Konstruktion eines Datenloggers für einen Ballonflug Richtung Stratosphäre

Eine Projektarbeit von Maximilian Hüter
Kurs: Projektkurs Physik
Q1 2020/2021

Inhaltsverzeichnis:

1. Einleitung

1.1. Eine Sonde Richtung Stratosphäre - Unser Projekt

Zunächst einmal beschreibe ich unser Projekt, dem wir uns im Rahmen des Physik-Projektkurses angenommen haben. Das Projekt besteht darin, dass wir einen Wetterballon bzw. eine Styropor-Box, die an an einem großen Ballon hängt, in die Luft aufsteigen lassen. Innerhalb dieser Styropor-Box befinden sich dann Sensoren, mit denen wir Daten sammeln. Als Beispiel kann man mit einem Temperatur-Sensor die Temperatur messen, die sich abhängig von der Höhe verändert. Jedoch kann man jetzt nicht einfach ein Thermometer in die Box legen und die Mission kann los gehen. Zum einen ist ein Thermometer, das z.B. mit Alkohol funktioniert, nicht all zu genau, aber noch ein größeres Problem ist, dass wir niemanden mit in die Box stecken können, der dann die Werte vom Thermometer abliest und notiert. Somit braucht man einen Datenlogger.

1.2. Was ist ein Datenlogger?

Ein Datenlogger ist quasi ein Gerät, das Sensoren auslesen kann und die ausgelesenen Daten dann in einer bestimmten Form speichern kann. Wie, als wenn eine Person mit in der Box fliegen würde, die Werte abliest und auf ein Blatt Papier notiert.

Ein bekannter Datenlogger ist der STRATO3[1]. Dieser Datenlogger hat ein Display, ein GPS-Modul, ein Luftdruck-Sensor, ein Luftfeuchtigkeit-Sensor, einen Außen- sowie einen Innentemperatur-Sensor. Die Daten, die er durch diese Sensoren erhält, speichert er auf einer MicroSD-Karte.

1.3. Das Thema dieser Arbeit

Hört sich ja schon nach ganz schön vielen Daten an. Allerdings wollten wir noch einen eigenen Datenlogger mitfliegen lassen, an dem wir unsere eigenen Sensoren anbringen können. Wir haben uns dann für weitere Temperatursensoren und einen Neigungs- und Beschleunigungs-Sensor entschieden.

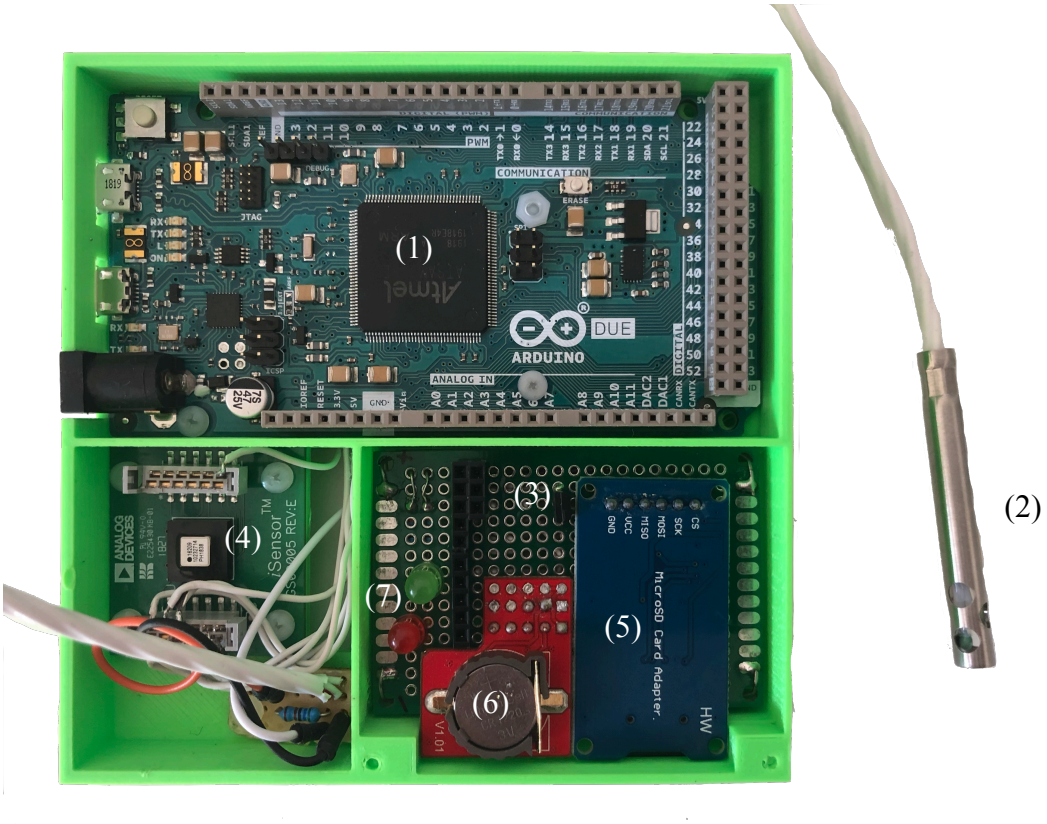
Somit habe ich mich wegen bereits vorhandenen Erfahrungen dafür entschieden diesen zu bauen. Um den Prozess, wie ich den Datenlogger konstruiert und programmiert habe

[1] Dies ist ein kommerziell produzierter Datenlogger, der von der Firma Stratoflights vertrieben wird.

sowie dessen Funktionsweise, soll es in den folgenden Texten gehen. Dabei werde ich auf jedes einzelne Teil in jeder Kategorie[2] jeweils eingehen, aber dann auch die Funktionsweise des Datenloggers insgesamt erläutern.

2. Ein Überblick über den Datenlogger

Abb. 1: Der Datenlogger ohne Deckel von oben, eigene Aufnahme



Auf der Abbildung sieht man den fertigen Datenlogger, so wie er auch auf der Mission Richtung Stratosphäre geflogen ist. Allerdings wurde hier der Deckel entfernt. Somit sind auf der Abbildung alle elektronischen Komponenten sichtbar, bis auf die Batterie, die den Datenlogger mit Strom versorgt. An den Zahlen im Bild erkennt man folgende Komponenten:

1. Arduino Due
2. NTC Temperatur-Sensor
3. TSIC506F digitaler Temperatur-Sensor
4. Neigungs- und Beschleunigungs-Sensor
5. SD-Karten-Modul

[2] Damit meine ich die Funktionsweise und Programmierung, aber auch spezielle Voraussetzungen des Bauteils und auch Probleme, die es damit gab.

6. RTC Echtzeit-Uhr
7. Status LEDs
8. Gehäuse aus Plastik (alles, was grün ist)

Die Komponenten werden jeweils in ihrem eigenen Abschnitt im Detail erläutert.

3. Der Arduino Due

3.1. Was ist ein Arduino?

Arduinos sind Micro-Controller, die von der Organisation Arduino-Foundation[3] produziert werden. Was ein Micro-Controller ist? Nun einen Micro-Controller kann man sich vorstellen wie einen kleinen Computer, der aber keinen Monitor oder eine Maus und Tastatur benötigt, um sein volles Potential auszuschöpfen. Micro-Controller sind sehr gut dafür geeignet, selbstständig kleine Aufgaben zu bewältigen. Es sind keine High-End-Geräte[4], aber zum Auslesen eines Sensors und das Speichern der ausgelesenen Daten, reichen sie bei Weitem aus. Also genau das, was unser Datenlogger braucht!

Allerdings kann man nicht einfach einen Arduino kaufen, der einem gefällt, dann anschließen und los gehts. Man muss Micro-Controller selber programmieren, ihnen also beibringen, was sie können und tun sollen.

3.2. Programmierung von Arduinos

Um einen Arduino zu programmieren, benötigt man eine Entwicklungsumgebung. Eine Entwicklungsumgebung ist eine Art Text-Editor auf dem Hauptcomputer. Die Entwicklungsumgebung nimmt das Programm und lässt es von einem Compiler in Maschinensprache[5] übersetzen, damit es auf den Arduino über ein Kabel hochgeladen werden kann und der Arduino so versteht, was er tun soll. Ein Arduino wird in einer C bzw. C++ sehr ähnlichen Sprache programmiert. Eine Entwicklungsumgebung, die diese Sprache übersetzen und hochladen kann, ist die Arduino IDE, die auch ich

---

[3] Link zur Homepage: https://www.arduino.cc

[4] Ein High-End-Gerät ist z.B. ein Nasa-Computer oder auch ein sehr aktueller Computer, der mehr bewältigen kann, als der Normalverbraucher benötigt.

[5] Sprache in der alle Computer arbeiten, die für den Mensch schwer zu verstehen ist.

verwendet habe. Man kann aber auch multisprachige Entwicklungsumgebungen verwenden, wie z.B. Visual Studio Code[6].

Die Ähnlichkeit zu C und C++ wird der „Arduino-Sprache“ eigentlich nur zugeschrieben, weil es zwei festgelegte Codeblöcke gibt, die ausgefüllt werden sollten:

```
1.  void setup() {
2.
3.  }
4.  void loop() {
5.
6.  }
```

Ohne diese Codeblöcke und ein Hinzufügen einer „main()“-Methode wäre die Sprache, mit der man den Arduino programmiert, identisch zu C bzw. C++.

In den ersten drei Zeilen steht der „void setup()“-Block. Dort werden vorzugsweise Attribute bzw. Variablen deklariert, aber generell wird alles, was dort drin steht, einmal ausgeführt. Darauf folgt der „void loop()“-Block. Dieser Teil wiederholt sich unendlich oft und kann nur beendet werden, indem man den Arduino von der Stromversorgung trennt. Hier steht somit die Aufgabe des Arduinos, die er immer wieder durchführt. Es ist zu vergleichen mit der „main()“-Methode.

3.3. Hardware des Arduinos

Auch wenn man seinen Arduino nun programmiert hat, dann ist noch immer nicht alles getan, um wie in unserem Fall einen laufenden Datenlogger zu haben. Denn der Arduino kann ohne Hilfe z.B. keine Temperaturen messen. Dafür benötigt er dann Sensoren, also Erweiterungen seiner Hardware. Um extra Hardware an den Arduino anzuschließen, besitzt er Pins, in die man Kabel stecken kann, die dann zum Sensor führen.

Der Arduino besitzt zwei Typen von Pins: analoge und digitale. Dabei gibt es nur analoge Eingänge, aber digitale gibt es auch als Ausgänge. Manche Pins haben auch noch die Funktion als digitaler Bus[7] zu dienen.

