

Hochschule Pforzheim
Fakultät für Technik
Studiengang Technische Informatik

Bachelorarbeit zur Erlangung des akademischen Grades

BACHELOR OF ENGINEERING

**Konzept und Implementierung einer SPS-
Kleinststeuerung auf einem Raspberry Pi mit
graphischer Programmierung und
IFTTT-Funktionalität**

Julian Wiche
306675

Datum: 19. Mai 2019
Erstprüfer: Prof. Dr. Karlheinz Blankenbach
Zweitprüfer: Prof. Dr.-Ing. Thomas Greiner

Zusammenfassung

In dieser Bachelor Thesis soll eine speicherprogrammierbare Steuerung mittels eines Raspberry Pi nachempfunden werden. Der Fokus liegt dabei darauf, eine möglichst günstige Lösung zu schaffen, um auch Einsteigern die Möglichkeit zu bieten einfache Steuerungsprojekte zu realisieren. Das Steuerungsprogramm hierfür, soll mittels Zeichnung intuitiv in einer Weboberfläche erstellt werden können.

Schlagwörter: SPS, Kleinststeuerung, Raspberry Pi

Abstract

Concept and implementation of a PLC- minicontroller using a Rasperry Pi featuring a graphical programming interface and IFTTT-functionality

Goal of this Bachelor-Thesis is, to adapt a programmable logic controller using a Raspberry Pi. The main focus is to achieve a affordable solution, enabling yet beginners to implement trivial controlling projects. The controlling programm for which, shall be creatable intuitively using a drawing tool within a web-gui.

Keywords: PLC, minicontroller, Rasperry Pi

Eidesstattliche Erklärung

Ich, Julian Wiche, Matrikel-Nr. 306675, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

Konzept und Implementierung einer SPS- Kleinsteuerung auf einem Raspberry Pi mit graphischer Programmierung und IFTTT-Funktionalität

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Pforzheim, den 19. Mai 2019

JULIAN WICHE

Inhaltsverzeichnis

Abbildungsverzeichnis	3
Tabellenverzeichnis	3
Quellcodeverzeichnis	3
1 Einleitung	4
1.1 Ziel der Arbeit	4
1.2 Verwandte Arbeiten	5
1.3 Aufbau der Arbeit	5
2 Grundlagen	6
2.1 Prinzip Speicherprogrammierbare-Steuerung	6
2.2 Ausgangssituation	7
2.2.1 Bedienoberfläche	7
2.2.2 Backend	9
2.2.3 Hardware	9
2.3 Versionsverwaltung	9
2.4 Entwicklungsumgebung	10
2.5 C++ Library LibPiFace	10
2.6 WebSocket	11
3 Umsetzung	13
3.1 Parsen von Logikausdrücken	13
3.2 Automatische Prüfung von Abhängigkeiten	14
3.3 Benamungsschema	14
3.4 Klassenstruktur und Kanäle	14
3.4.1 Hardware Kanäle	19
3.4.2 Merker Bausteine	19
3.4.3 Timer Bausteine	19
3.4.4 Virtuelle Kanäle	21
3.5 Laden der Programmlogik aus Datei	21
3.5.1 Erneutes Laden der Logik zur Laufzeit	22
3.6 Hauptschleife und Iterationslogik	22
3.7 Logik Prozessor	23
3.8 WebSocket Server	24
3.8.1 Basis	24
3.8.2 Erweiterung	25
3.8.3 Zugriffsbeschränkung	26

3.9	Benutzeroberfläche	26
3.9.1	Übersicht Komponenten	28
3.9.2	Apache Webserver	28
3.9.3	JavaScript Anwendung	29
3.9.4	Logik Editor CircuitVerse	29
3.9.5	PHP Adapter Programme	35
3.9.6	IFTTT	36
4	Auswertung	37
5	Ausblick	38
6	Literatur	39
7	Anhang	40

Abbildungsverzeichnis

2.1	Blockdiagramm SPS	7
2.2	Programm einer Easy Kleinsteuerng	8
3.1	Klassendiagramm Aggregation der Klassen	16
3.2	Vererbungshierarchie der Basisklasse IOChannel	17
3.3	Vererbungshierarchie der Basisklasse ChannelEntity	18
3.4	Zugriff auf die Weboberfläche	27
3.5	Networking Übersicht	28
3.6	Screenshot einer Logikschaltung in CircuitVerse	29
3.7	Darstellung der JSON Datenstruktur	30
3.8	Logikschaltung mit gekennzeichneten Knoten	31
3.9	Lookup Array	32
3.10	Programm Ablaufplan	33
3.11	Umgestaltetes CircuitVerse	34
3.12	Dialog mit Exportergebnis	35
7.1	Inhalt der beigelegten CD	40

Tabellenverzeichnis

7.1	Liste der eingesetzten Software	40
-----	---	----

Quellcodeverzeichnis

3.1	Initialisieren der Kanäle und Entitäten	15
3.2	Zugriff auf Kanal und Entität exemplarisch	15
3.3	Beispiel der Timer Konfigurationsdatei	21
3.4	Beispiel der Programmlogik Datei	22
3.5	Logik Engine	24
3.6	JSON Status Update von Privaten Entitäten	26

1 Einleitung

Speicherprogrammierbare Steuerungen oder kurz SPS tauchen überall dort auf, wo große elektrische Maschinen eingesetzt werden. Dies ist vor allem in der Industrie der Fall. Ihr kleiner Bruder ist die Kleinststeuerung. Sie bietet die selben Kernfunktionen, hat jedoch eine deutlich kleinere Anzahl an Ein- und Ausgängen. Sie werden häufig von Elektroinstallateuren eingesetzt, wenn eine klassische Verbindungsprogrammierte Steuerung zum Beispiel durch Drahtbruch oder defekte Spulen in eingesetzten Relais nicht mehr korrekt funktionieren. Der Hersteller Eaton hat mit seinem Produkt »Easy« genau diese Zielgruppe im Blick. Die Programmierung erfolgt hier, als würde man klassische Schütz-Kontakte in Reihe schalten. Die Einstiegsgeräte sind relativ preiswert, doch kauft man sich in eine proprietäre Produktwelt ein, welche aufgrund von inkompatiblen Bauteilen und Bussystemen schwer wieder zu verlassen ist. So gestaltet sich die Erweiterung einer bestehenden Steuerung um die Möglichkeit einer Fernabfrage übers Internet als nahezu unmöglich, oder setzt den Austausch der kompletten Steuerung voraus. Dabei sind Ein- und Ausgänge doch eigentlich das selbe wie an jedem Raspberry Pi vorhandene GPIOs. Auf Basis dieser Überlegung und günstigen Preisen hierfür, entstand die Idee eine Lösung mittels Raspberry Pi zu erarbeiten.

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist es dabei vor allem eine möglichst günstige Möglichkeit zu schaffen um eine Steuerung zu realisieren, welche intuitiv programmiert werden kann und die Grundsätzliche Funktion der vorher eingesetzten Easy Steuerung um Funktionen zur Fernabfrage übers Internet und weitere Funktionen erweitert. Dabei soll das Projekt mittels Git versioniert werden um es anderen Entwicklern auf GitHub als Open-Source Software zur Verfügung zu stellen. Dabei wird das Projekt als MIT Lizenziert, was eine Modifikation sowie private und gewerbliche Nutzung und Verbreitung ausdrücklich gestattet.

1.2 Verwandte Arbeiten

Weitere Projekte mit denen ähnliches möglich ist, sind hierbei das Kommerzielle Projekt Codesys *REF*. Auch zu nennen ist das Projekt Open-PLC *REF*

1.3 Aufbau der Arbeit

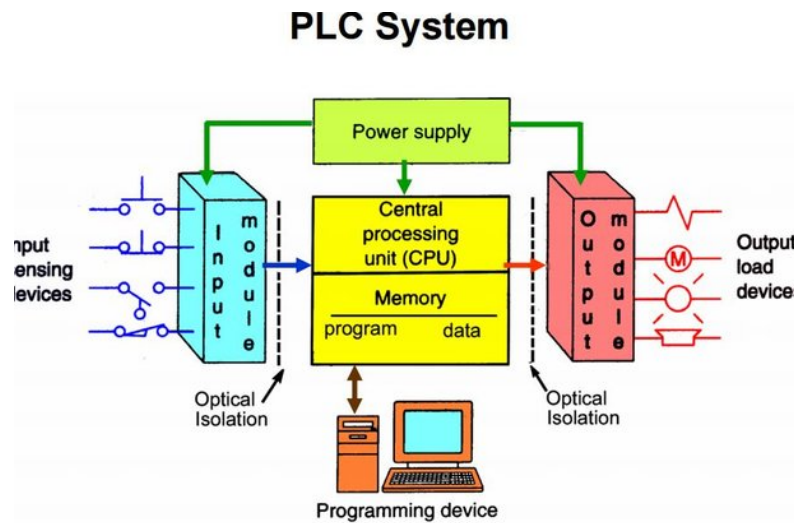
(TODO: Aufbau der Arbeit fertigstellen)

2 Grundlagen

(TODO: Struktur Grundlagen einfügen)

2.1 Prinzip Speicherprogrammierbare-Steuerung

Die Grundsätzliche Funktion einer Speicherprogrammierbaren Steuerung oder SPS ist, die Ermittlung der Ausgangswerte bzw. Schalterstellung durch eine logische Verknüpfung der Eingangswerte.[1] Im einfachsten Beispiel, könnte ein an einen Eingang angeschlossener Schalter als Sensor dienen. Als Aktor könnte eine Leuchte dienen. Der Benutzer der Steuerung muss nun durch eine Logik für jeden Ausgang festlegen, in welchen Fällen dieser Ausgang aktiv sein soll. Doch wieso schließt man dann nicht einfach die Leuchte direkt an den Schalter an? Dies wäre bei einer einfachen Lampensteuerung sicherlich zu bevorzugen, jedoch handelt es sich bei den Szenarien die mit einer solchen Steuerung realisiert werden für Gewöhnlich um deutlich komplexere Verschaltungen. Bei der klassischen Installation für eine Torsteuerung Beispielsweise, wären mehrere Elektromechanische Relais, auch Schütze genannt nötig. Zudem bedürfte ein automatisches schließen des Tores ein Zeitrelais. Der Verdrahtungsaufwand und Platzbedarf wären relativ hoch. Führt man stattdessen jedoch alle benötigten Sensoren auf eine Speicherprogrammierbare Steuerung wird der Verdrahtungsaufwand erheblich reduziert, was zu einer höheren Übersichtlichkeit führt und weniger Potential für Fehler bietet. Auch zieht eine Änderung im Logischen verhalten der Steuerung dann für gewöhnlich keinerlei Verdrahtungsänderungen mehr nach sich. Zuletzt sind auch die Kosten für Speicherprogrammierbare Steuerungen inzwischen auf einem Niveau, was klassische Steuerungen schnell unwirtschaftlich macht.



Quelle: <https://instrumentationforum.com/t/architecture-of-plc/7059>

Abbildung 2.1: Blockdiagramm SPS

2.2 Ausgangssituation

Als Vorbild für dieses Projekt dient die Kleinststeuerung Easy vom Hersteller Eaton. Das Einstiegsmodell bietet hier acht Eingänge und vier Ausgänge. Das Logikprogramm, welches die Eingänge der Steuerung logisch mit den Ausgängen verbindet wird hier, auf einem kleinen Display, direkt am Gerät erstellt. Dabei stehen neben den physikalischen Ein- und Ausgängen auch Zeitfunktionen oder Zählerbausteine zur Verfügung. *Erweiterbar* Im Programmiermodus wird links einen Pluspol und rechts einen Minuspol Symbolisiert. Der anzusteuernde Ausgang, welcher obligatorisch ist, steht dabei stets ganz rechts. Der Strompfad kann nunmehr bis zum Pluspol durch gezeichnet werden, oder aber durch Sensoren unterbrochen und verzweigt werden. Aus diesem Schaltplan werden dann die booleschen Gleichungen gewonnen, die die Steuerung im Betrieb durchläuft um die Werte der Ausgänge zu bestimmen.

2.2.1 Bedienoberfläche

Eine ähnliche Vorgehensweise ist auch in diesem Projekt geplant. Da der Raspberry Pi Netzwerkfähig ist, wurde jedoch anstatt einem Display am Gerät eine Bedien-

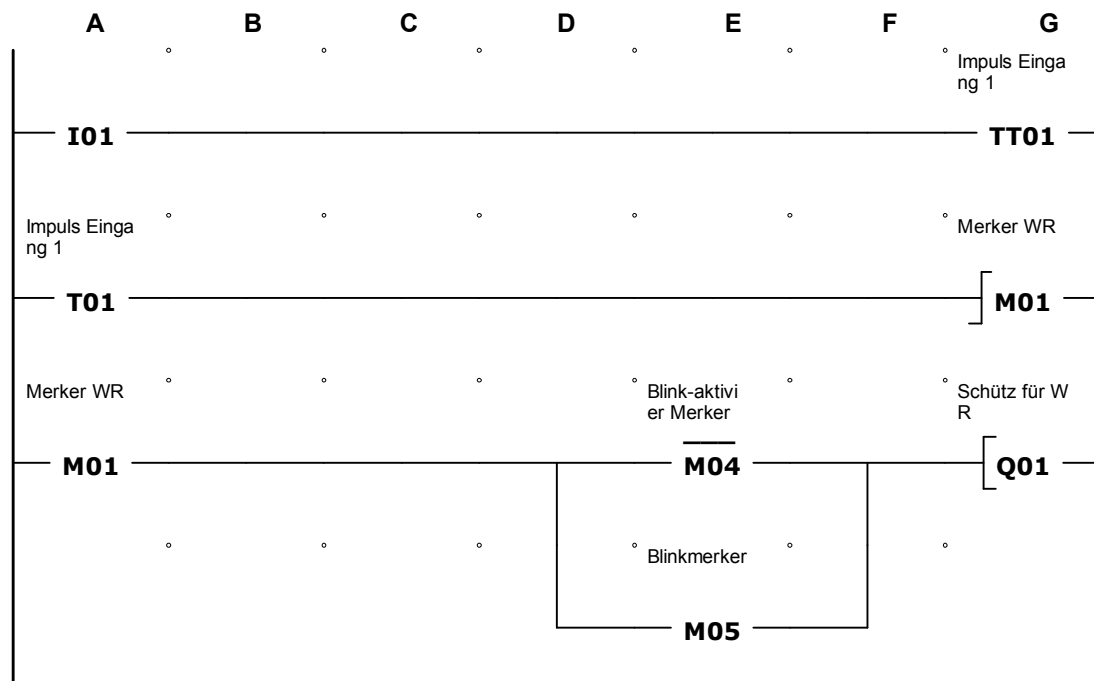


Abbildung 2.2: Programm einer Easy Kleinsteuerng

oberfläche gewählt, welche im Internetbrowser bedienbar ist. Als Basis für die Programmieroberfläche, wurde das Projekt CircuitVerse ^{*REF*} herangezogen. Hierbei handelt es sich um einen Logiksimulator, in welchem komplexe Logikschaltungen durch Drag&Drop erstellt werden können. Das Quelloffene Projekt ist auf GitHub ^{*REF*} verfügbar und dank der MIT ^{*REF*} Lizenz zur Erweiterung und Modifikation freigegeben. Dabei musste das Projekt vor allem durch eine Funktion ergänzt werden, um die Erstellte Logik in einem Format zu Exportieren, welche vom Backend verstanden wird. Weiterhin musste die zur Verfügung stehenden Ein- und Ausgänge dahingehend modifiziert werden, dass nur Schaltungen erstellt werden können, die auch vom Backend verstanden werden. Im Vorbild kann der Schaltplan auch Laufzeitinformationen wiedergeben. So wird ein (symbolisch) unter Spannung stehender Zweig als breite Linie dargestellt, während unbestromte zweige schmal gezeichnet werden. Diese Laufzeitinformationen sollen in dieser Bachelorarbeit ebenfalls dargestellt werden. Dazu wird für jeden Ein und Ausgang eine Grafik eingeblendet, welche erkennen lässt, ob der Eingang bzw. der Ausgang an oder aus

ist.

2.2.2 Backend

Als Schnittstelle zwischen Hardware und Bedienoberfläche wird eine Software eingesetzt, die das vom Benutzer erstellte Logikprogramm kontinuierlich durchläuft, und somit sicherstellt, dass eine Änderung an einem Eingang, der Ablauf eines Timers etc. die Werte der davon abhängigen Ausgänge entsprechend verändert. Wie dies im Vorbild der Easy Kleinststeuerung gelöst wird, bestand kein Einblick.

2.2.3 Hardware

Wie schon vorab beschrieben, bildet ein Raspberry Pi die Grundlage für dieses Projekt. Dieser bietet von Haus aus einige GPIOs, welche dazu verwendet werden können um Sensoren abzufragen oder um Aktoren anzusteuern. Jedoch fiel die Entscheidung darauf, eine Erweiterungskarte (HAT) zu diesem Zweck einzusetzen. Dies dient erstmals zum Schutz des Raspberry Pi, zudem bietet das eingesetzte Board jedoch Leuchtdioden an den Ausgängen, sowie Taster an den Eingängen, was das Testen erheblich vereinfacht. Da der Anschluss per SPI erfolgt, können theoretisch mehrere solcher Boards parallel betrieben werden.

2.3 Versionsverwaltung

Da im bisherigen Studium lediglich auf SVN als Versionsverwaltung tiefer eingegangen wurde wobei eine Versionsverwaltung für ein Projekt dieser Größe als notwendig erachtet wurde, fiel der Gedanke auf GIT. Git ist eine dezentrale Versionsverwaltung, die Notwendigkeit für einen Versionierungsserver entfällt hierdurch. Jedoch ist es in Git möglich ein- oder mehrere sogenannte Remotes hinzuzufügen. Das sind entfernte Git-Repositorys die mit dem lokalen Repository synchronisiert werden können. In der Praxis wird Git häufig eingesetzt, weshalb sich diese Bachelorarbeit als Einarbeitung anbot. Zunächst wurde ein lokales Git Repository erstellt, welches in Folge mit einem Remote-Repository auf GitHub verbunden wurde. Angedacht war ein Development-Branch und jeweils ein Feature Branch

welcher nach Vollendung des entsprechenden Features in den Development Branch zurückgeführt werden soll. Darüber hinaus, soll ein Tagging erfolgen. Dabei soll für jeden Zeitbalken **WORD** im, in der vorhergehenden Projektplanung erstellten Gantt Diagramm, ein Tag erstellt werden.

2.4 Entwicklungsumgebung

Ein Problem dass dieses Projekt bietet ist, dass nicht direkt auf dem Zielsystem entwickelt wird. Das hat zur Folge, dass das Programm entweder auf dem Entwicklungssystem Crosskompiliert werden muss, oder die Quelldateien auf das Zielsystem kopiert werden müssen um sie dort zu kompilieren. Da das Aufsetzen eines Crosskompilers inklusive der dazu nötigen Toolchain als zu Aufwändig erachtet wurde, blieb nur der Weg das Projekt direkt auf dem Zielsystem zu übersetzen. Um sich die Arbeit zu erleichtern wurde nach einer Entwicklungsumgebung recherchiert, die ein Übersetzen über eine SSH Verbindung zulässt. NetBeans bietet diese an, dabei wird entweder das gesamte Projekt beim Übersetzen per SSH auf das Zielsystem übertragen und dort übersetzt, oder es kann ein Mountpunkt definiert werden, an dem eine Systemfreigabe eingehängt ist (z.B. Samba-Freigabe). Im Laufe des Projektes wurde eigens dafür ein Samba Server auf dem Zielsystem installiert. Von nun an konnte also in der Entwicklungsumgebung direkt auf den Dateien des Raspberry Pi gearbeitet werden, wobei per Tastendruck eine SSH Verbindung aufgebaut wird um auf dem Zielsystem »*make*« auszuführen. Dabei werden sämtliche Comiler Ausgaben in einem Fenster angezeigt. Auch Debugging ist auf diesem Wege möglich.

2.5 C++ Library LibPiFace

Wie im vorhergehenden Abschnitt **REF** beschrieben, bildet die Erweiterungskarte PiFace Digital 2 **REF** die Grundlage für diese Bachelorarbeit. Im Lieferumfang befindet sich eine in C geschriebene Library inklusive eines Lauffähigen Tests, welche ebenfalls auf GitHub unter <https://github.com/piface/libpifacedigital> zu finden ist. Diese Library stützt sich wiederum auf die Library <https://github.com/piface/libmcp23s17> welche den verbauten SoC über SPI anspricht. Im Laufe der Arbeiten

fiel jedoch auf, dass die C Library nicht alle benötigten Funktionen enthielt. Da der Quellcode vorlag, und die Lizenzierung Veränderungen am Quellcode zulässt, lag die Überlegung nahe die benötigten Funktionen direkt in der Library zu ergänzen anstatt sie im eigentlichen Projekt unterzubringen. Weiterhin schien es auch ein erstrebenswertes Lernziel zu sein, das Erstellen und Übersetzen von statisch bzw. dynamisch gelinkten Bibliotheken kennenzulernen. Zuletzt schien es schlichtweg die Sauberste Lösung zu sein. Zunächst wurde angenommen, dass sich auch mehrere Hardwaremodule per SPI mit einem einzelnen Raspberry Pi verbinden lassen. Dies ist technisch auch möglich, so bieten die eingesetzten Boards die Möglichkeit über einen Jumper eine Hardwareadresse einzustellen. Der Hersteller bot auch die passende Hardware an, um mehrere Boards mit einem Raspberry Pi zu verbinden. Jedoch wurden diese scheinbar mangels Nachfrage aus dem Sortiment genommen. Obwohl eine Bastellösung es immer noch ermöglichen würde, ist der Aufwand hierfür sehr hoch und scheint unwirtschaftlich. Leider wurde bis zu dieser Erkenntnis schon einiges an Energie darin investiert, mehrere Boards zu unterstützen. Dies ist auch der Grund wieso ein Objektorientierter Ansatz in C++ gewählt wurde - eine Instanz für jedes Hardwaremodul. Ein weiterer Grund war, dass die Anwendung ohne Caching nicht performant genug war. Das heißt die zeitliche Lücke zwischen einer Änderung an einem Eingang bis zu dessen Auswirkung am Ausgang war deutlich spürbar. Dafür wurden Methoden vorgesehen, um das Caching ein und auszuschalten - bei eingeschaltetem Caching verändern die Methoden um Bytes und Bits zu schreiben, lediglich den Wert einer Instanzvariable. Derzeit muss das Leeren des Caches explizit mittels Aufruf der Methode *flush()* erfolgen. Über ein Automatisches verfahren wurde nachgedacht, jedoch erwies sich der manuelle Aufruf als einfacher.

2.6 WebSocket

Damit die in 2.2.1 beschriebene Benutzeroberfläche Statusinformationen von der Steuerung erhalten kann, muss ein Kommunikationskanal zwischen dem Internetbrowser des Anwenders und der Steuerung geschaffen werden. Eine Kommunikation per HTTP ist dabei jedoch wenig flexibel, da dieses Protokoll zustandslos ist. Das bedeutet dass eine Verbindung nach bearbeitung einer Anfrage sofort wieder abge-

baut wird. Es ist als per se nicht möglich, dass der Server den Client beim Eintreffen neuer Daten Benachrichtigt. Eine Lösung hierzu wäre, die Anwendung im Browser so zu schreiben, dass sie den Server zyklisch anfragt. Dabei muss ein Kompromiss aus Serverbelastung, also kleinen Abfrageintervallen und einer Annehmbaren Verzögerung gefunden werden. Eine Verbesserung hierzu bringt **Long-polling**. Hierbei wird auf Clientseite auf die selbe Weise vorgegangen, der Server blockiert jedoch Anfragen bis kurz vor einen Timeout. Bei diesen Methoden ist die Serverbelastung relativ hoch und beinhalten eine gewisse Latenz, wodurch Anwendungen weniger schnell auf Änderungen reagieren können. WebSockets hingegen ermöglichen eine Bidirektionale Kommunikation zwischen Server und Client, wobei die Latenz minimal ist. Dies begründet sich unter anderem darin, dass eine WebSocket-Verbindung persistent ist. Das heißt sie wird einmalig aufgebaut und bleibt dann bestehen bis eine Seite die Verbindung beendet. Dadurch sind nur zum Aufbau der Verbindung Header nötig, was die zu übermittelnde Datenmenge drastisch reduziert. In der Praxis besteht ein HTTP Header welcher jeder Anfrage mitgegeben werden muss oft aus mehr als 100 Bytes. Eine WebSocket Anfrage benötigt einen ebenso großen Header, jedoch nur um die Verbindung aufzubauen. Ist die Verbindung hergestellt, so werden lediglich zwei Byte fällig um die Verbindung zu steuern. Das steigert nicht nur die Verbindungsgeschwindigkeit sondern entlastet gleichzeitig den Server.

3 Umsetzung

(TODO: Struktur Umsetzung einfügen)

3.1 Parsen von Logikausdrücken

Das Ziel dieses Projektes ist es, dass die Ausgänge der Steuerung in Abhängigkeit von Eingängen wie physikalischen Eingängen oder zum Beispiel Timer-bausteinen ein beziehungsweise ausgeschaltet werden. Dafür ist in einer Textdatei für jeden Ausgang eine Zeile vorgesehen. Eine Zeile beginnt hierbei mit dem zu definierenden Ausgang, also zum Beispiel Ho1, worauf ein Gleichheitszeichen zu folgen hat. Der gesamte Ausdruck hinter dem Gleichheitszeichen wird zur Laufzeit des Programms durchlaufen, wobei jedes vorkommende Paar von [und] durch eine Null oder eine Eins ersetzt. Innerhalb der Klammern, findet sich gleich genau wie bei dem Bezeichner vor dem Gleichheitszeichen, die jeweilige Bezeichnung der Abhängigkeit. Lautet die Zeile also etwa $Ho0 = [Hi0] \& [Hi1]$ so wird die Komplette erste Klammer durch den Wert von Hi0 ersetzt, während die zweite Klammer durch den Wert von Hi1 ersetzt wird. Daraus ergibt sich dann, vorausgesetzt Hi0 und Hi1 sind im Zustand »Ein«, $Ho0 = 1 \& 1$. Das für den Leser offensichtliche Ergebnis dieser Gleichung ist 1 oder »true«. Jedoch gestaltet sich eine programmatische Lösung des Problems als deutlich komplexer. Denn sobald mehr als drei Ausdrücke im Spiel sind, müssen wie bei klassischer Mathematik Rechenregeln befolgt werden. Punkt vor Strich sowie die Beachtung von Klammern. Dabei kann ein Ausdruck beliebig komplex sein. Eine Recherche nach Ansätzen führte zu Stackoverflow. (Siehe ¹ und ²) Dieser Ansatz löste genau das Problem und wurde somit in das Projekt übernommen.

¹Abr. 11.03.2019 <https://stackoverflow.com/questions/8706356/boolean-expression-grammar-parser-in-c/8707598#8707598>

²Abr. 11.03.2019 <http://coliru.stacked-crooked.com/a/c40382620fb75b75>

3.2 Automatische Prüfung von Abhängigkeiten

Ein Problem was sich mit dem Einbinden der Lösung zum Parsen der Logikausdrücke ergab, war die Abhängigkeit zu Boost. Auf dem Desktop Computer auf dem die Lösung getestet wurde, konnte das Projekt dank installiertem boost Paket ohne Probleme übersetzt werden. Da es sich bei dem Zielsystem jedoch um eine ARM Architektur handelt, musste der code dort Übersetzt werden. In den Paketquellen des dort installieren Raspian, ist jedoch eine ältere Version des Boost Pakets hinterlegt, was dazu führt dass Boost manuell heruntergeladen und gebaut werden muss. Obwohl sich dieses Problem im Laufe der Zeit durch aktualisierung des Paketes in den Paketquellen von Raspian von selbst lösen wird, muss dennoch geprüft werden ob die installierte Version den Ansprüchen genügt. Hierfür wurde ein Bash script erstellt, welches auch alle weiteren abhängigkeiten Prüft und gegebenenfalls installiert. Dazu zählen auch die in *REF* erwähnten Bibliotheken um die Hardware anzusprechen und wie in *REF* erwähnt der verwendete Compiler.

3.3 Benamungsschema

Nachdem die Auswertung beziehungsweise das Parsing der Logikausdrücke funktionierte, sollte auch die Auswertung der Bezeichner automatisiert werden. Ein Benamungsschema wurde dabei schon vorher erdacht. Es besteht aus einem führenden Großbuchstaben, gefolgt von einem Kleinbuchstaben und einer Zahl. Dabei wird der führende Buchstabe als Kanal bezeichnet, der zweite als Entität und die Ziffer als Pin-Nummer. So könnte zum Beispiel der Buchstabe »H« den Kanal Hardware beschreiben, welcher wiederum die Entitäten »i« für Input und »o« besitzt, welche jeweils ein Byte, also 8 Bits oder Pins haben.

3.4 Klassenstruktur und Kanäle

Im Programm werden die zuvor beschriebenen Kanäle von einem Abkömmling der Basisklasse »*IO_Channel*« (siehe Abb. 3.2) repräsentiert. Wobei jeder Kanal als Eigenschaft eine Map mit Entitäten führt, welche durch Objekte des Typs »*Channel_Entity*« (siehe Abb. 3.3) repräsentiert werden. Eine Entität wiederum weiß,

wie groß ihre Breite ist, also wie viele Bits sie hat und ob sie nur lesbar oder auch beschreibbar ist. Eine weitere Klasse »*IO_Channel_AccessWrapper*« (siehe Abb. 3.1) bündelt alle vorhandenen Kanäle inklusive der jeweiligen Entitäten. Zudem erleichtert Sie mittels Überladung der Array-Operatoren den Zugriff. Ein Zugriff ist dann wie in Listing 3.2 gezeigt wird möglich. Dazu müssen die Entsprechenden Kanäle vorher wie in Listing 3.1 zu sehen initialisiert und einer Instanz der Klasse »*IO_Channel_AccessWrapper*« übergeben werden. Der für den späteren Zugriff auf den Kanal nötige Buchstabe, wird in diesem Zuge festgelegt, während die Buchstaben der untergeordneten Entitäten Bestandteil des jeweiligen Kanals sind und dementsprechend dort definiert werden.

```
chnl.insert(std::make_pair('T', IOChannelPtr(new IO_Channel_Virtual_Timer())));
chnl.insert(std::make_pair('P', IOChannelPtr(new IO_Channel_Virtual_Pipe("r7123d97a3", 0x07))));

// chnl is copied here!
commandProcessor cp(isg, chnl, webSocketSessions);
```

Quellcode 3.1: Initialisieren der Kanäle und Entitäten

```
// Initially read the timers config
std::string fn_timers = "timers.conf";
```

Quellcode 3.2: Zugriff auf Kanal und Entität exemplarisch

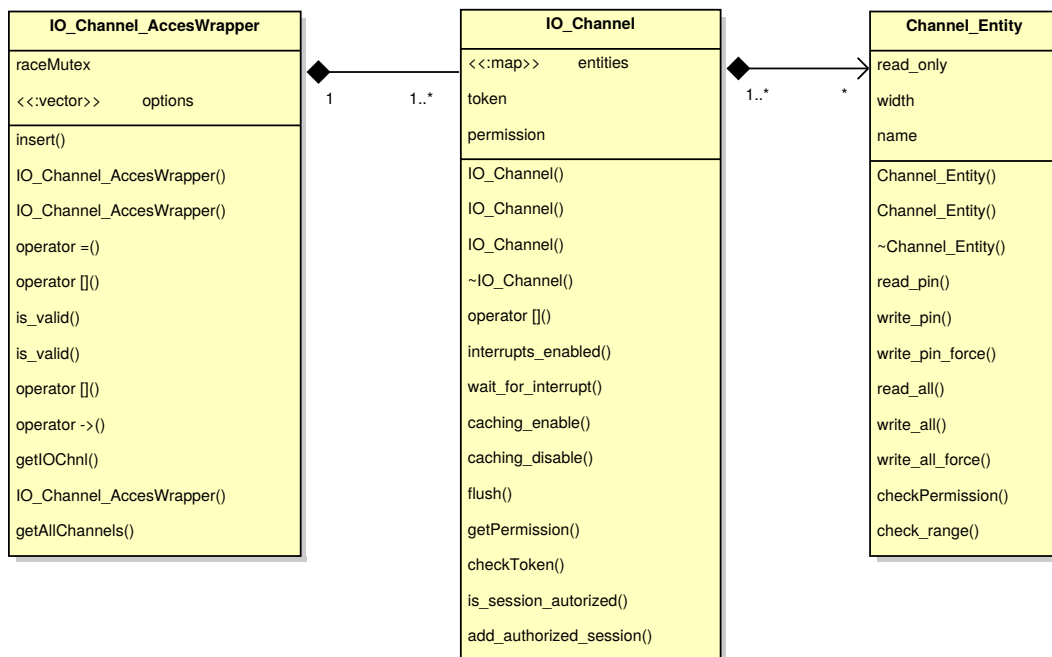


Abbildung 3.1: Klassendiagramm Aggregation der Klassen

Zu sehen ist die Aggregation zwischen dem »*IO_Channel_AccessWrapper*« den Kanälen »*IO_Channel*« und den Entitäten »*Channel_Entity*«. Alle Kanäle werden von dem »*IO_Channel_AccessWrapper*« zusammengefasst. Ein Zugriff erfolgt ausschließlich über diesen Weg.

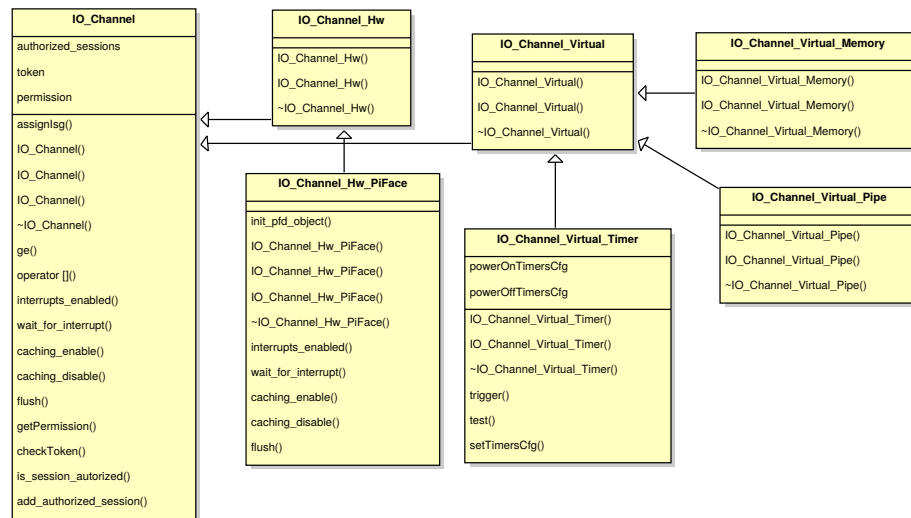


Abbildung 3.2: Vererbungshierarchie der Basisklasse IOChannel

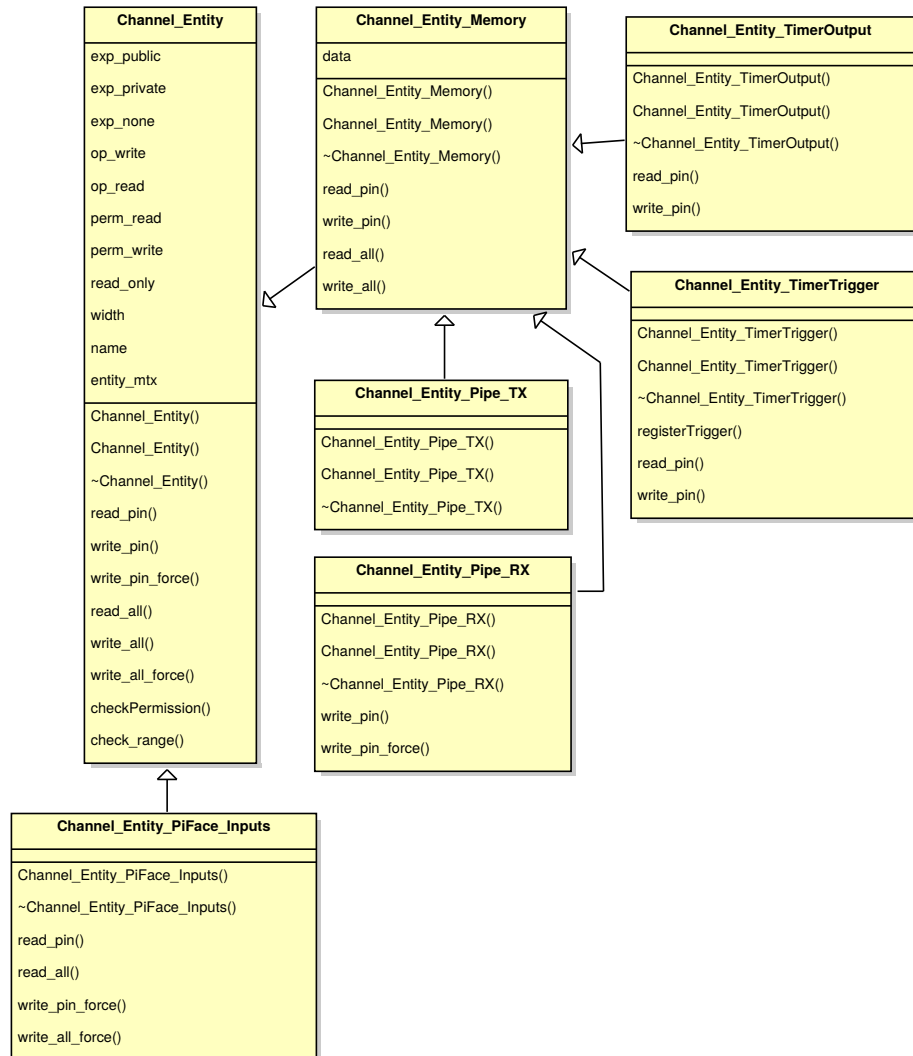


Abbildung 3.3: Vererbungshierarchie der Basisklasse ChannelEntity

Die Entitäten erben von der Basisklasse Entity. Sie überschreiben Methoden um lesend oder schreibend, entweder auf den gesamten Inhalt auf einmal oder Bitweise auf den Inhalt zuzugreifen. Dafür sind die Methoden `read_pin`, `write_pin` beziehungsweise `read_all` und `write_all` vorgesehen.

3.4.1 Hardware Kanäle

Der wohl wichtigste Kanal ist der Hardware Kanal. Die Klassenstruktur in Abb. 3.2 zeigt, dass alle Kanäle eine gemeinsame Basisklasse haben. Der Hardware Kanal ist eine Spezialisierung der Basisklasse, welche spezifische Funktionen für die Entsprechende Hardware beherbergt. So finden sich hier Methoden zur Steuerung des Caching sowie ein blockierender Aufruf, welcher auf einen Interrupt an der Hardware reagiert. Die Funktionen für den schreibenden und lesenden Zugriff auf die Ein- und Ausgänge der Hardware, sind in Entitäten gekapselt. Für eben jenen Zweck existieren zwei Spezialisierungen der »*Channel_Entity*« (siehe Abb. 3.3) Basisklasse, welche für den Zugriff mit »*i*« für den Eingang (input) beziehungsweise »*o*« für den Ausgang (output) referenziert werden.

3.4.2 Merker Bausteine

Die erste Erweiterung, welche wie in Abb. 3.1 zu sehen mit dem Buchstaben M referenziert wird, sind Memory Bausteine. Da ein Memory Baustein nicht richtungsorientiert sind, das heißt es existiert keine Eingangs oder Ausgangsgröße, wurde hier vom Benamungsschema für Entitäten abgewichen. Eine Referenzierung ist hier mittels eines oder mehrerer Buchstaben geplant. Somit bietet ein Merker Baustein Platz für 8 Zustandswerte pro verwendeter Entität. Bislang ist lediglich die Entität »*a*« angelegt, wobei geplant ist die Anzahl an Buchstaben beziehungsweise Entitäten durch einen Konstruktorparameter nach Außen zu führen, womit die Anzahl wie in Abb. 3.1 zu sehen beim Einfügen bestimmt werden könnte, oder in einem weiteren Schritt in eine Konfigurationsdatei exportiert werden kann.

3.4.3 Timer Bausteine

Timer Bausteine waren die wohl komplexesten Bausteine. Eine Verzögerung im Programmablauf bedeutet entweder ein blockieren, was aber bedeuten würde, dass das Programm in dieser Zeit auch nichts anderes tun kann und somit nicht auf Ereignisse reagieren kann. Die andere alternative ist Multithreading, also das Einführen von Parallelen Handlungssträngen, welche weitere Probleme bringen. So muss zum Beispiel beim Zugriff auf gemeinsam genutzte Ressourcen dafür Sorge getragen wer-

den, dass nicht gelesen wird während gerade geschrieben wird. Des weiteren musste die Hauptschleife überarbeitet werden. Bislang wurde die Programmlogik bei jedem eintreten eines Hardware-Ereignisses genau einmal durchlaufen. Jedoch muss die Programmlogik nun auch durch Ablauf eines Timers erneut durchlaufen werden. Gelöst wurde dies mithilfe einer Binären Variable als Schalter, welche zusammen mit eines Mutex zur Vermeidung von gleichzeitigem Zugriff, und einer Condition Variable zur Benachrichtigung des Hauptprozesses, in die Klasse »*iterationSwitchGuard*« gekapselt wurde. Ein Timerbaustein (mehrere sind Theoretisch möglich), bietet 8 Konfigurierbare Timer. Dabei gibts es eine Entität um einen Timer auszulösen »*t*« (Trigger) und eine Entität, welcher als Ausgang fungiert »*o*« (Output). So ergibt sich in der Programmlogik die selbe Syntax, welche sich auch bei Hardwarekanälen bietet: Ein explarisches triggern des Timers 3 würde erfolgen, indem eine Zeile der Logikdatei mit »*Tt3=*« beginnt. Eine Verwendung des Ausgangswertes desselben Timers, würde durch integrieren des Bezeichners »*[To3]*« in die Logikzuweisung eines anderen Bezeichners erfolgen. Die Parametrisierung des Timers erfolgt durch einbinden einer Konfigurationsdatei (siehe Listing 3.3) hierbei lassen sich für jeden Timer eine Einschaltverzögerung sowie eine Ausschaltverzögerung definiert werden. Jeder Wert ungleich null bewirkt eine Verzögerung, eine gleichzeitige Nutzung von Ein und Ausschaltverzögerung ist möglich. Werte gleich null bewirken eine sofortige Änderung am Ausgang, sobald sich der Eingangswert verändert. Beim Einlesen der Konfiguarionsdatei werden sämtliche Leerzeichen entfernt. Zudem kann mit einem Semikolon »*;*« oder einer Raute »*#*« ein Kommentar eingeleitet werden, welches alle restlichen Zeichen der Zeile beim Einlesen entfernt. Die Verzögerungszeiten sind in Millisekunden anzugeben, wobei die Reihenfolge der nicht sukzessive erfolgen muss. Der Maximalwert beläuft sich auf 4.294.967.295, was die größte in unsigned long darstellbare Zahl ist und umgerechnet etwa 49 Tagen entspricht.

```
timer =T 0
powerOnDelay=5000
powerOffDelay=0

timer=T7
powerOnDelay=1000
powerOffDelay=1000

timer=T1
powerOnDelay=10000
powerOffDelay=4294967295
```

Quellcode 3.3: Beispiel der Timer Konfigurationsdatei

3.4.4 Virtuelle Kanäle

Virtuelle Kanäle sind ein Konzept dass die Kommunikation der Steuerung mit entfernten Endpunkten ermöglichen. Ein Virtueller Kanal ist hierbei lediglich ein veränderter Memory Baustein, welcher entgegen anders als ein normaler Memory Baustein wieder richtungsorientiert ist. Das heißt es gibt eine Eingangs-Entität und eine Ausgangs-Entität. Hierbei muss beachtet werden, dass die Perspektive so gesetzt ist, dass ein Eingang die extern empfangenen Daten beinhaltet. Während in den Ausgang geschrieben wird. Die Gegenseite bildet der in **REF** beschriebene WebSocket Server, beziehungsweise was auch immer die Gegenseite bildet. Das kann das WebFrontend **REF** dieses Projektes sein, oder auch jeder andere WebSocket fähige Client. Angedacht ist auch die Implementierung eines WebSocket Clients in das Backend, womit sich zwei Einheiten verbinden ließen.

3.5 Laden der Programmlogik aus Datei

Das Logikprogramm wurde im nächsten Schritt in eine Textdatei ausgelagert. Sie soll künftig von der grafischen Benutzeroberfläche automatisch erstellt werden können, kann aber nach wie vor auch von Hand erstellt werden. Die Datei enthält einen Ausgang pro Zeile, gefolgt von einem Gleichheitszeichen und den Abhängigkeiten. Zeichen die auf ein Semikolon folgen, werden dabei als Kommentar gewertet und ausgelassen. Sie wird bei Programmstart eingelesen und verbleibt dann im Speicher. Zum Speichern der Daten wird ein »*Vector*« benutzt, welcher für jede Zeile der

Logikdatei einen weiteren Eintrag erhält. Da vor einem Gleichheitszeichen lediglich ein Ausgang stehen darf (z.B. ?? Zeile 1: »Ho0«), ist somit jeder Eintrag des Vectors die Definition genau einer Abgeschlossenen Zuweisung. Bei Mehrfacher Definition eines Ausgangs, überschreibt die spätere Definition die vorangegangene.

```

Ho0 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho1 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho2 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho3 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho4 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho5 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho6 = [Hi0] | [Hi1] | [Hi2] | [Hi3] | [Hi4] | [Hi5] | [Hi6] | [Hi7]
Ho7 = ( [Hi0] & [Hi1] ) | ( [Hi2] & [Hi3] ) | ( [Hi4] & [Hi5] ) | ( [Hi6] & [Hi7] )

```

Quellcode 3.4: Beispiel der Programmlogik Datei

3.5.1 Erneutes Laden der Logik zur Laufzeit

Zum erneuten einlesen der Logik, wurde ein Signalhandler vorgesehen, welcher auf das Signal SIGUSR1 hört. Die entsprechende Funktion ließt dann die Logikdaten erneut ein und überschreibt damit die vorherige Version im Speicher. Damit muss die Steuerung nicht neu gestartet werden und ermöglicht das Steuerungsprogramm zur Laufzeit zu verändern.

3.6 Hauptschleife und Iterationslogik

Das Logikprogramm liegt nun im Speicher und das Programm betritt die Hauptschleife. Doch treibt ein einfaches polling die Auslastung des Prozessors unnötig nach oben. Nun könnte nach jedem Durchlauf eine gewisse Zeit gewartet werden, bevor eine neue beginnt. Doch das verringert die Reaktionszeit der Steuerung. Die beste Lösung findet sich direkt im Treiber des PiFaceDigital. Hier bietet sich eine Funktion, die bis zum Eintreten einer Veränderung der Eingangswerte blockiert. Intern Eingesetzt wird hier ein Epoll, welcher einen File descriptor auf einen der GPIOs des Raspberry Pi überwacht und diesen somit als Interrupt Kanal nutzt.

3.7 Logik Prozessor

Den Kern des Projektes bildet der in Quellcode 3.5 gezeigte Logikprozessor. Es ist eine Methode, welche den in Unterabschnitt 3.5 geladenen Logik-Vektor zeilenweise durchläuft. Im ersten Schritt wird die Zeile nun auf Vorhandensein eines Gleichheitszeichens hin überprüft (Quellcode 3.5 Zeile 194). Eine Zeile ohne Gleichheitszeichen wird hierbei schlicht verworfen. Andernfalls wird zunächst der Teil vor dem Gleichheitszeichen näher untersucht. Dabei wird sichergestellt dass der Bezeichner aus drei Zeichen besteht, wobei die ersten beiden alphabetisch und der letzte numerisch sein muss. Diese drei Teile werden dann (Zeile 200-202) für die weitere Verarbeitung gespeichert. Der fertig ausgewertete Teil nach dem Gleichheitszeichen, auf dessen Verarbeitung nachfolgend noch etwas genauer eingegangen wird und der nunmehr entweder 0 oder 1 ist, wird dem entsprechenden Kanal nun zugewiesen. Wie in Quellcode 3.5 Zeile 222 zu sehen, wird nun eine Instanz `chnl` der Klasse `IO_Channel_AccessWrapper` für den eigentlichen Zugriff auf die Ressource verwendet. Zuvor müssen jedoch die vorher gespeicherten Teile des Bezeichners eingefügt werden verwendet. (Siehe Unterabschnitt 3.4 »*Klassenstruktur und Kanäle*«). Zudem wird dem Bezeichner in diesem Zuge ein Wert zugewiesen. Um diesen zu erhalten muss nun die Gleichung gelöst werden. Diese beinhaltet ebenfalls Bezeichner, welche vom Funktor `replaceIdentifier` durch dessen aktuellen Wert ersetzt wird. Zurück gegeben wird ein String, welcher nun nur noch aus Binärzahlen, arithmetischen Operatoren (& für und bzw. | für oder) und Klammern besteht. Die eigentliche Lösung der Gleichung erfolgt dann mittels eines Funktionsaufrufes auf einen in Boost Spirit geschriebenen Parser (siehe Unterabschnitt Unterabschnitt »3.1 Parsen von Logikausdrücken«).

```

186 void logicEngine(IO_Channel_AccesWrapper& chnl, std::vector<std::string>& softLogic){
187     bool      dbg      = true;
188     std::string delimiter = "=";
189     size_t     found;
190
191     // Iterates over Vercor row by row.
192     for (std::string softLogicRow: softLogic) {
193         // Splits the output string by the '=' sign in two parts, if found.
194         if ((found = softLogicRow.find('=')) != string::npos){
195             // The part before the '=' is the "output" to which the result will be assigned to
196             string assignedEntityStr = softLogicRow.substr(0,found);
197             if(assignedEntityStr.size() != 5){
198                 throw std::invalid_argument(errMsg);
199             }
200             char    c_io_channel    = assignedEntityStr.at(1);
201             char    c_channel_entity = assignedEntityStr.at(2);
202             int     c_pin_num       = assignedEntityStr.at(3) - '0';
203
204             // String after '=' is equation including variables called identifiers (e.g. [Ho0] ),
205             // arithmetic operators ( &, | ) and round brackets. Also allowed are literals (0 or 1)
206             string equationStringVariables = softLogicRow.substr(found+1, string::npos);
207
208             // Instantiate the replace-identifier Functor/Class thats used to
209             // inject the IO_Channel_AccessWrapper instance into the actual replace-function.
210             replaceIdentifier rplacIds(chnl);
211
212             // Replaces every occurance of a Identifier by its current state. (e.g. [Hi0] => 1 / 0 )
213             string equationStringLiterals = regex_replace( equationStringVariables,
214                                                         regex("\\[[A-Z][a-z][0-8]\\]"),
215                                                         rplacIds
216                                                         );
217
218             // Eventually the logic string will be evaluated (e.g. "!0 & 1 | 1" => 1)
219             bool logic_equation_res = evaluateLogicString(equationStringLiterals);
220
221             // And is then assigned to the Identifier before the '='
222             chnl [c_io_channel] [c_channel_entity] -> write_pin(logic_equation_res, c_pin_num);

```

Quellcode 3.5: Logik Engine

3.8 WebSocket Server

3.8.1 Basis

Um die Steuerung mit der Außenwelt zu Verbinden wurde ein WebSocket (siehe 2.6) Server in das Backend integriert. Als Basis hierfür diente das Beispiel, welches von Vinnie Falco im Rahmen eines Vortrages auf der CppCon 2018 erstellt wurde [2]. Die Funktion dieses Beispiels ist das Bereitstellen eines WebSocket Chatservers. Dieser baut auf einem rudimentären HTTP Server auf und ist deshalb in der Lage auch

normale HTTP Anfragen zu beantworten. Auf diesem Wege erhält der Webbrowser des Clients die HTML Webseite, in welche gleichzeitig auch der JavaScript Chat-Client enthalten ist der letztendlich die direkte Verbindung zum WebSocket Server aufbaut. Nachdem die Verbindung erfolgreich hergestellt wurde, kann der Benutzer Texteingaben tätigen und diese an den Server senden, welcher diese dann an alle verbundenen Clients verteilt. Da es nun relativ einfach umsetzbar ist, anstatt Text auch andere Formate wie etwa JSON über diesen Server an die Clients zu senden, schien dieses Beispiel eine ideale Grundlage zu sein. So könnte anstatt Text ein JSON mit dem Status der Ein- und Ausgänge der Steuerung an die verbundenen Clients verteilt werden, welche diese dann mit einer JavaScript Anwendung so umsetzen, dass der Status der Ein- und Ausgänge visuell dargestellt werden. Zudem ist über diesen Kommunikationskanal auch ein Schreibzugriff denkbar. So könnte ein symbolisch dargestellter Schalter im Frontend beim Anklicken per JavaScript Funktion einen Befehl an das Backend senden, welches diesen dann ausführt.

3.8.2 Erweiterung

Das vorher erwähnte Beispiel wurde in einem ersten Schritt in einen Unterordner des Projektes verschoben und das CMake-File dahingehend geändert, das anstatt eines eigenen Executables eine Library gebaut wird, welche dann vom Grundprojekt verwendet wird. Das Unterprojekt wurde zudem als Abhängigkeit deklariert sodass es beim bauen des Hauptprojektes automatisch neu gebaut wird, sofern sich Änderungen im Unterprojekt ergeben haben. Die einfachste Erweiterung war es, die schon vorhandene **broadcast** Funktion zu verwenden, um bei einer Änderung an den Ein- und Ausgängen eine Nachricht an alle verbundenen Clients zu senden. Da jedoch immer nur eine Nachricht gleichzeitig gesendet werden kann, wurde eine **Queue** verwendet, welche die Nachrichten sammelt, und diese dann nacheinander zustellt. Im nächsten Schritt wurden eingehende Nachrichten nun nicht mehr bloß eins zu eins an die **broadcast** Methode übergeben, und damit an alle Clients weitergeleitet. Stattdessen mündeten diese nun in eine weitere **Queue**: »*Command-Queue*«. Eingehende Nachrichten sollen damit gesammelt werden um die darin enthaltenen Befehle auszuführen.

3.8.3 Zugriffsbeschränkung

Um zu verhindern das jeder Client nun unkontrolliert vollen Zugriff auf alle Kanäle der Steuerung erhält, wurde daraufhin ein Konzept für die Zugriffsbeschränkung erarbeitet. Dafür wurden zwei Benutzerrollen eingeführt. **public** und **private**. Sie gelten für jeden Kanal und jeden Client individuell. Dafür besitzt jeder Kanal ein **private_token**. Authentifiziert sich ein Benutzer mittels **auth:<Kanal>:<Token>** für den Kanal, so wechselt er für diesen Kanal in die Rolle **private**. Da jeder Kanal aus mehreren Entitäten besteht, besitzt nun jede dieser Entitäten die Eigenschaften **expose_read** und **expose_write**. Zulässige Werte hierfür sind jeweils **public**, **private** oder **none**. Bei Zugriff auf den Kanal bzw. die Entität wird nun darauf geprüft ob der Benutzer **mindestens** die eingetragene Berechtigung hat. Nehmen wir zum Beispiel den Kanal Hardware »H«. Dessen Entitäten **i** für die Eingänge und **o** für die Ausgänge haben beide **expose_read** auf **private** gesetzt. Das **private_token** ist mit **top-secret-123** definiert. Ein Verbundener Client müsste sich nun erst mit dem Befehl **auth:H:top-secret-123** für den Kanal authentifizieren, bevor er Statusinformationen über die Entitäten dieses Kanals erhält. Wäre **expose_read** auf **public** gesetzt, würde er diese Informationen ohne vorherige Authentifikation erhalten. Anders herum, würde der Wert **none** bedeuten, dass niemand - nicht einmal ein Authentifizierter Client - Statusinformationen zu dieser Entität erhält. Listing 3.6 stellt ein typisches Statusupdate der privaten Entitäten **Qo** und **Qi** dar, darauf folgt das Update aller Entitäten, die als **public** definiert wurden.

Quellcode 3.6: JSON Status Update von Privaten Entitäten

```
1 {"Qo" : 4, "Qi" : 1}
2 {"Po" : 170, "Pi" : 4, "Ho" : 0, "Hi" : 6}
```

3.9 Benutzeroberfläche

(TODO: Grafik Erstellen)

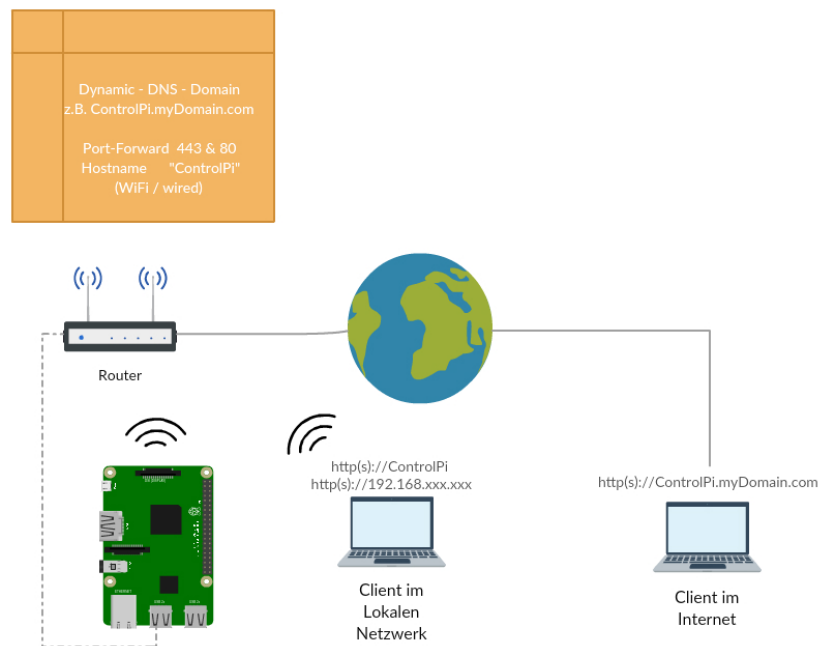


Abbildung 3.4: Zugriff auf die Weboberfläche

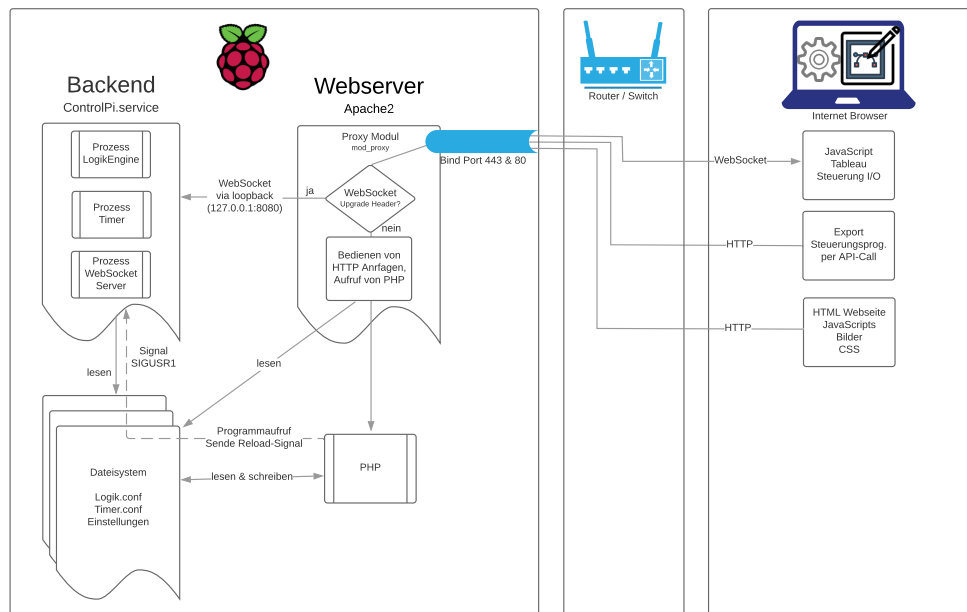


Abbildung 3.5: Networking Übersicht

3.9.1 Übersicht Komponenten

Die eigentliche Anwendung ist hauptsächlich in HTML und JavaScript und wird im Browser des Benutzers ausgeführt. Um die Benutzeroberfläche mit dem Rest der Anwendung kommunizieren zu lassen, bedarf es mehrerer Komponenten. Obwohl der Hauptteil der Anwendung über JavaScript Websockets eine direkte Verbindung vom Browser zum Backend herstellt, wird als direkte Schnittstelle zum Browser ein Apache2 Webserver eingesetzt. Dieser wird um das Proxy-Modul modproxy erweitert, was es ermöglicht, an das Backend gerichtete Anfragen mit dem WebSocket Header auf die lokale loopback Adresse 127.0.0.1 weiterzuleiten, während alle sonstigen Anfragen vom Webserver direkt beantwortet werden können. Zum einen wird hierdurch die Konfiguration von Verschlüsselten Verbindungen über SSL und den Abruf der dazu nötigen Zertifikate erheblich vereinfacht, denn einige Zertifizierungs-Anbieter wie zum Beispiel Let's Encrypt ^{*REF*} bieten extra auf Apache zugeschnittene Skripte, die ebendies fast vollständig automatisieren. Zum anderen ist es so möglich, die Auslieferung der HTML und JavaScript Quellen an den Webserver zu delegieren, anstatt bei einer eigenen Implementierung Sicherheitslücken und Performance-Probleme zu riskieren. Weiterhin ermöglicht es dieser Ansatz auch PHP Skripte zu verarbeiten, dies wird zum Beispiel eingesetzt, um Konfigurationsdateien vom Browser auf dem Server abzulegen, oder durch dem Backend Signale zu senden, wenn es die Logikdatei erneut einlesen soll.

3.9.2 Apache Webserver

3.9.3 JavaScript Anwendung

3.9.4 Logik Editor CircuitVerse

Die Wahl für einen Graphischen Logikeditor fiel auf den mit der MIT Lizenz gekennzeichneten Editor CircuitVerse ^{*REF*}. Die Lizenzierung gestattet es, das Programm zu verändern und weiter zu verbreiten. Der in Abbildung 3.6 gezeigte Screenshot, zeigt den Aufbau des Editors. Im linken Abschnitt kann dabei zwischen verschiedenen Bauteilen gewählt werden, welche dann per Drag & Drop auf die rechts daneben befindliche Zeichenfläche gezogen werden können. Grüne Punkte kennzeichnen dabei die Anschlussknoten, welche dann wiederum durch ziehen mit gehaltener linker Maustaste miteinander verbunden werden können. CircuitVerse bietet überdies die Möglichkeit, die erstellte Zeichnung in einem eigenen Format zu speichern, sowie sie als Grafik zu exportieren. Des weiteren gibt es die Möglichkeit eine Zeichnung anhand einer Logiktablelle automatisch erstellen zu lassen.

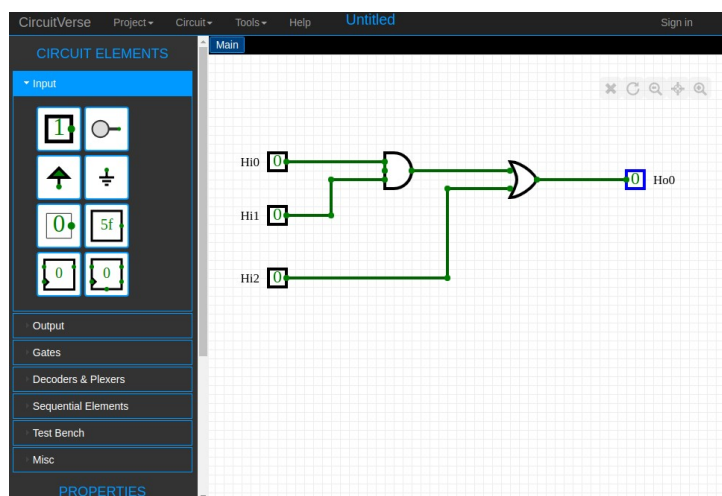


Abbildung 3.6: Screenshot einer Logikschaltung in CircuitVerse

Erweiterung für den Export

Eine Erweiterung von CircuitVerse ist nötig, um die gezeichnete Logikschaltung in ein Format umwandeln zu können, welches vom Backend verstanden wird. Weiterhin muss das Benamungsschema (siehe Unterabschnitt »3.3 Benamungsschema«)

eingbracht werden. Letztlich muss auch die Bauteile-Auswahl auf jene Bauteile reduziert werden, welche auch tatsächlich in der Steuerung verfügbar sind. Unterabschnitt »3.5 Laden der Programmlogik aus Datei« beschreibt das Format der Datei, die letztlich aus einer gezeichneten Logikschaltung gewonnen werden soll. CircuitVerse verwendet für die Speicherung der Schaltung eine JSON-Struktur. Dabei wird in Bauteile und Knoten unterschieden. Ein Bauteil hat ein oder mehrere Knoten. Wobei Knoten miteinander verbunden werden können, und dann jeweils Informationen über angeschlossene Knoten enthalten. Abbildung 3.8 verdeutlicht, wie CircuitVerse die Knoten verwendet. Die Grundlage für die Abbildung ist ein Grafikexport von CircuitVerse, wobei die Knotennummern zum besseren Verständnis der Funktionsweise besonders gekennzeichnet wurden. In »Abbildung 3.7 Darstellung der JSON Datenstruktur« ist die zugehörige JSON-Datenstruktur abgebildet.

```

▼ {name: "ControlPi", timePeriod: 500, clockEnabled: true, projectId: "dIZeQVPjuofBL5XFCmXS",...}
  clockEnabled: true
  focussedCircuit: 28983776459
  name: "ControlPi"
  projectId: "dIZeQVPjuofBL5XFCmXS"
  scopes: [{layout: {width: 100, height: 80, title_x: 50, title_y: 13, titleEnabled: true},...}]
  ▼ 0: {layout: {width: 100, height: 80, title_x: 50, title_y: 13, titleEnabled: true},...}
    ▶ AndGate: [{x: 320, y: 110, objectType: "AndGate", label: "", direction: "RIGHT", labelDirection: "LEFT",...}]
    ▶ Input: [{x: 140, y: 110, objectType: "Input", label: "[Hi0]", direction: "RIGHT", labelDirection: "LEFT",...}]
    ▶ NotGate: [{x: 230, y: 150, objectType: "NotGate", label: "", direction: "RIGHT", labelDirection: "LEFT",...}]
    ▶ Output: [...]}
  ▼ allNodes: [{x: 10, y: 0, type: 1, bitWidth: 1, label: "", connections: [5]},...]}
    ▶ 0: {x: 10, y: 0, type: 1, bitWidth: 1, label: "", connections: [5]}
    ▶ 1: {x: 10, y: 0, type: 0, bitWidth: 1, label: "", connections: [10]}
    ▶ 2: {x: -10, y: -10, type: 0, bitWidth: 1, label: "", connections: [5]}
    ▶ 3: {x: -10, y: 10, type: 0, bitWidth: 1, label: "", connections: [6]}
    ▶ 4: {x: 20, y: 0, type: 1, bitWidth: 1, label: "", connections: [8]}
    ▶ 5: {x: 150, y: 100, type: 2, bitWidth: 1, label: "", connections: [0, 2]}
    ▶ 6: {x: 310, y: 150, type: 2, bitWidth: 1, label: "", connections: [3, 13]}
    ▶ 7: {x: 10, y: 0, type: 1, bitWidth: 1, label: "", connections: [12]}
    ▶ 8: {x: 10, y: 0, type: 0, bitWidth: 1, label: "", connections: [4, 11]}
    ▶ 9: {x: 10, y: 0, type: 1, bitWidth: 1, label: "", connections: [10]}
    ▶ 10: {x: 440, y: 250, type: 2, bitWidth: 1, label: "", connections: [1, 9]}
    ▶ 11: {x: 430, y: 120, type: 2, bitWidth: 1, label: "", connections: [8]}
    ▶ 12: {x: -10, y: 0, type: 0, bitWidth: 1, label: "", connections: [7]}
    ▶ 13: {x: 20, y: 0, type: 1, bitWidth: 1, label: "", connections: [6]}
  id: 28983776459
  ▶ layout: {width: 100, height: 80, title_x: 50, title_y: 13, titleEnabled: true}
  name: "Main"
  ▶ nodes: [5, 6, 10, 11]
  timePeriod: 500

```

Abbildung 3.7: Darstellung der JSON Datenstruktur

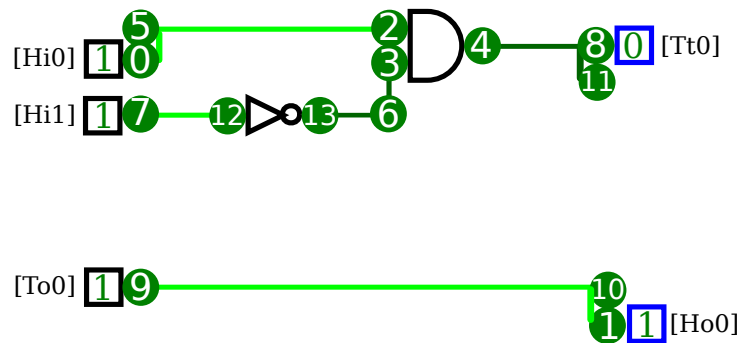


Abbildung 3.8: Logikschaltung mit gekennzeichneten Knoten

In der vorangehenden Abbildung 3.8 ist zu sehen, dass jedes Bauteil einen oder mehrere Knoten besitzt. In Abbildung 3.7 ist zu sehen, dass jeder dieser Knoten eine eigene Zeile unterhalb von `allNodes` besitzt. Hieraus geht hervor, mit welchen anderen Knoten er verbunden ist. Außerdem verbirgt sich im Feld `Typ`, die Information ob es sich um einen Ausgangsknoten einen Eingangsknoten oder einen Verbindungsknoten handelt. Ist der Knoten vom Typ Eingangsknoten oder vom Typ Ausgangsknoten so gehört er einem Bauteil wie einem Logikgatter, einem Eingangsbausteinen oder einem Ausgangsbausteinen. Die Anordnung der Knoten ist logischer weise anders herum wie die der Bauteile, so besitzt ein Ausgangsbau- stein genau einen Eingangsknoten (z.B. 3.8 Knoten 1), sowie ein Eingangsbaustein über einen Ausgangsknoten verfügt (z.B. Abbildung 3.8 Knoten 9). Ein Gatter hingegen hat einen Ausgangsknoten sowie einen oder mehrere Eingangsknoten. (z.B. Abbildung 3.8 Eingangssknoten 3,4 und Ausgangsknoten 4) Für die Umwand- lung in Textform schien es am sinnvollsten an einem Ausgang anzusetzen und eine Wegfindung durchzuführen, welche beim Erreichen eines Eingangsbausteins einen Endpunkt hat. Dazu durchläuft der in Abbildung 3.10 gezeigte rekursive Algorithmus den Weg von einem Ausgangsknoten bis zu einem Eingangsknoten. Ist ein Eingangsknoten erreicht, so wird geprüft ob es sich um den Ausgang eines Eingangsbausteins handelt, oder um den Ausgang eines Logikgatters. Da Circuit- Verse die Knoten jedoch nicht nach der Art des zugehörigen Bauteils, sondern nur in Eingangsknoten Ausgangsknoten oder Leitungsknoten unterteilt, muss die in 3.7 gezeigte Struktur oberhalb der Eigenschaft `allNodes` vorab durchlaufen

werden. Hierbei wird ein Lookup Array angelegt (siehe Abbildung 3.9), welches die Bauteile mit der Knotennummer ihres Ausgangs indiziert. Bezogen auf das in Abbildung 3.8 gezeigte Beispiel, würde für das AND-Gate ein Eintrag mit dem Index 4 angelegt. Abbildung 3.9 Zeigt den Eintrag für dieses AND-Gate in der aufgeklappten Zeile mit dem Index 4. Das Array `nodesIn` beinhaltet die Knoten 2 und 3. An diesem Punkt, ruft die Funktion zur Wegfindung sich nun selbst auf. Auf diese Weise werden sämtliche Logikgatter durchlaufen. In diesem Beispiel (siehe Abbildung 3.8) würde die Wegfindung von Knoten 2 über Knoten 5 auf Knoten 0 treffen. Dieser ist jedoch kein Gateknoten sondern der Knoten eines Eingangsbau- steines: Ein Endpunkt ist erreicht, das heißt es wird nun die Zeichenkette `[Hi0]` zurückgegeben. Zurück im vorher besprochenen AND-Gate, wird diese Zeichenkette nun mit dem Rückgabewert des Stranges an Knoten 3 und einem Kaufmännischen und (`&`) verkettet. Da das And Gate genau zwei Eingangsknoten hat, sind nunmehr alle Knoten abgearbeitet und der Baustein gibt die Zeichenkette `[Hi0] & ![Hi1]` zurück. Im Eingangsknoten wird diese Zeichenkette nun um den Bezeichner ergänzt. Somit ergibt sich `[Tt0]=[Hi0] & ![Hi1]` als Gesamtergebnis.

```

▶ 0: {nodesIn: Array(1), text: "[Hi0]", oType: "Input", type: "inp", nodeOut: 0}
▶ 1: {nodesIn: Array(2), text: "", oType: "AndGate", type: "gate", nodeOut: 4}
▶ 2: {nodesIn: Array(1), text: "[Hi1]", oType: "Input", type: "inp", nodeOut: 7}
▶ 3: {nodesIn: Array(1), text: "[To0]", oType: "Input", type: "inp", nodeOut: 9}
▼ 4:
  nodeOut: 4
  ▶ nodesIn: (2) [2, 3]
    oType: "AndGate"
    text: ""
    type: "gate"
  ▶ __proto__: Object
5: null
6: null
▶ 7: {nodesIn: Array(1), text: "[Hi1]", oType: "Input", type: "inp", nodeOut: 7}
8: null
▶ 9: {nodesIn: Array(1), text: "[To0]", oType: "Input", type: "inp", nodeOut: 9}
10: null
11: null
12: null
▶ 13: {nodesIn: Array(1), text: "", oType: "NotGate", type: "gate", nodeOut: 13}

```

Abbildung 3.9: Lookup Array

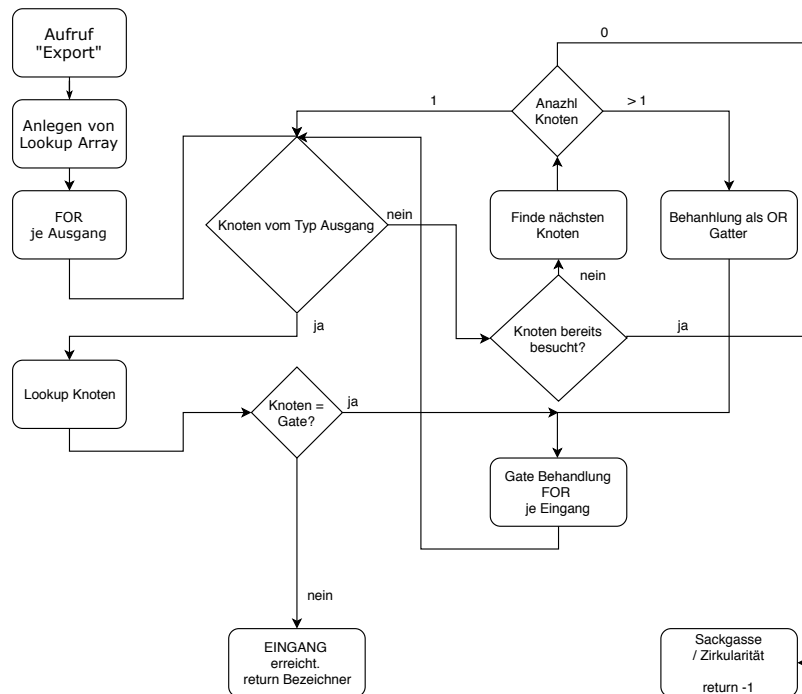


Abbildung 3.10: Programm Ablaufplan

Anpassung der Eingangsbausteine

Wie in »3.9.4 Erweiterung für den Export« besprochen, wurde CircuitVerse um einen Algorithmus zum Export der gezeichneten Schaltung erweitert. Dieser wird über einen neuen Menüpunkt (siehe 3.11) gestartet und zeigt dann wie in Abbildung 3.12 zu sehen, das Exportergebnis zur Überprüfung an. Nach der Bestätigung des Ergebnisses wird dieses per Ajax Aufruf an ein in Unterunterabschnitt 3.9.5 näher beschriebenes PHP Script übergeben, welches es dann als Textdatei speichert. Der Menüpunkt **Save Online** schreibt das bisherig verwendete JSON Format über die selbe Schnittstelle in eine weitere Textdatei auf dem Server. Dabei wird zudem eine Abbildung der Schaltung erzeugt und ebenfalls auf den Server übertragen. Letztlich dient die Schnittstelle auch dem lesenden Zugriff auf die JSON-Datei.

