crystal Oscillator (OCXO) oder Temperature Compensated Crystal Oscillator (TCXO). TCXO zeichnen sich durch Temperaturstabilität bei geringen Kosten aus, wodurch sie in Alltagsgegenständen, wie zum Beispiel Smartphones, verbaut werden.

1 `http://scipy-lectures.org/intro/scipy/auto_examples/plot_curve_fit.html`, Datum: 19.07.2019

OCXO sind deutlich stabiler und genauer. Dies wird erreicht, in dem der Quarz dauerhaft auf seine Betriebstemperatur erwärmt wird, wodurch diese als Konstante behandelt werden kann [Bau]. Sie werden vor allem in hochwertigen Geräten, wie einem NCS, eingesetzt.

2.3.2 Signal und Signalparameter

Jedes elektrische Signal weist verschiedene charakteristische Parameter, sogenannte Signalparameter, beispielsweise Amplitude, Frequenz oder Phase, gezeigt in Abbildung 2.2, auf [Sch13, S. 2]. Über Schwingungen erzeugt ein Oszillator ein Signal mit einer bestimmten Frequenz, welche als Trägerfrequenz bezeichnet wird. Um Informationen zu übertragen, wird auf diese Trägerfrequenz die eigentliche Informationsfrequenz moduliert. So ist es durch Verschieben der Informationen in unterschiedliche Frequenzbereiche möglich, ein Übertragungsmedium, wie zum Beispiel Luft, mehrfach zu verwenden [EK]. Der Frequenzbereich der Hochfrequenz reicht dabei von 10 Kilohertz (KHz) (High Frequency) [hft] bis zu 300 Gigahertz (GHz) (Ultra High Frequency) [itw].
Diese hohen Frequenzen werden in der Regel über sogenannte „Zwischenfrequenzen“ schrittweise erreicht. Hierzu erzeugt ein Lokaloszillator „vor Ort“ eine Hilfsfrequenz, die zur Umsetzung dient [Intb].
Als „Amplitude“ wird der maximale Wert einer sinusförmigen Größe bezeichnet. Diese verliert mit der Zeit, abhängig vom durchwanderten Medium, an Stärke. Dieser Effekt wird als „Dämpfung“ bezeichnet.
Erreichen zwei in der Frequenz gleiche Signale ihre Maxima, beziehungsweise Minima, zu unterschiedlichen Zeitpunkten, spricht man von einer Phasenverschiebung. Diese wird mit dem Winkel φ angegeben. Veranschaulicht ist dies in Abbildung 2.2 am Punkt „Phasenwinkel“.

2.3.3 Rauschen

In allen Bereichen, in denen Signale erfasst, verstärkt, gemessen oder übertragen werden, ist das sogenannte „Rauschen“ ein mehr oder weniger stark ausgeprägtes Problem, welches grundsätzlich immer vorkommt. Mathematisch gesehen ist Rauschen ein völlig stochastischer Vorgang, da seine Werte zu einem gegebenem Zeitpunkt nicht vorhersagbar sind [Inta]. Die Ursachen des Rauschens sind vielfältig, beispielsweise brownsche Bewegungen der Ladungsträger oder das Überqueren von Potentialbarrieren durch Ladungsträger [HAM, S. 3]. Die Rauschleistung ist über den Frequenzbereich verteilt, sodass

2 basierend auf `https://commons.wikimedia.org/wiki/File:Rot.-Zeiger2.svg`, “Creative Commons” von Saure Lizenz: CC BY-SA 3.0, Datum: 19.07.2019

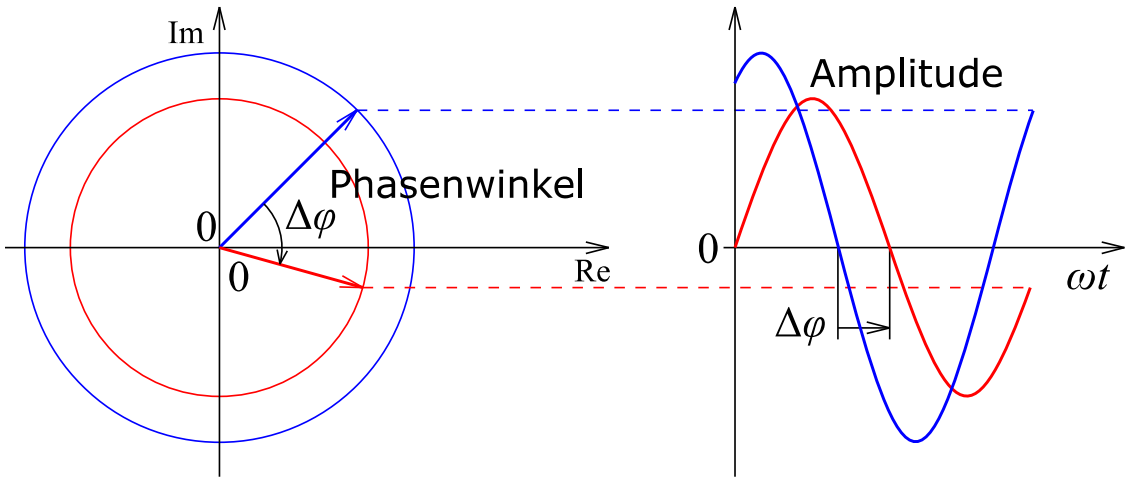

Abbildung 2.2 Darstellung der Signalparameter Amplitude, Frequenz und Phasenwinkel [2]

durch Filter die Anteile abseits der Signalfrequenz bis zu einem gewissen Grad entfernt werden können [sig].

2.3.4 Mischer

Im Bereich der Datenübertragung werden bei der sogenannten „Frequenzumsetzung“ Mischer für die Umsetzung von Eingangsfrequenzen in Ausgangsfrequenzen eingesetzt. Unterschieden wird dabei zwischen additiven und multiplikativen Mischern [uSCuGE02, S. 1409ff]. Zum Beispiel werden im Bereich der Hochfrequenz mithilfe eines Lokalsozillators Zwischenfrequenzen in Hochfrequenzsignale umgesetzt [Huf, S. 18]. In der Schaltung des Self-Calibration Detectors, siehe Abbildung 3.5, kommt ein additiver Mischer zum Einsatz. Bei diesem werden die Eingangssignale, entsprechend Formel 2.2, addiert.

$$x_{HF}(t) = sin(\omega_{ZF} * t) + sin(\omega_{LO} * t) \tag{2.2}$$

Der schaltungstechnische Aufwand bei additiven Mischern ist geringer, erfordert allerdings ein Filtern unerwünschter Frequenzanteile am Ausgang [uSCuGE02, S. 1411f].

3 Navigation Constellation Simulator

Der Begriff „NCS“ unterliegt keiner festen Definition, sondern ist die bei WORK Microwave gebräuchliche interne Bezeichnung für Navigationssimulatoren. In anderen Unternehmen mag es abweichende Formulierungen geben.
NCSs dienen der Erzeugung von GNSS-Signalen, womit Satellitenempfänger vor erlangen der Marktreife umfangreich getestet werden können. Für aussagekräftige Tests sind kontrollierte Bedingungen essentiell, um wiederholt und nachvollziehbar das Verhalten in Alltagssituationen, wie dem Verbindungsaufbau zu den Satelliten oder in Extremsituationen bei schlechtem oder gestörtem Empfang, zu testen. Die Fortschritte in der Entwicklung eines Produktes sollen erfasst oder das Produkt an die Weiterentwicklung der GNSS-Systeme angepasst werden. Zudem sollen bereits im Einsatz befindliche Empfänger zukünftig mit bislang nicht voll einsatzfähigen Systemen, wie Galileo oder Beidou, interagieren können. In diesen Fällen wird auf die kontrollierte Welt eines NCSs zurückgegriffen.

3.1 Satellitennavigation

Unter dem Begriff „Satellitennavigation“ wird die satellitengestützte Positionsbestimmung zur Navigation verstanden. Satelliten stellen Positions- und Zeitinformationen zur Verfügung, aus denen entsprechende Empfänger ihre Position auf, beziehungsweise über der Erdoberfläche berechnen können.
Entsprechend der Definition fasst GNSS alle global zur Verfügung stehenden Systeme, das US-amerikanische Navigational Satellite Timing and Ranging - Global Positioning System (GPS), das russische Global’naya Navigatsionnaya Sputnikovaya Sistema (GLONASS), das chinesische BaiDou und das europäische Galileo, zusammen [wha] [Ste].
Grundsätzlich stellen alle GNSS akkurate, kontinuierliche, weltweit verfügbare, dreidimensionale Positions- und Zeitinformationen sowie Geschwindigkeitsinformationen für alle Benutzer, welche über entsprechende Empfangsgeräte verfügen, bereit [DK17, S. 2].
Da die Kommunikation zwischen Empfänger und Satellit simplex ist, können beliebig viele Empfänger gleichzeitig mit den nötigen Informationen zur Positionsbestimmung versorgt werden [DK17, S. 2, 3].

Der grundsätzliche Aufbau der GNSS umfasst in der sogenannten „Core Constellation“ 24 Medium Earth Orbit (MEO)-Satelliten, die in drei oder sechs Bahnen um den Globus kreisen. Zusätzlich gibt es verteilte Bodenstationen zur Zustandsüberwachung der Satelliten. Um aus den empfangenen Signalen die Position zu bestimmen, wird die Laufzeit der Funksignale vom Empfänger zum Satelliten gemessen. Diese Laufzeit multipliziert mit der Lichtgeschwindigkeit gibt die zurückgelegte Strecke an. Zur Bestimmung der Lage in drei Dimensionen benötigt man die Signale von mindestens drei Satelliten und die Uhrzeit eines Vierten zur Synchronisation. Abbildung 3.1 gibt einen Überblick der Frequenzen, auf denen GPS, Galileo und GLONASS operieren.

3.1.1 GPS

Die Ära der GNSS wurde 1970 mit dem Start des US-amerikanischen Systems GPS eingeleitet. Im Rahmen von GPS existieren zwei Services, die sich an differente Zielgruppen richten. Der Standard Positioning Service (SPS) ist hierbei, entsprechend dem National Coordination Office for Space-Based Positioning, Navigation and Timing mit einer Genauigkeit von etwa fünf Metern in offenem Gelände [Nat], kommerziell verfügbar. An militärische Nutzung richtet sich der verschlüsselte Precise Positioning Service (PPS). Die GPS-Konstellation besteht normalerweise aus 24 Satelliten in sechs MEO Bahnen [DK17, S. 3].

3.1.2 GLONASS

Das 1982 gestartete russische System GLONASS verfügt, analog zu GPS, über einen zivilen und einen militärischen Service im Bereich des L-Bandes von 1100 Megahertz (MHz) bis 1500 MHz. Aktuell besteht es aus 24 aktiven Satelliten und bietet eine mit GPS vergleichbare Genauigkeit [Rus] [DK17, S. 4].

3.1.3 Galileo

Das im Auftrag der Europäische Union (EU) entwickelte europäische Pendant zu GPS und GLONASS heißt Galileo und soll seine volle Einsatzbereitschaft 2020 erreichen. Aufgrund des inkrementellen Ansatzes bei der Entwicklung ist die Verwendung in Teilen bereits seit 2016 möglich. Geplant sind insgesamt vier Services für verschiedene Zielgruppen. Einen frei zugänglichen kostenlosen Service (Open Service) mit einer Genauigkeit von bis zu einem Meter, einen kostenpflichtigen kommerziellen Service (High Accuracy Service) mit höherer Genauigkeit unter einem Meter, einen Speziellen für Regierungen mit erhöhter Sicherheit und Robustheit (Public Regulated Service) und einen separaten Service für

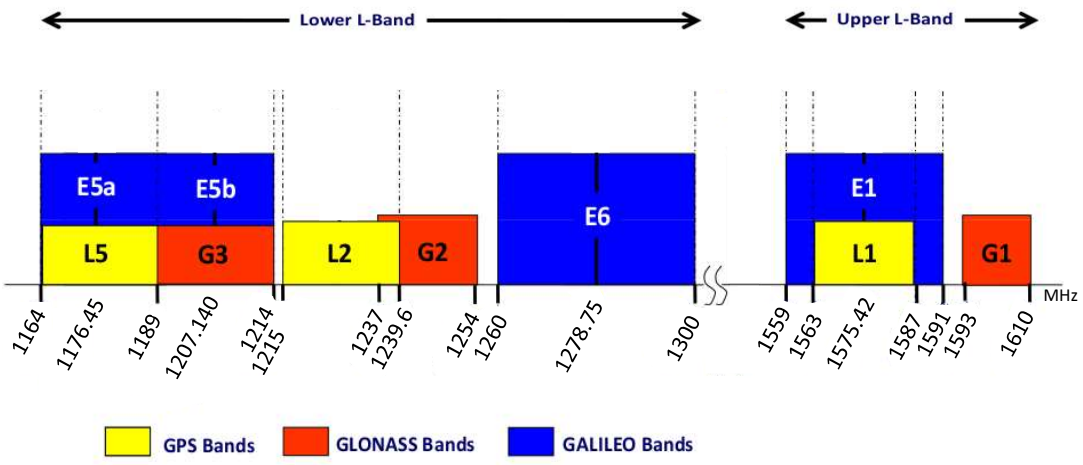

Abbildung 3.1 Überblick des Frequenzbandes der GNSSs GPS, Galileo und GLONASS [1]

Forschung und Rettung (Galileo Search and Rescue Service). Die 30 Satelliten sollen vollständig kompatibel zu GPS sein [DK17, S. 5, 6] [Plu].