---

[6] Link zur Homepage: https://code.visualstudio.com

[7] Ein Bus ist eine Schnittstelle, die komplexere digitale Signale übertragen kann z.B.: SPI oder I2C

Es gibt verschiedene Modelle von Arduinos, die unterschiedliche viele Pins, spezielle Bus-Systeme, EEROM-Speicher, Prozessorleistung und vielen andere Spezifikationen haben. Deshalb ist es wichtig, sich für einen passenden Arduino für sein Projekt zu entscheiden.

3.4. Warum der Arduino Due?

Ein bekannter Arduino, der für die meisten Projekte reicht, ist der Arduino UNO. Er ist sozusagen das Einsteiger-Modell. Für unseren Datenlogger haben wir uns aber für den Arduino Due entschieden. Dieser bietet viele Möglichkeiten, die der Arduino UNO nicht bieten kann. Für unser Projekt wichtige Vorteile waren die folgenden:

- Der Arduino Due besitzt ganze 114 Pins, sodass man sich keine Sorgen um zu wenige Pins machen muss. Der Arduino UNO hat nur 32.
- Ein weiterer Vorteil der Pins vom Arduino Due ist, dass die analogen Eingänge die eingehende Spannung in einer 12-bit Auflösung aufnehmen können. Der Arduino UNO schafft das Ganze nur in 10-bit, sodass der Arduino Due vier mal so genaue Werte liefert wie der Arduino UNO. Warum das für unser Projekt wichtig ist, wird im Kapitel „4. NTC - ein besonderer Temperatur-Sensor“ erklärt.
- Der Arduino Due basiert auf einem 32-bit ARM[8] Cortex-M3 SoC[9], was ihn sehr leistungsfähig macht und ihm viel Speicherplatz für das Programm bietet. Dazu ist dieser Chip auf 84 MHz getaktet und bietet eine hohe Arbeitsgeschwindigkeit.
- Alle aktuellen Top-Sensoren laufen mit einer Spannung von 3,3V. Der Arduino Due läuft auch mit 3,3V und kann somit problemlos mit ihnen kommunizieren.

[8] ARM ist eine Architektur für Computer-Prozessoren.

[9] SoC steht für System-on-a-Chip und beschreibt, dass alle notwendigen Komponenten auf einem Chip sind.

4. NTC - ein besonderer Temperatur-Sensor

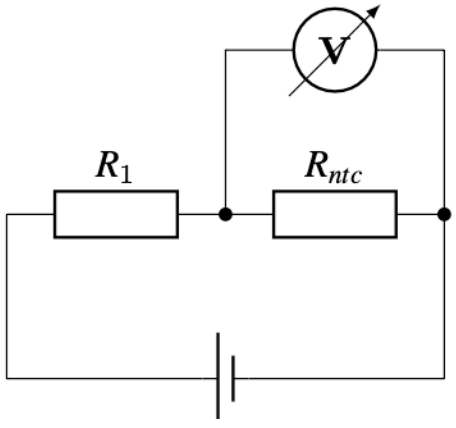
4.1. Was ist ein NTC?

NTC bedeutet Negative Temperature Coefficient und ist ein Widerstand, der, je kälter es wird, seinen Widerstand erhöht, das nennt man dann einen Heißleiter. Dabei ändert sich der Widerstand so, dass er in der Kälte immer kleinere Schritte macht. Das bedeutet, er wird in kälteren Bereichen sehr genau. Somit liegt auf der Hand, warum wir uns für so einen Temperatur-Sensor entschieden haben, denn, wie man vermutet und auch weiß, wird es in der höheren Atmosphäre immer kälter, so dass solch ein Heißleiter sich sehr gut für die Messung der dort herrschenden Temperaturen eignet.

Dies ist auch der Grund warum wir ihn mit fliegen lassen, obwohl der STRATO3 ja schon einen Außentemperatur-Sensor besitzt. Der NTC eignet sich für die Messungen einfach besser. Außerdem kann man die beiden Temperaturen dann auch noch miteinander vergleichen, um ein Gefühl dafür zu bekommen, ob die gemessenen Werte plausibel sind.

4.2. Schaltung und Auslesen des NTC

Die Änderung eines Widerstands kann der Arduino nicht erfassen, sodass man über einen Spannungsteiler die Spannung am NTC messen muss:



Dabei ist Widerstand R_{ntc} der NTC, der vom Arduino, wie es hier das Spannungsmessgerät macht, ausgelesen wird. Der Widerstand R_1 ist so gewählt, dass man eine möglichst weiten Messbereich vom NTC abdeckt. Unser Widerstand R_1 für

den Datenlogger sollte einen Widerstand von 5,6 kΩ haben. Für genaue Berechnungen wurde der exakte Widerstand des verbauten Widerstands von 5588,7 Ω gemessen.
Mit dem Widerstand des NTC kann die gemessene Temperatur bestimmt werden, so dass aus der gemessenen Spannung ein Widerstand errechnet werden muss.

4.3. Herleitung der Formel

Um aus der gemessenen Spannung den Widerstand des NTC zu berechnen, wird die Spannungsteiler-Gleichung benutzt:

$$R_{ntc} = \frac{U_{ntc}}{U_{ges} - U_{ntc}} \cdot R_1$$

Diese hier wurde schon nach R_{ntc} umgestellt, damit der Arduino aus den bekannten Daten den Widerstand berechnen kann. Dadurch, dass der Widerstand nicht antiproportional zur Temperatur steigt, sondern immer schneller wächst, kann es sich um eine Exponentielle Funktion handeln. Die allgemeine Formel zum Berechnen solcher Temperatursensoren sieht so aus:

$$T = \frac{1}{\frac{\ln\left(\frac{R_{ntc}}{A}\right)}{B} + \frac{1}{T_{25}}}$$

Diese Funktion wurde aus eine Exponentialfunktion nach T in Kelvin umgestellt, da diese zuvor R(T) war. Bei den Funktionen gibt es gewisse Konstanten:

- A ist der Widerstand des NTC bei 25°C. Bei unserem Sensor beträgt dieser genau 1000Ω.
- B ist der so genannte B-Wert. Dieser in der Ursprungsfunktion R(T) ist ein Faktor im Exponenten und beschreibt eine materialbehaftete Gegebenheit des Sensors. Unser B-Wert beträgt 3224 K, der eine verbesserte Version von dem aus dem Datenblatt ist.
- T_{25} gibt lediglich die Temperatur 25°C an bzw. 298,15 Kelvin, denn für diese Formeln werden Temperaturen in Kelvin benutzt.

Eine noch genauere Version dieser Formel ist die Steinhart-Hart-Gleichung[10]. Diese ergänzt einen Summanden unter dem Bruchstrich, der ein Produkt aus einen der Steinhart-Hart-Koeffizienten (hier ist a_3 gemeint) und einer Potenz ist:

[10] Formeln und Definitionen wurden von folgenden Link entnommen: https://de.wikipedia.org/wiki/Steinhart-Hart-Gleichung

$$\frac{1}{T} = a_0 + a_1 \cdot \ln\left(\frac{R_{ntc}}{A}\right) + a_3 \cdot \left(\ln\left(\frac{R_{ntc}}{A}\right)\right)^3$$

$$a_0 = \frac{1}{T_{25}}; \quad a_1 = \frac{1}{B}; \quad a_3 = 0{,}00000112$$

Somit sind a_0 und a_1 wieder zu finden. Den a_3-Wert haben wir mit einer Tabelle optimiert, indem wir ihn so lange angepasst haben, bis eine errechnete Kennlinie möglichst nahe an der des Datenblattes war.

4.4. Implementierung in den Arduino-Quellcode

Mit diesen Rechnungen ist es dem Arduino möglich, aus der gemessenen Spannung eine Temperatur in 1/Kelvin zu berechnen, was wie folgt umgesetzt wurde:

```
1.  wert = analogRead(analogPin);
2.  untc = wert/4095 * u0;
3.  rntc = untc/(u0-untc) * rvor;
4.  logResist = log(rntc/nennResist);
5.  oneByTempKelvin = AZero + AOne*logResist +
    AThree*pow(logResist, 3);
6.  currentTemp = makeToCelsius(oneByTempKelvin);
7.  sample[c] = currentTemp;
8.  c++;
9.
10. float makeToCelsius(float kelvinTemp){
11.   return 1/kelvinTemp - 273.15;
12. }
```

Anhand der meisten Namen ist die Variable aus den vorherigen Formeln zu erkennen. Eigens benannte Variablen werden erklärt.

In der ersten Zeile wird mit „analogRead()“ der Pin des Arduinos ausgelesen, an dem der NTC-Sensor sich befindet. Der gelesene Wert wird in der Variable „wert“ in einer 12-bit Auflösung gespeichert. Um aus diesem Bitwert eine Spannung in Volt zu berechnen, wird in Zeile zwei die Spannung pro Bitwert der Maximalspannung U_0 berechnet und somit das Gemessene in eine Spannung umrechnen zu können. Bei 3,3V und einer 12-bit Auflösung (0 bis 4095) beträgt somit 1 Bitwert die Spannung ≈ 0,00081V.

Danach wird der Widerstand des NTC mit der Spannungsteiler-Gleichung berechnet, wobei „rvor“ der Widerstand R_1 ist.

Mit der Variable „logResist“ wird in Zeile vier das Ergebnis des natürlichen Logarithmus aus dem Widerstand des NTC durch den Widerstand bei 25°C, hier „nennResist“, gespeichert. Um den natürlichen Logarithmus im Arduino-Code aufzurufen, muss man nur „log()“ schreiben, da die eulerische Zahl als Standard-Basis in der Math-Library steht.

Die Steinhart-Hart-Gleichung befindet sich in Zeile fünf. Vor dem „void setup()“ wurden die Steinhart-Hart-Koeffizienten bereits definiert. „pow()“ ist die Potenz-Methode der Math-Library. Als erster Parameter wird die Basis übergeben und dann der Exponent.

Um aus der Temperatur in 1/Kelvin normale Grad Celsius zu machen, habe ich die „makeToCelsius()“ Methode geschrieben, die mit dem übergebenen Parameter in Zeile sechs dieses zurück gibt.

Durch das Sampling in Zeile sieben und acht, also dem Aufnehmen mehrerer Werte innerhalb eines Messintervalls, benötigt man einen Ort, um alle diese Werte zu speichern, was mit dem Array „sample“ gemacht wird. Dafür wird auch die Laufvariable „c“ benötigt, die sich bis 20 Samples immer weiter erhöht. Ein Messintervall ist 2 Sekunden lang, wodurch also 10 Samples pro Sekunde aufgenommen werden.

