Wir drei wohnen alle in derselben Wohnung deshalb haben wir die meiste Zeit **zusammen am Projekt** gearbeitet. Alle drei hatten am Anfang gar **keine Vorkenntnisse** über die **AVR-Technologie** und über die **Programmiersprache** C. Um ein besseres Verständnis zu erlangen haben wir uns ein **Elegoo Uno Startet-Kit** gekauft, um uns mit allem bekannt zu machen. Das dabei enthaltene Elegoo Uno-Board **unterscheidet** sich in so gut **wie nichts** zu einem **Arduino** Uno, es nur eine billigere Alternative. Das Starter-Kit **beinhaltet** neben dem Elegoo **unterschiedlichste Sensoren** wie z.B. Bewegungssenore, Fernbedienung.. aber auch verschiedene Motoren.

In den **ersten Wochen arbeiteten** wir mit vielen Sensoren, um die **Grundprinzipe** der Programmierung und der Elektronik eines Mikrocontrollers **besser zu verstehen**.

Eines unserer Arduino Projekte war ein Einparksensor. Dazu verwendeten wir einen aktiven Buzzer und einen Ultraschallsensor.

Ein **aktiver** Buzzer kann **nur eine Tonfrequfenz** ausgeben und wird somit nur mit **HIGH** oder **LOW** betrieben. Im Gegensatz zum Aktiven gibt es auch einen **passiven Buzzer**, dieser kann über die analogen Ausgänge des Arduinos verschiedene Töne von sich geben.

Der Ultraschallsensor misst den **Abstand** zum **nächsten Objekt** bis zu einer Distanz von **2 bis maximal 5 Metern.** Dieses haben wir nun mit dem aktiven Buzzer verbunden.

Auf der Folie sehen wir den Arduino Code für dieses Projekt. Als erstes wurde die Library für den Ultraschallsensor importiert und ein Objekt davon erzeugt. Der Buzzer wurde an Pin 9 angeschlossen und als Output definiert. Durch den Aufruf der Methode „Distance()“ erhält man die Distanz in cm. Je nach Abstand gibt der Buzzer ein Signal von sich, dass umso schneller sich wiederholt je näher sich das gemessene Objekt befindet.

Nach den Projekten **mit** dem **Arduino** überlegten wir nun was sich am **besten für eine Übertragung an eine Station eignet.** Wir **entschieden uns für eine Wetterstation,** die wir auch je nach Schnelligkeit des Fortschritts im Laufe des Projekts noch durch einen weiteren Sensor erweitern können. Für den **Anfang** sollte unsere Wetterstation die **Temperatur** und die **Luftfeuchtigkeit** messen. Dazu verwenden wir einen DHT11-Sensor:
Mit dem DHT11 Sensor ist es möglich **Temperaturen von 0-50°C** und **Luftfeuchtigkeit von 20-90%** zu messen. Der Sensor **verwendet ein single-wire serial interface** und ist somit leicht und schnell zu programmieren. Der DHT ist klein und verbraucht nur **wenig Strom**.
Mit dem **Arduino** hatten wir den Temperatursensor **bald am Laufen**, da wir eigentlich nur die geeignete **Bibliothek** und den **Sketch einbinden** und ausführen mussten.

Nachdem der **Temperatursensor** mit unserem Arduino funktionierte wollten wir nun auch mit dem **CC1101 arbeiten**. Wir fanden eine dazu eine **Bibliothek** im **Internet**. Zusätzlich zu der Bibliothek kopierter wir zwei verschiedene Sketches, einmal für das **Gerät** das **sendet** und einmal für das **Gerät** das die Daten **empfangen** soll. Die Übertragung mit der gefundenen Bibliothek und Sketches testeten wir im Praktikumsraum mit dem Arduino und dem CC1101-Modul dort. Das Übertragen von Daten gelang uns auf Anhieb.

