Zentralinstitut für Engineering, Elektronik und
Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

Konzeptionierung eines Softwaremodells zur Abstraktion der Hardwaretreiber von Arbiträrgeneratoren

Lukas Lankes

Jül-4421

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Berichte des Forschungszentrums Jülich
Jül-4421 · ISSN 0944-2952
Zentralinstitut für Engineering,
Elektronik und Analytik (ZEA)
Systeme der Elektronik (ZEA-2)
DE-A96 (Master, FH Aachen, 2019)

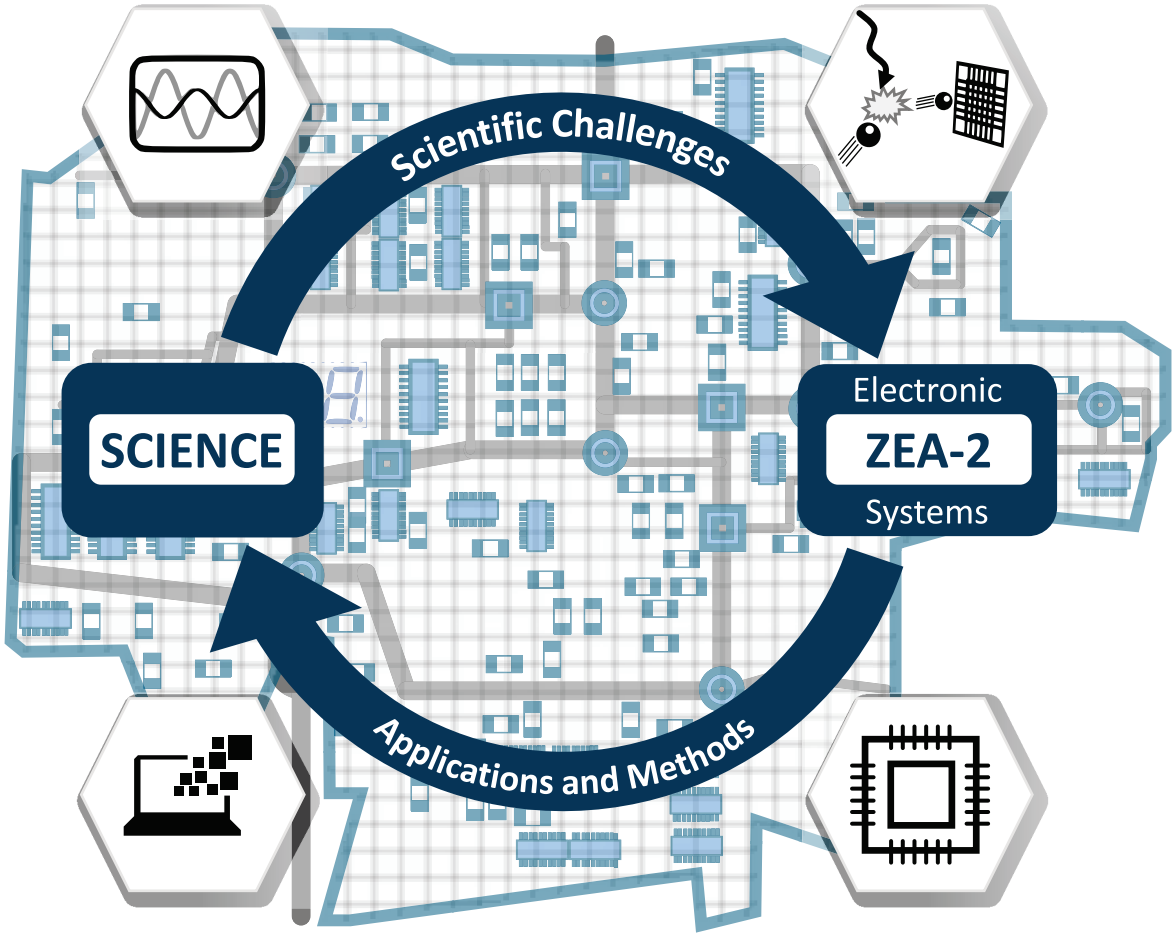
Vollständig frei verfügbar über das Publikations-
portal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess

Forschungszentrum Jülich GmbH · 52425 Jülich
Zentralbibliothek, Verlag
Tel.: 02461 61-5220 · Fax: 02461 61-6103
zb-publikation@fz-juelich.de
www.fz-juelich.de/zb

Zentralinstitut für Engineering, Elektronik und Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

Konzeptionierung eines Softwaremodells zur Abstraktion der Hardwaretreiber von Arbiträrgeneratoren

Lukas Lankes


Scientific Challenges
SCIENCE
Electronic
ZEA-2
Systems
Applications and Methods

Kurzfassung

Diese Arbeit beschäftigt sich mit der Konzeptionierung eines neuen Treibersystems für Arbiträrgeneratoren, welche hierbei im Umfeld der Forschung an Quantencomputern eingesetzt werden. Dabei handelt es sich um digitale Funktionsgeneratoren, die diskrete Spannungssignale erzeugen können. Das dabei konzipierte Treibersystem soll verständlich und intuitiv zu bedienen sein. Außerdem soll eine Erweiterung der Funktionalität und der unterstützten Geräte ermöglicht werden.

Zu Beginn der Arbeit wird eine Einführung in die Grundlagen der Quantenmechanik, insbesondere in den Bereich des Quantencomputings, gegeben. Dazu wird ein exemplarischer Versuchsaufbau beschrieben, in denen ein Arbiträrgenerator zum Einsatz kommt, um Qubits zu kontrollieren, welche die kleinste Recheneinheit eines Quantencomputers darstellen. Anschließend wird das Softwareframework *qupulse* thematisiert, welches unter anderem die Treiber der Arbiträrgeneratoren beinhaltet. Mit *qupulse* lassen sich diskrete Spannungssignale definieren und auf den Arbiträrgenerator übertragen. Die von dem Arbiträrgenerator abgespielten Spannungssignale dienen zur Manipulation und Messung von Qubits, die auf den Gesetzen der Quantenmechanik basieren.

Anschließend folgt die Analyse von drei Arbiträrgeneratoren, die momentan von *qupulse* unterstützt werden. Anhand der dabei herausgestellten Funktionen und Eigenschaften der Instrumente, wird ein neues Konzept zur Abstraktion von deren Treiber entwickelt. Durch die Abstraktion sollen alle Treiber einheitlich bedienbar sein. Außerdem soll sichergestellt werden, dass das Treibersystem um neue Treiber sowie die einzelnen Treiber um neue Funktionalität erweiterbar sind. Durch die Verwendung und Einhaltung von Entwurfsmustern und Richtlinien aus der Softwaretechnik wurde außerdem eine intuitive und verständliche Schnittstelle für die Anwender geschaffen.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zentralinstitut für Engineering, Elektronik und Analytik 1
1.2 JARA-Institut für Quanteninformation 2
1.3 Motivation 2

2 Grundlagen **5**

2.1 Quantenmechanik 5
2.1.1 Quantencomputer 9
2.2 Versuchsaufbau 12
2.2.1 Messung eines Qubits 15
2.3 Software zur Pulserzeugung 16
2.3.1 Geschichte 17
2.3.2 Ziele des Projekts 17
2.3.3 Aufbau 19

3 Hardwareanalyse **25**

3.1 Arbiträrgeneratoren 25
3.1.1 Tabor WX2184C 26
3.1.2 Tektronix AWG5014C 28
3.1.3 Zurich Instruments HDAWG8 30
3.2 Vergleich der Kernfunktionen und Eigenschaften 31

4 Konzeptionierung **39**

4.1 Zielsetzung 39
4.1.1 Hardwareabstraktion 40
4.1.2 Optimierung und Erweiterung der Funktionalität 40
4.1.3 Softwarearchitektur und Clean Code 41
4.2 Grobkonzept 46
4.2.1 Struktur der Hardwaretreiber 47
4.2.2 Abstraktion der hardwarespezifischen Funktionalität 52

4.3 Feinkonzept 59
4.3.1 Kernfunktionen und Eigenschaften der Hardware 60
4.3.2 Vereinfachung der Konfiguration der Treiberspezifikationen 68
4.3.3 Tests und Validierung 70

5 Zusammenfassung und Ausblick **77**

Abbildungsverzeichnis

2.1 Doppelspaltexperiment mit einem Elektronenstrahl 6
2.2 Schrödingers Katze 8
2.3 Bloch-Kugel 10
2.4 Elektronen-Spinqubits in GaAs 13
2.5 Schematischer Aufbau eines Quantencomputers 14
2.6 Messung des Zustands eines Singlet-Triplet-Qubits 16
2.7 Ablauf einer Messung mit „qupulse“ 20

3.1 Tabor WX2184C 26
3.2 Tektronix AWG5012C 28
3.3 Zurich Instruments HDAWG8 30

4.1 UML-Klassendiagramm der Struktur um die Klasse „Device“ 47
4.2 UML-Klassendiagramm der Klasse „Device“ 48
4.3 UML-Klassendiagramm der Klasse „Channel“ 49
4.4 UML-Klassendiagramm der Klasse „ChannelTuple“ mit der abstrakten Basisklasse „AWG“ 50
4.5 UML-Klassendiagramm der Klasse „ProgramManager“ 52
4.6 UML-Klassendiagramm zur Standardisierung der „synchronize“-Funktion in der Basisklasse 53
4.7 UML-Klassendiagramm zur Standardisierung der „synchronize“-Funktion in einem Interface 54
4.8 UML-Klassendiagramm zur Standardisierung hardwarespezifischer Funktionen in Anlehnung an das Bridge-Pattern 54
4.9 Abstraktionsschicht der AWG-Treiber 58
4.10 Graphansicht einer zweistufigen Sequenzierung 65

1 Einleitung

Diese Arbeit wurde am Institutsbereich Systeme der Elektronik des Zentralinstituts für Engineering, Elektronik und Analytik des Forschungszentrums Jülich angefertigt. Dies geschah in Kooperation mit dem JARA Institut für Quanteninformationen der Rheinisch-Westfälischen Technischen Hochschule Aachen. Im Folgenden werden diese beiden Institute vorgestellt. Daraufhin wird die Motivation dieser Arbeit erläutert.

1.1 Zentralinstitut für Engineering, Elektronik und Analytik

Das Zentralinstitut für Engineering, Elektronik und Analytik (ZEA) ist Teil der Forschungszentrum Jülich GmbH. Gemeinsam mit anderen Instituten des Forschungszentrums sowie externen Partnern werden am ZEA Experimente, Geräte, Prozesse, Mess-, Analyse- und Regelungsverfahren, Detektorsysteme und computergestützte Werkzeuge sowie bildgebende Verfahren entwickelt. Dabei kooperiert das Institut mit internen und externen Partnern bei technologischen Fragestellungen. Unter den externen Partnern befinden sich viele Universitäten und wissenschaftliche Einrichtungen aus der ganzen Welt. Der Fokus der Entwicklungen des Instituts liegt im technologischen und naturwissenschaftlichen Bereich. Das Institut setzt sich aus drei Institutsbereichen zusammen [1]:

1. Engineering und Technologie (ZEA-1)
2. Systeme der Elektronik (ZEA-2)
3. Analytik (ZEA-3)

Systeme der Elektronik (ZEA-2)

Das ZEA-2, wo auch diese Arbeit angefertigt wurde, entwickelt komplexe elektronische und informationstechnische Systeme für wissenschaftliche Zwecke. Die dort entwickelten Lösungen erstrecken sich von der Erfassung physikalischer Ereignisse über Detektoren

und Sensoren sowie Signal- und Datenverarbeitung bis hin zur Informationsextraktion und Visualisierung. Dabei wird teils auf bestehende Technologien zurückgegriffen und teils werden neue Technologien selbst entwickelt [2].

Unter anderem entwickelt das ZEA-2 auch integrierte Systemlösungen im Bereich Quantencomputing. Der Schwerpunkt liegt dabei auf der Entwicklung skalierbarer Kontroll- und Ausleseelektronik für Quantencomputer. Diese soll besonders rauscharm sein und mit geringer Leistungsaufnahme betrieben werden können. Außerdem muss die Elektronik auch bei niedrigen Temperaturen nahe dem absoluten Nullpunkt funktionieren [3]. Auch das in dieser Arbeit konzeptionierte Treibersystem wird für die Kontrollelektronik der Quantencomputer eingesetzt.

1.2 JARA-Institut für Quanteninformation

Als Teil der Jülich Aachen Research Alliance gehört das JARA-Institut für Quanteninformation sowohl zum zweiten physikalischen Institut der RWTH Aachen als auch zum Peter Grünberg Institut des Forschungszentrums Jülich. Das Institut für Quanteninformation arbeitet eng mit dem ZEA-2 zusammen und ist in vier Forschungsgruppen unterteilt, die sich mit sowohl mit der Theorie der Quanteninformation als auch mit experimenteller Quantentechnologie befassen [4].

Arbeitsgruppe Quantentechnologie

Eine dieser Gruppen ist die Arbeitsgruppe Quantentechnologie, die unter anderem an verschiedenen Experimenten mit *Elektronspin-Qubits* forscht [5]. Viele Grundlagen dieser Arbeit sind dort entwickelt worden und auch diese Arbeit wurde in Zusammenarbeit mit ebendieser Arbeitsgruppe angefertigt. Diverse Tests und Experimente zur Vorbereitung und Umsetzung dieser Arbeit wurden daher gemeinsam mit Mitarbeitern dieser Forschungsgruppe in den Laboren der RWTH Aachen durchgeführt.

1.3 Motivation

Ziel dieser Arbeit ist es, ein Konzept zur Abstraktion von Treibern für Arbiträrgeneratoren zu entwickeln. Bei Arbiträrgeneratoren handelt es sich um Instrumente, mit denen elektrische Spannungspulse erzeugt werden, die unter anderem verwendet werden, um die *Qubits* eines Quantencomputers zu kontrollieren und deren Zustände zu messen. Bei der

Konzeptionierung der neuen Treiber sollen die Eigenschaften der zurzeit von der Arbeitsgruppe Quantentechnologie verwendeten Geräte berücksichtigt werden. Das Konstrukt soll offen für Erweiterungen sein, jedoch darf der Mehraufwand, der durch ein hohes Maß an Flexibilität und Erweiterbarkeit erzeugt wird, nicht zu groß werden. Der Aufwand zur Umsetzung und Weiterentwicklung dieses Konzepts soll also in einem angemessenen Verhältnis zum Nutzen stehen. Dadurch soll erreicht werden, dass die neuen Treiber direkt oder mit geringem Aufwand in das bestehende Framework integrierbar sind. Außerdem sollen zukünftige Erweiterungen der Treiber von jedem Entwickler umsetzbar sein, weshalb das neue Treiberkonstrukt intuitiv und leicht verständlich sein soll.

Vor der Konzeptionierung des neuen Treiberkonstrukts waren die Treiber zum Teil sehr unflexibel und schwer erweiterbar. Es existierten zwar bereits mehrere Treiber für verschiedene Instrumente, aber diese waren nicht leicht austauschbar und es wurde nicht der volle Funktionsumfang der einzelnen Geräte genutzt. Es gab reichlich Optimierungspotential, wie beispielsweise bei der Konfiguration der Geräte oder der Speicherverwaltung auf der Hardware. Aus diesem Grund sind die primären Ziele bei der Konzeptionierung des neuen Treibersystems die Herstellung einer einheitlichen Schnittstelle zur Bedienung und Konfiguration der Treiber und eine Möglichkeit zur optimalen Ausnutzung der, von der Hardware gegebenen, Funktionalität.

2 Grundlagen

Zur Einordnung dieser Arbeit wird in diesem Kapitel zunächst eine Einführung in die Quantenmechanik erfolgen. Darauf aufbauend wird ein exemplarischer Versuchsaufbau gezeigt, in welchem ein Arbiträrgenerator zum Einsatz kommt, der über die hier thematisierten Treiber angesteuert wird. Im Anschluss wird das dabei zum Einsatz kommende Softwareframework beschrieben, das diese Treiber beinhaltet.

2.1 Quantenmechanik

> *„Denn wenn man nicht zunächst über die Quantentheorie entsetzt ist, kann man sie doch unmöglich verstanden haben.“*
>
> — Niels Bohr, 1969[1]

Wie *Niels Bohr* haben auch viele andere berühmte Physiker und Mathematiker ihre Probleme mit dem Verständnis der Quantenmechanik, darunter auch große Pioniere und Begründer der Quantentheorie, wie zum Beispiel *Richard Feynman*, *Albert Einstein* oder *Erwin Schrödinger*. Die Quantenmechanik scheint allem zu widersprechen, was aus der klassischen Physik bekannt ist, und vor allem scheint sie unvereinbar mit dem gesunden Menschenverstand zu sein. So geht es anfangs vielmehr darum, gewisse Gegebenheiten der Quantenmechanik zu akzeptieren und dabei nicht an Altbekanntem festzuhalten [7].

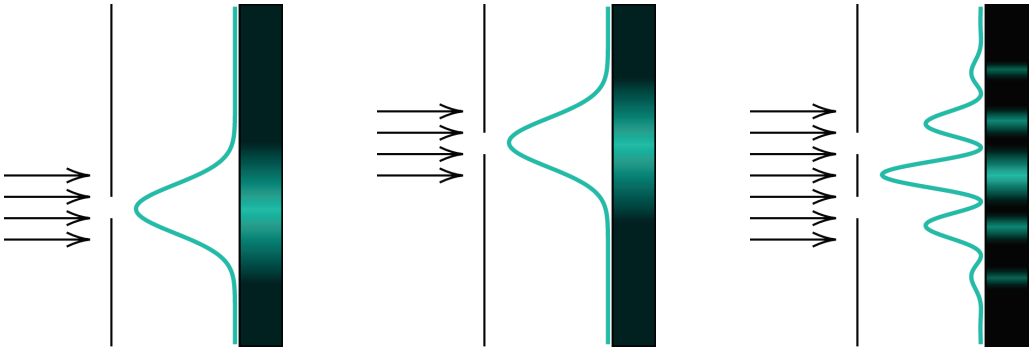
Welle-Teilchen-Dualismus

Ein wichtiges Phänomen der Quantenmechanik ist der Welle-Teilchen-Dualismus. In der klassischen Physik wurde zwischen Teilchen und Wellen strikt getrennt. Teilchen besitzen demnach eine Energie und einen Impulsvektor, wohingegen Wellen über eine Amplitude und einen Wellenvektor definiert werden. Doch in der quantenmechanischen Welt

[1] zitiert von *Werner Heisenberg* [6]

besitzen Wellen auch Teilcheneigenschaften und umgekehrt weisen Teilchen auch einen wellenartigen Charakter auf. Dieses Phänomen zeigt sich sehr anschaulich im Doppelspaltexperiment. Dabei wird zunächst Licht auf eine Wand mit zwei schmalen parallelen Spalten, dem Doppelspalt, geworfen. Dahinter befindet sich ein Schirm, auf dem das eintreffende Licht ein Interferenzmuster erzeugt. Das Erscheinen dieses Interferenzmusters liegt an den Welleneigenschaften von Licht, was bereits aus der klassischen Physik bekannt ist. Im Jahr 1923 stellte der Physiker *Louis de Broglie* jedoch eine Theorie vor, nach der auch Teilchen wellenartige Eigenschaften besitzen können. Dies konnte unter anderem bewiesen werden, indem das Doppelspaltexperiment mit einem Elektronenstrahl anstelle von Licht ausgeführt wurde. Nach der klassischen Physik wäre zu erwarten, dass ein Elektron durch einen der beiden Spalten hindurch fliegt und hinter dem jeweiligen Spalt auf dem Detektorschirm auftrifft [8].

In Abbildung 2.1 werden drei verschiedene Konfigurationen des Doppelspaltexperiments gezeigt. Dabei wird der Elektronenstrahl durch die Pfeile auf der linken Seite dargestellt. Die Elektronen treffen anschließend auf den Doppelspalt, fliegen durch diesen hindurch und landen schließlich auf dem Detektorschirm auf der rechten Seite. Der Detektorschirm misst, wie viele Elektronen an welcher Stelle des Schirms auftreffen. Dies wird in den Abbildungen durch die Färbung des Detektorschirms und durch das vertikale Histogramm zwischen dem Schirm und dem Doppelspalt dargestellt.



(a) Nur der untere Spalt ist geöffnet

(b) Nur der obere Spalt ist geöffnet

(c) Beide Spalte sind geöffnet

Abbildung 2.1: Doppelspaltexperiment mit einem Elektronenstrahl [8]

Das nach der klassischen Physik erwartete Verhalten für die Elektronen ist in den Abbildungen 2.1a und 2.1b zu beobachten, wo jedoch nur einer der beiden Spalte geöffnet ist. Dabei landen die Elektronen immer hinter dem jeweils geöffneten Spalt auf dem Schirm. Sind beide Spalte des Doppelspalts geöffnet, wäre nach der klassischen Physik mit einer Überlagerung dieser beiden Bilder zu rechnen. Stattdessen entsteht ein Interferenzmus-

ter, was in Abbildung 2.1c zu erkennen ist. Dies ist nach der klassischen Physik ist dies nur bei Wellen, wie zum Beispiel Licht, der Fall. Mit diesem Ergebnis konnte gezeigt werden, dass die Elektronen auch einen wellenartigen Charakter besitzen können. Demnach scheint es so, als verhielten sich die Elektronen manchmal wie klassische Teilchen und manchmal wie klassische Wellen. Stattdessen sind es jedoch quantenmechanische Teilchen mit quantenmechanischen Eigenschaften [8].

Superposition

Dass die Elektronen Interferenzmuster erzeugen können, liegt an der sogenannten *Superposition* – ein Grundprinzip der Quantenmechanik. Das *Superpositionsprinzip* besagt, dass sich Quantenteilchen in einer Überlagerung aus verschiedenen Zuständen befinden können. Das heißt in diesem Fall, dass die Elektronen beide Spalte des Doppelspalts gleichzeitig passieren können. Gemäß der klassischen Physik müsste sich ein Elektron hingegen für einen der beiden Spalte entscheiden, um den Schirm zu erreichen. Tatsächlich befindet sich das Elektron in einem Überlagerungszustand aus den beiden Möglichkeiten, wodurch es zu dem sichtbaren Interferenzmuster kommt. Dabei wird der Beobachter zu einem wichtigen Element des Experiments. Je nachdem, ob und wann ein Beobachter in einem quantenmechanischem Experiment „hinschaut“ oder eine Messung tätigt, können aufgrund der Superposition unterschiedliche Ergebnisse beobachtet werden [9].

Schrödingers Katze

Da die Entdeckung des Welle-Teilchen-Dualismus dem widerspricht, was die klassische Physik besagt, hatten viele Wissenschaftler Schwierigkeiten, die Welt der Quanten zu verstehen. So gab es viele Experimente, die Phänomene aufzeigten, die sich nicht widerspruchsfrei beschreiben ließen [8]. Im Jahr 1935 schlug *Erwin Schrödinger* ein Gedankenexperiment vor, welches das Superpositionsprinzip anschaulicher machte. Die Rede ist von *Schrödingers Katze*. Damit übertrug er die für die mikroskopische Quantenwelt geltenden Gesetze auf unsere makroskopische[2] Welt. Man stelle sich eine Katze in einer verschlossenen Kammer vor. Ebenfalls in der Kammer befindet sich ein Geigerzähler und eine geringe Menge einer radioaktiven Substanz. Dabei liegt die Wahrscheinlichkeit, dass eines der Atome dieser Substanz innerhalb einer Stunde zerfällt, bei 50 Prozent. Wenn dies eintritt, wird das Zerfallen des Atoms durch den Geigerzähler registriert. Dabei wird über einen Mechanismus ein Giftgas-Behälter mit einem Hammer zerschlagen, was den

[2] alle Objekte, die ein Mensch ohne besondere Hilfsmittel wahrnehmen kann, wie zum Beispiel materielle Gegenstände, Lebewesen oder kosmische Objekte.

sofortigen Tod der Katze nach sich ziehen würde. Da nicht in die Kammer hineingesehen werden kann, ist der Zustand der Atome nach einer Stunde nicht bekannt. Demnach befinden sich die radioaktiven Atome nach einer Stunde in einer Überlagerung aus den Zuständen „zerfallen“ und „nicht zerfallen“. Diese Superposition überträgt sich auf die Katze, da das Leben der Katze direkt vom Zustand der radioaktiven Substanz abhängt. Da der Beobachter nicht in die Kammer schauen kann, ist die Katze nach einer Stunde weder tot noch lebendig, sondern beides gleichzeitig. Erst beim Öffnen der Kammer, wenn also ein bewusster Beobachter in das Experiment eingreift, zwingt er das System in einen der beiden messbaren Zustände „Atom zerfallen, Katze tot“ oder „Atom nicht zerfallen, Katze lebt“. Dies wird auch *Kopenhagener Deutung* genannt, nach der ein bewusster Beobachter beziehungsweise eine Messung die Superposition aufhebt und das System in den gemessenen Zustand zwingt. [10, 11]

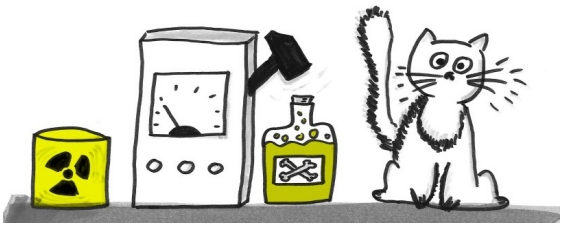


Abbildung 2.2: Illustration des Gedankenexperiments *Schrödingers Katze* [10]

Dekohärenz

Die *Kopenhagener Deutung* bringt jedoch ein Problem mit sich, denn diese verwendet den bewussten Beobachter als ausschlaggebenden Faktor. Jedoch ist das Bewusstsein keine messbare Größe und es existiert keine physikalische Definition für die Grenze zwischen „bewusst“ und „unbewusst“. Daher wurde die *Kopenhagener Deutung* mittlerweile durch die *Dekohärenz* abgelöst. Diese Theorie beschäftigt sich ebenfalls mit dem Übergang eines kohärenten Superpositionszustands in einen messbaren Zustand, der nicht mehr interferenzfähig ist. Nach der *Dekohärenz* ist dieser Kollaps der Überlagerungszustände jedoch nicht vom Bewusstsein eines Beobachters abhängig, sondern es handelt sich um einen dynamisch fortlaufenden Prozess. Ein quantenmechanisches System mit einer Messapparatur lässt sich nicht gänzlich von der Umgebung, wie zum Beispiel Luft, Licht oder Temperatur, isolieren. Somit findet die *Dekohärenz* kontinuierlich in Wechselwirkung mit der Umwelt statt. Wichtig dabei ist die *Dekohärenz-Zeit*, welche die Zeit bis zum Kollaps der Überlagerungszustände beschreibt. Es handelt sich also um die Zeitspanne, nach der ein quantenmechanisches System seine quantenmechanischen Eigenschaften verliert. Die Länge dieser Zeitspanne kann sich von System zu System stark

unterscheiden. Bei makroskopischen Systemen ist diese Zeit sehr kurz, weshalb bisher keine Superposition bei greifbaren Objekten beobachtet werden konnte. Mikroskopische Systeme können einen kohärenten Zustand hingegen für lange Zeit aufrechterhalten, denn die *Dekohärenz-Zeit* ist umgekehrt proportional zur Temperatur und Masse des Systems, wie Gleichung 2.1 zeigt [11]. Somit eignen sich sehr kleine und kalte Systeme am besten für quantenmechanische Versuche.

$$t_{Dekohärenz} \propto \frac{1}{T * m} \tag{2.1}$$

Verschränkung

Ein weiteres wichtiges Phänomen der Quantenmechanik ist die *Quantenverschränkung*, die einst von *Albert Einstein* als „spukhafte Fernwirkung“ bezeichnet wurde. Dieses Phänomen lässt sich sehr gut anhand eines Beispiels beschreiben: Angenommen zwei Personen haben jeweils eine Münze, die miteinander *verschränkt* seien. Dann fliegt der Eine mit seiner Münze beispielsweise auf den Mars, während der Andere mit seiner Münze auf der Erde verbleibt. Immer, wenn die Münzen geworfen werden und die eine Münze Kopf zeigt, zeigt die andere Münze Zahl, und umgekehrt. Dies geschieht im selben Moment und ohne Verzögerung – aufgrund der *Verschränkung* [7].

Genauso können auch die Eigenschaften quantenmechanischer Objekte miteinander *verschränkt* sein. Viele Science-Fiction-Autoren erhoffen sich dadurch eine verzögerungsfreie Kommunikation in Überlichtgeschwindigkeit. Dies ist jedoch nicht möglich, da die Münzen aus dem Beispiel keinen Informationsgehalt besitzen. Welche Seite oben liegt, ist zufällig und kann nicht beeinflusst werden. Somit wird es auch nicht möglich sein, verzögerungsfrei über weite Distanzen zu kommunizieren. Dennoch bietet die *Quantenverschränkung* andere Vorteile, die sich beispielsweise in einem Quantencomputer oder in der Quantenkryptographie nutzen lassen [7].

2.1.1 Quantencomputer

Während klassische Computer mit Bits rechnen, welche die Werte 0 oder 1 annehmen können, arbeiten Quantencomputer mit *Quantumbits* beziehungsweise *Qubits*. Dabei werden die Informationen nicht nur mit elektrischen Schaltkreisen kodiert, sondern mit Hilfe mikroskopischer Teilchen, wie zum Beispiel Atome, Elektronen oder Photonen. Für solche Teilchen gelten die Gesetze der Quantenmechanik. Dies führt zu einem der großen

Unterschiede zwischen herkömmlichen Computern und Quantencomputern: *Qubits* können nicht nur die Werte 0 oder 1 annehmen, sondern können sich auch in einer Superposition aus diesen beiden Werten befinden [12]. Der Zustand eines *Qubits* wird in der Ket-Schreibweise als „$|\psi\rangle$“ notiert. Dabei sind $|0\rangle$ und $|1\rangle$ die orthogonalen Basiszustände, aus denen der Gesamtzustand eines *Qubits* linear kombiniert werden kann. Dies ist in Gleichung 2.2 definiert [13].

$$|\psi\rangle = \alpha \cdot |0\rangle + \beta \cdot |1\rangle \quad \text{mit } \alpha, \beta \in \mathbb{C},\ |\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

Ein Zustand kann somit als eine Überlagerung der Basiszustände mit komplexen Vorfaktoren betrachtet werden. Dabei wird der Zustandsvektor auf eine Länge von 1 normiert, wodurch sich der Zustand mit einem komplexen Freiheitsgrad beschreiben lässt.

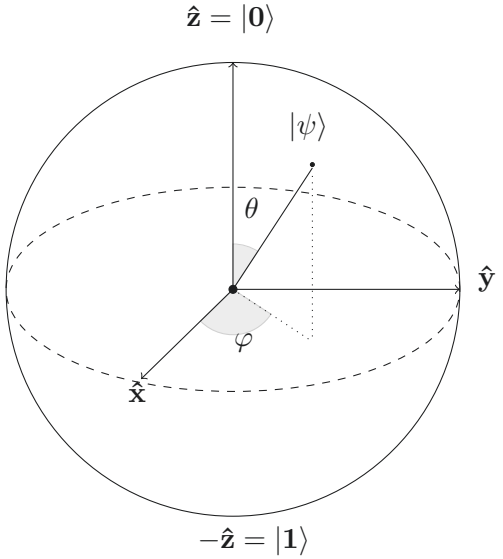


Abbildung 2.3: Bloch-Kugel [14]

Bei einem *Qubit* handelt es sich um ein Zweizustandssystem, wobei sich der Begriff „Zweizustandssystem“ nicht auf die Anzahl der möglichen Zustände bezieht, sondern die Anzahl der Basiszustände, aus denen sich ein Zustand zusammensetzen kann [15]. Ein solches System lässt sich grafisch auf einer sogenannten *Bloch-Kugel* darstellen, die in Abbildung 2.3 dargestellt ist. Der Zustandsvektor befindet sich in einem dreidimensionalen Raum und zeigt durch die Normierung der Länge auf die Oberfläche dieser Kugel mit dem Radius 1. Um den Zustandsvektor aus Gleichung 2.2 auf der *Bloch-Kugel* darzustellen, muss er zunächst in einen *Bloch-Vektor* konvertiert werden. Dieser besitzt die in Gleichung 2.3 beschriebene Form [14].

$$\begin{aligned} |\psi\rangle &= \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \quad \text{mit } \varphi, \theta \in \mathbb{R} \\ &= \cos\frac{\theta}{2}|0\rangle + (\cos\varphi + i\sin\varphi)\sin\frac{\theta}{2}|1\rangle \end{aligned} \tag{2.3}$$

Dadurch, dass der Zustandsvektor eines *Qubits* auf einen beliebigen Punkt der Oberfläche einer Kugel zeigen kann, übertrifft der Informationsgehalt eines einzelnen *Qubits* den eines klassischen Bits um ein Vielfaches. Der Zustand eines *Qubits* besitzt die beiden unabhängigen reellen Freiheitsgrade θ und φ. Der Informationsgehalt ist somit theoretisch unendlich groß. Da die überlagerten Zustände jedoch nicht messbar sind, wird der Informationsgehalt dadurch begrenzt. Bei einer Messung kann nur zwischen den Basiszuständen $|0\rangle$ und $|1\rangle$ unterschieden werden, wobei das Messergebnis zufällig ist. Die Wahrscheinlichkeit, welcher der beiden Basiszustände gemessen wird, hängt von dem quantenmechanischen Überlagerungszustand vor der Messung ab. Die Wahrscheinlichkeit, den Basiszustand $|0\rangle$ zu messen, liegt bei $|\alpha|^2$ und für $|1\rangle$ liegt sie bei $|\beta|^2$ [15].

Die Hauptschwierigkeit bei der Konstruktion und Benutzung eines Quantencomputers besteht in der zuvor erwähnten *Dekohärenz*, also dem Verlust der Interferenzfähigkeit. Durch das Kollabieren der Überlagerungszustände geht die Phaseninformation φ des *Qubits* – und somit ein Freiheitsgrad des Zustands – verloren. Da das System nicht gänzlich von seiner Umwelt isoliert werden kann, lässt sich die *Dekohärenz* nicht verhindern. Daher wird versucht, die *Dekohärenz-Zeit* möglichst auszudehnen, sodass das System einen *kohärenten* Zustand länger aufrechterhalten kann. Dies lässt sich erreichen, indem das System auf eine Temperatur von wenigen Kelvin heruntergekühlt wird, weil die *Dekohärenz-Zeit* umgekehrt proportional zur Masse und Temperatur des Systems ist. Als Nebeneffekt minimiert dies zusätzlich das thermische Rauschen, wodurch präzisere Messungen und Manipulationen der *Qubits* möglich sind [16].

Bei einem Quantencomputer wird nicht nur die Superpositionsfähigkeit der *Qubits* genutzt, sondern auch deren Fähigkeit zur *Verschränkung*. Dadurch können die *Qubits* nicht nur in einer Überlagerung mehrerer Zustände existieren und somit alle möglichen Werte gleichzeitig annehmen, sondern es können auch Operationen auf mehreren *verschränkten Qubits* gleichzeitig ausgeführt werden. Dies ermöglicht dem Quantencomputer eine einzigartige Möglichkeit zur Parallelisierung [17].

Durch die Ausnutzung quantenmechanischer Phänomene kann ein Quantencomputer gewisse Probleme effizienter als ein klassischer Computer lösen. Dazu gehören beispielsweise die Faktorisierung großer Zahlen und das Suchen in unsortierten Datenmengen [18].

2.2 Versuchsaufbau

Im Folgenden wird ein exemplarischer Versuchsaufbau für einen Quantencomputer beschrieben. Da verschiedene Möglichkeiten zur Realisierung eines Quantencomputers beziehungsweise eines *Qubits* existieren, liegt der Fokus in diesem Abschnitt auf den *Halbleiter Spinqubits*. Der hierbei verwendete Halbleiter ist Galliumarsenid (GaAs), worin bereits 2005 die kohärente Manipulation eines Qubits gezeigt werden konnte [19, 20].

Bei *Spinqubits* werden die Informationen im Elektronenspin gespeichert. Dabei handelt es sich um ein Zweizustandssystem, wodurch es sich hervorragend zur Verwendung als *Qubit* eignet [19]. Der Spin[3] ist eine Eigenschaft von Elektronen und kann analog zum mechanischen Drehimpuls verstanden werden [21]. Symbolhaft kann der Spin wie ein Kreisel gesehen werden. Dreht sich der Kreisel beispielsweise im Uhrzeigersinn, wird von *spin down* gesprochen; dreht er sich gegen den Uhrzeigersinn, wird dies *spin up* genannt. Daraus lassen sich direkt die Basiszustände $|\downarrow\rangle$ für *spin down* und $|\uparrow\rangle$ für *spin up* definieren. Dies ist jedoch nur eine Möglichkeit, *Spinqubits* umzusetzen. Im hier beschriebenen Versuch wird dies mit *Zwei-Elektronen-Spinqubits* beziehungsweise *Singlet-Triplet-Qubits* realisiert [19, 22]. Diese *Qubit*-Realisierungen ermöglichen eine vollkommen elektrische Steuerung der *Qubits*. Im Gegensatz zu anderen *Qubit*-Realisierungen kommt die Kontrolle der *Zwei-Elektronen-Spinqubits* ohne Mikrowellen aus, wodurch schnellere Gatter-Operationen möglich werden [23].

Bei einem *Zwei-Elektronen-Spinqubit* werden zwei Elektronen für ein *Qubit* verwendet. Dabei sind die Basiszustände $\uparrow\uparrow$, $\uparrow\downarrow$, $\downarrow\uparrow$ und $\downarrow\downarrow$ möglich. Diese lassen sich in die in Gleichung 2.4 beschriebene Form bringen. [24, 25]

$$\begin{aligned} |T_+\rangle &= |\uparrow\uparrow\rangle \\ |T_0\rangle &= \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle) \\ |S\rangle &= \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle) \\ |T_-\rangle &= |\downarrow\downarrow\rangle \end{aligned} \tag{2.4}$$

Der Gesamtspin m_s der Zustände $|T_0\rangle$ und $|S\rangle$ beträgt 0. Bei den Zuständen $|T_+\rangle$ und $|T_-\rangle$ ist der Gesamtspin hingegen $+1$ und -1. Die Zustände mit $m_s \neq 0$ lassen sich durch ein externes Magnetfeld herausfiltern, sodass nur noch die Zustände mit $m_s = 0$ angenommen werden können. Somit ergibt sich ein Zweizustandssystem, aus dem Singulett-Zustand $|S\rangle$ und dem Triplett-Zustand $|T_0\rangle$, die fortan als $|0\rangle$ und $|1\rangle$ definiert sind [25].

[3] engl. für Drall, Drehung

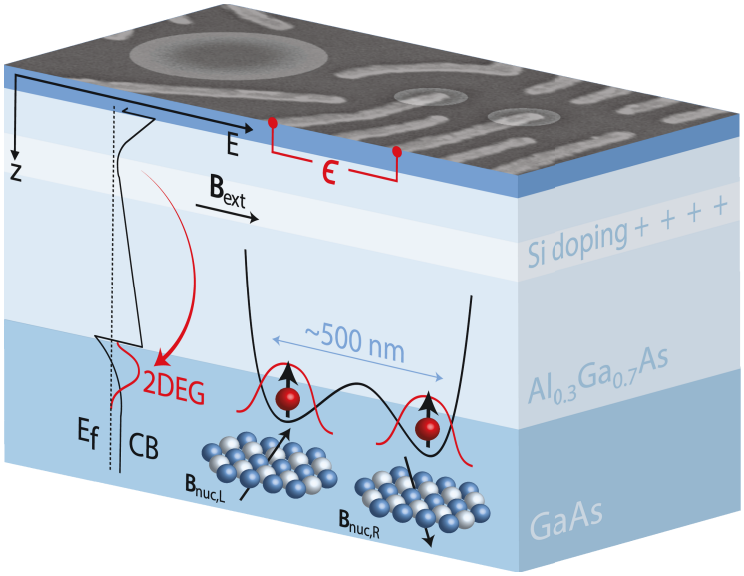


Abbildung 2.4: Elektronen-Spinqubits in GaAs [23]

Um ein *Qubit* technisch zu realisieren, müssen zwei Elektronen im GaAs isoliert werden. Durch das Anlegen einer Spannung über den Halbleiterblock, wird ein sogenannter Potentialtopf erzeugt, der zwei *Quantenpunkte*[4] erzeugt, in denen sich die Elektronen fangen lassen. Dies ist in Abbildung 2.4 zu erkennen. Die Abbildung zeigt den dreidimensionalen Aufbau einer *Halbleiter-Spinqubit*-Realisierung. In dem Galliumarsenid wird eine Schicht mit Elektronengas erzeugt. In diesem werden zwei Elektronen zur Realisierung des *Qubits* isoliert. Über die Gates auf der Oberseite des Blocks lassen sich an verschiedenen Stellen Spannungen anlegen, um die Elektronen zu manipulieren. Über die anderen Gates lässt sich unter anderem der Strom messen, um Erkenntnisse darüber zu erlangen, wie sich die Elektronen im Inneren des *Qubits* verhalten. Zur Kontrolle des Potenzials zwischen den beiden *Quantenpunkten* werden die beiden mit dem Parameter ϵ beschrifteten Gates verwendet. Zusätzlich müssen die *Qubits* auf unter 100 Millikelvin heruntergekühlt werden, um thermisches Rauschen zu minimieren und die *Dekohärenz-Zeit* zu optimieren. So bleiben die beiden Elektronen auch an der Position, wo sie benötigt werden, damit sie gemeinsam als *Qubits* verwenden werden können. Die Kühlung geschieht in mehreren Stufen. Dabei werden die Signale, die an das *Qubit* geleitet werden, an den Übergängen der Kühlebenen gedämpft und gefiltert, um unter anderem das thermische Rauschen zu mindern [26].

[4] engl. „quantum dots“

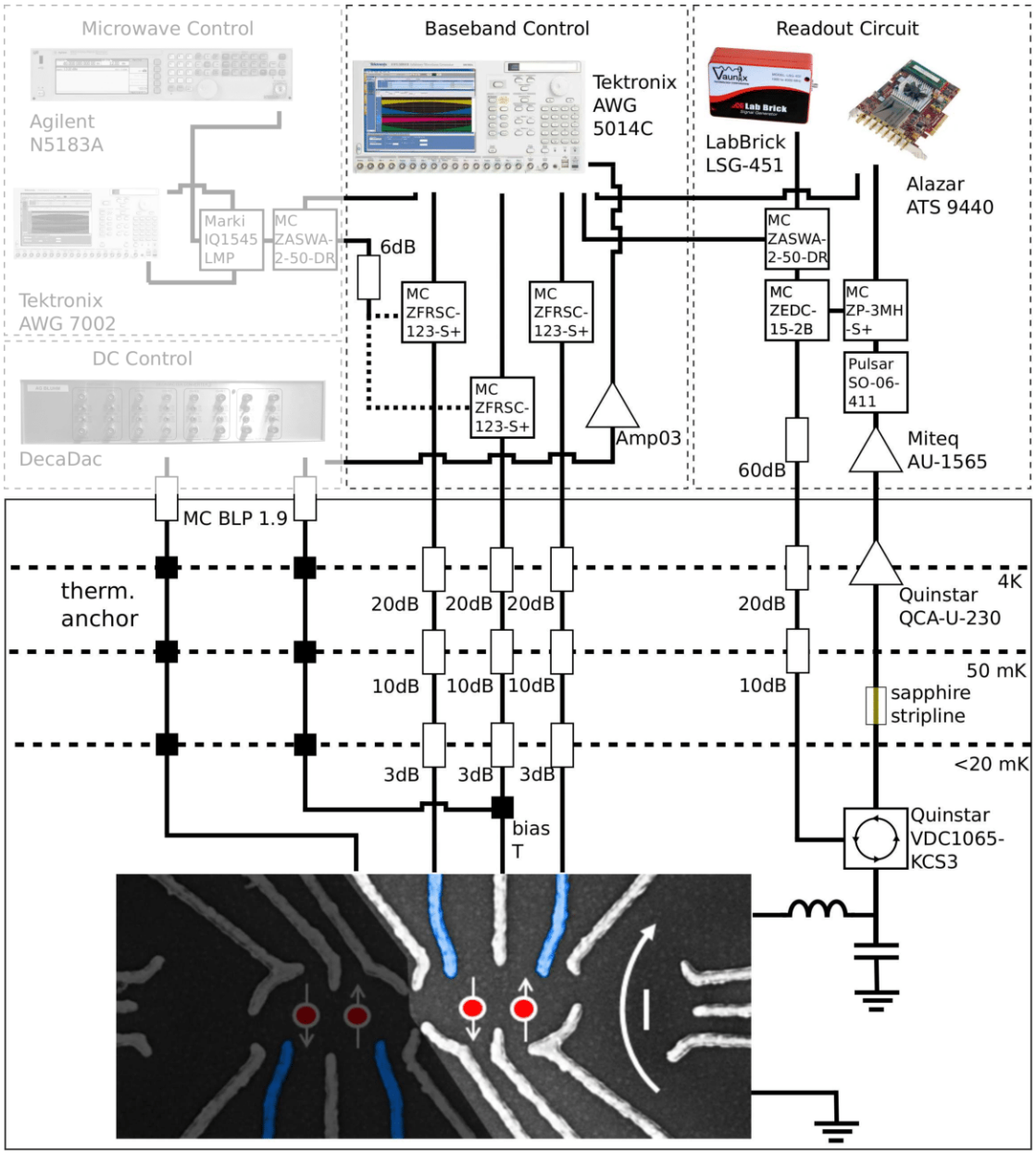


Abbildung 2.5: Schematischer Aufbau eines Quantencomputers [26]

In Abbildung 2.5 ist ein schematischer Aufbau eines Quantencomputers, bestehend aus zwei *Qubits*, zu sehen. Im oberen Bereich befinden sich die Mess- und Steuerinstrumente, die von gestrichelten Linien umrahmt sind. Im unteren von einer durchgezogenen Linie umrahmten Bereich ist der *Qubit*-Chip und die Kühlung zu sehen, wobei der unten dargestellte *Qubit*-Chip einer Draufsicht auf die Abbildung 2.4 entspricht. Die ausge-

grauten Bereiche *Microwave Control* und *DC Control* werden in dieser Arbeit nicht weiter beschrieben, denn für diese Arbeit ist vor allem der Bereich *Baseband Control* und zum Teil auch der *Readout Circuit* relevant, da diese für die Manipulation und Messung der *Qubits* zuständig sind. Im *Baseband Control* befindet sich der Arbiträrgenerator *Tektronix AWG5014C*, der auch im folgenden Kapitel, der Hardwareanalyse, genauer betrachtet wird. Arbiträrgeneratoren gehören dabei zu den digitalen Funktionsgeneratoren und können neben stetigen und fortlaufenden Standardsignalen auch arbiträre beziehungsweise diskrete Signale erzeugen. Hier wird der Arbiträrgenerator verwendet, um Spannungspulse zu erzeugen, welche die *Qubits* so manipulieren, dass diese Informationen speichern und Rechenoperationen ausführen können. Gleichzeitig werden im *Readout Circuit* Messungen aufgezeichnet, um die *Qubits* auszulesen und zu überwachen, wie sich diese verhalten haben. Die aufgezeichneten Daten werden von der Datenerfassungskarte *Alazar ATS9440* verarbeitet und können anschließend von den Benutzern ausgewertet werden. Die Signale der Mess- und Steuerinstrumente werden über die Verkabelung durch die einzelnen Kühlebenen bis hinunter zum *Qubit* geleitet, wo die Temperatur nur knapp über 0 Kelvin liegt. Die Kühlebenen sind in Abbildung 2.5 durch gestrichelte Linien voneinander getrennt und nutzen Temperaturen von 4 Kelvin, 50 Millikelvin und unter 20 Millikelvin. Die Mess- und Steuerinstrumente arbeiten bei Raumtemperatur [22, 25, 26].

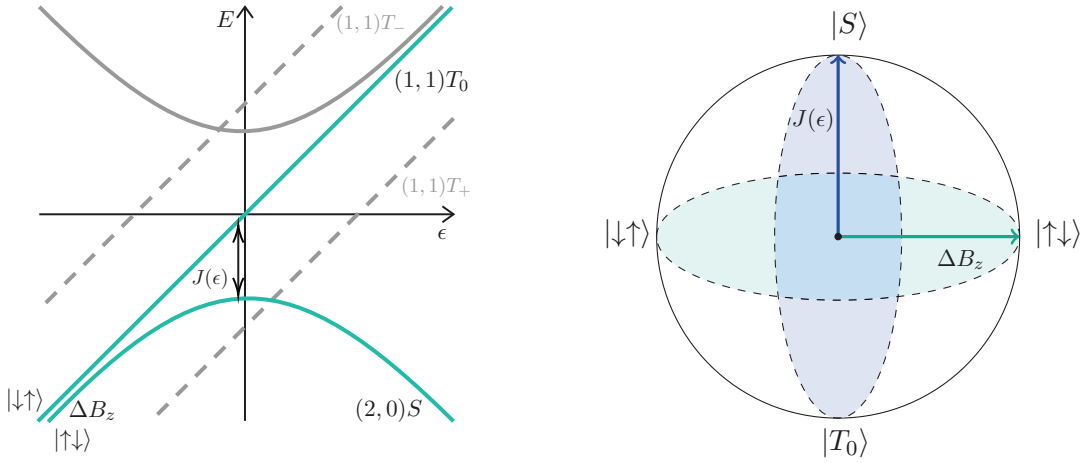
Durch verschiedene Spannungspulse, die auf die einzelnen Gates am *Qubit* angelegt werden, wurden anfangs Erfahrungen im Umgang mit den *Qubits* gesammelt, um die *Qubits* zu messen. Das mittelfristige Ziel der Experimente ist es jedoch, logische Gatter zu realisieren, um die *Qubits* in Zukunft für mathematische Rechenoperationen verwenden zu können, um daraus einen programmierbaren Prozessor gestalten zu können [19]. Des Weiteren gibt es Bestrebungen, einen sogenannten *Quantenbus* zu entwickeln, welcher ebenfalls über Arbiträrgeneratoren gesteuert wird und zur Datenübertragung innerhalb eines Quantenprozessors eingesetzt werden soll [27].

2.2.1 Messung eines Qubits

Um den Zustand eines *Qubits* zu messen, muss zwischen dem Singulett-Zustand S und dem Triplett-Zustand T_0 unterschieden werden können. Dazu wird das Potenzial zwischen den *Quantenpunkten* verändert. Dies führt dazu, dass die Elektronen entweder in getrennten *Quantenpunkten* bleiben oder eines in den anderen *Quantenpunkt* überspringt, sodass dieser beide Elektronen beherbergt. Um diesen Ladungszustand[5] darzustellen wird die Notation (n, m) verwendet, wobei n und m die Anzahl der Elektronen

[5] engl. „charge state“

im linken und rechten *Quantenpunkt* angeben. Möglich sind bei diesem Verfahren die Ladungszustände $(2,0)$ und $(1,1)$. Bei $(2,0)$ handelt es sich um den Singulett-Zustand S und bei $(1,1)$ um den Triplett-Zustand T_0 [26].



(a) Energiediagramm eines ST_0-Qubits

(b) Bloch-Kugel eines ST_0-Qubits

Abbildung 2.6: Messung des Zustands eines Singlet-Triplet-Qubits [26]

Zur Durchführung einer Messung wird ein Arbiträrgenerator benötigt, zum Beispiel der *Tektronix AWG5014C* aus dem schematischen Versuchsaufbau in Abbildung 2.5. Mit diesem wird ein Spannungspuls erzeugt, der zwischen zwei Gates auf dem *Qubit* angelegt wird. Diese Gates wurden in Abbildung 2.4 mit dem Parameter ϵ gekennzeichnet. Durch die Manipulation dieses Parameters ändert sich das Potenzial zwischen den beiden *Quantenpunkten* und der *Qubit*-Zustand lässt sich anhand der Energie messen. Dadurch kann der Ladungszustand des *Qubits* gemäß Abbildung 2.6 bestimmt werden. Befinden sich die Elektronen in demselben *Quantenpunkt*, ergibt sich der Ladungszustand $(2,0)$ und somit ist das *Qubit* im Singulett-Zustand S. Sind die Elektronen getrennt, also im Ladungszustand $(1,1)$, wurde der Triplett-Zustand T_0 gemessen [26].

2.3 Software zur Pulserzeugung

Zur Erzeugung der Spannungspulse mit einem Arbiträrgenerator und zur Auswertung der Messergebnisse wird ein Open Source Software Paket namens *qupulse* verwendet, welches in Python 3 implementiert wurde. Python 3 bietet sich als Programmiersprache an, da es sich um eine interpretierte Programmiersprache handelt, bei denen die Übersetzung eines Skripts zur Laufzeit stattfindet und kein Compiler notwendig ist. Dies

ist besonders in experimentellen Umgebungen von Vorteil. Außerdem bietet Python eine geringe Einstiegsbarriere und ist somit weit verbreitet [28]. Das *qupulse*-Framework ermöglicht die Erzeugung, Verwaltung, Speicherung und Verbreitung von Pulssequenzen. Es können ganze Messungen erstellt werden, in welchen die *Qubits* initialisiert, kontrolliert und ausgelesen werden. Die dazu benötigten Pulssequenzen sind wiederverwendbar und lassen sich leicht zwischen Mitarbeitern austauschen. Außerdem beinhaltet *qupulse* Treiber, welche die Möglichkeit bieten, die Pulse auf die Arbiträrgeneratoren hochzuladen. Neben den Treibern für die Arbiträrgeneratoren sind auch Treiber für die Datenerfassungsgeräte enthalten, sodass neben dem *Baseband Control* auch der *Readout Circuit* aus Abbildung 2.5 im Framework realisiert wird. Der Programmcode, Beispiele und die Dokumentation des *qupulse*-Projekts sind auf GitHub[6] verfügbar [25].

2.3.1 Geschichte

Die Entwicklung an *qupulse*, damals noch *qc-toolkit*, begann im Jahr 2015 am physikalischen Institut für Quanteninformation der RWTH Aachen und in Zusammenarbeit mit der Fakultät für Softwareentwicklung der RWTH. Das Ziel dabei war es, einen Standard zur Erzeugung von Pulssequenzen im Bereich der Quanteninformation zu schaffen. Die Entwicklung wurde schließlich im August 2016 abgeschlossen, woraufhin Simon Humpohl, ein Doktorand des Instituts für Quanteninformation, die Weiterentwicklung des Frameworks im Rahmen seiner Masterarbeit[7] aufgenommen hat. Dabei definierte er mehrere Ziele für *qupulse* und setzte diese um. Auch nach Humpohls Masterarbeit wurde *qupulse* inkrementell weiterentwickelt [25]. Es wurden Treiber für neue Geräte implementiert, diverse strukturelle Änderungen wurden vorgenommen und die Testabdeckung[8] wurde stetig gesteigert [29].

2.3.2 Ziele des Projekts

Die ursprünglichen Ziele bei der Entwicklung von *qupulse* waren die Folgenden [25]:

1. Leistungsfähige Software in Bezug auf die Komplexität von Pulsen:
   - Unterstützung von mehrkanaligen Pulsen (*Multi channel pulses*) für Geräte mit mehreren Ausgabekanälen.
   - Pulssequenzen mit bedingten Verzweigungen.

[6] `https://github.com/qutech/qupulse` [29]
[7] „Hardware Adapted Pulses and Software for Qubit Control“ [25]
[8] engl. „code coverage“ oder „test coverage“

2. Austausch von Pulssequenzen mit Kollegen, die andere Versuchsaufbauten verwenden.
3. Leichte Verständlichkeit des Front-Ends, auch mit wenig Programmiererfahrung.
4. Sorgfältig dokumentierter Quellcode
   - Front-End: Zum besseren Verständnis durch die Benutzer.
   - Back-End: Zur Verbesserung der Wartbarkeit, Erweiterbarkeit und Transparenz, sowie zur leichteren Einarbeitung neuer Entwickler.
5. Flexibles Back-End, welches viele verschiedene Geräte unterstützt.

Diese Ziele wurden größtenteils umgesetzt, jedoch gibt es noch einen gewissen Optimierungsbedarf. Diese Arbeit beschäftigt sich daher besonders mit Punkt 5, dem flexiblen Back-End. Besonders in Bezug auf die Erweiterbarkeit und Wartbarkeit ist die Flexibilität der Treiber eingeschränkt und kann weiter optimiert werden. Aber auch die nicht-einheitliche Umsetzung der verschiedenen Gerätetreiber macht das Framework weniger flexibel. Soll zum Beispiel in einem Experiment ein anderer Arbiträrgenerator verwendet werden als bisher, muss auch ein Großteil der dazu geschriebenen Programmskripte angepasst werden, um mit dem Treiber des neuen Geräts kompatibel zu sein. Außerdem ist die Konfiguration der Geräte nicht trivial, was gegen Punkt 3, der leichten Verständlichkeit des Front-Ends, verstößt. Auch dies ist unter anderem durch die nicht-einheitliche Realisierung der Treiber bedingt.

Das einfache Hochladen und Ausführen einer Pulssequenz ist bereits sehr flexibel. Es müssen nahezu keine Anpassungen bezüglich der Pulssequenz und dessen Ausführung getätigt werden, wenn ein anderes Gerät verwendet werden soll. Jedoch müssen die Instrumente vor der Ausführung einer Pulssequenz zunächst konfiguriert werden. Zum Beispiel muss eine Verbindung zum Gerät aufgebaut werden und die benötigten Kanäle müssen definiert und zugeordnet werden. Dazu müssen heute noch pro Gerät unterschiedliche Befehle ausgeführt werden, da jeder Hardwaretreiber bisher eine eigene Logik verwendet und nur gewisse Funktionen im Hintergrund vereinheitlicht wurden. Ziel ist es jedoch, dass ein Gerät mit dem kleinstmöglichen anschließenden Programmieraufwand ausgetauscht werden kann. Zudem soll es möglich sein, die Pulssequenzen so zu optimieren, dass sie möglichst wenig Speicher auf dem Instrument benötigen. Dadurch ließen sich deutlich mehr oder längere Pulssequenzen auf ein Gerät hochladen [25, 29].

2.3.3 Aufbau

Um mit *qupulse* eine Pulsvorlage zu erstellen und mit dieser eine Messung zu starten, gibt es einen festen Ablauf, der in Abbildung 2.7 zu erkennen ist. Zunächst beginnt der Benutzer damit, eine Pulsvorlage vom Typ `PulseTemplate` aufzubauen. Mit diesen lässt sich ein parametrisierter Spannungsverlauf definieren, welcher beliebig komplex sein kann, sofern genug Speicher vorhanden ist. Die Pulsvorlage kann anschließend unter Angabe der konkreten Parameterwerte in ein Programm vom Typ `Loop` konvertiert werden, wobei die Pulsvorlage auf die für die Hardware relevanten Informationen reduziert wird.

Um das `Loop`-Programm auf einen Arbiträrgenerator hochzuladen, wird der richtige Treiber benötigt. Die Treiber der Arbiträrgeneratoren[9] werden zurzeit durch die abstrakte Klasse `AWG` zusammengefasst. Diese Treiber haben den Zweck, das vorbereitete Programm für die Hardware aufzubereiten und auf das Gerät zu übertragen, sodass dieses die erstellte Pulssequenz anschließend abspielen kann. Dabei werden gleichzeitig die Messfenster, also Zeiträume, in denen Messungen stattfinden, aus dem Programm extrahiert, um diese an die Datenerfassungsgeräte zu übermitteln. Bei diesen handelt es sich in der Regel um PCIe-Karten mit direktem Zugriff auf den Arbeitsspeicher, um die großen Datenmengen schnell genug speichern zu können, ohne vom Betriebssystem ausgebremst zu werden. Die Geräte zeichnen auf, wie die Qubits während der Messzeiträume auf die Pulssequenzen reagieren. Die Treiber der Datenerfassungskarten[10] werden durch die abstrakte Klasse `DAC` zusammengefasst. In einem synchronisierten System aus Arbiträrgeneratoren und Datenerfassungskarten erlaubt die Verwendung von Messfenstern die Verknüpfung zwischen den Pulssequenzen und den daraus resultierenden Messwerten [29].

Der wichtigste Teil von *qupulse* für diese Arbeit sind die AWG-Treiber. Diese gilt es zu abstrahieren und zu optimieren. In Abbildung 2.7 ist der dafür relevante Bereich durch das gestrichelte Rechteck hervorgehoben.

