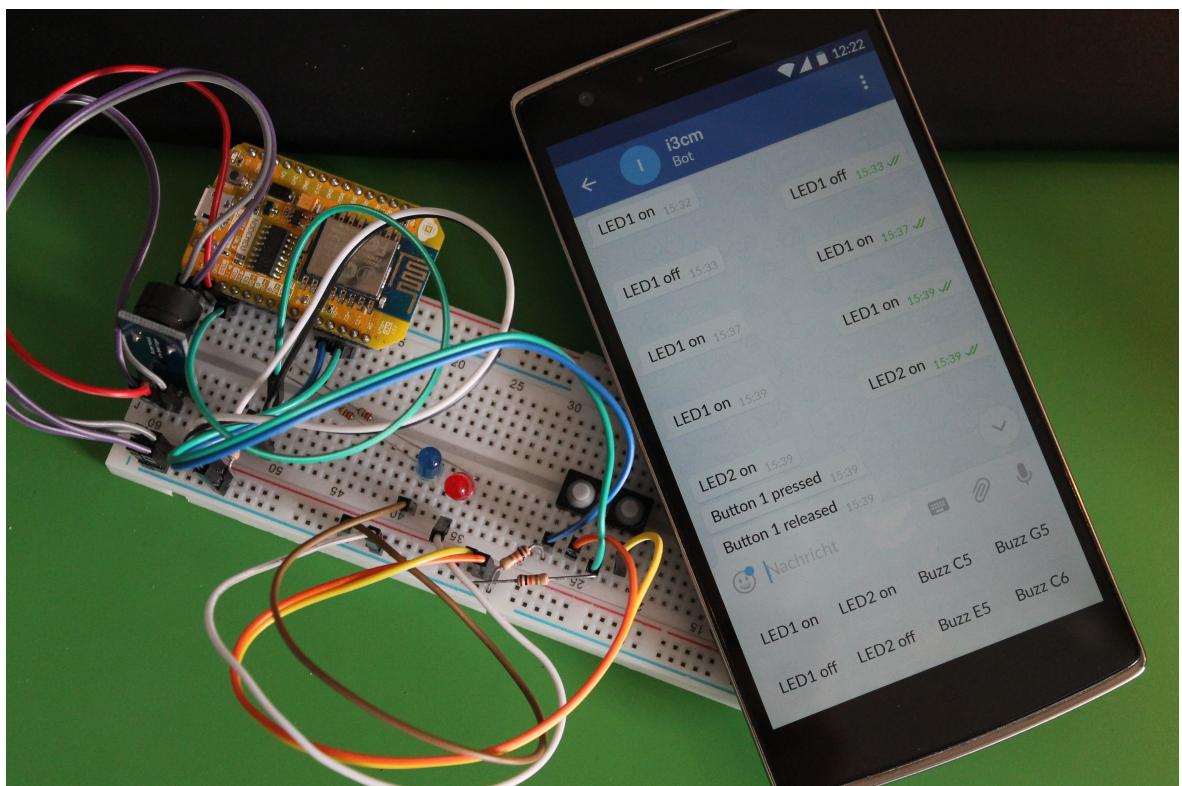


oLaF - offenes Labor Fulda

Arduinokurs Digitalisierung 2019

Jörn Schlingensiepen



©2018, 2019 Jörn Schlingensiepen, alle Rechte vorbehalten

Der Umdruck ist erst kurz vor dem Workshop fertig geworden. Daher werden sich noch etliche Fehler darin befinden und die ausgewiesenen Quellen sind leider unvollständig.
Melden Sie bitte alle Rechtschreib- oder sonstigen Fehlern und Anregungen per eMail an joern@schlingensiepen.com

Inhaltsverzeichnis

1 Einführung	5
1.1 Aufbau des Umdruckes	5
1.2 Links und Verweise	5
1.3 Menüeinträge und Tastenkombinationen	5
1.4 Quellcode	6
1.5 Begriffe und Definitionen	6
1.6 Beigefügte Informationen	6
2 Inbetriebnahme	7
2.1 Vorstellung der Hardware	8
2.1.1 Der ESP8266 - eine MCU (microcontroller unit)	8
2.1.2 Programmierung des ESP8266	8
2.1.3 Speicher für den ESP8266	8
2.1.4 Beispiel für ein Produkt mit einem ESP8266	9
2.1.5 ESP-XX, fertige Module für den Einsatz	9
2.1.6 NodeMCU Development Board mit ESP-12	10
2.2 Hardwareaufbau #1	12
2.3 Installation der Software	14
2.3.1 Installation der IDE	14
2.3.2 Verbinden der IDE zum NodeMCU	15
2.4 Das erste Programm starten	17
3 Eine (sehr) kurze Einführung in die Programmierung	19
3.1 Algorithmus, Variable und Fluss	19
3.1.1 Beispiel TurtleBot	19
3.1.2 Beispiel: Größter gemeinsamer Teiler	21
3.2 Zusammenfassung	26
4 Programmieren mit Arduino	29
4.1 Basics	29
4.1.1 Deklaration von Variablen	29
4.1.2 Zuweisungen	29
4.1.3 Initialisierung von Variablen	30
4.1.4 Kommentare	30
4.2 Flusskontrolle	31
4.2.1 Bedingungen	31
4.2.2 Verzweigung	31
4.2.3 Schleifen	32
4.2.4 Schleifenabbruch	33
4.3 Unterprogramme	34
4.3.1 Deklaration eines Unterprogramms	34

4.3.2	Aufruf eines Unterprogramms	35
5	Ein eigenes Programm schreiben	37
5.1	Analyse von myBlink.ino	37
5.2	Hardwareaufbau #2	39
5.3	Hardwareaufbau #3	40
5.4	Zustände	41
5.5	Kommunikation nach außen	42
5.6	Kommunikation von außen	43
6	Einbinden ins Netzwerk	45
6.1	Einbinden interner Bibliotheken	45
6.2	Einbinden ins WLAN	45
6.3	Verbinden mit einem WebServer	46
6.4	Hardwareaufbau #4	48
6.5	Steuern über Chat-Bot	49
6.5.1	Einbinden externer Bibliotheken	49
6.5.2	Einrichten eines eigenen Bots	50
6.5.3	Benutzen der Bibliothek TelegramBotClient	53
7	Weitere Akteure	55
7.1	Schieberegister	55
7.2	LED-Matrix	57
7.2.1	Funktionsweise	58
7.2.2	Serial Peripheral Interface	59
7.2.3	Ansteuern der LED-Matrizen	61
7.3	LCD Display	63

1 Einführung

Dies ist der Umdruck zum *oLaF-Arduinokurs Digitalisierung 2019*. Diese Arbeitsmappe kann während des Kurses genutzt werden um Abbildungen und Schaltungen nachzuschlagen oder etwas nachzulesen. Später können die Unterlagen als Nachschlagewerk oder zum nochmaligen Nachvollziehen der einzelnen Schritte genutzt werden.

Ziel des Kurses ist es, einen ESP8266 mit der Arduino-IDE zu programmieren, verschiedene Sensoren und Aktoren daran anzuschließen und über das Internet zu steuern. Das Beispiel kann dann genutzt werden, um eigene oLaF-Projekte mit einem Micro-Controller auszustatten.

1.1 Aufbau des Umdruckes

Dieser Umdruck ist so gesetzt, dass man beim Lesen wichtige Teile oder Abschnitte, die eine besondere Bedeutung haben, leicht erkennen kann.

1.2 Links und Verweise

In diesem Umdruck gibt es verschiedene Verweise auf Ressourcen im WWW. Ein Verweis auf eine Website sieht wie folgt aus:

<https://github.com/schlingensiepen/cloudification-esp8266>

Da es in der Regel lästig und fehleranfällig ist einen solchen URL abzutippen ist in der Regel noch ein Kurzlink mit angegeben:

Kurzlink: [drj](#)

Diese Kurzlinks können über die Seite <http://link.i3cm.de> angezeigt oder direkt aufgerufen werden.

Wenn Internetseiten als Quelle angegeben werden, ist das wie folgt notiert:

[<https://www.arduino.cc/>]

Zu diesem Kurs gibt es Beispielquellcode, der über <https://github.com/schlingensiepen/cloudification-esp8266>, Kurzlink: [drj](#) heruntergeladen werden kann. Laden Sie die kompletten Quellen als zip-Archiv herunter und entpacken dieses auf der lokalen Festplatte.

Verweise auf eine Beispieldatei aus diesem Archiv werden wie folgt dargestellt:

[gitHub:myBlink.ino](#)

1.3 Menüinträge und Tastenkombinationen

Anweisungen zum Wählen einer Funktion über das Funktionsmenü einer Softwareanwendung sind wie folgt notiert:

[Hauptmenü](#) → [Menüeintrag](#) → [Unterpunkt](#)

Die Pfeilrichtung gibt an, in welcher Reihenfolge die Punkte mit der Maus ausgewählt werden müssen.

Tastenkombinationen zum Wählen einer Funktion einer Softwareanwendung über die Tastatur sind wie folgt notiert:

[Taste1](#) + [Taste2](#)

Die Tasten sind in der Regel gleichzeitig zu drücken um die Funktionen aufzurufen.

Im Rahmen des Kurses werden Softwarebibliotheken aus den Arduino-Repositories verwendet. Um die Auswahl der richtigen Bibliothek zu erleichtern ist diese immer mit Name und Autor angegeben:

[TelegramBotClient by Jörn Schlingensiepen](#)

1.4 Quellcode

Zur Veranschaulichung sind Beispielquelltexte eingebunden. Diese werden wie folgt dargestellt:

```
int fak(int i)
{
    if (i==0) return 1;
    return fak(i-1);
}
```

Wird im Fließtext auf Quellcode Bezug genommen, so wird dieser innerhalb des Textes durch Verwendung einer anderen Schriftart gekennzeichnet. Aus dem Beispiel oben könnte so `int fak(int i)` im Fließtext dargestellt werden.

Neben Quellcodes werden auch Flussdiagramme verwendet. Ein Bezug im Text zu einem bestimmten Schritt im Flussdiagramm wird so dargestellt: (2).

nernswert ist, wir dieser wie folgt hervorgehoben:

«*Serielle Schnittstelle* »

Wird eine Abkürzung eingeführt sind die Buchstaben, die diese Abkürzung ergeben, wie folgt kenntlich gemacht:

Electrically Erasable Programmable Read-Only Memory kurz *EEPROM*

Für besonders wichtige Begriffe gibt es Definitionen, die wie folgt dargestellt werden:

Definition 1: Schüler

Ein Schüler oder eine Schülerin ist nach der Wortbedeutung ursprünglich eine Person, die im organisatorischen Rahmen einer Schule lernt. Dabei erhalten sie von entsprechenden Lehrern Unterricht, um in einem Lernprozess Fertigkeiten, Wissen, Einsichten und Verhaltensweisen zu erwerben, zu erweitern, einzubüben und zu vertiefen.

1.5 Begriffe und Definitionen

Wenn im Fließtext ein Begriff eingeführt wird, der zur Fachsprache gehört und deshalb erin-

1.6 Beigefügte Informationen

Zu einigen Themen gibt es Exkurse, die wie folgt dargestellt sind:

Exkurs 1: Bedeutung des Exkurses

Exkurse behandeln Themen, die beim Verständnis des Kurses helfen oder interessantes Hintergrundwissen darstellen. Grundsätzlich kann der Kurs auch ohne das Lesen der Exkurse bewältigt werden.

Zum Nachvollziehen und Vertiefen des Stoffes sind Aufgabenstellungen in den Kurs eingearbeitet, diese sind wie folgt dargestellt:

Aufgabe 1: Aufgabentitel

Beschreibung der Aufgabenstellung

Zusätzliche besondere Hinweise sind wie folgt dargestellt:

Anmerkung

Besonderer Hinweis.

2 Inbetriebnahme

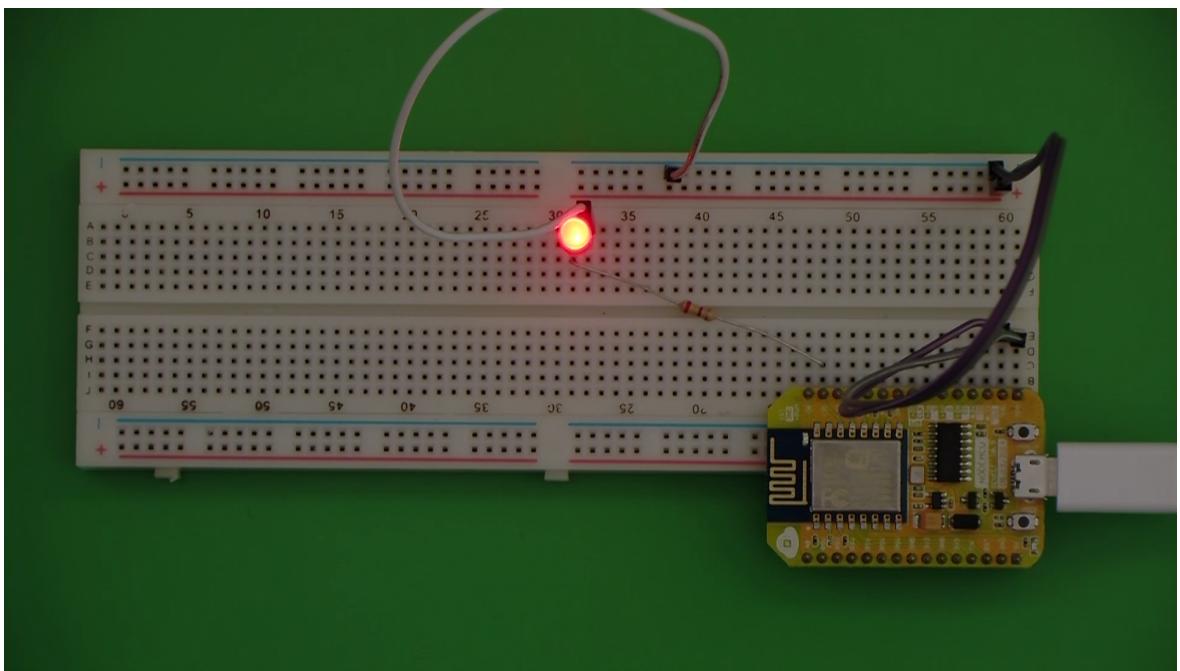


Abbildung 2.1: Ziel dieses Kapitels, der Aufbau zur Inbetriebnahme

Bevor es mit der Theorie losgeht, soll in diesem Kapitel die Hardware in Betrieb genommen werden, d.h. die Hardware wird vorgestellt, angeschlossen, mit einer LED verbunden und ein erstes Programm wird ausgeführt. Abbildung 2.1: *Ziel dieses Kapitels, der Aufbau zur Inbetriebnahme* (Seite 7) zeigt unsere erste Schaltung, die auf einem sog. «*Steckbrett*» (engl. «*Breadboard*») aufgesteckt ist.

2.1 Vorstellung der Hardware

2.1.1 Der ESP8266 - eine MCU (microcontroller unit)



Abbildung 2.2: Aufgelöteter ESP8266EX
[www.espressif.com]

Der Chip ESP8266 ist ein sog. *MCU* (*microcontroller unit*), d.h. ein kompletter Prozessor mit Peripherie auf einem Chip, der sich zur Integration in technische Systeme eignet und dort in der Regel Steuerungsaufgaben übernimmt (embedded applications). Der ESP8266 zeichnet sich dadurch aus, dass er sehr kostengünstig ist und einen kompletten WLAN-Chip enthält, für den auch ein kompletter TCP/IP-Stack implementiert ist. So ist das System in der Lage, über das Internet zu kommunizieren und einfache Informationen zu versenden und Befehle zu empfangen. Hersteller ist Expressif Systems (<http://espressif.com/>).

2.1.2 Programmierung des ESP8266



Abbildung 2.3: Beispiele für RS232 Schnittstellen

Der Chip ist programmierbar über ein *universal asynchronous receiver-transmitter*

(*UART*), also eine serielle Schnittstelle, ähnlich zu der seriellen Schnittstelle (RS232), die bis vor ein paar Jahren in PCs verbaut wurde. Diese Schnittstellen sind sehr weit verbreitet, es gibt viele elektronische Bausteine, die über eine solche Schnittstelle verfügen. Für den Anschluss an PCs gibt es elektronische Bausteine, die auf der einen Seite einen USB-Anschluss haben und auf der anderen Seite eine solche *UART-Schnittstelle*. Mit einem solchen «*USB-zu-UART-Konverter*» können verschiedenste elektronische Bausteine mit einem PC verbunden werden.

2.1.3 Speicher für den ESP8266

Die MCU braucht noch einen Speicher in dem das Programm abgelegt ist. Dieser Speicher darf nicht gelöscht werden, wenn der Strom ausgeschaltet wird. Diese Art von Speicher nennt man *Electrically Erasable Programmable Read-Only Memory* kurz *EEPROM*. Das heißt im Deutschen ungefähr «*elektrisch lösbarer programmierbarer Nur-Lese-Speicher*» und beschreibt einen Speicher, der im Normalbetrieb/Normalmodus nur lesbar, aber in einem speziellen Programmiermodus beschreibbar ist. Also genau das, was man für eine MCU braucht. Startet die MCU, dann lädt sie im Normalmodus aus dem Speicher das abgespeicherte Programm und startet so die Steuerung des angeschlossenen Gerätes.

Nach dem Zusammenbau oder wenn man das Programm nochmal ändern will, kann man die Schaltung in den Programmiermodus versetzen und ein neues, anderes Programm einspeichern. Nach dem Starten lädt die MCU dann das neue Programm und verhält sich entsprechend anders.

Im Fall des ESP8266 ist der Speicher über einen sog. *Serial Peripheral Interface Bus (SPI-Bus)* angeschlossen. Dies ist ein Standardanschluß, so dass man Bauteile verschiedener Hersteller oder Lieferanten einbauen kann.

2.1.4 Beispiel für ein Produkt mit einem ESP8266



Abbildung 2.4: Beispiel für ein Produkt: SOnOff, schaltet 230V über WLAN



Abbildung 2.5: Zusammengebauter SOnOff-Schalter

2.1.5 ESP-XX, fertige Module für den Einsatz

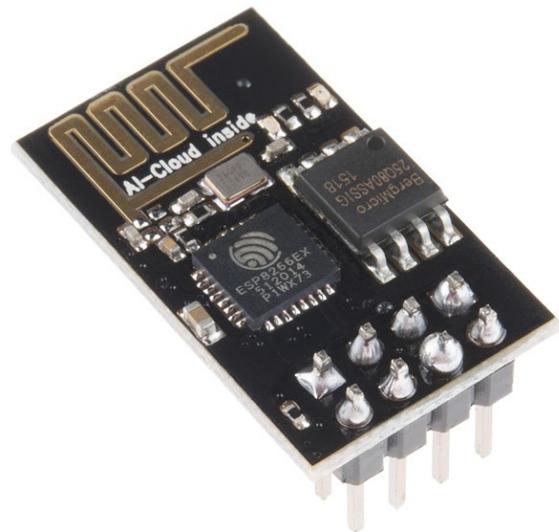


Abbildung 2.6: Der ESP-01 von AI-Thinker [www.antratek.de]

Es gibt auch fertige Produkte, die mit einem ESP8266 realisiert sind, z.B. den in Abbildung 2.4: *Beispiel für ein Produkt: SOnOff, schaltet 230V über WLAN* (Seite 9) und Abbildung 2.5: *Zusammengebauter SOnOff-Schalter* (Seite 9) abgebildeten Schalter SOnOff. Der Schalter kann 230V Wechselstrom, also den Strom, der aus einer üblichen Steckdose in Europa kommt, schalten. Auf der Abbildung kann man erkennen, dass in der Mitte ein Relais montiert ist, welches den Strom, der auf den breiten Leiterbahnen geführt wird, schaltet. In der Mitte kann man erkennen, dass dort auch eine UART-Schnittstelle auf der Platine ist. Diese kann z.B. mit einem «*USB-zu-UART-Konverter*» verbunden werden um den ESP8266 auf der Platine zu programmieren.

Seit 2014 stellt AI-Thinker (<https://www.ai-thinker.com/>) komplettete Module her, die aus einem ESP-8266, entsprechender Leistungselektronik, einem EEPROM, einem Sende- und Empfangsverstärker und einer WLAN-Antenne bestehen.

Diese Bausteine arbeiten mit einer Betriebsspannung von 3,3V und tragen je nach Ausstattung die Bezeichnung ESP-XX, wobei XX eine Nummer von 01 bis 12 darstellt. Mit diesen Systemen ist es einfach, schnell ein eingebettetes System zu realisieren und so Sensoren und Aktoren mit dem Internet zu verbinden um Daten zu erfassen oder Systeme zu steuern.

2.1.6 NodeMCU Development Board mit ESP-12

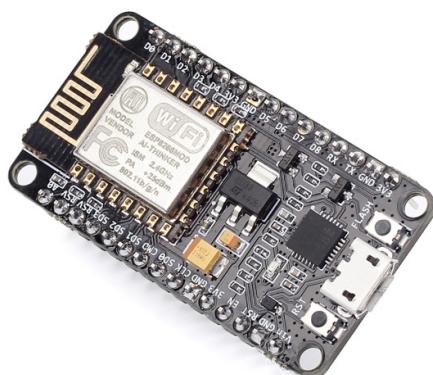


Abbildung 2.7: NodeMCU, Development Board mit ESP-12 [<https://www.exp-tech.de/>]

«*NodeMCU*» ist eine «*Firmware*» für ESP-12 und ein «*Development Kit*». Als Firmware bezeichnet man ein vorgefertigtes Programm für eine MCU, die direkt von der MCU ausgeführt werden kann und bestimmte Funktionen zur Verfügung stellt. Die Firmware von NodeMCU ermöglicht es einen ESP-12 mit der Interpreter-Sprache LUA zu programmieren (⇒ Exkurs 2: *Programm ausführen*, Seite 27).

Als Development Kit bezeichnet man eine Zusammenstellung von Werkzeugen und

Geräten, mit denen man Software für eine Anwendung programmieren kann. Im Fall von NodeMUC ist das Development Kit eine betriebsfertig verdrahtete Hardware ((⇒ Abbildung 2.7: *NodeMCU, Development Board mit ESP-12* [<https://www.exp-tech.de/>], Seite 10)), deren Pläne als OpenSource bereit stehen und die von verschiedenen Herstellern angeboten wird. Auf der Platine ist ein ESP-12 aufgelötet und die Platine kann über Steckerleisten mit Testaufbauten und einem «*Steckbrett*» («*Breadboard*») verbunden werden. Außerdem ist eine USB-Schnittstelle eingebaut, die mit einem USB-zu-UART-Konverter mit der seriellen Schnittstelle des ESP-12 verbunden ist, so dass man den ESP-12 bequem vom PC aus programmieren kann. Als letztes ist auf dem Board auch eine Leistungselektronik verbaut, die aus der 5V-Spannung des USB-Bus die 3,3V-Spannung des ESP-12 bereitstellen, so dass das ganze Ding direkt nach dem Anstecken in den USB-Bus programmiert und verwendet werden kann.

In diesem Kurs verwenden wir ein solches Development Kit und programmieren es mit der weit verbreiteten Entwicklungsumgebung von «*Arduino*» (⇒ Exkurs 1: *Arduino*, Seite 11) der sog. «*Arduino-IDE*». Dabei steht *IDE* für eine *integrated development environment*, also eine integrierte Entwicklungsumgebung. Integriert heißt hier, dass der Texteditor in dem man programmiert, auch Kommandos/Bedienelemente zur Ansteuerung von «*Compilern*» (⇒ Exkurs 2: *Programm ausführen*, Seite 27) und zum Ausführen und Debuggen des Programmes kennt. Im Fall der Arduino-IDE kann die IDE das fertige Programm also per Knopfdruck in Maschinensprache übersetzen und an die MCU übertragen und dort starten.

Exkurs 1: Arduino

Arduino ist eine aus Soft- und Hardware bestehende Plattform zur Entwicklung interaktiver, physische Systeme. Beide Komponenten sind im Sinne von «*Open Source*» quell offen. Die Hardware besteht aus einem einfachen E/A-Board mit einem Mikrocontroller und analogen und digitalen Ein- und Ausgängen. Die Entwicklungsumgebung basiert auf der Programmiersprache Processing und soll auch technisch weniger Versierten den Zugang zur Programmierung und zu Mikrocontrollern erleichtern. Die Programmierung selbst erfolgt in einer C bzw. C++-ähnlichen Programmiersprache, in sog. Sketches. Der «*Quelltext*» wird in Textdateien mit der Dateiendung .ino abgespeichert. Technische Details wie Header-Dateien sind vor den Anwendern weitgehend verborgen. Umfangreiche Bibliotheken und Beispiele vereinfachen den Einstieg in die Programmierung (In Anlehnung an [wikipedia:arduino]).

2.2 Hardwareaufbau #1

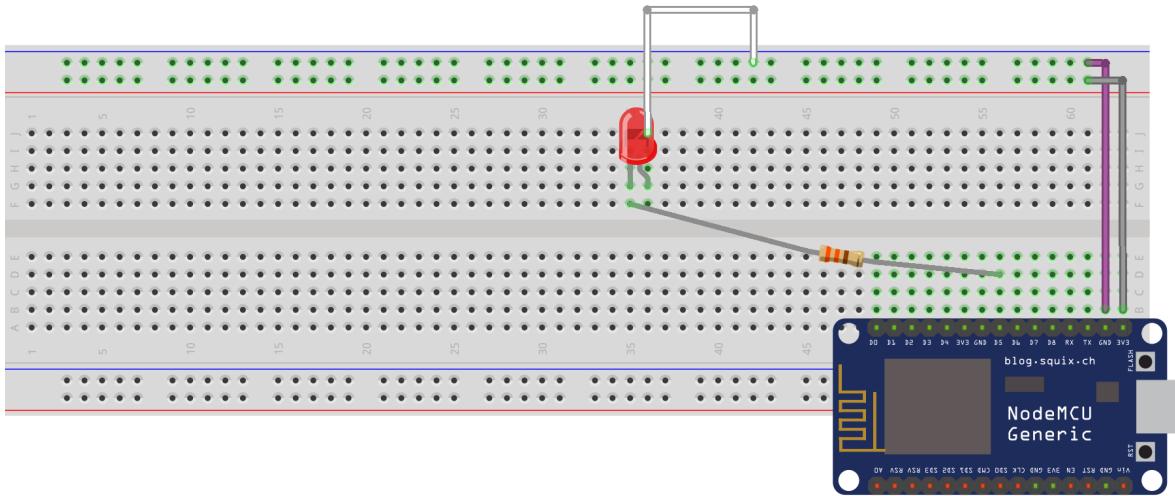


Abbildung 2.8: Schematische Darstellung des ersten Aufbaus

Für die ersten Versuche brauchen wir einen einfachen Aufbau. In Abbildung 2.8: *Schematische Darstellung des ersten Aufbaus* (Seite 12) ist dargestellt, wie der Grundaufbau aussieht. Die Schaltung ist auf einem sog. «*Steckbrett*» («*Breadboard*») aufgesteckt. Mit diesen Boards kann man einfache Schaltungen aufbauen und testen. Das Board hat Spalten und Zeilen mit Steckbuchsen, die das gleiche Lochraster haben, wie Standardplatinen, d.h. die Stecker haben genau den gleichen Abstand, wie die Bohrungen auf einer Platine, so dass man Standardbauteile aufstecken kann. Auf dem Board sind oben und unten Reihen mit Steckbuchsen (rot und blau markiert), die untereinander quer verbunden sind. Diese verwendet man um die Betriebsspannung in der Schaltung zu Verteilen. Die übrigen Stecker sind jeweils Spaltenweise untereinander verbunden. Diese Verbindung ist in der Mitte unterbrochen, so dass dort elektronische Bausteine im sog. «*DIL-Format*» aufgesteckt werden können. Jeder Pin eines solchen Bausteins ist dann mit einer Spalte des Boards verbunden. Wenn man jetzt etwas mit einem

solchen Pin verbinden will, dann braucht man diese nur mit einer Drahtbrücke zu verbinden. In Abbildung 2.9: *Drahtbrücke zum Verbinden der Spalten auf einem Steckbrett* (Seite 13) kann man erkennen, dass diese Drahtbrücken einfache Kabel mit Steckern an den Enden sind. Diese gibt es in verschiedenen Farben.



Widerstand mit der Spalte 35 verbunden. Dazu müssen die Enden des Widerstandes mit der Zange um 90° gebogen und dann in eine Flucht gedreht werden. Zum Einsticken des Widerstandes kann die kleine Zange verwendet werden.

Aufgabe 1: Aufbau 1

Bau die im Schema Abbildung 2.8: Schematische Darstellung des ersten Aufbaus (Seite 12) dargestellte Schaltung auf!

Abbildung 2.9: Drahtbrücke zum Verbinden der Spalten auf einem Steckbrett

Um den Kurs besser nachvollziehen zu können, empfiehlt es sich die Farben aus den schematischen Darstellungen zu übernehmen.

Für den ersten Aufbau wird zunächst das Board so gelegt, dass die blaue Markierung für den negativen Pol oben liegt. Der NodeMCU wird dann vorsichtig unten rechts in die Reihe A ab Spalte 0 eingesteckt. Dann wird die allgemeine Stromversorgung gesteckt, dazu wir die Spalte, die mit dem Pin GND verbunden ist, mit der lila Steckbrücke mit der blauen Reihe ganz oben verbunden. (GND steht für Ground und bezeichnet das untere Potential der Spannung, also den sog. Minuspol.) Analog wird die Spannung des Pins 3v3 mittels einer grauen Steckbrücke mit dem Pluspol (rote Reihe) verbunden.

Dann wird in Reihe G eine LED zwischen die Spalten 35 und 36 eingesteckt. Dabei muss der längere Anschluss in Spalte 35 eingesteckt werden. Jetzt können die beiden Anschlüsse der LED über die Spalte 35 und 36 angeschlossen werden. Die Spalte 36 wird mit einer weißen Steckbrücke mit dem Minuspol der Spannung (blaue Reihe) verbunden. Die LED soll über den Pin D5 angesteuert werden, dazu wird dieser Pin (Spalte 7) mit einem 220Ω -

2.3 Installation der Software

2.3.1 Installation der IDE

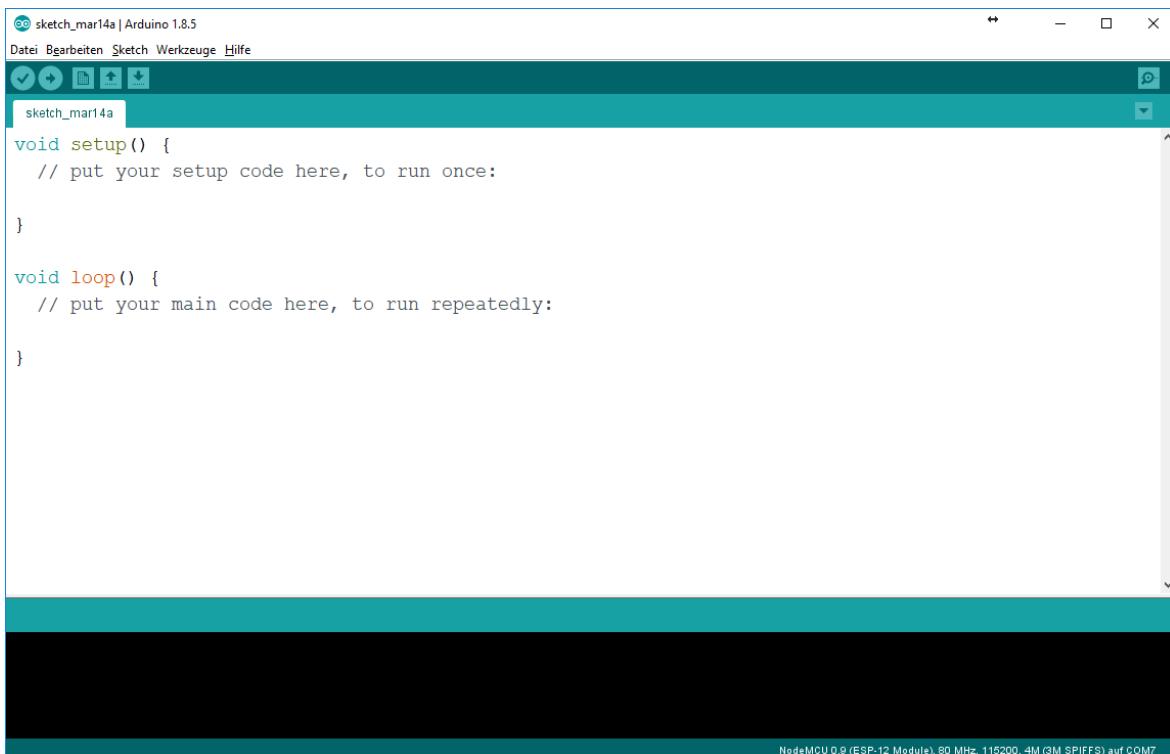


Abbildung 2.10: Arduino-IDE mit neuer Sketch

Zum Programmieren der Schaltung verwenden wir die Arduino IDE, die unter <https://www.arduino.cc/en/Main/Software>, Kurzlink: [ftq](#) heruntergeladen werden kann. Die IDE muss nicht installiert werden, es reicht die ZIP-Datei in einen Ordner auf der Festplatte zu entpacken. Und die Datei `arduino.exe` per Doppelklick zu starten. Abbildung 2.10: *Arduino-IDE mit neuer Sketch* (Seite 14) zeigt die gestartete IDE mit einem neuen Programm. Die Programme heißen auf der Arduino-Plattform Sketches. (Sollte beim Starten keine neue Sketch erstellt worden sein, kann diese über die Menüpunkte `Datei` → `Neu` (`Strg`+`n`) erstellt werden.

Die Bedeutung des angezeigten Quelltextes wird in späteren Abschnitten erläutert, zunächst soll die aktuelle Schaltung in Betrieb

genommen werden. Dazu muss zunächst die IDE so konfiguriert werden, dass sie mit dem ESP-12 auf dem NodeMCU-Board kommunizieren kann. Dann wird ein Beispielprogramm aus dem gitHUB-Repository geöffnet und auf der Plattform gestartet.

2.3.2 Verbinden der IDE zum NodeMCU

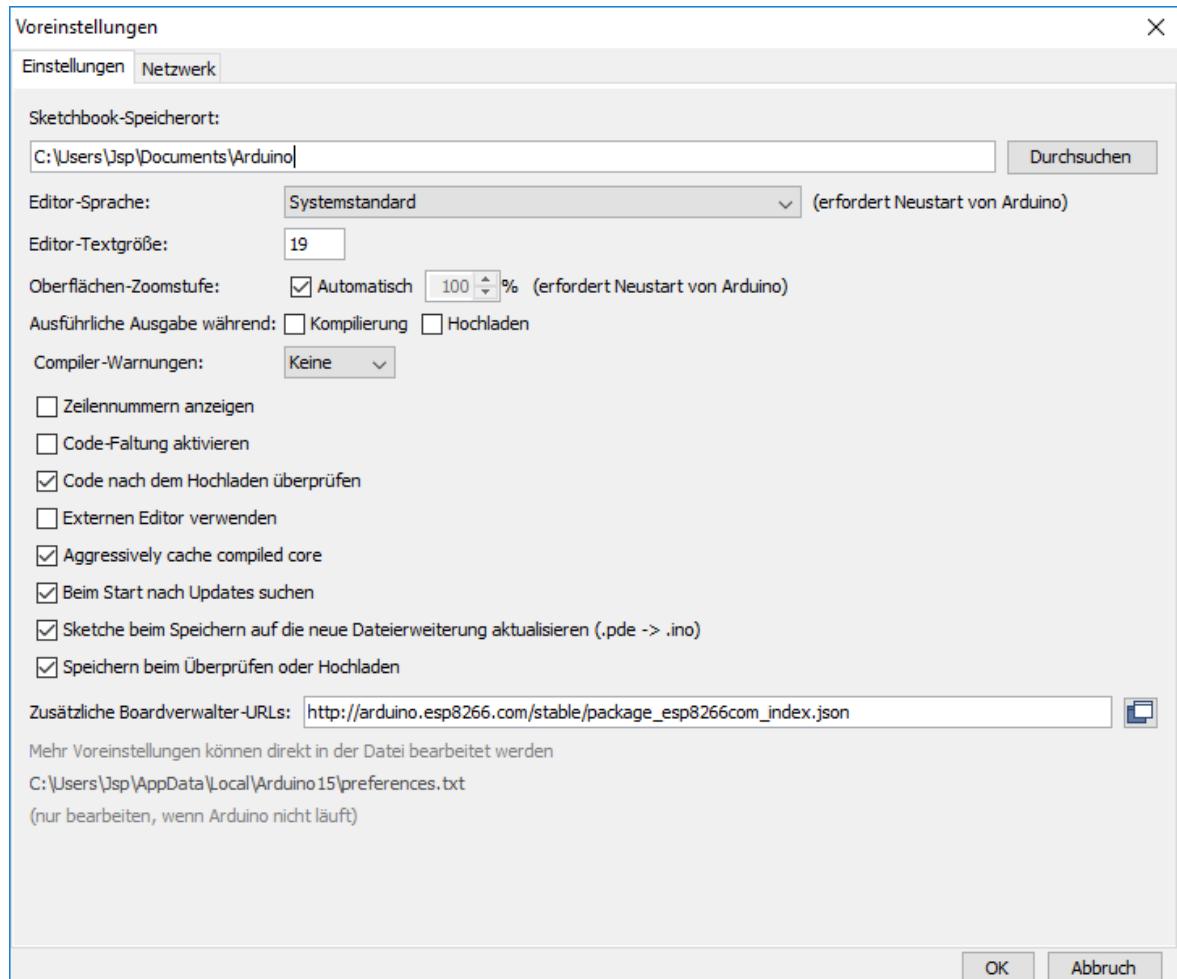


Abbildung 2.11: Dialog zur Konfiguration der IDE

Die IDE steuert den Compiler, so dass verschiedene Zielplattformen mit der gleichen IDE programmiert werden können, wenn man der IDE die Compiler für diese Plattformen bekannt macht. Das erledigt man über den sog. Board-Verwalter, der dafür sorgt, dass die passenden Programme heruntergeladen und konfiguriert werden. Dazu wird die Konfiguration aus dem Internet geladen. Über die Menüpunkte **Datei** →

Voreinstellungen (**Strg**+**,**) gelangt man zum Einstellungsdialog (⇒ Abbildung 2.11: *Dialog zur Konfiguration der IDE*, Seite 15). Im Feld **Zusätzliche Boardverwalter-URLs:** wird die URL für die Konfiguration eingegeben, in diesem Falle ist das http://arduino.esp8266.com/stable/package_esp8266com_index.json, Kurzlink: [hjk](#). Der Einstellungsdialog wird dann mit **Ok** geschlossen.

2 Inbetriebnahme

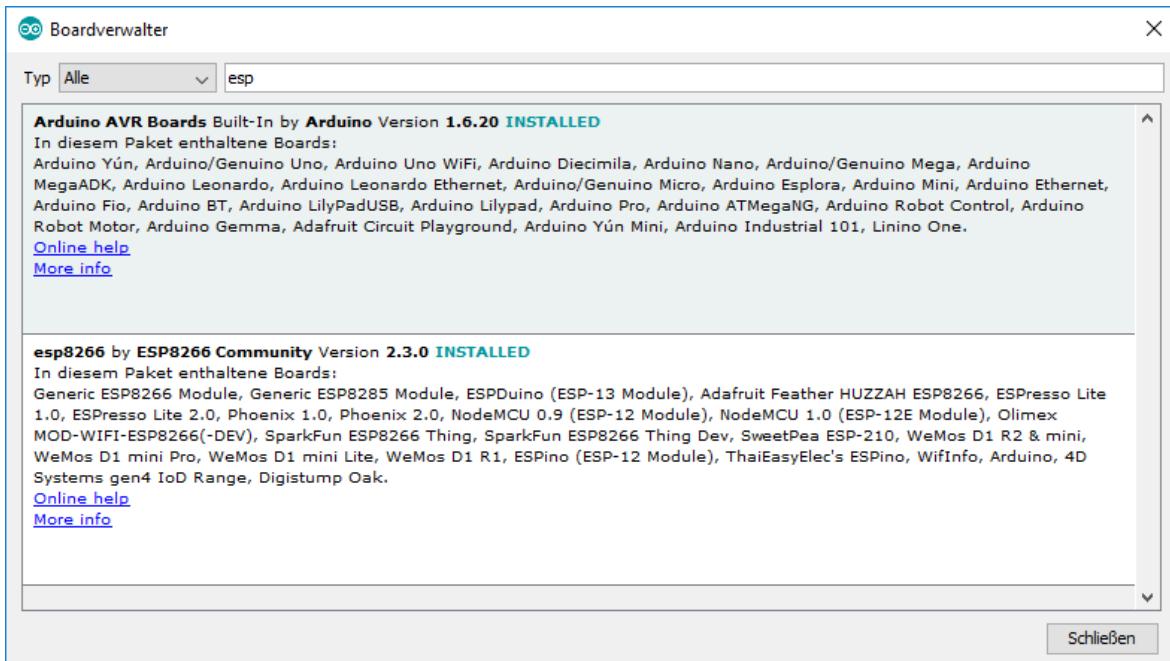


Abbildung 2.12: Boardverwalter: Dialog zur Installation der Einstellungen für zusätzliche Boards

Unter **Werkzeuge** → **Board** → **Boardverwalter...** kann man den eigentlichen Boardverwalter aufrufen (⇒ Abbildung 2.12: *Boardverwalter: Dialog zur Installation der Einstellungen für zusätzliche Boards*, Seite 16). Wenn man im Suchfeld 'esp' eingibt, wird die Liste entsprechend eingegrenzt und so wird u.a. **esp8266 by ESP8266 community** angezeigt. Über die Schaltfläche **Install** wird diese installiert. Der Boardmanager lädt nun alle notwendigen Datei herunter und installiert das Board in der IDE. Nach der Installation können wir das Board mit dem USB-Kabel am Rechner anschließen. Das Board wird über einen «*UART*» programmiert. An PCs heißen diese Anschlüsse «*serielle Ports*» und werden unter Windows als COM-Schnittstelle bezeichnet. Die einzelnen Schnittstellen heißen COM1, COM2, COM3 ... Wenn wir den NodeMCU über den UART-USB-Konverter am PC anschließen, wird ein Treiber installiert, der eine virtuelle COM-Schnittstelle einrichtet, d.h. die Schnittstelle wird nur simuliert; es ist ja in der Regel keine RS232-Schnittstelle mehr am PC verbaut.

Deshalb nimmt man den USB-Stecker und simuliert die Schnittstelle als sog. «*virtuelle Schnittstelle*». Alles, was ein Programm auf dem PC nun über diese virtuelle Schnittstelle schickt, landet über die USB-Verbindung beim UART-USB-Konverter und von dort am UART des ESP-12 und umgekehrt kann ein Programm, dass die virtuelle Schnittstelle ausliest, alles empfangen, was der ESP-12 über den UART versendet, weil der UART-USB-Konverter es umsetzt und es über die USB-Verbindung zum PC geschickt und dort vom Treiber auf die virtuelle serielle Schnittstelle umgesetzt wird. Die IDE kann jetzt also mit einer COM-Schnittstelle sprechen und das ganze verhält sich so, als wäre der ESP-12 direkt mit dem PC verbunden. Damit das funktioniert, müssen jetzt noch die COM-Schnittstelle und das richtige Board ausgewählt werden. Die Schnittstelle wählt man über **Werkzeuge** → **Port** aus. Hier werden in der Regel zwei Schnittstellen angezeigt, weil Windows schon eine virtuelle Schnittstelle mit der Bezeichnung COM1 eingebaut hat. Die andere Schnittstelle ist in der Regel die

neue virtuelle Schnittstelle zu unserem Board. Unter **Werkzeuge** → **Board** muss jetzt noch **NodeMCU 0.9 (ESP-12 Module)** ausgewählt werden, so dass die IDE die passenden Befehle für dieses Board erzeugt.

den. Mit dem Befehl **Sketch** → **Hochladen** (**Strg**+**U**) wird die Sketch kompliert und zum NodeMCU übertragen. Wenn das erfolgreich war, dann blinkt die LED jetzt immer für eine Sekunde (eine Sekunde an, eine Sekunde aus, ...).

2.4 Das erste Programm starten

Um zu prüfen, ob der Aufbau funktioniert, wird ein Beispielprogramm geladen und ausgeführt. Über **Datei** → **Öffnen...** kann man das Beispiel **gitHub:myBlink.ino** in die IDE la-

Aufgabe 2: Starten

Starte das Programm **gitHub:myBlink.ino** auf Deinem Aufbau.

3 Eine (sehr) kurze Einführung in die Programmierung

In der digitalen Welt werden verschiedene Geräte miteinander gekoppelt und können so miteinander interagieren. Also z.B. der sog. Intelligente Kühlschrank, der feststellt, dass die Milch leer oder schlecht ist und dann eine entsprechende Nachricht verschickt. Dazu müssen diese Geräte ein Verhalten haben und das muss programmiert werden. Warum das digital geschieht und ob alles, was so ein einfaches Verhalten hat, auch intelligent oder smart ist, wird in den späteren Kapiteln noch behandelt. Zum Einstieg folgt hier eine Einführung in die Programmierung, die sich auf die Teile beschränkt, die wir brauchen um erste Sketches (so heißen bei Arduino die Programme) schreiben zu können. Wenn im Laufe des Kurses noch weitere Elemente gebraucht werden, werden diese als Exkurs eingeführt. Wer gleich loslegen will, kann auch mit Abschnitt 5: *Ein eigenes Programm schreiben* (Seite 37) beginnen und später nochmal hier nachschauen, wenn etwas unklar ist.

3.1 Algorithmus, Variable und Fluss

Zu Beginn ein Hinweis: Ein Computer besitzt keine Intelligenz, er ist eine einfache Maschine, die Hintereinander eine Reihe von Anweisungen, die Befehle genannt werden, abarbeitet. Das wiederum machen die heutigen Rechner sehr schnell, so dass man durch geschickte Programmierung sehr komplexes Verhalten erzeugen kann.

3.1.1 Beispiel TurtleBot



Abbildung 3.1: Ein einfacher TurtleBot aus Lego

Ein einfaches Beispiel hierfür ist ein sog. Turtle-Bot.

In Abbildung 3.1: *Ein einfacher TurtleBot aus Lego* (Seite 19) kann man ein Beispiel eines solchen Roboters sehen. Der Roboter kann seine Räder vorwärts und rückwärts drehen und hat einen Sensor, der ausgelöst wird, wenn der Roboter auf ein Hindernis stößt.

Wenn ein solcher Roboter den Ausgang aus einem rechteckigen Raum finden soll, dann könnte man das Programmieren, indem man dem Roboter die folgenden Anweisungen gibt:

- Fahre Vorwärts (beide Räder vorwärts), bis Du auf ein Hindernis stößt (Sensor löst aus)
- Fahre kurz Rückwärts (beide Räder Rückwärts)
- Drehe dich etwas (rechtes Rad vorwärts, linkes Rad rückwärts)
- Fange vorne wieder an

Mit diesen Anweisungen würde der Roboter den Ausgang aus einem rechteckigen Raum finden. Wie man in Abbildung 3.2: Beispiele für den Fahrweg eines einfachen TurtleBots beim Verlassen eines Raumes (Seite 20) beispielhaft sieht, ist der Weg nicht der direkte oder besonders elegant, aber im Rahmen seiner Möglichkeiten löst er das Problem. (In diesem einfachen Beispiel allerdings ohne es zu merken, weil es keinen Anweisung zum Aufhören gibt.)

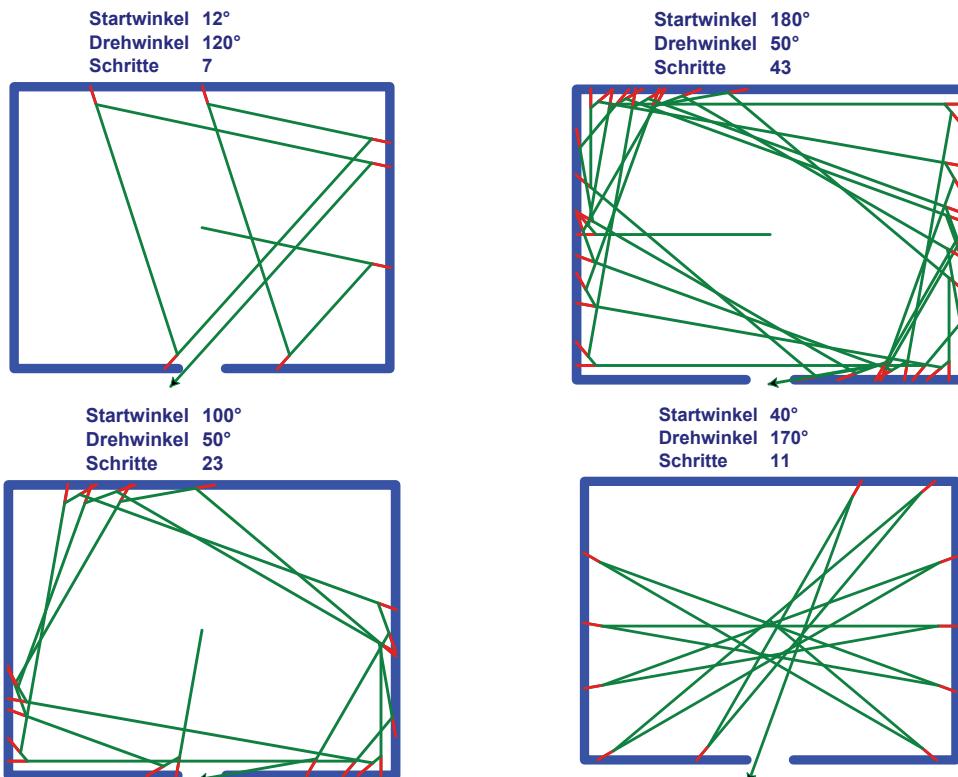


Abbildung 3.2: Beispiele für den Fahrweg eines einfachen TurtleBots beim Verlassen eines Raumes

Das einfache Beispiel zeigt, ein grundsätzliches Konzept der Programmierung, das bei der Abarbeitung der Befehle,

also dem Ablauf oder Fluss des Programmes wichtig ist, die sog. «*Flusskontrolle*». Der Roboter könnte das Problem nicht allgemein

(also für alle rechteckigen Räume) lösen, wenn wir ihm Anweisungen in der Form:

- Fahre 20cm vorwärts
- Drehe um 12 Grad nach rechts
- Fahre 40cm vorwärts
- ...

geben würden, so würde er ja nur von einer bestimmten Stelle in einem bestimmten Raum den Ausgang finden. Statt dessen haben wir vorgegeben, dass etwas getan werden muss, wenn etwas anderes eingetreten ist und wir haben vorgegeben, dass das Ganze wiederholt werden muss. Das sind die zwei Elemente der Flusskontrolle: «*Verzweigungen* » und «*Schleifen* ». Für beide gilt: Wir brauchen eine Bedingung, also eine Aussage, die **wahr** oder **falsch** ist, also in diesem Fall: Zeigt der Sensor an, dass ein Hindernis erreicht wurde? Wenn das **wahr** ist, wird die nächste Anweisung (Rückwärtssfahren) ausgeführt, wenn nicht, dann bleibt es beim Vorwärtssfahren. Eine solche Auflistung von Verhaltensweisen also vorgeschriebenen Handlungen nennt man auch eine Handlungsvorschrift und eine Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen (also ähnlichen Problemen) nennt man «*Algorithmus* ». Damit ein Computer so einen Algorithmus ausführen kann, muss dieser eindeutig beschrieben sein. Zur Beschreibung eines Algorithmus verwendet man sog. «*Höhere Programmiersprachen* », das sind Sprachen, die mit den von Menschen lesbaren Buchstaben geschrieben werden und einem sehr stark vorgegeben Aufbau folgen. Diesen nennt man «*formale Syntax* ». Meistens wird der Programmcode einfach in Textdateien abgespeichert und dann mit einem Compiler oder Interpreter eingelesen (⇒ Exkurs 2: *Programm ausführen*, Seite 27).

Aussagen, die in einer solchen Sprache formuliert sind, sind eindeutig und lassen keinen Interpretationsspielraum zu. Daher sind

diese Sprachen gut zum eindeutigen Beschreiben von Vorgängen geeignet. Wie in gesprochenen Sprachen werden dazu bestimmten Wörtern Bedeutungen zugeordnet. Lernt man eine Fremdsprache, muss man daher Vokabeln lernen. In Programmiersprachen nennt man diese Wörter «*reservierte Wörter* » oder «*Schlüsselwörter* ». Diese müssen ähnlich wie Vokabeln gelernt werden, es sind aber sehr viel weniger. In C++ gibt es 52 Schlüsselwörter während der Grundwortschatz Englisch für Level A2 (Elementare Sprachverwendung) schon 4000 Worte und Redewendungen umfasst.

3.1.2 Beispiel: Größter gemeinsamer Teiler

Neben den Elementen zur Flusskontrolle und der dazu notwendigen Beschreibung von Bedingungen, enthalten diese Programmiersprachen auch noch Elemente zur Beschreibung von Rechnungen. Dazu ein weiteres Beispiel: Ein sehr alter und bekannter Algorithmus ist der euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen. Der größte gemeinsame Teiler zweier ganzer Zahlen ist die größte natürliche Zahl, durch die beide Zahlen ohne Rest teilbar sind. Zur Herleitung stellen wir uns die beiden Zahlen als Stäbe vor, die Länge des Stabes repräsentiert dabei jeweils den Wert der Zahl:

Zahl1

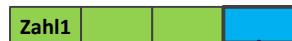
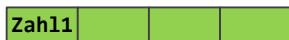
Zahl2

Dann müssten beide Zahlen ohne Reststück in kürzere, gleichlange Klötzchen zerlegbar sein:



Wir suchen jetzt also die Länge des größtmöglichen Klötzchens in das beide Stäbe zerlegbar sind.

Schaut man sich die folgende Skizze an, dann sieht man, dass nicht nur die beiden Zahlen, sondern auch das Stück, das übersteht, aus dem gesuchten Klötzchen zusammensetzbare sind.



Übertragen auf den Zahlenwert, ist dieser Überstand die Differenz zwischen den beiden Zahlen, also das, was rauskommt, wenn man von der größeren Zahl die kleinere Zahl abzieht.



Wenn man jetzt mit der kleineren Zahl und diesem Reststück weiterarbeitet, bleibt irgendwann als Überstand genau das gesuchte Klötzchen übrig:

Wenn beide Reststücke gleich groß sind, dann hat man den ggT gefunden.

Die Vorschrift zur Berechnung des ggT könnte also lauten:

1. Sind beide Zahlen gleich, dann ist dies der ggT.
2. Ziehe die kleinere von der größeren Zahl ab!
3. Verwende das Ergebnis und die kleinere Zahl als Zahlen!
4. Fange von vorne an!

Man erkennt sofort die Flusskontrolle mit der Schleife und der Beschreibung einer Be-

dingung dafür, dass der ggT gefunden worden ist. Diese Bedingung nennt man «*Abbruchbedingung einer Schleife.*»

Zur besseren Übersicht kann man diesen Algorithmus in einem Ablaufplan darstellen. Abbildung 3.3: *Ablaufplan des Algorithmus* (Seite 23) zeigt diesen Algorithmus als Ablaufplan. Jede Anweisung ist dabei in einem Kästchen eingetragen. Verzweigungen sind als Raute dargestellt und Rechenanweisungen als Rechtecke. Start und Ende sind an der Seite rund. Die Schleife ergibt sich daraus, dass von der untersten Anweisung aus der Pfeil wieder zur ersten Anweisung führt.

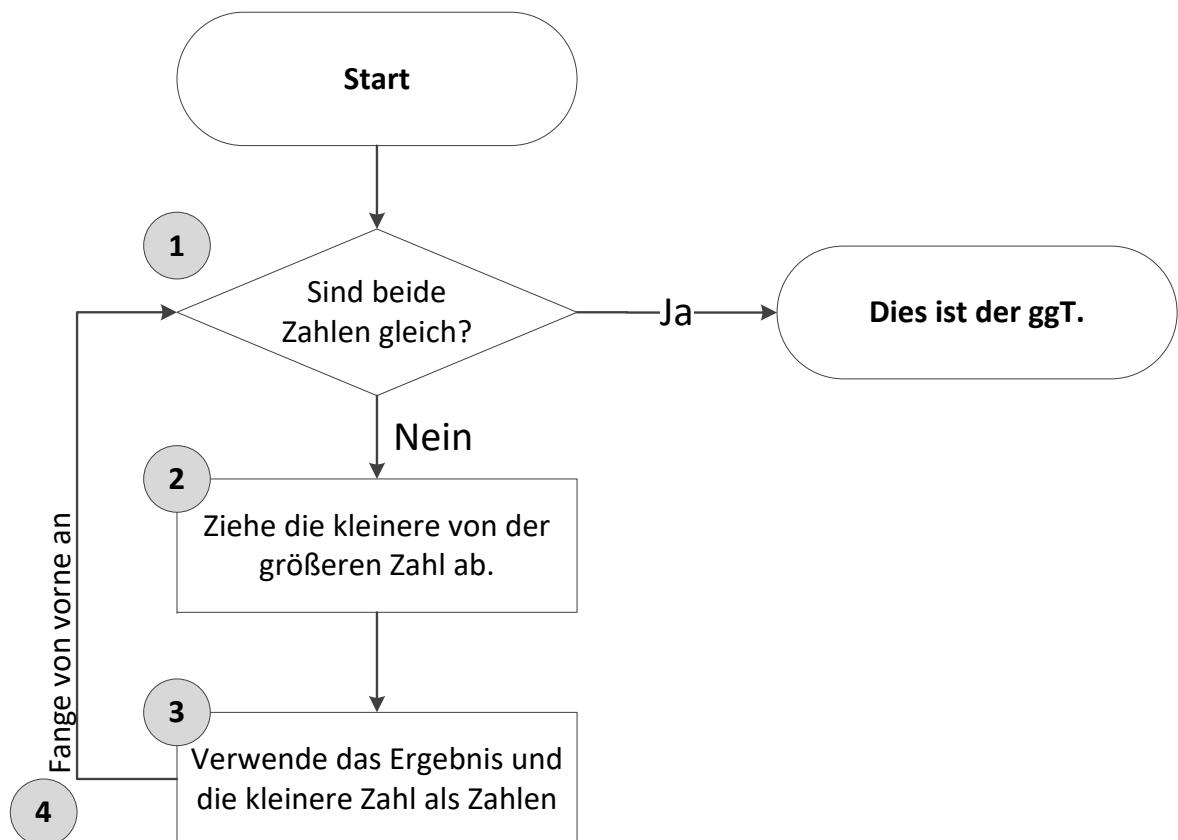


Abbildung 3.3: Ablaufplan des Algorithmus

Im Gegensatz zum ersten Beispiel wird hier gerechnet (Schritt ②). Wenn wir das mit unserer Intelligenz im Kopf machen, dann passiert das ganz automatisch. Ein Computer kann das nicht, der kann nur eindeutig be-

schriebene Algorithmen abarbeiten und dazu müssen die Werte, mit denen gerechnet wird, gespeichert werden. Dies geschieht in sog. Variablen und Parametern, die genauso funktionieren, wie man es aus der Mathe-

matik kennt. Wenn es dort eine Funktion in der Form $f(x) = x + 42$ gibt, dann rechnet man den Wert der Funktion für eine bestimmte Stelle z.B. $f(2)$ aus, indem man für die Variable x eine 2 einsetzt, dann nach der Vorschrift die 42 addiert und so den Funktionswert 44 erhält. Der Mathematiker schreibt dann $f(2) = 44$ und sagt: "f von 2 ist 44". Da das x in der Klammer hinter dem Funktionsnamen steht, bezeichnet man es als Parameter. Variablen sind also Platzhalter für Zahlen und Parameter sind Variablen, deren Wert von außen vorgegeben wird. In der Mathematik nennt man das: *Einen Wert in eine Funktion einsetzen* und bei der Programmierung sagt man: *Der Funktion wird ein Parameter übergeben*. Außerdem können beim Programmieren nicht nur Zahlen, sondern auch z.B. Zeichenketten vorkommen, so dass man nicht mehr vom Platzhalter für Zahlen, sondern allgemein für Daten spricht. Eine Variable ist also ein Platzhalter für ein Datum und hat in einer höheren Programmiersprache einen Namen.

Schauen wir uns die Vorschrift nochmal an:

1. Sind **beide Zahlen** gleich, dann ist dies der ggT.
2. Ziehe die **kleinere** von der **größeren** Zahl ab!
3. Verwende **das Ergebnis** und die **kleinere Zahl** als Zahlen!
4. Fange von vorne an!

Jetzt sieht man, dass es hier um mehrere Zahlen geht und dass diese hier über ihre Eigenschaft beschrieben werden (Ergebnis, kleinere und größere Zahl). Für das Schreiben eines Algorithmus müssen wir jetzt Variablen und Parameter mit Namen einführen, so dass wir damit rechnen können. Das nennt man *Deklaration einer Variablen*. Dazu gehört immer der Name der Variablen und in den meisten Programmiersprachen auch welche Art von Datum da abgespeichert werden soll. In diesem Beispiel also ganze Zahlen.

In Abbildung 3.4: *Ablaufplan des Algorithmus mit Parametern und Variablen* (Seite 25)

wurden für die beiden Zahlen die Parameter A und B eingeführt. Außerdem gibt es noch eine Variable E für das Zwischenergebnis. Die erste Verzweigung ① ist schon bekannt, hier wird geprüft ob beide Zahlen gleich sind. Dazu wird jetzt nur noch die Bedingung in die Rauten geschrieben. A==B bedeutet, dass die Werte der Variablen verglichen werden und die Bedingung *wahr* ist, wenn diese gleich sind. Die nächst Verzweigung 2 ist notwendig, weil der Computer eine exakte Beschreibung des Vorgehens braucht und wir bei der Aussage: *Ziehe die kleinere von der größeren Zahl ab!* im Kopf zwei Dinge machen: Zuerst bestimmen wir die größere Zahl und dann ziehen wir davon die andere ab. In ② wird also erstmal geprüft, ob A größer als B ist; die Bedingung dafür lautet A>B.

Wenn das der Fall ist, wird bei ③a weitergemacht. Aus der Anweisung *Ziehe die kleinere von der größeren Zahl ab!* wird jetzt A-B, da wir gerade festgestellt haben, dass A die größere Zahl sein muss. In ③a sieht man eine sog. Zuweisung E=A-B bedeutet, dass der Wert, der sich aus der Rechnung A-B ergibt in die Variable E gespeichert wird. Man sagt: *Der Wert von A-B wird der Variable E zugewiesen*.

Bitte darauf achten, dass immer von 'rechts nach links' zugewiesen wird, d.h. das Ziel steht links vom Gleichzeichen.

Danach wird in ③b der Wert aus der Variablen E der Variable A zugewiesen. Das A steht links. Dieser Schritt ist die Umsetzung der Anweisung: *Verwende das Ergebnis und die kleinere Zahl als Zahlen!* Da es im nächsten Schritt von vorne losgehen soll, müssen also jetzt die kleinere Zahl und das Ergebnis in den Variablen A und B landen. Und da B hier die kleinere Zahl ist, speichern wir einfach das Ergebnis der Subtraktion aus der Variable E in die Variable A um dann mit den beiden neuen Zahlen wieder von vorne zu beginnen ⑤. Für den Fall, dass sich in ② ergibt, dass B die größere Zahl ist, werden die Schritte ③b und ④b ausgeführt. Es wird also A von B abgezogen.

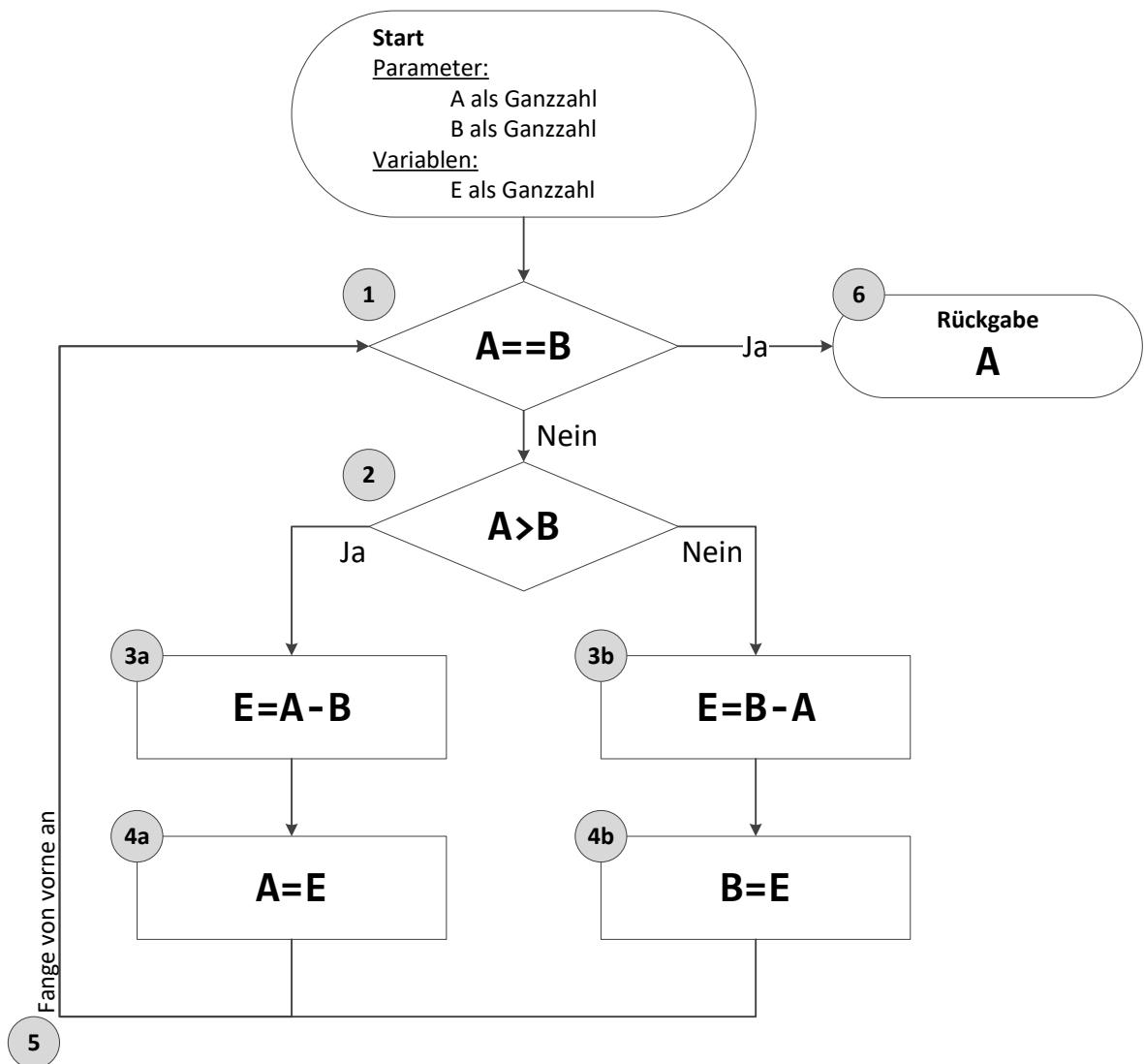


Abbildung 3.4: Ablaufplan des Algorithmus mit Parametern und Variablen

3.2 Zusammenfassung

In den letzten Abschnitten wurde dargestellt, dass man zum Programmieren nur die folgenden Elemente braucht:

- Flusskontrolle
 - Verzweigung
 - Schleife
 - Eine Möglichkeit Bedingungen eindeutig zu formulieren
- Zuweisungen
 - Deklaration von Variablen
 - Eine Möglichkeit Rechenterme eindeutig zu formulieren

Damit man in seinen Programmen die Übersicht behält, stellen die meisten Programmiersprachen noch Elemente zur Beschreibung von Unterprogrammen bereit. Diese Unterprogramme können ein Ergebnis zurückgeben, so wie es im vorangegangenen Beispiel zum ggT zu sehen ist. Folgende zusätzliche Elemente sind dazu notwendig:

- Flusskontrolle (Unterprogramme)
 - Deklaration des Namens eines Unterprogramms
 - Deklaration von Parametern zur Übergabe von Daten an das Unterprogramm (⇒ Abbildung 3.4: *Ablaufplan des Algorithmus mit Parametern und Variablen*, Seite 25)
 - Start
 - Deklaration der Art der Rückgabe (z.B. eine Zahl)
 - Ausführen der Rückgabe (⇒ Abbildung 3.4: *Ablaufplan des Algorithmus mit Parametern und Variablen*, Seite 25) (6)
- Zuweisungen (Unterprogramm)
 - Aufruf eines Unterprogramms

Da das Ausführen eines Unterprogramms ähnlich wie eine Schleife oder Verzweigung den Ablauf des Programms steuert, gehören diese Elemente zur **Flusskontrolle**. Da ein Unterprogramm genau wie eine Berechnung ein Ergebnis zurückgibt und dieses in der Regel in einer Variablen gespeichert werden soll, sind Aufrufe von Unterprogrammen in den meisten Programmiersprachen als **Zuweisungen** dargestellt.

Anmerkung

Neben der hier vorgestellten ablauforientierten Darstellung von Programmen, gibt es noch andere, die für bestimmte Aufgabenstellungen einfacher oder übersichtlicher sind. In diesem Kurs wird z.B. noch die objekt-orientierte Programmierung vorgestellt. Es gibt aber auch Programmiersprachen, die nicht als Text sondern graphisch definiert sind. Allen gemeinsam ist, dass zur Ausführung auf dem Rechner am Ende ein Programm erzeugt wird, bei dem Anweisungen nacheinander abgearbeitet werden.

Exkurs 2: Programm ausführen

Dieser Exkurs soll einen kurzen Überblick darüber geben, wie Programme auf einem Computer ausgeführt werden. Dazu braucht man ein paar Begriffsdefinitionen und deren Zusammenhänge:

Definition 1: Programm

Als Programm bezeichnet man die Beschreibung eines Algorithmus oder eine Reihe von Algorithmen in einer formal beschriebenen Sprache (Programmiersprache), die dazu dienen eine bestimmte Aufgabe zu unterstützen, wenn sie auf geeigneten Rechnern ausgeführt werden.

Definition 2: Maschinenbefehl

Jeder Prozessor verfügt über eigene Befehle, die er abarbeiten kann. Diese nennt man auch Maschinenbefehle. Die Menge aller Befehle, die ein Prozessor verarbeiten kann, nennt man «*Befehlssatz*». Die Programmiersprache des Prozessors, die aus diesen Befehlen zusammengesetzt ist, nennt man auch «*Maschinensprache*», «*Maschinencode*» oder «*nativen Code*».

Üblicherweise werden die Befehle im Arbeitsspeicher abgelegt und hintereinander abgearbeitet. Es sei denn, der Befehl ist ein Sprung, der angibt, dass woanders weitergearbeitet werden soll.

Beispiel für diese Befehle sind:

- Lesen von Daten aus dem Arbeitsspeicher
- Schreiben von Daten in den Arbeitsspeicher
- Ausführung elementarer arithmetischer und logischer Operationen
 - Addition
 - Subtraktion
 - Und-Verknüpfung
 - ...
- Sprung (Fortsetzung der Verarbeitung an einer bestimmten anderen Stelle)

Definition 3: Programmiersprache

Allgemein bezeichnet man Sprachen zur Formulierung von Rechenvorschriften, die von einem Computer ausgeführt (interpretiert) werden können, als «*Programmiersprache*». Auch die Maschinensprache ist also eine Programmiersprache. Leider ist sie für den Menschen sehr schwer verständlich, weil die Befehle sehr kleine Verarbeitungsschritte beschreiben. Deshalb hat man die sog. «*Höheren Programmiersprachen*» entwickelt. Diese werden auch als problemorientierte Programmiersprachen bezeichnet und mit ihnen lassen sich Programme auf höherem (also abstrakterem) Niveau als Maschinensprache beschreiben. Sie sind für Menschen einfacher verständlich. Allerdings braucht man zunächst ein weiteres Programm, dass die in der höheren Programmiersprache geschriebenen Programme interpretiert und umsetzt.

Es gibt zwei Arten der Umsetzung: «*Interpreter*» und «*Compiler*»:

Interpreter Umsetzer, der schrittweise nur einen Programmschritt nach dem anderen interpretiert und dann direkt ausführt

Compiler Übersetzer, der ein komplettes Programm in Maschinensprache übersetzt und nicht zur Ausführung bringt (z.B.: ein test.exe erzeugt)

4 Programmieren mit Arduino

Arduino ist eine Hard- und Softwareplattform, die als Open-Source angeboten wird (⇒ Exkurs 1: *Arduino*, Seite 11). Dazu gehört auch eine leicht zu erlernende Programmiersprache *Processing* in der die Programme sog. Sketches geschrieben werden können. Diese Sprache ist sehr stark an die Programmiersprache C++ angelehnt, ist aber etwas vereinfacht, um den Einstieg zu erleichtern. Beide Sprachen können innerhalb eines Projektes gemischt werden. In diesem Kapitel wird dargestellt, wie die notwendigen Elemente, die in den vorhergehenden Abschnitten dargestellt sind, in einer Sketch geschrieben werden.

4.1 Basics

4.1.1 Deklaration von Variablen

```
int A;
```

Bei der Deklaration einer Variablen muss zunächst angegeben werden, um welche Art von Datum es sich handelt. Man spricht hier vom Datentyp. Als erstes wird der Typ angegeben und dann der Name. In der ersten Zeile des Beispiels wird eine Variable für Ganzzahlen mit dem Namen A deklariert. Der Typ für Ganzzahlen heißt Integer und wird mit dem Schlüsselwort int bezeichnet. Man sagt: *Die Variable A ist vom Typ Integer*. Abgeschlossen wird die Deklaration mit einem Semikolon. Im Flussdiagramm sind die Anweisungen dadurch voneinander abgegrenzt, dass jede in einem eigenen Kästchen notiert ist. In einer Sketch wird jede Anweisung durch ein Semikolon abgeschlossen.

4.1.2 Zuweisungen

Wie schon beschrieben sind die Zuweisungen der Teil eines Programms, in dem gerechnet wird. Dazu muss man Berechnungen eindeutig formulieren können. Dazu sind bestimmte Zeichen definiert und einer Funktionen zugeordnet. Man spricht von «*Operatoren*». Es gibt «*Rechenoperatoren*», «*Zuweisungsoperatoren*» und «*Vergleichsoperatoren*». In der Regel sind diese an die Darstellung der Operatoren aus der Mathematik angelehnt. So ist der Zuweisungsoperator in einer Sketch das =, im Unterschied zum Vergleichsoperator ==, der Vergleichsoperator *größer als oder gleich* ≥ wird als >= dargestellt und Addition und Subtraktion sind mit + und - definiert.

```
int A;  
A = 5;  
int B;  
B = A + 4;
```

Im Beispiel sieht man zunächst die Deklaration der Variablen A vom Typ Integer und dann eine einfache Zuweisung. Es wird der Wert 5 zugewiesen. Steht eine Zahl direkt im Quelltext, spricht man von einer «*Konstante*». In der folgenden Zeile wird die Variable B vom Typ Integer deklariert und in der letzten Zeile wird gerechnet. Bei Ausführung des Programms wird erst der rechte Teil ausgewertet. Dort sind die Variable A und die Konstante 4 mit dem Additionsoperator + verknüpft, so dass sich eine 9 als Wert ergibt. Dieser Wert wird mit den Zuweisungsoperator = der Variable B zugewiesen.

4.1.3 Initialisierung von Variablen

```
int B = 5;  
String message =  
    "Dies ist ein Beispiel. ";
```

Die Deklaration einer Variablen kann auch gleich mit einer Zuweisung verbunden werden. Man spricht dann von einer «Initialisierung». Die erste Zeile liest man so: *Die Variable B ist vom Typ Integer und wird mit 5 initialisiert.* Oft benötigt man für eine Berechnung einen bestimmten Startwert in einer Variablen, zur Vermeidung von Programmierfehlern sollten daher alle Variablen gleich bei der Deklaration initialisiert werden. Die zweite Zeile zeigt die Deklaration und Initialisierung einer Variable `message`. Diese ist vom Typ `String`. Das ist der Typ zum Speichern von Text und wird im Deutschen oft als «Zeilchenkette» bezeichnet.

4.1.4 Kommentare

Wie man sieht, ist die Notation der Sprache eindeutig aber für den menschlichen Betrachter nicht sehr eingängig. Daher gibt es die Möglichkeit im Quelltext Kommentare einzufügen.

```
int A = 35; // Dies ist die erste Zahl  
int B = 25; // Dies ist die zweite Zahl
```

Fügt man in den Text zwei Schrägstriche // ein, dann wird der Rest der Zeile als Kommentar behandelt. Hier kann man dann Erläuterungen einfügen. Die nächste Zeile ist dann wieder normaler Quelltext.

```
int A = 35;  
/* Weil er mehrere Zeilen  
umfasst, ist dies ein  
ein mehrzeiliger Kommentar  
*/
```

Fügt man in den Text einen Schrägstrich gefolgt von einem Stern /* ein, dann wird alles als Kommentar behandelt. Bis das nächste mal ein Stern gefolgt von einem Schrägstrich */ im Code auftaucht.

4.2 Flusskontrolle

4.2.1 Bedingungen

Für die Flusskontrolle mit Verzweigungen und Schleifen müssen Bedingungen formuliert werden können. Dazu gibt es in einer Sketch die Vergleichsoperatoren, z.B.

```
A == B      // Ist gleich
A > B      // Ist größer als
A < B      // Ist kleiner als
A >= B     // Ist größer oder gleich
A <= B     // Ist kleiner oder gleich
A != B     // Ist ungleich
```

4.2.2 Verzweigung

Eine Verzweigung wird in einer Sketch mit dem Schlüsselwort **if** eingeleitet. Diesem folgt in runden Klammern ((...)) die Bedingung. Im folgenden Beispiel ist die Bedingung **A>B**:

```
// Verzweigung, abhängig von Bedingung:
// Ist A größer B?
if (A > B)
{
    // Block der ausgeführt wird
    // wenn A größer B
    E = A - B;
    A = E;
}
else
{
    // Block der ausgeführt wird
    // wenn A NICHT größer B
    E = B - A;
    B = E;
}
```

Im Beispiel werden die Schritte (2) bis (4) aus Abbildung 3.4: *Ablaufplan des Algorithmus mit Parametern und Variablen* (Seite 25) gezeigt. Nach der Bedingung folgen die Anweisungen, die ausgeführt werden sollen, wenn die Bedingung *wahr* ist. Dazu werden die Anweisungen, die den Schritten (3a) und (4a) entsprechen mit geschweiften Klammern ({... }) zu einem Block zusammengefasst. Nach diesem Block folgt das Schlüsselwort **else**, gefolgt von einem weiteren Block. Dieser Block wird ausgeführt, wenn die Bedingung nicht *wahr* ist und enthält in diesem Beispiel die Anweisungen, die (3b) und (4b) entsprechen.

4.2.3 Schleifen

Zur Beschreibung von Schleifen gibt es mehrere Möglichkeiten. Es gibt Schleifen, bei denen erst überprüft wird, ob die Abbruchbedingung erfüllt ist und dann die Schleife ausgeführt wird, bis diese Bedingung erfüllt ist. Außerdem gibt es noch Schleifen, die immer erst einmal ausgeführt werden, bevor geprüft wird, je nach dem ob also vor oder nach der Ausführung geprüft wird spricht man von «*pre-check-loop*» (vorher prüfende Schleife) oder «*post-check-loop*» (danach prüfende Schleife). Im Deutschen nutzt man die Begriffe «*kopfgesteuerte Schleife*» und «*fußgesteuerte Schleife*».

In einer Sketch sieht eine kopfgesteuerte Schleife so aus:

```
while (A != B)
{
    // Schleifenkörper
}
```

Eingeleitet wird mit dem Schlüsselwort **while**, dem (wie bei **if**) in runden Klammern () die Bedingung folgt. Das Beispiel zeigt die Bedinung aus Abbildung 3.4: *Ablaufplan des Algorithmus mit Parametern und Variablen* (Seite 25) ⑤ mit dem Ungleichoperator (!=). Die Schleife wird also so lange ausgeführt, wie die Zahlen in A und B ungleich sind. Danach folgt der Teil, der immer wieder ausgeführt wird, der sogenannte Schleifenkörper, der auch mit geschweiften Klammern ({...}) zusammengefasst ist.

An der Anordnung der Elemente (erst die Bedingung, dann der Schleifenkörper) kann man gut sehen, dass hier erst die Abbruchbedingung geprüft wird und dann der Schleifenkörper ausgeführt wird. Dementsprechend ist die Anordnung bei der fußgesteuerten Schleife andersherum:

```
do
{
    // Schleifenkörper
} while (A != B)
```

Eingeleitet wird diese Schleife mit dem Schlüsselwort «**do**» gefolgt vom Schleifenkörper an den dann das Schlüsselwort «**while**» mit der Bedingung in runden Klammern angehängt wird. An dieser Anordnung kann man schon erkennen: Hier wird erst der Schleifenkörper ausgeführt und dann die Bedingung geprüft.

Mit diesen beiden Schleifenarten kann man schon alles beschreiben, da man aber sehr oft Schleifen braucht, bei denen eine festgelegte Zahl von Wiederholungen ausgeführt wird, gibt es in den meisten Sprachen noch sog. Zählschleifen, bei denen eine Variable von einem Startwert bis zu einem Zielwert gezählt wird und für jedes Schritt wird der Schleifenkörper ausgeführt. Das könnte man auch mit einer Pre-Test-Loop machen, die Zählschleife ist aber übersichtlicher.

In einer Sketch sieht eine solche Zählschleife z.B. so aus:

```
for (int i = 0; i < 12; i++)
{
    // Schleifenkörper
}
```

Eingeleitet wird mit den Schlüsselwort **for** gefolgt, von den bekannten runden Klammern. Diesmal ist der Teil in den runden Klammern aber durch Semikolon in drei Abschnitte unterteilt:

```
for ( Initialisierung ;
      Abbruchbedingung ;
      Schritt )
{
    // Schleifenkörper
}
```

Die einzelnen Abschnitte haben folgende Bedeutung:

Initialisierung Dieser Teil wird vor Beginn des ersten Schleifendurchlaufes einmal ausgeführt. Im Beispiel sieht man, dass mit **int i = 0** eine Variable vom Typ **int** (Ganzzahl) deklariert und initialisiert wird. Die Variable **i** ist in der Schleife verfügbar und startet in unserem Beispiel mit dem Wert 0.

Abbruchbedingung Die Abbruchbedingung funktioniert, wie bei der anderen Pre-Check-Loop auch, sie wird vor jedem Schleifendurchlauf ausgewertet und wenn die Auswertung *falsch* ergibt, dann endet die Ausführung der Schleife. Das kann - wie bei jeder Pre-Check-Loop auch - schon vor dem ersten Schleifendurchlauf der Fall sein, dann wird der Schleifenkörper gar nicht ausgeführt.

Schritt Dieser Abschnitt wird nach jedem Durchlauf des Schleifenkörpers ausgeführt, bevor die Abbruchbedingung ausgewertet wird. Im Beispiel wird mit **i++** der Wert der Variablen **i** immer um eins erhöht. So dass nach dem zwölften Schleifendurchlauf die Abbruchbedingung erfüllt ist und die Ausführungendet.

4.2.4 Schleifenabbruch

Manchmal kann es notwendig sein, dass man die Ausführung des Schleifenkörpers mittendrin beenden will. Dann würde im Flussdiagramm ein Pfeil aus der Schleife ganz woanders hinzeigen. Dazu gibt es zwei Möglichkeiten:

continue Es wird nur die aktuelle Ausführung des Schleifenkörpers beendet, es wird also mit der Auswertung der Abbruchbedingung fortgefahrene und ggf. die Schleife weiter ausgeführt. Das Schlüsselwort hierfür ist **continue**.

break Es wird nur die komplette Schleife beendet, die Abbruchbedingung wird nicht noch einmal ausgewertet. Es wird mit der Ausführung des Codes nach der Schleife fortgefahrene. Das Schlüsselwort hierfür ist **break**.

Innerhalb des Schleifenkörpers sind beide Schlüsselworte nur innerhalb einer Verzweigung sinnvoll. Das folgende Beispiel zeigt die Verwendung von **continue** und **break**.

```
int a = 3;
int b = 5;
for (int i = 0; i < 12; i++)
{
    if (a > i)
    {
        continue;
    }
    if (b < i)
    {
        break;
    }
}
```

4.3 Unterprogramme

4.3.1 Deklaration eines Unterprogramms

Das folgende Beispiel ist aus den schon gezeigten Code-Teilen zusammengesetzt und zeigt exemplarisch das Unterprogramm zur Berechnung des ggT:

```
int ggT(int A, int B)
{
    int E = 0;          // Initialisierung der Variable E
    while (A != B)    // Solange A ungleich B ist wird die Schleife ausgeführt.
    {
        if (A > B)    // Verzweigung, abhängig von Bedingung: Ist A größer B?
        {
            E = A - B; // Block der ausgeführt wird wenn A größer B
            A = E;
        }
        else
        {
            E = B - A; // Block der ausgeführt wird wenn A NICHT größer B
            B = E;
        }
    }
    // Wenn wir hier hinter der Schleife ankommen
    // dann ist A nicht mehr ungleich B also
    // A ist gleich B und wir haben den ggT gefunden.
    return A; //Rückgabe eines der beiden gleichen Werte als ggT
}
```

Jedes Unterprogramm hat einen Namen über den es aufgerufen werden kann. Bei der Deklaration eines Unterprogramms wird zuerst der Datentyp der Rückgabe angegeben, dann der Namen eines Unterprogramms und dann in runden Klammern eine Liste von Parametern.

```
int ggT(int A; int B)
```

Bedeutet also, es gibt ein Unterprogramm mit dem *Namen ggT*. Dieses Programm *gibt*, wenn man es aufruft eine Ganzzahl (*int*) *zurück*. Beim Aufrufen des Programmes müssen zwei Ganzzahlen (*int*) als *Parameter* übergeben werden, die dann im Programm über die Variablennamen *A* und *B* verwendbar sind. Im Programm erfolgt dann die Berechnung des ggT von *A* und *B*. In der letzten Zeile des Pro-

gramms wird mit dem Schlüsselwort *return* das Ergebnis zurückgegeben, d.h. das Programm, das dieses Programm aufgerufen hat, bekommt das Ergebnis zurückgemeldet und sollte es dann mittels einer Zuweisung in einer Variablen abspeichern.

4.3.2 Aufruf eines Unterprogramms

Das Aufrufen eines Unterprogramms erfolgt in einer Zuweisung. Der Rückgabewert des Programms wird dabei in einer Variablen gespeichert. Das folgende Programm ruft das eben vorgestellte Programm ggT auf.

```
void main()
{
    int X = 25;
    int Y = 15;
    int Z = 0;
    Z = ggT(X,Y);
}
```

In dem Programm werden drei Ganzzahl-Variablen (X, Y, Z) deklariert und initialisiert. In der letzten Zeile erfolgt dann der Aufruf des Unterprogramms in einer Zuweisung. Links des Zuweisungsoperators (Gleichzeichen) steht die Variable Z , d.h. in dieser Variablen wird das abgespeichert, was am Ende des Programmablaufes mit `return` zurückgegeben wird. Als Parameter werden X und Y übergeben. Egal wie diese Variablen hier heißen, im Programm ggT landet der Wert aus dem ersten übergebenen Parameter in A und der zweite in B .

5 Ein eigenes Programm schreiben

5.1 Analyse von myBlink.ino

Das Programm beginnt mit einem mehrzeiligen Kommentar in dem beschrieben ist, was das Programm bei Ausführung machen wird.

```
/*
  Blink
  Let a LED on Port D5 blink.

  Sample by Joern Schlingensiepen
*/
```

Anschließend kommt ein sog. Definitionsblock. Hier wird festgelegt, dass überall, wo im Quelltext das Wort LED vorkommt, das Wort D5 eingefügt werden soll. D5 ist eine vorgegebene Konstante für den Pin D5, an dem die LED angeschlossen ist. Durch den Definitionsblock am Anfang kann man das Programm später leichter anpassen.

```
#define LED D5
```

Dann folgt ein Unterprogramm ohne Parameter mit den Namen `setup`. Per Definition startet der MCU dieses Programm direkt nachdem er gestartet wurde.

```
// the setup function runs once when
// you press reset or power the board
void setup () {
    // initialize digital pin
    //LED as an output.
    pinMode(LED, OUTPUT);
}
```

In `setup` wird das Programm `pinMode` aufgerufen. Diese Programm legt für einen Pin fest, ob der Pin als Ausgang zum Schalten oder als Eingang zum Einlesen von Information genutzt werden soll. Es bekommt als ersten Parameter die Nummer des Pins, hier also die in D5 gespeicherte Konstante, die für

LED eingesetzt wird und als zweiten Parameter die Richtung `OUTPUT` steht für Ausgang und `INPUT` stände für Eingang.

Dann folgt ein weiteres Unterprogramm ohne Parameter mit den Namen `loop`. Per Definition startet der MCU dieses Programm immer wieder neu, so dass es faktisch um eine Endlosschleife handelt. In diesem Programm muss die komplette Funktion der MCU implementiert werden.

```
// the loop function runs over
// and over again forever
void loop () {
    // turn the LED on by making
    // the voltage HIGH
    digitalWrite(LED, HIGH);
    // wait for a second
    delay(1000);
    // turn the LED off by making
    // the voltage LOW
    digitalWrite(LED, LOW);
    // wait for a second
    delay(1000);
}
```

Die `loop` ist recht einfach gehalten. Zuerst wird das Programm `digitalWrite` aufgerufen, es setzt den Zustand eines als Ausgang konfigurierten Pins. Dazu bekommt es als ersten Parameter die Nummer des Pins, hier also wieder die in D5 gespeicherte Konstante, die für LED eingesetzt wird und als zweiten Parameter den Wert, der am Pin angelegt werden soll, hier also `HIGH` für Spannung hoch. Nach dem Ausführen dieser Code-Zeile sollte die LED leuchten. Als nächstes wird das Programm `delay` aufgerufen, das einfach nur wartet. Als Parameter wird diesem Programm die Anzahl der Millisekunden, die gewartet werden soll, übergeben. Für

eine Sekunde also der Wert 1000. Während der Prozessor wartet, bleibt der Wert am Pin D5 konstant, also auf HIGH. Dann wird wieder `digitalWrite` aufgerufen. Diesmal mit dem zweiten Parameter LOW für Spannung 0. Ist dieser Punkt der Programmausführung erreicht, sollte die LED aus sein. Anschließend wird wieder gewartet, die LED bleibt währenddessen aus. Da die `loop` immer wieder ausgeführt wird, startet die Ausführung wieder oben im Programm mit dem Aufruf von `digitalWrite`, der die LED wieder einschaltet. Als Ergebnis blinkt die LED.

Aufgabe 3: Langsamer

Pass das Programm `gitHub:myBlink.ino` so an, dass die LED langsamer blinkt.

5.2 Hardwareaufbau #2

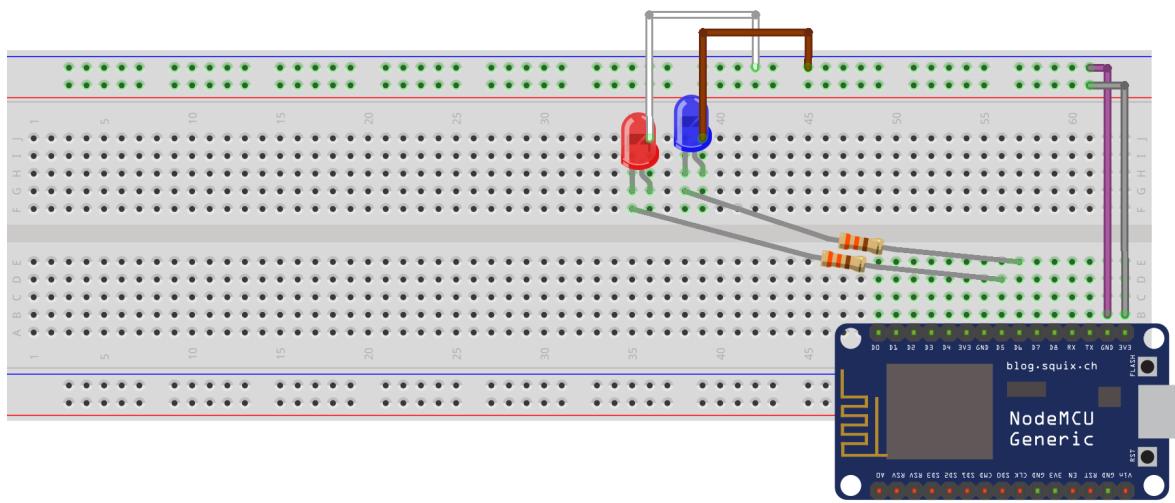


Abbildung 5.1: Schematische Darstellung des zweiten Aufbaus

Für die nächsten Versuche erweitern wir den Aufbau. In Abbildung 2.8: *Schematische Darstellung des ersten Aufbaus* (Seite 12) ist dargestellt, wie der erweiterte Aufbau aussieht.

Aufgabe 4: Aufbau 2

Bau die im Schema (⇒ Abbildung 5.1: *Schematische Darstellung des zweiten Aufbaus*, Seite 39) dargestellte Schaltung auf!

Aufgabe 6: Fade

Lade das Programm `gitHub:myFade.ino` und starte es. Dieses Programm nutzt das Programm `analogWrite(LED1, brightness)`. Überlege mit Deinem Arbeitspartner, was diese Programm macht und wie das Programm `gitHub:myFade.ino` funktioniert.

Aufgabe 5: Abwechselnd

Pass das Programm `gitHub:myBlink.ino` so an, dass die zwei LEDs abwechselnd blinken. Die Lösung findest Du in `gitHub:myBlink1.ino`.

5.3 Hardwareaufbau #3

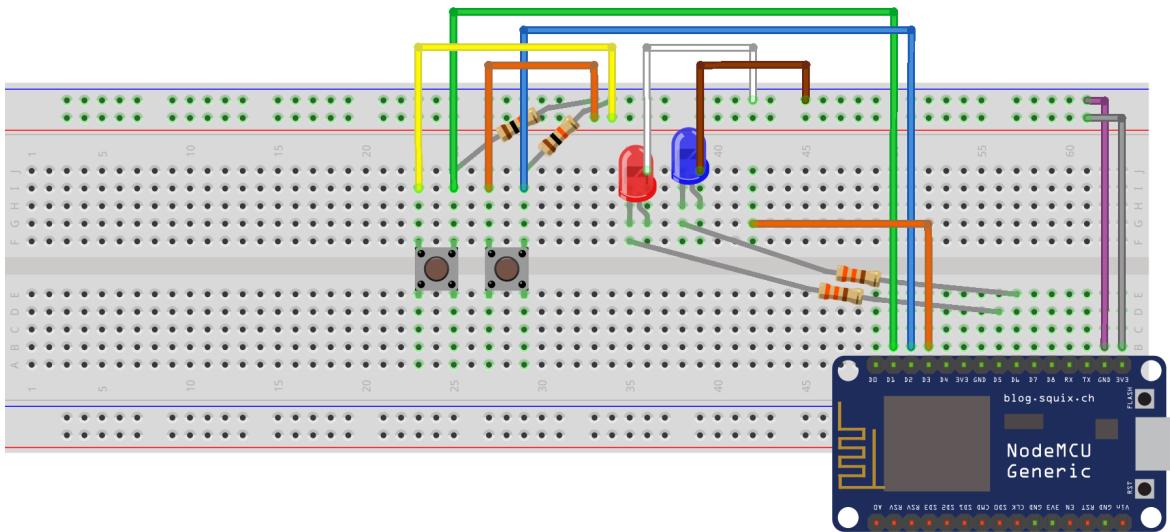


Abbildung 5.2: Schematische Darstellung des dritten Aufbaus

Für die nächsten Versuche erweitern wir den Aufbau. In Abbildung 2.8: *Schematische Darstellung des ersten Aufbaus* (Seite 12) ist dargestellt, wie der erweiterte Aufbau aussieht. Dazu werden zwei Taster eingebaut und mit dem Plus-Leiter verbunden. So liegt an den Pins D1 und D2 jeweils Plus an, wenn der entsprechende Taster gedrückt wird. Damit bei nicht gedrücktem Schalter auch wirklich 0 an den Pins anliegt, wird noch ein vergleichsweise großer Widerstand ($10k\Omega$) eingebaut. Über diesen Widerstand fließt, wenn geschaltet wird, nur ein sehr kleiner Strom.

Aufgabe 7: Aufbau 3

Bau die im Schema (\Rightarrow Abbildung 5.2: *Schematische Darstellung des dritten Aufbaus*, Seite 40) dargestellte Schaltung auf!

Aufgabe 8: Schalten

Lade das Programm [github:myButton1.ino](#) und führe es aus.

Im Programm [github:myButton1.ino](#) finden sich neue Konstrukte:

```
void setup() {
    // initialize the LED pins
    // as an output:
    pinMode(ledPin, OUTPUT);
    // initialize the pushbutton
    // pins as an input:
    pinMode(buttonPin, INPUT);
}
```

In **setup** wird das Programm **pinMode** mit dem Parameter **HIGH** aufgerufen, so dass der Pin mit dem Taster als Eingang fungiert.

```

void loop () {
    // read the state of the pushbutton value:
    buttonState =
        digitalRead (buttonPin);
    // check if the pushbutton is pressed.
    // If it is, the buttonState is HIGH:
    if (buttonState == HIGH) {
        // turn LED on:
        digitalWrite (ledPin , HIGH);
    } else {
        // turn LED off:
        digitalWrite (ledPin , LOW);
    }
}

```

In `loop` wird das Programm `digitalRead` aufgerufen. Es gibt den Zustand eines Eingangspins zurück, der dann in der Variablen `buttonState` abgelegt wird. Dann folgt eine Verzweigung mit der Bedingung `buttonState == HIGH`, es wird also geprüft, ob der Taster gedrückt war. Wenn er gedrückt war, dann wird die LED eingeschaltet. Ist er nicht gedrückt, dann wird sie ausgeschaltet. Da `loop` immer wieder und sehr schnell ausgeführt wird, bleibt die LED immer so lange angeschaltet, wie der Taster gedrückt ist. Denn nach dem Loslassen des Tasters wird im nächsten Durchlauf von `loop` der `buttonState` zu `LOW` und es wird der `else`-Zweig der Verzweigung durchlaufen und dort wird die LED ausgeschaltet.

Aufgabe 9: Zweimal Schalten

Passe das Programm [GitHub:myButton1.ino](#) so an, dass beim Drücken des zweiten Tasters die zweite LED leuchtet. Eine Lösung findest Du in [GitHub:myButton2.ino](#)

wieder los, dann wird sie wieder ausgeschaltet.

In dem Programm wird dazu eine Variable angelegt, in der der Zustand des System abgespeichert werden kann, so dass sich das System beim nächsten Durchlauf der `loop` entsprechend verhalten kann.

```

enum ledAutomatStateType
{
    pressedToOn ,
    pressedToOff ,
    lightOn ,
    lightOff
};

ledAutomatStateType
    ledAutomatState
        = lightOff;

```

In der `loop` wird dann je nach letztem Zustand entsprechend die LED ein- oder ausgeschaltet.

Aufgabe 10: Umschalten

Lade das Programm [GitHub:myButton3.ino](#), führe es aus und schaue Dir das Programm an. Es gibt hier eine neue Art der Flusskontrolle, versuche zu erkennen, wie diese funktioniert und diskutiere dies mit Deinem Arbeitspartner.

5.4 Zustände

Das Programm [GitHub:myButton3.ino](#) erweitert die Funktion so, dass nach Drücken und Loslassen des Tasters die LED dauerhaft angeschaltet ist. Drückt man nochmal und lässt

5.5 Kommunikation nach außen

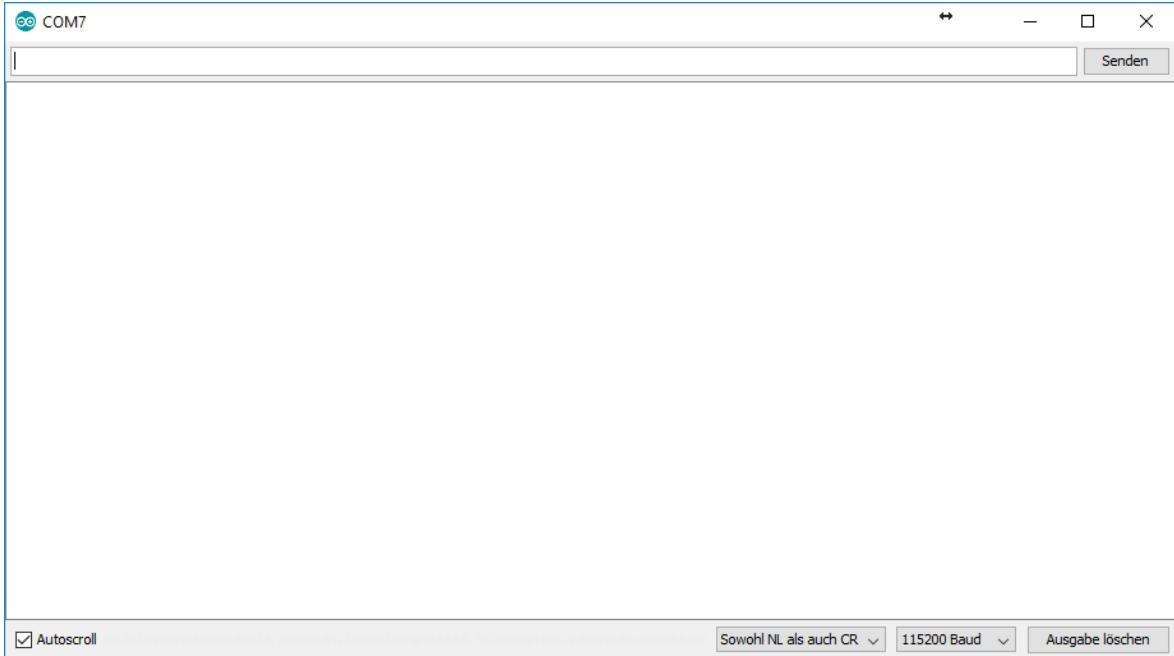


Abbildung 5.3: Der Serielle Monitor in der ArduinoIDE

Im Programm `gitHub:myButton4.ino` wird die Funktion um einen Kommunikationskanal erweitert. Im ersten Schritt wird dazu das verwendet, was ohnehin schon da ist. Der NodeMCU ist über die virtuelle serielle Schnittstelle ohnehin mit dem PC verbunden. Über diese Schnittstelle können Programme, die auf dem NodeMCU und dem PC ausgeführt werden, miteinander kommunizieren. Am einfachsten geht dies über den in der ArduinoIDE eingebauten «*Seriellen Monitor*», (⇒ Abbildung 5.3: *Der Serielle Monitor in der ArduinoIDE*, Seite 42), der über **Werkzeuge** → **Serieller Monitor** (**Strg**+**Umschalt**+**M**) gestartet werden kann. Unten rechts kann man die Übertragungsgeschwindigkeit einstellen, diese muss gleich der Geschwindigkeit sein, mit der man in der Sketch die Schnittstelle initialisiert. In den Beispielprogrammen ist das 9600 Baud.

```
void setup() {
    ...
    // Open internal serial port
    // with baud rate 9600
    Serial.begin(9600);
}
```

Die Initialisierung der Schnittstelle erfolgt in `setup` über den Aufruf `Serial.begin(9600)`. Dabei ist `Serial` ein sog. «*statisches Objekt*», d.h. es ist immer vorhanden und repräsentiert in diesem Fall die serielle Schnittstelle zum PC. Alle Funktionen, die ich an der Schnittstelle ausführen kann, werden über Programmaufrufe mit vorangestelltem `Serial.` ausgeführt. `Serial.begin` bekommt als Parameter die Geschwindigkeit, mit der die Schnittstelle arbeiten soll, übergeben. In diesem Beispiel sind das die schon erwähnten 9600 Baud.

```
void loop() {
    Serial.println(
        "Turned_Light_on");}
```

In der `loop` findet sich ein Aufruf von `Serial.println`. Am vorangestellten `Serial.` erkennt man, dass das Programm etwas mit der seriellen Schnittstelle machen soll. In diesem Fall sendet es die als Parameter übergebenen Zeichenkette zum PC.

Aufgabe 11: Umschalten

Lade das Programm [GitHub:myButton4.ino](#), führe es aus und starte den Seriellen Monitor. Schaue Dir an, wie sich das Programm verhält und versuche dann, den Quellcode nachzuvollziehen.

5.6 Kommunikation von außen

Im Programm [GitHub:myButton5.ino](#) wird die Funktion um einen Rückkanal erweitert.

```
void loop() {
    ...
    // check if chars in serial buffer
    if (Serial.available() != 0)
    {
        // read the string
        String input =
            Serial.readString();
        // converts to integer,
        // return 0 on conversion error
        int value = input.toInt();
        if (value >= 0 &&
            value <= 255)
        {
            analogWrite(
                ledPin1, value);
            Serial.print(
                "Set_Light_to_");
            Serial.println(value);
        }
    ...
}
```

Der ESP-12 legt alles, was über die serielle Schnittstelle eingeht in einem internen Puffer ab. Der Aufruf von `Serial.available` prüft, ob in diesem Puffer etwas abgelegt ist. Als Rückgabe bekommt man wahr oder falsch, also genau das, was man für eine Verzweigung braucht. Wenn da Daten liegen, dann sollen diese verarbeitet werden, wenn nicht, wird nichts gemacht. Da die `loop` immer wieder aufgerufen wird, kann man davon ausgehen, dass relativ schnell wieder geprüft wird, ob Daten eingegangen sind.

Der Aufruf von `Serial.readString` liest die eingegangenen Daten aus dem Puffer aus und gibt diese als Zeichenkette zurück. In dem Beispiel wird diese Zeichenkette der Variablen `input` zugewiesen, diese ist vom Typ `String`.

Der Aufruf `input.toInt` wandelt die Zeichenkette in eine Ganzzahl um. Hier sieht man wieder an der Schreibweise mit den vorangestellten `input.`, dass mit dem Inhalt der Zeichenkette gearbeitet werden soll. Die Umwandlung ist notwendig, weil wir über die Tastatur keine Zahlen, sondern Buchstaben eingeben. Die Zeichenkette "156" ist einfach nur eine Aufreihung der Buchstaben '1', '5' und '6'. Um damit rechnen zu können muss diese Zeichenkette in die Zahl 156 umgewandelt werden. An den Datentypen kann man erkennen, dass das eine als `String` (Zeichenkette) gespeichert ist, während das andere als `int` (Ganzzahl) abgelegt wird. Nach der Umwandlung wird mit `analogWrite` die LED auf den eingegebenen Wert gedimmt.

Aufgabe 12: Umschalten

Lade das Programm [GitHub:myButton5.ino](#), führe es aus und starte den Seriellen Monitor. Gib dort Zahlen zwischen 0 und 255 ein und sende diese an den NodeMCU! Schaue Dir an, wie sich das Programm verhält und versuche dann den Quellcode nachzuvollziehen.

Aufgabe 13: Umschalten

Lade das Programm [gitHub:-myButton6.ino](#), führe es aus und starte den Seriellen Monitor. Das Programm verhält sich anders, versuche nachzuvollziehen, wie es funktioniert.

6 Einbinden ins Netzwerk

Die Anbindung des NodeMCU an das Internet erfolgt in mehreren Schritten, der erste Schritt ist die Verbindung des MCU mit dem WLAN, zu finden in [GitHub:wifi.ino](#).

6.1 Einbinden interner Bibliotheken

Am Anfang dieses Programms findet sich folgender Eintrag:

```
#include <ESP8266WiFi.h>
```

Damit wird eine sog. «*Programmbibliothek*» (engl. «*software library*») eingebunden. Da Programme sehr schnell sehr groß und unübersichtlich werden können, organisiert man den Quellcode in diesen Bibliotheken. Der Vorteil ist, dass man Funktionen, die schon mal jemand implementiert hat, einfach durch Einbinden der Bibliothek selber nutzen kann. In diesem Beispiel ist das die Bibliothek, mit der der ESP auf das WLAN zugreifen kann.

6.2 Einbinden ins WLAN

```
const char* ssid
= "digitalisierung";
const char* pass
= "cloudification";

void setup() {
    ...
    WiFi.begin(ssid, pass);
    while (WiFi.status()
        != WL_CONNECTED)
    {
        delay(500);
```

```
        Serial.print(".");
    }
    Serial.println();
    Serial.println("OK");
    Serial.print("IP address.:.");
    Serial.println(WiFi.localIP());

    Serial.print("Strength.:.");
    Serial.println(WiFi.RSSI());
    Serial.println();
}
```

Ähnlich wie bei der seriellen Schnittstelle gibt es für das WLAN nach dem Einbinden der Bibliothek ein statisches Objekt. Dessen Name ist `WiFi`. Genau wie die serielle Schnittstelle wird das WLAN mit dem Aufruf von `begin` gestartet. Als Parameter müssen der Name des WLAN (die `ssid`) und ein Passwort übergeben werden. Defür wurden am Anfang zwei Zeichenketten als Variablen `ssid` und `pass` initialisiert. Mit den Voreinstellungen verbindet sich der NodeMCU mit dem WLAN-Hotspot für diesen Kurs.

Der Aufruf von `status` am `WiFi` gibt die Nummer des aktuellen Status des WLAN zurück, die vordefinierte Konstante `WL_CONNECTED` zeigt an, dass der NodeMCU mit dem WLAN verbunden ist. Die Schleife wird so lange durchlaufen, bis dieser Status erreicht wird, dabei wird jedes mal eine halbe Sekunde gewartet und ein `?` über die serielle Schnittstelle ausgegeben.

Damit man auf dem seriellen Monitor etwas sieht, werden noch mit `localIP` die Internet-Adresse des NodeMCU und mit `RSSI` die Signalstärke am `WiFi` abgefragt und über die serielle Schnittstelle ausgegeben.

Die `loop` ist relativ einfach gehalten:

```
void loop() {
    if (WiFi.status() == WL_CONNECTED) {
        Serial.print("Connected to ");
        Serial.println(WiFi.SSID());
        Serial.print("Strength: ");
        Serial.println(WiFi.RSSI());
    } else {
        Serial.println("Connection lost");
    }
    delay(1000);
}
```

In der `loop` werden nur jede Sekunde der Name des WLAN und die Signalstärke abgefragt und über die serielle Schnittstelle ausgegeben.

Aufgabe 14: WLAN testen

Lade das Programm `gitHub:wifi.ino`, führe es aus und starte den Seriellen Monitor. Beobachte was das Programm auf dem Seriellen Monitor ausgibt.

6.3 Verbinden mit einem WebServer

Das Programm `gitHub:httpClient1` implementiert den Zugriff auf einen Web-Server im Internet.

Dazu wird am Anfang eine zusätzliche Bibliothek eingebunden:

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
```

Diese stellt einen Web-Client zur Verfügung, der über das *hyper text transfer protocol (http)* Daten vom Web-Servern abrufen kann. Das Protokoll und die Inhalte, die

von einem Web-Server zurückgesendet werden, bestehen im wesentlichen aus Text, so dass die Rückgabe vom Server über die serielle Schnittstelle ausgegeben und dann gelesen werden kann.

Der übrige Aufbau des ersten Programmteils sieht aus, wie gewohnt, es werden eine Reihe von Konstanten definiert, die für den Zugriff auf den Server notwendig sind:

Im Gegensatz zum WLAN (`WiFi`) oder der seriellen Schnittstelle (`Serial`) wird der Web-Client nicht als statisches Objekt bereitgestellt. Das ist sinnvoll, weil man ggf. mehrere davon parallel nutzen will. Um trotzdem ein Objekt zu bekommen, muss es wie folgt erstellt werden:

```
HTTPClient http;
```

Das ist einfach die Deklaration einer Variablen mit dem Namen `http`. Diesmal ist aber nicht einfach nur eine Zahl oder eine Zeichenkette darin abgespeichert, sondern ein kompletter Web-Client.

Das Programm **setup** ist genau so aufgebaut, wie in der letzten Sketch. Es stellt die Verbindung zum WLAN her.

Der Zugriff auf den Web-Server erfolgt in der **loop**:

```
void loop() {
    ...
    http.begin (host , port , path);
    int err = http.GET();
    if (err == 200)
    {
        String content =
            http.getString();
        Serial.println(content);
    }
    else
    {
        Serial.print(err);
        delay(10000);
    }
    Serial.println(
        "Closing_connection");
    http.end();
    delay(3000);
}
```

Mit **begin** wird der Zugriff auf den Web-Server initialisiert. Dazu werden der Name des Servers (**host**), ein Port **port** und der Pfad **path**, auf den innerhalb des Servers zugegriffen werden soll, übergeben. Ein Port ist eine Art nummerierte Eingangstür, weil auf einem Rechner mehrere verschiedene Server laufen können, z.B. ein Web-Server, ein FTP-Server zum hochladen der Dateien und ein Mail-Server, braucht man eine Möglichkeit zu erkennen, mit welchem dieser Server der Client Kontakt aufnehmen möchte. Das erkennt man an der Port-Nummer.

Der eigentliche Zugriff auf den Server erfolgt in **GET**. Das **http** kennt nämlich verschiedene Arten des Zugriffs auf einen Server, deshalb sind Initialisierung und Zugriff getrennte Programmaufrufe. Der Rückgabewert ist ein sog. Error-Code, d.h. eine Zahl die repräsentiert, ob und was schief gegangen ist, das **http** kennt sehr viele dieser Codes; der bekannteste ist 404 für *file not found*, der wird

auch von Web-Browsern angezeigt und bedeutet, dass die Datei, auf die man zugreifen wollte, auf dem Server nicht gefunden wurde. Weniger bekannt ist der Code 200 für *Ok*, der einfach nur bedeutet, dass alles funktioniert hat.

In der Verzweigung, die auf den Aufruf folgt, wird einfach nur geprüft, ob der Error-Code 200 ist. Wenn ja, dann wird mit **http.getString** der Inhalt der Rückgabe - eine Zeichenkette - abgerufen und über die serielle Schnittstelle ausgegeben. Andernfalls wird die Fehlermeldung über die serielle Schnittstelle ausgegeben und 10 Sekunden gewartet. In jedem Fall wird danach mit **http.end()** die Verbindung zum Server geschlossen und dann 3 Sekunden gewartet.

Aufgabe 15: Einfacher Web-Zugriff

Lade das Programm [gitHub:httpClient1](#), führe es aus und starte den Seriellen Monitor. Beobachte was das Programm auf dem Seriellen Monitor ausgibt.

Aufgabe 16: Einfacher Web-Zugriff

Lade das Programm [gitHub:httpClient2](#), führe es aus und starte den Seriellen Monitor. Beobachte was das Programm auf dem Seriellen Monitor ausgibt und versuche im Quelltext nachzuvollziehen, was das Programm macht.

6.4 Hardwareaufbau #4

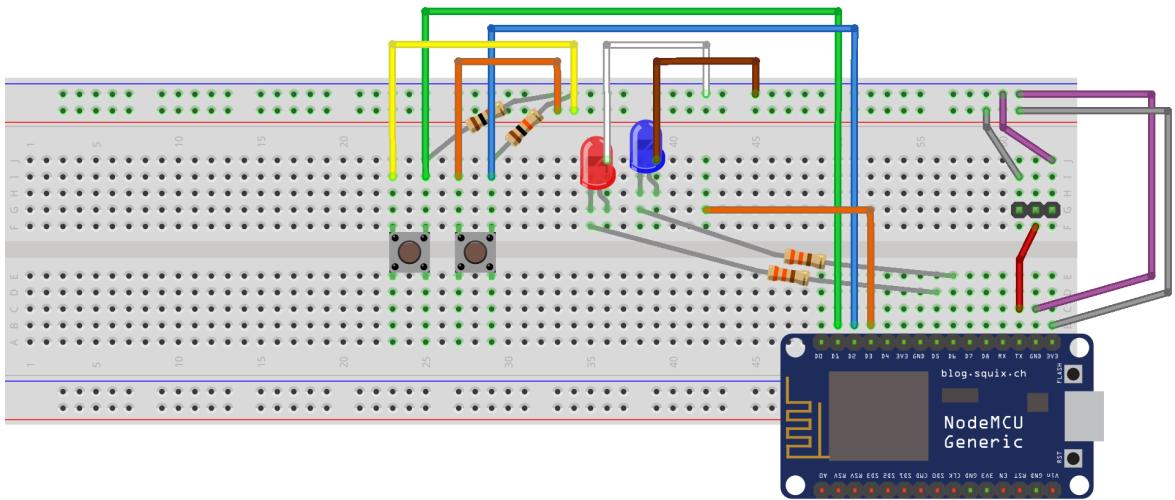


Abbildung 6.1: Schematische Darstellung des vierten Aufbaus

Als zusätzlicher Aktor soll ein Piezo-Summer/Buzzer eingebaut werden. Piezoelektrische Materialien verändern Ihre Form, wenn eine Spannung angelegt wird und erzeugen eine Spannung, wenn man ihre Form ändert. Man kann also Sensoren und Aktoren daraus bauen. In dem Buzzer, den wir einbauen, ist eine piezoelektrische Keramikelektrode verbaut. Durch Anlegen einer Spannung verändert diese ihre Form und bewegt damit eine kleine Membran. Wechselt man nun ständig die Spannung, beginnt die Membran zu schwingen. Wenn die Frequenz der Spannungswechsel im Hörbereich liegt, erklingt dabei ein Ton, den man hören kann. In Abbildung 6.1: Schematische Darstellung des vierten Aufbaus (Seite 48) ist der Aufbau schematisch dargestellt. Der Summer ist beschriftet: Bitte beim Einbauen darauf achten, dass die Polarität stimmt, also der Pluspol am Plusleiter und der Minuspol am Minusleiter angeschlossen wird.

Das Programm [gitHub:buzzer.ino](#) spielt eine Tonfolge auf dem Buzzer. Dazu werden in

der `loop` lediglich zwei neue Programmaufrufe eingeführt:

```
void loop() {
    tone(buzzPin, triad[counter]);
    ...
    noTone(buzzPin);
    ...
}
```

Der Aufruf von `tone` lässt einen Pin mit einer vorgegebenen Frequenz zwischen HIGH und LOW hin und her schalten. Wenig überraschend sind die zu übergebenden Parameter, die Nummer des Pin - im Beispiel `buzzPin` - und die Frequenz als Ganzzahl - im Beispiel `triad[counter]`. Die Funktion `noTone` schaltet das Hin- und Herschalten wieder ab.

Aufgabe 17: Buzzer

Lade das Programm `gitHub:buzzer.ino` und führe es aus. Höre was das Programm auf dem Buzzer ausgibt und versuche im Quelltext nachzu vollziehen, was das Programm macht.

Aufgabe 18: Buzzer Bonus

Versuche das Programm so anzupassen, dass eine Tonleiter oder eine Melodie gespielt wird.

die von einem Computer gesteuert werden. Um den Bot zu steuern, kann man über eine sog. Web-Schnittstelle auf den Telegram-Server zugreifen. Web-Schnittstelle bedeutet, dass der Server und der Client über das http miteinander kommunizieren, also genau so wie Web-Client und Web-Server. Als Text werden aber nicht Web-Seiten in html ausgetauscht, sondern sog. JSON-Messages.

Weitere Details sind im Rahmen dieses Kurses nicht von Bedeutung. Für den Zugriff auf den Telegram-Server wird eine Bibliothek verwendet. Wichtig ist nur, dass der Zugriff im Prinzip genau so funktioniert, wie in den Beispielen mit den Web-Clients.

6.5 Steuern über Chat-Bot

Zum Steuern des NodeMCU über das Internet wird die Chat-Applikation *Telegram* verwendet. Telegram bietet die Möglichkeit sog. Bots zu erstellen, das sind virtuelle Chat-Partner,

6.5.1 Einbinden externer Bibliotheken

Neben den internen Bibliotheken bietet die ArduinoIDE auch die Möglichkeit externe Bibliotheken einzubinden. Diese können aus Zip-Archiven importiert oder aus einem zentralen Repository im Internet geladen werden.

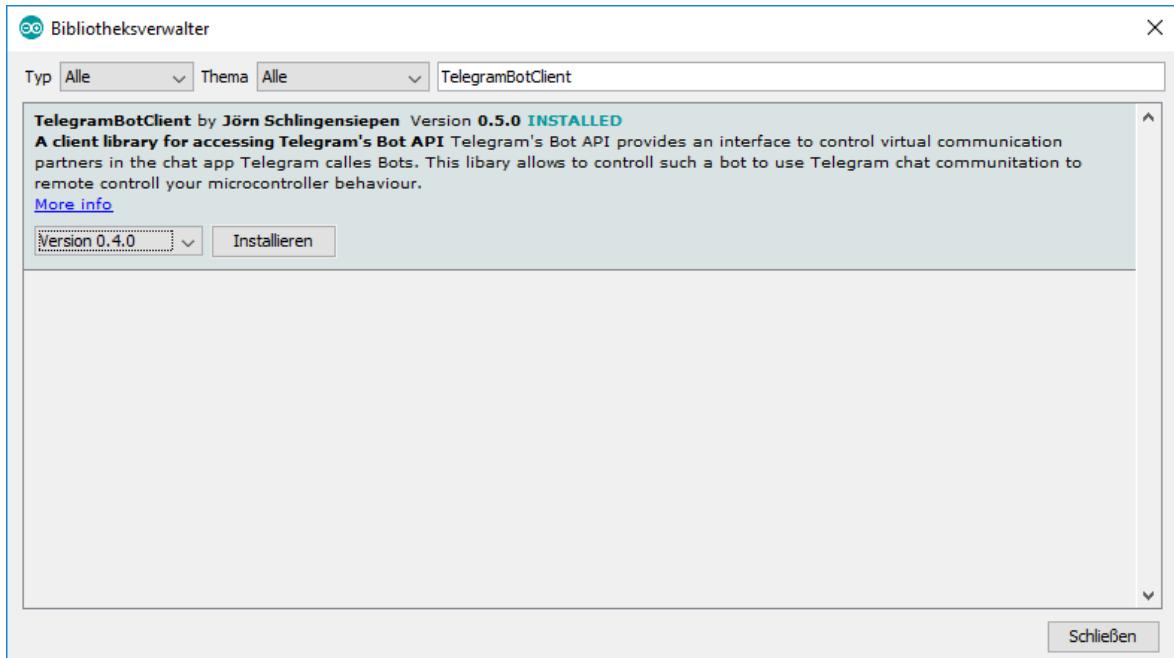


Abbildung 6.2: Bibliotheksverwalter der ArduinoIDE zum Einbinden von externen Bibliotheken

Zum Einbinden solcher Bibliotheken startet man den Bibliotheksverwalter über **Sketch** → **Bibliothek einbinden** → **Bibliotheken verwalten...**. Durch Eingaben in der oberen Text-Box können die verfügbaren Bibliotheken gefiltert werden. Über die Schaltfläche **Install** kann eine Bibliothek installiert werden. Im Rahmen des Kurses wird die Bibliothek **TelegramBotClient by Jörn Schlingensiepen** verwendet.

6.5.2 Einrichten eines eigenen Bots

Was kann dieser Bot?

BotFather is the one bot to rule them all. Use it to create new bot accounts and manage your existing bots.

About Telegram bots:

<https://core.telegram.org/bots>

Bot API manual:

<https://core.telegram.org/bots/api>

Contact [@BotSupport](#) if you have questions about the Bot API.

Abbildung 6.3: Telegram: Begrüßung durch den BotFather

Den Bot als virtuellen Kommunikationspartner richtet man direkt in der Telegram-App ein. Dazu startet man eine Konversation mit dem Kontakt *BotFather*.

Nach einer Begrüßung (⇒ Abbildung 6.3: *Telegram: Begrüßung durch den BotFather*, Seite 50) listet der BotFather die Funktionen, die er ausführen kann, auf (⇒ Abbildung 6.4: *Telegram: Funktionen des BotFathers*, Seite 51). Drückt man auf \newbot wird der Name des neuen Bots abgefragt (⇒ Abbildung 6.5: *Telegram: Neuer Bot erstellt*, Seite 52). Dieser muss auf Bot enden. Wenn alles funktioniert, legt BotFather einen neuen Bot an und zeigt auch einen sog. Token an. Mit diesem Token kann man später auf die Web-Schnittstelle zum Bot zugreifen, daher soll-

te man sich diesen Token abspeichern. Außerdem zeigt BotFather noch eine url in der Form t.me/<botname> an. Wenn man auf diesen Link klickt, wird der neue Bot als Kommunikationspartner angezeigt.

Zunächst antwortet der noch nicht, weil es noch kein Programm gibt, welches ihn über die Web-Schnittstelle steuert. Das soll im nächsten Schritt erstellt werden.

 **Jörn**
/start

 **BotFather**
I can help you create and manage Telegram bots. If you're new to the Bot API, please [see the manual](#).

You can control me by sending these commands:

- [/newbot](#) - create a new bot
- [/mybots](#) - edit your bots **[beta]**
- [/mygames](#) - edit your games **[beta]**

Edit Bots

- [/setname](#) - change a bot's name
- [/setdescription](#) - change bot description
- [/setabouttext](#) - change bot about info
- [/setuserpic](#) - change bot profile photo
- [/setcommands](#) - change the list of commands
- [/deletebot](#) - delete a bot

Bot Settings

- [/token](#) - generate authorization token
- [/revoke](#) - revoke bot access token
- [/setinline](#) - toggle inline mode
- [/setinlinegeo](#) - toggle inline location requests
- [/setinlinefeedback](#) - change inline feedback settings
- [/setjoiningroups](#) - can your bot be added to groups?
- [/setprivacy](#) - toggle privacy mode in groups

Games

- [/newgame](#) - create a new game
- [/listgames](#) - get a list of your games
- [/editgame](#) - edit a game
- [/deletetegame](#) - delete an existing game

 **Jörn** 09:43:11
/newbot



Abbildung 6.4: Telegram: Funktionen des BotFathers

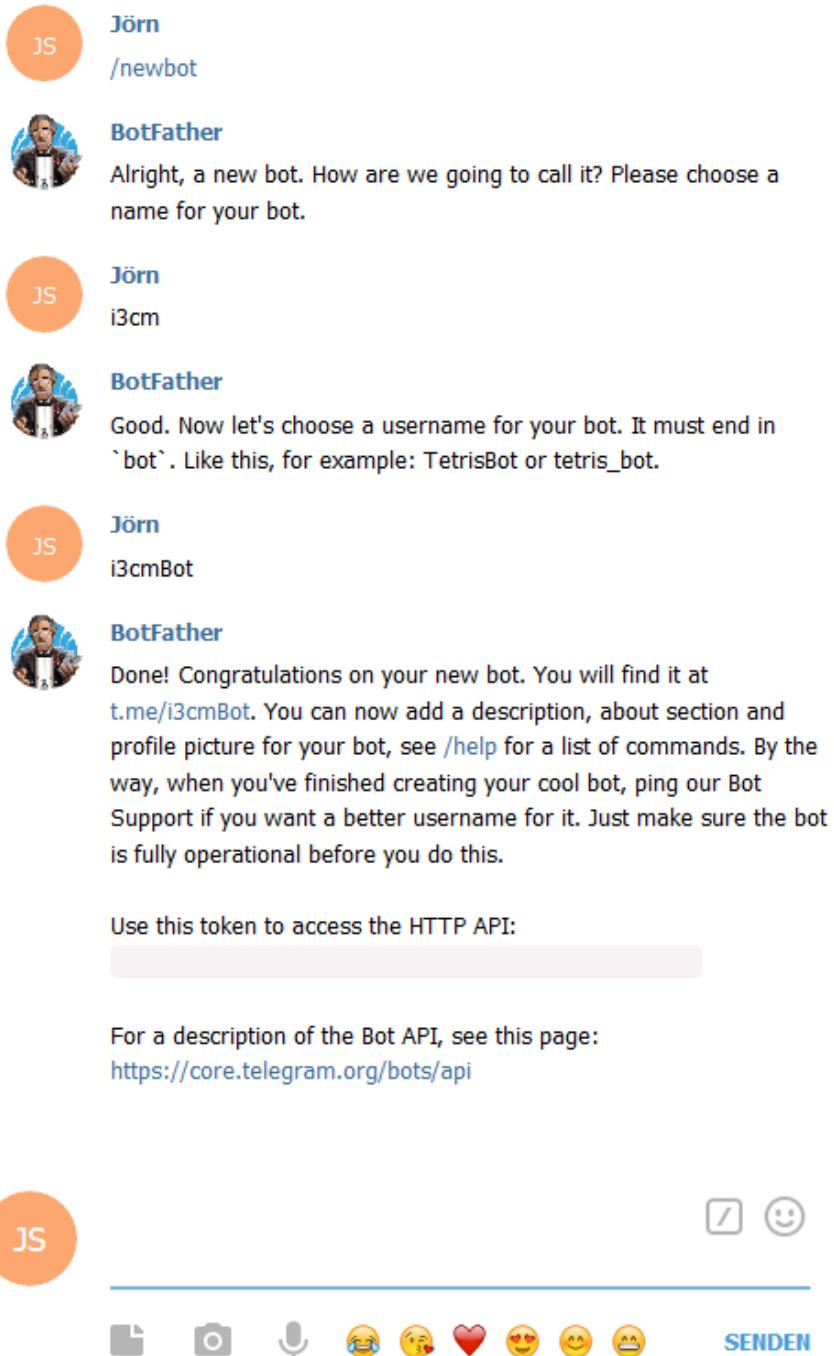


Abbildung 6.5: Telegram: Neuer Bot erstellt

6.5.3 Benutzen der Bibliothek TelegramBotClient

In [gitHub:TelegramEchoBot](#) findet man den Quellcode für einen EchoBot, das ist ein Bot, der immer das zurücksendet, was man ihm ge-

```
// Instantiate Telegram Bot secure token
// This is provided by BotFather
const String botToken = "<<_YOUR_BOT_TOKEN_>>" ;

// Instantiate the ssl client used to communicate with Telegram's web API
WiFiClientSecure sslPollClient ;

// Instantiate the client with secure token and client
TelegramBotClient client (
    botToken ,
    sslPollClient );
```

Bei `botToken` muss der Token eingetragen werden, den BotFather beim Erstellen des Bots erzeugt hat. `WiFiClientSecure sslPollClient` ist ein Client zum Zugriff auf das Internet, der alles TLS verschlüsselt. Mit der TLS Verschlüsselung kann auch http abgesichert

werden, man bezeichnet es dann als https. `TelegramBotClient client` ist der Client zum Zugriff auf den Telegram-Server, dem beim Instanziieren der Token (`botToken`) und der verschlüsselnde Client (`sslPollClient`) übergeben werden.

```
// Function called on receiving a message
void onReceive (
    TelegramProcessError tbcErr ,
    JwcProcessError jwcErr ,
    Message* msg)
{
    // Sending the text of received message back to the same chat
    // chat is identified by an id stored in the ChatId attribute of msg
    client . postMessage( msg->ChatId , msg->Text );
}
```

Dann wird ein neues Unterprogramm mit den Namen `onReceive` implementiert, es gibt nichts zurück, daher der Rückgabetyp `void`. Als Parameter erhält es zwei Error-Codes und eine Message. Das Programm wird später von `TelegramBotClient` aufgerufen, wenn eine neue Nachricht vom Server eingetroffen ist. In der Message selbst sind mehrere Informationen enthalten, die über den Operator `->` aus-

gelesen werden können. Enthalten sind: der Absender, die Absendezzeit usw.. Interessant für den EchoBot sind die `msg->ChatId` und der `msg->Text`. Der Bot kann wie ein echter Mensch mit mehreren Chat-Partnern chatten und in mehreren Chat-Gruppen Mitglied sein, daher ist es wichtig zu wissen, aus welchem Chat eine Nachricht gekommen ist. Die Chats können anhand der Id unterschieden werden.

Für den EchoBot ist das wichtig, damit er weiß in welchen Chat er zurückschreiben soll. Der Aufruf `client.postMessage` bekommt als Parameter eine Chat-Id und den Text, der versendet werden soll, übergeben. Im Fall des EchoBots genau die Chat-Id und den Text der eingehenden Nachricht.

```
// Function called if an
// error occurs
void onError (
    TelegramProcessError tbcErr ,
    JwcProcessError jwcErr )
{
    Serial.println("onError");
    ...
}
```

Analog zu `onReceive` gibt es auch ein `onError`, das der Client aufruft, wenn ein Fehler beim Empfangen oder Senden einer Nachricht aufgetreten ist. Der Rest des Programms besteht aus bekannten Aufrufen zum Verbinden mit dem Netzwerk und dem Starten der seriellen Schnittstelle. Interessant wird es im `setup`:

```
void setup() {
    ...
    // Sets the functions implemented
    // above as so called callback
    // functions, thus the client will
    // call this function on receiving
    // data or on error.
    client.begin(
        onReceive ,
        onError );
}
```

Mit `client.begin` wird der Client initialisiert. Dazu werden die beiden davor implementierten Unterprogramme `onReceive` und `onError` als Parameter übergeben.

Die `loop` ist dann relativ unspektakulär:

```
void loop() {
    // To process receiving
    // data this method has
```

```
// to be called each main
// loop()
client.loop();
}
```

Hier wird einfach nur das Programm `loop` am Client aufgerufen. Dieses Programm prüft, ob eine neue Nachricht angekommen oder ein Fehler aufgetreten ist und ruft dann entsprechend `onReceive` oder `onError` auf.

Aufgabe 19: EchoBot

Lade das Programm [gitHub:TelegramEchoBot.ino](#) und führe es aus. Sende dem Programm Nachrichten über die Telegram-App.

Bots können dem Anwender in Telegram Schaltflächen zur Auswahl vorgefertigter Antworten anzeigen. Das Programm [gitHub:TelegramEchoBotKeyboard.ino](#) demonstriert diese Möglichkeit.

Aufgabe 20: Keyboard

Lade das Programm [gitHub:TelegramEchoBotKeyboard.ino](#) und führe es aus. Sende dem Programm Nachrichten über die Telegram-App.

Die Funktion der vorgefertigten Antworten kann man auch nutzen, um dem Anwender Befehle zum Steuern des Testaufbaus als Knöpfe zur Verfügung zu stellen. Das Programm [gitHub:TelegramBotController.ino](#) demonstriert das.

Aufgabe 21: Controller

Lade das Programm [gitHub:TelegramBotController.ino](#) und führe es aus. Sende dem Programm Nachrichten über die Telegram-App.

7 Weitere Aktoren

7.1 Schieberegister

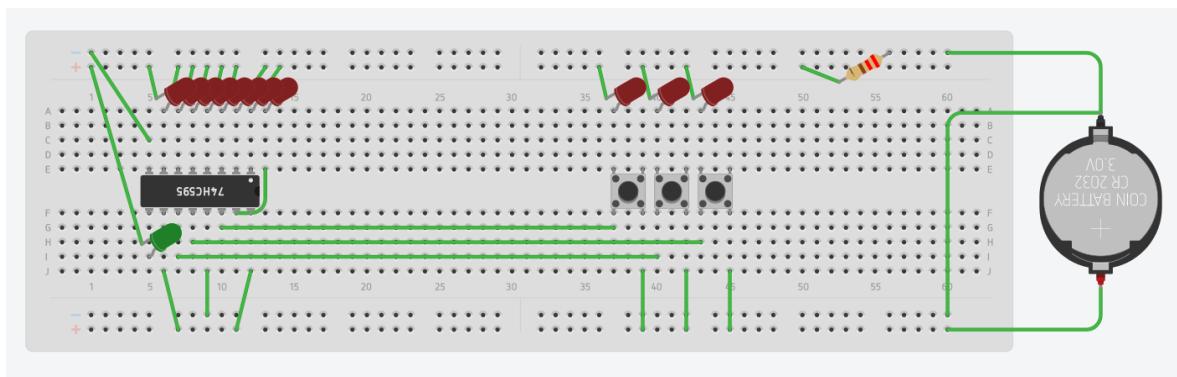


Abbildung 7.1: Schematische Darstellung des Aufbaus zum Schieberegister

Oft reichen die Pins, die eine MCU bereitstellt nicht aus, um alle Informationen darzustellen bzw. alle Aktoren gleichzeitig zu steuern. Dann müssen Informationen hintereinander übertragen werden. Eine einfache Möglichkeit dazu ist ein Schieberegister. Ein einfacher Aufbau zur Simulation eines solchen Schieberegisters findet sich unter <https://www.tinkercad.com/things/eyYTnw15L81>, Kurzlink: [gk1](#).

Dieses Schieberegister hat acht Datenausgänge, deren Zustand über drei Eingänge gesteuert wird. Im Baustein gibt es einen Datenspeicher mit acht Plätzen, in denen jeweils HIGH oder LOW gespeichert ist. Die drei Eingänge des Schieberegisters haben folgende Funktion:

Data Hier liegt die eigentliche Information - das Datum - als HIGH oder LOW an.

Clk Die Clock zeigt an, wann ein Datum eingelesen werden soll. Ändert sich der Zustand von CLK von LOW nach HIGH, dann werden alle Einträge im Datenspeicher um eins weiter geschoben (daher Schieberegister) und das Datum, das an Data anliegt, wird in den vorerste (jetzt freien) Speicherplatz eingespeichert.

Latch Zeigt an, dass die Daten, die im Datenspeicher eingespeichert sind, an den Datenausgängen ausgegeben werden sollen. Ändert sich der Zustand von LATCH von LOW nach HIGH, dann werden die Ausgänge entsprechend der Speichereinträge auf LOW oder HIGH geschaltet.

Aufgabe 22: Schieberegister

Starte die Simulation in TinkerCAD und versuche die vier mittleren LEDs zum Leuchten zu bringen. Die Taster können einfach mit der Maus angeklickt werden. Um in TinkerCAD einen Taster *festzuhalten*, muss er mit gedrückter Shift-Taste angeklickt werden.

7.2 LED-Matrix

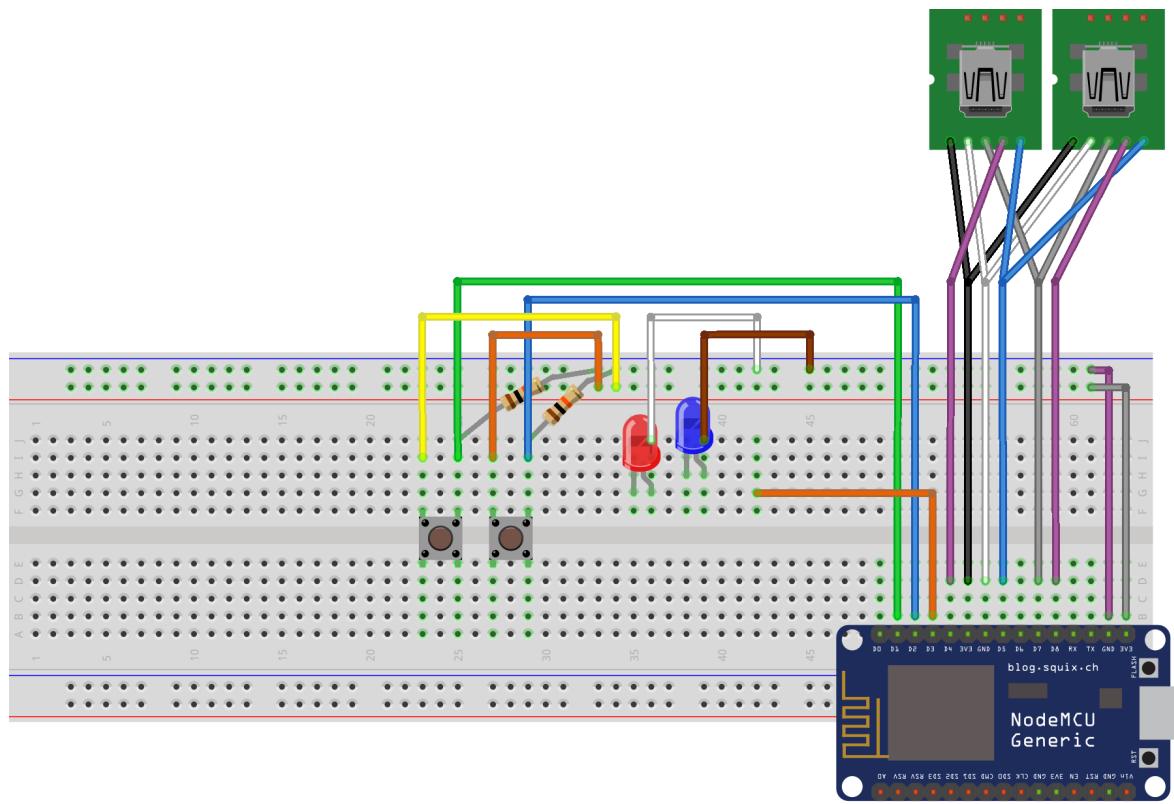


Abbildung 7.2: Schematische Darstellung des Anschlusses der LED-Matrizen

Ein typisches Beispiel für die Ansteuerung von sehr vielen Aktoren sind LED-Anzeigen. Daher soll hier am Beispiel einer LED-Matrix die Funktionsweise erläutert werden. Die LED-Matrix ist über vier Leitungen mit der MCU verbunden und steuert 8x8 also 64 LEDs, die auch ähnlich, wie beim Beispiel `gitHub:myFade.ino` in ihrer Helligkeit gesteuert werden können. Die Ansteuerung übernimmt ein Spezial-Chip MAX7219, der auch noch so in Reihe geschaltet werden kann, dass noch viel mehr LEDs angesteuert werden können.

7.2.1 Funktionsweise

Auch der MAX7219 hat keine 64 Ausgänge, die komplette Matrix wird über jeweils einen Eingang pro Spalte und einen Eingang pro Reihe, also 16 Eingänge (8 Reiheneingänge und 8 Spalteneingänge) geschaltet. Dabei macht man sich zum Einen die Eigenschaften von LEDs zu Nutze, nur in eine Richtung Strom zu leiten und zum Anderen die Trägheit des menschlichen Auges. Ein einfacher Aufbau zur Simulation einer LED-Matrix findet sich unter: <https://www.tinkercad.com/things/2gQoV7RvQVU>, Kurzlink: [khi](#).

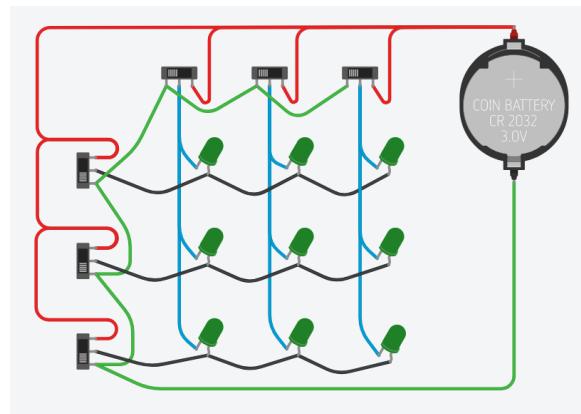


Abbildung 7.4: Simulation LED-Matrix

Das macht man sich zu Nutze um einzelne Reihen zu aktivieren. Die Simulation in Abbildung 7.4: *Simulation LED-Matrix* (Seite 58) zeigt, dass man durch die Selektion einer Reihe erreichen kann, dass die Information, die an den Spalten-Eingängen anliegen, sich genau auf die LEDs in dieser Reihe auswirken, während alle anderen aus bleiben.

Den Rest erledigt die Trägheit der menschlichen Wahrnehmung. Der Controller schaltet einfach ganz schnell hintereinander die einzelnen Reihen durch und legt immer die passende Information an den Reihen-Eingängen an. Sogar die Wahrnehmung der Helligkeit kann so gesteuert werden, für *dunkleres* Leuchten schaltet der Controller die LED einfach schnell wieder aus, soll *heller* geleuchtet werden, bleibt die LED bis zum Ende des Zeitslots für diese Reihe an.

Aufgabe 24: LED-Matrix

Starte die Simulation in TinkerCAD und schalte die einzelnen Reihen durch. Versuche eine Raute zu schalten!

Aufgabe 23: LED

Starte die Simulation in TinkerCAD und probiere die verschiedenen Schaltkombinationen durch, um zu sehen, wann die LED leuchtet.

7.2.2 Serial Peripheral Interface

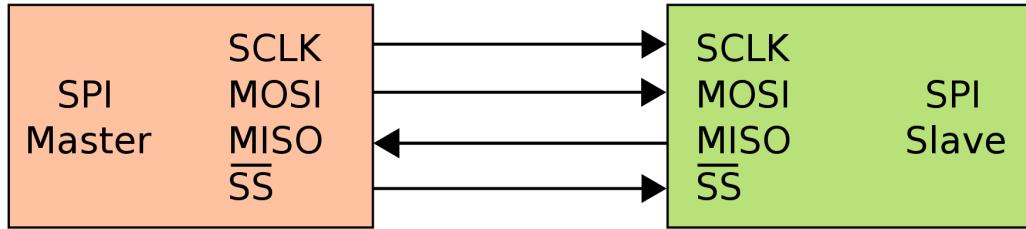


Abbildung 7.5: Schematische Darstellung des Serial Peripheral Interface Bus (SPI-Bus)

Am Beispiel des Schieberegisters konnte man sehen, dass zur Übertragung von Informationen zwei Leitungen reichen, wenn man die Information hintereinander überträgt. Diese Art der Datenübertragung ist für verschiedene Anwendungsfälle standardisiert. Der MAX7219 ist über einen sog. *Serial Peripheral Interface Bus (SPI-Bus)* angebunden. In Abbildung 7.5: *Schematische Darstellung des Serial Peripheral Interface Bus (SPI-Bus)* (Seite 59) kann man sehen, dass für den SPI-Bus insgesamt vier Leitungen und zwei Rollen standardisiert sind:

SCLK Die *Serial Clock* funktioniert genauso, wie beim Schieberegister. Wenn sich hier etwas am Schaltzustand ändert, dann liegen Daten an. Damit das funktioniert, müssen die Rollen *Master* und *Slave* festgelegt sein, weil nur ein Chip die SCLK schalten können darf.

MOSI *Master Out Slave In* ist die Datenleitung in Richtung Slave

MISO *Master In Slave Out* ist die Datenleitung in Richtung Master

SS *Slave Select* wählt den Slave aus, der Senden und Empfangen darf. In manchen Dokumentationen heißt diese Leitung auch *Chip Select (CS)*.

Zur eigentlichen Datenübertragung ist *SS* also nicht notwendig, es erlaubt aber den Betrieb von mehreren Slaves an den gleichen Leitungen. In Abbildung 7.6: *Schematische Darstellung des SPI-Bus mit drei Slaves* (Seite 60) kann man sehen, dass der Slave-Select-Mechanismus die Möglichkeit bietet, für jeden zusätzlichen Slave nur noch einen Datenpin zu verbrauchen, an dem das *SS* des zusätzlichen Chips angeschlossen wird.

Genau das macht sich die Schaltung in Abbildung 7.2: *Schematische Darstellung des Anschlusses der LED-Matrizen* (Seite 57) zu Nutze; die lila Drahtbrücken an D4 und D8 sind die Slave-Select-Leitungen.

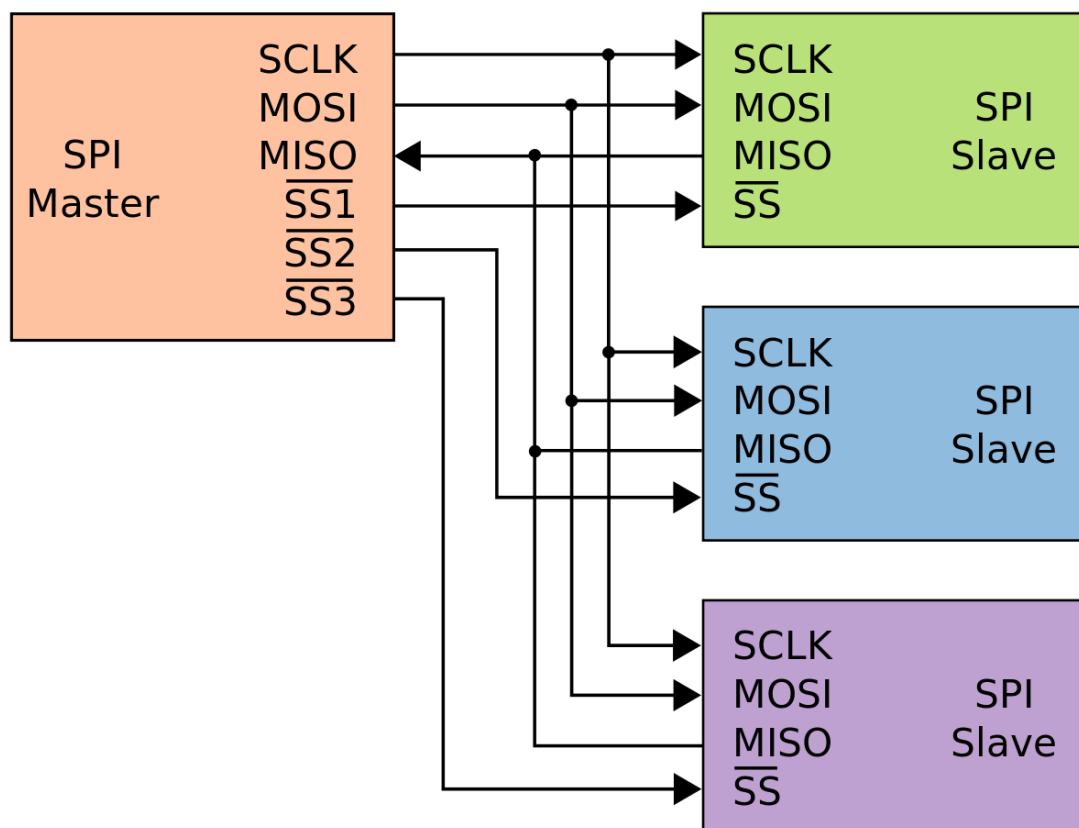


Abbildung 7.6: Schematische Darstellung des SPI-Bus mit drei Slaves

Aufgabe 25: Aufbau Matrix

Baue die im Schema Abbildung 7.2: *Schematische Darstellung des Anschlusses der LED-Matrizen* (Seite 57) dargestellte Schaltung auf! Entferne dazu die Widerstände, mit denen die LEDs verbunden sind. Die Anschlüsse an der LED-Matrix sind beschriftet. Achte darauf, dass VCC und GND richtig herum an 3v3 und GND angeschlossen sind und CS jeweils an D4 und D8.

7.2.3 Ansteuern der LED-Matrizen

In [gitHub:ledmatrix.ino](#) findet sich ein Beispielprogramm zum Ansteuern der beiden LED-Matrizen. Die Ansteuerung der LED-Matrix über den SPI-Bus und den MAX7219 erfolgt wieder über Bibliotheken, welche die grundsätzliche Kommunikation über SPI (`SPI.h`) und mit dem Chip (`LedMatrix.h`) schon implementieren.

Nach dem notwendigen Definitionsblock zur Festlegung der Pins etc. werden die Objekte für die Steuerung der beiden Matrizen initialisiert:

```
LedMatrix ledMatrix1 =
    LedMatrix(1, CS_PIN1);
LedMatrix ledMatrix2 =
    LedMatrix(1, CS_PIN2);
```

Als Parameter werden übergeben: Die Anzahl der kaskadierten MAX7219 (s.o. Der Chip selbst kann für größere LED-Flächen hintereinandergeschaltet werden.) und der Pin an dem der Slave-Select-Pin des Chip angeschlossen ist. Die übrigen Pins liegen standardmäßig schon da, wo sie nach dem vorgegebenen Schema verdrahtet sind.

Im `setup` werden dann die Objekte initialisiert:

```
void setup() {
    ...
    ledMatrix1.init();
    ledMatrix1.setIntensity(4);
    ...
}
```

`init` startet die Kommunikation mit dem MAX7219 und `setIntensity` setzt die Helligkeit, hier können Zahlen zwischen 0 und 15 übergeben werden. Das vorangestellte `ledMatrix1.` zeigt hier an, dass die Programme an dem Objekt für die erste LED-Matrix ausgeführt werden soll.

```
void loop()
{
    ledMatrix1.clear();
    ledMatrix1.setPixel(
        counter/8, counter%8);
```

```
ledMatrix1.commit();
...
counter++;
if (counter >= 64)
    counter=0;
}
```

In der **loop** wird dann immer die nächste LED angeschaltet. Der Aufruf von **clear** an der ersten Matrix schaltet alle LEDs aus. Mit **setPixel** wird ein einzelner Pixel eingeschaltet. Als Parameter werden dessen x und y Koordinaten auf der Matrix (Reihe und Spalte) übergeben. Dazu wird der Modulo-Operator (%), der den Rest einer ganzzahligen Division berechnet, genutzt, um mit Hilfe des **counter** immer die Koordinaten für die nächste LED zu berechnen. Der Aufruf **commit** startet die eigentliche Übertragung der LED-Schaltzustände an den MAX7219. So können auch mehrere Aufrufe von **setPixel** hintereinander gemacht werden, die dann zu einer Übertragung zusammengefasst werden.

Aufgabe 26: Pixel testen

Lade das Programm [gitHub:ledmatrix.ino](#) und führe es aus. Beobachte was das Programm auf den LED-Matrizen ausgibt.

Das Programm [gitHub:ledmatrix1.ino](#) demonstriert noch eine weitere Funktion der LEDMatrix-Bibliothek. Über die Bibliothek können Texte auf der LED-Matrix ausgegeben werden. Dazu wird in **setup** der Text in das jeweilige Objekt eingespeichert:

```
void setup() {
    ...
    ledMatrix1.setText(
        "Digitalisierung");
    ...
    ledMatrix2.setText(
        "Cloudification");
    ...
}
```

```
void loop() {
    ledMatrix1.clear();
    ledMatrix1.scrollTextLeft();
    ledMatrix1.drawText();
    ledMatrix1.commit();
    ...
}
```

In der **loop** wird dann die Matrix gelöscht (**clear**), der Text jeweils weitergescrollt (**scrollTextLeft**), dann der Text an der jeweiligen Scrollposition in die Matrix gemalt (**drawText**) und zum Schluß wird alles an die Matrix übertragen (**commit**).

Aufgabe 27: Text ausgeben

Lade das Programm [gitHub:ledmatrix1.ino](#) und führe es aus. Beobachte was das Programm auf den LED-Matrizen ausgibt.

Aufgabe 28: Text CinemaSkope

Passe das Programm [gitHub:ledmatrix1.ino](#) so an, dass, wenn man beide LED-Matrizen nebeneinander hält, ein Text über beide Matrizen läuft.

Aufgabe 29: Text von Bot

Erstelle ein Programm, das über Telegram Nachrichten erhält und diese über die beiden nebeneinander hängenden LED-Matrizen ausgibt. (Eher ein langfristiges Projekt).

7.3 LCD Display

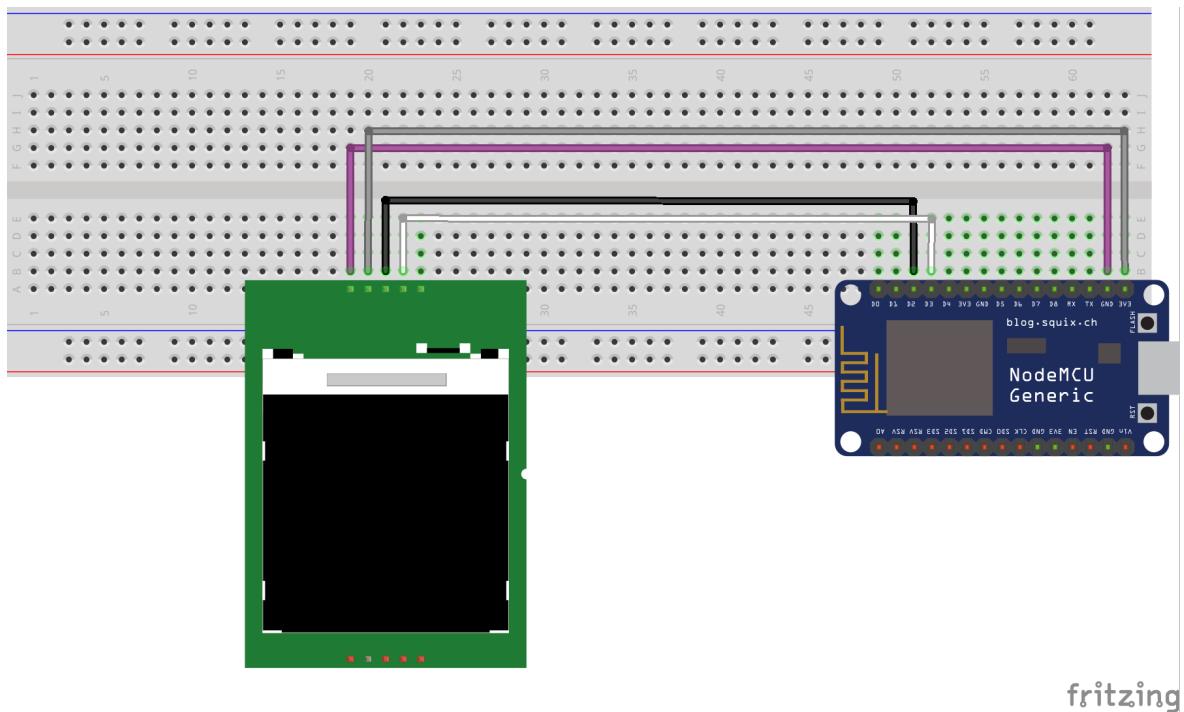


Abbildung 7.7: Schematische Darstellung des Anschlusses des LCD-Displays

Abbildung 7.7: Schematische Darstellung des Anschlusses des LCD-Displays (Seite 63) zeigt den Anschluß eines LCD-Displays an den NodeMCU. Neben der Stromversorgung sind nur zwei Datenkabel notwendig. Das Display ist über einen sog. I2C-Bus angebunden, über den auch mehrere Geräte über die gleichen Datenleitungen angebunden werden können, weil die Geräte auf dem Bus über eine interne ID adressiert werden.

Für eine solche Anbindung bedient man sich mehrerer Bibliotheken. Die interne Bibliothek `wire` stellt die Anbindung verschiedener Geräte über I2C-Bus zur Verfügung. Das Programm `github:i2c.ino` nutzt diese Bibliothek um alle verfügbaren Geräte auf dem Bus anzuzeigen.

Für die Ansteuerung des Displays kommt die externe Bibliothek **Adafruit SSD1306** by Adafruit zum Einsatz. Und damit man nicht jeden Pixel einzeln malen muss, kommt auch die Bibliothek **Adafruit GFX Library** by Adafruit zum Einsatz. Diese Grafik-Effekt-Bibliothek stellt Funktionen bereit, um z.B. Linien zu malen oder Text auszugeben. Sie implementiert also Funktionen um die höheren Grafikkonzepte *Linie*, *Kreis*, *Ellipse* etc. in Schaltzustände der Pixel auf dem Display umzurechnen.

`gitHub:screentest.ino` und `gitHub:screentest.ino` enthalten eine entsprechende Demo und sind ein guter Startpunkt für eigene Experimente.

Abbildungsverzeichnis

2.1	Ziel dieses Kapitels, der Aufbau zur Inbetriebnahme	7
2.2	Aufgelöteter ESP8266EX [www.espressif.com]	8
2.3	Beispiele für RS232 Schnittstellen	8
2.4	Beispiel für ein Produkt: SOnOff, schaltet 230V über WLAN	9
2.5	Zusammengebauter SOnOff-Schalter	9
2.6	Der ESP-01 von AI-Thinker [www.antratek.de]	9
2.7	NodeMCU, Development Board mit ESP-12 [https://www.exp-tech.de/]	10
2.8	Schematische Darstellung des ersten Aufbaus	12
2.9	Drahtbrücke zum Verbinden der Spalten auf einem Steckbrett	13
2.10	Arduino-IDE mit neuer Sketch	14
2.11	Dialog zur Konfiguration der IDE	15
2.12	Boardverwalter: Dialog zur Installation der Einstellungen für zusätzliche Boards	16
3.1	Ein einfacher TurtleBot aus Lego	19
3.2	Beispiele für den Fahrweg eines einfachen TurtleBots beim Verlassen eines Raumes	20
3.3	Ablaufplan des Algorithmus	23
3.4	Ablaufplan des Algorithmus mit Parametern und Variablen	25
5.1	Schematische Darstellung des zweiten Aufbaus	39
5.2	Schematische Darstellung des dritten Aufbaus	40
5.3	Der Serielle Monitor in der ArduinoIDE	42
6.1	Schematische Darstellung des vierten Aufbaus	48
6.2	Bibliotheksverwalter der ArduinoIDE zum Einbinden von externen Bibliotheken	49
6.3	Telegram: Begrüßung durch den BotFather	50
6.4	Telegram: Funktionen des BotFathers	51
6.5	Telegram: Neuer Bot erstellt	52
7.1	Schematische Darstellung des Aufbaus zum Schieberegister	55
7.2	Schematische Darstellung des Anschlusses der LED-Matrizen	57
7.3	Simulation LED	58
7.4	Simulation LED-Matrix	58
7.5	Schematische Darstellung des Serial Peripheral Interface Bus (SPI-Bus)	59
7.6	Schematische Darstellung des SPI-Bus mit drei Slaves	60
7.7	Schematische Darstellung des Anschlusses des LCD-Displays	63