Nachdem wir auf dem Ardiuno den Temperatursensor und den CC1101 getestet haben, wollten wir mit dem **Bau unseres eigenen Boards** starten. Wir haben uns für einen **AVR**-**Microcontroller** entschieden, denn sie sind **ausgelegt**, um **effizienten C-Code auszuführen**. Die AVR **Familie** wird hauptsächlich in **drei Typen klassifiziert: tinyAVR, megaAVR** und **XMEGA** welche sich lediglich in der Leistungsfähigkeit und ein wenig in der Funktionalität unterscheiden. Das **naheliegendste** war es den **gleichen Chip** wie auf dem Arduino Uno da wir davon Vorkenntise haben, also den Atmega328p zu verwenden.

Der **Atmel ATmega328p ist ein 8-bit AVR-Microkontroller**, welcher mit einer **Spannung von zwischen 1,8 bis 5,5 Volt** und bis zu einer maximalen **Taktrate** bis zu **20MegaHerz** arbeitet.
Die Atmega Chips haben immer das gleiche Schema in der Namensgebung z.B. bei Atmega 328p

1. Die **Baureihe** (hier Atmega)
2. Nach der Baureihe kommt eine Nummer (immer Zweierpotenz), welche die Größe des **Flash**-**Speichers** angibt. Also in unserem Beispiel sind das 32 KiB
3. Nach dem Flash-Speicher können bis zu **drei weitere Ziffern** folgen. Diese definieren wieder die **Baureihe**, die **Anzahl der I/O Pins und Zusatzfunktionen** an. Bei unserem Chip haben wir nur eine Ziffer, die 8. Wir haben hier die **neuste Baureihe** von 4 bis 32 KiB Flash-Speicher, ein mit **28 – 32** pinkompatibel Gehäuse und **verbesserte Funktionen**, wie z.B. **Timern**. Mit ihrer Hilfe ist es möglich, in regelmäßigen Zeitabständen Aktionen zu veranlassen.
4. An vierter Stelle kommen bis zu zwei Buchstaben, die für die **Revision** und spezielle **stromsparende Architektur** stehen. Das P steht für **Pico**-**Power**. **Es erlaubt den Mikrocontroller** mit einer relativ niedrigeren Volt Anforderungen zu arbeiten
5. Zuletzt kommt ein Bindestrich und bis zu zwei weiteren Buchstaben, die die **Bauform** angeben. PU steht für ein bleifreies DIP-Gehäuse (Dual in-line package)

Als erstes, haben wir im Internet nach einer Anleitung gesucht, um den Chip aus dem Arduino zu nehmen, und ihn auf einem Breadboard laufen zu lassen. Auf der Arduino Website fanden wir ein Tutorial, wie man den Chip nur mit Hilfe eines Arduino Boards programmieren und mit Strom versogen kann. Dafür sollte man den Chip vom Arduino nehmen und ihn auf ein Breadboard stecken.
Im Falle eines leeren Chips musste man als ersten Schritt den Arduino Bootloader auf den Chip laden, aber das mussten wir nicht tun, da wir den Arduino-Chip verwendeten. Der nächste Schritt war das Board ans Arduino anzuschließen. Einfach Volt und Ground verbinden, den Reset vom Arduino und VCC mit einem 10k Widerstand an den Reset Port des Chips anschließen und noch den TX vom Arduino an den RX vom Chip und andersrum anstecken. Wo sich diese Ports alle befinden, ließ sich leicht aus dem Datasheet des Chips herauslesen.

In der Arduino IDE mussten wir Port und den richtigen Chip auswählen, um den ersten Blink Sketch hochzuladen. Aber das hat bei uns, auch nach längerer Fehlersuche nicht funktioniert. Deshalb suchten wir im Internet nach weiteren Anleitungen, um den Chip auch ohne Arduino zum Laufen zu bringen. Aber bei den meisten Tutorials benötigte man einen 16MHZ Crystal, den wir aber nicht besaßen und wir auch im Praktikumsraum keinen finden konnten.

