Universität Bremen

Fachbereich 3: Informatik und Mathematik

Entwurf und Entwicklung einer Schnittstelle zur Einbindung von Multifunktionssensoren in die SAUL-Abstraktion des Betriebssystems RIOT am Beispiel des Farbsensors APDS9960

Bachelorarbeit

29. Oktober 2019

Daniel Lux

Erstgutachter Dr.-Ing. Olaf Bergmann
Zweitgutachter Prof. Dr.-Ing. Carsten Bormann

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Bremen, 29. Oktober 2019

Daniel Lux

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Viele Sensoren sind auf dem Markt verfügbar. Für jeden Anwendungsfall gibt es die unterschiedlichsten Sensoren. Ein einzelner Sensor benötigt auf einer gefertigten Platine einen gewissen Platz. Benötigt man daher mehrere Sensoren, kann es nützlich sein einen Multifunktionssensor zu verwenden. Multifunktionssensoren beinhalten mehrere Funktionen in einem Gehäuse. So wird der Multifunktionssensor APDS9960 von Avago Technologies [1], der Bestandteil dieser Arbeit ist, mit vier Sensoren ausgestattet. Mit dem APDS9960 ist es möglich Entfernungen zu messen, die Helligkeit zu ermitteln, eine Farbe zu bestimmen und Gesten zu erkennen. Dabei ist das Gehäuse des Multifunktionssensors laut dem Datenblatt unter Package Outline Dimensions (Seite 35 [1]) nicht größer als 4mm x 2.4mm x 1.4mm.

Ein Sensor kann Daten erfassen. Allerdings wird auch ein Host benötigt, der bestimmte Sensoren kalibriert und die Daten in Empfang nimmt. Zum Beispiel können Mikrocontroller für so eine Aufgabe verwendet werden. Damit wir uns nicht auf einen speziellen Mikrocontroller konzentrieren, verwenden wir das Betriebssystem RIOT [2]. RIOT ist implementiert für eine Vielzahl von Mikrocontrollern mit unterschiedlichen Hardwareeigenschaften. Außerdem wird viel Entwicklungszeit eingespart, da grundlegenden Funktionen vorhanden sind (Seite 3f. [2]).

Das Betriebssystem unterstützt unterschiedliche Treiber, die mit SAUL eingebunden sind. SAUL ist die Schnittstelle zwischen Betriebssystem und Treibern. Daher soll der Umgang mit Multifunktionssensoren evaluiert werden.

1.1 Zielsetzung und Erkenntnisinteresse

Basieren wird diese Bachelorarbeit auf dem Betriebssystem RIOT. RIOT bietet eine Sensorschnittstelle SAUL an. Dabei soll evaluiert werden, ob SAUL in der Lage wäre, Multifunktionssensoren, wie den APDS9960 zu integrieren. Dabei soll ein Konzept vorgestellt werden, wie so eine Integration aussehen könnte.

Untersucht wird der Multifunktionssensor APDS9960 von Avago Technologies. Der APDS9960 hat einen Licht-, Farb-, Näherungs- und Gestensensor eingebaut(Seite 1 [1]). Dabei soll die Gestensteuerung in das Betriebssystem RIOT integriert werden. Es muss ebenfalls evaluiert werden, ob der Multifunktionssensor für die SAUL Abstraktion geeignet ist.

Auf der Basis des Treibers soll ein Funktionsdemonstrator entwickelt werden. Der Demonstrator soll die Funktionen der Gestensteuerung unterstützen. Dabei werden die Rohdaten des Sensors ermittelt und dargestellt. Diese Daten

können von jeder Anwendung individuell benutzt werden. Den Multifunktionssensor APDS9960 besitzt zwei unterschiedliche Modi. Dabei sollen beide Modi funktionsfähig integriert sein.

Die Gestenerkennung kann sehr vielfältig sein. Fast jedes Produkt im Alltag könnte durch eine Gestensteuerung eine bessere Usability erlangen. So könnten bestimmte Produktfunktionen intuitiver gestaltet werden. Zum Beispiel könnte man durch eine Geste nach links oder rechts das Fernsehprogramm ändern. Auch könnte eine Lichtersteuerung durch eine Geste eingestellt werden. Im öffentlichen Raum könnten an Türen Sensoren ausgestattet sein, die mit einer Gestensteuerung geöffnet werden können, ohne etwas berühren zu müssen. Dazu könnte ein Fahrstuhl mit einem Gestensensor ausgestattet werden. So müsse nicht jeder Mensch auf einen Knopf drücken und Angst vor unbekannten Bakterien haben.

1.2 Forschungsstand und theoretische Grundlage

Das Betriebssystem RIOT soll um eine API bereichert werden. RIOT ist ein Betriebssystem für Mikrocontroller. Mit diesem ressourcenschonenden Betriebssystem können Anwendungen für das Internet der Dinge entwickelt werden (Seite 1 [2]). Das Betriebssystem unterstützt im Voraus schon einige Mikrocontroller(Seite 5 [2]). Die Entwicklung auf der Plattform soll laut Hersteller sehr einfach sein und Zeit in der Entwicklung einsparen.

Einige Funktionen des Multifunktionssensors APDS9960 werden theoretische in RIOT unterstützt. Der Farb-, Umgebungslicht- und Näherungssensor sind in einem Pull-Request [3] zu RIOT enthalten. Allerdings fehlt noch die Funktionalität der Gestensteuerung.

Um den Gestensensor des APDS9960 in RIOT zu integrieren, muss ein Treiber entwickelt werden. Dabei wird evaluiert, ob die Gestenfunktion in SAUL integriert werden kann. SAUL heißt „Sensor Actuator Uber Layer“und bildet die Schnittstelle für Sensoren in RIOT. SAUL funktioniert ohne Probleme mit einzelnen Sensoren. Hierbei soll untersucht werden, ob SAUL eine geeignete Schnittstelle für Multifunktionssensoren bietet, oder ob die Schnittstelle überarbeitet werden müsste.

Für den Demonstrator wird der Mikrocontroller STM32F103C8T6 verwendet [4]. Diesen Mikrocontroller gibt es auf einem sehr günstigen BluePill Board bestückt [1]. Außerdem wird der Mikrocontroller von RIOT unterstützt und bietet alles, was für die Steuerung des Multifunktionssensors APDS9960 benötigt wird.

1.3 Forschungskonzept

Im Rahmen dieser Bachelorarbeit wird untersucht, wie der Sensor APDS9960 in das praxisnahes Betriebssystem RIOT für eingeschränkte System integriert

[1] https://www.heise.de/developer/artikel/Keine-bittere-Pille-die-Blue-Pill-mit-ARM-Cortex-M3-4009580.html [Zugegriffen: 28.10.2019]

werden kann. Dabei wird die Schnittstelle zwischen dem Betriebssystem und den Sensoren analysiert. Des Weiteren soll durch einen Funktionsdemonstrator gezeigt werden, dass der Multifunktionssensor APDS9960 angesteuert werden kann.

1.4 Eigene Motivation/Vorarbeiten

Mein Interesse an Embedded Software Entwicklung ist geweckt. Daher interessiere ich mich für Software auf eingeschränkten Systemen. Da Software alleine nicht Zielführend ist, sollten einfache Hardwarekenntnisse nicht fehlen.

Im Rahmen eines Bachelorprojekts WADI habe ich mich mit einigen Technologien beschäftigt, die auch in dieser Arbeit eine Rolle spielen. So habe ich schon kleinere Programme in RIOT geschrieben. Auch habe ich zwei Treiber entwickelt. Mit SAUL habe ich mich noch nicht auseinandergesetzt, hoffe aber, dass ich die Schnittstelle besser verstehe und effizienter Treiber für diese Plattform schreiben kann.

In einer Werkstudententätigkeit habe ich mit Mikrocontrollern von STMicroelectronics gearbeitet, sodass mir diese Systeme nicht fremd sind. Das Arbeiten in Datenblättern (und bei STM Mikrocontrollern in den Reference Manuals) gehört zur täglichen Arbeit und ist daher kein Problem.

2 Grundlagen für diese Bachelorarbeit

Für diese Arbeit werden bestimmte Technologien verwendet. Die nötigen Informationen zu diesen Technologien sollen in diesem Kapitel übermittelt werden. Als Grundlage für einen Mikrocontroller wird RIOT verwendet. RIOT ist ein Betriebssystem und nimmt viel Arbeit in der Entwicklung ab. Verwendet werden soll der Sensor APDS9960, welcher ein Multifunktionssensor ist. Dieser wird mittels einer Kommunikationsschnittelle namens I^2C angesprochen werden.

2.1 RIOT

RIOT ist ein Betriebssystem für Mikrocontroller. Da die meisten Mikrocontroller wenig Ressourcen und Rechenkapazitäten besitzen, muss die Software angepasst werden[2]. Windows, Linux oder andere Betriebssysteme sind für Rechner konzipiert worden, die wesentlich mehr Leistung als ein Mikrocontroller haben.

Das Betriebssystem RIOT ist sehr freundlich im Umgang mit IoT-Geräten (Internet der Dinge) (Seite 1 [2]). Zudem wird RIOT kostenlos und open source unter der LGPLv2.1 Lizenz veröffentlicht (Seite 11 [2]), sodass jeder das Betriebssystem ein bisschen besser machen kann. Außerdem strebt das Entwicklerteam von RIOT an, alle relevanten und offenen Standards zu implementieren (Seite 3 [2]).

In RIOT ist es möglich Anwendungen in der Programmiersprache C oder C++ zu entwickeln (Seite 3 [2]). Des Weiteren werden Standardtools, wie gcc oder gdb verwendet (Seite 9 [2]). Das Betriebssystem ist so aufgebaut, dass möglichst wenig Mikrocontroller spezifischer Programmcode verwendet wird. So kann die Portierung auf andere Systeme vereinfacht werden (Seite 6/9 [2]).

Damit RIOT auf möglich vielen Plattformen verwendet werden kann, muss das Betriebssystem mit sehr wenig Speicherplatz zurechtkommen. Das wird durch einen Mikrokernel gelöst (Seite 4 [2]). Dass bedeutet, dass der Kern des Betriebssystems sehr stark vereinfacht und minimalisiert wurde. So bleiben eventuell fortschrittliche Funktionen in einem Betriebssystem aus, allerdings kann das System auf sehr vielen unterschiedlichen Systemen laufen. Bei Bedarf kann man den RIOT Kern auch Mikrocontroller spezifisch mit Funktionen erweitern. Aufgrund der Softwarearchitektur ist das Betriebssystem RIOT auch

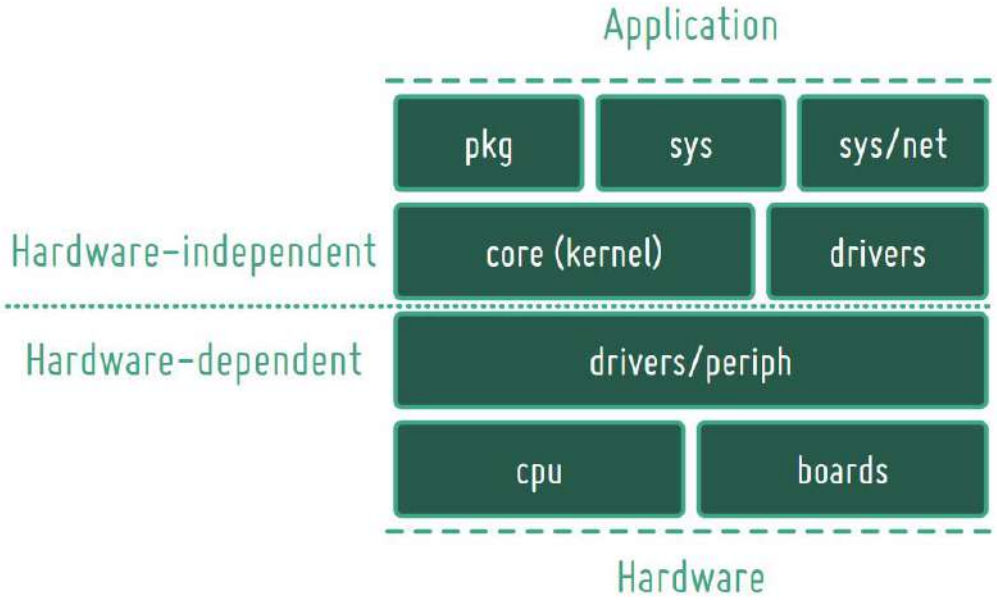

Abbildung 2.1: Softwarearchitektur von RIOT[1]

sehr Energieeffizient und kann als Echtzeitsystem verwendet werden. Außerdem hat RIOT eine Menge an Kommunikationsprotokolle implementiert. Dies ist nötig, um im Umgang des IoT mit anderen Geräten kommunizieren zu können. Zudem unterstützt RIOT eine breite Palette von Low-Power-Hardware. [5]

In Abbildung 2.1 ist die Softwarearchitektur von dem Betriebssystem RIOT dargestellt (Seite 3f. [2]). Auf der untersten Ebene findet sich die Hardware wieder. Die Hardware kann ein von RIOT unterstützer Mikrocontroller sein. Für jedes System benötigt man eine minimale Menge an Hardware-abhängigen Programmcode. Dies Betrifft zum Beispiel die CPU, die initialisiert werden muss. Außerdem bietet ein Development Board noch weitere Möglichkeiten an, als nur der Mikrocontroller. Da ein Mikrocontroller nicht nur aus einer Prozessoreinheit besteht, kann noch die interne Peripherie definiert werden. Aufbauend auf die Hardwaregrundlagen kommt der hardwareunabhängige Programmcode mit dem eigentlichen Betriebssystem RIOT. Der Mikrokernel ist in Core implementiert. Zudem kann zusätzlich angebrachte Hardware an das Board aktiviert werden. System- aber nicht Hardware relevante Module sind in sys niedergeschrieben. Die gesamte Netzwerkfunktionalität für IoT-Geräten von RIOT ist in sys/net definiert. In pkg sind zusätzliche externe Bibliotheken und Abhängigkeiten implementiert. In RIOT existiert auch ein Beispiel- und ein Testordner.

[1] https://riot-os.org/api/

2.2 Inter Integrated Circuit (I2C)

Philips Semiconductors entwickelte 1982 einen bidirektionalen Bus mit zwei Leitungen mit einer seriellen Datenleitung (SDA) und einer Taktleitung (SCL) (Revision history, Seite 2 [6]). In der heutigen Fassung spezifiziert I^2C eine Datenrate von bis zu 5 Mbit/s. I^2C bietet die Grundlage von weiterführenden Protokollen, wie zum Beispiel den System Management Bus (SMBus), den Power Management Bus (PMBus) und das Intelligent Platform Management Interface (IPMI) (Seite 3 [6]).

Wichtige Features von I^2C sind, dass mit zwei Leitungen mehrere Geräte verbunden werden können. Jedes I^2C-Gerät besitzt eine Adresse, die von der Software angesprochen werden kann. Darüber hinaus gibt es eine Master-Slave Beziehung, dabei kann der Master entweder senden oder empfangen. Es können sogar mehrere Master angeschlossen werden. Der Datentransfer erfolgt seriell mit 8 zugehörigen Bits und einem Acknowledge-Bit (Seite 3f. [6]).

Die beiden Leitungen von I^2C müssen jeweils durch einen Pull-Up Widerstand - wie in Abbildung 2.2 - an eine Versorgungsspannung angeschlossen werden. Je nach Größenordnung der Widerstände kann die Kommunikationsgeschwindigkeit beeinflusst werden. Die Pull-Up Widerstände müssen angeschlossen werden, weil I^2C-Geräte die beiden Leitungen nur gegen Ground runterziehen können (Slide 31, Seite 13 [7]). Damit kein undefiniertes Verhalten entsteht, zieht die Versorgungsspannung die Leitungen wieder hoch. Leitungen besitzen kleine Kapazitäten, daher können niederohmige Widerstände die Leitung schneller aufladen, als nutze man hochohmige Widerstände [8]. Allerdings ist durch das ohmsche Gesetz der Stromverbrauch erhöht. Wenn man keinen Widerstand benutzt, entsteht ein Kurzschluss zwischen der Versorgungsspannung und dem vom I^2C-Gerät heruntergezogenem Ground.

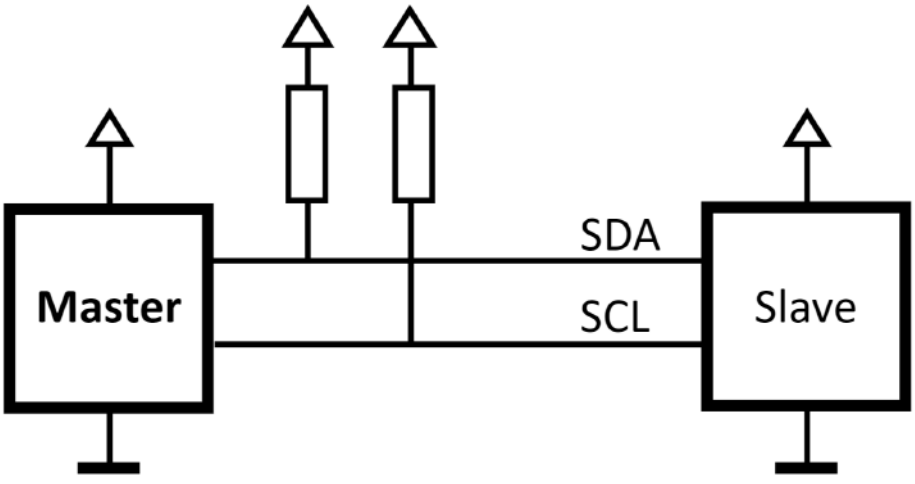

Abbildung 2.2: Elektrische Integration von I2C

Es gibt zwei Arten, wie man I^2C-Geräte ansprechen kann. Zum einen können I^2C-Geräte eine 7 Bit oder 10 Bit ansprechbare Adresse besitzen. Wie in Abbildung 2.3 erkennbar, wird beim Starten einer Kommunikation zuerst eine

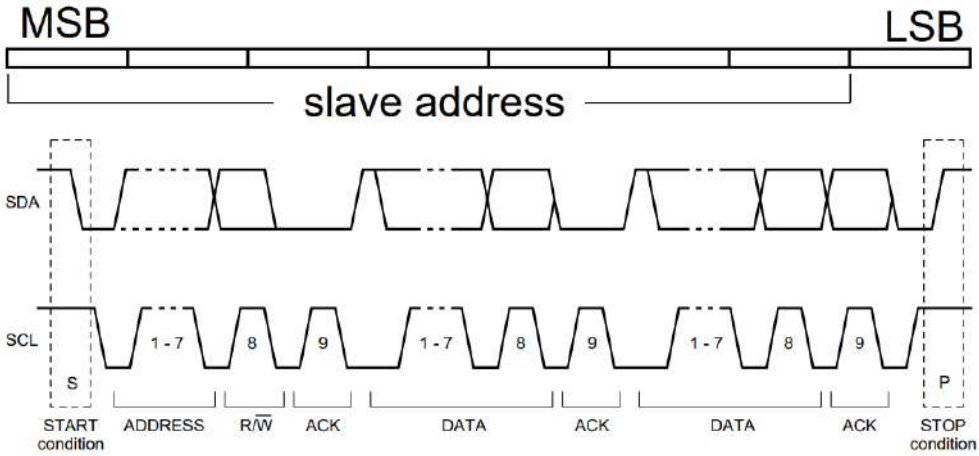

Abbildung 2.3: Datentransfer auf der I²C-Busleitung[2]

Startbedingung gesendet, gefolgt von der Adresse (Seite 13f. [6]). I²C ist spezifiziert mit einem seriellen 8 Bit Datentransfer. Daher wird die 7 Bit Adresse um einen nach links verschoben und das 8. Bit beschreibt ein Lese- oder Schreibkommando, welches im nächsten Datenpaket gesendet oder empfangen wird (Seite 13f. [6]).

Nicht jede Adresse ist für ein Gerät verfügbar. So gibt es auch reservierte Adressen. Diese werden im Dokument UM10204 von NXP Semiconductors näher beschrieben. So kann zum Beispiel durch eine Bitfolge eine 10 Bit Adresse angesprochen werden. Die 10 Bits werden einfach in zwei Bytes gesendet. Danach startet erst der Lese oder Schreibprozess (Seite 17 [6]).

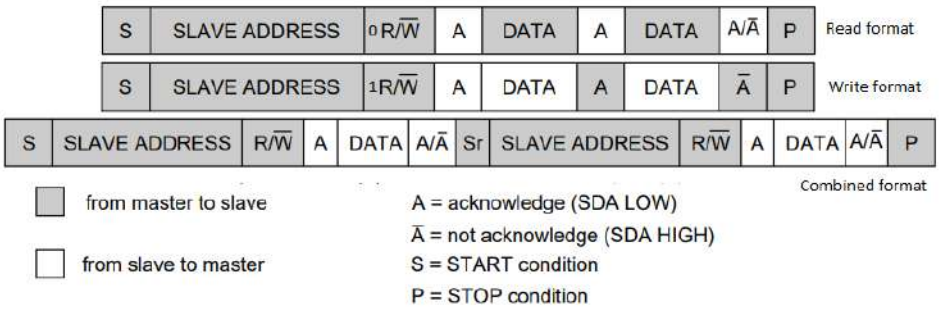

Abbildung 2.4: Unterschiedliche Datenformate auf der I²C-Busleitung[3]

Mit I²C sind drei unterschiedliche Datenformate möglich. Zum einen kann auf dem I²C-Bus geschrieben oder gelesen werden. Zum anderen kann es Kombinationen zwischen lesen und schreiben geben (Seite 14f. [6]). Die Abbildung 2.4 beschreibt drei unterschiedlichen Vorgänge einer Kommunikation. Die Kommunikation beginnt immer beim Master-Gerät. Der Master spricht eine bestimmte Adresse an und teilt dem I²C-Gerät mit, ob gerade gelesen oder geschrieben werden soll. Nach jedem Bytetransfer wird vom Opponenten die Nachricht mit einem Acknowledge-Bit bestätigt. Nachdem alle Daten gesendet

[2]Fig.9/Fig.10, Seite 13[6]
[3]Fig.11/Fig.12/Fig.13, Seite 15[6]

oder empfangen wurden, wird vom Master eine Stop Bedingung gesendet und beendet damit die Kommunikation (Seite 9 [6]).

2.3 APDS9960

Der APDS9960 ist ein Multifunktionssensor von Avago Technologies. Alle Angaben in diesem Kapitel beziehen sich auf das Datenblatt des Sensors [1]. Dieser Multifunktionssensor integriert vier Sensoren. Diese Sensoren sind ein Näherungssensor, ein Helligkeitssensor, ein Farbsensor und ein einfacher Gestensensor. Angesteuert wird der Multifunktionssensor mit I^2C. Der APDS9960 ist in der Lage ein Interrupt auszulösen und kann eine Datenrate bis zu 400 kHz erreichen. Betrieben wird das Bauteil mit 3 V (Seite 1ff. [1]). Augenmerk dieser Arbeit werden der Näherungs- und der Gestensensor werden.

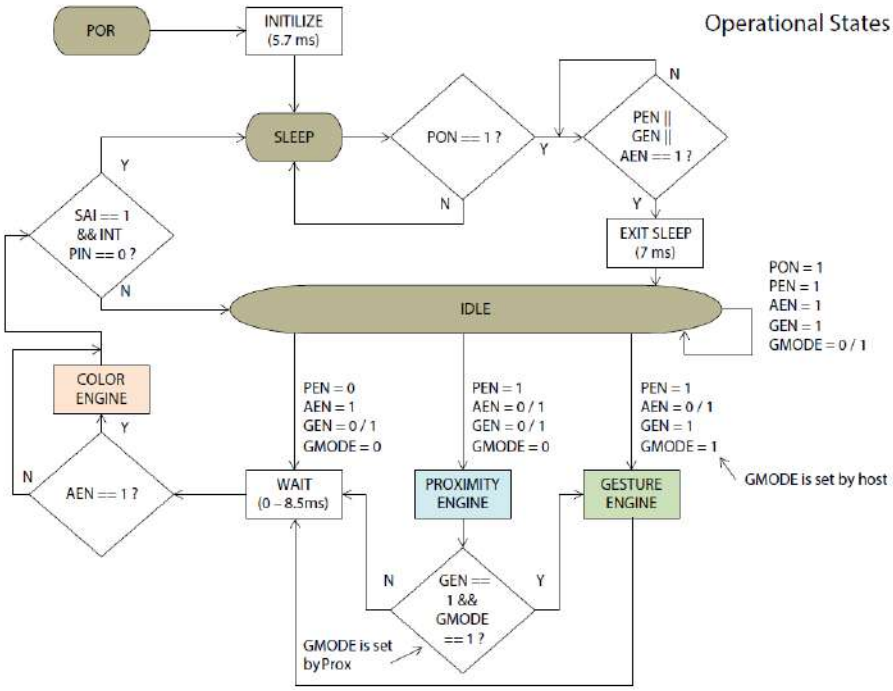

Abbildung 2.5: Zustandsdiagramm des APDS9960[4]

Der Multifunktionssensor APDS9960 wird durch eine in Abbildung 2.5 dargestellte Zustandsmaschine gesteuert. Innerhalb der Zustandsmaschine können funktionale Zustände ausgewählt oder übersprungen werden. So ist es möglich den Farbsensor zu überspringen, wenn die Funktionalität nicht gewünscht ist (Seite 9 [1]). Beim Anlegen der Versorgungsspannung initialisiert sich der Multifunktionssensor und befindet sich in einem Schlafzustand. Bei einer Anfrage über I^2C wird der Oszillator temporär aktiviert. Beim Setzten des Power-Bits im ENABLE-Register wird die interne Schaltung, dazu auch der Oszillator, dauerhaft aktiviert. Zu diesem Zeitpunkt wird sehr wenig Energie verbraucht, bis ein funktionaler Zustand erreicht wird (Seite 9 [1]). Beim erstmali-

[4]Fig.7, Seite 9[1]

gen Verlassen des Schlafzustandes in einen funktionalen Zustand benötigt der Multifunktionssensor eine kleine Pause von 7 ms. Anschließend werden je nach Features in die folgenden Zustände abgearbeitet: idle, proximity, gesture, wait, color/ALS und sleep. Es kann eine Wartezeit von 2,78 ms bis 8,54 s eingestellt werden(Seite 9 [1]).

Der Multifunktionssensor APDS9960 wird mit I^2C angesprochen. Die 7-Bit Adresse des Sensors ist 57 oder auch als Hexadezimal Wert 0x39. Möglich sind die drei Datenformate dargestellt in Abbildung 2.4. Zuerst muss ein Kommando gesendet werden und dann die Daten. Der Lesevorgang gibt das Register aus dem gesendeten Kommandobyte zurück (Seite 8 [1]).

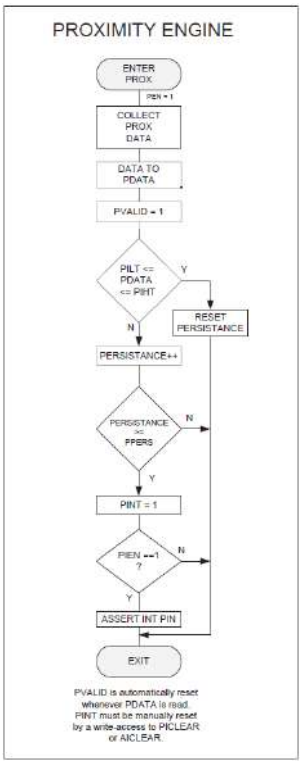

Abbildung 2.6: Näherungs Engine des APDS9960[5]

2.3.1 Näherungssensor

Der Näherungssensor funktioniert ähnlich wie bei einem Smartphone, welches man an das Ohr hält. Es ist eine Infrarot-LED verbaut und eine Fotodiode (Seite 2 [1]). Es gibt zwei Events für das Erkennen und Freigeben eines Objektes bei denen ein Interrupt ausgelöst werden kann. Mit Schwellenwerten können bestimmte Abstände zum Sensor ignoriert werden und mit einem Offsetwert können ungewollte Reflexionen am Sensor umgangen werden. Das optimale Ergebnis wird durch drei unterschiedlichen Werten beeinflusst: die Infrarot-LED-Leuchtkraft, der Infrarot-Empfang und Umgebungsfaktoren, wie zum Beispiel die Oberflächenreflexion von Objekten (Seite 10f. [1]).

[5]Fig.8, Seite 11[1]

In der Tabelle 1 im Anhang gibt es eine Liste von der Näherungssteuerung. Die Register können so eingestellt werden, dass die Näherungs-Engine in Abbildung 2.6 nach individuellem Verhalten befolgt wird.

Die Abbildung 2.6 beschreibt den Vorgang beim Betreten in den Funktionszustand der Näherungs Engine.

Zuerst wird die Entfernung gemessen. Dazu sendet die Infrarot-LED gepulste Signale und die vier Fotodioden empfangen die Reflexion. Die gemessene Entfernung wird nun in das PDATA-Register geschrieben und in den Statusinformationen wird vermerkt, dass im PDATA-Register ein valider Wert steht. Nun wird noch geprüft, ob der Wert innerhalb der Schwellwerte liegt. Sind jetzt noch Interrupts aktiviert, wird noch ein dementsprechendes Bit gesetzt und der Interrupt-Pin wird hochgezogen (Seite 11 [1]).

2.3.2 Gestensensor

Der Gestensensor besteht ebenfalls aus einer Infrarot-LED und vier Fotodioden, wie in Abbildung 2.7 gezeigt wird. So können einfache Gesten in der Horizontalen und Vertikalen erkannt werden. Es ist aber auch möglich komplexe Gesten zu erkennen. Es wurden Features entwickelt, die eine möglichst robuste Erkennung von Gesten ermöglichen soll (Seite 16 [1]).

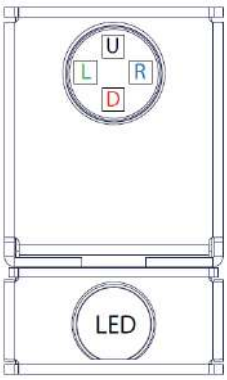

Abbildung 2.7: Positionierung der Dioden des APDS9960[6]

Die Tabelle 2 B.2 im Anhang gibt Aufschluss darüber, welche Register zu der Konfiguration der Gestensteuerung gehören. Dazu gibt es in Abbildung 2.8 ein passendes Flussdiagramm, wie der Ablauf der Gestenerkennung funktioniert. Zuerst werden Daten erfasst. Je nach Gebrauch können die erkannten Richtungen der Gesten im GDIMS-Register konfiguriert werden (Seite 16 [1]).

Die Werte der aufgenommenen Geste werden in eine FIFO-Liste hinzugefügt. Sollte die FIFO-Liste mit 32 Einträgen voll sein, können keine weiteren Daten ermittelt werden. Sollte der Mikrocontroller langsam in der Verarbeitung der Daten sein, so bleiben die ersten 32 Eintrage bestehen. Ist die FIFO Liste bereits voll, wird der nächste Zustand der Gesten Engine angesteuert. Befindet sich noch Platz in der Liste, dann werden die Werte dieser FIFO Liste hinzugefügt und ein Zähler namens GFLVL, der die Füllhöhe der Liste beschreibt, in-

[6]Fig.11, Seite 17[1]

krementiert. Ist der Wert im GFLVL Register höher als der FIFO Schwellenwert im GFIFOTH Register, wird das Gesture Valid Bit und das Gesture Interrupt Bit gesetzt. Nun muss nur noch überprüft werden, ob Interrupts aktiviert wurden. Ist dem so, wird ein Interrupt ausgelöst (Seite 16 [1]).

In dem GEXTH-Register wird definiert, wann eine Geste endet. Beträgt der Wert 0, dann endet die Geste bis der Mikrocontroller den Modus des Sensors ändert. Andernfalls kommt ein persistenter Filter zum Einsatz, der eine bestimmte Anzahl an Endzuständen erfasst. Wurde das Ende häufig genug erkannt, wird überprüft, ob die Daten in der FIFO Liste valide sind und kann dabei wieder ein Interrupt auslösen. Ist eine Geste noch nicht zu Ende, springt der Zustand in eine Wartemaschine und anschließend fängt die Gesten Engine von vorne an. (Seite 16 [1]).

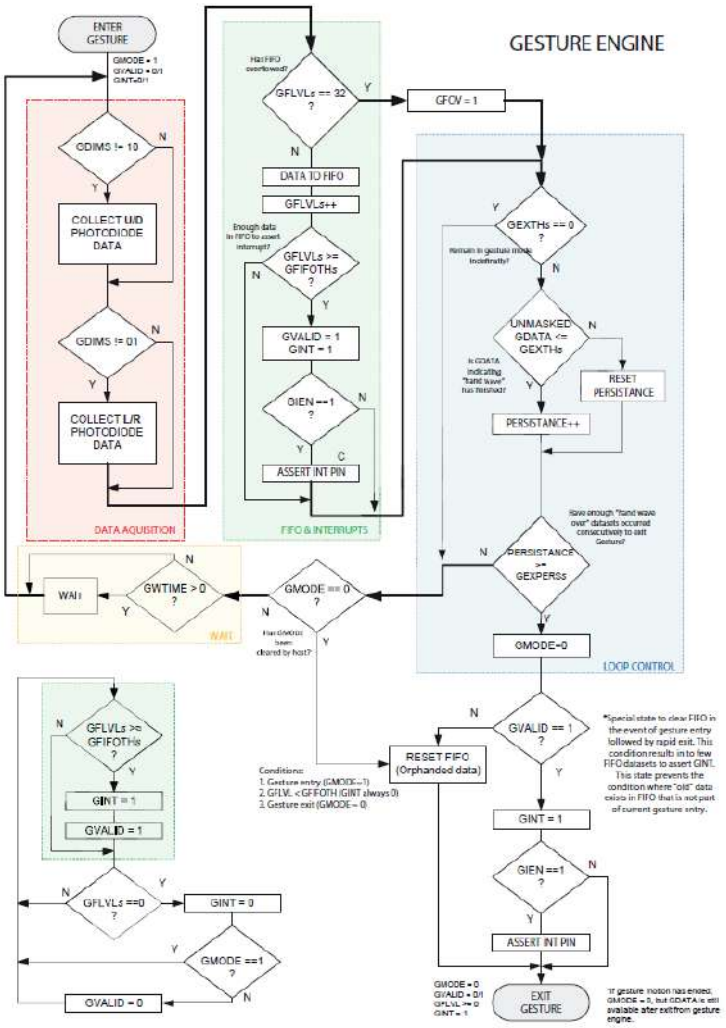

Abbildung 2.8: Gesten Engine des APDS9960[7]

[7]Fig.10, Seite 15[1]

3 Analyse von SAUL

SAUL bedeutet auf Englisch „Sensor Actuator Uber Layer API“und dient zum einen dazu einen herstellerunabhängigen Zugriff auf Sensoren und Aktoren zu bekommen und zum anderen, dass heterogene Mikrocontroller dieselben Funktionen verwenden können (Seite 6, [2]).

Saul hat einen einfachen Aufbau. Die Schnittstelle beinhaltet Geräte Deskriptoren und ein Datentyp namens phydat. Ein Geräte Deskriptor wird benötigt um unterschiedliche Sensoren und Aktoren anzusprechen. Die physikalischen Daten der Geräte werden in den Datentypen gespeichert. Saul bietet zudem eine Geräteregistrierung für Sensoren und Aktoren an. So könne zum Beispiel bei der Initialisierung des Mikrocontrollers alle Geräte automatisch konfiguriert werden (Seite 6, [2]).

3.1 Aufbau von SAUL

Das Modul SAUL von RIOT besteht nur aus drei Dateien. Wie in der Abbildung 3.1 erkennbar ist, besteht Saul aus den Dateien **saul.h** [9], **saul_reg.h** [10] und **phydat.h** [11]. Aus der Sicht des Betriebssystems RIOT wird Saul in der Datei **auto_init.c** initialisiert [12]. Sofern die Autoinitialisierung nicht explizit deaktiviert wird, wird die Datei **auto_init.c** immer automatisch ausgeführt.

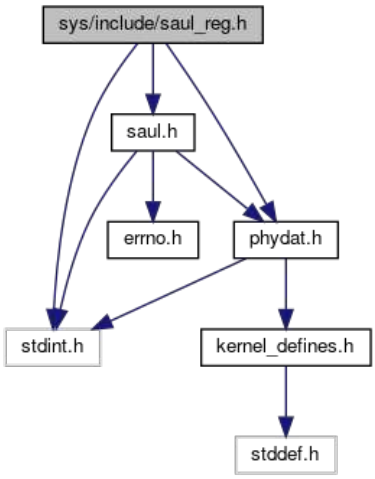

Abbildung 3.1: Abhängigkeiten der SAUL-Abstraktion[1]

[1]https://riot-os.org/api/saul__reg_8h.html [28.10.2019]

3.1.1 phydat.h

Die Datei **phydat.h** definiert einen Standard von physischen Daten [11]. Es wird unter anderem ein generischer Datentyp phydat_t definiert. phydat_t besteht aus einem Wert, einer Einheit und einer Skalierung. Damit ist es möglich physische Daten darzustellen. Der Datentyp ist 16 Bit groß. Damit ist es kein Problem, Daten mit geringerer Auflösung zu verarbeiten. Des Weiteren gibt es noch eine Auflistung von verschiedenen Einheiten. Die wichtigsten Funktionen sind phydat_fit und phydat_dump. Die fit-Funktion speichert den gegebenen Wert in dem phydat_t Datentypen ab. Die dump-Funktion gibt die gegebenen Werte einer Struktur auf dem Ausgabekanal zurück.

3.1.2 saul.h

Die Datei **saul.h** definiert die SAUL-Schnittstelle in RIOT [9]. SAUL definiert eine Struktur saul_driver_t mit einem Lese- und Schreibepointer und einem Typen. Zudem existiert eine Funktion zum Auslesen eines Sensors und zum Beschreiben eines Aktors.

3.1.3 saul_reg.h

Die Datei **saul_reg.h** stellt eine Schnittstelle für die SAUL-Registrierung bereit [10]. Auch hier existiert eine Struktur namens saul_reg_t und es gibt nützliche Funktionen zum Verwenden der SAUL-Schnittstelle. saul_reg_t besteht aus einem Geräte Deskriptor, einem Namen zur Identifikation des Gerätes und einen Gerätetreiber. Als Funktionen gibt es saul_reg_add zum Hinzufügen eines Treibers in die Registrierung, saul_reg_rm zum Entfernen eines Treibers, unterschiedliche Funktionen zum Suchen einer Ressource und das Lesen und Schreiben aus den phydat_t-Daten.

3.1.4 auto_init.c

Die Datei **auto_init.c** dient dazu Sensoren und Aktoren automatisch zu initialisieren [12]. Voraussetzung ist, dass das dementsprechende Modul in der Makefile eines Projektes hinzugefügt wurde. Ist dem so, wird die SAUL Registrierung in einer externen Datei aufgerufen. In der externen Datei soll die SAUL Registrierung und die Initialisierung des Gerätes durchgeführt werden.

3.2 Anforderungen an SAUL

SAUL sollte in der Lage sein den Multifunktionssensor APDS9960 als Gerät mit vollem Funktionsumfang zu integrieren. Werden keine Parameter explizit angegeben, soll der Sensor automatisch initialisiert werden. Auch soll ein Konzept bestehen andere Multifunktionssensoren zu integrieren. Der Sensor wird einmalig initialisiert und dem Betriebssystem stehen für jede Funktion jeweils eine SAUL-Ressource zur Verfügung.

3.3 Entwurf einer Schnittstelle zur Integration von Multifunktionssensor in SAUL

Generell kann über die Datei **auto_init.c** ein SAUL Gerät initialisiert werden. Jedes dem Projekt zugefügtem Modul wird abgefragt und ruft die dementsprechende Initialisierungsfunktion zu dem Modul auf. Möchte man daher einen neuen Treiber für einen Sensor für das Betriebssystem RIOT entwickeln, so muss ein neues Modul erstellt werden. Die **auto_init.c** Datei ruft eine Funktion namens auto_init_`Name des Multifunktionssensors`() auf. Diese Funktion befindet sich in der gleichnamigen Datei im Ordner RIOT/sys/auto_init/saul/. Da ein Multifunktionssensor mehrere Sensoren in einem Gerät integriert, muss nun eine Auswahl der Funktionen getroffen werden.

Die Parameter eines Sensors lässt sich einfach in einer Datenstruktur darstellen. Hier können Informationen stehen, ob eine bestimmte Funktion verwendet oder ob eine Funktion ausgeschlossen werden soll. Jede Anwendung benötigt unterschiedliche Funktionen des Multifunktionssensors. Daher ist es sinnvoll, dass diese Datenstruktur austauschbar ist. Entweder kann diese Datenstruktur überschrieben werden, bevor die Initialisierungsfunktion aufgerufen wurde. Zum Beispiel durch den Präprozessor der Programmiersprache C. Oder die Datenstruktur wird direkt vor dem Kompilieren manipuliert.

Eine Autoinitialisierung benötigt eine parametrisierte Datenstruktur. Da der Verwendungszweck pro Anwendung variieren kann, wird der Multifunktionssensor mit seinem kompletten Funktionsumfang aus folgendem Grund initialisiert. Der Entwickler, der ein Produkt entwerfen möchte, hat sich aus ganz bestimmten Gründen dazu entschieden genau diesen Multifunktionssensor zu verwenden. Das bedeutet, dass der Sensor die Wünsche des Entwicklers abdeckt. Werden Funktionen nicht verwendet, könne ein anderer Multifunktionssensor verwendet werden und spart dabei noch kosten. Werden noch weitere Sensoren benötigt, werden diese als weitere Bauteile in die Schaltung integriert. Sollte aber tatsächlich eine Funktion nicht benötigt werden, so muss die Datenstruktur angepasst werden.

Wurde eine valide, parametrisierte Datenstruktur erstellt, wird der Sensor mit seiner Konfiguration initialisiert. Beim Zugreifen auf den Sensor werden bestimmte Kommunikationsprotokolle verwendet. Im Falle des Multifunktionssensors APDS9960 wird das Kommunikationsprotokoll I^2C verwendet. Sollte der Treiber neu entwickelt werden, muss der Entwickler selber die Initialisierungsroutine designen. Nach erfolgreicher Initialisierung wird für jede Funktion ein SAUL Gerät registriert. So kann SAUL die physischen Daten unabhängig vom Gerät verarbeiten.

Es ist ohne Probleme möglich Multifunktionssensoren in die SAUL Abstraktion zu integrieren. Dazu müssen die einzelnen Sensoren bei SAUL registriert werden. Voraussetzung dabei ist, dass die vom Sensor ermittelten Daten in die Phydat Datenstruktur passen. Wenn wir uns die Daten der Gestensteuerung des Multifunktionssensors APDS9960 anschauen, stellen wir fest, dass die Gestendaten in Datensets gespeichert werden. Diese Datensets werden in eine FIFO Liste abgelegt. Ein Datenset beinhaltet für jede vier Himmelsrichtungen einen 8

Bit großen Wert, der durch seine Auflösung Werte von 0 bis 255 darstellen kann. Das sind für einen Datenset 32 Bit große Daten. Die Datenstruktur Phydat hat allerdings nur Platz für 16 Bit große Daten, die mit einem Scale Faktor versehen sind. Dieser Scale Faktor nützt uns allerdings nicht, weil wir 4 unterschiedliche Daten codiert haben und nicht einen großen. Daher ist die Gestenfunktion des Multifunktionssensor APDS9960 nicht für die SAUL Abstraktion ohne weiteres geeignet.

4 Entwickeln eines Treibers

Eine Grundlage des Treibers wird die Arbeit auf GitHub von Gunar Schorcht sein, der sich schon mit dem Multifunktionssensor APDS9960 befasst hat [3]. Allerdings fehlt der Gestensensor, daher ist das Ziel eine saubere Integration der Gestenfunktion zu implementieren. Gunar Schorcht implementiert in 15 Dateien einen grundlegenden Treiber und einen kompletten Treiber. Der Unterschied liegt an den Interrupts, die bei dem kompletten Treiber aktiviert sind. Der Sensor APDS9960 unterstützt den Polling Modus und den Interrupt basierten Modus. Im Polling Modus wartet der Mikrocontroller so lange auf eine Antwort, bis der Sensor ihm diese beantwortet. Im Interrupt Betrieb werden die Ressourcen des Mikrocontrollers nur dann beansprucht, sofern der Sensor neue Daten bekommen hat.

Als erste Maßnahme müssen die Register der Gestensteuerung in der Datei apds99xx_reg.h (Listing 4.1) definiert werden. Jedes Register wird durch eine Basisadresse und einem Offset charakterisiert. Die Basisadresse des Multifunktionssensor APDS9960 beträgt 0x80.

Listing 4.1: Register für die Gestensteuerung mit einer Registerbasis von 0x80

```
85 #define APDS99XX_REG_GPENTH     (APDS99XX_REG_BASE + 0x20)  /*
       *< Gesture proximity enter threshold */
86 #define APDS99XX_REG_GEXTH      (APDS99XX_REG_BASE + 0x21)  /*
       *< Gesture exit threshold*/
87 #define APDS99XX_REG_GCONF1     (APDS99XX_REG_BASE + 0x22)  /*
       *< Gesture configuration one */
88 #define APDS99XX_REG_GCONF2     (APDS99XX_REG_BASE + 0x23)  /*
       *< Gesture configuration two */
89 #define APDS99XX_REG_GOFFSET_U  (APDS99XX_REG_BASE + 0x24)  /*
       *< Gesture UP offset register */
90 #define APDS99XX_REG_GOFFSET_D  (APDS99XX_REG_BASE + 0x25)  /*
       *< Gesture DOWN offset register */
91 #define APDS99XX_REG_GOFFSET_L  (APDS99XX_REG_BASE + 0x27)  /*
       *< Gesture LEFT offset register */
92 #define APDS99XX_REG_GOFFSET_R  (APDS99XX_REG_BASE + 0x29)  /*
       *< Gesture RIGHT offset register */
93 #define APDS99XX_REG_GPULSE     (APDS99XX_REG_BASE + 0x26)  /*
       *< Gesture pulse count and length */
94 #define APDS99XX_REG_GCONF3     (APDS99XX_REG_BASE + 0x2A)  /*
       *< Gesture configuration three */
95 #define APDS99XX_REG_GCONF4     (APDS99XX_REG_BASE + 0x2B)  /*
```

```
    *< Gesture configuration four */
96 #define APDS99XX_REG_GFLVL      (APDS99XX_REG_BASE + 0x2E)  /*
    *< Gesture FIFO level */
97 #define APDS99XX_REG_GSTATUS    (APDS99XX_REG_BASE + 0x2F)  /*
    *< Gesture status */
98 #define APDS99XX_REG_GFIFO_U    (APDS99XX_REG_BASE + 0x7C)  /*
    *< Gesture FIFO UP value */
99 #define APDS99XX_REG_GFIFO_D    (APDS99XX_REG_BASE + 0x7D)  /*
    *< Gesture FIFO DOWN value */
100 #define APDS99XX_REG_GFIFO_L    (APDS99XX_REG_BASE + 0x7E)  /*
    *< Gesture FIFO LEFT value */
101 #define APDS99XX_REG_GFIFO_R    (APDS99XX_REG_BASE + 0x7F)  /*
    *< Gesture FIFO RIGHT value */
```

Damit die Gestenfunktion sauber in die Arbeit von Gunar Schorcht integriert wird, wurde in der Datei apds99xx.h (Listing 4.2) eine Struktur namens gesture_params_t erstellt. Diese Struktur beinhaltet alle Einstellungen, die das Verhalten der Gestensteuerung beeinflusst. Auch werden vordefinierte Enumeratoren für bestimmte Parameter entwickelt, die das Arbeiten mit dem Treiber erleichtern soll.

Listing 4.2: Datenstruktur der Gestenparametern

```
320 #if MODULE_APDS9960
321 typedef enum {
322     APDS99XX_GES_FIFO_TH_1 = 0,
323     APDS99XX_GES_FIFO_TH_4,
324     APDS99XX_GES_FIFO_TH_8,
325     APDS99XX_GES_FIFO_TH_16,
326 } apds99xx_ges_FIFO_th_t;
327
328 typedef enum {
329     APDS99XX_GES_PULSE_LEN_4 = 0,
330     APDS99XX_GES_PULSE_LEN_8, // default
331     APDS99XX_GES_PULSE_LEN_16,
332     APDS99XX_GES_PULSE_LEN_32,
333 } apds99xx_ges_pulse_len_t;
334
335 typedef enum {
336     APDS99XX_GES_ENABLE_OFF = 0,
337     APDS99XX_GES_ENABLE_ON,
338 } apds99xx_ges_enable_t;
339
340 typedef enum {
341     APDS99XX_GES_INTERRUPT_OFF = 0,
342     APDS99XX_GES_INTERRUPT_ON,
343 } apds99xx_ges_interrupt_t;
344
345 typedef struct
346 {
```

```
347     uint8_t offset_up; // 0xa4
348     uint8_t offset_down; // 0xa5
349     uint8_t offset_left; // 0xa7
350     uint8_t offset_right; // 0xa9
351 } gesture_offset_t;
352
353
354
355 typedef struct
356 {
357     apds99xx_ges_enable_t enable; // 0x80 6
358     apds99xx_ges_interrupt_t interrupt; // 0xab 1
359
360     uint8_t entry_threshold; // 0xa0 dependent on prx data
361     uint8_t exit_threshold; // 0xa1 dependent on prx data
362
363     uint8_t exit_mask; // 0xa2 2345
364     uint8_t persistence; // 0xa2 01
365     uint8_t gain_control; // 0xa3 56
366     uint8_t led_strength; // 0xa3 34
367     uint8_t wait_time; // 0xa3 012
368     gesture_offset_t offset;
369     uint8_t pulse_count; // 0xa6 012345
370     apds99xx_ges_pulse_len_t pulse_length; // 0xa6 67
371     uint8_t dimension_select; // 0xaa 01
372 } gesture_params_t;
373
374 #endif
```

Nachdem die Parameter spezifiziert wurden, wird ein vordefinierter Datensatz (Listing 4.3) unter dem Namen APDS9960_GESTURE_PARAMS angelegt. Standardmäßig wird der Gestensensor beim Start aktiviert und der Interrupt ist ausgeschaltet. Die Eingangsschwelle wurde auf 20 gesetzt, weil der Nährungssensor ein Grundrauschen wahrnimmt. Die Ausgangsschwelle wurde ebenfalls auf 20 gesetzt, damit das Ende einer Geste erkannt werden kann. Würde der Schwellwert 0 betragen, würde der Gestensensor erst aufhören Daten zu sammeln, wenn der Benutzer den Gestenmodus zurücksetzt. Es gibt mehrere Parameter, die im Normalbetrieb nicht benutzt werden müssen. Die Wartezeit des Sensors wurde auf die höchste Stufe 7 gesetzt. Das dient dazu, um einen Überlauf an Daten zu verhindern und einen reduzierten Stromverbrauch zu erreichen. Die LED Eigenschaften wurde wie im Datenblatt empfohlen eingestellt.

Listing 4.3: Überschreibbare Standardparameter der Gestenfunktion

```
73 #ifndef APDS9960_GESTURE_PARAMS
74 #define APDS9960_GESTURE_PARAMS { \
75         .enable = APDS99XX_GES_ENABLE_ON, \
76         .interrupt = APDS99XX_GES_INTERRUPT_OFF, \
77         .entry_threshold = 20, \
78         .exit_threshold = 20, \
```

```
79          .exit_mask = 0, \
80          .persistence = 0, \
81          .gain_control = 0, \
82          .led_strength = 0, \
83          .wait_time = 7, \
84          .offset = { \
85              .offset_up = 0, \
86              .offset_down = 0, \
87              .offset_left = 0, \
88              .offset_right = 0, \
89          }, \
90          .pulse_count = 8, \
91          .pulse_length = APDS99XX_GES_PULSE_LEN_8, \
92          .dimension_select = 0, \
93      }
94  #endif
```

Ist der Multifunktionssensors APDS9960 in die Applikation eingebunden, so stehen dem Entwickler zwei weitere Funktionen zur Verfügung. Diese Funktionen werden in der Headerdatei apds99xx.h (4.4) definiert. Zum einen erlaubt die Funktionen apds99xx_data_ready_ges() das Abfragen nach neuen Daten im Polling-Modus. Zum anderen ermöglicht die Funktion apds99xx_read_ges_raw() das Empfangen der Daten. Die Daten werden in Datensets gespeichert. Realisiert wird das durch die Struktur apds99xx_ges_dataset_t. Diese Struktur beinhaltet jeweils einen 8 Bit großen Integer für die Variablen UP, DOWN, LEFT und RIGHT. Da allerdings der Sensor eine 32 Datenset große FIFO Liste beinhaltet, werden die Datensets in der Struktur apds99xx_ges_t in eine FIFO Liste eingetragen. Je nach Anwendung werden unterschiedliche Algorithmen für die Gestenerkennung verwendet. Daher werden nur die Rohdaten durch diesen Treiber angeboten.

Listing 4.4: Realisierung der Speicherung von rohen Gestendaten

```
517 #if MODULE_APDS9960
518 /**
519  * @brief   Gesture data structure (APDS9960 only)
520  */
521 typedef struct {
522     uint8_t UP;
523     uint8_t DOWN;
524     uint8_t LEFT;
525     uint8_t RIGHT;
526 } apds99xx_ges_dataset_t;
527
528 typedef struct {
529     apds99xx_ges_dataset_t FIFO[32];
530 } apds99xx_ges_t;
531
532
533 int apds99xx_read_ges_raw(const apds99xx_t *dev, apds99xx_ges_t
```

```
    *ges);
534 int apds99xx_data_ready_ges (const apds99xx_t *dev);
535
536 #endif
```

Eine Datenstruktur namens apds99xx_int_config_t (4.5) ist für die Interrupt Steuerung zuständig. Innerhalb der Struktur wird definiert, ob Interrupts aktiviert werden sollen und setzt zudem noch den Persistence Wert und den FIFO Schwellwert. Der FIFO Schwellwert gibt ein Interrupt aus, sofern die Liste einen Bestimmten füllstand aufweist. Diese Werte werden erst hinzugefügt, wenn der Sensor APDS9960 eingebunden wurde. Ebenfalls wurde die Struktur apds99xx_int_source_t um einem booleschen Wert ges_int erweitert.

Listing 4.5: Interrupteinstellungen

```
459 #if MODULE_APDS9960
460     bool ges_int_en;
461     uint8_t ges_pers;
462     apds99xx_ges_FIFO_th_t ges_fifo_th; // 0xa2 67
463 #endif
```

In der Datei apds99xx.c werden konkret die Funktionen und das Verhalten des Sensors implementiert. In der Initialisierungsroutine werden die Register des Sensors APDS9960 beschrieben (4.6). Zuerst muss geprüft werden, ob der Sensor überhaupt aktiviert werden soll. Diese Information steht in den Parametern. Anschließend werden die Parameter an den Sensor gesendet.

Listing 4.6: Die Initialisierungsroutine der Gestensteuerung

```
136 #if MODULE_APDS9960
137     // if gesture is enabled then initialize gesture params
138     if(dev->params.ges_param.enable == APDS99XX_GES_ENABLE_ON)
139     {
140         reg = 0;
141
142         _reg_write(dev, APDS99XX_REG_GPENTH, &dev->params.
               ges_param.entry_threshold, 1);
143         _reg_write(dev, APDS99XX_REG_GEXTH , &dev->params.
               ges_param.exit_threshold , 1);
144
145         //reg =  (dev->params.ges_param.persistence << 0) | (
               dev->params.ges_param.exit_mask << 2) | (dev->params
               .ges_param.FIFO_threshold << 6);
146         //_reg_write(dev, 0xA2, &reg, 1);
147
148         reg = (3 << 4);
149         _reg_write(dev, 0x90, &reg, 1);
150
151         reg =  ((dev->params.ges_param.wait_time & 0x7) << 0) |
                   ((dev->params.ges_param.led_strength & 0x3) << 3) |
                   ((dev->params.ges_param.gain_control & 0x3) << 5);
```

```
152         _reg_write(dev, APDS99XX_REG_GCONF2, &reg, 1);
153
154         _reg_write(dev, APDS99XX_REG_GOFFSET_U, &dev->params.
               ges_param.offset.offset_up, 1);
155         _reg_write(dev, APDS99XX_REG_GOFFSET_D, &dev->params.
               ges_param.offset.offset_down, 1);
156         _reg_write(dev, APDS99XX_REG_GOFFSET_L, &dev->params.
               ges_param.offset.offset_left, 1);
157         _reg_write(dev, APDS99XX_REG_GOFFSET_R, &dev->params.
               ges_param.offset.offset_right, 1);
158
159         _reg_write(dev, APDS99XX_REG_GCONF3, &dev->params.
               ges_param.dimension_select, 1);
160
161         reg = ((dev->params.ges_param.pulse_count & 0x3F) | ((
               dev->params.ges_param.pulse_length & 0x3) << 6));
162         _reg_write(dev, APDS99XX_REG_GPULSE, &reg, 1);
163     }
164 #endif
```

Wurden alle Parameter korrekt übertragen, muss ein entsprechendes Bit im Sensor gesetzt werden, damit dieser die Gesten Engine ansteuern kann (4.7).

Listing 4.7: Aktivieren der Gestensteuerung

```
173 #if MODULE_APDS9960
174     if(dev->params.ges_param.enable == APDS99XX_GES_ENABLE_ON)
175         _set_reg_bit(&reg, APDS99XX_REG_GEN, 1);
                                  /* enable GES */
176 #endif
```

Wird der Gestensensor im Polling-Modus verwendet, so kann mit der Funktion apds99xx_data_ready_ges() (4.8) geprüft werden, ob neue Daten im Sensor vorhanden sind. Dabei fragt der Mikrocontroller nach dem Statusregister. Ist das dementsprechende Bit gesetzt, so wird ein erfolgreiches Statement zurückgeschickt.

Die Funktion apds99xx_read_ges_raw() ermittelt die Rohdaten des Sensors. Es können maximal $32 * 4 = 128$ Daten in der FIFO Liste vorhanden sein. Ein Datenset besteht aus 4 Bytes und die FIFO Liste ist 32 Datensets groß. Dementsprechend werden 128-mal die Register ausgelesen. Der Sensor verhält sich wie folgt: die Adresse von der Variable UP entspricht 0xFC, die von DOWN 0xFD, von LEFT 0xFE und von RIGHT 0xFF. Fängt man bei der ersten Adresse 0xFC an 4 Register zu lesen, so erhält man in der Reihenfolge ein Datenset. Wird der Lesebefehl ein weiteres Mal aufgerufen, so springt der Lesezeiger des Sensors auf die Adresse 0xFC zurück, sodass der nächste FIFO Wert ausgelesen werden kann. Befinden sich keine Datensets mehr in dieser Liste und es wird versucht weiterhin etwas zu lesen, dann werden nur nullen zurückgegeben. Daher ist es nicht schlimm, würde man versuchen mehr Daten, als sich in der FIFO Liste befinden, zu lesen.

Listing 4.8: Funktion zum Handhaben der Gestendaten

```
297 #if MODULE_APDS9960
298
299 /** Is new gesture data available?
300  * @param dev the APDS9960 device reference.
301  * @return result
302  **/
303 int apds99xx_data_ready_ges (const apds99xx_t *dev)
304 {
305     assert(dev != NULL);
306     DEBUG_DEV("", dev);
307     uint8_t reg = 0;
308
309
310     if (_reg_read(dev, APDS99XX_REG_GSTATUS, &reg, 1) !=
        APDS99XX_OK)
311     {
312         return -APDS99XX_ERROR_I2C;
313     }
314
315     return (reg & 1) ? APDS99XX_OK : -APDS99XX_ERROR_NO_DATA;
316 }
317
318
319
320 /** Read raw gesture data.
321  * @param dev the APDS9960 device reference.
322  * @param ges gesture structure
323  * @return result
324  **/
325 int apds99xx_read_ges_raw(const apds99xx_t *dev, apds99xx_ges_t
    *ges)
326 {
327     assert(dev != NULL);
328     assert(ges != NULL);
329     DEBUG_DEV("ges=%p", dev, ges);
330     uint8_t data[32*4] = { };
331
332
333     // Read all datasets of fifo for getting all data and for
        clearing interrupt
334     if (_reg_read(dev, 0xFC, data, 4 * 32) != APDS99XX_OK) {
335         return -APDS99XX_ERROR_RAW_DATA;
336     }
337
338
339     for(size_t i = 0; i < 32; i++) {
340         ges->FIFO[i].UP    = data[4 * i + 0];
341         ges->FIFO[i].DOWN  = data[4 * i + 1];
```

```
342         ges->FIFO[i].LEFT  = data[4 * i + 2];
343         ges->FIFO[i].RIGHT = data[4 * i + 3];
344     }
345
346     return APDS99XX_OK;
347 }
348 #endif
```

Des Weiteren wurden einige Stellen im Quellcode angepasst, damit der Sensor funktionsfähig ist.

5 Validierung des Treibers

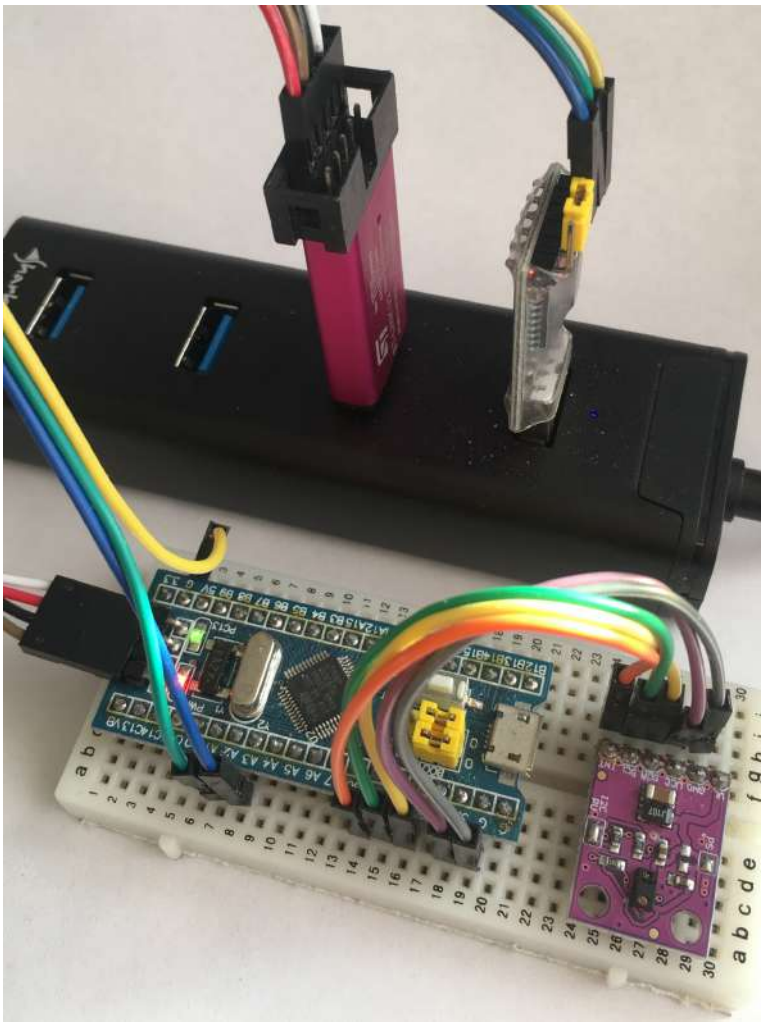

Abbildung 5.1: Aufbau des Funktionsdemonstrators

Damit der in dieser Arbeit erweiterte Treiber getestet werden kann, wird ein Demonstrator erstellt. Der Demonstrator ist ein Minimalsystem der verwendeten Technologien. Verwendet wird der Mikrocontroller STM32F103C8T6 auf einem sogenannten BluePill-Board und der Multifunktionssensor APDS9960. Diese werden, wie in Abbildung 5.1 dargestellt, zusammen auf einem Breadboard verbunden. Da der Multifunktionssensor auch auf einem Development-Board bestückt ist, werden keine Pull-Up Widerstände für I^2C benötigt. Der

Schaltplan in Abbildung 5.2 zeigt, wie die Pins miteinander verbunden sind. Aufschluss darüber, wie die Pinbelegung des Mikrocontrollers ist, kann im Anhang C.3 oder im Datenblatt nachgelesen werden (Seite 21ff. [4]).

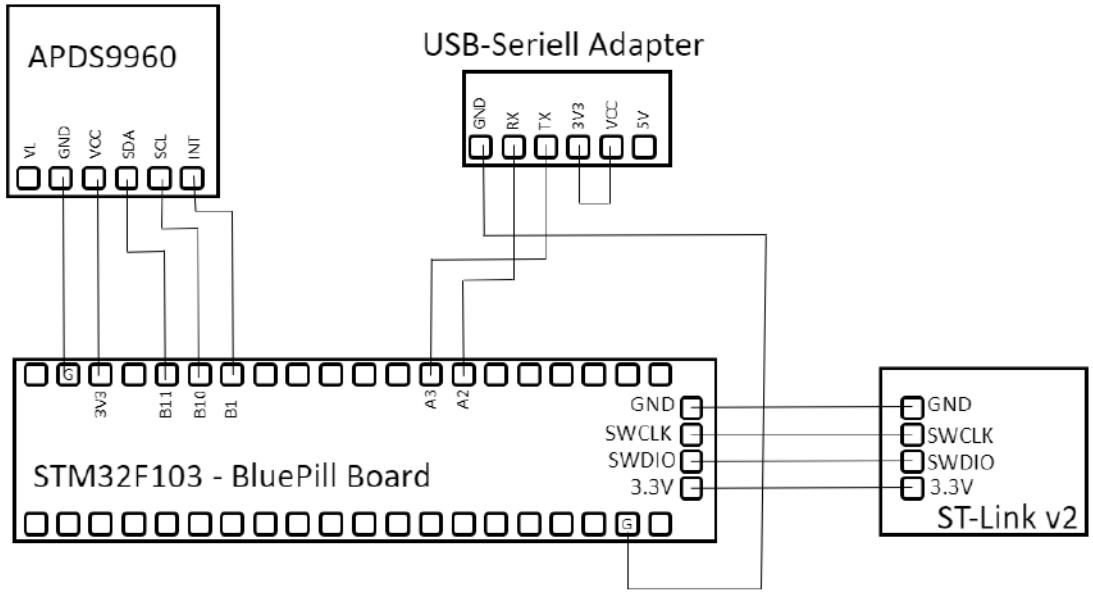

Abbildung 5.2: Schaltplan des Demonstrators

Ein Programmieradapter wird benötigt, um den Mikrocontroller programmieren zu können. Verwendet wird hier ein ST-Link v2 Klon aus China, die es schon zu sehr günstigen Preisen gibt [1]. Hierbei müssen nur vier Leitungen angeschlossen werden. Die Tabelle 5.1 gibt Aufschluss über die Pinbelegung des ST-Link v2 und dem BluePill-Board.

ST-Link v2	BluePill
3.3 V	VCC
SWDIO	SWDIO
SWCLK	SWCLK
GND	GND

Tabelle 5.1: Anschluss zwischen dem ST-Link v2 und dem BluePill

Nun fehlt nur noch ein USB zu UART Adapter, damit wir die Ausgabe des Mikrocontrollers auf einem Rechner sehen und unsere Daten vom Multifunktionssensor validieren können. Verwendet wird hier ein USB TO TTL HW-597, welcher auf dem IC CH340 basiert [2]. Vorteil dieses Adapters ist es, dass dieser sehr günstig aus China erworben werden kann. Bei diesem Adapter werden nur drei Pins benötigt, die wie in Tabelle 5.2 mit dem BluePill-Board angeschossen sind. Wichtig hierbei ist ein Jumper, der zwischen VCC und 3.3V

[1] https://www.ebay.de/i/272512190503 [Zugegriffen: 28.10.2019]
[2] https://www.ebay.de/itm/183934025981 [Zugegriffen: 28.10.2019]

angeschlossen ist. Der Jumper ist daher notwendig, weil dem Adapter hierbei mitgeteilt wird, dass der TTL Pegel 3.3V betragen soll und nicht die üblichen 5V. Mit einem falschen Pegel können wir den Mikrocontroller zerstören.

USB-Seriell	BluePill
TX	RX (Pin A3)
RX	TX (Pin A2)
GND	GND

Tabelle 5.2: Anschluss zwischen dem USB-Seriell-Adapter und dem BluePill

Nach der Hardware folgt die Software. Als Betriebssystem wird Windows 10 verwendet. Da RIOT allerdings für Linux oder Mac entwickelt wurde, wurde eine Linux Subsystem installiert. Hierbei handelt es sich um ein für Windows angepasstes Debian System [3]. Standardmäßig benötigt man noch git, um RIOT aus dem Internet herunterladen zu können. Zum Kompilieren der Programme wird das Programm make in der RIOT Umgebung verwendet. Da wir nicht alle Funktionen von Debian benutzen können, müssen wir make mit dem Kommando **make hexfile** mitteilen, dass eine Hex-Datei generiert werden soll. Diese Hex Datei wird anschließend in ein offizielles Programm von STM namens STM32 ST-LINK Utility geladen. Hiermit können wir unser selbstgeschriebenes Programm auf den Mikrocontroller programmieren [4]. Für den USB zu Seriell Adapter wird das Programm HTerm verwendet. HTerm ist ein sehr komplexes Programm um serielle Daten darzustellen und senden zu können [5]. Die Standardeinstellungen von HTerm entsprechen den seriellen Einstellungen von RIOT. So bleibt die Baudrate 115200.

5.1 Programmcode

Es werden zwei kleine Applikationen demonstriert. Die erste befasst sich mit dem Multifunktionssensor APDS9960 im Polling-Modus. Der Polling-Modus bedeutet, dass der Sensor periodisch abgefragt und auf die Daten gewartet wird. Das zweite Programm beschäftigt sich mit den Interrupts des Sensors. Interrupts unterbrechen den aktuell laufenden Quellcode und werden priorisiert ausgeführt. Ein Interrupt kann damit direkt verarbeitet werden, ohne aktiv auf die Daten warten zu müssen. Die Beispieldateien werden im RIOT Ordner **RIOT/examples/** erstellt.

5.1.1 Polling-Modus

Das Polling-Modus Projekt heißt apds9960-polling. Orientiert habe ich mich an den Testdateien von Gunar Schorcht. Das Makefile D.1 sieht sehr ähnlich aus zu

[3] https://www.microsoft.com/store/productId/9MSVKQC78PK6 [Zugegriffen: 28.10.2019]

[4] https://www.st.com/en/development-tools/stsw-link004.html [Zugegriffen: 28.10.2019]

[5] http://www.der-hammer.info/pages/terminal.html [Zugegriffen: 28.10.2019]

dem von Gunar (RIOT/tests/driver_apds99xx/Makefile). Lediglich der Name der Applikation und ein Default-Board wurde hinzugefügt und das inkludieren einer Testdatei wurde entfernt. In dem Makefile steht, dass die Applikation **apds9960-polling** heißt, dass das Default-Board als BluePill-Board definiert wurde und die Module apds9960, sowie den xtimer verwendet werden. xtimer wird aus dem Grund verwendet, da wir zwischen unseren Anfragen warten müssen.

Die **main.c** Datei D.2 ist sehr schlicht und einfach gehalten. Vor dem Inkludieren definieren wir das I2C Interface des Multifunktionssensor APDS9960. Weiterhin definieren wir eine Wartezeit, wie schnell die Hauptschleife durchlaufen darf. In diesem Beispiel beträgt die Wartezeit 100ms. In der Hauptfunktion wird der Sensor mit den Standardparametern initialisiert. Bei einem Fehler würde das Programm beendet werden. Nach der Initialisierung gelangen wir in die Hauptschleife. Hier warten wir jedes Mal 100 ms. Es wird die ganze Zeit gewartet, bis neue Gestendaten vorhanden sind. Sind im Sensor valide Daten vorhanden, werden die Rohdaten des Sensors, wie in Abbildung 5.3 gelesen und dargestellt. Da die Rohdaten als Datensets in einer FIFO Liste gespeichert sind, müssen diese mit einer Schleife ausgegeben werden. Leere Datensets werden dabei ignoriert.

```
main(): This is RIOT! (Version: 2016.10-devel-12266-g30f83-apds99xx)
APDS99XX proximity and gesture application

Initializing APDS99XX sensor
[OK]

Raw gesture data (UDLR):
  Dataset[0]:  11  18  20   3
  Dataset[1]:   3   1   0   0
Raw gesture data (UDLR):
  Dataset[0]:  12  18  25   6
Raw gesture data (UDLR):
  Dataset[0]:  27  33  48  19
  Dataset[1]:  60  57  68  57
  Dataset[2]:  64  50  47  74
Raw gesture data (UDLR):
  Dataset[0]:  51  35  18  61
  Dataset[1]:  18   9   1  26
  Dataset[2]:   4   0   0   5
```

Abbildung 5.3: Rohdaten vom Polling-Modus

5.1.2 Interrupt basiert

Der Projektname des Interrupt basierten Programms heißt apds9960-interrupt. Auch diese Applikation ist an die Implementierung von Gunar Schorcht angelehnt. Während Gunar die normalen Sensorfunktionen abdeckt, wird hier der Nutzen der Gestensteuerung gezeigt. Das Makefile D.3 ist eine angepasste Version aus der Testapplikation von Gunar (RIOT/tests/driver_apds99xx_full/-Makefile). Hier wurde wie im Polling-Modus, der Name der Applikation benannt, das BluePill-Board zum Standard gewählt und das RIOT Oberverzeichnis definiert.

Am Anfang des Quelltextes D.4 muss der APDS99xx Treiber konfiguriert werden. Dabei wird das I^2C Gerät und der Interrupt Pin definiert. Sobald der Multifunktionssensor ein Interrupt auslöst, wird dieser am Interrupt Pin des Mikrocontrollers erkannt.

Neben der Standard Ein- und Ausgabe, sowie den sensorspezifischen Dateien, werden auch Threads inkludiert. Ein Thread wird beim Erkennen eines Interrupt Signals ausgeführt. Da es in einer Anwendung mehrere Threads geben kann, werden diese mit Flags unterschieden. In dieser Anwendung wird das Flag unter APDSXX_IRQ_FLAG definiert. Eine globale Referenz des Threads wird unter t_main gespeichert.

Die ISR (Interrupt Service Routine) des Multifunktionssensors setzt lediglich die Information, dass die Interrupt Funktion ausgeführt wurde. Eine Interrupt Funktion sollte so kurz wie möglich erfolgen, damit der operationale Betrieb nicht eingeschränkt wird. Mit dem Setzen eines Flags symbolisiert man im Programm, dass ein Interrupt erkannt wurde und sobald das Programm Zeit hat, kümmert es sich um die Auswertung der neuen Daten.

In der Hauptfunktion wird zum Anfang ein Thread gestartet. Anschließend wird der Sensor mit den Standardeinstellungen initialisiert. Bei erfolgreicher Initialisierung werden die Interrupt Bedingungen konfiguriert. Zugelassen wird die Gestenerkennung und bei jedem Datenset (ein Datenset besteht aus den vier 8 Bit Informationen der vier Fotodioden) wird ein Interrupt ausgelöst.

In der Hauptschleife des Programms wird auf ein Flag des Interrupts gewartet, welches beim Ausführen des Threads gesetzt wird. Konnte dieses Flag erkannt werden, wird die Interrupt Quelle des Sensors abgefragt. Nun können unterschiedliche Interrupts des Sensors unterschiedlich behandelt werden. In dieser Applikation wird nur der Interrupt verwendet, der für die Erkennung einer Geste zuständig ist. Sollte eine Geste erkannt werden, werden die Daten wie in Abbildung 5.4 im Rohformat ausgegeben.

```
main(): This is RIOT! (Version: 2016.10-devel-12266-g30f83-apds99xx)
APDS99XX proximity and gesture test application with interrupts
Initializing APDS99XX sensor [OK]
Raw gesture data (UDLR):
  Dataset[0]:  48  78 101  15
Raw gesture data (UDLR):
  Dataset[0]:  70 107 142  26
Raw gesture data (UDLR):
  Dataset[0]:  50  75 103  14
Raw gesture data (UDLR):
  Dataset[0]:  46  61  94  10
Raw gesture data (UDLR):
  Dataset[0]:  59  72 108  11
Raw gesture data (UDLR):
  Dataset[0]:  81  87 150  20
```

Abbildung 5.4: Rohdaten vom interruptbasierten Modus

6 Ausblick

Der Multifunktionssensor APDS9960 wurde vollständig in RIOT integriert. Die Gestenfunktion liefert dem angeschlossen Hostsystem, wie alle weiteren Funktionen des Sensors, die Rohdaten zurück. Somit ist der Treiber vollständig. Für weiterführende Anwendungen die den Multifunktionssensor APDS9960 benutzen, können durch eigene Algorithmen die Rohdaten ausgewertet werden, um ggf. einfachere oder auch komplexe Gesten erkennen zu können. Da jede Anwendung sehr unterschiedlich sein können, wurde hier auf eine Analyse der Rohdaten verzichtet. Vor allem würde die Analyse weiter greifen, als eine Treiberimplementierung des Sensors. Daher wäre es für zukünftige Projekte, zum Beispiel in der Algorithmik, interessant eine komplexe Gestenerkennung zu implementieren.

Die SAUL Abstraktion von RIOT ist auf jeden Fall geeignet Multifunktionssensoren zu integrieren. Es kann ein Gerät mit mehreren Funktionen registriert werden. Auch die von SAUL aufgerufenen Funktionen können je nach Treiber die korrekten Daten ermitteln. Allerdings ist der Multifunktionssensor APDS9960 mit der Gestensteuerung nicht geeignet, da ein einziges Datenset (32 Bit) nicht in die Phydat Daten (16 Bit) passt. Daher müsse weiterhin speziell für eine Gestensteuerung evaluiert werden, ob die Daten ggf. normiert werden können oder ob das Verhältnis zwischen den Richtungsdaten des Sensors ausreichen würde. Dies führt allerdings zu dem Entschluss, dass spezielle Anwendungen keine vollständige Analyse der Rohdaten betreiben können, da diese schon manipuliert wurden. Eventuell kann es auch eine andere Form der Datenspeicherung für Gestendaten existieren.

Ziel dieser Arbeit war es eine Schnittstelle der Gestensensorfunktion des APDS9960 in RIOT zu integrieren. Interessant wären weitere Projekte, die die Gestensteuerung konkret in eine Anwendung einbinden. Der erarbeitete Funktionsdemonstrator stellt zwar die Rohdaten dar, allerdings kann damit noch kein Fernsehprogramm gewechselt werden.

7 Fazit

Meiner Meinung nach wurde das Ziel dieser Bachelorarbeit erreicht. Es wurde eine Schnittstelle für den Multifunktionssensor APDS9960 erarbeitet. Diese Schnittstelle in Form eines Treibers bindet den Sensor in das Betriebssystem RIOT ein. Des Weiteren wurde evaluiert, ob die SAUL Abstraktion Multifunktionssensoren einbinden kann. Leider ist es ohne weiteres nicht möglich, die Gestenfunktion vom APDS9960 in die SAUL Abstraktion zu integrieren, weil die aufgenommene Datengröße nicht in die Datenstrukturen von SAUL passen.

Zum Testen des Treibers wurde ein Funktionsdemonstrator mit zwei unterschiedlichen Modi des Sensors APDS9960 erstellt. Dieser Funktionsdemonstrator liefert in beiden Fällen das gewünschte Ergebnis zurück. Der eine Modus verarbeitet die Daten wartend. Das bedeutet, dass der Mikrocontroller die ganze Zeit auf ein Event wartet. Im Interrupt basierten Modus muss nicht auf ein Event explizit gewartet werden, sondern wird dann ausgeführt, sobald dieses Event stattfindet. In einer Produktion würden Interrupts verwendet werden, damit der Kontrollfluss der normalen Anwendung nicht beeinträchtigt wird.

Literaturverzeichnis

[1] A. Technologies, "Datenblatt; apds-9960; digital proximity, ambient light, rgb and gesture sensor." https://cdn.sparkfun.com/assets/learn_tutorials/3/2/1/Avago-APDS-9960-datasheet.pdf. Zugegriffen: 29.05.2019.

[2] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, pp. 4428–4440, 12 2018.

[3] G. Schorcht, "drivers: add driver for apds99xx ambient light and proximity sensors." https://github.com/RIOT-OS/RIOT/pull/10420. Zugegriffen: 27.04.2019.

[4] STMicroelectronics, "Stm32f103x8, stm32f103xb: Medium-density performance line arm-based 32-bit mcu with 64 or 128 kb flash, usb, can, 7 timers, 2 adcs, 9 com. interfaces." https://www.st.com/resource/en/datasheet/stm32f103c8.pdf. Zugegriffen: 28.10.2019.

[5] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot documentation." https://doc.riot-os.org. Zugegriffen: 04.10.2019.

[6] N. Semiconductors, "Um10204 - i2c-bus specification and user manual." https://www.nxp.com/docs/en/user-guide/UM10204.pdf. Zugegriffen: 19.10.2019.

[7] P. Semiconductors, "An10216 - i2c manual." https://www.nxp.com/docs/en/application-note/AN10216.pdf. Zugegriffen: 28.10.2019.

[8] F. J. Gensicke, "Berechnung der kapazität von leiterbahnen." https://www.elektronikentwickler-aachen.de/allgemeines/kapazitaet_leiterbahnen.htm. Zugegriffen: 28.10.2019.

[9] H. Petersen, "saul.h." https://doc.riot-os.org/saul_8h.html. Zugegriffen: 28.10.2019.

[10] H. Petersen, "saul_reg.h." https://doc.riot-os.org/saul__reg_8h.html. Zugegriffen: 28.10.2019.

[11] H. Petersen, "phydat.h." https://doc.riot-os.org/phydat_8h.html. Zugegriffen: 28.10.2019.

[12] K. Schleiser, O. Hahm, H. Petersen, and A. Abadie, "auto_init.h." https://doc.riot-os.org/auto__init_8h.html. Zugegriffen: 28.10.2019.

Anhang

A Näherungssensorsteuerung

Register/Bit	Address	Description
ENABLE<PON>	0x80<0>	Power ON
ENABLE<PEN>	0x80<2>	Proximity Enable
ENABLE<PIEN>	0x80<5>	Proximity Interrupt Enable
PILT	0x89	Proximity low threshold
PIHT	0x8B	Proximity high threshold
PERS<PPERS>	0x8C<7:4>	Proximity Interrupt Persistence
PPULSE<PPLEN>	0x8E<7:6>	Proximity Pulse Length
PPULSE<PPULSE>	0x8E<5:0>	Proximity Pulse Count
CONTROL<PGAIN>	0x8F<3:2>	Proximity Gain Control
CONTROL<LDRIVE>	0x8F<7:6>	LED Drive Strength
CONFIG2<PSIEN>	0x90<7>	Proximity Saturation Interrupt Enable
CONFIG2<LEDBOOST>	0x90<5:4>	Proximity/Gesture LED Boost
STATUS<PGSAT>	0x93<6>	Proximity Saturation
STATUS<PINT>	0x93<5>	Proximity Interrupt
STATUS<PVALID>	0x93<1>	Proximity Valid
PDATA	0x9C	Proximity Data
POFFSET_UR	0x9D	Proximity Offset UP/RIGHT
POFFSET_DL	0x9E	Proximity Offset DOWN/LEFT
CONFIG3<PCMP>	0x9F<5>	Proximity Gain Compensation Enable
CONFIG3<PMSK_U>	0x9F<3>	Proximity Mask UP Enable
CONFIG3<PMSK_D>	0x9F<2>	Proximity Mask DOWN Enable
CONFIG3<PMSK_L>	0x9F<1>	Proximity Mask LEFT Enable
CONFIG3<PMSK_R>	0x9F<0>	Proximity Mask RIGHT Enable
PICLEAR	0xE5	Proximity Interrupt Clear
AICLEAR	0xE7	All Non-Gesture Interrupt Clear

Abbildung A.1: Näherungssensorsteuerung[1]

[1] Table 1, Seite 10 [1]

B Gestensensorsteuerung

Register/Bit	Address	Description
ENABLE<PON>	0x80<0>	Power ON
ENABLE<GEN>	0x80<6>	Gesture Enable
GCONFIG4<GIEN>	0xAB<1>	Gesture Interrupt Enable
GPENTH	0xA0	Gesture Proximity Entry Threshold
GEXTH	0xA1	Gesture Exit Threshold
GCONFIG1<GFIFOTH>	0xA2<7:6>	Gesture FIFO Threshold
GCONFIG1<GEXMSK>	0xA2<5:2>	Gesture Exit Mask
GCONFIG1<GEXPERS>	0xA2<1:0>	Gesture Exit Persistence
GCONFIG2<GGAIN>	0xA3<6:5>	Gesture Gain Control
GCONFIG2<GLDRIVE>	0xA3<4:3>	Gesture LED Drive Strength
GCONFIG2<GWTIME>	0xA3<2:0>	Gesture Wait Time
STATUS<PGSAT>	0x93<6>	Gesture Saturation
CONFIG2<LEDBOOST>	0x90<5:4>	Gesture/Proximity LED Boost
GOFFSET_U	0xA4	Gesture Offset, UP
GOFFSET_D	0xA5	Gesture Offset, DOWN
GOFFSET_L	0xA7	Gesture Offset, LEFT
GOFFSET_R	0xA9	Gesture Offset, RIGHT
GPULSE<GPULSE>	0xA6<5:0>	Pulse Count
GPULSE<GPLEN>	0xA6<7:6>	Gesture Pulse Length
GCONFIG3<GDIMS>	0xAA<1:0>	Gesture Dimension Select
GCONFIG4<GIEN>	0xAB<1>	Gesture Interrupt Enable
GCONFIG4<GMODE>	0xAB<0>	Gesture Mode
GFLVL	0xAE	Gesture FIFO Level
GSTATUS<GFOV>	0xAF<1>	Gesture FIFO Overflow
GSTATUS<GVALID>	0xAF<0>	Gesture Valid
GFIFO_U	0xFC	Gesture FIFO Data, UP
GFIFO_D	0xFD	Gesture FIFO Data, DOWN
GFIFO_L	0xFE	Gesture FIFO Data, LEFT
GFIFO_R	0xFF	Gesture FIFO Data, RIGHT
CONFIG1<LOWPOW>	0x8D	Low Power Clock Mode

Abbildung B.2: Gestensensorsteuerung[2]

[2]Table 4, Seite 14 [1]

C Pinbelegung des BluePill-Boards

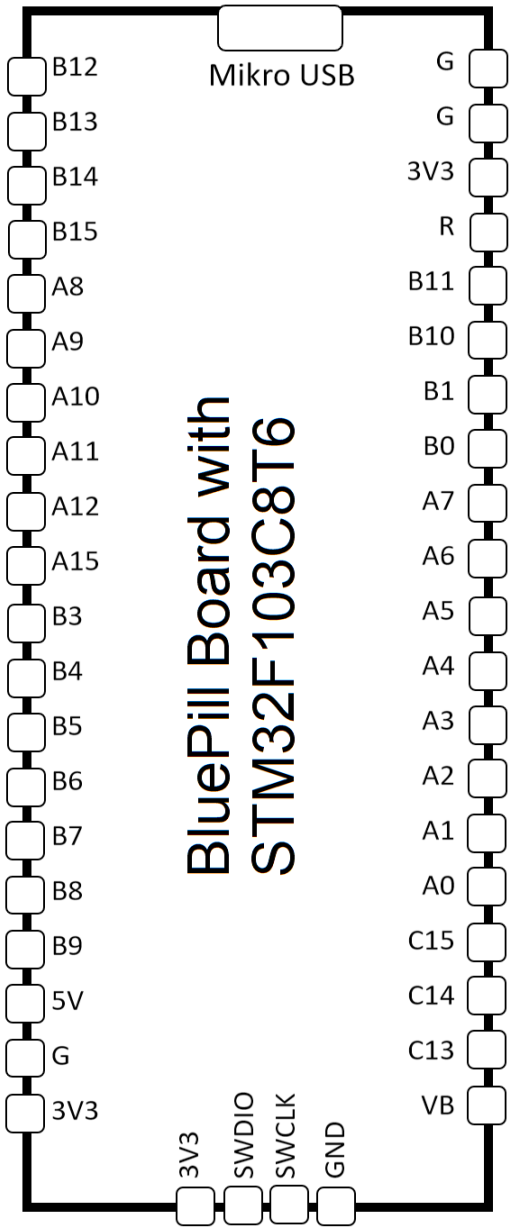

Abbildung C.3: STM32F103 - Pinbelegung

D Quelltext

D.1 Polling-Modus

Listing D.1: Makefile von apds9960-polling

```
APPLICATION = apds9960-polling
BOARD ?= bluepill
RIOTBASE ?= ../..

DRIVER ?= apds9960

USEMODULE += $(DRIVER)
USEMODULE += xtimer

include $(RIOTBASE)/Makefile.include
```

Listing D.2: Quelltext von apds9960-polling

```
/** apds9960_polling: main.c
 * @author  Daniel Lux <lux@uni-bremen.de>
 * @brief   This is an example application for the gesture and
    proximity function of the sensor APDS9960.
 */



// define I2C device on bluepill board
#define APDS99XX_PARAM_DEV          I2C_DEV(1)



/* INCLUDES */
#include <stdio.h>
#include "xtimer.h"
#include "apds99xx.h"
#include "apds99xx_params.h"



/* DEFINES */
#define SLEEP   (100 * US_PER_MS)



/** Entrypoint of this application.
 * @param none
 * @return successful or faillure application exit
 **/
```

```
int main(void)
{
    apds99xx_t dev;
    apds99xx_ges_t ges;


    puts("APDS99XX␣proximity␣and␣gesture␣application\n");
    puts("Initializing␣APDS99XX␣sensor␣");


    // Initialize sensor
    if (apds99xx_init(&dev, &apds99xx_params[0]) == APDS99XX_OK
       ) {
        puts("[OK]\n");
    } else {
        puts("[Failed]");
        return 1;
    }


    // main-loop
    while (1) {

        // sleep every loop 100ms
        xtimer_usleep(SLEEP);

        /* Is gesture data available? */
        if (apds99xx_data_ready_ges(&dev) == APDS99XX_OK) {
            if (apds99xx_read_ges_raw(&dev, &ges) ==
               APDS99XX_OK) {
                printf("Raw␣gesture␣data␣(UDLR):\n");

                for (size_t i = 0; i < 32; i++)
                {
                    if (ges.FIFO[i].UP == 0 && ges.FIFO[i].DOWN
                        == 0 && ges.FIFO[i].LEFT == 0 && ges.
                       FIFO[i].RIGHT == 0)
                        continue;

                    printf("␣␣Dataset[%d]:␣%3d␣%3d␣%3d␣%3d\n",
                       i, ges.FIFO[i].UP, ges.FIFO[i].DOWN, ges
                       .FIFO[i].LEFT, ges.FIFO[i].RIGHT);
                }
            }
        }
    }

    return 0;
}
```

D.2 Interruptbasiert

Listing D.3: Makefile von apds9960-interrupt

```
APPLICATION = apds9960-interrupt
BOARD ?= bluepill
RIOTBASE = ../..

DRIVER ?= apds9960
USEMODULE += $(DRIVER)
USEMODULE += apds99xx_full
USEMODULE += core_thread_flags

include $(RIOTBASE)/Makefile.include
```

Listing D.4: Quelltext von apds9960-interrupt

```
/** apds9960-interrupt: main.c
 * @author  Daniel Lux <lux@uni-bremen.de>
 * @brief   This is an example application for the gesture and
    proximity function of the sensor APDS9960 with the function
     of interrupts.
 **/


// configure apds99xx driver with i2c device and interrupt pin
#define APDS99XX_PARAM_DEV          I2C_DEV(1)
#define APDS99XX_PARAM_INT_PIN      (GPIO_PIN(PORT_B, 1))



/* INCLUDES */
#include <stdio.h>
#include "thread.h"
#include "thread_flags.h"
#include "apds99xx.h"
#include "apds99xx_params.h"



/* DEFINES */
#define APDS99XX_IRQ_FLAG  0x1000



/* GLOBAL VARIABLES */
thread_t* t_main;


```

```
32 /** APDS9960 hardware pin interrupt.
33  *
34  * This function is needed to set a flag to indicate an
       interrupt. It is
35  * recommended to leave this function as short as possible.
36  *
37  * @param arg arguments
38  * @return none
39  **/
40 static void apds99xx_isr (void *arg)
41 {
42     (void)arg;
43     thread_flags_set(t_main, APDS99XX_IRQ_FLAG);
44 }
45
46
47
48 /** Entrypoint of this application.
49  * @param none
50  * @return exit status
51  **/
52 int main(void)
53 {
54     apds99xx_t dev;
55     t_main = (thread_t*)sched_threads[sched_active_pid];
56
57
58     printf("APDS99XX␣proximity␣and␣gesture␣test␣application␣
          with␣interrupts\n");
59     printf("Initializing␣APDS99XX␣sensor␣");
60
61
62     // initialize sensor with default params
63     if (apds99xx_init(&dev, &apds99xx_params[0]) == APDS99XX_OK
          ) {
64         printf("[OK]\n");
65     }
66     else {
67         printf("[Failed]");
68         return 1;
69     }
70
71
72     /* configure interrupts */
73     apds99xx_int_config_t int_cfg = {
74         .als_int_en = false,
75         .als_pers = 0,
76
77         .prx_int_en = false,
78         .prx_pers = 1,
```

```
79          .prx_thresh_low = 0,
80          .prx_thresh_high = 20,
81
82          .ges_int_en = true,
83          .ges_pers = 0,
84          .ges_fifo_th = APDS99XX_GES_FIFO_TH_1,
85      };
86      apds99xx_int_config(&dev, &int_cfg, apds99xx_isr, 0);
87
88
89      // main-loop
90      while (1) {
91
92          thread_flags_wait_one(APDS99XX_IRQ_FLAG);
93
94
95          // get triggered interrupt
96          apds99xx_int_source_t int_src;
97          apds99xx_int_source(&dev, &int_src);
98
99
100         // gesture interrupt?
101         if (int_src.ges_int) {
102             apds99xx_ges_t ges;
103
104             if (apds99xx_read_ges_raw(&dev, &ges) ==
                    APDS99XX_OK) {
105                 printf("Raw␣gesture␣data␣(UDLR):\n");
106
107                 for (size_t i = 0; i < 32; i++)
108                 {
109                     if (ges.FIFO[i].UP == 0 && ges.FIFO[i].DOWN
                            == 0 && ges.FIFO[i].LEFT == 0 && ges.
                        FIFO[i].RIGHT == 0)
110                         continue;
111
112                     printf("␣␣Dataset[%d]:␣%3d␣%3d␣%3d␣%3d\n",
                        i, ges.FIFO[i].UP, ges.FIFO[i].DOWN, ges
                        .FIFO[i].LEFT, ges.FIFO[i].RIGHT);
113                 }
114
115             }
116         }
117     } // main-loop
118
119     return 0;
120 }
```