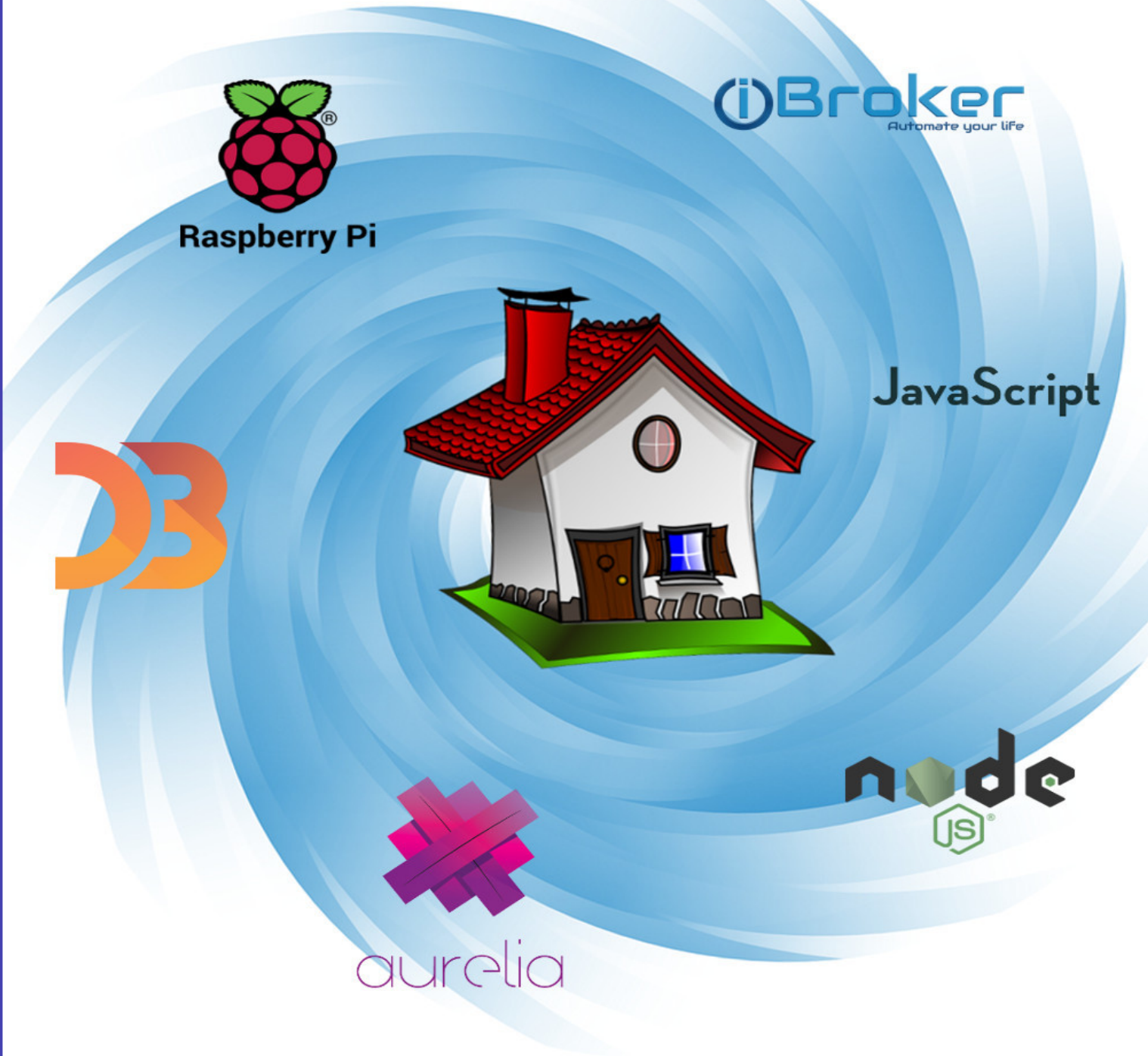
# Smarthome

## do it yourself

**Überwachen und steuern Sie Ihr Haus mit Raspberry Pi, ioBroker und selbst programmierten WebApps**




Roger Inigo

# Smarthome DIY

## Überwachen und steuern Sie Ihr Haus mit Raspberry Pi, ioBroker und selbst programmierten WebApps.

Roger Inigo

Dieses Buch wird verkauft unter http://leanpub.com/smarthdiy

Diese Version wurde veröffentlicht am 2020-08-30

# Inhaltsverzeichnis

**Kapitel 3: IoBroker Scripting** . . . . . **1**
Einrichtung des Script-Hosts . . . . . 2
Einfache Skripte . . . . . 5
Skripte sichern . . . . . 24

**Kapitel 4: Standalone Front-End** . . . . . **25**
Teil 3: DoubleGauge . . . . . 35
Teil 4: Druckknopf . . . . . 49
Teil 5: Tri-State Button . . . . . 59
Teil 6: Lineare Anzeigegeräte . . . . . 66

**Kapitel 6: Bastelstunde** . . . . . **71**
Grundätzliches und was man zum Basteln braucht . . . . . 72
Barometer . . . . . 85
Motor . . . . . 101
Elektroboiler mechanisch steuern . . . . . 106

# Kapitel 3: IoBroker Scripting

Nachdem wir jetzt verschiedene Komponenten eingerichtet und im ioBroker UI geschaltet und ausgelesen haben, wollen wir anfangen, ein paar Sachen zu automatisieren. Dazu verwenden wir den in ioBroker enthaltenen Scripting-Host.

Es sei an dieser Stelle darauf hingewiesen, dass ioBroker verschiedene Methoden kennt, um Abläufe zu automatisieren. Ich verwende hier die "pure" JavaScript-Methode. Dies einerseits, weil es mir gefällt, exakt kontrollieren zu können, was ein Programm tut, und andererseits, weil man auf diese Weise "alles" kann und nicht auf die Möglichkeiten der jeweiligen Automatisierungstechnik begrenzt ist bzw. nur mit zusätzlicher Mühe daraus ausbrechen kann. Dennoch: Wenn Ihnen visuelles Programmieren besser liegt, schauen Sie sich zum Beispiel den "Node Red"-Adapter oder den "Scenes"-Adapter genauer an. Insbesondere Node Red ist inzwischen eine weitverbreitete Programmiertechnik geworden, für die es auch viel Hilfe im Netz gibt. (https://nodered.org).

Falls Sie aber auf die "harte Tour" skripten wollen, folgen Sie mir bitte weiter.

Alle Skripte, die ich im Folgenden zeige, finden Sie auch im Verzeichnis "iobroker-scripts" der Begleitsoftware, welche Sie wie im Anhang gezeigt auf Ihren Computer klonen können. Ich empfehle Ihnen, dies auch zu tun da ich hier machmal nur die relevanten Teile der Skripte vorstelle. Den vollständigen Code finden Sie dann jeweils im Quellen-Verzeichnis.

Die hier besprochenen Skripts befinden sich im Unterverzeichnis 'iobroker', wenn Sie das Repository wie im Anhang gezeigt clonen, und dann den Haupt-Branch auschecken:

```
git checkout master
```

# Einrichtung des Script-Hosts

Richten Sie Ihren Browser auf ioBroker (http://homeview.local:8081) und öffnen Sie den Reiter ‘Adapter’. Sie benötigen die ‘Script Engine’, bzw. ‘Skriptausführung’ in der deutschen Version (Sie können auch einfach ‘script’ in das Feld ‘Filter’ eingeben, dann brauchen Sie nicht die immer länger werdende Liste zu durchsuchen). Installieren Sie diesen Adapter in gleicher Weise wie im vorigen Kapitel gezeigt, und lassen Sie alle Einstellungen der Instanz-Konfiguration auf den Vorgaben. Nach erfolgreicher Installation haben Sie einen neuen Reiter namens ‘Skripte’ in Ihrer ioBroker Oberfläche.

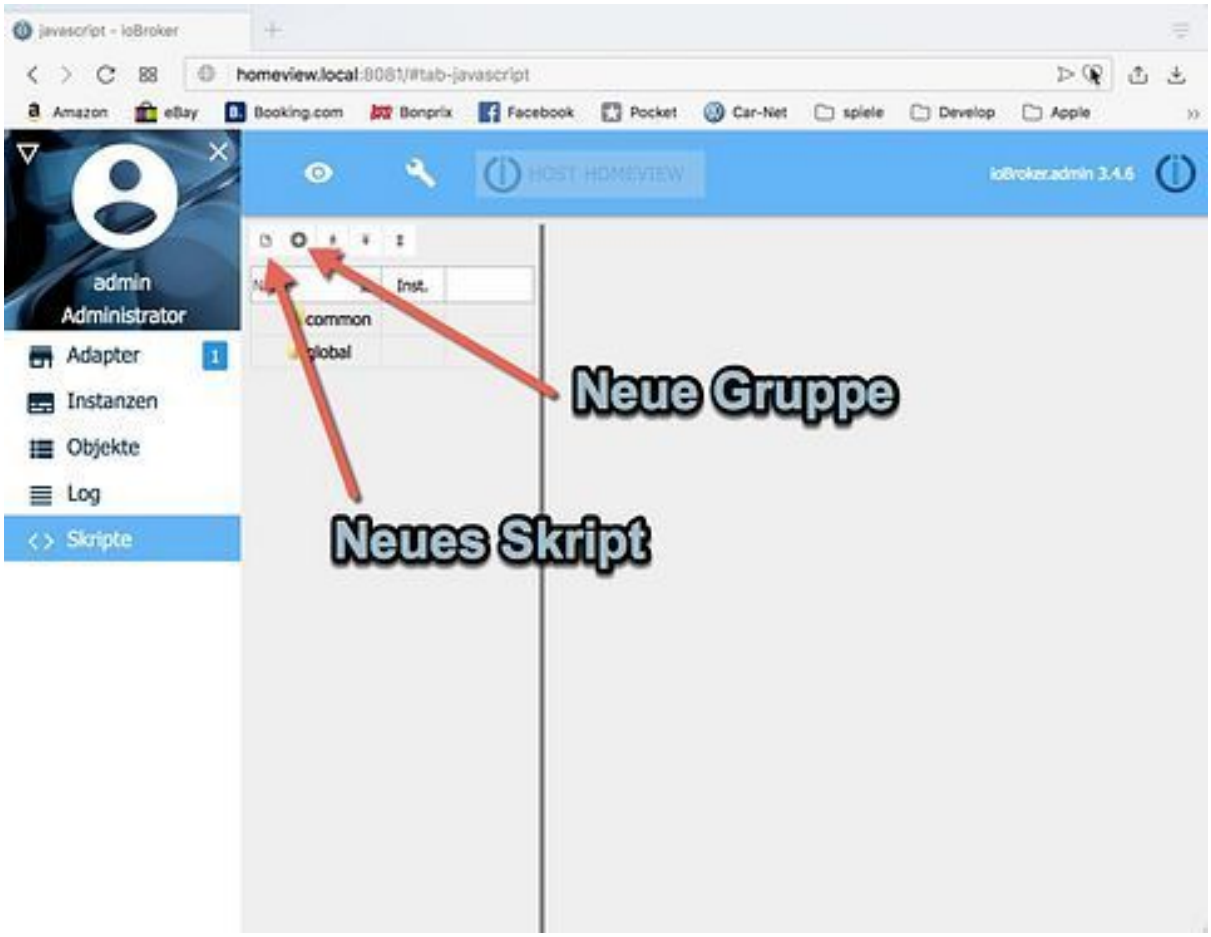



**Abb. 3.1: Scripting-Host**

Klicken Sie dann zunächst auf den runden plus-Button für neue Gruppe und erstellen Sie eine Gruppe namens *Smarthome-diy* (Oder einen Namen Ihrer Wahl). Markieren Sie dann die eben erstellte Gruppe und klicken Sie auf den Button ‘neues Script’. Wählen Sie als “type” dann ‘JavaScript’. Geben Sie dem Script, das nun als “Skript1” aufgelistet ist, den wesentlich fantasievolleren Namen “test” und klicken Sie ›speichern‹.

Schreiben Sie dann in das weiße Feld rechts:

```
log("Hello, World", "info")
```

Klicken Sie auf ›speichern‹, und dann auf den ›Play‹ Button im linken Teilfenster neben dem

Namen des Scripts. Im rechten unteren Teilfenster erscheinen einige Textzeilen, darunter auch eine, die “Hello, World” enthält. Gratuliere, Sie haben Ihr erstes ioBroker Script geschrieben.

Einschub: Möglicherweise ist im log-Feld nichts erschienen. Auch hier hakelt Admin3 manchmal, aber glücklicherweise nur am Anfang. Folgendes kann helfen: Klicken Sie in der linken Spalte auf ‘Log’, um die volle Log-View zu öffnen. Dort sehen Sie die Ausgaben Ihres Skripts auf jeden Fall. Löschen Sie das Log durch Klick aufs rote Papierkorb-Symbol, das vierte Icon von links. Löschen Sie dann noch das lokale Log durch Klick auf das blaue Papierkorb-Symbol daneben. Wenn Sie dann wieder auf ‘Skripte’ gehen, sollte das untere Teilfenster zum Leben erwacht sein. Wenn nicht, starten Sie den Raspi neu.

“log” bedeutet dabei, dass das Skript etwas in die “Log-Datei” schreiben soll. Den Inhalt der Log-Datei sehen Sie jeweils im Reiter **Log** der Admin-Oberfläche. In diese Log-Datei schreiben alle Adapter und Skripte, was sie dem Administrator mitteilen wollen. Das sind Mitteilungen unterschiedlicher Wichtigkeitsstufen, die man als “debug”, “info”, “warn” und “error” bezeichnet. Allgemeine Informationen erhalten die Stufe ›info‹, während Nachrichten, die man nur zum Debuggen benötigt, mit dem Level ›debug‹ geschrieben werden. Ernsthaftere Störungen werden als ›warn‹ ausgegeben, und Fehler, die die normale Funktion unmöglich machen, als ›error‹. Im Log-Fenster kann man einstellen, welche Stufen man sehen will. Ich werde Ihnen das später noch zeigen. Interessant ist für uns, dass wir Log-Meldungen unserer Skripte auch direkt im untersten Teilfenster der Skript-View sehen.

Also zusammengefasst: Unser erstes Skript schreibt “Hello World” mit der Wichtigkeit “info” in die Logdatei, was der Skripting-Host uns gleichzeitig auch hier im Fenster anzeigt.

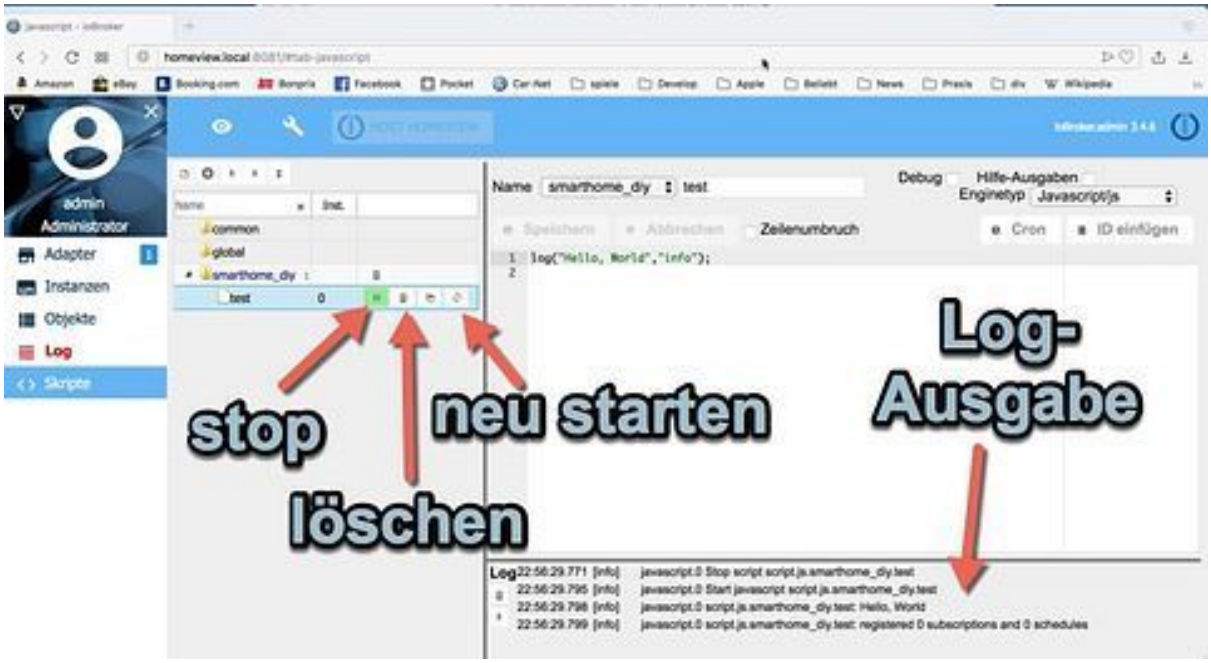



**Abb. 3.1: Log Ausgabe**

Im Folgenden werden wir einige einfache Aufgaben per Skript erledigen.

Doch zunächst eine kurze Erklärung zu den Ordnern, die wir bereits in ‘Skripte’ vorgefunden haben:

- Common ist einfach ein Ablageort, gedacht für allgemein benötigte Code Teile
- Mit ‘Global’ hat es aber eine besondere Bewandtnis: Was hier drin steht, wird an den Anfang jedes anderen Skripts gesetzt. Wir nutzen da gleich aus, um ein paar Konstanten zu definieren:

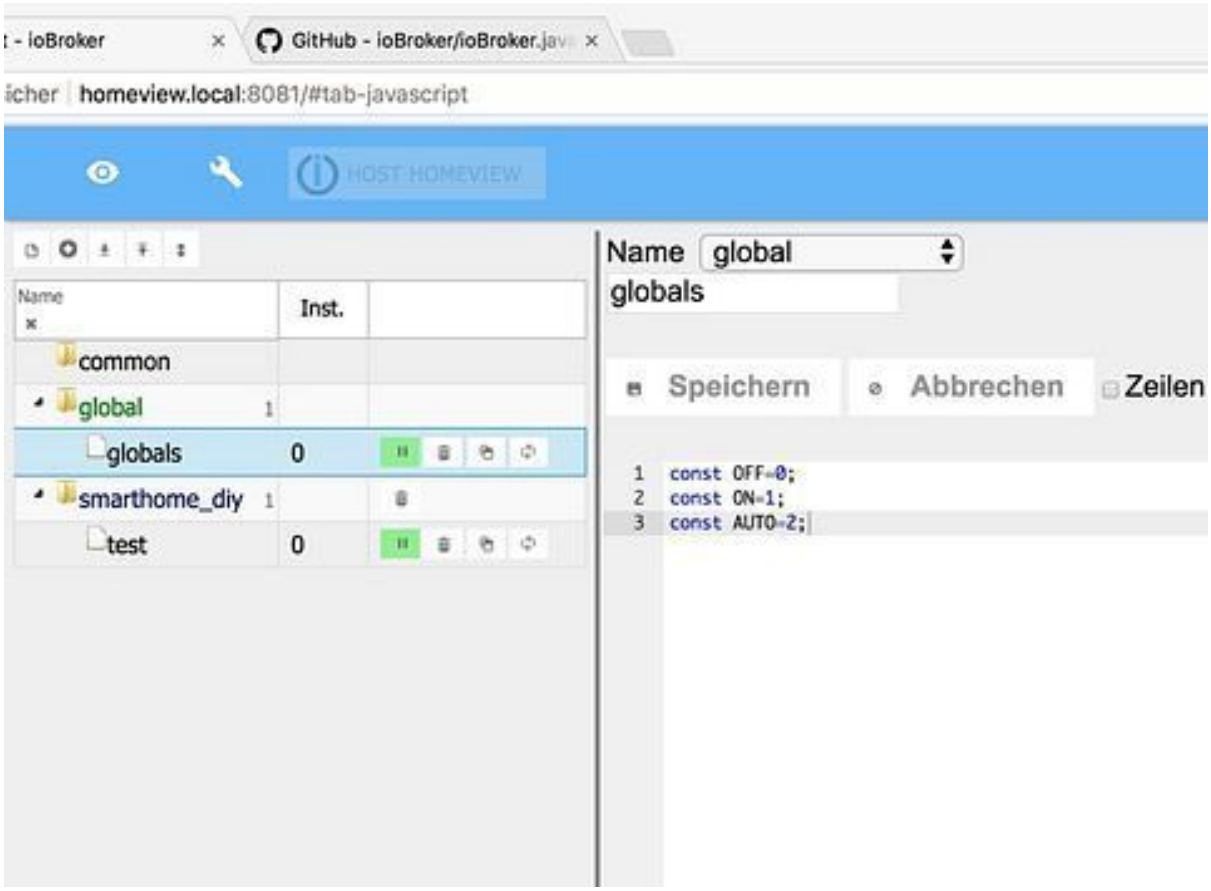



**Abb. 3.2: Globale Skripte**

Wir werden diese Konstanten später benutzen und ggf. erweitern. Beachten Sie, dass das Skript in “global” gestartet (grün) sein muss, damit andere Skripte die Werte benutzen können.

# Einfache Skripte

## Lampe einschalten

Erstellen Sie ein neues Skript und geben Sie ein:

```
setState(
```

Klicken Sie dann rechts oben im Skriptfenster auf ‘ID Einfügen’. Im daraufhin öffnenden Dialog wählen Sie das gewünschte Licht oder die gewünschte Lichtgruppe aus, markieren dort die Eigenschaft ‘on’ und klicken danach ‘Auswählen’.

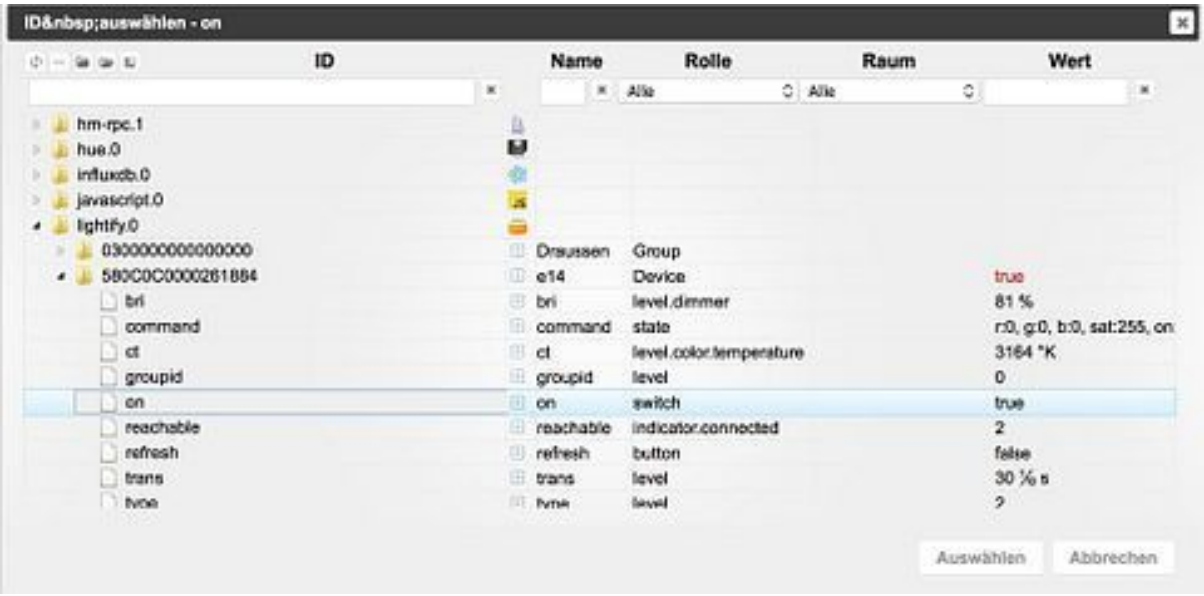



**Abb. 3.3: IoBroker State auswählen**

Im Skriptfenster steht dann etwas wie:

```
setState("lightify.0.663254887.on"/*on*/)
```

IoBroker hat also für uns die manchmal etwas unhandliche Geräte-ID eingefügt (die bei Ihnen selbstverständlich anders aussehen wird, als hier gezeigt). Den Teil zwischen `/*` und `*/` kann man löschen, der ist nur als Information gedacht (und hier überflüssig), dafür muss man aber noch ergänzen, was ioBroker nun genau tun soll:

```
setState("lightify.0.663254887.on", true);
```

Speichern und starten Sie dieses Skript, und wenn Sie alles richtig gemacht haben, wird dadurch das genannte Licht oder die genannte Lichtgruppe eingeschaltet. Wenn Sie ‘false’ statt ‘true’ einsetzen, wird es ausgeschaltet. True und false sind sogenannte bool’sche Konstanten, die nur diese beiden Werte annehmen können, und darum gern für Schaltvorgänge verwendet werden. Man hätte auch 1 und 0 schreiben können, JavaScript (und damit ioBroker) ist da recht tolerant.

## Zwei Lampen miteinander verknüpfen

Nun wollen wir Folgendes erreichen: Wann immer jemand das Licht im Esszimmer einschaltet, soll auch das Licht im Korridor angehen. Das könne man bei Hue und Lightify auch dadurch erreichen, dass man diese beiden Lampen in einer Gruppe zusammenfasst, und dann die Gruppe ein- und ausschaltet, aber die Lösung mit dem Skript ist besser, da es egal ist, in welcher Weise das Esszimmerlicht eingeschaltet wird (Lichtschalter oder App), und da es nur in einer Richtung geht (man kann das Korridorlicht ohne Esszimmer schalten, aber nicht umgekehrt), und nicht zuletzt auch, weil es flexibler ist (Das Korridorlicht kann in mehreren verschiedenen Schaltlogiken eingebunden sein), und die beiden Lichter können zu ganz verschiedenen Gerätearten gehören: Man kann auf diese Weise problemlos Hue, Lightify und myStrom Lampen verknüpfen (und hier sehen Sie nun somit das erste Mal in diesem Buch den 'Mehrwert', den Sie dank der Bemühungen mit ioBroker erzielen).

```
const esszimmerlicht = "lightify.0.64EADA0002261884"
const korridorlicht = "hue.0.Philips_hue.Korridor"
const esszimmer_an_aus = esszimmerlicht + ".on"
const korridor_an_aus = korridorlicht + ".on"
on({ id: esszimmer_an_aus }, function () {
  var State = getState(esszimmer_an_aus).val
  setState(korridor_an_aus, state)
}
```

In den ersten beiden Zeilen weisen wir die unhandlichen Objekt-IDs etwas menschenverständlicheren Variablen zu. Es bewährt sich, das immer so zu halten. Wenn man später einmal eine Lampe auswechselt oder ändern möchte, muss man nur die Zuweisung am Anfang ändern, und nicht das ganze Skript nach Verweisen auf das betroffene Gerät durchsuchen.

Die nächsten zwei Zeilen sind vielleicht etwas schwieriger zu verstehen: Wir addieren ".on" zu einem Variablennamen? Nanu?

Dazu ein kleiner Exkurs, den Sie überspringen können, wenn Sie sich mit JavaScript schon ein wenig auskennen:

Ein besonders vielseitiger Variablentyp ist das "Objekt". Ein Objekt wird so dargestellt:

```
var irgendein_name = {
  frage: "Das Leben etc.",
  antwort: 42,
  attribut3: false,
  attribut4: [1, 2, 3],
  attribut5: { name: "Rumpelstilzchen" }
}
```

Jedes Attribut kann nur einmal vorkommen und einen beliebigen Namen haben, der aus Buchstaben, Zahlen und Unterstrich bestehen darf, aber mit einem Buchstaben beginnen muss. Jeder Wert kann vom Typ String (Zeichenkette), number (Zahl), Array, Boolean oder Object sein.

Attribute innerhalb eines Objekts sind durch Komma getrennt, das ganze Objekt ist immer mit geschweiften Klammern umschlossen.

In ioBroker wird ein Objekt, das einen Zustand eines Gerätes anzeigt, ›State‹ genannt, und solche State-Objekte sind entsprechend aufgebaut, zum Beispiel könnte eine Lampe so definiert sein:

```
const licht = {
  on: true,
  bri: 70,
  r: 255,
  g: 255,
  b: 125,
  reachable: true
}
```

Wenn ich jetzt ›licht.on‹ referenziere, dann meine ich damit das Attribut **on** des Objekts **licht**. Somit ist klar: esszimmer_an_aus ist hier: esszimmerlicht.on.