Das Problem jetzt war, dass wir einen Programmer brauchten, um den Chip zu programmieren, weil nun nicht mehr das Arduino als Programmer herhalten konnte. Im Praktikumsraum fanden wir dazu mehrere Pololu Avr Programmer. Der Pololu USB AVR-Programmer ist ein kompakter ISP (In-System Programmer) für AVR Mikrocontroller. Der Programmer stellt ein Interface zur Übertragung eines kompilierten AVR Programms vom PC zum Flash Speicher des Chips zur Verfügung. Nach schlau machen auf deren Seite, haben wir den Programmer an unserem Microcontroller angeschlossen.

08-Atmel Studio und Tera Term:

Neben Programmer verwendeten wir ab jetzt nun auch nicht mehr die Arduino IDE sondern suchten nach einer passenden Entwicklungsumgebung. Nach kürzeren Recherchen wurde uns Atmel Studio (ehemaliges AVR Studio) für Windows empfohlen und so entschieden wir uns für diese IDE. Bevor man mit Atmel Studio programmieren kann, muss man noch ein paar Anwendungen installieren. Dazu zählt WinAVR. WinAVR enthält neben dem C und C++ Compiler für alle AVR Mikrocontroller , noch einige andere Pakete, z. B. die Bibliotheken avr-libc oder avr-io, die Programmiersoftware AVRDUDE und den Editor Programmer's Notepad.

Atmel Studio besitzt leider keinen wirklich brauchbaren Seriellen Monitor, deshalb installierten wir uns die Software “Tera Term” mit der wir die Ausgaben kontrollieren konnten. Tera Term ist ein kostenloser Terminal-Emulator, der einfach einzurichten ist. Beim starten des Terminals kann man den Port auswählen, dessen Ausgaben man anzeigen lassen will. Da der Pololu Programmer zwei Ports besitzt, konnten wir einen zum Programmieren und den anderen als Schnittstelle für unseren seriellen Monitor benutzen.

Wir dachten, dass unser Microcontroller jetzt auch mit Strom versogt war, da er ja durch dem Programmer an Vcc und Ground angeschlossen wäre. Allerdings leuchtete die LED auf dem Pololu immer noch rot und wir gelangen nicht in den Programming-Mode.

Wir mussten uns daher um eine externe Stromversorgung kümmern. Die meiste Zeit arbeiteten wir zu Hause und konnten so nicht die Geräte im Praktikumsraum verwenden. Anstelle verwendeten wir den Elegoo Power MB v2 der auch bei dem Starter Kit mitgeliefert wurde. Mit diesem waren wir nun in Stande unseren Microcontroller mit 3,3 oder 5 Volt zu versorgen. Manchmal verwendeten wir auch das Elegoo Uno Board zur reinen Stromversorgung.

Nun haben wir alles angeschlossen und konnten unser erstes C++ Programm auf den Chip laden. Wie beim Arduino war das ein einfacher Blink Sketch.

Hier auf der Folie sieht man den Code für den Sketch, mit dem wir eine LED zum Blinken bringen wollen. Im Gegensatz zum Arduino sieht es jetzt schon deutlich komplizierter aus. Da wir kein Arduino Framework zum Programmieren verwenden, konnten wir Methoden wie pinMode, digitalRead oder digitalWrite nicht verwenden. Deshalb haben wir eine Header Datei erstellt, welche wir importieren, in der wir die 3 Methoden selbst geschrieben haben, die die Arduin.h ersetzten sollen.

