# Hardwareansteuerung mit Node.JS
# unter Berücksichtigung von Zeitbedingungen

**Hardware control with Node.JS**

**in consideration of time requirements**

Wolfgang Hackenberg

Martikelnummer: 04962209

BACHELORARBEIT

eingereicht im Bachelorstudiengang

INFORMATIK

an der Hochschule für angewandte Wissenschaften München

Wintersemester 2012/2013

Diese Arbeit entstand unter der Betreuung

von

Prof. Dr. Axel Böttcher

Januar bis März 2013

# Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Hochschule für angewandte Wissenschaften München, am 28. März 2013

Wolfgang Hackenberg

# Inhaltsverzeichnis

**Erklärung** ..... ii

**Kurzfassung** ..... v

**Abstract** ..... vi

**Hinweise** ..... vii

**Kapitel 1**
**Einleitung** ..... 1
- 1.1 Aufgabenstellung ..... 1
- 1.2 Motivation ..... 2

**Kapitel 2**
**Grundlagen** ..... 4
- 2.1 ECMAScript ..... 4
- 2.2 Google V8 ..... 5
- 2.3 Node.JS ..... 7
- 2.4 Erweiterungen in Node.JS ..... 10
- 2.5 Parallel-Schnittstelle ..... 11
- 2.6 Flipperautomat ..... 12

**Kapitel 3**
**Eignung von Node.JS** ..... 16
- 3.1 Definition der Grenze zu C++ ..... 16
- 3.2 Test der Geschwindigkeit ..... 16
- 3.3 Parallelschnittstelle unter Linux ..... 18
- 3.4 Anbindung der Parallel-Schnittstelle ..... 20
- 3.5 Test der Parallelschnittstelle ..... 20
- 3.6 Anpassung des Schnittstellenzugriffs ..... 22

**Kapitel 4**
**Probleme mit Node.JS** ..... 24
- 4.1 Übersicht ..... 24
- 4.2 Garbage-Collector der V8 ..... 24
- 4.3 Zeitgeber mit Node.JS ..... 25

**Kapitel 5**
**Architekturentwurf** ..... 29
- 5.1 Abgrenzung zur Implementierung in RTS Java ..... 29
- 5.2 Struktur der Konfiguration ..... 31
- 5.3 Entwurf der Parallelport-Kommunikation ..... 32
- 5.4 Grundstruktur des Architekturentwurfs ..... 33
- 5.5 Schicht HAL ..... 34
- 5.6 Schicht Spiellogik ..... 37
- 5.7 Verwendete Entwurfsmuster ..... 40

**Kapitel 6**
**Untersuchungen zu Coding Standards** ..... **46**

**Kapitel 7**
**Referenzimplementierung** ..... **53**
7.1 Implementierung HAL ..... 53
7.2 Konfiguration HAL ..... 61
7.3 Implementierung Spiellogik ..... 63
7.4 Konfiguration Spiellogik ..... 66
7.5 Fehlerbehandlung ..... 70
7.6 Testen ..... 71

**Kapitel 8**
**Zusammenfassung** ..... **74**
8.1 Vorteile und Nachteile ..... 74
8.2 Fazit ..... 75
8.3 Ausblick ..... 76

**Literaturverzeichnis** ..... **77**

# Kurzfassung

Diese Bachelorarbeit untersucht die Eignung der *JavaScript*-Plattform *Node.JS* zur Ansteuerung von Hardware unter Zeitbedingungen. Hierbei kommt *JavaScript* im für die Sprache ungewöhnlichen Kontext der *Embedded Systems* zum Einsatz. Mit Entwurf und Implementierung einer Architektur, die als Steuerung eines Flipperautomaten vom Typ *Pinball 2000* der Firma *Williams Electronics Games Inc.* dient, konnte die Realisierbarkeit einer solchen Anwendung mit *JavaScript* erfolgreich nachgewiesen werden.

Der erste Teil der Arbeit beschäftigt sich mit der Evaluierung der dieser Arbeit zu Grunde liegenden Technologien und identifiziert mögliche Problemfelder. Im Rahmen dieser wird die grundsätzliche Eignung der Plattform *Node.JS* untersucht und die Anbindung des Parallelports entworfen. Dieser dient als Kommunikationsschnittstelle zur Platine des Flipperautomaten. Für die auftretenden Probleme, wie die Präzision der Zeitgeber und die Ansteuerung des Parallelports, wurden eigene Lösungen implementiert.

Anschließend wurde ausgehend von den Ergebnissen der Voruntersuchung eine Architektur in *JavaScript* entworfen. Hier wurde eine schlanke Zwei-Schicht Architektur gewählt, bei der jede Schicht um eine zentrale Aufgabe der Steuersoftware gestaltet ist: Eine Schicht dient der Hardwareabstraktion und eine Schicht zur Ausführung der Spielregeln. Die Kommunikation zwischen den Schichten erfolgt über eine kompakt gehaltene API. Als zentrale Entwurfsmuster wurden *State Pattern*, *Build Pattern* und *Command Pattern* verwendet.

Im letzten Abschnitt der Arbeit wird die Implementierung und Konfiguration der Steuersoftware beschrieben. Bei Gestaltung der Regelstruktur wurde insbesondere auf die Testbarkeit der Regeln selbst geachtet. Ergänzend zur eigentlichen Software wurden Tests für beide Schichten und zum Testen der Kommunikation mit dem Flipperautomaten erstellt. Zusätzlich wurden einige Beispielregeln entworfen, um korrektes Initialisieren und Ausführen der hinterlegten Regelkonfigurationen zu zeigen. Ergebnis der Arbeit ist eine funktionsfähige und frei konfigurierbare Steuersoftware.

# Abstract

With this bachelor thesis the suitability of the *JavaScript* platform *Node.JS* for hardware control in consideration of time requirements is examined. *JavaScript* is used in an unusual context of *embedded systems*. An architecture to control a pinball machine *Pinball 2000* by *Williams Electronics Games Inc.* was designed and implemented. With this implementation, the feasibility of such a software written in *JavaScript* could be demonstrated successfully.

The first part of the thesis evaluates the underlying technologies and identifies the potential problems. The fundamental suitability of the platform *Node.JS* should be investigated. The communication interface should be created utilizing the parallel port. For occurring problems like the precision of the timer or the access to the parallel port, own solutions were implemented.

Based on the results of the evaluation an architecture was designed in *JavaScript*. A simple two-layer architecture was chosen. Each layer is focused on one central aspect of the control software: One layer serves as hardware abstraction while the other executes the game rules. A slim API is utilized for communication between those layers. *State pattern*, *build pattern* and *command pattern* were used as primary *design patterns*.

The final part describes the implementation and configuration of the control software. In addition to the control software, tests for both abstraction layers and for the communication with the pinball were implemented. Additionally, some basic rules were drafted to demonstrate the correct initialization and execution of the rules of the game. The outcome of this thesis is a fully operational and freely configurable control software.

# Hinweise

Fachbegriffe wurden soweit sinnvoll ins Deutsche übertragen. Wurde der Begriff aus der englischsprachigen Fachliteratur übernommen, oder ist die deutsche Übersetzung eines Fachbegriffs unüblich, wurden die Begriffe in der Originalsprache belassen.

*Eigennamen,* (englische) *Fachbegriffe* sind in der Arbeit *kursiv* gesetzt. **Klassen** und **Module**, sowie **Funktionen** und **Methoden** nebst deren **Parametern**, sind in **Fettschrift** gedruckt.

# Kapitel 1

# Einleitung

## 1.1 Aufgabenstellung

Ziel dieser Arbeit ist es, die Eignung der Technologie *Node.JS* [1] für die zeitkritischen Ansteuerung von Hardware zu untersuchen. Bei *Node.JS* handelt es sich um eine für Netzwerkapplikationen gedachte *JavaScript*-Plattform. Als Anschauungsobjekt für die Untersuchung dient die Flipper-Plattform *Pinball 2000* des Herstellers *Williams Electronics Games Inc.* Der Einsatz von *JavaScript* [2] im Umfeld von *Embedded Systems* ist ungewöhnlich, die Machbarkeit einer solchen Lösung daher im Rahmen dieser Arbeit zu untersuchen. Um die Eignung der Technologie nachzuweisen, soll die in der Arbeit von Herrn Kratzer [3] mittels der *Real-Time Specification for Java* [4] erstellte Software in *JavaScript* umgesetzt werden.

Die im Rahmen dieser Arbeit untersuchte Aufgabe umfasst drei Teile: In einem ersten Schritt soll die grundsätzliche Eignung der verwendeten Technologien für diese Aufgabe untersucht werden. Diese Untersuchung ist notwendig, da die Plattform *Node.JS* einerseits, mit ihrem ereignisorientiertem Aufbau und ihrer Optimierung für sehr hohen Datendurchsatz, auch in abweichenden Anwendungsfällen Potential verspricht. Auf der anderen Seite ist sie nicht für Echtzeitfähigkeit entworfen, was ihren Einsatz unter Zeitbedingungen schwierig erscheinen lässt. Auch die Speicherverwaltung mit *Garbage Collection* kündigt sich als potentielle Problemstelle an. Diese Vor- und Nachteile sollen im Rahmen des ersten Teils untersucht und abgewogen werden.

Im Anschluss an diese Voruntersuchung ist in einem zweiten Schritt die Schnittstelle zur Steuerplatine des Flipperautomaten zu erstellen. Diese Schnittstelle soll als *C++*-Erweiterung für die Plattform *Node.JS* erstellt werden. Hierbei soll möglich viel der Funktionalität direkt in *JavaScript* implementiert werden. Es ist zu untersuchen, inwieweit sich dieses Ziel realisieren lässt, und wo genau die Trennung zwischen *JavaScript*- und *C++*-Funktionalität vorzunehmen ist.

Als letzter Schritt soll eine Referenzimplementierung in *JavaScript* erstellt werden, die auf der Plattform *Node.JS* zur Ausführung kommt. Hierfür soll eine geeignete Architektur entworfen werden, wobei die folgenden Zielen relevant sind. Diese stellen eine Variation der Ziele der ursprünglichen Implementierung in *RTS Java* dar: [3, Kapitel 3]

- **Zeitbedingungen:** Der Flipperautomat gibt einerseits mit dem *Watchdog*[1] eine technisch bedingte Zeitbedingung vor. Andererseits ergeben sich zusätzliche Zeitbedingungen auch physikalisch aus dem Spielablauf, wie etwa durch die hohe Geschwindigkeit der Kugel. Diese sind unbedingt einzuhalten, um einen Spielbetrieb zu ermöglichen.
- **Konfiguration:** Durch Konfiguration von Hardwareaufbau und Spiellogik soll die Steuersoftware flexibel, ihre Implementierung zudem einfach erweiterbar sein.
- **Schlankheit:** Die Softwareplattform soll überschaubar und insbesondere die Regel-Konfiguration kompakt bleiben. Dies soll es später ermöglichen, auch hochkomplexe Spielregeln zu konfigurieren. Hierzu sollen die Möglichkeiten, die *ECMAScript* bietet, genutzt werden, um eine Verbesserung im Vergleich zur bisherigen Regelimplementierung zu erzielen [3, Kapitel 6]
- **Testbarkeit:** Die Implementierung und die konfigurierten Spielregeln müssen einfach testbar sein.

Die grundsätzliche Eignung der Lösung zur Ansteuerung der *Pinball 2000* Plattform kann nun mit einigen Beispielregeln für den Flipperautomaten untersucht werden. Eine kompletter Regelsatzes würde aufgrund des enormen Umfangs den Rahmen dieser Arbeit sprengen, folglich ist das Erstellen der Spielregeln nicht Teil dieser Arbeit.

## 1.2 Motivation

Im Zuge des Trends zum Einsatz von Webapplikationen als plattformunabhängige Anwendungen wird der Skriptsprache *JavaScript* zunehmend Aufmerksamkeit zu Teil. Eine steigende Komplexität dieser Anwendungen führt zusammen mit der vergleichsweise geringen Rechenleistung von Mobilgeräten zu immer höheren Anforderung an die Effizienz der Ausführungsumgebungen, wie auch an die der in

1 Der *Watchdog* setzt einen *Blanking* [5] genannten Mechanismus ein, der weite Teile des Flippers von der Stromversorgung nimmt, wenn die erwartete Kommunikation mit dem Steuerrechner nicht so häufig wie erwartet erfolgt.

dieser Arbeit eingesetzten *V8-Engine* [6]. Durch die gesteigerte Effizienz wird *JavaScript* wiederum für neue Aufgabenfelder interessant. So wurde während dem Erstellen dieser Arbeit *JavaScript* als Hauptsprache zum Erstellen von Applikationen für die Desktopumgebung *GNOME* ausgewählt [7]. Und mit *Node.JS* findet *JavaScript* auch serverseitig zunehmend Anwendung und wird in Unternehmen wie *LinkedIn Corporation*, *Microsoft Corporation* und *Yahoo! Inc.* eingesetzt [1].

Im Rahmen dieser Bachelorarbeit soll anhand der Ansteuerung eines Flipperautomaten vom Typ *Pinball 2000* untersucht werden, ob eine weitere Ausweitung des Einsatzgebietes der Sprache *JavaScript* möglich ist. Kann die die Technologie zum aktuellen Stand auch im Umfeld mit Zeitbedingungen zum Einsatz kommen?

# Kapitel 2

# Grundlagen

## 2.1 ECMAScript

### Geschichte

Die Skriptsprache *JavaScript* wurde 1995 von Brendan Eich für Netscape entwickelt. Im 1996 veröffentlichten Webbrowser *Netscape Navigator 2.0* war *JavaScript* erstmals enthalten. [8] Microsoft adaptierte für ihren Webbrowser *Internet Explorer* eine Variante der Skriptsprache unter dem Namen *JScript* [9].

Im November 1996 startete die Entwicklung eines Industriestandards unter dem Namen *ECMAScript*, der in erster Version im Juni 1997 fertiggestellt wurde. Im April 1998 wurde er als ISO/EIC 16262 als ISO-Norm veröffentlicht. Der Standard wird im Juni 2002 um viele Sprachelemente erweitert. Die dritte Version des Standards spezifiziert ECMA 5. Sie integriert vor allem viele Spracherweiterungen in den Sprachkern, die in verschiedenen Implementierungen des Standards diesen erweitert hatten [2].

### Einsatzzweck

Die Spezifikation erläutert den gedachten Einsatzzweck von *ECMAScript* als Skriptsprache wie folgt:

> “A ***scripting language*** is a programming language that is used to manipulate, customise, and automate the facilities of an existing system. In such systems, useful functionality is already available through a user interface, and the scripting language is a mechanism for exposing that functionality to program control.” [2, Seite 2]

*ECMAScript* ist zur Einbettung in ein Host-System gedacht, weshalb sie keinerlei eigene Funktionen zur Ein- und Ausgabe beinhaltet. Derartige Funktionen sollen von der Ausführungsumgebung in Form von *host*-Objekten zur Verfügung gestellt werden. Auch wenn die Sprache als Skriptsprache für clientseitige Internetanwendungen gedacht ist, soll die Verwendung nicht auf diesen Zweck beschränkt sein. Aus diesem

Grund ist der Sprachstandard allgemein gehalten und nimmt keine Rücksicht auf spezifische Sprachelemente der Ausführungsumgebung. Zu solchen spezifischen Sprachelementen gehören Objekte, die beispielsweise das Fenster oder einzelne Seitenelemente wie Textboxen in einem Webbrowser repräsentieren. Aber auch Webbrowser-spezifische Ereignisse wie Mausklicks auf einzelne Elemente sind Teil dieser Erweiterung [2]. Die Verwendung von serverseitigem *Scripting* wird im Standard als Möglichkeit explizit erwähnt:

> "A web server provides a different host environment for server-side computation including objects representing requests, clients, and files; and mechanisms to lock and share data." [2, Seite 2]

Dies ist der Weg, den *Node.JS* wählt.

## 2.2 Google V8

Im Zuge der Entwicklung des eigenen Webbrowsers *Chrome* hat *Google Inc.* mit der V8 eine quelloffene *JavaScript*-Ausführungsumgebung gemäß den *ECMAScript*-Spezifikationen geschaffen. Sie ist in *C++* implementiert und kann einfach in andere *C++*-Anwendungen auf einer Vielzahl an Rechnerplattformen eingebettet werden [6]. Aufgrund der durch zahlreiche Optimierungen erreichten Ausführungsgeschwindigkeit ist es *Google Inc.* möglich, einen Teil der V8 selbst in *JavaScript* zu implementieren [10, Kapitel 8].

Um eine hohe Ausführungsgeschwindigkeit zu erzielen, kommen unter anderem folgende Techniken zum Einsatz, die im Anschluss noch kurz erläutert werden sollen [11]:

- *Hidden Classes* zum schnellen Nachschlagen von Objektfeldern
- *Just-in-Time Compiler* mit Maschinencodegenerierung
- Effiziente *Garbage Collection*

#### Hidden Classes

Obwohl *JavaScript* als objektorientierte Sprache dem Prototyp-Ansatz folgt, also Objekte ohne Klassenkonzept direkt voneinander erben lässt, erzeugt die *V8* bei Ausführung im Hintergrund Klassen für alle in *JavaScript* angelegten Objekte. Bei diesen Klassen handelt es sich um eine Informationsstruktur, die fortan mit der Objektinstanz verknüpft ist. Die Informationen dienen der Ausführungsumgebung im folgenden zur Optimierung der Objektzugriffe: In ihnen ist gespeichert, an welchen

Stellen die einzelnen Attribute im Speicher hinterlegt sind. Diese Informationen werden vom *Just-in-Time Compiler* zur Optimierung des Feldzugriffs eingesetzt. Da es in *JavaScript* möglich ist, jederzeit neue Felder in einem Objekt anzulegen, muss die *Hidden Class* bei Bedarf durch eine neue Klasse ersetzt werden. Dieses Verhalten führt zu erhöhten Kosten bei der Manipulation von Objekten, wird aber durch den Umstand abgemildert, dass einmal erzeugte *Hidden Classes* bei Bedarf wiederverwendet werden [11].

**Just-in-Time Compiler**

Der *Just-in-Time Compiler* kann mithilfe der *Hidden Class* Maschinencode erzeugen, in welchem die Zugriffe auf Objektfelder hart kodiert sind. Für eine Funktion wird ein derartiger Block beim ersten Aufruf generiert. Im Zuge dieses Vorgangs werden die *Hidden Classes* der als Funktionsparameter übergebenen Objekte ausgewertet. Zugriffe auf die Felder der Objekte können mit den Informationen aus den *Hidden Classes* im Bytecode direkt in Form von Adressabständen hinterlegt werden. Für einen einmal auf diese Weise erzeugten Bytecode-Block muss bei Ausführung lediglich noch überprüft werden, ob die zur Erzeugung verwendeten *Hidden Classes* mit den *Hidden Classes* der zur Laufzeit vorliegenden Objekte übereinstimmen. Diese Überprüfung wird ebenfalls in den Maschinencode integriert, indem die Identifikatoren der Klassen verglichen werden. Ergebnis ist eine extrem hohe Ausführungsgeschwindigkeit, solange vom Aufbau identische Objekte verarbeitet werden. Wird ein Objekt mit anderer Struktur und damit einer abweichenden *Hidden Class* verarbeitet, wird neuer Maschinencode unter verwendung der geänderten *Hidden Class* erzeugt [11].

**Speicherverwaltung**

Die Speicherverwaltung dient dazu, die in *ECMAScript* erzeugten Variablen im Arbeitsspeicher anzulegen. Hierzu muss der in ISO/EIC 16262 [2] definierte Gültigkeitsbereich der Variablen (*Scope*) umgesetzt werden. In *ECMAScript* erzeugt jeder Funktionsblock einen eigenen Namensraum. Ein korrekter Umgang mit der Speicherverwaltung wird wichtig, wenn *Add-ons* für die V8 Umgebung erstellt werden, um so die Möglichkeiten von *Node.JS* zu erweitern [12]. In einem solchen Add-on können in *C++* Objekte angelegt und im Namensraum von JavaScript eingeblendet werden. Hierbei ist es wichtig, die manuell angelegten Instanzen korrekt in die Speicherverwaltung einzubinden. Aus diesem Grund wurden die folgenden Ausführungen trotz ihres technischen Charakters in die Arbeit aufgenommen.

Um die Speicherverwaltung mit Gültigkeitsbereichen zu realisieren, wird für jede Variable in *ECMAScript* ein *C++*-Objekt **Handle** erzeugt, dass den Speicherverweis mit dem Inhalt der Variablen verwaltet. Die *Handle* von lokale Variablen werden in **HandleScope**-Instanzen verwaltet, ihre Lebensdauer ist an diese gebunden. Wird ein *Handle* weder als für eine interpretierte *ECMAScript*-Funktion automatisch erzeugt, noch manuell in einem im *C++*-Code erzeugten *Scope* verwaltet, muss er als *persistent handle* angelegt werden. *Handles* und *Scopes* werden in einem Ausführungskontext angelegt. Bei einem Webbrowser hätte beispielsweise jedes Tab einen eigenen Kontext, damit jede dargestellte Internetseite unabhängig voneinander *ECMAScript* ausführen kann. Es erfolgt keine explizite Verknüpfung von *Handles* und *Scopes*. Dies geschieht implizit im Hintergrund über *Stacks*, d.h. ein neu erzeugter *Handle* wird automatisch dem zuletzt erzeugtem **HandleScope** hinzugefügt [13].

#### Garbage Collection

Bei der *Garbage Collection* werden die im vorherigen Abschnitt beschriebenen *Handles* herangezogen. Ein Objekt im Arbeitsspeicher, auf das kein *Handle* mehr verweist, kann gelöscht werden um den von ihm belegten Speicher freizugeben. Bei der *V8* kommt ein *Garbage Collector* zum Einsatz, der die gesamte Ausführungsumgebung anhält [11]. Dieses Verhalten kann sich für der im Rahmen dieser Bachelorarbeit untersuchten Aufgabenstellung als problematisch erweisen. Potentielle Probleme an dieser Stelle werden im Kapitel 4 näher beschrieben.

## 2.3 Node.JS

Bei *Node.JS* handelt es sich um eine Plattform zur Ausführung von Netzwerkapplikationen. Gestartet wurde die Entwicklung von Ryan Dahl [14, Vorwort Ryan Dahl]. Die Plattform wurde ereignisgestützt und nicht-blockierend gestaltet, um eine hohe Geschwindigkeit und eine gute Skalierung zu erreichen. Anwendungen für die Plattform werden in *JavaScript* implementiert:

> “Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.” [1]

Die Plattform ist für Anwendungen gedacht, die von der Möglichkeit profitieren können, viele Verbindungen zu Clients offenzuhalten, um diesen bei Bedarf Daten aktiv Daten übermitteln (*push*) zu können. Hierfür war der klassische Serveransatz,

der für jede Verbindungen einen *Thread* mit damit einhergehenden Speicherverbrauch erzeugt, aufgrund der damit verbunden schlechten Skalierung nur unzureichend geeignet. Um eine hohe Ausführungsgeschwindigkeit zu erreichen, wurden einige grundlegenden Designentscheidungen getroffen: Kern von *Node.JS* ist eine einzelne Ereigniswarteschlange, in die ausschließlich nicht-blockierende Ereignisse zur Verarbeitung eingereiht werden. Blockierende Aufgaben hingegen werden im Hintergrund bearbeitet. Sobald deren Ergebnisse vorliegen werden die Aufgaben wieder in der Warteschlange berücksichtigt. Durch diese Auslagerung bleibt die Geschwindigkeit bei Abarbeitung der Ereigniswarteschlange stets hoch [16][2] [14, Vorwort Ryan Dahl & Kapitel 1].

**Aufbau**

Ryan Dahl bezeichnet die Plattform als eine Sammlung von *Bindings* für *Googles V8 JavaScript*-Ausführungsumgebung. Dies zeigt sich, wenn man den grundsätzlichen Aufbau von *Node.JS* betrachtet: Als *JavaScript*-Laufzeitumgebung kommt Googles *V8* zum Einsatz. In der *C*-Bibliothek *libuv* bzw. deren Teilkomponenten *libeio* und *libev* werden die plattformabhängigen Implementierungen wie Ereigniswarteschlange und Zeitgeber gebündelt. Des weiteren wird hier ein *Thread Pool* für blockierende Aufgaben wie Dateizugriffe und DNS-Auflösung vorgehalten. Darauf setzten die *Bindings* auf, die diese Funktionen in Form von Objekten und Funktionen im Namensraum von *JavaScript* zur Verfügung stellen. Selbst in *JavaScript* implementiert hingegen ist die *Standard Library* [17]. Diese sorgt für Grundfunktionalität für Netzwerkprotokolle und Dateizugriff.

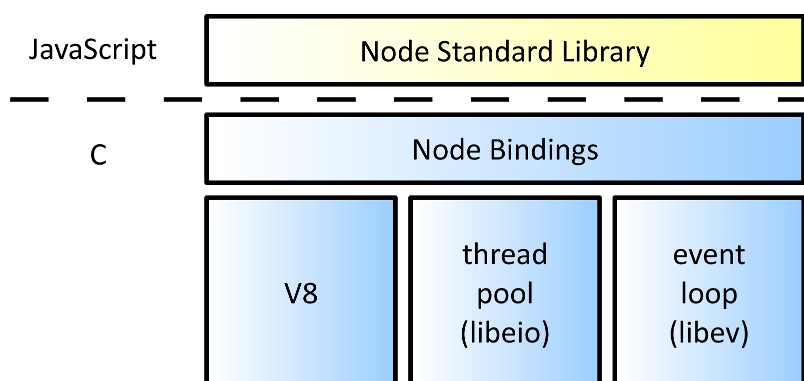



**Abbildung 1:** Übersicht der Architektur von *Node.JS* nach Ryan Dahl [16, Folie 23].

2 Zu Ryan Dahls Einführungsvorträgen existieren zahlreiche Videomitschnitte, beispielsweise kann unter unter [15] der Mitschnitt des im Rahmen der *Cinco de Node.js* am 05.05.2010 gehaltenen Vortrags abgerufen werden.

Die Implementierung von Anwendungen für die *Node.JS*-Plattform erfolgt in *JavaScript*. Diese Sprache wurde von Ryan Dahl gewählt, da diese Eigenschaften aufweist, die sie zur Verwendung mit einer auf Ereigniswarteschlangen gestützten Architektur geeignet macht. Anonyme Funktionen und *Closures*[3] ermöglichen ein komfortables Implementieren von Rückruffunktionen, desweiteren erfolgt bereits die Interaktion innerhalb eines Webbrowsers ereignisgestützt. Diese Syntax konnte innerhalb von *Node.JS* weiterverwendet werden. Auch der Umstand, dass es in *JavaScript* noch keine Mechanismen für Datei- und Netzwerkzugriffe gibt, machte die Sprache interessant, um genau diese Funktion gemäß der Anforderungen von Node.JS einzuführen [16] [14, Vorwort Ryan Dahl].

Die in *C* oder *C++* geschriebenen Kernkomponenten werden über *Bindings* im Namensraum von *JavaScript* bekannt gemacht und können wie nativ in *JavaScript* implementierte Funkionen verwendet werden. Diese *Bindings* entsprechen jenen, die man auch für selbst entwickelte Erweiterungen anlegen kann, und werden in einem folgenden Abschnitt näher behandelt.

#### Asynchrone Kommunikation

Bedeutendes Gestaltungsmerkmal von *Node.JS* ist eine nicht-blockierende Ereigniswarteschlange, die stets schnell auf anstehende Aufgaben reagieren kann. Da die Ereigniswarteschlange in einem einzelnen *Thread* bearbeitet wird, dürfen weder Netzwerkkommunikation noch etwaige zusätzliche Ein-/Ausgabe-Operationen im Betrieb als Webserver blockieren. Die blockierenden Tätigkeiten werden stattdessen für den Entwickler transparent im Hintergrund unter Verwendung des *Thread-Pools* durchgeführt. Aus *JavaScript* ist lediglich der Zugriff auf den Ausführungsthread der *V8*-Umgebung möglich (*WebWorker*[4] seien hier unberücksichtigt), während die C-Schicht Zugriff auf den *Thread-Pool* und damit auf mehrere *Threads* hat. Dadurch reicht es laut Ryan Dahl, wenn der *Node.JS*-Entwickler sein Augenmerk auf das Vermeiden von blockierendem Code richtet, während klassische Probleme der Nebenläufigkeit von ihm unberücksichtigt bleiben können [16]. Eine Möglichkeit, blockierende Funktionen zu verhindern, ist deren asynchroner Aufruf: Eine Ein-/Ausgabe-Methode wartet nicht bis sie das Ergebnis der Operation liefern kann. Statt dessen wird ihr beim Aufruf eine Rückruffunktion mitgegeben, welche das Ergebnis verarbeiten kann. Die Methode selbst kehrt sofort zurück. Sobald die Daten vorhanden sind, wird die Rückruffunktion in die Ereigniswarteschlange eingereiht

3 Eine gute Erläuterung zu Closures findet sich in „JavaScript: The Good Parts“ von Douglas Crockford [18, Kapitel 4].

4 Siehe *node-worker* als Umsetzung der *WebWorker* API für *Node.JS* [19].

und anschließend zeitnah ausgeführt [14, Kapitel 3]. Ermöglicht werden Rückruffunktionen durch den Umstand, dass Funktionen in *JavaScript* Objekte sind. Dies ist einer der Ansätze der Sprache, der dem funktionalem Paradigma folgt. [20, Kapitel 8]

Zusätzlich zur Rückruffunktion ist noch ein zweiter Mechanismus zur asynchronen Verarbeitung fester Bestandteil von *Node.JS*: *JavaScript*-Objekte können selbst Ereignisse aussenden, auf die Interessenten hören können. In der nativen API kommen Rückruffunktionen meist dann zum Einsatz, wenn es um einmalige Weiterverarbeitung von Daten geht. Ein Beispiel hierfür wären das Lesens eines Blocks aus einer Datei [21]. Der Ereignis-Mechanismus kommt hingegen meist dort zum Einsatz, wo regelmäßig wiederkehrenden Nachrichten verarbeitet werden müssen. Als Beispiel hierfür sei die Verarbeitung von Daten beim Lesen aus einem *Streams*, der das Einlesen jedes Blocks als Ereignisse kommuniziert [22].

## 2.4 Erweiterungen in Node.JS

*Node.JS* sieht zwei Arten der Erweiterung vor: Zum einen können Add-ons in *JavaScript* implementiert werden. Reichen die damit erzielten Möglichkeiten nicht aus, kann die Erweiterung auch in *C++* geschrieben. Mittels *Bindings* können Funktionen im Namensraum von *JavaScript* zur Verfügung gestellt werden. Hierbei wird auf das Erweiterungssystem der *V8*-Ausführungsumgebung zurückgegriffen [14, Kapitel 8].

Die in *JavaScript* geschriebenen Erweiterungen von *Node.JS* heißen Module. Bei Modulen handelt es sich um Dateien mit *JavaScript*-Code. Diese haben jeweils einen eigenen Namensraum und müssen explizit definieren, welche Variablen und Funktionen von außen sichtbar sein sollen [23, Kapitel 3].

Die Erstellung von Add-ons in *C++* stellt die zweite Möglichkeit der Erweiterung dar. Diese ist aber noch komplex, was auch die Entwickler von Node.JS so kommunizieren:

> “The API (at the moment) is rather complex, involving *knowledge* of several libraries.” [12]

Damit eine *C++*-Funktion innerhalb der *JavaScript*-Ausführungsumgebung korrekt ausgeführt und innerhalb des *JavaScript*-Namensraums eingebettet werden kann, muss sich diese wie eine *JavaScript*-Funktion verhalten. Hierfür muss das Plug-in folgende Aufgaben erfüllen [12]:

- Funktionen müssen mittels *Bindings* in der *V8*-Ausführungsumgebung registriert werden.
- *Context* und *Scope* für die Funktion müssen passend erzeugt und geschlossen werden.
- Ein- und Ausgabeparameter müssen korrekt mit *Wrappern* ausgestattet oder aus diesen entnommen werden.
- Eine Vielzahl an möglichen Fehlern muss abgefangen und sinnvoll weiterverarbeitet werden.

Um dem Entwickler bei diesen Definitionen zu helfen, existieren mit *V8-Juice* [24] und *V8U* [25] unabhängig voneinander zwei Hilfsbibliotheken. Letztere unterstützt mit vordefinierten *C++*-Makros das Erstellen von Erweiterungen. Mithilfe dieser Makros ist es möglich, ein Containerobjekt zu erzeugen, dass die Funktionalität in *C++* implementiert und diese in der *V8* als *JavaScript*-Funktionen registriert. Aufrufe dieser Funktionen werden auf die in der Erweiterung implementierten *C++*-Funktionen abgebildet.

Durch diese Form der Erweiterung ist es möglich, in *Node.JS* Funktionalität einzubringen, welche rein durch Erweiterungen in *JavaScript* nicht realisierbar ist. Dadurch wird es im Laufe dieser Arbeit möglich sein, den Parallel-Port aus *JavaScript* heraus anzusprechen.

## 2.5 Parallel-Schnittstelle

Bei der Parallelschnittstelle handelt es sich um eine Computerschnittstelle, die in Spezifikation *IEEE 1284-1994* [26] definiert ist. Sie besteht aus drei 8-Bit Ports. Der erste ist ein *Data-Port*. Er ist mit Pin 2-9 des physischen Steckverbinders verbunden und kann bidirektional benutzt werden. Der zweite Port *Status* stellt Statusinformationen lesend zur Verfügung. Diese Pins sind für druckerspezifische Signale wie „*Out of Paper*“ gedacht. Der dritte Byte-Port ist das Register *Control*. Es dient zur Steuerung der Kommunikation über die Datenleitungen [27, Kapitel 9].

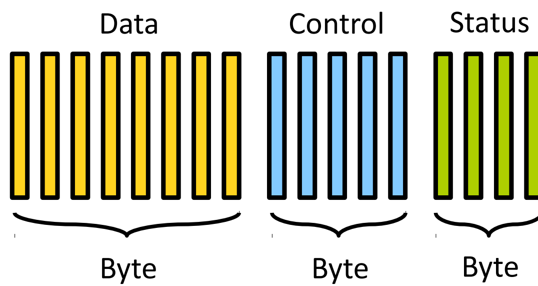



**Abbildung 2:** Die Leitungen der Parallelschnittstelle lassen sich als drei Byte-Blöcke interpretieren.

Neben der Möglichkeit, festgelegte Protokolle zur Ansteuerung von Druckern und anderen Geräten zu verwenden, besteht die Möglichkeit, den Status aller Leitungen manuell zu kontrollieren. Dies ermöglicht einen sehr universellen Einsatz der Schnittstelle. Bei dem im folgenden vorgestellten Flipperautomaten wurde der Parallel-Port als Schnittstelle zur Ansteuerung gewählt.

## 2.6 Flipperautomat

Bei dem in dieser Arbeit verwendeten Flipperautomaten handelt es sich um ein Gerät vom Typ *Star Wars Episode 1 Pinball 2000* von *Williams Electronics Games Inc.* [28]. Die Ansteuerung erfolgt durch einen im Gehäuse integrierten Computer, der über die Parallelschnittstelle mit einer Ansteuerungsplatine kommuniziert. Auf dieser werden mit diskreten Logikbauteilen die übertragenen Befehle zur Ansteuerung der einzelnen Komponenten des Flippers umgesetzt [29]. Der verbaute Steuercomputer mit der originalen Steuersoftware wird im Rahmen dieser Arbeit komplett ersetzt.

**Abbildung 3:** Spielfläche des Flippers *Star Wars Episode 1 Pinball 2000*.

#### Aufbau

Bei den im Rahmen dieser Arbeit angesteuerten Komponenten handelt es sich um Schalter, Spulen und Lampen. Schalter dienen zur Eingabe von Benutzerbefehlen ebenso wie zur Bestimmung der Position der Kugel auf dem Spielfeld. Hierfür sind an für den Spielablauf wichtige Stellen Taster in das Spielfeld eingebracht. Die Spulen dienen als Ausgabeelemente zur Manipulation von Kugel und Spielfläche. Neben der Möglichkeit, die Kugel an bestimmten Stellen festzuhalten, können auch Begrenzungen in das Spielfeld bewegt oder Kugel an *Bumpern* abgestoßen werden. Auch die vom Spieler kontrollierten Flipper-Finger werden über Spulen angesteuert. Die Lampen dienen als zweiter Typ von Ausgabekomponenten zur Visualisierung von Spielzielen und dem aktuellen Status des Spiels. Eine ausführlichere Beschreibung des Flipperautomaten findet sich in der Arbeit von Herrn Kratzer auf den Seiten 11 bis 14 [3, Kapitel 2].

#### Kommunikation

Die Kommunikation mit der Ansteuerplatine des Flipperautomaten erfolgt registerbasiert. Das heißt, dass der Flipper über eine Vielzahl an Registern verfügt, wobei jedes Register eine festgelegte Funktion hat. Register sind hierbei entweder zum Lesen oder zum Schreiben vorgesehen, eine Mischbelegung existiert nicht [5].

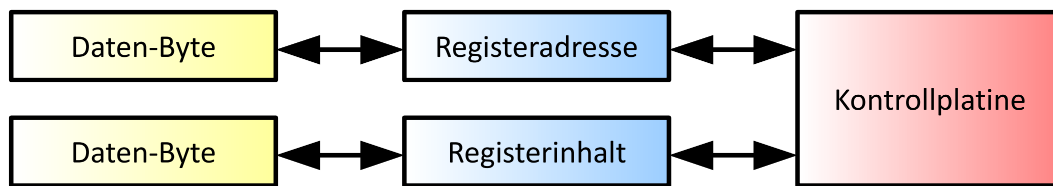



**Abbildung 4:** Abstraktionsebenen bei der Kommunikation mit der Steuerplatine des Flipperautomaten.

Der Zugriff auf die Register erfolgt über die Parallelschnittstelle, wobei Registeradresse und Registerwert über die Datenleitungen übertragen werden, während eine Koordination der Kommunikation über die Steuerleitungen erfolgt [3, Kapitel 2].

Die Kommunikation mit der Steuerplatine des Flippers lässt sich in zwei Aufgaben unterteilen:

- regelmäßiges Lesen des aktuellen Status der Eingabeelemente
- regelmäßiges Schreiben des gewünschten Zustands aller Ausgabeelemente

Die verschieden Komponenten des Flipperautomaten sind hierbei unterschiedlich in den Registern organisiert [3, Kapitel 2]:

- Lampen in Form einer 16x8 Bit-Matrix
- Schalter in einer 8x8 Bit-Matrix und teils in Einzelregistern
- Spulen grundsätzlich in Einzelregistern

Jedes Bit in einer Matrix oder einem Register entspricht dem Zustand eines bestimmten Bauteils des Flipperautomaten. Dieser kann bei Eingabekomponenten geschrieben, bei Ausgabekomponenten gelesen werden.

#### Matrix-Zugriff

Während bei Komponenten, die direkt über Register angesprochen werden, durch Lesen oder Schreiben eines Registers der Wert für acht Bauteile des Flipperautomaten gelesen oder geschrieben wird, gestaltet sich der Zugriff auf die Matrizen komplizierter. Für das Ansprechen einer Matrix stehen jeweils Spalten- und Zeilenregister zur Verfügung. Diese sind für jede Matrix fest definiert. Im Spaltenregister wird ein einzelnes Bit auf eins gesetzt, um eine Spalte der Matrix auszuwählen. Soll explizit keine Spalte ausgewählt werden, muss dieses Register mit null beschrieben werden. Anschließend enthält das Zeilenregister der jeweiligen Matrix den Inhalt der über das Spaltenregister ausgewählten Spalte. Dieser kann je

nach Bauteiltyp ausgelesen oder neu beschreiben werden. Da die Register nur ein Byte bereit sind, existieren im Falle der Lampenmatrix zwei Zeilenregister, um den Zustand der 16 Lampen einer Lampenmatrixzeile zu setzen [3, Kapitel 2].

**Zeitliche Anforderungen**

Je nach Bauteil sind bei der Ansteuerung unterschiedliche Anforderungen zu erfüllen:

- Schalter müssen regelmäßig ausgelesen werden, um die Anforderungen des *Watchdogs* von Seiten des Flipperautomaten zu erfüllen. Diese Schaltung auf der Hardware-Platine des Flippers deaktiviert sämtliche Ein-/Ausgabe wenn innerhalb von 1,5 ms kein Zugriff auf das Spaltenregister der Schaltermatrix erfolgt [3, Kapitel 2]. Des weiteren ist eine häufige Aktualisierungsrate wichtig, um die Reaktionsgeschwindigkeit hoch zu halten. Aus diesem Grund sollen in jedem Schritt der Steuersoftware alle Schalter ausgelesen werden.
- Änderungen an den Zuständen der Spulen werden sofort in die Register geschrieben, um eine möglichst hohe Reaktionsgeschwindigkeit zu erreichen und die Kugeln im Spiel so zuverlässig zu manipulieren.
- Lampen sind über eine Matrix verdrahtet. Aus diesem Umstand ergibt sich ein Pulsbetrieb: Jeweils eine Lampenreihe wird je Schritt der Steuersoftware angesteuert. Durch Überspannung leuchten die aktivierten Lampen lange genug, bis die jeweilige Reihe bei erneutem Durchlauf der Lampenmatrix erneut angesteuert wird. Während die Werte für die neue Lampenreihe in die Register geschrieben werden, wird das Spaltenauswahlregister mit null beschrieben. So wird während des Schreibens der Zeilenregister keine Lampe aktiv angesteuert. Nachdem der Status der Reihe korrekt geschrieben wurde, wird die neue Reihe der Lampenmatrix über das Spaltenregister aktiviert. Entsprechend der aus der originalen Steuersoftware gewonnenen Messwerten, ergibt sich eine Pulsdauer von 1,5 ms [3, Kapitel 2], was durch die acht Zeilen der Lampenmatrix zu Pausen von 10,5 ms zwischen den einzelnen Impulsen führt.

Für jeden der drei Komponententypen muss im Rahmen dieser Arbeit untersucht werden, ob eine Ansteuerung mit *JavaScript* über die Plattform *Node.JS* die zeitlichen Anforderungen erfüllt.

# Kapitel 3

# Eignung von Node.JS

## 3.1 Definition der Grenze zu C++

Da ein Ansteuern von Rechnerschnittstellen wie der Parallel-Schnittstelle in *Node.JS* nicht Teil des Funktionsumfangs ist, muss diese Funktion als Erweiterung realisiert werden. Beim Entwurf einer solchen Erweiterung ergeben sich folgende Möglichkeiten, zwischen denen sich die am Ende umgesetzte Lösung bewegen muss:

- Die Ansteuerung der Parallelschnittstelle wird über eine *C++*-Erweiterung für *Node.JS* realisiert. Die im vorhergehenden Kapitel beschriebenen Ansteuerungsvarianten für die einzelnen Komponenten des Flipperautomaten werden in *JavaScript* implementiert.
- Die Ansteuerung der Komponenten wird innerhalb einer *C++*-Erweiterung umgesetzt. Eine Schnittstelle zu *JavaScript* bietet die Möglichkeit die Zustände der einzelnen Komponenten des Flipper zu manipulieren. Es wird also lediglich die Spiellogik und die Konfiguration der Software in *JavaScript* implementiert.

Da sich die Arbeit mit der Untersuchung von *Node.JS* beschäftigt, wird zunächst versucht, *C++* nur zur Ansteuerung des Parallel-Ports zu verwenden, um möglichst viel der Funktionalität innerhalb von *Node.JS* umzusetzen. Bei der Implementierung der hardwarenahen Softwarekomponenten soll aber darauf geachtet werden, dass diese bei Bedarf möglichst einfach in *C++* übersetzt werden können. Zu diesem Zweck muss zunächst aber grundsätzlich untersucht werden, inwiefern *Node.JS* für Steueraufgaben unter Zeitbedingungen geeignet ist.

## 3.2 Test der Geschwindigkeit

Die Einhaltung der im vorherigen Kapitel vorgestellten Zeitbedingungen stellt besondere Anforderungen an die Steuersoftware. Ob diese diese erfüllt werden können ist zunächst unklar, da die *Node.JS*-Plattform für Netzwerkapplikationen und

damit für einen ganz anderen Anwendungsfall konzipiert ist [1]. Im folgenden soll nun untersucht werden, inwieweit die Ereigniswarteschlange, die den Kern von *Node.JS* darstellt, diesen Anforderungen gewachsen ist.

Zu diesem Zweck soll die Geschwindigkeit beim Abarbeiten der Ereigniswarteschlange untersucht werden. *Node.JS* bietet mit der Methode **process.nextTick** die Möglichkeit, eine Rückruffunktion ans Ende der Ereigniswarteschlange einzureihen. Dadurch ist es möglich eine Funktion einmal pro Durchlauf der Warteschlange aufzurufen. So kann die Geschwindigkeit, mit der *Node.JS* die Schleife ohne Last ausführt, untersucht werden. Wird bereits hier die erforderlichen Zeitvorgaben nicht erreicht, muss von einer Implementierung der zeitkritischen Gerätekommunikation in *JavaScript* abgesehen werden.

```
(function benchEventLoop(last) {
    console.log(process.hrtime(last));
    var current = process.hrtime();
    process.nextTick(function () { benchEventLoop(current) });
})(process.hrtime());
```

**Quellcode 1:** Code zur Demonstration der Ereigniswarteschlangengeschwindigkeit.

Die Funktion holt die aktuelle Systemzeit als *Array* mit Sekunden- und Nanosekundenanteil und gibt die Differenz zum vorherigen Durchlauf auf der Konsole aus. Anschließend wird die aktuelle Zeit in einer Variable hinterlegt und **process.nextTick** genutzt, um die Funktion beim nächsten Durchlauf der Warteschleife erneut auszuführen. Hierbei ist zu beachten, dass die Ausgabe auf Konsole betriebssystemspezifisch ist und in der Regel blockierend erfolgt. Dies kann zu einem verlangsamten Schleifendurchlauf führen. Da hier lediglich das Erreichen einer Mindestgeschwindigkeit demonstriert werden soll, muss der dadurch bedingte verlangsamte Ablauf nicht weiter berücksichtigt werden.

Ausgeführt wurde dieser und die weiteren Tests in dieser Arbeit auf dem Entwicklungsrechner. Dieser ist mit einer *APU* vom Typ *AMD A6-3410MX* ausgestattet. Hierbei handelt es sich um vier CPU-Kerne mit einem moderaten Takt von 1,6 GHz. Als Betriebssystem kommt *Ubuntu 12.04 64 Bit* zum Einsatz.

```
[ 0, 5859 ]
[ 0, 5718 ]
[ 0, 12852 ]
[ 0, 6126 ]
[ 0, 5725 ]
[ 0, 13153 ]
```

**Ausgabe 1:** Messwerteausschnitt des Beispielquellcodes in Sekunden und Millisekunden.

Hierbei ergeben sich auf dem Entwicklungsrechner Laufzeiten durch die Warteschlange von um die 6 µs mit einzelnen Ausreißern in den niedrigen zweistelligen Mikrosekundenbereich. Die Ausgabe auf Konsole könnte zu einer Verzögerung geführt haben. Die grundsätzliche Ausführungsgeschwindigkeit in der *Node.JS*-Umgebung stellt sich als geeignet heraus, um eine Ansteuerung unter Zeitanforderungen im niedrigen Millisekundenbereich umzusetzen. Zusammen mit der hohe Ausführungsgeschwindigkeit von *Googles V8* und der Möglichkeit die Plattform mit nativen *C++*-Add-ons zu erweitern, lässt sich eine grundsätzliche Eigung von *Node.JS* feststellen.

Die Zeit der Zugriffe auf den Parallelport soll im folgenden untersucht werden. Mit dem *Garbage Collector* und der Eignung der in der *V8* integrierten Zeitgeberfunktionen sind zwei weitere Problemstellen vorhanden, die im nächsten Kapitel näher untersucht werden.

## 3.3 Parallelschnittstelle unter Linux

Unter Linux gibt es drei Möglichkeiten auf Parallel-Port zuzugreifen. Ihnen gemeinsam ist, dass es sich bei den Zugriffstreibern um *Character-Driver* handelt: Der Treiber schreibt und liest Byte-weise [30, Kapitel 5 & Kapitel 19]. Eine Übersicht über die Zugriffsmöglichkeiten findet sich unter [5], für die genauere Beschreibung wurde die im folgenden angegebene Fachliteratur verwendet:

#### Zugriff über /dev/lpX

Dieser Zugriff ist für die Ansteuerung von Druckern gedacht. Da hier die Datenleitungen nur geschrieben, nicht aber gelesen werden können, ist er für die Ansteuerung des Flipperautomaten nicht geeignet. Erschwerend zu dieser Einschränkung kommt hinzu, dass bei diesem Zugriffsmodus Antwortsignale automatisch ausgelöst werden. Auch wird auf Antwort des angesteuerten Gerätes gewartet [31, Kapitel 7].

#### Roher IO-Zugriff

Hier wird der Zugriff mittels des Befehls *ioperm* erhalten, anschließend kann mittels der Kernelmakros *outb* und *inb* schreibend bzw. lesend auf die Parallelport-Schnittstelle zugegriffen werden. Für diese Art des Zugriffs sind *root*-Rechte erforderlich. Die Adresse des Parallel-Ports ist eine im BIOS des jeweiligen Rechners

5 Eine Übersicht über die drei Zugriffsmöglichkeiten mit Erläuterungen für den praktischen Einsatz zur Hardware-Ansteuerung findet sich unter http://mockmoon-cybernetics.ch/computer/linux/programming/parport.html.

konfigurierte Basisadresse, unter der in den folgenden drei Bytes die Daten-, Kontroll- und Statusleitungen der Parallelschnittstelle abgebildet werden. Diese Lösung wurde von Herrn Zamanis ergriffen und in dessen Arbeit in Kapitel 7 näher vorgestellt [32, Kapitel 7] [30, Kapitel 19].