3.1.4 BeiDou

Die Volksrepublik China arbeitet seit einigen Jahren an der Entwicklung eines eigenen GNSS unter dem Namen „BeiDou“, welches zeitgleich zu Galileo 2020 fertig sein soll. Der Fokus liegt, obwohl es sich um ein GNSS handelt, auf China und seinen angrenzenden Regionen, für welche BeiDou eine Genauigkeit von einem Meter bieten soll. Zusätzlich soll es für die chinesische Region eine Möglichkeit zum Versenden von Short Messaging Service (SMS), schnelle Positionserfassung sowie einen präzisen Zeitservice bieten. Analog zu GPS und GLONASS wird es einen zivilen und einen militärischen Service geben [DK17, S. 7, 8]. Entsprechend den Angaben, die China gegenüber der International Telecommunications Union (ITU) gemacht hat, wird BeiDou Teile des Frequenzbereiches von GPS und Galileo überlagern [Rod].

3.2 Aufbau Navigation Constellation Simulator

Die NCSs der zweiten Generation, schematisch dargestellt in Abbildung 3.2, dienen als Basis für aktuelle Weiterentwicklungen. Sie bestehen aus bis zu acht autarken Signalmodulen, welche jeweils von einem 32-bit PowerPC powerpc-e500v2 Mikroprozessor

1 basierend auf `https://gssc.esa.int/navipedia/images/f/f6/GNSS_navigational_frequency_bands.png`, Datum: 05.07.2019

angetrieben werden, auf welchem als Betriebssystem ein speziell angepasstes Linux mit dem Long Term Support (LTS)-Kernel 4.14.26 läuft. Jedes dieser Module verfügt neben dem Dualcore Mikroprozessor über 512 Megabyte (MB) Random Access Memory (RAM), eigenem Flashspeicher sowie Schnittstellen zur Ansteuerung von Peripherie. Ein auf diesem Konzept aufbauendes Modul wird als System on Module (SoM) bezeichnet [Tex]. Über einen Switch erfolgt mittels Ethernet die Anbindung an die Außenwelt sowie an den Arbiter. Auf diesem läuft ein Linux identisch zu den Signalmodulen, welches unter anderem die Kalibrierung ausführt sowie die Anbindung an die Combiner ermöglicht. Jeder der vier Combiner kann die Ausgabe eines Signalmoduls mit Signalen anderer Signalmodule sowie zusätzlich die Ausgabe eines Noise-Generators und eines optionalen externen Interferez-Signals mischen. Abschließend wird das Signal über einen Ausgang zur weiteren Verwendung bereitgestellt.
Auf der Hardwareebene besteht jedes Signalmodul aus einer Baseband-Platine, einer Synthesizer-Platine und einer Hochfrequenz-Router-Platine. Auf Softwareebene spricht man hier von Modulen.
Die Kommunikation zwischen diesen Modulen erfolgt jeweils über SPI. Mittels Peripheral Component Interconnect Express (PCIe) ist das SoM mit einem Field Programmable Gate Array (FPGA) verbunden, welcher das digitale Basissignal erzeugt.
Sofern nicht anders aufgeführt, basieren diese Module auf einem 8-bit AVR ATxmega128A1 von Microchip Technology Inc. und sind über einen Treiber abstrahiert. Jedes Modul kann über ein eigenes Command Line Interface (CLI), welches innerhalb des Betriebssystems verfügbar ist, direkt angesprochen werden. Der integrierte Synthesizer ist mittels SPI mit drei weiteren Mikrocontrollern, dem Stepattenuator zur analogen Dämpfung, dem Phase-Locked Loop (PLL)-Synthesizer zur Generierung von Lokaloszillator-Signalen und einem Hochfrequenz-Router, verbunden. Die Signalerzeugung erfolgt auf vier Signalpfaden, sogenannten Bundles, von denen jedes acht Kanäle mit theoretisch beliebiger Frequenz ausgeben kann, die jeweils einem simulierten Satelliten entsprechen. Dabei senden die „Satelliten“ eines Bundles immer auf der selben Frequenz, welche als Trägerfrequenz bezeichnet wird. In einer „Switching Matrix“ können die Ausgaben der Bundle wieder kombiniert werden. So können alle aktuell vorhandenen GNSS-Signale simuliert werden. Der schematische Aufbau eines Signalpfades ist in Abbildung 3.3 dargestellt.
Um mit den vier Bundle arbeiten zu können, besitzen die Digital-Analog-Converter (DAC) jeweils zwei Ausgänge. Die DACs werden intern durch einen jeweils 16 Bit großen Wert repräsentiert.
Das Referenzmodul erzeugt als Basis für alle weiteren internen Signale über einen OCXO ein Signal mit einer Frequenz von 122,76 MHz. Zur Erzeugung der, für die Kalibrierung relevanten, Trägerfrequenzen im Bereich von circa 1100 MHz bis 2600 MHz sitzt hinter

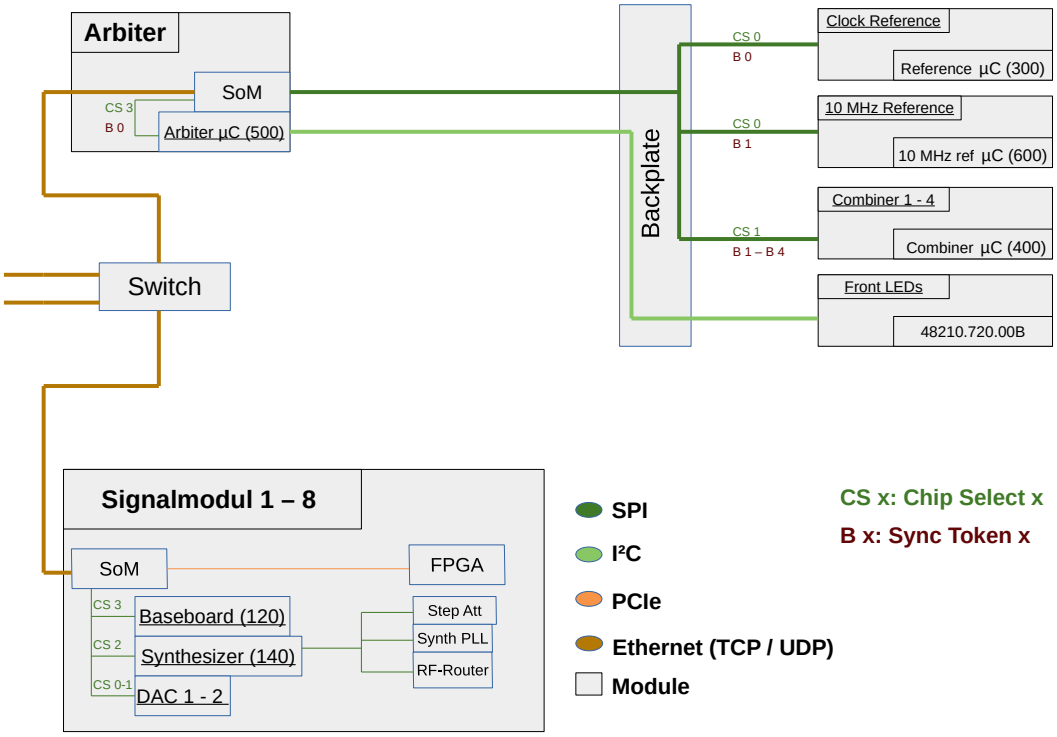

Abbildung 3.2 Schematischer Aufbau eines NCS

dem Referenzmodul ein Multiplikator mit den Stufen 9, 11, 14 und 21. Die Produkte entsprechen dabei dem Wert des Synthesizer-Lokaloszillators und ergeben gemischt mit einer einstellbaren Frequenz zwischen 0 MHz und 100 MHz aus dem DAC die Trägerfrequenzen. Diese werden, wie in Kapitel 5.1 beschrieben, zur Phasenkalibrierung im Combiner mit der Frequenz eines sogenannten „Hilfslokaloszillators“ gemischt, welche +-10 MHz von der zugehörigen Trägerfrequenz liegt. Das relevante Frequenzband leitet sich aus den in Abbildung 3.1 gezeigten Frequenzen der GNSS ab und ist in Abbildung 3.4 explizit aufgezeigt.

3.3 Self-Calibration Detector

Da das Messen der Phasenverschiebung in den verwendeten Frequenzbereichen, siehe Abbildung 3.4, eine technische Herausforderung darstellt, wird auf die sogenannte „Phasenauslöschung“, siehe hierzu auch Kapitel 2.3 und Abbildung 2.2, zurückgegriffen. Die eigentliche Messung wird vom sogenannten „Self-Calibration Detector“ ausgeführt, dessen Schaltung in Abbildung 3.5 schematisch dargestellt ist.

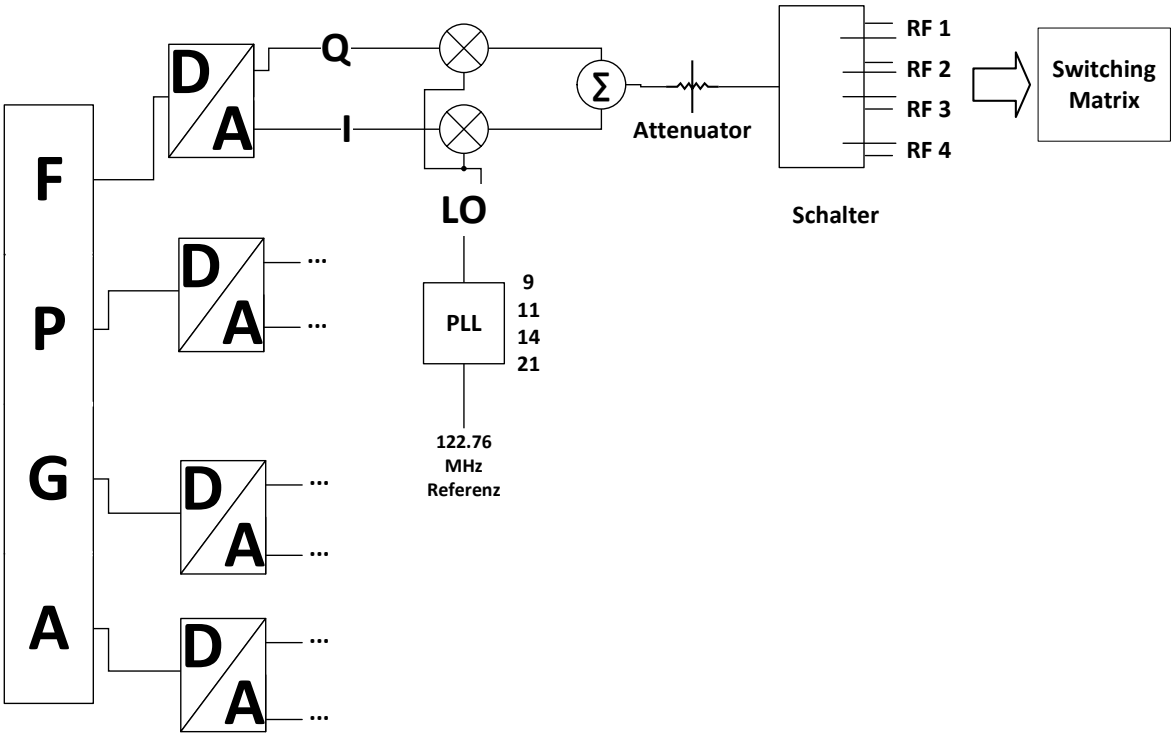

Abbildung 3.3 Schematischer Aufbau eines Signalpfades (Bundle) eines Signalmoduls

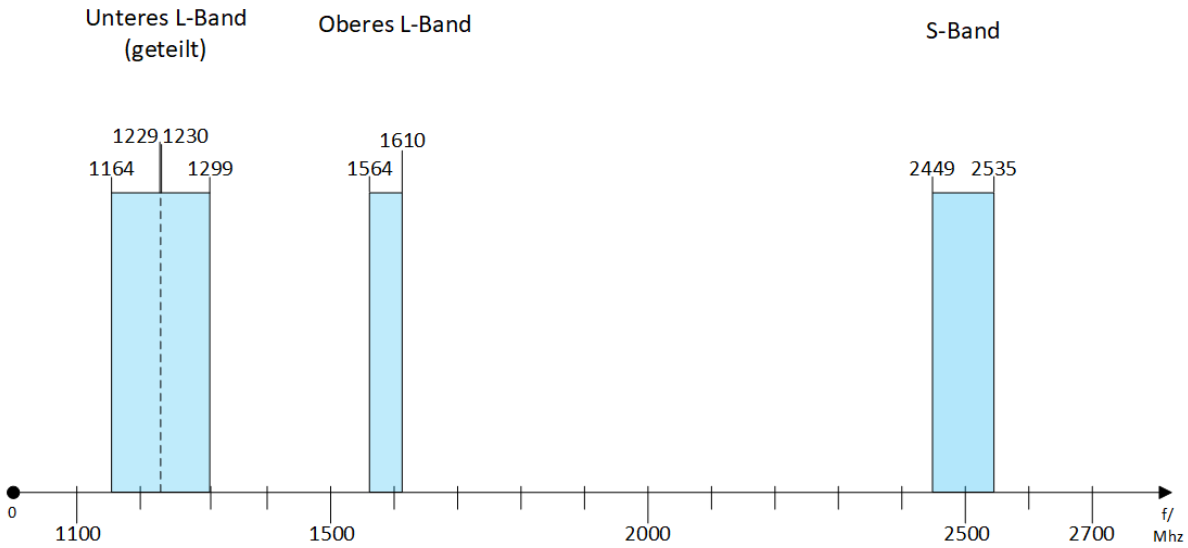

Abbildung 3.4 Plan der Kalibrierfrequenzen

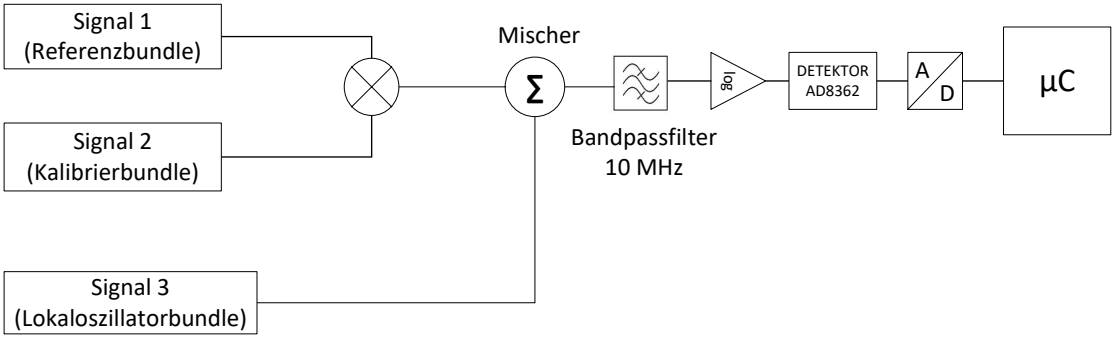

Abbildung 3.5 Schematischer Aufbau der Schaltung des Self-Calibration Detectors

Zwei in der Frequenz gleiche Signale werden in einem Mischer, vergleiche Kapitel 2.3.4, mit einer Hilfsfrequenz, die von einem Lokalsoszillator generiert wird, gemischt. In einem Bandpassfilter werden die unerwünschten Frequenzanteile herausgefiltert. Dieser Aufbau entspricht im Wesentlichen einem sogenannten „additiven Mischer“. Das zu messende Signal wird anschließend über einen logarithmischen Verstärker vom Typ „AD8362“ der Firma Analog Devices Inc. verstärkt und an das Messsystem geleitet. Dessen Ausgabe wird in einem Analog/Digital Umwandler gewandelt und an den Mikrocontroller weitergegeben. Um den Aufwand für die Detektorschaltung gering zu halten und sie auf dem Printed circuit board (PCB) des Combiners integrieren zu können, wird die Hilfsfrequenz von einem der vier Bundle des Signalmoduls erzeugt.

3.4 Ausgangslage Kalibrierung

Da die aktuellen Simulatoren nicht über die nötige für eine Selbstkalibrierung verfügen, wurde jeder NCS als Komplettsystem betrachtet und entsprechend manuell kalibriert. Somit wurde nicht unterschieden, an welcher Stelle auf dem Signalpfad eine Beeinträchtigung auftritt und entsprechend nicht spezifisch bei der Kalibrierung berücksichtigt.
Eine manuelle Kalibrierung mittels externer Werkzeuge, wie Spektrumanalysator und Powermeter, nimmt im Vollausbau mit acht Signalmodulen circa acht Stunden in Anspruch. Dabei ist zu berücksichtigen, dass einzelne Module bis zu zwei Stunden Einschwingzeit benötigen, um auf Betriebstemperatur zu kommen. Eine solche Kalibrierung kann nicht vor Ort beim Kunden durchgeführt werden und verursacht somit unerwünschte Ausfallzeiten. Der autarke, modulare Aufbau der Signalmodule soll es zukünftig ermöglichen, einzelne Module beim Kunden zu tauschen, beziehungsweise nachzurüsten. Dies macht eine Einzelkalibrierung mit isolierter Ermittlung der jeweiligen Beeinträchtigungsanteile unabdingbar.

Grundsätzlich sind dabei alle verbauten Komponenten bis hin zu den verwendeten Kabeln Quellen von Beeinträchtigung, jedoch üben sie unterschiedlich starken Einfluss auf das Ergebnis aus. Den größten Einfluss übt der Phasenfehler im Signalmodul aus. Hier sind viele aktive Komponenten, welche einem stärkeren Temperaturdrift unterliegen, verbaut. Hinzu kommt die selbstauferlegte Genauigkeit von 0.5 Grad bis maximal 1 Grad Abweichung des Phasenwinkels.
Diese Genauigkeit ist erforderlich um simulieren zu können, wie ein Flugzeug, welches über drei Antennen verfügt, über den Empfang des selben Signals an diesen Antennen seine Lage über die x-y-z-Achse im Raum bestimmen kann. Da bei einer Abweichung von 1 Grad bei einer Frequenz von 1 GHz bereits eine Ungenauigkeit von $1/(360 * 10^9)$ Sekunden auftritt, ist es entscheidend, dass ein Signalmodul möglichst phasengleiche Signale ausgeben kann.
Die Modularität erfordert die lokale Ablage der Kalibrierdaten auf den entsprechenden Modulen. Bedingt durch große Datenmengen, die sich aufgrund der komplexen Abhängigkeiten von Frequenz, Temperatur, Amplitude, et cetera. ergeben, kann dies zu Kapazitätsproblemen im 2 Kilobyte (KB) Electrically Erasable Programmable Read-Only Memory (EEPROM) der Mikrocontroller führen. Aus diesem Grund wird aktuell ein Binärdatenformat verwendet, welches weder menschenlesbar noch erweiterbar ist und aufgrund des neuen Speicherorts im Flash der SoM, siehe Kapitel 4.2, ersetzt werden kann.
Eine Übersicht aller zu kalibrierenden Komponenten sowie die Abhängigkeiten untereinander, welche maßgeblichen Einfluss auf die Reihenfolge der Kalibrierung haben, kann Abbildung 4.1 entnommen werden. Bei der Kalibrierung handelt es sich aus technischer Sicht um eine Minimumsuche, somit ist es entsprechend Kapitel 5.1 erforderlich, den Punkt zu finden, an dem das gemessene Resultat bei möglichst geringer Leistung möglichst niedrig ist.
Die eigentliche Kalibrierroutine läuft auf dem Arbiter ab, welcher beim Systemstart die Kalibrierdaten der einzelnen Module einsammelt und anschließend die Routine entsprechend der Einstellungen anstößt. Abschließend schreibt er die neuen Werte in die Dateien zurück und legt sie auf den zugehörigen Modulen wieder ab.

4 Zuarbeit Kalibrierung

Die in diesem Kapitel beschriebenen Tätigkeiten sind, wie Tabelle 1.1 zu entnehmen ist, sowohl Teil des Meilensteins „Entwurfs- und Vorbereitungsphase“ , als auch Teil der „Implementierungs- und Integrationsphase“. Zu Ersterem gehören die Kapitel 4.1, 4.3 sowie das Kapitel 4.4. Die Kapitel 4.2 und 4.6 sind Teil der „Implementierungs- und Integrationsphase“. Aus der Integration in das Embedded-Linux ergab sich die in Kapitel 4.5 beschriebene Tätigkeit „Systemaktualisierung“, welche die erlangte Flexibilität im sequentiellen Vorgehen durch das Einbeziehen von iterativen Aspekten verdeutlicht.

4.1 Datenspeicherung

Da die Module die ermittelten Kalibrierdaten zukünftig in einer 16 MB großen beschreibbaren Partition des Betriebssystems ablegen, war es möglich das bisher verwendete, auf geringen Speicherbedarf optimierte, binäre Datenformat abzulösen. Die Verwendung dieses Binärformates brachte eine Reihe von Nachteilen mit sich, aus welchen sich die Anforderungen an das neue textbasierte Format ableiten ließen. Dieses Format soll menschenlesbar sein, damit Informationen schnell und ohne Aufwand ausgelesen werden können. Dies schließt auch eine übersichtliche Darstellung großer Informationsmengen mit ein. Um in Zukunft weitere Parameter in die Kalibrierung mit aufnehmen zu können, soll das Format einfach erweiterbar sein und dabei Abwärtskompatibilität gewährleisten. Ein Augenmerk sollte auf die Unterstützung des Formates durch die Entwicklergemeinschaft gelegt werden, da somit eine langfristige Unterstützung, beispielsweise in Form aktueller und performanter Parser für die eingesetzten Programmsprachen C und Python, gewährleistet wird. Damit einhergehend ist eine umfangreiche und aktuelle Dokumentation des Formates wünschenswert. Obwohl die neuen Module über einen deutlich größeren Speicher verfügen, bleibt die Verfügbarkeit von Speicher aufgrund der komplexen Abhängigkeiten der Parameter ein Kriterium. Deshalb war darauf zu achten, dass ein potentielles Format keine unnötig komplexe und umfangreiche Syntax aufweist.
Um das Problem des strukturierten Abspeicherns von Daten zu lösen, wurden die drei gängigen Formate JavaScript Object Notation (JSON), eXtensible Markup Language (XML) und Comma Separated Value (CSV) einer genaueren Analyse unterzogen. Das Kriterium

der Dokumentation und langfristigen Unterstützung erfüllen zum jetzigen Stand alle drei Formate, sodass hieran keine Unterscheidung getroffen wurde.
Aufgrund seines Overheads bei der Informationsrepräsentation, bedingt durch öffnende und schließende Tags für jedes Element, wurde XML ausgeschlossen. CSV wurde nicht weiter berücksichtigt, da sich die Trennung der Elemente durch Kommata bei großen Datenmengen als unübersichtlich herausgestellt hat. Als geeignetste Variante erwies sich JSON, da es eine gut lesbare Strukturierung der Daten ermöglicht sowie eine native Unterstützung in Form der Datenstruktur „Dictionary“ in Python bietet. JSON ist gut dokumentiert und wird aufgrund seiner großen Popularität innerhalb der Entwicklergemeinschaft voraussichtlich noch viele Jahre verwendet und unterstützt werden. Zudem besteht die Möglichkeit JSON mittels Binary JavaScript Object Notation (BSON) zur Reduzierung des Speicherbedarfs binär abzulegen. Darüber hinaus ist das unkomplizierte Ablegen der Datendateien in No-Structured Query Language (SQL) Datenbanken zur Archivierung mit JSON möglich.

4.2 Datenstrukturierung

Neben der Kalibrierung der Phase gibt es eine Reihe weiterer Signalparameter, die in Zukunft kalibriert werden sollen, aber über den Kontext dieser Arbeit hinausgehen. Da das Format für alle Kalibrierungen, entsprechend Abbildung 4.1, Anwendung finden soll, wurden sie bei der Ausarbeitung bereits berücksichtigt.
Jeder Komponente aus Abbildung 4.1 wurde eine spezifische JSON-basierte Datei als Vorlage zugewiesen, welche alle im Flashspeicher des SoM hinterlegt sind und auf einer analogen Struktur basieren.
Alle Dateien verfügen über den in Code 4.1 abgebildeten Aufbau aus einem „header“-Objekt mit Metainformationen und einem „data“-Objekt, welches die eigentlichen Ergebnisse der Kalibrierung, in Abhängigkeit der anliegenden Lokaloszillatorfrequenz, enthält. Der Vorteil einer Aufteilung in „header“- und „data“-Objekte sowie die weitere Unterteilung in Arrays liegt im schnellen und unkomplizierten Parsen der Ergebnisse.
Das „header“-Objekt ist aufgebaut aus einer Versionsnummer, welcher im Autocalibration-modul zur Phasenkalibrierung ein entsprechender Parser zur Garantie der Abwärtskompatibilität zugewiesen ist, dem Identifier (ID) der Komponente, welche das Objekt abbildet und dem Namen der Datei. Weitere Eigenschaften sind ein Zeitstempel zur Angabe der letztmaligen Dateiänderung sowie die ID der Kalibriermethode. So können zukünftig unterschiedliche Abläufe, beziehungsweise Kalibriermethoden abgebildet werden.
Das eingebettete „correction_values“-Objekt gibt den Bezeichner des ermittelten Korrekturwertes und seinen Index innerhalb des Ergebnisarrays im „data“-Objekt an. Im

„params“-Objekt werden die Parameter aufgelistet, die für die vorliegende Kalibrierung benötigt wurden, ihre jeweilige Indizes im Ergebnisarray des „data“-Objekts sowie im „params_ranges“-Objekt den Wertebereich, welchen sie annehmen. Um die Ergebnisse später einfacher automatisiert verarbeiten zu können, kann ein Parser einen Datenpunkt im Parameterraum nachschlagen, ohne das nachfolgende „data“-Objekt parsen zu müssen.
Das „data“-Objekt, Beispiel entsprechend Code 4.2, besteht aus einem Array für jedes kalibrierte Bundle mit der Bundle-ID als Schlüssel sowie den ermittelten Korrekturwerten der Kalibrierung als entsprechende Einträge. Die Werte werden dabei, entsprechend den im „header“-Objekt hinterlegten, Indizes abgespeichert. Die Trägerfrequenz „tx_freq“ wird in den Multiplikatorstufen, siehe hierzu Kapitel 3.2, eingetragen. Der Wert für die aktuelle Temperatur ist durch einen Platzhalter belegt, da bislang nicht abschließend geklärt wurde, wie die Temperatur ermittelt werden soll. Denkbar wären neben den exakten Werten, die von einem lokalen Temperatursensor ermittelt werden, auch ein Mittelwert aus verschiedenen Sensoren oder möglicherweise eine Funktion, welche den Temperaturverlauf über die Betriebszeit beschreibt.

Code 4.1 Vorlage der Datenstruktur für die Phasenkalibrierung

```
{
   "header": {
      "version": 0,
      "type_id": 7,
      "type_name": "corr_phase_synth",
      "date": "2019-05-13T13:18",
      "calib_method": 0,
      "correction_values": {},
      "params": {},
      "param_ranges": {
         "lo_freq": [],
         "tx_freq": [],
         "temperature": []
      },
      "bundles": []
   },
   "data": {}
}
```

Code 4.2 Auszug des „data“-Objekts der Datenstruktur für die Phasenkalibrierung

```
{
   ...
   "correction_values": {
      "phase_offset": 0
   },
   "params": {
      "tx_freq": 1,
      "lo_freq": 2,
      "temperature": 3
   },
   ...
   "data": {
   "1": [
      [
         0.541,
         9,
         1164.174,
         20
      ],
      [
         0.528,
         9,
         1229.646,
         20
      ],
      [
         0.529,
         11,
         1230.669,
         20
      ],
   ...
     ]
   }
}
```

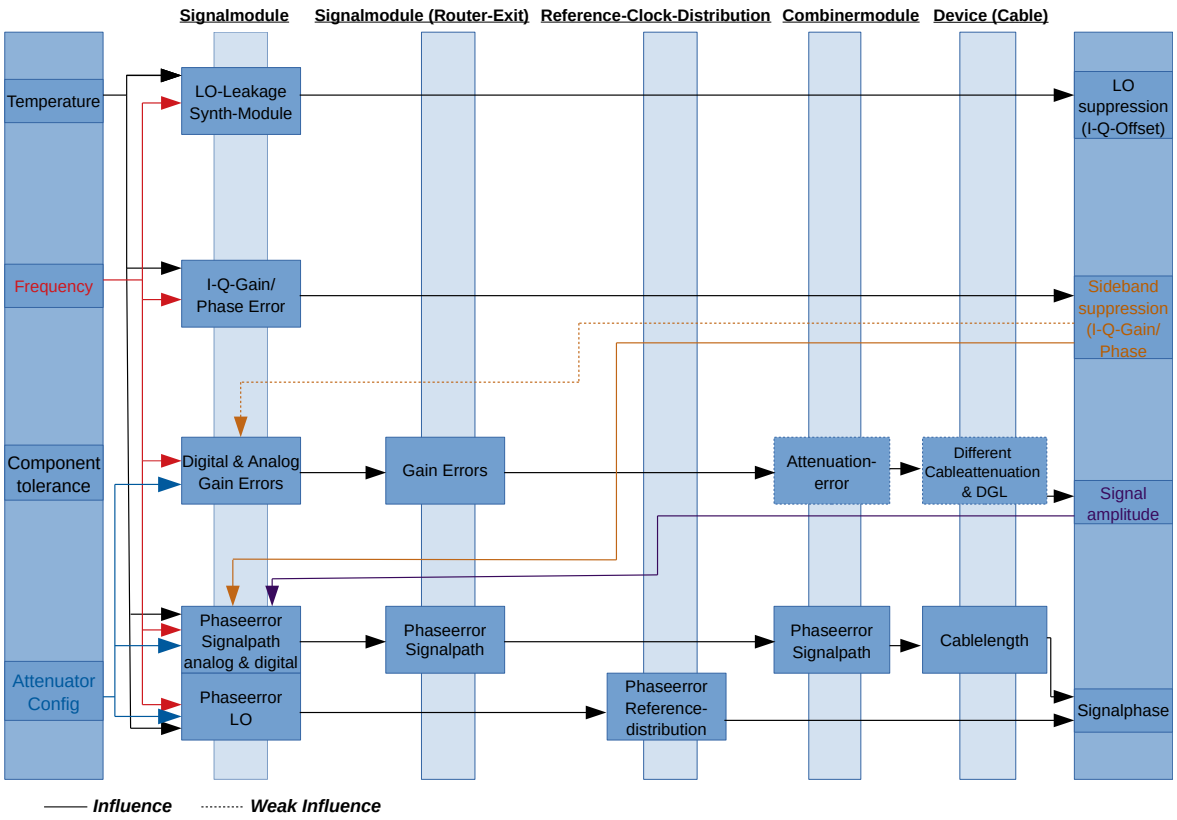

Abbildung 4.1 Überblick der zu berücksichtigenden Komponenten für den Aufbau der Datenstruktur zur Abspeicherung der Kalibrierdaten

4.3 Adaption Treiber und Firmware

Sowohl dem Arbiter als auch den Signalmodulen ist es dank ihrer stärkeren Mikroprozessoren vom Typ PowerPC powerpc-e500v2 möglich, ein Embedded-Linux, welches auf dem LTS Kernel 4.14.26 basiert, auszuführen. Innerhalb dieses Betriebssystems stehen für die ansprechbaren Komponenten in C geschriebene CLIs zur Ansteuerung über SPI zur Verfügung. Die CLIs greifen auf Treiber zurück, welche die verfügbaren Steuerbefehle sowie entsprechende Funktionen um diese abzusetzen, bereitstellen.
Entwickelt wurden diese Komponenten auf dem Buildserver, da auf diesem eine Cross-Compiling-Toolchain für die PowerPC-Architektur verfügbar ist. Die Befehle, um die Leistungsmessung mittels Self-Calibration-Detectors, siehe hierzu Kapitel 3.3, auszuführen, waren firmwareseitig auf einem experimentellen Gitbranch bereits vorhanden. Auf Combinerseite musste der Treiber entsprechend erweitert werden. Anschließend wurde der Zugriff auf den Combiner mittels CLI abstrahiert und die Treiberfunktionen um Informationen zur Fehlerbehandlung ergänzt. Die Einzelschritte der Leistungsmessung wurden zur Zugriffserleichterung zu einem Befehl zusammengefasst.

Code 4.3 Pseudocode der Funktion zur Messung der Leistung mittels des Self-Calibration Detectors

```
int start_measurement_Detection() {

   Detection_t tmp = {0};

   // read current state
   driver_read_Detection(&COMBINER, &tmp);

   // set to calibration mode if necessary
   if(tmp.config != CALIBRATION) {
      driver_write_config(&COMBINER, CALIBRATION);
   }

   // start new measurement
   if(tmp.state != CALIBRATION_STATE_START) {
      driver_write_Detection_state(&COMBINER, CALIBRATION_STATE_START);
   }

   while(tmp.state != CALIBRATION_STATE_COMPLETE) {
      sleep(1);
   }

   // read back result
   driver_read_Detection(&COMBINER, &tmp);

   //print result
   printf("\tresult: %i\n}\n", le16toh(tmp.result));

   return EXIT_SUCCESS;
   }
```

Der Pseudocode in Code 4.3 gibt den grundsätzlichen Ablauf der Messung über die CLI des Combiners wieder. Zunächst wird der aktuelle Status des Detektors gelesen und, falls noch nicht geschehen, in den Modus zur Selbstkalibrierung geschaltet. Anschließend kann die eigentliche Messung angestoßen werden, in dem der Status auf „CALIBRATION_STATE_START“ gesetzt wird. Mittels Polling wird auf das Ende des Messvorganges gewartet, das Ergebnis ausgelesen und auf die Konsole ausgegeben. Von hier aus kann das Resultat weiterverarbeitet werden.

4.4 Ortung der Lokaloszillator-Arbeitsbereiche

Zur Phasenkalibrierung der Bundle werden, entsprechend Kapitel 2.3, die phasenverschobenen Frequenzen mit dem Signal eines dritten frei gewählten Bundles gemischt. Dieses Bundle wird als Lokalsoszillatorbundle oder allgemein als Lokaloszillator bezeichnet, da es lokal eine Frequenz erzeugt. Die Frequenz des Lokaloszillator ist dabei um +-10 MHz verschoben, sodass sie sich innerhalb des jeweiligen Frequenzblockes, siehe hierzu Abbildung 3.4, befindet.
Im ersten Schritt wurde der ideale Arbeitsbereich des Lokaloszillators in Abhängigkeit der eingestellten Frequenz ermittelt. Dieser Arbeitsbereich ist charakterisiert durch ein möglichst hohes Messresultat des Self-Calibration Detectors, siehe Abbildung 3.5, bei zugleich möglichst geringer Leistung des Bundles. Dieser Schritt ist prinzipiell für jeden neuen NCS einmal durchzuführen, die ermittelten Werte können aber auf baugleiche Modelle übertragen werden.
Hierzu wurden Skripte zum Messen und Plotten in Python 3.6 geschrieben und lokal auf dem Entwicklungsrechner mit Verbindung via secure shell (SSH) auf das Signalmodul ausgeführt. Im Rahmen der Ortung der Arbeitsbereiche wurden die, in Kapitel 4.3 vorgestellten, Treiber und Firmware erstmalig im Einsatz getestet. Dabei musste nachträglich berücksichtigt werden, dass der eingesetzte powerpc-e500v2 big-endian ist, die AVR ATxmega128A1 allerdings little-endian.
Zur Ortung wurde für jede der folgenden Frequenzen 1138, 1200, 1690, 1524, 1564, 2394 und 2468 MHz, der Parameter „power_level“ des Lokaloszillator auf -90 Dezibel Milliwatt (dBm) und der Parameter „shifter“ auf Null gestellt. Diese Frequenzen weichen von denen aus Abbildung 3.4 ab, da erst im späteren Verlauf der Kalibrierung auf die exakten Werte gewechselt wurde. Das „power_level“ steht dabei für den Leistungspegel, gemessen in dBm und entspricht in etwa dem der Satellitensignale. Der „shifter“ ermöglicht weiteres Anheben der Leistung durch verschieben dieses Leistungspegels in Bereiche oberhalb von -90 dBm, die über das „power_level“ nicht direkt eingestellt werden können. Praktisch entspricht dies einer bitweisen Verschiebung eines 12 Bit großen Kanals im 16 Bit großen DAC, entsprechend Kapitel 3.2. Somit stehen drei Stufen zur Anhebung des Leistungspegels, welche jeweils 6 Dezibel (dB) entsprechen, zur Verfügung.
Anschließend wurde ein beliebiges anderes Bundle des Signalmoduls bestimmt und beginnend bei einem power_level von -110 dBm, in Schrittweite von 2.5 dBm und einem konstanten Dämpfungswert von 20 bis zum Endwert von -130 dBm vermessen. Dieser Vorgang wurde für alle Frequenzen wiederholt.
Die Angaben in dBm beziehen sich auf den Referenzleistungswert am vorderen Ausgang. Der Self-Calibration Detector misst tatsächlich eine höhere Leistung.

Tabelle 4.1 Optimale Einstellungen für den Arbeitsbereich des Lokaloszillators

Lokaloszillatorfrequenz in MHz	Leistungspegel in dBm
1138	-88
1200	-89
1260	-88
1524	-89
1564	-88
2394	-85
2468	-84

Da es sich bei diesem Vorgang grundsätzlich um eine Maximumsuche handelte, konnten die Ergebnisse, beispielsweise in Abbildung 4.2 für eine Lokaloszillatorfrequenz von 1138 MHz, geplottet und anschließend manuell ausgewertet werden. Die Ergebnisse der Messung entsprechen dabei einem dimensionslosen Wert zwischen 0 und 65535, welcher vom Self-Calibration Detector ausgegeben wird. Tabelle 4.1 fasst die ermittelten idealen Arbeitsbereiche des Lokaloszillators für alle Frequenzen zusammen.

4.5 Systemaktualisierung

Auf den powerpc-e500v2 basierten Modulen des NCS wird ein linuxbasiertes Betriebssystem ausgeführt, welches speziell für den Einsatz in einem NCS angepasst wurde. Zur Erstellung von Bootloader, Kernel und Root-Dateisystem wurde das Framework „buildroot“ eingesetzt. Hierzu stellt es eine Toolchain für die Zielarchitektur des 32-bit PowerPC zur Verfügung. Diese besteht aus der power-pc-GNU Compiler Collection (GCC), den entsprechenden GNU Binary Utilities und der C-Bibliothek uClibC-ng.
Der verwendete generische Bootloader „universal bootloader (u-boot)“ findet vor allem im Umfeld von eingebetteten Systemen Anwendung und läuft auf unterschiedlichen Architekturen.
Die uClibC-ng ist eine Embedded-C-Bibliothek, die aus der eingestellten uClibC hervorgegangen ist und hinsichtlich Größe und Funktionsumfang speziell für den Einsatz auf eingebetteten System zugeschnitten wurde. Innerhalb der Toolchain war als aktuellste Version 1.0.28 verfügbar, welche ein paar Monate hinter der letzten veröffentlichten Version 1.0.31 liegt.
Zur Konfiguration des Kernels wurde auf das grafische „make menuconfig“ zurückgegriffen. Hier ließ sich die verwendete Pythonversion sowie die benötigten Module aus der Standardbibliothek manuell auswählen.
Anschließend wurde durch eine Aktualisierung der Makefiles die CLIs sowie das Autocali-

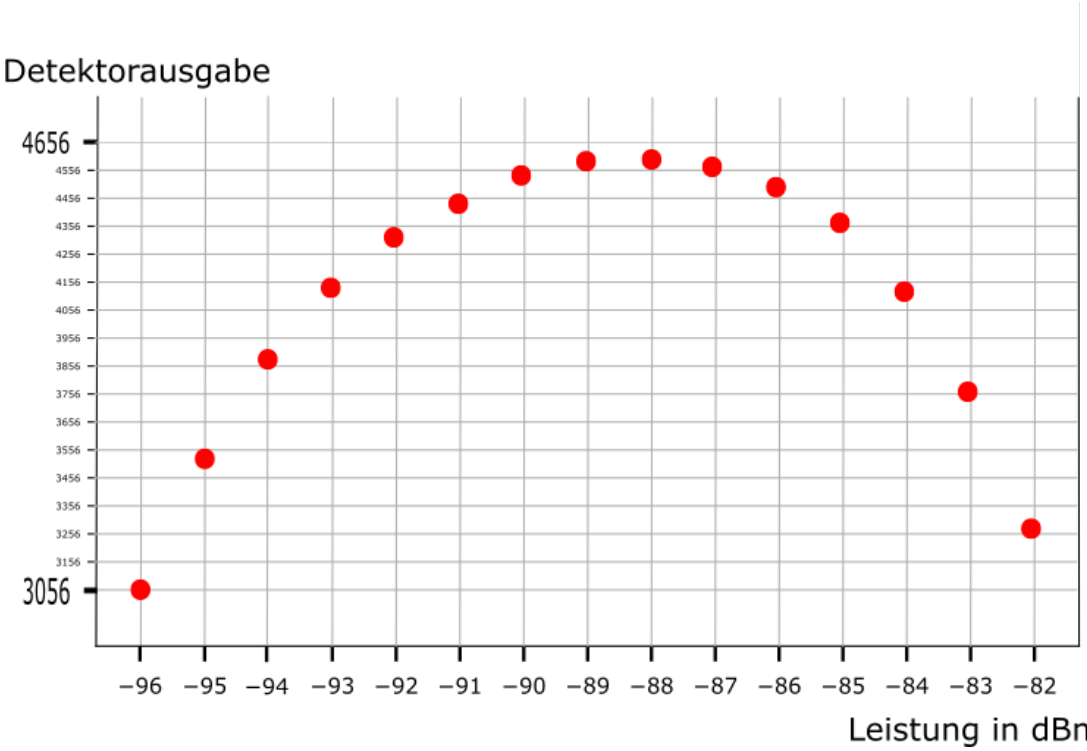

Abbildung 4.2 Plot der Messresultate zur Ortung des Lokaloszillator-Arbeitsbereiches für die Frequenz 1138 MHz

brationmodul mitsamt seiner Abhängigkeiten in das Root-Dateisystem einkompiliert.

4.6 Systemintegration

Die Entwicklung des Autocalibrationmoduls sowie die Implementierung der Routinen für die Lokalisierung des Lokaloszillatorarbeitsbereiches wurden zunächst lokal auf dem Entwicklungsrechner vorangetrieben. Mittels SSH wurden die CLIs, entsprechend Kapitel 4.3, angesprochen, sodass neue Iterationen der Software nicht erst auf den Arbiter übertragen werden mussten. Somit war es möglich, den Pythondebugger der Integrated Development Environment (IDE) PyCharm zu verwenden.
Die Entwicklung erfolgte zunächst in Python 3.6. Jedoch wurde, da kein Upgrade innerhalb des Embedded-Linux geplant war, auf das dort verfügbare Python 2.7 zurückportiert. Notwendige Änderungen, vor allem im Zusammenhang mit der Verarbeitung von Strings, wurden protokolliert, um eine einfache Rückportierung zu gewährleisten.
Nach Abschluss der ersten Entwicklungsstufe innerhalb der „Implementierungs- und Integrationsphase“, die mit der grundsätzlichen Funktionalität des Moduls erreicht war, erfolgte die dauerhafte Integration in das Betriebssystem. Damit einhergehend wurde das

Code 4.4 Pythonfehler nach der Systemintegration des Autocalibrationmoduls

```
(1.) sem_wait: Unknown error -11
Fatal Python error: ceval: tstate mix-up
Aborted

(2.) sem_wait: Unknown error -11
Fatal Python error: PyEval_AcquireThread: non-NULL old thread state
Aborted

(3.) sem_wait: Unknown error -11
Segmentation fault
Fatal Python error: GC object already tracked
Aborted
```

Autocalibrationmodul in die Continous Integration (CI)/Continous Delivery (CD)-Pipeline des Betriebssystems aufgenommen.
Da das Modul somit lokal auf dem Arbiter verfügbar gemacht wurde, konnte der SSH-Code zur Steuerung der CLIs entfernt sowie die interne Kommunikation zwischen Arbiter und DAC von SSH auf das proprietäre navx_gen2-Protokoll umgestellt werden. Die Ansteuerung der CLIs erfolgt nun über Aufrufe aus der Shell heraus. Anschließend wurde das Autocalibrationmodul sowie seine Abhängigkeiten in den NCSTools, entsprechend Kapitel 5.2, in das Embedded-Linux aufgenommen.
Nach der Integration kam es zu einer Reihe von Fehlern, zusammengefasst in Code 4.4, welche die weitere Entwicklung des Autocalibrationmoduls blockierten.

Das Auftreten der Fehler folgte keinem festen Muster, sodass als Anhaltspunkt das *„sem_wait: Unknown error -11“* blieb, welches auf einen Ursprung in den Semaphoren des Betriebssystems hinweist. Zur Eingrenzung der Ursache wurde das Modul auf ein Minimalbeispiel reduziert. Der Fehler konnte so in das threading-Modul der Python-Standardbibliothek, welches von den NCSTools bei der TCP, beziehungsweise User Datagram Protocol (UDP) basierten Kommunikation mit den Signalmodulen verwendet wird, zurückverfolgt werden. Mittels *„print()“*-Statements wurde der Ursprung in der *„wait()“*-Funktion des threading-Moduls, welches intern auf die *„sem_wait()“*-Funktion der uClibC-ng zugreift, lokalisiert. Ein Workaround, basierend auf einer Substitution der *„wait()“*-Funktion durch eine Kombination aus *„sleep()“* und Polling, konnte die Häufigkeit der Fehlerauftritte zwar reduzieren, jedoch nicht ausreichend, um einen stabilen Betrieb des Autocalibrationmoduls zu gewährleisten.
Nach weiteren Recherchen konnte die Verwendung der *„CLOCK_REALTIME“* in der Implementierung von *„PyThread_acquire_lock_timed()“* , welche auf *„sem_wait()“* basiert,

als potentielle Ursache identifiziert werden, da das SoM nicht über eine integrierte Echtzeituhr verfügt. Dieses Problem ist der Pythongemeinschaft bekannt und in Issue 23428 des Python Bugtracker dokumentiert[1].
Da die Implementierung der Semaphore-Advanced Programming Interface (API) abhängig von der C-Bibliothek ist, wurde die eingesetzte uClibC-ng hinsichtlich einer Aktualisierung auf Version 1.0.31 analysiert. In den Veröffentlichungshinweisen waren keine Änderungen im Zusammenhang mit Semaphoren aufgeführt, sodass alternativ ein Wechsel auf die „musel“ -Bibliothek in Erwägung gezogen wurde. Da im buildroot allerdings keine Toolchain für diese Bibliothek mit der PowerPC-Architektur angeboten wird, wurde dieser Ansatz nicht weiter verfolgt.
Entsprechend den Hinweisen in den Kommentaren zu Issue 23428 im Bugtracker wird ab Python 3.5 *„CLOCK_MONOTONIC“* statt *„CLOCK_REALTIME“* eingesetzt. Somit wurde sich zunächst für eine Aktualisierung des eingesetzte Python 2.7 entschieden.
Nachdem der Support für Python 2.7 Ende 2019 eingestellt wird[2], konnte die nötige allerdings noch nicht geplante Aktualisierung auf Python 3.6, der aktuellen Version auf dem Buildserver, vorgezogen werden.
Tatsächlich konnte das Autocalibrationmodul nach der Aktualisierung erfolgreich ausgeführt werden. Der konstant neu auftretende Fehler *„sem_wait: Unknown Error -110“* , hat keine direkten feststellbaren Auswirkungen auf den Programmablauf. Die Ergebnisse entsprachen den selben Werten der Ausführung mittels SSH. Der Fehlercode „-110“ entspricht dabei dem *„errno“* Status *„ETIMEDOU“* und steht für einen „Timeout“, der mutmaßlich irgendwo im Modul oder den NCSTools auftritt.
Abschließend war es erforderlich, die auf Python basierenden Programme des Betriebssystems, wie den Bottle-basierten Webserver und die System-Update-Routine zur Firmwareaktualisierung, an die neue Sprachversion anzupassen.

1 https://bugs.python.org/issue23428, abgerufen am 28.06.2019
2 https://www.python.org/dev/peps/pep-0373/

5 Implementierung Kalibrierung

Die nachfolgenden Kapitel 5.2 und 5.3 sind Teil des Meilensteins „Implementierungs- und Integrationsphase“, entsprechend dem Zeitplan 1.1. Sie beschäftigen sich mit der Umsetzung und Implementierung des Kalibrieralgorithmus. Der in Kapitel 5.1 beschriebene theoretische Ablauf der Phasenkalibrierung wurde in der „Entwurfs- und Vorbereitungsphase“ festgelegt.