Mit der folgenden Methode werden alle Samples addiert und dann durch 20 geteilt, um einen Schnitt des Messintervalls zu erhalten:

```
1.  float giveTemp(){
2.    float total = 0;
3.    float temp = 0;
4.    for(int i = 0; i < c; i++){
5.      total = total + sample[i];
6.    }
7.    return temp = total/c;
8.  }
```

5. TSIC506F - der digitale Temperatur-Sensor

5.1. Unterschied zum NTC und Funktionsweise

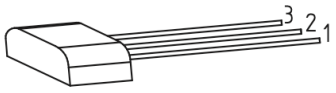
Die TSIC50X Serie ist eine Reihe von Temperatur-Sensoren, die digital operieren. Das bedeutet, dass dieser Sensor kein einfacher Widerstand wie der NTC ist, und deshalb auch kein Spannungsteiler nötig ist. Dazu übertragen die TSIC50X ihre Daten digital, weshalb am Arduino ein digitaler Pin benutzt werden muss.

Der TSIC506F wurde als Vergleich zum Innentemperatur-Sensor des STRATO3 eingebaut, so dass man auch hier einen Vergleichswert hat.

Für den TSIC506F haben wir uns entschieden, da dieser am genausten misst mit Abweichungen im positiven Temperaturbereich von bis zu +0,3°C und im negativen Bereich von bis zu -0,1°C. Außerdem ist seine Temperaturreichweite von -10°C bis +60°C ausreichend, um die Innentemperatur zu messen, da es innerhalb der Sonde nicht so kalt werden sollte. Dazu ist es sehr vorteilhaft, dass auch der TSIC506F mit einer 3,3V Spannung arbeitet.

5.2. Verbindung zum Arduino

Der TSIC506F besitzt drei Pins. Zwei sind für die Spannungsversorgung, nämlich V_{dd} (3) und GND (1). Die beiden Versorgungspins befinden sich an den Rändern des Gehäuses. In der Mitte zwischen den beiden Pins liegt der Daten-Pin (2), wo ein digitaler Eingang angeschlossen wird. (Abb. 2: TSIC506F Pinbelegung, stammt aus „TSIC50X Datasheet“)



5.3. Auslesen und Einbinden in den Arduino-Quellcode

Um einen digitalen Eingang mit dem Arduino auszulesen und korrekt interpretieren zu können, benötigt man ein Verständnis davon, in welcher Weise der Sensor seine Daten übermittelt und was das Empfangene konkret bedeutet. Der TSIC506F übermittelt seine Daten mit Hilfe eines 11-bit Serial Interfaces. Wir sind zum Glück aber nicht die Ersten,

die diesen Sensor benutzen wollen. Deshalb konnte ich im Internet eine geeignete Library[11] für den Arduino namens: „TSIC“[12] finden.

Der Quellcode im Arduino für den Sensor ist der folgende:

```
1.  #include <TSIC.h>
2.  TSIC thermo(52, 50, TSIC_50x);
3.  void setup(){
4.    uint16_t innerTemp;
5.    float innerTempC;
6.    String innerTempString;
7.  }
8.  void loop(){
9.    if(thermo.getTemperature(&innerTemp)){
10.     innerTempC = thermo.calc_Celsius(&innerTemp);
11.     innerTempString = String(innerTempC, 2);
12.   }
13. }
```

(In diesem Codestück wurden alle Inhalte entfernt, die nichts mit dem TSIC zu tun haben. Außerdem wird ein Verständnis des Objektorientierten Programmierens vorausgesetzt.)

Zu Beginn wird die Library inkludiert und ein Objekt der Klasse TSIC, die mit der Library kommt, erzeugt. Das Objekt heißt hier „thermo“ und an ihm wird zuerst in Zeile zwei der Konstruktor aufgerufen und die Parameter für den Data-Pin, dann V_{dd}-Pin und zum Schluss der Typ des Sensors übergeben. Da diese Library viele TSIC Sensoren interfacen kann, benötigt sie den Typ.

Innerhalb von „void setup()“ von Zeile drei bis sieben werden Variablen deklariert. Dabei dient „innerTemp“ als Art globale Variable der Library, die sich dann auch im Arduino-Code befindet. In „innerTempC“ soll dann die Temperatur in Grad Celsius gespeichert werden. Der „innerTempString“ ist für später.

Im sich wiederholenden Teil von Zeile acht bis dreizehn wird mit der if-Abfrage überprüft, ob der Sensor Daten hat, indem die Methode „getTemperature()“ aufgerufen wird. Diese gibt eine 1 zurück, wenn sie erfolgreich war, so dass nur weiter gerechnet wird, wenn es auch nötig und möglich ist. Dazu liest die Methode den gemessenen Temperaturwert des Sensors aus, allerdings in einem 11-bit Wert. Dazu wird

---

[11] Eine Library ist auch Quellcode, der sich in seinen Hauptcode einbinden lässt. Library’s bieten Methoden, um sie zu benutzen.

[12] Lässt sich im Arduino Library-Manager finden.

„getTemperature()“ die Speicheradresse durch das & der „innerTemp“ Variable übergeben, damit die Library dort ihren geschifteten Bit-Wert speichern kann. Die Library kann mit der Adresse auf die Variable mit dem „Difference Operator“[13] zugreifen ohne, dass sie ein Teil von ihr ist. Mit der „calc_Celsius()“ Methode wird der gelesene 11-bit Wert zu einer float-Zahl in Grad Celsius umgewandelt. Diese gibt sie als Rückgabewert zurück und wird dann in „innerTempC“ gespeichert. Somit wurde der TSIC ausgelesen und seine Daten in einem Float und in einem String, mit der „String()“ Methode umgewandelt, gespeichert. Die 2, die der „String()“ Methode beigefügt wird, dient dazu, die Nachkomma-Stellen auf zwei zu beschränken.

6. Der Neigungs- und Beschleunigungs-Sensor

6.1. Der ADIS16209

Dieser 8-bit Neigungs- und Beschleunigungs-Sensor ist ein absolutes Spitzenmodell und besitzt viele eingehende Quellen, um seine vielen Daten zu erfassen. Mit dem ADIS16209 kann man die Z-Achsen-Rotation, Neigung in X- und Y-Achse, Beschleunigung in X- und Y-Achse und die Temperatur messen.

Um mit dem Arduino zu kommunizieren, benutzt er den SPI-Bus, welcher hohe Transferraten ermöglicht. Außerdem läuft auch der ADIS16209 mit einer 3,3V Versorgungsspannung, was von hoher Wichtigkeit für den Betrieb des Sensors mit unserem Arduino ist.

6.2. Der SPI-Bus

SPI (Serial Peripheral Interface) ist eine weit verbreitete Methode unter Micro-Controllern, um sie mit ihren Hardwareerweiterungen oder sogar anderen Micro-Controllern in Verbindung zu setzen, so dass z.B. Daten ausgetauscht werden können.

Ein solcher SPI-Bus[14] besteht aus drei Haupt-Pins: MOSI, MISO und SCL. Diese sind die besonderen Pins, von denen die meisten Arduinos nur je einen haben:

---

[13] Der „Difference Operator“ kann mit eine Speicheradresse die Variable aufrufen, zu der diese Speicheradresse gehört.

[14] Busse sind Verbindungen, die mit nur einer Schnittstelle eine Kommunikation zu mehreren Hardwareerweiterungen ermöglichen.

- MOSI bedeutet Master-out-Slave-in. Mit Master ist der Arduino gemeint, der mit seinem Slave, dem Sensor, kommunizieren möchte. Über diesen Pin kann also der Master Daten an seinen Slave senden.
- MISO ist quasi dasselbe wie MOSI nur in die andere Richtung und bedeutet Master-in-Slave-out. Über diesen Pin erhält der Master Daten von seinem Slave.
- SCL steht für Serial-Clock. Dieser Pin gibt in einer bestimmten Frequenz Impulse aus. Mit dem Impuls synchron übermittelt der Sensor z.B. dem Master einen Bit pro Impuls. Somit weiß der Master, wann ein neues Bit des Sensors beginnt und wann es aufhört. Ohne diese Clock wäre eine Datenübertragung unmöglich und bei falscher Frequenz entstehen Fehler bei der Datenübertragung.

Dazu gibt es noch den CS-Pin, der jeder gewöhnliche digitale Pin des Arduino sein kann. Während die anderen drei Pins an alle SPI-Geräte gehen, besitzt jedes SPI-Gerät einen eigenen CS-Pin. Mit diesem Pin bestimmt der Arduino, von welchem seiner Sensoren er gerne Daten hätte oder ihm etwas sagen möchte. Dafür wird der CS-Pin des entsprechenden Gerätes auf HIGH[15] gestellt, damit es zuhört, und auf LOW[16], damit es weghört.

Damit die SPI-Kommunikation auch korrekt funktioniert, müssen drei Parameter zwischen Slave und Master übereinstimmen. Dies ist die Frequenz, mit der die Serial-Clock Impulse ausgibt, ob beim Übermitteln von Daten zuerst der „First-Signifikant-Bit“ oder „Least-Signifikant-Bit“ übertragen wird und der SPI-Mode.

6.3. ADIS16209- und SPI-Library für den Arduino

Zur leichteren Implementierung des Sensors gibt es die „ADIS16209“-Library von Juan Jose Chong, die er auf Git-Hub bereitstellt[17]. Diese Library benutzt auch die von der Arduino-Foundation bereitgestellte SPI-Library, um die SPI-Schnittstelle zu aktivieren und korrekt einzustellen.

Allerdings gab es ein Problem mit der ADIS16209-Library, nämlich, dass diese nur für 8-bit Arduinos geschrieben wurde. Dadurch funktionierte bei einem ersten Test der Sensor nicht mit unseren 32-bit Arduino Due und der ganze Micro-Controller ist

---

[15] Mit HIGH ist gemeint, dass der Pin 3,3V Spannung aus gibt.

[16] Mit LOW ist gemeint, dass der Pin 0,0V Spannung aus gibt.