Auf diese möchte ich jetzt kurz eingehen.
Mit der Methode PinMode kann man einen Pin als OUTPUT oder INPUT definieren. Um einen Pin als INPUT/OUTPUT zu definieren, muss man diesen im zuständigen Register eintragen. DDRD verwendet man für die Ports D, DDRB für die Ports B. Um jetzt den Port D4 als Output einzutragen, schreibt man DDRD |= (1<<4). Um das jetzt erklären, nehmen wir an, dass das Register DDRD so aussieht. Der Ausdruck 1<<4 ist ein leftshift um 4 stellen. Als Startwert nehmen wir 00000001 und verschieben alle Bits um vier Stellen nach Links. Damit erhalten wir 00010000. Diese Bitfolge verknüpft man mit dem "oder" Operator und man erhält diesen Ausdruck. Das vierte Bit ist nun eine 1, also ein Output.
So ziemlich ähnlich sieht es aus, wenn man einen Port als INPUT definieren möchte. Der Unterschied zwischen den 2 ist, dass man nun, nach dem leftshift, alle Bits negieren muss, und dann nicht mit einem Oder verknüpfen, sondern mit einem Und Operator. Bei diesem Beispiel hier möchte ich den Port2 als Input definieren. Wie man auf der Folie sehen kann, ist das zweite Bit nun eine 0, also ein Input.
Die Methode digitalWrite benötigt einen Pin und den Parameter High oder Low. Der Ausdruck für High ist derselbe wie der, den man verwendet, wenn man einen Port als OUT bzw INPUT definieren will. Der Unterschied zur PinMode liegt darin, dass man nun das Port Register verwenden muss, anstatt des DDRD. Das Gleiche gilt für LOW.
DigitalRead bekommt als Parameter nur den Pin. Um zu überprüfen, ob der Input von Port 4 High ist, muss der Ausdruck PIND & (1<<4) true ergeben. Für Low logischerweise false. Zum Code. Nehmen wir an, PORT 4 ist HIGH. PIND besteht zum Beispiel aus der folgenden Bitfolge. Der leftshift ist wieder derselbe wie bei den Beispielen vorher. Nach dem Anwenden des Und Operators erhalten wir eine Bitfolge, die nicht 0 ergibt, also High. In unserem Beispiel haben wir 16 erhalten. Sollte der Port4 LOW sein, dann würden wir 0 herausbekommen.

Der LED-Sketch funktionierte. Der nächste Schritt war nun den Temperatursensor auch auf unserem Standalone-MCU zum Laufen zu bringen.
In erster Linie verwendeten wir den gleichen Code wir vorher bei dem Arduino. Das heißt wir importierten dieselbe Library und kopierten auch den Code von der Main-Datei. Den Temperatursensor zum Laufen zu bringen brachte jedoch in Hinblick auf die anderen Sketches zuvor mehrere Schwierigkeiten. Beim Compilieren des Codes traten viele Fehler auf. Da die Library für Arduino bestimmt war, mussten wir noch ein paar Dinge anpassen. Zum Beispiel mussten wir den Datentyp Byte, den Arduino verwendet, selber hinzufügen. Für pinMode, digitalRead und digitalWrite haben wir unsere Methoden, wie vorher angesprochen, verwendet und wir mussten zusätzlich eine Library für Strings einbinden, sodass es möglich war, den Code erfolgreich zu builden. Nach den Basteleien traten keine Fehler bei der Compilierung des Codes mehr auf. Beim hochladen auf den Chip allerdings funktionierte der Temperatursensor immer noch nicht. Den Wert, den der Sensor uns zuschickt, haben wir in Tera Term ausgeben lassen. Aber anstatt des Wertes für die Temperatur/Luftfeuchtigkeit, wurden nur weiße Kästchen ausgegeben. Der einzige Grund für den Fehler lag nach unserer Meinung an dem Takt des Chips.
Wir haben uns entschlossen, statt dem internen Oszillator des Microcontrollers, der eine Taktrate von bis zu 8Mhz erreichen kann, auf einen externen Quarz mit denselben Mhz wie dem Arduino, also 16Mhz, umzusteigen. Zwar wird in mehreren Foren erwähnt, dass für Sensoren wie einem Temperatur-Sensor der interne Oszillator leicht ausreichen müsste, aber wir versuchten es trotzdem.