5.1 Theoretischer Ablauf des Kalibrieralgorithmus

Da alle Frequenzen im NCS ihren Ursprung im selben 122.76 MHz -OCXO-Oszillator des Referenzmoduls haben, sind sie in der Theorie phasengleich. Allerdings verursachen auf dem Signalweg verbaute Bauteile, beispielsweise Verstärker oder Abschwächer, verschiedene Abweichungen, welche neben der bereits genannten Bauteilalterung der Grund für eine Kalibrierung sind.
Eine technische Herausforderung stellt das Messen dieser Abweichungen in den verwendeten Frequenzbereichen, siehe Abbildung 3.4, dar, da selbst hochwertige Messgeräte an die Grenzen ihres Messbereiches kommen. Aus diesem Grund wird auf die sogenannte „Phasenauslöschung“, siehe hierzu Kapitel 2.3 und Abbildung 2.2, zurückgegriffen, welche den Phasenfehler in einen Amplitudenfehler, zur weiteren Verarbeitung, transformiert. Prinzipiell löschen sich zwei Signale mit identischer Frequenz aus, wenn die Phase eines Signals um $\varphi = 180\,°$ gedreht wird. Somit ist es durch Justierung der Phase möglich zwei phasengleiche Signale zu generieren.
Für die Messung jeder Frequenz aus dem Frequenzband aus Abbildung 3.4, wird die zuvor ermittelte, ideale Einstellung für das Bundle, welches als Lokaloszillator dient, eingestellt. Referenzbundle und Kalibrierbundle werden auf identische, feste Frequenzwerte, die sogenannte Trägerfrequenz, aber mit um 180 Grad gedrehter Phase gestellt. Anschließend werden diese drei Bundle, wie in Kapitel 3.3 beschrieben, im Mischer gemischt und eine Zwischenfrequenz für das weitere Vorgehen erzeugt. Anschließend wird die Phase über das CLI des DAC, in einer Schrittweite von 0.001, entsprechend 0.36 *rad*, über das manuell festgelegte Intervall von 0,5 – 0,6 gestellt und die Ergebnisse des Self-Calibration Detectors aufgezeichnet. Die 360 Grad eines vollständigen Umlaufes

werden intern auf das geschlossene Intervall von 0 – 1 übertragen, sodass das Testintervall von 0,5 – 0,6 einem Winkel von 180 Grad – 216 Grad entspricht. Die Abhängigkeit zwischen der Frequenz des Lokaloszillators und der Trägerfrequenz beträgt +-10 MHz, sodass die Lokalsozillatorfrequenz innerhalb des verwendeten Frequenzblockes bleibt. Zur weiteren Auswertung wurden die Messergebnisse geplottet. Abbildung 5.1 zeigt den Plot der Messresultate für die Frequenz von 1164.174 MHz des Lokaloszillators.
Vermutet wurde zunächst eine Streuung der Ergebnisse, sodass zunächst eine Kurve definiert werden müsste, aus der sich das Minimum ableiten ließe. Hierzu war angedacht, mittels „Curve-Fitting“, siehe Kapitel 2.2, eine möglichst optimale Kurve über die Messpunkte zu legen und anschließend eine Funktion zu bestimmen, deren Minimum sich mittels einer Reihe von Stützpunkten berechnen ließe.
Die ermittelten Phasenwerte zur Korrektur werden abschließend in das Format der neuen Datenstruktur überführt und abgelegt. Diese Phasenkorrekturwerte müssen bei der nächsten Anwendung des NCSs eingestellt werden um identische Signale ausgeben zu können.

5.2 Autocalibrationmodul

Als Grundlage für das Autocalibrationmodul standen die NCSTools, eine Sammlung von Pythonmodulen zur Arbeit mit dem NCS, zur Verfügung. Der Zugriff auf den NCS wurde dabei über die bereitgestellte API des „driver“-Moduls hergestellt. Aus diesem Grund wurde neben der Verfügbarkeit von „scipy“, siehe Kapitel 2.2, entschieden, dass Autocalibrationmodul in Python zu implementieren. Über den Pythondatentyp „Dictionary“ lässt sich zusätzlich die neue JSON-basierte Datenstruktur nativ verwenden.
Die aktuelle Werkskalibrierung ist ebenfalls Teil der NCSTools, sodass hier zum Aufbau der Verbindung über das proprietäre navx_gen2-Protokoll zur Kommunikation mit den Signalmodulen und den DACs sowie zum Setzen der Einstellungsparameter angeknüpft werden konnte.
Bei der Programmierung wurde mit Hilfe des Programms „pylint“ sichergestellt, den Richtlinien von PEP-8 zu entsprechen. Hierbei konnte die maximale Wertung von zehn möglichen Punkten erreicht werden. Zum besseren Verständnis der textbasierten Funktionalitätsbeschreibung des Autocalibrationmoduls wurde mit dem Programm „pyreverse“ ein Klassendiagramm auf Basis von UML, siehe Abbildung 5.2, angefertigt und anschließend manuell angepasst. Zum besseren Verständnis des logischen Programmablaufes, ohne detailliert jeden Funktionsaufruf abzubilden, dient das im Anhang beigefügte Ablaufdiagramm A.1.
Das vollständig in Python 3.6 geschriebene Modul zur automatischen Phasenkalibrierung

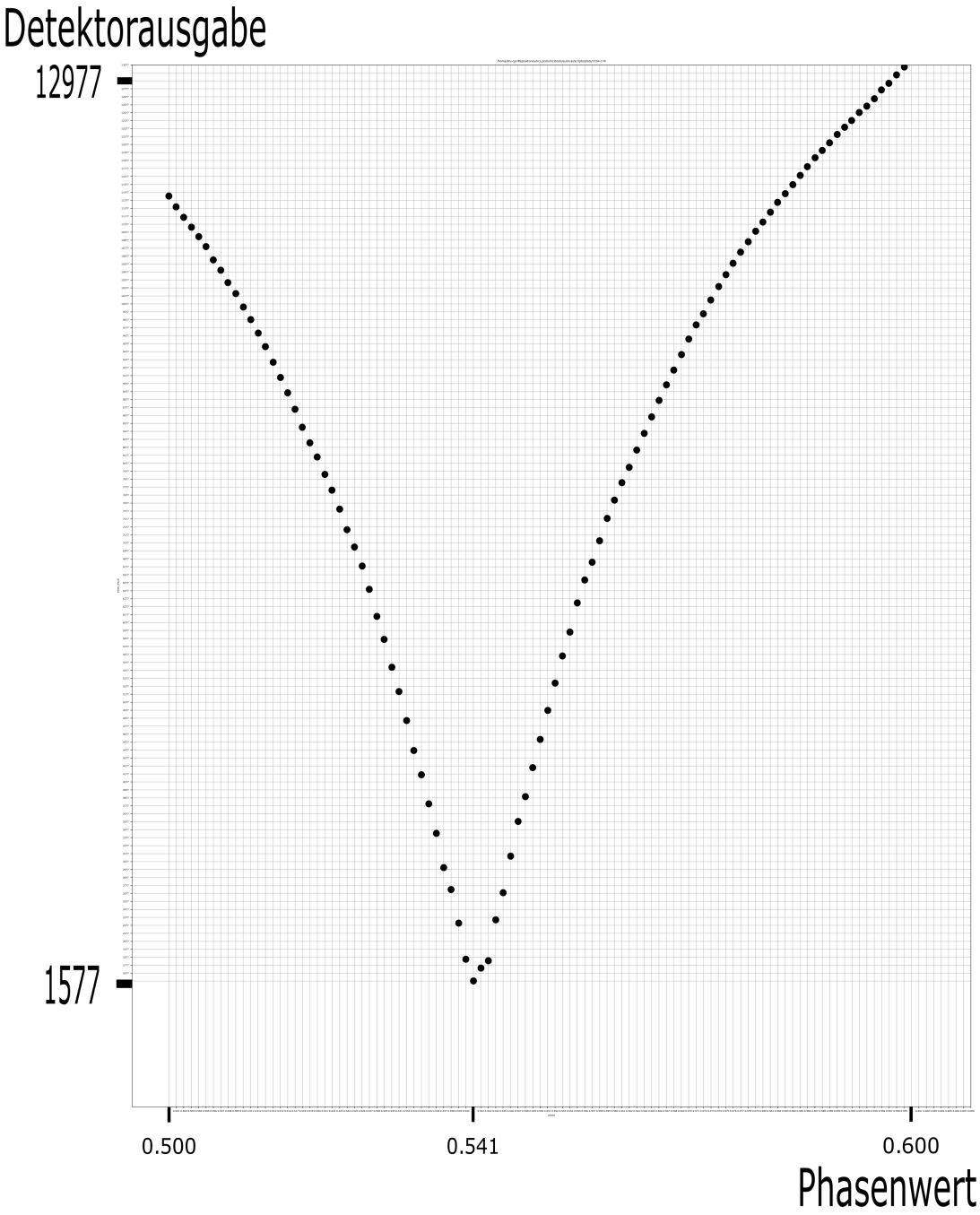

Abbildung 5.1 Plot der Ergebnisse der Phasenmessung für die Frequenz 1164.174 MHz

besteht aus drei Klassen, welche Daten zur Konfiguration des Moduls entgegennehmen, die eigentliche Kalibrierroutine ausführen und der Aufbereitung sowie Abspeicherung der gewonnenen Informationen dienen. Zur Unterstützung gibt es außerdem eine Reihe von Hilfsfunktionen und -klassen. Das Zusammenspiel der Komponenten untereinander und mit den in Kapitel 4.3 beschriebenen C-Komponenten kann Abbildung 5.2 entnommen werden.
Sämtliche variable Parameter, deren Einstellung für die Kalibrierung erforderlich sind, werden aus einer JSON-basierten Konfigurationsdatei ausgelesen. Hierzu gehören die Trägerfrequenzen und die Lokaloszillatoreinstellungen sowie die Einteilung der Bundle in Kalibrierbundle, Referenzbundle und Lokaloszillator. Zudem lassen sich die Parameter der Messroutine, wie das Intervall des Phasenwinkels sowie die Schrittweite der Messung konfigurieren. Ebenfalls variabel bestimmbar ist das Verzeichnis, in welchem die ermittelten Werte innerhalb der beschreibbaren Partition des Speichers des Betriebssystems abgelegt werden. Des Weiteren werden innerhalb dieser als „Configuration“ bezeichneten Klasse für den Zugriff auf die Module entsprechende Wrapper für die Combiner- und DAC-CLIs angelegt. Nach dem Einlesen der Konfigurationsdatei prüft die Klasse alle Werte auf Plausibilität, übermittelt die Einstellungen und stellt dem Anwender entsprechendes Feedback über eine Logging-Schnittstelle zur Verfügung.
Die Klasse „MeasurementValuesManager“ ist für die Aufbereitung der Informationen in die in Kapitel 4.2 vorgestellte JSON-Datenstruktur zuständig, wobei unterschiedliche Versionsstände berücksichtigt werden können. Dazu wird eine sogenannte „Template“-Datei, welche der leeren Datenstruktur aus Code 4.2 für die aktuell ausgeführte Kalibriermethode entspricht, eingelesen. Die Ergebnisse werden parallel in einem internen Puffer zwischengespeichert, welcher abschließend in die Struktur geschrieben wird. Nach Abschluss der Messung wird aus diesem Puffer das Minimum der Messwerte ermittelt und der entsprechende Phasenwert als Phasenkorrekturwert abgespeichert. Die Ergebnisdatei wird zum Schluss in das spezifizierte Verzeichnis geschrieben. Dem Anwender wird über die Logging-Schnittstelle Rückmeldung über den Fortschritt der Messung zur Verfügung gestellt.
Die Wrapper für die Combiner-CLI und die DAC-CLI sind als eigenständige Klassen implementiert und werden als „Manager“ bezeichnet. Sie stellen den Zugriff über die Shell auf die entsprechenden Anwendungen bereit und kümmern sich um das Behandeln von Ausnahmen.
Auf eine weitere Unterteilung wurde aufgrund der geringen Komplexität verzichtet, sodass die „PhaseMeasurementRunner“ Klasse, die Instanzen der „Configuration“ und „MeasurementValuesManager“ Klassen hält.
Entsprechend ihrem Namen ist diese Klasse für die eigentliche Kalibrierroutine verant-

wortlich. Hierzu werden bei der Ausführung alle in der Konfigurationsdatei hinterlegten Frequenzen durchlaufen, die entsprechenden Einstellungsparameter gesetzt und die, als „Simulation“ bezeichnete Routine, innerhalb derer die Messung stattfindet, gestartet. Für jede Frequenz wird die Phase über die Instanz der Wrapperklasse „DACPhaseParam“ in der definierten Schrittweite, beispielsweise 0.001, gesetzt und anschließend über die entsprechende Methode des „CombinerCLIManager“ die Messung mit dem Self-Calibration Detector angestoßen. Das Ergebnis wird aus der Rückgabestruktur extrahiert und der MeasurementValuesManager-Instanz zur Weiterverarbeitung übergeben.
Für den Fall, dass die Ergebnisse von Rauschen verdeckt werden, kann über ein Erhöhen des „Shiftfactor“-Parameters die Signalleistung angepasst und die Messung anschließend wiederholt werden. Dieses Vorgehen ist ähnlich dem, welches in Kapitel 4.4 beschrieben ist.
Um die Schwankungen der einzelnen Messpunkte, sichtbar in Abbildung 5.3, auszugleichen, werden die Messungen, die im Umkreis von +-5 Phasenschritte um das gefundene Minimum angesiedelt sind, jeweils 16 mal wiederholt. Anschließend wird aus den Ergebnissen eines jeden Schrittes der Mittelwert berechnet und als Endergebnis in die Datenstruktur eingetragen. Der Nachteil dieses Vorgehens ist die verringerte Ausführungsgeschwindigkeit der Kalibrierung.
Während des gesamten Ablaufs erhält der Anwender Feedback über die Loggingschnittstelle auf die Konsole.
Um später einen Vergleich mit den Ergebnissen der Werkskalibrierung zu ermöglichen, werden die Resultate des einen Messvorgangs, in dem der finale Korrekturwert bestimmt wurde, in einer eigenen Logdatei gespeichert und auf dem Arbiter abgelegt.
Das Autocalibrationmodul liegt als Bytecode im Embedded-Linux vor und kann mittels entsprechendem Aufruf über den Pythoninterpreters gestartet werden.

5.3 Optimierungsmöglichkeiten und Ausblick

Um neue Funktionalitäten zu testen, Fehlerbehandlungsroutinen einzubauen und Messergebnisse vergleichen zu können, wurde das Autocalibrationmodul im Rahmen der inkrementellen Weiterentwicklung wiederholt ausgeführt. Dabei wurde nach einer unregelmäßigen Anzahl von Ausführungen immer ein Fehler in der SoM protokolliert. Entsprechend dem Log wurde die Verbindung zum Combiner verloren, womit das Autocalibrationmodul nicht mehr in der Lage war über den Treiber mit diesem zu kommunizieren und in der Folge abstürzte. Erst nach einem Neustart des NCS war die Kommunikation wieder möglich.
Die SPI-Schnittstelle konnte als potentielle Ursache ausgeschlossen werden, da die Kommunikation mittels Lock-File, über die Betriebssystemfunktion „flock()“ geschützt ist. Auch

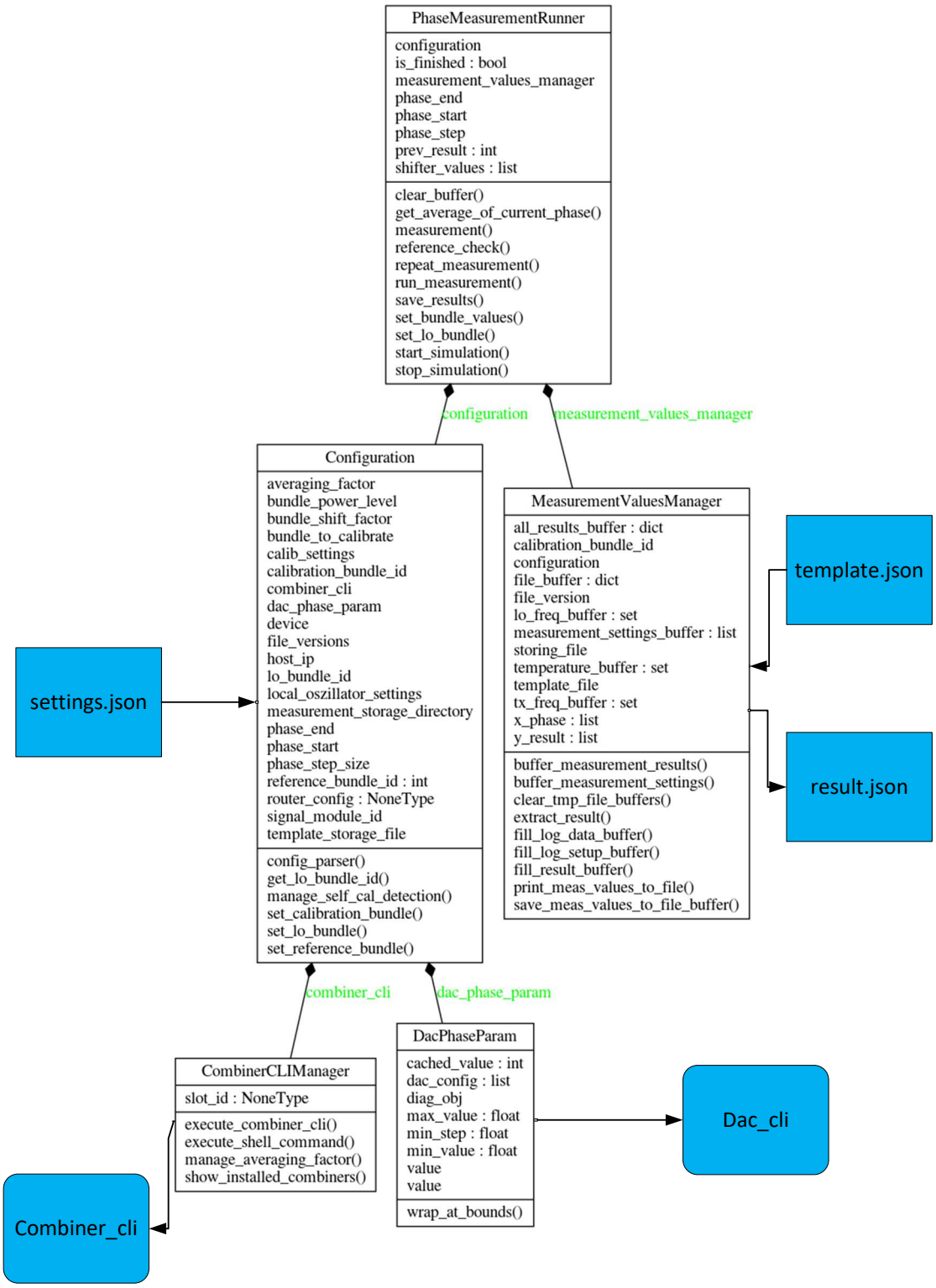

Abbildung 5.2 UML-Klassendiagramm des Autocalibrationmoduls zur Kalibrierung der Phase

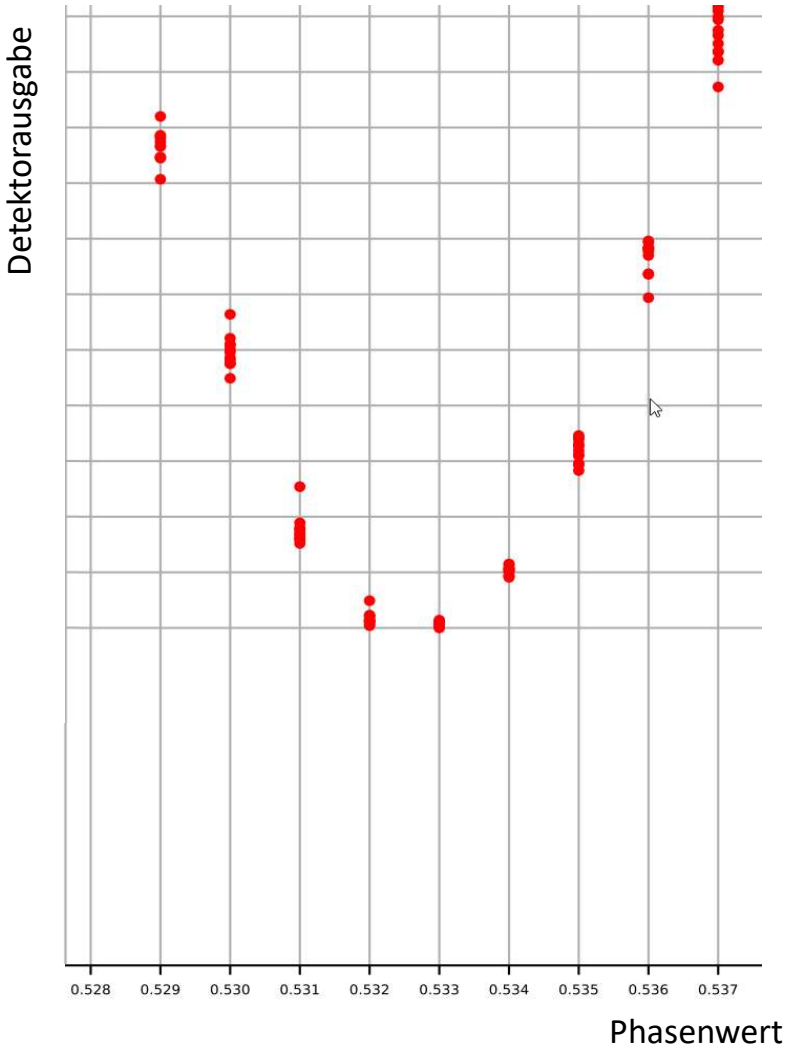

Abbildung 5.3 Auzug aus dem Plot der Messung zur Feststellung des Grades der Verrauschung der Messergebnisse bei einer Lokaloszillatorfrequenz von 1148 MHz

eine Verringerung der Anfragen über die Schnittstelle, in dem ein *„sleep()“* von einer Millisekunde beim Pollen der Messergebnisse, vergleiche Code 4.3, eingebaut wurde, brachte keine Lösung.
Somit bleibt die Ursache für dieses Problem unklar und wurde entsprechend dokumentiert.
Beim Vergleich der Ergebnisse des Autocalibrationmoduls mit denen der Werkskalibrierung ergaben sich teils deutliche Differenzen bei den ermittelten Phasenkorrekturwerten. Der Grund dafür liegt, obwohl in der Theorie kein Einfluss ausgeübt wird, in der Amplitudenabweichung der Signale. Diese scheint stark genug zu sein, über den Self-Calibration Detector einen Einfluss auf den Messvorgang auszuüben, sodass es zu einer Verfälschung der Messergebnisse kommt.
Daraus folgt die Notwendigkeit einer Kalibrierung sowohl über die Phase als auch über die Amplitude, bis in beiden Fällen ein Minimum gefunden wurde. Die Erweiterung der Phasenkalibrierung zu einer zweidimensionalen Kalibrierung konnte aus zeitlichen Gründen nicht mehr durchgeführt werden. Um weitere Nachforschungen in diesem Bereich zu ermöglichen, wurde zusätzlich ein Logging-Mechanismus eingebaut, der alle ermittelten Werte der vollständigen Kalibrierung in einer Datei ablegt. So kann der komplette Ablauf nachvollzogen werden.
Die ermittelten Messpunkte lagen, anders als zunächst vermutet wurde, nicht stark verteilt vor, sondern ermöglichten das direkte Auslesen eines Minimums aus den aufgezeichneten Messwerten. Somit konnte auf den Einsatz von Curve-Fitting, siehe Kapitel 2.2, verzichtet werden. Allerdings kann nicht garantiert werden, tatsächlich das Minimum der Messpunkte gefunden zu haben, da gegebenenfalls die Schrittweite weiterhin zu groß sein könnte. Eine Abschätzung hierzu kann erst bei einem aussagekräftigen Vergleich mit der Werkskalibrierung getroffen werden. Sollte der Einsatz von Curve-Fitting eine Verbesserung versprechen, muss berücksichtigt werden, dass die scipy-Bibliothek für die PowerPC Architektur nicht verfügbar ist. Hier würde sich gegebenenfalls eine Eigenimplementierung anbieten.
Die Laufzeit einer vollständigen Messreihe über alle Frequenzen aus dem Frequenzplan in Abbildung 3.4 beträgt etwa reale 20 Minuten. Gemessen wurde dieser Wert mit der Betriebssystemfunktion „time“. Um diesen Wert zukünftig noch optimieren zu können, wäre es beispielsweise möglich, nach dem Erhöhen des Shifters nur einen Teil des Intervalls zu durchsuchen. Dies ist möglich, da die Messkurven im Intervall stetig fallen, beziehungsweise steigen und somit abgeschätzt werden kann, wo es zu einem Minimum kommt. Abbildung 5.1 unterstreicht dies grafisch.
Zudem kann vermutlich Zeit gespart werden, wenn die Aufrufe des Self-Calibration Detectors nicht mehr in eigenen Subprozessen erfolgen, sondern gebündelt werden können.

6 Zusammenfassung

Im Rahmen dieser Arbeit wurden erstmalig Informationen im Zusammenhang mit dem Self-Calibration Detector und dem Aufbau einzelner Module des NCS zentral gebündelt und in zahlreichen Skizzen grafisch aufbereitet. So wird zukünftig die Einstiegsschwelle in die Thematik der automatische Selbstkalibrierung eines NCS niedriger ausfallen.
Die Funktionsweisen sowie potentielle Optimierungsmöglichkeiten der CLI und des Autocalibrationmoduls wurden dokumentiert. Basierend auf dieser Arbeit kann mit der Weiterentwicklung einer vollständigen automatischen Selbstkalibrierung die manuelle Kalibrierung entfallen.
Der abschließende Vergleich der durchgeführten Tätigkeiten mit dem Zeitplan zeigt, dass die Mehrzahl der Vorgaben erfolgreich erfüllt werden konnten. Das optionale Webinterface konnte aus zeitlichen Gründen nicht mehr implementiert werden, sodass eine Möglichkeit fehlt, bequem von außen mit dem Modul zu interagieren. Auch fehlt eine grafische Aufbereitung der Ergebnisse für den Endanwender. Hierzu können die im JSON-Format aufbereiteten Messergebnisse direkt genutzt werden.
Der aktualisierte SPI-Treiber und die Firmware integrieren den Zugriff auf den Self-Calibration Detector, welcher über die neue CLI des Combiners angesteuert und seine Rückgabestrukturen ausgelesen werden kann.
Das eingeplante Testen der Hardware war nicht erforderlich, da diese wie erwartet funktioniert. Dafür war es notwendig, das Embedded-Linux für die Integration des Autocalibrationmoduls anzupassen. Erst Python ab Version 3.5 ermöglicht den Einsatz von „CLOCK_MONOTONIC“, was möglicherweise die Lösung der Probleme mit den Semaphoren war. Da das eingesetzte Python 2.7 aus dem Support fällt, garantiert dieser Schritt, das auch nach Ende 2019 eine unterstütze Pythonversion verfügbar ist.
Einem ersten Praxistest wurden Treiber, Firmware und CLI bei den Testreihen zur Bestimmung des optimalen Arbeitsbereiches des Self-Calibration Detectos unterzogen. In diesem Schritt wurde auch das Signalmodul erstmalig über die neue Software angesteuert.
Anschließend konnte die Datenstruktur zur Speicherung der Korrekturwerte und der zugehörigen Einstellungen auf Basis von JSON entworfen werden. Hierzu wurde zunächst eine Analyse der potentiellen Formate JSON, CSV und XML anhand einer Reihe von Kriterien, die sich aus den Unzulänglichkeiten des bisher verwendeten binären Formates ergeben

