

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	2
Tabellenverzeichnis	2
Quellcodeverzeichnis	2
1 Einleitung	3
1.1 Ziel der Arbeit	3
1.2 Verwandte Arbeiten	4
1.3 Aufbau der Arbeit	4
2 Grundlagen	5
2.1 Was ist eine SPS	5
2.2 Ausgangssituation	5
2.2.1 Bedienoberfläche	6
2.2.2 Backend	7
2.2.3 Hardware	7
2.3 Versionsverwaltung	7
3 Umsetzung	9
3.1 C++ Library LibPiFace	9
3.2 Parsen von Logikausdrücken	10
3.3 Automatische Prüfung von Abhängigkeiten	10
3.4 Benamungsschema	11
3.5 Klassenstruktur und Kanäle	11
3.5.1 Hardware Kanäle	16
3.5.2 Merker Bausteine	16
3.5.3 Timer Bausteine	16
3.5.4 Virtuelle Kanäle	18
3.6 Laden der Programmlogik aus Datei	18
3.6.1 Erneutes Laden der Logik zur Laufzeit	19
3.7 Hauptschleife und Iterationslogik	19
4 Auswertung	20
5 Ausblick	21
6 Anhang	22

Abbildungsverzeichnis

2.1	Programm einer Easy Kleinsteuerng	6
3.1	Klassendiagramm Aggregation der Klassen	13
3.2	Vererbungshierarchie der Basisklasse IOChannel	14
3.3	Vererbungshierarchie der Basisklasse ChannelEntity	15
6.1	Inhalt der beigelegten CD	22

Tabellenverzeichnis

6.1	Liste der eingesetzten Software	22
-----	---	----

Quellcodeverzeichnis

3.1	Initialisieren der Kanäle und Entitäten	12
3.2	Zugriff auf Kanal und Entität exemplarisch	12
3.3	Beispiel der Timer Konfigurationsdatei	18
3.4	Beispiel der Programmlogik Datei	19

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

2.1 Was ist eine SPS

Die Grundsätzliche Funktion einer Speicherprogrammierbaren Steuerung ist, die Ermittlung der Ausgangswerte bzw. Schalterstellung durch eine logische Verknüpfung der Eingangswerte. 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.

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 anzusteuern 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.

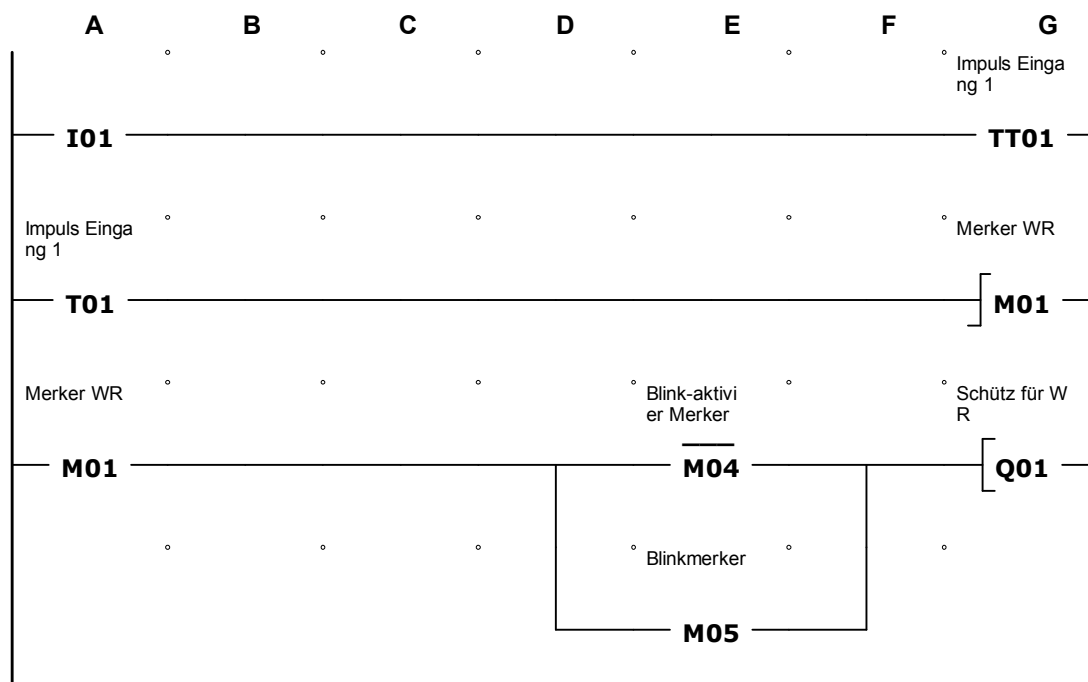


Abbildung 2.1: Programm einer Easy Kleinsteuerung

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 Bedienoberfläche 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. Dies Laufzeitinformationen sollen in dieser Bachelorarbeit ebenfalls dargestellt werden.

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, das 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 viel 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.

3 Umsetzung

3.1 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.

3.2 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.

3.3 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

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>

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.

3.4 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.

3.5 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.

```
IO_Channel_AccessWrapper chnl(&isg);
chnl.insert(std::make_pair('H', IOChannelPtr(new IO_Channel_Hw_PiFace("none", 0x07))));
chnl.insert(std::make_pair('M', IOChannelPtr(new IO_Channel_Virtual_Memory())));
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))));
```

Quellcode 3.1: Initialisieren der Kanäle und Entitäten

```
outputs = chnl['H']['o']->read_all();
printf("Outputs: 0x%x\n", outputs);
```

Quellcode 3.2: Zugriff auf Kanal und Entität exemplarisch

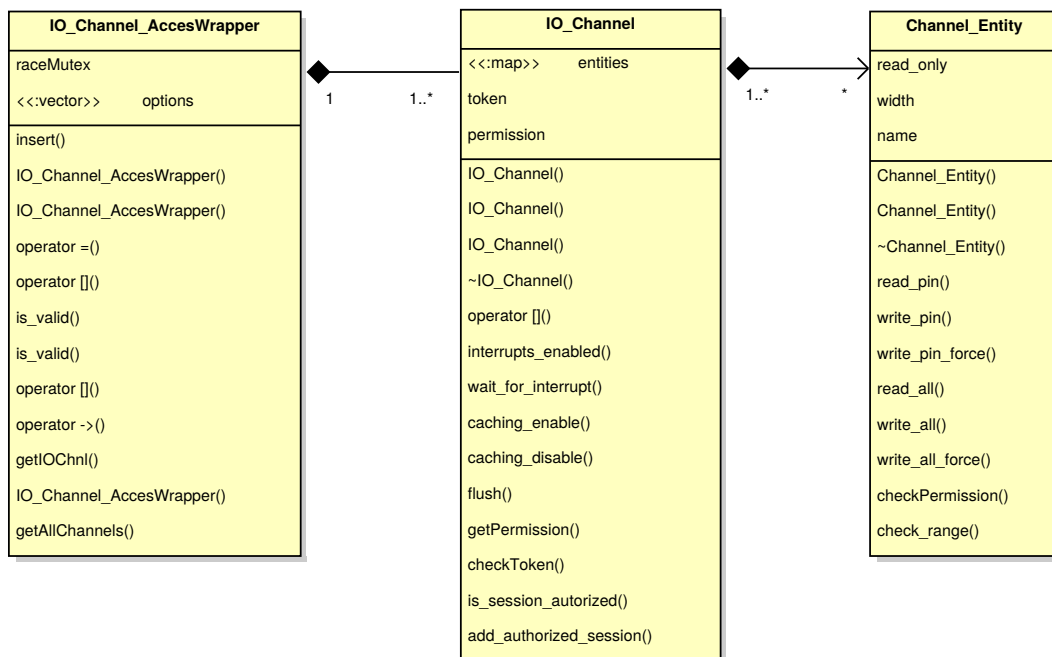


Abbildung 3.1: Klassendiagramm Aggregation der Klassen

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.

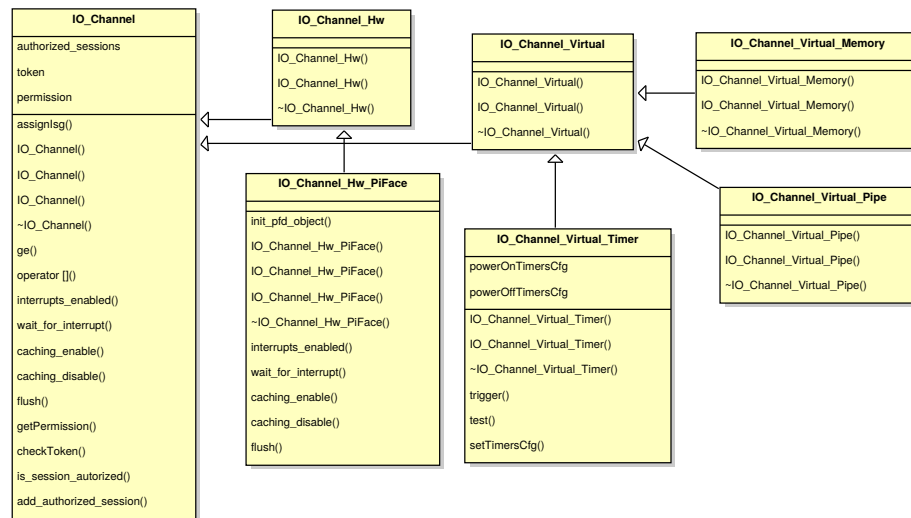


Abbildung 3.2: Vererbungshierarchie der Basisklasse IOChannel

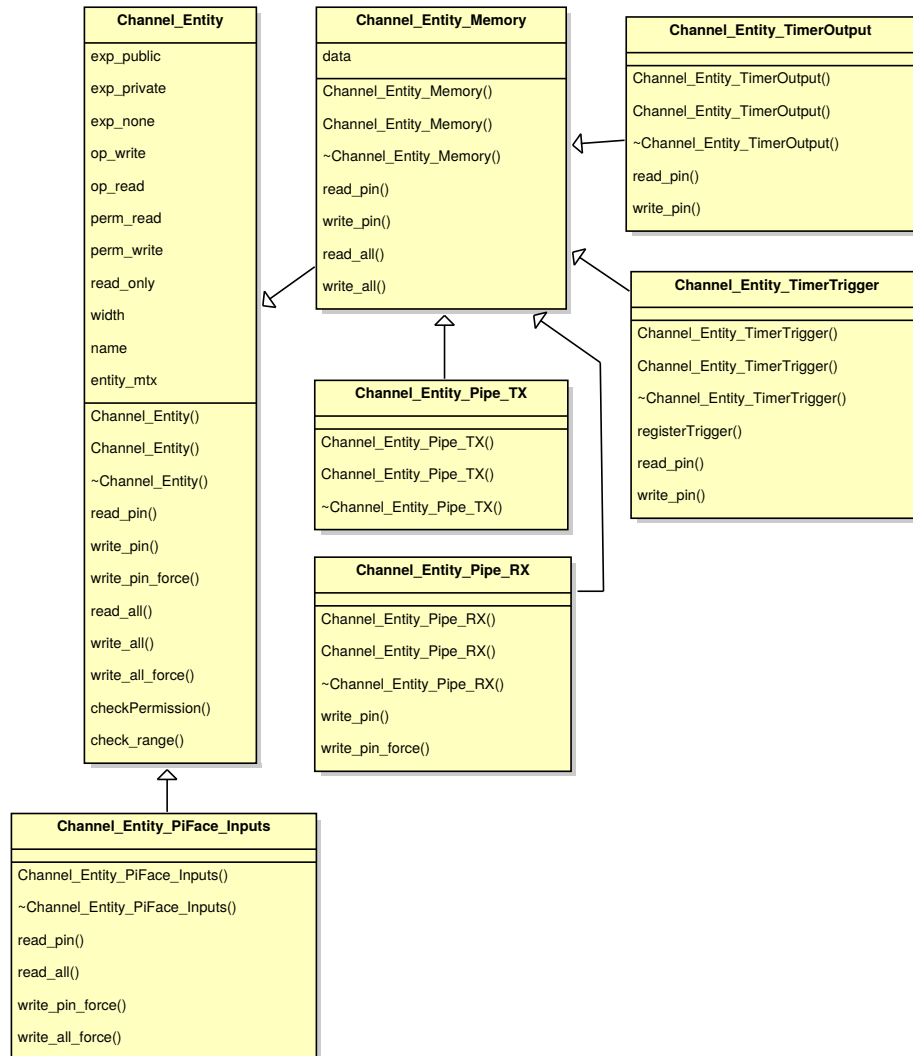


Abbildung 3.3: Vererbungshierarchie der Basisklasse ChannelEntity

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.

3.5.1 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.

3.5.2 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.

3.5.3 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

3.5.4 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.

3.6 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.

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

3.6.1 Erneutes Laden der Logik zur Laufzeit

Zum erneuten einlesen der Logik, wurde ein Signalhandler vorgesehen, welcher auf das Signal SIGUSR1 hört. Die entsprechende Funktion liest 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.

3.7 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.

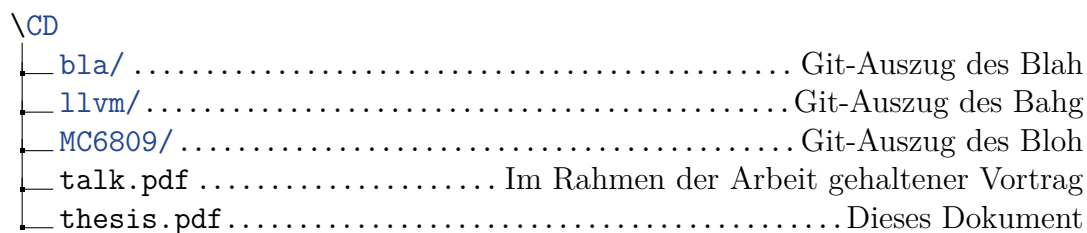
Beschreibung, Begründung

4 Auswertung

5 Ausblick

6 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 6.1: Inhalt der beigelegten CD

Eingesetzte Software

In Tabelle 6.1 werden die wichtigsten Programme, die zum Entwickeln verwendet wurden, aufgelistet.

Tabelle 6.1: 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