Sie wird beim ersten Aufruf des Logikeditors somit vom Server heruntergeladen. Damit kann die bereits vorhandene Steuerung erweitert oder verändert werden. Um zu vermeiden dass sich die Logik in der JSON Datei von der Logik in der vom Backend lesbaren Logikdatei unterscheidet, soll die Speicherung in einen Menüpunkt zusammengeführt werden. Dies dient vor allem auch dem einfacheren Verständnis durch den Endbenutzer. Es schränkt jedoch die Möglichkeit ein, neben der Steuerung die gerade „in Betrieb“ ist, weitere alternative Steuerungsprogramme abzuspeichern.

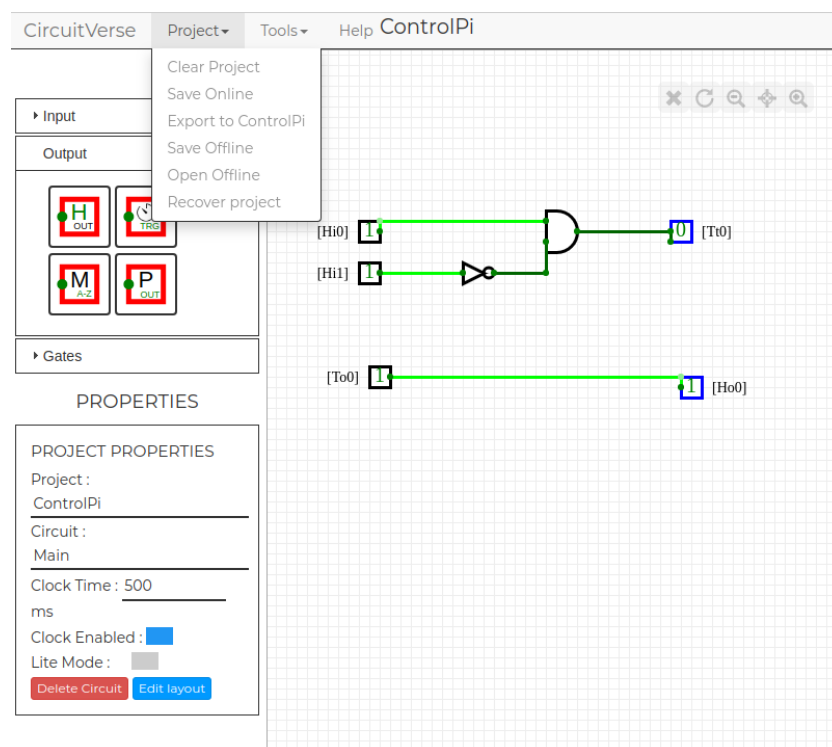


Abbildung 3.11: Umgestaltetes CircuitVerse

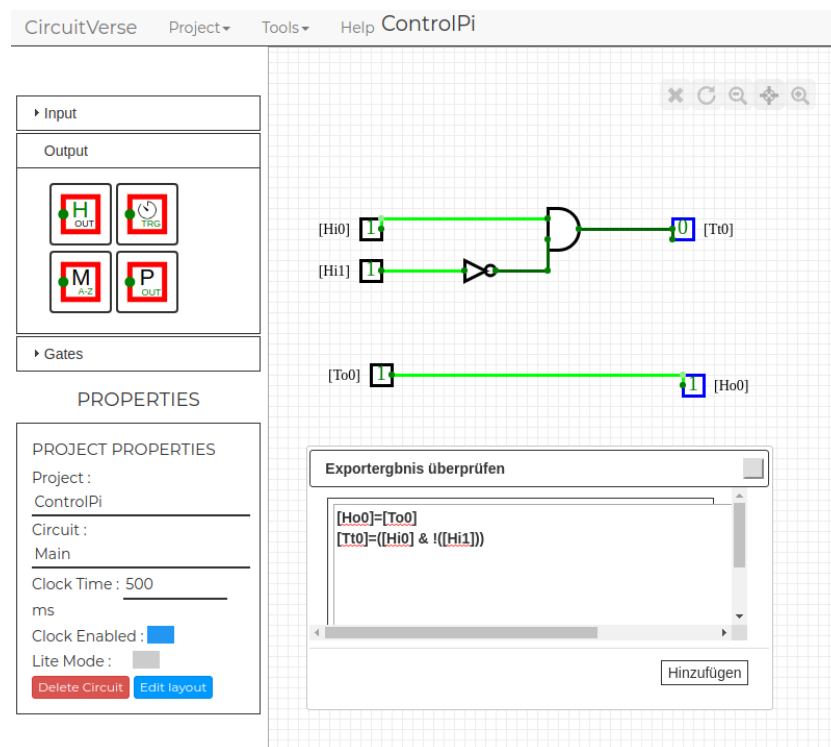


Abbildung 3.12: Dialog mit Exportergebnis

3.9.5 PHP Adapter Programme

Schnittstelle zum Speichern als Datei

Da CircuitVerse als JavaScript Anwendung im Browser ausgeführt wird und somit keinen direkten Zugriff auf das Dateisystem des Servers hat, ruft es beim Speichern oder beim Laden per Ajax eine URL auf. Um Dateien nun persistent in das Dateisystem des Servers ablegen zu können wird eine API Schnittstelle benötigt, welche über ein rudimentäres PHP-Skript `logicUpdateApi` realisiert wurde. Dies geschieht über ein anderes PHP Skript, welches ebenfalls über eine Ajax Anfrage angesprochen wird. Außerdem ließt dieses Skript ebendiese Datei, wenn es beim Laden dieser CircuitVerse Version über JavaScript mittels Ajax aufgerufen wird. Zwischen lesendem und schreibendem Zugriff wird über die Verwendete Zugriffsmethode unterschieden, wobei POST einen schreibenden Zugriff gewährt und GET

den Inhalt der Datei liest. Da der Logikeditor in ein Helles Template eingebettet werden soll, wurde außerdem das Farbschema und die Schriftart modifiziert.

WebSocket Adapter

Das im letzten Abschnitt 3.9.5 beschriebene Verfahren zum Ablegen von Konfigurationsdaten ist ein einmaliges Ereignis, dementsprechend wird hier auch eine normale HTTP Anfrage eingesetzt, welche nach vollständiger Übertragung abgebaut wird. Im Unterschied dazu wird für die Ansteuerung der Virtuellen Kanäle sowie für die Statusabfrage der Physischen ein und Ausgänge eine persistente Verbindung aufgebaut. Dabei baut der Client eine Verbindung zu dem WebSocker Server auf welcher in Kapitel 3.8 näher beschrieben wird. Um durch einmalige Ereignisse von externen Diensten per HTTP zu ermöglichen, baut ein PHP Script nun **on-demand** eine WebSocket Verbindung zu dem im Backend *REF* integrierten Server auf, setzt den gewünschten Befehl um und baut die Verbindung daraufhin wieder ab.

3.9.6 IFTTT

4 Auswertung

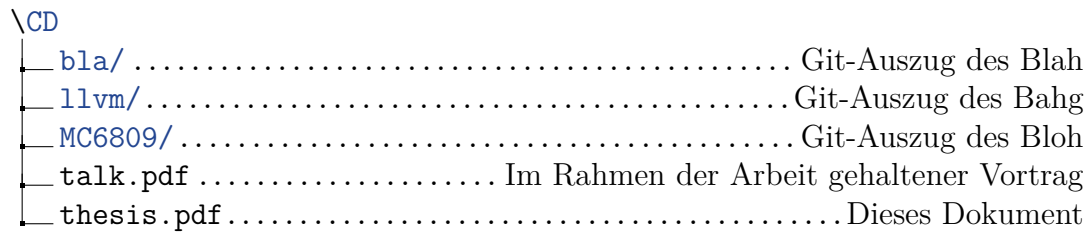
5 Ausblick

6 Literatur

- [1] Günter Wellenreuther und Dieter Zastrow. *Automatisieren mit SPS - Theorie und Praxis: Programmieren mit STEP 7 und CoDeSys, Entwurfsverfahren, Bausteinbibliotheken Beispiele für Steuerungen, ... PROFINET, Ethernet-TCP/IP, OPC , WLAN*. Mannheim, Jan. 2015.
- [2] Vinnie Valco. *Get Rich Quick! Using Boost.Beast WebSockets and Networking TS*. URL: <https://github.com/vinniefalco/CppCon2018> (besucht am 01.11.2018).

7 Anhang

Inhalt der beigelegten CD



The diagram shows a directory tree for a CD. The root is labeled \CD. It contains five entries: bla/, llvm/, MC6809/, talk.pdf, and thesis.pdf. Each entry is connected to a description by a dotted line. The descriptions are: Git-Auszug des Blah, Git-Auszug des Bahg, Git-Auszug des Bloh, Im Rahmen der Arbeit gehaltener Vortrag, and Dieses Dokument.

\CD	
└─ bla/ Git-Auszug des Blah
└─ llvm/ Git-Auszug des Bahg
└─ MC6809/ Git-Auszug des Bloh
└─ talk.pdf Im Rahmen der Arbeit gehaltener Vortrag
└─ thesis.pdf Dieses Dokument

Abbildung 7.1: Inhalt der beigelegten CD

Eingesetzte Software

In Tabelle 7.1 werden die wichtigsten Programme, die zum Entwickeln verwendet wurden, aufgelistet.

Tabelle 7.1: Liste der eingesetzten Software

(TODO: Look for Boost install script on RPI 3) Programm	Version
CMake	3.13.4
GCC	8.1.0
Boost	0.0.0
glibc	2.28
libstdc++	3.4.25
Linux Kernel	4.19.0
LLVM	7.0.0
Ninja	1.8.2
Python2	2.7.15
Python3	3.7.1