Die Bedeutung der einzelnen Attribute eines ioBroker-Objekts ist übrigens rein vom Adapter abhängig. Hier bedeutet **on** den Einschaltzustand, **bri** die Helligkeit in Prozent, **r**, **g** und **b** sind die Rot- Grün- und Blau-Anteile des Lichts, als Werte zwischen 0 und 255 ausgedrückt. **Reachable** gibt Auskunft darüber, ob das Gerät überhaupt per Software erreichbar ist (das ist nicht der Fall, wenn es zum Beispiel mit dem Lichtschalter ausgeschaltet wurde). All diese Attribute können wir in ioBroker Skripten auslesen und setzen. Nicht alle Adapter sind gleich gut dokumentiert, und manchmal muss man ein wenig mit den Werten herumspielen, um herauszufinden, was sie bedeuten.

## Zeitschaltuhr

Zum Aufwärmen bilden wir jetzt eine dieser billigen Zeitschaltuhren mit unserem teuren Smarthome-System in Software nach.

Erstellen Sie ein neues Skript namens ‘zeitschaltuhr’ in der Gruppe smarthome-diy.

```
1  const licht ="lightify.0.904AA200AA3EB07C.on"
2
3  schedule("0 17 * * *", function(){
4      log("Schaltuhr ein")
5      setState(licht,true)
6  })
7
8  schedule("30 23 * * *",function(){
9      log("Schaltuhr aus")
10     setState(licht,false)
11 })
```

Eingangs definieren wir die Lampe, die wir steuern wollen. Dann folgen zwei schedule Ausdrücke, die vielleicht noch näher erklärt werden sollten: Damit teilt man ioBroker (bzw. der V8-Engine, auf der ioBroker läuft) mit, dass eine Funktion zu bestimmten Zeiten ausgeführt werden soll. Die allgemeine Syntax ist:

```
schedule("Zeitausdruck", function () { })
```

Der Zeitausdruck kann ganz einfach vom Skripteditor aus eingegeben werden, indem man rechts oben auf ›Cron‹ klickt. Es geht dann dieses Fenster auf:

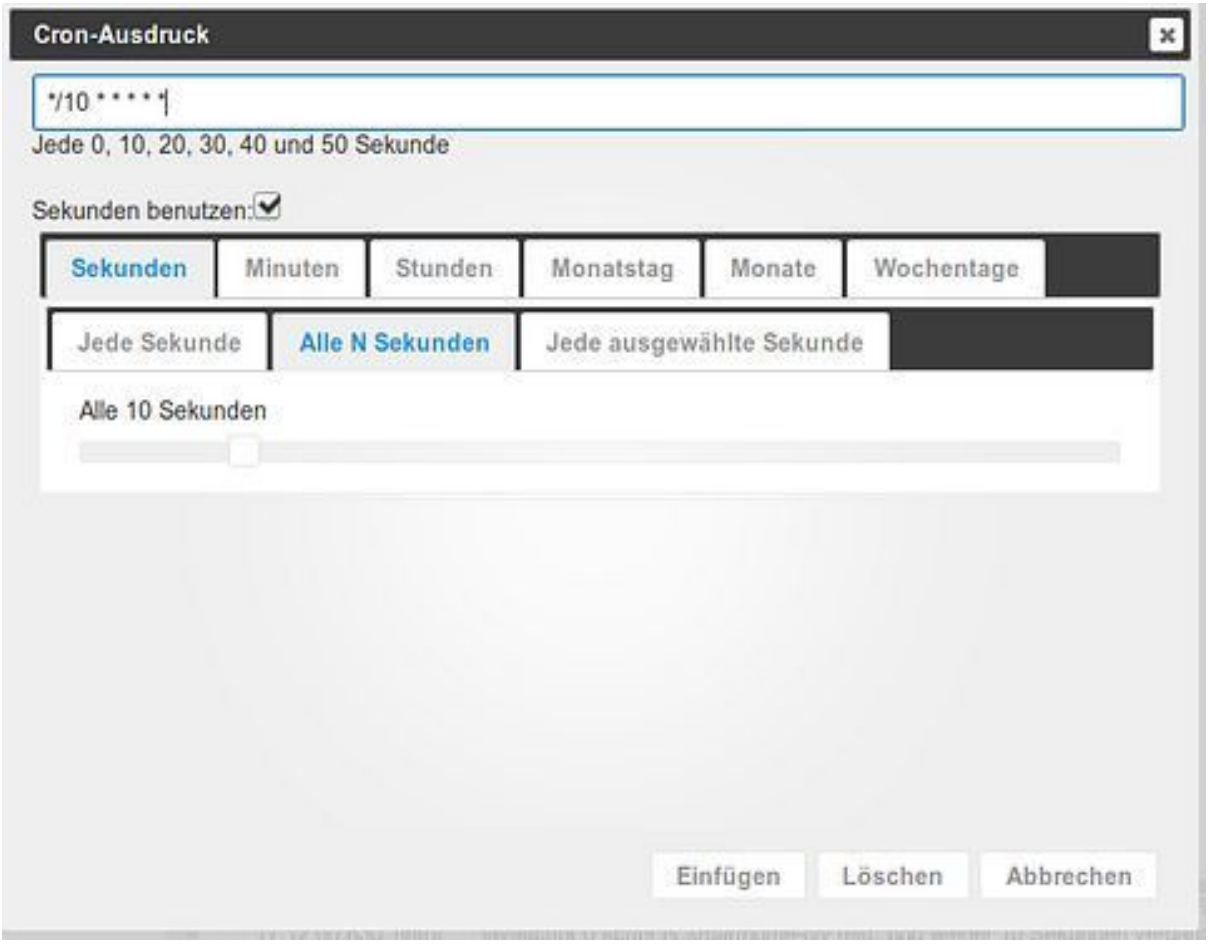



**Abb. 3.5: Cron-Ausdruck per UI**

Hier kann man ganz einfach die gewünschten Zeiten angeben. Die Syntax ist dem Unix cron nachempfunden, daher der Name: Man schreibt 5 oder 6 Sterne (je nachdem, ob Sekunden benutzt werden sollen). der Stern ganz links ist jeweils die kleinste Maßeinheit, dann bedeutet jedes Feld von links nach rechts: (Sekunde, wenn vorhanden), Minute, Stunde, Tag des Monats, Monat, Tag der Woche. So würde etwa "30 3,15 * * *" bedeuten: Um 3:30 und 15:30 Uhr an jedem Tag in jedem Monat, während "0 12 * 5,7,9 1-6" bedeuten würde: Jeden Montag bis Samstag im Mai, Juli und September um 12:00 Uhr. Sie sehen, man kann da ziemlich jeden gewünschten Zeitpunkt ausdrücken. Und glücklicherweise muss man dank der intuitiven Cron-Auswahlbox von ioBroker diese komplizierte Schreibweise gar nicht kennen.

Erweitern Sie unser allererstes Script so:

```
log("Hello, World", "info")
schedule("*/10 * * * * *", function () {
  log("und wieder 10 Sekunden vergangen", "info")
})
```

Wenn Sie es starten, wird es brav alle 10 Sekunden laut geben. Allerdings nicht ganz genau. Cron ist kein Präzisionstimer, sondern wird die Zeiten nur ungefähr einhalten. Wenn es auf Sekundengenauigkeit ankommt, braucht man andere Methoden.

Bewaffnet mit diesen Wissen erkennen Sie nun, dass obige Zeitschaltuhr das Licht um 17:00 Uhr ein- und um 23:30 Uhr wieder ausschaltet, und zwar jeden Tag.

Wir werden das Konzept gleich ein wenig ausbauen:

## Außenbeleuchtung nach Sonnenuntergang bis Mitternacht einschalten

Die Außentreppe ist im Dunkeln ein wenig gefährlich. Damit niemand hinauf- und hinunterstolpert, soll sie wenigstens bis Mitternacht beleuchtet sein. Das soll einerseits automatisch erfolgen, andererseits muss es aber auch möglich sein, manuell einzugreifen, damit man die Treppe etwa für späte Partygäste auch (und gerade!) zu vorgerückter Stunde beleuchten kann.

Erstellen Sie ein neues Skript namens ›aussenlicht_nachts‹ in der Gruppe smarthome-diy.

```
createState("aussenlicht_manuell",AUTO)

const treppenlicht ="lightify.0.904AA200AA3EB07C.on"

function toggle(mode){
    log("toggle "+mode)
    if(getState("aussenlicht_manuell").val==AUTO){
        setState(treppenlicht,mode)
    }
}


on({id: "javascript.0.aussenlicht_manuell", val:OFF}, function(){
    log("manuell aus")
    setState(treppenlicht,false)
})
on({id: "javascript.0.aussenlicht_manuell",val: ON},function(){
    log("manuell an")
    setState(treppenlicht,true)
})
on({id: "javascript.0.aussenlicht_manuell", val:AUTO},function(){
    log("switched to auto")
    if(isAstroDay()){
```

```
24          log("it's day")
25          setState(treppenlicht,false)
26      }else{
27          log("it's night")
28          setState(treppenlicht,compareTime('sunset','23:59','between'))
29      }
30  })
31
32  schedule({astro: "sunset", shift: 15}, function(){
33      log("sunset, lights on")
34      toggle(true)
35  })
36
37  schedule("30 23 * * *", function(){
38      log("23:30, lights off")
39      toggle(false)
40  })
```

Hier geschieht eine ganze Menge Neues. Zunächst erstellen wir mit “createState” einen Software-State. Sie erinnern sich: Ein ‘State’ ist der Zustand eines von ioBroker kontrollierten Objektes, also etwa der Zustand “an” einer Lampe. Wir können in einem Skript aber auch Pseudo-States oder Software-States mit beliebigen Namen erstellen, auf die man dann genau so zugreifen kann, wie auf “echte” States. Hier erstellen wir einen solchen State mit einem initialen Wert von AUTO. AUTO wiederum ist (ebenso wie ON und OFF) eine Konstante, die wir vorhin in unserem globals-skript in der global-Gruppe des Skripting Hosts erstellt hatten.

Nachdem Sie dieses Skript zum ersten Mal ausgeführt haben, finden Sie unter Objekte in der Gruppe javascript.0 einen neuen State namens ‘aussenlicht_manuell’:


**Abb. 3.4: Software-State ›aussenlicht_manuell‹**

Diesen State können Sie genauso wie “echte” States von dieser Admin- Benutzeroberfläche (und später von Visualisierungen und Programmen) aus schalten, wie z.B. den “on” Status einer Lampe.

Dann folgt eine Funktion **toggle**, die das Treppenlicht je nach mode-Argment ein- oder ausschaltet, aber nur, wenn unser vorhin definierter **aussenlicht_manuell** State auf AUTO steht. Wenn der State also nicht auf “Automatik” steht, dann ignoriert diese Funktion die Aufforderung.

Ab Zeile 13 reagieren wir auf manuelles Ausschalten also auf die Situation, dass "irgendwer" den state **aussenlicht_manuell** auf OFF stellt. ioBroker schickt uns dann eine Benachrichtigung, die wir mit einer solchen "on({state},value)" Funktion verarbeiten können. In diesem Fall schalten wir das Treppenlicht aus. Unser Software-State ist jetzt nicht mehr auf AUTO, sondern auf OFF. Künftige automatische Schaltvorgänge werden also nicht ausgeführt.

Ab Zeile 17 kommt dasselbe für den Fall dass der State ON geht.

Ab Zeile 21 verarbeiten wir die Nachricht dass unser State auf AUTO geschaltet wurde. In diesem Fall schauen wir zuerst mal nach, ob es gerade Tag oder Nacht ist.

Dabei kommt eine schöne Funktion von ioBroker zum Zuge: **isAstroDay()** liefert **true** zurück, wenn es Tag ist, sonst **false**. Das ist übrigens auch der Grund, warum ioBroker bei der Installation Ihre Koordinaten wissen wollte. Nur so kann diese Funktion (und andere tageszeitabhängige Funktionen, die wir später noch sehen werden) korrekt funktionieren.

Die letzten beiden Funktionen implementieren nun die Automatik: Um 15 Minuten nach Sonnenuntergang ({astro: sunset, shift 15}) wird das Licht eingeschaltet (falls es im Automatik-Modus ist), und um 23:30 wird es ausgeschaltet (falls es im Automatik-Modus ist).

## Außenbeleuchtung mit Bewegungssensor verknüpfen

Wir wollen das Außenlicht nicht nur zu bestimmten Zeiten einschalten, sondern auch, wenn jemand sich in der Nähe der Treppe befindet. Wir möchten sie also mit einem Bewegungssensor zusammenschalten. Ausserdem soll es, wenn es manuell oder per Sensor aktiviert wurde, heller leuchten, als das Dauerlicht. Und last but not least soll auch ein Licht an der Eingangstür angehen, wenn jemand in die Nähe kommt.

Dazu müssen wir nur relativ wenig ergänzen:

```
1  const DEFAULT_BRI=35;
2
3  createState("aussenlicht_manuell",AUTO)
4
5  const treppenlicht="lightify.0.904AA200AA3EB07C.on"
6  const treppenlicht_bri="lightify.0.904AA200AA3EB07C.bri"
7  const tuerlicht="lightify.0.64EADA0000261884.on"
8  const sensor="hm-rpc.0.NEQ0320745.1.MOTION"
9  const helligkeit="hm-rpc.0.NEQ0320745.1.BRIGHTNESS"
10
11 function toggle(mode){
12     log("toggle "+mode)
13     if(getState("aussenlicht_manuell").val==AUTO){
14         setState(treppenlicht,mode)
15     }
16 }
17
18
19 on({id: "javascript.0.aussenlicht_manuell", val:OFF}, function(){
```

```
20      log("manuell aus")
21      setState(treppenlicht,false)
22      setState(treppenlicht_bri,DEFAULT_BRI)
23      setState(tuerlicht,false)
24  })
25
26  on({id: "javascript.0.aussenlicht_manuell",val: ON},function(){
27      log("manuell an")
28      setState(treppenlicht,true)
29      setState(treppenlicht_bri,90)
30      setState(tuerlicht,true)
31  })
32
33  on({id: "javascript.0.aussenlicht_manuell", val:AUTO},function(){
34      log("switched to auto")
35      setState(treppenlicht_bri,DEFAULT_BRI)
36      setState(tuerlicht,false)
37      if(isAstroDay()){
38          log("it's day")
39          setState(treppenlicht,false)
40      }else{
41          log("it's night")
42          setState(treppenlicht,compareTime('sunset','23:59','between'))
43      }
44  })
45
46  // (1)
47  on({id: sensor, val: true},function(){
48      if(getState(helligkeit).val<90){
49          setTimeout(function(){
50              log("motion sensor timeout")
51              setState("javascript.0.aussenlicht_manuell",AUTO)
52          },120000)
53          log("motion sensor activated")
54          setState(tuerlicht,true)
55          setState(treppenlicht, true)
56          setState(treppenlicht_bri,100)
57      }
58  })
59
60  schedule({astro: "sunset", shift: 15}, function(){
61      log("sunset, lights on")
62      toggle(true)
63  })
64
65  schedule("30 23 * * *", function(){
```

```
66      log("23:30, lights off")
67      toggle(false)
68  })
```

Grundsätzlich hat sich nicht viel geändert, ausser dass beim manuellen Ein- und Ausschalten auch das Türlicht geschaltet wird, und dass ein zusätzlicher setState() auf die Helligkeit des Treppenlichts erfolgt. Im Auto-Modus wird die Helligkeit auf 35% zurückgefahren und das Türlicht ausgeschaltet. Bis jemand in den Erfassungsbereich des Sensors tritt, was in der Funktion bei (1) behandelt wird: Hier wird für 120 Sekunden alles eingeschaltet, falls die Hellgkeit unter einem bestimmten Grenzwert ist. Der Grenzwert von 90 ist empirisch gefunden: Wenn es vor unserer Haustür so dunkel ist, dass man gerne etwas mehr Licht hätte, dann zeigt der Sensor bei uns diesen Wert an.

Die Schaltfunktion bedarf vielleicht einer kleinen Erklärung:

```
 setTimeout(function(){
        setState("javascript.0.aussenlicht_manuell",AUTO)
    },120000)

    setState(tuerlicht,true)
    setState(treppenlicht, true)
    setState(treppenlicht_bri,100)
```

Die JavaScript Standardfunktion setTimeout führt die im ersten Argument genannte Funktion nach der im zweiten Argumenten Zeit (in Millisekunden) aus. Hier wird also nach 120 Sekunden der State für den Aussenlicht-Schalter auf “AUTO” gestellt. Das bewirkt, dass dann die on(...,AUTO) Funktion ausgeführt wird, die alles auf den Normalzustand setzt. Nach dieser Vorbereitung werden *sofort* drei setState-Kommandos abgesetzt, die die Lichter einschalten und auf volle Helligkeit setzen.

## Fernsehbeleuchtung einschalten, wenn der Fernseher läuft

Wenn der Fernseher läuft, sollen die Philips Living-Colors Leuchten angehen.

```
var hue="hue.0.Philips_hue.Wohnzimmer.on";
createState("fernsehlicht_manuell",AUTO)

// (1)
function licht(val){
    if(getState("fernsehlicht_manuell").val==AUTO){
        if(getState("hue.0.Philips_hue.Wohnzimmer.on").val!=val){
            console.log("schalte Licht: "+val,'debug')
            setState(hue,val)
        }
    }
}
```

```
// (2)
on({id: "javascript.0.fernsehlicht_manuell", val:ON},function(){
    setState(hue,true)
})

// (3)
on({id: "javascript.0.fernsehlicht_manuell", val:OFF},function(){
    setState(hue,false)
})

// (4)
schedule({astro: "sunset", shift: -15}, function () {
    log("sunset",'info')
    if(getState("lgtv.0.on"/*TV is ON*/).val){
        licht(true)
    }
});

// (5)
on({id: 'lgtv.0.on', val: true, change: "ne"}, function(){
    log("tv switched on",'info')
    if(!isAstroDay()){
        licht(true)
    }
})

// (6)
on({id: 'lgtv.0.on', val: false, change: "ne"}, function(){
    log("tv switched off",'info')
    licht(false)
})

// (7)
schedule({astro: "sunrise"}, function(){
    log("sunrise")
    licht(false)
})
```

Am Anfang definieren wir, wie nun schon gewohnt, die benötigten Geräte-IDs und den State, den wir zur Kontrolle verwenden wollen. Dann folgt bei (1) die Funktion “licht”, die die Aufgabe hat, dann, im Automatik Modus das Licht ein oder auszuschalten. Bei (2) und (3) werden manuelle Schaltvorgänge verarbeitet. Die Schedule Funktion bei (4) wird nur einmal pro Tag 15 Minuten vor Sonnenuntergang aufgerufen und prüft dann, ob der Fernseher läuft. Wenn ja, schaltet sie das Licht ein. (5) und (6) werden aufgerufen, wenn der Fernseher ein- oder ausgeschaltet wird. (7) schliesslich kommt dann zum Zug, wenn der Fernseher bei Sonnenaufgang immer noch läuft

und schaltet dann das Licht aus.

## Auto laden, wenn genug Strom von der Solaranlage da ist

Vermutlich sind unter den ‘early adoptern’ der Heimautomation auch viele Menschen, die auch in anderen Dingen einen Hang zu moderner Technik haben. Zum Beispiel Photovoltaik oder Elektroautos. Daher wollen wir in dieser Übung den Strom vom Dach, das Auto und die Heimautomation miteinander verknüpfen:

Das Auto soll tagsüber nur dann laden, wenn die Photovoltaikanlage dafür ausreichend Strom liefert. Andernfalls soll es nachts den Niedertarif nutzen.

Wir brauchen also die momentane Leistung des Solardachs und wir brauchen eine Möglichkeit, das Ladegerät des Autos davon abhängig zu schalten. Ersteres ist einfach: Es gibt ja bereits einen ioBroker Adapter für den Fronius Wechselrichter, der unter Anderem auch die Momentleistung und die Netto-Leistung vom/ins Netz ausspuckt. Falls Sie einen Wechselrichter eines anderen Herstellers haben, gibt es auch da vermutlich Möglichkeiten, die Leistungsdaten auszulesen.

Unser Auto lässt sich zwar ebenfalls per Fernsteuerung laden, aber die Schnittstelle dafür ist leider nicht offengelegt, und ausserdem lässt der Hersteller sich die Fernsteuerung recht teuer per Jahresabo bezahlen.

Aber es gibt eine andere Möglichkeit: Man kann das Ladegerät an eine schaltbare Steckdose anschließen. Davon gibt es eine ganze Reihe, zum Beispiel von Homematic etc. Ich hatte aber ein Exemplar von myStrom.

Für dieses existierte zum Zeitpunkt dieses Schreibens noch kein “offizieller” ioBroker Adapter. Ich habe Ihnen im vorigen Kapitel gezeigt, wie man einen inoffiziellen Adapter verwenden kann. Da das myStrom API offengelegt und recht simpel ist, wollen wir die Gelegenheit auch nutzen, einen eigenen ioBroker Adapter für myStrom zu programmieren. Um den Schreib- und Lesefluss hier nicht zu unterbrechen, habe ich das aber in Kapitel 5 ausgelagert. Lesen Sie dort weiter, wenn Sie wissen wollen, wie man einen eigenen Adapter programmiert, oder lesen Sie hier weiter, wenn Sie zunächst sehen wollen, wie die Lade-Logik programmiert wird.

**Theorie**

Wir beginnen damit, uns zu überlegen. wie wir die Schaltvorgänge steuern wollen. Dazu gibt es folgende Bedingungen zu berücksichtigen.

Erstens: Der myStrom Switch ist für 9.9 A Last spezifiziert, und der Hersteller bestätigte mir auf Anfrage, dass dies als mögliche Dauerlast zu verstehen ist. Allerdings zieht der Netzadapter des Autos in Standardeinstellung 10A. Auch wenn das nur ein kleiner Unterschied zu sein scheint, möchte ich das dem filigranen myStrom-Switch nicht zumuten (Haben Sie schon einmal die Temperatur an einem gewöhnlichen Netzstecker gefühlt, durch den einige Stunden lang 10 A, also 2400 Watt geflossen sind?)

Glücklicherweise lässt sich das Ladegerät auch so konfigurieren, dass es maximal 5A, also 1200 Watt zieht. Das geht dann problemlos, allerdings dauern Ladevorgänge dann natürlich doppelt so lang. Da unser typischer Auto-Tagesbedarf aber nur bei 6-8 kWh liegt, ist das gerade noch erträglich.

Zweitens: Vermutlich ist es nicht so gut, wenn das Ladegerät bei jeder vorbeiziehenden Wolke aus- und dann wieder eingeschaltet wird.

Um diese Anforderungen zu erreichen, spezifizieren wir:

Das Ladegerät wird eingeschaltet, wenn ausreichend Strom zur Verfügung stehen, und es zieht dann maximal 1200 Watt.

Wenn es einmal eingeschaltet ist, soll es mindestens 10 Minuten lang eingeschaltet bleiben, egal wieviel Strom hereinkommt.

Danach soll es sich wieder ausschalten, sobald die Solaranlage nicht mehr ausreichend Strom liefert. Frühestens nach 10 Minuten kann es sich wieder einschalten.

**Praxis**

Obige Spezifikation wollen wir nun als Script formulieren. Das erweist sich dann, nach diesen Vorüberlegungen, doch wieder als erstaunlich einfach:

```
var power_pv="fronius.0.powerflow.P_PV" // was wir produzieren
var power_use="fronius.0.powerflow.P_Grid" // was wir vom/zum Netz beziehen/liefe\
rn
var mystrom_switch="mystrom.1.switchState" // Der Schalter

createState("loadcar_manual",2) // Ein, Aus oder Automatik.

/*
 Ladevorgang Ein oder Ausschalten, falls Automatik-Modus
*/
function toggle(mode){
    if(getState("javascript.0.loadcar_manual").val==2){
        if(getState(mystrom_switch).val != mode){
            log("toggle "+mode, 'info')
            setState(mystrom_switch,mode)
        }
    }
}

/*
  manuelles Einschalten
*/
on({id: "javascript.0.loadcar_manual",val: 0},function(){
    log("manual on")
    setState(mystrom_switch,true)
})

/*
  manuelles Ausschalten
*/
on({id: "javascript.0.loadcar_manual",val: 1},function(){
```

```
    log("manual off")
    setState(mystrom_switch,false)
})

schedule("*/10 7-19 * * *",function(){
    var net_flow=getState(power_use).val
    // log("available: "+getState(power_pv).val+", net flow: "+net_flow,'info')
    if( net_flow < -1500){
        toggle(true)
    }else if(net_flow > 0){
        toggle(false)
    }
})

schedule("5 21 * * *", function(){
    log("night schedule: on")
    toggle(true)

})

schedule("59 6 * * *", function(){
    log("day schedule: off")
    toggle(false)
})
```

Die Funktion **toggle** schaltet an oder aus, wenn der Schalter auf ‘automatik’ steht, ganz ähnlich, wie wir es bei der Außenbeleuchtung gemacht haben. Die nächsten zwei Funktionen schalten den Strom absolut ein oder aus. Das können wir brauchen, wenn wir das Auto auch mal aufladen wollen, ohne genug Sonnenstrom zu haben.

Mittels der dann folgenden Schedule-Funktion führen wir alle 10 Minuten zwischen 7 und 19 Uhr Code aus, welcher den Stromfluss vom bzw. zum Netz prüft. Wenn mehr als 1500 Watt exportiert werden, wird das Ladegerät eingeschaltet und zieht dann maximal 1200 Watt. Wenn kein Strom mehr exportiert wird (also die Leistung nicht mehr reicht, um den Verbrauch zu decken), wird die Steckdose wieder ausgeschaltet. Da die Funktion nur alle 10 Minuten ausgeführt wird, werden zu schnelle Schaltvorgänge von vornherein vermieden.

Die nächsten beiden Schedule-Funktionen sorgen dafür, dass die Steckdose jeden Abend um 21:05 eingeschaltet, und jeden Morgen um 06:59 wieder ausgeschaltet wird, falls sie auf ‘Automatik’ steht. (Niedertarif ausnutzen, falls die Sonne des vorigen Tages nicht für volle Ladung gereicht hat).

Den etwas kryptischen Inhalt der “schedule” Ausdrücke können Sie automatisch setzen lassen, wenn Sie oben rechts auf “Cron” klicken.

## Fernseher leise stellen, wenn das Telefon klingelt

Das ist nun sehr einfach:

```
const ringing="tr-064.0.callmonitor.ringing"
const volume="lgtv.0.volume"

on({id: ringing,val:true},function(){
    setState(volume,10);
})
```

## Heizung regeln

Hier sind ein wenig mehr Gedanken notwendig. Die Heizung ist ja, vor allem in ihrer modernen Ausprägung als Bodenheizung, ein vergleichsweise träges Gerät. Wenn wir sie erst dann aufdrehen, wenn die Temperatur unangenehm kühl ist, dann werden wir einige Zeit frieren, und dann schwitzen, weil die Regelung überschießt. Wir müssen zusätzliche Parameter in die Regelung einbeziehen. Dies ist in erster Linie die Außentemperatur, und die Erwartung über deren künftigen Verlauf. Wenn die Aussentemperatur sinkt, müssen wir damit rechnen, heizen zu müssen. Wenn die Vorhersage aber baldiges Steigen prophezeit, müssen wir weniger heizen, als wenn es noch kälter wird. Wenn die Innentemperatur nur sinkt, weil jemand ein Fenster geöffnet hat, dann sollten wir überhaupt nicht nachheizen, sondern warten, bis das Fenster wieder geschlossen ist.

Das sind bereits eine ganze Menge “unscharfer” Parameter, die berücksichtigt werden müssen. Dazu kommt eine ganz wesentliche weitere einzubindende Konstante: Die Qualität der Isolation. Bei exakt gleicher Außen- und Innentemperatur wird der Heizbedarf dennoch je nach Isolation unterschiedlich sein. Wir brauchen also eine Konstante, die dieses Element berücksichtigen kann. Leider ist das nicht wirklich eine Konstante: Bei hoher Differenz von Außen- zu Innentemperatur wird der Wärmeverlust größer sein, als bei kleiner Differenz. Es wird also eine von Haus zu Haus unterschiedliche Kennlinie sein, die den Heizbedarf je nach Außen- und Innentemperatur angibt. Wenn Ihr Haus neuer als vielleicht ca. 20 Jahre ist, wird der Hersteller Ihrer Heizung diese Kennlinie bereits in die Regelung eingespeist haben. Leider nützt das nicht unbedingt viel für eine in der Heimautomation einzubindende Steuerung. Wir wollen eher die Temperatur je Zimmer regeln, während die Heizungsautomatik die gesamte Heizleistung der Anlage anpasst.