[17] Link zur Library von ihm: https://github.com/juchong/ADIS16209-Arduino-Demo

abgestürzt. Deshalb machte ich mich auf die Suche nach einer Lösung für das Problem, welches ich in der von Herr Chong selber verwendeten SPI-Library fand. Sobald die ADIS16209-Library versuchte mit dem SPI-Objekt die SPI-Schnittstelle zu konfigurieren, stürzte der Arduino ab. Ich vermute, dass Herr Chong für seine Sensor-Library die SPI-Library schon kompiliert hatte, aber eben nur für 8-bit Arduino Boards.
Somit habe ich einfach eine neue ADIS16209-Library geschrieben, wobei ich weite Teile von Herr Chongs Library übernehmen konnte: die meisten Methoden und die Daten zur Initialisierung der SPI-Schnittstelle und des Sensors, sowie die Registeradressen, um vom Sensor Daten zu erfragen, konnten kopiert werden. Das Wichtigste war, dass die SPI-Library neu kompiliert wird. Dies geschah durch das Neu-schreiben der Library, da diese zuerst komplett durchkompiliert werden musste. Dazu wird die SPI-Library nun je nach Board kompiliert, so dass sie mit jedem Board funktioniert.
Im Großen und Ganzen bietet die Library für den Sensor alles, was man benötigt:

- „regWrite()“ übermittelt Daten zum Sensor. Sie benötigt als Parameter die Registeradresse und die zu transferierenden Daten.
- „regRead()“ fragt Daten beim Sensor an. Als Parameter wird die Registeradresse benötigt. Dazu gibt die Methode als Rückgabewert die empfangenen Daten.
- Die „-Scale()“ Methoden haben als Präfix immer die Datenherkunft, z.B. „tempScale()“. Mit diesen Methoden wird der empfangene 16-bit Wert des Sensors für je seine Datenherkunft interpretiert und als float ausgegeben in einer uns bekannten Einheit.
- Hand in Hand gehen der Konstruktor und die „begin()“ Methode. Beim Konstruktor werden weitere außerhalb der SPI-Schnittstelle benötigte Pins deklariert. Darunter fällt der CS-Pin, aber auch der RST-Pin und der DR-Pin, die zum Zurücksetzten/Neu-starten des Sensor bzw. zum Überprüfen, wann Daten bereit sind, gedacht sind. Mit der „begin()“ Methode wird der Sensor gestartet.

6.4. Implementierung in den Arduino-Quellcode

Das Einbinden erweist sich durch die Library als sehr einfach: Durch das Erstellen eines Objektes der Klasse kann auf alle Methoden zugegriffen werden, so dass nur ein paar nötige Voreinstellungen an den Sensor übermittelt werden müssen, denn den Rest übernimmt die Library. Z.B. muss das Sample-Intervall für den Neigungssensor

bestimmt werden, was durch das Schreiben in ein Kontrollregister erledigt wird. Zum Erfassen der Daten werden alle Datenregister nacheinander ausgelesen. Dabei ist es sehr wichtig, dass immer vor einer Auslesesession die korrekten SPI-Parameter eingestellt sind, denn es ist möglich, dass diese durch andere SPI-Geräte verändert wurden. Dies ist bei unserem Datenlogger der Fall, da wir ein weiters SPI-Gerät haben, welches im nächsten Kapitel „7. Das SD-Karten-Modul“ erläutert wird.

7. Das SD-Karten-Modul

7.1. Aufgabe und Funktionsweise des Moduls

Die SD-Karte ist der Speicher unseres Datenloggers, auf den wir alle Daten schreiben werden. Um auf eine SD-Karte zu schreiben, benötigt der Arduino ein SD-Karten-Modul. Wir haben uns für das Lese-/Schreib-Modell HW120 entschieden. Dieses Modul erlaubt dem Arduino Due mit 42 MHz Daten auf Dateien zu übertragen, was einer Geschwindigkeit von ca. 5,25 MB/s entspricht abzüglich der Datenverwaltung. Das ist mehr als genug. Auch dieses Modul läuft über den SPI-Bus und benutzt somit die SPI-Library und seine eigene SD-Library.

Die Library übernimmt jegliche Einstellungen, die man für den SPI-Bus benötigt. Um die Library zu benutzen, erstellt man aber kein SD-Objekt, denn dieses wird von der SD-Library bereit gestellt, sondern ein File-Objekt, auf das das SD-Objekt zugreift. Die wichtigsten Methoden sind:

- Mit „begin()“ wird an dem SD-Objekt alles gestartet, was größtenteils das Einstellen der SPI-Parameter übernimmt. Dazu wird der Methode die CS-Pin übergeben, damit die Library das Modul bei Bedarf ein- bzw. ausschalten kann.
- „open()“-Methode ist zum Öffnen eines File-Objekts gedacht. Ihr wird als Parameter der Name der Datei gegeben und ggf., ob von der Datei gelesen oder auf ihr geschrieben werden soll. Dazu muss ihr Rückgabewert in einem File-Objekt gespeichert werden. Sollte keine Datei mit diesem Namen existieren, wird eine neue Datei erstellt.
- Nach dem Öffnen muss mit „close()“ am File-Objekt die Datei wieder geschlossen werden, damit beim unerwarteten Trennen des Stroms keine Daten zerstört werden können.

- Mit „println()“ wird ein String in die nächste Zeile der Datei geschrieben. Als Parameter wird ein String benötigt.

7.2. Einbau in den Arduino-Quellcode

Die Implementierung in den Arduino-Quellcode wurde wie folgt umgesetzt. Dabei wurden alle Teile entfernt, die nichts mit der SD-Karte zu tun haben. Der Programmcode ist einzeln auf der nächsten Seite zu finden.

```
1.  #include <SPI.h>
2.  #include <SD.h>
3.
4.  int sdCs = 5;
5.  char fileNameBuffer[64];
6.  File logFile;
7.
8.  void setup(){
9.    SPI.begin();
10.   digitalWrite(sdCs, LOW);
11.   if(!SD.begin(sdCs)){
12.     digitalWrite(redLEDPin, HIGH);
13.     Serial.println("Error during setup! There is a SD-card
    issue!");
14.     while(true);
15.   }
16.   digitalWrite(sdCs, HIGH);
17.   digitalWrite(sdCs, LOW);
18.   fileName = makeLogName(moment);
19.   fileName.toCharArray(fileNameBuffer, fileName.length()
    +1);
20.   logFile = SD.open(fileNameBuffer);
21.   if(!
    logFile.readStringUntil('\r').equals("secondsSinceStart;ut
    c;temp;innerTemp;inclTemp;AX;AY;ROT;INCX;INCY;SUPPLY") ==
    false){
22.     logFile.close();
23.     logFile = SD.open(fileNameBuffer, FILE_WRITE);
24.     if(logFile){
25.
    logFile.println("secondsSinceStart;utc;temp;innerTemp;incl
    Temp;AX;AY;ROT;INCX;INCY;SUPPLY");
26.       logFile.close();
27.     }
28.     else{
29.     Serial.println("Error creating logFile!");
30.     digitalWrite(greenLEDPin, LOW);
31.     digitalWrite(redLEDPin, HIGH);
32.     while(true);
33.     }
34.   }
35.   digitalWrite(sdCs, HIGH);
36. }
37.
38. void loop(){
39.   digitalWrite(sdCs, LOW);
40.   Serial.print("Writing to LogFile now…");
41.   logFile = SD.open(fileNameBuffer, FILE_WRITE);
42.   logFile.println(makelogString(moment));
43.   logFile.close();
44.   Serial.println("done.");
45.   Serial.print("Written says: ");
46.   Serial.println(makelogString(moment));
47.   digitalWrite(sdCs, HIGH);
48. }
```

In diesem Stück Code werden alle Aufgaben des SD-Karten-Moduls durchgeführt. Dabei werden Methoden verwendet, die zur Vereinfachung des Codes dienen. Sie erledigen lediglich Aufgaben, bei denen ein String gebildet wird, der inhaltlich den Methodennamen beinhaltet. Im Folgenden wird der Code in Segmente eingeteilt und deren Funktion erläutert:

Zunächst werden die Libraries importiert und Variablen initialisiert. Diese sind zum Einen eine Variable für den CS-Pin, der für das SD-Karten-Modul zuständig ist, ein char-Array und ein File-Objekt. Die Größe 64 ist ohne besondere Bedeutung gewählt. Die File Library wurde durch die SD-Library mit importiert.

Während „void setup()“ wird die SPI Schnittstelle gestartet. Danach sollen die Einstellungen für das SD-Modul und des SD Objekts vorgenommen werden. Damit das SD-Karten-Modul zuhört, muss der CS-Pin auf LOW gezogen werden. Diese Einstellungen übernimmt die „begin()“ Methode, die innerhalb einer if-Abfrage in Zeile 11 aufgerufen wird. Mit der if-Abfrage kann man überprüfen, ob die Methode erfolgreich war, da diese eine 1 zurück gibt, wenn sie erfolgreich war. Wenn nicht, dann gibt sie eine 0 zurück, was sie wegen verschiedener Fehler liefert. Der schlechteste Fehler ist, dass das SD-Karten-Modul keine SD-Karte finden oder lesen konnte. Bei einem Fehler der „begin()“ Methode würde die if-Abfrage weiter machen und über den Seriellen Monitor den Fehler anzeigen bzw. durch die rote LED, die in Zeile 12 ein geschalten wird. Dazu wird eine while-Schleife betreten, die eine Endlosschleife ist, die das Programm somit an diesem Punkt anhält.

Wenn nun das Einrichten des Moduls und der Library erfolgreich war, muss eine neue Datei für die Log-Session erstellt werden. Die Datei braucht einen Namen, der mit der „makeLogName()“ Methode in dem String „fileName“ gespeichert wird. Um eine Datei so zu benennen, muss der String in ein char-Array umgewandelt werden, was in Zeile 19 mit „toCharArray()“ in „fileNameBuffer“ gespeichert wird. Mit dem SD-Objekt kann nun eine neue Datei geöffnet werden, die dann in „logFile“ gespeichert wird.

Von Zeile 21 bis 34 wird die neu erstellte Datei zu einer Log-Datei, indem man eine Headline schreibt, die die Reihenfolge der aufgezeichneten Daten angibt. Beim ersten Öffnen soll diese in die erste Zeile geschrieben werden. Allerdings ist es durchaus möglich, dass man eine Datei mehrmals an Tag von dem Arduino beschreiben lässt. Aber ein einfaches „println()“ würde bei jedem Start jedoch diese Headline eintragen, weshalb eine if-Abfrage in Zeile 21 überprüft, ob bereits eine Headline in der Datei ist.

Falls eine in der Datei ist, wird einfach keine mehr eingetragen. Im Großen und Ganzen wird beim Erstellen der Datei auch wieder mit einer if-Abfrage geprüft, ob das Erstellen der Log-Datei erfolgreich war, was in Zeile 24 überprüft wird. Nach dem erfolgreichen Einrichten des Moduls und der Datei wird der CS-Pin wieder auf HIGH gezogen, damit das Modul nicht mehr angesprochen wird.

Im Teil der „void loop()“ ist die Aufgabe des Moduls natürlich, den Daten-String zu speichern. Dies wird in den Zeilen 41 bis 43 erledigt, indem die Datei geöffnet, der String geschrieben und zur Sicherheit die Datei wieder geschlossen wird. Dazu werden auf dem Seriellen Monitor Informationen für zum Bespiel Debugging ausgegeben oder damit man sehen kann, was auf die Karte geschrieben wird. Auch wird wieder der CS-Pin auf LOW und HIGH gesetzt, um das SD-Karten-Modul zu aktivieren bzw. zu deaktivieren.

7.3. Probleme mit der MISO-Verbindung

Da das SD-Modul ja auch die SPI-Schnittstelle benutzt, ist es somit mit unserem ADIS16209 über die gemeinsamen MISO-, MOSI- und SCL-Leitungen verbunden. Somit gehen auch Signale von dem Arduino Due für das SD-Modul an den ADIS16209. Allerdings hört der ja nicht zu, da er durch die CS-Leitung für die Kommunikation deaktiviert wird. Jedoch gibt es bei dem SD-Modul HW120 in manchen Modellen einen technischen Defekt, bei dem das Modul trotz seiner kommunikativen Deaktivierung über die MISO-Leitung immer noch versucht, Daten an den Arduino Due zu übertragen. Dies stellt ein sehr großes Problem dar, weil somit auch die MOSI-Leitung des ADIS16209 geblockt wird, wenn dieser Daten übermitteln will. Damit ist der Neigungs- und Beschleunigungs-Sensor unbrauchbar, denn beim Arduino Due kommen dann nur zerstörte Daten an. Tatsächlich hat sogar dieser unkontrollierte Datenstrom den Arduino Due eingefroren, so dass dieser gar nicht mehr reagierte.

Um dieses Problem zu lösen, muss somit die Deaktivierung durch den CS-Pin des SD-Moduls quasi künstlich nachgeahmt werden. Dazu habe ich in dem Arduino-Forum einen Post gefunden, bei dem jemand dasselbe Problem schilderte, aber eine Lösung hatte: ein Gate. Ein Gate ist ein elektronisches Bauteil, das, wenn eine Spannung an einem Aktivierungs-Pin anliegt, Daten weiter gibt und, wenn keine Spannung anliegt, keine Daten weiter gibt. Somit liegt als Lösung auf der Hand, dass man das Gate die MISO-Leitung des SD-Moduls kontrollieren lässt, indem man mit dem CS-Pin auch das

Gate noch kontrolliert. Allerdings ist ein invertiertes Gate zu benutzen, das Daten bei keiner Spannung durchlässt, da SPI-Geräte zuhören bzw. senden sollen, wenn ihr CS-Pin auf LOW ist.

Solch ein Gate ist der IC 74HC 125, welches genau so invertiert ist. Es wurde für den Datenlogger verwendet und ist zwischen der MOSI-Leitung des SD-Moduls und der gemeinsamen Leitung von Arduino Due und ADIS16209 geschaltet, sodass es keine Interferenzen mehr gibt. Dazu geht der CS-Pin auch an das Gate, um es zu aktivieren.

8. RTC - eine Uhr an Bord

8.1. Nutzen einer Uhr

Eine Uhr ist mit auf unserem Flug, um eine Möglichkeit zu haben, die Zeit mit in unseren Log-String zu schreiben. Dies dient dazu, dass man den aufgenommenen Daten auch einen Zeitpunkt zuweisen kann, an dem sie aufgenommen wurden. Somit kann man die Daten des Arduino Due Datenloggers dann mit anderen einfach vergleichen, indem man Werte der gleichen Uhrzeit gegenüber stellt.

Auch der Arduino besitzt eine Art Uhr, die jedoch nur die Millisekunden seit dem Start aufnimmt und somit keine einfach zu vergleichende Zeitangabe hat.

Mit der Uhr kann der Arduino Due immer nach der Zeit fragen, wenn er gerade einen Datensatz auf unsere SD-Karte schreiben möchte. Solche Module werden RTCs genannt, was Real-Time-Clock also Echtzeit-Uhr bedeutet. Diese Module besitzen einen Chip, der auch wie der Arduino Due seit seinem Start die Zeit zählt, allerdings addiert er diese dann auf eine Uhrzeit. Da eine RTC eine Hardware-Erweiterung ist, wird sie vom Arduino mit Strom versorgt und geht aus, wenn der Arduino aus geht. Dadurch würde die Uhr aufhören zu laufen und somit die korrekte Uhrzeit verloren gehen. Deshalb besitzen RTCs eine kleine Batterie, die für ein Weiterlaufen der Uhr sorgen und somit die korrekte Uhrzeit behalten. Für den Datenlogger habe ich die RTC DS1307 gewählt (Abb. 3: DS1307 Unterseite, stammt von https://www.amazon.de/DS1307-Module-Battery-Raspberry-Powered/dp/B01FR7AE1U):

Dieses Modul kann nicht nur die Uhrzeit mit Stunde, Minute und Sekunde aufzeichnen, sondern auch das Datum mit Jahr, Monat, Tag und Wochentag. Deshalb eignet sich diese RTC auch sehr gut, um unserer Log-Datei einen Namen zu vergeben, da diese dann einfach nach dem Datum benannt werden kann, an dem sie erstellt wurden und es dann für jeden Tag eine Log-Datei gibt.

8.2. Kommunikation mit der Uhr

Die RTC DS1307 kommuniziert entweder über eine Serielle Schnittstelle oder über das I²C-Bussystem. Für eine Verbindung mit dem I²C-Bus steht bereits eine Library für den Arduino zur Verfügung, die RTClib[18] heißt. Sie bietet einfache Methoden, um die Uhr benutzen zu können und bringt DateTime Objekte mit, in denen alle zeitlichen Angaben der Uhr gespeichert werden können. Wichtige Methoden der Library sind die folgenden:

- „adjust()“ wird an dem RTC-Objekt ausgeführt, welches vom Typ der passenden Uhr ist, und sorgt für eine Übergabe der Startzeit der Uhr.
- „now()“ gibt alle zeitlichen Angaben der RTC als DateTime Objekt zurück, das gespeichert werden kann.
- An den DateTime-Objekten kann man jede einzelne Zeit-Angabe herausfinden, indem man an ihnen die Methode ausübt, die den Namen der Zeit-Angabe trägt. Zum Beispiel mit „seconds()“ erhält man einen Integer, der die Sekunden angibt.

Der I²C-Bus ist ähnlich wie das SPI-Bussystem, jedoch benötigt es weniger Kabel. Es werden nur die Verbindungen SCL und SDA benötigt. Dabei ist SCL das selbe wie bei dem SPI-Bus. Die SDA-Verbindung übernimmt Datentransfer wie MOSI und MISO, sowie das auswählen des korrekten Sensors über eine Adresse, was bei dem SPI-Bus die individuelle CS-Verbindung macht. Wegen des I²C-Bus ist auch die Wire-Library nötig, um die Schnittstelle benutzen zu können. Jegliche Kommunikation über die I²C-Bus übernimmt aber die RTClib.

8.3. Gebrauch der Uhr im Arduino-Quellcode

Nach Einbinden der Libraries muss ein RTC Objekt sowie ein DateTime Objekt erstellt werden. Danach kann durch die „now()“ Methode das DateTime Objekt initialisiert und somit jegliche zeitliche Angaben gespeichert werden. Mit diesen kann dann weiter

[18] Diese Library kann im Arduino Bibliotheken-Manager gefunden werden.

gearbeitet werden. An dem DateTime Objekt, das im Code „moment“ heißt, können somit die entsprechenden Methoden angewandt werden und mit Hilfe der „String()“ Methode von Integern in Strings umgewandelt werden, so dass zum Beispiel die SD-Karte die Zeit schreiben kann. So funktioniert das Ganze im Quellcode, der für den Flug war.

Ein weiterer wichtiger Teil des Programms ist es, die Uhrzeit korrekt auf der Uhr einzustellen. Dazu wird ja ein festes DateTime Objekt benutzt. Dieses kann vorher eingegeben werden oder durch eine spezielle Schreibweise kann die Computerzeit zum Zeitpunkt des Hochladens auf den Arduino verwendet werden. Allerdings muss die Zeit abgezogen werden, die zum Hochladen benötigt wird, da die Uhrzeit des letzten Speicherns der Datei verwendet wird, was kurz vor dem Hochladen gemacht wird. Das Hochladen dauert etwas 15 Sekunden, so dass die RTC 15 Sekunden nach ging. Dazu sollte die Uhrzeit in Universal Time angeben sein, da der SRTATO3 seine Werte auch so angibt. Somit muss von unserer mitteleuropäischen Zeit 59 Minuten und 45 Sekunden abgezogen werden, damit die Uhr korrekt läuft. Dies ist möglich mit „TimeSpan()“, was eine Zeit angibt, die man einem DateTime Objekt zuweisen kann.

Alle soeben genannten Vorgänge sollen in diesem kleinen Beispiel Code einmal gezeigt werden, in dem jede Minute die Minute der Uhrzeit über den Seriellen Monitor ausgegeben wird:

```
1.  #include <Wire.h>
2.  #include <RTClib.h>
3.
4.  RTC_DS1307 rtc;
5.  DateTime moment;
6.  void setup() {
7.    Serial.begin(9600);
8.    rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
9.    rtc.begin();
10. }
11.
12. void loop() {
13.   moment = rtc.now();
14.   moment = rtc.now() - TimeSpan(0, 0, 59, 45);
15.   String minute = String(moment.minute());
16.   Serial.println(minute);
17.   delay(60000);
18. }
```

9. Bildung des Log-Strings

9.1. Überblick über den Log-String

Der Log-String ist eine Datenkette, die Daten in einer Zeile der SD-Karte angibt. Ihre Daten sind durch Semikolons getrennt und sind somit durch ein Programm wie Excel oder andere Auswertungsprogramme gut einlesbar. Durch die Headline, die am Anfang der Datei geschrieben wird, weiß das Auswertungsprogramm, an welcher Stelle die Daten für eine Kategorie sind. Hier ist ein Beispiel-Log-String:

```
12471;14:16:48;-4.27;14.80;10.90;-0.16;-0.03;-9.70;-9.10;-1.52;3.28
```

Am Anfang befinden sich die Sekunden nach dem Start, die nach dem Semikolon von der Universal Time gefolgt sind. Nach den beiden Zeitangaben folgen jegliche Daten, die statt eines Kommas einen Punkt haben, um die Dezimalstellen beginnen zu lassen. Dies ist dem geschuldet, dass man auf Englisch programmiert und somit auch float-Variablen ihre Dezimalstellen, wie es im Englischen üblich ist, mit Punkten abtrennt. Ein Problem für die Auswertung stellt dies nicht dar, denn Excel zum Beispiel fragt beim Auswerten, wie die Dezimalstellen getrennt sind.

9.2. Zusammenbau der Daten zum Log-String

In diesem Kapitel wird einmal zusammengefasst, wie ein Log-String erstellt wird. Dies kann nun leicht erläutert werden, da alle Hardwareerweiterungen und Datenerfassungen in den vorherigen Kapiteln erklärt wurden. Für die Übersichtlichkeit des Codes habe ich das Bilden des Log-Strings in mehrere Methoden aufgeteilt. Dabei führt eine einzelne Methode alle anderen an den korrekten Stellen aus.

Mit der „makeLogString()“ Methode beginnt das Ganze. Sie gibt nach dem Aufrufen einen fertigen Log-String aus, der natürlich vom Datentyp String ist. Diese Methode wird immer am Schluss des Aufnahmeintervalls aufgenommen, damit alle Daten vorhanden sind. Zum einfachen Aufschreiben des Strings wird die Methode innerhalb der „println()“ des File Objekts ausgeführt, was in dem Kapitel „7.2. Einbau in den Arduino-Quellcode“ der SD-Karte beschrieben wird. Die Methode benötigt außerdem als Parameter ein DateTime Objekt, damit in den LogString die korrekte Uhrzeit

geschrieben werden kann. Dieses DateTime Objekt wurde durch „now()“ aktualisiert und ist das globale Objekt „moment“.

Hier ist die „makeLogString()“ Methode aus dem Quellcode. Die anderen Methoden werden nur qualitativ erläutert und sind im digitalen Anhang im Programm zu finden:

```
1.  String makelogString(DateTime moment){
2.    String logString;
3.    logString.concat(millis()/1000);
4.    logString.concat(";");
5.    logString.concat(makeTime(moment));
6.    logString.concat(";");
7.    logString.concat(tempString);
8.    logString.concat(";");
9.    logString.concat(innerTempString);
10.   logString.concat(";");
11.   logString.concat(makeInclString());
12.   return logString;
13. }
```

Zunächst einmal wird ein String deklariert, der dann später zurückgegeben wird, hier „logString“. Mit Hilfe der „concat()“ Methode kann an einen String ein weiterer String angehangen werden. Die Methode wird an einem String ausgeführt, an dem dann der übergebene String einfach angehangen wird. Somit werden alle Segmente des Log-Strings in dieser Methode an den String gehangen.

Zuerst werden die Sekunden nach dem Start angehangen (Zeile 3). Mit „millis()“ wird die Zeit in Millisekunden nach dem Start des Arduino Due ausgegeben. Durch das Teilen durch 1000 werden aus den Millisekunden Sekunden. Dieser berechnete Integer kann auch angehangen werden. Da nun eine der Angaben fertig geschrieben wurde, wird in Zeile 4 ein Semikolon zum Abtrennen eingefügt.

Darauf folgt die Universal Time, die durch die „makeTime()“ Methode als String zurück gegeben wird. Sie benötigt als Parameter das DateTime Objekt, um an ihr die Stunde, Minute und Sekunde abzurufen. Diese werden wieder alle an einen String angehangen, der mit Doppelpunkten zwischen die Zeitangaben ausgestattet wird. Nachdem die Zeit fertig ist, kann sie auch an den Log-String angehangen werden.

In den Zeilen 7 und 9 werden globale Strings an den Log-String angehangen, die zuvor durch die Auslese der Temperatur-Sensoren mit Werten initialisiert wurden, so dass diese einfach angehangen werden können.

Zum Schluss wird noch mit der „makeInclString()“ Methode ein String gebildet, der aus allen Daten des Neigungs- und Beschleunigungs-Sensor besteht, mit Semikolons getrennt. Alle Daten wurden zuvor aufgenommen und durch die „String()“ Methode zu Strings umgewandelt.
Somit ist der Log-String fertig und kann mit dem return-Statement ausgegeben und auf die SD-Karte und den Seriellen Monitor geschrieben werden.

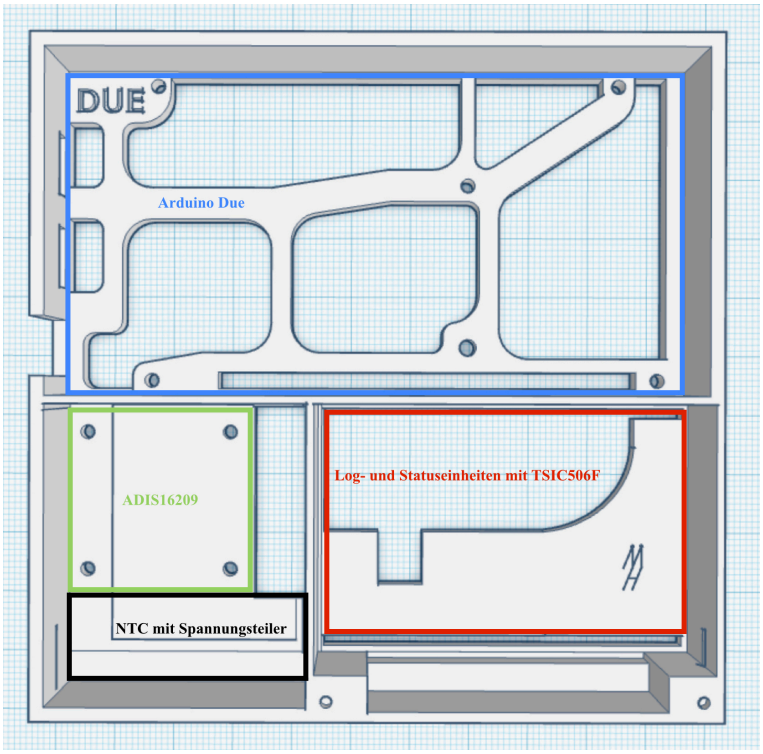
10. Übersicht über die Hardware

10.1. Entwurf des Gehäuses

Ein Gehäuse dient dazu, alle Komponenten als eine Einheit, also den Datenlogger zu fassen. Der Arduino Due benötigt feste Verbindungen über Kabel, die direkt an seine Pins gelötet werden sollten. Für eine Struktur und einen festen Aufbau war ein Gehäuse nötig, das viel Platz für Kabel und Sensoren bietet.
Wegen des Besitzes eines 3D-Druckers habe ich die Möglichkeit ein Gehäuse ganz nach unseren Anforderungen zu erstellen. Angefangen habe ich mit einer 3D-Vorlage eines Gehäuses[19] für den Arduino Due aus dem Internet. Dieses hat die Öffnungen für alle Pins berücksichtigt und bietet sogar Löcher, um den Arduino Due fest zu schrauben.
Das restliche Gehäuse ist der Größe des Arduino Due angepasst, so dass nur vor seinem Gehäuse neue Teile für alle Sensoren angebracht sind.
Es gibt zwei weitere Räume für Sensoren: einmal für die Platine, auf der sich die Log- und Statuseinheiten, welche die RTC, das SD-Modul und die Status-LEDs sind, und dazu noch der TSIC506F befinden und dann noch den Raum für den ADIS16209 und den NTC mit seinem Spannungsteiler:

[19] Das Gehäuse ist unter folgenden Link oder im digitalen Anhang unter 3D-Modelle zu finden: https://www.thingiverse.com/thing:2794329

(Abb. 4: Datenlogger Gehäuse von oben, Screenshot aus Tinker-CAD)

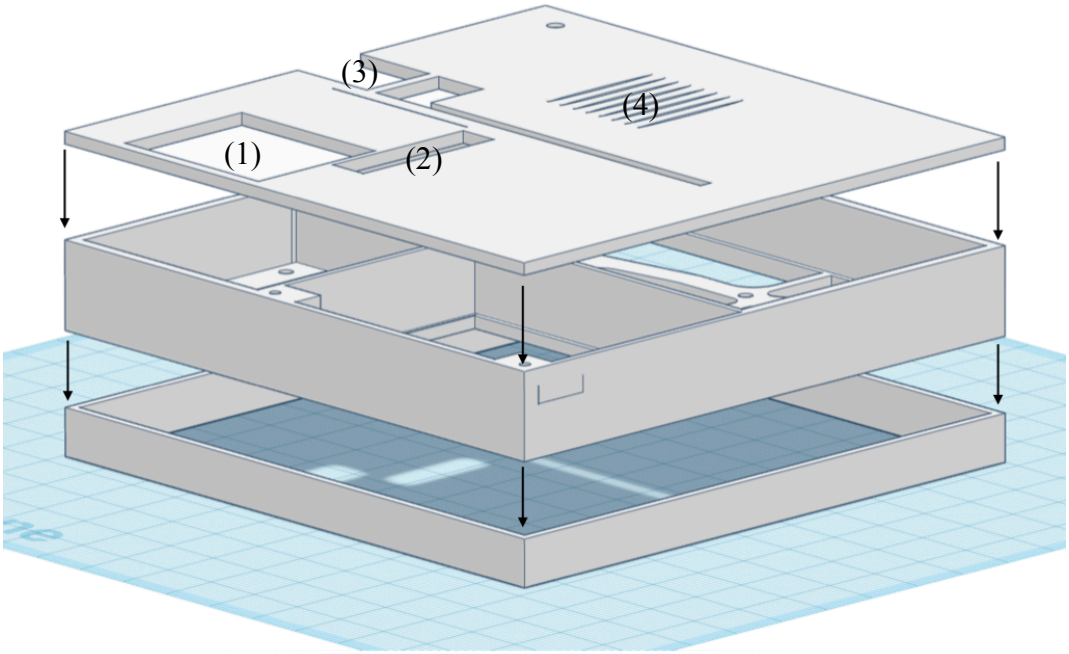


Das abgebildete Gehäuse kann also alle Komponente beherbergen und bietet gute Möglichkeiten, um von unten Kabel zu legen, die dann durch die Löcher im Boden an den Sensor bzw. den Arduino Due gelangen. Auf dem Bild ist das Gehäuse von oben zu sehen.

Von unten verlötet, würde das Gehäuse auf seinen Kabeln liegen und somit keinen festen und geraden Halt in der Sonde finden können. Dabei werden die Kabel gequetscht und Vibrationen und Stöße würden auf sie übertragen werden, so dass das ganze Gehäuse von einem Rahmen von unten erhöht werden muss, damit alle Kabel frei schweben können und die Standhaftigkeit des Datenloggers gewährleistet ist. Dieser Rahmen wird einfach angeklebt.

Auch auf der Oberseite des Hauptkörpers muss etwas hinzugefügt werden, nämlich ein Deckel, der die sonst frei liegenden Sensoren vor umherfliegenden Teilen oder einem eignen Versagen der Befestigung und somit vor Stürzen schützt. Der Deckel wird mit Tape angeklebt, um ihn nach dem Flug wieder lösen zu können:

(Abb. 5: 3D-Gehäuse, Screenshot aus Tinker-CAD)



Der Deckel des Gehäuses bietet auch Löcher, die aber natürlich nicht für Kabel sind, sondern für Erreichbarkeiten:

1. Hier kann der NTC von seinem Spannungsteiler aus zum Ausgang aus der Sonde.
2. An dieser Stelle kann man die Status LEDs ablesen.
3. Die Aussparung verhindert die Wärmebildung an der Stromversorgung.
4. Ebenso sind hier Luftspalte für eine kühlere CPU.

Die Aussparungen für die Stromversorgung und CPU scheinen unter normalen Umständen als unnötig, sind aber unter diesen Umständen in der gut isolierten Sonde sehr wohl angebracht. Andere elektrische Geräte, insbesondere die Kameras, produzieren viel Hitze, so dass möglichst viele Dämmungen von Hitzequellen verhindert werden müssen. Aus den Daten von unserem Flug ist zu erkennen, dass die maximale Innentemperatur des Datenloggers (gemessen durch den TSIC506F) bei 42,4°C lag. Das ist schon ganz schön hoch für einen Mikrocontroller.

Nach Abschluss des Designs musste das Gehäuse nur noch gedruckt werden. Es ist aus generic PLA, ein Kunststoff, der für viele Anwendungszwecke geeignet, robust und nicht brüchig ist. Es ist auch relativ leicht, so dass das gesamte fertige Gehäuse[20] in die Sonde passende Maße von 10,65 cm x 10,6 cm x 2,49 cm hat und dabei nur 56g wiegt.

[20] Das 3D-Modell kann im digitalen Anhang unter 3D-Modelle oder unter folgenden Link mit Tinker-CAD Account gefunden werden: https://www.tinkercad.com/things/6iXEeDlUae7-arduino-case-final-version/edit

10.2. Schaltungen der Komponenten

Wie schon im Kapitel zuvor erläutert, besteht der Datenlogger also aus drei Kammern. Es werden nun die Schaltungen innerhalb der Kammern und zwischen ihnen erläutert.

Zunächst einmal bilden die Log- und Statuseinheiten mit dem TSIC506F eine kompakte Platine. Somit sind an dieser Platine die folgenden Pins zu finden:

- SPI-Pins für das SD-Modul, die bis auf die MISO Leitung alle direkt an den Arduino Due gelötet werden können. Zwischen den beiden MISO Anschlüssen befindet sich noch das Gate.
- SDA und SDL sind die Pins für den I²C-Bus der RTC und können auch einfach verbunden werden.
- Pin 52 ist der digitale Pin, der die Daten des TSIC506F empfängt und kann somit auch einfach angelötet werden. Ebenso wie Pin 50 direkt daneben, der für die Stromversorgung des Sensors verantwortlich ist.
- Pin 6 und 7 sind die Versorgungspins der Status-LEDs. Da LEDs mit weniger als 3,3V operieren, die sie von diesen digitalen Output Pins erhalten würden, ist für je eine LED ein Widerstand von 330Ω nötig.
- Die Stromversorgung der RTC und des SD-Moduls ist getrennt, denn die RTC benötigt eine Spannung von 3,3V und das SD-Modul eine Spannung von 5V. Dafür gibt es einen Ground, der alle Bauteile verbindet, und somit nur einer benötigt wird.

Die Komponenten sind auf der Platine alle mit ihren Pins eingelötet, die sie dann auch auf der Platine halten. Für den Ground gibt es an der Unterseite einen langen Draht, der sich an der Seite erstreckt und somit alle Bauteile einfach daran angelötet werden können. Jegliche Kabel treffen sich etwa an der Mitte der Platine zu einer Leiste an Steckverbindungen, an deren Unterseite auch die Kabel für die Lötverbindung sind, die durch ein Loch im Boden zum Arduino gelangen. Mit den Steckern konnte eine gesteckte Testversion[21] des Datenloggers getestet werden. Die Platine wird mit doppelseitigem Tape in ihrer Kammer gehalten. Ein Bild der Platine ist im Anhang zu finden unter Bild 1.

Links von dieser Platine liegt eine Kammer mit dem ADIS16209 und des NTC mit Spannungsteiler. Die beiden Bauteile verbindet nur ein Ground und eine 3,3V Spannungsversorgung, aber sonst haben die Bauteile nichts miteinander zu tun.

---

[21] Ein Bild dieser Steckversion ist im Anhang zu finden unter Bild 3.

- Die SPI-Leitungen werden direkt mit dem Arduino Due verlötet. Allerdings sind alle Kabel am ADIS16209 an Steckern angelötet, die an seine Header gesteckt sind.
- Dazu liegen noch die RST und DR Verbindungen vor, die digitale Signale weiter geben und auch einfach an ihre Pins 12 bzw. 13 angelötet werden können.
- Der NTC ist als einer der Widerstände in seinen Spannungsteiler eingelötet. Zwischen dem NTC und seinem Vorwiderstand liegt ein Analoger Input an Pin A4, der auch direkt verbunden wird.

Beide Bauteile sind passend in die Kammer eingeschraubt, so dass insbesondere der Neigungs- und Beschleunigungs-Sensor fest an dem Datenlogger Gehäuse bleibt. Die Kabel verlaufen durch ein quadratisches Loch im Boden der Kammer zum Arduino Due. Auch hier ist wieder ein Bild im Anhang zu finden unter Bild 2.

In der letzten Kammer über den beiden anderen befindet sich der Arduino Due, der mit drei Schrauben befestigt ist. Von unten kann alles angelötet werden. An der Unterseite befindet sich dazu noch in der Nähe des SPI-Bus das Gate, welches mit Tape angeklebt ist.

Zusammengefasst kann man die Schaltung als sehr gradlinig bezeichnen, da die meisten Komponenten einfach miteinander verbunden werden können.

11. Ein guter Datenlogger?

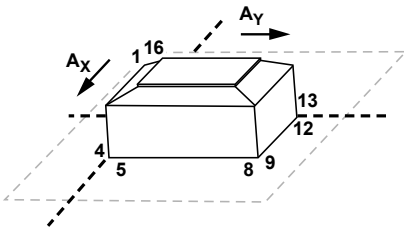
11.1. Gute Design-Entscheidungen

Zunächst einmal ist natürlich gut, dass alles funktioniert. Dies ist schon mal ein großer Schritt in die richtige Richtung gewesen. Dinge, die man besonders hervorheben kann, sind die folgenden:

1. Das Gehäuse ist für den Inhalt, den es trägt, sehr kompakt und passt sehr gut in die Sonde.
2. Dank des Arduino Due's gibt es keine Probleme bei zu schwierigen Aufgaben und die Anzahl der Pins bietet viele Möglichkeiten. Dazu konnte das Potenzial des Temperatur-Sensors ausgeschöpft werden dank der 12-bit Auslese.
3. Auch gut für die Sicherheit ist die Befestigung mit Schrauben und das Löten der Kabel, was das Risiko für zum Beispiel das Ausfallen eines Sensors stark verringert.
4. Die Spannungsversorgungen passen alle gut zusammen, da alle Geräte intern mit 3,3V Spannung laufen und es somit keine Interferenzen gibt.
5. Gut ist natürlich auch, dass alle Bauteile bis auf den Arduino Due in ihrem Potenzial ausgeschöpft werden bzw. alle möglichen Daten auch ausgelesen und aufgezeichnet werden.

11.2. Fehlende Z-Beschleunigung

Mit dem ADIS16209 können Beschleunigungen in zwei zueinander im 90° Winkel stehenden Achsen gemessen werden, die sich beide in einer 2-dimensionalen Fläche befinden. Als Koordinatensystem betrachtet sind dies die X- und Y-Achse (Abb. 6: Basic Accelerometer Operation, stammt aus „ADIS16209_REVE"):



Allerdings ist es für Messungen sehr interessant auch die Beschleunigung in der Höhe messen zu können, also in der Z-Achse. Dies ist mit dem Aufbau des Sensor so nicht möglich, sondern nur mit einer 90° Neigung, um zum Beispiel die X-Achse nach oben zeigen zu lassen. Allerdings würde das die anderen Messungen unbrauchbar machen.

Somit wäre eine Idee noch einen Beschleunigung-Sensor für nur eine Richtung in die Höhe einzubauen.

11.3. Weitere Verbesserungen: Aufnahmerate, Kalibrierung und STRATO3

Um den Datenlogger weiter zu verbessern, sind genauere Daten nötig. Diese lassen sich zum Einen durch eine höhere Aufnahmerate erreichen. Mit der höheren Aufnahmerate sollen insbesondere die Daten des Neigungs- und Beschleunigungs-Sensor besser gemacht werden. Mit dem jetzigen Aufnahmeintervall von zwei Sekunden lassen sich besonders bei der Rotation nur sehr unzuverlässige Daten generieren, da wir beim Start erkannt haben, dass sich die Sonde deutlich mehr dreht als erwartet und somit in einem Aufnahmeintervall es möglich ist, dass manche Bewegungen gar nicht aufgezeichnet werden. Bei so schnellen Drehungen sind höhere Aufnahmeraten gefragt, die sich in einem Bereich von 1 bis 10 mal pro Sekunde aufhalten sollten.

Mit 10 pro Sekunde würden sich innerhalb einer Stunde 36.000 Log-Strings bilden. Ein Log-String ist etwa 34 Byte groß, was sich aus einer vorhandene Log-Datei berechnen lässt, mit der gesamten Größe durch Anzahl der Strings. Nach einer Stunde wäre somit die Datei 1,22 MB groß. Jedoch hält uns wenig davon ab, Aufnahmeraten von bis 10 Hz zu erreichen, denn zum Beispiel dauert das Auslesen des ADIS16209 nur 500 Mikrosekunden. Das Auslesen des NTC ist mit der „analogRead()“ Methode alle 100 Mikrosekunden. Und bei 340 Byte pro Sekunden kommt die SD-Karte auch noch gut mit. Allgemein ist überall eine Aufnahmerate von bis zu 10 Hz möglich, wobei zum Beispiel eine Aufnahmerate von 4 Hz auch ausreichen würde. Denn mit 4 Hz würden vier Messungen pro Sekunde statt finden, was 0,25 Sekunden Intervalle sind. Für Messungen der Rotation, welche die kleinsten Zeitintervalle von allen Sensor-Daten benötigt zum Messen, wäre dies ausreichend, denn es ist unwahrscheinlich, dass sich die Sonde innerhalb einer Viertelsekunde mehr als 360° dreht. Ob das ganze funktioniert, muss natürlich erstmal getestet werden.

Des Weiteren kann man noch generell die Daten aller Sensoren durch eine bessere Kalibrierung verbessern und korrekte Tests anstellen mit Vergleichsgeräten, um zum Beispiel die Formel für den NTC weiter an zu passen. Dazu muss die zeitliche Abfrage des TSIC506F kalibriert werden. Denn ich vermute diese ist nicht korrekt eingestellt und fragt Daten ab, obwohl noch gar keine Daten bereit sind und somit werden einfach maximalwerte über 1000 °C ausgegeben.

Des Weiteren gäbe es vielleicht die Möglichkeit, sich die Uhrzeit über den STRATO3 zu besorgen, denn dieser hat einen Seriellen Ausgang, über den sich eine Kommunikation realisieren lassen könnte. So wären Arduino Due und STRATO3 immer synchron.

12. Abschluss

Alles in allem kann man sagen, dass der Datenlogger in seinem jetzigen Zustand eine gute Arbeit geleistet hat. Durch ihn konnten viele Daten auf unserem Flug aufgezeichnet werden, die zu interessanten Erkenntnissen führten.
Diese Arbeit hat die Dokumentation des Datenloggers übernommen und bietet somit nun viele Informationen für nachfolgende Gruppen, die im Rahmen des Projektkurses Physik einen Wetterballon aufsteigen lassen werden.

Vielen Dank fürs Lesen!

Ich erkläre, dass ich die Projektarbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Leverkusen, 2. Juni 2021 Maximilian Hüter

Quellen- und Literaturverzeichnis

Internet-Quellen und Internet-Fußnoten in der Reihenfolge, wie sie im Text benutzt wurden:

1. Arduino Foundation: https://www.arduino.cc/ abgerufen 30.03.2021; angegeben Fußnote 3, verwendet in Kapitel 3.1.
2. Microsoft: https://code.visualstudio.com abgerufen 30.03.2021; angegeben Fußnote 6, verwendet in Kapitel 3.1.
3. Wikipedia, „Steinhart-Hart-Gleichung“: https://de.wikipedia.org/wiki/Steinhart-Hart-Gleichung zuletzt bearbeitet 11.03.2021; angegeben Fußnote 10, verwendet in Kapitel 4.3.
4. Juan Chong auf Git-Hub, „ADIS16209-Arduino-Demo“: https://github.com/juchong/ADIS16209-Arduino-Demo 26.10.2015; angegeben Fußnote 17, verwendet in Kapitel 6.3.
5. Arduino Forum, „anyone run SDfat library for long period at spi_full_speed without corruption?“: https://forum.arduino.cc/t/anyone-run-sdfat-library-for-long-period-at-spi_full_speed-without-corruption/168770 Juni 2013, verwendet in Kapitel 7.1. zum Berechnen der Datenraten des SD-Moduls
6. ElectricalEngineering, „SPI communication - bits per second vs hz“: https://electronics.stackexchange.com/questions/122500/spi-communication-bits-per-second-vs-hz 23.07.2014, verwendet in Kapitel 7.1. zum Berechnen der Datenraten des SD-Moduls
7. DIYhans auf Thingivers, „Arduino DUE case“: https://www.thingiverse.com/thing:2794329 14.02.2018; angegeben Fußnote 19, verwendet in Kapitel 10.1.
8. maximilian.hueter (ich) auf Tinker-CAD, „Arduino Case Final-Version“: https://www.tinkercad.com/things/6iXEeDlUae7-arduino-case-final-version/edit 22.02.2021; angegeben Fußnote 20, verwendet in Kapitel 10.1.
9. Willem Maes, „Howto make an Arduino fast enough to…“: http://www.optiloading.be/willem/Arduino/speeding.pdf 01.05.2018, verwendet in Kapitel 11.3. zum Berechnen der Auslesedauern

Auflistung der Literaturquellen, die im digitalen Anhang zu finden sind:

- Analog Devices: „High Accuracy, Dual-Axis Digital Inclinometer and Accelerometer ADIS16209“ USA, 2008-2015, Dateiname: „ADIS16209_REVE.pdf“
- Innovative Sensor Technologie: „TSic™ 506F/503F/501F Temperature Sensor IC“ Schweiz, o. J., Dateiname: „TSIC50X Datasheet“
- SGS-Thomson Microelectronics: „Quad Bus Buffers (3-State)“ o. O., September 1993, Seite 1/11, Dateiname: „M54/74HC125/74HC126“

Internet-Quellen als QR-Code für Papierversionen:

Arduino Foundation	
Visual Studio Code	
Steinhart-Hart-Gleichung	
Juan Chong auf Git-Hub	
Arduino Forum	
ElectricalEngeneering	
DIYhans auf Thingivers	
maximilian.hueter auf Tinker-CAD	
Willem Maes „Howto make an Arduino fast enough to…“	

Anhang

Bild 1:

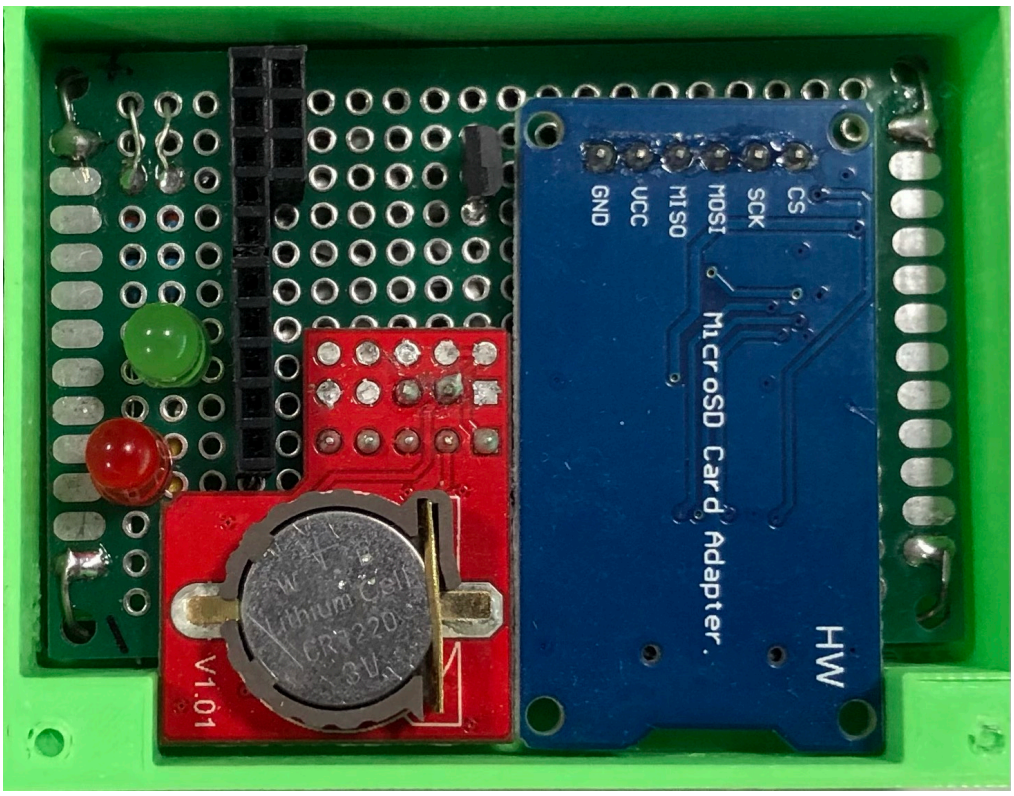


Bild 2:

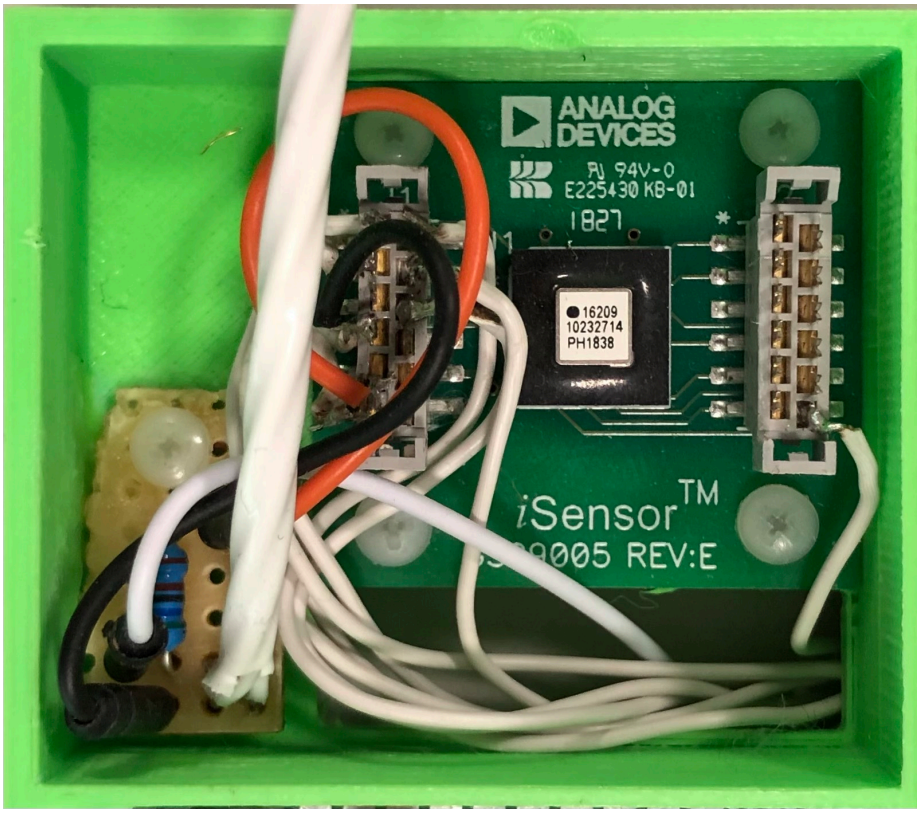


Bild 3: