

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 Raspberry 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, Raspberry 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	3
1.3 Ausbau der Arbeit	3
2 Grundlagen	4
2.0.1 Was ist eine SPS	4
2.1 Konzept?	5
2.2 Ausgangssituation	5
2.2.1 Bedienoberfläche	5
2.2.2 Backend	6
2.2.3 Hardware	6
2.3 Versionsverwaltung	6
3 Umsetzung	8
3.1 C++ Library LibPiFace	8
3.2 Parsen von Logikausdrücken	9
3.3 Automatische Prüfung von Abhängigkeiten	9
3.4 Benamungsschema und Klassenstruktur	10
3.5 Laden der Programmlogik aus Datei	11
3.5.1 Erneutes Laden der Logik zur Laufzeit	11
3.6 Hauptschleife und Interrupt	11
3.7 Merker Bausteine	11
3.8 Timer Bausteine	11
3.9 WebSockets und Virtuelle Kanäle	11
4 Auswertung	12
5 Ausblick	13
6 Anhang	14

Abbildungsverzeichnis

6.1	Inhalt der beigelegten CD	14
-----	-------------------------------------	----

Tabellenverzeichnis

6.1	Liste der eingesetzten Software	14
-----	---	----

Quellcodeverzeichnis

1 Einleitung

1.1 Ziel der Arbeit

1.2 Verwandte Arbeiten

1.3 Ausbau der Arbeit

2 Grundlagen

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. Nachfolgend wird kurz darauf eingegangen was die eine Speicherprogrammierbare Steuerung eigentlich ist und weshalb man sie benötigt.

2.0.1 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.

2.1 Konzept?

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 Stompfad kann nunmehr bis zum Pluspol durchgezeichnet 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 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 ge-

zeichnet werden. Diese 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, 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 abzugraben 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 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 anbietet. 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 Zeitblanken *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 und Klassenstruktur

Nachdem die Auswertung beziehungsweise das Parsing der Logikausdrücke funktionierte, sollte auch die Auswertung der Bezeichner automatisiert werden. Ein Benennungsschema 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. Im Programm repräsentiert eine Klasse mit dem Namen Channel die Kanäle, wobei jeder Kanal als Eigenschaft einen Vektor mit Entitäten führt. Eine Entität wiederum weiß, wie groß ihre Breite ist, also wie viele Bits sie hat und ob sie nur Lesbar oder auch schreibbar ist. Durch Überladung der Operatoren der beiden Klassen ist ein Zugriff im Format `chnl['H']['i']` möglich. 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 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.

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 Interrupt

Das Logikprogramm liegt nun im Speicher und das Programm betritt die Hauptschleife. Doch treibt ein einfaches polling die Auslastung des Prozessors nach oben.

3.7 Merker Bausteine

3.8 Timer Bausteine

3.9 WebSockets und Virtuelle Kanäle

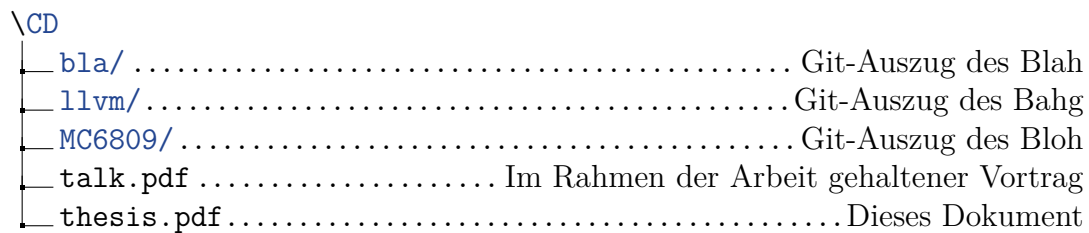
Beschreibung, Begründung

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 the root by a vertical line, and each entry has a horizontal line extending to its description. The descriptions are: bla/ Git-Auszug des Blah, llvm/ Git-Auszug des Bahg, MC6809/ Git-Auszug des Bloh, talk.pdf Im Rahmen der Arbeit gehaltener Vortrag, and thesis.pdf 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

Programm	Version
Clang	7.0.0
CMake	3.12.3
GCC	8.2.1
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