#### Zugriff über /dev/parportX

Der *ppdev* Treiber ermöglicht direkten und kompletten Zugriff auf Parallel-Port [31, Kapitel 7]. Beim Zugriff über „/dev/parportX" wird für den Gerätepfad mittels des **open** Befehls ein Deskriptor geöffnet. Über diesen kann im Anschluss mittels des Befehls **ioctl** die Leitungen der Parallelschnittstelle manipuliert werden und so mit dem angeschlossenen Gerät kommuniziert werden [31, Kapitel 7]. Durch Verwendung von **ioctl** kann das Programm im Benutzermodus laufen, da es sich bei **ioctl** um einen generischen Systemaufruf handelt [27, Kapitel 6].

#### Problematik mit ioctl

Die Verwendung von **ioctl** in zeitkritischen Anwendungsfällen war innerhalb der Linuxgemeinde umstritten. Dies lag im Umstand begründet, dass ein Systemaufruf mit **ioctl** zunächst vom unter Echtzeit laufenden Programm in den Kernel wechselte. Dort konnte es beim Gerätezugriff zum Einsatz eines *Superlocks* kommen, ein weitreichender *Locking*-Mechanismus des Linux-Kernels. Beim *Superlocks* handelt es sich um den ersten *Locking*-Mechanismus für Multiprozessorsystem, welcher den ganzen Kernel als kritischen Bereich ansieht. Durch die frühe Verfügbarkeit fand er historisch gewachsen breite Anwendung. Bereits im Kernel 2.6 war die Reichweite dieses *Locks* massiv eingeschränkt [27, Kapitel 5].

In der Entwicklung des Linux-Kernels wurde daher die Verwendung eines derart weitreichenden *Locking*-Mechanismus immer weiter reduziert, der *Git-Commit* [33] entfernt den *Superlock* endgültig aus dem Kernel. Die oft in älteren Quellen zu findenden Bedenken gegenüber eines Parallelport-Zugriffs mittels **ioctl**, aufgrund zu weitreichender *Locking*-Mechanismen, sind nicht mehr zutreffend.

## 3.4 Anbindung der Parallel-Schnittstelle

Zur Kommunikation mit der Parallelschnittstelle war geplant das *Node.JS*-Add-on *parport* von Xavier Mendez zum Einsatz [34] zu bringen. Nach ersten Tests stellte sich heraus, dass Anpassungen nötig sind. Auf diese wird in einem folgenden Abschnitt eingegangen. Beim Add-on *parport* handelt es sich um *Bindings* für die Bibliothek *Parallel-Port* von Matheus Neder [35], das heißt die Erweiterung *parport*

stellt einige Methoden der in *C++* implementierten Bibliothek im *JavaScript*-Namensraum zur Verfügung. Bei *Parallel-Port* handelt es sich um eine Bibliothek, die die Parallelschnittstelle Betriebssystem-unabhängig zu Verfügung stellt:

> "Parallel-Port provides a portable API for *C++* and Java to get raw access to the parallel port on IBM/PC based machines." [35]

Eine Analyse der offenen Quellcodes zeigt, dass unter Linux [36] der Befehl **ioctl** verwendet wird, um Befehle an die Parallelschnittstelle abzusetzen und über */dev/parportX* auf den Parallel-Port zugegriffen wird. Dies entspricht Variante drei der im vorherigen Abschnitt beschriebenen Zugriffsmöglichkeiten auf die Parallelschnittstelle.

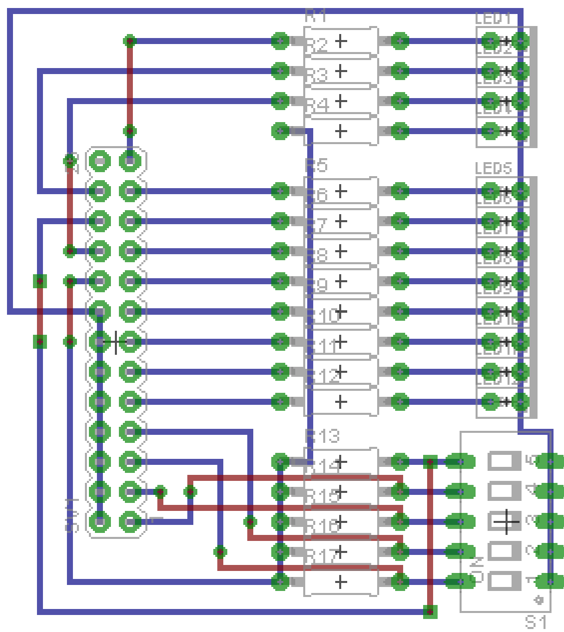
Die Verwendung einer fertigen Erweiterung hat diverse Vorteile: Statt vorhandene Funktionen selbst neu zu implementieren, wird auf ein bewährtes Add-on zurückgegriffen. Die im zweiten Kapitel beschriebe komplexe Implementierung von Erweiterungen für *Node.JS* kann umgangen werden. Sollte sich die gewählte Zugriffsmethode aus irgendeinem Grund als ungeeignet herausstellen, kann aufgrund der Quelloffenheit des Moduls *parport* leicht auf einen Rohzugriff auf die Parallelschnittstelle umgestellt werden. Diese Umarbeitung der Erweiterung ist im folgenden Abschnitt „Anpassung des Schnittstellenzugriffs“ näher beschrieben.

## 3.5 Test der Parallelschnittstelle

Um die Funktion des Parallelport-Zugriffs aus *Node.JS* zu prüfen, wurde sowohl die grundsätzliche Funktion als auch die Geschwindigkeit untersucht.

### Testplatine

Um die Funktion des Schnittstellenzugriffs testen zu können, wurde vom Autor eine Testplatine erstellt. Die Schaltung basiert auf einem Vorschlag, der im Fachmagazin *Elektor* veröffentlicht wurde [37]. Um den dort vorgestellten Schaltplan auf Lochrasterplatine umsetzen zu können, wurde vom Autor ein Platinenlayout entworfen und die Testschaltung entsprechend aufgebaut.



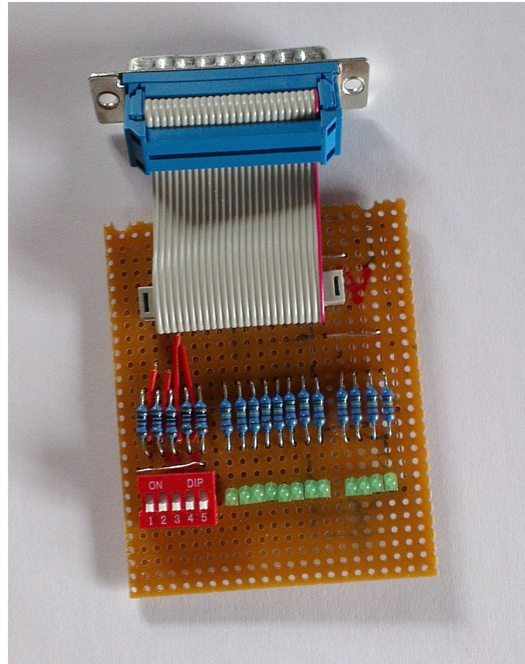




**Abbildung 5:** Lochraster-Platinenlayout für Parallelport-Testschaltung erstellt ausgehend vom Schaltplan nach [37]. Erstellt mit der Software EAGLE der Firma CadSoft [38].

**Abbildung 6:** Foto der fertiggestellten Testplatine.

Die Testplatine ermöglicht es, den Status der Kontrollleitungen (vier LEDs) und der Datenleitungen (acht LEDs) zu kontrollieren. Des weiteren ist es möglich, die Statusleitungen über einen DIP-Schalterblock zu setzen. Die Schalter werden über Kontrollleitung Nummer vier mit Strom versorgt, zum Auslesen des Statusblocks muss diese also aktiviert werden.

### Geschwindigkeit

Um die Eignung der Parallelportansteuerung abschätzen zu können, wurde die Anzahl der möglichen Schreiboperationen auf dem Entwicklungsrechner untersucht. Hierzu wurde eine Schnittstellenkarte für den *PCI Express Mini Card* Bus genutzt, angesprochen wurde die Karte über eine *Node.JS*-Konsole mit installierter Erweiterung *parport*. Als Zugriffsgeschwindigkeit konnten ca. 500.000 Schreibzugriffe auf die Datenleitungen pro Sekunde ermittelt werden, also ca. 500 Zugriffe je Millisekunde. Bei Ansteuerung des Flippers werden je Millisekunde folgende Registerzugriffe benötigt:

- sechzehn Zugriffe für ein komplettes Auslesen der Schaltermatrix mit zwei Registerzugriffen je Zeile für Zeilenauswahl und Lesen des Zeileninhalts
- vier Zugriffe für das Auslesen der vier Schalterregister

- drei Zugriffe für das Schreiben einer Lampenzeile zur Auswahl der Matrixzeile und Lesen der beiden Teile des Zeileninhalts

Bei den Lampenzugriffen handelt es sich um die Zugriffsanzahl im ungünstigsten Fall: Die Iteration über die Lampenzeilen erfolgt eigentlich nur alle 1,5 ms, daher wird nicht in jedem Millisekunden-Zyklus auf diese Register zugegriffen.

Zu diesen 23 regelmäßigen Registerzugriffen kommen sporadische Schreibzugriffe in die Spulenregister hinzu. Jeder Registerzugriffe bedingt sechs Zugriffe auf die Parallelschnittstelle, gemäß des unter [32, Kapitel 9] beschriebenen Kommunikationsprotokolls. Dadurch lässt sich abschätzen, dass eine Geschwindigkeit von etwa 150 Zugriffen je Millisekunde den Anforderungen genügen würde. Dies liegt deutlich unter den testweise erreichten 500 Zugriffen je Millisekunde, womit sich die Ansteuerung der Parallelschnittstele unter *Node.JS* von Seiten der Geschwindigkeit als geeignet erweist.

## 3.6 Anpassung des Schnittstellenzugriffs

Um die gewählten Anbindung der Parallelschnittstelle auf Funktionsfähigkeit zu prüfen, wurde versucht mit der Platine des Flipperautomaten zu kommunizieren. Leider verliefen erste Tests nicht erfolgreich, was weitere Untersuchungen nötig machte. Um den Problemen auf den Grund zu gehen, wurde zunächst manuell über die *Node.JS*-Konsole auf den Parallel-Port zugegriffen. Hier stellte sich die Steuerleitung mit Bit-Wert 32 als problematisch heraus. Deren Aufgabe ist es, die Zugriffsrichtung der Datenleitungen zwischen Lese- und Schreibmodus zu wechseln. Dies funktioniert im gewählten Zugriffsmodus zumindest nicht auf die einfache Art, wie sie das Kommunikationsprotokoll des Flipperautomaten vorsieht, unabhängig davon ob kein Gerät, die Testplatine oder der Flipperautomat angeschlossen ist. Um dieses Problem zu untersuchen wurde vom Autor ein Testprogramm in *C* geschrieben, das Schreib- und Leseoperationen auf den Datenleitungen durchführt. Diese Testsequenz wurde sowohl für Zugriff über *dev/parportX* als auch für den Rohzugriff umgesetzt. Hier zeigt sich, dass ein Zurückwechseln vom Lese- in den Schreibmodus nur bei Rohzugriff auf den Parallelport möglich ist. Auch ein Versuch die Zugriffsrichtung mittels eines entsprechenden **ioctl** Kommandos zu manipulieren war nicht erfolgreich.

Mit diesen Erkenntnissen wurde das *Node.JS*-Add-on *parport* umgearbeitet. Zunächst wurde eine komplette Kopie des Quellcodes, der unter *GPLv3* [34] veröffentlicht ist, angelegt. Für die verwendete *Parallel-Port* Bibliothek wurde eine eigene

Ersatzimplementierung geschrieben. Die neu erstellte Bibliothek *Low-Level-Port* ist schnittstellenkompatibel zur Originallösung, setzt aber sämtliche Befehl über den Rohzugriff auf den Parallelport um. Anschließend wurde Projektbeschreibungs-dateien und die Konfigurationsdateien für das *Build-Tool Generate Your Projects* [39] angepasst, so dass beim Kompilieren die neu geschriebene Bibliothek statt der originalen Bibliothek *Parallel-Port* als Abhängigkeit eingebunden wird. Das Plug-in kann nun, wie von einem *Node-JS*-Add-on erwartet, gebaut werden, indem man im Wurzelverzeichnis des Add-ons **npm install** aufruft. Das Plug-in ist für die Veröffentlichung im Add-on *Repository* von *Node.JS* vorbereitet, diese ist aber zumindest im Rahmen dieser Arbeit nicht erfolgt.

# Kapitel 4

# Probleme mit Node.JS

## 4.1 Übersicht

Während den Voruntersuchungen der verwendeten Technologien taten sich aufgrund deren Eigenschaften einige Problemfelder auf. In diesem Kapitel sollen zwei dieser potentiellen Problemstellen näher untersucht werden. Während es beim *Garbage Collector* bei einer theoretischen Betrachtung bleiben konnte, stellten sich die Zeitgeber als echtes Problem für die Implementierung dar. Hier musste eine eigene Lösung entworfen und gegen die in *Node.JS* integrierten Lösungen getestet werden. In diesem Kapitel sollen diese beiden Themen beleuchtet werden.

## 4.2 Garbage-Collector der V8

Der in der *V8*-Ausführungsumgebung verwendete *Garbage Collector* stoppt die Ausführung des Programms, um effizient arbeiten zu können [11]. Derartige Unterbrechungen in der Ausführung sind zur Einhaltung von Zeitbedingungen fatal. Eine Einschätzung wird dadurch erschwert, dass sich die Angaben zur Unterbrechungsdauer auf Server-Anwendungen mit enorm hoher Belastung beziehen. Erik Corry, einer der Entwickler der *V8* bei *Google* in Dänemark, nennt in einem Beitrag auf *Stackoverflow.com* vom 9. April 2011 1 ms pro Megabyte als Richtwert für die alte *Garbage Collection* der *V8*-Ausführungsumgebung [40]. Seitdem wurde die Speicherverwaltung mit Hinblick auf die Unterbrechungszeiten stetig weiterentwickelt.

Am 21. November 2011 geben Vyacheslav Egorov und Erik Corry im offiziellen Entwicklerblog von *Chrome* die Einführung eines inkrementellen *Garbage Collectors* bekannt. Waren in der alten Implementierung die von der Speicherverwaltung verursachten Pausen abhängig von der Speicherausnutzung der ausgeführten Anwendung, kommt es in der neuen Implementierung auch bei hohem Speicherbedarf zu deutlich kürzeren Unterbrechungen in der Ausführung [41].

Sollten sich die Verzögerungen durch die *Garbage Collection* trotz dieser Verbesserungen als zu schwerwiegend herausstellen, besteht die Möglichkeit, die Kommunikation mit der Platine des Flipperautomaten in *C++* zu implementieren. Dadurch kann ein guter Teil der Zeitbedingungen außerhalb von *JavaScript* erfüllt werden. Die entspricht dem unter „Definition der Grenze zu C++“ beschriebenen alternativen Ansatz.

## 4.3 Zeitgeber mit Node.JS

Regelmäßige Kommunikation unter Einhaltung von Zeitbedingungen erfordert präzise Zeitgeber. In den nächsten Abschnitten sollen zunächst die in *Node.JS* vorhandenen Zeitgeber untersucht werden, um im Anschluss ein Konzept für die Zeitgeber zur Ansteuerung des Flipper-Automaten zu erarbeiten.

#### Verwendung von setTimeout mit kleinen Zeiten

Mit **setTimeout** und **setInterval** bietet *Node.JS* zwei Zeitgebermethoden, die sich vom Syntax an gängige Browser-Implementierungen von *ECMAScript* anlehnen. Erzeugt man einen Zeitgeber mit einer Wartezeit von 0,5 ms, wird dieser Wert im zurückgegebenen Objekt auf ein Minima von eins aufgerundet. Ein solche Vorgabe von Mindestzeiten mag mit der Herkunft von *ECMAScript* zu begründen sein: In Webbrowsern wurden teils deutlich höhere Werte eingesetzt, wie das Beispiel des Browsers *Mozilla Firefox* zeigt [42]. Das Minimum von einer Millisekunde wird für die Anwendung problematisch, da für die Lampenansteuerung eigentlich ein Zeitgeber mit einer Wartezeit von einer halben Millisekunde notwendig wäre.

Eine Lösungsmöglichkeit zum Erreichen kürzerer Zeiten ist *aktives Warten*. Da dies aufgrund des in Kapitel 2 beschriebenen strikt nicht-blockierenden Aufbaus von *Node.JS* nicht möglich ist, kann *aktives Warten* unter Verwendung die Ereigniswarteschlange nachgebildet werden: Eine Hilfsfunktion wird ans Ende der Warteschlange gestellt. Sobald diese Ausführungszeit bekommt, prüft sie die aktuelle Zeit in Nanosekunden: Ist die gewünschte Wartezeit erreicht, wird der ursprüngliche *Call-back* aufgerufen, ansonsten trägt sich die Hilfsfunktion erneut in die Warteschlange ein. Dadurch kann eine zeitliche Präzision erreicht werden, die nur von der Durchlaufzeit der Ereigniswarteschlange und der Genauigkeit und Aktualisierungsrate des vom Betriebssystems gelieferten Zeitstempels abhängig ist. Ermöglicht wird dies durch die Einführung einer hochauflösenden Zeitgeberfunktion **hrtime** durch *Commit* [43] von Nathan Rajlich in die Versionsverwaltung des *Node.JS*-Projektes vom 05.03.2012.

Der folgende Beispiel-Code implementiert mit den gerade beschriebenen Funktionen einen Zeitgeber, der einmalig nach gegebener Wartezeit eine Rückruffunktion ausführt. Ein derart implementierter Zeitgeber löst frühestens beim Erreichen der gewünschten Wartezeit aus, spätestens bei erneutem Durchlauf der Warteschlange:

```
var waitNanoseconds = 500000;
var startTime = process.hrtime();
var targetTime = [startTime[0], startTime[0] + waitNanoseconds];
var callback = function() { console.log('Timer activated!') };
process.nextTick(function checkTimer() {
  var currentTime = process.hrtime();
  if (targetTime[0] < currentTime[0] || (targetTime[0] == currentTime[0]
      && targetTime[1] <= currentTime[1])) {
    callback();
  } else {
    process.nextTick(checkTimer());
  }
});
```

**Quellcode 2:** Beispiel-Implementierung eines Einmal-Zeitgebers.

Bei dieser Lösung ist zu beachten, dass der Betriebssystemzeitgeber unter Umständen deutlich seltener aktualisiert wird, als die hohe Auflösung in Nanosekunden suggeriert. Eine kürzere Mindestwartezeit als mit der **setTimeout**-Funktion von *Node.JS* sollte sich aber auf alle Fälle erreichen lassen.

### Periodische Zeitgeber

Im folgenden Abschnitt soll eine geeignete Möglichkeit untersucht werden, um periodische Zeitgeber in JavaScript umzusetzen. Mit diesen können die regelmäßig wiederkehrenden Schreib- und Leseoperationen auf die Register der Steuerplatine des Flipperautomaten angestoßen werden.

Die von *Node.JS* für einen periodischen Zeitgeber zur Verfügung gestellte Funktion **setInterval** ist für genaue Intervalle ungeeignet: Nachdem ein Aufruf der hinterlegten Funktion nach Ablauf der Wartezeit fällig wird, erfolgt diese noch leicht zeitverzögert, da die Ausführungsumgebung unter Umständen gerade noch ein anderes Ereignis abarbeitet. Erst dann wird der Zeitgeber für den nächsten Aufruf aktiviert. Aufgrund dieses Verhaltens gilt es schon außerhalb von *Node.JS* im Umfeld der Webentwicklung als *Best Practice* statt dessen iterativ den Einmalzeitgeber **setTimeout** zu nutzen und den Fehler durch Zeitverzögerung bei Berechnung der nächsten Wartezeit einfließen zu lassen.

Eine kurze Prüfung unter *Node.JS* mit einem kurzen Intervall von 1 ms zeigt einen weiteren gravierenden Nachteil: **setInterval** ist für derart kurze Wartezeiten um ein vielfaches zu langsam, ein periodischer Zeitgeber mit **setTimeout** beendet zu schnell.

Beide Probleme verschwinden, wenn deutlich höhere Intervallzeiten gewählt werden. Um dieses Problem und eine mögliche Lösung zu untersuchen, wurden im Rahmen der Arbeit vier Zeitgeber implementiert, die periodisch eine Rückruffunktion aufrufen. Diese Rückruffunktion wird im folgenden in Anlehnung an die Sprache *Java Runnable* genannt:

- **NativeInterval** verwendet die Methode **setInterval** zur regelmäßigen Ausführung der *Runnable*-Methode.
- **NativeTimeout** hinterlegt per **setTimeout** eine Rückruffunktion, die die *Runnable*-Methode ausführt und sich mittels **setTimeout** selbst aufruft.
- **SpinningRunner** führt das *Runnable* einmal je Durchlauf der Ereigniswarteschlange aus. Durch diesen ungebremsten Durchlauf kann geprüft werden, ob die Rechenlast des *Runnable* im gewünschten Zeitintervall zu bewältigen ist: Beendet dieser Zeitgeber später als gewünscht, ist die Last zu hoch.
- **AdaptiveRunner** nutzt das im vorhergehenden Abschnitt vorgestellte aktive Warten mittels Ereigniswarteschlange, er nutzt dazu die Methoden **process.hrtime** und **process.nextTick**.

Für diese vier Implementierungen wurde vom Autor ein Test in Form eines *Benchmarks* erstellt, um die Leistungsfähigkeit der Implementierungen vergleichen zu können. In diesem werden alle vier Implementierungen mit einem Intervall von 1 ms 1000 mal ausgeführt und anschließend die Differenz zwischen Start- und Endzeitpunkt ermittelt:

```
Interval length (ms): 1
Interval count: 1000
Difference for NativeInterval (ms): 6779.140886
Difference for NativeTimeout (ms): -813.646538
Difference for SpinningRunner (ms): -994.726188
Difference for AdaptiveRunner (ms): -0.04269
```

**Ausgabe 2:** Konsolenausgabe von des *Benchmarks* IntervalRunnerTest.js.

Bei einer theoretischen Ausführungszeit von einer Sekunde weichen die beiden nativen Zeitgeberfunktionen massiv ab und müssen als für diese Aufgabenstellung ungeeignet betrachtet werden. Der **SpinningRunner** ist erwartungsgemäß deutlich zu schnell fertig. Die Abweichung des **AdaptiveRunner** ist sehr gering. Durch den fehlenden schleichenden Fehler ist der **AdaptiveRunner** unabhängig von der Anzahl der Iterationen: Der Fehler stellt lediglich den Fehler der letzten Iteration dar.

Hierbei ist zu berücksichtigen, dass die je Iteration durchzuführende Rechenlast der Zeitgeber im Test vernachlässigbar ist. Aufgrund der stärker ausgelasteten Ereigniswarteschlange, ergeben sich im praktischen Einsatz größere Fehler. Dies bedeutet aber im Umkehrschluss auch, dass eine Optimierung des je Intervalls ausgeführten Codes sich positiv auf die Genauigkeit auswirkt.

Der Versuch, beim **AdaptiveRunner** den Restfehler mittels einer Formel zu berücksichtigen und die Zielzeiten etwas früher anzusetzen, war nicht erfolgreich. Die Idee war es eventuelle Verzögerungen auszugleichen, wie im vorherigen Abschnitt für den Einmalzeitgeber erläutert. Hierfür ändert sich der Fehler aber zu sprunghaft: Bei Betrachtung der Einzelfehler jeder Iteration zeigt sich, dass sich die Zahlen zu schnell ändern. Für die Implementierung der Steuersoftware wichtig ist die zuverlässige Funktion im Bereich einzelner Mikrosekunden, des weiteren gilt es einen schleichenden Fehler zu verhindern. Beides wurde mit der Lösung ohne Fehlerausgleich bereits erreicht, weshalb eine weitere Optimierung nicht ratsam erschien.

# Kapitel 5

# Architekturentwurf

## 5.1 Abgrenzung zur Implementierung in RTS Java

Bei der ursprünglichen Implementierung wurde *RTS Java* als Programmiersprache und *XML* als Dateiformat für die Konfiguration gewählt. Die hier erarbeitete Lösung hingegen setzt beides mit *JavaScript* um. Daraus ergibt sich eine Reihe an Unterschieden: Die eine Sprache wird kompiliert, bei der anderen handelt es sich um ist eine Skriptsprache. *RTS Java* ist im Gegensatz zu *JavaScript* echtzeitfähig. Die Java-Lösung setzt auf ausgeprägtes *Multithreading*, währen von der Steuerlogik in JavaScript direkt nur ein *Thread* angesprochen werden kann. Aus diesen Unterschieden ergeben sich sowohl eine Reihe an Problemen, aber auch Chancen bei der *JavaScript*-Umsetzung. Vieles lässt sich in *Node.JS* einfacher umsetzen, was im folgenden erläutert werden soll.