Nach dem Einbauen des 16Mhz Quarz und dem Einstellen der Fuse-Bits bei den Programmer-Einstellungen lief der DHT11-
Temperatursensor allerdings immer noch nicht. Wir schlossen somit aus, dass es sich um die Taktrate handelt, weil es sowohl mit internen als auch mit externen Oszillator nicht funktio nierte. Das Programm brach immer mit dem Fehler 105 ab.
Um diesen Fehler euch näher zu bringen, erkläre ich zuerst wie der Sensor
und unser Programm arbeitet.
Vielleicht erster noch ein paar generellere Informationen über den Sensor.
Der DHT11 kann mit 3,3 – 5.5VDC versorgt werden. Bei unserem Microcontroller verwenden wir für die Versorgung eine Spannung von 3,3 Volt.
Ein Kommunikationsprozess zwischen MCU und Sensor dauert in etwa 4 ms.
Für die Kommunikation und die Synchronisation zwischen dem MCU und dem Temperatursensor wird das Single-bus data format verwendet. Das heißt zur Übertragung dient allein 1 Kabel – one-wire.
Der Kommunikations-Prozess gestaltet sich wie folgt: Zu Beginn schickt der Microcontroller ein Start-Signal an den Sensor. Daraufhin schaltet der Sensor in den running-mode. Nun antwortet der Sensor mit einem Signal von 40-bit Daten und sobald die Daten übermittelt wurden geht der Sensor wieder in den low-power-consumption mode über.

Hier der Kommunaktionsprozess noch mal genauer:
Man sieht in dieser Abbildung noch einmal gut, dass der Single-Bus sowohl von MCU als auch vom Sensor verwendet wird und sich die Verwendung je nach Fortschreiten des Prozesses verändert.
Unser Programm setzt den Data Single-bus von high auf low, dieser Vorgang muss eine bestimmte Zeit dauern, sodass der Sensor unser Start-Signal erhält. Nun können wir den Bus wieder auf high setzen und warten auf die Antwort des DHT.
Sobald der Sensor das Start Signal erhalten hat passieren noch einige Start-Prozeduren. Wir können dadurch nochmals feststellen ob für die Übertragung der Daten alles funktioniert, ansonsten wird die Übertragung abgebrochen und ein Fehler zurückgegeben. Sind diese Prozeduren erfolgreich abgeschlossen beginnt nun der eigentliche Datenaustausch.

Jedes Bit wird folgendermaßen übertragen:
Der Sensor beginnt mit einem 50us low-voltage-level.
Die confirm-Methode, die wir hier sehen, wird immer dann verwendet, wenn eine Übertragung vom Sensor stattfindet, um zu überprüfen ob sie rechtmäßig, wie von uns erwartet, abgelaufen ist. Im Laufe des Programms gibt es insgesamt 6 Stellen wo bei Fehlern abgebrochen wird, um so genauer zu identifizieren wo das Problem liegt. Ein Beispiel wäre hier:
Wir hatten ein 50us low-voltage-level dieses wir nun von einem high-voltage-level gefolgt. Die Länge dieses Sinals bestimmt den Wert eines Bits also 1 oder 0. Dauert das Signal zwischen 26-28us dann überträgt der Sensor eine “0”, hat das Signal eine Länge von 70us überträgt er eine “1”. Diese us werden hier in der for-Schleife gezählt mit Hilfe von ticks zu je 10 us. Waren es über 3 ticks, also eine 1, sind es weniger als 3 ticks eine 0.
Ist das 40. und letzte Bit übertragen folgt ein pull down des voltage level für 50us. Danach geht der Single-Bus voltage zurück in den free status.

Nach der Übertragung der 40 Bits müssen diese natürlich noch richtig geparsed werden, sodass sie auch im richtigen Format ausgegeben werden können, darauf gehen wir aber jetzt nicht genauer ein.
Hier aber noch ein kurzer Einblick auf die Checksumme. Als letztes der 5 übertragenen Bytes handelt es sich um eine Checksumme. Addiert man alle vorherigen Bytes und nimmt die ersten 8 Bits dieser Summe, so muss man die Checksumme erhalten. Dies wird hier im Code überprüft. Ist das der Fall, dann kann Temperatur und Luftfeuchtigkeit ausgegeben werden, ist das nicht so, dann hat sich ein Übertragungsfehler ereignet und die Übertragung war wertlos und muss wiederholt werden. Dieser Fehler hat die die Nummer 105.