hatten, durchgeführt. Da die Datenstruktur im weiteren Verlauf der Entwicklung produktiv eingesetzt wurde, konnte in einer ersten Abschätzung ihr Aufbau positiv bewertet werden. Noch offen ist, in welcher Form die ermittelte Temperatur des NCS in die Datenstruktur eingetragen wird.
Im Rahmen des zweiten Meilensteins wurde anschließend die Implementierung des Autocalibrationmoduls in Python 3.6 begonnen. Die Entwicklung erfolgte in iterativen Schritten und zog sich, stetig verbessert, über die anschließenden Wochen hin. Erfreulicherweise waren die Messergebnisse in ihrer Verteilung so genau, dass der Einsatz von Curve-Fitting nicht nötig war. Stattdessen konnte über das gemessene Minimum der Phasenkorrekturwert einfach ausgelesen werden.
Beim Vergleich der ermittelten Messwerte mit denen der Werkskalibrierung stellte sich heraus, dass die Ungenauigkeit der Amplitude das Messergebnis negativ beeinflusst, sodass neben der Phase auch die Amplitude optimiert werden müsste. Dies war aus Zeitgründen nicht mehr möglich.
Davor wurde das neue Modul mit seinen Abhängigkeiten in die bestehende Anwendung integriert. Hierzu mussten eine Reihe kleinerer Anpassung an der Codebasis, sowie eine Aktualisierung der Makefiles vorgenommen werden. Das Modul kann zur Ausführung auf dem Arbiter dem Pythoninterpreter übergeben werden.
Die Ausführung der Messung ist bezüglich ihrer Laufzeit noch nicht optimal. So wäre es nach der Erhöhung des Shifters nicht mehr nötig, das gesamte Messintervall zu durchfahren, da anhand der Ergebnisse bereits eine Abschätzung möglich ist, an welcher Stelle es zu einem Minimum kommen wird. Aktuell wird beim Ansteuern des Self-Calibration Detectors ein neuer Subprozess gestartet. Dies ließe sich gegebenenfalls optimieren.
Um die Kalibrierung anzustoßen gibt es verschiedene denkbare Szenarien, zum Beispiel zu einem regelmäßigen Zeitpunkt oder sobald ein zu definierender Schwellwert erreicht wurde. Diesbezüglich wurde bislang keine Entscheidung getroffen.
Die abschließend angesetzte Verifikationsmessung musste aus genannten Gründen entfallen. Stattdessen wurde eine Möglichkeit eingebaut, mit der alle Messwerte, die zur Ermittlung des Phasenkorrekturwertes herangezogen wurden, abgespeichert werden können.

A Ablaufdiagramm Autocalibrationmodul

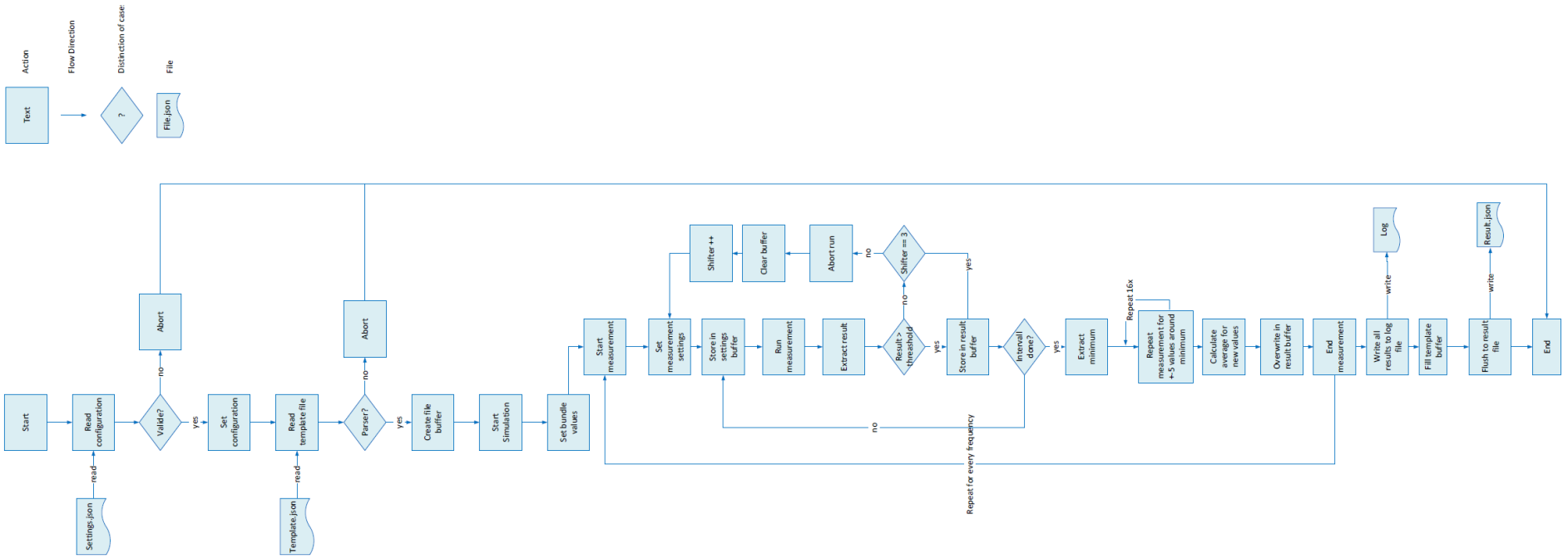

Abbildung A.1 Logischer Programmablauf des Autocalibrationmoduls

Literaturverzeichnis

[All18] B. Allbee. *Hands-On Software Engineering with Python: Move Beyond Basic Programming and Construct Reliable and Efficient Software with Complex Code*. Packt Publishing, 2018. ISBN: 1788622014, 9781788622011.

[Ana18] M. Anaya. *Clean Code in Python: Refactor your legacy code base*. Packt Publishing, 2018. ISBN: 9781788837064.

[Bau] J. Bausch. TCXO vs. OCXO. `https://www.electronicproducts.com/Passive_Components/Oscillators_Crystals_Saw_Filters/TCXO_vs_OCXO.aspx`, abgerufen am 09.07.2019.

[Deu] Deutsches Zentrum fuer Satelliten Kommunikation e.V. WORK Microwave GmbH. `https://desk-sat.com/index.php?id=158`, abgerufen am 25.04.2019.

[DK17] E. D. Kaplan und C. Hegarty. *Understanding GPS: Principles And Applications*. Artech House, 11 2017. ISBN: 1630810584, 978-1630810580.

[EK] Elektronik-Kompendium.de. Modulation / Modulationsverfahren. `https://www.elektronik-kompendium.de/sites/kom/0211195.htm`, abgerufen am 06.07.2019.

[HAM] HAMEG Instruments. Was ist Rauschen? `https://cdn-reichelt.de/documents/datenblatt/TIPP/HAMEG_WasistRauschen.pdf`, abgerufen am 09.07.2019.

[hft] LEXIKON DER PHYSIK Hochfrequenztechnik. `https://www.spektrum.de/lexikon/physik/hochfrequenztechnik/6785`, abgerufen am 09.07.2019.

[Huf] M. Hufschmid. Mischer. `http://www.informationsuebertragung.ch/Extras/Mischer.pdf`, abgerufen am 10.08.2019. Ergaenzung zum Buch Information und Kommunikation, Grundlagen und Verfahren der Informationsuebertragung ISBN 978-3-8351-9077-1, Vieweg + Teubner Verlag, 2007.

[Ife] Ifen GmbH. PRESS RELEASE: IFENS NEW NCS TITAN GNSS SIMULATOR - THE MOST POWERFUL AND FLEXIBLE GNSS SIMULATOR, IN A COST EFFECTIVE PACKAGE. https://bit.ly/2Yo5RGj, abgerufen am 10.07.2019.

[Inta] International Electrotechnical Vocabulary. 161-02-14 Rauschen. https://www2.dke.de/de/Online-Service/DKE-IEV/Seiten/IEV-Woerterbuch.aspx?search=Rauschen, abgerufen am 09.07.2019.

[Intb] International Electrotechnical Vocabulary. 713-10-06 Lokaloszillator. https://www2.dke.de/de/Online-Service/DKE-IEV/Seiten/IEV-Woerterbuch.aspx?search=Lokaloszillator.

[itw] HF (Hochfrequenz). https://www.itwissen.info/Hochfrequenz-HF-high-frequency-HF.html, abgerufen am 03.08.2019.

[Nat] National Coordination Office for Space-Based Positioning, Navigation and Timing. GPS Accuracy. https://www.gps.gov/systems/gps/performance/accuracy/#how-accurate, abgerufen am 23.04.2019.

[Pap17] L. Papula. *Mathematische Formelsammlung: Für Ingenieure und Naturwissenschaftler*. Springer Fachmedien Wiesbaden, 2017. ISBN: 9783658161958.

[Plu] W. Pluta. Galileo fiel durch unglücklichen Zufall aus. https://www.golem.de/news/satellitennavigation-galileo-ist-wieder-online-1907-142814.html, abgerufen am 02.08.2019.

[Pop] G. Popp. *Konfigurationsmanagement mit Subversion, Maven und Redmine: Grundlagen fuer Softwarearchitekten und Entwickler*. dpunkt.verlag GmbH. ISBN: 9783864900815.

[Pro] F. Prommer. Der erste Mieter steht fest: Diese High-Tech-Firma zieht in das Ex-Panasonic-Gebaeude. https://bit.ly/31dqIZS, abgerufen am 25.04.2019.

[Rod] J. A. Rodriguez. BeiDou Signal Plan. https://gssc.esa.int/navipedia/index.php/BeiDou_Signal_Plan, abgerufen am 11.07.2019.

[Rus] Russian institute of space device engineering. Precision of GLONASS navigation definitions. http://www.sdcm.ru/smglo/st_glo?version=eng&repdate&site=extern, abgerufen am 23.04.2019.

[Sch97] R. Schröter. *Allgemeine Meßtechnik*, S. 3–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN: 978-3-642-60437-9.

[Sch13] A. Schöne. *Digitaltechnik und Mikrorechner*. Vieweg+Teubner Verlag, 2013. ISBN: 9783322842350.

[sig] S/N (Signal-Rausch-Verhaeltnis). https://www.itwissen.info/Signal-Rausch-Verhaeltnis-S-N-signal-to-noise-ratio-SNR.html, abgerufen am 09.07.2019.

[Spi] Spirent Communications, Inc. TESTING GPS WITH A SIMULATOR. https://www.spirent.com/assets/eb/eb_testing-gps-with-a-simulator, abgerufen am 01.08.2019.

[Spi14] A. Spillner, T. Linz, T. Rossner und M. Winter. *Praxiswissen Softwaretest - Testmanagement*. dpunkt.verlag GmbH, 2014. ISBN: 978-3-86490-052-5.

[Ste] P. Steigenberger, A. Hauschild, O. Montenbruck und U. Hugentobler. Galileo, Compass und QZSS: Aktueller Stand der neuen Satellitennavigationssysteme. https://geodaesie.info/zfv/heftbeitrag/1114, abgerufenam18.04.2019. Fachbeitrag in der zfv - Zeitschrift fuer Geodaesie, Geoinformation und Landmanagement vom 01.2013.

[Ten] F. Tenzer. Prognose zum Absatz von Smartphones weltweit von 2010 bis 2023. https://de.statista.com/statistik/daten/studie/12865/umfrage/prognose-zum-absatz-von-smartphones-weltweit/, abgerufen am 24.06.2019.

[Tex] Texas Instruments. System-On-Module. http://processors.wiki.ti.com/index.php/System-On-Module, abgerufen am 11.07.2019.

[The] The SciPy community. scipy optimize curve fit. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html, abgerufen am 17.07.2019.

[uSCuGE02] T. U. und Schenk C. und Gamm E. *Halbleiter-Schaltungstechnik 12. Auflage*. Springer-Verlag GmbH, 2002. ISBN: 3540428496.

[wha] What is GNSS. https://www.gsa.europa.eu/european-gnss/what-gnss, abgerufen am 04.03.2019.

[WOR] WORK Microwave GmbH. About us. https://work-microwave.com/about/, abgerufen am 04.03.2019.