#### Aufbau

Abstrahiert man die Hardware und betrachtet nur die Schnittstelle zu dieser, lassen sich Schaltern, Spulen und Lampen als Ein- und Ausgabe-Elemente identifizieren. Gemäß dieser Einschätzung von Herrn Kratzer soll auch in dieser Arbeit der Flipperautomat lediglich als Schnittstelle mit Ein- und Ausgabekomponenten betrachtet werden [3, Kapitel 4].

Bei der grundsätzlichen Struktur wurde ein jedoch ein anderer Weg gewählt, um den sprachspezifischen Besonderheiten von *JavaScript* gerecht zu werden und diese zum Vorteil zu nutzen. Ziel der Änderungen ist es, die Architektur schlanker zu halten. Deshalb wird vom doppeltem *MVC Pattern* [3, Kapitel 4] Abstand genommen. Statt dessen wurde eine Grundstruktur mit zwei Abstraktionsebenen entworfen, aus denen auf die Komponenten des Flippers zugegriffen wird: In der Spiellogik wird das Verhalten des Flipperautomaten hinterlegt. Die Struktur des Automaten hingegen ist in der Ansteuerungsschicht konfiguriert. Diese Trennung wird sowohl in der Konfiguration als auch in der Implementierung der Ansteuerungssoftware als Ausgangspunkt verwendet.

### Konfiguration

Die Spielregeln sind in Herrn Kratzers Lösung teilweise im Code hinterlegt. Es existieren über 50 *Observer*, welche bestimmte Spielsituationen abbilden [3, Kapitel 6]. Die konkreten Konfigurationswerte sind in der XML-Konfiguration hinterlegt, bei Initialisierung der Software werden die *Observer* passend instanziiert. Hier soll bei der Umsetzung in *Node.JS* einer der Vorteile einer Skriptsprache eingesetzt werden: Die Regelkonfiguration selbst wird in *JavaScript* umgesetzt und kann neben den Konfigurationsparametern Verhalten in Form von *JavaScript*-Funktionen enthalten. Dadurch ist die Definition einer Spielregel komplett an einer Stelle gesammelt, und kann im Gegensatz zu fest hinterlegten Regelklassen deutlich flexibler konfiguriert werden. Eine ausführlichere Betrachtung zu diesem Thema erfolgt im Abschnitt „Struktur der Konfiguration".

### Kommunikation

Aufgrund der Einschränkungen durch den Einsatz einer Echtzeitumgebung wurde in Herrn Kratzers Arbeit die Kommunikation streng ereignisorientiert durchgeführt. Eine Vielzahl an Ereignissen signalisieren nicht nur eine Statusänderung von Bauteilen, sondern auch die Ansteuerung selbst erfolgt wiederum über Ereignisse. Dieser Ansatz ergibt sich unter anderem aus der Auftrennung in Echtzeit- und Nicht-echtzeitprogrammkomponenten [3, Kapitel 5]. Neben dem Fehlen dieser Anforderung gibt es in *Node.JS* wegen des einzelnen *JavaScript-Threads* auch keinen *Thread*-übergreifenden Datenaustausch, für die ereignisbasierte Kommunikation ein Lösungsansatz ist.

Dies führt zu folgender Vereinfachung: Da die Veränderung nur den jeweils zuständigen Komponententreiber als einzigen Empfänger hat, soll hier von einem ereignisgestützten Nachrichtenübermittlung Abstand genommen werden. Statt dessen werden die Steuerbefehle über direkten Methodenaufruf kommuniziert. Die Änderungen der Schalterzustände sollen weiterhin über Ereignisse kommuniziert werden, da sich diese innerhalb der Spiellogik an eine unbekannte Anzahl an Empfängern richtet. Dies lässt eine lose Kopplung, wie sie durch ereignisgestützte Kommunikation erreicht wird, von Vorteil erscheinen.

Eine weitere Vereinfachung ergibt sich aus dem Umstand, dass *Node.JS* intern auf eine Ereigniswarteschlange setzt, welche über eine API vom Entwickler für eigene Objekte mitgenutzt werden kann [44]. Auf das Einbinden oder Implementieren eines eigenen

Ereignissystems, wie es in der Arbeit von Herrn Kratzer notwendig war, kann hier verzichtet werden. Dadurch entfällt auch der Initialisierungsaufwand, der bei einem eigenen System notwendig ist.

## 5.2 Struktur der Konfiguration

Bei einer Software, bei der ein Großteil der Funktionalität über die Konfiguration bestimmt wird, ist die Wahl derselben von großer Bedeutung. Deshalb sollen im folgenden noch einmal ausführlicher die Gründe erläutert werden, aufgrund derer die Konfiguration in der in dieser Arbeit beschriebenen Form entworfen wurde.

Für die Wahl der Konfiguration waren folgende Ziele relevant:

- Lesbarkeit
- Erweiterbarkeit
- Testbarkeit

Ein naheliegender Ansatz für die Konfiguration der Spielregeln wäre gewesen, lediglich die Konfigurationswerte zu hinterlegen. Dies hätte in Form von *XML* oder *JSON*-Dateien erfolgen können. Dieser Ansatz wurde in Kevin Kratzers Bachelorarbeit gewählt. Herr Kratzer schreibt von über 50 *Observern* die er implementiert hat [3, Kapitel 6]. Da eine rein wertegestützte Konfiguration nicht genug Möglichkeiten bot, musste entsprechend Funktionalität im Quellcode in Form der *Observer* hinterlegt werden. Dies hat allerdings den Nachteil, dass die Spielregeln über Quellcode und Konfiguration verteilt sind, bzw. dass man in den Spielregeln nur Verhalten erreichen kann, dass bereits im Quellcode vorbereitet ist. Hinzu kommt, dass die Verständlichkeit der Regeln durch die Verteilung leidet.

Aus dieser Überlegung heraus, und dem Umstand dass *JavaScript* in seiner Eigenschaft als Skriptsprache eben genau hierfür die Möglichkeiten bot, wurde der Entschluss gefasst, Verhalten und Konfigurationswerte innerhalb der Konfiguration zu bündeln. Durch die Bündelung soll sowohl die Lesbarkeit als auch die Flexibilität erhöht werden. Der Benutzer soll seine Spielregeln nicht in Form von Werten, sondern in Funktionen hinterlegen können. Hierdurch soll maximale Flexibilität bei den Spielregeln erreicht werden. Durch eine klare Grundstruktur soll die Anforderung bei der Regelerstellung aber überschaubar bleiben, um nicht tiefgreifende Kenntnisse in *JavaScript* oder gar dem internen Aufbau der in dieser Arbeit implementierten Software zu erfordern.

Die Funktion **eval** würde das Hinterlegen von Funktionen als Strings ermöglichen, was auch im Rahmen einer *JSON*-Konfiguration umgesetzt werden könnte. Gegen Einsatz dieser Funktion sprechen allerdings viele gute Gründe. Relevant für die hier beschriebene Anwendung wäre die langsame Ausführung, da viele Optimierungsschritte nicht ausgeführt werden können [10, Kapitel 2] und die schlechte Lesbarkeit des Quellcodes, auch aufgrund von fehlendem Syntax-Hervorhebung [18, Anhang B].

Aus diesem Grund kam nur eine Umsetzung direkt in Form von *JavaScript*-Dateien in Betracht. Nachteil hierbei ist, dass die Konfigurationsdateien einen gewissen *Overhead* haben, der sie zu syntaktisch korrekten *Node.JS*-Modulen macht. Dieser ist für den Nutzer nicht von Belang und dient als potentielle Fehlerquelle, sollte er fälschlicherweise verändert werden. Dieser Nachteil wird aber nach Meinung des Autors von den Vorteilen aufgewogen. Eine Beschreibung des Aufbaus dieser Dateien erfolgt im Abschnitt „Konfiguration Spiellogik“.

Auch im Hinblick auf die Testbarkeit ergeben sich bei dieser Art der Konfiguration Vorteile: Die einzelnen Teile der Spielregeln sind Funktionen, die sämtliche Kooperationsobjekte als Parameter übergeben bekommen. Hier ist es im Rahmen von *Unit Tests* möglich, die Funktionen mit *Mock*-Objekten als Parameter aufzurufen. Das Zusammenführen der einzelnen Funktionen zu ausführbaren Regeln übernehmen dann die einzelnen Module der Spiellogik. Diese Initialisierung ist von der Konfiguration unabhängig und soll im Rahmen dieser Arbeit separat getestet werden. Es ist also ausreichend beim Testen der Regeln sich auf die einzelnen Funktionen zu konzentrieren.

## 5.3 Entwurf der Parallelport-Kommunikation

Um mit der Ansteuerungsplatine des Flipperautomaten zu kommunizieren, wird im folgenden ein Modul mit Namen **ParallelPort** erstellt. In diesem wird das in der Arbeit von Herrn Zamanis [32] beschriebene Kommunikationsprotokoll umgesetzt. Nach außen bietet dieses Modul die Möglichkeit die Register der Flipperplatine zu lesen und zu schreiben und kapselt so das eigentliche Kommunikationsprotokoll. Die Ansteuerung der Parallelschnittstelle erfolgt mit dem im Rahmen dieser Arbeit angepassten *Node.JS*-Modul *parport2*. Dessen Erstellung wurde im Abschnitt „Anpassung des Schnittstellenzugriffs“ bereits beschrieben.

Mit dem Modul **ParallelPort** wird es allen weiteren hardwarenahen Modulen ermöglicht, auf eine einfache und abstrahierte Art und Weise mit dem Flipperautomaten zu kommunizieren.

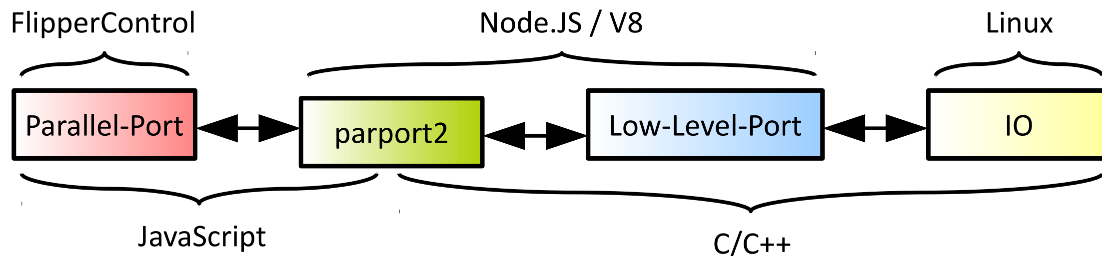



**Abbildung 7:** Abstraktionsebenen beim Zugriff aus der Steuersoftware auf die Parallelschnittstelle.

## 5.4 Grundstruktur des Architekturentwurfs

Auf alle Komponenten des Flipperautomaten wird in zwei Abstraktionsebenen zugegriffen:

Einerseits gibt es eine Spiellogik, die die Spielregeln ausführt und dazu den Zustand der Komponenten im Falle von Schaltern abfragt oder im Falle von Spulen und Lampen setzt. Hierbei handelt es sich um eine stark abstrahierte Verwendung der Komponenten, welche über einen eindeutigen Namen angesprochen werden.

Auf der anderen Seite gibt es die Hardware-Ansteuerung, die den Zustand des Flipperautomaten mit dem des internen Modells abgleicht und zu diesem Zweck über die Parallelschnittstelle mit dem Automaten kommuniziert. Die einzelnen Komponenten sind auf dieser Ebene über Registeradressen und *Offsets* oder Matrixzeilen und -spalten repräsentiert. Um der Spiellogik eine Schnittstelle zu bieten, muss diese Darstellung auf die Namen abgebildet werden. Der Zugriff auf den Flipperautomaten wird über die Parallelschnittstelle realisiert, wobei hier durch eine weitere Abstraktionsschicht direkt Register gesetzt und gelesen werden können, ohne sich mit der genauen Ansteuerung des Parallel-Ports zu befassen.

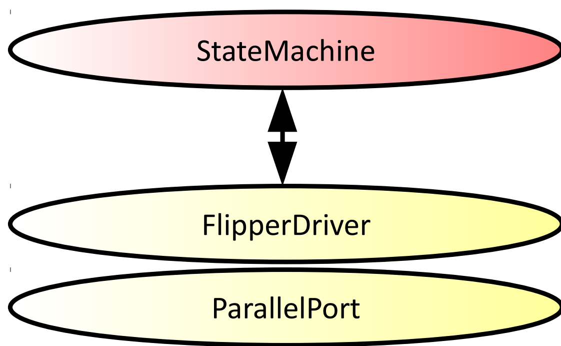



**Abbildung 8:** Grundstruktur der Zwei-Schicht-Architektur mit *StateMachine* und *HAL*.

## 5.5 Schicht HAL

In der *HAL* werden alle Module gebündelt, die direkt zur Ansteuerung des Flipperautomaten dienen. Sie stellt die Schnittstelle zwischen der Steuerplatine des Flipperautomaten und der Spiellogik her. In Richtung der Spiellogik stellt sie den Gesamtzustand zur Verfügung und kommuniziert Änderungen. In Richtung Flipperautomat beachtet sie dessen Ansteuerungsbedingungen und -anforderungen. Die Struktur der Implementierung soll derart erstellt werden, dass der Quellcode Methode für Methode in *C++* übertragen werden kann. Dies ermöglicht es, die Trennlinie zwischen *C++* und *JavaScript* bei Problemen auch nachträglich noch zu verlegen und die *HAL* teilweise in *C++* umzuschreiben.

### Modul FlipperModel

Das **FlipperModel** sammelt den Gesamtzustand des Flippers. Es kennt die Zustände der Schalter, Lampen und Spulen und bietet den Spielregeln eine Schnittstelle diese abzufragen. Der Zustand des Modells wird von den folgenden Modulen aktualisiert, um stets ein exaktes Abbild des realen Flipperautomaten zu bieten.

Eine Aktualisierung eines Schalterzustandes wird in Form eines Ereignisses kommuniziert, wobei jeder Schalter durch ein eigenes Ereignis repräsentiert wird. Interessenten können beim Modell eine Rückruffunktion registrieren, welche das Ereignis verarbeiten kann.

### Modul SwitchScanner

Der **SwitchScanner** fragt je Intervall *alle* Schalter ab. Die Zeitvorgaben, welche die Länge des Abgrageintervall bedingen, hängen einerseits mit dem *Watchdog* [3, Kapitel 2] zusammen. Sie ergeben sich aber auch aus der Ballgeschwindigkeit, wie in [3, Kapitel 2] korrekt erläutert wird: Die Kugelgeschwindigkeit von bis zu 3 m/s

bedingt ein hohes Abfrageintervall. So kann eine Reaktion sichergestellt werden, bevor die Stahlkugel den Wirkbereich der aktiven Elemente verlässt. Aus diesem Grunde werden alle Schalter jede Millisekunde abgefragt.

#### Modul LampDriver

Dieses Modul bietet eine Schnittstelle um Lampen ein- und auszuschalten bzw. ihren Zustand umzukehren. Da alle Lampen im Pulsbetrieb angesteuert werden [3, Kapitel 2], kann diese Information nicht einfach an die Hardware geschrieben werden. Statt dessen muss der Zustand der Lampen intern gespeichert werden.

Bei der Ansteuerung im Pulsbetrieb werden regelmäßig alle gewünschten Lampen in einer Zeile der Matrix aktiviert und nach einem Zeitintervall von 1,5 ms wieder deaktiviert. Dieses Intervall ergibt sich nach [3, Kapitel 2] aus dem Messungen mit der originalen Steuersoftware.

#### Modul SolenoidDriver

Der **SolenoidDriver** bietet eine Schnittstelle um Spulen zu aktivieren. Anders als bei den Lampen wird der Zustand nicht dauerhaft geändert, statt dessen bleiben die Spulen ein individuell konfiguriertes Zeitintervall aktiv. Anschließend werden sie vom **SolenoidDriver** wieder deaktiviert. Haltespulen bleiben hingegen dauerhaft aktiv und müssen manuell deaktiviert werden. Da es beim zu langem Betrieb einer Spule zu Hardwareschäden kommen kann, muss eine Reaktivierung der Spule während sie aktiv ist unterbunden werden. Anderenfalls wird die Gesamtdauer des Betriebs über das zulässige Maß hinaus verlängert.

Die Aktivierung von Spulen wird sofort umgesetzt, um Reaktionszeiten klein und die maximale Betriebsdauer zuverlässig einzuhalten.

#### Module DriverFacade

Im Modul **DriverFacade** werden **SolenoidDriver** und **LampDriver** unter einer einheitlichen Schnittstelle zur Ansteuerung von Flipperkomponenten angeboten. Das *Facade Pattern*, das hierbei Anwendung findet, wird im Abschnitt „Verwendete Entwurfsmuster“ in diesem Kapitel näher erläutert.

#### Konfiguration

Die Konfiguration der Bauteile besteht zunächst aus der Definition von individuellen Namen. Dies ermöglicht einen hohen Abstraktionsgrad, da Komponenten des Flipperautomaten nur noch über ihren Namen angesprochen werden, während technische Details verborgen bleiben. Für jeden dieser Namen wird die

Bauteiladressierung hinterlegt, d.h. die Position innerhalb des Registers oder der Matrix an der sich die Komponente befindet. Bei Spulen wird zusätzlich die maximale Betriebsdauer hinterlegt. Des weiteren soll in der Konfiguration die Zugriffsinformationen für den Parallelport hinterlegt werden.

**Initialisierung**

Da die Konfiguration der Module einen großen Anteil am Quellcode hat, wird das *Builder-Pattern* [45, Kapitel 3] verwendet: Jedes Modul bietet einen individuellen Erbauer an. Nachdem das eigentliche Objekt des Moduls von ihm erzeugt und konfiguriert wurde, kann es zurückgegeben werden.

Die Komplexität der Konfiguration ist einerseits mit der Anzahl der Ein- und Ausgabekomponenten zu erklären. Des weiteren führt sowohl die asynchrone als auch die ereignisgestützte Kommunikation dazu, dass viele Rückruffunktionen verwendet werden. Diese lassen sich beim Erzeugen der Objekte durch die Erbauer generieren und können hinterlegen werden. Im Betrieb werden diese nur noch ausgeführt bzw. als Rückruffunktion an andere Funktionen übergeben.

Anhand der beschriebenen Module wurde folgende Grundstruktur als Vorlage für die Implementierung entworfen:

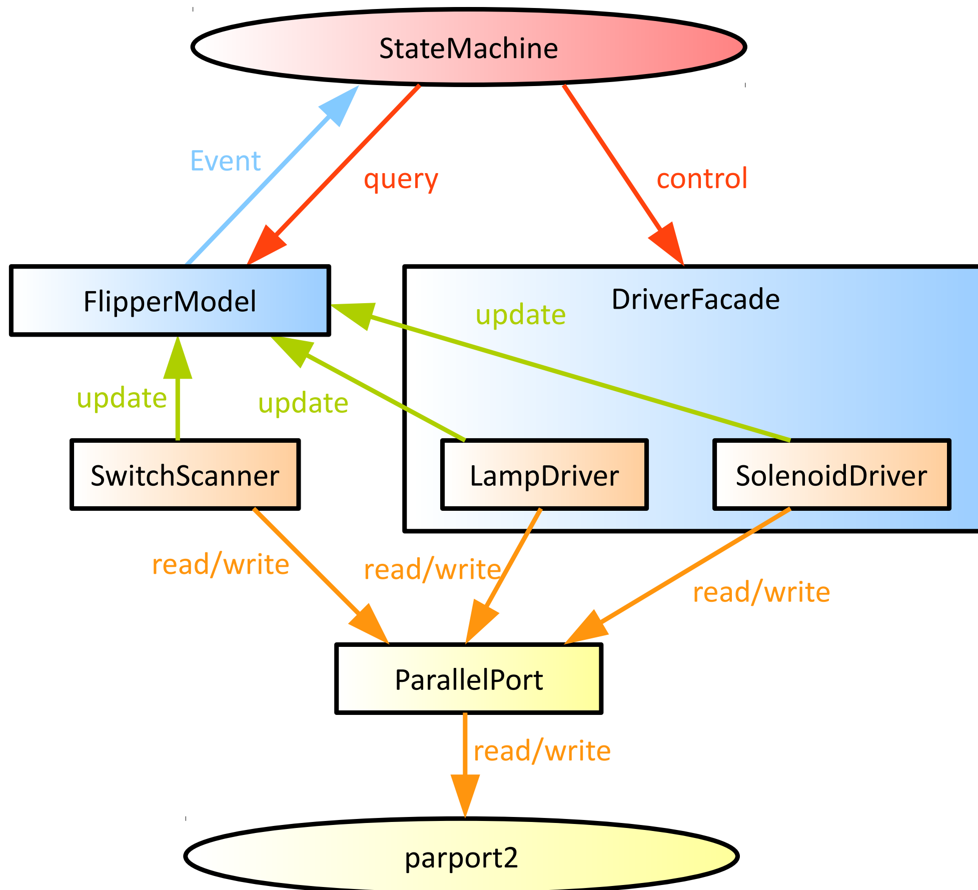



**Abbildung 1:** Schematische Darstellung der *HAL*-Schicht.

## 5.6 Schicht Spiellogik

In der Spiellogik werden alle Module gesammelt, die zur Umsetzung der Spielregeln und der Durchführung eines Flipperspiels notwendig sind. Die Spiellogik soll als Zustandsautomat umgesetzt werden. Hierbei wird ein je nach Spielzustand unterschiedlicher Satz an Bedingungen geprüft und bei Bedarf entweder ein interner Zustandswechsel der Spiellogik oder eine Ansteuerung des Flipperautomaten ausgelöst. Um die nötigen Änderungen am Flipperautomaten durchzuführen greift sie auf die durch die *HAL* abstrahierte Steuerplatine zu.

Hierbei stellt die Implementierung nur ein Grundgerüst für die Initialisierung und den Betrieb der Spiellogik zur Verfügung. Das genaue Verhalten wird von einer ebenfalls in *JavaScript* implementierten Spielregelsammlung festgelegt, welche von der Implementierung an den passenden Stellen aufgerufen wird.

### Modul Action

Das Modul **Action** stellt die Verknüpfung eines Bedingung mit einem Kommando dar. Es sorgt dafür, dass diese passend verknüpft werden. Bei Aktivieren registriert die **Action** zudem die von der Bedingung benötigten Ereignisse.

Bedingungen bestehen aus einer Liste aus Schalter-Ereignissen, bei denen eine Prüfung erfolgen soll. Des weiteren enthalten sie eine Prüfbedingung in Form einer Funktion. Ereignisliste und Prüffunktion sind hierbei Teil der Regelkonfiguration. Diese werden bei Initialisierung der Konfiguration passend mit einem Kommando oder Zustandswechsel verknüpft.

Bei einem Kommando handelt es sich um eine Sammlung aus einem oder mehreren Steuerbefehlen für den Flipperautomaten zur Ansteuerung von Lampen und Spulen. Verwendet werden die Kommandos primär als Reaktion auf ausgelöste Bedingungen. Aber auch Zustandswechsel der Spiellogik können die Ausführung eines Kommandos auslösen. Kommandos werden ausschließlich in der Konfiguration der Regeln als Funktion definiert, da sie keinerlei Initialisierung benötigen. Die von ihnen benötigten Parameter erhalten sie vom Aufrufer.

### Modul Timer

Um komplizierte Abläufe komfortabel zur Verfügung stellen zu können, sollen in der Implementierung drei Hilfsstrukturen Verwendung finden: Mit Hilfe eines vordefinierten Zeitgebers soll es möglich sein, ein Kommando verzögert oder in einem Intervall wiederholt auszuführen. Auch soll das Definieren einer Sequenz von Kommandos mit festgelegter Verzögerungszeit möglich sein. Letzteres ist insbesondere für die Zusammenstellung komplexer Lampenansteuerungsfolgen gedacht. Sequenzen können permanent aktiv sein oder sich nach einmaligem Durchlauf selbst beenden.

### Modul State

Ein **State** repräsentiert einen Zustand der Spiellogik. Er besteht aus *Actions* und *Transitions*. Actions dienen zur Abbildung der im aktuellen Zustand der Spiellogik gültigen Spielregeln. Eine *Transition* besteht ebenfalls aus einer Bedingung, löst bei Erfüllung dieser aber einen Zustandswechsel des Zustandsautomaten aus. Mit dem neuen Zustand wird ein neuer Satz aus Aktionen und Transitionen aktiviert. Zusätzlich soll ein **State** bei Betreten und Verlassen eine in der Konfiguration

hinterlegte Rückruffunktion auslösen. Dies ermöglicht es, für den Zustand nötige Initialisierungen am Flipperautomaten vorzunehmen bzw. diese bei Verlassen wieder aufzuheben.

#### Modul StateMachine

Bei der **StateMachine** handelt es sich um das Herzstück der Spielregeln. Diese werden in Form von *States* organisiert, wodurch die aktiven Regeln in Abhängigkeit der aktuellen Spielsituation aktiviert werden können.

Für Regeln, die unabhängig vom aktuellen Zustand aktiv sein sollen, enthält die **StateMachine** neben den Zustandsdefinitionen auch *Watcher*. Diese bestehen wie bereits die Aktionen aus einer Bedingung und einem Kommando. Während die Zustände und die mit ihnen verknüpften Aktionen dazu dienen, Missionen [3, Kapitel 7] umzusetzen, dienen die Überwacher dazu, situationsunabhängige Regeln zu überwachen, wie beispielsweise das Freigeben einer zusätzlichen Kugel bei Aktivieren bestimmter Schalter.

#### Konfiguration

Die Konfiguration besteht aus drei Elementen. Um dauerhaft aktive Regeln zu definieren, muss eine Sammlung aus Konfigurationen für **Action**-Instanzen erstellt werden. Diese werden in Form von *Watchern* im Zustandsautomaten hinterlegt. Zweites Element der Konfiguration ist eine Liste der Zustände. Dies ist nötig, weil die Anzahl und Namensgebung derselben dem Benutzer überlassen bleibt. Hier kann auch Start- und Endzustand markiert werden. Drittes Element sind Konfigurationen für **States**. Diese werden für jeden in der Zustandsliste aufgelisteten Zustand erstellt. Sie bestehen aus Eintritts- und Austrittsfunktionen und einer Liste aus **Actions**. Des weiteren wird eine Liste mit *Transitions* hinterlegt, um zu Beschreiben unter welchen Bedingungen ein Wechsel in einen anderen Zustand erfolgen soll.

#### Initialisierung

Wie bei der Initialisierung der *HAL* wird auch hier durch den hohen Anteil, den die Konfiguration am Gesamtaufwand hat, auf das *Builder-Pattern* [45, Kapitel 3] zurückgegriffen. Die einzelnen Erbauer der Module verarbeiten und prüfen einen Teil der Konfiguration und erzeugen daraus Objektinstanzen. Diese bieten in der Regel nur noch Methoden zum Aktivieren und Deaktivieren an, um im Rahmen eines Zustandsautomaten eingesetzt werden zu können.

Mit den vorgestellten Modulen ergibt sich für die **StateMachine** folgende Grundstruktur:

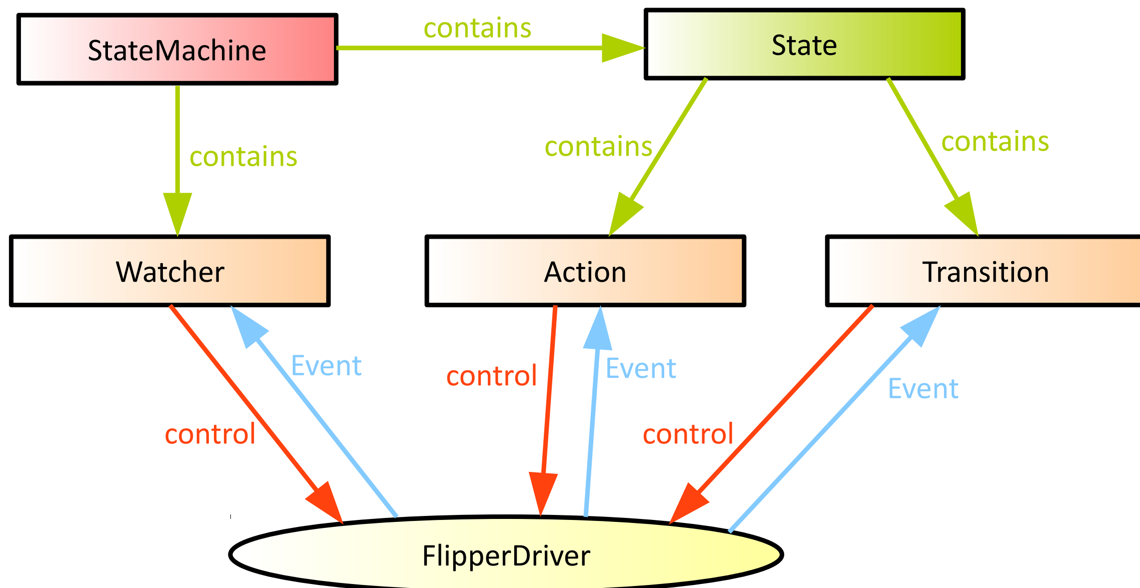



**Abbildung 9:** Schematische Darstellung der Schicht StateMachine.

## 5.7 Verwendete Entwurfsmuster

Bei der Gestaltung der Architektur kamen einige Entwurfsmuster zum Einsatz. Diese sollen auftretende Probleme nicht nur auf bewährte Art und Weise lösen, sondern auch die Verständlichkeit der Lösung durch Verwendung bekannter Strukturen erhöhen.

### State Pattern

Beim *State Pattern* handelt es sich um ein Muster, mit dem ein Zustandsautomat objektorientiert abgebildet werden kann. Hierzu wird das Verhalten jedes Zustands in Zustandsklassen gebündelt, die einen gemeinsamen abstrakten Zustand implementieren. Der Kontext tauscht nun je nach Bedarf das von ihm referenzierte Zustandsobjekt aus [45, Kapitel 5].

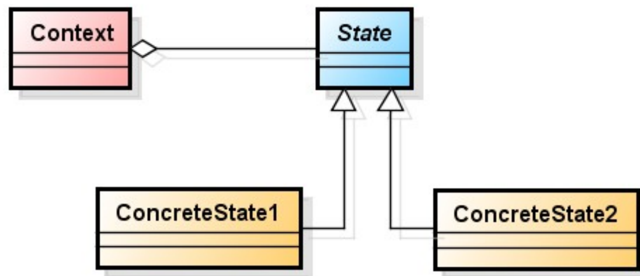



**Abbildung 10:** UML-Diagramm für Grundstruktur des *State Patterns* nach [46, Seite 410].

Um den Gesamtzustand der Spiellogik abzubilden wird auf das *State Pattern* zurückgegriffen. Zustände werden in der Konfiguration der Spiellogik erstellt. In der Original-Software [29] des Flipperautomaten sind beispielsweise Missionen hinterlegt. Diese geben bestimmte Spielziele vor, das heißt sie fordern vom Spieler bestimmter Areale des Spielfeldes mit einer Kugel zu treffen. Da die Mission weitreichenden Einfluss auf Zustand und Verhalten des Flipperautomaten hat, lässt sie sich gut als Zustand in der Steuerlogik abbilden: Bei Eintritt in eine Mission wird die Beleuchtung des Automaten angepasst. Gleichzeitig gelten spezifische Regeln der aktiven Mission, zu deren Prüfung bestimmte Zielschalter in Folge überwacht werden müssen.

### Command Pattern

Beim *Command Pattern* werden Objekte genutzt, um Verhalten zu kapseln. Die Kommando-Objekte werden vom Clienten erzeugt und an den Aufrufer weitergegeben. Dieser legt fest, wann die Kommandos ausgeführt werden, hat aber selbst keinerlei Kenntnis über den Empfänger, der durch die Kommandos manipuliert wird [45, Kapitel 5].

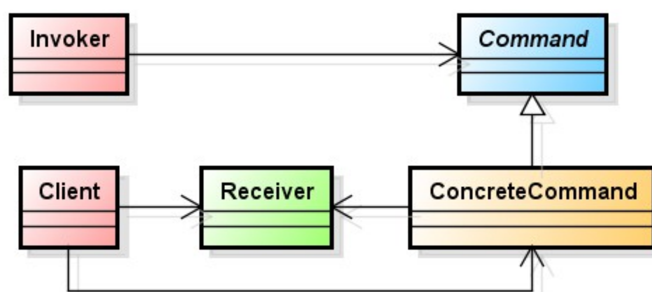



**Abbildung 11:** UML-Diagramm für Grundstruktur des *Command Patterns* nach [46, Seite 207].

Das *Command Pattern* findet ebenfalls bei der Konfiguration der Spiellogik Anwendung. Es ermöglicht, dass Änderungen mehrerer Komponenten des Flipperautomaten gebündelt werden.

Betrachtet man die obige Skizze, agieren in diesem Fall Module der hardwarenahen Schicht als Empfänger. Aufrufer ist eine Aktion mit erfüllter Bedingung, in der das Kommando hinterlegt ist. Die Initialisierung der Steuersoftware agiert als Client, er setzt gemäß seiner Konfiguration Kommandos und Ansteuerungsbefehle für Komponenten des Flipperautomaten zusammen.

Die Vorteile, die sich daraus ergeben, sind:

- Die Konfiguration des Zustandsautomaten bleibt schlank, da statt der Ansteuerung von Einzelkomponenten nur noch das Durchführen von Kommandos konfiguriert wird.
- Der Zustandsautomat wird völlig von der Hardware abstrahiert, weil er nicht mehr auf Einzelkomponenten des Flipperautomaten arbeitet.
- Durch die höhere Abstraktion wird auch die Testbarkeit erhöht, da Kommandos und Zustandsautomat getrennt getestet werden können.

Eine weitere Stelle wo das *Command Pattern* in abgewandelter Form Einsatz findet, sind die Handler, wie sie beispielsweise der **SwitchScanner** enthält: Dessen Erbauer erzeugt eine Scanner-Funktion für jedes Schalterregister und hinterlegt eine Liste derartiger Funktionen im **SwitchScanner**. Dieser ruft im Anschluss nur noch die Scanner auf, er selbst weiß nicht auf welches Bit unter welcher Registeradresse dadurch zugegriffen wird. Hier dient das *Command Pattern* zur Entkopplung von Abhängigkeiten.

#### Observer Pattern

Bei dem *Observer Pattern* verwaltet ein Subjekt eine Liste mit Beobachtern, welche bei Eintreten eines Ereignisses vom Subjekt informiert werden. Objekte die als Beobachter agieren wollen, implementieren die Benachrichtigungsschnittstelle und können sich beim Subjekt als Beobachter eintragen lassen [45, Kapitel 5].

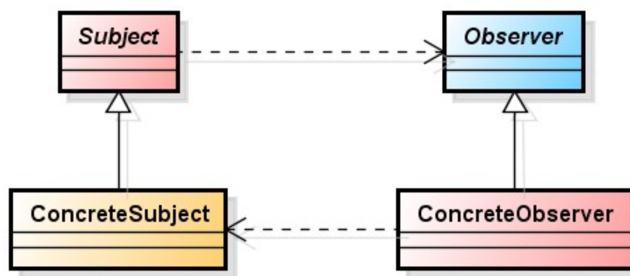



**Abbildung 12:** UML-Diagramm für Grundstruktur des *Observer Patterns* nach [46, Seite 52].

Dieses *Pattern* kommt bei der Ereigniswarteschlange von *Node.JS* zum Einsatz, und ist damit keine Designentscheidung, sondern durch die verwendete Plattform vorgegeben. Das Module **FlipperModel** erbt von **EventEmitter** [44]. An den Schalterereignissen interessierte Elemente der Spiellogik können sich als Beobachter für jeden Schalter eintragen und werden bei Ereigniseintritt informiert.

### Facade Pattern

Das *Facade Pattern* dient zur Einführung einer vereinfachten Schnittstelle zu einem Subsystem, mit dem Ziel die Abhängigkeiten zwischen Subsystemen zu minimieren. Anstatt die einzelnen Teilkomponenten direkt zu verknüpfen und so zahlreiche Abhängigkeiten zu erzeugen, erfolgt sämtliche Kommunikation über die Fassade [45, Kapitel 4].

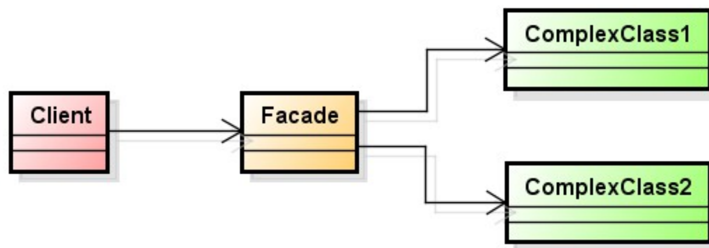



**Abbildung 2:** UML-Diagramm für Grundstruktur des *Facade Patterns*.

Das *Facade Pattern* fasst die beiden Treibermodule für Spulen und Lampen unter einer einheitlichen Schnittstelle zusammen. Die angebotenen Methoden tragen zudem stark verkürzte Namen, um deren häufigen Verwendung in den Spielregeln entgegenzukommen.

### Dependency Injection

Der Begriff *Dependency Injection* wurde erstmals von Martin Fowler verwendet [47, Kapitel 1]. Er beschreibt Mechanismen, bei denen Hilfsinstanzen nicht vom Konstruktor einer Klasse selbst erzeugt werden, sondern als Abhängigkeiten von

außen eingefügt werden. Dies sorgt für eine Bündelung der Konfiguration aller Abhängigkeiten an einem Ort. Die konkreten Klassen müssen nur bei der Konfiguration bekannt sein, implementiert wird rein gegen *Interfaces* [48].

Die Art *Dependency Injection*, die in dieser Arbeit zum Einsatz kommt, ist *Setter Injection* [48]. Durch Verwendung des *Builder Patterns* erfolgt diese nicht direkt in die Objektinstanzen, sondern in die Erbauer. Diese übergeben die Abhängigkeiten direkt an die erzeugte Objektinstanz. Dadurch kommt der Nachteil der *Setter Injection*, dass im Gegensatz zur *Constructor Injection* temporär ungültige, da nicht vollständige konfigurierte Instanzen erzeugt werden [48], nicht zum tragen. Der Erbauer stellt sicher, dass er Objekte erst zurückgibt, wenn sie vollständig und mit allen Abhängigkeiten konfiguriert sind.

**Builder Pattern**

Beim *Builder Pattern* wird das Erzeugen von Produkten an einen Erbauer delegiert. Dieser erzeugt die meist komplexen Instanzen auf Anweisungen des Direktors und bietet eine Schnittstelle zum Auslesen der fertiggestellten Instanz. Der Direktor benötigt Kenntnis der zur korrekten Erzeugung nötigen Schritte, nicht aber deren technische Details [45, Kapitel 3].

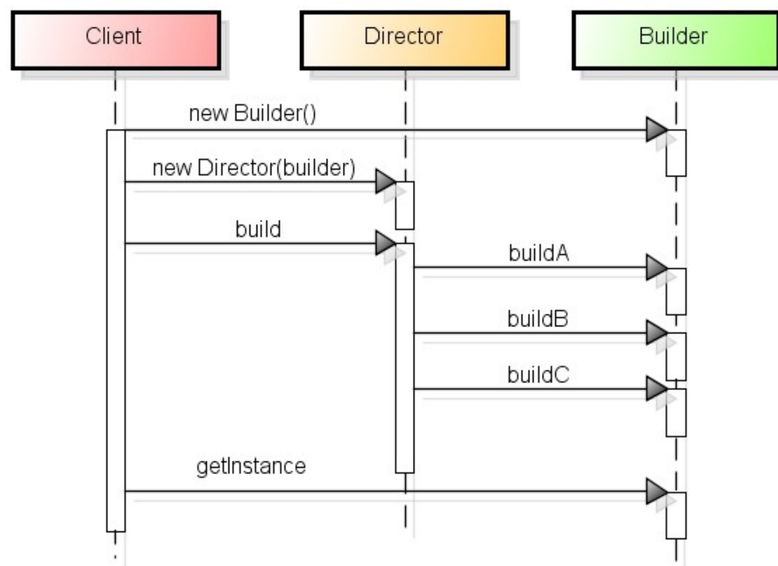



**Abbildung 3:** UML-Diagramm für Anwendung des *Builder Patterns* nach [45, Seite 99].

Das *Builder Pattern* kommt in der gesamten Software immer dann zum Einsatz, wenn die von einem Modul erstellten Objekte eine gewissen Komplexität überschreiten. Dies ist der Fall wenn die Anzahl der Abhängigkeiten zu groß scheint, um sie im Aufruf der *Factory-Methode* als Parameter mitzugeben. Die Erbauer bieten jeweils

passende bind-Methoden, mittels derer den Erbauern die Abhängigkeiten in Form von *Setter-Injection* übergeben werden. Aber auch die Erzeugung von sowohl temporärer Strukturen, als auch von Strukturen innerhalb der erzeugten Objektinstanzen gehören zu ihren Aufgaben. Dadurch wird die Funktionsumfang auf Erbauer und Objektinstanz verteilt, was für eine klare Trennung der Zuständigkeiten sorgt.

Als Direktor kommt ein zentrales Modul in jeder der beiden Abstraktionsebenen zum Einsatz. Das Erzeugen dieser beiden Module wiederum wird von der Hauptmethode vorgenommen.

# Kapitel 6

# Untersuchungen zu Coding Standards

Dieses Kapitel umfasst eine Reihe von Voruntersuchungen und Recherchen, die der Umsetzung des Entwurfs in eine Implementierung vorausgegangen sind. Die Notwendigkeit dieser ergab sich aus den besonderen Umständen des Projekts: Eine Skriptsprache, die primär für clientseitige Anwendung im Webbrowser gestaltet wurde, soll zur Ansteuerung von Hardware verwendet werden. Dies macht es notwendig, ganz andere Aspekte der Sprache zu untersuchen, aber auch gängige *Best Practices* ob ihrer Anwendbarkeit für diesen Fall zu hinterfragen. Denn nicht alles, was im Webapplikationsumfeld Sinn ergibt, muss auch in diesem speziellen Fall sinnvoll sein.

#### Variablenzugriff

*Literale* und lokale Variablen sind sehr schnell, ein Zugriff auf Arrays und Objekt-Felder langsamer. Deshalb empfiehlt es sich, wo möglich auf lokale Variablen zurückzugreifen. Optimierende *JavaScript*-Ausführungsumgebungen versuchen diese Zugriffe aber zu beschleunigen. Verschachtelte Objektzugriffe in Form von „x.y.z“ sind durch die Optimierung nicht teurer als andere Objekt-Feld-Zugriffe [10, Kapitel 2].

#### Bit-Operationen

*ECMAScript* arbeitet intern mit einem *Number*-Datentyp um Zahlen abzubilden, welcher als 64-Bit Gleitkommazahl gestaltet ist. Bit-Operationen in *ECMAScript* sind für vorzeichenbehaftete 32-Bit Ganzzahlen definiert [2, Seite 82]. Dies führt bei der Anwendung dieser Operationen zu einer doppelten Umwandlung: Zuerst von Gleitkommazahl zu Ganzzahl, dann wieder zurück zu Gleitkommazahl. Dies führt zu einer langsameren Ausführung der Operationen, als man es von anderen Programmiersprachen erwarten würde. Aus diesem Grunde werden Bit-Operationen von Douglas Crockford auch zu den „Bad Parts“ des Sprachstandards gezählt. Durch die langsamere Ausführung würden Bit-Operationen sehr selten zum Einsatz kommen und können so durch das seltene Auftreten leicht für fehlerhaft eigegebene boolesche Operationen gehalten werden [18, Anhang B]. Dem gegenüber steht die

Meinung von Nicholas C. Zakas. Dieser bestätigt zwar die Geschwindigkeitseinbußen, weißt aber auch darauf hin, dass Bit-Operationen in *JavaScript* im Vergleich zu booleschen oder mathematischen Operationen trotzdem schnell sind [10, Kapitel 8]. Da Bit-Operationen zudem einen intuitiven Ansatz darstellen, um auf einzelnen Bits innerhalb eines Registers zu arbeiten, soll in der Implementierung ein direktes Verarbeiten von Registerwerten mit Bit-Operationen verwendet werden.

**Parametertypen und Tests**

Bei dem Erstellen der Tests wurde angenommen, dass die Parameter der Funktionen stats sinnvoll auswertbar sind. Aufgrund der typlosen Natur von *JavaScript* können beliebige Objekte und Primitive als Parameter verwendet werden: Wird innerhalb der Funktion der Parameter als Zahl verwendet, mag ein String ein gültiger Parameter sein, sofern er sich als Zahl interpretieren lässt. Will man dies garantieren, müssen die Parameter stets aufwendig geprüft werden. Für korrekt auswertbare Parameter werden positive und negative Tests [49, Kapitel 4] geschrieben. Das Verhalten aller Funktionen für inkompatible Parameterwerte ist dadurch undefiniert. Parameter mit kompatiblen, aber ungültigen Wert werden korrekt (in der Regel durch Werfen einer *Exception*) behandelt. Eine Ausnahme wird hier das Auswerten von Konfigurationsdateien sein. Hier wird weitgehend auf Typ des Variablenwertes geprüft, um das Erstellen von korrekten Spielregeln zu erleichtern.

**Event-Mapping**

Ändert ein Schalter des Flipperautomaten seinen Zustand, wird dies von der Ansteuerungssoftware durch Auslösen einen *Events* signalisiert. Um die Zuordnung zwischen Schalter und Ereignis einfach zu halten, entspricht der Schaltername dem Ereignisnamen. Die Unterscheidung zwischen Ereignis „Schalter aktiviert“ und „Schalter deaktiviert“ erfolgt durch Übergabe eines booleschen Parameters an die Verarbeitungsfunktion des jeweiligen Schalterereignisses.

Aufgrund der Implementierung bietet es sich an beim Aussenden von Ereignissen maximal zwei Parameter zusätzlich zum Ereignisnamen zu verwenden:

```
if (arguments.length <= 3) {
  handler.call(this, arguments[1], arguments[2]);
} else {
  var args = Array.prototype.slice.call(arguments, 1);
  handler.apply(this, args);
}
```

**Quellcode 3:** Pseudo-Code für Verarbeitung der Parameter in **emit** nach [14, Seite 57].

Durch Verwendung des oberen Zweiges ist es möglich eine relativ aufwendige Array-Manipulation zu vermeiden [14, Kapitel 4]. Gemäß des Hinweises aus „*Node Up & Running*“ soll darauf geachtet werden, die Parameteranzahl der Rückruffunktion klein zu halten.

**Funktionsdefinitionen und Funktionsobjekte**

JavaScript bietet die Option anonyme Funktionen zu definieren. Dies kann eingesetzt werden, um eine Rückruffunktionen an der Stelle zu definieren, wo sie benötigt wird. Dies ist zwar ein intuitiver Ansatz, hat aber bei komplexerer Software einige Nachteile. Deshalb wurde sich bewusst gegen diese Möglichkeit des Sprachstandards entschieden. Funktionen werden nicht innerhalb anderer Funktionen definiert, sondern als Methoden an einen Objekt-Prototypen gebunden. Einzige Ausnahmen sind Funktionsobjekte, die als Rückgabewert dienen. Auch Rückruffunktionen werden im Produktivcode nicht intern erzeugt, sondern sind ihrerseits Rückgabewert einer Erzeugungsfunktion. Dies hat folgende Vorteile:

- Einfache Testbarkeit: Anstatt die Methode zu *mocken*, um beim Aufruf die ihr übergebene Rückruffunktion zu extrahieren und zu testen, hat diese einen klaren Namen und Ort und kann so direkt für Testzwecke aufgerufen werden.
- Die Wiederverwendbarkeit auch kleiner Methoden innerhalb des Objekts wird sichergestellt.

**Erzeugung von Hilfsobjekten ("Handler")**

Aufgrund der vielen Einzelkomponenten des Flipperautomaten wird für die gewählte Implementierung eine Vielzahl an Hilfsobjekten benötigt. So wird beispielsweise für jede Lampe ein Objekt benötigt, dass die jeweilige Lampe ansteuern kann und für diesen Zweck die Adressinformationen kennt. Durch Delegation dieser Funktion an Hilfsobjekte können diese Informationen gekapselt werden. Beim Erzeugen dieser Hilfsobjekte wird der Umstand ausgenutzt, dass Funktionen in *ECMAScript* bereits Objekte sind. Anstand einer Vielzahl an Objekten zu initialisieren, wird eine Gruppe Funktionen generiert. Deren Definition ist deutlich schlanker als die von Objekten mit gleicher Funktionalität.

Um die Funktionen mit den Ansteuerparametern zu verknüpfen existieren in *ECMAScript* zwei grundsätzliche Möglichkeiten: Die Variablen können mittels *Closure* der Funktion zur Verfügung gestellt werden, oder sie können mittels *Partial Application* an die Funktion gebunden werden.

```
function closureExample() {
  var someVar = "closure";
  return function innerFunction() {
    console.log(someVar)
  };
}
```

**Quellcode 4:** Beispiel für Funktion mit *Closure*.

*ECMAScript* bietet *Closures* als Mechanismus an, um auf Variablen außerhalb des Funktionsnamensraums zugreifen zu können. Im Beispiel heißt das, dass die Funktion **innerFunction** auf die Variable **someVar**, die außerhalb der Funktion definiert ist zugreifen kann.

```
function partialExample() {
  var someVar = "partial";
  return function() {
    console.log(someVar)
  }.bind(this, someVar);
}
```

**Quellcode 5:** Beispiel für Funktion mit *Partial Application*.

*Partial Application* hingegen hat erst mit der neusten Spezifikation Einzug in die Sprachwelt von *ECMAScript* gehalten [2]. Bei der *Partial Application* werden die Funktionsparameter einer Funktion mit Werten belegt. Das Ergebnis ist eine Funktion, deren Parameteranzahl um die der vorbelegten Parameter verringert ist.

