Bachelorarbeit

Messgeräteansteuerung unter Python

Thomas Klima

Sept. 2016

Ausgeführt am Institut für Sensor- und Aktuatorsysteme
der technischen Universität Wien

unter der Anleitung von

Dipl.-Ing. Dr.techn. Wilfried Hortschitz

Dipl.-Ing. Dr.techn. Andreas Kainz

am September 6, 2017

durch

Thomas Klima
Matrikelnummer 0425668

Thomas Klima, Technical University Vienna

Abstract of Bachelor's Thesis, submitted 6th September 2017 at the institute of sensor and actuator systems, TU Wien.

The primary purpose of this bachelor's thesis was to evaluate and improve upon the software controlling special laboratory equipment over the GPIB bus. To achieve this goal a novel hardware adapter and software driver was developed for the raspberry pi and linux operating system. In the first two chapters of the thesis an introduction to the involved technologies and the problems with current solutions are presented. Subsequently the Raspberry Pi GPIB Shield is presented, followed by a analysis of the projects ramifications.

Das Ziel dieser Arbeit war, die Software zur Ansteuerung von GPIB-fähigen Messgeräten neu zu Evaluieren und zu verbessern. Um die Nachteile aktuell verfügbarer Adapter zu umgehen wurde ein neuer GPIB-Adapter für Linux, speziell den Raspberry Pi, entwickelt. Die ersten beiden Kapitel bieten eine Einführung in die Thematik und zeigen die Probleme mit bestehenden Adaptern auf. Anschließend wird der entworfene RaspberryPi-GPIB-Shield vorgestellt, sowie aufgenommene Testergebnisse präsentiert. Den Abschluss bildet ein Ausblick auf die weiteren Möglichkeiten welche die Ergebnisse dieser Arbeit bieten.

Contents

1 Einleitung

In vielen technischen Labors finden sich ”veraltete” Messgeräte, etwa der 1980er Jahre, die zwar ausgezeichnete Messeigenschaften bieten, jedoch in einen modernen Messaufbau nicht integrierbar sind da ihnen moderne PC-Schnittstellen fehlen. Diese Schnittstellen ermöglichen Fernbedienung und Messdatenerfassung, üblicherweise über USB oder Ethernet, eine Aufgabe die der damals weit verbreitete GPIB-Bus übernahm. Folgende Grafik stellt einige Möglichkeiten zur Anbindung von Messgeräten an den Raspberry Pi dar:

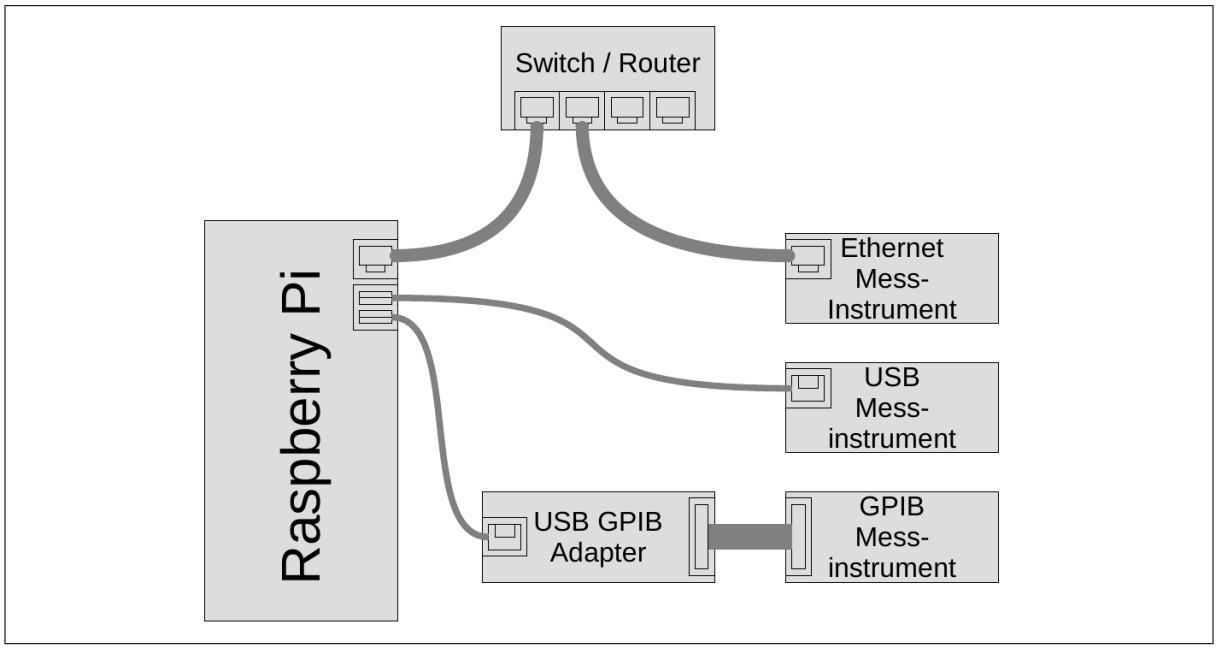

Figure 1: Übersicht üblicher Messgeräteschnittstellen und deren Verbindung zum Raspberry Pi Mikrocomputer.

Nachdem am ISAS (Institut für Sensor- und Aktuatorsysteme [1] der TU Wien) einige Geräte in Verwendung sind die ausschließlich GPIB besitzen wurde eine Lösung zur Anbindung an den an der TU verwendeten Raspberry Pi [2] gesucht. Die Option über USB-Adapter hat jedoch neben dem Preis der Adapter den Nachteil, dass man zumeist auf proprietäre Software (u.A. NI LabView) und Betriebssysteme (MS Windows) beschränkt ist. Im Rahmen dieser Arbeit wurde eine neue GPIB-Schnittstelle für den Raspberry Pi sowie dazu gehörende Treibersoftware entwickelt, welche geringe Kosten aufweist und eine freie, offen zugängliche Alternative darstellt.

In den folgenden Kapiteln werden zuerst die verwendeten Technologien und das Software-Umfeld dargestellt, sowie die dadurch erwachsenen Anforderungen dargelegt. Im Anschluss daran wird das Raspberry Pi GPIB Shield und dessen Treiber für den Linux- Kernel beschrieben und die selbstauferlegte Verpflichtung zu open-source und -hardware sowie deren Folgen besprochen.

1.1 Die GPIB- Schnittstelle

Der GPIB-Bus geht auf eine Entwicklung von Hewlett-Packard in den 1960er Jahren zurück, den HP-Interface-Bus (HP-IB). Das IEEE standardisierte die Schnittstelle mit dem Standards IEEE-488, später erweitert durch IEEE-488.1 und IEEE-488.2, das IEC führt den Standard unter IEC-625.
Der GPIB-Bus wurde bis in die 1990er Jahre in Messgeräte-Neuentwicklungen einbezogen, dann jedoch durch USB und Ethernet verdrängt.

> IEC-625-Bus ist die internationale Normbezeichnung für einen externen parallelen Datenbus, der vorrangig zur Verbindung von Messgeräten und Peripheriegeräten wie Plottern und Druckern mit einem Computer eingesetzt wird, wobei bis zu 15 Geräte angeschlossen werden können. Die maximale Geschwindigkeit der Standardausführung beträgt 1 MByte/s.
> Quelle:Wikipedia [21]

Standardisiert wurden anfangs die Hardware-Schnittstelle und die grundlegende Datenübertragung (Signalnamen, Handshake), später auch Kommandos und Datenübertragungsprotokolle.

1.2 Der Raspberry Pi Einplatinenrechner

Der Raspberry Pi ist ein weit verbreiteter, kostengünstiger Einplatinenrechner der britischen Raspberry Pi Foundation. Das Projekt zielte ursprünglich darauf ab eine günstige Plattform für Bildungszwecke zu bieten, fand jedoch bald Verwendung in vielfältigen Bereichen wie zum Beispiel home-automation, prototyping, embedded systems und Miniatur-PCs. Die Grundausstattung des ersten Raspberry Pi (RPi 1 Model A) bot als Schnittstellen neben USB und Ethernet auch mehrere general-purpose IO-Ports (GPIO) (mit 3.3 V Logikpegel) an. Um eigene Schaltungen an den Raspberry Pi anzuschließen hat es sich bewährt diese auf den GPIO Header aufzustecken, man spricht dann von, wie auch bei Arduino üblich, Shields oder Hats. Ein typischer Messplatz in einem Labor kann somit im Wesentlichen aus einem Raspberry Pi mit HDMI-fähigem Monitor, USB-Tastatur/Maus und Netzwerkverbindung erstellt werden. Die Rechenleistung liegt je nach Modell zwischen ein und vier CPU-Kernen mit bis zu 1.2 GHz und bis zu 1 GB Hauptspeicher.

1.3 Die Programmiersprache Python

Als Beschreibung der Programmiersprache Python findet man auf python.de:

> Python ist eine objektorientierte Skriptsprache, die Anfang der 1990er-Jahre von Guido van Rossum am Centrum voor Wiskunde en Informatica in Amsterdam entwickelt wurde und heute auf einer Vielzahl von Betriebssystem-Plattformen (Unix/Linux, Windows, MacOS, etc.) verfügbar ist. Ihre leicht lesbare Syntax und umfangreiche Standard-Bibliothek ("Batteries included") sowie eine Vielzahl von Erweiterungen aus den verschiedensten Bereichen (GUI, Netzwerke, Datenbanken, Graphik, 3D, Audio, Video, Web, GIS, Numerik, Spiele, etc.) haben sie

zum Mittel der Wahl bei vielen Open-Source-Projekten und in namhaften Unternehmen und Organisationen gemacht.
Quelle:python.de [6]

Gerade im Bildungssektor erfreut sich Python zunehmender Beliebtheit, da es leicht zugänglich ist, eine lebendige community besitzt und die Fähigkeit hat als "glue-language" für übergeordnete Logik zu wirken [7]. Gerade die letzte Eigenschaft ist in einem Umfeld stark unterschiedlicher und voneinander getrennt entwickelter Software von Vorteil. Die in dieser Arbeit erstellte Soft- und Hardware ist zur Verwendung mit der Software GPIB-USBTMC-Serial-Connector [3] vorgesehen, welche ebenfalls am ISAS entwickelt wurde.

1.4 Die linux-gpib Bibliothek

Das linux-gpib Projekt bietet eine offene Schnittstelle zu gängigen GPIB Adaptern. Es umfasst neben Hardwaretreibern und Programmen auch Bibliotheken für einige gängige Programmiersprachen wie Python und C um direkt mit GPIB-Geräten zu kommunizieren. Die Verwendung dieser offenen Schnittstelle bietet neben der prinzipiellen Möglichkeit diese nach eigenen Vorstellungen zu Erweitern, da open-source, auch eine gewisse Zukunftssicherheit die durch proprietäre Systeme wie etwas LabView von National Instruments nicht gegeben sein kann. Im Rahmen dieser Arbeit wurde ein linux-gpib kompatibler Kerneltreiber für Linux erstellt, sowie zum testen desselben die mitgelieferten Programme "ibterm" und "ibtest" neben Python-Skripten verwendet.

1.5 Lösungsansatz und Ausblick

Eine Evaluation der am Markt verfügbaren GPIB-Adapter soll entscheiden ob es Alternativen zu den GPIB-USB-HS Adaptern von National Instruments gibt und falls ja sollten deren Beschaffungssicherheit und zukünftige Software-Unterstützung bewertet werden. Es gibt GPIB Adapter in vielen Ausführungen, am häufigsten USB, PCI(e) und Ethernet. Am weitesten verbreitet ist USB, welches sich für den Raspberry Pi auch anbietet, deshalb beschränkt sich diese Analyse auf USB-Adapter. Eine Recherche ergab folgende am Markt erhältliche GPIB-USB Adapter:

Der Markt teilt sich, vereinfacht dargestellt, somit grob in drei Gruppen:

- kommerzielle Adapter namhafter Hersteller ab 400€ (NI, LeCroy, Adlink)
- Prologix und Agilent/Keysight 82357B Clones bis 200€
- Eigenbauten unter 100€.

Einen Hinweis auf die Notwendigkeit dieses Projekts zeigt sich in der Unterstützung der verbreiteten USB-GPIB-HS Adapter von National Instruments, da diese mit dem Betriebssystem des RPi, raspbian [8] nur unzureichend kompatibel sind. Die von NI gelieferten Treiber gibt es zum Zeitpunkt dieser Analyse (Sommer 2016) als Archiv (ni4882-2.9.1f0.tar.gz Release date: 07-17-2012) oder für rpm-basierte Distributionen wie RedHat und Suse als .iso

Bezeichnung	Preis ca.	Open Hardware	bes. Merkmale
NI-GPIB-USB-HS	€850 (ni.com)	nein	8MBps, NI-488.2 kompatibel
Prologix GPIB-USB	$149 (Sparkfun)	nein	-
Agipibi [10] GPIBerry [11] Galvant GPIBUSB [12]	€50 (Eigenbau)	ja	Interface zu Arduino SN75160/161 Treiber für ARM, AVR und PIC
Adlink USB-3488A	€350 (Mouser)	nein	-
LeCroy USB2-GPIB	€1000 (Mouser)	nein	-
Wisamic USB Interface	€114 (Amazon)	nein	rebrand Agilent

Table 1: Übersicht kommerzieller GPIB-USB Adapter

(NI4882-3.2.0f0.iso Release date: 08-01-2014). Manuelles entpacken des rpm ist möglich mittels rpm2cpio, das mitgelieferte tool updateNIdrivers schlägt jedoch fehl.

Nachdem die Kosten des GPIB-USB Adapters jene des Raspberry Pi Arbeitsplatzrechners deutlich übersteigen und zusätzlich noch problematische Software-Abhängigkeiten zugekauft werden, liegt es nahe alternative Lösungen zu suchen.
Nachdem die nötige Software durch linux-gpib schon zum Teil vorhanden, sowie dokumentiert und von Außenstehenden erweiterbar ist, bleibt nur die Frage nach der zu verwendenden Hardware. Sucht man jedoch ein Interface unter 100€, was in etwa dem Werte des Raspberry Pi samt Peripherie entspricht, findet man zwar einige Projekte von Privatpersonen, diese sind jedoch meist mit einigem Aufwand (Beschaffung und Aufbau) verbunden. Im Rahmen dieser Arbeit wird eine Schnittstelle vorgestellt, die im einfachsten Fall auf eine reine Verdrahtung Raspberry Pi zu GPIB reduziert werden kann. Die für die GPIB-Kommunikation notwendige Logik wird vollständig in den linux-gpib Treiber ausgelagert, der frei verfügbar im Internet zu finden sein wird.

2 Aufgabenstellung und detaillierte Anforderungen

In diesem Kapitel werden nach Darstellung der vorhandenen Hardware (Raspberry Pi) einige Details des GPIB-Busses beleuchtet um aufzuzeigen welche Anforderungen gestellt werden sowie vorhandene Lösungen analysiert.

2.1 Der Raspberry Pi und dessen general purpose IO (GPIO)

General-purpose IO, oder GPIO, sind Anschlüsse eines Mikrocontrollers, welche je nach Konfiguration von der Software aus gesteuert (Logik-) Spannungspegel ausgeben oder einlesen

können. Der verwendete Raspberry Pi 2 Model B+ verfügt, wie der Raspberry Pi 3, über folgende nach außen geführte GPIO pins:

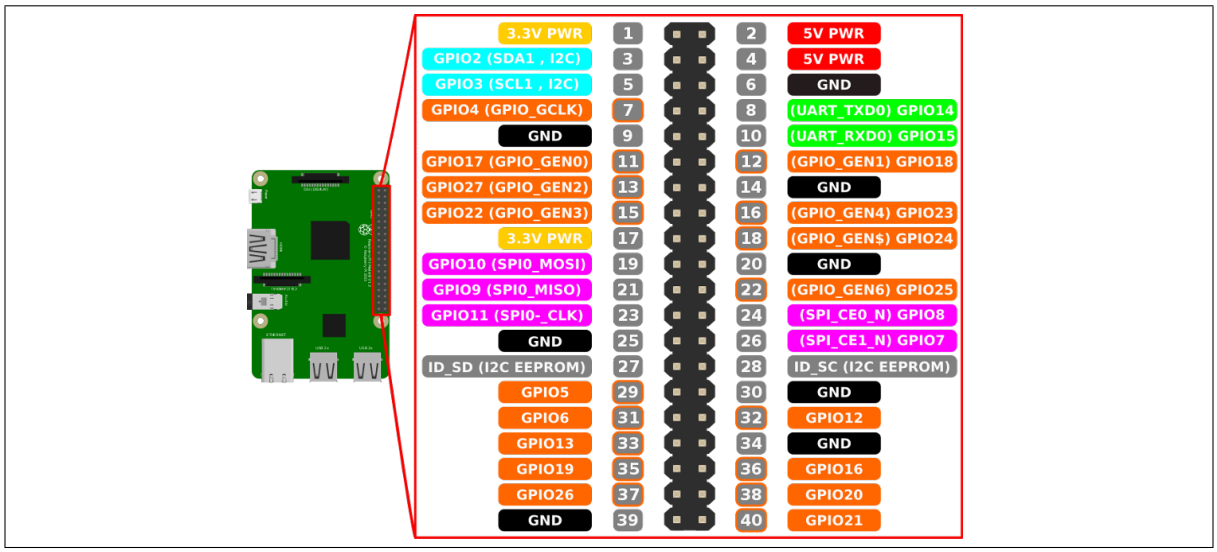

Figure 2: Anordnung der GPIO beim RaspberryPi 2 und 3 [14]

2.1.1 Der GPIO-Header

Diese doppelte Stiftleiste ermöglicht einfachen Zugang zu den GPIO sowie internen Schnittstellen wie zum Beispiel seriellen Schnittstellen (USART), I2C oder SPI. Weiters ist sie als Verbindung für Erweiterungsplatinen, sogenannten shields, vorgesehen.

Der Raspberry Pi 1 bietet im Gegensatz zu den späteren Versionen folgende Pinbelegung:

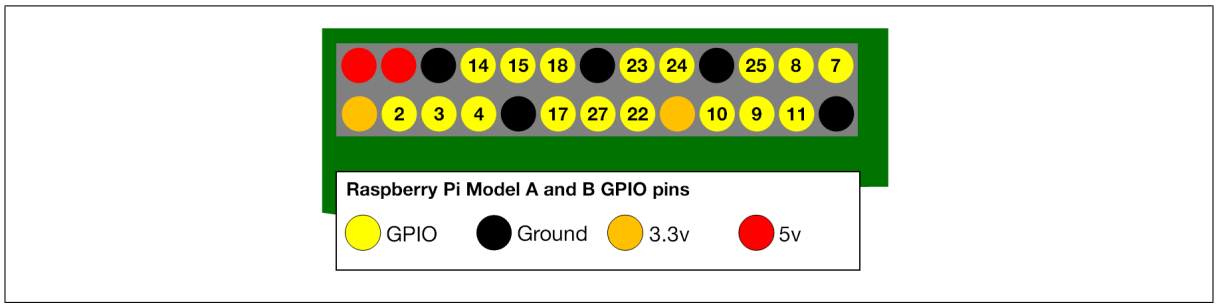

Figure 3: Anordnung der GPIO beim RaspberryPi 1 [15]

2.1.2 Das GPIO-Interface des Linux Kernels

Der Linux Kernel ist als Verwalter aller Treiber die grundlegende Schnittstelle zur Hardware. Der zu entwerfende linux-gpib Treiber wird als reguläres Kernelmodul erstellt, sodass er mit modprobe geladen und mit rmmod wieder entfernt werden kann. Dazu muss er der API (application programming interface) des Kernels folgen, sowie um die general purpose-IO ansprechen zu können das GPIO-Interface linux/gpio.h nutzen. Innerhalb des Kernels wird jedem IO-Pin eine Nummer zugewiesen, welche aus dem DeviceTree (einer Konfigurationsdatei welche die Hardware beschreibt) ausgelesen wird. Nachdem der SoC (System on Chip)

des RaspberryPi von mitgelieferten Kernels direkt unterstützt wird, sind die Pin-Definitionen bereits vollständig vorhanden, es bleiben folgende Initialisierungsschritte:

- Deklarieren eines struct gpio, welches die benötigten GPIO-Pins mitsamt deren Anfangszustand und Namen enthält.
- gpio_request_array fordert die Nutzung der Pins an und prüft deren Verfügbarkeit.

Anschließend kann jeder Pin mit gpio_direction_output(), bzw. -input() gesetzt und mit gpio_get_value() abgefragt werden. Weiters ist es möglich jedem GPIO-Pin einen Pin-change Interrupt zuzuweisen, indem man die durch gpio_to_irq() erhaltene Interrupt-Nummer request_irq() übergibt. Der Vollständigkeit halber sei auch erwähnt, dass es um die GPIO aus dem Userspace heraus anzusprechen auch Dateien im sysfs unter /sys/class/gpio gibt.

2.2 Der GPIB-Messgerätebus im Detail

Im Folgenden werden die elektrischen und mechanischen Spezifikationen der verschiedenen GPIB Standards dargelegt.

2.2.1 Allgemeines

Der GPIB-Bus ermöglicht das Verbinden von bis zu 15 Teilnehmern mit einer maximalen Kabellänge von 4 m und einer Gesamtausdehnung von 10 m. Jedes Bussegment besitzt einen Master, der Datenaustausch erfolgt von diesem ausgehend über einen festgelegten Ablauf (Handshake).

2.2.2 Mechanischer Aufbau

Obwohl in IEC-625 ein alternativer Stecker (D-Sub 25) erlaubt wurde, hat sich der 24-polige Centronics-Stecker (auch: “Micro Ribbon Connector”) durchgesetzt. Dieser ist zumeist zweiseitig ausgeführt um ein Stapeln der Stecker zur Bildung des Busses zu ermöglichen.

2.2.3 Elektrische Spezifikationen

Verwendet wird ein 24-poliger Centronics Stecker der, zumeist stapelbar, folgende 16 Signale trägt [13]:

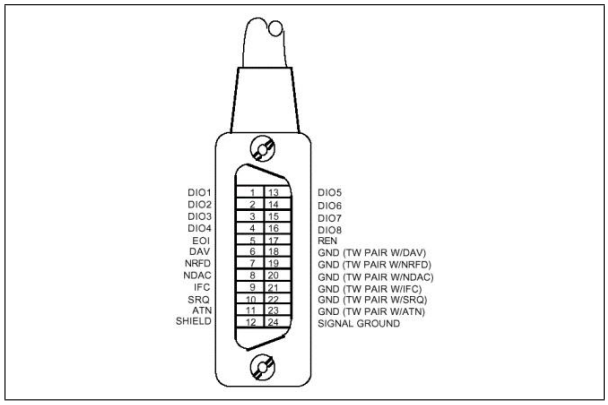

Figure 4: Steckerbelegung GPIB-Connector (National Instruments)

- Data Lines: D1..D8.
- Handshake Lines: DAV, NRFD, NDAC.
- Interface Management Lines: ATN,IFC,REN,SRQ und EOI.

Alle Signalleitungen haben 5 V-TTL Pegel und erwarten eine Busterminierung von 6.2 kΩ gegenüber Masse und 3 kΩ zu 5 V. NRFD, NDAC und SRQ sind OpenCollector-Ausgänge, die übrigen Leitungen haben Push-Pull Ausgänge.

2.2.4 Logischer Aufbau

Um bei Parallelschaltung die Busarbitrierung zu regeln, gibt es immer einen primären Master, der den angeschlossenen Geräten die Sendeerlaubnis (talk enable) erteilt und aktiv Daten ausliest.
Der Standard IEEE-488 definiert mehrere Funktionen (“capabilities”) wie etwa:

Function	Abbrevation	Description/Example
Basic Talker	T	
Extended Talker	TE	
Basic Listener	L	L4 - Unlistens if talk address received
Service Request	SR	SR0 - no service request capability SR1 - complete
Remote-Local	RL	RL0 - no local lockout
Parallel Poll	PP	PP0 - does not respond to Parallel Poll
Controller	C	C0 - no controller function

Table 2: Auszug GPIB Capabilities, Quelle: Wikipedia

Es muss jedoch nicht jedes GPIB-Gerät alle capabilities unterstützen. Das zum testen verwendete “DMM196” von Keithley bietet zum Beispiel folgende Funktionen: SH1, AH1, T6, TE0, LE0, SR1, RL1, PP0, DC1, DT1, C0, E1. Eine weitere interessante Funktion wäre das

serielle bzw. parallele Polling mehrerer Geräte, welches jedoch vom erstellten Treiber (noch) nicht unterstützt wird.
Der Standard IEEE 488.2-1987 erweiterte die capabilities und definierte notwendige sowie optionale Klassen von capabilities.

Description	Control Sequence	Compliance
Send ATN-true commands	SEND COMMAND	Mandatory
Set address to send data	SEND SETUP	Mandatory
Send ATN-false data	SEND DATA BYTES	Mandatory
Send a program message	SEND	Mandatory
Set address to receive data	RECEIVE SETUP	Mandatory
Receive ATN-false data	RECEIVE RESPONSE MESSAGE	Mandatory
Receive a response message	RECEIVE	Mandatory
Pulse IFC line	SEND IFC	Mandatory
Place devices in DCAS	DEVICE CLEAR	Mandatory
Place devices in local state	ENABLE LOCAL CONTROLS	Mandatory
Place devices in remote state	ENABLE REMOTE	Mandatory
Place devices in remote with local lockout state	SET RWLS	Mandatory
Place devices in local lockout state	SEND LLO	Mandatory
Read IEEE 488.1 status byte	READ STATUS BYTE	Mandatory
Send group execution trigger (GET) message	TRIGGER	Mandatory
Give control to another device	PASS CONTROL	Optional
Conduct a parallel poll	PERFORM PARALLEL POLL	Optional
Configure devices' parallel poll responses	PARALLEL POLL CONFIGURE	Optional
Disable devices' parallel poll capability	PARALLEL POLL UNCONFIGURE	Optional

Figure 5: IEEE-488.2 capabilities Übersicht (ni.com)

Weiters wurden Befehle standardisiert, welche vormals oft in der Form von gerätespezifischen Funktionen und zwischen verschiedenen Herstellern inkompatibel, implementiert waren.

Mnemonic	Group	Description
*IDN?	System Data	Identification query
*RST	Internal Operations	Reset
*TST?	Internal Operations	Self-test query
*OPC	Synchronization	Operation complete
*OPC?	Synchronization	Operation complete query
*WAI	Synchronization	Wait to complete
*CLS	Status and Event	Clear status
*ESE	Status and Event	Event status enable
*ESE?	Status and Event	Event status enable query
*ESR?	Status and Event	Event status register query
*SRE	Status and Event	Service request enable
*SRE?	Status and Event	Service request enable query
*STB?	Status and Event	Read status byte query

Figure 6: IEEE-488.2 commands Übersicht (ni.com)

2.3 Analyse verfügbarer Eigenbau-GPIB-Interfaces

Die im Internet frei zugänglichen GPIB-Interfaces aus Tabelle 1 lassen sich wie folgt hinsichtlich ihrer Funktion zusammenfassen:

Name	Schnittstelle	Treiber-IC's	Treiber	link
Galvant GPIB to USB Adapter	USB	nein	ser. Schnittstelle	[12]
GPIBerry	USB	ja	ser. Schnittstelle	[11]
Pic-Plot2	USB	nein	ser. Schnittstelle	[17]

Table 3: Überblick Eigenbau-GPIB-Interfaces

Die Unterschiede sind hauptsächlich durch die Eigenschaften des eingesetzten Mikrocontrollers und dessen verfügbaren Schnittstellen bestimmt. Die verwendeten 8-bit Mikrocontroller, Atmel AVR und Microchip PIC, weisen oft keine integrierte USB Schnittstelle auf, bedingt durch einen internen Systemtakt von etwa 20 MHz, bedürfen sie externer Wandler wie etwa den FT232.

Der "Galvant GPIB to USB"-Adapter[12] besitzt keinerlei Schnittstellentreiber und verwendet einen Mikrocontroller (ArduinoNano), der die Kommando-Logik übernimmt:

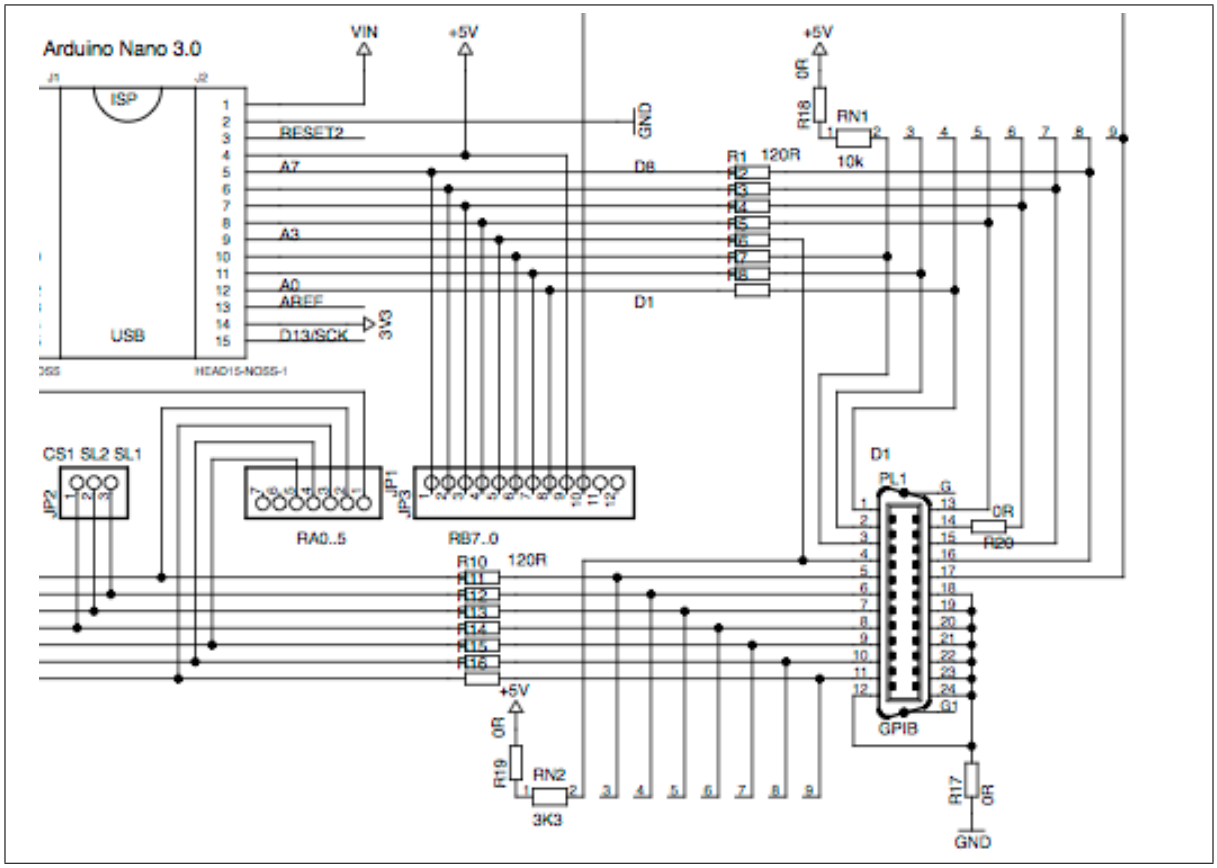

Figure 7: Ausschnitt Schaltplan Galvant GPIB to USB Adapter [12]

Der “GPIBerry” verwendet als Schnittstellentreiber SN75160/161, entspricht ansonsten dem “Galvant”-Adapter insofern, als er einen Mikrocontroller mit USB Verbindung einsetzt. Der “Pic-Plot2”-Adapter verbindet den Mikrocontroller direkt mit dem GPIB-Bus, stellt die USB- Verbindung jedoch mittels FT232 her.
Die verwendeten SN75160/161, von Texas Instruments hergestellt, sind GPIB-Bus-Transceiver und beherbergen außer den eigentlichen Treibern auch die notwendigen Widerstände für die Busterminierung. Alle I/Os haben einen Ausgangswiderstand von etwa 30 Ω und können einen Strom von 46 mA pro Kanal treiben. Obwohl das Design aus 1980 stammt, und TI der einzige Lieferant ist, ist der Treiber immer noch verfügbar in PDIP-20 oder SOIC-20 Gehäusen und derzeit (Stand 2016-09) bei mehreren Distributoren (Mouser, Farnell, RS Components) lagernd.

2.4 Die Distribution Raspbian

Der Raspberry Pi wird standardmäßig mit Raspbian, einer Linux-Distribution, ausgeliefert. Eine Distribution ist eine Zusammenstellung aller Software, die ein vollständiges Linux-System darstellt. Genaugenommen ist Linux der Kernel und dessen Treiber, die weiters nötige System-Software für das Betriebssystem, wird vom GNU-Projekt bereitgestellt, weshalb von GNU/Linux gesprochen wird. Raspbian ist die offizielle Linux-Distribution für den Raspberry Pi und ist ein Debian [5] Derivat für die armhf Hardware-Architektur.

2.5 Die linux-gpib userspace Bibliothek

Das Angebot des Linux Kernels bezüglich GPIB-Hardware ist leider auf einige ISA-Karten für PC und eine ZORRO-Karte für den Amiga beschränkt. Linux-gpib ist ein Projekt, das Kerneltreiber für zeitgemäße GPIB-Hardware sowie eine Bibliothek für Anwenderprogramme bietet.

> The Linux GPIB Package is a support package for GPIB (IEEE 488) hardware. The package contains kernel driver modules, and a C user-space library with Guile, Perl, PHP, Python and TCL bindings. The API of the C library is intended to be compatible with National Instrument's GPIB library. The Linux GPIB Package is licensed under the GNU General Public License.[4]

Erhältlich ist linux-gpib als Sourcecode über Sourceforge [4]. Das Paket bietet neben den genannten Bibliotheken und APIs auch noch u.A. Kommandozeilen-Tools um die Schnittstelle einzurichten und mit den Geräten zu kommunizieren. Das Programm gpib-config liest z.B. die aktuelle Konfigurationsdatei ein und lädt die nötigen Kerneltreiber, mit ibtest lassen sich Zeichenfolgen senden und empfangen, sowie u.A. Steuerleitungen setzen. Entgegen der Beschreibung auf der linux-gpib Homepage muss das Paket selbst kompiliert werden. Um dafür eine geeignete Entwicklungsumgebung vorzubereiten mussten noch die Kernel-Quellen (z.B. mit git clone `https://github.com/notro/rpi-source.wiki.git`), python-dev, git, build_essentials, make, libc6-dev-i386 und deren Abhängigkeiten installiert werden.

2.6 Anforderungsanalyse

Nachdem alle Eigenbau-Adapter mit bescheidener Rechenleistung (zum Beispiel Atmel AVR mit 20 MHz) ausgestattet fähig waren die notwendigen GPIB-Signale zu liefern, kann man voraussetzen, dass der Raspberry Pi dafür ebenfalls geeignet ist. Doch obwohl die IO-Ports des Raspberry Pi fähig sind Signale von mehreren Megahertz zu erzeugen [16], wird es jedoch kaum möglich sein auf die Übertragungsgeschwindigkeiten eines ASIC, wie im NI GPIB-USB Adapter verbaut, zu kommen. Die folgenden Anforderungen entstammen hauptsächlich der Anwendung selbst (Raspberry Pi und GPIB-Bus), es wurde jedoch auch Wert darauf gelegt Nachbauten zu erleichtern.

- Busfähigkeit:
  Durch korrekte Terminierung und Einsatz von optionalen Treiberbausteinen soll sichergestellt werden, dass, wie im Standard vorgesehen, mehrere Meter Kabellänge ermöglicht werden.

- Mechanik:
  Da GPIB-Stecker und -Kabel gegenüber dem RaspberryPi dominant groß und schwer sind, ist eine sichere mechanische Verbindung über Schraubverbindung vorzusehen. Weiters sollen auf der Unterseite nur Bauteile mit niedriger Höhe vorzufinden sein, um Kollision mit einem gegebenenfalls darunterliegenden Shield (Erweiterungsboard) oder Raspberry Pi zu vermeiden.

- Externe Abhängigkeiten:
  Um eine erhöhte Nachbausicherheit (ohne vendor lock-in) zu gewähren soll auch ein (gegebenenfalls eingeschränkter) Betrieb ohne Spezialbausteine ermöglicht werden. Auch bei den eingesetzten CAD-Programmen ist zu prüfen, ob die erstellten Dateien auch in Zukunft verarbeitet werden können. Ein aktuelles Beispiel für die Problematik ist das in Bastlerkreisen beliebte Eagle (vormals Cadsoft EAGLE, mit Beschränkungen frei verwendbar), das nun als "Autodesk Eagle 8.0" einen online Account sowie eine monatliche Reaktivierung erfordert.

- Stromversorgung:
  Wenn möglich soll die Schaltung über den GPIO-Header des Raspberry Pi versorgt werden.

- linux-gpib Unterstützung:
  Linux-gpib bietet als einziges Projekt eine Gesamtlösung für den Zugriff auf GPIB, also sollte die zu erstellende Software damit kompatibel sein. Dies führt dazu, dass ein Linux Kerneltreiber geschrieben werden muss, um sich nahtlos in das Projekt einzufügen.

- GPIO-Schnittstelle:
  Der zu erstellende Treiber soll im Kernel auf die GPIO-Schnittstelle zugreifen.

- Kompatibilität:
  Je nach Aktualität des verwendeten Raspbian kann die Version des Linux-Kernels zwischen Version 3.6 und 4.4 (Stand April 2017) liegen. Obwohl das GPIO-Interface in der

Zwischenzeit keine großen Änderungen erfahren hat, ist zu prüfen, ob sich der Treiber auch auf alten Linux-Versionen kompilieren lässt.

- Performance:
  Obwohl keine konkrete Forderung an die Übertragungsgeschwindigkeit besteht, soll doch ein Betrieb mit mehreren Kilobytes pro Sekunde möglich sein ohne die Systemleistung merkbar zu beeinflussen. Die Verwendung von Interrupts und Polling ist abzuwägen.

3 Das Raspberry Pi GPIB Shield

Das folgende Kapitel stellt das entworfene GPIB-Interface vor, und erläutert anhand von Beispielen dessen Anwendung mit Python-Skripten sowie der mit linux-gpib mitgelieferten Diagnosewerkzeuge.

3.1 Überblick

Das Raspberry Pi GPIB Shield besteht hardwareseitig aus einer Adapterplatine, dem Shield, softwareseitig aus einer Reihe von Schichten die den spezifischen Hardwaretreiber von der Applikationsschicht trennen und abstrahieren. Folgende Grafik soll einen groben Überblick geben:

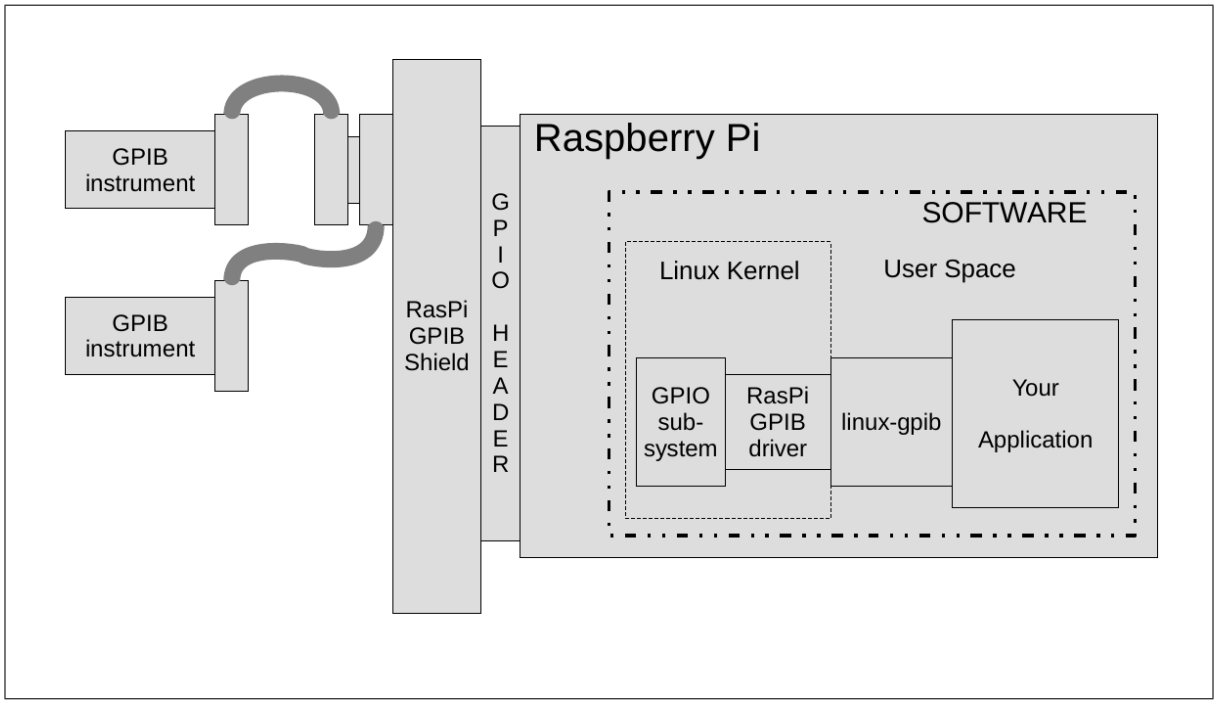

Figure 8: Blockdiagramm Übersicht Raspberry Pi GPIB Shield und Gliederung der Software.

Zu sehen ist die hierarchische Struktur mit dem zentralen Kernel, GPIO Treiber eingeschlossen, der Übergang zu linux-gpib mittels RasPi-GPIB-Treiber, und der Anwendersoftware. Es ist weiters anzumerken, dass durch die von linux-gpib vorgegebene Schnittstelle das entworfene Interface vollständig abstrahiert wird. Die Anwendersoftware muss also nicht auf das verwendete Interface zugeschnitten werden, sofern man gegebenenfalls Einbußen in der Übertragungsrate in Kauf nimmt.

Folgendes Foto zeigt den zweiten Prototypen, gefertigt mit Hardware V0.4:

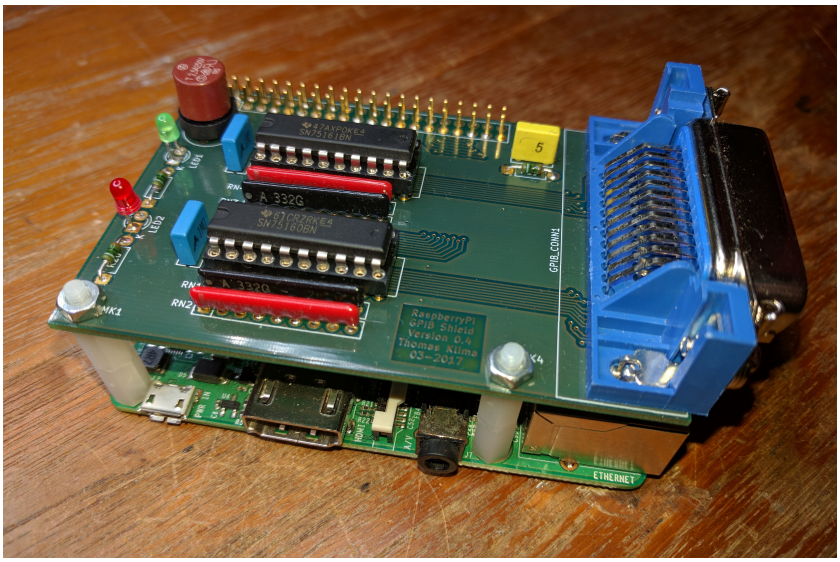

Figure 9: Foto RasPi GPIB Shield V0.4

3.2 Hardware

In diesem Kapitel wird gezeigt, wie die gestellten Anforderungen erfüllt wurden, sowie die entwickelte Schaltung vorgestellt. Folgende Grafik kann zur Orientierung herangezogen werden.

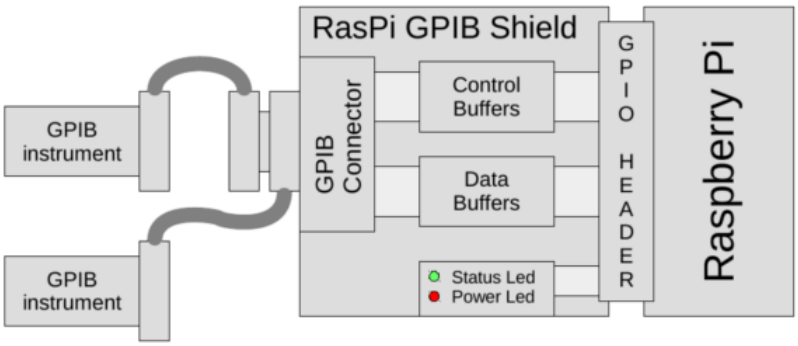

Figure 10: Blockdiagramm Aufbau der Hardware

- Busfähigkeit:
  Um dem Nutzer die Wahlfreiheit bezüglich der Bustreiber zu geben, sind die Widerstandsbrücken RP1 bis RP4 vorgesehen. Sind sie bestückt ist der Betrieb von wenigen Geräten mit kurzen Kabellängen und geringer Busgeschwindigket möglich. Bei mehreren Busteilnehmern oder Kabellängen ab 10 m, empfiehlt es sich die Bustreiber zum Einsatz zu bringen um Übertragungsfehler zu vermeiden. Die Widerstandswerte

von RR1..RR4 ergeben sich direkt aus dem GPIB-Standard [13] und sorgen für eine korrekte Terminierung.

- Mechanik:
  Das entworfene Layout der Platine kann direkt auf den Raspberry Pi aufgesteckt werden mit zusätzlichen Befestigungslöchern für Abstandshalter (M2.5, Länge 20 mm). Hierbei ist zu beachten, dass gewöhnliche Buchsenleisten zu kurze Kontaktstifte haben, es braucht eine Länge von mindestens 10 mm. Zum Anschluss des GPIB-Busses ist ein Footprint für einen 24-poligen Centronics Stecker (gewinkelt) vorgesehen, wie bei RS Components unter der Nummer 239-1162 erhältlich. Es ist weiters zu bedenken, dass der GPIB-Stecker mit 15 mm Höhe das oberste Shield sein muss, oder die Stiftleisten extra verlängert werden müssen.

- externe Abhängigkeiten:
  Die Schaltung lässt sich auch ohne spezielle Bauteile (Bustreiber) betreiben, wobei jedoch die Funktionalität eingeschränkt wird. Um sicherzustellen, dass der erstellte Schaltplan sowie das Layout auch in Zukunft verwendet werden können, wurde die Schaltung mit KiCad ([23]) entworfen.

- Stromversorgung:
  Der RPi liefert über den GPIO-Header eine Spannung von 5 V die lediglich durch das verwendete Netzteil und eine Polyfuse mit 2 A begrenzt ist. Im Betrieb mit Bustreiber-Bausteinen werden diese mit 5 V versorgt, der Verbrauch liegt bei etwa 72 mW max pro Kanal, insgesamt etwa 1 W maximal. Bei einem Kurzschluss auf der Busseite steigt der Stromverbrauch auf 100 mA pro Kanal, eine vorgesehene Sicherung mit 500 mA löst in diesem Fall aus. Ohne Bustreiber muss der GPIO-Port des SoC die Signale bereitstellen, ist hier jedoch auf 16 mA pro IO-Pin, 50 mA gesamt, begrenzt. Die Werte von RP1..RP4 ergeben sich aus diesem maximalen Strom der GPIO $R = \frac{5\,\mathrm{V}}{16\,\mathrm{mA}} = 312\,\Omega$, gewählt wurden $R = 330\,\Omega$.

Weiters wurden noch zwei Leuchtdioden zur Information über Aktivität und Stromversorgung vorgesehen. Folgende Materialkosten ergaben sich für den Prototypen (Preise für Einzelstücke), wobei sich die Referenzen auf den Schaltplan im Anhang beziehen:

3.3 Der RasPi GPIO Treiber

Um das Grundgerüst des Treibers zu erstellen wurde der Quellcode des Treibers lpvo_usb_gpib (erstellt von Marcello Carla der Universität Florenz) untersucht. Dieser dient dazu den usb-gpib-Adapter der Universität von Ljubljana [18] anzusteuern. Beide Projekte sind gut dokumentiert und open-source, womit sich der Ablauf der Treibersoftware gut nachvollziehen lässt.

3.3.1 Überblick

Die Quellen der linux-gpib-Treiber finden sich unter “drivers/gpib/”. Dort wurde der Treiber in einem eigenen Unterverzeichnis “raspi_gpib” inklusive Makefile abgelegt. Einige der beste-

Ref.Nr	Bezeichnung	Preis (EUR)	BestellNr
GPIO_CONN	Raspberry Pi Header	6,82	RS 254-6110
GPIB_CONN	GPIB-Buchse	2,93	RS 239-1162
U1	SN75160	2,82	Farnell 1470448
U2	SN75161	2,82	Farnell 1470452
Rxx	Widerstände	ca. 4	diverse
Cxx	Kondensatoren	ca. 1	diverse
F1	Sicherung, Halter	1,16	Farnell 9515950
Gesamt:		21,55	

Table 4: Stückliste bzw. Bill of materials (BOM) des Raspi-GPIB-Shields

henden Treiber verwenden den selben GPIB-Controller IC und geben die Funktionsaufrufe direkt an bestehende Bibliotheksfunktionen weiter, wie zum Beispiel

```
1 int agilent_82350b_take_control(gpib_board_t *board,int synchronous) {
2   agilent_82350b_private_t *priv = board->private_data;
3   return tms9914_take_control(board,&priv->tms9914_priv,synchronous);
4 }
```

Listing 1: Beispiel linux-gpib-Treiber Code

Der RasPi-GPIO Treiber ersetzt diese Aufrufe durch Befehle an die GPIO-Ports um die benötigten Signale korrekt einzustellen. Weiters ist eine (stark verkürzte) priv_t Struktur nötig, welche Konfigurations-Optionen wie etwa IRQ-Nummern beinhaltet. Im Folgenden ein Überblick über die implementierten Funktionen:

- Initialisierung: init/exit_module, gpib_attach/detach:
  In init_module() werden mit gpio_request_array die GPIO-pins konfiguriert und mit gpib_register_driver der Treiber im System gemeldet. In attach() wird das struct priv allokiert und der IRQ registriert. Die zugehörigen exit- und detach- Funktionen stellen den Ausgangszustand wieder her.

- Interface Management: interface_clear, take_control, ...
  In take_control() wird die ATN-Leitung gesetzt, womit alle clients in den listen-Zustand versetzt werden und Kommandos gesendet werden können, go_to_standby() gibt ATN wieder frei. interface_clear() setzt die IFC-Leitung, was die clients in den Grundzustand zurückversetzt. remote_enable() setzt die REN-Leitung, um bei angeschlossenen Geräte die lokale Steuerung zu deaktivieren.

- I/O Funktionen: read, write, command
  write() sendet eine Zeichenfolge indem es den GPIB-write-Handshake ausführt. command() sendet ebenfalls eine Zeichenfolge, jedoch mit gesetzter ATN-Leitung, womit die Nachricht als Kommando interpretiert werden muss. read() liest von der GPIB-Schnittstelle unter Zuhilfenahme von read_byte(), das den GPIB-read-Handshake implementiert.

- Status Management: en/disable eos, update_status, primary_address, ...
  enable_eos() aktiviert die Prüfung auf Ende der Übertragung durch CR/LF. primary/sec-

ondary_address() liefert schlicht die jeweilige Adresse retour. update_status() verwaltet das Status-Wort in der private-Struktur.

- Utility Funktionen: allocate_private, t1_delay
  allocate_private() initialisiert die private-Struktur, die diverse Konfigurationseinstellungen enthält. t1_delay() führt eine Verzögerung aus, die der Konfiguration in priv entspricht.

Weiters gibt es noch Hilfsfunktionen welche die Richtung der einzelnen IO's (in-/output, weak-pullup) setzt.

3.3.2 Installation

Die Installation erfolgt über Herunterladen der Sourcen von Sourceforge [4], entpacken, konfigurieren, kompilieren und installieren, zum Beispiel wie folgt:

```
pi@raspberrypi:/home/pi/#  wget <insert\_current\_linux\_gpib\_link\_here>
pi@raspberrypi:/home/pi/# tar xzf linux-gpib-x.y.tar.gz
pi@raspberrypi:/home/pi/# cd linux-gpib-x.y
pi@raspberrypi:/home/pi/linux-gpib-x.y/# ./configure
pi@raspberrypi:/home/pi/linux-gpib-x.y/# make
pi@raspberrypi:/home/pi/linux-gpib-x.y/# sudo make install
```

Listing 2: Installationsschritte für linux-gpib

Nach diesen Schritten befinden sich, unter Anderem, die Kernelmodule unter "/lib/modules/<kernel_version>/
dynamic libraries unter "/usr/local/lib" und die Hilfsprogramme unter "/usr/local/bin". Weiters werden Skripte für udev erstellt.

3.3.3 Konfiguration

Parametrisiert werden die verwendeten Geräte über die systemweite Konfigurationsdatei "/etc/gpib.conf", eine einfache Konfigurationsvariante könnte wie folgt lauten:

```
interface {
  minor = 0
  board_type = "raspi_gpio"
  pad = 0
  master = yes
}
device {
  minor = 0
  name = "Keithley 196 DMM"
  pad = 9
  sad = 96
  eos = 0xa
  set-reos = yes
  set-bin = no
  timeout = T1s
}
device {
  minor = 0
  name = "AWG"
  pad = 7
  sad = 97
  eos = 0xa
  set-reos = yes
  set-bin = no
  timeout = T1s
}
```

Listing 3: gpib.conf Beispiel für raspi_gpio und zwei Geräte

Über die erstellte Konfigurationsdatei können nun die nötigen Links, "/dev/gpibX", mittels dem Kommando `gpib_config` erzeugt werden. Damit ist die Installation und Konfiguration abgeschlossen, und die Geräte können nun zum Beispiel direkt über "ibterm -d9 -s96" (Primäradresse 9, das Multimeter aus obiger gpib.conf) angesprochen werden.

3.3.4 Funktionelle Spezifikation

- linux-gpib Unterstützung:
  Der Treiber ist, abgesehen von der Änderung eines Makefiles, minimal invasiv, lässt sich also unabhängig von der aktuellen linux-gpib Version entwickeln und wird auf längere Sicht kompatibel bleiben sofern sich nicht projektinterne Datenstrukturen wie gpib_interface_t ändern.

- GPIO-Schnittstelle und Kompatibilität:
  Das verwendete GPIO Interface ist seit Jahren stabil und wird vielfältig eingesetzt, es gibt keine Anzeichen, dass es in naher Zukunft entfernt werden könnte. Der Treiber kompilierte unter Linux Kernel 3.16 sowie der aktuellen Version 4.4.

- Performance:
  Die erste Version nutzte intensiv Warteschleifen mit sleep im Mikrosekundenbereich, trotzdem konnte keine messbare Beeinträchtigung festgestellt werden da aktuelle Raspberry Pi mehrere CPU Kerne besitzen.

- GPIB-Features:
  Die grundlegenden Funktionen wie schreiben/lesen, Talk Enable und ähnliche sind unterstützt. Von den fortgeschrittenen Funktionen ist zum Beispiel serial-poll noch nicht implementiert.

3.4 Funktionsüberprüfung des RasPi GPIB Shield

Um die Funktion des Treibers zu testen wurde mit Hilfe der bei linux-gpib inkludierten Tools ”ibtest” und “ibterm” die Kommunikation mit einem Keithley DMM 196 Multimeter hergestellt sowie mit Hilfe eines Logikanalysator der Zustand der Leitungen aufgenommen:

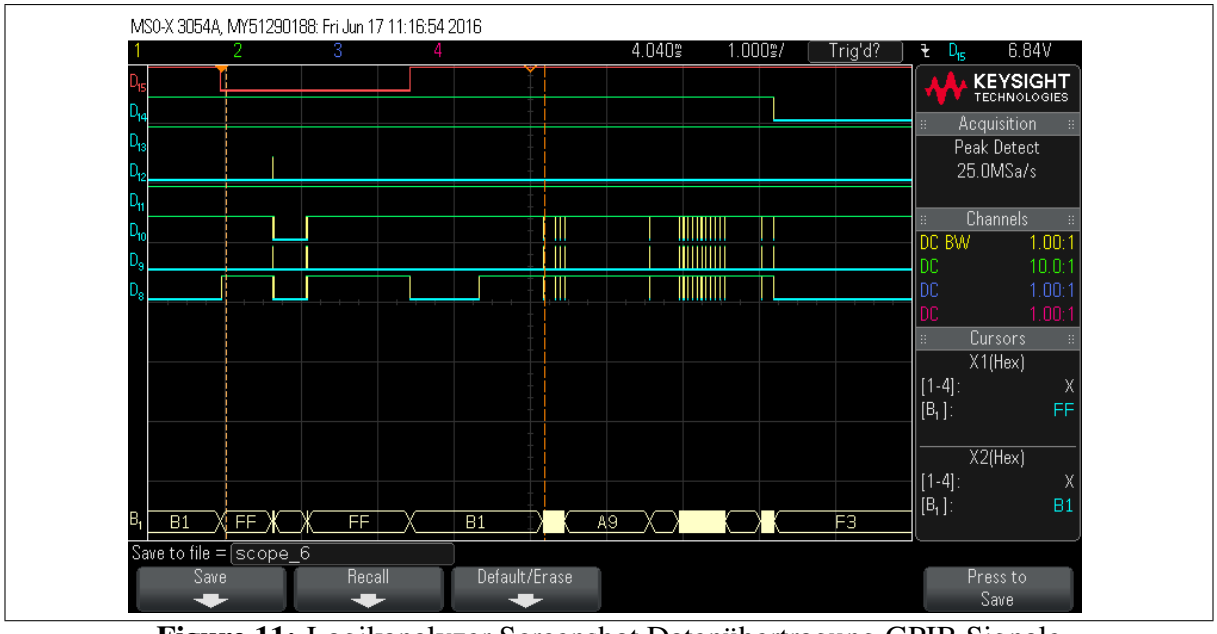

Figure 11: Logikanalyzer Screenshot Datenübertragung GPIB Signale

Zu sehen sind: D15=ATN, D14=EOI, D10=DAV, D9=NRFD, D8=NDAC, D7..D0=Datenleitungen, B1=Wert D0,..,D7

Folgend eine Kommunikation mit einem DMM196, welches den originalen IEEE-488 Standard unterstützt und folglich Kommandos wie "TE0", "DC1" und ähnliche, jedoch nicht "*IDN?" unterstützt. Im Falle, dass das Kommando nicht verstanden wird, sendet das Gerät schlicht den aktuellen Messwert.

```
root@raspberrypi:~# gpib_config
root@raspberrypi:~# ibterm -d9 -s96
Attempting to open /dev/gpib0
pad = 9, sad = 96, timeout = 10, send_eoi = 1, eos_mode = 0x0000
ibterm> <INPUT: ENTER>
NDCV+000.0002E+0
ibterm>*IDN?
NDCV+000.0002E+0
ibterm>F2X
OOHM+9.999999E+9
ibterm> <INPUT: CTRL-D>
ibterm: Done.
root@raspberrypi:~#
```

Listing 4: ibterm Output DMM196

Das ebenfalls mit linux-gpib mitgelieferte Programm "ibtest" erlaubt ein setzten von Steuerleitungen und somit auch zum Beispiel den Wechsel zu einem sekundären GPIB-Controller. Wenn diese Funktionalität auch nicht getestet wurde folgt ein beispielhafter Programmaufruf mit dem DMM196 an GPIB Adresse 9:

```
root@raspberrypi:~# ibtest
Do you wish to open a (d)evice or an interface (b)oard?
        (you probably want to open a device): d
enter primary gpib address for device you wish to open [0-30]: 9
trying to open pad = 9 on /dev/gpib0 ...
You can:
        w(a)it for an event
        write (c)ommand bytes to bus (system controller only)
        send (d)evice clear (device only)
        change remote (e)nable line (system controller only)
        (g)o to standby (release ATN line, system controller only)
        send (i)nterface clear (system controller only)
        ta(k)e control (assert ATN line, system controller only)
        get bus (l)ine status (board only)
        go to local (m)ode
        change end (o)f transmission configuration
        (q)uit
        (r)ead string
        perform (s)erial poll (device only)
        change (t)imeout on io operations
        request ser(v)ice (board only)
        (w)rite data string
: r
enter maximum number of bytes to read [1024]:
trying to read 1024 bytes from device...
received string: 'OOHM+9.999999E+9
Number of bytes read: 18
gpib status is:
ibsta = 0x2100  < END CMPL >
iberr= 0
ibcnt = 18
You can:
        w(a)it for an event
        write (c)ommand bytes to bus (system controller only)
        send (d)evice clear (device only)
        change remote (e)nable line (system controller only)
        (g)o to standby (release ATN line, system controller only)
        send (i)nterface clear (system controller only)
        ta(k)e control (assert ATN line, system controller only)
        get bus (l)ine status (board only)
        go to local (m)ode
        change end (o)f transmission configuration
        (q)uit
        (r)ead string
        perform (s)erial poll (device only)
        change (t)imeout on io operations
        request ser(v)ice (board only)
        (w)rite data string
```

```
:q
```

Listing 5: ibterm Output DMM196

Folgendes Python-Skript zeigt eine Möglichkeit auf linux-gpib zuzugreifen. Nachdem die linux-gpib Bibliothek importiert ist, wird das Kernelmodul neu geladen und eine einzelne Anfrage an das Gerät gestellt:

```
#!/usr/bin/python

import Gpib
import os

os.system("rmmod raspi_gpib")
os.system("insmod drivers/gpib/raspi_gpib/raspi_gpib.ko")
os.system("gpib_config")

d = Gpib.Gpib(0,9)
print("Sende: IDN?")
d.write("IDN?\n")
print("Antwort: "+d.read())
```

Listing 6: linux-gpib Python Test-Skript

Ein weiteres Skript zeigt die Verwendung von IVI, einer weiteren Abstraktionsschicht, welche von linux-gpib zur Verfügung gestellt wird. Der Ablauf erfolgt prinzipiell wie im vorangegangenen Beispiel:

```
#!/usr/bin/python

import ivi
import os

os.system("rmmod raspi_gpib")
os.system("insmod drivers/gpib/raspi_gpib/raspi_gpib.ko")
os.system("gpib_config")

dmm = ivi.interface.linuxgpib.LinuxGpibInstrument("GPIB0::10::INSTR")
print("current reading:")
dmm.write("DSPS?")
print(dmm.read())
```

Listing 7: IVI Python Test-Skript

4 Ausblick

Das Ziel der Arbeit, die Anbindung von Messgeräten mit GPIB-Schnittstelle zu erleichtern, wurde erreicht durch die Entwicklung des RaspberryPi-GPIB-Shields. Diese Neuentwicklung hat jedoch das Potential Auswirkungen auch auf Anwendungen außerhalb des Instituts zu haben, da es die finanzielle Schwelle um GPIB-fähige Geräte zu verwenden deutlich senkt. Es ist nun mit geringem Aufwand möglich geworden, GPIB-fähige Geräte, die aufgrund fehlender USB oder Netzwerkverbindung bereits als veraltet galten, weiter zu verwenden. Ein Beispiel für solche Messgeräte ist das in den Tests verwendete DMM196 von Keithley aus dem Jahr 1986, welches nun einen weiteren Anwendungsbereich erhalten hat. Es ist anzunehmen, dass sich durch die geringen Kosten auch Hobbyisten angesprochen fühlen Altgeräte von Tauschbörsen wie Ebay zu reaktivieren.
Die Lösung der Kommunikation per GPIO Schnittstelle ermöglicht einen Betrieb mit einer breiten Klasse von embedded Linux Systemen, nicht nur des Raspberry Pi. Dieser ist wohl der am weitesten verbreitete Einplatinenrechner, jedoch lassen sich GPIO Ports auch an einem klassischen PC per USB-Adapter nachrüsten mittels Bausteinen wie zum Beispiel dem FT232 von FTDI. Es ist auch anzunehmen, dass durch die Weiterentwicklung im Bereich der 32-bit Mikrocontroller in Bezug auf Speicher und Rechenleistung es auch bei Systemen die früher mit anwendungspezifischer Firmware gebaut wurden, nun zum vermehrten Einsatz von Linux kommt. Ein Beispiel dazu wäre neben vielen IOT (Internet of things) Bausteinen der STM32 von ST-Microelectronics mit einem Preis von etwa 10 EUR, der zwar ein klassischer Mikrocontroller ist, jedoch mit externem SD-RAM Linux-fähig ist.
Im Bereich Software gibt es das für Linux-Entwicklungen nicht unübliche Problem der instabilen APIs, oft hervorgerufen durch die rapiden Änderungen der involvierten Technologien und der flexiblen Entwicklungs-Kultur. Der Linux Kernel hat eine sich selbst auferlegte Regelung, Kernel-APIs stabil zu halten soweit vom Userspace aus ersichtlich, für interne Interfaces gilt dies jedoch nur teilweise. Das GPIO Interface hat in den letzten Jahren einige neue Anforderungen bezüglich atomarem Zugriff, Energieverbrauch und Flexibilität hinzubekommen und reagierte darauf mit dem Übergang zum deskriptor-based GPIO interface. Der vorliegende Sourcecode verwendet die "legacy"-Schnittstelle, um Kompatibilität zu Kernels bestehender Systeme zu gewährleisten, eine Überarbeitung ist jedoch bereits in Planung.
Das linux-gpib Projekt ist in einem stabilen, ruhigen Zustand und scheint allen Interessenten ausreichend Features zur Verfügung zu stellen. Die Aktivität ist nicht sehr hoch, vermutlich da kaum neue GPIB Hardware entwickelt wird, weder Messgeräte noch Schnittstellen. Mittlerweile wurde der Treiber über die linux-gpib Mailingliste veröffentlicht und hat positives Feedback bekommen, ist jedoch noch nicht in die offiziellen Projektsourcen aufgenommen. Die in dieser Arbeit besprochene Version des Treibers entspricht der Erstveröffentlichung, Marcello Carla (University of Florence - Dept. of Physics) verbesserte den Treiber durch einen Interrupt-basierten Handshake.

In Absprache mit meinen Betreuern DI Dr. Wilfried Hortschitz von der Donau Universität Krems und Dipl.-Ing. Andreas Kainz von der TU Wien ist geplant das gesamte Projekt unter einer open-source Lizenz zu veröffentlichen. Hierzu wurden die Teilbereiche Hardware, die

Python-Software und der Treiber als github-Projekte online veröffentlicht. Dort sind sie unter raspi_gpib_shield und raspi_gpib_driver zu finden und bieten Raum für Diskussionen, Änderungswünsche und Austausch von Code.

Ich bedanke mich herzlich bei meinen Betreuern Wilfried Hortschitz und Andreas Kainz, die mir Anregungen und die Möglichkeiten gegeben haben das Projekt durchzuführen. Weiters bedanke ich mich bei meiner Familie, besonders meiner Frau Magdalena welche mir die erforderliche Zeit im Familienleben einberaumt hat sowie meinen Kindern Cornelius, Emilia und Maximilian sowie meinem Vater Josef Klima.

5 Anhang

5.1 Hardware - Layout und Schaltplan

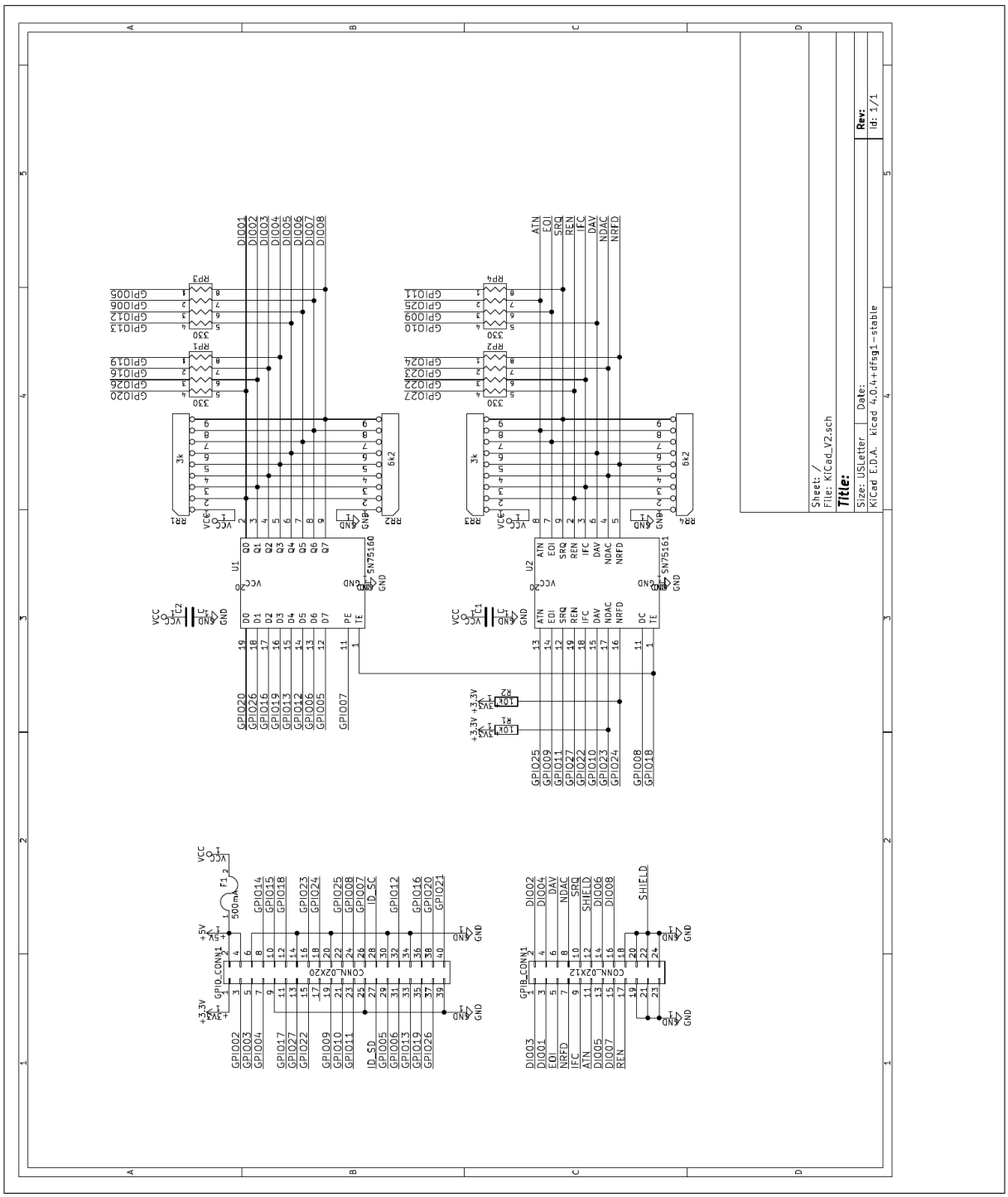

Figure 12: Schaltplan des Raspberry Pi Shields

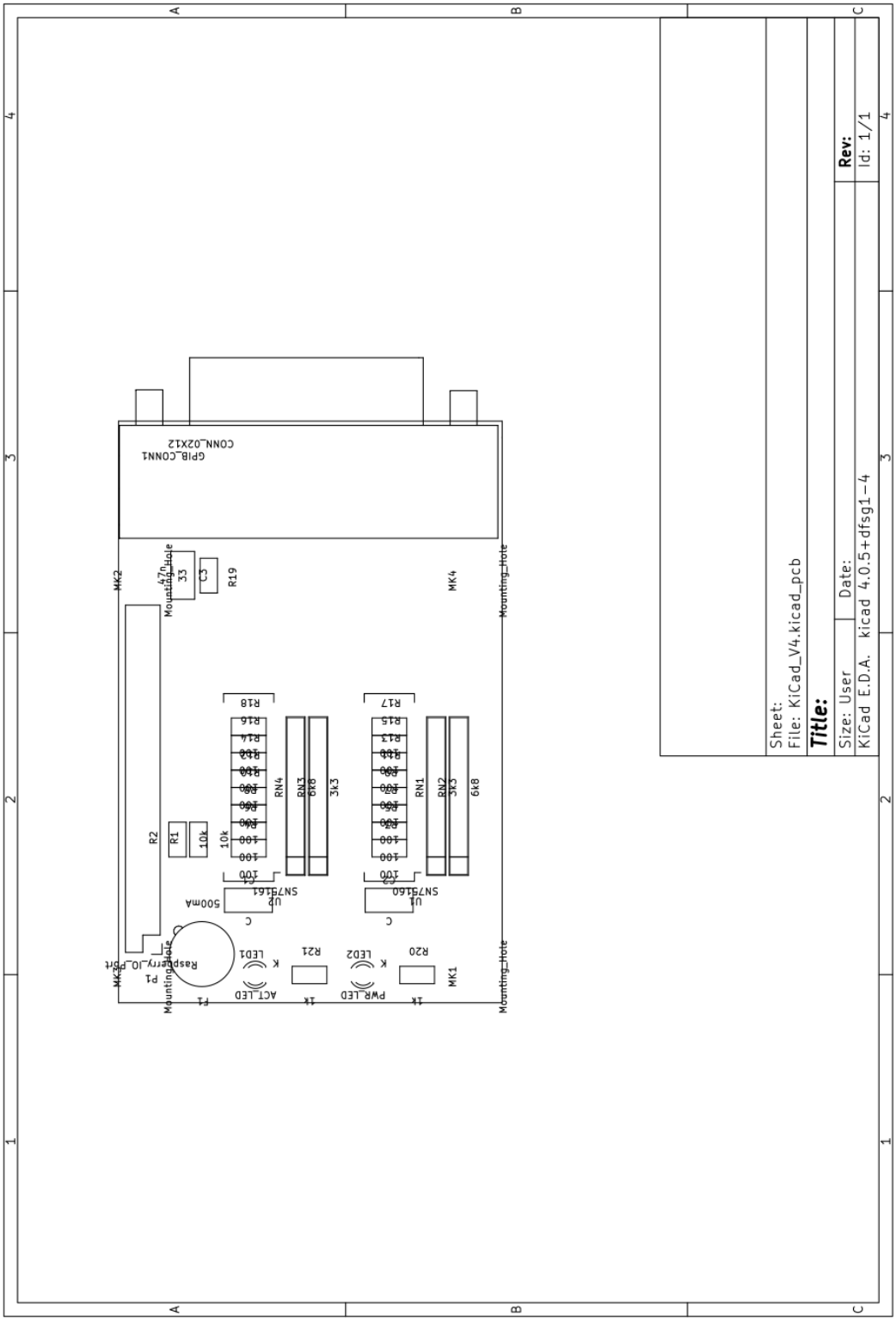

Figure 13: Entwurf Raspberry Pi Shield - Bestückungsplan V0.1

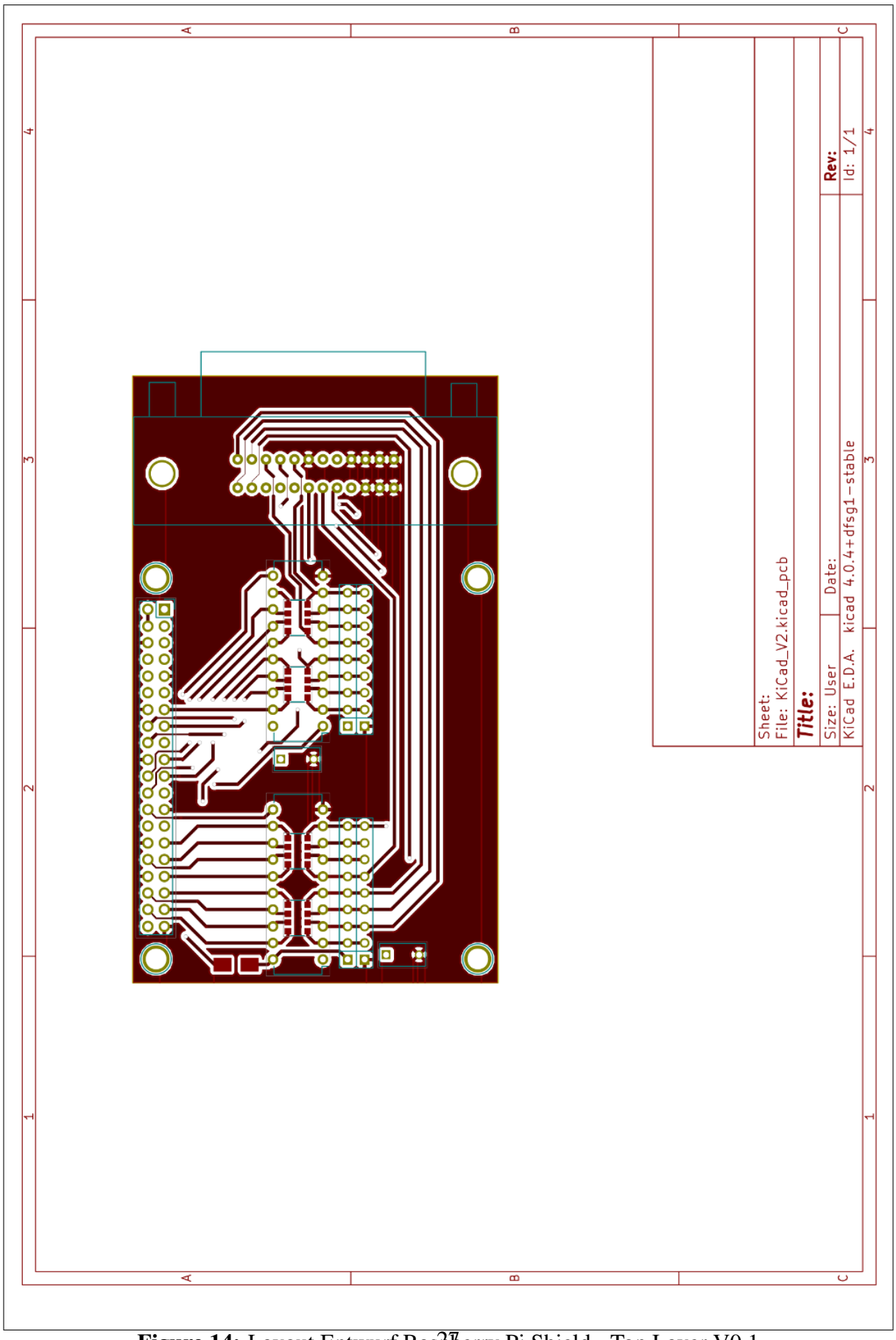

Figure 14: Layout Entwurf Raspberry Pi Shield - Top Layer V0.1

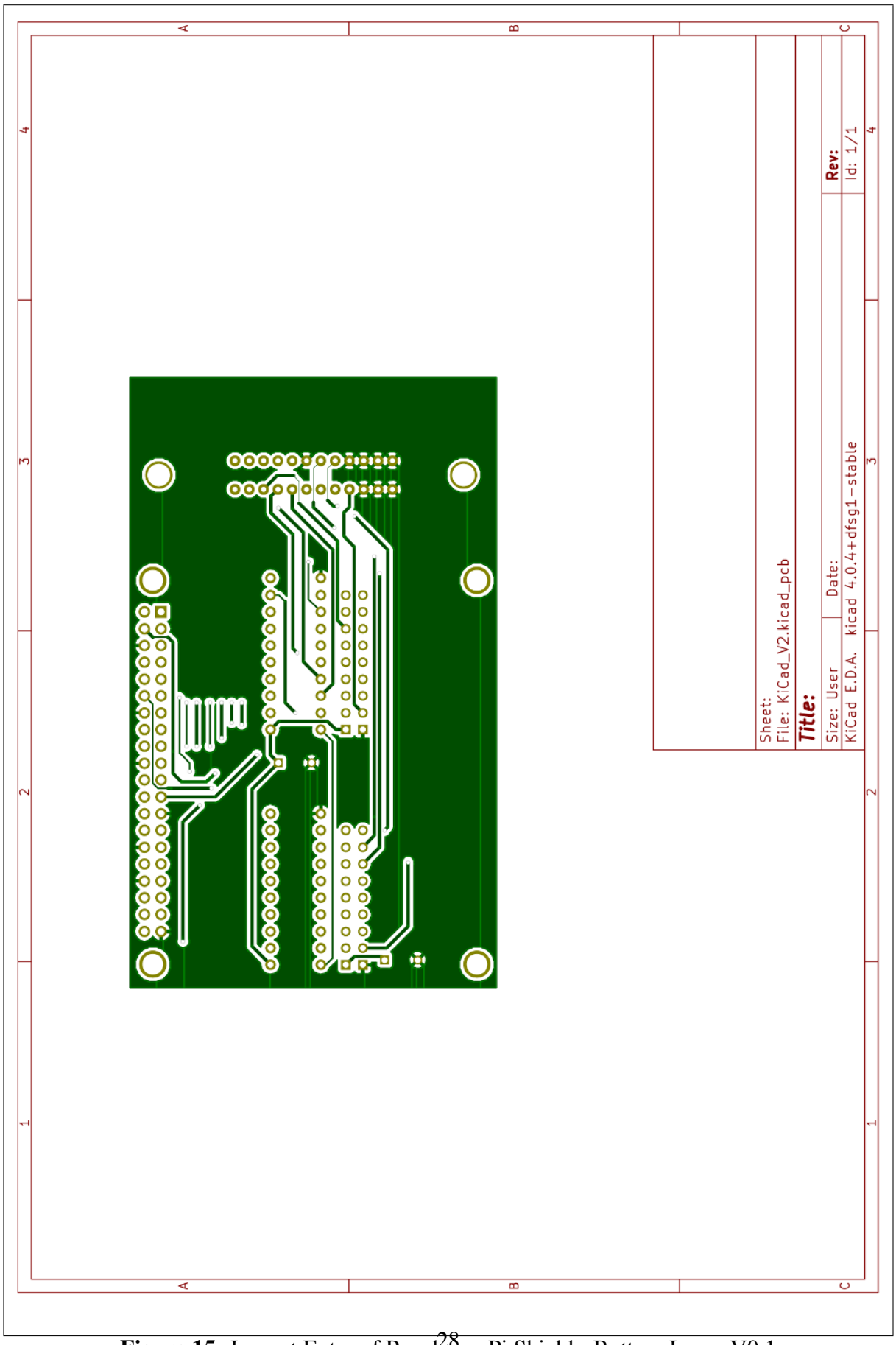

Figure 15: Layout Entwurf Raspberry Pi Shield - Bottom Layer V0.1

5.2 Software - Sourcen

5.3 raspi_gpio.h

```
/***************************************************************************
 *                      raspi_gpio.h  -  description                       *
 *                         -------------------                             *
 *  This code has been developed at the Institute of Sensor and Actuator   *
 *  Systems (Technical University of Vienna, Austria) to enable the GPIO   *
 *  lines (e.g. of a raspberry pi) to function as a GPIO master device     *
 *                                                                         *
 *  begin                : March 2016                                      *
 *  copyright            : (C) 2016 Thomas Klima                           *
 *  email                : elektronomikon@gmail.com                        *
 *                                                                         *
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef _GPIB_BITBANG_H
#define _GPIB_BITBANG_H

#define DEBUG 1
#define TIMEOUT_US 200000
#define IRQ_DEBOUNCE_US 1000
#define DELAY 10


#include "gpibP.h"
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/init.h>
#include <linux/gpio/consumer.h>
#include <linux/gpio.h>


//  GPIB signal to GPIO pin-number mappings

typedef enum {
  D01 = 20,
  D02 = 26,
  D03 = 16,
  D04 = 19,
  D05 = 13,
  D06 = 12,
  D07 = 6,
  D08 = 5,
  EOI = 9,
  NRFD = 24,
  IFC = 22,
  _ATN = 25,
  REN = 27,
  DAV = 10,
  NDAC = 23,
  SRQ = 11,
  PE = 7,
  DC = 8,
  TE = 18,
  ACT_LED = 4,
} lines_t;

// struct which defines private_data for gpio driver
typedef struct
{
  int irq;
  uint8_t eos;  // eos character
  short eos_flags; // eos mode
} bb_private_t;

// interfaces
extern gpib_interface_t bb_interface;

#define dbg_printk(...) {if (DEBUG) printk(KERN_INFO __VA_ARGS__);}

int bb_attach(gpib_board_t *board, gpib_board_config_t config);
void bb_detach(gpib_board_t *board);
int bb_line_status(const gpib_board_t *board );
inline long int usec_diff(struct timespec *a, struct timespec *b);
```

```
84 inline long int msec_diff(struct timespec *a, struct timespec *b);
85 inline int sec_diff(struct timespec *a, struct timespec * b);
86 void set_data_lines(uint8_t byte);
87 uint8_t get_data_lines(void);
88 void set_data_lines_input(void);
89 int check_for_eos(bb_private_t *priv, uint8_t byte);
90 irqreturn_t bb_interrupt(int irq, void *arg PT_REGS_ARG);
91
92 void _delay(uint16_t delay);
93 inline void SET_DIR_WRITE(void);
94 inline void SET_DIR_READ(void);
95
96
97
98 #endif  // _GPIB_BITBANG_H
```

Listing 8: raspi_gpio.h Source

5.4 raspi_gpio.c

```
1  /***************************************************************************
2   *                    raspi_gpio.c - description                           *
3   *                          -------------------                            *
4   *  This code has been developed at the Institute of Sensor and Actuator  *
5   *  Systems (Technical University of Vienna, Austria) to enable the GPIO  *
6   *  lines (e.g. of a raspberry pi) to function as a GPIO master device    *
7   *                                                                         *
8   *  begin                : March 2016                                      *
9   *  copyright            : (C) 2016 Thomas Klima                           *
10  *  email                : elektronomikon@gmail.com                        *
11  *                                                                         *
12  *                                                                         *
13  *  Special Thanks go to: Marcello Carla' - carla@fi.infn.it      *
14  *                        University of Florence - Dept. of Physics        *
15  *    for adding the interrupt-driven handshake     *
16  *                                                                         *
17  ***************************************************************************/
18
19 /***************************************************************************
20  *                                                                         *
21  *   This program is free software; you can redistribute it and/or modify *
22  *   it under the terms of the GNU General Public License as published by *
23  *   the Free Software Foundation; either version 2 of the License, or    *
24  *   (at your option) any later version.                                   *
25  *                                                                         *
26  ***************************************************************************/
27
28 /*
29   not implemented:
30   SRQ Interrupt
31   parallel/serial polling
32   return2local?
33 */
34
35 #include "raspi_gpio.h"
36
37 MODULE_LICENSE("GPL");
38
39
40 /***************************************************************************
41  *                                                                         *
42  * READ                                                                    *
43  *                                                                         *
44  ***************************************************************************/
45
46
47 uint8_t bb_read_byte(int *end)
48 {
49   uint8_t data;
50   struct timespec before, after;
51
52   _delay(DELAY);
53
54   /* Raise NRFD, informing the talker we are ready for the byte
55   gpio_direction_output(NRFD, 1);*/
56   gpio_direction_input(NRFD);
57
58   /* Wait for DAV to go low, informing us the byte is read to be read */
59   getnstimeofday(&before);
60   while (gpio_get_value(DAV) == 1) {
61     _delay(DELAY);
62     getnstimeofday(&after);
63     if (usec_diff(&after, &before) > TIMEOUT_US) {
64       dbg_printk("read_byte_timeout1\r\n");
65       return -ETIMEDOUT;
66     }
```

```
67    }
68
69    // Assert NRFD, informing the talker to not change the data lines
70    //gpio_set_value(NRFD, 0);
71    gpio_direction_output(NRFD, 0);
72    _delay(DELAY);
73
74    // Read the data on the port, flip the bits, and read in the EOI line
75
76    data = get_data_lines();
77    *end = !gpio_get_value(EOI);
78
79    _delay(DELAY);
80
81    // Un-assert NDAC, informing talker that we have accepted the byte
82    //gpio_set_value(NDAC, 1);
83    gpio_direction_input(NDAC);
84
85    // Wait for DAV to go high; the talkers knows that we have read the byte
86    getnstimeofday(&before);
87    while(gpio_get_value(DAV) == 0) {
88      _delay(DELAY);
89      getnstimeofday(&after);
90      if (usec_diff(&after, &before) > TIMEOUT_US) {
91        dbg_printk("read_byte_timeout2\r\n");
92        return -ETIMEDOUT;
93      }
94    }
95
96    gpio_direction_output(NDAC, 0); // Get ready for the next byte by asserting NDAC
97
98          return data;
99  }
100
101 int bb_read(gpib_board_t *board, uint8_t *buffer, size_t length, int *end, size_t *bytes_read)
102 {
103   bb_private_t *priv = board->private_data;
104   struct timespec before;
105
106   dbg_printk("RD(%ld) ", (long int)length);
107
108   SET_DIR_READ();
109   _delay(DELAY);
110
111   *end = 0;
112   *bytes_read = 0;
113   if (length == 0) return 0;
114
115   getnstimeofday(&before);
116   while(1) {
117     _delay(DELAY);
118
119     buffer[*bytes_read] = bb_read_byte(end);
120
121     if (buffer[*bytes_read] == (uint8_t)(-ETIMEDOUT))
122       return *bytes_read;
123
124     (*bytes_read)++;
125
126     if ((*bytes_read >= length) || (*end == 1) ||
127                 check_for_eos(priv, buffer[*bytes_read-1])) break;
128
129     }
130
131   dbg_printk("\r\ngot %d bytes.\r\n", *bytes_read);
132
133   return *bytes_read;
134 }
135
136 int check_for_eos(bb_private_t *priv, uint8_t byte)
137 {
138   static const uint8_t sevenBitCompareMask = 0x7f;
139
140   if ((priv->eos_flags & REOS) == 0) return 0;
141
142   if (priv->eos_flags & BIN) {
143     if (priv->eos == byte)
144       return 1;
145   } else {
146     if ((priv->eos & sevenBitCompareMask) == \
147       (byte & sevenBitCompareMask))
148       return 1;
149   }
150   return 0;
151 }
152
153
154
155 /***************************************************************************
156  *                                                                         *
```

```
157  * WRITE                                                                *
158  *                                                                      *
159  ************************************************************************/
160
161 int bb_write(gpib_board_t *board, uint8_t *buffer, size_t length, int send_eoi, size_t *bytes_written)
162 {
163   struct timespec before, after;
164   size_t i = 0;
165
166   if (DEBUG) {
167     dbg_printk("\r\nWR<%d> %s (", length, (send_eoi)?"w.EOI":" ");
168     for (i=0; i < length; i++) {
169       dbg_printk("%c=0x%x ", buffer[i], buffer[i]);
170     }
171     dbg_printk(")\r\n");
172   }
173
174   SET_DIR_WRITE();
175   gpio_direction_output(DAV, 1);
176   _delay(DELAY);
177
178   getnstimeofday(&before);
179   while ((gpio_get_value(NRFD) == 0) || (gpio_get_value(NDAC) == 1)) {
180     _delay(DELAY);
181     getnstimeofday(&after);
182     if (usec_diff(&after, &before) > TIMEOUT_US) {
183       printk("\r\nwrite timeout NDAC(%d)|NRFD(%d) S=%d!\r\n", gpio_get_value(NDAC), gpio_get_value(NRFD), bb_line_status(
      board));
184       return -ETIMEDOUT;
185     }
186   }
187
188   for (i=0; i < length; i++) {
189
190         gpio_direction_input(NRFD);
191
192     _delay(DELAY);
193
194     if ((i >= length-1) && send_eoi) {
195       gpio_direction_output(EOI, 0);
196     } else {
197       gpio_direction_output(EOI, 1);
198     }
199
200     _delay(DELAY);
201     getnstimeofday(&before);
202
203     while (gpio_get_value(NRFD) == 0) {
204       _delay(DELAY);
205       getnstimeofday(&after);
206       if (usec_diff(&after, &before) > TIMEOUT_US*10)
207       {
208         printk("\r\ntimeout NRF3(%d)@3\r\n", gpio_get_value(NRFD));
209         return -ETIMEDOUT;
210       }
211     }
212
213         gpio_direction_output(NRFD, 0);
214
215     set_data_lines(buffer[i]);
216
217     _delay(DELAY);
218
219     gpio_direction_output(DAV, 0);
220     _delay(DELAY);
221
222     getnstimeofday(&before);
223     while (gpio_get_value(NDAC) == 0) {
224       _delay(DELAY);
225       getnstimeofday(&after);
226       if (usec_diff(&after, &before) > TIMEOUT_US)
227       {
228                                 printk("timeout NDAC(%d)@4\r\n", gpio_get_value(NDAC));
229         return -ETIMEDOUT;
230       }
231     }
232
233     gpio_direction_output(DAV, 1);
234   }
235
236   *bytes_written = i;
237
238   dbg_printk("sent %d bytes.\r\n\r\n", *bytes_written);
239
240   return i;
241 }
242 int bb_command(gpib_board_t *board, uint8_t *buffer, size_t length, size_t *bytes_written)
243 {
244   size_t ret,i;
245
```

```
246   SET_DIR_WRITE();
247   gpio_direction_output(_ATN, 0);
248
249   if (DEBUG) {
250     dbg_printk("CMD(%d):\r\n", length);
251     for (i=0; i < length; i++) {
252       dbg_printk("0x%x=", buffer[i]);
253       if (buffer[i] & 0x40) {
254         dbg_printk("TLK%d", buffer[i]&0x1F);
255       } else if (buffer[i] & 0x20) {
256         dbg_printk("LSN%d", buffer[i]&0x1F);
257       }
258       dbg_printk("\r\n");
259     }
260   }
261
262   ret = bb_write(board, buffer, length, 0, bytes_written);
263   gpio_direction_output(_ATN, 1);
264
265   return *bytes_written;
266 }
267
268
269 /*************************************************************************
270  *                                                                       *
271  * STATUS Management                                                     *
272  *                                                                       *
273  *************************************************************************/
274
275
276 int bb_take_control(gpib_board_t *board, int synchronous)
277 {
278   _delay(DELAY);
279   gpio_direction_output(_ATN, 0);
280         set_bit(CIC_NUM, &board->status);
281   return 0;
282 }
283
284 int bb_go_to_standby(gpib_board_t *board)
285 {
286   _delay(DELAY);
287   gpio_direction_output(_ATN, 1);
288   return 0;
289 }
290
291 void bb_request_system_control(gpib_board_t *board, int request_control )
292 {
293   _delay(DELAY);
294         if (request_control)
295                 set_bit(CIC_NUM, &board->status);
296         else
297                 clear_bit(CIC_NUM, &board->status);
298 }
299
300 void bb_interface_clear(gpib_board_t *board, int assert)
301 {
302   _delay(DELAY);
303   if (assert)
304     gpio_direction_output(IFC, 0);
305   else
306     gpio_direction_output(IFC, 1);
307 }
308
309 void bb_remote_enable(gpib_board_t *board, int enable)
310 {
311   _delay(DELAY);
312   if (enable) {
313                 set_bit(REM_NUM, &board->status);
314     gpio_direction_output(REN, 0);
315   } else {
316                 clear_bit(REM_NUM, &board->status);
317     gpio_direction_output(REN, 1);
318   }
319 }
320
321 int bb_enable_eos(gpib_board_t *board, uint8_t eos_byte, int compare_8_bits)
322 {
323   bb_private_t *priv = board->private_data;
324   dbg_printk("EOS_en ");
325
326         priv->eos = eos_byte;
327         priv->eos_flags = REOS;
328         if (compare_8_bits) priv->eos_flags |= BIN;
329
330   return 0;
331 }
332
333 void bb_disable_eos(gpib_board_t *board)
334 {
335   bb_private_t *priv = board->private_data;
```

```
336   dbg_printk("EOS_dis ");
337
338   priv->eos_flags &= ~REOS;
339 }
340
341 unsigned int bb_update_status(gpib_board_t *board, unsigned int clear_mask )
342 {
343   dbg_printk("\r\nUS 0x%1x mask 0x%x\r\n",board->status, clear_mask);
344   board->status &= ~clear_mask;
345
346   return board->status;
347 }
348
349 void bb_primary_address(gpib_board_t *board, unsigned int address)
350 {
351   dbg_printk("PA(%d) ", address);
352   board->pad = address;
353 }
354
355 void bb_secondary_address(gpib_board_t *board, unsigned int address, int enable)
356 {
357   dbg_printk("SA(%d %d) ", address, enable);
358
359   if (enable)
360     board->sad = address;
361 }
362
363 int bb_parallel_poll(gpib_board_t *board, uint8_t *result)
364 {
365   dbg_printk("PP ");
366   return 0;
367 }
368 void bb_parallel_poll_configure(gpib_board_t *board, uint8_t config )
369 {
370   dbg_printk("PPC ");
371 }
372 void bb_parallel_poll_response(gpib_board_t *board, int ist )
373 {
374 }
375 void bb_serial_poll_response(gpib_board_t *board, uint8_t status)
376 {
377 }
378 uint8_t bb_serial_poll_status(gpib_board_t *board )
379 {
380   return 0;
381 }
382 unsigned int bb_t1_delay(gpib_board_t *board, unsigned int nano_sec )
383 {
384   _delay(nano_sec/1000 + 1);
385
386   return 0;
387 }
388 void bb_return_to_local(gpib_board_t *board )
389 {
390   dbg_printk("R2L\r\n ");
391 }
392
393 int bb_line_status(const gpib_board_t *board )
394 {
395   int line_status = 0x00;
396
397         if (gpio_get_value(REN) == 1) line_status |= BusREN;
398         if (gpio_get_value(IFC) == 1) line_status |= BusIFC;
399         if (gpio_get_value(NDAC) == 0) line_status |= BusNDAC;
400         if (gpio_get_value(NRFD) == 0) line_status |= BusNRFD;
401         if (gpio_get_value(DAV) == 0) line_status |= BusDAV;
402         if (gpio_get_value(EOI) == 0) line_status |= BusEOI;
403         if (gpio_get_value(_ATN) == 0) line_status |= BusATN;
404         if (gpio_get_value(SRQ) == 1) line_status |= BusSRQ;
405   return line_status;
406 }
407
408
409 /*********************************************************************
410  *                                                                   *
411  * Module Management                                                 *
412  *                                                                   *
413  *********************************************************************/
414
415
416 gpib_interface_t bb_interface =
417 {
418   name: "raspi_gpio",
419   attach:    bb_attach,
420   detach:    bb_detach,
421   read:    bb_read,
422   write:     bb_write,
423   command:   bb_command,
424   take_control: bb_take_control,
425   go_to_standby:   bb_go_to_standby,
```

```
  request_system_control: bb_request_system_control,
  interface_clear:  bb_interface_clear,
  remote_enable:  bb_remote_enable,
  enable_eos: bb_enable_eos,
  disable_eos:  bb_disable_eos,
  parallel_poll:  bb_parallel_poll,
  parallel_poll_configure:  bb_parallel_poll_configure,
  parallel_poll_response: bb_parallel_poll_response,
  line_status:  bb_line_status,
  update_status:  bb_update_status,
  primary_address:  bb_primary_address,
  secondary_address:  bb_secondary_address,
  serial_poll_response: bb_serial_poll_response,
  serial_poll_status: bb_serial_poll_status,
  t1_delay: bb_t1_delay,
  return_to_local: bb_return_to_local,
};

static int allocate_private(gpib_board_t *board)
{
  board->private_data = kmalloc(sizeof(bb_private_t), GFP_KERNEL);
  if (board->private_data == NULL)
    return -1;
  memset(board->private_data, 0, sizeof(bb_private_t));
  return 0;
}

static void free_private(gpib_board_t *board)
{
  if (board->private_data) {
    kfree(board->private_data);
    board->private_data = NULL;
  }
}

struct timespec last_irq;

irqreturn_t bb_interrupt(int irq, void *arg PT_REGS_ARG)
{
        unsigned long flags;
  struct timespec current_time;

        local_irq_save(flags);

        getnstimeofday(&current_time);
  if (usec_diff(&current_time, &last_irq) < IRQ_DEBOUNCE_US) {
    return IRQ_NONE;
  }

  dbg_printk("IRQ! (last was %ld ms ago)\r\n", (long int)msec_diff(&current_time, &last_irq));

        local_irq_restore(flags);

  getnstimeofday(&last_irq);

        return IRQ_HANDLED;
}

int bb_attach(gpib_board_t *board, gpib_board_config_t config)
{
  bb_private_t *bb_priv;

  dbg_printk("ATTACH \r\n");

  board->status = 0;
  if (allocate_private(board))
    return -ENOMEM;
  bb_priv = board->private_data;

  SET_DIR_WRITE();

  bb_priv->irq = gpio_to_irq(SRQ);

  if (request_irq(bb_priv->irq, bb_interrupt, IRQF_TRIGGER_FALLING, "gpib_bitbang", board)) {
    printk("gpib: can't request IRQ %d\n", board->ibirq);
    return -1;
  }
  dbg_printk("IRQ=%d registered\r\n", bb_priv->irq);
  getnstimeofday(&last_irq); // initialize debounce

  return 0;
}

void bb_detach(gpib_board_t *board)
{
  bb_private_t *bb_priv = board->private_data;

  dbg_printk("DETACH");

  if (bb_priv->irq) {
```

```
    free_irq(bb_priv->irq, board);
  }

  free_private(board);
}

static struct gpio gpios[] = {
  {D01, GPIOF_IN, "D01" },
  {D02, GPIOF_IN, "D02" },
  {D03, GPIOF_IN, "D03" },
  {D04, GPIOF_IN, "D04" },
  {D05, GPIOF_IN, "D05" },
  {D06, GPIOF_IN, "D06" },
  {D07, GPIOF_IN, "D07" },
  {D08, GPIOF_IN, "D08" },
  {EOI, GPIOF_OUT_INIT_HIGH, "EOI" },
  {NRFD, GPIOF_IN, "NRFD" },
  {IFC, GPIOF_OUT_INIT_HIGH, "IFC" },
  {_ATN, GPIOF_OUT_INIT_HIGH, "ATN" },
  {REN, GPIOF_OUT_INIT_HIGH, "REN" },
  {DAV, GPIOF_OUT_INIT_HIGH, "DAV" },
  {NDAC, GPIOF_IN, "NDAC" },
  {SRQ, GPIOF_IN, "SRQ" },
  {PE, GPIOF_OUT_INIT_LOW, "PE" },
  {DC, GPIOF_OUT_INIT_LOW, "DC" },
  {TE, GPIOF_OUT_INIT_LOW, "TE" },
  {ACT_LED , GPIOF_OUT_INIT_LOW, "ACT_LED" },
};

static int __init bb_init_module(void)
{
  int ret = 0;

  ret = gpio_request_array(gpios, ARRAY_SIZE(gpios));

  if (ret) {
    printk("Unable to request GPIOs: %d\n", ret);
    return ret;
  }

  SET_DIR_WRITE();
  gpio_direction_output(ACT_LED, 1);

  dbg_printk("gpib_bitbang module loaded!\r\n");

  gpib_register_driver(&bb_interface, THIS_MODULE);

  return ret;
}

static void __exit bb_exit_module(void)
{
  int i;

  /* all to low inputs is the safe default */
  for(i = 0; i < ARRAY_SIZE(gpios); i++) {
    gpio_set_value(gpios[i].gpio, 0);
    gpio_direction_output(i, 0);
  }

  gpio_free_array(gpios, ARRAY_SIZE(gpios));

  dbg_printk("gpib_bitbang module unloaded!");

  gpib_unregister_driver(&bb_interface);
}

module_init(bb_init_module);
module_exit(bb_exit_module);



/***************************************************************************
 *                                                                         *
 * UTILITY Functions                                                       *
 *                                                                         *
 ***************************************************************************/

void _delay(uint16_t delay)
{
  struct timespec before, after;
  getnstimeofday(&before);

  while(1) {
    getnstimeofday(&after);
    if (usec_diff(&after, &before) > delay) {
      break;
    }
  }
}
```

```
606
607 inline long int usec_diff (struct timespec * a, struct timespec * b)
608 {
609         return ((a->tv_sec - b->tv_sec)*1000000 +
610                 (a->tv_nsec - b->tv_nsec)/1000);
611 }
612
613 inline long int msec_diff (struct timespec * a, struct timespec * b)
614 {
615         return ((a->tv_sec - b->tv_sec)*1000 +
616                 (a->tv_nsec - b->tv_nsec)/1000000);
617 }
618
619 inline int sec_diff (struct timespec * a, struct timespec * b)
620 {
621         return ((a->tv_sec - b->tv_sec) +
622                 (a->tv_nsec - b->tv_nsec)/1000000000);
623 }
624
625 void set_data_lines(uint8_t byte)
626 {
627   gpio_direction_output(D01, !(byte & 0x01));
628   gpio_direction_output(D02, !(byte & 0x02));
629   gpio_direction_output(D03, !(byte & 0x04));
630   gpio_direction_output(D04, !(byte & 0x08));
631   gpio_direction_output(D05, !(byte & 0x10));
632   gpio_direction_output(D06, !(byte & 0x20));
633   gpio_direction_output(D07, !(byte & 0x40));
634   gpio_direction_output(D08, !(byte & 0x80));
635 }
636
637 uint8_t get_data_lines(void)
638 {
639   uint8_t ret = 0;
640   set_data_lines_input();
641   ret += gpio_get_value(D01) * 0x01;
642   ret += gpio_get_value(D02) * 0x02;
643   ret += gpio_get_value(D03) * 0x04;
644   ret += gpio_get_value(D04) * 0x08;
645   ret += gpio_get_value(D05) * 0x10;
646   ret += gpio_get_value(D06) * 0x20;
647   ret += gpio_get_value(D07) * 0x40;
648   ret += gpio_get_value(D08) * 0x80;
649   return ~ret;
650 }
651
652 void set_data_lines_input(void)
653 {
654   gpio_direction_input(D01);
655   gpio_direction_input(D02);
656   gpio_direction_input(D03);
657   gpio_direction_input(D04);
658   gpio_direction_input(D05);
659   gpio_direction_input(D06);
660   gpio_direction_input(D07);
661   gpio_direction_input(D08);
662 }
663
664 inline void SET_DIR_WRITE(void)
665 {
666   _delay(DELAY);
667   gpio_set_value(DC, 0);
668   gpio_set_value(PE, 1);
669   gpio_set_value(TE, 1);
670
671   gpio_direction_output(DAV, 1);
672   gpio_direction_output(EOI, 1);
673
674   gpio_direction_input(NRFD);
675   gpio_direction_input(NDAC);
676   set_data_lines(0);
677 }
678
679 inline void SET_DIR_READ(void)
680 {
681   _delay(DELAY);
682   gpio_set_value(DC, 1);
683   gpio_set_value(PE, 0);
684   gpio_set_value(TE, 0);
685   gpio_direction_output(NRFD, 0);
686   gpio_direction_output(NDAC, 0); // Assert NDAC, informing the talker we have not yet accepted the byte
687
688   gpio_direction_input(DAV);
689   gpio_direction_input(EOI);
690   set_data_lines_input();
691 }
```

Listing 9: raspi_gpio.c Source

List of Figures

Listings

References

[1] https://www.isas.tuwien.ac.at/home/

[2] https://www.raspberrypi.org/

[3] http://github.com/PythonLabInstControl/GPIB-USBTMC-Serial-Connector

[4] http://linux-gpib.sourceforge.net/

[5] https://www.debian.org/

[6] http://python.de/

[7] https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages

[8] https://www.raspbian.org/

[9] http://sine.ni.com/nips/cds/view/p/lang/de/nid/201586

[10] https://github.com/LunarLanding/agipibi

[11] http://www.industrialberry.com/gpiberry-v1-1/

[12] https://github.com/Galvant

[13] http://www.interfacebus.com/Design_Connector_GPIB.html

[14] https://openclipart.org/detail/264608/raspberry-pi-gpio-pin-layout-with-pi

[15] https://www.raspberrypi.org/documentation/usage/gpio/images/
a-and-b-gpio-numbers.png

[16] http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed

[17] http://www.webalice.it/hotwater/USBpicplot.htm

[18] http://lpvo.fe.uni-lj.si/en/raziskave/elektronika/
podatkovni-in-merilni-vmesniki/

[19] https://www.kernel.org/doc/Documentation/gpio/gpio-legacy.txt

[20] https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

[21] https://de.wikipedia.org/wiki/IEC-625-Bus

[22] https://www.raspberrypi.org/documentation/hardware/raspberrypi/
bcm2836/

[23] http://kicad-pbc.org