Das heißt wie vorher angesprochen, unser Programm hatte immer wieder Bitübertragungsfehler.
Wir fanden heraus, dass diese in der Implementierung der delay-Funktion lagen. Um den Code der DHT-Bibliothek nicht verändern zu müssen haben wir nämlich eine eigene delay-Methode geschrieben.
Wir tauschten alle Methodenaufrufe durch die ursprüngliche Form aus und nun endlich funktionierte unser Temperatursensor. Die Abweichung, die zwischen den zwei Varianten liegt ist uns bei den vorherigen Sketches, die wir schrieben, nicht aufgefallen da diese Programme keine so genauen Delays wie der DHT Sensor benötigen.
Kommen wir wieder zurück zu den Oszillatoren. Durch den externen Quarz muss der Microcontroller für ein sichergestelltes Laufen mit ca. 5 Volt versorgt werden und durch die erhöhte Taktzahl steigt der Stromverbrauch. Wir versuchten somit nochmals unseren Sensor auch ohne externen Oszillator zum Laufen zu bringen. Durch den Austausch der delay-Methoden, funktionierte der Temperatur und Luftfeuchtigkeitssensor jetzt aber auch nur mit dem interne Oszillator des Atmega 328p.

Da nun der Temperatur-Sensor verlässlich arbeitet, haben wir uns wieder unserer Übertragungstechnologie, dem CC1101, gewidmet. Zum Schluss gelang uns dort noch eine Kommunikation mit unserem Standalone Microcontroller und dem Arduino, wie am Anfang von Arduino zu Arduino. Allerdings erhielt der Arduino nicht die richtigen Daten sondern wieder nur undefinierte Kästchen. An diesem Problem werden wir nach der Klausurenpause als erstes weiter arbeitet.
Das Projekt bis jetzt hat uns allen drei sehr gut gefallen auch wenn es immer wieder zu frustrierenden Phasen gekommen ist. Uns fehlt es einfach an Grundverständnis bei diesen ganzen Sachen, vor allem bei der Elektronik, bei Hardware bezogenen Dingen, aber auch bei der Entwicklung mit C und bei gewissen Grundeinstellungen, die vorgenommen werden müssen bis der Microcontroller überhaupt läuft. Daher haben wir immer wieder ziemlich viel Zeit in das Projekt investiert, ohne große Fortschritte zu machen. Weil jedes Mal beim Auftreten von kleinen Fehlern konnten wir diese nicht sofort beheben. Das Programmieren des Microcontrollers an sich fanden wir weniger schwierig.
Wir haben in diesem Praktikum bis jetzt alle ziemlich viel dazugelernt, vor allem natürlich über die AVR-Technologie, aber auch unsere C-Kenntisse konnten wir erweitern. Weiteres konnten wir auch das Grundverständis von Seiten der Elektronik verbessern.

Aktuell sieht unser Mikrocontroller wie folgt aus ohne den Pololu avr Programmer und den Transceiver CC1101:
Links das vorübergehend verbaute Netzteil Elegoo mb v2 für unsere Stromversorgung. An dieses Netzteil kann entweder eine Batterie oder einen Kabel zur Steckdose angeschlossen werden. Wir versorgen den Mikrocontroller mit 3,3 Volt. An unsere Chip haben wir einen Resetbutton an den RESET-Pin hinzugefügt und hier recht ist unser Temperatur und Luftfeuchtigkeit Sensor welcher mit Pin 3 verbunden ist.

Der DHT11-Sensor funktioniert nun, vielleicht werden wir unsere Wetterstation um einen weiteren Sensor erweitern wie z.B. einen Luftdrucksensor oder Luftqualitätssensor. Natürlich müssen wir als nächstes das Transceiver Modul fertigstellen und Details mit dem Station-Team besprechen um herauszufinden was dann die nächsten Schritte sein werden. Zusätzlich muss noch geklärt werden wie unsere Stromversorgung aussieht und wie wir sie gestalten werden. Und zum Schluss muss noch alles auf die Platine. Aber das werden wir nach der Klausurenpause klären. Der Entwurf für die Platine sieht bis zum jetzigen Zeitpunkt wie folgt aus: