noch einzupflegen

Institut für eingebettete Systeme
und angewandte Informatik

Hochschule Bochum
Bochum University
of Applied Sciences
Campus
Velbert/Heiligenhaus

Integration eines OPC-Servers mit einer modularen, auf Linux basierenden speicherprogrammierbaren Steuerung

von

Lukas Friedrichsen und **Philipp Stenkamp**
Matrikelnummer: 013 200 111 Matrikelnummer: 014 215 793

Hausarbeit im Master-Studiengang
Mechatronik & Informationstechnologie

Eingereicht am: 5. Oktober 2018

1. Prüfer: Prof. Dr. rer. nat. Peter Gerwinski

Kurzfassung

Abstract

Serverless system integration on an industrial level This case study aims to demonstrate the advantages of a serverless system architecture in the field of IoT and Industrie 4.0. It is designed to serve as an example for how a traditional SCADA system can be substituted by a Linux-based PLC with connection to a cloud service via an edge gateway.

To evaluate the technological challenges the design of this architecture imposes, the model of an industrial plant has been interfaced to a cloud-based process automation system via OPC UA. The process logic - originally implemented solely onto a single PLC - is decentralized and split up between the cloud and the edge (PLC / OPC UA) level.

The core technologies utilized include Amazon Web Services (AWS) IoT as a cloud service and AWS Greengrass on the fog layer for redundancy. The main objective of this case study is the evaluation of the benefits and downsides of such a decentralized architecture in conjunction with new, holistic programming paradigms in the context of Industrie 4.0.

Part of the secondary objectives is the integration of a digital twin in the style of AutomationML. Additional attention is given to the aspects of redundancy and availability as well as deployment of software changes.

The project can be divided in three parts:

1. Setting up an OPC UA-server as an interface to the machinery
2. Setting up an AWS Greengrass instance on fog-level
3. Implementing the process logic

endenumerate

Inhaltsverzeichnis

Nomenklatur **V**

1 Einleitung und Motivation **1**

2 Grundlagen **2**
2.1 Speicherprogrammierbare-Steuerung und Linux – Revolution Pi 2
2.1.1 Kunbus RevolutionPi 2
2.1.2 Zugriff auf IO-Module 3
2.2 Echtzeit und Multithreading unter Linux – preemptRT und posix . . . 3
2.2.1 preemptRT 4
2.2.2 posix 4
2.3 OPC-UA und open62541 4
2.3.1 OPC UA 4
2.3.2 open62541 5

3 Systemkonzept **7**
3.1 Anbindung der IO an den OPC-Server 7
3.2 Integration des OPC-Servers in das System 7

4 Implementierung **8**

5 Test des OPC-Servers im Gesamtsystem **12**

6 Zusammenfassung und Ausblick **13**
6.1 Zusammenfassung 13
6.2 Ausblick 13

Literatur- und Quellenverzeichnis **14**

Bilderverzeichnis **15**

Tabellenverzeichnis **16**

Verzeichnis der Listings **16**

Anhang **17**

Nomenklatur

TODO .....

1 Einleitung und Motivation

Ziel dieses Projektes ist die Integration eines OPC-Servers mit einer auf Linux basierenden speicherprogrammierbaren Steuerung (SPS). Angeschlossen an diese SPS ist jeweils ein digitales Ein-/ bzw. Ausgabemodul. Die von diesen bereitgestellten Ein-/ bzw. Ausgänge (IO) sollen in der Datenstruktur des OPC-Servers abgebildet und über diesen für OPC-Clients les-/ und schreibar sein. Weiterhin sollen einige Funktionen zur Überwachung und Steuerung der an die SPS angeschlossenen Aktoren und Sensoren direkt im OPC-Server implementiert werden. Hiermit stellt dieses Projekt eine der Grundlagen für ein übergeordnetes Projekt, die cloudbasierte Steuerung eines miniaturisierten Produktions-Systems, dar.

Der hier verwendete OPC-Server ist Teil des sog. open62541 Projekts. Er ist in C geschrieben und implementiert bereits einen großen Teil der im OPC-UA-Standard spezifizierten Funktionen. Als SPS findet ein Revolution Pi 3 der Firma Kunbus Verwendung. Dieser integriert ein sog. Compute Module der Raspberry Pi Foundation in ein industrietaugliches Gehäuse und erlaubt die Erweiterung mittels IO- oder Gateway-Modulen. Über diese erfolgt die Kommunikation mit weiteren Komponenten der Automatisierungstechnik.

Motiviert ist dieses Projekt durch die Beobachtung, dass die Verbreitung offener Standards sowie freier Software auch in der Automatisierungstechnik zunimmt. Linux ist ein freies Betriebssystem, OPC-UA ein offen zugänglicher, aktiv gepflegter und weit verbreiteter Standard. Der Raspberry Pi findet sowohl bei Hobby-Anwendern als auch in den Bereichen Forschung und Entwicklung sowie bei industriellen Anwendern Verwendung. Dieses Projekt stellt somit eine für unterschiedliche Anwender interessante Entwicklung dar.

Im Anschluss an diese einleitende Übersicht im Abschnitt 1 folgt die Darstellung der wichtigsten Grundlagen in Abschnitt 2. Aufbauend auf diesen Grundlagen folgt die konzeptuelle Ausarbeitung im Abschnitt 3. Die Umsetzung wird im Abschnitt 4 erläutert. Die Leistungsfähigkeit der Implementierung wird in Abschnitt 5 untersucht. Eine Zusammenfassung und ein Ausblick schließen die Arbeit in Abschnitt 6 ab. Eventuell noch benötigte Anhänge finden sich in den Anhängen [...] bis [...].

2 Grundlagen

2.1 Speicherprogrammierbare-Steuerung und Linux – Revolution Pi

2.1.1 Kunbus RevolutionPi

Der RevolutionPi 3 ist eine speicherprogrammierbare Steuerung (SPS) des Herstellers Kunbus GmbH. Kern dieser SPS ist das von der Raspberry Pi Foundation entwickelte und vertriebene Raspberry Pi Compute Module 3. Dieses integriert ein Broadcom BCM2837 System-on-Chip (SoC) mit vier 1,2GHz Prozessorkernen, 1GB RAM, 4GB eMMC Anwendungsspeicher und sonstige Peripherie in ein Modul im DDR2-SODIMM Formfaktor. Diese Spezifikationen sind weitgehend identisch zu denen des ausgesprochen populären Raspberry Pi 3. Der Revolution Pi profitiert daher von dem gleichen großen Angebot an Software und Unterstützung wie der Raspberry Pi, ergänzt dessen Hardware jedoch um eine 24V Spannungsversorgung, die Möglichkeit der Erweiterung durch mehrere industrietaugliche Ein-/ Ausgabemodule und Gateways sowie ein Gehäuse zur Montage auf einer DIN-Schiene.

- Prozessor: BCM2837
- Taktfrequenz 1,2 GHz
- Anzahl Prozessorkerne: 4
- Arbeitsspeicher: 1 GByte
- eMMC Flash Speicher: 4 GByte
- Betriebssystem: Angepasstes Raspbian mit RT-Patch
- RTC mit 24h Pufferung über wartungsfreien Kondensator
- Treiber / API: Treiber schreibt zyklisch Prozessdaten in ein Prozessabbild, Zugriff auf Prozessabbild über Linux-Filesystem als API zu Fremdsoftware.
- Kommunikationsanschlüsse: 2 x USB 2.0 A (je 500 mA belastbar), 1 x Micro-USB, HDMI, Ethernet (RJ45) 10/100 Mbit/s
- Stromversorgung: min. 10,7 V, max. 28,8 V, maximal 10 Watt
- Zulässige Umgebungstemperatur: -40 bis +55 C
- Gehäuseabmessungen: (HxBxL) 96 mm x 22,5 mm x 110,5 mm (ohne gesteckte Stecker)
- ESD Schutz: 4 kV / 8 kV gemäß EN61131-2 und IEC 61000-6-2
- Surge / Burst Prüfungen: gemäß EN61131-2 und IEC 61000-6-2 eingekoppelt auf Versorgungsspannung, Ethernet und IO-Leitungen
- EMI Prüfungen: gemäß EN61131-2 und IEC 61000-6-2

Kunbus bietet eine Auswahl an IO- und Gateway-Modulen zur Erweiterung des Revolution Pi an. Gateways dienen der Kommunikation mit Systemen oder Komponenten der Automatisierungstechnik über Protokolle wie PROFIBUS oder EtherCAT. IO-Module

erlauben die Überwachung und Steuerung von digitalen oder analogen Ein- und Ausgängen.

2.1.2 Zugriff auf IO-Module

Der Zugriff auf die Ein- und Ausgänge der IO-Module erfolgt über ein Prozessabbild und einen hierfür von Kunbus bereitgestellten Treiber, genannt piControl. Dieser aktualisiert das Prozessabbild zyklisch. Die angestrebte Zykluszeit beträgt 5ms, kann jedoch je nach Anzahl der angeschlossenen Module auch größer sein. Kunbus garantiert bei drei IO-Modulen und zwei Gateway-Modulen eine Zykluszeit von 10 ms. Jedes der IO-Module stellt ein eigenständiges eingebettetes System dar. Es verfügt über einen Microcontroller, welcher die IOs bereitstellt und über einen RS485-Bus mit dem Revolution Pi kommuniziert.

Lizenz: GPL

Listing 1: Setzen der Scheduler-Priorität auf SCHED_FIFO in revpi_common.c

```
226 param.sched_priority = ktprio->prio;
227 ret = sched_setscheduler(child, SCHED_FIFO,
228         &param);
```

2.2 Echtzeit und Multithreading unter Linux – preemptRT und posix

Der Linux-Kernel verfügt über mehrere unterschiedliche Preemtion-Modelle:

- No Forced Preemption (server): Ausgelegt auf maximal möglichen Durchsatz, lediglich Interrupts und System-Call-Returns bewirken Präemption.
- Voluntary Kernel Preemption (Desktop): Neben den implizit bevorrechtigten Interrupts und System-Call-Returns gibt es in diesem Modell weitere Abschnitte des Kernels in welchen Preämption explizit gestattet ist.
- Preemptible Kernel (Low-Latency Desktop): In diesem Modell ist der gesamte Kernel, mit Ausnahme sog. kritischer Abschnitte präemptible. Nach jedem kritischen Abschnitt gibt es einen impliziten Präemptions-Punkt.
- Preemptible Kernel (Basic RT): Dieses Modell ist dem zuvor genannten sehr ähnlich, hier sind jedoch alle Interrupt-Handler als eigenständige Threads ausgeführt.
- Fully Preemptible Kernel (RT): Wie auch bei den beiden zuvor genannten Modellen ist hier der gesamte Kernel präemtible, die Anzahl und Dauer der nichtpräemtiblen kritischen Abschnitte ist auf ein notwendiges Minimum beschränkt. Alle Interrupt-Handler sind als eigenständige Threads ausgeführt, Spinlocks durch Sleeping-Spinlocks und Mutexe durch sog. RT-Mutexe ersetzt.

Lediglich mit dem vollständig präemtiblen Kernel kann Echtzeit-Verhalten realisiert werden.

Spinlock und Mutexe sov die RT-Variante dieser e ren!

2.2.1 preemptRT

Das dem PREEMPT RT Kernel zugrunde liegende Prinzip lässt sich in einer einfachen Regel ausdrücken: Nur Code, welcher absolut nicht-präemtible sein darf, ist es gestattet nicht-präemtible zu sein. Das erklärte Ziel des PREEMPT_RT Patches ist es folglich, die Menge des nicht-präemtiblen Codes im Linux-Kernel auf das absolut notwendige Minimum zu reduzieren.

Dies wird durch Verwendung folgender Mechanismen erreicht:

- Hochauflösende Timer
- Sleeping Spinlocks
- Threaded Interrupt Handlers
- rt_mutex
- RCU

2.2.2 posix

Ist posix hier wirklich relevant? Debian bzw. Raspbian sind weitgehend posix kompatibel, aber wird es hier genutzt? -> JA, open62541 nutzt pthread.h piControl nutzt kthread.h, und semaphore.h

2.3 OPC-UA und open62541

2.3.1 OPC UA

Open Platform Communications (OPC) ist eine Familie von Standards zur herstellerunabhängigen Kommunikation von Maschinen (M2M) in der Automatisierungstechnik. Die sog. OPC Task Force, zu deren Mitgliedern verschiedene große Firmen der Automatisierungsindustrie gehören, veröffentlichte die OPC Specification Version 1.0 im August 1996. Motiviert ist dieser offene Standard durch die Erkenntniss, dass die Anpassung der zahlreichen Herstellerstandards an individuelle Infrastrukturen und Anlagen einen großen Mehraufwand verursachen. Die Wikipedia beschreibt das Anwendungsgebiet für OPC wie folgt:

„OPC wird dort eingesetzt, wo Sensoren, Regler und Steuerungen verschiedener Hersteller ein gemeinsames Netzwerk bilden. Ohne OPC benötigten zwei Geräte zum Datenaustausch genaue Kenntnis über die Kommunikationsmöglichkeiten des Gegenübers. Erweiterungen und Austausch gestalten sich entsprechend schwierig. Mit OPC genügt es, für jedes Gerät genau einmal einen OPC-konformen Treiber zu schreiben. Idealerweise wird dieser bereits vom Hersteller zur Verfügung gestellt. Ein OPC-Treiber lässt sich ohne großen Anpassungsaufwand in beliebig große Steuer- und Überwachungssysteme integrieren.

OPC unterteilt sich in verschiedene Unterstandards, die für den jeweiligen Anwendungsfall unabhängig voneinander implementiert werden können. OPC lässt sich damit verwenden für Echtzeitdaten (Überwachung), Datenarchivierung, Alarm-Meldungen und neuerdings auch direkt zur Steuerung (Befehlsübermittlung).“

OPC basiert in der ursprünglichen Spezifikation auf Microsofts DCOM-Spezifikation. DCOM macht Funktionen und Objekte einer Anwendung anderen Anwendungen im Netzwerk zugänglich. Der OPC-Standard definiert entsprechende DCOM-Objekte um mit anderen OPC-Anwendungen Daten austauschen zu können. Die Verwendung von DCOM bindet Anwender an Betriebssysteme von Microsoft. Die ursprüngliche OPC Spezifikation wird durch die Entwicklung von OPC Unified Architecture (OPC UA) abgelöst. OPC UA setzt auf einem eigenen Kommunikationionsstack auf, die Verwendung von DCOM und damit die Bindung an Microsoft wurden aufgelöst.

Die OPC-UA-Architektur ist eine Service-orientierte Architektur (SOA), deren Struktur aus mehreren Schichten besteht.

Das OPC-Informationsmodell ist nicht mehr nur eine Hierarchie aus Ordnern, Items und Properties. Es ist ein sogenanntes Full-Mesh-Network aus Nodes, mit dem neben den Nutzdaten eines Nodes auch Meta- und Diagnoseinformationen repräsentiert werden. Ein Node ähnelt einem Objekt aus der objektorientierten Programmierung. Ein Node kann Attribute besitzen, die gelesen werden können (Data Access (DA), Historical Data Access (HDA)). Es ist möglich Methoden zu definieren und aufzurufen. Eine Methode besitzt Aufrufargumente und Rückgabewerte. Sie wird durch ein Command aufgerufen. Weiterhin werden Events unterstützt, die versendet werden können (AE (Alarms & Events), DA DataChange), um bestimmte Informationen zwischen Geräten auszutauschen. Ein Event besitzt unter anderem einen Empfangszeitpunkt, eine Nachricht und einen Schweregrad. Die o. g. Nodes werden sowohl für die Nutzdaten als auch alle anderen Arten von Metadaten verwendet. Der damit modellierte OPC-Adressraum beinhaltet nun auch ein Typmodell, mit dem sämtliche Datentypen spezifiziert werden.

2.3.2 open62541

open62541 ist eine offene und freie Implementierung von OPC UA. Die in C geschriebene Bibliothek stellt eine beständig zunehmende Anzahl der im OPC UA Standard definierten Funktionen bereit. Sie kann sowohl zur Erstellung von OPC-Servern als auch -Clients genutzt werden. Ergänzend zu der unter der Mozilla Public License v2.0 lizensierten Bibliothek stellt das open62541 Projekt auch Beispielprogramme unter einer CC0 Lizenz zur Verfügung.

Die Bibliothek eignet sich auch für die Entwicklung auf eingebetteten Systemen und Microcontrollern. Je nach Umfang der gewünschten Funktionen und des OPC Informationsmodells beträgt die Größe einer Server-Binary weniger als 100kb.

Nodes e
klären!
Evtl. ob

Folgende Auswahl an Eigenschaften und Funktionen zeichnet die in dieser Arbeit verwendete Version 0.3 von open62541 aus:

- Kommunikationionsstack
  - OPC UA Binär-Protokoll (HTTP oder SOAP werden gegenwärtig nicht unterstützt)
  - Austauschbare Netzwerk-Schicht, welche die Verwendung eigener Netzwerk-APIs erlaubt.
  - Verschlüsselte Kommunikationion

  - Asynchrone Dienst-Anfragen im Client
- Informationsmodell
  - Unterstützung aller OPC UA Node-Typen, inkl. Methoden
  - Hinzufügen und Entfernen von Nodes und Referenzen zur Laufzeit.
  - Vererbung und Instanziierung von Objekt- und Variablentypen
  - Zugriffskontrolle auch für einzelne Nodes
- Subscriptions
  - Erlaubt die Überwachung (subscriptions / monitoreditems)
  - Sehr geringer Ressourcenbedarf pro überwachtem Wert
- Code-Generierung auf XML-Basis
  - Erlaubt die Erstellung von Datentypen
  - Erlaubt die Generierung des serverseitigen Informationsmodells

Mozilla Public License CC0 Lizenz für Beispiele und Plugins

3 Systemkonzept

Auf Basis der in Abschnitt [...] vorgestellten Möglichkeiten folgt nun die Ausarbeitung eines Konzepts.

3.1 Anbindung der IO an den OPC-Server

3.2 Integration des OPC-Servers in das System

4 Implementierung

revpi.c - 58

Listing 2: Setzen eines phsikalischen, digitalen Ausgangs

```
97  extern void PI_writeSingleIO(char *pszVariableName, bool *bit, bool
        verbose)
98  {
99    int rc;
100   SPIVariable sPiVariable;
101   SPIValue sPIValue;

103   strncpy(sPiVariable.strVarName, pszVariableName, sizeof(sPiVariable.
          strVarName));
104   rc = piControlGetVariableInfo(&sPiVariable);
105   if (rc < 0) {
106     printf("Cannot find variable '%s'\n", pszVariableName);
107     return;
108   }

110     sPIValue.i16uAddress = sPiVariable.i16uAddress;
111     sPIValue.i8uBit = sPiVariable.i8uBit;
112     sPIValue.i8uValue = *bit;
113     rc = piControlSetBitValue(&sPIValue);
114     if (rc < 0)
115       printf("Set bit error %s\n", getWriteError(rc));
116     else if (verbose)
117       printf("Set bit %d on byte at offset %d. Value %d\n", sPIValue.
            i8uBit, sPIValue.i16uAddress,
118               sPIValue.i8uValue);
119 }
```

Listing 3: Methode piControlSetBitValue in piControlIf.c

```
301 /*
        ******************************************************************************
        */
302 /*!
303  * @brief Set Bit Value
304  *
305  * Set the value of one bit in the process image.
306  *
307  * @param[in/out]   Pointer to SPIValue.
308  *
309  * @return 0 or error if negative
310  *
311  ******************************************************************************
        */
312 int piControlSetBitValue(SPIValue *pSpiValue)
313 {
314     piControlOpen();

316     if (PiControlHandle_g < 0)
317   return -ENODEV;

319     pSpiValue->i16uAddress += pSpiValue->i8uBit / 8;
320     pSpiValue->i8uBit %= 8;

322     if (ioctl(PiControlHandle_g, KB_SET_VALUE, pSpiValue) < 0)
```

```
323   return errno;

325     return 0;
326 }
```

Listing 4: Auszug aus der RevolutionPi Programmers Manual

```
KB_SET_VALUE SPIValue *argp
Write one bit or one byte to the process image. Before the call the
    elements i16uAddress and
i8uBit must be set to the address of the value. i8uValue must set to
    the value to write. This call is
more efficient than the usual calls of seek and write because only one
    function call is necessary. If
more than on application are writing bits in one output byte, this call
     is the only safe way to set a
bit without overwriting the other bits because this call is doning a
    read-modify-write-cycle.
The struct SPIValue used by this ioctl is defined as
typedef struct SPIValueStr
{
uint16_t i16uAddress; // Address of the byte in the process image
uint8_t i8uBit; // 0-7 bit position, >= 8 whole byte
uint8_t i8uValue; // Value: 0/1 for bit access, whole byte otherwise
} SPIValue;
```

Listing 5: Methode piControlOpen in piControlIf.c

```
74 /*
       ******************************************************************************
       */
75 /*!
76  * @brief Open Pi Control Interface
77  *
78  * Initialize the Pi Control Interface
79  *
80  ******************************************************************************
       */
81 void piControlOpen(void)
82 {
83     /* open handle if needed */
84     if (PiControlHandle_g < 0)
85     {
86   PiControlHandle_g = open(PICONTROL_DEVICE, O_RDWR);
87     }
88 }
```

Listing 6: Methode piControlIoctl in piControlMain.c

```
719   /*
          ******************************************************************************
          */
720   /*     I O C T L
                                                                          */
721   /*
          ******************************************************************************
          */
722   static long piControlIoctl(struct file *file, unsigned int prg_nr,
          unsigned long usr_addr) // <-
```

```
723   {
724     int status = -EFAULT;
725     tpiControlInst *priv;
726     int timeout = 10000;  // ms

728     if (prg_nr != KB_CONFIG_SEND && prg_nr != KB_CONFIG_START && !
          isRunning()) {
729       return -EAGAIN;
730     }

732     priv = (tpiControlInst *) file->private_data;

734     if (prg_nr != KB_GET_LAST_MESSAGE) {
735       // clear old message
736       priv->pcErrorMessage[0] = 0;
737     }

739     switch (prg_nr) {

741     [...]

743     piControlMain.c - 865
744     case KB_SET_VALUE:  // <-
745       {
746         SPIValue *pValue = (SPIValue *) usr_addr;

748         if (!isRunning())
749           return -EFAULT;

751         if (pValue->i16uAddress >= KB_PI_LEN) {
752           status = -EFAULT;
753         } else {
754           INT8U i8uValue_l;
755           my_rt_mutex_lock(&piDev_g.lockPI);
756           i8uValue_l = piDev_g.ai8uPI[pValue->i16uAddress];

758           if (pValue->i8uBit >= 8) {
759             i8uValue_l = pValue->i8uValue;
760           } else {
761             if (pValue->i8uValue)
762               i8uValue_l |= (1 << pValue->i8uBit);
763             else
764               i8uValue_l &= ~(1 << pValue->i8uBit);
765           }

767           piDev_g.ai8uPI[pValue->i16uAddress] = i8uValue_l;  // <-
768           rt_mutex_unlock(&piDev_g.lockPI);

770   #ifdef VERBOSE
771           pr_info("piControlIoctl Addr=%u, bit=%u: %02x %02x\n", pValue
                ->i16uAddress, pValue->i8uBit, pValue->i8uValue,
                i8uValue_l);
772   #endif

774           status = 0;
775         }
776       }
777       break;
```

```
779       [...]

781       default:
782         pr_err("Invalid Ioctl");
783         return (-EINVAL);
784         break;

786       }

788       return status;
789     }
```

Listing 7: Definition des Struct spiControlDev in piControlMain.h

```
62  typedef struct spiControlDev {
63    // device driver stuff
64    int init_step;
65    enum revpi_machine machine_type;
66    void *machine;
67    struct cdev cdev; // Char device structure   // <-> #include <linux/
          cdev.h>
68    struct device *dev;                          // <-> #include <linux/
          device.h>
69    struct thermal_zone_device *thermal_zone;

71    // process image stuff
72    INT8U ai8uPI[KB_PI_LEN];
73    INT8U ai8uPIDefault[KB_PI_LEN];
74    struct rt_mutex lockPI;
75    bool stopIO;
76    piDevices *devs;
77    piEntries *ent;
78    piCopylist *cl;
79    piConnectionList *connl;
80    ktime_t tLastOutput1, tLastOutput2;

82    // handle open connections and notification
83    u8 PnAppCon;     // counter of open connections
84    struct list_head listCon;
85    struct rt_mutex lockListCon;

87    struct led_trigger power_red;
88    struct led_trigger a1_green;
89    struct led_trigger a1_red;
90    struct led_trigger a2_green;
91    struct led_trigger a2_red;
92    struct led_trigger a3_green;
93    struct led_trigger a3_red;
94  } tpiControlDev;
```

5 Test des OPC-Servers im Gesamtsystem

6 Zusammenfassung und Ausblick

Der folgende Abschnitt 6.1 fasst die gewonnenen Erkenntnisse und den Stand der Implementierung zusammen. Den Abschluss dieser Arbeit bildet der Ausblick in Abschnitt 6.2.

6.1 Zusammenfassung

6.2 Ausblick

Literatur- und Quellenverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Setzen der Scheduler-Priorität auf SCHED_FIFO in revpi_common.c . 3
2 Setzen eines phsikalischen, digitalen Ausgangs 8
3 Methode piControlSetBitValue in piControlIf.c 8
4 Auszug aus der RevolutionPi Programmers Manual 9
5 Methode piControlOpen in piControlIf.c 9
6 Methode piControlIoctl in piControlMain.c 9
7 Definition des Struct spiControlDev in piControlMain.h 11

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Die Regelungen der geltenden Prüfungsordnung zu Versäumnis, Rücktritt, Täuschung und Ordnungsverstoß habe ich zur Kenntnis genommen.

Diese Arbeit hat in gleicher oder ähnlicher Form keiner Prüfungsbehörde vorgelegen.

Heiligenhaus, den 5. Oktober 2018

Unterschrift