Hard- und Software für die Messung und Auswertung des (Raum-)Klimas

# 1. Vorwort

# 2. Wahl der Komponenten / Programmiersprachen

2.1. Wahl der Hardwarebasis

Für den Bau eines Gerätes, das Daten verarbeiten / aufzeichnen soll, gibt es eine Vielzahl von Möglichen Plattformen, auf denen ein Projekt realisiert werden kann. Zum einen gibt es diverse Mikrocontroller, zum anderen (Linux) Einplatinencomputer.

Für dieses Projekt muss die Möglichkeit gegeben sein, relativ viele Sensoren anzuschließen und die Sensordaten aufzuzeichnen, Auswertung oder größere Berechnungen am Gerät sind nicht notwendig.

Die Entscheidung welche Plattform die geeignetere ist fiel darauf, einen Mikrocontroller als Herzstück des Gerätes zu verwenden, da ein solcher zwar weniger Rechenleistung bietet als ein Einplatinencomputer, aber dafür meistens mehr nutzbare I/O Pins besitzt. So hat z.B. der Raspberry PI 40 Pins, von denen prinzipiell 26 als GPIO (General Purpose Input / Output) verwendet werden können, jedoch können z.B. Pins GPIO 14 und GPIO 15 nicht wirklich beliebig verwendet werden, da der Raspberry PI eine Konsole über diese beiden Pins ausgibt. Dies Kann natürlich Deaktiviert werden, ist aber nur eine der Schwierigkeiten, die bei Einplatinencomputern auftreten können. Außerdem ist die Ansteuerung der Pins und die Verfügbarkeit von Libraries dürftig. Mikrocontroller wie der Arduino MEGA 2560 haben 70 IO Pins, von denen 16 analoge Signale messen können (mit einem Integrierten ADC (=Analog-Digital-Converter)) und 15 PWM-Signale senden können (PWM=Puls Weiten Modulation; pseudo Analog). Viele Einplatinencomputer wie z.B. der Raspberry PI haben keinen ADC, der aber für einige Sensoren zwingend notwendig ist. Das wichtigste Argument gegen einen Einplatinencomputer ist die Beschränkung auf 3.3V. Viele Sensoren arbeiten mit 5V, weswegen diese auch ihr Ausgangs Signal mit 5V senden, was die meisten Einplatinencomputer zerstören würde. Einige Mikrocontroller wie z.B. der Arduino Due arbeiten ebenfalls mit maximal 3.3V und scheiden deswegen aus.

Nach diesen Überlegungen bleibt (abgesehen von schlecht dokumentierten, mit denen man nicht unbedingt arbeiten möchte) nur eine Option: Arduino.

Während der Entwicklung und möglicherweise im finalen Produkt wird ein Arduino MEGA 2560 verwendet werden, da dieser mehr Pins und mehr Speicher als andere Arduino-Boards hat. Daher muss während der Entwicklung nicht mit Mehreren Controllern zur Evaluierung der Sensoren gearbeitet werden. Wenn möglich soll aber im finalen Produkt ein Arduino UNO oder Arduino Pro Mini zum Einsatz kommen.

2.2. Wahl der Programmiersprache

Die Programmiersprache mit der die Firmware des Geräts geschrieben wurde war, nachdem feststand, dass ein Arduino verwendet werden würde bereits klar: Die angepasste Mischform aus C und C++, mit der Arduinos üblicherweise programmiert werden. Für die Auswertungssoftware gab es im Grunde zwei Möglichkeiten: Java oder C#. Für Java sprach, dass bereits bekannt war, dass etwas ähnliches Projekt bereits existierte und es daher möglich sein musste das Projekt durchzuführen. Außerdem bietet Java gegenüber C# den Vorteil Plattformunabhängig zu sein. C# ist zunächst nur auf einigermaßen aktuelle Windows-Versionen beschränkt. Es ist zwar möglich C# Code auf Linux auszuführen, dies bedarf allerdings einer speziellen Software und der Code muss gesondert für Linux kompiliert werden. C# bietet für die meisten Anwendungsfälle zwar weit überlegene APIs, jedoch war mir keine Library bekannt, mit der das zeichnen von Graphen ohne weiteres möglich ist.

Aufgrund der Plattformunabhängigkeit und dem fehlenden wissen von passenden C# Libraries fiel die Wahl auf Java, obwohl C# sehr viel angenehmer gewesen währe.

# 3. Ursprüngliche Anforderungen und erste Pläne für die Umsetzung

3.1. Anforderungen der Fachschaften Physik und Geographie

3.2. Umsetzungspläne

# 4. Probleme / Technische Schwierigkeiten bei der Umsetzung

4.1. Probleme beim Bau der Hardware

Beim ersten Test des Geräts schien irgendetwas mit der SD-Karte oder der SD-Library nicht in Ordnung zu sein, da nach einer Test-Aufzeichnung immer noch keine Datei auf der SD-Karte zu finden war. Es stellte sich heraus, dass das Problem darin lag, dass die SD-Library nur 8.3 Dateinamen unterstützt (<https://de.wikipedia.org/wiki/8.3>). Die Lösung für dieses Problem war denkbar einfach: Den Dateinamen kürzen. Nachdem nun die Messwerte aufgezeichnet werden konnten, kristallisierte sich ein weiteres Problem heraus. In der Datei mit den Messwerten fand sich häufig „NaN“ (= Not a Number) anstatt eines Messwertes. Bei genauerer Untersuchung stellte sich heraus, dass „NaN“ etwa alle 5-10 Minuten in den Messreihen des DHT22-Sensors fand. Diese Beobachtung führte zu dem Entschluss zwei andere Sensoren zusätzlich anzuschließen und über den Zeitraum von einigen Wochen zu testen, welcher Sensor am besten geeignet ist. Die beiden zusätzlich angeschlossenen Sensoren waren beide von Bosch: der BMP180 und der BME280. Beide Sensoren können die Temperatur und den Luftdruck messen, der BME280 kann außerdem auch, wie der DHT22, die Luftfeuchtigkeit messen. Spätestens jetzt müsste klarwerden, dass es Probleme beim Anschließen geben muss, da BMP180, BME280, die RTC (Real-Time Clock / Uhr) und ein LCD Adapter alle an die Pins 20 und 21 am Arduino Mega 2560 angeschlossen werden müssen, da der Arduino nur über diese beiden Pins durch das I²C-Protokoll kommunizieren kann, das von allen vorher aufgeführten Bauteilen verwendet wird. Glücklicherweise ist I²C ein Bus-Protokoll d.h. können mehrere Geräte an diesen Bus angeschlossen werden, die dann nur Daten senden, wenn der Master (dem steuernden Gerät, in diesem Fall der Arduino) z.B. von Gerät LCD-Adapter mit der Adresse 0x27 Daten anfordert. Das bedeutet, dass der Arduino in regelmäßigen Intervallen ein einzelnes Gerät ansprechen muss, das daraufhin seine Messwerte zurücksendet und danach wartet bis wieder Daten angefordert werden.

Low power library funzt nicht => MosFet+AVR libs

Manche sensoren falsche temperature => zu nah am mq135

4.2. Probleme bei der Software

Das erste Problem, das sich einstellte war, dass es keine geeigneten Beispiele bzw. keine Dokumentation zu JFreeChart, also der Library, mit der die Wertverläufe gezeichnet werden sollten, gab. Weitergeholfen hat hier ein Video auf YouTube, in dem jemand einen Photowiderstand an einen Mikrocontroller angeschlossen hatte und die Messwerte in Echtzeit als Graph darstellen ließ (<https://www.youtube.com/watch?v=cw31L_OwX3A>). Natürlich konnte das aus dem Quellcode gewonnene Wissen nicht direkt für dieses Projekt verwendet werden, da in dem gefundenen Quellcode nur eine Messreihe gezeichnet werden konnte und es keine Option gab, wie von der Fachschaft Physik angefordert, die Messwerte an einer bestimmten Stelle auszugeben, nachdem auf einen Punkt in der Zeichenebene geklickt wurde. Aus der Demo-Datei, die beim Download der Library mitgeliefert wird, kann abgelesen werden wie einzelne Features heißen, jedoch nicht wie sie Implementiert werden sollen. Anhand der Namen der gesuchten Klassen konnte mithilfe der Code-Vorschläge von Eclipse erraten werden wie auch die von der Fachschaft Physik angeforderte Funktion implementiert werden konnte.

Nachdem nun eine Grundlegende Funktionalität gegeben war musste nun das Layout geordnet werden, da manches so seltsam angeordnet wurde, dass Teile der Bedienoberfläche gar nicht zu sehen waren oder einfach nicht lesbar waren. (siehe Bilder 1-8) Möglicherweise sind die Zuordnungen welches Element an welcher Stelle stehen soll redundant (z.B. übergeordnetes Element und untergeordnetes Element werden beide angewiesen sich am linken Rand anzuordnen). Teilweise traten hier auch sehr merkwürdige Phänomene wie das völlige Ignorieren von Anweisungen oder das ausführen der gegenteiligen Reaktion. Beispielsweise haben sich manche Elemente, nachdem sie mit der Anweisung „element.setAlignmentX(Component.LEFT\_ALIGNMENT);“ nicht wie gewünscht an den linken Rand bewegt, sondern sich so weit nach rechts bewegt, dass das Element zu etwa 75% aus dem Sichtbereich verschwunden ist. Stattdessen hatte die Anweisung „element.setAlignmentX(Component.RIGHT\_ALIGNMENT);“ den Effekt, dass sich das Element am linken Bildschirmrand anordnete.

Um mehr Platz für die Graphen zu haben kam die Idee auf die Java Full-Screen Exclusive Mode API zu verwenden auf. Wie in den Programmierhinweisen (http://docs.oracle.com/javase/tutorial/extra/fullscreen/exclusivemode.html) erwähnt, wurde mit dem Methodenaufruf isFullScreenSupported()überprüft, ob die Funktion unterstützt wird. Das Ergebnis dieser Abfrage war stets *true,* obwohl der Aufruf der Methode setFullscreenWindow(JFrame)zwei unterschiedliche Ergebnisse hervorbrachte. (Siehe Bild 5 und Bild 6) Die Ursache für die unterschiedlichen Ergebnisse war, dass beim ersten Test eine AMD Radeon HD5450 und beim zweiten Test, aus anderen Gründen diese dedizierte GPU ausgebaut war und die Bildausgabe über die integrierte Intel HD 610 erfolgte. Aufgrund der Tatsache, dass die Software, auch während die Intel Grafikeinheit für die Bildausgabe verantwortlich war, meldete, dass der Vollbildmodus möglich sei, wurden weitere Tests auf anderen Computern ausgeführt. Als Ergebnis konnte festgehalten werden, dass auf allen Computern die eine Integrierte Grafikeinheit von Intel verwenden das Problem, wie in Bild 6 zu erkennen auftrat. Es Gab zwei mögliche Lösungen für dieses Problem. Die erste war, die Vollbildfunktion gänzlich zu entfernen und die zweite Möglichkeit war, eine Fallunterscheidung einzufügen, die zunächst überprüft auf welchem Betriebssystem das Programm ausgeführt wird und je nach System per Befehlszeile den Namen der verwendeten Grafikeinheit abfragt. Wenn der Name der Grafikeinheit das Wort „Intel“ enthält wird das Fenster nur maximiert, und kein Vollbildmodus verwendet. Wenn der Name jedoch das Wort „Intel“ nicht enthält wird der Vollbildmodus verwendet. Das Funktionieren dieser Unterscheidung wurde sowohl auf Windows (Windows 7, 8 und 10) als auch unter Linux (Debian und Ubuntu) mit ATI/AMD Grafikkarten (sowohl Integriert als auch dediziert), Nvidia Grafikkarten und Intel Grafikeinheiten getestet.

Nachdem nun die Software weitgehend stabil lief wurden nun auch Aufzeichnungen langer Messperioden mit etwa 1.000.000 Datensätzen zu je 16 Messwerten durchgeführt. Berits beim ersten Test wurde klar, dass Anpassungen zwingend notwendig waren: die Ladezeit für diese etwa 16.000.000 Messwerte betrug ungefähr 4 Stunden. Als Lösung für dieses Problem wurde die Möglichkeit die Werte zu Interpolieren eingefügt d.h. dem Benutzer wurde die Möglichkeit gegeben auszuwählen wie viele Werte zwischen den zu zeichnenden Werten übersprungen werden sollten. Außerdem werden in der finalen Version nicht mehr 16 Werte pro Datensatz gespeichert, sondern voraussichtlich nur noch 5 Werte.

# 5. Erweiterte Anforderungen und neues Konzept für die Umsetzung

5.1. Hinzugefügte Anforderung

5.2. Überarbeitetes Konzept

5.3. Probleme bei der Fertigstellung

# 6. Fertiges Projekt

# 7. Betriebsanleitung

# 8. Inhaltsverzeichnis