Auch wenn der **bind**-Syntax zunächst von Vorteil schien, wurde er im Laufe der Implementierung wieder Fallen gelassen. Eine Analyse der zugrundeliegenden Funktion **FunctionBind(this_arg)** [50] zeigt aber potentielle Probleme auf. Die erzeugte Methode führt mehrere Überprüfungen nebst Array-Operationen bei jedem Aufruf durch. Dies erscheint unter den gegeben zeitkritischen Umständen nicht zielführend. Hintergrund ist, dass *ECMAScript* für **bind** viele Prüfbedingungen definiert [2].

Um die Eignung von beiden Techniken selbst zu überprüfen, wurde folgender Versuch durchgeführt: Ein **SwitchScanner** der implementierten Steuersoftware liest 30.000 mal sämtliche Schalterregister und die Schaltermatrix aus. Für jedes Schalteregister und jedes Matrixspalte ist eine Auslesefunktion vorbereitet. Diese sind einmal unter Nutzung der *Closure* und das andere mal mittels **bind** erzeugt worden:

```
[ 6, 295874242 ]
[ 6, 255758632 ]
[ 6, 295488925 ]
[ 6, 244524317 ]
[ 6, 271954945 ]
```

**Ausgabe 3:** Auslesen der Schalter mit *Closure*-basiertem *Scanner* in Sekunden und Millisekunden.

```
[ 6, 723600728 ]
[ 6, 724368778 ]
[ 6, 715204102 ]
[ 6, 668973840 ]
[ 6, 644287566 ]
```

**Ausgabe 4:** Auslesen der Schalter mit *Bind*-basiertem *Scanner* in Sekunden und Millisekunden.

Der augenscheinlich hoch optimierte Closure-Mechanismus der V8 wird deutlich schneller ausgeführt, weshalb erste Lösung für sämtlichen implementierten Hilfsfunktionen gewählt wurde.

#### Handhabung von Stringliteralen

Sowohl die einzelnen Elemente des Flipperautomaten als auch die zugehörigen Ereignisse sind durch Namen repräsentiert, die innerhalb des Programms als *Strings* abgebildet werden. Dies hat zwar gegenüber Sprachen wie beispielsweise Java den Vorteil, dass keine Event-Objekte erzeugt werden müssen. Der Nachteil ist aber, dass sich der Name als String in einer Vielzahl Dateien an unterschiedlichen Stellen wiederfindet, was eine nachträgliche Bearbeitung des Quellcodes erschwert. Um dies zu vermeiden, werden statt dessen die Namen in Form einer *Map* hinterlegt:

```
var Solenoids = {
  LEFT_SAUCER: 'LeftSaucer',
  LEFT_DROP_TARGET_UP: 'LeftDropTargetUp',
  LEFT_DROP_TARGET_DOWN: 'LeftDropTargetDown'
}
```

**Quellcode 6:** Beispiel für Defintion dreier Schalternamen.

In der weiteren Konfiguration können die so hinterlegten Namen eingesetzt werden, anstatt die Bezeichner an dieser Stelle erneut in Form von *Strings* einzusetzen: (Bei den Registern kommt ein analoger Ansatz zum tragen.)

```
var Flipper = {};
var solenoids = {};
solenoids[Solenoids.LEFT_SAUCER] = {
  register: Registers.SOLENOID_GROUP_A, index: 0, duration: 20
};
solenoids[Solenoids.LEFT_DROP_TARGET_UP] = {
  register: Registers.SOLENOID_GROUP_A, index: 1, duration: 20
};
solenoids[Solenoids.LEFT_DROP_TARGET_DOWN] = {
  register: Registers.SOLENOID_GROUP_A, index: 2, duration: 20
};
Flipper.solenoids = solenoids;
```

**Quellcode 7:** Beispiel für Konfiguration von drei Spulen.

Daraus ergeben sich zwei Vorteile:

- Mit den Werkzeugen zum *Refactoring* moderner IDEs kann die Vorkommen von Variablen und Feldnamen zuverlässig dateiübergreifend identifiziert und ersetzt werden, um beispielsweise eine falsche Benennung später zu korrigieren.
- Bei der Verwendung der Bauteilnamen kann die IDE eine Autovervollständigung sämtlicher hinterlegter Bauteilnamen anbieten. Dies macht Eingabefehler durch den Benutzer unmöglich und vereinfacht vor allem die Konfiguration der Spielregeln, bei der die Bezeichner besonders häufig eingesetzt werden müssen.

Nachteilig an diesem Ansatz ist, dass die Ausdrücke im Quellcode teilweise länger werden. Dieser Nachteil wurde aufgrund der Vorteile in Kauf genommen.

Auch aus Sicht der Fehleranfälligkeit ist dieses Vorgehen von Vorteil. So empfiehlt der Autor Mark Ethan Trostler in seinem Buch „*Maintainable JavaScript*“ dringend, bei Verwendung ereignisgestützter Architektur die Ereignisse in Form von *Enums* oder *Hashtables* zu hinterlegen: Ein Tippfehler bei einem Methodennamen führe zu einem Laufzeitfehler, ein falsch geschriebenes Event hingegen ist nicht einfach erkennbar [51, Kapitel 3]. Dies trifft in dieser Bachelorarbeit auf die Schalternamen zu, die direkt als Eregeignisnamen Verwendung finden.

**Code-Qualitätswerkzeuge**

Für *JavaScript* existieren Werkzeuge zur Verbesserung der Codequalität. Diese sorgen für das Einhalten von Codekonventionen und fangen potentielle Programmierfehler ab. Im Rahmen dieser Arbeit kam das Werkzeug *JSLint* [52] zum Einsatz. Hierbei zeigen sich die Voreinstellungen als sehr restriktiv. Aufgrund der Art der hier implementierten Anwendung wurden einige Warnungen abgeschwächt, wobei hier im besonderen zwei Einstellungen zu nennen sind:

- Bit-Operation: Wie in diesem Kapitel beschrieben, sollen für die Registerzugriffe Bit-Operationen zum Einsatz kommen.
- Warnung vor unbenutzten Variablen: Im Rahmen der Konfiguration kommen Funktionen mit fixer Parameteranzahl zum Einsatz, unabhängig davon, ob der Benutzer wirklich beide Parameter in seiner Regel verwendet. Dadurch ist ein einheitlicher Regelsyntax möglich.

# Kapitel 7

# Referenzimplementierung

## 7.1 Implementierung HAL

Bei der Implementierung wurden die im Entwurf beschriebenen Schichten umgesetzt und wo nötig durch Hilfsobjekte ergänzt. Nachdem im Entwurf dargestellt wurde, welche Module nötig sind, soll nun im folgenden auf einige Details der Implementierung eingegangen werden.

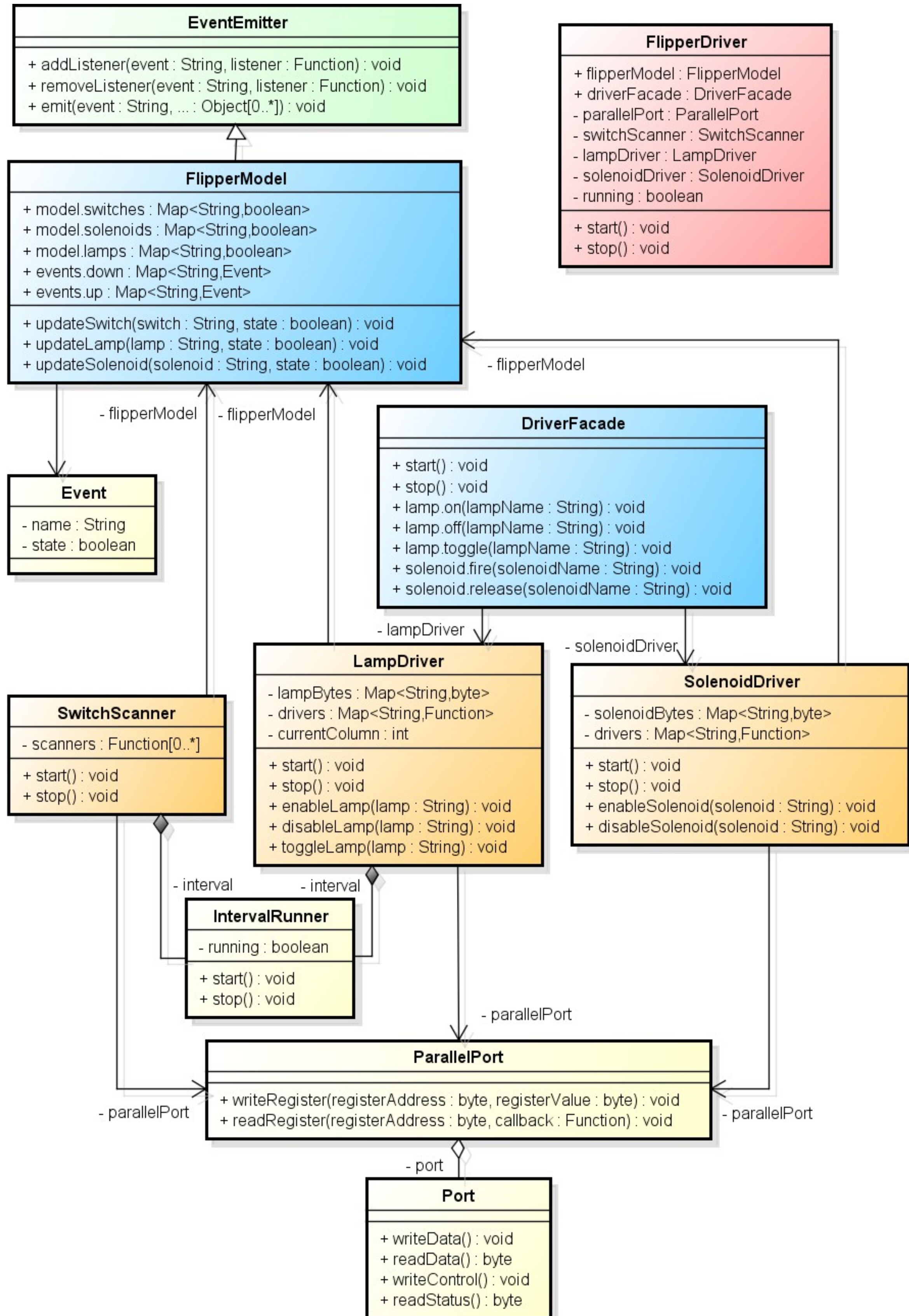



**Abbildung 4:** UML Klassendiagramm für die Schicht *HAL*.

**Kommunikation**

Um mit der Spiellogik-Schicht zu kommunizieren wird, wie im Entwurf vorgesehen, mit **FlipperModel** eine den gesamten Flipperzustand repräsentierenden Struktur erzeugt. Über die Aktualisierungsmethoden können die drei Hauptmodule dieser Schicht durch sie ausgelöste oder festgestellte Änderungen hinterlegen. Das Modell selbst wird den Modulen der Spiellogikschicht zur Verfügung gestellt, um stets den Zustand einzelner Komponenten prüfen zu können. Um diese Funktionalität zu erhalten, erbt **Flippermodel** durch *Prototype*-Vererbung von **EventEmitter** aus *Node.JS* [44]. Dadurch kann das *Model* das in *Node.JS* integrierte Ereignissystem nutzen. Als **EventEmitter** darf **Flippermodel** selbst keine Fehler werfen. Dies kann die Ereignisverarbeitung der Ereigniswarteschlange stören, so dass Ereignisse nicht mehr korrekt verarbeitet werden [23, Kapitel 5].

Zur Kommunikation mit der Hardware ist das Modul **Parallel-Port** vorgesehen. Dieses baut eine Verbindung zum Flipperautomaten auf und stellt eine API zur Verfügung um Register des Automaten zu lesen und zu schreiben. Beim Lesen von Registern wird eine Rückruffunktion neben der Registeradresse als Parameter erwartet, die das eingelesene Ergebnis verarbeiten kann. Dadurch wird die synchrone Leseoperation auf niedriger Ebene als asynchrone Kommunikation weitergegeben. Um mit dem Parallelport des Computers zu interagieren, wurde die in der Voruntersuchung zum Parallelport skizzierte Zugriffskette verwendet.

**Driver und Scanner**

Mit **SwitchScanner**, **LampDriver** und **SolenoidDriver** enthält die Schicht *HAL* drei Module, die sich jeweils um die Ansteuerung einer der drei Typen von Ein- und Ausgabekomponenten des Flipperautomaten kümmern. Bei Ausgabekomponenten des Flipperautomaten kann der Zustand gesetzt werden: Lampen können aktiviert, deaktiviert und umgeschaltet werden. Spulen können nur aktiviert werden, die Deaktivierung erfolgt durch die in der Konfiguration hinterlegte Betriebszeit automatisch. Schalter hingegen lösen bei Änderung eine Aktualisierung des *Models* aus.

Sowohl **SwitchScanner** als auch **LampDriver** müssen regelmäßig ihren internen Zustand mit dem Zustand des Flipperautomaten abgleichen. Zu diesem Zweck wurde das Hilfsmoduls **IntervalRunner** erstellt. Ein dafür geeigneter Zeitgeber wurde im Abschnitt „Zeitgeber mit Node.JS“ in Kapitel 4 erläutert. Der **IntervalRunner** ruft

nicht nur eine Rückrufmethode periodisch auf, sondern kann bei Start und Stopp des Intervalls weitere Rückruffunktionen ausführen. Dies wird beim **LampDriver** genutzt um bei Programmende alle Lampen zu deaktivieren.

Wie im Entwurf bereits skizziert, sollen Änderungen an den Spulen vom **SolenoidDriver** sofort an die Steuerplatine des Flipperautomaten kommuniziert werden. Die Lampen werden periodisch angesteuert, Steuerbefehle werden vom **LampDriver** also zunächst intern zwischengespeichert. Auch der **SwitchScanner** speichert intern den alten Zustand der von ihm verwalteten Bauteile. Dadurch kann er Änderungen erkennen und bei Bedarf das Modell des Flippers aktualisieren.

Im folgenden werden einige Abläufe, wie sie im Betrieb der Software vorkommen, in Form von Sequenzdiagrammen dargestellt:

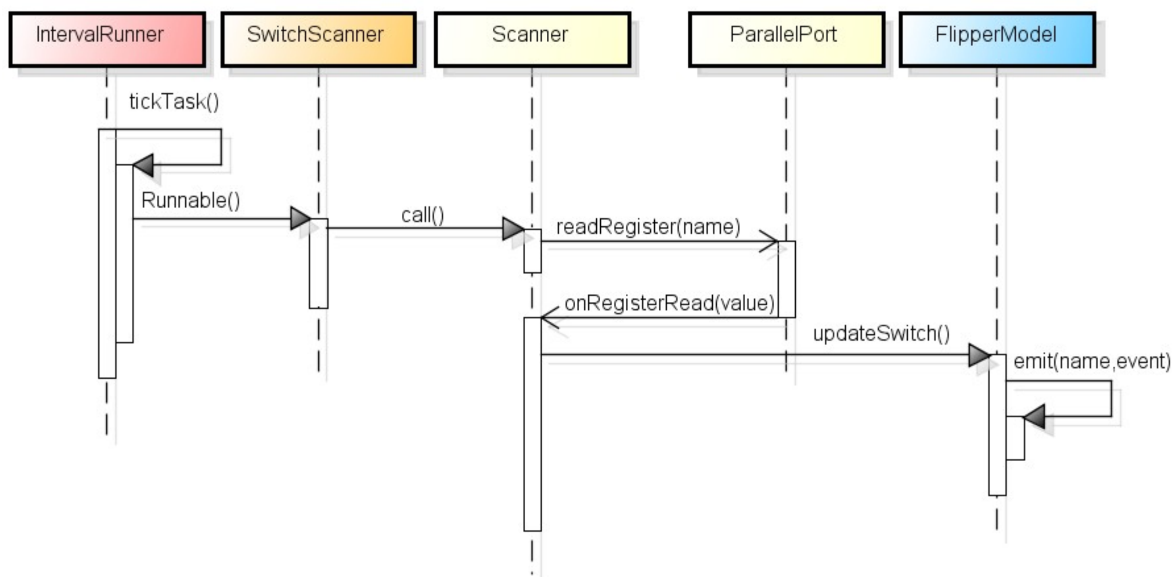



**Abbildung 13:** Periodischer Aufruf des Scanners eines Schalter. Das Einlesen erfolgt asynchron über die Rückrufmethode **onRegisterRead**. Eine erkannte Änderung führt zu einer Aktualisierung des Flippermodells, welches wiederum ein Schalterereignis auslöst.

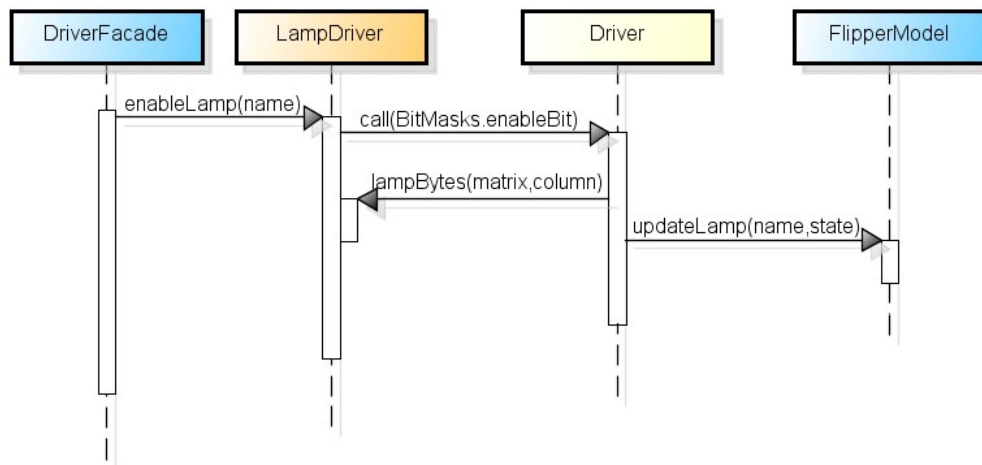



**Abbildung 14:** Aktivieren einer Lampe über den ihr zugeordneten Treiber . Dieser manipuliert eine hinterlegte Matrixzeile und aktualisiert das Flippermodell.

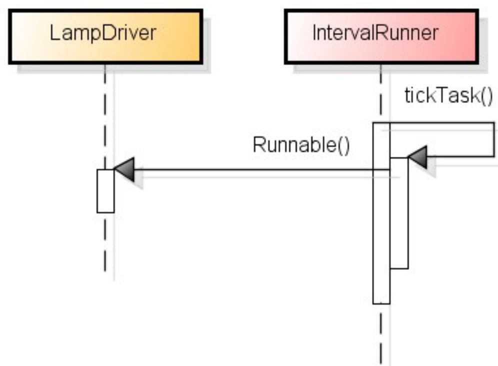



**Abbildung 15:** Periodisches schreiben der Lampenmatrix in die Register der Steuerplatine.

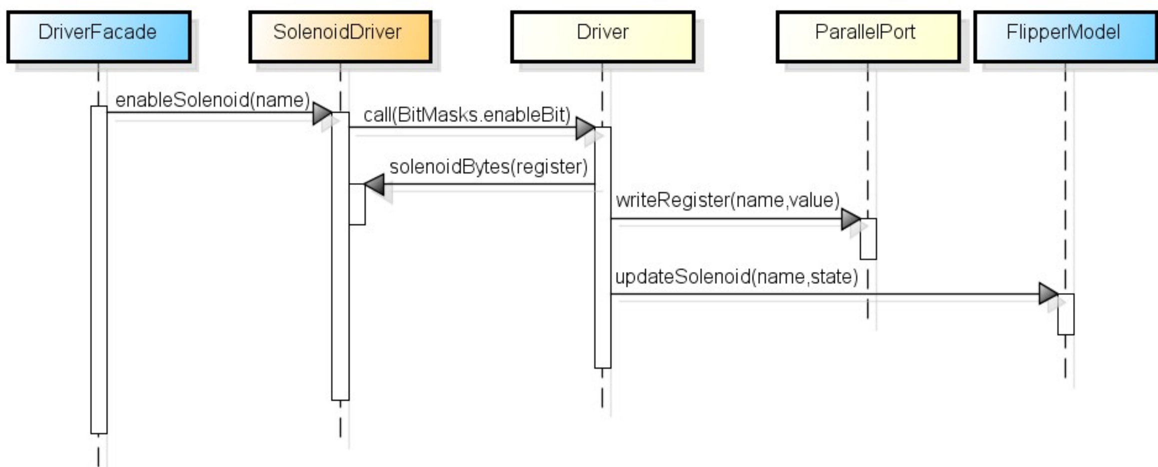



**Abbildung 16:** Aktivieren einer Spule über den ihr zugeordneten Treiber. Dieser manipuliert das hinterlegte Spulenregister, aktualisiert das Flippermodell und schreibt den Registerwert.

Der Hauptaufwand der Module liegt in der Initialisierung: Für jede Flipper-Komponente wird ein individuelle Ansteuerungsfunktion erzeugt und hinterlegt, um nicht ständig die entsprechenden Parameter aus den Datenstrukturen auslesen zu müssen. Hier findet die Eigenschaft von *ECMAScripts* Anwendung, Funktionsobjekte zu erzeugen. Dies wird bei der Beschreibung der Erbauer näher erläutert.

**Builder**

Wie im Abschnitt zum Entwurf bereits beschrieben, sollen zur Initialisierung der Software *Builder* [45, Kapitel 3] zum Einsatz kommen. Diese ermöglichen es, den zur Initialisierung nötigen Quellcode klar vom zum Betrieb nötigen zu trennen. Im folgenden soll der verwendete Ansatz näher erläutert werden.

Gemäß der Konvention in *Node.JS* exportieren die Module keine Konstruktoren, sondern eine Fabrikmethode [14, Kapitel 8]. Im Falle der mit Hilfe von Erbauern konfigurierten Objekten ist dies die Methode **createBuilder()**, die eine Instanz zur Erzeugung des eigentlich gewünschten Objekts enthält.

Im Falle von Modulen zur Ansteuerung von Komponenten des Flipperautomaten bekommt diese Fabrikmethode die Struktur des Automaten als Parameter mitgeteilt, die Signatur lautet **createBuilder(flipper)**. Mittels der Erzeugermethoden **bindModel(flipperModel)** und **bindPort(parallelPort)** werden mittels *Setter Injection*[6] passende Instanzen in den Erzeuger eingefügt. Durch diesen Mechanismus wird mit den **bind**-Methoden der Erbauer eine Entkopplung erreicht. Zusätzlich erleichtert man Tests, da man die Abhängigkeiten einfach durch *Mock*-Objekte austauschen kann. [49, Kapitel 2]. Ist der Erzeuger komplett initialisiert, kann anschließend mit **build()** die Erzeugung der jeweiligen Objektinstanz angestoßen werden. Dank *Chaining Pattern* [53, Kapitel 5] kann die Erzeugung von Instanzen in einer Zeile vorgenommen werden.

**build()** erzeugt eine neue Instanz des Hauptobjekts des jeweiligen Moduls und führt auf diesem bestimmte Initiierungsaufgaben durch, bevor die fertige Instanz als Rückgabewert die Methode verlässt. Zu diesen Aufgaben zählen:

1. Übertragen von Abhängigkeiten und notwendigen Teilen der Konfigurationsobjekte mittels *Constructor Injection*[6]. Hierbei werden die an den Erbauer gebundenen Hilfsinstanzen und Ausschnitte der im Erbauer hinterlegten Konfigurationsobjekte als Konstruktor-Parameter verwendet.

6 siehe Erläuterung im Abschnitt „Verwendete Entwurfsmuster".

2. Prüfen der Konfigurationsdaten. Diese wurden als Flipperstruktur im Erzeuger hinterlegt. Hier wird überprüft, ob alle notwendigen Felder vorhanden sind.

3. Erzeugen nötiger Hilfsstrukturen. Hiermit können einerseits temporär nötige Strukturen im Erbauer gemeint sein, wie es im folgenden Beispiel der Fall ist. Es kann aber auch eine Struktur direkt in der erbauten Instanz sein: Beispielsweise kann eine *Map* erzeugt werden, die für jedes Register den Registerinhalt in Form eines Byte-Wertes speichert.

4. Anlegen von speziellen Hilfsfunktionen zur Ansteuerung von Flipperkomponenten. Hierbei werden die nötigen Informationen direkt in der Funktion hinterlegt, technisch realisiert durch das Nutzen von *Closures*.

Gerade durch die letzten beiden Schritte ist der Anteil der Konfiguration, der in den Objektinstanzen hinterlegt wird sehr klein. Statt dessen wird die Konfiguration bereits vom Erbauer weitgehend ausgewertet und feingranular in Hilfsfunktionen hinterlegt. Durch diesen Aufbau wird ein hohes Maß an Datenkapselung erreicht [54, Teil A].

#### Beispiel SwitchScanner

Um die Aufgaben der Erbauer zu veranschaulichen, sollen im folgenden die Schritte vorgestellt werden, die der Erbauer des **SwitchScanners** durchführt. Der fertig initialisierte **SwitchScanner** hat die Aufgabe, die Schalterzustände aus den Registern lesen und im Modell zu aktualisieren.

1. Setzen der Abhängigkeiten und Konfigurationswerte

Als Abhängigkeiten benötigt der Scanner weder die an den Erbauer gebundene Referenz auf das **FlipperModel**, noch den **ParallelPort**. Beide werden ausschließlich direkt in den Hilfsfunktionen verwendet. Dies ist in anderen Modulen anders, da beispielsweise der **SolenoidDriver** direkt die Initialisierung der Spulen durchführt, für welche er Zugriff auf den Parallelport benötigt.

Als Konfigurationsobjekt hat der Erbauer die Gesamtkonfiguration des Flippers zur Verfügung. Aus dieser entnimmt er die Intervallzeit, in der die Schalter abgefragt werden sollen, aus dem Feld **flipper.intervals.switches**. Dies ist der einzige Wert, der für den **SwitchScanner** direkt von Belang ist.

2. Überprüfung der Konfigurationsdaten

Bei der Überprüfung der Konfiguration muss die Vollständigkeit jedes Schaltereintrags überprüft werden. So muss Spalte und Zeile angegeben sein, falls es sich um einen in der Schalter-Matrix hinterlegten Schalter handelt. Anderenfalls muss Register und Index für einen in einem Register hinterlegten Schalter vorhanden sein. Hierfür wird auf **flipper.switches** zurückgegriffen. Jedes Feld in diesem Objekt stellt einen Schalter dar, dessen Felder wiederrum die genannten Bedingungen erfüllen müssen.

3. Erzeugung von Hilfsstrukturen

Die Schalter sind in **flipper.switches** unter ihrem Namen organisiert. Anders als bei den Treibern, bei denen ein Steuerbefehl für eine bestimmten Namen vorliegt, und anschließend auf eine bestimmte Registeradresse oder Matrixzeile zugegriffen werden muss, ist hier der umgekehrte Weg erforderlich. Aus diesem Grund werden im Erbauer des **SwitchScanners** zwei Tabellen angelegt: Eine sortiert die direkt in Registern hinterlegten Schalter nach der Registeradresse, eine zweite die in der Schaltermatrix hinterlegten nach der Schalterspalte.

4. Anlegen der Hilfsfunktionen

Nachdem alle Schalter sortiert sind, kann für jedes Register und jede Matrixspalte eine *Scanner*-Funktion erzeugt werden. Unter Nutzung von *Closures* werden in dieser Funktion sowohl Registeradresse bzw. Matrixzeile hinterlegt, als auch die Information welche Schalternamen sich hinter den einzelnen Bit verbergen. Dies hat den Vorteil, dass die in Schritt drei erzeugten Zuordnungstabellen im Betrieb nicht mehr benötigt werden. Der **SwitchScanner** enthält lediglich eine Liste derartig erzeugter *Scanner*, welche er regelmäßig ausführen muss. Kommt ein solcher *Scanner* zur Ausführung, ließt er den aktuellen Byte-Wert ein, überprüft die Bits auf Änderung und stößt mit den hinterlegten Schalternamen Aktualisierungen im **FlipperModel** an.

#### Initialisierung

Um die *HAL* einfach zu starten, wird für die Verwaltung der Schicht ein Modul **FlipperDriver** eingeführt. Dieses erzeugt alle der bisher vorgestellten Module und verknüpft sie untereinander passend. Das so erzeugte Objekt bietet direkten Zugriff auf **FlipperModel** und **DriverFacade**. Hierdurch genügt es der Spiellogik die **FlipperDriver**-Instanz bekannt zu machen, um ihr zu ermöglichen auf alle benötigten Module zuzugreifen. Zusätzlich werden mit **start()** und **stop()** zwei Methoden angeboten, die das Starten und Stoppen der Intervallzeitgeber in **DriverFacade** und

**SwitchScanner** anstoßen. **DriverFacade**, der als Fassade zu den Ansteuerungsmodulen **LampDriver** und **SolenoidDriver** dient, delegiert den Start wiederum an die beiden enthaltenen Treiber.

#### API

Zur Kommunikation mit der Spiellogik kann unter Berücksichtigung der vorhergegangenen Abschnitte folgende API definiert werden.

Für eine von Modul **FlipperDriver** erzeugte Instanz:

- Für den einfachen Zugriff auf die internen Module sind in **flipperModel** und **driverFacade** die gleichnamigen Module hinterlegt.
- **start()** und **stop()** ermöglichen ein zentrales Steuern der internen Zeitgeber.

Der **FlipperDriver** selbst kann über **createDriver(flipper)** aus einer Flipper-strukturdefinition erzeugt werden.

Für eine von Modul **FlipperModel** erzeugte Instanz:

- Das Modell des Flippers enthält eine Struktur **model** mit Schalter, Lampen- und Spulenzuständen.
- Auf dem Modell können mittles **addListener(switchName, listener)** Ereignisse registriert und mittels **removeListener(switchName, listener)** deregistriert werden [44].

Für eine von Modul **DriverFacade** erzeugte Instanz:

- Zur Ansteuerung der Lampen stehen die Methoden **lamp.on(lampName)**, **lamp.off(lampName)** und **lamp.toggle(lampName)** zur Verfügung.
- Zur Ansteuerung der Spulen kann **solenoid.fire(solenoidName)** genutzt werden. Haltespulen werden mit **solenoid.release(solenoidName)** deaktiviert.

## 7.2 Konfiguration HAL

Die Konfiguration der *HAL* besteht zunächst aus der Definition von Registernamen und zugehörigen Adressen, des weiteren aus der Beschreibung der Struktur des Flipperautomaten. Die Konfiguration ist auf zwei Orte verteilt: Zum einen gibt es die Kommunikation und Ansteuerung betreffenden Daten. Die Information, dass die Lampen über Matrizen angesteuert werden findet zum Beispiel im Lampentreiber

Anwendung. Da eine Änderung an diesen Einstellungen auch Änderungen in der Implementierung nach sich ziehen würde, sind diese Konfigurationssammlungen in den Quellordner hinterlegt.

Bauteilnamen und -parameter hingegen können geändert werden, ohne die Implementierung anzupassen. Derartige Einstellungen sind daher in einem Konfigurationsordner hinterlegt.

#### Ansteuerung des Flipperautomaten

Zur Kommunikation mit dem Flipperautomaten sind zwei Dinge nötig: Einerseits werden für das Kommunikationsprotokoll die Bedeutungen der einzelnen Steuer- und Datenleitungen benötigt, des weiteren müssen die in der Steuerplatine verwendeten Registeradresse gesammelt werden.

Der Zugriff auf die Parallelschnittstelle erfolgt byteweise, da die Leitungen in Gruppen gebündelt sind. Die Bedeutung der einzelnen Bits ist im Modul **BitMasks** hinterlegt. Dieses enthält zusätzlich einige Hilfsfunktionen, um ein einfaches Arbeiten mit Bitmasken zu ermöglichen.

Die verwendeten Register hingegen sind im Modul **Registers** gesammelt. So kann unter anderem in der weiteren Strukturkonfiguration mit Namen anstatt mit Zahlenwerten für Register gearbeitet werden. Die speziellen Spulen, wie das Spannungsrelais, dass die anderen Spulen mit Strom versorgt, werden in **Relay** hinterlegt. Dadurch ist sichergestellt, dass ihre Namen nicht in der Spulenliste auftauchen. Damit ist die versehentliche Ansteuerung durch den Ersteller der Spielregeln erschwert. Die Ansteuerinformationen selbst liegen wie bei normalen Spulen im Modul **Flipper**, welches im nächsten Abschnitt vorgestellt wird.

#### Struktur des Flipperautomaten

Mit **Lamps**, **Solenoids** und **Switches** liegen im Konfigurationsordner drei Module vor, in welchen die Namen der Komponenten hinterlegt sind. Diese Sammlung dient zur Identifizierung der einzelnen Bauteile, sie wird sowohl in der weiteren Konfiguration und vor allem bei der Regeldefinition angewandt. Die diversen Vorteile derartiger Strukturen im Vergleich zur Verwendung von Strings als Identifikatoren wurde im Kapitel 6 erläutert.

Im Modul **Port** ist die Geräteadresse der Parallel-Schnittstelle hinterlegt.

Im Modul **Flipper** sind die Ansteuerparameter für jede Komponente hinterlegt. Im Fall direkt über Register ansprechbarer Bauteile handelt es sich hierbei um Registeradresse und Bit-Index des Bauteils. Für über Matrizen angesteuerte Bauteile ist hingegen Zeile und Spalte in der jeweiligen Matrix hinterlegt. Da es für Lampen zwei Register für die Matrixreihe gibt, wird hier die Reihenadresse selbst zusätzlich hinterlegt. Für die Spulen, als einzige Bauteilgruppe mit individuellen Ansteuerparametern, ist zusätzlich die erlaubte Aktivierungsdauer gespeichert. Zusätzlich sind in diesem Modul die Zeitintervalle für das Einlesen der Schalter sowie das Schreiben der Lampenzeilen hinterlegt.

Das Ergebnis dieser Konfigurationsdateien ist eine Beschreibung der kompletten Struktur des Flipperautomaten. Aus diesem können die Erbauer der einzelnen *HAL*-Module die notwendigen internen Strukturen zur Bauteilansteuerung erzeugen.

## 7.3 Implementierung Spiellogik

In der Spiellogik wird durch die Anwendung von Spielregeln Ansteuerinformationen für den Flipperautomaten erzeugt. Analog zu den beiden vorhergehenden Abschnitten soll bei der Beschreibung der Implementierung vor allem auf Besonderheiten eingegangen werden.

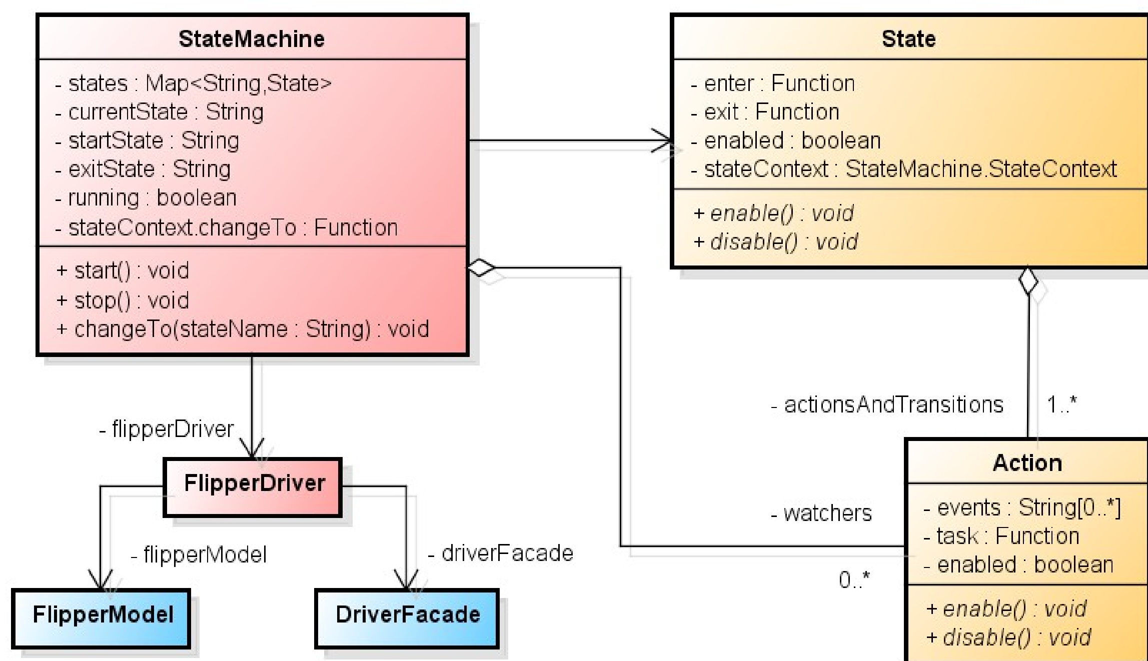



**Abbildung 17:** UML Klassendiagramm für die Schicht StateMachine.

#### Zustandsautomat

Die Umsetzung des Zustandsautomaten, bestehend aus **StateMachine** und **State**, ist sehr nahe am *State Pattern* [45, Kapitel 5] erfolgt. Diese Module dienen primär dem Laden und Initialisieren ihrer Konfiguration. Ihre Struktur wird von der Konfiguration festgelegt und wird dort näher erläutert, die von ihnen verwendeten Hilfsstrukturen werden im folgenden beschrieben.

#### Zeitgeber

Um bei der Erstellung von komplexen Spielregeln zu unterstützen wurde eine Reihe von Zeitgebern implementiert:

- **OneShotTimer** führen eine Funktion einmalig aus.
- **IntervalTimer** führen eine Funktion periodisch aus.
- **OneShotSequences** führen eine Liste von Funktionen aus.
- **IntervalSequences** führen eine solche Liste zyklisch aus.

Diese Zeitgeber stehen in zwei verschiedenen Modulen zur Verfügung: **PrecisionTimer** und **Timer**. Während die ersteren sich auf die für die *HAL* implementierten **IntervalRunner** stützen, setzen die normalen Zeitgeber auf die *JavaScript*-interne Funktion **setTimeout**. Im normalen Betrieb, wenn weder hohe Genauigkeit noch Zeiten unter einer Millisekunde erreicht werden müssen, sollte auf die normalen Zeitgeber zurückgegriffen werden. Diese dürften aufgrund der einfacheren Implementierung und der Nutzung von Funktionen der Ausführungsumgebung performanter sein.

#### Builder

Analog zum Ansatz in der *HAL* kommen auch im Zustandsautomaten Erbauer zum Einsatz. Diese bekommen den für sie relevanten Ausschnitt der Konfiguration in Form eines *Object Specifiers* [18, Kapitel 5] als Konstruktorparameter. Da es sich bei den Konfigurationsfelder oftmals um Funktionen mit Parametern handelt, muss sichergestellt werden, dass diese bei Aufruf spezifiziert werden. Wie dies bereits in den Erbauern vorbereitet wird, wird im nächsten Abschnitt beispielhaft anhand des Moduls **Action** beschrieben.

### Action, Transitions und Watcher

Im Entwurf wurden die drei Elemente *Action*, *Transition* und *Watcher* erwähnt. Allen dreien gemeinsam ist, dass sie in Abhängigkeit einer Bedingung aktiviert werden. Aufgrund dieser Gemeinsamkeiten wurden alle drei in Form eines Moduls **Action** umgesetzt. In diesem Modul ist der Erbauer verantwortlich, die verschiedenen Funktionen aus den Konfigurationsobjekten in passendem Kontext zu hinterlegen:

*Action* und *Watcher* lösen hierbei eine Kommando aus. Zur Aktivierung durch ein Ereignis wird eine Rückruffunktion benötigt. Diese wird nach folgendem Ansatz generiert:

```
return function Task(event) {
    if (condition(event, model)) {
        command(solenoid, lamp)
    }
};
```

**Quellcode 8:** Code zur Erzeugung einer Task-Funktion mit Bedingung und Kommando aus der Konfiguration.

**condition** und **command** sind Funktionen, die als Teil der Konfiguration hinterlegt sind. **model**, **solenoid** und **lamp** werden außerhalb der Methode hinterlegt und sind über deren *Closure* innerhalb der generierten Funktion bekannt. Ein analoger Ansatz kann für die *Transition* ergriffen werden:

```
return function Transition(event) {
  if (condition(event, model)) {
    context.changeTo(state);
  }
};
```

**Quellcode 9:** Code zur Erzeugung einer Transition-Funktion mit Bedingung und Zustand aus der Konfiguration.

Statt ein Kommando auszuführen wird hier wiederum über die *Closure* auf den **context** zugegriffen und über dessen Methode auf einen anderen Zustand gewechselt. Die Variablen **model** und **state** sind ebenfalls in der *Closure* definiert.

Das gleiche Vorgehen kommt auch bei den Zuständen selbst zum Einsatz. Deren Konfiguration kann Funktionen enthalten, welche bei Betreten und Verlassen des Zustands aufgerufen werden. Diesen Initialisierungsfunktionen stehen Referenzen auf Modell und die beiden Treiberschnittstellen zur Verfügung:

```
return function TransitionCallback() {
  callback(model, solenoid, lamp);
};
```

**Quellcode 10:** Code zur Erzeugung einer Enter- oder Exit-Funktion mit Rückruffunktion aus der Konfiguration.

#### API

Zur Verwendung der Spiellogik steht folgende API zur Verfügung:

Für eine von Modul **StateMachine** erzeugte Instanz:

- Der verwendete Flipper-Treiber ist in **flipperDriver** in Form eines gleichnamigen Moduls hinterlegt.
- **start()** und **stop()** ermöglichen ein zentrales Steuern der internen Zeitgeber.

Die **StateMachine** selbst kann durch Erzeugung eines Erbauers mittels **createBuilder(states, watchers)** aus einer Zustands- und Überwacher-Konfigurationen erzeugt werden. Anschließend wird mit **bindDriver(flipperDriver)** eine Instanz des Flipper-Treibers gebunden und mit **build()** die Erzeugung abgeschlossen.

## 7.4 Konfiguration Spiellogik

Das Umsetzen der Spielregeln war aufgrund des enormen Umfangs nicht Teil dieser Arbeit. Um aber die Möglichkeiten, die der Zustandsautomat bietet, zu dokumentieren, wurde eine einfache Beispielkonfiguration angelegt. In dieser sind die Anforderungen und Möglichkeiten nochmals in Form von Kommentaren hinterlegt. Eine Begründung für die Wahl des Aufbaus und der Implementierung der Konfiguration findet sich im Abschnitt „Struktur der Konfiguration“ in Kapitel 5.

#### States

Bei diesem Modul handelt es sich um eine Auflistung aller Zustände für den Zustandsautomaten. Jeder Zustand hat seinerseits ein gleichnamiges Modul im Konfigurationsordner, dass bei der Initialisierung dynamisch nachgeladen wird. Zwei der Schlüssel in diesem Modul haben besondere Bedeutung: Als Startzustand ist der Name eines Zustands angegeben, der bereits in der normalen Auflistung enthalten ist. Dieser wird bei Start der Zustandsmaschine aktiviert. Des weiteren ist ein Zustand zum Beenden des Automaten hinterlegt. Bei diesem handelt es sich um einen virtuellen Zustand, der ein Herunterfahren von Zustandsautomat und *HAL* auslöst. Dieser Zustand hat daher keine gleichnamige Konfigurationsdatei.

### DefaultState

Beim Modul **DefaultState** handelt es sich um einen Beispielzustand, der die Möglichkeiten der Konfiguration demonstrieren soll.

```
{
  enter: function (model, solenoid, lamp) { ... },
  exit: function (model, solenoid, lamp) { ... },
  transitions: {
    transition1: {
      events: [ ... ],
      condition: function(event, model) { ... },
      state: ...
    }
  },
  actions: {
    action1: {
      init: function(solenoid, lamp) { ... },
      events: [ ... ],
      condition: function (event, model) { ... },
      command: function(solenoid, lamp) { ... }
    }
  }
}
```

**Quellcode 11:** Vorlage für eine Zustandskonfiguration in Form eines Objektliterals.

Zunächst enthält die Konfiguration zwei Rückruffunktionen **enter** und **exit**, welche mit jeweils drei Parametern beim Aktivieren bzw. Deaktivieren des Zustands aufgerufen werden. Durch diese haben die Funktionen Zugriff auf Flippermodell sowie den Lampen- und Spulentreiber. Diese Rückruffunktionen sind optional.

Die nächste Komponente der Konfiguration ist eine Liste mit **transitions**, welche Übergangsbedingungen in andere Zustände konfigurieren. Jeder dieser Einträge besteht aus einer Liste mit Ereignissen, einer Prüfbedingung und den Namen des Zustands in den gewechselt werden soll.

Des weiteren sind hier Aktionen in der Auflistung **actions** definiert. Diese sind analog zu einer Transition aufgebaut, enthalten aber statt dem Namen des Zustands, eine Kommandofunktion, die bei Erfüllen der Bedingung ausgelöst wird.

### Watchers

In diesem Modul werden Aktionen konfiguriert, die unabhängig vom aktuell aktiven Zustand ausgeführt werden sollen. Ihr Aufbau entsprecht dem des in den Zuständen hinterlegten **actions** Block.

### Transition, Action und Watcher im Detail

Im folgenden soll die Konfiguration von Transitionen und Aktionen, welche im letzten Abschnitt bereits eingesetzt wurden, näher erläutert werden.

Das Feld **events** enthält eine Auflistung aller Namen derjenigen Schalter, die bei Statusänderung die Prüfung der Bedingung auslösen sollen.

Diese wiederum ist im Feld **condition** hinterlegt. Diese Funktion erhält als Parameter das Ereignis **event**, welches die Felder **name** und **state** bietet. Ersteres ist der Name des auslösenden Schalters und letzteres der Zustand des Schalters als boolescher Wert. Der zweite Parameter **model** hingegen bietet mit **solenoids**, **lamps** und **switches** drei *Maps*, welche den Gesamtzustands des Flippers beschreiben.

Das Feld **state** legt bei Transitionen den Namen des Zustands in den gewechselt werden soll fest.

Aktionen enthalten statt dessen das Feld **command**. Hierbei handelt es sich um eine Funktion, die mit zwei Parametern aufgerufen wird. Der erste Parameter **solenoid** bietet die Methoden **fire(name)** und **release(name)** an, mit denen Spulen manipuliert werden können. Der zweite Parameter **lamp** ermöglicht dies für Lampen. **on(name)**, **off(name)** und **toggle(name)** stehen hier zur Verfügung.

Das Feld **init**, welches nur bei Aktionen ausgewertet wird, besteht aus einer Funktion, die zur Initialisierung gedacht ist. Sie erhält mit **solenoid** und **lamp** die gleichen Parameter wie ein Kommando.

In **Watchers** werden die hier beschriebenen Möglichkeiten unter anderem eingesetzt, um die Finger des Flipperautomaten mit den dafür vorgesehenen Eingabetasten zu verknüpfen. Im **DefaultState** wird beispielhaft eine Lampensequenz bei der Initialisierung vorbereitet, und im Betrieb auf Tastendruck aktiviert. Für mehr Beispiele nebst ausführlicheren Erläuterungen sei auf die gleichnamigen Konfigurationsdateien verwiesen.

### Globals und Locals

Ein Problem, dass es zu lösen galt, war die regelübergreifende Kommunikation. Zusätzlich war es aufgrund des Umstands, dass Regeln aus Funktionen bestehen notwendig, eine Struktur anzubieten, in der diese bei Aufruf Informationen persistent hinterlegen können. Hierfür wurden zwei Konventionen innerhalb der Konfiguration eingeführt. Zum einen enthält jeder Zustand und auch die Konfiguration der *Watcher* ein leeres Objekt unter dem Namen **Local**. Dieses kann von den Regeln in Form einer *Map* genutzt werden.

Zusätzlich steht in jeder Zustandskonfiguration das Modul **Global** zur Verfügung. Hier können zustandsübergreifende Informationen analog zu **Local** hinterlegt werden. Diese Struktur eignet sich beispielsweise zum Mitzählen der vom Spieler erzielten Punkte. Eine derartige Verwendung eines Moduls ist möglich, da *Node.JS* Module nur einmalig lädt [55]. Dadurch teilen sich alle Zustände die gleiche Instanz des **Global**-Moduls.

#### Beispiel zur Übernahme einer XML-Konfiguration

Im folgenden soll anhand eines Beispiels aufgezeigt werden, wie aus der *XML*-Konfiguration des *Java*-Projekts von Kevin Kratzer [3] eine *JavaScript*-Konfiguration erstellt werden kann. In Projekt von Hern Kratzer besteht die Möglichkeit, Lampensequenzen zu hinterlegen. Anhand des Quellcodes lässt sich nachvollziehen, dass dieser *XML*-Ausschnitt zur Instanzierung einer Lampenensequenz-Klasse, welche ihrerseits eine Warteschlangenimplementierung verwendet, genutzt wird.

```
<sequence timingMillis="200" timingNanos="0">
  <scene>
    <lampEvent type="Off" matrix="A" lamp="61" />
    <lampEvent type="On" matrix="A" lamp="63" />
  </scene>
  <scene>
    <lampEvent type="Off" matrix="A" lamp="63" />
    <lampEvent type="On" matrix="A" lamp="61" />
  </scene>
</sequence>
```

**Quellcode 12:** Konfigurationsbeispiel entnommen aus Kevin Kratzers Arbeit [3, Seite 47].

Bei der Umsetzung in *JavaScript* soll nun ohne spezielle Klasse im Hintergrund die Konfiguration innerhalb einer **Action** erfolgen. Hierzu wird das **init**-Feld der Konfiguration genutzt, um einen Zeitgeber mitsamt der Lampenansteuerung zu hinterlegen. Die Sequenz kann nun jederzeit mit **start()** und **stop()** kontrolliert werden, etwa im Rahmen eines Kommandos. Derartig ist auch der Start des Zeitgebers im folgenden Beispiel hinterlegt.

```
{
  init: function (solenoid, lamp) {
    Locals.myLampSequence = Timer.createIntervalSequence([
      function () {
        lamp.off(Lamps.RIGHT_STANDUPS_UPPER);
        lamp.on(Lamps.RIGHT_STANDUPS_LOWER);

      },
      function () {
        lamp.on(Lamps.RIGHT_STANDUPS_UPPER);
        lamp.off(Lamps.RIGHT_STANDUPS_LOWER);
      }
    ], 200);
  }
  ...
  command: function(solenoid, lamp) {
    Locals.myLampSequence.start();
  }
}
```

**Quellcode 13:** Ausschnitt einer **Action**-Definition mit hinterlegtem Sequenz-Zeitgeber zur Ansteuerung einiger Lampen.

Analog zu diesem Beispiel können auch die anderen Teile der Konfiguration umgesetzt werden.

## 7.5 Fehlerbehandlung

Fehler werden in der Implementierung bewusst erst auf höchster Ebene gefangen. Grund für diese Entscheidung ist der Umstand, dass eine fehlerhafte Software (oder Konfiguration) Schäden am Flipper auslösen kann. Werden beispielsweise die Schalter noch regelmäßig ausgelesen, schaltet die Platine des Flipperautomaten nicht ab[7]. Ist in dieser Zeit der Spulentreiber inaktiv, kann es zur Überschreitung der maximalen Aktivierungszeiten bei einzelnen Spulen kommen. Aus diesem Grund muss muss die Software zuverlässig komplett gestoppt werden. Dies kann zuverlässig erreicht werden, in dem die Ausnahmen erst auf oberster Ebene im Rahmen der **main**-Funktion gefangen werden.

Durch das Aufrufargument **verbose** erhält der Anwender zudem die Möglichkeit, die Fehler komplett auf der Konsole auszugeben. So ist es möglich über Angabe der Zeilennummer und des *Stacktraces* den Fehler näher zu untersuchen. Bei fehlerhaften Konfigurationsdateien werden zahlreiche verschiedene Fehlermeldungen genutzt, welche den fehlerhaften Ausschnitt aus der Konfiguration ausgeben, sofern dies an der Stelle sinnvoll erscheint.

7 Die Platine des Flipperautomaten ist aktiv, solange regelmässig auf die Schaltermatrix zugegriffen wird. Ausbleibende Zugriffe auf die Spulenregister bleiben hingegen für die Ansteuerung ohne Folgen.

## 7.6 Testen

Tests wurden in zwei verschiedenen Formen erstellt: Zum einen wurden die beiden Ebenen der Softwareimplementierung mit *Unit-Tests* abgedeckt. Hierbei wurde besonderes Augenmerk auf die korrekte Überprüfung der Konfiguration gerichtet. Der Nutzer, der Spielregeln erstellen will, ist hier auf hilfreiche Rückmeldung angewiesen ist. Diese Tests sind für automatische Auswertung vorgesehen. Zum anderen wurden Test implementiert, mit der ein Benutzer mit dem Flipperautomaten und dem Zustandsautomaten interagieren kann.

#### Unit-Tests

Um bei der Entwicklung regelmäßig Rückmeldung über den aktuellen Zustand der Software zu erhalten, wurde sowohl für *HAL* als auch für die **StateMachine** ein Satz *Unit-Tests* erstellt. Als *Test-Framework* kommt hierbei *Nodeunit* [56] zum Einsatz.

Das Laden von Konfigurationsmodulen und das Erzeugen von Objekten wurden zwar mit Hinblick auf die Testbarkeit weitgehend mittels *Dependency Injection*[8] an zentrale Stellen gebündelt. An Stellen wo ein solches Vorgehen nicht ratsam erschienen ist und Module direkt geladen werden, kann es für die Tests nötig sein das zu testende Modul mit *gemockten* Abhängigkeiten zu laden. Für diesen Zweck kommt die *Node.JS*-Erweiterung *Proxyquire* zum Einsatz. Ihr Vorteil ist, dass am Produktivcode keinerlei Änderungen erfolgen müssen und sich die Verwendung der Erweiterung rein auf den Testcode beschränken kann [57].

Mit 88 Tests für die *HAL* und 78 Tests für die **StateMachine** konnte eine gute Testabdeckung erreicht werden, welche sich bei Erstinbetriebnahme der Software als sehr hilfreich herausstellte.

#### CommunicationTest

Zusätzlich wurden Tests implementiert, die einen Teil der Software initialisieren, und dabei teilweise Benutzerinteraktion erfordern. Die Tests für die Zeitgeber und die Kommunikation mit dem Parallelport wurden bereits in den vorausgegangenen Kapiteln besprochen.

Zum Test der *HAL* wurde mit **CommunicationTest** eine Test-*Suite* erstellt, die zum Testen der Kommunikation mit dem echten Flipperautomaten dient. Hierfür wird die komplette *HAL*-Schicht initialisiert und gestartet. Hier zeigt sich, ob die Kommunikation mit dem Flipperautomaten stabil aufrechterhalten werden kann,

8 Siehe die Erläuterungen zur *Dependency Injection* im Abschnitt zu den verwendeten *Design Pattern*.

oder ob sie aufgrund von verletzten Zeitbedingungen abbricht. Des weiteren stehen drei Testprogramme zur Verfügung um das Ansteuern der einzelnen Komponenten zu testen:

- Der Lampentest aktiviert und deaktiviert nach und nach alle Lampen.
- Der Spulentest aktiviert abwechselnd die beiden *Saucer*-Spulen. Zusätzliche aktivert, hält und deaktivert er die beiden Finger des Flippers.
- Der Schaltertest gibt alle Schalterereignisse auf der Konsole aus.

Um ein *Debugging* der Parallelport-Kommunikation zu ermöglichen, wurde zudem ein **ParallelPortSniffer** erstellt. Bei diesem handelt es sich um einen *Wrapper* für das **ParallelPort**-Modul. Er verwendet *Proxyquire* um die *Node.JS*-Erweiterung *parport2* im **ParallelPort** durch eine Instanz zu ersetzen, die sämtliche Übertragungen über den Parallelport auf unterster Ebene abgreift und aufzeichnet.

Im Rahmen des Tests wurden die Zeiten gemessen, in denen der Lampentreiber die einzelnen Zeilen der Lampenmatrix auf den Parallelport schreibt. Als Intervallzeit wurde hier 0,5 ms gewählt. Dies ist das Intervall, das im Quellcode von Herrn Kratzer genutzt wird. Die im Rahmen seiner Arbeit [3, Kapitel 2] und auch in dieser Arbeit bisher dokumentierten 1,5 ms führen zu einer für das bloße Auge identischen Ansteuerung. Für den Test wurde jedoch trotzdem die schnellere, und damit fordernde, Ansteuerungsfrequenz gewählt.

```
[ 0, 567237 ]
[ 0, 336390 ]
[ 0, 565821 ]
[ 0, 335998 ]
```

**Ausgabe 5:** Aufrufintervall des Lampentreibers in Sekunden und Nanosekunden.

Trotz Schwankungen kann die Zielzeit von 0,5 ms ausreichend eingehalten werden. Beim Auslesen der Schalter ergibt sich ein noch besseres Bild. Hier besteht keine Gefahr, die durch die Flipperplatine vorgegeben Auslesefrequenz von 1,5 ms zu verletzen:

```
[ 0, 943193 ]
[ 0, 945775 ]
[ 0, 941779 ]
[ 0, 945161 ]
```

**Ausgabe 6:** Aufrufintervall des Schalterscanners in Sekunden und Nanosekunden.

**StateMachineTest**

Dieser Test dient zum manuellen Untersuchen des in der Konfiguration hinterlegten Zustandsautomaten. Für die komplette *HAL* wird ein *Mock* erzeugt, der die notwendigen Funktionen nachbildet: Das **FlipperModel** kann wie gewohnt gelesen werden. Ebenso können Ereignisse wie im Original registriert werden. Dem Benutzer stehen Konsolen-Kommandos zur Verfügung, mit denen er den Status der einzelne Komponenten manipulieren kann. Geänderte Schalter lösen hierbei Ereignisse aus. Die durch die Spielregeln ausgelösten Zugriffe auf die Treiber der *HAL* werden auf der Konsole ausgegeben. Zusätzlich wird regelmäßig der aktuelle Zustand des Zustandsautomaten ausgelesen und bei Änderung ausgegeben. Dies ermöglicht es die definierten Regeln in ihrer Gesamtheit zu überprüfen.

**CommunicationTest** und **StateMachineTest** decken zusammen den kompletten Funktionsumfang der Steuersoftware ab. Während der erstgenannte Test mit dem Flipperautomaten kommuniziert und dabei die komplette *HAL* unter Verwendung ihrer API einsetzt, ersetzt der zweite Test eben alles unter der API der Hardwareschicht durch *Mock*-Objekte und testet so die Spiellogik darüber.

# Kapitel 8

# Zusammenfassung

## 8.1 Vorteile und Nachteile

Durch die übersichtlich gehaltene Zwei-Schicht-Architektur wurde sich bei Gestaltung der Software auf das Wesentliche beschränkt. Die Bauteile sind abstrahiert, die Spiellogik arbeitet rein mit Namensstrings, die genaue Ansteuerung ist für sie völlig transparent. Durch diese Struktur ist die Spiellogik komplett von der Hardware entkoppelt. Die Abhängigkeiten sind durch die schlanke API zwischen den beiden Schichten gering. Der Schicht der Spiellogik muss lediglich das Modell des Flipperautomaten und die Treiberfassade bekannt sein. Beide Schichten könnten einfach durch eine andere Implementierung ausgetauscht werden.

Kurze Methoden sorgen für eine feingranulare Testbarkeit durch *Unit-Tests*. Durch Einsatz von *Injection* sowohl für Objekte wie auch für Konfigurationsdateien können *Mock*-Objekte weitgehend einfach im Rahmen von Tests genutzt werden.

*JavaScript* als Sprache ist durch ihren Typenlosigkeit und einen für Programmierer ungewohnten, aber für weniger versierte Anwender intuitivem, *Closure*-Ansatz leicht erlernbar. Dies konnte für eine sehr flexible und dennoch verständliche Konfiguration genutzt werden. Durch Verwendung von *JavaScript* kann beim Erstellen der Regeln auf Unterstützung einer IDE zurückgegriffen werden. Bauteilnamen lassen sich beispielsweise über die Mechanismen zur Autovervollständigung auswählen. Die Beispiel-Konfiguration kann übernommen und erweitert werden, auch ohne Kenntnisse der Steuersoftware. Zudem ist die Konfiguration in ihrem Aufbau aus Funktionen ohne Zusatzwerkzeuge testbar.

Die Nachteile der Implementierung ergeben sich ebenfalls aus der gewählten Entwicklungssprache. Wie schon in der Einleitung angedeutet, ist der Einsatz unter Zeitbedingungen kein typisches Anwendungsfeld für JavaScript. Dies führt dazu, dass beispielsweise die Zeitgeber nicht für derartige Anforderungen vorbereitet sind. Trotz

der hohen Ausführungsgeschwindigkeit der *V8*-Umgebung dürfte eine Implementierung in einer kompilierten Sprache effizienter sein. Auch ist die gewählte Lösung in keinster Weise zur Erfüllung von Echtzeitbedingungen geeignet.

Dennoch konnte mit dieser Arbeit gezeigt werden, dass trotz einiger für diesen Anwendungsfall potentiell ungeeigneten Eigenschaften dennoch eine lauffähige Implementierung erstellt werden konnte.

Für den praktischen Einsatz könnte sich folgender Mittelweg empfehlen, um Vorteile mit Nachteilen zu kombinieren: Die Konfiguration verbleibt in JavaScript und erhält so die genannten Vorteile. Für die Funktionsgruppen darunter wird sukzessiv eine Implementierung beispielsweise in *C++* in Erwägung gezogen. Dies ist auch der Ansatz den die *V8* selbst wählt: Sie ist teilweise in *C++*, aber auch in *JavaScript* implementiert. Eine derartige Aufteilung erhöht aber den Aufwand für den Entwickler.

## 8.2 Fazit

JavaScript entwickelt sich trotz aller Kritik [18, Anhang A & Anhang B] zu einer immer mächtigeren Sprache, was den immer weiter optimierten Ausführungsumgebungen, aber auch den darauf aufsetzenden Technologien wie *Node.JS* zu verdanken ist. Das diese mittlerweile auch abseits klassischer Einsatzfelder zu brauchbaren Ergebnissen führen, konnte im Rahmen dieser Abschlussarbeit bestätigt werden. Für alle Problemfelder konnte eine Lösung gefunden werden, so dass eine lauffähige Softwaresteuerung aus dieser Arbeit hervorgeht.

Darüber hinaus ist davon auszugehen, dass die Entwicklung der Ausführungsumgebung stets weiter voran schreitet, wie am Beispiel der *Garbage Collection* zu sehen ist [41]. Teils wurden für die Arbeit essentielle Methoden wie **hrtime** erst wenige Monate vor Beginn dieser Arbeit implementiert [43]. Auch an dieser Stelle ist zu sehen, dass sich die Technologie gerade in der Entwicklung befindet und für die Zukunft noch einiges erwartet werden kann.

Der Eingangs erwähnte Trend zu aufwendigen Webapplikation wird diese Entwicklung weiter befeuern. Und was die Reaktionszeit im Webbrowser verbessert, wirkt sich auch positiv auf noch höhere zeitliche Anforderungen aus.

**Abbildung 18:** Flipperautomat unter Ansteuerung der Lampen mit der in dieser Arbeit erstellten *JavaScript*-Software.

## 8.3 Ausblick

Zunächst kann die Implementierung genutzt werden, um den Regelsatz der originalen Steuersoftware umzusetzen. Neben der Deklaration der Regeln wäre auch ein Erstellen von *Unit-Tests* Teil dieser Aufgabe.

Aber auch an der Software selbst kann noch angesetzt werden: Anstatt für **LampDriver** und **SwitchScanner** jeweils eine eigene Instanz der **IntervalRunners** zu erzeugen, könnte eine globale Instanz genutzt werden, bei der sich Interessenten registrieren können. Dies erhöht die Komplexität der Implementierung aber deutlich, da die einzelnen Module unterschiedliche Intervalle benötigen. Auch ist denkbar, direkt am *Node.JS*-Quellcode anzusetzen und die Implementierung von **setTimeout** zu optimieren. Da diese aber auf die Ereigniswarteschlange aufsetzt, kann der Umfang der nötigen Änderungen beträchtlich sein.

Als Gesamtoptimierung könnte man den Quellcode mit *JavaScript-Compilern* verarbeiten. Dieser erzeugen neuen *JavaScript*-Quellcode und lassen zahlreiche Optimierungen einfließen. Hier ist zu untersuchen, ob sich deutliche Verbesserungen zur bereits stark optimierenden V8-Ausführungsumgebung ergeben. Ein Beispiel für solch einen optimierenden *Compiler* wäre *Googles Closure Compiler* [58].

## Literaturverzeichnis

Hinweis: Aufgrund des Themenfeldes dieser Arbeit, welche sich mit jungen Technologien im Fachbereich Informatik beschäftigt, handelt es sich bei vielen der angegebenen Quellen um Internetseiten. Für Seiten, bei denen die Autoren nicht oder nicht eindeutig identifizierbar waren, wird die Entwicklergruppe oder das Unternehmen hinter der jeweiligen Internetpräsenz genannt.


[1] Joyent, Inc: *node.js*. Webseite, abgerufen Januar 2013, `http://nodejs.org/docs/v0.8.20/`.

[2] Kratzer, K.: *Design und Implementierung einer Echtzeitsteuerung mit Hilfe der Real-Time Specification for Java*. Bachelorarbeit, Hochschule für angewandte Wissenschaften München, 2010.

[3] RTSJ Technical Interpretation Committee: *RTSJ Version 1.0.2*. Webseite, abgerufen Januar 2013, `http://www.rtsj.org/specjavadoc/book_index.html`.

[4] Joint Technical Committee ISO/IEC JTC 1: *Information technology — Programming languages, their environments and system software interfaces — ECMAScript language specification*. Technische Spezifikation, 2011.

[5] o. A.: *Repairing Williams/Bally Pinball 2000*. Webseite, nicht mehr verfügbar, `www.pinrepair.com/pin2000/makep2k.htm`.

[6] Google Inc.: *Chrome V8 — Google Developers*. Webseite, abgerufen Januar 2013, `https://developers.google.com/v8/`.

[7] Reitter, T.: *Let's Push Things Forward - Answering the question: "How do I develop an app for GNOME?"*. Webseite, abgerufen März 2013, `http://treitter.livejournal.com/14871.html`.

[8] Netscape Communications Corporation: *Netscape and Sun Announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet*. Webseite, abgerufen Januar 2011, `http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html`.

[9] Microsoft Corporation: *Microsoft Internet Explorer 3.0 Beta Now Available*. Webseite, abgerufen Januar 2013, `http://www.microsoft.com/en-us/news/press/1996/may96/ie3btapr.aspx`.

[10] Zakas, N. C.: *High Performance JavaScript*. O'Reilly Media Inc., 1. Auflage, 2010.

[11] Google Inc.: *Design Elements - Chrome V8 — Google Developers*. Webseite, abgerufen Januar 2013, https://developers.google.com/v8/design/.

[12] Joyent Inc.: *Addons Node.js v0.8.20 Manual & Documentation*. Webseite, abgerufen Januar 2013, http://nodejs.org/docs/v0.8.20/api/addons.html.

[13] Google Inc.: *Embedder's Guide - Chrome V8 — Google Developers*. Webseite, abgerufen Januar 2013, https://developers.google.com/v8/embed.

[14] Hughes-Croucher T. und Wilson, M.: *Node: Up and Running*. O'Reilly Media Inc., 1. Auflage, 2012.

[15] Rabinovich, A.: *YUI Theater — Ryan Dahl: "Introduction to NodeJS" (58 min.)*. Webseite, abgerufen Januar 2013, http://www.yuiblog.com/blog/2010/05/20/video-dahl/.

[16] Dahl, R.: *node.js*. Webseite, abgerufen Januar 2013, http://joyeur.files.wordpress.com/2010/11/qcon.pdf.

[17] Joyent Inc.: *Node.js v0.8.20 Manual & Documentation*. Webseite, abgerufen Januar 2013, http://nodejs.org/docs/v0.8.20/api/.

[18] Crockford, D.: *JavaScript: The Good Parts*. O'Reilly Media Inc., 1. Auflage, 2008.

[19] Ubl, M.: *cramforce/node-worker · GitHub*. Webseite, abgerufen Januar 2013, https://github.com/cramforce/node-worker.

[20] Flanagan, D.: *JavaScript: The Definitive Guide*. 'Reilly Media Inc., 6. Auflage, 2011.

[21] Joyent Inc.: *File System Node.js v0.8.20 Manual & Documentation*. Webseite, abgerufen Januar 2013, http://nodejs.org/docs/v0.8.20/api/fs.html.

[22] Joyent Inc.: *Stream Node.js v0.8.20 Manual & Documentation*. Webseite, abgerufen Januar 2013, http://nodejs.org/docs/v0.8.20/api/stream.html.

[23] Teixeira, P.: *Professional Node.js: Building Javascript Based Scalable Software*. Wrox, 1. Auflage, 2012.

[24] Beal, S.: *v8-juice - C++ toolkit for extending the Google v8 JavaScript engine - Google Project Hosting*. Webseite, abgerufen Januar 2013, http://code.google.com/p/v8-juice/.

[25] Mendez, X.: *jmendeth/v8u · GitHub*. Webseite, abgerufen Januar 2013, https://github.com/jmendeth/v8u.

[26] Microprocessor Standards Committee: *1284-1994 - IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers*. Technische Spezifikation, 1994.

[27] Corbet, J., Rubini, A. und Kroah-Hartman, G.: *Linux Device Drivers*. O'Reilly Media Inc., 3. Auflage, 2005.

[28] Williams Electronics Games Inc.: *Home - WMS.COM*. Webseite, abgerufen Januar 2013, http://www.wms.com/.

[29] Williams Electronics Games Inc.: *Pinball 2000, Star Wars Episode I, Operations Manual*. Betriebsanleitung, 1999.

[30] Venkateswaran, S.: *Essential Linux Device Drivers*. Prentice Hall, 1. Auflage, 2008.

[31] Waugh, T.: *The Linux 2.4 Parallel Port Subsystem*. Freies Buch unter GNU Free Documentation License veröffentlicht, 1. Auflage, 1999.

[32] Zamanis, P.: *Steuerung eines Flipperautomaten mittels Java Real-Time und Anbindung von C-Funktionen über das Java Native Interface*. Bachelorarbeit, Hochschule für angewandte Wissenschaften München, 2009.

[33] Bergmann, A.: *linux/kernel/git/torvalds/linux.git - Linux kernel source tree*. Webseite, abgerufen Januar 2013, `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4ba8216cd90560bc402f52076f64d8546e8aefcb`.

[34] Mendez, X.: *parport*. Webseite, abgerufen Januar 2013, `https://npmjs.org/package/parport`.

[35] Neder, M.: *parallel-port - Parallel port portable API for C++ and Java - Google Project Hosting*. Webseite, abgerufen Januar 2013, `http://code.google.com/p/parallel-port/`.

[36] Neder, M.: *ParallelPortLinux.cpp - parallel-port - Parallel port portable API for C++ and Java - Google Project Hosting*. Webseite, abgerufen Januar 2013, `https://code.google.com/p/parallel-port/source/browse/trunk/parallel-port/src/ParallelPortLinux.cpp?r=3`.

[37] Walraven, K. und Buiting, J.: *LPT exerciser, gateway to the PC parallel port*. Artikel erschienen in Elektor (Englische Ausgabe), Ausgabe 284, 2000.

[38] CadSoft Computer GmbH: *CadSoft EAGLE PCB Design Software - Circuit Board Design Software*. Webseite, abgerufen März 2013, `http://www.cadsoftusa.com/eagle-pcb-design-software/`.

[39] Generate Your Projects Developers: *gyp - Generate Your Projects - Google Project Hosting*. Webseite, abgerufen Januar 2013, `http://code.google.com/p/gyp/`.

[40] Corry, E. und o. A.: *Node.js and V8 garbage collection - Stack Overflow*. Webseite, abgerufen Januar 2013, `http://stackoverflow.com/questions/5603011/node-js-and-v8-garbage-collection`.

[41] Egorov, V. und Corry, E.: *Chromium Blog: A game changer for interactive performance.*. Webseite, abgerufen Januar 2013, `http://blog.chromium.org/2011/11/game-changer-for-interactive.html`.

[42] Mozilla Developer Network: *window.setTimeout - Document Object Model (DOM) | MDN*. Webseite, abgerufen Januar 2013, `https://developer.mozilla.org/en/docs/DOM/window.setTimeout#Minimum_delay_and_timeout_nesting`.

[43] Rajlich, N.: *process: add `process.hrtime()` · 07c886f · joyent/node · GitHub*. Webseite, abgerufen Januar 2013, `https://github.com/joyent/node/commit/07c886f94446daeb64aaab34904b319992895da4`.

[44] Joyent Inc.: *Events Node.js v0.8.20 Manual & Documentation*. Webseite, abgerufen Januar 2013, `http://nodejs.org/docs/v0.8.20/api/events.html`.

[45] Gamma, E., Helm, R., Johnson, R. und Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1. Auflage, 1994.

[46] Freeman, E., Robson, E., Bates, B. und Sierra K.: *Head First Design Patterns*. O'Reilly Media Inc., . Auflage, 2004.

[47] Prasanna, D. R.: *Dependency Injection*. Manning Publications, 1. Auflage, 2009.

[48] Fowler, M.: *Inversion of Control Containers and the Dependency Injection pattern*. Webseite, abgerufen Januar 2013, `http://martinfowler.com/articles/injection.html`.

[49] Trostler, M. E.: *Testable JavaScript*. O'Reilly Media Inc., 1. Auflage, 2013.

[50] Google Inc.: *v8natives.js - v8 - V8 JavaScript Engine - Google Project Hosting*. Webseite, abgerufen März 2013, `http://code.google.com/p/v8/source/browse/trunk/src/v8natives.js?r=13979`.

[51] Zakas, N. C.: *Maintainable JavaScript*. O'Reilly Media Inc., 1. Auflage, 2012.

[52] Crockford, D.: *JSLint,The JavaScript Code Quality Tool*. Webseite, abgerufen Januar 2013, `http://www.jslint.com/`.

[53] Stefanov, S.: *JavaScript Patterns*. O'Reilly Media Inc., 1. Auflage, 2010.

[54] Meyer, B.: *Object Oriented Software Construction*. Prentice Hall, 2. Auflage, 2000.

[55] Joyent Inc.: *Modules Node.js v0.8.20 Manual & Documentation*. Webseite, abgerufen März 2013, `http://nodejs.org/docs/v0.8.20/api/modules.html`.

[56] Reinstein, M.: *caolan/nodeunit · GitHub*. Webseite, abgerufen Januar 2013, `https://github.com/caolan/nodeunit`.

[57] Lorenz, T.: *thlorenz/proxyquire · GitHub*. Webseite, abgerufen Januar 2013, `https://github.com/thlorenz/proxyquire`.

[58] Google Inc.: *Closure Tools — Google Developers*. Webseite, abgerufen März 2013, `https://developers.google.com/closure/compiler/`.