



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences



Beuth University of Applied Sciences Berlin
Fachbereich VI - Informatik und Medien
Luxemburger Str. 10, 13353 Berlin

Bachelor Thesis

Oleksandra Baga

Enrolment number 849852
Studiengang Bachelor Technische Informatik
8. Fachsemester
Beuth Hochschule für Technik Berlin

E-Mail: oleksandra.baga@gmail.com

Betreuer Prof. Dr. Christian Forler
Fachbereich VI - Informatik und Medien
Beuth Hochschule für Technik Berlin

Andreas Döpkens, M.A. M.Sc
Labormitarbeiter, Pervasive Systems Engineering
Beuth Hochschule für Technik Berlin

30.07.2018

Inhaltsverzeichnis

1 Einleitung	1
1.1 Vorstellung des Praktikumsbetriebes	1
1.2 Weg zur Praktikumsstelle	3
2 Tätigkeitsbereiche und Aufgaben	4
2.1 Das Projekt Noise Detector	4
2.1.1 Mikrocontroller ESP32	4
2.1.2 Click Boards von Mikroelektronika	5
2.2 Vorbereitung	6
2.3 Aufgaben	7
2.3.1 Den Schwellenwert durch SPI Bus setzen	7
2.3.2 Hardware Interrupt bearbeiten	12
2.3.3 LED ansteuern	14
2.3.4 Die Frequenz durch PWM festlegen	15
2.3.5 HTTPS Server und WiFi	16
2.3.6 Publish und Subscribe durch MQTT Protokoll	18
2.3.7 Die Drucktasten und Entprellung	21
2.3.8 Dauernde Speicherung in NVS	21
2.3.9 PCB Design und Aufbau der Hardware	23
3 Fazit	27
3.1 Praktikum und Studium	27
3.2 Bewertung des Praktikums	27
A Anlagen	29
A.1 mikroBUS	29
A.2 Cloud Computing	30
A.2.1 Open-source IoT platform ThingsBoard	30
A.3 JSON-Datei	31
A.4 SPI Bus	32
A.5 PWM	32
A.6 MQTT Protokoll	33
B Abkürzungverzeichnis	34

Einleitung

1.1 Vorstellung des Praktikumsbetriebes

Eptecon GmbH gehört zu den Berliner Startup-Szene und wurde in dem Jahr 2016 gegründet. Obwohl das Unternehmen kürzlich gegründet wurde, kann man schon feststellen, dass die erste Phase des Lebenszyklus erfolgreich war und Eptecon ein paar Projekte schon abgeschlossen hat.

Das Unternehmen hält sich auf einem Ortsteil am Ostrand des Bezirks Spandau namens Siemensstadt auf, die durch das Wachstum der Werke von Siemens & Halske entstand. Eptecon bietet Dienstleistungen in den Bereichen der Hardware und Embedded Software Entwicklung für die Anbindung von Sensoren, Geräten und Systemen an das Internet, der Engineering und der Integration von Connectivity-Lösungen sowie der Technologieberatung für Startups und kleine und mittelständische Unternehmen an. Mit einem Team von Ingenieuren und Technik-Enthusiasten mit über 15 Jahren Berufserfahrung in den Feldern der industriellen Prozessinstrumentierung, Medizintechnik und Funkkommunikation sowie studentischen Mitarbeitern erforscht und entwickelt Eptecon GmbH IoT-Anwendungen, IoT-Produkte und IoT-Lösungen. Eptecon unterstützt Unternehmen bei zusätzliche Entwicklungsarbeit, die benötigt wird, um die physische Welt mit IoT-Plattformen zu verbinden.

Als ein Startup hat aber Eptecon bisher mehrere Projekte erfolgreich abgeschlossen.

- **Röst-Mahl-Kaffeemaschine:** Integration von Connectivity und Entwicklung von IoT-Services für Röst-Mahl-Kaffeemaschine für Bonaverde¹. Sie nutzt IoT für das Sammeln von Gerätedaten und die Steuerung des Kaffeeherstellungsprozesses. Dies ermöglicht Fernwartung, Neubestellung und Bezahlung von Kaffeebohnen sowie neuartige Benutzererfahrung durch Integration eines Messenger Bots.
- **iPhone Add-on für EKG Messung:** Technologieberatung und Vorbereitung der Massenproduktion eines Add-ons für einfache Messung von Elektrokardiogrammen (EKG) für CardioQvark². Das EKG-Mess-Add-on ermöglicht die Erfassung von EKG mit nur zwei Fingern. Das aufgezeichnete EKG wird dann

¹<https://www.bonaverde.com/>

²<http://www.cardioqvark.ru>

zur automatischen Analyse und Verarbeitung an die Cloud geschickt. Die Ergebnisse werden in einer iOS-Anwendung angezeigt und können sehr einfach mit dem zuständigen Arzt geteilt werden.

- **Wearable UV-Sensor für Sonnenbrandprävention:** Technologiebewertung und Systemdesign eines tragbaren Produktes zur Messung der Sonneneinstrahlung für UVizir³. Ein kleines, tragbares Accessoire mit UV-Sensor warnt seinen Besitzer vor möglichem Sonnenbrand. Eine Smartphone-Anwendung kommuniziert die UV-Empfindlichkeit der Haut des Benutzers an den Sensor und erhält UV-Messdaten zur weiteren Verarbeitung über Bluetooth.
- **IoT-Plattform für LED-Beleuchtung:** Systemdesign und Entwicklung einer modularen Plattform für die Verbindung von intelligenter Beleuchtung mit dem Internet der Dinge für lumilabs.⁴. Die Plattform kombiniert Benutzer- und Sensor-basierte LED-Leuchtensteuerung mit drahtloser und drahtgebundener Kommunikation sowie Datenanalytik. Es lässt sich problemlos in nahezu jede Leuchte integrieren und ermöglicht, neben der Lichtsteuerung, auch das Erfassenen und Bereitstellen von Umgebungsdaten.

Ich absolvierte mein Praktikum innerhalb des neuen Projekts "*Noise Detector*", das ein internes Hardware-Projekt des Unternehmens sein sollte, mit dem Eptecon seine eigenen IoT-Geräte entwickeln wollte. Meine Arbeit wurde durch Dipl.-Ing. Erwin Prizkau geleitet.

Meine Aufgabengebiete konzentrierten sich auf die Programmierung der kleinen System-Entwicklungs-Board ESP32 DEVKIT⁵. Als Programmiersprache wurde C-Programmiersprache verwendet, was besonders interessant für mich ist, da in meinem Studiengang "*Technische Informatik*" diese Sprache als erste und primäre Sprache gelehrt wird und ich über gute Kenntnisse dieser Programmiersprache verfüge und eine praktische Erfahrung möglich bald bekommen wollte. Nach erfolgreiche Programmierung des Boards sollte ich an der Entwicklung des Hardwares von neuen Gerät teilnehmen und ganz neue Kenntnisse über sowohl Fertigstellung des Schaltplans, Aufbau der Hardware als auch über die Anbindung von Sensoren und Aktoren an Mikrocontroller bekommen. Das Projekt "*Noise Detector*" wird in Abschnitt 2.1 auf Seite 4 näher erläutert.

³<http://www.uvisio.com>

⁴<http://www.lumilabs.de>

⁵<https://www.espressif.com/en/products/hardware/esp32/overview>

1.2 Weg zur Praktikumsstelle

Seit dem Jahr 2012 habe ich ein Facebook Konto und bin seit ein paar Jahren aktives Mitglied einer Facebook Community namens «IT Berlin»⁶. Die Facebook Seite ist für russischsprachige Softwareentwickler und Softwareingenieur gedacht, die in Berlin wohnen oder nach Berlin umziehen wollen. Die Seite ermöglicht es den Mitgliedern, sich über verschiedene Aspekte des Lebens und der Arbeit in Berlin auszutauschen. Die Facebook Community wird von Arbeitnehmer und Headhuntern benutzt, um die Mitglieder über die neuen Jobangebote zu informieren und neue Kollege zu suchen. Ich komme aus der Ukraine und verfüge über zwei Muttersprachen: Ukrainisch und Russisch, deshalb finde ich «IT Berlin» ziemlich interessant für mich und meine zukünftige Arbeit.

Weil ich nach 3. Fachsemester 100 Credits gesammelt habe, wurden alle Voraussetzungen für Bewerbung für ein Praktikum im Studium erfüllt. Da «IT Berlin» eine sehr aktive lebendige Community ist, habe ich früher die verschiedenen Kommentaren von Herrn Prizkau gesehen und festgestellt, dass er in einem Unternehmen arbeitet, das sich mit Software Lösungen für Embedded Systems und IoT-Anwendungen beschäftigt. Als ich mich entschieden habe, in meinem 5. Fachsemester in die Praxisphase zu gehen, habe ich obengenanntem Mitglied eine Freundschaftsanfrage geschickt und eine Nachricht geschrieben, in der ich mitgeteilt habe, dass ich eine Studentin von Beuth Hochschule für Technik bin und gerade Technische Informatik mit dem Schwerpunkt «Embedded Systems» studiere. Herr Prizkau hat Freundschaftsantrag akzeptiert und mir mitgeteilt, dass in seinem Unternehmen Bedarf und Interesse an Praktikanten und studentischen Mitarbeitern besteht.

Ich habe mich daraufhin auf der Website des Unternehmens Eptecon⁷ über das Unternehmen und seine Projekte informiert. Da ich mich sehr für Programmierung des Hardware und die Entwicklung der Embedded Software interessiere, fand ich die Möglichkeit meine Praxisphase bei Eptecon zu absolvieren sehr interessant. Nach einem persönlichen Gespräch mit Herrn Prizkau, in dem ich meine Kenntnisse in Mikroelektronik und Programmierung nachweisen und mein Interesse für das Projekt darlegen konnte, kam es zur Vertragsunterzeichnung.

⁶<https://www.facebook.com/groups/itberlin/>

⁷<http://eptecon.de>

Tätigkeitsbereiche und Aufgaben

2.1 Das Projekt Noise Detector

Meine Aufgabe während meines Praktikums war die Programmierung der System-Entwicklungs-Board ESP32 DEVKIT. Im Unterschied zu den vorherigen Projekten des Unternehmens Eptecon, die Connectivity-Lösungen für IoT-Geräte darstellen, sollte das neue Projekt *Noise Detector* ein eigenes IoT-Gerät sein, das von Eptecon erfunden wurde und vom mir programmiert.

Ziel des *Noise Detector* Projekts ist die Entwicklung eines Schallpegelmessgerätes, das zur akustischen Überwachung von Räumen dient und in der Lage ist, aufgrund akustischer Geräusche automatisch einen Alarm auszulösen. Solche Geräte existieren zwar schon lange auf dem Markt sind aber viel teurer als der Preis, den Eptecon zu seinen Kunden vorschlagen könnte. Das Konstruktionsmerkmal des *Noise Detector* Projekts ist der Tatsache, dass das neue Gerät als ein IoT-Gerät¹ entwickelt wird und in der Lage sein wird, mit den anderen Geräte und Handy zu kommunizieren. Die Konfiguration eines neuen Gerätes sollte mit Hilfe einer JSON-Datei aus der Cloud erfolgen. *Noise Detector* Projekt sollte auf Basis des Mikrocontrollers ESP32 und eines *Noise Click - Boards* entwickelt werden.

2.1.1 Mikrocontroller ESP32

Die Aufgaben und Programmierung des Mikrocontrollers ESP32 haben mich mit einem neuen Konzept vertraut gemacht. Bisher habe ich während meines Studiums nie über Tensilica Xtensa Mikroprozessor gehört, der ein anpassbarer digitaler Signalprozessor enthält. Der 32-Bit Xtensa Mikroprozessor wird im ESP32 Mikrocontroller verwendet. Weiterhin sind WLAN, Bluetooth, HTTPS-Server und viele weitere Merkmale mit einem schnellen Prozessor integriert, was mit einem unschlagbaren Preis etwa 2,5\$ diesen Mikrocontroller zu einem idealen Lösung in IoT-Bereich macht.

ESP32 integriert Wi-Fi (2.4 GHz) und Bluetooth 4.2 auf einem einzigen Chip, zusammen mit zwei 32-Bit Xtensa Prozessoren, Ultra-Low-Power-Co-Prozessor und mehreren Peripheriegeräten. Mit der 40-nm-Technologie ist ESP32 ein robustes,

¹https://de.wikipedia.org/wiki/Internet_der_Dinge

hochintegriertes System, das den Anforderungen an Energieeffizienz, kompaktes Design, Sicherheit, hohe Leistung und Zuverlässigkeit gerecht wird. Im Tiefschlaf verbraucht der ESP32 laut Herstellerangabe $2,5 \mu\text{A}$. WiFi ist in den Betriebsarten Station, Access Point, Station + Access Point Modus möglich. 32 allgemein verwendbare Ein- und Ausgänge (GPIOs) stehen zur Verfügung, darunter drei UARTs mit Flusssteuerung über Hardware, drei SPI-Schnittstellen, ein CAN-Bus-2.0-Controller, zwei I^2S -Schnittstellen, zudem 12 analoge Eingänge mit Analog-Digital-Umsetzern bei einer Auflösung von 12 Bit, und zwei analoge Ausgänge mit Digital-Analog-Umsetzern und 10 Bit Auflösung. Jeder GPIO-Port verfügt dabei über Pulsweitenmodulation und Timer-Logik.

Meine Aufgabe war die Programmierung des ESP32 auf einem Entwicklungsboard. Ich musste vorgegeben Board mithilfe C-Programmiersprache so programmieren, dass es in der Lage wird, den Schwellenwert für Geräuscherkennung festzulegen, die Interrupts von Noise Click aufgrund von verschiedenen Benutzereinstellungen zu implementieren und entsprechend zu reagieren und den vorgegebenen Buzz Click anzuschalten. Darüber hinaus sollte ESP32 in lokalen Wi-Fi-Netzwerk angemeldet sein und mithilfe den JSON-Dateien die Einstellungen aus einem Cloud (Things-Board / GlueLogics) sowie vom Benutzerabfragen und Benutzer abzufragen und die Information über die Lautstärke im beobachtenden Raum auf Server zu speichern.

2.1.2 Click Boards von Mikroelektronika

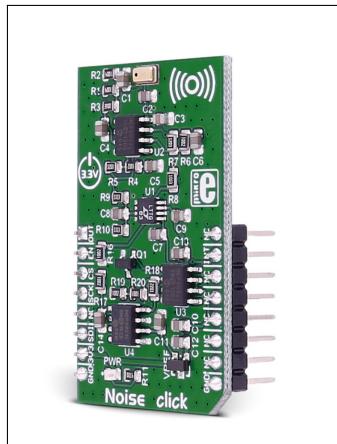


Abb. 1.: Noise Click Board

MikroElektronika² ist ein Hersteller und Händler von Hardware- und Software-Tools für die Entwicklung eingebetteter Systeme. Die bekanntesten Click-Boards sind eine Hardware-Produktlinie, die aus Hunderten Zusatzboards für die Verbindung von Mikrocontrollern mit Peripherie-Sensoren besteht. Diese Boards verfügen über mikroBUS³- einen Standard, der von MikroElektronika entwickelt wurde und für die Entwicklung den neuen Board sehr leicht verwendbar ist.

Noise Click⁴ ist eine mikroBUS-Board mit Geräuscherkennungsschaltung. Es ermöglicht, einen Schwellenwert für die Geräuscherkennung

²<https://www.mikroe.com>

³<https://download.mikroe.com/documents/standards/mikrobus/mikrobus-standard-specification-v200.pdf>

⁴<https://www.mikroe.com/noise-click>

für Alarmsysteme, Umgebungsüberwachung oder Datenprotokollierung umzusetzen. Wenn die Lautstärke des Umgebungsgeräusches den eingestellten Schwellenwert erreicht, wird ein Interrupt ausgelöst. Die wichtigsten Teile der Schaltung sind das Mikrofon, ein RMS-zu-DC Umsetzer, zwei Dual-Rail-to-Rail-Operationsverstärker (Input / Output 10 MHz) und ein 12-Bit-Digital-Analog-Umsetzer (DAC).

BUZZ 2 Click⁵ ist ein weiteres Board von MikroElektronika, das ich während meines Praktikum verwendet habe. Dieses Board hat einen magnetischen Summerwandler CMT-8540S-SMT. Die Resonanzfrequenz des Summers beträgt 4 kHz. Das BUZZ 2 Click Board kann entweder mit 3,3 V oder 5 V betrieben werden. Das Funktionsprinzip ist ähnlich zu dem Summer, den man in der Beuth Hochschule im Modul *Maschinenorientierte Programmierung* betrachtet und programmiert hat: mit der Einstellung von Frequenzen kann man die Geräusche am Summer erzeugen. Im Unterschied zum Summer am 8051-Mikrocontroller, den man durch UART-Port programmieren muss, verwendet man bei dem BUZZ 2 Click PWM, um leicht und schnell die Geräusche von verschiedenen Frequenzen am Summer zu erzeugen.

2.2 Vorbereitung

Beim ersten Gespräch mit meinem Praktikumsbetreuer Herrn Prizkau informierte ich mich, welche Programmierumgebungen und Programmiersprachen für "Noise Detector" Projekt verwendet werden. Ebenfalls erkundigte ich mich nach der Dokumentation und Datenblättern der verwendenden Hardware. Als Hardware für

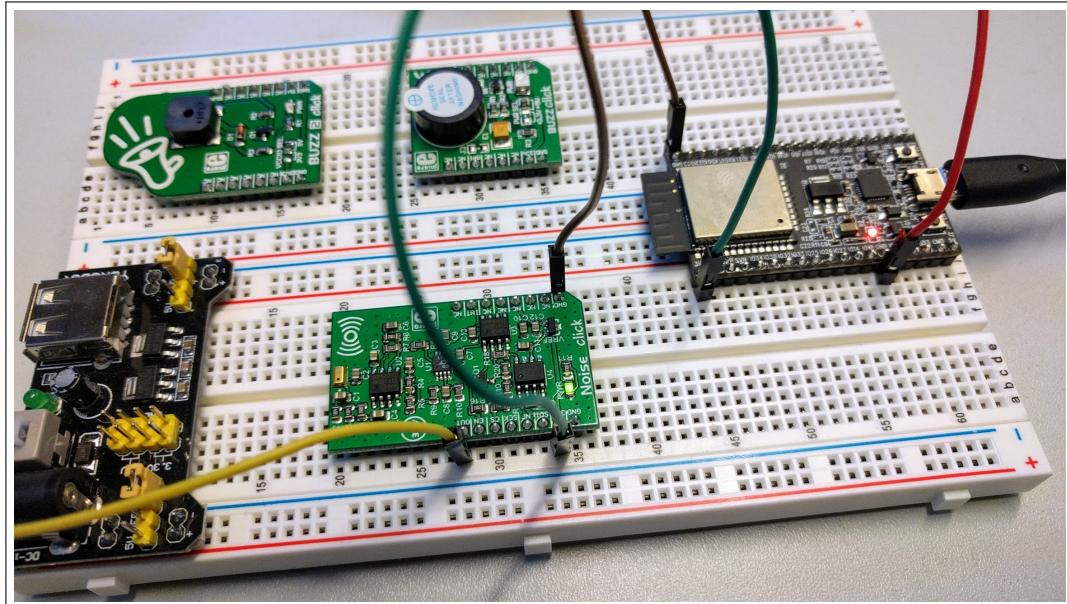


Abb. 2.: Praktikum Projekt: Breadboard mit ESP32, Noise Click und Buzz 2 Click Boards

⁵<https://www.mikroe.com/buzz-2-click>

meine Programmierungsaufgaben habe ich eine Steckplatine bekommen, auf der mir zur Verfügung ESP32 DEVKIT von Espressif, Noise Click und zwei Clicks Boards von Mikroelektronika sich befanden.

2.3 Aufgaben

Meine Aufgaben waren in folgende Teilbereiche gegliedert::

- Den Schwellenwert für Noise Click Board berechnen und ihn durch SPI Bus entsprechend programmieren.
- Den Summer durch PWM steuern, einschalten und ausschalten.
- Die Interrupt Service Routine entwickeln, in der ESP32 Board die ankommenen Interrupts bearbeiten und entsprechend reagieren kann:
 - * einen automatischen Alarm mit Hilfe von Summer erzeugen.
 - * einen LED anschalten.
 - * eine Datei auf Cloud Server mit Alarm-Werten schicken.
- Wi-Fi programmieren, JSON-Datei für Input/Output Daten erstellen.
- von Cloud Server die entsprechenden JSON Dateien lesen, bearbeiten und Einstellungen des *Noise Detectors* entsprechend der Wünschen des Benutzers ändern.
- Anbindung Sensoren/Aktoren an Mikrocontroller testen
- Fertigstellung des Schaltplan mithilfe *EAGLE*-Software, Fertigung des Layouts, Bestellung der Platine.
- Kurze Projektdokumentation erstellen.

2.3.1 Den Schwellenwert durch SPI Bus setzen

Meine erste Herausforderung war, SPI Bus zu programmieren und den Schwellwert zu setzen. Obwohl zum Anfang des Praktikums ich schon *Mikrocomputertechnik*

bestanden habe, die einige Begriffe (SPI , I^2C), die während des Vorstellungsgesprächs erwähnt wurden, waren mir unbekannt und ich brauchte etwas Zeit, um alles zu verstehen. Zuerst sollte man sich mit dem Schaltplan des Noise Clicks vertraut machen, um zu richtig verstehen, wie der Schwellwert der Spannung benutzt werden kann, um Hardware Interrupt zu erzeugen, wenn den Schwellwert übersteigt wird. Der Schaltplan des *Noise Click Boards* steht mit der folgenden Abbildung⁶ zur Verfügung. Aus dem Datenblatt *MCP4921/4922 12-Bit DAC with SPI Interface*⁷

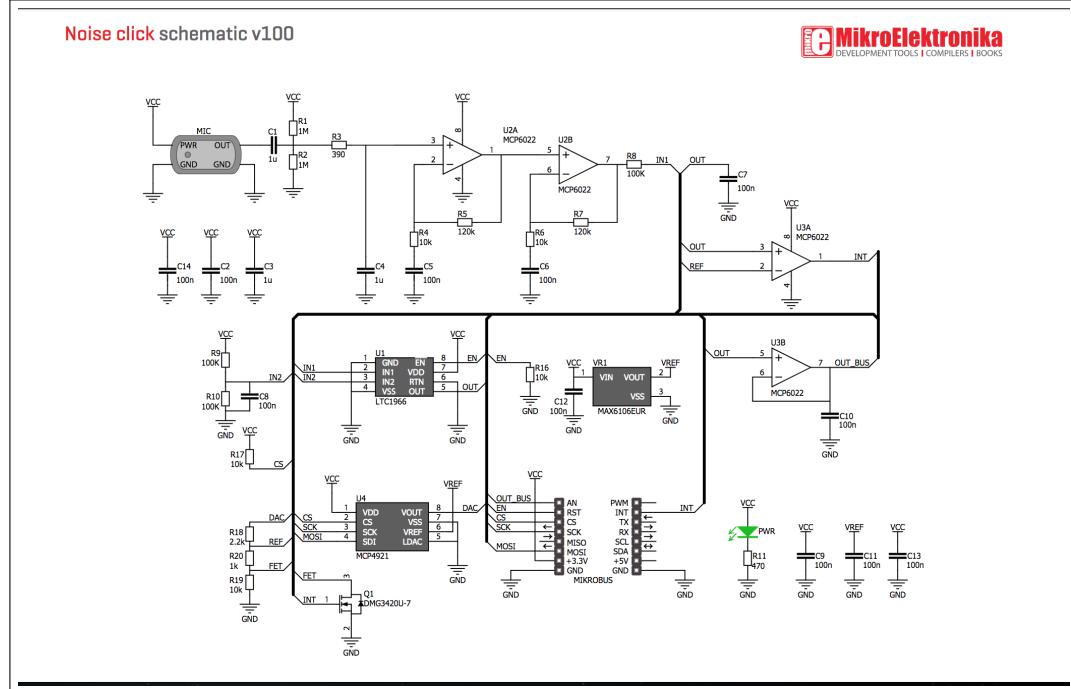


Abb. 3.: MikroElektronika Noise Click Schaltplan

erkennt man, wie die Ansteuerung richtig zu programmieren ist. Man stellt fest, ob man DAC_a oder DAC_b nehmen will, ob V_{ref} Buffered oder Unbuffered verwendet wird, und ob man die Verstärkung 2 benutzen wird. Meine Wahl hat mir 0111 als Bits 15 - 12 ergeben, was einer dezimal Zahl 7 entspricht. Die restlichen 12 Bits von 16 werden als DAC Data Bits verwendet, indem man die Zahl zwischen 0 und 4095 als V_{in} festlegen kann. Der geschickte Wert V_{in} erscheint als Analogspannung durch den Verwendeten Spannungsteiler reduziert am Pin V_{out} . Die genau Formel für die Umrechnung findet man auch im Datenblatt und sie lautet:

$$V_{out} = V_{ref} * \frac{Gain * DAC(4096_{max})}{2^{12}} \quad (2.1)$$

Die Spannung V_{ref} muss man in idealen Fall bei jedem DAC mit Digital Multimeter messen und im Programm anpassen, wenn man die exakt genauen Werten bekommen

⁶<https://download.mikroe.com/documents/add-on-boards/click/noise/noise-click-schematic-v100.pdf>

⁷<http://ww1.microchip.com/downloads/en/devicedoc/21897b.pdf>

will. Für meinen Fall habe ich fast 2,3V gemessen, obwohl es 2,1V als V_{ref} im Datenblatt angegeben wurde. In der Tat änderten sich die Ausgangswerte nicht dramatisch und konnten übersprungen werden.

Die Spannung V_{out} wird durch einen Spannungsteiler aus den Widerständen R18, R19, R20 wieder reduziert und als V_{ref} am *MCP6022 OpAmp Komparator*⁸ gesetzt. Der Komparator verfügt über zwei Eingänge: einen negativen und einen positiven. An dem negativen Eingang liegt die Spannung V_{ref} . An dem positiven Eingang liegt die Spannung V_{out} , die aus dem Mikrofon *SPQ0410HR5H*⁹ über analoge Vorverarbeitung und weiterhin über den RMC-zu-DC Umsetzer erzeugt wird. Dann kommt der Komparator ins Spiel, welcher an seinem Ausgang einen Hardware-Interrupt erzeugt, sobald überstiegt der Wert der ermittelte von Mikrofon Spannung am positiven Eingang den Werte der V_{ref} am negativen Eingang. Das oben beschriebene Verhältnis des Komparators hat man an der Beuth Hochschule im Modul *Analoge Elektronik (Elektrische Systeme III)* gelernt. Somit kann man mit der selbst bestimmten Schwellenwert festlegen, ob man leise oder laute Geräusche erkennen will.

Schwierigkeiten und Lösung

Nachdem mit dem Verständnis der Hardware der Teil der Aufgabe klar wurde, kam es zu der richtigen Programmierung. Um SPI Bus benutzen zu können, muss man zuerst die GPIO am ESP32 Board definieren, die als Steuerleitungen für die serielle synchrone Datenübertragung gesteuert werden. Um nur einen Schwellwert am *Noise Click* zu setzen und danach nichts mehr durch SPI Bus von *Noise Click* zu bekommen, braucht man nur MOSI Pin (Master Out -> Slave In) zu definieren und die Leitung MISO mit der Zahl - 1 als nicht benutzt zu markieren. Damit spart man einen GPIO Pin des Mikrocontrollers für zukünftige Anwendungen und Erweiterungen. Bei der GPIO Wahl muss man auch das Datenblatt von ESP32 durchlesen, weil manche GPIOs stehen gar nicht zu Verfügung des Entwicklers, da sie schon innen des ESP32 benutzt werden. Manche GPIOs sind uni-direktional und man muss aufpassen, dass man die Richtung richtig programmiert hat.

Als ich die alle GPIO Pins in meinem Programm festgelegt hatte und auf Steckplatine die ESP32 Pins mit dem *Noise Click* Pins verbunden habe, durfte ich den Quellcode endlich ausführen. Als Erstes wollte ich feststellen, ob der Schwellwert richtig gesetzt wurde. Das kann man am Pin 8 von *MCP4921/4922 12-Bit DAC with SPI Interface*¹⁰ mit einem Digitalen Multimeter direkt ermitteln. Dies fand ich sehr spannend, da es zum ersten Mal zu einem reellen Anwendung der ermittelten

⁸<http://ww1.microchip.com/downloads/en/DeviceDoc/20001685E.pdf>

⁹<https://download.mikroe.com/documents/datasheets/spq0410hr5h-b.pdf>

¹⁰<http://ww1.microchip.com/downloads/en/devicedoc/21897b.pdf>

Kenntnisse in Hardware und Elektrotechnik kam und mir zur Verfügung ein reeler Mikrocontroller stand statt die verschiedene Bauteile, die in Kapseln gelötet wurden. Als ich von mir gesetzte und am Pin 8 erschienene Spannung V_{ref} mit einem Digitalen Multimeter gemessen habe, sollte ich feststellen, dass der Wert überhaupt nicht in der Nähe von gewünschten Bereich war, und ziemlich zufällig aussah. Nach einigen Versuchen den Fehler in Quellcode zu finden, wurde mir von meinem Praktikumsleiter empfohlen, es mit einem Oszilloskop anzuschauen, welche Werte durch SPI vom Mikrocontroller geschickt wurde. Mithilfe des Oszilloskops kann man einen Moment anschauen, wenn durch SPI etwas übertragen wird. Die 16 Bits der Information wurden von ESP32 erzeugt und während der Datenübertragung erscheint eine logische 1 als eine Spannung 3.3V und eine logische 0 als die kleine Spannung 0,2V. Die Abbildung 4 zeigt die Übertragung das Steuerwort 0111 0011 1110 1000, das nach der Fehlerfindung korrigiert wurde.

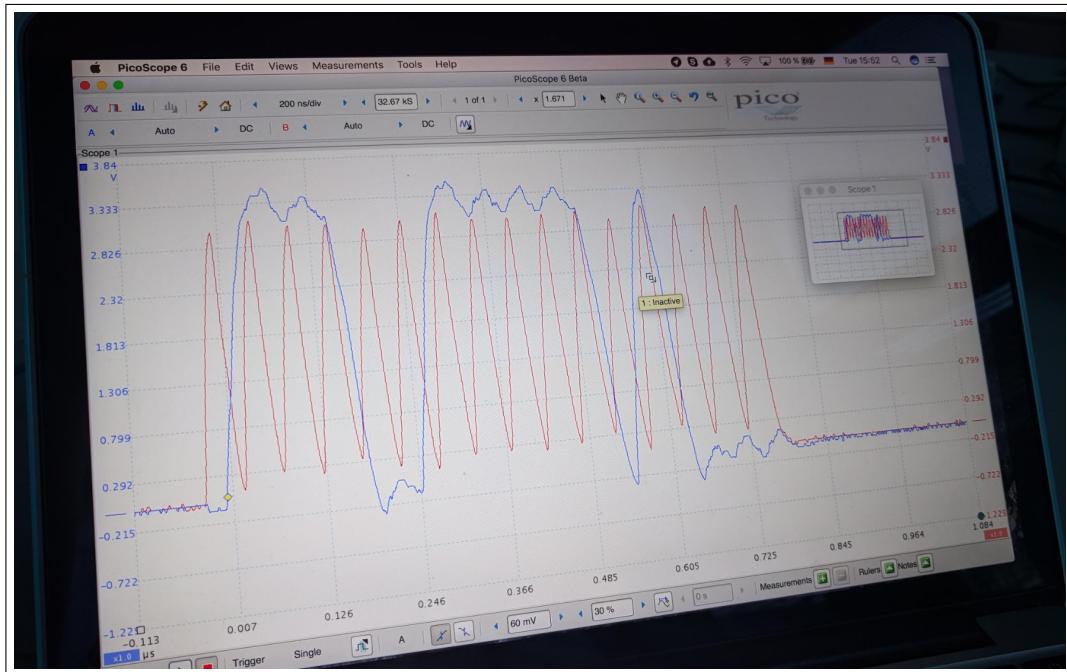


Abb. 4.: Datenübertragung durch SPI

Als der Moment der Datenübertragung eingefangen werden konnte, wurde es klar, dass die ersten 8 Bits des Steuerworts am Ende der Übertragung kamen und am Anfang der Übertragung die letzten 8 Bits des Steuerworts übertragen wurden. Auf diese Weise habe ich ein Problem der Byte-Reihenfolge (Endianness) kennengelernt. Der Grund liegt darin, dass die Rechnerarchitektur und Betriebssystem meines Laptops und ESP32 Xtensa über verschiedene Byte-Reihenfolge verfügen, was ein Problem verursacht hat, indem man ein umgekehrtes Steuerwort bekommt. Ich habe das Problem in Programm gelöst und endlich konnte ich das Steuerwort richtig übertragen.

Nachher sollte ich feststellen, wie man einen Schwellwert setzen muss, um Zimmerlautstärke von Industrie- und Gewerbelärm zu unterscheiden. Nach der Reihe von Versuchs, habe ich festgestellt, dass die V_{ref} höher als 2.5V unmöglich ist für einen Mensch mit einem Schrei zu übersteigen. Obwohl man V_{ref} bis 4.9V setzen darf, kann man tatsächlich bis 3.3V als V_{ref} benutzen, weil die ganze Schaltung von 3,3V versorgt wird und am Ausgang des Mikrofons maximal 3,3V liegen können. Und obwohl die Werten bis 3.3V zur Verfügung stehen, kann man nur die Werte bis 2 - 2.5V als V_{ref} durch SPI setzen, entweder es unmöglich wäre, den Schwellwert zu übersteigen. Tatsächlich liegen die Werte für Zimmerlautstärke zwischen 400 und 800 mV. Die merklich niedrigen Werten von 70 - 150 mV können sogar ein Flüstern erkennen und die Werten höhere als 1V sind geeignet, um sehr laute Geräusche zu erlauben.

Damit wurde festgestellt, dass der Bereich der Schwellenwerte mit 2V begrenzt wird und der Arbeitspunkt bei 0,5V liegt, welcher als ein standardmäßiger Schwellenwert in jedem *Noise Detector* benutzt wird. Nach dem Erwerben des *Noise Detectors* kann Benutzer seinen eigenen Schwellwert setzen. Das manuelle Setzen des Schwellenwertes war ebenfalls eine Teilaufgabe meines Praktikums.

Quellcode Beispiel

```
/*
 * Sends a data to the Noise Click.
 * Creates control word with a custom ref voltage
 */

void spi_to_noise_click_transmit_cmd(spi_device_handle_t spi, int
vInNoiseClick){

    // creating the CW + Reference Voltage
    uint16_t data = ((COMMAND_SPI & 0xF) << 12) | (vInNoiseClick
    & 0xFFFF);
    // solving the issue with big endian and little endian
    data = ((data & 0xFF) << 8) | ((data >> 8) & 0xFF);

    // calculating decimal value V_out, PIN 8, MCP4921, V_ref
    int V_out = ((V_REF_DAC * vInNoiseClick * GAIN_V_REF) / 4096)
    * 1000;
    int V_ref = (V_out * 11) / 13.2;

    printf("SPI: transmit decimal: %d\n", vInNoiseClick);
    printf("SPI: transmit HEX: %x\n", data);
    printf("V_out = %d mV\n", V_out);
    printf("V_ref = %d mV\n", V_ref);
}
```

```

    esp_err_t ret;
    spi_transaction_t t;
    memset(&t, 0, sizeof(t)); //Zero out the transaction
    t.length = 16;           //length is in bits.
    t.rx_buffer = NULL;
    t.tx_buffer = &data;      //Data
    t.user = (void *) 1;

    // Transmit the CW
    ret = spi_device_transmit(spi, &t);
    assert(ret == ESP_OK);
    printf("Transmitted\n\n");
}

```

2.3.2 Hardware Interrupt bearbeiten

Sobald der Schwellenwert der V_{ref} überschritten wird, wird ein Hardware Interrupt am Pin *Int* des *Noise Clicks* erzeugt. Diesen Pin verknüpfte ich mit einem GPIO Pin des Mikrocontrollers, den ich jetzt für die Erkennung der Interrupts programmieren sollte. Dafür stehen die Funktionen des ESP32 Toolchains und FreeRTOS zur Verfügung, die mir erlaubten, die von *Noise Clicks* erzeugten Interrupts zuerst in eine Queue (Schlange) zu sammeln und als eine logische 1 ins Interrupt Service Routine zu schicken. Wenn ein ankommender Interrupt aus der Queue abgewonnen wird, kann Programm dann entsprechend reagieren. Ich brauchte bei jedem Übersteigung des Schwellwerts ein Lämpchen (LED) und den Summer einzuschalten.

Schwierigkeiten und Lösung

Sobald die Programmierung des Interrupts Service Routine erfolgreich abgeschlossen wurde, konnte man beobachten, wie oft die Interrupts ankommen und wie überhaupt die Übersteigung des Schwellenwerts aussieht. Es wurde festgestellt, dass bei menschlichen Gesprächen, wenn die Lautstärke höher als Schwellenwert ist, es zu einer Reihe von Interrupts kam. Der Grund liegt darin, dass ein Mensch nicht in der Lage ist, nach der ersten Meldung des *Noise Detectors* seine Stimme sofort zu mäßigen und bräuchte einige Zeit, um sich zu beruhigen. Diese Tatsache hat mir gezeigt, dass nicht jeden ankommenden Interrupt (also Übersteigung des Schwellenwerts) das Einschalten des Lämpchens und Summers verursachen sollte. So wurde eine zusätzlichen Variable namens *Threshold* hinzugefügt. Mit Hilfe von *Threshold* kann der Benutzer die Empfindlichkeit einstellen. Je niedriger die Empfindlichkeit ist, desto mehr Hardware Interrupts werden ohne Softwaremeldung durchgelassen, was die Verwendung des *Noise Detectors* in einem lauter Raum zu ermöglichen sollte. Je höher die Empfindlichkeit ist, desto schneller wird die Softwaremeldung

angezeigt. Dies sollte dem Zweck dienen, die leise Atmosphäre in einem Raum zu sichern.

Es wurde eine Struktur namens *TInterrupt* erstellt, die nebenbei die Zeit der Hardware Interrupt speichern soll.

```
typedef struct {
    int interruptNo;
    int interruptVoltage;
    int interruptTime;
} TInterrupt;
```

Nach jedem folgenden Interrupt sollte einer Zeitunterschied berechnet werden, der von *Threshold* abhängig ist. Je höhere Empfindlichkeit ist, desto kleiner ist Zeitunterschied, der die Hardware Interrupt in eine Schlange ohne Software Meldung sammeln erlaubt. Zusätzlich ändert sich anhand von *Threshold* die erlaubte Länge der Schlange. Wenn es mehr Hardwareinterrupts als die erlaubte Länge der Schlange vor dem Ablauf eines erlaubten Zeitunterschieds ankommen wird, wird eine Softwaremeldung angezeigt, Lämpchen und Summer eingeschaltet.

```
int check_threshold(TQueue *q, TInterrupt *ir) {
    ...
    remove_outdated_interrupts(q, ir->interruptTime, T_TH3);

    int amountInterrupts = q_length(q);
    TInterrupt *tmpLastInt = q_getlast(q);

    // calculating the elapsed time
    double elapsedTime = ((double) (ir->interruptTime -
        tmpLastInt->interruptTime)) / CLOCKS_PER_SEC;

    if (amountInterrupts >= TH_AMOUNT
        && elapsedTime <= TH_TIME) {
        flag = 1;
    }

    // we didn't exceed the amount of interrupts and time is still
    // running
    else if (amountInterrupts < TH_AMOUNT
        && elapsedTime < TH_TIME) {
        flag = 0;
    }
    ...
}
```

Quellcode Beispiel

```

 ****
 * GPIO Task
 * Detects an interrupt from the ESP32 inner queue ->
 * reads the voltage on the moment of interrupt
 * sets the interrupt flag ->
 * the RTOS tasks can detect the global variable (interrupt flag) and
 * start perform
 * their own interrupt routines. Separately tasks for each hardware
 * and software task
 * allow to switch on/off some functionality regarding to the client
 * needs.
 ****

void gpio_noise_interrupt_task(void* arg) {
    uint32_t io_num;
    while(1) {
        if(xQueueReceive(gpio_noise_interrupt_queue, &io_num,
                         portMAX_DELAY) && io_num == GPIO_INPUT_INTERRUPT) {
            // get voltage from ADC
            detected_voltage = adc1_to_voltage(ADC1_TEST_CHANNEL,
                                                &characteristics);
            printf("%d mV\n", detected_voltage);
            printf("GPIO[%d]: #%d Interrupt\n", io_num, count);

            // add the new interrupt into the queue
            count++;
            detected_time = clock();
            printf("Detected time: %i\n", (int)detected_time);
            TInterrupt *new_interrupt = register_interrupt(count,
                detected_voltage, detected_time);
            q_add(queueInterrupt, new_interrupt);

            // check threshold and setting/resetting the
            // interrupt flag
            detected_interrupt = check_threshold(queueInterrupt,
                new_interrupt);
        }
        else {
            // resetting interrupt flag if no interrupt occurs
            detected_interrupt = 0;
        }
    }
}

```

2.3.3 LED ansteuern

Das Ein- und Ausschalten des Lämpchen gehört bei der Embedded Programmierung zum ersten Schritten, die man mit einem neuen Mikrocontroller überhaupt

macht, um sich mit Toolchain, Programmiersprache und dem Aufbau des konkreten Mikrocontroller vertraut zu machen.

Ich habe dafür eine *FreeRTOS Task* entwickelt, die permanent läuft und in einer Schleife die Variable abfragt, die in Interrupt Service Routine gesetzt werden sollte. Wenn nach der Reihe der Hardwareinterrupts ein Ereignis *Software Interrupt* vorkommt, wird die Variable mit 1 gesetzt und die entsprechenden *FreeRTOS Tasks* können LED und Summer einschalten.

Das LED Anschalten geschieht in ESP32 mit einem GPIO Pin, der auf 1 gesetzt werden soll. Das Ausschalten geschieht mit dem umgekehrten Setzung auf 0.

```
gpio_set_level(GPIO_OUTPUT_BLINK, 1);
```

Interessant war, dass es nicht ausreicht, einfach das Lämpchen einzuschalten und in ein Paar Sekunden wieder auszuschalten. Wenn *FreeRTOS Task* für einige Zeit nichts macht (es gab keine Interrupt, also kein LED Steuerung), dann erscheint einen fatal Fehler bei ESP32:

```
Guru Meditation Error: Core 0 panic'ed (Timeout on CPU1)
CPU halted.
```

ESP32 wurde so vom Hersteller programmiert, dass bei jedem fatal Fehler der Mikrocontroller einen Neustart kriegt. Wenn ein Fehler zufällig ist, stellt der Mikrocontroller sein Programm und die Aufgaben nach einigen Sekunden wieder her. Wenn der Fehler immer wieder vorkommt, kann der Mikrocontroller in eine Schleife geraten. Für meinen Anwendungsfall sollte ich einen Zeitverzug (engl. Delay) für jede *FreeRTOS Task* vorsehen, die Task einfach für die kleinste Zeit verzögern kann. Damit wird es für CPU so aussehen, dass *FreeRTOS Task* seine Aufgabe macht und nicht hängengeblieben ist und keinen Neustart notwendig ist.

```
vTaskDelay(10 / portTICK_RATE_MS);
```

2.3.4 Die Frequenz durch PWM festlegen

PWM steht für (engl.) Pulse Width Modulation und auf Deutsch bedeutet das Pulsbreitenmodulation. Es ist ein bekanntes Verfahren im Embedded Systems, um LEDs und auch Motoren zu steuern. Es wurde bisher in meinem Studiengang nicht behandelt, obwohl es ein Standard heutzutage ist, der sehr leicht zu verwenden und zu verstehen ist.

Die Steuerung des Summers ist mit PWM sehr leicht und besteht im Prinzip aus zwei Schritten. Zuerst muss man die PWM initialisieren, indem man festlegt, welche

Frequenz und welchen Arbeitszyklus man braucht. Außerdem muss man festlegen, welche PWM Einheit von zwei bei ESP32 vorhandenen man benutzen will.

```
*****
* PWM Config FOR NOISE ALARM INTERRUPT
*****  
  
void pwm_modul_config_noiseAlarm(mcpwm_config_t *pwm_config) {
    pwm_config->frequency = FREQUENCY_NOISE_ALARM; // default no
    case frequency
    pwm_config->cmpr_a = DUTY_CYCLE;
    pwm_config->counter_mode = MCPWM_UP_COUNTER;
    pwm_config->duty_mode = MCPWM_DUTY_MODE_1;
    mcpwm_init(MCPWM_UNIT_0, MCPWM_TIMER_0, pwm_config);
    mcpwm_stop(MCPWM_UNIT_0, MCPWM_TIMER_0);
}
```

Die Steuerung durch PWM geschieht auch mit Hilfe eines GPIO Pins, den man als Ausgang programmieren muss. Wenn die entsprechende *FreeRTOS Task* feststellt, dass ein Software Interrupt vorgekommen ist, gibt sie durch Ausgangspin einen Signal mit dem entsprechenden Arbeitszyklus und der Frequenz. Die zwei Parameter sorgen dafür, welchen Klang mit Summer erzeugt wird.

```
mcpwm_start(MCPWM_UNIT_0, MCPWM_TIMER_0);
vTaskDelay(INTERUPT_DELAY / portTICK_RATE_MS);
mcpwm_stop(MCPWM_UNIT_0, MCPWM_TIMER_0);
```

Für die Zeit, in der *FreeRTOS Task* keinen Summer steuert, sollte auch einen Zeitverzug vorgesehen werden.

2.3.5 HTTPS Server und WiFi

Normalerweise ist es sehr leicht, den ESP32 Mikrocontroller mit einem WiFi Netzwerk zu verbinden, weil der Zweck des ESP32 eine leichte Anwendung in Bereich *Interner der Dinge* ist. Jedoch muss der Benutzer dafür eine mobile Anwendung des ESP32-Herstellers verwenden, indem er die Credentials des WiFi Netzwerks (SSID und Password) an den EPS32 übergeben kann. Ich wollte es vermeiden, dass unsere Kundin / unserer Kunde gezwungen wird, eine fremde und zusätzliche mobile Anwendung herunterladen und installieren müssen. Ich wollte es ermöglichen, die Einstellungen für WiFi Netzwerk direkt an den ESP32 übergeben. Außerdem sollten SSID und Password dauerhaft gespeichert werden, nach einem Ausschalten und Einschalten sollte ESP32 in der Lage sein, wieder die Verbindung herzustellen. Dafür wurde die externe Datei von Neil Kolban¹¹ verwendet. Seine cpp-Datei *BootWiFi.cpp*

¹¹<https://github.com/nkolban/>

Select WiFi

SSID:	<input type="text"/>
Password:	<input type="password"/>
IP address:	<input type="text"/>
Gateway address:	<input type="text"/>
Netmask:	<input type="text"/>

The IP address, gateway address and netmask are optional. If not supplied these values will be issued by the WiFi access point.

Abb. 5.: HTTPS Server (Access Point mode)

ermöglichte mir genau das, was ich wollte. Das aber hat mich gezwungen, das ganze Projekt von der Programmiersprache C auf C++ umzustellen.

Es gibt jetzt die folgende Möglichkeiten *Noise Detector* mit einem WiFi Netzwerk zu verbinden:

1. Falls es immer noch keine Credentials für das WiFi Netzwerk gespeichert wurden, läuft *Noise Detector* in diesem Fall als eigenständiger Zugangspunkt (Access Point mode). Das bedeutet, dass das ESP32 zu einem WiFi-Zugangspunkt wird, mit dem sich andere WLAN-Geräte verbinden können. Jetzt kann der Kunde sich über sein Handy oder ein anderes WLAN-Gerät mit dem ESP32 verbinden. Nach der Verbindung, können wir einen Browser öffnen. Auf der Browserseite werden wir nach der SSID und dem Passwort gefragt, die wir später verwenden möchten. Dies wird im *Noise Detector* gespeichert. Anschließend verbindet sich das Gerät mit diesem Netzwerk.
2. Wenn *Noise Detector* in eine neue Umgebung gebracht wird, indem der zuvor gespeicherte Zugangspunkt nicht mehr zugänglich ist oder es einfach unmöglich ist, eine gespeicherte Verbindung herzustellen, wird *Noise Detector* wieder zu einem Zugangspunkt und der Benutzer kann neue Informationen wie oben beschrieben eingeben.
3. Wenn der Benutzer seinen Zugangspunkt ändert will, kann *Noise Detector* einen GPIO-Pin beim Start abfragen. Wenn dieser Pin auf 1 hoch gesetzt wird (standard 0), kann dies den ESP32 wieder in Acces Point Modus versetzen.

2.3.6 Publish und Subscribe durch MQTT Protokoll

MQTT Protokoll kommt zum Einsatz meistens in den Bereichen *Maschine zu Maschine* und *Internet der Dinge* und ist sehr einfach für Menschen zu lesen und für Maschinen zu bearbeiten. Ich erinnere mich daran, dass im Modul *Informatik 2* es am schwierigste war, die Aufgabe mit der Bearbeitung der XML-Datei zu erledigen. Es kam immer zu verschiedenen Fehler und die eigentliche Lösung ist nur in einer XML-Datei anwendbar. Falls sich die XML-Datei ändert, muss der Entwickler sein Programm wieder anpassen.

Während der Übertragung durch MQTT Protokoll verwendet man JSON-Datei, in der alle Variables als entsprechende *Methods* mit gesetzten *Values* übertragen wird. Mit der Hilfe der C-Bibliothek *cJSON.h* kann man sehr leicht eine JSON-Datei erzeugen, die neue Werte hinzufügen, die ankommende JSON-Datei lesen und die notwendigen Werte erkennen und speichern.

```
void parseJsonCommand(char *json, command_holder_t *ch) {
    ...
    if (strcmp(method_name, "setVoltageValue") == 0) {
        param = cJSON_GetObjectItemCaseSensitive(monitor_json
            , "params");

        if (cJSON_IsString(param) && param->valuestring != NULL) {
            ch->method = METHOD_SET_VOLTAGE;
            ch->voltage = atoi(param->valuestring);
        }
    } else if (strcmp(method_name, "setSoundValue") == 0) {
        param = cJSON_GetObjectItemCaseSensitive(monitor_json
            , "params");
        if (cJSON_IsBool(param)) {
            ch->method = METHOD_SET_SOUND;
            ch->set_sound = cJSON_IsTrue(param);
        }
    } else if ...
}
```

Lösung

Eptecon hat eine eigene Webanwendung namens GlueLogics¹² basierend auf der Open Source IoT-Plattform ThingsBoard eingerichtet, um seine Geräte als *Connected Devices* zu verwenden. Jedes Gerät wird bei GlueLogics als *Connected Device* registriert und enthält damiteinen Token für Zugang (oder Zugriff). Mithilfe dieses

¹²<http://gluelogics.com>

Tokens und TLS-Protokolls (Transport Layer Security) geschieht eine Verbindung durch MQTT Protokoll zwischen *Noise Detector* und GlueLogics. Nach der erfolgreichen Programmierung, wurde endlich die Verbindung hergestellt und die an die Anwendung übertragenen Werte erscheinen in *Latest Telemetry* Fenster.

Last update time	Key ↑	Value
2018-07-19 00:18:06	getBatteryValue	95
2018-07-19 00:18:06	getLEDValue	false
2018-07-19 00:18:06	getSoundValue	true
2018-07-19 00:18:06	getThresholdValue	5
2018-07-19 00:18:06	noise_level	272

Abb. 6.: Latest Telemetry Fenster

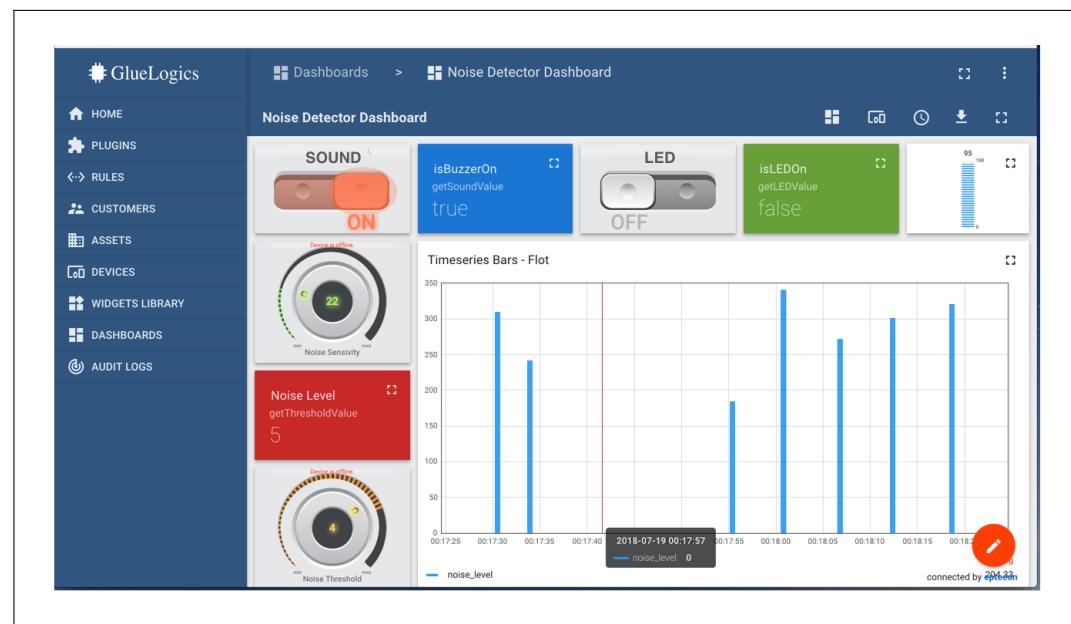
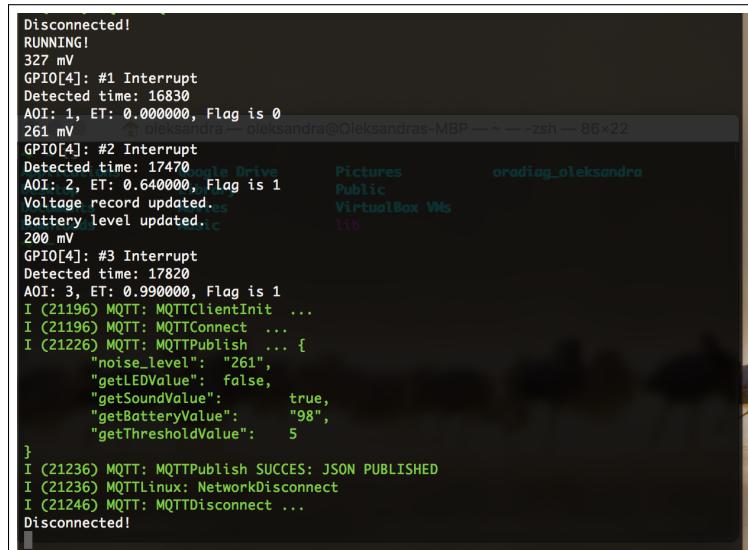


Abb. 7.: Noise Click Dashboard mit Widgets

Jetzt kann man die gewünschte Variable wählen und sie auf einem Widget zeichnen lassen. Ich habe das Fenster (Abbildung 7) für *Noise Detector* eingerichtet, in dem man sowohl den Summer und die LED ein- und ausschalten kann als auch die Einstellungen für den Summer und die LED überprüfen kann. Das wichtigste ist, natürlich, dass *Noise Detector* jetzt einen festgestellten Lärmwert direkt an die Cloud in die GlueLogics Anwendung überträgt und in einem Zeitgraphen anzeigt. Mit den ausgeschalteten Summer und LED kann Administrator trotz beobachten, wie laut in Raum ist.

Quellcode Beispiel

```
*****  
* MQTT Update JSON-File if Interrupt occurs  
*****  
  
void mqtt_interrupt_task(void *arg){  
    while(1) {  
        if (detected_interrupt) {  
            mqtt_update_json_voltage(detected_voltage);  
            mqtt_update_json_battery();  
            mqtt_publish(TOPIC2, mqtt_msg);  
            detected_interrupt = 0; }  
        else {  
            detected_interrupt = 0;  
            vTaskDelay(10 / portTICK_RATE_MS); }  
    }  
}
```



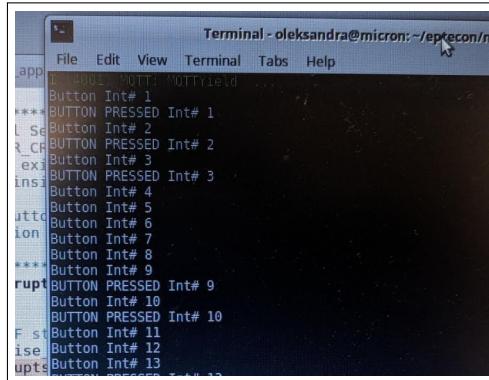
The screenshot shows a terminal window with the following log output:

```
Disconnected!  
RUNNING!  
327 mV  
GPIO[4]: #1 Interrupt  
Detected time: 16830  
AOI: 1, ET: 0.000000, Flag is 0  
261 mV  
GPIO[4]: #2 Interrupt  
Detected time: 17470  
AOI: 2, ET: 0.640000, Flag is 1  
Voltage record updated.  
Battery level updated.  
200 mV  
GPIO[4]: #3 Interrupt  
Detected time: 17820  
AOI: 3, ET: 0.990000, Flag is 1  
I (21196) MQTT: MQTTClientInit ...  
I (21196) MQTT: MQTTConnect ...  
I (21226) MQTT: MQTTPublish ... {  
    "noise_level": "261",  
    "getLEDValue": false,  
    "getSoundValue": true,  
    "getBatteryValue": "98",  
    "getThresholdValue": 5  
}  
I (21236) MQTT: MQTTPublish SUCCES: JSON PUBLISHED  
I (21236) MQTTLinux: NetworkDisconnect  
I (21246) MQTT: MQTTDisconnect ...  
Disconnected!
```

Abb. 8.: MQTT Publish Beispiel

Die Abbildung 8 zeigt den Programmverlauf bei einem erfolgreichen MQTT Publish Ereignis.

2.3.7 Die Drucktasten und Entprellung



The screenshot shows a terminal window titled "Terminal - oleksandra@micron: ~esp32con/r". The window displays a continuous stream of button press events. The text in the terminal reads:

```
File Edit View Terminal Tabs Help
app
I... 401 NOTT: NOTTYield
**** Button Int# 1
Button Int# 2
Button Int# 3
Button Int# 4
Button Int# 5
Button Int# 6
Button Int# 7
Button Int# 8
**** Button Int# 9
Button Int# 10
Button Int# 11
Button Int# 12
Button Int# 13
```

Abb. 9.: Die entprellten Drucktaste

Während der Programmierung der Connectivity des *Noise Clicks* kam es zur Notwendigkeit das Gerät mit einer Drucktaste zu versorgen, um es zu ermöglichen, die Credentials des jeweiligen WiFi-Netzwerks zu ändern. Ich habe auf das Problem der Tastenprellung gestoßen. Bei einmaligen Tastendruck registrierte mein Programm vielmals darauf.

Ich fand es spannend, das Problem in der Programmiersprache C zu lösen, weil ich die Entprellung während meines Studiums kennengelernt habe und in VHDL schon erledigt habe. Es gab keine großen Schwierigkeiten, das Problem zu lösen und die Entprellung der Taste richtig zu programmieren.

2.3.8 Dauernde Speicherung in NVS

Um es dem Benutzer zu ermöglichen, seine Einstellungen am *Noise Detector* sowohl übertragen als auch dauerhaft speichern, sollte non-volatile storage (NVS) library benutzt werden. NVS bedeutet aud Deutsch *Nichtflüchtiger Speicher* und bezieht sich auf einen Flash-Speicher (ähnlich wie in USB Sticks), der Daten permanent speichern kann, wenn keine Stromversorgung vorhanden ist, und keine periodischen Aktualisierungen seiner Speicherdaten erfordert. ESP32 NVS arbeitet mit Schlüssel-Wert-Paaren. Schlüssel sind ASCII-Zeichenfolgen, die maximale Schlüssellänge beträgt 15 Zeichen. Werte können einen der folgenden Typen aufweisen:

- integer types: $uint8_t, int8_t, uint16_t, int16_t, uint32_t, int32_t, uint64_t, int64_t$
- zero-terminated string
- variable length binary data (blob)

Für das Projekt *Noise Detector* brauchte ich nur bestimmte Einstellungen speichern. Es wurde festgestellt, dass der Benutzer es wünschen könnte, sein *Noise Detector* ohne einen Klang des Summers und/oder LED einzustellen. Zusätzlich möchte man den Schwellenwert V_{ref} und die Empfindlichkeit $Threshold$ speichern. Die Möglich-

keit die Einstellungen nach dem Neustart oder dem Ausschalten wiederherzustellen macht *Noise Detector* zu einem fertigen Gerät, das dem Benutzer zur Verfügung steht. Abbildung 10 zeigt, wie das Programmablauf bei Neustart aussieht. Nach

```

Opening Non-Volatile Storage (NVS) handle... Done
Reading reference voltage from NVS ...Done
NVS Value = 630

SPI: transmit decimal: 630
SPI: transmit HEX: 7672
V_out = 353 mV
V_ref = 294 mV
Transmitted

Opening Non-Volatile Storage (NVS) handle... Done
Reading noise threshold from NVS ...Done
NVS Value = 5

JSON created.
Voltage record added to JSON.
LED status record added to JSON.
Buzzer status record added to JSON.
Battery record added to JSON.
Threshold record added to JSON.

Opening Non-Volatile Storage (NVS) handle... Done
Reading LED settings from NVS ...Done
NVS Value = 0 oleksandra — oleksandra@Oleksandras-MBP ~ ~ -zsh ~ 86x22

LED alarm signal is turned off.
Current LED Status updated.

Opening Non-Volatile Storage (NVS) handle... Done
Reading buzzer settings from NVS ...The value is not initialized yet!
Buzzer alarm signal is turned on.
Current Buzzer Status updated.
I (4466) wifi: pm start, type:0

I (6726) MQTT: MQTTClientInit ...
I (6726) MQTT: MQTTConnect ...
I (6776) MQTT: MQTTPublish ... {
    "noise_level": 0,
    "getLEDValue": false,
    "getSoundValue": true,
    "getBatteryValue": 100,
    "getThresholdValue": 5
}
I (6776) MQTT: MQTTPublish SUCCES: JSON PUBLISHED
I (6786) MQTTlinux: NetworkDisconnect
I (6786) MQTT: MQTTDisconnect ...
Disconnected!

```

Abb. 10.: NVS Meldungen bei Neustart

GPIO Initialisierung wird Nichtflüchtiger Speicher geöffnet und die gespeicherte Werte entsprechend ihren Schlüsseln gesucht. Falls der jeweilige Schlüssel gesetzt ist und der Wert ausgelesen werden kann, wird den Wert als Variable in die entsprechenden Funktionen übertragen und die Funktionen werden ausgeführt. Zum Beispiel, falls V_{ref} von Benutzer eingestellt wurde, wird durch SPI Bus MOSI Pin der gespeicherte Wert am *MCP4921/4922 12-Bit DAC*¹³ übertragen. Damit wird bei jedem Neustart den gespeicherten Schwellenwert wiederherstellt und *Noise Click* wird immer mit den Einstellungen des Benutzers starten. Falls es keine Spannung V_{ref} gespeichert wurde (also existiert keinen Schlüssel namens VARIABLE SET VOLTAGE) wird ein standardmäßiger Schwellenwert 0.5V durch SPI übertragen. Damit können wir sicherstellen, dass im Fall eines Fehlers *Noise Detector* seine standardmäßigen Einstellungen wiederherstellt wird.

¹³<http://ww1.microchip.com/downloads/en/devicedoc/21897b.pdf>

Quellcode Beispiel

```
switch (varType) {
    case NO_VARIABLE:
        break;
    case VARIABLE_SET_VOLTAGE:
        printf("Updating reference voltage in NVS ... ");
        err = nvs_set_i16(handle, "refVoltage", value);
        break;
    case VARIABLE_SET_SOUND:
        printf("Reading reference voltage from NVS ... ");
        err = nvs_get_i16(handle, "buzzer", value);
        break;
    case VARIABLE_SET_LED:
        printf("Reading reference voltage from NVS ... ");
        err = nvs_get_i16(handle, "led", value);
        break;
    case VARIABLE_SET_THRESHOLD:
        printf("Reading reference voltage from NVS ... ");
        err = nvs_get_i16(handle, "threshold", value);
        break;
    default:
        break;
}
```

2.3.9 PCB Design und Aufbau der Hardware

Als ich mit der Programmierung und Testen fertig war, kam es endlich zu einem voll unbekannten und interessanten Aufgabe: Eine Leiterplatte zu bauen. Dafür habe ich die Software *EAGLE CAD*¹⁴ verwendet. Bevor ich beginnen konnte, Leiterbahnen und andere Dinge zu zeichnen, musste ich wissen, welche Schaltung ich bauen möchte. Ich musste also Schaltplan für meine Schaltung entwerfen.

Dafür habe ich die alle Datenblätter von *ESP32 Mikrocontroller, Noise Click und Buzz Click* wieder durchgelesen und dann alle benötigten Komponenten (Kondensatoren, Widerstände, Summer, Mikrofon u.s.w) zu meinem Schaltplan hinzugefügt. Für viele Sachen kann man schon existierende Bibliotheken von Herstellern benutzen. Ich musste nur eine lbr-Datei für Summer selbst entwerfen, weil diese im EAGLE Format noch nicht gab. Während des Entwurfs sollte ich den Summer genau Vermessen (Abbildung 11), damit er auf Leitplatine richtig positioniert werden kann.

Dann sollte man genau alle GPIO Pins, die in Projekt verwendet wurden, mit entsprechenden Ein- und Ausgängen anderer Bauteile zu verknüpfen. Hier muss man immer aufpassen und alles richtig machen, damit die Leitplatine nach der Herstellung und

¹⁴<https://www.autodesk.com/products/eagle/overview>

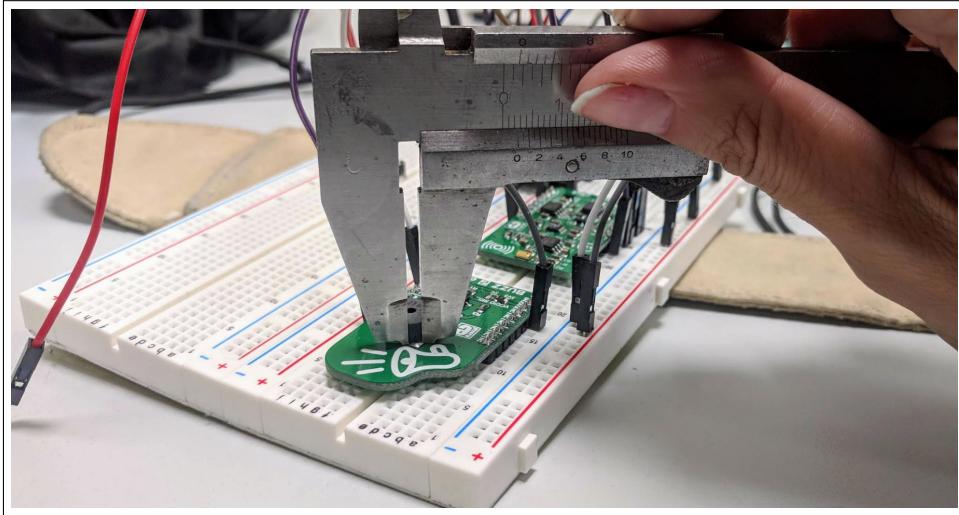


Abb. 11.: Messung von Summe

Löten funktioniert. Abbildung 12 zeigt den fertigen Schaltplan in Eagle Software. Es ist ESP32 zu sehen, der schon mit allen Bauteilen verbunden ist.

Nach der Fertigung des Schaltplans, fängt man mit dem Design der Leiterplatte an. Ich musste mein Schaltbild in eine Zeichnung meiner Leiterplatte übertragen. Man muss sich viele Gedanken darüber machen und viele Sachen während des PCB Designs überlegen. Zum Beispiel, wie wir die Leiterplatte in ein Gehäuse stecken werden. Welche Bauteile wie platziert werden müssen, um in unsere festgelegte Leiterplattengröße zu passen. Wenn alle Fragen geklärt wurden, konnte man mit dem PCB Design weitermachen.

Ich habe die Bauteile wie Kondensatoren und Widerstände entsprechend der BOM-Datei (BOM bedeutet Bill Of Materials - oder auf Deutsch: Stückliste) auf der Mouser.de¹⁵ Website zum Bestellung ausgewählt. BOM-Datei wird automatisch aus dem Schaltplan von Eagle erstellt. Die Datei ist wie eine Einkaufsliste für elektronisches Design. Sie enthält alle Bauteile, die für die Fertigstellung PCB erforderlich sind. Aber statt einer allgemeinen Einkaufsliste mit einem Ein-Wort-Wert für einen Artikel enthält ein Artikel in der BOM-Datei viel Information, die es erleichtert, die richtigen Bauteile zu identifizieren und zu kaufen.

Zum Schluss meines Praktikums wurde die Leiterplatte bei einem spezialisierten Fertiger bestellt. Die Fertigung sollte noch zwei Wochen nach dem Ablauf meines Praktikums dauern. Das Projekt *Noise Detector* wurde voll programmiert und eine Leitplatine dafür entwickelt.

¹⁵<https://www.mouser.de/>

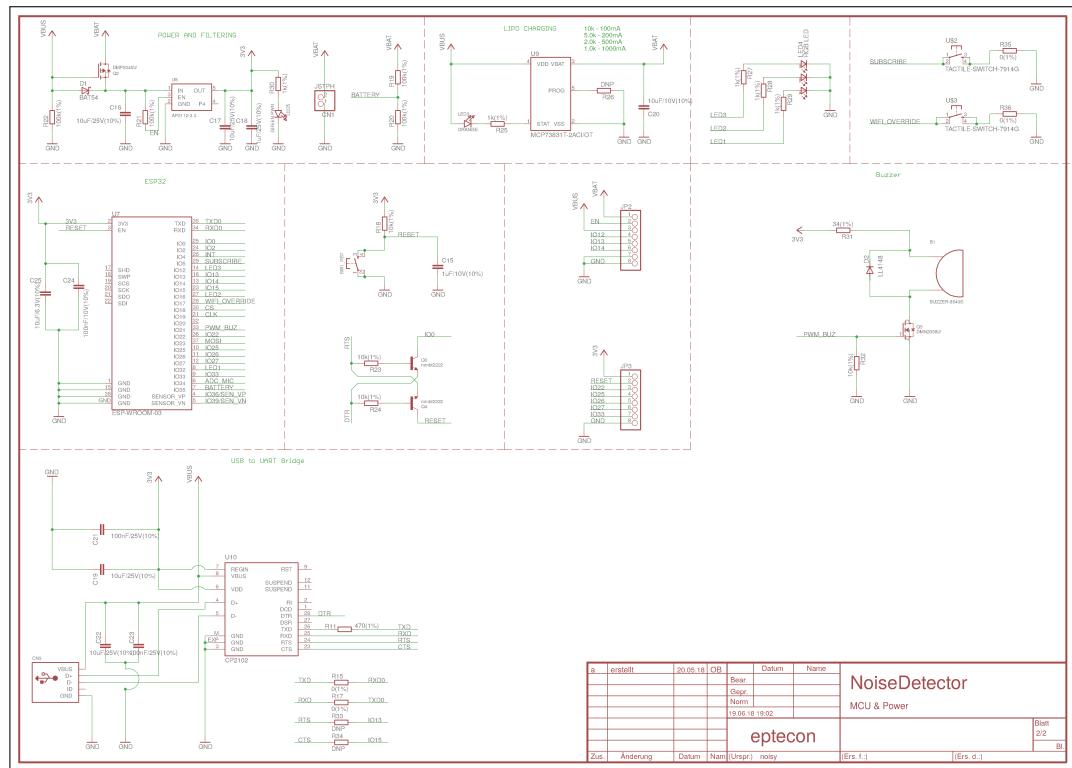


Abb. 12.: Schaltplan des Noise Clicks

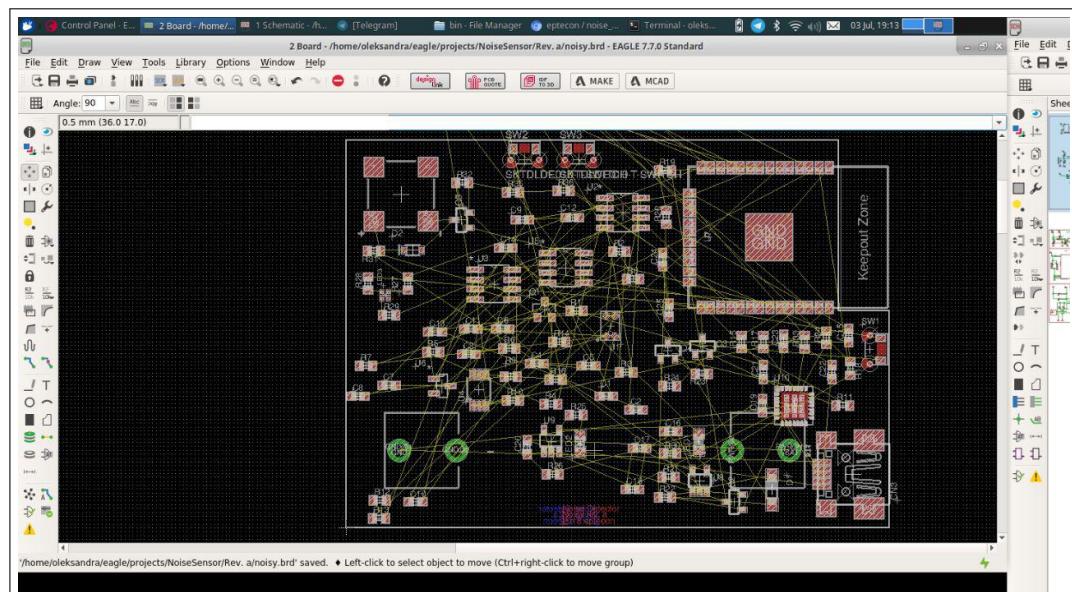


Abb. 13.: Leiterplatte des Noise Clicks

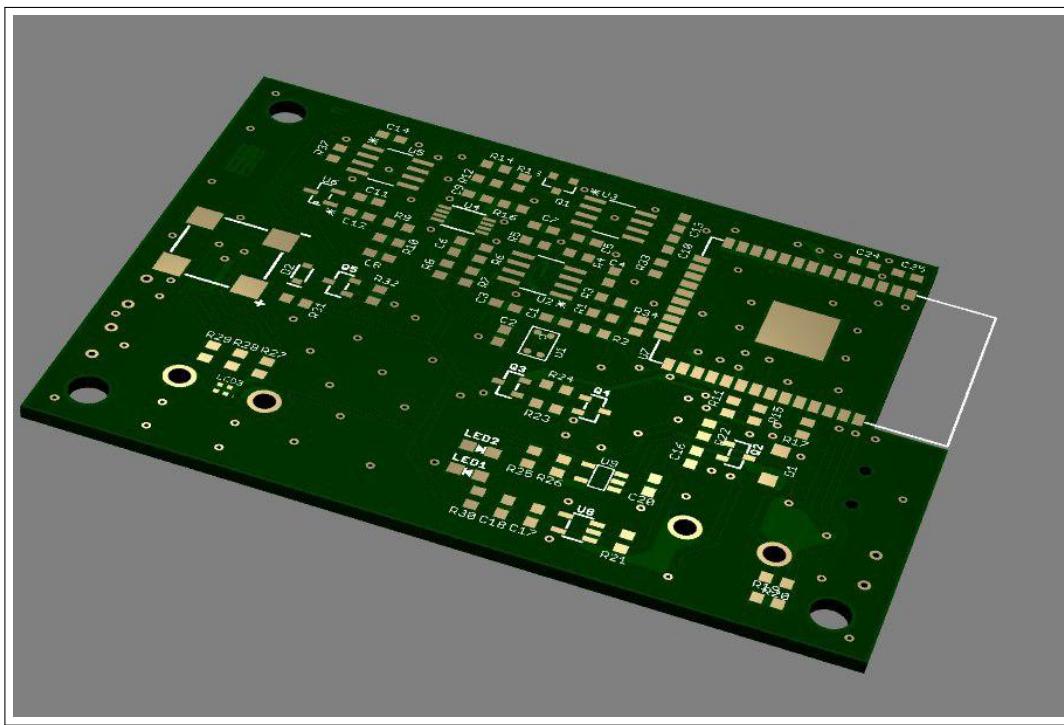


Abb. 14.: 3D Visualisation

A	B	C	D	E	F	G	H	I	J	
Part	Value	Device	Package	Mouser-Nr.	Price 1	Price 5	Description	Datasheet	Mouse.R.D	DIG
1										
2	B1	BUZZER-8540S	BUZZER-8540S	BUZZER-8540S	490-CMT18R61A105KA61J	2.590	2.240	MAGNETIC BUZZER TRANSDUCER	Link	Link
3	C1	1uF/10V(10%)	C-EUC0603	C0603	81-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
4	C2	100nF/10V(10%)	C-EUC0603	C0603	81-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
5	C3	1uF/10V(10%)	C-EUC0603	C0603	81-CRM18R61A105KA61J	0.090	0.074	CAPACITOR, European symbol	Link	Link
6	C4	1uF/10V(10%)	C-EUC0603	C0603	81-CRM18R61A105KA61J	0.090	0.074	CAPACITOR, European symbol	Link	Link
7	C5	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
8	C6	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
9	C7	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
10	C8	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
11	C9	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
12	C10	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
13	C11	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
14	C12	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
15	C13	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
16	C14	1uF/10V(10%)	C-EUC0603	C0603	81-CRM18R61A105KA61J	0.090	0.074	CAPACITOR, European symbol	Link	Link
17	C15	1uF/10V(10%)	C-EUC0603	C0603	81-CRM18R61A105KA61J	0.090	0.074	CAPACITOR, European symbol	Link	Link
18	C16	10uF/25V(10%)	C-EUC0603	C0603	80-C0603C103K3RAC	0.082	0.074	CAPACITOR, European symbol	Link	Link
19	C17	10uF/25V(10%)	C-EUC0603	C0603	80-C0603C103K3RAC	0.082	0.074	CAPACITOR, European symbol	Link	Link
20	C18	1uF/25V(10%)	C-EUC0603	C0603	963-TMK107B7105KA-T	0.123	0.074	CAPACITOR, European symbol	Link	Link
21	C19	10uF/25V(10%)	C-EUC0603	C0603	80-C0603C103K3RAC	0.082	0.074	CAPACITOR, European symbol	Link	Link
22	C20	10uF/10V(10%)	C-EUC0603	C0603	81-CRM18R61A105KA61J	0.090	0.074	CAPACITOR, European symbol	Link	Link
23	C21	100nF/25V(10%)	C-EUC0603	C0603	810-CGA3E2X8R1E104K	0.148	0.101	CAPACITOR, European symbol	Link	Link
24	C22	10uF/25V(10%)	C-EUC0603	C0603	80-C0603C103K3RAC	0.082	0.074	CAPACITOR, European symbol	Link	Link
25	C23	100nF/25V(10%)	C-EUC0603	C0603	810-CGA3E2X8R1E104K	0.148	0.101	CAPACITOR, European symbol	Link	Link
26	C24	100nF/10V(10%)	C-EUC0603	C0603	80-C0603C104K8RAUTO	0.090	0.074	CAPACITOR, European symbol	Link	Link
27	C25	10uF/10V(10%)	C-EUC0603	C0603	81-CRM18R61A105KA61J	0.090	0.074	CAPACITOR, European symbol	Link	Link
28	CN1	JSTPH	JSTPH2					JST 2-Pin Right-Angle Connector		
29	CN3	USBMINIBLARGE	USB-MINIB_LARGER					USB Connectors		
30	D1	BAT54	BAT54	SOT23				Schottky Diodes		
31	D2	LL4148	DIODE-MELF-MLL41	MELF-MLL41				DIODE		
32	JP2	PINHD-X8	PINHD-X8	1X8				PIN HEADER		
33	JP3	PINHD-X8	PINHD-X8	1X8				PIN HEADER		

Abb. 15.: BOM Datei des Noise Clicks: Kondensatoren

Fazit

3.1 Praktikum und Studium

Die im Bachelorstudiengang Technische Informatik erworbenen Grundlagen und Fähigkeiten ermöglichen es mir, Aufgaben im Praktikum zu erfüllen. Besonderer Fokus lag hier auf den praktischen Erfahrungen mit den Programmiersprachen C und C++, Analog- und Digitaltechnik, sowie grundlegenden Problemlösungsstrategien und die Fähigkeiten mit dem neuen Stoff umzugehen und die Lösungen der Probleme zu finden, die ich während meines Studiums nicht betrachtet habe.

Ein Beispiel hierfür ist die Beherrschung der Software *EAGLE* für das Design und Fertigstellung des Schaltplans, Aufbau der Platine und Festlegung der Komponenten in der Datei namens "Bill of Materials".

Von den Modulen, die ich an der Beuth Hochschule schon absolviert habe, besonders hilfreich für meinen Praktikumsablauf waren *Systemprogrammierung*, *Maschinenorientierte Programmierung*, *Analoge Elektronik (Elektrische Systeme III)* und *Mikrocomputertechnik*. Da auf ESP32 FreeRTOS läuft, habe ich viele Kenntnisse aus dem Modul *Echtzeitsysteme* in der Realität verwendet. Letztendlich wären die Aufgaben, die der von mir programmierte *Noise Detector* ausführen konnte, ohne mehreren parallelen Tasks nicht möglich. Obwohl während des Studiums FreeRTOS nicht genau betrachtet wurde, konnte ich leicht nachvollziehen, wie man die Tasks in der RTOS Umgebung programmiert.

Während meines Praktikum habe ich festgestellt, dass die Programmierung mir den größten Spaß macht und der Schwerpunkt meiner zukünftigen Arbeit wird. Hier wurde mein Interesse für die Embedded Programmierung unterstützt und die Neugier für die wachsende Branche der IoT-Geräte geweckt.

3.2 Bewertung des Praktikums

Mein Praktikum bei Eptecon hatte eine angenehme, offene Atmosphäre. Als Praktikantin kriegte ich alle meine Fragen von meinem Praktikumsleiter und anderen

Mitarbeitern beantwortet. Ich fühlte mich nie unter Druck gesetzt etwas einfach nur schnell abarbeiten zu müssen, sondern es wurde viel Wert auf Verständnis und das Erlernen neuer Fähigkeiten gelegt.

Ein Beispiel hierfür ist, dass mir die Zeit gegeben wurde, mich ganz genau Zeit mit der Verwendung der Software *EAGLE* zu befassen. Am Anfang meines Praktikums hatte ich auch entsprechend genug Zeit, um alle Datenblätter durchzulesen, Tool Chain zu installieren und mich mit dem Mikrocontroller ESP32 vertraut zu machen.

Es war eine sehr anregende Zeit, in der ich sehr viel lernte und viele neue Fachgebiete kennenlernenlehrte. Ich habe mich gefreut, dass ich auch schon viel an der Beuth Hochschule gelernt habe, was man in der Praxis bei einer realen Arbeit verwenden kann und muss. Ich weiß jetzt genau, worin ich die Vertiefung machen will.

Anlagen

A.1 mikroBUS

Der Standard legt das physikalische Layout der mikroBus-Pinout-Verbindung, die verwendeten Kommunikations- und Stromversorgungspins auf dem Mainboard fest. Der Zweck von mikroBUS ist es, eine einfache Erweiterbarkeit der Hardware mit einer großen Anzahl von standardisierten kompakten Zusatzboards zu ermöglichen, von denen jede einen einzelnen Sensor, Display, Encoder oder Motortreiber, eine integrierte Schaltung hat. Der von MikroElektronika entwickelte mikroBUS ist ein offener Standard - jeder kann mikroBUS in seinem Hardwaredesign implementieren. Die Abbildung¹ 16 zeigt die Pinout Spezifikation des Herstellers, die man entsprechend ändern kann und die neuen Verbindungen für den eigenen Projekt feststellen. Wenn ein Modul eine Schnittstelle verwendet, die bereits auf mikroBUS vorhanden ist, benutzt man diese exakten Pins und markiert diese entsprechend. Wenn ein Pin nicht verwendet wird, sollte er als NC (für "Not Connected") markiert sein.

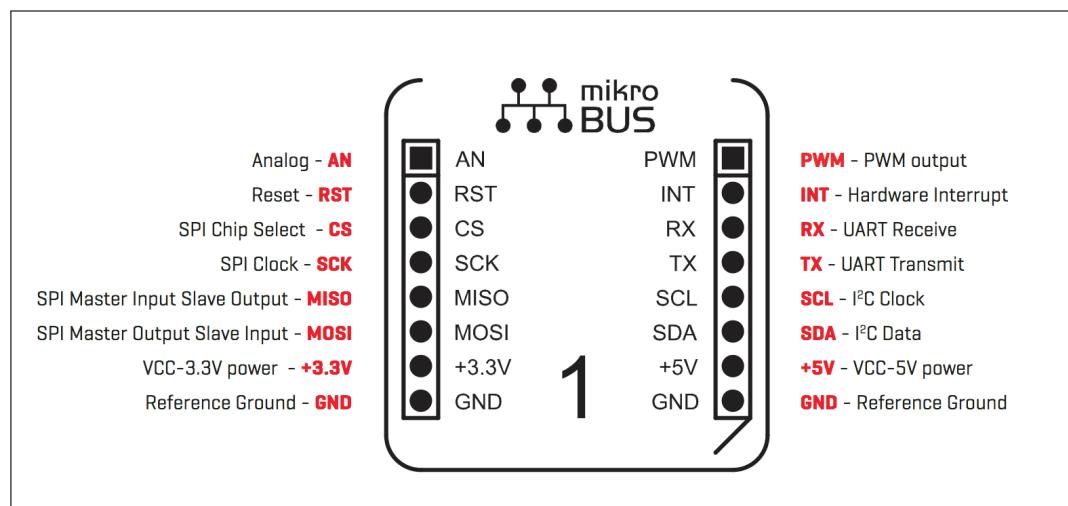


Abb. 16.: MikroBus Pinout Spezifikation

¹<https://download.mikroe.com/documents/standards/mikrobus/mikrobus-standard-specification-v200.pdf>

A.2 Cloud Computing

Die *Cloud* oder *Cloud Computing* Begriff kommt offensichtlich aus dem Englischen und heißt auf Deutsch "Wolke". Der Begriff beschreibt einen oder mehrere entfernte Server, auf die man seine Daten von einem Gerät über das Internet hochladen kann. Dann übernimmt die Cloud die Aufgaben wie die Datenverarbeitung oder komplizierte Programmabläufe. Während der Datenverarbeitung weiß der Nutzer nicht, wie viele Server hinter der Cloud stecken und welche komplizierte Hardware für die Berechnungen benötigt werden.

Der Name *Cloud* stammt davon, dass Cloud Computing quasi wolkig (unklar) für den Nutzer ist, weil ein Nutzer nicht wissen muss und kann, wo sich der Server physikalisch befindet. Selbst wenn ein Server ausfällt, hat dies keine Auswirkungen auf das gesamte System und diese Eigenschaft nennt man "Wolke". Cloud hilft bei IoT-Anwendungen, da die IoT-Hersteller die anfallenden Datenmengen mit der eigenen IT-Infrastruktur in der Regel nicht bewältigen und umfassend nutzen können. Cloud bietet die nötige Flexibilität, schnelle Skalierbarkeit und ständige Verfügbarkeit ohne langfristige, kapitalintensive Investitionen in eigene Rechenzentren.

Cloud erlaubt auch mehreren Nutzer einen gemeinsamen Zugriff auf die auf dem Server gespeicherten Dateien zu bekommen. So kann man mit einem berühmten Cloud Service von Google namens "*Google Drive*"² an einer Präsentation oder anderen Dokumenten zusammenarbeiten. Man kann die von den jeweiligen Benutzern am Dokument vorgenommenen Änderungen in Echtzeit anzeigen.

A.2.1 Open-source IoT platform ThingsBoard

ThingsBoard³ ist eine Open-Source-IoT-Plattform zur Erfassung, Verarbeitung, Visualisierung und Verwaltung der Daten. Es ermöglicht Gerätekonnektivität über Industriestandard-IoT-Protokolle - MQTT, CoAP und HTTP - und unterstützt sowohl Cloud- als auch lokale Bereitstellungen. ThingsBoard kombiniert Skalierbarkeit, Fehlertoleranz und Hardware- und Software Leistungen.

Mit ThingsBoard kann man umfassende IoT-Dashboards für die Datenvisualisierung und Remote-Gerätesteuerung in Echtzeit erstellen. Während der Entwicklung des Projekts "*Noise Detector*" die Leistungen von ThingsBoard wurden dafür verwendet, um die Einstellungen auf entwickelten Gerät zu speichern. Das wurde erfolgreich mithilfe JSON-Datei realisiert, deren Format in Abschnitt A.3 auf Seite 31 kurz erklärt wird.

²<https://drive.google.com/drive/my-drive>

³<https://thingsboard.io>

ThingsBoard ist unter Apache License 2.0 lizenziert, so dass man es kostenlos in seinem kommerziellen Produkten verwenden kann. Es hat auch eine wichtige Rolle für das Projekt gespielt, da "Noise Detector" ohne Finanzierung für eigenes Geld des Unternehmens entwickelt wurde. Die Abbildung zeigt ein Beispiel der Verwendung von ThingsBoard für Visualisierung⁴.

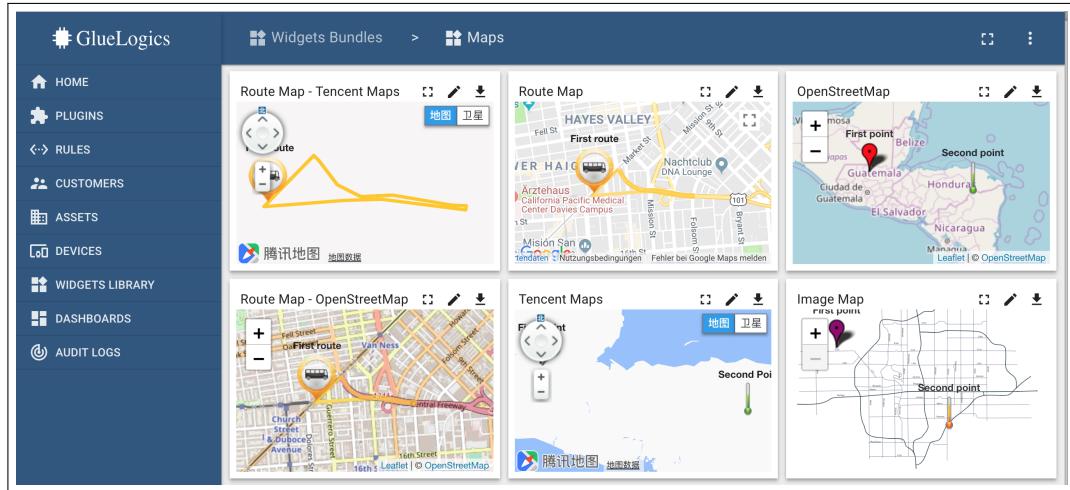


Abb. 17.: ThingsBoard Beispiel: Maps Widgets

A.3 JSON-Datei

JSON⁵ ist die Abkürzung für JavaScript Object Notation, was auf Deutsch bedeutet "JavaScript Objekt-Bezeichnung". Eine JSON-Datei speichert Daten ähnlich wie CSV-, Javascript oder XML-Datei. Sie ist einfach für Menschen zu lesen und zu schreiben. Sie ist einfach für Maschinen zu parsen (Analysieren von Datenstrukturen) und zu generieren. Diese Eigenschaft spielt eine wichtige Rolle, weil sie viel den Arbeitsaufwand für Menschen ersparen kann, womit die Daten in verschiedenen Formaten und aus unterschiedlichen Schnittstellen schnell und leicht zu verarbeiten sind. Dank dieser Eigenschaft in der Zeit des Internets der Dinge wird die Auszeichnungssprache XML zunehmend von einfacheren Formaten wie z. B. JSON ersetzt. ThingsBoard arbeitet mit JSON-Datei und die Datei-Format wurde für das Projekt meines Praktikums verwendet, um die Einstellungen auf unseren IoT-Gerät zu ändern. Mithilfe der JSON-Datei kann der Benutzer die Empfindlichkeitsgrenze des Geräts ändern. So kann man ein Gerät entweder für die Bemessung und Feststellung von leisen Geräusche oder für die Arbeit in sehr lauten Räumen so einstellen, dass nach der Überschreitung des Schwellenwerts ein automatischer Alarm ausgelöst wird. Mit

⁴<http://gluelogics.com/widgets-bundles>

⁵<https://www.json.org/>

JSON-Datei kann man zusätzlich einen von mit programmierten Warnsummer ein- oder ausschalten.

Beispiel einer JSON Datei

```
"title": "Noise Detector Dashboard",
"configuration": {
    "description": "ESP32",
    "name": "Noise Detector Dashboard"
    "widgets": {
        "settings": {
            "stateControllerId": "entity",
            "showTitle": false,
            "showDashboardsSelect": true,
            "showEntitiesSelect": true,
            "showDashboardTimewindow": true }}}
```

A.4 SPI Bus

Das Serial Peripheral Interface, kurz SPI ist ein Bussystem, das aus drei Leitungen für eine serielle synchrone Datenübertragung besteht:

- MOSI (Master Out -> Slave In);
- MISO (Master In <- Slave Out);
- SCK (Serial Clock) - Schiebetakt

Zusätzlich zu diesen drei Leitungen benutzt man für jeden Slave eine Slave Select (SS) oder auch Chip Select (CS) genannte Leitung, durch die der Master den Slave zur aktuellen Kommunikation selektiert.

A.5 PWM

PWM steht für (engl.) Pulse Width Modulation und heißt eigentlich übersetzt Pulsbreitenmodulation. PWM wird zum Erzeugen eines analogen Signals anhand einer digitalen Signalquelle benutzt. Ein pulsweitenmoduliertes Signal (PWM-Signal) ist durch zwei Hauptkomponenten charakterisiert: Tastverhältnis und Frequenz. Das Tastverhältnis ist das Verhältnis der High-Dauer des Signals zur Gesamtdauer des Impulses. Die Frequenz bestimmt, wie schnell eine Periode durchlaufen wird (1000 Hz entspräche z. B. 1000 Perioden pro Sekunde), und gibt somit an, wie schnell das

Signal zwischen High und Low wechselt. Die Abbildung⁶ zeigt wie unterschiedlich ein Signal aussehen kann, wenn man Tastverhältnis ändert.

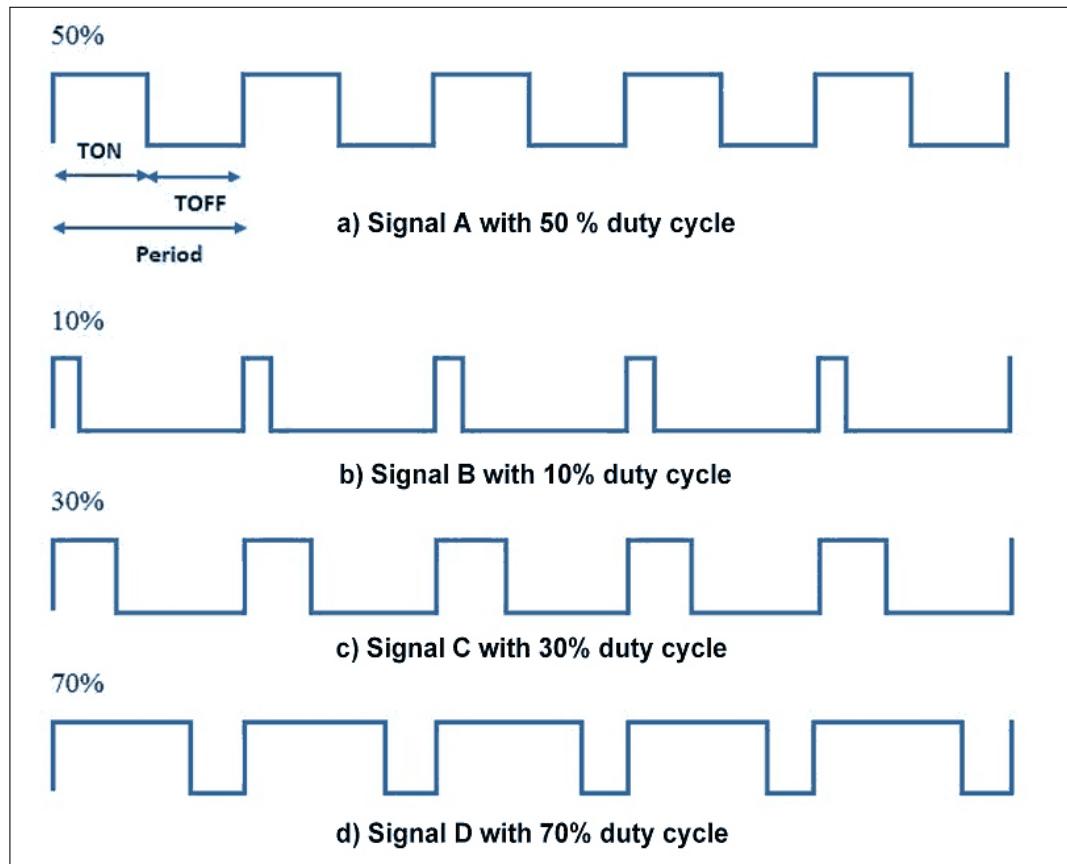


Abb. 18.: PWM Signale mit unterschiedlichem Tastverhältnis

A.6 MQTT Protokoll

MQTT (MQ Telemetry Transport oder Message Queue Telemetry Transport) wurde 1999 zur M2M-(Maschine zu Maschine)-Kommunikation im Zuge eines gemeinsamen Projekts von IBM und Arcom Control Systems entwickelt. Die Internet Assigned Numbers Authority (IANA) reserviert für MQTT die Ports 1883 und 8883. MQTT ist ein Client-Server-Protokoll. Clients senden dem Server nach Verbindungsaufbau Nachrichten mit einem Topic, die mit dem TLS-Protokoll verschlüsselt werden können.

⁶<http://www.electronicwings.com/pic/pic18f4550-pwms>

Abkürzungverzeichnis

BOM Bill of Materials

DEVKIT Development Kit

Cloud Cloud Computing

EKG Elektrokardiogramm

GmbH Gesellschaft mit beschränkter Haftung

I²S Inter-IC Sound

IoT Internet of Things, Internet der Dinge

IT Informationstechnik, Bereich der Informations- und Datenverarbeitung

JSON JavaScript Object Notation

LED Light-emitting diode, Leuchtdiode

MQTT Message Queuing Telemetry Transport

NVS Non-volatile storage

PCB Printed circuit board

PWM Pulse Width Modulation

SPI Serial Peripheral Interface Bus

UV Ultraviolettstrahlung

Abbildungsverzeichnis

Abb. 1	Noise Click Board	5
Abb. 2	Praktikum Projekt	6
Abb. 3	MikroElektronika Noise Click Schaltplan	8
Abb. 4	Datenübertragung durch SPI	10
Abb. 5	HTTPS Server (Access Point mode)	17
Abb. 6	Latest Telemetry Fenster	19
Abb. 7	Noise Click Dashboard mit Widgets	19
Abb. 8	MQTT Publish Beispiel	20
Abb. 9	Die entprellten Drucktaste	21
Abb. 10	NVS Meldungen bei Neustart	22
Abb. 11	Messung von Summer	24
Abb. 12	Schaltplan des Noise Clicks	25
Abb. 13	Leiterplatte des Noise Clicks	25
Abb. 14	3D Visualisation	26
Abb. 15	BOM Datei des Noise Clicks: Kondensatoren	26
Abb. 16	MikroBus Pinout Spezifikation	29
Abb. 17	ThingsBoard Beispiel	31
Abb. 18	PWM Signale mit unterschiedlichem Tastverhältnis	33