Pulsvorlagen

Das Kernkonzept von *qupulse* sind die bereits erwähnten Pulsvorlagen, welche eine parametrisierbare Beschreibung einer Pulssequenz angeben, die von einem Arbiträrgenerator abgespielt werden soll. Sie bestehen aus Sequenzen und Wiederholungen von Spannungsniveaus und Laufzeiten. Daraus ergeben sich für das Zielgerät arbiträre beziehungsweise

[9] engl. „arbitrary waveform generator“ (kurz AWG)
[10] engl. „data acquisition card“ (kurz DAC)

diskrete Spannungssequenzen, die auf festgelegten Kanälen abgespielt werden sollen. Die Pulsvorlagen sind so gestaltet, dass es für den Benutzer möglichst einfach ist, auch komplizierte Pulssequenzen zu definieren. Dafür sollen keine tiefen Kenntnisse der Programmierung notwendig sein [30].

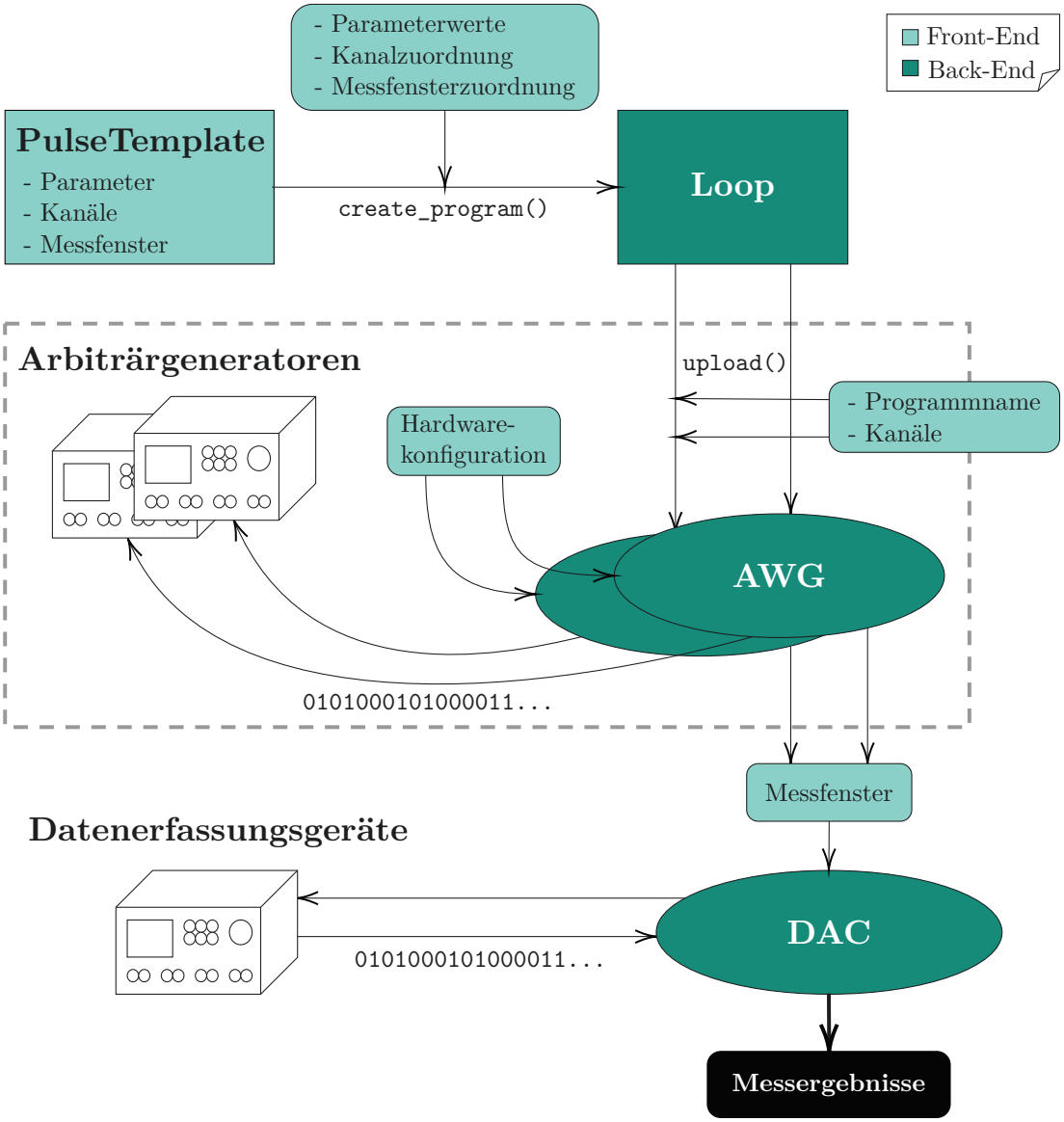


Abbildung 2.7: Ablauf einer Messung mit *qupulse*

Die Parameter der Pulsvorlagen müssen erst kurz vor dem Kompilieren und Hochladen auf den Arbiträrgenerator vom Benutzer aufgelöst werden. Neben den Spannungsniveaus können auch Messfenster in den Pulsvorlagen definiert werden. Diese sind zwar nicht für

die Arbiträrgeneratoren relevant, aber für die Datenerfassungsgeräte. Die in den Pulsvorlagen definierten Messfenster werden von *qupulse* beim Kompiliervorgang in feste, relative Zeiträume umgewandelt, die anschließend an die Datenerfassungsgeräte übermittelt werden können. Dadurch wird sichergestellt, dass die Messgeräte die Messdaten im richtigen Zeitraum aufzeichnen [25].

Die Pulsvorlagen werden durch die abstrakte Klasse `PulseTemplate` dargestellt, für welche verschiedene Spezialisierungen existieren. Diese können auf unterschiedliche Arten parametrisierbare Spannungssequenzen beschreiben. Die verschiedenen Klassen, die von der `PulseTemplate`-Klasse abgeleitet sind, lassen sich dabei in atomare (`AtomicPulseTemplate`) und nicht-atomare Pulsvorlagen einteilen. Zu den atomaren Pulsvorlagen gehören die Klassen `TablePulseTemplate`, `PointPulseTemplate`, `FunctionPulseTemplate` und `AtomicMultiChannelPulseTemplate`, da diese Objekte die Blätter, also die kleinste Einheit, in einem verschachtelten Baum an Pulsvorlagen darstellen. Alle weiteren Pulsvorlagen sind somit nicht atomar, weil diese noch weitere Pulsvorlagen beinhalten, wodurch eine verschachtelte Struktur entsteht. Im Folgenden werden die wichtigsten Pulsvorlagen kurz vorgestellt [30]:

TablePulseTemplate: Das `TablePulseTemplate` ermöglicht es dem Benutzer, Spannungspulse in Form von Wertepaaren aus Zeit und Spannung zu definieren. Außerdem kann eine Interpolationsstrategie angegeben werden, die beschreibt, wie die Spannung von einem zum nächsten Zeitpunkt verändert wird. Dabei existieren die Möglichkeiten `hold`, `jump` und `linear`. Ein `TablePulseTemplate` kann außerdem über mehrere Kanäle definiert werden, wobei für jeden Kanal eine unabhängige Pulssequenz definiert werden kann.

PointPulseTemplate: Das `PointPulseTemplate` funktioniert bei der Verwendung eines Kanals genauso wie das `TablePulseTemplate`. Bei Mehrkanal-Pulssequenzen können die Kanäle jedoch nicht gänzlich unabhängig voneinander agieren. Die Spannungsniveaus lassen sich pro Kanal individuell einstellen. Die Zeitpunkte, an denen ein Spannungswert gesetzt wird, sind jedoch für alle Kanäle einheitlich.

FunctionPulseTemplate: Beim `FunctionPulseTemplate` lässt sich eine beliebige mathematische Funktion in Textform angeben, die eine Abbildung der Spannung über die Zeit darstellt. Vor dem Hochladen wird diese Funktion abgetastet beziehungsweise diskretisiert, um ein arbiträres Signal mit der Abtastrate[11] des Geräts zu erhalten. Ein

[11] engl. „sample rate“

`FunctionPulseTemplate` unterstützt nur einen Kanal. Zur Definition eines mehrkanaligen Pulses, der durch mathematische Funktionen beschrieben wird, können mehrere `FunctionPulseTemplates`, sowie auch andere atomare Pulsvorlagen, in einem `AtomicMultiChannelPulseTemplate` zusammengefasst werden.

Anschließend folgen einige nicht-atomare Pulsvorlagen, die die Eigenschaft besitzen, dass diese eine oder mehrere Pulsvorlagen eines beliebigen Typs beinhalten.

SequencePulseTemplate: Mit dem `SequencePulseTemplate` lassen sich mehrere Pulsvorlagen aneinander ketten. Vom Arbiträrgenerator werden diese direkt hintereinander in Form einer fortlaufenden Pulssequenz abgespielt.

RepetitionPulseTemplate: Mit dieser Pulsvorlage lässt sich die eingebettete Pulsvorlage beliebig oft wiederholt. Die Anzahl der Wiederholungen lässt sich konstant oder als Parameter angeben.

ForLoopPulseTemplate: Das `ForLoopPulseTemplate` funktioniert analog zur `for`-Schleife, wie aus in vielen Programmiersprachen bekannt. Es wird ein Parameter deklariert, über welchen iteriert werden soll. Für diesen wird eine Liste an Werten angegeben, die der Parameter nacheinander annehmen soll.

MappingPulseTemplate: Wie bereits zuvor erwähnt, sind alle Pulsvorlagen parametrisierbar. Mit dem `MappingPulseTemplate` lassen sich einzelne Parameter auf andere Parameter oder mathematische Ausdrücke abbilden. Dadurch sind die Pulsvorlagen flexibler einsetzbar, da auf diese Weise auch Pulsvorlagen aus anderen Versuchsaufbauten integriert werden können, ohne dass Konflikte zwischen den verschiedenen Parametern auftreten.

Alle verschiedenen Pulsvorlagen sind serialisierbar. Dies bedeutet, dass sie sich beispielsweise in Dateien speichern lassen. Dabei kann die Wurzel-Pulsvorlage mit allen verschachtelt enthaltenen Pulsvorlagen in eine Datei gespeichert werden. Alternativ ist es möglich, einzelne Pulsvorlagen separat zu speichern und mit einem eindeutigen Namen auf diese zu verweisen. Hierdurch wird Speicherplatz gespart und Änderungen an zentralen Pulsvorlagen werden automatisch in den übergeordneten Pulsvorlagen wirksam. Dies erleichtert zudem den Austausch von Pulsvorlagen zwischen Mitarbeitern [29].

Programme und Waveforms

In den vorherigen Kapiteln wurde hauptsächlich das Front-End beschrieben, wobei das Front-End in diesem Fall keine Benutzeroberfläche ist, sondern eine Programmierschnittstelle. Um die von den Benutzern definierten Pulsvorlagen auf die Hardware zu transferieren, müssen diese zunächst im Back-End für die Geräte aufbereitet werden. Das heißt, dass alle Daten, die für die Hardware irrelevant sind, entfernt werden, um diese für die Hardware zu vereinfachen. Dabei werden beispielsweise die Parameter aufgelöst und es werden gewisse Änderungen an der verschachtelten Struktur der Pulsvorlagen vorgenommen. Dies geschieht in einem Zwischenschritt über die Programme und Waveforms, noch bevor die Pulssequenzen auf die Instrumente geladen werden.

Um die Pulsvorlagen in ein allgemeines Programm zu konvertieren, müssen zunächst alle Parameter aufgelöst werden, da die Programme auf der Hardware momentan nicht parametrisierbar sind. Außerdem besteht die Möglichkeit, die definierten Kanäle und Messfenster neu zuzuordnen, sodass die Pulsvorlagen nicht angepasst werden müssen, um in unterschiedlichen Experimenten verwendet zu werden. Unter Auflösung der Parameter werden die Pulsvorlagen in Objekte der `Waveform`-Klasse umgewandelt. Diese Klasse ist, wie die `PulseTemplate`-Klasse, abstrakt und weist eine ähnliche Vererbungshierarchie auf. So gibt es beispielsweise auch die Klassen `TableWaveform`, `FunctionWaveform` und `SequenceWaveform`. Die Waveforms sind also die direkten Gegenstücke zu den Pulsvorlagen. Der Hauptunterschied besteht darin, dass diese keine Parameter mehr beinhalten. Außerdem besitzen die Waveforms eine Methode, mit der die Waveforms pro Kanal gesampelt werden können. Dies bedeutet, dass die Waveforms mit einer festgelegten Rate abgetastet werden können, um daraus einen diskreten Spannungsverlauf zu erhalten. Dies wird im folgenden Abschnitt genauer beschrieben [25, 29].

Die auf diese Weise erhaltenen Waveforms werden anschließend in `Loop`-Objekte gepackt. Bei den `Loops` bleibt die verschachtelte Struktur beziehungsweise Hierarchie der Pulsvorlagen erhalten. Dadurch werden die Schleifen, Wiederholungen und Sequenzen abgebildet. Das Gesamtkonstrukt, begonnen bei dem Wurzel-`Loop`-Objekt, wird auch als Programm bezeichnet. Ein solches Programm ist eine vereinfachte Darstellung einer verschachtelten Pulsvorlage. Es ist für das Hochladen auf die Hardware vorbereitet und muss nur noch an die hardwarespezifischen Bedingungen angepasst werden, die sich von Instrument zu Instrument unterscheiden können [29].

Hardwaretreiber

Der letzte Schritt für die Pulssequenzen in *qupulse* ist das Hochladen auf die Hardware. Dies geschieht im jeweiligen `AWG`-Hardwaretreiber. Dazu erhält der Hardwaretreiber das zuvor erstellte Programm. Außerdem benötigt der Treiber Informationen über die Kanäle, auf denen die Pulssequenzen abgespielt werden sollen. Anschließend beginnt der Treiber damit, das Programm kompatibel zur Hardware zu machen. Dazu müssen gewisse durch die Hardware gegebene Restriktionen eingehalten werden, was die Länge, Auflösung und Verschachtelungstiefe des Programms betrifft. Daraufhin wird das Programm mit der momentanen Abtastfrequenz des Instruments abgetastet, um die diskreten Spannungsniveaus pro Abtastpunkt herauszubekommen und die für das Instrument optimale zeitliche Auflösung zu erreichen. Gleichzeitig werden die Zeiträume der Messfenster aus dem Programm extrahiert, die später für die Konfiguration der Messungen benötigt werden. Daraufhin wird das gesampelte Programm auf den Arbiträrgenerator hochgeladen und kann anschließend abgespielt werden. Dies geschieht in der Praxis jedoch nicht sofort. Stattdessen wird das Programm auf der Hardware scharf geschaltet, woraufhin das Gerät auf ein Triggersignal wartet. Dadurch wird sichergestellt, dass alle mit dem Trigger verbundenen Instrumente synchron starten. Dies funktioniert je nach Hardware bis auf wenige Nanosekunden genau [29].

Die zuvor extrahierten Messfenster werden dem Benutzer zurückgegeben, sodass er die Möglichkeit hat, die Messungen auf einem Datenerfassungsgerät zu definieren. Dies geschieht über die Treiber vom Typ `DAC`. Sobald das Datenerfassungsgerät die Messfenster gespeichert hat, kann auch dieses scharf geschaltet werden. Anschließend wartet auch das Messgerät auf das Triggersignal und beim Auslösen des Triggers werden alle Instrumente gleichzeitig die Programme und Messungen starten [29].

3 Hardwareanalyse

In diesem Kapitel werden die momentan von der Arbeitsgruppe Quantentechnologie verwendeten Arbiträrgeneratoren vorgestellt. Die drei vorgestellten Geräte werden hinsichtlich ihrer Eigenschaften und Funktionen analysiert und verglichen, um anschließend in der Konzeptionierung besonders auf die Kernfunktionen der betrachteten Geräte eingehen zu können.

3.1 Arbiträrgeneratoren

Ein Arbiträrgenerator ist ein digitaler Funktionsgenerator, der beliebig geformte Spannungssignale erzeugen kann. Wie auch einfache analoge Funktionsgeneratoren, unterstützt ein Arbiträrgenerator Standardsignale, wie zum Beispiel Sinus-, Dreieck- oder Rechtecksignale. Zusätzlich können arbiträre beziehungsweise diskrete Signale generiert werden, die frei vom Benutzer definiert werden können. Dazu lässt sich ein Arbiträrgenerator meist über verschiedene Schnittstellen mit einem Computer verbinden, worüber das Gerät programmiert werden kann. Die am Computer definierten Signale werden im Speicher des Geräts abgelegt und können von dort aus abgespielt werden. Dabei werden die digitalen Signale von einem Digital-Analog-Wandler in analoge Signale umgesetzt und verstärkt, um das Ausgangssignal zu erzeugen [31].

Die wichtigsten Eigenschaften in Bezug auf die Leistungsfähigkeit eines Arbiträrgenerators sind die Abtastrate, die Wortbreite und die Speicherkapazität. Die Abtastrate gibt an, mit welcher Frequenz das Ausgangssignal ausgegeben werden kann, und die Wortbreite gibt die maximale vertikale Auflösung, also die Genauigkeit, eines Signals in Bits an [31].

Die zurzeit von der Arbeitsgruppe Quantentechnologie verwendeten Arbiträrgeneratoren sind die Geräte *Tabor WX2184C*, *Tektronix AWG5014C* und *Zurich Instruments HDAWG8*. Nachfolgend werden diese vorgestellt und hinsichtlich ihrer Spezifikationen und ihres Funktionsumfangs analysiert.

3.1.1 Tabor WX2184C

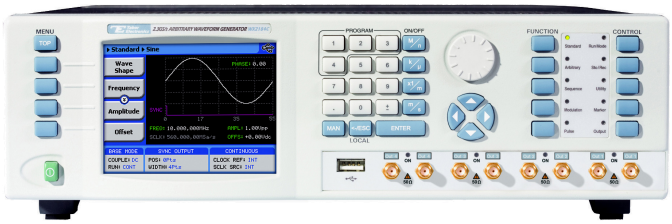


Abbildung 3.1: Tabor WX2184C [32]

Der in Abbildung 3.1 dargestellte *WX2184C* von *Tabor Electronics* ist ein Hochgeschwindigkeits-Arbiträrgenerator und das Top-Modell aus der *WaveXciter*-Baureihe. Diese umfasst außerdem die Modelle *WX1281C*, *WX1282C*, *WX1284C*, *WX2181C* und *WX2182C*. Der *Tabor WX2184C* grenzt sich durch eine höhere Abtastrate und mehr Kanäle von den anderen Modellen der Baureihe ab [33]. Außerdem bietet das Gerät eine hohe Signalqualität mit sehr geringem Phasenrauschen, wodurch die *WaveXciter*-Serie nach Aussage des Herstellers zu den qualitativ hochwertigsten Geräten auf dem Markt zu zählen ist [32].

Der *Tabor WX2184C* besitzt vier Ausgabekanäle. Jeder Ausgabekanal besteht aus zwei Differenzausgängen, zwischen denen die gewählte Spannung angelegt wird, und einem Marker-Kanal. Dieser kann als Trigger für andere Instrumente eingesetzt werden, um das Gesamtsystem zu synchronisieren. Zur Taktung hat das Instrument zwei Taktgeber verbaut. Diese können als gemeinsamer Taktgeber oder separat genutzt werden. Mit einem gemeinsamen Taktsignal können alle vier Kanäle mit einem Zeitversatz von höchstens 10 Pikosekunden[12] synchronisiert werden. Alternativ kann im separaten Modus gearbeitet werden, wobei die Kanäle 1 und 2 sowie die Kanäle 3 und 4 jeweils paarweise synchronisiert werden. Dabei können die Kanalpaare wie getrennte unabhängige Geräte genutzt werden. Wenn mehr als vier Kanäle benötigt werden, ist es zusätzlich möglich, das Gerät mit einem weiteren Instrument aus der WX-Serie zu koppeln, um insgesamt bis zu acht synchrone Kanäle nutzen zu können [32].

Bei arbiträren Signalen ist eine Abtastrate von bis zu 2,3 Gigasample pro Sekunde möglich, was einer zeitlichen Auflösung von unter 500 Pikosekunden entspricht. Die minimale Abtastrate liegt bei 75 Megasample pro Sekunde. Die Wortbreite der Waveforms beträgt 14 Bits, was einer relativen Genauigkeit von 1/16384 entspricht. Bei einer maximalen Schwingungsbreite von 4 Volt-peak-to-peak[13] zwischen den Differenzausgän-

[12] eine Pikosekunde (ps) entspricht 10^{-12} Sekunden
[13] Spitze-Tal-Wert der Wechselspannung

gen, entspricht dies einer absoluten Genauigkeit von 0,25 Millivolt. Wird eine geringe Spannungsbreite gewählt, verbessert sich die absolute Genauigkeit entsprechend. Wird anstelle der Differenzausgänge ein asymmetrischer „single-ended“ Ausgang verwendet, halbiert sich die maximale Amplitude. Die Kapazität des Waveformspeichers beträgt standardmäßig 16 Megasample pro Kanal, also 16 Millionen Abtastpunkte. Optional ist das Gerät auch mit 32 Megasample erhältlich. Der gesamte Speicher kann für nur eine einzelne Waveform genutzt werden, kann aber auch in bis zu 32000 Segmente eingeteilt werden. [32, 34]

Zuvor wurde bereits erwähnt, dass je nach Hardware gewisse Restriktionen bei den Waveforms eingehalten werden müssen. Für den *Tabor WX2184C* muss jedes Segment hardwarebedingt eine Mindestlänge von 192 Abtastpunkten und eine Auflösung von 16 Abtastpunkten aufweisen. Die Gesamtlänge muss demnach ein Vielfaches von 16 Abtastpunkten sein, mindestens jedoch 192. Diese Einschränkungen betreffen die Segmentlänge und bewirken, dass die enthaltene Waveform verlängert werden muss, falls sie noch nicht diesen Vorgaben entspricht. Die Waveform selbst ist auf einen Abtastpunkt genau. Unter Einhaltung dieser Vorgaben kann der Waveformspeicher beliebig segmentiert werden [34].

Für eine höhere Flexibilität und zur Speicheroptimierung besitzt der *WX2184C* zusätzlich einen Sequenzierungsmechanismus. Dies ermöglicht, dass die Waveforms aus dem Speicher in einer Sequenzierungstabelle referenzierbar sind. Darin können die Waveforms beliebig wiederholt oder aneinander gekettet werden. So müssen wiederkehrende Bestandteile einer Waveform nur einmal im Speicher hinterlegt werden, wodurch Redundanzen größtenteils vermeidbar sind. Es können maximal 1000 Szenarios zur Sequenzierung mit jeweils bis zu 48000 Schritten erstellt werden. Mit dem Modus *Advanced Sequencing* können außerdem bis zu 1000 geschachtelte Sequenzen, also „sequenzierte Sequenzen“ oder „Subsequenzen“, erstellt werden. Dadurch wird die Verschachtelungstiefe der möglichen Waveformstrukturen um eine zusätzliche Ebene erweitert [34].

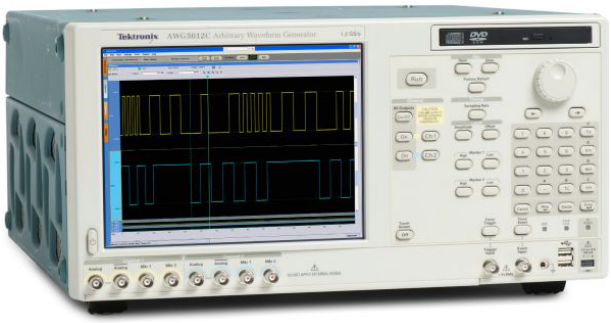
Der *Tabor WX2184C* lässt sich über mehrere Schnittstellen konfigurieren. Zum einen kann ein Großteil der Funktionalität autark mit Hilfe von Bedienelementen und einem Bildschirm am Gerät konfiguriert werden. Ein mächtigeres Werkzeug bietet jedoch die Programmierschnittstelle. Dazu muss das Gerät per USB, LAN oder GPIB[14] mit einem Computer verbunden werden. Dadurch lässt sich das Gerät fernsteuern und es können Daten, wie beispielsweise Waveforms und Sequenzen, auf das Gerät transferiert werden. Zur Programmierung des Geräts wird *SCPI*[15] verwendet. Dabei handelt es sich um einen

[14] Abk. für „General Purpose Interface Bus“; ein Industriestandard (IEEE-488) zur Verbindung eines Computers mit programmierbaren Messgeräten [35].

[15] Abk. für „Standard Commands for Programmable Instruments“

ASCII-basierten und im IEEE-488.2-Standard veröffentlichten Befehlssatz, der für Mess- und Testinstrumente entwickelt wurde. Die Befehle sind in einer hierarchischen Struktur gegliedert und werden in Textform an das Gerät übermittelt. Einige Befehle sind bereits im Standard definiert, während die meisten Befehle jedoch hardwarespezifisch sind. Die Wahl des Kommunikationsmediums und Übertragungsprotokolls spielt bei *SCPI* keine Rolle [34].

3.1.2 Tektronix AWG5014C



Der *Tektronix AWG5012C* ist nahezu baugleich mit dem Modell *AWG5014C*. Lediglich die Anzahl der Ausgabekanäle ist beim *AWG5012C* geringer.

Abbildung 3.2: Tektronix AWG5012C [36]

Der *Tektronix AWG5014C* ist ein vierkanaliger Arbiträrgenerator aus der Modellreihe *AWG5000*, zu der außerdem der nahezu baugleiche *AWG5012C* aus Abbildung 3.2 und das etwas leistungsschwächere Modell *AWG5002C* gehören, die jeweils zwei Ausgabekanäle besitzen [36].

Für jeden der vier Ausgabekanäle des *AWG5014C* existieren zwei Marker-Kanäle zum Triggern anderer Instrumente. Arbiträre Signale können mit bis zu 1,2 Gigasample pro Sekunde erzeugt werden. Somit ist das Gerät nur etwa halb so schnell, wie der *Tabor WX2184C*. Die Wortbreite liegt jedoch ebenfalls bei 14 Bits, wodurch die Amplitude genauso fein und präzise einstellbar ist. Die Schwingungsbreite der Amplitude kann zwischen 40 Millivolt-peak-to-peak und 9 Volt-peak-to-peak variiert werden, bei Verwendung der Differenzausgänge eines Kanals. Als asymmetrischer Ausgang liegt die mögliche Amplitude des Kanals auch hier bei der Hälfte, also maximal 4,5 Volt-peak-to-peak [36].

Zur Konfiguration des *Tektronix AWG5014C* kann der integrierte Bildschirm oder ein externer Rechner genutzt werden. Bei dem Gerät selbst handelt es sich um einen vollwertigen Computer mit dem vorinstallierten Betriebssystem „Windows 7“ von Microsoft.

Dadurch ist der volle Funktionsumfang direkt am Gerät verwendbar. Außerdem kann der *Tektronix AWG5014C* per GPIB direkt an einen Computer angeschlossen werden oder per LAN mit dem Netzwerk verbunden werden, um jederzeit von einem beliebigen Rechner aus dem Netzwerk auf das Instrument zuzugreifen. So können am PC erstellte Waveforms auf den Arbiträrgenerator hochgeladen werden. Eine Verbindung per USB ist nicht möglich [37].

Die Programmierung des Instruments erfolgt, wie auch beim *Tabor WX2184C*, über *SCPI* [37]. Somit ist auch die Kommunikation mit dem *Tektronix* unabhängig von der Programmiersprache, dem Kommunikationsmedium und dem Übertragungsprotokoll. Der Waveformspeicher umfasst eine Kapazität von 16 Megasample pro Kanal und ist optional auf 32 Megasample erweiterbar [36]. Komplexe Waveforms lassen sich durch einen Sequenzierungsmechanismus, ähnlich dem des *Tabor WX2184C*, strukturieren, um dadurch Speicherplatz einzusparen. Dabei können bis zu 8000 Sequenzschritte definiert werden, mit denen die einzelnen Waveforms wiederholt und aneinander gekettet werden können. Ein „Subsequencing“ beziehungsweise „sequenzierte Sequenzen“ sind jedoch nur mit einer zusätzlichen Option verfügbar. Diese Option ermöglicht außerdem eine Echtzeit-Sequenzierung, mit der bedingte Verzweigungen erstellt werden können. Dadurch kann das Gerät in Echtzeit zum Beispiel auf Messwerte oder Signale externer Geräte reagieren [37].

Tektronix Modellreihe AWG70000

Neben der *AWG5000*-Baureihe von *Tektronix* ist auch die Serie *AWG70000* zu berücksichtigen. Diese wird zurzeit zwar noch nicht von der Arbeitsgruppe Quantentechnologie der RWTH Aachen eingesetzt, aber wird zukünftig eine Rolle spielen, da von der Modellreihe *AWG5000* nur noch Restbestände erhältlich sind [36]. Der Funktionsumfang der beiden Modellreihen ist sehr ähnlich, doch in den Spezifikationen gibt es einige Unterschiede. Mit nur 10 Bits vertikaler Auflösung, beziehungsweise 8 Bits bei Verwendung der Marker-Kanäle, sind die Geräte aus dieser Reihe weniger präzise, als der *AWG5014C*. Außerdem sind nur bis zu zwei analoge Ausgabekanäle verfügbar. Der entscheidende Vorteil der Modellreihe *AWG70000* ist die hohe Abtastrate. Mit einer maximalen Abtastrate von 12 Gigasample pro Sekunde, optional bis zu 24 GS/s, sind in dieser Serie bis zu zwanzigfach höhere Ausgangsfrequenzen möglich, wodurch die Waveforms mit einer feineren Granularität abgespielt werden können. Die maximale Amplitude beträgt beim stärksten Modell der Reihe 4 Volt-peak-to-peak. Bei der Programmierung des Geräts bestehen jedoch kaum Unterschiede zur Baureihe *AWG5000* [38].

3.1.3 Zurich Instruments HDAWG8

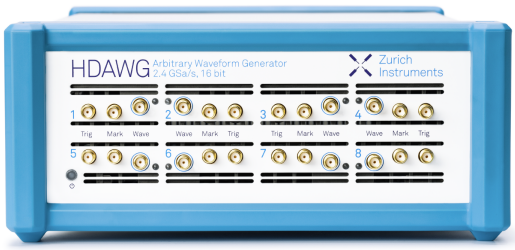


Abbildung 3.3: Zurich Instruments HDAWG8 [39]

Der *Zurich Instruments HDAWG8* ist ein achtkanaliger Arbiträrgenerator, der in Abbildung 3.3 zu sehen ist. Gemeinsam mit dem vierkanaligen *HDAWG4* bildet dieser die *HDAWG*-Baureihe. Die Modelle der Reihe zeichnen sich durch kurze Reaktionszeiten im Echtzeitbetrieb aus und sind vom Hersteller unter anderem für die Anwendung im Bereich Quantencomputing vorgesehen [39].

Die Ausgabekanäle des *HDAWG8* erreichen eine maximale Amplitude von 5 Volt-peak-to-peak. Dabei gibt es pro Kanal nur einen einzelnen physischen Ausgang. Es existieren also keine Differenzausgänge, wie bei den anderen vorgestellten Instrumenten. Jeder Kanal besitzt zusätzlich zwei Marker-Kanäle. Die acht Kanäle lassen sich einzeln oder synchronisiert nutzen. Dabei können sowohl alle acht Kanäle gekoppelt werden, als auch paarweise oder in Vierergruppen. Sollten die acht vorhandenen Kanäle nicht ausreichen, können mehrere *HDAWG8s* synchronisiert werden, um noch mehr gekoppelte Kanäle nutzen zu können. Bei der Erzeugung arbiträrer Signale erreicht der *Zurich Instruments HDAWG8* eine Abtastrate von 2,4 Gigasample pro Sekunde. Für die Waveforms steht pro Kanal ein 64 Megasample großer Speicher zur Verfügung. Dieser ist optional auf ganze 500 Megasample erweiterbar. Die darin gespeicherten Waveforms müssen eine minimale Länge von 32 Samples aufweisen und einer Granularität von 16 Samples. Die Wortbreite des Geräts beträgt 16 Bits. Dies entspricht einer relativen Genauigkeit von 1/65536. Somit ist das Gerät bei 5 Volt-peak-to-peak auf unter 0,1 Millivolt genau [39].

Um eine Verbindung mit einem PC herzustellen, kann der *HDAWG8* direkt per USB an einen PC angeschlossen werden. Alternativ lässt sich dieser per Ethernet in das lokale Netzwerk einbinden, um das Gerät über den integrierten Datenserver zu steuern. Dazu nutzt *Zurich Instruments* ein mächtiges Webinterface, über welches sich das Gerät konfigurieren lässt. Eine Konfiguration direkt am Gerät ist aufgrund nicht vorhandener Bedienelemente nicht möglich [40].

Bei der Programmierung des Instruments wird nicht auf *SCPI* gesetzt, wie es bei den zuvor vorgestellten Geräten der Fall ist. Stattdessen wird ein eigenes Framework namens *LabOne* verwendet, mit welchem sich das Gerät gesteuert und programmiert werden kann. Für das Framework existieren Schnittstellen für diverse Programmiersprachen, unter anderem Python, MATLAB und C. Außerdem kommt der *HDAWG8* mit einem eigenen Compiler, sodass die Waveformsequenzen in Form eines Skripts definiert werden. Dieses Skript lässt sich am Computer kompilieren und auf das Gerät hochladen. Das kompilierte Waveformprogramm kann anschließend auf dem Gerät ausgeführt werden. Die dazu entwickelte Programmiersprache unterstützt alle klassischen Kontrollstrukturen, wie Schleifen, Sprünge und bedingte Verzweigungen. So können Sequenzen mit bis zu 16384 Instruktionen erstellt werden und die Verschachtelungstiefe ist lediglich durch den Speicher begrenzt. Die Bedingungen können in Echtzeit geprüft werden und auch auf externe Trigger und Messergebnisse zugreifen. Dazu besitzt der *Zurich Instruments HDAWG8* Eingangskanäle, die im Waveformprogramm abgefragt werden können. Die Reaktionszeit des Geräts beträgt dabei unter 50 Nanosekunden [40, 41].

3.2 Vergleich der Kernfunktionen und Eigenschaften

Zur Entwicklung eines geeigneten Softwarekonzepts zur Abstraktion der Hardwaretreiber, werden zunächst die Gemeinsamkeiten und Unterschiede zwischen den wichtigsten Funktionen und Eigenschaften der Instrumente herausgestellt. Auf der Grundlage dieser Erkenntnisse ist es anschließend möglich, ein Softwarekonzept zu entwerfen, das verschiedene Eigenschaften und unterschiedlich implementierte Funktionen flexibel zusammenfassen kann. Im folgenden Abschnitt geben die Kernfunktionen und Spezifikationen der Arbiträrgeneratoren einen Überblick darüber, worin sich der Funktionsumfang der Instrumente unterscheidet und wo es Überschneidungen gibt. Die aufgeführten Eigenschaften der Geräte umfassen neben der von allen Geräten unterstützten Basisfunktionalität auch relevante Funktionen, die häufig verwendet werden und ein hohes Optimierungspotential besitzen. In der anschließenden Konzeptionierung werden die hier gewonnen Erkenntnisse berücksichtigt, um ein optimales Softwarekonzept zu erhalten.

Spezifikationen

In den Spezifikationen der Hardware ist eine hohe Varianz zwischen den Modellen festzustellen. Dabei handelt es sich jedoch nur um Beschränkungen, die nicht struktureller Natur sind. So machen diese sich in der späteren Implementierung nur durch unterschiedliche Zahlenwerte bemerkbar, die ohnehin variabel sein werden. Zu den relevanten

Eigenschaften der Hardwarespezifikationen zählen die Kanalanzahl, die Abtastrate, die Wortbreite, die Amplitude und die Kapazität des Waveformspeichers. In Tabelle 3.1 sind diese Eigenschaften für die einzelnen Geräte gegeneinander aufgestellt. Darin ist zu erkennen, dass sich die Geräte in manchen Eigenschaften gleichen oder ähneln und an anderen Eigenschaften zum Teil sehr unterschiedlich sind. Für die spätere Treiberentwicklung stellt dies jedoch kein Problem dar, da diese Zahlenwerte für jeden Treiber separat angepasst werden können. Die Anzahl der Kanäle, die maximale Amplitude und die Speicherkapazität können eventuell dafür sorgen, dass gewisse mehrkanalige Pulsvorlagen nicht auf einem Gerät abgespielt werden können, da zu wenige Kanäle vorhanden sind, ein zu hoher Spannungsbereich benötigt wird oder der Puls zu lang ist. Die Abtastrate und Wortbreite wirken sich hingegen nur auf den Vorgang des Abtastens der Pulssequenzen aus. Dabei bestimmt die Abtastrate die zeitliche Auflösung des gesampelten Programms und aus der Wortbreite ergibt sich die vertikale Auflösung der Spannungsniveaus.

Tabelle 3.1: Vergleich der Hardwarespezifikationen

	Tabor WX2184C	**Tektronix AWG5014C**	**Zurich Instruments HDAWG8**
Kanäle	4 Kanäle, 1 Marker pro Kanal	4 Kanäle, 2 Marker pro Kanal	8 Kanäle, 2 Marker pro Kanal
Abtastrate	min. 75 MS/s max. 2,3 GS/s	min. 10 MS/s max. 1,2 GS/s	min. 50 MS/s max. 2,4 GS/s
Wortbreite	14 Bit	14 Bit	16 Bit
max. Amplitude*	2 V_{pp} (s.e.) 4 V_{pp} (diff.)	4,5 V_{pp} (s.e.) 9 V_{pp} (diff.)	5 V_{pp} (s.e.)
Speicherkapazität	16 MS pro Kanal (optional: 32 MS)	16 MS pro Kanal (optional: 32 MS)	64 MS pro Kanal (optional: 500 MS)

* Die Kanäle der Geräte *Tabor WX2184C* und *Tektronix AWG5014C* unterstützten neben dem „single-ended"-Modus (s.e.) mit einem asymmetrischen Ausgang auch einen differentialen Modus (diff.), bei dem die Spannung zwischen den zwei Differenzausgängen angelegt wird. Der *Zurich Instruments HDAWG8* besitzt keine Differenzausgänge, sondern nur asymmetrische Ausgänge.

Die vorgestellten Eigenschaften sind teils sehr unterschiedlich zwischen den Instrumenten. Da jedoch alle Instrumente diese Eigenschaften aufweisen, nur in unterschiedlichen Ausprägungen, wird dies die Struktur des späteren Softwarekonzepts nur geringfügig beeinflussen. Die Spezifikationen können für jedes Instrument auf die gleiche Weise umgesetzt werden, wodurch der Entwurf einer verbesserten Treiberstruktur erleichtert wird.

Kommunikation und Programmierung

Zur Kommunikation mit den Instrumenten müssen diese zunächst mit einem Computer verbunden werden. Wie in der Beschreibung der Geräte bereits erwähnt, werden dazu verschiedene Schnittstellen, wie USB, GPIB oder Ethernet, verwendet. Was die Kommunikation und Programmierung betrifft, lassen sich die Geräte in zwei Klassen einteilen. Daher werden zunächst die Geräte *Tabor WX2184C* und *Tektronix AWG5014C* betrachtet. Diese verwenden die *Virtual Instrument Software Architecture* oder kurz *VISA*. Dabei handelt es sich um eine standardisierte API zur Kommunikation mit Mess- und Testinstrumenten. *VISA* ist unabhängig vom Übertragungsmedium und kann dadurch mit allen vom Instrument bereitgestellten Schnittstellen genutzt werden. Zur Kommunikation werden Kommandos und Antworten von den *VISA*-Bibliotheken in Textform zwischen Computer und Instrument ausgetauscht [42]. Diese sind wiederum unabhängig von *VISA*, da die Programmierung in einer eigenen Ebene abstrahiert ist. Die beiden Instrumente von *Tabor* und *Tektronix* werden mit der zuvor erwähnten Schnittstelle *SCPI* programmiert. Die ASCII-basierten Kommandos von *SCPI* sind hierarchisch in einer Baumstruktur gegliedert und zusammengehörende Kommandos werden unter einem Knoten gruppiert. Die Schlüsselwörter der einzelnen Baumebenen werden durch Doppelpunkte voneinander getrennt [34]. Ein beispielhaftes Kommando zur Konfiguration eines Sinussignals beim *Tabor WX2184C* ist in Programmcode 3.1 zu sehen.

```
1 :SOURce:FUNCtion:SHAPe SINusoid
2 :SOURCE:FUNCTION:SHAPE SINUSOID
3 :SOUR:FUNC:SHAP SIN
```

Programmcode 3.1: Beispielkommando in *SCPI* [34]

Alle drei Zeilen von Programmcode 3.1 sind Varianten desselben Kommandos, da *SCPI* case-insensitive ist. Es wird also nicht zwischen Groß- und Kleinschreibung unterschieden. Außerdem existiert eine Kurzschreibweise, sodass die Befehle nicht zwingend ausgeschrieben werden müssen, wie in der dritten Zeile zu sehen ist. In den Programmierreferenzen der Geräte werden die Kommandos wie in der ersten Zeile aufgeführt. Dabei zeigen die Großbuchstaben die mögliche Kurzschreibweise. Der erste Teil des Kommandos

ist vergleichbar mit einer Funktion in höheren Programmiersprachen. Dahinter folgen die Funktionsparameter, in diesem Fall `SINusoid` [34]. Obwohl das Format der *SCPI*-Kommandos standardisiert ist, sind es die Kommandos selbst nicht und können sich somit von Gerät zu Gerät unterscheiden. Nur einige wenige Kommandos sind bereits im *SCPI*-Standard definiert, wie beispielsweise das Zurücksetzen eines Instruments mit `*RST` [43].

Als Nächstes wird der *Zurich Instruments HDAWG8* hinsichtlich der Programmierschnittstelle betrachtet. Dieser verwendet nicht die Kombination aus *VISA* und *SCPI*, sondern ein eigenes serverbasiertes Kommunikationsmodell. Es existiert ein Softwareframework namens *LabOne* mit Schnittstellen für diverse Programmiersprachen und Plattformen, mit denen sich das Instrument steuern lässt. So lässt sich der *HDAWG8* unter anderem über Python, MATLAB oder C programmieren, sowie über ein Webinterface namens *LabOne UI*, das in allen gängigen Internetbrowsern genutzt werden kann [39].

Jede Kommunikation mit dem *Zurich Instruments HDAWG8* verläuft über einen Datenserver, der zwischen dem Instrument und den Clients vermittelt. Dieser stellt auch den Webserver für das Browserinterface bereit. Auch die Softwareframeworks, wie zum Beispiel `ziPython` für Python oder `ziDAQ` für MATLAB, kommunizieren mit diesem Datenserver, der die Anfragen an das Instrument weiterreicht. Die Instrumentensteuerung verläuft beim *HDAWG8* größtenteils über hierarchisch gegliederte, ASCII-basierte Kommandos, die sich jedoch nicht an *SCPI* orientieren, aber ähnlich aufgebaut sind. Manche Aktionen, wie beispielsweise der Verbindungsaufbau zum Datenserver, werden jedoch auch objektorientiert zur Verfügung gestellt.

```
1 wave w_gauss = 1.0 * gauss(640, 320, 50);
2
3 repeat (5) {
4     playWave(1, w_gauss);
5 }
```

Programmcode 3.2: Beispielskript für den *Zurich Instrumemts HDAWG8* [41]

Eine weitere Besonderheit des Arbiträrgenerators von *Zurich Instruments* ist, dass die Waveforms in Form von Programmcode definiert werden. Soll beispielsweise ein Signal in Form einer Gaußschen Glocke erzeugt und fünfmal wiederholt werden, kann dies mit dem Programmcode 3.2 umgesetzt werden. Der Quelltext aus dem Beispiel erinnert an die Syntax von C oder anderen C-basierten Sprachen. Bevor die Waveforms abgespielt werden können, wird das Skript auf dem Computer kompiliert, sodass ein ausführbares Waveformprogramm entsteht. Dieses wird in den Speicher des Instruments geladen und lässt sich anschließend ausführen [41].

Aufgrund der unterschiedlichen Schnittstellen und Kommunikationsprotokolle muss die Programmierung und Kommunikation der Instrumente bei der Konzeptionierung berücksichtigt werden. Obwohl es auch Gemeinsamkeiten zwischen dem *Tabor WX2184C* und *Tektronix AWG5014C* gibt, müssen selbst diese gesondert betrachtet werden. Diese nutzen zwar das gleiche Übertragungsprotokoll, aber verschiedene *SCPI*-Befehlssätze, wodurch eine Vereinheitlichung nur schwer umsetzbar wäre. Verglichen mit dem *Zurich Instruments HDAWG8* sind die Unterschiede sogar noch größer. Die Hardware-Kommunikation und die Programmierung der Geräte sind der Kern eines Treibers und die zentrale Verbindung zwischen Gerät und Treiber. Somit wird dies insgesamt zu einer der wichtigsten Aufgaben für die Konzeptionierung werden.

Synchronisation von Kanälen und Geräten

Ein weiterer wichtiger Aspekt bei der Entwicklung eines soliden Softwarekonzepts für die Arbiträrgeneratoren ist die Synchronisation von Kanälen. Die Kanalsynchronisation ist bei den Geräten *Tabor WX2184C* und beim *Zurich Instruments HDAWG8* möglich und unterscheidet sich lediglich im Umfang. Bei der Kopplung von Kanälen teilen sich mehrere Kanäle einen gemeinsamen Taktgeber, sodass die synchronisierten Kanäle gleich getaktet sind. Der *WX2184C* hat standardmäßig jeweils zwei Kanäle paarweise synchronisiert, unterstützt aber auch eine Vollsynchronisation aller vier Kanäle. Der *HDAWG8* unterstützt aufgrund der höheren Anzahl an Kanälen neben der paarweisen Synchronisation und der Vollsynchronisation aller acht Kanäle auch die Synchronisation in Vierergruppen. Bei beiden Geräten können die Kanäle nicht beliebig gruppiert werden, sondern nur in den Standardkonstellationen [34, 41].

Neben der Kanalkopplung unterstützen die beiden Geräten auch eine Gerätesynchronisation. Der *Tabor WX2184C* lässt sich mit einem weiteren *WX2184C* synchronisieren, während sich der *Zurich Instruments HDAWG8* auch mit zwei oder mehr weiteren Geräten synchronisieren lässt [32, 39]. Bei der Bedienung soll es sich für den Nutzer so anfühlen, als würde er mit einem einzigen Arbiträrgenerator arbeiten, der lediglich mehr Kanäle besitzt. Dies wird bereits durch die Treiber der Hersteller unterstützt [32, 41]. Der *Tektronix AWG5014C* unterstützt hingegen keine Kanal- oder Gerätesynchronisation, sondern lediglich die Synchronisation der Messungen per Marker und Trigger, was mit allen Geräten möglich ist [36].

Die Synchronisierung der Kanäle wird bei der Struktur des neuen Treiberkonzepts eine wichtige Rolle spielen. Dieser Punkt ist eng mit der intuitiven Bedienbarkeit der Treiber verbunden, denn die synchronisierten Kanalgruppen eines Geräts bilden ein neues virtuelles Gerät, das unabhängig von den anderen Kanalgruppen agieren kann. Dies macht die

Kanalsynchronisierung zu einer zentralen Funktion der Treiber. Die daraus resultierenden Kanalgruppen werden dem Benutzer als zentrale Kommunikationsobjekte nützen, über welche ein Großteil der Messkonfiguration erfolgen wird. Die Gerätesynchronisation auf der anderen Seite ist wiederum eine spezifische Funktion, die nicht von allen Instrumenten unterstützt wird und zudem in unterschiedlichem Umfang. Für diese spezifischen Funktionen muss ein eigenes Konzept erstellt werden, damit solche Funktionen flexibel aber dennoch einheitlich bereitgestellt werden können.

Speicherverwaltung und Sequenzierung der Waveforms

Die Waveformspeicher der drei Geräte sind aus allgemeiner Sicht sehr ähnlich aufgebaut. Im Speicher lassen sich Waveforms beliebiger Länge definieren. Diese beinhalten zum einen den eindeutigen Namen beziehungsweise die ID der Waveform, womit diese identifizierbar ist. Zum anderen besteht eine Waveform aus den arbiträren Spannungsniveaus pro Abtastpunkt, die für jeden Kanal getrennt definiert werden können. Beim *Tabor WX2184C* und *Tektronix AWG5014C* werden die Waveforms per *SCPI*-Kommandos auf die Geräte hochgeladen und in den Speicher geschrieben [34, 37]. Beim *Zurich Instruments HDAWG8* werden die Waveforms in Dateien vom CSV-Format gespeichert, welche anschließend in den Waveformspeicher des Geräts geladen werden [41]. Die Waveforms lassen sich jeweils einzeln abspielen oder können in Sequenzen referenziert werden, um komplexere Waveformstrukturen erstellen zu können.

Den einfachsten Sequenzierungsmechanismus bietet der *Tektronix AWG5014C* in der Basisausstattung. Dabei wird eine Sequenzierungstabelle bereitgestellt, in der Waveformsequenzen definiert werden können. Die Sequenzen enthalten alle einen eindeutigen Namen und Referenzen auf die Waveforms. Optional können die Waveforms beliebig oft wiederholt werden, Wartezeiten können eingebaut werden und es lässt sich eine Sequenz referenzieren, die im Anschluss an die momentane Sequenz gestartet wird [37].

Der *Tabor WX2184C* besitzt serienmäßig zwei Sequenzierungstabellen, was beim *Tektronix AWG5014C* nur als optionale Erweiterung erhältlich ist. Die eine Sequenzierungstabelle funktioniert analog zu der des *Tektronix AWG5014C*, da in dieser die Waveforms referenziert werden. Aus der anderen Sequenzierungstabelle wird hingegen auf die Sequenzen aus der ersten Tabelle verwiesen. Dadurch ergibt sich eine zweistufige Sequenzierung der Waveforms. Somit können mit diesem Instrument Sequenzen und Subsequenzen angelegt werden, was eine höhere Komplexität der Waveformstruktur ermöglicht [34].

Noch komplexere Strukturen sind mit dem *Zurich Instruments HDAWG8* möglich. Dieser bietet mit der eigens entwickelten Programmierschnittstelle eine sehr flexible Sequenzierung mit quasi unbegrenzter Komplexität. Im Waveformskript können Steueranweisungen, wie zum Beispiel Verzweigungen mit `if` und `else` oder `for`-, `while`- oder `repeat`-Schleifen, verwendet werden. Diese werden vom Compiler in Sequenzen übersetzt, welche die enthaltenen Waveforms entsprechend wiederholen und aneinander ketten. Das kompilierte Programm lässt sich anschließend in den Speicher des Instruments laden, um es schließlich ausführen zu können [41].

Die verschiedenen Sequenzierungsmechanismen sind mächtige Werkzeuge zur Speicheroptimierung auf der Hardware. So können längere und komplexere Waveforms gestaltet werden, die normalerweise den Speicher des Instruments zum Überlauf bringen würden. Durch die ein-, zwei- oder mehrstufige Sequenzierung der Waveforms lassen sich wiederkehrende Waveforms platzsparend strukturieren. Die Optimierung dieses Prozesses wird bei der Feinkonzeptionierung eine zentrale Rolle einnehmen. Ziel ist es, die Waveforms möglichst effizient zu speichern, sodass auch große Waveforms flexibel gehandhabt werden können.

4 Konzeptionierung

In diesem Kapitel werden die Erkenntnisse aus der Hardwareanalyse genutzt, um ein Softwaresystem zu konzipieren, dass sich an den Funktionen und Eigenschaften der Hardware orientiert. Zunächst wird eine Übersicht über die Zielsetzung der Konzeptionierung gegeben, wobei unter anderem auf allgemeine Konventionen und Richtlinien aus der Softwareentwicklung zurückgegriffen wird, um eine transparente und intuitive Schnittstelle zu erhalten. Darauf folgt die grobe Konzeptionierung der Treiber, in der es darum geht, den Treibern einen strukturellen Rahmen vorzugeben und eine einheitliche Schnittstelle zu erzeugen. Anschließend folgt die Verfeinerung des Konzepts. Dabei werden die Kernfunktionen der Hardware in das Konzept eingepflegt, Optimierungen werden vorgenommen und die Anbindung von Tests wird diskutiert.

4.1 Zielsetzung

Die Konzipierung eines neuen Treibersystems für die Arbiträrgeneratoren der Arbeitsgruppe Quantentechnologie der RWTH Aachen hat zwei primäre Ziele, welche sich an ursprünglichen für *qupulse* definierten Zielen orientieren. Zum einen sollen die Treiber grundsätzlich vereinheitlicht werden und zum anderen gilt es, den Funktionsumfang der Treiber zu optimieren und zu erweitern. Neben diesen beiden Zielen soll die Software intuitiv strukturiert sein und sauberer Programmcode entwickelt werden. Das heißt, die Benutzer der Treiber sollen sich leicht in dem System der Treiber zurechtfinden und der Programmcode muss auch für Erstanwender gut lesbar sein. Um dies zu erreichen werden allgemein geltende Konventionen und Regeln der Softwaretechnik, insbesondere der Programmierung mit Python, berücksichtigt. Außerdem ist zu beachten, dass die Software durch zunehmende Flexibilität sehr komplex werden kann. Stattdessen sollen die Treiber so umgesetzt werden, dass auch nachträgliche Erweiterungen mit geringem Aufwand möglich sind.

4.1.1 Hardwareabstraktion

Im derzeitigen Zustand der AWG-Treiber ist lediglich die Kommunikation mit dem *qupulse*-Back-End durch eine einheitliche Schnittstelle definiert. Dies ist erforderlich, um überhaupt einen Treiber in das Framework integrieren zu können. Daher stellt die Schnittstelle nur den vom *qupulse*-Back-End benötigten Anteil an Funktionalität bereit. Alle übrigen Funktionen der Treiber sind nicht standardisiert und können daher variieren.

Vor jedem Experiment müssen die Instrumente vorbereitet und konfiguriert werden. Dies geschieht mit Hilfe der Hardwaretreiber, an denen die erforderlichen Einstellungen vorgenommen werden. Da diese Konfiguration noch nicht vereinheitlicht ist, kann keine intuitive Bedienung gewährleistet werden. Daher ist ein Ziel dieser Arbeit, eine Virtualisierung beziehungsweise Abstraktion der Hardwaretreiber vorzunehmen. Dadurch kann sichergestellt werden, dass alle Treiber auf dieselbe Weise bedient werden, wodurch die Geräte und dessen Treiber sich mit minimalem Aufwand in einer bestehenden Konfiguration austauschen lassen. Dies vereinfacht einerseits die Bedienung der Treiber und erhöht andererseits die Wartbarkeit und Erweiterbarkeit der Treiber.

4.1.2 Optimierung und Erweiterung der Funktionalität

Das zweite Ziel ist die Erweiterung und Optimierung der von den Treibern bereitgestellten Funktionalität. Bislang wurden die Hardwaretreiber sehr schlicht und einfach gehalten. Viele Funktionen wurden dabei auf eine unkomplizierte und geradlinige Weise implementiert, um möglichst schnell einsatzfähig zu sein. Dabei wurde jedoch das mögliche Optimierungspotential der einzelnen Geräte vernachlässigt. Besonders beim Hochladen und bei der Speicherung der Waveforms auf einem Instrument wird die mögliche Leistung der Instrumente nicht ausgereizt. Mit Sequenzierungsmechanismen und intelligenter Speicherverwaltung wäre es hingegen möglich, komplexere und längere Waveforms als bisher auf einem Gerät zu speichern und diese schneller auf das Gerät zu übertragen. Daher soll das Treiberkonzept darauf ausgelegt werden, möglichst den vollen Funktionsumfang der Instrumente nutzen zu können, wobei die Bedienung der Treiber jedoch einheitlich sein soll.

Außerdem soll das Treibersystem auch um neue Treiber erweiterbar sein, die bisher noch nicht zum Einsatz kommen. Somit sollen auch Arbiträrgeneratoren angebunden werden, die in der Hardwareanalyse nicht betrachtet wurden. Aus diesem Grund muss das

Treiberkonzept möglichst flexibel sein und darf keine strengen strukturellen Vorgaben machen, die nur mit den hier thematisierten Geräten kompatibel sind. Dadurch soll die Treiberstruktur offen für Erweiterungen werden.

4.1.3 Softwarearchitektur und Clean Code

Zur Entwicklung eines intuitiven und selbsterklärenden Softwaresystems existieren heutzutage eine Vielzahl an Konventionen und Richtlinien. Diese beschäftigen sich unter anderem mit der Strukturierung von Softwarearchitekturen und der Entwicklung von *Clean Code*, also „sauberem“ Programmcode. Die Softwarearchitektur befasst sich mit den groben Strukturen und blickt aus einer größeren Entfernung auf ein Softwaresystem. Die Bewertung von sauberem Programmcode erfolgt hingegen auf tieferer Ebene und bezieht sich direkt auf den Quellcode. Eine strukturierte Softwarearchitektur und sauberer Programmcode verbessern dabei nicht nur die Erweiterbarkeit und reduzieren die Fehleranfälligkeit eines Systems. Ebenfalls erleichtern diese das Verständnis des Systems durch Dritte, wodurch sich zum Beispiel Einarbeitungszeiten in das System verringern und dessen Wartbarkeit zunimmt [44, 45].

Die verschiedenen Richtlinien können sowohl die Softwareentwicklung im Allgemeinen als auch die Programmierung mit einer bestimmten Programmiersprache betreffen. Bei den im Folgenden vorgestellten Konventionen handelt es sich jedoch nicht um menschengemachte Prinzipien, die willkürlich festgelegt wurden. Stattdessen handelt es sich bei den meisten dieser Richtlinien um Erfahrungen und *best practices*, die sich bereits in der Praxis bewährt haben. Diese Praktiken wurden von diversen Autoren zusammengefasst und werden seither als allgemeingültige Konventionen geachtet [46].

SOLID-Prinzipien

Eine Gruppe von Prinzipien zur Entwicklung von sauberem Code und übersichtlichen Softwarearchitekturen sind zum Beispiel die *SOLID*-Prinzipien, die *Robert C. Martin* erstmals in seinem Werk „Design Principles and Design Patterns“ aus dem Jahre 2000 veröffentlichte [47]. Das Akronym *SOLID* wurde kurze Zeit später von *Michael Feathers* eingeführt und steht für die folgenden fünf Prinzipien [48]:

- **S**ingle Responsibility Principle (SRP)
- **O**pen Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Die *SOLID*-Prinzipien beschäftigen sich im Allgemeinen mit Vererbung, Komposition und Aggregation von Klassen. Es handelt sich um Prinzipien der Softwareentwicklung, die bei korrekter Umsetzung zum Teil automatisch für einen gut wartbaren und leicht verständlichen Quellcode sorgen. Dadurch sollen die Klassen und Funktionen so selbsterklärend sein, dass Kommentare überflüssig werden [49]. Im Folgenden werden die einzelnen *SOLID*-Prinzipien kurz thematisiert.

Das *Single Responsibility*-Prinzip besagt, dass jede Klasse nur eine einzige Verantwortung haben darf. Dies meint auch, dass es nie mehr als einen Grund zur Änderung einer Klasse geben soll. Mehrere Verantwortungen in einer Klasse führen dazu, dass die Klasse mehrere Bereiche abdeckt, in denen zukünftig Änderungen anfallen können. Dies steigert gleichzeitig die Fehleranfälligkeit der Klasse. Unter Einhaltung dieses Prinzips bleibt die Klasse so einfach wie möglich, wodurch sie leichter verständlich, effizienter und weniger fehleranfällig ist [50].

Das zweite *SOLID*-Prinzip ist das *Open Closed Principle*. Dieses besagt konkret, dass eine Softwareentität, wie zum Beispiel eine Klasse, ein Interface oder eine Funktion, offen für Erweiterungen und geschlossen gegenüber Veränderungen sein muss. Dadurch wird die Erweiterung der Software ermöglicht, ohne bestehende Komponenten modifizieren zu müssen. Dieses Ziel ist in der Softwareentwicklung sehr erstrebenswert, aber in bestehenden Systemen meist schwer realisierbar, sodass alte Softwareartefakte dennoch angepasst werden müssen. Bei einer Neuentwicklung oder Umstrukturierung eines Systems, wie in dieser Arbeit, sollte dieses Prinzip jedoch nicht vernachlässigt werden [51].

Ein weiteres Prinzip ist das *Liskovsche Substitutionsprinzip*, welches nach *Barbara Liskov* benannt ist, die dieses Prinzip bereits 1993 auf einer Konferenz vorstellte [52]. Es besagt, dass eine Basisklasse jederzeit durch eine Unterklasse ersetzbar, also substituierbar, sein muss. Dies bedeutet, dass eine Unterklasse die Funktionalität der Basisklasse zwar erweitern, aber nicht einschränken darf [53].

Das *Interface Segregation*-Prinzip ist das Vierte der *SOLID*-Prinzipien. Dieses sagt aus, dass die Subklassen eines Interfaces nicht „gezwungen“ werden dürfen, eine Schnittstellenmethode zu implementieren, die weder benötigt noch unterstützt wird. Ziel dieses Prinzips ist die Vereinfachung von Interfaces und die Vermeidung von großen und aufgeblasenen Interfaces. Dies ist beispielsweise durch die Aufteilung auf mehrere kleine Interfaces zu erreichen [45].

Der letzte Buchstabe von *SOLID* steht für das *Dependency Inversion*-Prinzip. Ziel dieses Prinzips ist die Entflechtung und Isolierung von Klassen. Eine starke Kopplung zwischen Klassen erschwert die Wartung oder Erweiterung, da dies bei Änderungen zu

unerwünschten Nebeneffekten führen kann. Um dies zu vermeiden, dürfen zum einen keine Abhängigkeiten zwischen verschiedenen Abstraktionsebenen entstehen. Zum anderen sollen Abstraktionen nicht von Details und Implementierungen abhängen, sondern umgekehrt. Wenn sich Implementierungen an der Abstraktion orientieren, bleiben die Klassen und Schnittstellen eines Systems lose gekoppelt. Dadurch lassen sich einzelne Klassen einfach erweitern, anpassen, neu entwickeln oder austauschen, ohne dass andere Teile des Systems betroffen sind [54].

Entwurfsmuster

Konkrete Vorschläge zur Umsetzung verschiedener Prinzipien aus der Softwaretechnik bieten die *Entwurfsmuster* beziehungsweise *Design Patterns*, die zum Teil auf die *SOLID*-Prinzipien aufbauen. Der Begriff der *Entwurfsmuster* kommt ursprünglich aus der Architektur und wurde vom Architekten *Christopher Alexander* geprägt. Mit diesen lassen sich wiederkehrende Probleme oder Szenarios einfach, vor allem aber einheitlich, lösen. Im Bereich der Softwaretechnik geht es bei den *Entwurfsmustern* zwar nicht um architektonische Komponenten, wie Gebäude, Räume und Wände, sondern um Softwareentitäten, wie Klassen, Interfaces und Funktionen. Das Prinzip der *Entwurfsmuster* lässt sich aber auch im Softwarebereich optimal umsetzen. *Entwurfsmuster* bieten strukturelle Vorlagen für häufig wiederkehrende Problemstellungen bei der Neu- oder Weiterentwicklung von Software. Ein besonderer Fokus liegt dabei auf der Interaktion zwischen Mensch und Computer, indem die *Entwurfsmuster* für einheitliche Schnittstellen und für strukturierten und verständlichen Programmcode sorgen [55].

Große Bekanntheit erlangten die *Entwurfsmuster* vor allem durch das Buch „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software“[16] von *Erich Gamma* und drei weiteren Mitautoren, die häufig als *Gang of Four* bezeichnet werden [55]. In ihrem Buch stellen die Autoren 23 verschiedene *Entwurfsmuster* vor, die in die Kategorien Erzeugungsmuster, Strukturmuster und Verhaltensmuster gegliedert sind. Jedes *Entwurfsmuster* wird darin detailliert beschrieben, Vor- und Nachteile werden abgewägt und die Vorgehensweise wird anhand von Beispielen erläutert [46]. Bei der Konzeptionierung des Treibersystems wird auf manche dieser *Entwurfsmuster* zurückgegriffen, die in diesem Kontext genauer erläutert werden. Zwei zentrale Aussagen der *Gang of Four* sind dabei besonders wichtig für die Entwicklung eines Softwaresystems:

Eine dieser Aussagen lautet: „Programmiere auf eine Schnittstelle hin, nicht auf eine Implementierung“ [46, S. 25]. Dies bedeutet, dass die Implementierung einer Klasse dem Entwickler nicht bekannt sein muss. Er soll die Möglichkeit haben, lediglich mit den

[16] im engl. Original „Design Patterns: Elements of Reusable Object-Oriented Software“

Interfaces oder abstrakten Basisklassen zu kommunizieren, ohne die konkrete Implementierung berücksichtigen zu müssen. Dazu muss die Software so konzipiert sein, dass die Schnittstelle bereits alle notwendigen Funktionen bereitstellt und diese intern mit einer konkreten Implementierung verbindet. Dem Entwickler muss im Idealfall nicht bekannt sein, welche Implementierung im Hintergrund angesprochen wird, da dies dem Back-End des Systems überlassen wird. Dieses Ziel soll auch für die AWG-Treiber gelten.

Die zweite Aussage betrifft die Verwendung von Vererbung. Es wird gefordert, die „Objektkomposition der Klassenvererbung [vorzuziehen]“ [46, S. 27]. Dieses Prinzip erinnert an das *Single Responsibility Principle*, denn durch große Vererbungshierarchien werden Klassen immer größer und erweitern somit fortlaufend ihr Aufgabenfeld. Es kommen regelmäßig neue Funktionen und Aufgabe zu einer Klasse hinzu, wodurch die ursprüngliche Aufgabe der Klasse nach und nach in den Hintergrund rückt. Durch die Verwendung von Objektkomposition werden die verschiedenen Funktionen und Aufgaben auf mehrere Klassen verteilt. Dadurch teilen sich die Verantwortungen auf mehrere Klassen auf und das *Single Responsibility*-Prinzip wird ebenfalls eingehalten [46].

Programmierung mit Python

Neben allgemeingültigen Richtlinien zur Softwareentwicklung, wie die *SOLID*-Prinzipien oder die *Entwurfsmuster*, gibt es für jede Programmiersprache eigene Konventionen. Diese hängen von den speziellen Gegebenheiten einer Programmiersprache ab, zum Beispiel von der Syntax oder davon, ob die Sprache funktional, dynamisch, deklarativ oder objektorientiert ist. Alle Eigenschaften einer Sprache werden dabei berücksichtigt, denn jede Programmiersprache deckt eigene Einsatzgebiete ab und besitzt einen charakteristischen Stil [56].

Auch Python besitzt einige Besonderheiten, die unter anderem bei der Anwendung der *Entwurfsmuster* beachtet werden müssen. Zum Beispiel besitzt Python eine implizite und dynamische Typisierung und es handelt sich um eine interpretierte und nicht kompilierte Sprache. Python ist eine Sprache, die auf der Philosophie gut durchdachter *best practices* basiert, weshalb sie sich sehr gut für die Verwendung von *Entwurfsmustern* eignet. Die Umsetzung ist jedoch nicht zwingend so trivial, wie in anderen Sprachen, aufgrund von Pythons Dynamik und Flexibilität: Zum Beispiel werden Python-Skripte direkt interpretiert und müssen nicht kompiliert werden und Objekte lassen sich zur Laufzeit um neue Funktionen und Attribute erweitern. Außerdem bietet Python eine mächtige Standardbibliothek sowie eine Vielzahl an externen Bibliotheken. Zum Teil stellt Python bereits automatisch Realisierungen von *Entwurfsmustern* bereit, sodass

diese bereits implizit vom Entwickler genutzt werden. Python selbst gibt bereits einige Regeln vor, wie eleganter Python-Code auszusehen hat und wie dieser strukturiert sein soll [56].

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious
 ↪ way to do it.
Although that way may not be obvious at first unless
 ↪ you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad
 ↪ idea.
If the implementation is easy to explain, it may be a
 ↪ good idea.
Namespaces are one honking great idea -- let's do more
 ↪ of those
```

Programmcode 4.1: Zen of Python

Der zentrale Style Guide für Python nennt sich *PEP-8*[17] und wurde im Jahr 2001 zur Verbesserung der Lesbarkeit von Python-Code entwickelt. Dieser beschäftigt sich ausschließlich damit, wie Python-Code geschrieben wird. Dazu zählen unter anderem das Code-Layout, Benennungskonventionen, Einrückung und Kommentare [57, 58]. Da es sich dabei jedoch um feine Implementierungsdetails handelt, ist *PEP-8* für diese Arbeit zunächst weniger relevant. Anders verhält es sich mit dem *Zen of Python*, das aus allgemeinen Regeln für die Programmierung mit Python besteht und den Rahmen für

[17] Abk. für „Python Enhancement Proposal“

eleganten Python-Code vorgibt. Ein Teil dieser Regeln bezieht sich ebenfalls auf Implementierungsdetails, während andere auch allgemeine oder strukturelle Aussagen treffen. Das *Zen of Python* ist fest in Python integriert. In Programmcode 4.1 ist zu sehen, wie sich dieses durch das Importieren des Pakets `this` anzeigen lässt [56].

Die Kombination von *Zen of Python* und *PEP-8* werden besonders bei der Implementierung im Anschluss an diese Arbeit eine große Rolle spielen. Sie dienen als Grundlage für sauberen, lesbaren und wartbaren Programmcode. Unter Hinzunahme der *Entwurfsmuster* und *SOLID*-Prinzipien in der Konzeptionsphase wird ein konsistentes und erweiterbares Softwarepaket entstehen. Dies vereinfacht nicht nur die Arbeit mit den AWG-Treibern und steigert so die Akzeptanz von Seiten der Benutzer, sondern kann auch zu einer geringeren Fehleranfälligkeit beitragen [56].

4.2 Grobkonzept

Die Konzeptionierung der Treibervirtualisierung erfolgt. Dazu wird ein Top-Down-Ansatz gewählt. Ein solcher ist zwar weniger flexibel als ein agiler Bottom-Up-Ansatz, bietet jedoch andere Vorteile. Zum einen können auf diese Weise zuerst die Schnittstellen definiert werden, um im Anschluss das Feinkonzept daran anzupassen und zum anderen können bereits existierende Schnittstellen besser berücksichtigt werden. Außerdem werden die feinen Implementierungsdetails in dieser Arbeit nicht in Gänze thematisiert, da diese nur die Konzeptionierung zum Ziel hat. Bei einem Bottom-Up-Ansatz dagegen spielen die feinen Details und die Implementierung von Beginn an eine Rolle. Dennoch wird bei der Konzeptionierung bereits darauf geachtet, dass es möglich ist, die Treiber nach der Implementierung dieses Konzepts inkrementell weiterzuentwickeln. Dazu wird das Konzept auf eine modulare Struktur ausgerichtet. Dies ermöglicht eine agile Weiterentwicklung nach dem Bottom-Up-Prinzip, wodurch Erweiterungen in kleineren Schritten erfolgen können. Umstrukturierungen oder großer Planungsaufwand werden dabei nicht notwendig sein, um die Software weiterzuentwickeln [59].

Im Folgenden werden verschiedene Klassen und Strukturen vorgestellt, die im Laufe dieser Arbeit konzeptioniert wurden. Dazu werden UML-Klassendiagramme verwendet, die jedoch nicht auf die Syntax von Python ausgelegt sind. Da es in Python zum Beispiel keine Sichtbarkeiten gibt und jedes Attribut öffentlich ist, können die Sichtbarkeiten aus dem UML-Klassendiagramm nicht genauso in Python umgesetzt werden. Daher werden die im UML-Klassendiagramm als privat gekennzeichneten Attribute bei der Umsetzung in Python mit einem Unterstrich als Präfix gekennzeichnet, wie es die allgemeine Konvention nach *PEP-8* vorsieht. Die Getter- und Setter-Funktionen zu den privaten

Attributen werden dabei nicht explizit aufgeführt, um die Abbildungen übersichtlicher zu halten. Zudem werden in den Klassendiagrammen bereits die Benennungskonventionen aus Python verwendet [58]. Bei den Namen der Klassen und Funktionen handelt es sich jedoch nur um Vorschläge, die bei der Implementierung im Anschluss abweichen können. Außerdem werden in diesem Kapitel zunächst nur die allgemeinen Funktionen und Attribute der Klassen aufgeführt. Die hardwarespezifischen Eigenschaften werden anschließend im Feinkonzept betrachtet.

4.2.1 Struktur der Hardwaretreiber

Im ersten Schritt der Konzeptionierung wurde die grobe Klassenstruktur der Treiber, unter Berücksichtigung der Kompatibilität mit dem *qupulse*-Back-End, entworfen. Die im Folgenden vorgestellte Struktur basiert auf der Schnittmenge verschiedener Arbiträrgeneratoren. Es ist also die allgemeine Struktur, die auf möglichst viele Arbiträrgeneratoren anwendbar ist, mindestens jedoch auf die in dieser Arbeit analysierten Instrumente. Die dabei entwickelten Klassen werden in der Umsetzung als Basisklassen für die jeweiligen Implementierungen der AWG-Treiber dienen, sodass sichergestellt ist, dass alle Treiber auf die gleiche Art und Weise aufgebaut sind.

Die Treiber bestehen aus der zentralen Klasse `Device`, welche die Kanäle vom Typ `Channel` und deren Synchronisation verwaltet. Die synchronisierten Kanalgruppen werden durch die Klasse `ChannelTuple` dargestellt. Diese sind für die Kommunikation mit dem *qupulse*-Back-End verantwortlich und beinhalten einen `ProgramManager` für die Verwaltung der hochgeladenen Pulssequenzen beziehungsweise Programme. Eine Übersicht über die interne Klassenstruktur eines Treibers gibt Abbildung 4.1.

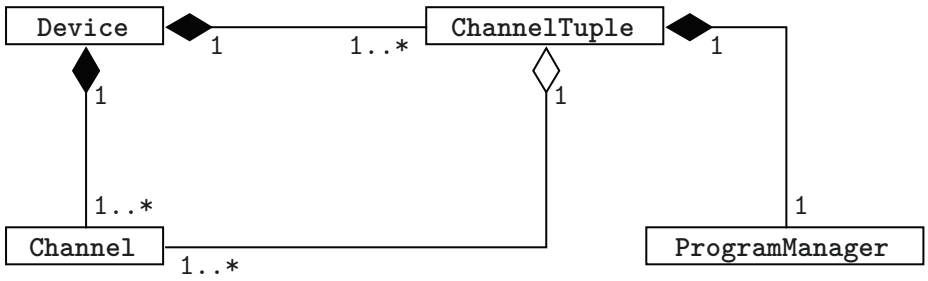


Abbildung 4.1: UML-Klassendiagramm der Struktur um die Klasse `Device`

Device

Die zentrale Klasse der Treiber wird hier `Device` genannt und dient als äußere Hülle des Treibers. Sie ist für die Initialisierung der Treiber verantwortlich und verwaltet die Verbindung mit dem Instrument. Dazu kommuniziert die Klasse direkt mit der Hardware. Dies wird in Abbildung 4.2 durch die Funktionen `send_cmd` und `send_query` verdeutlicht, mit denen sich Kommandos und Anfragen an das Instrument senden lassen. Neben der grundlegenden Kommunikation mit der Hardware ist die Klasse `Device` ebenfalls für die Verwaltung der Kanäle zuständig. Auch die Kanalsynchronisation findet in der `Device`-Klasse statt, wobei die einzelnen Kanäle vom Typ `Channel` der jeweiligen Kanalgruppe vom Typ `ChannelTuple` zugeordnet werden. Dies geschieht in der Funktion `synchronize_channels`. Eine detailliere Auseinandersetzung mit der Kanalsynchronisation folgt in Kapitel 4.3.1.

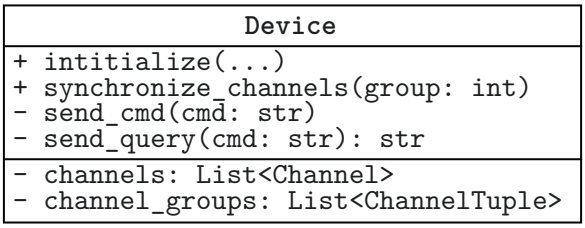


Abbildung 4.2: UML-Klassendiagramm der Klasse `Device`

Channel

Die Kanäle eines Treibers werden durch die Klasse `Channel` abgebildet, die in Abbildung 4.3 skizziert ist. Sie besitzen ebenfalls Methoden zur Kommunikation mit der Hardware, welche die Kommandos an das übergeordnete `Device`-Objekt weiterleiten. Dabei wird automatisch der richtige Kanal ausgewählt, was bei manchen Arbiträrgeneratoren über einen separaten Befehl erfolgen muss. Daher erhält der `Channel` eigene Funktionen, um die Kanalauswahl zentral zu steuern. Außerdem beinhaltet die `Channel`-Klasse Attribute zur Identifikation des Kanals, wie die Kanalnummer `channel_id`, eine Referenz auf das `Device`-Objekt sowie eine Referenz auf das `ChannelTuple`-Objekt und die Kanalnummer innerhalb dieser Kanalgruppe. Mit Hilfe dieser Attribute ist der Kanal immer genau identifizierbar und kann mit den übergeordneten Objekten verknüpft werden. Neben den im Klassendiagramm 4.3 aufgelisteten Attributen wird die `Channel`-Klasse einige, zum Teil gerätespezifische, Attribute erhalten. Dazu gehören beispielsweise die Sperre und Freigabe der Spannungsausgabe oder gewisse Parameter zur Manipulation

des ausgegebenen Spannungssignals, wie zum Beispiel ein Offset. Da diese Eigenschaften jedoch nicht unbedingt von allen Instrumenten unterstützt werden, sind diese nicht in der allgemeinen Struktur enthalten. Stattdessen werden diese Eigenschaften von der Treiberstruktur entkoppelt, was in Kapitel 4.2.2 thematisiert wird.

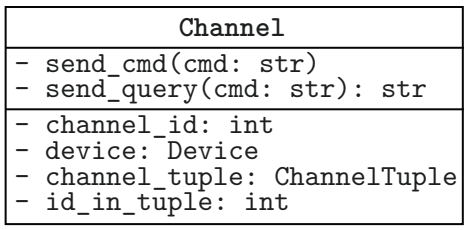


Abbildung 4.3: UML-Klassendiagramm der Klasse `Channel`

ChannelTuple

Ein wichtiger Bestandteil des Treibers ist das `ChannelTuple`, welches die Schnittstelle für das *qupulse*-Back-End bereitstellt. In Abbildung 4.4 ist die Klasse `ChannelTuple` mit der Schnittstelle `AWG` für das Back-End skizziert. Diese Schnittstelle existierte bereits in der alten Treiberstruktur und hat sich nicht geändert. Dadurch sind die Treiber direkt in das bestehende Framework integrierbar, ohne dass das Back-End in diesem Zuge angepasst werden muss. Damit die alte Schnittstelle mit den neuen Treibern kompatibel ist, wird der `ChannelTupleAdapter` verwendet.

Ein *Adapter*, auch *Wrapper* genannt, ist ein *Entwurfsmuster* der *Gang of Four* und wird zu den Strukturmustern gezählt. Dieses *Entwurfsmuster* hat den Zweck, eine Klasse an eine vorgegebene Schnittstelle anzupassen, indem der *Adapter* die Schnittstelle implementiert und dessen Aufrufe an die eigentliche Implementierung weiterleitet. So können Klassen mit inkompatiblen Schnittstellen zusammenarbeiten, indem der *Adapter* als Vermittler arbeitet [46]. Dies hat in diesem Fall den Vorteil, dass die neue Treiberstruktur nicht mehr von der Struktur des Back-Ends abhängt, aber dennoch mit diesem kompatibel ist. Sollten in Zukunft auch Umstrukturierungen am Back-End von *qupulse* vorgenommen werden, kann eine neue Schnittstelle definiert werden, die auf die neuen Treiber aufbaut. In dem Fall wird der *Adapter* nicht mehr benötigt werden oder kann ersetzt werden. Auf diese Weise sind die Treiber weniger stark mit dem Back-End verstrickt und die notwendigen Änderungen können in kleineren Schritten erfolgen, wodurch der Aufwand reduziert wird. Außerdem können Fehler bei der Integration der neuen Treiber vermieden werden und es ist sichergestellt, dass das Back-End weiterhin so funktioniert wie zuvor. Zur Realisierung des *Adapter-Patterns* existieren zwei Möglichkeiten: der

Klassenadapter und der *Objektadapter*. Ein *Klassenadapter* wird in Form einer Klasse umgesetzt, die sowohl die Schnittstelle implementiert als auch von der adaptierten Klasse abgeleitet ist. So wird nur ein Objekt benötigt und der *Klassenadapter* hat die Möglichkeit, manche Funktionen der adaptierten Klasse zu überschreiben. Jedoch bietet diese Realisierung den Nachteil, dass nur die adaptierte Klasse selbst mit der Schnittstelle verbunden wird, aber nicht deren Unterklassen. Da die `ChannelTuple`-Klasse aber nur eine abstrakte Basisklasse der jeweiligen Treiberimplementierungen sein soll, ist ein solcher *Klassenadapter* nicht geeignet [46].

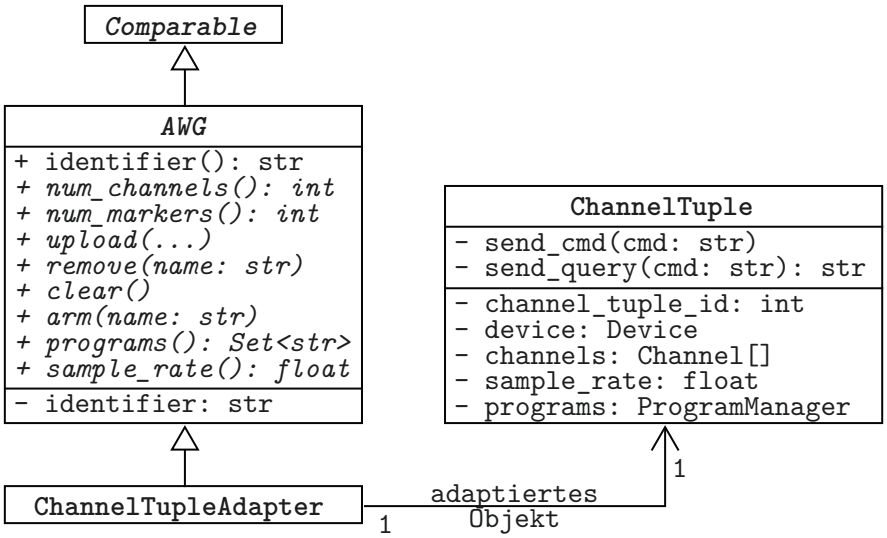


Abbildung 4.4: UML-Klassendiagramm der Klasse `ChannelTuple` mit der abstrakten Basisklasse `AWG`

Stattdessen wird ein *Objektadapter* verwendet. Dieser implementiert die Zielschnittstelle und beinhaltet die adaptierte Klasse als Objekt. So werden zwar zwei Objekte benötigt und die Funktionen der adaptierten Klasse sind nur bedingt überschreibbar. Jedoch muss nicht für jede Unterklasse ein eigener *Adapter* implementiert werden, was den Mehraufwand und die Code-Redundanz verringert und somit die Übersichtlichkeit steigert, denn ein *Objektadapter* kann sowohl mit der adaptierten Klasse selbst als auch mit allen Unterklassen arbeiten [46].

Die `AWG`-Schnittstelle ist das Interface zum *qupulse*-Back-End und enthält alle Treiber-Methoden, die von diesem aufgerufen werden. Der `identifier` wird lediglich für Fehlermeldungen und in Metadaten verwendet, um die Kanäle als Benutzer identifizieren zu können. Die Methoden `num_channels` und `num_markers` enthalten die Anzahl der Ausgabekanäle und der Markerkanäle, um prüfen zu können, ob genug Kanäle vorhanden sind, um eine bestimmte Pulssequenz abspielen zu können. Die Funktionen `upload`,

`remove`, `clear`, `arm` und `programs` dienen zur Verwaltung der Pulsprogramme auf der Hardware. Diese werden im neuen `ChannelTuple` vom `ProgramManager` verwaltet, um das *Single Responsibility*-Prinzip nicht zu verletzen. Dadurch wird die Verantwortung der Speicherverwaltung in die `ProgramManager`-Klasse ausgelagert [50]. Die `sample_rate` ist eine feste Eigenschaft des `ChannelTuples` und wird ebenfalls für die `AWG`-Schnittstelle benötigt, damit das Back-End die Pulssequenzen im richtigen Takt abtastet. Diese Eigenschaft wird von jedem Gerät unterstützt, da die Kanalgruppen über die Synchronisation des Takts definiert sind. Außerdem ist `AWG` von der `Comparable`-Schnittstelle abgeleitet, damit die existierenden `ChannelTuple`-Objekte unterscheidbar sind.

Das `ChannelTuple` selbst basiert auf den `channel_pairs`, die in den bisherigen Treibern aus *qupulse* verwendet wurden. Diese waren in Form von Attributen, wie zum Beispiel `channel_pair_AB` und `channel_pair_CD`, in die Treiber integriert. Dieses Konstrukt ist jedoch sehr starr und lässt keine beliebige Kanalsynchronisation zu. Die Kanäle aller drei Arbiträrgeneratoren konnten mit den bisherigen Treibern somit nur paarweise genutzt werden. Pulssequenzen mit drei oder mehr Ausgabekanälen waren gar nicht oder nur über Umwege umsetzbar. Ebenso, wie die `Channel`-Klasse, besitzt auch das `ChannelTuple` Funktionen zur Hardwarekommunikation, die die Kommandos an die `Device`-Klasse weiterleiten und eventuell zusätzliche Kommandos zur Auswahl des richtigen Kanals ausführen. Außerdem beinhaltet auch das `ChannelTuple` Attribute zur Identifikation des Objekts, wie die Kanalgruppennummer `channel_tuple_id`, eine Referenz auf das `Device` und das Tupel `channels`, welches die einzelnen gekoppelten Kanäle umfasst. Des Weiteren kann ein `ChannelTuple` weitere hardwarespezifische Eigenschaften enthalten, die jedoch separat abgehandelt werden.

ProgramManager

Zur Verwaltung der Pulssequenzen, die bereits auf den Arbiträrgenerator hochgeladen wurden, verwendet das `ChannelTuple` die Klasse `ProgramManager`, die in Abbildung 4.5 skizziert ist. Diese nimmt in der `add`-Funktion ein Objekt der Klasse `Program` entgegen. Diese Programme lassen sich anhand von Pulssequenzen vom Typ `Loop` erzeugen, welche bereits im Kapitel Grundlagen vorgestellt wurden. Die `Loops` werden aus `PulseTemplates` generiert, wobei die Parameter der Pulsvorlagen aufgelöst werden. Über die weiteren Methoden `get`, `remove` und `clear` lassen sich die gespeicherten Programme verwalten. Neben der Pulssequenz enthalten die Programme einen Namen zur Identifikation des Programms. Außerdem werden die verwendeten Ausgabe- und Markerkanäle aus der `Loop` extrahiert, um zu prüfen, ob die Pulssequenz mit dem `ChannelTuple` kompatibel ist. Die Kanäle und Marker werden anhand ihrer ID gespeichert. Durch einen Aufruf

der `get_sampled`-Funktion eines Programms wird eine Liste von Programmsegmenten erzeugt. Dabei wird die Pulssequenz mit der Abtastrate der Kanalgruppe gesampelt. Die abgetasteten Werte werden pro Kanal im `Segment`-Objekt gespeichert, wobei jedes Segment einem Abtastpunkt des arbiträren Spannungssignals entspricht. Beim Hochladen auf das Gerät wird über die einzelnen Segmente iteriert. Dabei können die Segmentdaten mit der Funktion `get_binary_data` für die Hardware aufbereitet werden. Die resultierenden Binärdaten können direkt in den Speicher des Geräts geschrieben werden.

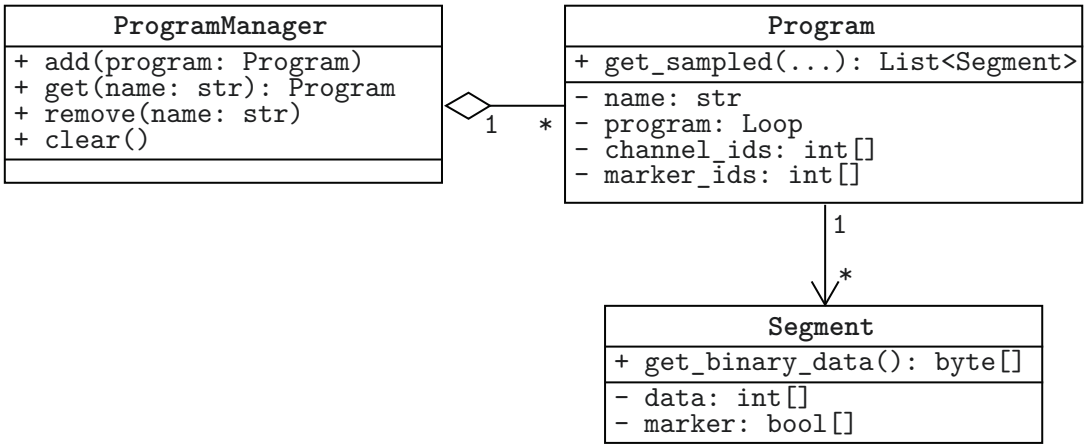


Abbildung 4.5: UML-Klassendiagramm der Klasse `ProgramManager`

4.2.2 Abstraktion der hardwarespezifischen Funktionalität

Die Umsetzung hardwarespezifischer Funktionen ist essenziell für ein flexibles und übersichtliches Treibersystem. Jeder der Arbiträrgeneratoren unterstützt unterschiedliche Merkmale und Funktionen, wobei es, besonders bei zunehmender Anzahl an Treibern, zu Überschneidungen kommen kann. Damit die verschiedenen Treiber einheitlich bedient werden können, müssen diese spezifischen Funktionen generalisiert werden. Wenn dementsprechend zwei oder mehr Geräte dieselbe Funktion unterstützen, muss diese auch auf die gleiche Weise umgesetzt werden. Dies bedeutet konkret, dass die Funktionssignatur, also der Funktionsname und die Typen der Übergabeparameter und der Rückgabe, einheitlich sein muss. Normalerweise wird dies mit Hilfe von Vererbung gelöst, was in diesem Fall jedoch nicht möglich ist, da je nach Umsetzung gewisse Richtlinien aus dem Softwaredesign verletzt werden. Zum Beispiel unterstützen der *Tabor WX2184C* und der *Zurich Instruments HDAWG8* die Synchronisation mehrerer Geräte. Zur Standardisierung könnte beispielsweise eine `synchronize`-Funktion in der Basisklasse deklariert werden, wie in Abbildung 4.6 skizziert. Dadurch ist sichergestellt, dass beide Geräte

die Funktion auf dieselbe Weise in ihrer Schnittstelle bereitstellen. Kommt jedoch der *Tektronix AWG5014C* hinzu, muss dieser die Funktion ebenfalls implementieren, obwohl sie nicht unterstützt wird. Somit müsste dieser eine Exception werfen, wenn die nicht unterstützte Funktion aufgerufen wird. An diesem Punkt wird jedoch das *Liskovsche Substitionsprinzip* verletzt, denn eine Klasse darf die Funktionalität ihrer Basisklasse nur erweitern und nicht einschränken [53].

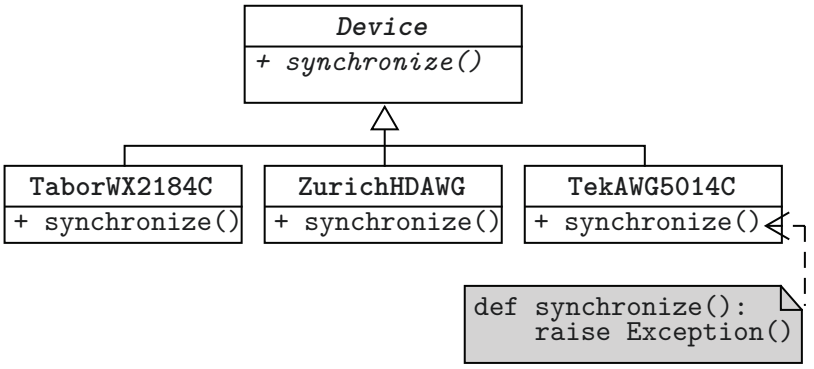


Abbildung 4.6: UML-Klassendiagramm zur Standardisierung der `synchronize`-Funktion in der Basisklasse

Ein weiterer Ansatz wäre eine Realisierung mit Interfaces. Obwohl diese in Python nicht direkt unterstützt werden, können aufgrund der Mehrfachvererbung abstrakte Basisklassen dazu verwendet werden. Dabei würde für jede Funktionalität ein Interface erstellt werden, mit Methoden, die diese Funktionalität bereitstellen. Alle Geräte, die diese Funktionalität unterstützen, können das Interface implementieren und es ist sichergestellt, dass alles einheitlich aufrufbar ist. Somit würde die `synchronize`-Funktion nicht in der Basisklasse der Treiber deklariert werden, sondern in einem separaten Interface, wie in Abbildung 4.7 zu sehen. Dadurch muss nicht auf die Implementierung hin programmiert werden, sondern es kann allein mit den Schnittstellen gearbeitet werden, wie es bereits von der *Gang of Four* gefordert wurde. Dies verhindert Abhängigkeiten von den konkreten Treiber-Implementierungen [46]. Aber auch dieser Ansatz hat zwei Nachteile: Zum einen sollte Vererbung, soweit möglich, vermieden werden, da das Konstrukt der Vererbung eher starr und unflexibel ist. Zum anderen verliert das System bei zu vielen Interfaces an Übersichtlichkeit. Dennoch ist dieser Ansatz vielversprechend und kann in einer optimierten Form umgesetzt werden.

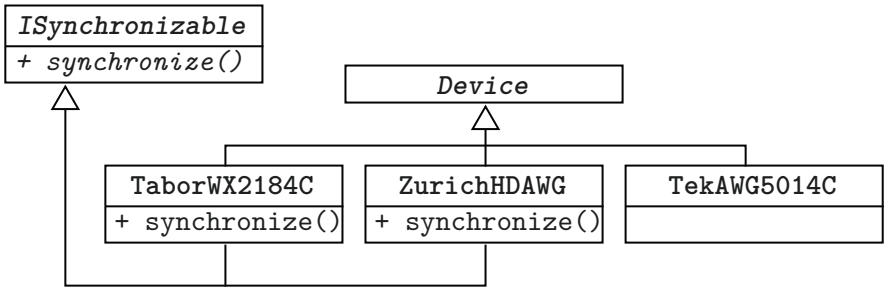


Abbildung 4.7: UML-Klassendiagramm zur Standardisierung der `synchronize`-Funktion in einem *Interface*

Der finale Entwurf zur Standardisierung der hardwarespezifischen Funktionen orientiert sich an der Prämisse der *Gang of Four*: „Ziehe Objektkomposition der Klassenvererbung vor“ [46, S. 25]. Zur Vermeidung starrer Vererbungsstrukturen um die Treiber herum, wurde daher das *Bridge-Pattern* als *Entwurfsmuster* hinzugezogen. Es wurde von der *Gang of Four* als Strukturmuster definiert und dient der Trennung von Implementierung und Abstraktion. Anstatt also die Implementierung direkt von der Abstraktion der Funktionalität abhängig zu machen, zum Beispiel durch Interfaces, werden beide Strukturen voneinander entkoppelt [46].

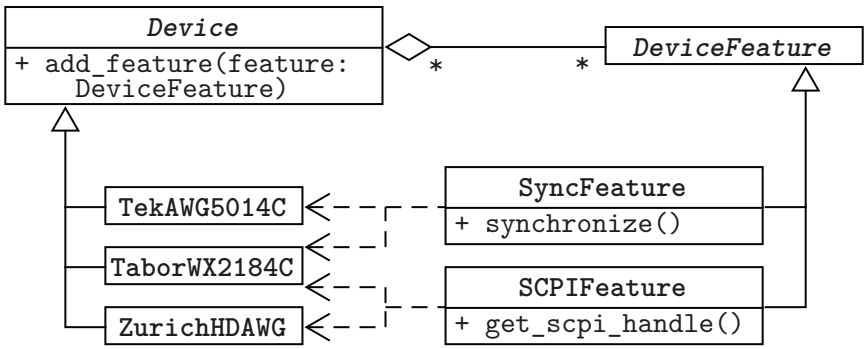


Abbildung 4.8: UML-Klassendiagramm zur Standardisierung hardwarespezifischer Funktionen in Anlehnung an das *Bridge-Pattern*

Die Trennung von Abstraktion und Implementierung ist in Abbildung 4.8 dargestellt. Auf der linken Seite befinden sich die drei AWG-Treiber, die von der Basisklasse `Device` abgeleitet sind. Davon getrennt befindet sich auf der rechten Seite die Abstraktion der Funktionalität, die hier als `Feature` bezeichnet wird. Die `Features` werden in einer Vererbungshierarchie strukturiert, wobei hier nur die `Features` der `Device`-Klassen dargestellt sind. Parallel dazu werden auch `Features` für `Channels` und `ChannelTuples` bereitgestellt, denn die diversen Funktionen können sich sowohl auf das gesamte Gerät

beziehen als auch auf einzelne oder gekoppelte Kanäle. Exemplarisch werden hier jedoch nur die `Device`-Klassen betrachtet. Als exemplarisches `DeviceFeature` wird neben der Gerätesynchronisation auch die *SCPI*-Fähigkeit betrachtet. Geräte, die *SCPI* unterstützen, könnten demnach beispielsweise ein `scpi_handle` besitzen, mit welchem direkt auf die Hardware zugegriffen werden kann, um diese mit *SCPI*-Befehlen zu steuern. Die Gerätesynchronisation wird vom *Tabor WX2184C* und vom *Zurich Instruments HDAWG8* unterstützt. *SCPI* wird hingegen zur Programmierung des *Tektronix AWG5014C* sowie des *Tabor WX2184C* verwendet. Welches `Feature` von welchem Treiber unterstützt wird, ist in Abbildung 4.8 durch die gestrichelten Pfeilen dargestellt. Somit ergeben sich zwei `Features`, die von unterschiedlichen Instrumenten bereitgestellt werden, wobei der *Tabor WX2184C* beide unterstützt. Um diese Gegebenheit zu realisieren, werden die `Features` nicht in Form von Interfaces auf die Geräte verteilt, sondern per Objektkomposition in die `Device`-Objekte integriert. Dazu besitzt die Klasse `Device` die Methode `add_feature` für das Hinzufügen eines `Feature`-Objekts. Zur Übersichtlichkeit lassen sich die bereitgestellten `Features` eines Treibers über eine Liste einsehen. Beim Hinzufügen einer neuen Funktion wird zum einen das `Feature`-Objekt in diese Liste eingefügt und zum anderen werden die bereitgestellten Funktionen des `Features` dem Treiber-Objekt dynamisch hinzugefügt. Dies erlaubt die Dynamik von Python und kann mit Hilfe der im Standard enthaltenen Methode `setattr` realisiert werden. Dadurch muss nicht erst das richtige `Feature`-Objekt in der Liste gefunden werden, um die Funktion aufzurufen, sondern die Funktion kann direkt vom Treiberobjekt aus aufgerufen werden. Daraus ergibt sich dasselbe Bild, wie bei einer Lösung mit Interfaces. Dies stellt außerdem sicher, dass der Aufruf für alle AWG-Treiber einheitlich ist.

Bisher blieb die Implementierung der `Feature`-Funktionen unerwähnt. Da die `Feature`-Objekte nicht direkt mit der Hardware kommunizieren können, muss die Implementierung an anderer Stelle erfolgen. Dazu existieren zwei Möglichkeiten. Die erste Möglichkeit ist, dass dem `Feature`-Objekt bei der Konstruktion ein Funktionsobjekt übergeben wird, mit welchem der Funktionsaufruf an das `Device`-Objekt umgeleitet wird. Dabei wird das Prinzip der *Dependency Injection* verwendet, indem die Abhängigkeiten von außen in das `Feature`-Objekt „injiziert“ werden. Die *Dependency Injection* leitet sich aus dem Prinzip der *Inversion of Control* ab, also der Umkehrung des Kontrollflusses. Beide Prinzipien haben zum Ziel, Abhängigkeiten zwischen verschiedenen Klassen und Schnittstellen zu reduzieren und somit die Wartbarkeit und Flexibilität der Software zu steigern [60]. In diesem Fall dient das bei der Konstruktion des `Features` übergebene Funktionsobjekt als „injizierte“ Abhängigkeit. Diese Funktion ließe sich entweder als eine private Funktion des Treibers oder eine anonyme Lambda-Funktion realisieren. Das resultierende Funktionsobjekt kann somit vom Treiber aus an die `Feature`-Schnittstelle

weitergegeben werden, um die gewünschte Funktionalität zu „injizieren“. Dies könnte beispielsweise wie in Programmcode 4.2 realisiert werden. Diese Realisierungsmöglichkeit hat den Vorteil, dass die Funktion auf alle Ressourcen des Treiber-Objekts zugreifen kann, um beispielsweise mit der Hardware kommunizieren zu können.

```
class SyncFeature(DeviceFeature):
    def __init__(self, sync_function: Callable):
        self._sync_function = sync_function

    def synchronize(self):
        return self._sync_function()

class ZurichHDAWG(Device):
    def __init__(self):
        self.add_feature(SyncFeature(self._synchronize))
        # ...

    def _synchronize(self):
        # do something
        pass
```

Programmcode 4.2: `Feature`-Implementierung mit einem Funktionsobjekt

Die zweite Möglichkeit zur Platzierung der Implementierung, ist die Verwendung des *Adapter-Patterns*, welches bereits beim `ChannelTuple` zum Einsatz kommt. Dabei wird für jeden Treiber, der ein `Feature` unterstützt, ein *Adapter* angelegt, welcher für die Implementierung des Features zuständig ist. Die `Feature`-Klasse wird in dem Fall zu einer abstrakten Basisklasse für die *Adapter*, welche die abstrakten Funktionen des `Features` überschreiben. Die *Adapter*-Klassen können beispielsweise als innere Klassen der jeweiligen Treiber implementiert werden, sodass diese ebenfalls Teil des Treibers sind. Dies bietet den Vorteil, dass die Funktionen im Programmcode dort gefunden werden, wo sie erwartet werden. Außerdem sind die Funktionen nach ihren `Features` gegliedert, wodurch eine gewisse Struktur in den Treibern hergestellt wird. In Programmcode 4.3 befindet sich eine exemplarische Implementierung eines solchen *Adapters*.

```
class SyncFeature(DeviceFeature):
    @abstractmethod
    def synchronize(self):
        pass

```

```
class ZurichHDAWG(Device):
    def __init__(self):
        self.add_feature(self.SyncFeatureAdapter())
        # ...

    class SyncFeatureAdapter(SyncFeature):
        def synchronize(self, parent: ZurichHDAWG):
            # do something
            pass
```

Programmcode 4.3: `Feature`-Implementierung mit dem Adapter-Pattern

Die Verwendung eines *Adapters* hat jedoch den Nachteil, dass der *Adapter* nicht direkt mit der Hardware kommunizieren kann. Dazu wird weiterhin das `Device`-Objekt benötigt, welches in Programmcode 4.3 in Form des Parameters `parent` übergeben werden muss. Dies führt zu starken gegenseitigen Abhängigkeiten, die das *Dependency Inversion*-Prinzip verletzen, welches starke Bindungen zwischen Klassen verbietet, denn dies würde wiederum die Wartbarkeit und Flexibilität dieser Entitäten einschränken [54]. Außerdem widerspricht dies der Funktionsweise eines *Adapters*, da ein solcher als Vermittler zwischen zwei unabhängigen Schnittstellen dient. Dabei werden die Schnittstellenfunktionen an die Implementierung weitergeleitet, wobei kleine Änderungen bei den Funktionsaufrufen möglich sind [46]. Wenn der `Device`-Klasse dazu jedoch neue Funktionen hinzugefügt werden müssen, die der *Adapter* aufruft, können diese Funktionen direkt an die Schnittstelle angepasst werden, wodurch kein *Adapter* mehr notwendig wäre. Aus diesem Grund empfiehlt es sich für die AWG-Treiber, die Realisierung mit einem Funktionsobjekt vorzuziehen. So werden keine Prinzipien und Konventionen der Softwareentwicklung verletzt, was auch die Softwarequalität verbessert. Außerdem kann durch Ausnutzung von *Dependency Injection* gezielt eine hohe Wartbarkeit hergestellt werden, wodurch der Entwicklungsaufwand reduziert wird.

Durch die vorgegebene Treiberstruktur und die Generalisierung der hardwarespezifischen Funktionen mit dem *Bridge-Pattern* kann die Konfiguration der Instrumente von den eigentlichen Treibern entkoppelt werden. Dabei sind die Treiber zwar nicht komplett abstrahiert, dies ist jedoch für die praktische Anwendung nicht erforderlich. Da bei jedem Experiment zuvor die Geräte angeschlossen werden, ist ohnehin bekannt, welcher Treiber benötigt wird. Außerdem werden bei komplexen Experimenten zum Teil spezifische Funktionen einzelner Instrumente benötigt, die durch eine totale Abstraktion nicht verfügbar wären.

Mit der dazugewonnenen Hardwareabstraktionsschicht können die in Experimenten verwendeten Arbiträrgeneratoren leicht ausgetauscht werden, sofern sie den erforderlichen Funktionsumfang unterstützen. Durch die einheitliche Treiberstruktur und die Generalisierung der hardwarespezifischen Funktionen ist sichergestellt, dass die Instrumente die gemeinsamen Funktionen auf die gleiche Weise bereitstellen. Dadurch muss bei einem Austausch eines Arbiträrgenerators lediglich der Verbindungsaufbau angepasst werden, da die Verbindung über eine andere Schnittstelle läuft und der Zugriffspunkt eventuell abweicht.

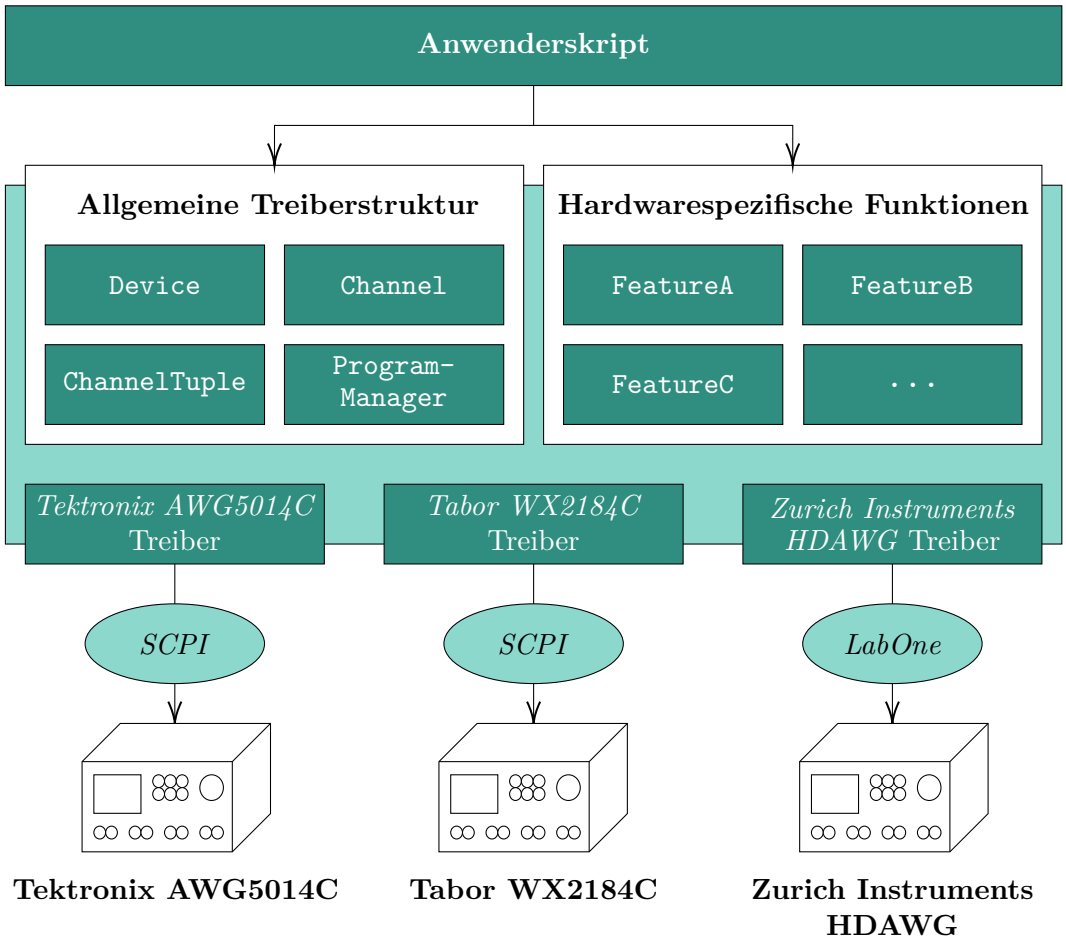


Abbildung 4.9: Abstraktionsschicht der AWG-Treiber

Das in Abbildung 4.9 dargestellte Modell zeigt die Abstraktionsschicht, welche auf die jeweiligen Treiber aufbaut und diese einen standardisierten Rahmen vorgibt. Die Anwender arbeiten zwar immer noch direkt mit den einzelnen AWG-Treibern, deren Funktionen

können jedoch unabhängig von den Treibern genutzt werden, da diese von den vereinheitlichten `Features` bereitgestellt werden. Abgesehen von der Erstellung des Treiberobjekts, bei welcher der Verbindungsaufbau stattfindet, ist die gesamte Funktionalität der Instrumente durch die Treiberstruktur und die `Features` abgebildet. Ist einmal die Verbindung zum Arbiträrgenerator hergestellt, kann der Anwender die Geräte anhand der verfügbaren Funktionalität in den `Feature`-Objekten bedienen. So lässt sich ein Arbiträrgenerator im Versuchsaufbau leicht durch einen anderen austauschen, der ebenfalls den benötigten Funktionsumfang bietet.

Ein charakteristisches Merkmal von Python ist die Möglichkeit des *Duck-Typings*. Dabei handelt es sich um ein Konzept, bei welchem der Typ eines Objekts nicht allein durch die Klasse definiert wird, sondern durch die bereitgestellten Funktionen und Attribute eines Objekts. Ein Großteil der Funktionen und Attribute kommen in der Regel zwar von der Klasse eines Objekts, doch es können bei Python dynamisch neue Eigenschaften hinzugefügt werden, wie bei den vorgestellten `Features`. Daher ist es in Python unüblich, Interfaces zu definieren. Stattdessen wird die Schnittstelle eines Objekts anhand der vorhandenen Funktionen und Attribute beschrieben. Beim *Duck-Typing* wird diese Eigenschaft ausgenutzt. Dabei wird der sogenannte *Ententest* angewendet, welcher besagt: „Wenn es aussieht wie eine Ente und quakt wie eine Ente, dann ist es eine Ente!“ [56]. Das heißt, wenn ein Objekt eine bekannte Funktion bereitstellt, kann diese auch so aufgerufen werden, wie es erwartet wird. Diese Eigenschaft verleiht Python ein hohes Maß an Dynamik und Flexibilität, da die Schnittstelle durch die Funktionsaufrufe des Anwenders bestimmt wird. Daher ist es wichtig, dass die gemeinsame Funktionalität zwischen verschiedenen Geräten durch dieselben Methoden repräsentiert werden, wie es mit Hilfe der `Feature`-Struktur sichergestellt wird.

4.3 Feinkonzept

Nach dem strukturellen Entwurf im Grobkonzept folgt die Verfeinerung dieses Konzepts. Zunächst werden dazu die in der Hardwareanalyse herausgestellten Eigenschaften und Funktionen in das Konzept eingearbeitet. Anschließend folgt ein Ansatz zur Vereinfachung der Treiberkonfiguration. Dabei soll der Aufwand zur Konfiguration des Treibers für eine Messung reduziert werden, um die Vorbereitungszeit eines Experiments zu verringern. Zum Schluss wird die Anbindung von Tests diskutiert, die der Validierung der Software und der frühzeitigen Erkennung von Fehlern dienen soll.

4.3.1 Kernfunktionen und Eigenschaften der Hardware

Nachfolgend werden die Kernfunktionen und Eigenschaften der Arbiträrgeneratoren in das Treiber-Konzept eingebunden. Diese werden dabei unter anderem in die Treiberstruktur integriert, über die `Feature`-Struktur angebunden oder im *qupulse*-Back-End eingebaut. Des Weiteren werden Optimierungen konzipiert, um von der Leistungsfähigkeit jedes einzelnen Geräts zu profitieren.

Spezifikationen

Die verschiedenen Spezifikationen und Eigenschaften der Arbiträrgeneratoren spiegeln sich an unterschiedlichen Stellen der Treiber wieder. Die relevanten Eigenschaften der Geräte sind unter anderem die Anzahl der Kanäle, die Amplitude, die Wortbreite und die Abtastrate sowie die Kapazität des Waveformspeichers, welche jedoch erst im Kontext der Speicherverwaltung thematisiert wird.

Die Realisierung der Kanäle wird direkt über die Struktur der AWG-Treiber abgebildet. Jeder Treiber vom Typ `Device` enthält ein Array von Kanälen der Klasse `Channel`. Diese Kanäle werden dynamisch bei der Konstruktion des Treibers angelegt und können je nach Gerät variieren. Die Kanäle lassen sich in vorgegebenen Szenarios synchronisieren, was weiter unten noch genauer thematisiert wird. Die Kanäle selbst besitzen feste Eigenschaften zur Identifikation und zur Kommunikation mit der Hardware. Dazu gehören die Kanalnummer, die Zugehörigkeit zu einer synchronisierten Kanalgruppe und ein Verweis auf das übergeordnete `Device`-Objekt. So ist jeder Kanal innerhalb eines Geräts, sowie innerhalb einer Kanalgruppe, eindeutig identifizierbar. Außerdem kann jeder Kanal über das `Device`-Objekt mit der Hardware kommunizieren. Neben diesen Attributen wird das *Bridge Pattern* auch für Kanäle angeboten, sodass kanalspezifische Eigenschaften dort abbildbar sind. Dazu zählen beispielsweise Eigenschaften, wie die (De-)Aktivierung des Kanalausgangs sowie die Amplitude und das Offset der Spannungsausgabe. Das Freigeben und Sperren eines Kanals sowie die Einstellung von Amplitude und Offset werden von allen drei in der Hardwareanalyse diskutierten Geräten unterstützt. Durch das Sperren eines Kanals lässt sich die Spannungsausgabe für einen Kanal gänzlich deaktivieren. Dies wäre beispielsweise durch ein `EnableChannelFeature` mit den Funktionen `enable(bool)` und `is_enabled()` realisierbar. Dadurch kann die Spannungsausgabe eines Kanals aktiviert und deaktiviert werden und der aktuelle Aktivierungszustand lässt sich abfragen. Weitere Kanaleinstellungen sind das Offset und die Amplitude, die zur Manipulation des Spannungssignals und der Optimierung der Auflösung dienen. Mit dem Offset ist es möglich, das gesamte Spannungssignal in der Spannungshöhe verschieben

und mit der Amplitude wird die maximale Ausdehnung des Spannungssignals gesteuert, sodass die Amplitude zum Strecken und Stauchen der Spannung eingesetzt werden kann. Bei gleichbleibender Wortbreite lässt sich die vertikale Auflösung des Pulses durch die optimale Wahl von Offset und Amplitude verbessern. Auf diese Weise kann die volle Wortbreite auf den benötigten Spannungsbereich der Pulssequenz angewendet werden, um somit ein Spannungssignal von möglichst feiner Granularität zu erhalten. So lassen sich qualitativ hochwertigere Signale erzeugen, als bei der standardmäßigen Verwendung der maximalen Amplitude und einem Offset von 0.

Eine weitere Eigenschaft ist die Abtastrate, die den synchronisierten Kanalgruppen zugeordnet wird. Diese gibt die horizontale Auflösung einer Pulssequenz an und bestimmt somit auch die Abspielgeschwindigkeit der arbiträren Spannungswerte. Anhand der Abtastrate bestimmt das Back-End, mit welcher Geschwindigkeit die Pulssequenz für das Hochladen auf das Gerät abgetastet werden muss, sodass der Arbiträrgenerator das Signal anschließend mit derselben Geschwindigkeit abspielen kann. Je höher die Abtastrate eingestellt wird, desto feiner ist die zeitliche Auflösung des Signals und desto höhere Frequenzen lassen sich erzeugen. Damit die Pulssequenzen vom *qupulse*-Back-End abgetastet werden können, um einen arbiträren Spannungsverlauf zu erhalten, wurde die `sample_rate` auch bereits im `AWG`-Interface definiert.

Kommunikation und Programmierung

Die Kommunikation mit der Hardware und dessen Programmierung erfolgt direkt in den Treiberimplementierungen. Je nach Modell muss hierbei anders vorgegangen werden. Daher erfolgt bereits der Verbindungsaufbau im Konstruktor der einzelnen Implementierungen der Treiber. Der *Tabor WX2184C* und der *Tektronix AWG5014C* nutzen *VISA* als Kommunikationsprotokoll. Dieses ermöglicht den Austausch von Textbefehlen zwischen Treiber und Hardware. Auch der *Zurich Instruments HDAWG8* nutzt mit dem *LabOne*-Framework ebenfalls eine textbasierte Kommunikation. Daher wurden im Grobkonzept bereits die Funktionen `send_cmd` und `send_query` vorgesehen. Mit diesen Funktionen lassen sich *String*-Kommandos an die Hardware senden und die `send_query`-Funktion erwartet zusätzlich eine Antwort vom Gerät, ebenfalls in Form eines *Strings*. Trotz der einheitlichen Schnittstelle, werden die übergebenen Befehle im Hintergrund auf unterschiedliche Arten verarbeitet, also je nach Instrument per *VISA*, *LabOne* oder anderen Schnittstellen und Protokollen.

Doch nicht nur die Art der Übertragung unterscheidet sich zwischen den Instrumenten, sondern auch die Befehle selbst. Die beiden vorgestellten Geräte von *Tabor* und *Tektronix* verwenden *SCPI*-Befehle und das Gerät von *Zurich Instruments* nutzt das

LabOne-Framework, welches neben der Übertragungsschnittstelle auch einen Befehlssatz bereitstellt. Für jedes `Feature`, das von einem Treiber unterstützt wird, werden pro Gerät unterschiedliche Befehle oder Befehlsfolgen ausgeführt. Dies gilt sogar für die beiden *SCPI*-fähigen Geräte, da *SCPI* nur die Syntax und die Struktur der Befehle vorgibt. Direkt mm *SCPI*-Standard sind nur wenige Befehle definiert [43]. Der Großteil der Befehle unterscheidet sich von Hersteller zu Hersteller und zum Teil auch zwischen den Modellen. Daher findet sich die Realisierung der Programmierung und Kommunikation auf unterster Ebene in den konzipierten AWG-Treiber, sodass diese Heterogenität dem Anwender verborgen bleibt.

Zur Erhöhung der Funktionalität der Treiber, vor allem in der anfänglichen Entwicklungsphase, kann den Benutzern auch eine Möglichkeit angeboten werden, um mit Hilfe der Treiber direkt mit der Hardware zu kommunizieren. So könnten die Funktionen zum Senden der Kommandos öffentlich angeboten werden, sodass die Benutzer in der Anfangszeit auch Funktionen der Instrumente nutzen können, die zu diesem Zeitpunkt noch nicht von den neuen AWG-Treibern abgebildet werden. Dabei ist anzumerken, dass dies nur zu Übergangszwecken dient, bis die Treiber ausreichend Funktionalität in abstrahierter Form bereitstellen. Sobald sich die Treiber in einem fortgeschrittenen bis finalen Zustand befinden, wird die direkte Kommunikation mit der Hardware nicht mehr von den Anwendern benötigt werden. Daher wird die direkte Hardwarekommunikation langfristig nur innerhalb der Treiberimplementierung stattfinden.

Zur Optimierung der direkten Kommunikation mit der Hardware können beispielsweise auch *SCPI*-Objekte bereitgestellt werden, die die gesamte Befehlsstruktur eines Geräts abbilden. Dadurch ist es nicht mehr erforderlich, die Befehle als *String* einzutippen, da es dabei schnell zu Fehlern kommen kann. So könnte beispielsweise der Befehl zur Erzeugung eines Sinussignals, wie bereits in Programmcode 3.1 auf Seite 33 beschrieben, auf zwei Arten programmiert werden. Die einfachste Variante wäre per `send_cmd`. Dabei wird der Befehl per *String* übergeben und an das Gerät gesendet, wie in der ersten Zeile von Programmcode 4.4. Bei der Verwendung eines Objekts, das die *SCPI*-Befehlsstruktur abbildet, könnte der Aufruf gemäß der zweiten Zeile ablaufen. Dazu könnte die im Grobkonzept als Beispiel eingeführte Funktion `get_scpi_handle` verwendet werden.

```
1 awg.send_cmd(":SOURCE:FUNCTION:SHAPE SIN")
2 awg.get_scpi_handle().source().function().shape("sin")
```

Programmcode 4.4: Objektorientierte Repräsentation von *SCPI*-Kommandos

Die Erstellung einer solchen Objektstruktur zur Abbildung der Kommandos, erleichtert einerseits die Bedienung des Geräts und stellt den Anwendern bereits von Beginn an den vollen Funktionsumfang bereit. Andererseits können auf diese Weise Fehler in

der Kommunikation verhindert werden, da Tippfehler bei der Ausführung sofort vom Python-Interpreter bemerkt werden. Wird jedoch ein fehlerhafter *String*-Befehl an das Gerät gesendet, kann es je nach Gerät vorkommen, dass der Fehler unbemerkt auftritt. Doch eine Forderung des *Zen of Python* lautet: „Errors should never pass silently“ (siehe Programmcode 4.1, Zeile 13). Damit solche Fehler nicht unbemerkt auftreten, sollten *String*-Kommandos vermieden werden oder nur an zentraler Stelle eingesetzt werden, wo diese auch zentral getestet werden können.

Intern sendet jede Funktion aus dieser Objektstruktur den entsprechenden *String*-Befehl an das Gerät, woraufhin die Antworten des Geräts interpretiert und für den Benutzer aufbereitet werden können. Dadurch muss jeder Befehl nur einmal in Textform implementiert werden. Alle weiteren Aufrufe des Befehls können fortan über die objektorientierte Schnittstelle laufen.

Ähnliches lässt sich für das *LabOne*-Framework bereitstellen. Dieses bietet zwar bereits einige objektorientierte Funktionen an, ein Großteil der Funktionalität wird jedoch auch hierbei per *String*-Kommandos realisiert. Auch diese lassen sich objektorientiert abbilden, um Fehler zu vermeiden und dem Benutzer mehr Funktionalität bereitzustellen. Besonders für die anfängliche Entwicklungszeit des neuen Treibersystems bietet sich ein solches Vorgehen an, da nicht von Beginn an die gesamte Funktionalität durch die Treiber gekapselt werden kann. Aber auch innerhalb der Treiber kann diese Form der Abstraktion die Fehleranfälligkeit senken und zu mehr Übersichtlichkeit führen.

Synchronisation von Kanälen und Geräten

Eine weitere Kernfunktion der Arbiträrgeneratoren ist die Möglichkeit zur Synchronisierung mehrerer Kanäle und zum Teil auch ganzer Geräte. Die Grundfunktionalität der Kanalsynchronisation ist bereits in der Treiberstruktur erkennbar. Jedes Gerät besitzt eine gewisse Anzahl an Kanälen vom Typ `Channel`, welche sich in Kanaltupeln vom Typ `ChannelTuple` synchronisieren lassen. Da es sich bei der Kanalsynchronisation um eine essentielle Kerneigenschaft der Arbiträrgeneratoren handelt, ist die Treiberstruktur daran ausgerichtet. Je nachdem, wie die Kanäle miteinander gekoppelt sind, setzt sich ein Treiber aus unterschiedlichen Kombinationen von Kanälen und Kanaltupeln zusammen. Bei den vorgestellten Geräten lassen sich die Kanäle in Gruppen von mindestens zwei Kanälen synchronisieren. Die Synchronisation von vier Kanälen ist ebenfalls möglich, sowie auch die Synchronisation von acht Kanälen. Dies unterstützt jedoch nur der *Zurich Instruments HDAWG8*, aufgrund seiner höheren Anzahl an Kanälen. Anhand der konfigurierten Kanalsynchronisation ergeben sich die Anzahl an Kanaltupeln und die

darin enthaltenen Kanäle. Alle Kanaltupel nutzen dabei einen einzigen Taktgeber oder mehrere synchronisierte Taktgeber, sodass diese gekoppelten Kanäle als eigenständiges Instrument betrachtet werden können. Während die Kanalstruktur fest in die Treiberstruktur integriert ist, wird die Konfiguration der Kanalsynchronisation per `Feature` angebunden, wie auch sämtliche andere Konfigurationsmöglichkeiten. Dadurch lassen sich zukünftig auch Treiber für neue Geräte anbinden, die eventuell keine dynamische Kanalsynchronisation unterstützen.

Werden mehr als vier beziehungsweise acht synchronisierte Kanäle benötigt, können auch Arbiträrgeneratoren miteinander synchronisiert werden. Dies ist zum Beispiel mit dem *Tabor WX2184C* oder dem *Zurich Instruments HDAWG8* möglich. Dazu kann die in der Grobkonzeptionierung exemplarisch eingeführte Funktion `synchronize` verwendet werden. Nachdem die Gerätesynchronisation konfiguriert wurde, lassen sich die synchronisierten Geräte mit nur einem Treiberobjekt steuern, das die Kanäle aller synchronisierten Geräte beinhaltet.

Speicherverwaltung und Sequenzierung der Waveforms

Eines der Ziele dieser Arbeit ist die Leistungsoptimierung der Geräte. Dazu sind die Speicherverwaltung und der Sequenzierungsmechanismus besonders wichtig. Durch eine optimierte Speicherverwaltung und eine effiziente Sequenzierung der Waveforms, lässt sich viel Speicherplatz sparen, sodass mehr Waveforms oder längere und komplexere Waveforms von den Instrumenten gespeichert werden können. Außerdem lässt sich durch den reduzierten Datenverbrauch auch die Übertragungszeit zum Hochladen der Waveforms verringern.

Die Waveformspeicher der drei vorgestellten Geräte sind sehr ähnlich zueinander aufgebaut. Die Speicherkapazität von 16 Megasample, beziehungsweise 64 Megasample beim *Zurich Instruments HDAWG8*, ist pro Kanal angegeben. Ein Sample entspricht dabei einem Abtastpunkt, also einem diskreten Spannungswert der Waveform. Der Waveformspeicher kann als eine Tabelle mit 16 (beziehungsweise 64) Millionen Zeilen und jeweils einer Spalte für jeden Kanal beschrieben werden. Jede Zelle dieser Tabelle gibt somit das Spannungsniveau pro Kanal und pro Abtastpunkt an. Dabei wird ein Spannungsniveau mit einer Wortbreite von 14 Bit (16 Bit beim *HDAWG8*) gespeichert, wobei gilt: Je höher die Wortbreite ist, desto genauer lassen sich die Spannungssignale definieren. Beim Abspielen der Waveform iteriert der Arbiträrgenerator über diese Tabelle und erzeugt die darin gespeicherten Spannungen, um daraus ein arbiträres Signal zu erhalten.

In welcher Reihenfolge die Waveforms aus dem Speicher abgespielt werden, lässt sich ebenfalls einstellen. Im einfachsten Modus werden die Zeilen der Speichertabelle nacheinander durchgegangen. Um dies zu optimieren, kann auf einen Sequenzierungsmechanismus zurückgegriffen werden. Dieser unterscheidet sich zwischen den Geräten in seiner Ausprägung und Komplexität. So unterstützt der *Tektronix AWG5014C* nur eine einstufige und der *Tabor WX2184C* eine zweistufige Sequenzierung. Die Sequenzierungstiefe des *Zurich Instruments HDAWG8* ist sogar quasi unbegrenzt; lediglich der verfügbare Speicherplatz schränkt bei diesem die Tiefe der Sequenzen ein.

Bei der Sequenzierung des *Tektronix*-Geräts können in einem Sequenzelement die Waveforms aus dem Speicher referenziert werden. Dabei lässt sich angeben, wie oft diese Waveform wiederholt werden soll. Im Anschluss an ein Sequenzelement wird das nächste Element abgespielt. So lassen sich die Waveforms aus dem Speicher wiederverwenden und umstrukturieren [37]. Dadurch wird der benötigte Speicherplatz reduziert. Im *Tabor WX2184C* kommt eine weitere Sequenzierungsebene hinzu. Dabei funktioniert die zweite Sequenzierungsebene genau wie die des *AWG5014C* von *Tektronix*. In der ersten Ebene werden keine Waveforms referenziert, sondern die Sequenzen aus der zweiten Ebene [34]. Daraus ergibt sich eine Art Baumstruktur mit einem Wurzelknoten, also dem Programm, welches den gesamten Spannungsverlauf widerspiegelt. Darunter befindet sich die erste Sequenzierungsebene, welche auf die Subsequenzen der zweiten Sequenzierungsebene verweist. Ganz unten befindet sich die Ebene der Waveforms, die die Blätter des Baums darstellen. Dies ist in Abbildung 4.10 dargestellt.

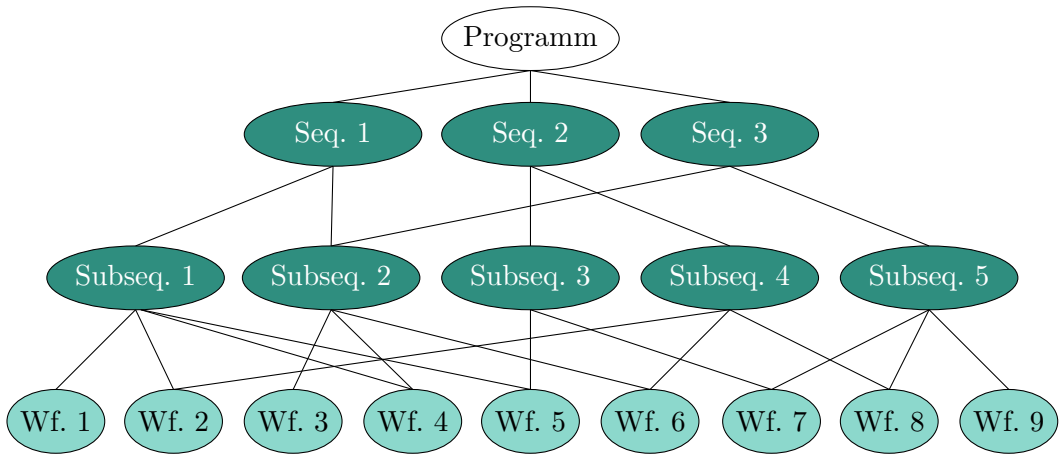


Abbildung 4.10: Graphansicht einer zweistufigen Sequenzierung

Da die Knoten in dieser Struktur zum Teil auch mehrere Elternknoten beziehungsweise Vorgänger haben können, handelt es sich per Definition nicht um einen Baum, sondern um einen Graphen. Dennoch ist die Struktur des Speichers baumartig eingeteilt, da die Sequenzierungsebenen sowohl voneinander als auch vom Waveformspeicher getrennt sind [61]. Dieser Graph enthält in diesem Fall zwei Sequenzierungsebenen. Jede Sequenz und Subsequenz verweist auf die darunterliegende Ebene und kann beliebig aus den Elementen zusammengesetzt werden. Dabei kann für jedes Element eine Wiederholungszahl angegeben werden, die angibt, wie oft das referenzierte Element bei der Ausführung des Programms wiederholt wird.

Diese Struktur wird bereits von den Pulsvorlagen genutzt, die der Benutzer definiert. Die atomaren Pulsvorlagen entsprechen demnach der untersten Ebene, also der Ebene der Waveforms. Alle darüberliegenden Ebenen werden für die strukturellen Pulsvorlagen genutzt, wie zum Beispiel `LoopPulseTemplates` oder `SequencePulseTemplates`. Die `RepetitionPulseTemplates` benötigen hingegen keine eigene Sequenzierungsebene, da diese lediglich durch die Wiederholungszahl umgesetzt werden können. Da diese beiden Strukturen so ähnlich sind, ist die Vorbereitung zum Hochladen der Waveforms sehr einfach umsetzbar.

Die Pulsvorlagen werden zunächst in Programme vom Typ `Loop` übersetzt. Dabei werden alle Parameter aufgelöst und jegliche Informationen entfernt, die für die Hardware nicht relevant sind. Dadurch werden gleichzeitig die `MappingPulseTemplates` aufgelöst, da diese nur zur Umbenennung und Zuordnung der Parameter dienen. So befinden sich die Pulssequenzen nun in einem verflochtenen Graphen, der jedoch prinzipiell bereits von der Hardware verstanden wird. Lediglich die Anzahl der Sequenzierungsebenen könnte zu hoch sein. Daher werden diese auf die maximale unterstützte Anzahl an Ebenen reduziert, indem die Sequenzen und Waveforms der Pulsvorlage zu größeren Sequenzen und Waveforms zusammengefasst werden. Dies geschieht im *qupulse*-Back-End und wird bereits ähnlich praktiziert. Dazu wurde in der vergangenen Entwicklungszeit von *qupulse* ein Algorithmus zur Abflachung und Balancierung der `Loops` implementiert. Dabei werden außerdem bereits die Restriktionen der Hardware berücksichtigt, die die Länge und Auflösung der Waveforms betreffen.

Eine Besonderheit unter den vorgestellten Arbiträrgeneratoren stellt der *HDAWG8* von *Zurich Instruments* dar. Dieser hat keine feste Begrenzung der Sequenzierungsebenen. Da die Sequenzierung bei diesem Gerät in Form eines Programmskripts definiert wird, können nahezu beliebig komplexe Strukturen abgebildet werden, sofern der Speicherplatz ausreicht. Beim Kompilieren des Skripts werden die darin verwendeten Kontrollstrukturen in eine Sequenzstruktur übersetzt, die anschließend vom Arbiträrgenerator

ausgeführt werden kann. Dadurch lässt sich die Struktur der vom Anwender definierten Pulsvorlagen sogar direkt in der Hardware übernehmen. Dies geschieht bisher jedoch nicht im Treiber dieses Geräts und lässt sich daher noch optimieren.

Nachdem die Sequenzierung im Back-End stattgefunden hat und das Waveform-Programm kompatibel zur Hardware gemacht wurde, werden die Daten vom Treiber auf das Gerät hochgeladen. Dazu wird zunächst der Speicher für die einzelnen Waveforms reserviert, um die Waveforms anschließend in den Waveformspeicher zu schreiben. Jede Waveform ist anschließend eindeutig identifizierbar, wobei die Art der Adressierung je nach Gerät variieren kann. Beispielsweise werden die Waveforms auf dem *Tabor WX2184C* mit einer eindeutigen Segment-Nummer adressiert, wohingegen beim *Tektronix AWG5014C* ein Name für jede Waveform festgelegt werden kann [34, 37]. Mit Hilfe dieser Adressierung können die Waveforms aus dem Speicher in den Sequenzen referenziert werden. Die Identifikation der Waveforms wird auch im `ProgramManager` hinterlegt, sodass die hochgeladenen Waveforms dort verwaltet werden können. Zum einen lässt sich dadurch vermeiden, dass dieselben Waveforms mehrfach hochgeladen werden. Zum anderen muss ein Programm, das sich geändert hat und erneut hochgeladen werden soll, nicht in Gänze übertragen werden. Stattdessen kann der `ProgramManager` analysieren, welche Waveforms und Sequenzen sich verändert haben und nur diese erneut hochladen. Daraus ergibt sich eine hohe Zeitersparnis beim Hochladen von Programmen. Ohne Optimierung des Hochladens kann die Übertragungszeit je nach Gerät und Komplexität der Pulssequenz bis zu mehreren Minuten betragen. Im Best-Case kann durch diese Optimierung ein erneutes Hochladen eines Programms extrem verkürzt oder sogar ganz verhindert werden.

Nach den Waveforms werden die Sequenzen auf den Arbiträrgenerator übertragen. Dabei werden die Sequenzen beim *WX2184C* und *AWG5014C* in die Sequenzierungstabelle(n) geschrieben. Für den *HDAWG8* wird hingegen ein Skript vom Treiber erzeugt, welches die Sequenzstruktur in Form von Schleifen und Verzweigungen abbildet. Dieses Skript wird anschließend kompiliert und daraufhin auf das Gerät hochgeladen. Der gesamte Prozess des Hochladens eines Programms geschieht, aufgrund der großen Unterschiede zwischen den Geräten, innerhalb der Treiberimplementierungen. Die Funktionsaufrufe für das Hochladen der Waveforms und Sequenzen sollen hingegen mit Hilfe der `Features` vereinheitlicht werden, sodass alle Treiber einheitlich bedient werden können.

Zusammengefasst ist zu erkennen, dass die Treiberstruktur bereits so gewählt ist, dass die Funktionalität der hier vorgestellten Geräte darin unterzubringen ist. Die einzelnen Funktionen und Eigenschaften werden zum Teil bereits durch die Struktur der Treiber repräsentiert. Zum Teil können die Funktionen hingegen auch durch die `Features` rea-

lisiert werden, welche ein hohes Maß an Erweiterbarkeit erzeugen. Lediglich die direkte Hardwarekommunikation und hardwarespezifische Vorgänge befinden sich auf unterster Ebene in den Treibern. Dabei wird sichergestellt, dass die Struktur und die Schnittstelle sowohl für die Benutzer als auch für das Back-End einheitlich umgesetzt werden.

4.3.2 Vereinfachung der Konfiguration der Treiberspezifikationen

Eine Schwierigkeit im Umgang mit *qupulse* kann das Treiberkonzept im bisherigen Zustand nicht lösen und zwar die Konfiguration der Treiber. Diese konnte zwar bereits deutlich vereinfacht werden, da die Treiberkonfiguration nun für alle Geräte einheitlich funktioniert, der Aufwand ist jedoch nicht geringer geworden. Um auch diesen zu reduzieren, wird zunächst ein weiteres *Entwurfsmuster* vorgestellt: das *Builder Pattern* oder seltener auch *Erbauer*. Bei diesem werden die Erzeugung und die Repräsentation eines Objekts voneinander getrennt, um den Erzeugungsprozess auch für andere Repräsentationsmöglichkeiten zu verwenden. Dieses Konzept erinnert an das *Fabrikmuster* oder *Factory Pattern*, wobei hier jedoch nicht nur die Erzeugung des Objekts übernommen wird, sondern auch dessen initiale Konfiguration oder die Zusammensetzung aus Teilobjekten [46].

Für die AWG-Treiber bietet sich das *Builder Pattern* aus dem Grund an, da es zum einen den Konfigurationsaufwand verringert und zum anderen die Möglichkeit bietet, Standardkonfigurationen zu erstellen, die zwischen den Mitarbeitern ausgetauscht werden können. Eine beispielhafte Implementierung eines *Erbauers* für einen Treiber vom Typ `Device` befindet sich in Programmcode 4.5.

```
1  class AWGBuilder:
2      def __init__(self, connection: str):
3          self._connection    = connection
4          self._channel_group = 2   # Default group size
5          self._amplitude     = 4.0 # Default amplitude [Vpp]
6          self._offset        = 0.0 # Default offset [V]
7
8      def withChannelGrouping(self, n: int) -> AWGBuilder:
9          self._channel_group = n
10         return self
11
12     def withAmplitude(self, ampl: float) -> AWGBuilder:
13         self._amplitude = ampl
14         return self
15
```

```
16      def withOffset(self, offs: float) -> AWGBuilder:
17          self._offset = offs
18          return self
19
20      def build(self):
21          device = Device(self._connection)
22          device.synchronize_channels(self._group_size)
23
24          # Set amplitude and offset for all channels
25          for channel in device.channels():
26              channel.set_amplitude(self._amplitude)
27              channel.set_offset(self._offset)
28
29          return device
30
31  device = AWGBuilder("192.168.123.45:6789")\
32      .withChannelGrouping(4)\
33      .withAmplitude(2.4)\
34      .withOffset(-0.5)\
35      .build()
```

Programmcode 4.5: Vereinfachung der Treiberkonfiguration mit dem *Builder Pattern*

Der in Programmcode 4.5 vorgestellte `AWGBuilder` dient der vereinfachten Erzeugung eines Treiberobjekts. Dabei werden gleichzeitig gewisse Konfigurationen vorgenommen, sodass das erbaute Objekt bereits die benötigte Grundfunktionalität liefert, um dieses direkt einsetzen zu können. In diesem Beispiel lassen sich vier Attribute konfigurieren. Als `connection` werden die IP-Adresse und der Port des Geräts konfiguriert, was bereits im Konstruktor angegeben werden muss, da es ein Pflichtattribut ist. Die anderen Attribute sind das `ChannelGrouping` zur Kanalsynchronisation und die kanalspezifischen Eigenschaften `Amplitude` und `Offset`, die für jeden einzelnen Kanal eingestellt werden. Diese Attribute sind optional und müssen somit nicht angegeben werden. Wenn diese nicht angegeben werden, wird auf Standardwerte zurückgegriffen, die hier im Konstruktor des `AWGBuilders` definiert sind. Die Konfiguration der optionalen Attribute erfolgt über die mit `with` beginnenden Methoden. Dabei wird in jeder Methode das `self`-Objekt zurückgegeben, sodass die Methodenaufrufe aneinander gekettet werden können, um die Konfiguration mit möglichst wenig Programmcode vornehmen zu können, wie in Programmcode 4.5 ab Zeile 32 zu erkennen ist [62]. Durch den finalen Aufruf der `build`-Funktion wird das Treiberobjekt erstellt und konfiguriert. Im Beispiel findet die gesamte Konfiguration in der `build`-Methode statt. Alternativ kann auch ein Konstruktor für den Treiber angeboten werden, der ein `AWGBuilder`-Objekt als Übergabepara-

meter erwartet. In diesem Fall würde die gesamte Konstruktion des Treiberobjekts im Konstruktor des Treibers stattfinden. Dadurch können die konfigurierten Werte initial im Treiber verwendet werden, wodurch die Konstruktion etwas schneller abläuft. Dabei würde die `build`-Methode lediglich aus der Zeile `return Device(self)` bestehen und der `Device`-Konstruktor ist für die Extraktion und Anwendung der Konfigurationsdaten verantwortlich. Zur besseren Übersichtlichkeit wird hier jedoch die Treiberkonfiguration in der `build`-Methode bevorzugt.

Zur weiteren Vereinfachung der Treiberkonfiguration können auf diese Weise auch *Builder*-Objekte mit Standardkonfigurationen erstellt werden. Diese können zwischen den Mitarbeitern ausgetauscht werden, um bei wiederkehrenden Versuchsaufbauten den Konfigurationsaufwand zu verringern. Dabei kann gegebenenfalls gänzlich auf eine eigene Konfiguration verzichtet werden oder die Standardkonfiguration wird als Ausgangsbasis für die eigene verwendet, sodass nur noch die Feinabstimmung vorgenommen werden muss.

Die hier exemplarisch aufgeführten Konfigurationsattribute stellen nur einen kleinen Ausschnitt der Möglichkeiten dieses Entwurfs dar. Jedes Attribut, jede Funktion und jedes `Feature` lassen sich durch den *Erbauer* abbilden. Auch komplexere Einstellungen können in einem *Erbauer* gekapselt werden, sodass auch Konfigurationen vorgenommen werden können, die normalerweise aus mehreren aufeinanderfolgenden Funktionsaufrufen bestehen. Daraus ergibt sich ein sehr mächtiges Werkzeug zur Basiskonfiguration der Treiber, das die Arbeit mit den Treibern deutlich vereinfachen kann.

4.3.3 Tests und Validierung

Ein häufig vernachlässigter Teil der Softwareentwicklung ist das Testen. Dabei sollten parallel zur Konzeptionierung und Entwicklung immer auch Tests berücksichtigt werden, um Fehler in der Software zu vermeiden. Wie alle Menschen, machen auch Softwareentwickler Fehler und das Ziel von Tests ist es, diese Fehler frühzeitig zu erkennen. Unerkannte Fehler können schnell in die Produktivumgebung einer Software gelangen, wo eventuell ein größerer Schaden angerichtet wird. Mit einem durchdachten Testkonzept, das im besten Fall automatisiert abläuft, lassen sich viele Fehler im Vorhinein vermeiden und die Suche nach den Fehlern kann beschleunigt werden [63].

Testverfahren

Das Ziel von Softwaretests ist also die gezielte Erkennung und Vermeidung von Fehlern. Dabei existieren zwei grundsätzliche Verfahren zur Auswahl der Tests. Bei den *Black-Box-Tests* werden die Tests unabhängig von Programmstruktur und Quelltext ausgewählt. Das Softwaresystem wird als „schwarze Box“ betrachtet, in die nicht hineingesehen werden kann. Der Tester hat somit keine Informationen über den inneren Aufbau des Systems und erstellt die Testfälle anhand der Dokumentation und der Anforderungen des Systems. Für gewöhnlich werden *Black-Box-Tests* nicht von den Softwareentwicklern durchgeführt, sondern von fachlich orientierten Testern, die die Tests funktionsorientiert auswählen. Werden Tests von Softwareentwicklern durchgeführt, haben diese meist Kenntnis von der internen Systemstruktur. In dem Fall werden die Tests strukturorientiert durchgeführt und es ist von *White-Box-Tests* die Rede. Dabei wird vorausgesetzt, dass der Tester die Struktur und häufig auch den Quelltext des Systems kennt und anhand dieses Wissens gezielt die Testfälle auswählt. Beide Testauswahlverfahren bieten Vor- und Nachteile, weshalb der Einsatz beider Verfahren sinnvoll ist. Die *White-Box-Tests* bieten sich an, um eine hohe Testabdeckung zu erzielen. Dabei soll jede Komponente und optimalerweise jede Zeile des Programmcodes durch Tests abgedeckt werden. Das heißt, jede Zeile des Quelltexts soll in mindestens einem Test durchlaufen werden. Daher ist es zur Erhöhung der Testabdeckung zwingend erforderlich, *White-Box-Tests* durchzuführen, da dies ohne Kenntnis des Quelltextes nicht möglich wäre. Besonders bei zeilenweise interpretierten Programmiersprachen, wie Python, ist dies sinnvoll, da nur auf diese Weise alle Syntaxfehler verlässlich aufgedeckt werden können. Dies geschieht bei kompilierten Programmiersprachen bereits während des Kompilationsvorgangs. Die *Black-Box-Tests* hingegen dienen dazu, Standardfälle zu testen, die der Praxis möglichst nahekommen. Außerdem können Tester, die keine Softwareentwickler sind und den Quelltext nicht kennen, dabei helfen, Missverständnisse der Entwickler aufzudecken [64, 65].

Eine weitere Kategorisierung von Tests ist dessen Einteilung in vier Teststufen [63]:

Komponententests: Die *Komponententests* beziehungsweise *Unit-Tests* testen ein Softwaresystem auf unterster Ebene. Dabei werden möglichst alle Bestandteile eines Systems separat und isoliert voneinander getestet. Die zu testenden Komponenten sind dabei in der Regel die Funktionen. Dabei wird für jede Funktion mindestens ein Test definiert, um dessen Korrektheit zu validieren. An jeder Stelle, wo die getesteten Komponenten eingesetzt werden, kann durch die Tests davon ausgegangen werden, dass jene das erwartete Ergebnis liefert. Hierbei sollte bereits eine hohe Testabdeckung des Quellcodes angestrebt werden, um sicherzustellen, dass die Komponenten auch für alle Sonderfälle

funktionieren. In der Regel handelt es sich bei *Unit-Tests* um *White-Box-Tests*, da diese für gewöhnlich vom Entwickler selbst definiert werden [63]. Eine Ausnahme stellt die *testgetriebene Entwicklung*[18] dar, bei der die Testfälle zwar auch meistens von den Entwicklern erstellt werden, dies geschieht jedoch vor der Implementierung. Die Testdefinition basiert dabei auf der Beschreibung der vom Auftraggeber angeforderten Funktionen. Nach der Erstellung des Testfalls schlägt dieser zunächst fehl, da die zu entwickelnde Funktionalität noch nicht existiert. Im Anschluss wird die gewünschte Funktionalität auf den Test hin implementiert, bis dieser nicht mehr fehlschlägt. Dadurch ist sichergestellt, dass genau die vom Auftraggeber beschriebene Funktion realisiert wurde und dass keine Funktion ungetestet ist [66]. Da die Tests hierbei bereits vor der Implementierung definiert werden, handelt es sich um einen *Black-Box-Test*.

Die *Unit-Tests* ermöglichen eine hohe Testabdeckung und erleichtern die Fehlersuche. Weil die einzelnen Komponenten separat getestet werden, beschränkt sich die Fehlersuche im Falle eines fehlgeschlagenen Tests nur auf eine Methode und nicht auf das gesamte System [63]. Zur Durchführung von *Unit-Tests* bietet Python das Framework `unittest`. Dieses ist mit diversen Entwicklungsumgebungen kompatibel und ermöglicht die einfache Implementierung, Ausführung und Auswertung der Testfälle.

Integrationstests: Nachdem die *Komponententests* sicherstellen, dass jede Funktion eines Systems so funktioniert, wie erwartet, folgt die Teststufe der *Integrationstests*. Diese Teststufe konzentriert sich auf die Schnittstellen der Komponenten und deren Abhängigkeiten untereinander. So wird getestet, wie die einzelnen Komponenten zusammenarbeiten und miteinander kommunizieren. Dies ist erforderlich, da die Komponenten häufig von verschiedenen Entwicklern stammen. Somit wird sichergestellt, dass Änderungen an Schnittstellen keine unerwünschten Effekte in anderen Programmteilen hervorrufen. Außerdem lassen sich mit *Integrationstests* bereits praxisnahe Abläufe testen, mit denen die Zusammenhänge zwischen den Komponenten geprüft werden [63].

Systemtests: Auf der Ebene über den *Komponenten-* und *Integrationstests* befinden sich die *Systemtests*. Diese dienen zur Validierung der Anforderungen des Kunden. Dabei geht es um die großen Zusammenhänge der Systemmodule, wobei nicht nur funktionale, sondern auch nicht-funktionale Anforderungen geprüft werden. Die funktionalen Anforderungen beschreiben dabei die konkreten Aufgaben einer Software, wohingegen die nicht-funktionalen Anforderungen solche Themen wie Zuverlässigkeit, Wartbarkeit und Benutzungsfreundlichkeit umfassen. Die Testumgebung für Systemtests ist meistens bereits nah an der Produktivumgebung der Kunden, findet jedoch noch beim Softwarehersteller statt [63].

[18] engl. „test-driven development“ (TDD)

Abnahmetests: Erst die *Abnahmetests* werden beim Kunden selbst durchgeführt. In dieser Testphase wird meistens mit Echtdaten oder zumindest sehr realistischen Daten getestet. Der *Abnahmetest* dient, wie der Name bereits ausdrückt, der Abnahme eines Softwaresystems durch den Kunden. Wurden diese Tests erfolgreich durchgeführt, übernimmt der Kunde offiziell die Software. Wenn alle vorherigen Testphasen bereits erfolgreich waren, ist davon auszugehen, dass auch diese Testphase positiv verläuft [63].

Während der Entwicklung der AWG-Treiber sollten bereits *Unit-* und *Integrationstests* durchgeführt werden. Diese können bereits vor Beginn der Implementierung definiert werden, wie es beim *test-driven development* gehandhabt wird. Dies bietet sich an, da durch die ex-ante-Konzeptionierung des Treibersystems bereits die funktionale Definition der Treiber bekannt ist. Besonders bei der späteren Weiterentwicklung, bei der stetig neue `Features` hinzugefügt werden, empfiehlt sich ein Vorgehen gemäß der *testgetriebenen Entwicklung*. Auch die *Integrationstests* sind enorm wichtig für die Treiber, da damit die Zusammenhänge der verschiedenen Schnittstellen getestet werden können.

Im Anschluss an die Implementierung der neuen Treiberstruktur muss diese in das *qupulse*-Framework integriert werden. Daraufhin müssen *Systemtests* durchgeführt werden, die das *qupulse*-Framework in seiner Gesamtheit testen. Dafür existieren in der Dokumentation bereits einige Codebeispiele, die verwendet werden können, um die Zusammenarbeit aller Module testen zu können [30]. Zuletzt folgen abschließende Abnahmetests in den Laboren der Arbeitsgruppe Quantentechnologie an der RWTH Aachen. Dort werden Messsysteme, bestehend aus verschiedenen Instrumenten, installiert, an denen das neue Treibersystem mit gemessenen Echtdaten getestet werden kann.

Hardwaresimulation

Ein Problem beim Testen von Hardwaretreibern stellt das Testen ohne Hardware dar. Besonders in den frühen Entwicklungsphasen sind die Treiber noch nicht vollständig oder die Hardware steht nicht zur Verfügung. Bei einem Großteil der *Unit-Tests* ist dies zwar kein Problem, da auf kleinster Ebene nicht für jede Funktion eine Geräteverbindung erforderlich ist. Vor allem aber die *Integrations-* und *Systemtests* sind auf eine angeschlossene Hardware angewiesen. Da die Hardware jedoch nicht immer verfügbar ist, besonders mit Blick auf die Testautomatisierung, müssen auch Tests ohne Hardware ermöglicht werden.

Auf Ebene der *Komponententests* können zur Vermeidung einer Hardwareverbindung sogenannte *Mock-Objekte* verwendet werden. Diese Objekte werden als Attrappen für andere Objekte verwendet, die eigentlich eine Verbindung zur Hardware benötigen. Dazu

kann ein *Mock-Objekt* beliebig Funktionen des eigentlichen Objekts überschreiben, um die Ergebnisse zu erhalten, die bei korrekter Implementierung erwartet werden. Dadurch können die zu testenden Komponenten isoliert von dessen Abhängigkeiten getestet werden [67]. Wenn die „gemockte“ Funktionalität bereits erfolgreich getestet und validiert wurde, kann davon ausgegangen werden, dass die erwarteten Ergebnisse geliefert werden. Daher ist es bei einem *Unit-Test* nicht notwendig, die darin enthaltenen Abhängigkeiten mitzutesten. Dies geschieht erst auf Ebene der *Intergrationstests*. Bei diesen bietet sich die Verwendung von Simulatoren an. Simulatoren werden zum Teil bereits vom Hersteller der eigentlichen Hardware bereitgestellt. Diese simulieren die Kommunikation mit der Hardware und reagieren genauso auf die gesendeten Kommandos, wie die echte Hardware. Auch der Verbindungsaufbau und die Programmierschnittstelle eines Simulators stimmen mit der echten Hardware überein. Wichtig dabei ist, dass der Simulator fehlerfrei ist, da ein Test ansonsten keine Aussagekraft hätte [67].

Ist ein Simulator für die zu testende Hardware nicht verfügbar oder tatsächlich fehlerhaft, muss auf Alternativen zurückgegriffen werden. Für die drei vorgestellten Arbiträrgeneratoren existiert zum Beispiel nur ein Simulator für den *Tabor WX2184C* [29]. Für die anderen Geräte empfiehlt sich daher die Implementierung eines eigenen Simulators. Dieser muss die empfangenen Kommandos verarbeiten und dem Treiber dieselbe Antwort liefern, wie die echte Hardware. Um dies zu realisieren ist unter Umständen ein enormer Aufwand erforderlich. Daher sollte zunächst abgewogen werden, ob der Simulator die gesamte Funktionspalette der Hardware bereitstellt oder nur die zurzeit benötigten Funktionen. Wird nur der zu einem Zeitpunkt benötigte Funktionssatz von der Simulation bereitgestellt, kann der Simulator immer dann erweitert werden, wenn ein neues Kommando getestet werden soll. Alternativ kann auch nur ein fester Satz an Kernfunktionen umgesetzt werden, um den Treiber im Allgemeinen testen zu können. Für detailliertere Tests wäre es dann erforderlich, die echte Hardware hinzuzuziehen. So könnte jedoch bereits ein Großteil der Funktionalität ohne Hardware getestet werden.

Spätestens bei den *Systemstests* ist jedoch die Verwendung der echten Hardware empfehlenswert, denn beim Systemtest müssen nicht nur einzelne Geräte getestet werden, sondern das gesamte *qupulse*-Framework. Dabei wird auch unter anderem das Zusammenspiel zwischen Arbiträrgeneratoren und Datenerfassungsgeräten getestet, wobei die von den Arbiträrgeneratoren erzeugten Spannungssignale die Messergebnisse beeinflussen. Daher müssen bei den *Systemstests* definierte Versuchsaufbauten verwendet werden, in welchen die Messergebnisse bis auf geringe Abweichungen reproduzierbar sind. Diese Tests müssen jedoch von Menschenhand ausgewertet werden, da die Ergebnisse je nach Versuchsaufbau variieren und das Testframework nicht erkennen kann, wie die Instrumente miteinander verbunden sind.

Automatisierung

Zur weiteren Optimierung der Softwaretests, sollten diese automatisiert werden. Da die *Systemtests* an der echten Hardware durchgeführt werden müssen und optimalerweise vom Menschen ausgewertet werden, ist dabei keine Automatisierung möglich. Bei den *Integrationstests* und vor allem bei den *Unit-Tests* kann jedoch eine Testautomatisierung vorgenommen werden.

Für die Testautomatisierung bietet sich die aktuelle Umgebung von *qupulse* besonders an. Mit Python wurde eine plattformunabhängige Sprache gewählt und, da das Projekt in einem *GitHub*-Verzeichnis verwaltet wird, können automatisierte Testwerkzeuge eingesetzt werden. Zum automatisierten Testen bietet sich *Continuous Integration* an. Dabei wird der Quellcode eines Projekts regelmäßig und automatisch kompiliert, installiert und getestet wird. Welche Schritte dabei genau vorgenommen werden, kann vom Administrator des Projekts selbst in der sogenannten *Continuous-Integration-Pipeline* verwaltet werden. Die Durchführung der *fortlaufenden Integration* erfolgt in den meisten Projekten bei jeder Änderung am Quellcode, die in das *GitHub*-Verzeichnis übertragen wird. Dabei sind die produktiven Zweige der Quellcodeverwaltung in der Regel gesperrt, sodass dort nur die Änderungen einfließen können, die bereits erfolgreich von der *Continuous-Integration*-Umgebung getestet wurden [68].

Da es sich bei Python um eine interpretierte Sprache handelt, fällt der erste Schritt, die Kompilation, weg. Daher startet die *Continuous-Integration-Pipeline* bei der Installation, wobei das Projekt in einer leeren Umgebung installiert wird. Dabei lässt sich feststellen, ob die Installation selbst funktioniert und ob diese auch alle benötigten Abhängigkeiten berücksichtigt. Wurde das Projekt erfolgreich installiert, lassen sich anschließend die Tests durchführen. Dies wird automatisch von der *Continuous-Integration*-Umgebung gesteuert und ausgewertet. Erst, wenn alle Schritte der *Continuous-Integration-Pipeline* erfolgreich ausgeführt wurden, werden die Änderungen im *GitHub*-Verzeichnis freigegeben, sodass sie auch in die Produktivzweige der Quellcodeverwaltung übernommen werden dürfen [68].

Durch die Verwendung von *Continuous Integration* kann sichergestellt werden, dass keine Änderungen in der Produktivumgebung landen, die zu fehlerhaften Tests führen. Außerdem lässt sich in der *Continuous Integration Pipeline* eine Prüfung zur Qualität des Programmcodes durchführen. Diese kann unter anderem feststellen, ob der neu hinzugefügte Quelltext durch einen Test abgedeckt ist. So lassen sich auch weitere Ziele zur Codequalität definieren, die automatisch geprüft werden und zur Gesamtqualität der Software beitragen.

5 Zusammenfassung und Ausblick

Zusammenfassung

Im Laufe dieser Arbeit konnte ein Treibersystem konzipiert werden, das die Funktionalität der vorgestellten Arbiträrgeneratoren abstrahiert und vereinheitlicht. Dazu wurden die Eigenschaften und Funktionen der drei momentan eingesetzten Arbiträrgeneratoren analysiert, um das Konzept darauf hin zu optimieren. Außerdem wurden Entwurfsmuster und Konventionen der Softwareentwicklung berücksichtigt, um eine einheitliche und intuitive Schnittstelle zu erhalten.

In Abschnitt 2.3.2 wurden die ursprünglichen Ziele der Entwicklung von *qupulse* beschrieben, die zuvor von *Simon Humpohl* in seiner Masterarbeit definiert wurden [25]. Besonders bei zwei dieser Ziele konnte ein Optimierungsbedarf festgestellt werden:

Leichte Verständlichkeit des Front-Ends: Das eigentliche Front-End von *qupulse* war bereits einfach aufgebaut und auch für Nicht-Softwareentwickler leicht verständlich. Dazu zählen unter anderem die Pulsvorlagen, deren Erstellung bereits nach kurzer Lektüre der Dokumentation gelingt. Jedoch reicht es nicht aus, die Pulsvorlagen zu definieren, um *qupulse* zu verwenden. Um diese Pulse auf den Arbiträrgenerator hochzuladen, müssen zunächst die Geräte mit Hilfe der Treiber konfiguriert werden. Da dies auch direkt vom Benutzer gemacht wird, zählt die Benutzerschnittstelle der Treiber ebenfalls zum Front-End. Zu Beginn dieser Arbeit besaßen die Treiber lediglich eine Schnittstelle, die zur Kommunikation mit dem Back-End diente. Der Rest der Treiber-Klassen wurde nicht vereinheitlicht und konnte somit von Gerät zu Gerät variieren. Um das Ziel der leichten Verständlichkeit des Front-Ends auch für das Treiber-Front-End zu erreichen, wurde eine Hardwareabstraktion im Treibersystem als primäres Ziel dieser Arbeit definiert, an dem sich auch das neue Treiberkonzept orientiert.

Flexibles Back-End: Ein weiteres Ziel war ein flexibles Back-End, welches viele verschiedene Geräte unterstützen soll. Dies wurde bisher durch die `AWG`-Schnittstelle der Treiber erreicht, die nur die vom Back-End benötigte Funktionalität in den Treibern

bereitstellt. Dies führte zu einer hohen Flexibilität der Treiber, da nur ein geringer Funktionsumfang vorgegeben wurde. Um diese Flexibilität nicht einzuschränken, darf die Abstraktion der Treiberstruktur nicht dessen Funktionalität und Erweiterbarkeit einschränken. Daher wurde im neuen Konzept die Erweiterung des bereitgestellten Funktionsumfangs angestrebt, wodurch gleichzeitig die Wartbarkeit und Flexibilität der Treiber optimiert wurde.

Diese Ziele konnten erreicht werden, indem eine Hardwareabstraktion konzipiert wurde, welche die einzelnen Treiber vereinheitlicht, ohne dabei deren Funktionalität einzuschränken. Außerdem wird ein hohes Maß an Erweiterbarkeit und Wartbarkeit erreicht, indem auf Entwurfsmuster und allgemeine Konventionen der Softwareentwicklung zurückgegriffen wurde.

Die Hardwareabstraktion wurde umgesetzt, indem eine Struktur für die Treiber vorgegeben wurde. Diese setzt keine besonderen Eigenschaften der Geräte voraus, außer dass es sich um Arbiträrgeneratoren handeln muss. Die Treiberstruktur gibt ein Konstrukt vor, in welchem ein Arbiträrgenerator aus mehreren Kanälen besteht, die synchronisiert werden können. Diese Struktur bleibt dabei offen für Arbiträrgeneratoren mit geringerem Funktionsumfang, die beispielsweise nur einen Kanal besitzen. So ist auch das Ziel des flexiblen Back-Ends erfüllt, das möglichst viele verschiedene Geräte unterstützen soll. Dadurch, dass weiterhin die bestehende `AWG`-Schnittstelle zwischen Treiber und Back-End verwendet wird, sind die neuen Treiber direkt in das bestehende *qupulse*-Framework integrierbar und schränken dessen Funktionalität nicht ein. Stattdessen erhöht sich die Flexibilität sogar durch die Verwendung des *Adapters*, mit dem die Treiber an die `AWG`-Schnittstelle angeschlossen werden. Dadurch sind die Treiber gänzlich vom restlichen *qupulse*-Back-End entkoppelt, sodass nun auch Treiber angebunden werden können, die mit der bisherigen Schnittstelle eventuell nicht realisiert werden konnten.

Da die vorgegebene Treiberstruktur sehr einfach gehalten wurde, um die Integration vieler verschiedener Geräte zu ermöglichen, wurde zur Abstraktion der Funktionalität der Geräte ein anderer Ansatz gewählt. Anstatt die benötigte Funktionalität direkt durch Vererbung in der Treiberstruktur zu verankern, wurde eine deutlich flexiblere Realisierung gewählt, die gleichzeitig die Erweiterbarkeit und Wartbarkeit des Systems optimiert. Sowohl die hardwarespezifischen als auch allgemeingültige Funktionen werden in Anlehnung an das *Bridge Pattern* umgesetzt. Dabei wird die Implementierung der Treiber von deren Abstraktion getrennt, um jede Funktion und Eigenschaft der Treiber auf Seite der Abstraktion durch die `Features` zu vereinheitlichen. Die Umsetzung dieser `Features` erfolgt innerhalb der Treiberimplementierungen und ist somit unabhängig von der Abstraktion. Dadurch kann sichergestellt werden, dass alle Treiber einheitlich

bedienbar sind und dennoch jeder einzelne Treiber das volle Potential der Arbiträrgeneratoren nutzen kann. Da Python die dynamische Erweiterung von Objekten ermöglicht, kann dieses Konstrukt dort optimal realisiert werden.

Insgesamt konnte die Verständlichkeit des Front-Ends deutlich verbessert werden, da alle Treiber einheitlich bedienbar sind und die gleiche Struktur aufweisen. Außerdem wurde die Flexibilität des Back-Ends durch die lose Kopplung zwischen den Treibern und dem restlichen *qupulse*-Back-End optimiert, sodass beide Komponenten unabhängig voneinander weiterentwickelt werden können. Mit diesem Ansatz lässt sich nahezu jeder Arbiträrgenerator in das Treibersystem integrieren, ohne dass dessen Funktionsumfang und Leistungsfähigkeit eingeschränkt werden.

Ausblick

Im Anschluss an diese Arbeit folgt zunächst die Implementierung des Konzepts. Dabei wird zuerst die Treiberstruktur aus dem Grobkonzept realisiert, woraufhin die an das *Bridge Pattern* angelehnte `Feature`-Struktur umgesetzt wird. Dabei werden zunächst keine konkreten `Features` implementiert, außer der Abtastrate, da diese Eigenschaft bereits für die `AWG`-Schnittstelle erforderlich ist. Sobald die neue Treiberstruktur fertiggestellt ist, können die bestehenden Treiber in die neue Struktur überführt werden und bei Bedarf können bereits neue Treiber angebunden werden. Die Implementierung der einzelnen Treiber kann anschließend auch parallel erfolgen, wobei jedoch darauf geachtet werden muss, dass die dazu erstellten `Features` konsistent bleiben und keine Redundanzen entstehen. Durch einen regelmäßigen Abgleich mit dem *qupulse*-Repository können Inkonsistenzen und Redundanzen dabei größtenteils vermieden werden.

Nachdem die Treiberstruktur erstellt und die drei bestehenden Treiber darin eingepflegt wurden, ist es möglich, das System agil und inkrementell weiterzuentwickeln. Durch die Verwendung der `Feature`-Struktur können Funktionserweiterungen in kleinen Schritten erfolgen. Dies reduziert die Fehleranfälligkeit des Systems und optimiert die Wartbarkeit, denn jedes einzelne `Feature` kann für sich getestet werden, sodass Fehler und unerwünschte Seiteneffekte direkt erkannt und behoben werden können.

Nach erfolgreicher Implementierung des neu konzipierten Treibersystems, wird dieses zur Erforschung und Weiterentwicklung der Quantencomputer eingesetzt werden. Zum einen sollen Spannungspulse gefunden werden, die sich zur Realisierung von Quantengattern eignen. Diese sollen, analog zu elektronischen Gattern in klassischen Computern,

die elementaren Operationen auf Qubits durchführen. Zum anderen soll ein Quantenbus entwickelt werden, mit welchem ein Quantenpunkt, inklusive des darin enthaltenen Elektrons und dessen Informationsgehalt, in einem Quantenprozessor transportiert werden soll. Dadurch sollen auch Informationen in großen skalierbaren Systemen übertragen werden können [5, 27].

Literaturverzeichnis

[1] Forschungszentrum Jülich GmbH. *Das Zentralinstitut für Engineering, Elektronik und Analytik (ZEA)*. 3. Januar 2019. URL: https://www.fz-juelich.de/portal/DE/Institute/ZentralinstitutEngineering/_node.html.

[2] Forschungszentrum Jülich GmbH. *Systeme der Elektronik (ZEA-2)*. 16. Mai 2019. URL: https://www.fz-juelich.de/zea/zea-2/DE/Home/home_node.html.

[3] Forschungszentrum Jülich GmbH. *Nano- und Mikroelektronische Systeme*. 25. Juni 2018. URL: https://www.fz-juelich.de/zea/zea-2/DE/Forschung/Nano-und-Mikroelektonische-Systeme/_node.html.

[4] *JARA-Institut für Quanteninformation – Das Institut*. 21. Mai 2019. URL: http://www.quantuminfo.physik.rwth-aachen.de/cms/Quantuminfo/~dqvq/Das-Institut/.

[5] *JARA-Institut für Quanteninformation – Gruppe Bluhm*. 20. Mai 2019. URL: http://www.quantuminfo.physik.rwth-aachen.de/go/id/dqwq.

[6] Werner Heisenberg. *Der Teil und das Ganze. Gespräche im Umkreis der Atomphysik*. München, 1969, S. 280.

[7] Matthias Manning. *Quantenphysik: Im Kleinen spielt das Universum verrückt*. 20. April 2017. URL: https://www.golem.de/news/quantenphysik-im-kleinen-spielt-das-universumverrueckt-1704-127350.html.

[8] Steven Holzner. *Quantenphysik für Dummies*. Übers. von Regine Freudenstein. 2013.

[9] Oriol Romero-Isart und Anika Pflanzer. *Quantenmechanik am Limit: Superpositionen massiver Objekte*. 2011. URL: https://www.mpg.de/4981741/Quantenphysik_am_Limit.

[10] LEIFIphysik.de. *Schrödingers Katze – Ein Gedankenexperiment*. URL: https://www.leifiphysik.de/atomphysik/quantenmech-atommodell/versuche/schroedingers-katze-ein-gedankenexperiment.

[11] Birgit Bomfleur. *Schrödingers Katze kann aufatmen – und sei es auch nur ein letztes Mal*. Ismaning, Oktober 2001.

[12] Forschungszentrum Jülich GmbH. *Quantencomputer*. URL: https://www.fz-juelich.de/portal/DE/Forschung/schlaglichter/quantencomputer.html.

[13] Hendrik Degering. *Quantencomputing I: Oberseminar Theoretische Informatik SS 2005*. Göttingen, 2005. URL: http://www.nld.ds.mpg.de/~hecke/archiv/osemSS05.pdf.

[14] Wikipedia. *Bloch-Kugel*. 9. Mai 2019. URL: https://de.wikipedia.org/w/index.php?title=Bloch-Kugel&oldid=188405945.

[15] Wikipedia. *Qubit*. 7. Juni 2019. URL: https://de.wikipedia.org/w/index.php?title=Qubit&oldid=189339678.

[16] Hendrik Bluhm u. a. *Electrons in Solids*. April 2019. Kap. 3 Quantum computing, qubits and decoherence, S. 125–204. ISBN: 978-3-11-043832-1.

[17] ARTE. *Wie funktioniert ein Quantencomputer? Interview mit Prof. Rainer Blatt*. 2016. URL: https://www.arte.tv/de/videos/069293-002-A/wie-funktioniert-ein-quantencomputer/.

[18] Wikipedia. *Quantencomputer*. 3. Juni 2019. URL: https://de.wikipedia.org/w/index.php?title=Quantencomputer&oldid=189218033.

[19] *GaAs Spinqubits*. 24. April 2017. URL: http://www.quantuminfo.physik.rwth-aachen.de/cms/Quantuminfo/Forschung/Institut-fuer-Quantentechnologie/~dvux/Bluhm-GaAs/.

[20] J. R. Petta u. a. *Coherent Manipulation of Coupled Electron Spins in Semiconductor Quantum Dots*. 30. September 2005. URL: https://doi.org/10.1126/science.1116955.

[21] Wikipedia. *Spin*. 10. März 2019. URL: https://de.wikipedia.org/w/index.php?title=Spin&oldid=186424130.

[22] Simon Humpohl. *A multithreaded data acquisition dll*. Aachen, Juli 2014.

[23] Pascal Cerfontaine. *High-Fidelity Single- and Two-Qubit Gates for Two-Electron Spin Qubits*. Aachen, 17. Mai 2019.

[24] Wikipedia. *Triplet state*. 16. Februar 2019. URL: https://en.wikipedia.org/w/index.php?title=Triplet_state&oldid=883563302.

[25] Simon Humpohl. *Hardware Adapted Pulses and Software for Qubit Control*. Aachen, März 2017.

[26] Tim Botzem. *Coherence and high fidelity control of two-electron spin qubits in GaAs quantum dots*. Aachen, 2016.

[27] Si-QuBus Consortium. *What is a Si-QuBus good for?* 2019. URL: http://www.siqubus.rwth-aachen.de.

[28] Jochen Schulz. *Python - die Alternative zu Matlab?* Göttingen, 1. Oktober 2013. URL: http://num.math.uni-goettingen.de/~schulz/data/python_talk.pdf.

[29] Quantum Technology Group RWTH Aachen. *Qupulse's repository.* URL: https://github.com/qutech/qupulse.

[30] Quantum Technology Group RWTH Aachen. *Qupulse's documentation.* 17. Mai 2019. URL: https://qupulse.readthedocs.io/en/0.4/index.html.

[31] Wikipedia. *Arbiträrgenerator.* 6. August 2017. URL: https://de.wikipedia.org/w/index.php?title=Arbitr%C3%A4rgenerator&oldid=167923341.

[32] Tabor Electronics Ltd. *Models WX1284C/WX2184C: Datasheet.*

[33] Tabor Electronics Ltd. *Tabor – WaveXciter AWG.* URL: https://stantronic.com/de/signalquellen/arbitraer-funktionsgeneratoren/tabor-wavexciter-awg/.

[34] Tabor Electronics Ltd. *Models WX1284C / WX2184C: User Manual.* 15. Juli 2015.

[35] National Instruments. *Was ist GPIB?* URL: https://www.ni.com/gpib/d/what_is.htm.

[36] Tektronix Inc. *Datenblatt zur Baureihe AWG5000.* 13. April 2017.

[37] Tektronix Inc. *AWG5000 and AWG7000 Series – Arbitrary Waveform Generators: Programmer Manual.* 22. August 2011.

[38] Tektronix Inc. *AWG70000 Series Data Sheet.* 4. Juni 2012.

[39] Zurich Instruments AG. *HDAWG Arbitrary Waveform Generator: Product leaflet.* Februar 2019.

[40] Zurich Instruments AG. *HDAWG User Manual.* 20. Dezember 2018.

[41] Zurich Instruments AG. *LabOne Programming Manual.* 15. Dezember 2018.

[42] Tektronix Inc. *What is VISA?* URL: https://www.tek.com/support/faqs/what-visa.

[43] SCPI Consortium. *Standard Commands for Programmable Instruments (SCPI) - Syntax and Style.* USA, Mai 1999. URL: http://www.ivifoundation.org/docs/SCPI-99.PDF.

[44] Stefan Dirschnabel. *SOLID - Die 5 Prinzipien für objektorientiertes Softwaredesign.* 2. Mai 2018. URL: https://www.informatik-aktuell.de/entwicklung/methoden/solid-die-5-prinzipien-fuer-objektorientiertes-softwaredesign.html.

[45] Stefan Dirschnabel. *SOLID [4] - Interface Segregation Principle*. 26. Juni 2018. URL: https://www.informatik-aktuell.de/entwicklung/methoden/solid-4-das-interface-segregation-principle.html.

[46] Erich Gamma u. a. *Entwurfsmuster: Elemente wiederkehrender objektorientierter Software*. Übers. von Dirk Riehle. München, 2001. ISBN: 978-3-82-731862-6.

[47] Robert C. Martin. *Design Principles and Design Patterns*. objectmentor.com, 2000.

[48] Wikipedia. *SOLID*. 29. Mai 2019. URL: https://en.wikipedia.org/w/index.php?title=SOLID&oldid=899311582.

[49] Andreas Wintersteiger und Christoph Mathis. *Clean Code*. 12. Juli 2012. URL: https://entwickler.de/online/agile/clean-code-134128.html.

[50] Andre Krämer. *SOLID [2] - Das Single Responsibility Principle*. 29. Mai 2018. URL: https://www.informatik-aktuell.de/entwicklung/methoden/solid-2-das-single-responsibility-principle.html.

[51] Andre Krämer. *SOLID [5] - Das Open Closed Principle*. 4. September 2018. URL: https://www.informatik-aktuell.de/entwicklung/methoden/solid-5-das-open-closed-principle.html.

[52] Wikipedia. *Liskovsches Substitutionsprinzip*. 16. Juli 2019. URL: https://de.wikipedia.org/w/index.php?title=Liskovsches_Substitutionsprinzip&oldid=190469056.

[53] Stefan Dirschnabel. *SOLID [3] - Das Liskov Substitution Principle*. 12. Juni 2018. URL: https://www.informatik-aktuell.de/entwicklung/methoden/solid-3-das-liskov-substitution-principle.html.

[54] Andre Krämer. *SOLID [1] - Das Dependency Inversion Principle*. 15. Mai 2018. URL: https://www.informatik-aktuell.de/entwicklung/methoden/solid-1-das-dependency-inversion-principle.html.

[55] Wikipedia. *Entwurfsmuster*. 25. Oktober 2018. URL: https://de.wikipedia.org/w/index.php?title=Entwurfsmuster&oldid=182142916.

[56] Andrei Boyanov. *Python Design Patterns: For Sleek And Fashionable Code*. Februar 2016. URL: https://www.toptal.com/python/python-design-patterns.

[57] Jasmine Finer. *How to Write Beautiful Python Code With PEP 8*. 19. Dezember 2018. URL: https://realpython.com/python-pep8/.

[58] Guido van Rossum, Barry Warsaw und Nick Coghlan. *PEP 8 – Style Guide for Python Code*. 5. Juli 2001. URL: https://www.python.org/dev/peps/pep-0008/.

[59] Jürgen Dehmer. *Problemlösetechniken: Module (1)*. 4. Juli 2003. URL: https://www.lehrer.uni-karlsruhe.de/~za714/informatik/infkurs/module1.html.

[60] Stephan Augsten. *Was ist Dependency Injection?* 2. April 2019. URL: https://www.dev-insider.de/was-ist-dependency-injection-a-814452/.

[61] Wikipedia. *Baum (Graphentheorie*. 16. Juni 2019. URL: https://de.wikipedia.org/w/index.php?title=Baum_(Graphentheorie)&oldid=189573585.

[62] Mirko Eberlein. *Das Builder Pattern*. 7. Dezember 2017. URL: https://www.digicomp.ch/blog/2017/12/07/das-builder-pattern.

[63] t3n. *Software-Testing: In 4 Schritten zum besseren Software-Entwickler*. 21. Juli 2014. URL: https://t3n.de/news/software-testing-unit-tests-556167/.

[64] Benjamin Aunkofer. *Testverfahren: White-Box vs Black-Box*. 2. August 2009. URL: https://www.der-wirtschaftsingenieur.de/index.php/testverfahren-white-box-vs-black-box/.

[65] Wikipedia. *Softwaretest*. 18. August 2019. URL: https://de.wikipedia.org/w/index.php?title=Softwaretest&oldid=191464368.

[66] it agile. *Was ist Testgetriebene Entwicklung?* URL: https://www.it-agile.de/wissen/agiles-engineering/testgetriebene-entwicklung-tdd/.

[67] Jonathan Seroussi. *How To Automate Device Drivers Testing In IoT Embedded Software Projects*. 4. April 2018. URL: https://medium.com/jumperiot/how-to-automate-device-drivers-testing-in-iot-embedded-software-projects-44c164158f43.

[68] Kritijan Ivancic. *Continuous Integration With Python: An Introduction*. 26. November 2018. URL: https://realpython.com/python-continuous-integration/.

Jül-4421 • Oktober 2019
ISSN 0944-2952

Mitglied der Helmholtz-Gemeinschaft