Derartige Aufgaben übernimmt in vielen Fällen bereits ein Raum- oder Etagenthermostat. Dieser berücksichtigt aber oben genannte Faktoren gar nicht, sondern vergleicht nur eine Soll- mit einer Ist-Temperatur.

Damit wird unsere Aufgabe ein wenig einfacher: Wir brauchen nicht bei Adam und Eva anfangen, sondern wir müssen nur den existierenden Thermostaten so “aufbohren”, dass er mehr Parameter einbezieht, und dass er zu spezifischen Regelprogrammen je nach Tages- und Nachtzeit, Ferien etc. fähig ist.

Die für jeden zu regelnden Raum gültige Kennlinie kann man nicht errechnen. Zu vielfältig sind die Einflüsse. Man kann der Steuerung aber einen Lernmodus spendieren: Eine Zeit lang soll sie nur beobachten, wie die existierende Regelung bei jeder Außen- und Innentemperatur reagiert, und wie gross der Fehler jeweils ist. Dann beginnt sie, in die Regelung einzugreifen, wobei sie weiterhin versucht zu lernen, in welchen Fällen sie nicht perfekt reguliert hat (nämlich dann, wenn die Ist-Temperatur von der Soll-Temperatur abweicht), und die Kennlinie so laufend anzupassen.

Sie sehen, dieses Projekt wird deutlich aufwändiger, als die Bisherigen. Ich werde es deshalb in mehreren Etappen angehen, und im Rahmen dieses Buches auch nicht ganz vollständig implementieren.

## Wetterdaten bereitstellen

Wir möchten, dass unsere Skripte und später auch externe Programme eine gewisse Vorstellung davon haben können, wie das Wetter ist und wird. Dinge wie Heizungsautomatisierung, Markisensteuerung und Schaltung von grösseren Stromverbrauchern in Abhängigkeit von der erwarteten Windstärke und Sonneneinstrahlung lassen sich dann besser planen.

Es gibt mehrere Wetterdienste, die ein REST-API bieten. Ich habe mehr oder weniger zufällig den Dienst von DarkSky ausgewählt. Ein Hauptvorteil dieses Dienstes ist, dass tausend Abfragen pro Tag kostenlos sind. Das sollte für unsere Zwecke reichen. Bevor man den Dienst nutzen kann, muss man sich allerdings registrieren: https://darksky.net/register. Man erhält dann einen API-KEY, denn man seinen Abfragen mitgeben muss (und den man tunlichst geheim halten sollte).

Um lokale Wetterdaten zu erhalten, benötigt man die exakten Koordinaten als dezimale Längen, und Breitengrade. Die kann man zum Beispiel bei http://mygeoposition.com durch Angabe der Adresse erfahren.

Eine simple Abfrage wie `https://api.darksky.net/forecast/API_KEY/BREITENGRAD,LAENGENGRAD?units=si&l` liefert dann eine ganze Reihe von Wetterdaten im handlichen JSON-Format.

Wir wollen diese Daten alle paar Stunden abholen und für uns relevante Teile davon für interne Abfragen vorhalten.

Für einfacheres Rechnen mit Datum und Zeit verwenden wir die moment.js Library. Diese ist leider nicht standardmässig mit ioBroker installiert. Das wollen wir nun nachholen:

```
ssh pi@homeview.local
cd /opt/iobroker/node_modules/iobroker.javascript
sudo npm install --save moment
```

Vielleicht wundern Sie sich über das Verzeichnis, in das wir wechseln, um moment.js zu installieren. Es ist das Verzeichnis des javasript-interpreters in ioBroker, der wiederum, wie alle Adapter, im node_modules-Verzeichnis der ioBroker Installation gespeichert ist. Generell müssen alle Libraries, die man in Skripten verwenden will, in iobroker.javascript installiert sein.

Dann richten Sie Ihren Browser auf die Skript-Konsole http://homeview.local:8081/#javascript und geben dort folgendes Skript ein (oder fügen es mit copy&paste aus der Begleitsoftware dieses Buchs ein):

**Wetterbericht einlesen**

```
var latitude=46.631094047;
var longitude=7.72370708;
var API_KEY="Sollte geheim bleiben";

const request=require('request')
const moment=require('moment')

const NOW = "wetter.darksky.jetzt."
const TODAY = "wetter.darksky.heute."
const TOMORROW = "wetter.darksky.morgen."

const attribution = "https://darksky.net/poweredby/"
createState("wetter.darksky.lastcall","")
createState(NOW + "temp", 0)
createState(NOW + "bedeckt", 0)
createState(NOW + "wind", 0)
createState(NOW + "niederschlag", 0)


createState(TODAY + "maxtemp", 0)
createState(TODAY + "mintemp", 0)
createState(TODAY + "bedeckt", 0)
createState(TODAY + "wind", 0)
createState(TODAY + "niederschlag", 0)

createState(TOMORROW + "maxtemp", 0)
createState(TOMORROW + "mintemp", 0)
createState(TOMORROW + "bedeckt", 0)
createState(TOMORROW + "wind", 0)
createState(TOMORROW + "niederschlag", 0)


var call = "https://api.darksky.net/forecast/" + API_KEY + "/" + latitude + "," +\
 longitude + "?units=si&lang=de&exclude=minutely,daily,flags,alerts"

console.log("powered by: " + attribution)

const getMinMax = function (range, curr, accum) {
  const currtime = moment(curr.time * 1000)
  if (currtime.isAfter(range[0]) && currtime.isBefore(range[1])) {
    accum.minTemp = Math.min(accum.minTemp, curr.temperature)
    accum.maxTemp = Math.max(accum.maxTemp, curr.temperature)
    accum.wind = Math.max(accum.wind, curr.windSpeed)
    accum.cloudsum = accum.cloudsum + curr.cloudCover
    accum.precipsum = accum.precipsum + curr.precipIntensity * curr.precipProbabi\
```

```
lity
    accum.counter = accum.counter + 1
    accum.cloud = accum.cloudsum / accum.counter
    accum.precip = accum.precipsum / accum.counter
  }
  return accum
}

function fetch() {
  request(call, function (error, response, body) {
    if (error) {
      log.warning("Error! " + error)
    } else {
      if (response && (response.statusCode == 200)) {
        const forecast = JSON.parse(body)
        const today = forecast.hourly.data
        const now = forecast.currently
        setState(NOW + "temp", now.temperature)
        setState(NOW + "bedeckt", Math.round(100 * now.cloudCover))
        setState(NOW + "wind", now.windSpeed)
        setState(NOW + "niederschlag", now.precipIntensity)

        const tsNow = moment(now.time * 1000)
        const tomorrow = tsNow.clone()
        tomorrow.add(1, 'days')
        const spanToday = [tsNow.clone(), tsNow.clone().endOf('day')]
        const spanTomorrow = [tomorrow.clone().startOf("day"), tomorrow.clone().e\
ndOf("day")]
        const accumTemplate = {
          minTemp: 100, maxTemp: -100, wind: -5, cloudsum: 0, counter: 0, precips\
um: 0
        }

        const todayMinMax = today.reduce((accum, curr) => getMinMax(spanToday, cu\
rr, accum)
          , Object.assign({}, accumTemplate))
        const tomorrowMinMax = today.reduce((accum, curr) => getMinMax(spanTomorr\
ow, curr, accum), Object.assign({}, accumTemplate))
        setState(TODAY + "mintemp", todayMinMax.minTemp)
        setState(TODAY + "maxtemp", todayMinMax.maxTemp)
        setState(TODAY + "wind", todayMinMax.wind)
        setState(TODAY + "bedeckt", Math.round(100 * todayMinMax.cloud))
        setState(TODAY + "niederschlag", Math.round(100 * todayMinMax.precipsum))
        setState(TOMORROW + "mintemp", tomorrowMinMax.minTemp)
        setState(TOMORROW + "maxtemp", tomorrowMinMax.maxTemp)
        setState(TOMORROW + "wind", tomorrowMinMax.wind)
```

```
92          setState(TOMORROW + "bedeckt", Math.round(100 * tomorrowMinMax.cloud))
93          setState(TOMORROW + "niederschlag", Math.round(100 * tomorrowMinMax.preci\
94  psum))
95          setState("wetter.darksky.lastcall",tsNow.toString())
96        } else {
97          console.log("no response")
98        }
99      }
100   })
101 }
102
103 schedule("20 4,8,11,14,17,20,23 * * *",fetch)
```

Ganz am Anfang kommen die Daten zu Koordinaten und API-KEY, die Sie für Ihre Gegebenheiten anpassen müssen.

Dann muss man sich überlegen, welche der Angaben von DarkSky man überhaupt benötigt. Ich habe mich hier entschlossen, nur die momentanen Daten, sowie die Tageshöchst- und Tiefstwerte der Temperatur für den Rest des heutigen und des ganzen morgigen Tages zu speichern, ausserdem die maximale Windgeschwindigkeit, den durchschnittlichen Bewölkungsgrad und schließlich die erwartete Niederschlagsmenge und -wahrscheinlichkeit. Für jede dieser Positionen wird ein State erstellt. Die Attribution geben wir auf der Konsole aus, da die Lizenzbedingungen von DarkSky verlangen, dass man die Herkunft der Daten deklariert. Ein kleiner Preis für so einen Service.

Die Funktion getMinMax() in Zeile 30ff werden wir später nutzen, um die jeweiligen Daten aus einem zu übergebenden Zeitraum zu fischen. Dann kommt in der Funktion fetch() ab Zeile 44 die eigentliche Arbeit: Mit “request” setzen wir einen REST-Call an DarkSky ab, und wenn die Antwort Erfolg signalisiert, parsen wir deren body nach JSON. Dann kommt ein wenig Zeit-Rechnerei, um Anfang und Ende der interessierenden Zeiträume zu definieren, und mit diesen (spanToday und spanTomorrow) füttern wir eine reduce-Funktion, welche die vorhin genannte getMinMax()-Funktion als Parameter erhält. Reduce gehört zu den Funktionen, die man fast bei jeder funktionalen Sprache findet. Sie “reduziert” eine Collection auf einen einzigen Wert, indem sie auf jedes Element dieselbe Funktion anwendet, die jeweils das bisherige Ergebnis (accum) und das momentane Element (curr) als Parameter erhält. Was wir am Ende erhalten, ist todayMinMax resp. tomorrowMinMax, also zwei Objekte mit den jeweils zu speichernden Extrakten aus den Gesamtdaten. Diese Extrakte schreiben wir dann in die vorhin definierten States.

Ganz am Ende sorgen wir dafür, dass fetch() 7 Mal pro Tag ausgeführt wird, damit die Daten einigermassen aktuell sind (Das ist natürlich eine vollkommen willkürliche Zahl, die weit genug von den 1000 Abfragen weg ist, die DarkSky erlaubt, so dass Sie sie gern auch erhöhen dürfen.)

Vielleicht ist Ihnen aufgefallen, dass der Ausdruck am Ende lautet:

```
schedule("* 4,8,11,14,17,20,23 * * *",fetch)
```

und nicht etwa:

```
schedule("* 4,8,11,14,17,20,23 * * *",fetch())
```

Man muss als Callback den Namen der Funktion angeben, und nicht etwa die Funktion ausführen. Also immer ohne Klammerpaar schreiben.

Und weiter? Nun, die so erzeugten States können wir mit anderen Skripten oder von aussen auslesen. Ich werde das später noch zeigen.

## SMS Warnung

Es kommt zwar nicht sehr häufig vor, aber stellen Sie sich vor, während Ihres Skiurlaubs fällt die Heizung aus. Die Leitungen frieren ein, und wenn Sie nach Hause kommen, empfängt Sie eine Überschwemmung. An sich hätten Sie ja eine Fernabfrage für alle Bestandteile Ihres Smarthomes gehabt, aber Sie hatten nicht daran gedacht, jeden Tag darauf zu schauen. Wieso auch.

Das ist verhinderbar, wenn Ihr Smarthome Ihnen bei Problemen aller Art eine SMS schickt. Die werden Sie vermutlich bemerken.

Aber wie kann der Raspberry Pi eine SMS versenden? Nun, dafür gibt es mehrere Möglichkeiten. Beispielsweise könnten Sie ihn via Bluetooth oder Kabel mit einem nicht mehr gebrauchten Handy oder einem GSM-Modem verbinden. Allerdings brauchen Sie dann immer noch eine SIM-Karte, die mehr kostet, als die Lösung, die ich Ihnen im Folgenden vorstellen werde. Falls Sie doch den Weg übers Handy gehen wollen, liefere ich Ihnen das Stichwort ›Gnokii‹ zum weiter googeln.

Ich werden Ihnen hier aber den Weg über einen SMS Service zeigen. Von diesen gibt es mehrere (Such-Stichwörter sind z.b. ›sms api‹ pder ŝeb sms‹), allen gemeinsam ist, dass man über ein vorgegebenes API via Internet darauf zugreifen und SMS versenden kann. Ich verwende hier aspsms.com, weil es unter Anderem ein nodejs.api hat, und weil man einige kostenlose SMS zum Testen bekommt, und weil einmal gekaufte SMS Credits ungegrenzt gültig bleiben. Der Preis einer SMS ist bei ungefähr 8 cents.

Als Erstes müssen Sie einen Account bei https://www.aspsms.com/de/registration/ eröffnen. Sie bekommen dann einen Userkey und ein Passwort. Mit diesen Credentials kann der Raspberry dann auf das API zugreifen.

Zunächst installieren Sie das NodeJS API:

```
ssh pi@homeview.local
cd /opt/ioBroker
npm install mod-aspsms
```

— tbd —

# Skripte sichern

Nachdem wir jetzt schon so viel Arbeit in unsere Skripte gesteckt haben, stellen Sie sich vermutlich die Frage, was eigentlich bei einem Defekt des Raspberry oder der SD-Karte geschieht. Nun, wenn die SD-Karte beschädigt wird, dann sind die Skripte futsch. Und leider geht eine SD-Karte mit Sicherheit früher oder später kaputt, denn sie erträgt prinzipbedingt nur eine begrenzte Zahl von Schreibzugriffen. Der Wert der Skripte für Sie dürfte zu diesem Zeitpunkt den Wert der SD-Karte um ein Vielfaches übersteigen. Stellen Sie sich vor, Sie haben Ihr Smarthome perfekt eingerichtet, wie viel Zeit würde es Sie wohl kosten, alle Skripte wieder zu rekonstruieren? Eben.

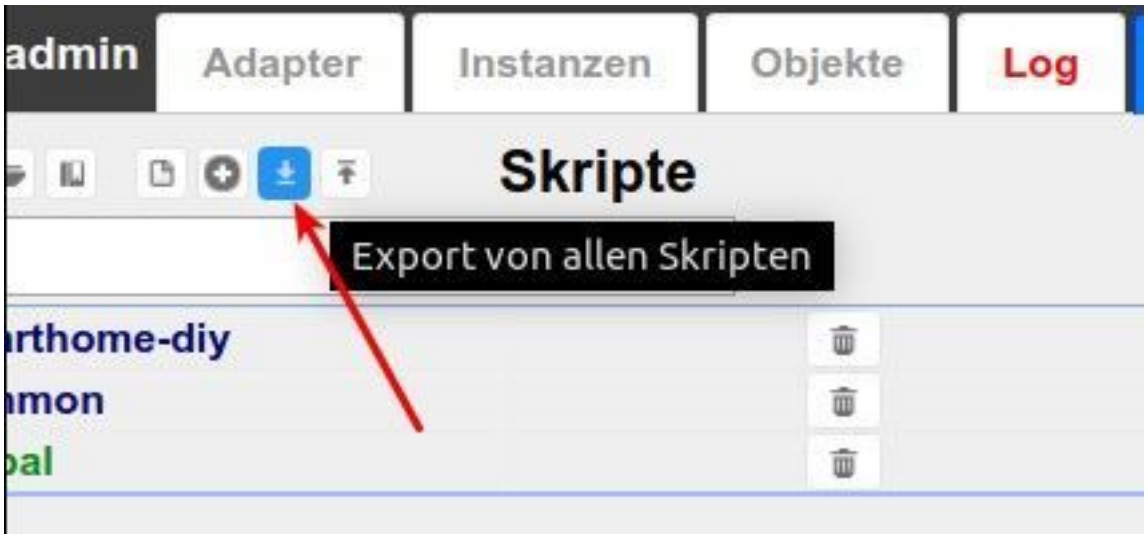



**Abb. 3.6: Skripte exportieren**

Glücklicherweise ist guter Rat hier ausnahmsweise billig (und im Preis dieses Buches bereits enthalten): Klicken Sie im Skripte Fenster einfach auf den Button ‘Export von allen Skripten’. Diese werden dann gezippt, je nach Einstellung Ihres Browsers entweder mit oder ohne Frage nach dem gewünschten Speicherort, als Backup heruntergeladen. Falls Sie Ihren Heimserver jemals neu aufsetzen müssen, können Sie die so gespeicherten Skripte ebenso einfach mit dem daneben liegenden Button “Import von allen Skripten” im neuen ioBroker wieder installieren. Sie müssen nur noch dafür sorgen, dass die heruntergeladene .zip Datei an einem sicheren und wiederauffindbaren Ort gespeichert wird. Idealerweise bewahrt man mehrere Generationen von mehreren Daten auf, damit man einen eventuell irgendwann eingeschlichenen Fehler ausbügeln kann.

# Kapitel 4: Standalone Front-End

... Auszug ...

---

## Echte Messwerte

Bisher haben wir ja nur Werte angezeigt, die unser “Mock” lieferte. Nun wird es Zeit, dass wir uns echte Messwerte anzeigen lassen. Ich gehe im Folgenden davon aus, dass Sie irgend ein Messgerät in ihrer ioBroker Konfiguration eingebunden haben. Hier benutzen wir zur Demonstration ein Homematic IP Innenthermometer / Hygrometer ‘HmIP-STH’ (z.B. hier: https://www.elv.ch/homematic-ip-temperatur-und-luftfeuchtigkeitssensor-innen.html).

Damit ioBroker seine ‘States’ überhaupt an ein Fremdprogramm herausrückt, muss man eine entsprechende Schnittstelle installieren. Gehen Sie auf die ioBroker Admin Seite (homeview.local:8081) und installieren Sie den “SimpleAPI Adapter” aus der Gruppe “Kommunikation. Dieser Adapter bietet auf Port 8087 ein REST API zum Lesen und Schreiben von States an. Lassen Sie bei der Installation alles auf Default-Werten. Zum Testen können Sie nach der Installation folgendes in Ihren Browser eingeben:

```
http://homeview.local:8087/get/system.adapter.simple-api.0.uptime?prettyPrint
```

Das Resultat sollte ein JSON Objekt ähnlich wie dieses sein:


```
{
  "val": 785918,
  "ack": true,
  "ts": 1513019750949,
  "q": 0,
  "from": "system.adapter.simple-api.0",
  "lc": 1513019750949,
  "_id": "system.adapter.simple-api.0.uptime",
  "type": "state",
  "common": {
    "name": "simple-api.0.uptime",
    "type": "number",
    "read": true,
    "write": false,
    "role": "indicator.state",
    "unit": "seconds"
  },
  "native": {}
}
```

Wenn Sie statt ‘system.adapter.simple-api.0.uptime’ die ID eines Ihrer Geräte-States eintragen, wird der Zustand dieses States angezeigt. Damit Sie die länglichen IDs nicht abtippen müssen, können Sie sie durch Klick auf “kopieren” direkt in die Zwischenablage nehmen und in den Programmeditor einfügen.


**Abb. 4.10: State copy&paste**

Jetzt können Sie in app.ts das echte Gerät eintragen:

```
 constructor(private ea:EventAggregator){

    setInterval(()=>{
        this.fetcher.getIobrokerValue("hm-rpc.1.000F570AA11B84.1.ACTUAL_TEMPERATU\
RE").then(result=>{
        this.ea.publish("temperatur", result)
        },reason=>{
          alert("an error occured "+reason)
        })
    },1000)
  }
```

Allerdings werden Sie, wenn Sie das Programm laufen lassen, immer noch dasselbe wilde Hüpfen des Zeigers sehen, wie zuvor. Das liegt daran, dass der FetchService auch die Variable env.mock beachtet. Die müssen wir nun auf false setzen. Das tun wir in aurelia_project/environments/-dev.ts:

```
export default {
  debug: true,
  testing: true,
  mock: false,
  iobroker: 'http://homeview.local:8087'
};
```

Bei dieser Gelegenheit haben wir auch gleich die korrekte Adresse und Port des ioBroker REST API eingetragen.

(Diesen Stand des Projekts kann ich Ihnen nicht zum Auschecken anbieten, da ich ja nicht weiß, wie die IDs Ihrer Geräte sind. Ich werde in diesem Buch daher weiterhin mit Mock-Werten arbeiten.)

Wenn Sie das Programm jetzt starten, wird der Zeiger unserer CircularGauge nicht mehr so fröhlich umherspringen, sondern uns die reale Temperatur anzeigen. Um die Netzwerkbelastung

und den Stromverbrauch nicht unnötig hoch zu lassen, würde ich in app.ts nun auch das Abfrageintervall von 1000 auf etwa 60000 hoch setzen, damit das Thermometer nur noch jede Minute statt jede Sekunde ausgelesen wird.

Natürlich können Sie auch problemlos zwei oder mehr CircularGauges als Thermometer in die Site einbinden. Allerdings werden Sie dann auf ein Problem stossen: Da alle CircularGauges auf dieselbe Nachricht - "temperatur" vom EventAggregator abonniert sind, werden alle Anzeigen auf alle Thermometer reagieren. Wir müssen also auch die Nachricht parametrisierbar machen. Das erreichen wir mit einem zusätzlichen Attribut "message" in cfg, und einer kleinen Anpassung in der attached() Funktion der CircularGauge:

```
attached() {
  this.configure()
  this.body = select(this.element).append("svg:svg")
    .attr("class", "circulargauge")
    .attr("width", this.cfg.size)
    .attr("height", this.cfg.size)
  this.render()
  this.ea.subscribe(this.cfg.message, data => {
    this.redraw(data)
  })
}
```

"Subscribe" geht jetzt auf den in this.cfg.message festgelegten String, anstatt auf den hartcodierten Text "temperatur". So können wir in der Konfiguration jedem Thermometer eine eigene Nachricht mitgeben, auf die es lauschen soll.

Um eine ganze Website mit Innen- und Aussenthermometer zu demonstrieren, brauchen wir eine "vernünftige" Möglichkeit, die IDs er interessierenden Geräte irgendwo zu sammeln. Das Verteilen dieser IDs im ganzen Code ist *keine* gute Idee, wie Sie spätestens beim Ersetzen eines Geräts nach einem halben Jahr oder so merken würden, wenn Sie keine Ahnung mehr haben, wo im Code die ID für dieses Gerät sich befindet. Stattdessen ergänzen wir unsere Environment-Definitionen config/environment.json und config/environment.production.json um einen Abschnitt "devices", so dass Sie jetzt so aussehen:

**config/environment.json**

```
{
  "debug": true,
  "testing": true,
  "mock": false,
  "iobroker": "http://homeview.local:8087",
  "influx": "http://homeview.local:8086",
  "devices":{
    "barometer": "mqtt.0.Wetter.Wohnzimmer.Luftdruck",
    "aussen_temp": "hm-rpc.0.OEQ00XXXX4.1.TEMPERATURE",
    "aussen_hygro": "hm-rpc.0.OEQ00XXXX4.1.HUMIDITY",
    "wohnzimmer_temp": "hm-rpc.1.000E5569AXXXXE.1.ACTUAL_TEMPERATURE",
```

```
    "wohnzimmer_hygro": "hm-rpc.1.000E5569AXXXXE.1.HUMIDITY",
    "dusche_temp": "hm-rpc.1.000E5709AXXXX4.1.ACTUAL_TEMPERATURE",
    "dusche_hygro": "hm-rpc.1.000E5709AXXXX4.1.HUMIDITY",
    "dachstock_temp": "hm-rpc.1.000E5709AXXXX3.1.ACTUAL_TEMPERATURE",
    "dachstock_hygro": "hm-rpc.1.000E5709AXXXX3.1.HUMIDITY",
    "treppenlicht_direkt": "lightify.0.904AA200AA3XXXXC.on",
    "treppenlicht_modus": "javascript.0.aussenlicht_manuell" ,
    "fernsehlicht_direkt": "hue.0.Philips_hue.Wohnzimmer.on",
    "fernsehlicht_modus": "javascript.0.fernsehlicht_manuell",
    "helligkeit": "hm-rpc.0.NEQ0320745.1.BRIGHTNESS",
    "esstisch_helligkeit": "hue.0.Philips_hue.Esstisch.bri",
    "esstisch_schalter": "hue.0.Philips_hue.Esstisch.on",
    "_car_loader_manual": "javascript.0.loadcar_manual",
    "_car_loader_state": "mystrom-wifi-switch.1.switchState",
    "_car_loader_power": "mystrom-wifi-switch.1.power",
    "ACT_POWER": "fronius.0.powerflow.P_PV",
    "DAY_ENERGY": "fronius.0.inverter.1.DAY_ENERGY",
    "YEAR_ENERGY": "fronius.0.inverter.1.YEAR_ENERGY",
    "TOTAL_ENERGY": "fronius.0.inverter.1.TOTAL_ENERGY",
    "energy_grid_flow": "fronius.0.powerflow.P_Grid",
    "MAX_POWER": 10000,
    "MAX_DAILY_ENERGY": 70000
  }
}
```

(Beachten Sie die noch nicht benötigten Einträge einfach nicht :-))

Auf diese Weise muss ich, im Fall eines späteren Gerätetauschs, nur an einer Stelle nachsehen und ändern.

Sie erhalten den Quellcode dieses Teils, wenn Sie eingeben:

```
git checkout -f origin/stufe_05
git clean -f
npm install
```

Doch hier die wichtigsten Codeänderungen:

In app.html setzen wir zwei Anzeigen nebeneinander:

**src/app.html**

```
<template>
  <require from="components/circulargauge"></require>
  <div class="container">
    <h1 class="h1">Temperatur-Demo</h1>
    <div class="row">
      <div class="col">
        <h2>Aussen</h2>
        <circular-gauge cfg.bind="aussentemp_cfg"></circular-gauge>
      </div>
      <div class="col">
        <h2>Wohnzimmer</h2>
        <circular-gauge cfg.bind="wohnzimmertemp_cfg"></circular-gauge>
      </div>
    </div>
</div>
</template>
```

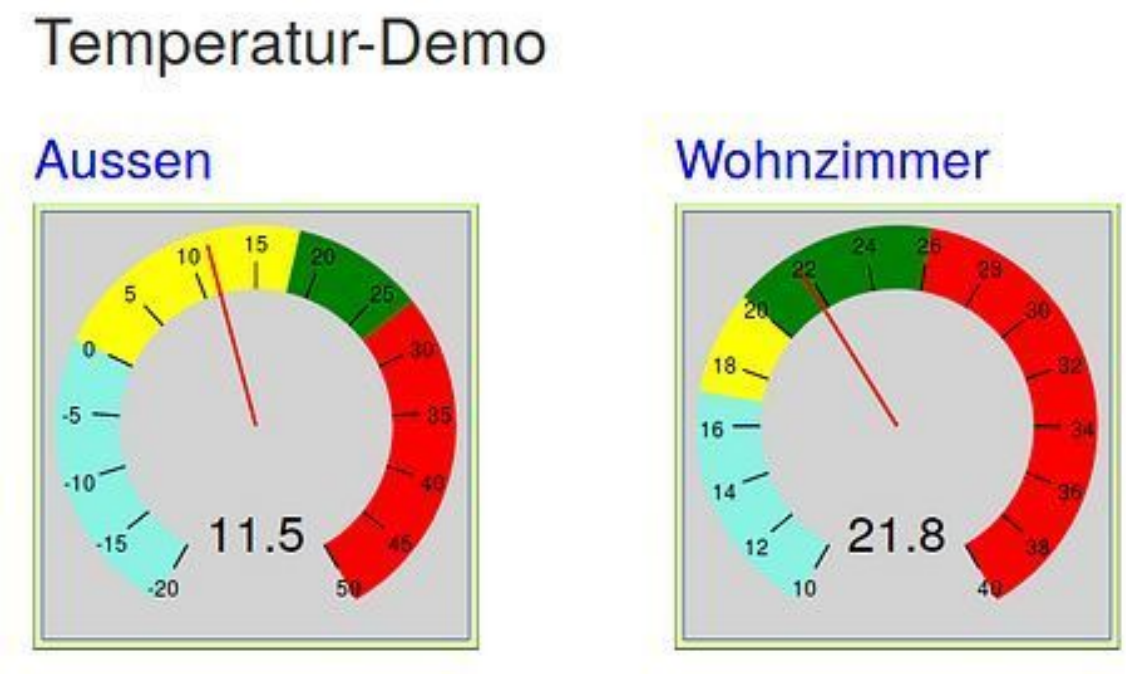



**Abb. 4.11: Zwei Anzeigen**

Hier sehen Sie zwei unterschiedlich parametrisierte CircularGauges nebeneinander. Der entsprechende Code in app.ts braucht vielleicht ein wenig Erläuterung:

**src/app.ts**

```
import {FetchService} from './services/fetchservice'
import {autoinject} from 'aurelia-framework'
import  {EventAggregator} from 'aurelia-event-aggregator'
import * as env from '../config/environment.json'
const dev=env.devices


@autoinject
export class App {
  message = 'Hello World!'
  fetcher=new FetchService()
  wohnzimmertemp_cfg={
    "device": dev.wohnzimmer_temp,
    "size":250,
    bands: [{ from: 10, to: 17, color: "#8cf2e4" },
          {from: 17, to: 20, color: "yellow"},
          {from: 20, to: 26, color: "green"},
          {from: 26, to: 40, color: "red"}],
    MAX_ANGLE:300,
    min: 10,
    max: 40,
    message: "wohnzimmer_temp"
  }
  aussentemp_cfg={
    "device": dev.aussen_temp,
    "size":250,
    bands: [{ from: -20, to: 0, color: "#8cf2e4" },
          {from: 0, to: 18, color: "yellow"},
          {from: 18, to: 27, color: "green"},
          {from: 27, to: 50, color: "red"}],
    MAX_ANGLE:300,
    min: -20,
    max: 50,
    message: "aussen_temp"
  }
  constructor(private ea:EventAggregator){
    let devices=[this.wohnzimmertemp_cfg,this.aussentemp_cfg]
    setInterval(()=>{
        this.fetcher.getIobrokerValues(devices.map(dev=>dev.device))
          .then(results=>{
            for(let i=0;i<results.length;i++){
              this.ea.publish(devices[i].message,results[i])
            }
          },reason=>{
            alert("an error occured "+reason)
```

```
            })
    },10000)
  }
}
```

Wir erstellen zwei Konfigurationsobjekte, wohnzimmertemp_cfg und aussentemp_cfg, die die beiden Geräte referenzieren, und die unterschiedliche Anzeigebereiche und unterschiedliche message-attribute haben. Im constructor packen wir die beiden Objekte in ein Array namens devices. Danach kommt die setInterval-Funktion, die wir schon früher betrachtet haben. Darin rufen wir die `getIoBrokerValues()`-Funktion des FetchServices aus. Diese holt in einem Rutsch beliebig viele States, deren IDs es in einem Array als Parameter erwartet. Dieses Array erstellen wir on the fly mit `devices.map(dev=>dev.device)` (Die Javascript-Standardfunktion map erstellt ein Resultat-Array, indem es auf jedes Element des Ursprungs-Arrays eine Operation anwendet, hier dev.device. Es entsteht also ein Array aus Strings, welche die "device"-Attribute jedes Elements von "devices" sind). Dieses Array ist dann das Argument für `getIoBrokerValues()`. Als Rückgabewert erhalten wir eine Promise, welche wieder zu einem Array resolved, diesmal einem Array der Resultate, in derselben Reihenfolge, wie die Elemente des Eingangs-Arrays. Diese Resultate lesen wir aus und schicken sie über den EventAggregator zum passenden Empfänger.

## Komplexe Figuren erstellen und rotieren

Ich möchte die Entwicklung unserer CircularGauge mit einer letzten Verschönerung abschliessen: Der Zeiger soll nicht einfach nur ein Strich sein, sondern, eben ein Zeiger. Das gibt mir Gelegenheit zu zeigen, wie man Formen jenseits von einfachen geometrischen Figuren mit D3 resp. SVG erstellen kann, und vor allem, wie man solche Figuren verschiebt und rotiert, so dass es natürlich aussieht.

Sie erhalten diesen Stand mit:

```
git checkout -f origin/stufe_06
git clean -f
npm install
```

Für komplexe Figuren hält SVG die Elemente *Polygon*, *Polyline* und *Path* bereit, wobei *Path* das bei Weitem flexibelste ist. Wir hatten es schon bei der arc-Funktion unseres Helper-Objekts kennengelernt, dort hat allerdings D3 uns die Details abgenommen, und wir mussten nur die Parameter des Kreisbogens angeben. Diesmal werden wir selber die Ärmel hochkrempeln und *Path* von Hand bedienen.

Für die Definition eines *Path* benutzt SVG eine einfache Beschreibungssprache. Falls Sie TurtleGraphics oder Logo kennen, wird Ihnen das bekannt vorkommen, aber auch ohne dies ist es nicht besonders schwierig. Die wichtigsten Befehle[1] sind:

- M - Moveto
- L - Lineto

[1] Eine vollständige Befehlsreferenz finden Sie z.B. bei https://www.w3schools.com/graphics/svg_path.asp

- C - Curveto
- A - Arc
- Z - Close Path

Ein Rechteck von 10/10 nach 30/20 könnte man zum Beispiel so definieren:

```
M 10 10 L 30 10 L 30 20 L 10 20 Z
```

Die Beschreibung darf im Prinzip beliebig lang sein, aber es ist klar, dass es doch arg unübersichtlich und schwer korrigierbar wird, wenn sich so eine Anweisung über eine halbe Seite erstreckt. Dann ist man froh, wenn ein Toolkit wie D3 die Feinarbeit übernimmt.

Für unseren Zeiger reicht aber Handarbeit:

In unserer render() Funktion definieren wir einige Konstanten für den Zeiger, und mit pointer_stroke den Path. Dann erstellen wir ein "g" Element. Die g (group) Elemente dienen in SVG dazu, andere Komponenten zusammenzufassen. Beachten Sie, dass wir dieses g-Element nach der Erstellung sofort nach (center,center) verschieben, und um unsere Skalenrotation drehen. Dies ist darum notwendig, weil SVG Rotationen sich stets auf den Ursprung (0,0) des übergeordneten Elements beziehen. Wenn Sie sehen wollen, was ich meine, kommentieren Sie den transform-Ausdruck einfach mal aus und starten Sie das Programm.

In dieses g Element malen wir dann den Pointer und einen kleinen Kreis ums Zentrum.

**src/components/circulargauge.ts**

```
  render() {
    // basic setup
    let dim = this.cfg.size
    let center = dim / 2
    let size = (dim / 2) * 0.9

    const pointer_width = 10
    const pointer_base = 0.3
    const pointer_stroke = `M ${-size * pointer_base} 0
    L 0 ${-pointer_width / 2}
    L ${size * (1 - pointer_base)} 0
    L 0 ${pointer_width / 2}
    Z`


    this.hlp.frame(this.body, dim)

    /*
      Draw the coloured bands for the scale
    */
    this.cfg.bands.forEach(band => {
      this.hlp.arch(this.body, center, center, size - this.arcsize, size,
        this.hlp.deg2rad(this.scale(band.from)),
        this.hlp.deg2rad(this.scale(band.to)), band.color, this.rotation())
```

```
    })

    // Draw the pointer
    let pframe = this.body.append("g")
      .attr("transform",
      `translate(${center},${center}) rotate(${this.rotation() - 90})`)
    this.pointer = pframe.append("g")
    this.pointer.append("svg:path")
      .attr("d", pointer_stroke)
      .classed("pointer", true)
    this.pointer.append("svg:circle")
      .attr("cx", 0)
      .attr("cy", 0)
      .attr("r", 8)


    /* field for actual measurement */
    let valuesFontSize = Math.round(size / 4)
    this.valueText = this.hlp.stringElem(this.body, center, center + size / 2,
      valuesFontSize, "middle")

    /* create tickmarks  */
    let tickmarkFontSize=this.arcsize/3
    this.scale.ticks(15).forEach(tick => {
      let p1 = this.valueToPoint(tick, 1.2)
      let p2 = this.valueToPoint(tick, 1.0)
      let p3= this.valueToPoint(tick,1.35)
      this.hlp.line(this.body, center - p1.x, center - p1.y, center - p2.x, cente\
r - p2.y, "black", 1.2)
      this.hlp.stringElem(this.body,center-p3.x,center-p3.y,tickmarkFontSize,"mid\
dle").text(tick)
    })

    this.update(0)
  }
```

Und in update() zeichnen wir jetzt den Zeiger nicht mehr neu, sondern drehen ihn einfach um den gewünschten Winkel:

**src/components/circulargauge.ts**

```
// called if new value arrives
  update(value) {

    this.pointer
      .transition()
      .duration(700)
      .attr("transform",`rotate(${this.scale(value)})`)

    this.valueText.text(value)
  }
```

Wie Sie sehen wurde zwar das initiale Zeichnen des Zeigers ein wenig komplizierter, aber dafür wurde die update() Funktion wesentlich einfacher. Und das Problem mit der 'unnatürlichen' Bewegung des Zeigers, welches wir vorhin hatten, hat sich damit ebenfalls erledigt. Jetzt können wir eine relativ lange 'duration()' einstellen, die bewirkt, dass die Drehung des Zeigers recht natürlich wirkt.

Vermutlich haben Sie bemerkt, dass wir dem Zeiger noch eine CSS Klasse mitgegeben haben:

```
this.pointer.append("svg:path")
      .attr("d", pointer_stroke)
      .classed("pointer", true)
```

Solange wir diese Klasse nicht definiert haben, bleibt der Zeiger unschön schwarz. Das wollen wir ändern und schreiben dazu in styles.scss:

```
svg{
  .pointer{
    stroke: #ff0000;
    fill: #ff1100;
    opacity: 0.8;
  }
}
```

Mit dem 'opacity' Attribut bewirken wir, dass der Zeiger leicht durchscheinend wirkt. Opacity muss eine Zahl zwischen 0 (ganz durchsichtig) und 1 (ganz undurchsichtig) sein.

Natürlich dürfen Sie den Zeiger gerne noch beliebig schön gestalten, ich wollte Ihnen nur zeigen, wie man eine komplexere SVG Figur erstellen und drehen kann.

# Teil 3: DoubleGauge

Nun haben wir so teure Homematic Thermo/Hygrometer angeschafft und lesen nur die Temperatur ab. Natürlich könnten wir je zwei CircularGauges pro Instrument bereitstellen, und je eines für Temperatur und eines für Feuchtigkeitsanzeige parametrisieren. Das Anzeige-Widget ist ja flexibel genug programmiert. Aber das finde ich, ist Platzverschwendung. Vor allem, wenn ich es dann auch auf einem Handy-Bildschirm ablesen will.


Abb. 4.12: Double Gauge

Als letzten Teil unserer Exkursion durch die Welt der Rundinstrumente werde ich daher mit Ihnen, wenn Sie wollen, eine DoubleGauge bauen: Ein Rundinstrument mit zwei Anzeigen. Nach unserer Vorarbeit in den letzten Kapiteln ist das eher trivial. Wir müssen einfach jedes Element doppelt programmieren, und vielleicht ein, zwei Gedanken an die entgegengesetzte Drehrichtung des unteren Zeigers aufwenden, dann sollte es klappen. Den Ausgangspunkt dieses Kapitels erhalten Sie mit

```
git checkout -f origin/stufe_07
git clean -f
npm install
```

Als erstes definieren wir die Konvention, dass alles, was die obere Anzeige betrifft, mit der präfix ‘upper’ gekennzeichnet wird, alles für die untere Anzeige mit ‘lower’.

Von dieser Verdoppelung mal abgesehen, sieht der Anfang unserer DoubleGauge sehr ähnlich aus, wie der der CircularGauge:

**src/components/doublegauge.ts**

```
const MIN_ANGLE = 15
const MAX_ANGLE = 165

@autoinject
@noView
export class DoubleGauge {
  @bindable cfg;
  private arcsize;
  private upperScale
  private lowerScale
  private upperPointer
  private lowerPointer
  private upperValueText
  private lowerValueText
```

```
  private body

  constructor(private hlp: Helper, public element: Element,
    private ea: EventAggregator) { }


  configure() {
    this.cfg = Object.assign({}, {
      size: 150,
      upperMin: 0,
      upperMax: 100,
      lowerMin: 0,
      lowerMax: 100,
      message: ["doublegauge_upper_value",
                "doublegauge_lower_value"],
      upperBands: [{ from: 0, to: 100, color: "blue" }],
      lowerBands: [{ from: 0, to: 100, color: "green" }]
    }, this.cfg)
    this.upperScale = scaleLinear()
      .domain([this.cfg.upperMin, this.cfg.upperMax])
      .range([MIN_ANGLE, MAX_ANGLE])
    this.upperScale.clamp(true)

    this.lowerScale = scaleLinear()
      .domain([this.cfg.lowerMin, this.cfg.lowerMax])
      .range([MAX_ANGLE, MIN_ANGLE])
    this.lowerScale.clamp(true)

    this.arcsize = this.cfg.size / 7
  }

  attached() {
    this.configure()
    this.body = select(this.element).append("svg:svg")
      .attr("class", "circulargauge")
      .attr("width", this.cfg.size)
      .attr("height", this.cfg.size)
    this.render()

    this.ea.subscribe(this.cfg.message[0], data => {
      this.upperRedraw(data)
    })
    this.ea.subscribe(this.cfg.message[1], data => {
      this.lowerRedraw(data)
    })
  }
```

```
  // .... geht nachher noch weiter
```

MIN_ANGLE und MAX_ANGLE sind zu Konstanten geworden, da es angesichts des beengteren Platzes in unserer Gauge keinen Sinn mehr macht, den Kreisbogen konfigurierbar zu machen.

Beachten Sie, dass die .range in der unteren Skala nicht von MIN_ANGLE zu MAX_ANGLE geht, sondern umgekehrt. Grössere Werte werden hier also auf kleinere Winkel umgesetzt und umgekehrt.

Die attached() Funktion ist fast identisch zu CircularGauge, nur dass wir auf zwei verschiedene Nachrichten lauschen. Theoretisch hätte man natürlich auch eine einzelne Nachricht definieren können, die beide Werte für oben und unten enthält. (Der mit dem EventAggregator übergebene data-Wert kann ein beliebiger Datentyp sein, auch ein Array oder ein beliebig komplex aufgebautes Objekt). Ich habe mich hier aber entschieden, die beiden Teile von DoubleGauge als unabhängige Instrumente zu behandeln, weil sie dadurch flexibler zu handhaben sind.

Spätestens an dieser Stelle fällt nun aber auf, dass wir ziemlich viel geschrieben haben, was fast gleich aussieht, wie in der CircularGauge. Das ist schlecht. Ein wichtiges Prinzip des Programmierens lautet: DRY (Don't repeat yourself, wiederhole dich nicht). Das hat nicht nur mit Faulheit zu tun, sondern auch mit Folgendem: Wenn Sie irgendwann einen Fehler in einem solchen Stück Code finden, oder eine Verbesserung oder nur Veränderung einbauen, dann müssen Sie mühsam jedes Programmteil suchen, wo Sie diesen Code verwendet haben. Wenn Sie sich aber an DRY gehalten hatten, dann brauchen Sie nur eine einige Stelle zu ändern, und die Änderung erscheint automatisch überall.

Wir machen deshalb hier einen kleinen Exkurs, um den gemeinsamen Code zu definieren

---

## Exkurs: DRY

Wir lagern also den Teil jeder Komponente, der immer gleich ist (auch "boilerplate code" genannt, Textbaustein), in eine gemeinsame Komponente aus. Dazu gibt es verschiedene mögliche Ansätze. Wir können das, was gemeinsam ist, in eine gemeinsame Oberklasse unserer Komponentenklassen packen. Oder wir packen den Initialiserungscode in eine Helferfunktion. So oder so müssen wir uns überlegen, was es ist, das unsere Komponenten ausmacht, wo also die Gemeinsamkeit liegt. In Typescript kann man solche Dinge in einem Interface deklarieren:

**in src/components/helper.ts**

```
export type eaMessage={
  message: string,
  data: any
}
export interface Component {
  configure()           // Konfiguration der Komponente
  render()              // Zeichnen der Komponente
  update(data:eaMessage)// Einen neuen Wert anzeigen
```

```
  cfg: any              // Konfigurationsdaten
  element: Element      // DOM Element
  body: Selection       // SVG Bssiselement
  component_name: String
}
```

Ein Interface ist einfach eine Beschreibung, welche Eigenschaften ein Objekt mindestens hat, das dieses Interface implementiert. Jede Klasse, die "Component" implementiert, hat also mindestens die oben genannten Funktionen und Felder.

Dann lagern wir den Initialisierungscode in eine `initialize()` Funktion aus, die wir ebenso wie das Interface in unsere Helper-Klasse verschieben (Das ist eine mehr oder weniger willkürliche Entscheidung. Wir hätten auch eine eigene Klasse dafür erstellen können, oder die Helper-Klasse zu einer Oberklasse unserer Komponenten machen können).

**src/components/helper.ts**

```
export class Helper {
  static BORDER = 5

  constructor(public ea: EventAggregator) { }

  initialize(component: Component, defaultCfg: any) {
    component.cfg = Object.assign(
      {
        width: component.cfg.width || component.cfg.size || 150,
        height: component.cfg.height || component.cfg.size || 150,
        modify: a => a
      }, defaultCfg, component.cfg)
    component.configure()
    component.body = select(component.element).append("svg:svg")
      .attr("class", component.component_name)
      .attr("width", component.cfg.width || component.cfg.size || 180)
      .attr("height", component.cfg.height || component.cfg.size || 80)
    component.render();
    ([].concat(component.cfg.message)).forEach(msg => {
      this.ea.subscribe(msg, data => {
        component.update(<eaMessage>{
          message: msg,
          data: component.cfg.modify(data)
        })
      })
    });
  }

  // ... rectangle, arch, stringElem, line und deg2rad hier weggelassen ...
```

```
  defaultFrame(c: Component): { x: number, y: number, w: number, h: number } {
    const yoff = this.frame(c.body, c.cfg.width, c.cfg.height, c.cfg.caption, c.c\
fg.capsize)
    return {
      x: Helper.BORDER,
      y: yoff,
      w: c.cfg.width - 2 * Helper.BORDER,
      h: c.cfg.height - yoff - Helper.BORDER
    }
  }

  frame(parent, outer_width: number, outer_height: number, caption: string = unde\
fined, capsize: number = outer_height / 8): number {
    this.rectangle(parent, 0, 0, outer_width, outer_height, "frame")
    let x_offset = Helper.BORDER
    let y_offset = Helper.BORDER
    let width = outer_width - 2 * Helper.BORDER
    let height = outer_height - 2 * Helper.BORDER
    if (caption) {
      let fontsize = Math.round(capsize)
      y_offset = y_offset + fontsize
      height = height - fontsize - 2
      this.rectangle(parent, x_offset, y_offset, width, height, "inner")
      let off = (y_offset - fontsize) / 2
      this.stringElem(parent, outer_width / 2, Helper.BORDER + off, fontsize, "mi\
ddle").text(caption)
      return y_offset
    } else {
      this.rectangle(parent, x_offset, y_offset, width, height, "inner")
      return Helper.BORDER
    }
  }
}
```

Sie sehen, der `initialize()` - Teil ist recht ähnlich dem, was wir vorher im `configure()` Teil der Komponente gemacht haben: Zunächst wird ein Default-cfg mit dem per cfg.bind in app.html übergebenen Konfigurationsobjekt überlagert. Etwas seltsam erscheint vielleicht die Zeile `modify: a=>a` bei (1). Die Auflösung folgt bei (2): Um die Komponente möglichst universell zu gestalten, kann der anzuzeigende Wert anwendungsspezifisch modifiziert werden, bevor er an die Komponente übergeben wird. In diesem Fall kann man in der individuellen cfg eine entsprechende Funktion als 'modify' Attribut eintragen. Die Default-Implementation liefert einfach den unmodifizierten Wert (a⇒a). Funktionen wie 'a⇒1/a' oder 'a⇒a*a' könnten je nach Anwendung eine bessere Grafik ergeben. Einen konkreten Anwendungsfall zeige ich später.

Ausserdem haben wir Helper.ts eine Funktion "defaultFrame" spendiert, die einen Standard-Rahmen um eine Komponente zeichnet, und die Innenmasse des Rahmens, also den eigentlichen

Zeichenbereich der Komponente, zurückliefert. Auf diese Weise erzielen wir ein einheitliches Design unserer Komponenten, das wir bei Bedarf auch sehr einfach ändern können.

Also fangen wir noch einmal von Vorne an und erstellen das Grundgerüst unserer DoubleGauge besser.

Falls Sie keine Lust haben, das abzutippen, können Sie es auch auschecken:

```
git checkout -f origin/stufe_08
git clean -f
npm install
```

Das hier sind die Anderungen:

**components/doublegauge.ts redesigned**

```
import { bindable, noView, autoinject } from 'aurelia-framework'
import { scaleLinear } from 'd3-scale'
import 'd3-transition'
import { Helper, Component, eaMessage } from './helper'

const MIN_ANGLE = 15
const MAX_ANGLE = 165

@autoinject
@noView
export class DoubleGauge implements Component {
  @bindable cfg;
  component_name = "DoubleGauge";

  private arcsize;
  private upperScale
  private lowerScale
  private upperPointer
  private lowerPointer
  private upperValueText
  private lowerValueText
  body

  constructor(private hlp: Helper, public element: Element) { }

  configure() {
    this.upperScale = scaleLinear()
      .domain([this.cfg.upper.minValue, this.cfg.upper.maxValue])
      .range([MIN_ANGLE, MAX_ANGLE])
    this.upperScale.clamp(true)

    this.lowerScale = scaleLinear()
```

```
      .domain([this.cfg.lower.minValue, this.cfg.lower.maxValue])
      .range([MAX_ANGLE, MIN_ANGLE])
    this.lowerScale.clamp(true)

    this.arcsize = this.cfg.size / 7
  }

  attached() {
    this.hlp.initialize(this, {
      size: 150,
      upper: {
        minValue: 0,
        maxValue: 100,
        bands: [{ from: 0, to: 100, color: "blue" }],
      },
      lower: {
        minValue: 0,
        maxValue: 100,
        bands: [{ from: 0, to: 100, color: "green" }]
      },

      message: ["doublegauge_upper_value",
        "doublegauge_lower_value"],

    })
  }

  render() {
    const dim=this.hlp.defaultFrame(this)
    let size = { w: (dim.w / 2) * 0.9, h: (dim.h / 2) * 0.9 }
    let center= {x: dim.w/2 +dim.x, y: dim.h/2+dim.y}

    const pointer_width = 10
    const pointer_base = 0.3
    const pointer_stroke =
      `M ${-size.w * pointer_base} 0
      L 0 ${-pointer_width / 2}
      L ${size.w * (1 - pointer_base)} 0
      L 0 ${pointer_width / 2}
      Z`

    /*
      Draw the coloured bands for the scale
    */
    const drawBands = (bands, scale, angle) => {
      bands.forEach(band => {
```

```
      this.hlp.arch(this.body, center.x, center.y,
        size.w - this.arcsize, size.w,
        this.hlp.deg2rad(scale(band.from)),
        this.hlp.deg2rad(scale(band.to)), band.color, angle)

    })
  }
  drawBands(this.cfg.upper.bands, this.upperScale, 270)
  drawBands(this.cfg.lower.bands, this.lowerScale, 90)

  // Draw the pointers
  const pframe = this.body.append("g")
    .attr("transform",
      `translate(${center.x},${center.y}) rotate(180)`)

  this.upperPointer = pframe.append("svg:path")
    .attr("d", pointer_stroke)
    .classed("pointer", true)

  this.lowerPointer = pframe.append("svg:path")
    .attr("d", pointer_stroke)
    .classed("pointer", true)

  pframe.append("svg:circle")
    .attr("cx", 0)
    .attr("cy", 0)
    .attr("r", 10)


  /* field for actual measurement */
  const valuesFontSize = Math.round(size.h / 5)
  this.upperValueText = this.hlp.stringElem(this.body, center.x,
    center.y - size.h / 2, valuesFontSize, "middle")
  this.lowerValueText = this.hlp.stringElem(this.body, center.x,
    center.y + size.h / 2, valuesFontSize, "middle")

  /* create tickmarks  */
  const tickmarkFontSize = this.arcsize / 3
  const createTickmarks = (scale, f) => {
    scale.ticks().forEach(tick => {
      const valueToPoint = (val, factor, scale) => {
        let arc = scale(val)
        let rad = this.hlp.deg2rad(arc)
        let r = ((dim.w / 2) * 0.9 - this.arcsize) * factor
        let x = r * Math.cos(rad)
        let y = r * Math.sin(rad)
```

```
          return { x, y }
        }
        let p1 = valueToPoint(tick, 1.2, scale)
        let p2 = valueToPoint(tick, 1.0, scale)
        let p3 = valueToPoint(tick, 1.35, scale)
        this.hlp.line(this.body, center.x + p1.x * f, center.y + p1.y * f,
          center.x + p2.x * f, center.y + p2.y * f, "black", 1.2)
        this.hlp.stringElem(this.body, center.x + p3.x * f, center.y + p3.y * f,
          tickmarkFontSize, "middle").text(tick)
      })

    }
    createTickmarks(this.upperScale, -1)
    createTickmarks(this.lowerScale, 1)

    this.upperRedraw(0)
    this.lowerRedraw(0)
  }

  update(newVal: eaMessage) {
    if (newVal.message === this.cfg.message[0]) {
      this.upperRedraw(newVal.data)
    } else {
      this.lowerRedraw(newVal.data)
    }
  }

  upperRedraw(upper) {
    if (isNaN(upper)) {
      this.upperValueText.text("Fehler");
      this.upperPointer.attr("style", "opacity:0.1")
    } else {
      this.upperPointer
        .transition()
        .duration(600)
        .attr("transform", `rotate(${this.upperScale(upper)})`)
      this.upperValueText.text(upper + this.cfg.upper.suffix)
    }
  }
  lowerRedraw(lower) {
    if (isNaN(lower)) {
      this.lowerValueText = lower
      this.lowerPointer.attr("style", "opacity:0.1")
    } else {
      this.lowerPointer
        .transition()
```

```
        .duration(600)
        .attr("transform", `rotate(${180 + this.lowerScale(lower)})`)
      this.lowerValueText.text(lower + this.cfg.lower.suffix)
    }
  }
}
```

Wie Sie sehen, wurde die Initialisierung doch deutlich vereinfacht. Wir werden künftig alle Elemente in dieser Weise erstellen. Sie können dann Ihr eigenes “look&feel” ganz einfach durch ändern von style.css und der defaultFrame() Funktion in helper.ts erstellen.

Wegen der Änderungen in helper.ts sind auch einige kleine Anpassungen in der CircularGauge notwendig. Versuchen Sie es selber zu korrigieren, oder schauen Sie sich an, wie CircularGauge jetzt im Quellcode zu stufe_08 aussieht.

Das Umbauen bereits existierenden Codes, um neue Aspekte oder Verbesserungen einzubringen, nennt man auch ‘Refactoring’. Bei grösseren Projekten kann das ein recht fehlerträchtiger Prozess sein. Es empfiehlt sich darum generell, dass man, sobald man erkennt, dass etwas nicht optimal gelöst wurde, möglichst früh über ein Refactoring nachdenkt, und es dann (und nur dann), wenn die Vorteile die Nachteile überwiegen, zügig umsetzt, dann aber gründlich testet um keinen existierenden Programmcode zu bersehen, der angepasst werden müsste.

**Endes des Exkurses**

---

In der render() Funktion mussten wir wieder jeden Schritt verdoppeln. Um nicht alles zweimal schreiben zu müssen, habe ich `drawBands()` und `createTickmarks()` als interne Funktionen definiert, die dann jeweils zweimal mit passenden Parametern für die obere und die untere Anzeigehälfte aufgerufen werden.

Und last but not least gibt es nicht nur eine, sondern zwei `redraw()` Funktionen: `upperRedraw()` und `lowerRedraw()`.

Der Einbau dieser Komponente in app.html erfolgt erwartungsgemäß:

**src/app.html**

```
<template>
  <require from="components/circulargauge"></require>
  <require from="components/doublegauge"></require>
  <div class="container">
    <h1 class="h1">Klima-Demo</h1>
    <div class="row">
      <div class="col">
        <double-gauge cfg.bind="conf.aussentemp_cfg">
        </double-gauge>
      </div>
      <div class="col">
```

```
        <double-gauge cfg.bind="conf.wohnzimmertemp_cfg">
        </double-gauge>
      </div>
    </div>
  </div>
</template>
```

In app.ts habe ich nun aber, wo ich schon mit dem Refactoring beschäftigt war, ebenfalls eine Änderung eingebaut: Der Teil mit den Konfigurationsdefinitionen für die Anzeigen wird immer länger, und er droht mit zunehmender Komplexität des Programms, noch länger zu werden. Das macht app.ts unnötig unübersichtlich. Ich lagere darum den Teil mit den Konfigurationsdateien aus in eine neue Datei namens *config.ts*:

**src/config.ts**

```
import env from './environment'
const gauge_size=242;
const switch_size=80;

const climate={
  temperature: {
    caption: "Temperatur",
    suffix: "°C",
    minValue: 17,
    maxValue: 35,
    precision: 1
  },
  humidity:{
    caption: "Luftfeuchte",
    suffix: "%",
    minValue:20,
    maxValue:80,
    precision: 0
  },
  temp_scale: {
    bands: [{ from: 10, to: 17, color: "#8cf2e4" },
    { from: 15, to: 20, color: "yellow" },
    { from: 20, to: 26, color: "green" },
    { from: 26, to: 35, color: "red" }],
    minValue: 15,
    maxValue: 35,
    suffix: "°C",
    precision: 1
  },
  humid_scale:{
    bands: [{ from: 20, to: 40, color: "yellow" },
    { from: 40, to: 60, color: "green" },
```

```
    { from: 60, to: 80, color: "yellow" }],
    minValue: 20,
    maxValue: 80,
    suffix: "%",
    precision: 0
  }

}

export default {
  "SWITCH_ON": 1,
  "SWITCH_OFF": 0,
  "SWITCH_AUTO": 2,

  wohnzimmertemp_cfg: {
    "devices": [env.devices.wohnzimmer_temp, env.devices.wohnzimmer_hygro],
    "size": gauge_size,
    upper: climate.temp_scale,
    lower: climate.humid_scale,
    message: ["wohnzimmer_temp", "wohnzimmer_hygro"],
    caption: "Wohnzimmer",
    timeout: 86400,
    visible: true
  },
  aussentemp_cfg: {
    "devices": [env.devices.aussen_temp, env.devices.aussen_hygro],
    "size": gauge_size,
    upper: {
      bands: [{ from: -15, to: 0, color: "#8cf2e4" },
      { from: 0, to: 10, color: "yellow" },
      { from: 10, to: 25, color: "green" },
      { from: 25, to: 40, color: "red" }],
      minValue: -15,
      maxValue: 40,
      suffix: "°C",
      precision: 1,
    },
    lower:{
      bands: [{ from: 20, to: 40, color: "yellow" },
      { from: 40, to: 60, color: "green" },
      { from: 60, to: 80, color: "yellow" }],
      minValue: 20,
      maxValue: 80,
      suffix: "%",
      precision: 0
    },
```

```
    message: ["aussen_temp", "aussen_hygro"],
    caption: "Aussen",
    timeout: 86400,
    visible: true
  }
}
```

Wie Sie sehen, habe ich nebst dem Auslagern der Konfigurationsobjekt auch noch einige Vereinfachungen eingebaut, indem ich Teile, die wiederholt benötigt werden, in die vorgelagerte Konstante 'climate' verschob. Dies hat erneut den Vorteil, dass man, wenn man beispielsweise Temperaturskalen verändern will, nur an einer Stelle anpassen muss.

Config.ts wiederum lese ich als externes Modul in app.ts ein.

**src/app.ts**

```
import { FetchService } from './services/fetchservice'
import { autoinject } from 'aurelia-framework'
import { EventAggregator } from 'aurelia-event-aggregator'
import configs from './config'
import env from './environment'
const dev=env.devices


@autoinject
export class App {
  message = 'Hello World!'
  fetcher = new FetchService()
  conf=configs        // (1)

  constructor(private ea: EventAggregator) {
    let devices = [configs.wohnzimmertemp_cfg, configs.aussentemp_cfg]
    let ids = []
    let messages = []
    devices.forEach(dev => {
      ids = ids.concat(dev.devices)
      messages = messages.concat(dev.message)

    })
    setInterval(() => {
      this.fetcher.getIobrokerValues(ids).then(results => {
        for (let i = 0; i < results.length; i++) {
          this.ea.publish(messages[i], results[i])
        }
      }, reason => {
        alert("an error occured " + reason)
      })
    }, 3000)
```

```
  }
}
```

Vielleicht wundern Sie sich über die Zeile: *conf=configs* bei (1). Schließlich greife ich später in *app.ts* nicht mehr auf *conf* zu. Das Rätsel löst sich, wenn Sie etwas weiter oben app.html noch einmal anschauen.

```
...
<circular-gauge cfg.bind="conf.aussentemp_cfg">
</circular-gauge>
...
```

Dort benötigen wir die Variable *conf*, um auf die Konfigurationen zugreifen zu können. Die Binding-Engine hätte nicht direkt einen Ausdruck wie *configs.aussentemp_cfg* auswerten können. Die View kann nur Variablen lesen, die direkt in ihrem ViewModel definiert sind. Wenn man hier etwas falsch macht, sind die Fehler oft schwer zu finden, weil die IDE einem keine Hilfestellung geben kann. Der Editor "weiß" nichts von der Bindung zwischen View und ViewModel, die Aurelia erst zur Laufzeit herstellt. Lassen Sie die Zeile *conf=configs* in app.ts mal weg und starten Sie das Programm. Es wird keine Fehlermeldung geben, aber es funktioniert nicht.

Zusammenfassung:

Jetzt haben wir einen deutlich robusteren und lesbareren Grundstock für unsere späteren Komponenten. Ich verlasse daher für jetzt die Rundinstrumente, und überlasse Ihnen die weitere Verschönerung der Zeiger und Skalen, und vielleicht unterschiedliche Farben oder Formen für obere und untere Zeiger, als Übung.

# Teil 4: Druckknopf

Bisher haben wir nur Daten abgelesen. Jetzt wollen wir aktiv werden. Also Dinge ein- und ausschalten. Wir fangen mit einem einfachen Druckknopf an. Den Startpunkt dieses Teils erhalten Sie, wenn Sie Folgendes eingeben:

```
git checkout -f origin/stufe_09
git clean -f
npm install
```

Ein Druckknopf hat zwei visuell und funktional unterscheidbare Zustände: Gelöst und gedrückt. manchmal bleibt er nur solange gedrückt, wie er bedient wird, manchmal rastet er im gedrückten Zustand ein. Für unsere derzeitige Position als UI-Designer stellt sich die Frage, wie wir den aktuellen Zustand des Schalters darstellen wollen. Ich fange mit einem sehr einfachen Beispiel an, um das Konzept zu zeigen:

Zunächst bereiten wir in app.html ein Plätzchen für unsere neue Komponente:

```
<template>
  <require from="components/doublegauge"></require>
  <require from="components/pushbutton"></require>
  <div class="container">
    <h1 class="h1">Klima-Demo</h1>
    <div class="row">
      <div class="col">
        <double-gauge cfg.bind="conf.aussentemp_cfg">
        </double-gauge>
      </div>
      <div class="col">
        <double-gauge cfg.bind="conf.wohnzimmertemp_cfg">
        </double-gauge>
      </div>
      <div class="col">
        <push-button cfg.bind="conf.pushbutton_cfg" pressed="pb_on">
        </push-button>
      </div>
    </div>
  </div>
</template>
```

Und sorgen dafür, dass in config.ts die referenzierten Variablen bereit stehen

**src/config.ts**

```
// ...
pushbutton_cfg: {
    message: "treppenlicht",
    size: switch_size
  },
// ...
```

Dann erstellen wir die Komponente, diesmal als Einstieg zuerst ganz konventionell mit .ts und .html-Teilen:

**src/components/pushputton.ts**

```
import { bindable, autoinject } from 'aurelia-framework'
import { EventAggregator } from 'aurelia-event-aggregator'

@autoinject
export class PushButton{
  @bindable cfg
  @bindable pressed:boolean;

  constructor(private ea:EventAggregator){}

  attached(){
    this.cfg=Object.assign({},{
      message: "pushbutton"
    },this.cfg)
  }

  toggle(){
    this.pressed=!this.pressed
    this.ea.publish(this.cfg.message,this.pressed)
  }
}
```

**src/components/pushbutton.html**

```
<template>
  <p>Pushbutton</p>
  <div if.bind="pressed" click.trigger="toggle()">
    On!
  </div>
  <div if.bind="!pressed" click.trigger="toggle()">
    Off!
  </div>
</template>
```

Möglicherweise kannten Sie das von Aurelia bereitgestellte *if.bind* Konstrukt bisher noch nicht. Damit binden wir ein beliebiges HTML-Element in Abhängigkeit von einer Bedingung ein, hier abhängig vom Zustand der *pressed* - Variable im Viewmodel. Starten Sie das Programm und probieren Sie es aus! Wenn Sie auf On! klicken, wird es zu Off! und umgekehrt. Damit ist auch gleich klar geworden, was die click.trigger-Ausdrücke bedeuten: Sie rufen die angegebene Funktion auf, wenn auf das Element geklickt wird.

Nun sind wir natürlich nicht auf die Wörter On und Off limitiert. Wir können beliebig viel beliebiges HTML und SVG in diese DIVs packen, die wir per if.bind ein- und ausblenden.

Zum Beispiel ein Bild:

```
<template>
  <div if.bind="pressed" click.trigger="toggle()">
    <img src="img/on_button.png">
  </div>
  <div if.bind="!pressed" click.trigger="toggle()">
    <img src="img/off_button.png">
  </div>
</template>
```

Eine ähnliche Technik hatten wir ja schon im Kapitel über Schalter beim Scripting-host besprochen. Der Nachteil dieser Darstellungsweise ist: Da der Schalter ein Pixel-Bild ist, skaliert er nur bedingt auf andere Grössen. Bei so einer einfachen Figur ist das allerdings kein allzu grosses Problem. Wir können es ja mal testen (Die beiden Button Bilder befindne sich, wenn Sie die aktuellen Quellen ausgecheckt haben, im Ordner “static”. Sie können aber natürlich nach belieben zwei eigene Bilder für “an” und “aus” dort hin kopieren)

```
<template>
  <div if.bind="pressed" click.trigger="toggle()">
    <img src="/on_button.png" style="width: 100px">
  </div>
  <div if.bind="!pressed" click.trigger="toggle()">
    <img src="/off_button.png" style="width: 100px">
  </div>
</template>
```

Das sieht immer noch einigermassen akzeptabel aus. Aber schöner wäre natürlich eine Vektorgrafik. Einfaches Beispiel:

```
<template>
  <div if.bind="pressed" click.trigger="toggle()">
    <svg width="100px" height="100px">
      <rect class="frame" width="100px" height="100px"></rect>
      <rect class="inner" x="5px" y="5px" width="90px" height="90px"></rect>
      <rect x="10px" y="10px" width="80px" height="80px" fill="#e5fc1b" rx="10px"\
 ry="10px"></rect>
    </svg>
  </div>
  <div if.bind="!pressed" click.trigger="toggle()">
      <svg width="100px" height="100px">
          <rect class="frame" width="100px" height="100px"></rect>
          <rect class="inner" x="5px" y="5px" width="90px" height="90px"></rect>
          <rect x="10px" y="10px" width="80px" height="80px" fill="#41454c" rx="1\
0px" ry="10px"></rect>
        </svg>
  </div>
</template>
```

Doch im Sinn eines einheitlichen Designs unserer Oberfläche erstellen wir den Pushbutton jetzt wieder über unsere Standardmethode, und rein mit d3js ohne HTML Teil. Diesen Teil erhalten Sie mit

```
git checkout -f origin/stufe_10
git clean -f
npm install
```

Die Komponente sieht vertraut aus:

**src/components/pushbutton2.ts**

```
import { autoinject, noView, bindable } from "aurelia-framework";
import { Component, eaMessage, Helper } from "./helper";
import { detect } from 'detect-browser'

// Leider will Safari es anderes als die Anderen...   (1)
let LINK = "href"
const browser = detect.detect()
if (browser && (browser.name === 'safari' || browser.name === 'ios')) {
  LINK = "xlink:href"
}

@autoinject
@noView
export class PushButton2 implements Component {
  @bindable cfg
  body: any;
```

```
  component_name: "Push Button"
  private state = "off"
  private button;

  constructor(private hlp: Helper, public element: Element) { }
  attached() {
    this.hlp.initialize(this, {
      size: 110,
      message: "pushbutton2",
      caption: "Klick mich"
    })
  }

  configure() {}

  render() {
    const dim=this.hlp.defaultFrame(this)
    this.button=this.body.append("svg:image")
      .attr(LINK,"/off_button.png")
      .attr("x",dim.x)
      .attr("y",dim.y)
      .attr("width",dim.w)
      .attr("height",dim.h)
      .on("click",event=>{
        this.state=this.state==="on" ? "off" : "on"
        if(this.state === "on"){
          this.button.attr(LINK,"/on_button.png")
        }else{
          this.button.attr(LINK,"/off_button.png")
        }
      })

  }


  update(val: eaMessage) {
      // TODO
  }

}
```

Die Stelle der LINK-Definition bei (1) ist einem Phänomen geschuldet, dass trotz aller Standardisierungsbemühungen immer noch existiert: Browser-Inkompatibilität. Während sonst meistens IE die Rolle des Schlusslichts bei der Implementation von Neuerungen übernimmt, sind es hier Safari und sein mobiler iOS-Kollege: Der Link zu einem eingebetteten Bild in einem SVG Element musste in SVG Version 1 mit dem Attribut “xlink:href” deklariert werden. Diese Form wurde

in SVG 2 durch “hlink” ersetzt, und die alte Variante wurde als “deprecated” erklärt (https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/xlink:href), was bedeutet, dass sie in künftigen Browser-Versionen wohl nicht mehr unterstützt wird. Leider scheint Safari sich hier nicht um Standards zu kümmern: “href” wird schlicht nicht beachtet, und die Bilder werden nur mit “xref:href” korrekt angezeigt (probieren Sie es aus!). Das bedeutet, man muss sich entscheiden, ob man zu Safari inkompatibel sein will, oder zu künftigen Versionen der anderen Browser. Oder man baut, so wie hier, eine Entscheidung in den Code ein. (Um das zu tun müssen Sie `npm install --save detect-browser` ausführen, damit dieses Tool eingebunden werden kann - wenn Sie teil_11 ausgecheckt haben, habe ich das bereits für Sie getan, und Sie müssen nur noch `npm install` ausführen).

Generell empfehle ich: Wenn Code sich “seltsam” verhält, testen Sie ihn zunächst in Chrome, der sich meist recht eng an Standards und deren Neuerungen orientiert. Wenn etwas in Chrome erwartungsgemäß funktioniert, in einem anderen Browser aber nicht, dann googeln Sie - meist wird schon jemand dasselbe Problem gehabt haben. Hier lieferte zum Beispiel die Suchanfrage “safari svg href” ausreichend Antworten. Im Zweifelsfall verwenden Sie dann einen Adapter, wie hier gezeigt. Der Fairness halber sei gesagt, dass das nur noch sehr selten nötig ist. Erstens sind Browser generell viel standardkonformer geworden, und zweitens bügeln Framworks wie Aurelia die meisten noch verbliebenen Unterschiede hinter den Kulissen sauber aus.

Ansonsten ist der Code nun frei von überraschenden Stellen. Es wird ein “image”-Element namens “button” eingehängt, und je nachdem, ob das Feld *state* gerade “off” oder “on” ist, wird das passende Bild angebunden. Bei Klick wird *state* und das Bild zum jeweils anderen Zustand gewechselt. Binden Sie die Komponente ein, indem Sie in app.html den Teil mit dem Push Button so ändern:

```
<!-- ... -->
<require from="components/pushbutton2"></require>
<!-- ... -->
<div class="col">
  <push-button2 cfg="conf.pushbutton_cfg" pressed="pb_on"></push-button2>
</div>
<!-- -->
```

Und probieren Sie das Programm aus. Sie können den Schalter ein- und ausschalten, aber natürlich passiert noch nichts, wenn Sie das tun, ausser dass das Bild im Schalter sich ändert.

Wie könnten wir nun zum Beispiel die Aussenbeleuchtung einschalten? Bisher haben wir nur betrachtet, wie wir über das SimpleAPI aus ioBroker States auslesen können. Die Vermutung liegt nahe, dass wir über dasselbe REST API auch States setzen können. Die allgemeine Syntax ist: `http://homeview.local:8087/set/<state>?value=x`, wobei x für den gewünschten Wert steht.

Um diese Funktion an zentraler Stelle als Service bereitzustellen, schreiben wir eine neue Funktion im services/fetchservice.ts:

**src/services/fetchservice.ts: setIoBrokerValue**

```
public async setIoBrokerValue(id, value) {
    if (env.mock) {
      return {
        "id": id,
        "value": value
      }
    } else {
      let raw = await this.http.fetch(`${env.iobroker}/set/${id}?value=${value}`)
      let result = await raw.json();
      return result
    }
}
```

(Lassen Sie sich nicht davon irritieren, dass ein *fetch* Befehl abgesetzt wird, um einen Wert zu setzen. Das REST API merkt anhand der URL, dass ein Wert gesetzt werden soll, und der Begriff "fetch" bezieht sich hier auf die Antwort des REST Services. Diese Antwort liefern wir auch zurück.)

Wir könnten jetzt beispielsweise in der `toggle()` Funktion von pushbutton.ts oder im `on('click')` Handler von pushbutton2.ts diesen Service aufrufen, und es würde funktionieren. Allerdings würden wir damit unseren Vorsatz verletzen, universelle, lose gekoppelte Komponenten zu programmieren. Dieser Button wäre dann eng ans Treppenlicht gekoppelt, und wenn wir nach ein paar Jahren nicht mehr genau wissen, wie er im Einzelnen funktioniert, dann wird es uns schwer fallen, ihn für ein anderes Projekt anzupassen und zu verwenden.

Daher verwenden wir lieber unsere vorhin schon geübte Technik mit dem EventAggregator, nur eben umgekehrt: Diesmal initiiert die Komponente eine Nachricht, die von Interessenten aufgefangen und ausgewertet werden kann. Wir teilen der PushButton Komponente beim Instanzieren mittels des cfg-Parameters mit, welche Nachricht sie versenden soll, und können dann entspannt darauf warten, dass sie uns diese Nachricht zukommen lässt. Die Komponente braucht nicht zu wissen, was der Empfänger mit der Nachricht zu tun gedenkt.

**src/components/pushbutton2.ts**

```
// ...
 render() {
    const dim = this.hlp.defaultFrame(this)
    this.button = this.body.append("svg:image")
      .attr(LINK, "img/off_button.png")
      .attr("x", dim.x)
      .attr("y", dim.y)
      .attr("width", dim.w)
      .attr("height", dim.h)
      .on("click", event => {     //(1)
        this.hlp.ea.publish(this.cfg.message,
          { state: (this.state == "on") ? "off" : "on" ,
          direction: "out"})
```

```
      })

  }

  update(val: eaMessage) {
    const img = val.data.state === "on" ? "/on_button.png" : "/off_button.png"
    this.state = val.data.state
    this.button.attr(LINK, img)
  }

...
```

Da wir ohne View auskommen müssen, können wir nicht, wie beim ersten PushButton, einen `click.trigger()` im HTML definieren, sondern wir hängen bei (1) einen click-Handler (`.on("click",function(){})`) an die Definition des Buttons.

Wieso sendet `render()` eine Nachricht, anstatt den Button einfach direkt im click-Handler korrekt zu setzen? Nun, stellen Sie sich vor, mehrere Leute im Haus haben diese WebApp geöffnet. Jemand drückt auf den Pushbutton, und das Licht geht an resp. aus. Aber was zeigen die anderen Instanzen der WebApp an? Genau: Der PushButton muss nicht nur darauf reagieren, dass er von jemandem gedrückt wird, sondern er muss auch auf die Nachrichten reagieren, die eintreffen, wenn jemand anders woanders darauf gedrückt hat. Eine solche Nachricht wird ja vom Initialisierungscode in Helper.ts zu update() gesendet. Würden wir nun schon im Click-Handler den Zustand des Knopfs ändern, und dann die Nachricht versenden, dann würden wir unsere eigene Nachricht kurz darauf empfangen und den Knopf noch einmal anpassen. Also versenden wir lieber im click.Handler nur die Nachricht und ändern den Knopf erst dann, wenn eine (in unserem Fall die eigene) Nachricht bei `update()` eintrifft.

Natürlich müssen wir die Nachricht des PushButtons auch irgendwo anders auffangen, damit sie überhaupt irgendeine Wirkung ausser der Änderung der Anzeige hat. Das können wir in jedem Objekt tun, das Zugriff auf den EventAggregator hat. Wir tun es für jetzt gleich in app.ts.

**src/app.ts**

```
import { FetchService } from './services/fetchservice'
import { autoinject } from 'aurelia-framework'
import { EventAggregator } from 'aurelia-event-aggregator'
import configs from './config'
import env from './environment'
const dev=env.devices


@autoinject
export class App {
  message = 'Hello World!'
  fetcher = new FetchService()
  pb_on=false
  conf=configs
```

```
  constructor(private ea: EventAggregator) {
    let devices = [configs.wohnzimmertemp_cfg, configs.aussentemp_cfg]
    let ids = []
    let messages = []
    devices.forEach(dev => {
      ids = ids.concat(dev.devices)
      messages = messages.concat(dev.message)

    })
    this.switches()
    setInterval(() => {
      this.fetcher.getIobrokerValues(ids).then(results => {
        for (let i = 0; i < results.length; i++) {
          this.ea.publish(messages[i], results[i])
        }
      }, reason => {
        alert("an error occured " + reason)
      })
    }, 3000)
  }

  switches(){
    this.ea.subscribe(configs.pushbutton_cfg.message,pushed => {
      this.fetcher.setIoBrokerValue(dev.treppenlicht_direkt,pushed)
    })
  }
}
```

Wie Sie sehen, habe ich eine neue Funktion *switches()* eingeführt, die die Nachricht des Pushbuttons abfängt und auswertet. Natürlich hätte man den *subscribe*-Ausdruck auch direkt in den `constructor()` setzen können, aber dann würde dieser langsam zu gross und unübersichtlich. Eine Faustregel besagt, dass eine einzelne Funktion nicht grösser als eine Bildschirmseite sein sollte.

App.html bleibt unverändert:

**src/app.html**

```
<template>
  <require from="components/doublegauge"></require>
  <require from="components/pushbutton2"></require>
  <div class="container">
    <h1 class="h1">Klima-Demo</h1>
    <div class="row">
      <div class="col">
        <h2>Aussen</h2>
        <double-gauge cfg.bind="conf.aussentemp_cfg"></double-gauge>
```

```
      </div>
      <div class="col">
        <h2>Wohnzimmer</h2>
        <double-gauge cfg.bind="conf.wohnzimmertemp_cfg"></double-gauge>
      </div>
      <div class="col">
        <h2>Push Button</h2>
        <push-button2 cfg="conf.pushbutton2_cfg" pressed="pb_on"></push-button2>
      </div>
    </div>
</div>
</template>
```

# Teil 5: Tri-State Button

Bevor ich das Reich der Knöpfe verlasse, möchte ich einen weiteren Button-Typ vorstellen, den wir für unsere HomeView-Einrichtung benötigen: Einen Knopf, der die Stellungen Ein, Aus und Automatik einnehmen kann. Wir hatten einen solchen Schalter ja schon mit der Vis-Oberfläche erstellt.

Den Startpunkt dieses Teils erhalten Sie, wenn Sie folgendes eingeben:

```
git checkout -f origin/stufe_11
git clean -f
npm install
```

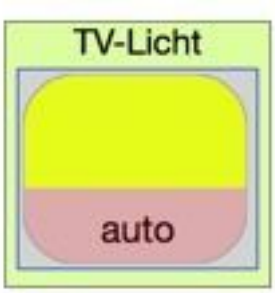



**Abb. 4.13: Tri State Button**

Das Grundgerüst der Klasse TristateButton ist sehr ähnlich wie das der anderen @noView Komponenten; ich gehe darum hier nicht mehr weiter darauf ein. Ich habe in app.html ein neues Feld für den TriState Button eingefügt und in *config.ts* eine neue cfg. Die TristateButton Klasse beginnt wie gehabt mit `configure()` und `attached()`; interessant wird erst die `render()`-Methode. Hier zeige ich Ihnen zunächst etwas, was wir bisher noch nicht benutzt haben: SVG clipping:

**src/components/tristatebutton.ts#render()**

```
render() {
    const dim = this.hlp.defaultFrame(this)
    const ratio = 0.6

    this.upper = this.hlp.rectangle(this.body, dim.x,
      dim.y, dim.w, dim.h, "light_on")
      .attr("rx", 15)
      .attr("ry", 15)
      .attr("clip-path", "url(#clip-bottom)")       // (1)
      .on("click", () => {
        this.ea.publish(this.cfg.message, {
          state: this.state === "on" ? "off" : "on",
          mode: "manual",
          direction: "out"
        })
      })

    this.lower = this.hlp.rectangle(this.body, dim.x,
      dim.y, dim.w, dim.h, "mode_auto")
      .attr("rx", 15)
```

```
      .attr("ry", 15)
      .attr("clip-path", "url(#clip-top)")          // (2)
      .on("click", () => {
        this.ea.publish(this.cfg.message, {
          state: this.state,
          mode: this.mode == "auto" ? "manual" : "auto",
          direction: "out"
        })
      })

    const defs = this.body.append("svg:defs")
    defs.append("svg:clipPath")
      .attr("id", "clip-bottom")
      .append("svg:rect")
      .attr("x", dim.x)
      .attr("y", dim.y)
      .attr("width", dim.w)
      .attr("height", dim.h * ratio)

    defs.append("svg:clipPath")
      .attr("id", "clip-top")
      .append("svg:rect")
      .attr("x", dim.x)
      .attr("y", dim.y + dim.h * ratio)
      .attr("width", dim.w)
      .attr("height", dim.h * (1 - ratio))

    let fontsize = Math.floor(dim.h / 5)
    let text_cx = this.cfg.size / 2
    let text_y = Math.round(dim.y + dim.h * ratio + (dim.h * (1 - ratio)) / 2) - 1
    let txtgroup = this.body.append("svg:g")
      .attr("transform", `translate(${text_cx},${text_y})`)
    this.autoText = this.hlp.stringElem(txtgroup, 0, 0, fontsize, "middle")
      .text("auto")

    this.powerText = this.hlp.stringElem(this.body, text_cx, dim.h / 2, fontsize,\
 "middle", 0)
      .style("pointer-events", "none");

  }
```

Zunächst erstellen wir zwei Rechtecke mit abgerundeten Ecken. Den Füllstil der Rechtecke, “light_on” etc. haben wir in der styles-Datei hinzugefügt:

**src/style.scss**

```
svg {
  .pointer {
    stroke: #ff0000;
    fill: #ff1100;
    opacity: 0.8;
  }
  .light_on{
    fill: #e5fc1b;
    stroke: #a79ea3;
    stroke-width: 0.8;
  }
  .light_off{
    fill: #41454c;

  }
  .mode_auto{
    fill: #f714205d;
    stroke: #a79ea3;
    stroke-width: 0.8;

  }
  .mode_manual{
    fill: #d3d3d3;
    stroke: #a79ea3;
    stroke-width: 0.8;
    opacity: 0.2;

  }
}
```

Das einzig Neue ist das Attribut “clip-path” bei (1) und (2). Mit diesem Attribut weisen wir SVG an, nur einen Teil des Objekts zu zeichnen. Nämlich den Teil, der sich mit der in clip-path definierten (und beliebig komplexen) Struktur überschneidet. Der eine Clip-Path beschneidet das obere, der andere das unter Rechteck.

Diese clip-paths wiederum definieren wir im Element “defs”, welches wir unserem body ebenfalls anhängen, hier unterhalb der Rechtecke, aber der Ort ist für SVG egal. Das Resultat ist jedenfalls, dass wir von jedem Rechteck nur einen Teil sehen.

Schließlich sorgen wir dafür, dass die beiden Teile auf Mausklicks reagieren.

Weiter unten definieren wir noch zwei Textfelder, deren Position und Schriftgrösse wir anhand der Bemaßung ausrechnen, um im oberen und unteren Teil Informationen auszugeben.

Die update() Methode sollte keine großen Probleme machen: Wir geben einfach jedem der Rechtecke die passende Klasse und blenden den Text “auto” ein, wenn der Modus auf auto ist.

Ausserdem wird optional im oberen Teil im eingeschalteten Zustand diese Verbrauchsangabe eingeblendet.

Dieses Textfeld hat keinen Inhalt, wird also standardmäßig nicht angezeigt. Wir zeigen es in der Funktion `update()` nur dann an, wenn die Variable *power* gesetzt ist, und runden es (bei (1)) je nach Größenordnung der Zahl auf keine, eine oder zwei Stellen:

**src/components/tristatebutton.ts#update()**

```
update(val: eaMessage) {
    let newstate = val.data
    //  console.log(val.message+", "+JSON.stringify(val.data))
    let bActState = (newstate.state === "on")
    let bActMode = (newstate.mode === "auto")
    this.upper
      .classed("light_on", bActState)
      .classed("light_off", !bActState)
    this.lower
      .classed("mode_manual", !bActMode)
      .classed("mode_auto", bActMode)
    this.autoText.attr("opacity", bActMode ? 1.0 : 0.2)

    if (newstate.power) {             // (1)
      let tx = newstate.power
      if (newstate.power > 1000) {
        tx = Math.round(newstate.power)
      } else if (newstate.power > 100) {
        tx = Math.round(10 * newstate.power) / 10
      } else {
        tx = Math.round(100 * newstate.power) / 100
      }
      this.powerText.text(tx + " W")
    }
    this.state = newstate.state
    this.mode = newstate.mode
  }
```

Spielen Sie ein wenig mit dem Button. Wenn Sie auf die untere Fläche klicken, wechseln Sie zwischen auto und manual, wenn Sie auf die obere Fläche klicken, zwischen an und aus, wobei ein Klick auf die obere Fläche ein manueller Eingriff ist, und darum auch in jedem Fall in den Modus “manual” führt.

Allerdings tut unser Button noch nichts. Wir müssen ihn noch verdrahten. Da er ausschliesslich über EventAggregator-Nachrichten gesteuert wird, erfolgt das “verdrahten” drahtlos. Im Grund kann die Steuerung an beliebiger Stelle im Programm liegen. Der Einfachheit halber packen wir sie zu allem Anderen nach *app.ts*.

Dabei müssen wir berücksichtigen, dass es jetzt nicht mehr um reine Einweg-Kommunikation geht. Der bisherige Aufbau von app.ts genügt nicht, um sowohl Anzeigen (“Licht ist an”), als

auch Aktionen (“Schalte Licht an”) zu ermöglichen.

App.ts wurde mit diesen zusätzlichen Abfragen ein wenig unübersichtlich. Ich habe daher die Gelegenheit ergriffen, die Klasse gleich aufzuräumen und ein wenig zu reorganisieren. Also ein Weiteres Refactoring. Sie erhalten diesen Stand des Projekts, wenn Sie eingeben:

```
git checkout -f origin/stufe_12
git clean-f
npm update
```

**src/app.ts**

```
import { FetchService } from './services/fetchservice'
import { autoinject } from 'aurelia-framework'
import { EventAggregator } from 'aurelia-event-aggregator'
import env from './environment'
const dev=env.devices
import configs from './config'
import { tristateMessage } from './components/tristatebutton';

@autoinject
export class App {
  fetcher = new FetchService()
  conf = configs
  constructor(private ea: EventAggregator) {
    let gauges = [configs.wohnzimmertemp_cfg, configs.aussentemp_cfg]
    let switches = [configs.fernsehlicht_cfg]
    this.subscribe(switches)
    setInterval(() => {
      this.getGaugeValues(gauges)
      this.getSwitchValues(switches)

    }, 3000)
  }

  /**
   * Subscribe to messages of all configured TriStateButtons
   * and send changed values to ioBroker
   * @param switches Array with TristateButton configurationa
   */
  subscribe(switches) {
    switches.forEach(sw => {
      this.ea.subscribe(sw.message, msg => {
        let value = 2
        if (msg.mode == "manual") {
          value = msg.state == "on" ? 0 : 1
        }
```

```
        this.fetcher.setIoBrokerValue(sw.mode_id, value)
      })
    });
  }

  /**
   * Fetch ioBroker values for all configured Gauges
   * @param gauges Array with gauge configurations
   */
  getGaugeValues(gauges) {
    let gauge_ids = []
    let gauge_messages = []
    gauges.forEach(dev => {
      gauge_ids = gauge_ids.concat(dev.devices)
      gauge_messages = gauge_messages.concat(dev.message)

    })
    this.fetcher.getIobrokerValues(gauge_ids).then(results => {
      for (let i = 0; i < results.length; i++) {
        this.ea.publish(gauge_messages[i], results[i])
      }
    }, reason => {
      alert("an error occured " + reason)
    })
  }

  /**
   * Fetch ioBroker values for all configured TristateButtons
   * @param switches Array with tristatebutton configurations
   */
  getSwitchValues(switches) {
    this.fetcher.getIobrokerValues(switches.map(sw => sw.mode_id)).then(async res\
ult => {
      for (let i = 0; i < result.length; i++) {
        let msg: tristateMessage;
        if (result[i] == 2) {
          let state = await this.fetcher.getIobrokerValue(switches[i].state_id)
          msg = {
            state: state ? "on" : "off",
            mode: "auto"
          }
        } else {
          msg = {
            state: result[i] == 1 ? "off" : "on",
            mode: "manual"
          }
```

```
        }
        this.ea.publish(switches[i].message, msg)
      }
    })
  }
}
```

Eine Faustregel lautet, dass eine einzelne Funktion normalerweise nicht grösser als eine Bildschirmseite sein sollte, damit das Programm gut lesbar bleibt. Ich habe darum die Abfragen der Messinstrumente und der Schalter in jeweils eigene Funktionen ausgelagert und rufe in `setInterval()` diese beiden Funktionen auf. `getGaugeValues()` entspricht im Wesentlichen dem, was vorher hier stand, während `getSwitchValues()` neu ist:

Mit `switches.map(sw => sw.mode_id)` erstellen wir 'on the fly' ein Array, welches nur die mode_id Attribute der TristateButton-Konfigurationen enthält, also die im ioBroker Javascript programmatisch erstellten States für die entsprechenden Schaltelemente. Dieses Array senden wir an den fetcher, der die entsprechenden Werte vom Homeview-Server abholt und wieder als Array zurück liefert. Das sind Werte zwischen 0 und 2, die für ein, aus und auto stehen. Der TriStateButton erwartet aber eine Message des Typs tristateMessage, also ein Objekt mit den Attributen *state* und *mode*. Damit wir dieses Objekt herstellen können, brauchen wir im Fall von "2" (auto) die zusätzliche Information, ob das Licht derzeit gerade an oder aus ist. Wir deklarieren deshalb die Resultatfunktion für `.then` als *async*, damit wir mit *await* eine zweite Abfrage zum ioBroker schicken können, um den Schaltzustand abzufragen. Dieser hat den Wert *true* oder *false*, den wir dann in die vom TristateButton erwartete Form "on" bzw. "off" wandeln. Die so zusammengebastelte Nachricht schicken wir dann, nein, nicht zum TristateButton, sondern wir publizieren sie über den EventAggregator, so dass jeder Interessent sie abfangen kann.

Um den TriStateButton zu verwenden, habe ich in config.ts eine neue Konfiguration für ein Licht eingefügt, die mit dem Fernsehgerät kombiniert ist (Vgl. auch in Kapitel 3 )

**src/config.ts**

```
// ...
fernsehlicht_cfg: {
    "state_id": env.devices.fernsehlicht_direkt,
    "mode_id": env.devices.fernsehlicht_modus,
    "size": switch_size,
    "message": "fernsehlicht",
    "caption": "TV-Licht"
  }
```

# Teil 6: Lineare Anzeigegeräte

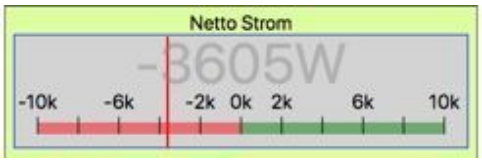



**Abb. 4.14: Lineare Anzeige**

Da wir grössere Stromverbraucher wie Waschmaschine, Tumbler und Geschirrspüler gern dann einschalten, wenn gerade genug Sonnenstrom zur Verfügung steht, wollen wir nun eine Anzeige zum schnellen Überblick über die momentane Stromsituation erstellen. Eine lineare Skala, welche positiven oder negativen Stromfluss anzeigt. Damit wir die neue Anzeige nicht nur hierfür verwenden können, machen wir sie erneut konfigurierbar.

```
git checkout -f origin/stufe_13
git clean -f
npm installl
```

Wir beginnen mit dem dem nun schon gut bekannten Grundgerüst:

**components/lineargauge.ts**

```
import {autoinject, bindable, noView} from 'aurelia-framework';
import {EventAggregator} from "aurelia-event-aggregator"
import {scaleLinear} from "d3-scale";
import 'd3-transition'
import {Helper, Component, eaMessage} from './helper'

@autoinject
@noView
export class Lineargauge implements Component{
  @bindable cfg
  readonly component_name = "Lineargauge"
  private scale
  body
  private indicator
  private value

  constructor(public element: Element,
              private hlp: Helper) {}

  attached() {
    this.hlp.initialize(this,{
      message: "Lineargauge_value",
      caption: "",
      suffix: "",
      minValue    : 0,
      maxValue    : 100,
      height : 50,
```

```
      width  : 180,
      padding: 0,
      bands  : [{from: 0, to: 100, color: "blue"}]
    })
  }

  configure() {
    this.scale = scaleLinear()
      .domain([this.cfg.minValue, this.cfg.maxValue])
      .range([5 + this.cfg.padding, this.cfg.width - 5 - this.cfg.padding])
      .clamp(true)
  }

  // ...
  // render()
  // update()
  //.. .
}
```

Die `render()` Methode macht etwas ganz Ähnliches, wie die der CircularGauge: Sie malt farbige Streifen und Tickmarks. Nur diesmal eben in Form von Geraden anstatt Kreisbogen. Der Zeiger ist diesmal ein simpler roter Strich. Für die Textanzeige des Messwerts müssen wir ein wenig rechnen, um die Grösse so hinzukriegen, dass alles gut lesbar bleibt. Dann setzen wir sie halbtransparent mitten in die Anzeige.

```
render() {
    // Rahmen
    const dim=this.hlp.defaultFrame(this)
    const baseline = dim.y+4*dim.h/5
    const scalesize=dim.h/10
    // draw colored bands
    this.cfg.bands.forEach(band => {
      this.hlp.line(this.body, this.scale(band.from), baseline, this.scale(band.t\
o), baseline, band.color, scalesize).attr("opacity", 0.5)
    })

    // draw tick marks and text on every second tick
    const ticks = this.scale.ticks(10)
    const tickFormat = this.scale.tickFormat(10, "s")
    const fontSize = dim.h / 6
    let even = true
    ticks.forEach(tick => {
      const pos = this.scale(tick)
      this.hlp.line(this.body, pos, baseline + scalesize/2, pos, baseline - scale\
size, "black", 0.6)
      if (even || (tick == 0)) {
```

```
        this.hlp.stringElem(this.body, pos, baseline-1.5 * fontSize, fontSize, "m\
iddle")
          .text(tickFormat(tick))
      }
      even = !even
    })

    // Zeiger
    this.indicator = this.hlp.line(this.body, this.scale(0), dim.y+1, this.scale(\
0), dim.y+dim.h-2, "red", 1.2)
      .attr("id", "indicator1")


    // Textanzeige des Messwerts
    let valueFontSize = 0.8 * 3*dim.h/5
    this.value = this.hlp.stringElem(this.body,
      this.scale(this.cfg.minValue + ((this.cfg.maxValue - this.cfg.minValue) / 2\
)),
      valueFontSize-3,
      valueFontSize,
      "middle",
      0
    ).attr("opacity", 0.4).style("fill", "grey")

  }
```

Die update() Funktion ist dann wieder sehr einfach: Wir schieben einfach den Zeiger an die neue Position, die uns von der d3.scaleLinear() aus dem übergebenen Messwert errechnet wird und beschriften das Textfeld neu.

```
  update(newVal:eaMessage) {
    const value=newVal.data
    const x = this.scale(value)
    this.indicator.transition()
      .duration(300)
      .attr("x1", x)
      .attr("x2", x)
    this.value.text(value + this.cfg.suffix)
  }
```

Damit die neue Komponente überhaupt angezeigt wird, muss sie in *app.html* eingetragen werden:

```
<require from="components/lineargauge"></require>
  ...
 <div>
   <lineargauge cfg.bind="conf.powermeter_cfg"></lineargauge>
</div>
```

und die dort angegebene Konfiguration muss in *config.ts* vorhanden sein. Und erst mit dieser Konfiguration wird aus einem universellen Linear-Anzeigeinstrument eine Stromfluss-Anzeige:

**src/config.ts**

```
export default {
  // ...
   powermeter_cfg: {
    devices: [env.devices.energy_grid_flow],
    width: gauge_size,
    height: switch_size,
    minValue: -10000,
    maxValue: 10000,
    caption: "Netto Strom",
    message: "fronius_net",
    suffix: "W",
    padding: 10,
    bands: [
      {from: -10000, to: 0, color: "red"},
      {from: 0, to: 10000, color: "green"}
    ],
    modify: x => -x
  }
  // ...
}
```

Hier sehen Sie auch eine Anwendung der modify-Funktion in cfg: In diesem Fall multiplizieren wir den erhaltenen Wert jeweils mit -1. Der Grund ist: Der ioBroker State für den GRID_FLOW, also den Netto-Fluss von oder zum Elektrizitätswerk, liefert negative Werte für Export und positive Werte für Import von Strom. Ich möchte es aber lieber umgekehrt haben und positive Werte sollen dann angezeigt werden, wenn wir mehr Strom produzieren, als verbrauchen. Daher also alles mal -1.

Zu guter Letzt muss auch app.ts angewiesen werden, die neue Komponente mit Daten zu versorgen. Das ist sehr einfach:

**src/app.ts**

```
export class app{
  // ...
  constructor(){
      // ...
      let gauges = [configs.wohnzimmertemp_cfg, configs.aussentemp_cfg,
                    configs.powermeter_cfg]
    // ...
  }
  // ...
}
```

Wir mussten nur unsere powermeter_cfg in die Liste der abzuklappernden gauges eintragen, den Rest erledigt die früher schon geschriebene Logik. Der anfangs erbrachte Aufwand zahlt sich langsam aus.

# Kapitel 6: Bastelstunde

Das Hauptgewicht dieses Buchs lag bisher ja auf der Software-Seite der Hausautomation. Nun gibt es leider nicht immer für alles eine Fertigkomponente. Oder der Preis für eine solche Fertigkomponente ist in einer für Privatleute inakzeptablen Höhe. Andererseits gibt es heute alles, was zum selber bauen (fast) beliebiger Komponenten nötig ist, zu durchaus akzeptablen Preisen und in riesiger Auswahl.

Natürlich wird ein selbstgebautes Gerät nie so kompakt und elegant sein können, wie ein industriell gefertigtes Produkt. Für viele Zwecke wird es aber dennoch gut genug sein.

# Grundätzliches und was man zum Basteln braucht

Das “Internet der Dinge” besteht aus unabhängigen Einzelkomponenten, die miteinander und mit übergeordneten Steuerungen vernetzt sind, und gemeinsam ein “intelligentes” System bilden. In diesem Kapitel werden zunächst die grundsätzlichen Bestandteile besprechen und dann auch selber kreativ tätig sein.

## Jargon

Natürlich hat sich im IoT-Bereich ein eigener Jargon entwickelt. Ich denke, es genügt zunächst, wenn Sie mit folgenden Begriffen etwas anfangen können

- IoT (Internet of Things, Internet der Dinge) - Ein Sammelbegriff für alle autonomen “Dinge” die sich via Internet vernetzen können.
- GPIO (general purpose input ouput) - Universalanschlüsse, die entweder etwas ausgeben oder etwas einlesen können. Was welcher Anschluss tut, wird softwareseitig bestimmt.
- ADC (analog digital converter) - Ein Pin, der eine Spannung am Eingang in einen proportionalen Zahlenwert umwandeln kann.
- PWM (pulse width modulation) - Ein Pin, der “pseudo-analoge” Ausgangssignale erzeugen kann, indem er schnelle Ein/Ausschaltvorgänge produziert. Je nach auszugebendem Wert ist das Verhältnis von Ein- zu Ausschaltzeit anders. So kann beispielsweise ein Licht gedimmt werden, indem man es nur in 50% statt in 100% der Zeit einschaltet, diese Schaltvorgänge aber so schnell nacheinander ausübt, dass das Auge davon getäuscht wird. In gleicher Weise kann man auch die Leistung von Motoren variieren.
- Output Current (Ausgangsstrom) - Die mögliche Ausgangsleistung. Wenn man mehr Leistung verlangt, als diese Zahl angibt, dann wird es nicht einfach “nicht gehen”, sondern es wird möglicherweise etwas durchbrennen. Und zwar etwas im Inneren des Chips, was man nicht reparieren kann. Daher: Wenn Sie nicht genau wissen, was Sie tun (und wie man es berechnet), betrachten sie alle Ausgangspins als praktisch “leistungslos” und hängen sie niemals etwas Stärkeres als etwa eine LED an. Alles, was mehr Leistung braucht, braucht einen Verstärker.
- Input Voltage (Eingangsspannung) - Manche der gebräuchlichen Chips wollen 5 Volt, manche 3.3 Volt. Wenn Sie 3.3 Volt statt 5 Volt liefern, wird es oft klappen. Wenn Sie aber 5 Volt statt 3.3 Volt anschliessen, wird meist etwas kaputt gehen. Also schließen Sie, wenn es sein muss, 5 Volt Peripherie an 3.3 Volt Ausgänge an, aber niemals umgekehrt. Oder nur über einen Spannungswandler.
- serial Port/serielle Schnittstelle - ein altehrwürdiger asynchroner Kommunikationsstandard, der ursprünglich als RS 232 mit 25 Leitungen definiert wurde, später als 9-poliger Anschluss am PC Einzug hielt, und bei Mikrokontrollern auf 3 Leitungen (Rx,Tx, Gnd) reduziert wurde. Da bei einer asynchronen Kommunikation kein gemeinsamer Takt vorgegeben wird, müssen sich die beteiligen Partner im Voraus auf bestimmte Eckwerte einigen (Baudrate, Zahl der Start- Daten- und Stopbits, Parität), sonst kommt es zu keiner Verständigung. Diese Daten sind im Standard nicht fest integriert, sondern müssen für jede Verbindung individuell festgelegt werden. Oft ist man mit 9600,8,n,1 auf der sicheren Seite, aber im Zweifel muss man das Datenblatt studieren. Es geht aber nichts kaputt, wenn man

falsche Parameter einsetzt, sondern es kommt lediglich kein Kontakt zustande. Man darf also ruhig ausprobieren.

- I2C (I square C: Inter-integrated circuit) - Ein synchroner Kommunikationsstandard, der mit zwei Drähten auskommt (Daher wird es manchmal auch TWI - Two Wire Interface genannt). An diesen zwei Drähten können bis zu 1024 Peripheriegeräte hängen. Es wird häufig verwendet, um Peripheriebausteine mit Steuergeräten zu verbinden. (Synchron bedeutet: Der Master gibt an einem Draht (SCL) den Takt vor, der für alle Geräte gilt). Es gibt daher bei diesen Bus viel weniger Raum für Missverständnisse, als bei RS232&Co.
- SPI (Serial Peripheral Interface) - Ein weiterer synchroner Kommunikationsstandard, der allerdings drei gemeinsame Drähte und einen weiteren Draht je Peripheriegerät benötigt.
- 1 Wire - Ein asynchroner Kommunikationsstandard, der sogar mit nur einem Draht (und Masse) auskommt. Sogar die Stromversorgung erfolgt dabei über den Signaldraht. Das Peripheriegerät braucht daher keine eigene Energiequelle, muss aber für die Dauer, in denen das Steuergerät das Signal auf "Low" zieht, ausreichend Energie speichern können (z.B. mit einem Kondensator).
- Breakout-Board: Damit ist eine Platine gemeint, die einen für Hobbyisten kaum handhabbaren winzigen SMB-Chip mit Anschlüssen versieht, die der durchschnittlich begabte Bastler mit seinen Projekten verlöten kann, oder die man mit entsprechenden Stiftleisten in eine Platine oder ein Breadboard stecken kann. Ausserdem enthält ein Breakout oft auch vom Chip benötigte Peripherie wie Widerstände und Kondensatoren, manchmal auch Schutzschaltungen gegen Überspannung etc.
- Firmware: Eine Software-Schicht, die zwischen Hardware und Anwendungsprogrammen vermittelt, und oft fest in einen nichtflüchtigen Speicher "gebrannt" wird. Bei PCs wird die Firmware meist BIOS genannt, und als Anwender hat man selten damit zu tun. Bei Mikrokontrollern dagegen muss man je nach gewünschter Entwicklungs- und Ablaufumgebung unterschiedliche Firmwares verwenden. Den Prozess, Firmware zum Controller zu übertragen, nennt man meist "flashen" oder "brennen", und er erfolgt fast immer über eine serielle Schnittstelle, die bei "besseren" Boards in einen USB Anschluss konvertiert wird.
- SOC (System on a Chip) - Ein nicht ganz scharfer Begriff, der Geräte meint, die im Prinzip aus einem einzigen Chip bestehen, welcher dann eben Mikroprozessor, Speicher und Steuerlogik beinhaltet. in der Realität brauchen solche SOCs trotzdem noch eine gewisse periphere Beschaltung mit geregelter Stromversorgung, Pufferschaltungen etc., aber diese ist vergleichsweise einfach und preisgünstig machbar.

## Endkomponenten

Sensoren oder steuerbare Schalter, wie zum Beispiel Licht-, Wärme-, Druck-, Tast-Sensoren, LEDs, LCD-Anzeigen, Lautsprecher, Relais, Optokoppler usw. Es gibt eine Reihe von online-Shops, bei denen man solche Dinge beziehen kann. Nebst alteingesessenen Betrieben wie Conrad oder ELV gibt es auch eine Menge kleinerer Firmen, die zum Teil günstigere Preise und gerade in diesem Bereich auch eine grössere Auswahl bieten. Als Beispiele seien Mikroshop (https://www.mikroshop.ch), Funduino (https://www.funduinoshop.com/) oder DIY-Shop (https://www.diy-shop.ch/de/) genannt Googlen Sie am besten einfach nach dem Namen bzw. der Produktnummer der Komponente die Sie benötigen, und Sie werden fündig werden.

Viele der ersten Suchtreffer werden von Aliexpress sein. Das ist eine Sammelfirma für tausende von winzigen bis riesigen chinesischen Herstellern. Wenn Sie einige Wochen warten können, und

auch damit umgehen können, wenn die Produkte manchmal etwas anders sind, als erwartet und ohne jede Dokumentation kommen, dann kann das ein gute Option sein. Die Produkte kosten oft weniger als ein Fünftel der hiesigen Preise.

In der Regel ist es aber meines Erachtens vernünftig, hiesige Geschäfte zu berücksichtigen: Die Lieferung erfolgt schneller, bei Problemen bekommt man rascher Hilfe, die Garantiebedingungen sind klar und durchsetzbar, und man ist auch nie in Gefahr, unwissentlich gegen irgendwelche Zoll- oder Einfuhrbestimmungen zu verstoßen.

## Steuergeräte

Natürlich benötigen wir irgendein "intelligentes" Gerät, das die Endkomponente ansteuern kann. Dieses Gerät ist der Vermittler zwischen dem Endgerät und unserer Heimautomations-Zentrale. Hier gibt es zwei grundsätzlich verschiedene Herangehensweisen: Das Programm kann direkt auf diesem Gerät laufen, Steueraufgaben mit einer gewissen eigenen Intelligenz erledigen, oder aber es beschränkt sich darauf, Sensordaten weiterzuleiten und Steuerbefehle entgegenzunehmen und den Aktor entsprechend anzusteuern.

Welchen Weg man wählt, hängt von der Aufgabe, von den zur Verfügung stehenden Mikrocontrollern/SOCs und natürlich den individuellen Präferenzen ab. Auf einem Raspberry wird man auch komplexere Programme direkt ablaufen lassen können, während man auf einem Arduino nur vergleichsweise einfache Aufgaben direkt erledigen kann. Für komplexere Aufgaben wird man ihn eher fernsteuern - zum Beispiel mit einem Raspberry.

## Raspberry Pi

Zur Steuerung von Komponenten kann man den uns schon bestens bekannten Raspberry Pi einsetzen. Auch wenn wir sie bisher noch nicht benutzt haben, sind Ihnen die GPIO-Pins mit Sicherheit schon aufgefallen. Diese Anschlüsse kann man zum Lesen oder Schreiben von Schaltimpulsen verwenden. Da Steueraufgaben nicht viel Rechenleistung brauchen, ist es im Grunde egal, welchen Raspberry Sie verwenden. Für kleine Peripheriegeräte empfiehlt sich meines Erachtens der Zero W, den es für knapp 10 Euro gibt, und der bereits per WLAN angebunden werden kann. Oder vielleicht haben Sie von Ihren ersten Raspberry-Gehversuchen noch einen alten, langsamen Model B im Keller. Auch der tut es, braucht allerdings ein Netzwerkkabel oder einen Wifi-Stick für die Einbindung ins Netz.

Die Pin-Nummerierung der Raspberries ist leider reichlich verwirrend: es gibt verschiedene Systeme (Broadcom-Pin, Header-Nummer, GPIO, Wiring-Pi usw.), die bezüglich Nummernfolge keiner Logik zu folgen scheinen. Leider sind auf dem Pi selbst auch keine Pin-Nummern aufgedruckt. Schauen Sie im Zweifelsfall auf einem entsprechenden Schaubild, z.B. http://pi4j.com/pins/model-3b-rev1.html oder http://pinout.xyz wie es bei Ihrem Pi ist. Prüfen Sie zweimal, ob Sie jeden Draht richtig gesteckt haben, bevor Sie einschalten. Wenn Sie falsche Verbindungen ziehen, und dann Strom darauf geben, geht höchstwahrscheinlich etwas kaputt. Oft leider der Pi.

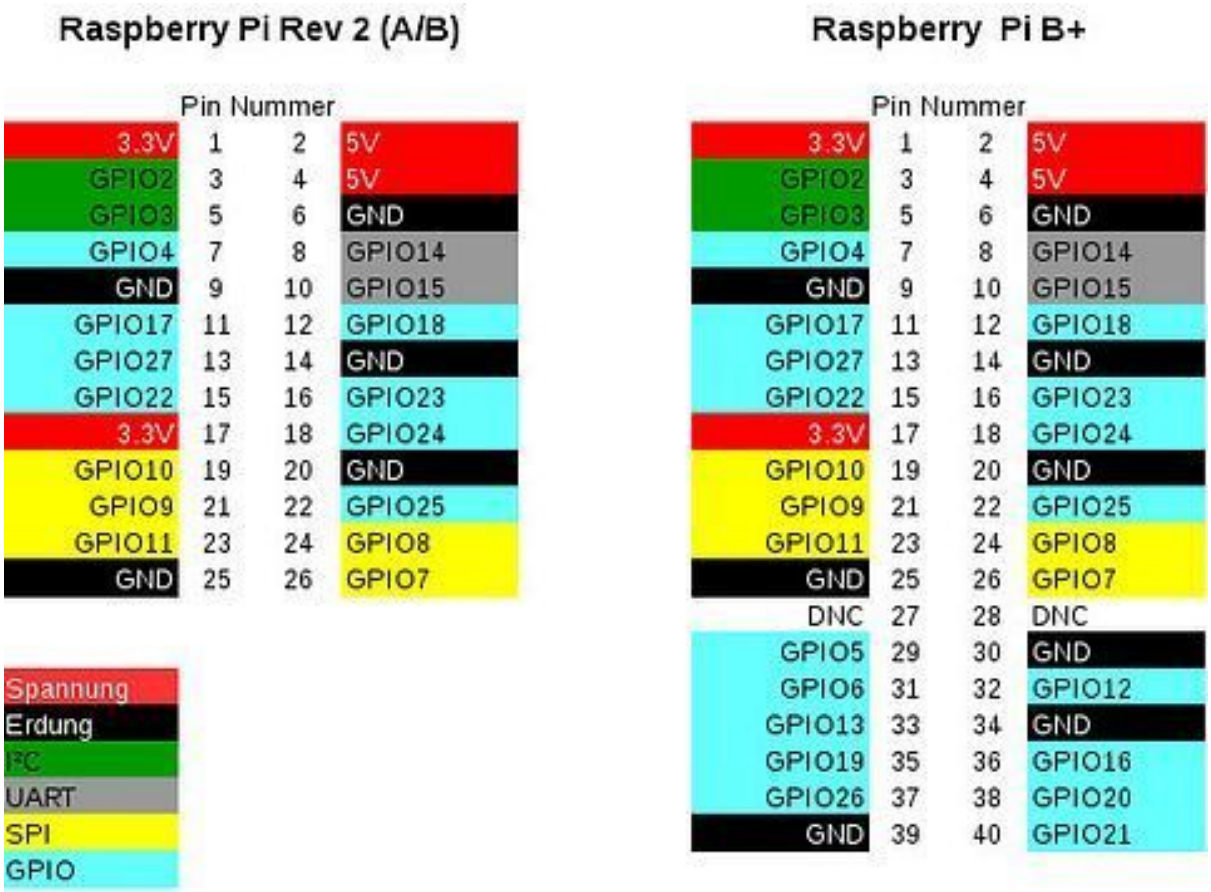



**Abb. 6.1: Raspberry Pin layout**

Raspberries können über I2C, SPI und die serielle Schnittstelle mit Peripheriegeräten kommunizieren. Man muss dabei aber beachten, dass der Raspi keine Spannungen über 3.3 Volt verträgt, viele andere Geräte aber hemmungslos mit 5 Volt kommunizieren.

## Arduino

Der Arduino ist klar der Platzhirsch unter den Mikrocontrollern im Hobbybereich. Er steht hier nur deshalb erst an zweiter Stelle, weil in diesem Buch der Raspberry schon in Verwendung und darum beim Leser vermutlich schon vorhanden ist. Arduino ist eine ursprünglich für italienische Schulen entwickelte OpenSource-Plattform[2]. Wegen Lizenzquerelen entstanden verschiedene Markennamen (Arduino, Genuino), und es gibt inzwischen eine unübersehbare Zahl von Varianten und Nachahmerprodukten. Dazu sollte man noch wissen, dass es bei vielen Clones, die einen billigeren Chip zur USB Anbindung verwenden, manchmal knifflig sein kann, sie mit dem Computer zu verbinden. Anfängern würde ich in jedem Fall zu Original Arduino/Genuino Modellen raten. Im Folgenden meine ich jeweils den Arduino/Genuino Uno R3, aber alle Varianten sind ähnlich genug, dass das keine grosse Rolle spielt.

Arduinos sind vergleichsweise schwachbrüstig: Der Uno hat ganze 32 kB Speicher und läuft mit 16 Mhz. Ihr Handy hat vermutlich mehr als hunderttausendmal so viel Speicher und die hundertfache Taktfrequenz. Das darf aber keinen falschen Eindruck erwecken: Ein Mikrocontroller

[2]OpenSource Plattform heisst, dass auch die Hardware OpenSource ist - Im Gegensatz etwa zum Raspberry Pi darf somit jeder den Arduino nachbauen. Allerdings darf man so einen Nachbau dann nicht Arduino nennen - eine Regel, die viele Nachahmer leider nicht berücksichtigen.

muss keine Videos abspielen, sondern er soll nur Mess- und Schaltaufgaben erledigen. Dafür ist die Leistung absolut ausreichend. Da er kein Betriebssystem laden muss, beginnt ein Arduino (bzw. dessen Firmware) sofort nach dem Einschalten, sein Programm abzuarbeiten. Man muss ihn am Ende auch nicht irgendwie "herunterfahren" sondern kann einfach den Stecker ziehen. Programme für den Arduino kann man komfortabel auf dem PC in der "Arduino IDE" (https://www.arduino.cc/en/main/software) entwickeln und dann über den USB Anschluss auf den Arduino übertragen. Die Programmiersprache ist im Prinzip C++ mit einigen Einschränkungen und dafür einigen systemspezifischen Erweiterungen, etwa Befehle, um GPIOs zu schalten. Für viele Standardaufgaben existiert ein riesiger Fundus an vorgefertigten Libraries, so dass das Programmieren sich oft darauf beschränkt, die richtige Library zu finden und diese mit ein paar wenigen Programmzeilen anzubinden.

Das Programm wird in einem nichtflüchtigen Speicher gehalten und startet bei jedem Einschalten sofort, bis es von einem anderen Programm überschrieben wird. Beim Arduino sind nie mehrere Programme gleichzeitig im Speicher. Arduinos brauchen relativ wenig Strom und können daher, speziell in der "Arduino Nano"-Variante auch gut mit Batterien betrieben werden.

Der Arduino Uno ist einsteigerfreundlich, denn er kommt mit verschiedenen Stromquellen zwischen 5 und 15 Volt zurecht und verzeiht auch schonmal eine falsche Verschaltung ohne kapitalen Schaden.

Eine weitergehende Einführung würde den Rahmen dieses Buches sprengen; bei Interesse empfehle ich, Google mit "Arduino" zu füttern oder direkt auf der Seite http://arduino.cc vorbeizuschauen.

Für uns interessant: Auch Arduinos können via I2C, SPI und serielle Schnittstelle kommunizieren.

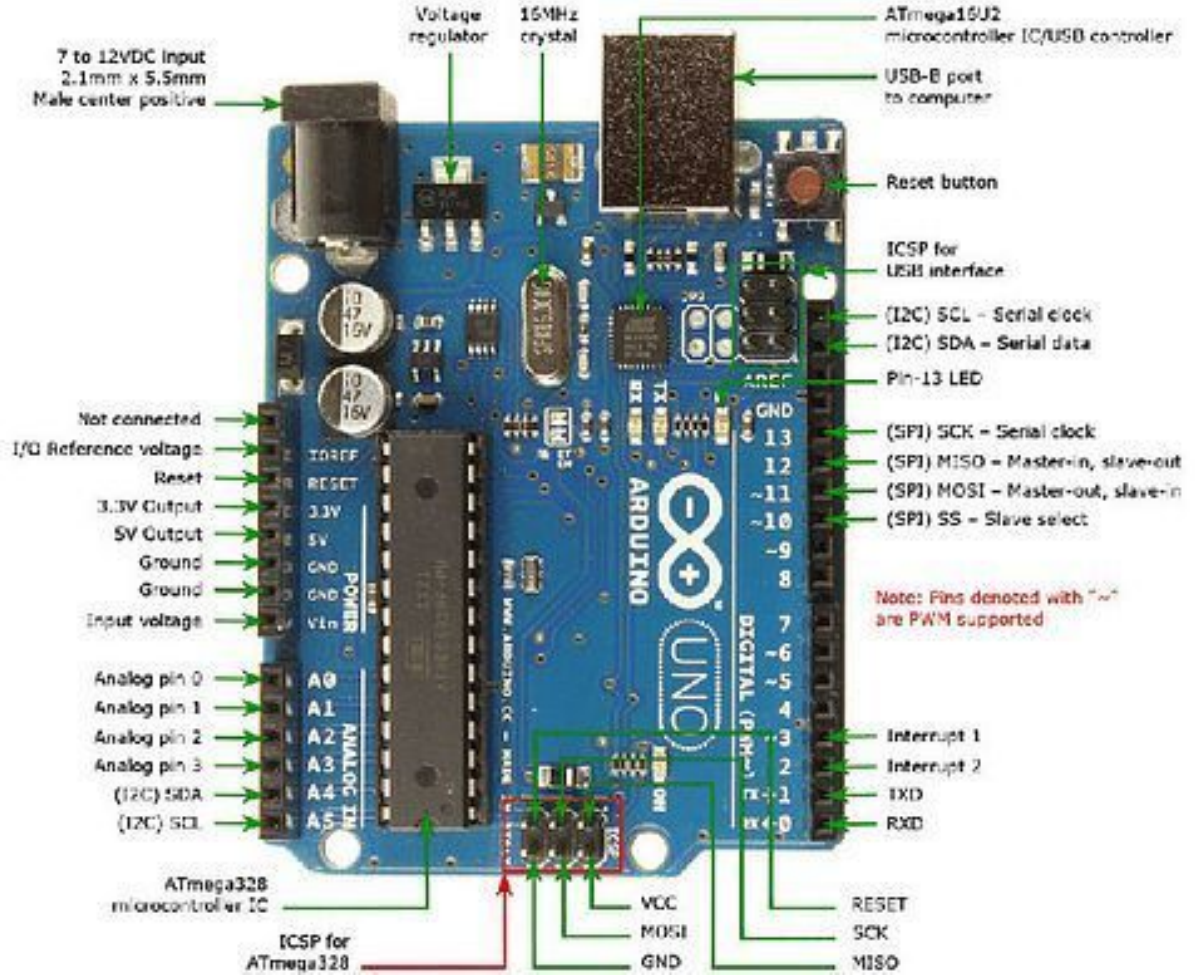



**Abb. 6.2: Arduino Pin Layout**

Ein großer Vorteil für Bastler gegenüber dem Raspberry: Die Pins sind beschriftet, die Verwechslungsgefahr daher deutlich geringer.

Zur Anschauung hier ein kurzes Arduino-Programm, welches eine LED an GPIO 3 blinken lässt:

```
  #define LED 3
  void setup() {
    pinMode(LED, OUTPUT);
  }
  void loop() {
    digitalWrite(LED, HIGH);
    delay(500);
    digitalWrite(LED, LOW);
    delay(1000);
}
```

## ESP8266 und ESP32

Diese Chips des chinesischen Herstellers Espressif sind vielleicht wegen des etwas sperrigeren Namens im Hobby-Bereich weniger bekannt, als die Arduinos, sind aber gleichwohl in den

letzten Jahren mit über 100 Millionen verkauften Exemplaren inzwischen zu den meistverkauften IoT-Geräten geworden. Viele WiFi-fähige IoT-Geräte haben einen dieser Chips an Bord.

Das ist an sich kein Wunder: Ein ESP8266-Board kostet weniger als ein Arduino, hat aber einen mit bis zu 160 Mhz deutlich schneller getakteten Prozessor und je nach Version auch mehr Speicher und ebenso frei programmierbare GPIOs. Und als wichtigstes Merkmal haben diese Chips integriertes WLAN und, im Fall des ESP32, auch Bluetooth an Bord. Beides braucht beim Arduino relativ teure Zusatzhardware.

Der ESP8266 kam 2014 auf den Markt, der leistungsfähigere ESP32 folgte 2016. Für den älteren, aber für die meisten Zwecke ausreichenden ESP8266 findet man generell mehr Unterstützung im Internet und mehr kompatible Software, so dass ich mich im Folgenden auf diesen beschränke.

Man kann den ESP8266 (ebenso den ESP32) entweder direkt als "nackten" Chip oder in einer minimalen Breakout-Package kaufen. Beides ist nicht empfehlenswert. Stattdessen sollte man sich als Amateur eher für ein "Development Board" entscheiden. Ein solches Board ist immer noch recht klein, enthält aber alles, was man braucht, um den ESP8266 direkt an den USB Port eines Computers anzuschliessen und dort zu programmieren. Auch bei den Development Boards hat man allerdings die Qual der Wahl zwischen einer Vielzahl von Herstellern. Als besonders empfehlenswert würde ich derzeit die NodeMCU, die Wemos D1 Mini und die Adafruit Feather-Boards hervorheben. Diese Geräte kosten bei europäischen Händlern um 5-15€, aus China 1-2€, sind also für Smarthome Projekte ohne Weiteres erschwinglich. Der Markt entwickelt sich allerdings so schnell, dass Sie besser eigene Recherchen machen, als sich auf meine möglicherweise bereits veralteten Empfehlungen hier abzustützen. Das, was ich Ihnen im Weiteren zeigen möchte, sollte mit jedem beliebigen ESP8266-Board möglich sein, das Sie erfolgreich mit Ihrem Computer verbinden können.

Hier als Beispiel eine NodeMCU:

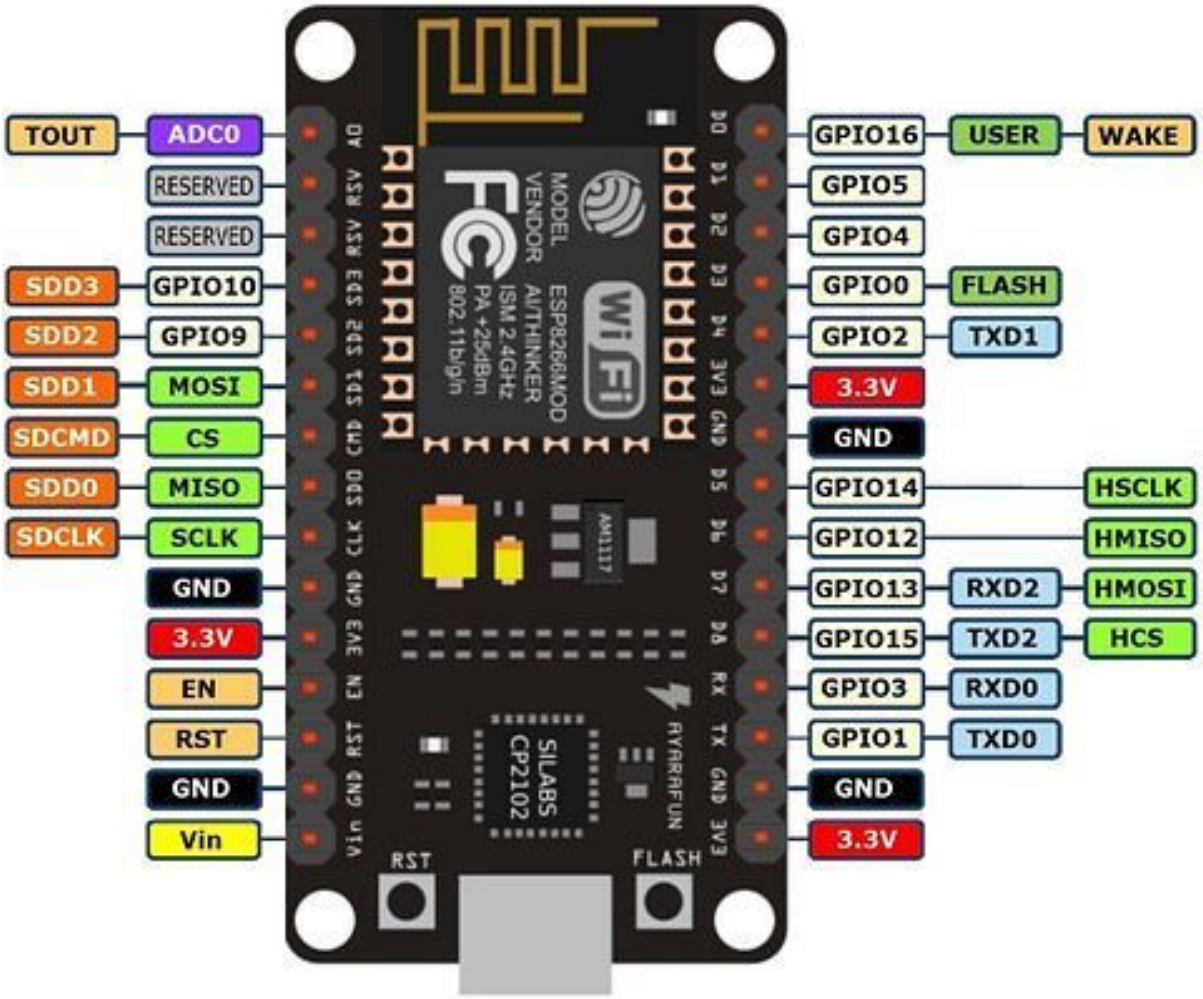



**Abb. 6.3: NodeMCU Pin Layout**

Zum Programmieren kann man eine Vielzahl von Tools und IDEs verwenden, die meisten Programmiersprachen von JavaScript über C/C++ bis Python haben ihren Platz. NodeMCU etwa ist nicht nur eine Platine, sondern auch ein ganzes Ökosystem aus Firmware und Programmierphilosophie, basierend auf der Skriptsprache Lua. Viele Hobbyisten nutzen jedoch die Tatsache, dass man die ESP-Familie auch ganz einfach in die Arduino IDE einbinden und Arduino-Programme ohne grosse Änderungen verwenden kann. Das macht nicht nur die Lernkurve flacher, sondern hat auch den Vorteil, dass man die existierende riesige Zahl von Bibliotheken für den Arduino mit allenfalls minimalen Änderungen verwenden kann, und somit das Rad nicht immer wieder neu erfinden muss.

Wenn das Verbinden des NodeMCU mit dem Computer nicht klappt, müssen Sie eventuell einen USB-Treiber laden: https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

Wenn Sie sich allerdings bisher noch nicht an die Arduino IDE "gewöhnt" haben, sollten Sie sich vielleicht zunächst auch andere SDKs anschauen, an vorderster Front für den NodeMCU natürlich dessen eigene firmware und SDK: https://github.com/nodemcu/nodemcu-firmware, erwähnenswert aber auch PlatformIO https://platformio.org und MongooseOS https://mongoose-os.com/software.html

Und, last but not least, eine sehr interessante Alternative ist ESPEasy (https://www.letscontrolit.com/wiki/index.php/ESPEasy). Hier enthält die Firmware einen kompletten Webserver, der sich bequem per Web-Interface programmieren lässt. Zu diesem Zweck sind eine ganze Reihe von Sensoren vorinstalliert. Ich werde etwas später ein Projekt auf einem NodeMCU via ESPEasy erstellen.

## Breadboards und Jumperkabel

Fast nie lötet man ein Projekt sofort zusammen. Stattdessen baut man es zunächst auf einem Experimentierbrett auf und prüft, ob es sich tatsächlich so verhält, wie es soll. Als Standard hat sich ein bestimmtes Layout ebenso wie der Name Breadboard[3] etabliert.

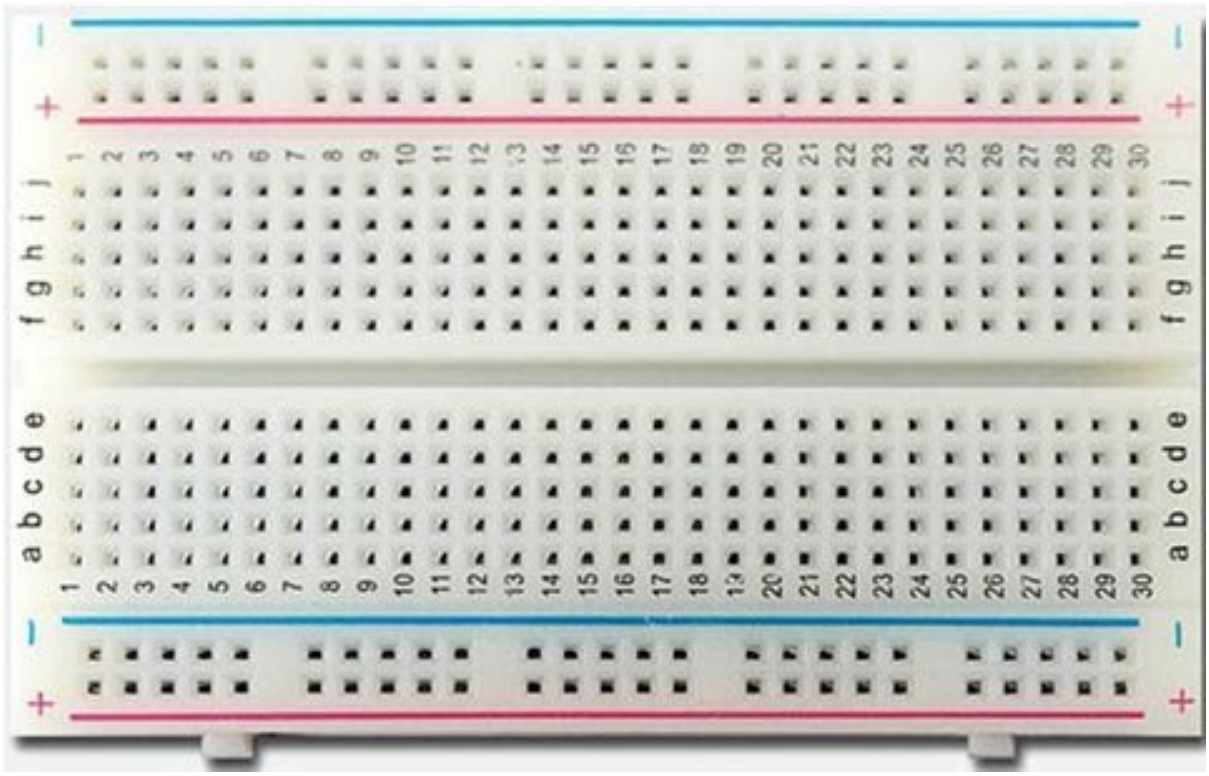


**Abb. 6.4: Breadboard**

Meist (nicht immer) befinden sich auf beiden Längsseiten Kontaktreihen für die Stromversorgung, wobei alle Anschlüsse derselben Seite und derselben Polung miteinander verbunden sind. In der Mitte des Bretts finden sich zwei oder mehrere Reihen von Fünferblöcken, die jeweils miteinander verbunden sind. Zur besseren Beschreibbarkeit eines Versuchsaufbaus sind die Reihen und Spalten oft auch beschriftet, wie hier. Hier sind also beispielsweise 3 a-e und 3 f-j jeweils miteinander verbunden, nicht aber 3e und 3f oder 2e und 3e.

Die Löcher sind in einem bestimmten Rastermass (2.54 mm bzw. 1/10 Zoll) angeordnet, das nicht ganz zufällig exakt zu den Beinchen vieler Elektronik-Komponenten passt. Zur Verdrahtung verwendet man am besten vorgefertigte “Jumper Wires”, die es in verschiedenen Längen zu

[3]Der Name kommt angeblich daher, dass Elektroniker früher ihre Prototypen tatsächlich auf Brotbrettchen aufgebaut haben. Und falls das nicht stimmen sollte, so ist es doch gut erfunden: http://www.instructables.com/id/Use-a-real-Bread-Board-for-prototyping-your-circui/

kaufen gibt. Zur Überbrückung kurzer Distanzen kann man sich auch mit Heftklammern behelfen.

## Platinen etc. für definitiven Aufbau

Wenn das selbst gebaute IoT Gerät tut, was es soll, sollte man es allerdings nicht als Steckbrett-Aufbau in den harten Alltag entlassen. Die Kontakte sind etwas wacklig, das Ganze ist nicht stabil, und die Lieblings-Katze könnte in einer destruktiven Phase die Arbeit von vielen Tagen zerstören.

Es hilft nichts: Für den definitiven Aufbau sollten Sie sich an den Gedanken gewöhnen, einen Lötkolben in die Hand zu nehmen. Das ist nicht so schwierig, wie man es sich als Einsteiger oft vorstellt. Am besten schauen Sie sich dazu ein Lehrvideo an, wie zum Beispiel dieses: https://www.youtube.com/watch?v=4DWUZp1t7Ls. Am Lötkolben würde ich nicht unbedingt sparen - mit einem 10-Euro-Gerät werden Sie vermutlich bald ein Mehrfaches der Einsparung für kaputte Bauteile und Magentabletten ausgeben. Der Lötkolben soll gut in der Hand liegen und in nützlicher Frist 300-350°C erreichen können.

Als Träger für die Elektronik bieten sich Lochrasterplatinen an (wenn Sie nicht gleich Leiterplatten selber ätzen wollen). Da gibt es verschiedene Varianten mit Lötstreifen, Lötpunkten, Lötdoppelpunkten etc. und verschiedene Materialien von Glasfaser bis Hartpappe, getränkt mi Epoxydharz, Melamin oder Polyester. Ich bevorzuge nach einigem Experimentieren simple Lötpunkte auf Hartpappe. Letzteres, weil es sich einfacher auf Mass schneiden lässt. Auch die Hartpappe ist übrigens mit Kunststoff getränkt und ausgehärtet, so dass sie wesentlich beständiger ist, als der Begriff suggeriert. Aber sie lässt sich trotzdem leichter bearbeiten, als eine Glasfaser-Basis.

## Bonus: 3D Drucker fürs Gehäuse

Um die Katze noch weiter zu desavouieren, sollte man seinen Elektronikbauwerken ein schützendes Gehäuse spendieren. Gehäuse gibt es in allen möglichen Formen und Grössen zu kaufen, aber natürlich ist just gerade diese spezielle Form mit dieser speziellen Befestigungsart, die wir jetzt gerade brauchen, entweder nicht vorrätig oder überhaupt nicht zu bekommen. Dann hilft nur selber machen. 3D Drucker sind absolut erschwinglich geworden. Wenn Sie nicht gerade sehr grosse Stücke bauen wollen, und mit einer Genauigkeit im Fünftelmillimeterbereich leben können, kommen Sie mit 200 bis 300 Euro zu einem brauchbaren Gerät, das Ihnen robuste und formschöne Gehäuse, Verankerungen, Schienen und Klammern herstellen kann. Falls Sie Inspiration brauchen, können sie sich zum Beispiel auf Thingiverse umsehen. Hier einige Beispiele für NodeMCU Gehäuse: https://www.thingiverse.com/search?q=nodemcu. Diese Entwürfe können Sie direkt ausdrucken, ausgedruckt kaufen, verändern, oder einfach nur als Anregung für eigene Projekte verwenden.

Der Entwurf solcher 3D-Stücke braucht entweder ein gewisses gestalterisches Talent, oder mathematisches Verständnis. Wenn Sie eher zu Ersterem tendieren, hilft Ihnen ein visuelles 3D Programm, wie zum Beispiel das sehr einsteigerfreundliche kostenlose Tinkercad https://www.tinkercad.com/. Für fortgeschrittene Designer gibt es eine Vielzahl von Alternativen, aber fortgeschrittene Designer haben bestimmt schon ihr Lieblingsprogramm gefunden, und

brauchen hier keine Hinweise. Wenn Sie hingegen eher mathematisches, als künstlerisches Geschick haben, sei Ihnen OpenScad (http://www.openscad.org) ans Herz gelegt. Mit OpenScad erstellen Sie dreidimensionale Objekte mit einer Programmiersprache.

Beispielsweise konstruiert das hier:

```
  translate([10,0,0])
    cube([10,10,10]);
```

Einen Würfel mit 10 Millimetern Kantenlänge an der Position 10,0,0.

Und dieses Programm:

```
// Diameter of the fan
blade_length=60; // [20:100]
// Width of a blade
blade_width=7; //[5:10]
// Diameter of the axe
axe_diameter=2;  // [2:10]
// Diameter of the center hub
hub_diameter=18; // [10:30]
// Angle of the blades
angle=45; // [30:60]
// Height of the center hub
height=blade_width*2.5*cos(angle);

// compensate shrink for axe hole
shrink=1.1;

propeller();

module propeller(){
  difference(){
    cylinder(d=hub_diameter,h=height,$fn=100);
      for(i=[0:60:360])
        rotate([0,0,i])
            blade();
    cylinder(d=axe_diameter*shrink,h= 8,$fn=40);
  }
}

module blade(){
   translate([0,0,height/2])
    rotate([90,angle,0])
      linear_extrude(height=blade_length/2)
        scale([0.2,2.5])
          circle(d=blade_width,$fn=50);
}
```

erzeugt das hier:

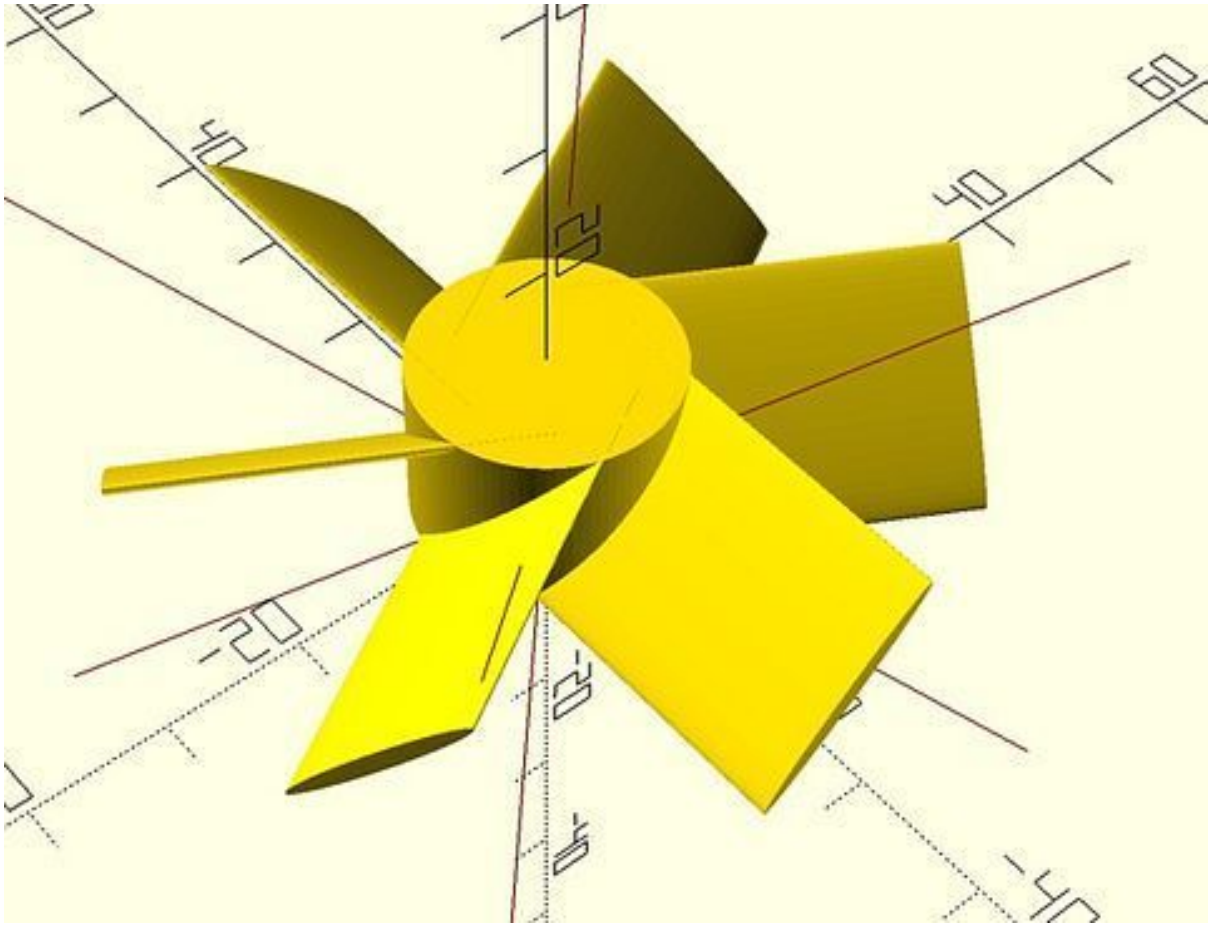



**Abb. 6.5: Propeller**

Sie können sich vorstellen, dass so ein "technisches" Objekt mit Freihandzeichnen, künstlerische Begabung hin oder her, eher schwierig herzustellen und zu variieren wäre. In diesem Programm dagegen ist es ein Klacks, die Rotorblätter etwas länger und in einem anderen Anstellwinkel zu fertigen. Also wählen Sie Ihr Werkzeug je nach Talent und Anforderung, und probieren Sie anfangs auch unterschiedliche Werkzeuge aus.

Egal, mit welchem Programm Sie das Objekt erstellt haben, im Anschluss müssen Sie es als *.stl-Datei (oder ein anderes Format, das der Slicer versteht) exportieren, und ein weiteres Programm, eben diesen Slicer bemühen. Ein Slicer schneidet, wie der Name sagt, das 3D Objekt in hauchdünne horizontale Scheiben (Wobei "hauchdünn" je nach Druckermodell und Einstellungen etwas wie 0.2 oder 0.4 Millimeter bedeutet.) Bekannte Slicer-Programme sind etwa "cura" oder "slic3r", wobei letzteres mein Favorit ist.

Diese Scheiben werden dann in eine Reihe von Aktionsbefehlen für den Drucker in einer Sprache namens "Gcode" umgesetzt. Keine Angst, das geht vollautomatisch. Diese .gcode-Datei wiederum kann man dann dem 3D-Drucker zum Frass vorwerfen, und dieser wird, falls er mit ausreichend Filament versehen ist, in etlichen Minuten bis Stunden Druckzeit Scheibe um Scheibe das Objekt aufbauen. Und dieses wird eine verblüffend hohe Stabilität haben. Ein echtes Kunststoff-Objekt, allerdings mit dem Vorteil, dass dieser Kunststoff (Wenn Sie PLA verwenden) aus nachwachsenden Rohstoffen hergestellt und biologisch abbaubar ist.

Es wird eine Weile dauern, bis Sie brauchbare Resultate bekommen, und noch länger, bis Sie richtig gute Resultate bekommen. Man braucht eine gewisse Erfahrung, um sofort zu erkennen, welche Strukturen druckbar sind, und welche nicht (z.B. Überhänge nur bis zu einem gewissen Winkel, Brücken nur bis zu einer gewissen Länge). Und zu viele Parameter kann und muss man beim 3D-Druck variieren und zu viel kann schief gehen. Aber es macht auch viel Freude, "echte" Objekte herzustellen, die einzigartige Unikate sind (zumindest so lange, bis Sie sie ein zweites Mal ausdrucken oder auf Thingiverse publizieren).

Gehäuse für einige der in diesem Kapitel diskutierten Basteleien finden Sie im Quellcode Verzeichnis, das Sie wie im Anhang gezeigt clonen können, und dann:

```
git checkout -f origin/master
git clean -f
```

Und zwar im Unterverzeichnis "Bastelstunde", jeweils als *.stl, *.scad und *.png Datei.

Genug der Theorie, jetzt bauen wir ein richtiges IoT-Gerät, und das mit einem Aufwand von wenigen Euro.

# Barometer

## Einleitung

Ein Heimautomationssystem braucht zweifellos auch ein Monitoring des Luftdrucks. Schließlich wollen wir den Boiler und das Auto nur dann mit Nachtstrom laden, wenn die Wetteraussichten für den nächsten Tag wenig Sonne versprechen, und die Kaffeemaschine für extra starken Kaffee vorbereiten, wenn die Wetteraussichten trübe sind. Sicher, es gibt Wettervorhersage-Apps, aber wir erhalten genauere Prognosen, wenn wir deren Angaben mit eigenen Luftdruck- und Temperaturmessungen kombinieren.

Für den Luftdruck fand ich keine Homematic Komponente. Wir müssen uns da also selbst behelfen. Von Bosch gibt es Barometer-Thermometer-Chips für kleines Geld, die sich für derartige Zwecke ausgezeichnet eignen. Die Chips tragen je nach Version die Bezeichnungen BMP-180, BMP-280 oder BME-280 (Mit Hygrometer). Es handelt sich allerdings um winzige Dinger, die man als Hobbyist unmöglich direkt irgendwo einsetzen kann. Die Lötstellen sind mikroskopisch klein, und es ist periphere Beschaltung notwendig. Glücklicherweise gibt es von verschiedenen Herstellern Breakout Boards, die die periphere Beschaltung erledigen und nur die interessierenden Anschlüsse in Hobbylöt-freundlicher Grösse nach Aussen führen. Von Adafruit und Grove gibt es derartige Boards für unter 20 Euro; wenn man ein paar Wochen warten und damit leben kann, dass man im Voraus nicht ganz sicher weiss, welche Chip-Variante dann tatsächlich verbaut ist, kann man BMP280/BME280/BMP180-Breakouts auch bei AliExpress für unter 4 Euro bekommen.

Ich habe diesen hier verwendet:

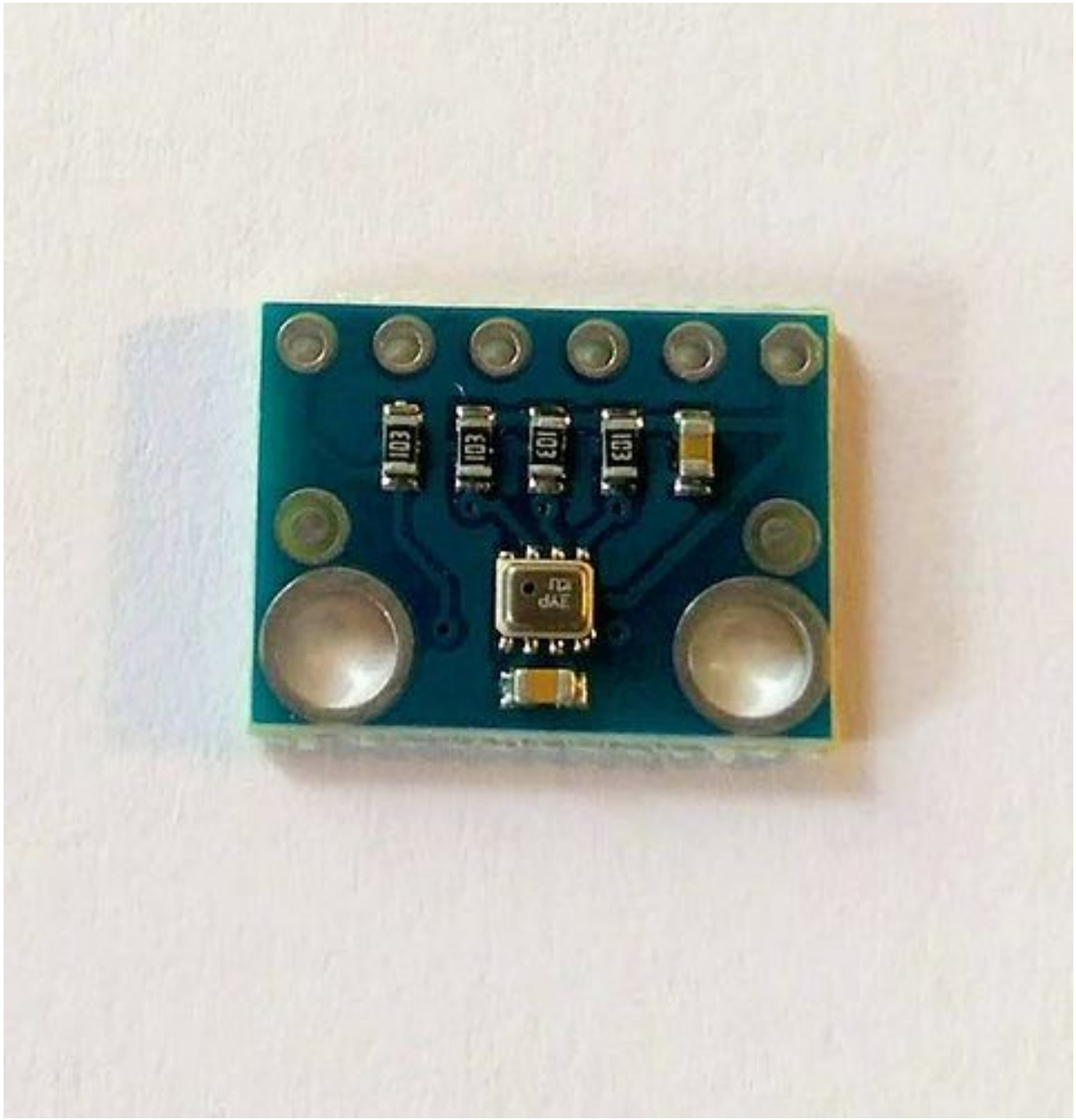


**Abb. 6.6: BMP280 breakout**

Wie Sie sehen, muss man die Pins selber anlöten. Aber das sollte Sie ja nun nicht mehr schrecken. Eine solche Platine ist nicht so empfindlich, wie man vielleicht glaubt, Man kann den Lötkolben durchaus einige Sekunden lang an die Pins halten, ohne dass etwas durchbrennt. Der eigentliche Chip ist ja auch ein ganzes Stück weit von den Lötösen entfernt.

Machen Sie aber bitte nicht denselben Fehler wie ich beim ersten Mal, die Platine "richtig herum", also mit dem Chip nach oben, zu löten. Dann sehen Sie nämlich die Beschriftung der Anschlüsse nicht mehr, wenn sie im Breadboard sitzt… Setzen Sie die Header stattdessen so ein, wie hier gezeigt:

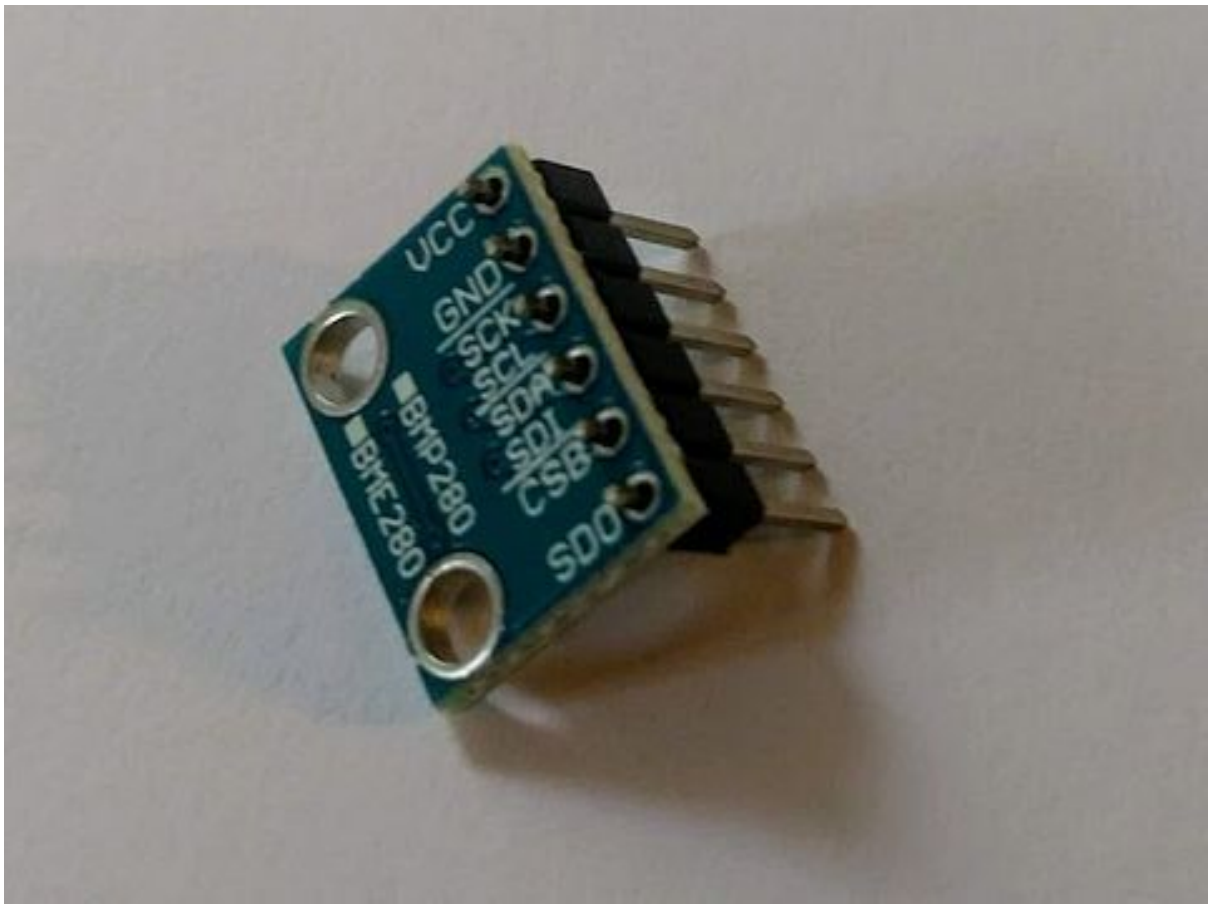



**Abb. 6.7: BMP280 Lötempfehlung**

Dem Chip ist es völlig egal, ob er auf dem Kopf steht, und so sehen Sie jederzeit, welcher Pin wofür ist. Das gilt natürlich nur für den Versuchsaufbau. Im definitiven Aufbau später sollten Sie der Messgenauigkeit wegen darauf achten, dass der Chip möglichst frei und möglichst weit von anderen Komponenten entfernt platziert wird.

Wie liest man die Messwerte aus, die ein solches Bauteil liefert? Dafür gibt es, wie nicht anders zu erwarten, mehrere Standards, die ich in der Einleitung kurz angesprochen hatte. Bosch hat diesem Chip gleich zwei davon spendiert: I2C und SPI. Die Implementationsdetails brauchen uns jetzt nicht zu interessieren und würden auch zu weit führen, denn wir werden ohnehin eine Library benutzen, die uns das mühsame Einsammeln der Bits abnimmt. Wir müssen nur genug wissen, damit wir für Gerät und Software den richtigen Standard auswählen können.

Zunächst steht die Entscheidung an, wie wir die Daten überhaupt ins Netz holen. Da wir ohnehin einen Raspberry Pi als Heimserver verwenden, bietet es sich an, diesem gleich noch den Nebenjob aufzuhalsen, den BMP-280 auszulesen und dessen Messwerte im ioBroker bereitzustellen. Und wie der glückliche Zufall so spielt, hat der Raspberry schon alles an Bord, was es braucht, um mit einem Peripheriegerät über I2C und SPI zu kommunizieren. Es fehlt nur noch die Software. Weil wir uns inzwischen mit JavaScript und dem NodeJS Ökosystem recht gut auskennen, suchen wir eine NodeJS-Lösung.

Auftritt Johnny-Five (http://johnny-five.io). Fragen Sie mich nicht, warum das so heisst. Johnny-Five ist viel mehr, als nur eine Möglichkeit, den BMP 280 auszulesen. Es ist ein universelles Konzept, Hardware mit verschiedensten Geräten zu steuern. Der Raspberry Pi als Host ist

eigentlich nur zweite Wahl. Sein eigentliches Milieu hat Johnny im Land der Arduinos, Espruinos und wie sie alle heissen: Kleinen Mikrocontrollern mit vielen Ein- und Ausgängen. Man kann ein Johnny-Five Programm mit minimalen Änderungen (Im Wesentlichen die Nummern der Pins) für jede der unterstützen Plattformen verwenden. Wenn Sie den Barometer später an einen ferngesteuerten Arduino hängen wollen, kein Problem!

Aber damit wir den BMP 280 nicht gleich durch falsches Anschließen grillen, geht es zunächst ans Datenblatt: https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMP280-DS001-19.pdf. Ich habe es bereits für Sie studiert, und das Wesentliche ist:

- Johnny-Five kommuniziert via I2C mit einem BMP 280.
- Der BMP 280 wird mit 3.3 Volt betrieben, was just exakt die Spannung unserer Raspberry-I2C Pins ist.
- Wenn wir wollen, dass der BMP 280 den I2C Bus benutzt, müssen wir vor oder spätestes mit Anlegen der Betriebsspannung dafür sorgen, dass der CSB Pin auf HIGH ist. Wenn CSB irgendwann nach dem Anlegen der Spannung nicht mehr HIGH ist, wird der Chip sich für I2C tot stellen. Wir fixieren diesen Pin also auf 3.3 Volt.
- Der BMP 280 kann zwei I2C Adressen benutzen (das ist nützlich, wenn man zwei BMP280 am selben Bus hängen hat). Welche Adresse das ist, entscheidet der Zustand des Pins SDO. Zieht man ihn auf LOW, ist die Adresse 0x76, zieht man ihn auf HIGH, hört er auf 0x77 (und das erwartet Johnny-Five standardmässig). Wenn man SDO gar nicht beschaltet, ist die Adresse undefiniert, was dazu führt, dass der Chip mal erreichbar ist und mal nicht. Kein erwünschtes Verhalten, darum verbinden wir auch SDO fix mit 3.3 Volt. Es gibt allerdings auch Breakout-Boards, die die Adresse bereits fix eingestellt haben, und den SDO-Pin gar nicht nach aussen führen.

Mit diesem Vorwissen geht es weiter auf dem Raspi. Ich würde vorschlagen, zunächst zur Übung, und zum schauen, ob überhaupt alles funktioniert, zunächst ein kleines Testprogramm zu erstellen. Gehen Sie per SSH auf den Raspberry (oder schliessen Sie ihn an Tastatur und Bildschirm an) und erstellen Sie ein Verzeichnis “barometer”. Geben Sie dann Folgendes ein:

```
cd barometer
npm init -y
npm install --save johnny-five raspi-io
```

Dies wird eine Weile dauern. Eventuell (je nach bereits vorhandener Software) meckert der Installer, dass pigpio noch nicht installiert sei, und wird versuchen, es zu installieren, was mangels Admin-Rechten scheitern wird. Brechen Sie in diesem Fall ab und geben Sie zunächst ein:

```
sudo npm install pigpio
```

Und starten Sie dann erneut die Installation von johnny-five und raspi-io

Danach müssen Sie den Raspberry neu starten, damit der I2C Bus aktiviert wird. Ich würde vorschlagen, Sie fahren ihn zunächst herunter, stecken ihn dann aus, erledigen die Verkabelung

zum BMP280 und stecken ihn dann wieder ein. Auf diese Weise ist die Gefahr versehentlicher Beschädigungen am geringsten.

Da der Raspi je nach Version nur einen oder zwei 3.3 Volt Anschlüsse hat, wir aber drei Pins (CSB, SDO und VCC) mit 3.3 Volt verbinden wollen, müssen Sie z.B. drei Kabel zusammenlöten, mit einem Verbinder zusammenfassen oder über ein Steckbrett auf drei Pins verteilen. Achtung: Hier ist Sorgfalt gefragt, Wenn Sie den Sensor versehentlich mit 5 Volt verbinden, können Sie ihn anschließend höchstwahrscheinlich wegwerfen (Ausser, Sie haben eines jener Breakout-Boards erwischt, die auch einen Spannungsregler beinhalten).

Wir verbinden also:

- SDA mit Pin 3/GPIO 8 des Raspberry (Das ist, wenn Sie den Raspberry so vor sich halten, dass er mit der Schmalseite mit den USB-Anschlüssen zu Ihnen zeigt und die GPIO Stiftleise rechts ist, der zweite Pin von oben der linken Stiftleiste).
- SCL mit Pin 5/GPIO 9 (das ist der dritte Pin links von oben)
- GND mit PIN 6 / GROUND (das ist der dritte Pin rechts, gegenüber von SCL)
- CSB,SDO und VCC mit Pin 1 (das ist der oberste Pin links).

Kontrollieren Sie noch einmal, ob alle Kabel richtig verbunden sind und starten Sie dann denn Raspi neu. Öffnen Sie dann wieder die ssh-Konsole und geben Sie ein:

```
cd barometer
nano bmp280.js
```

Dann geben Sie bitte folgendes Prögrämmchen ein:

```
const five=require('johnny-five')
const Raspi=require('raspi-io');
const board=new five.Board({
        io: new Raspi()
})

board.on('ready',()=>{
        const bmp280=new five.Multi({
        controller: 'BMP280',  // Bei Verbindungsproblemen versuchen Sie 'BME280' oder '\
BMP180'
        freq: 1000      // Abfragehäufigkeit in ms.
        })
        bmp280.on('data',()=>{
                let temperature=bmp280.thermometer.celsius.toFixed(1);
                let pressure=bmp280.barometer.pressure.toFixed(1);
                let humidity=bmp280.hygrometer.relativeHumidity
                console.log(`${temperature}°C, ${pressure} kPa, ${humidity}%rH`);
        })
})
```

Beenden Sie den Editor mit CTRL-O und CTRL-X und starten Sie das Programm mit `sudo node bmp280.js`. Vermutlich werden noch ein paar Fehlermeldungen wegen Tippfehlern kommen, aber dann werden Sie mit einer Ausgabe wie dieser belohnt:

```
27.7°C, 96.5 kPa, 0%rH
27.7°C, 96.5 kPa, 0%rH
27.7°C, 96.5 kPa, 0%rH
27.7°C, 96.5 kPa, 0%rH
27.7°C, 96.5 kPa, 0%rH
27.7°C, 96.5 kPa, 0%rH
```

(Da verschiedene Chipversionen im Umlauf sind, die nicht alle dieselben Messwerte liefern, habe ich hier Code für alles eingegeben. Wenn Sie z.B. wie hier für humidity unsinnige Werte erhalten, bedeutet das, dass der eingesetzte Chip die Luftfeuchtemessung nicht unterstützt. Alle Varianten unterstützen aber pressure und temperature.)

Da unterschiedliche Chips im Handel sind, kann es auch sein, dass ein als BMP-280 verkauftes Board einen BME-280 enthält und daher im Skript nur als "BME280" korrekt angesprochen werden kann. Sie müssen eventuell ein wenig experimentieren. Wenn es mit keinem der ähnlichen Chips funktioniert, oder wenn es nur instabil läuft, ist vielleicht die Baudrate des I2C Bus für Ihren vielbeschäftigten Raspi zu schnell eingestellt. Das können Sie direkt auf dem Raspi so ändern:

```
sudo nano /boot/config.txt
```

Suchen Sie dort die Zeile **dtparam=i2c_arm_baudrate=100000** und ändern Sie den Wert in 10000. Speichern Sie config.txt mit CTRL-O und verlassen Sie den Editor mit CTRL-X. Starten Sie dann denn Raspi mit `sudo reboot` neu. Dann sollte es klappen.

Wenn Sie an korrektem Wert für die Temperatur interessiert sind, sollten Sie den Chip weit genug vom Raspberry weg montieren, sonst messen Sie (wie hier) eher dessen Temperatur, als die der Umgebung.

An dieser Stelle möchte ich darauf hinweisen, dass man ein auf dem RaspberryPi laufendes NodeJS Programm auch vom Arbeitscomputer aus debuggen kann, was bei komplizierteren Programmen oft einfacher ist, als einem Fehler mit verstreuten console.log() Anweisungen auf die Spur zu kommen. Sie finden die Anleitung zum Remote-Debugging im Anhang.

## Anbindung an ioBroker

Nachdem unser Barometer grundsätzlich funktioniert, muss es an ioBroker angebunden werden. Die erste Idee ist: Wir benötigen einen ioBroker Adapter. Eine schnelle Websuche liefert allerdings keine Ergebnisse für BMP/BME-280, wir müssten somit selber einen schreiben.

Das kann man natürlich tun; wie es geht, habe ich ja im Kapitel 5 gezeigt. Allerdings gibt es einen viel einfacheren Weg, und den möchte ich Ihnen hier vorstellen.

Dazu hole ich erst mal ganz weit aus und benutze einen NodeMCU zum Anbinden des BMP280. Sie erinnern sich, NodeMCU ist ein Development Board rund um den ESP8266, einen Mikrocontroller mit WLAN.

Wenn Sie eine NodeMCU zur Hand haben, können Sie das Folgende gleich mitmachen, wenn nicht, überfliegen Sie Sie diesen Abschnitt nur; ich werde später das Vorgehen mit dem Raspberry Pi zeigen.

- Besorgen Sie zunächst die Arduino IDE von der Originalquelle https://www.arduino.cc/en/Main/Software.
- Installieren Sie dann die ESP8266 Tools in die Arduino IDE wie hier beschrieben: http://blog.opendatalab.de/codeforbuga/2016/07/02/arduino-ide-mit-nodemcu-esp8266. (Falls Sie die NodeMCU bisher noch nie an den Computer angeschlossen hatten, beachten Sie bitte, dass Sie meist zunächst den Treiber von SiliconLabs (https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers) installieren müssen, damit das Board erkannt wird.)
- Besorgen Sie sich dann die ESPEasy Firmware von der Originalseite: https://www.letscontrolit.com/wiki/index.php/ESPEasy. Folgen Sie den Installationsanweisungen für Ihr System.

Das Einbinden des NodeMCU mit ESPEasy ist ein zwei-Schritt-Prozess, ähnlich dem, den Sie vielleicht von der Einrichtung der Osram Lightify Bridge in Erinnerung haben: Zunächst spannt der Chip einen Accesspoint auf, mit dem man sich verbinden kann, um die Zugangsdaten zum "Echten" Netzwerk einzugeben. Danach kann man ihn über dieses Netzwerk erreichen. Also:

- Nach erfolgreichem Hochladen der Firmware drücken Sie auf den "RST" Knopf des NodeMCU und suchen mit Ihrem Computer nach dem Wifi AccessPoint **ESP_Easy_0** (oder so ähnlich) und verbinden Sie sich mit dem Password *configesp* mit diesem WLAN. Sie gelangen auf eine Seite, auf der Sie die Zugangsdaten für Ihr richtiges WLAN eintragen können.
- Verbinden Sie Ihren Computer wieder mit dem richtigen WLAN und rufen Sie mit dem Browser die Adresse auf, die Ihnen beim Einrichten gezeigt wurde. Wenn alles geklappt hat, sollten Sie jetzt von der Steuerseite des ESPEasy begrüsst werden. Ziehen Sie jetzt vor dem Verkabeln bitte wieder den Stecker (Herunterfahren ist bei Mikrocontrollern nicht nötig), damit wir das Gerät nicht kaputt machen.
- Nehmen Sie das BMP280 Board und Verbinden Sie:
  - VCC, SDO und CSB mit 3.3 Volt
  - GND mit GND
  - SCL mit D1
  - SDA mit D2

Die fliegende Verkabelung könnte etwa so aussehen:

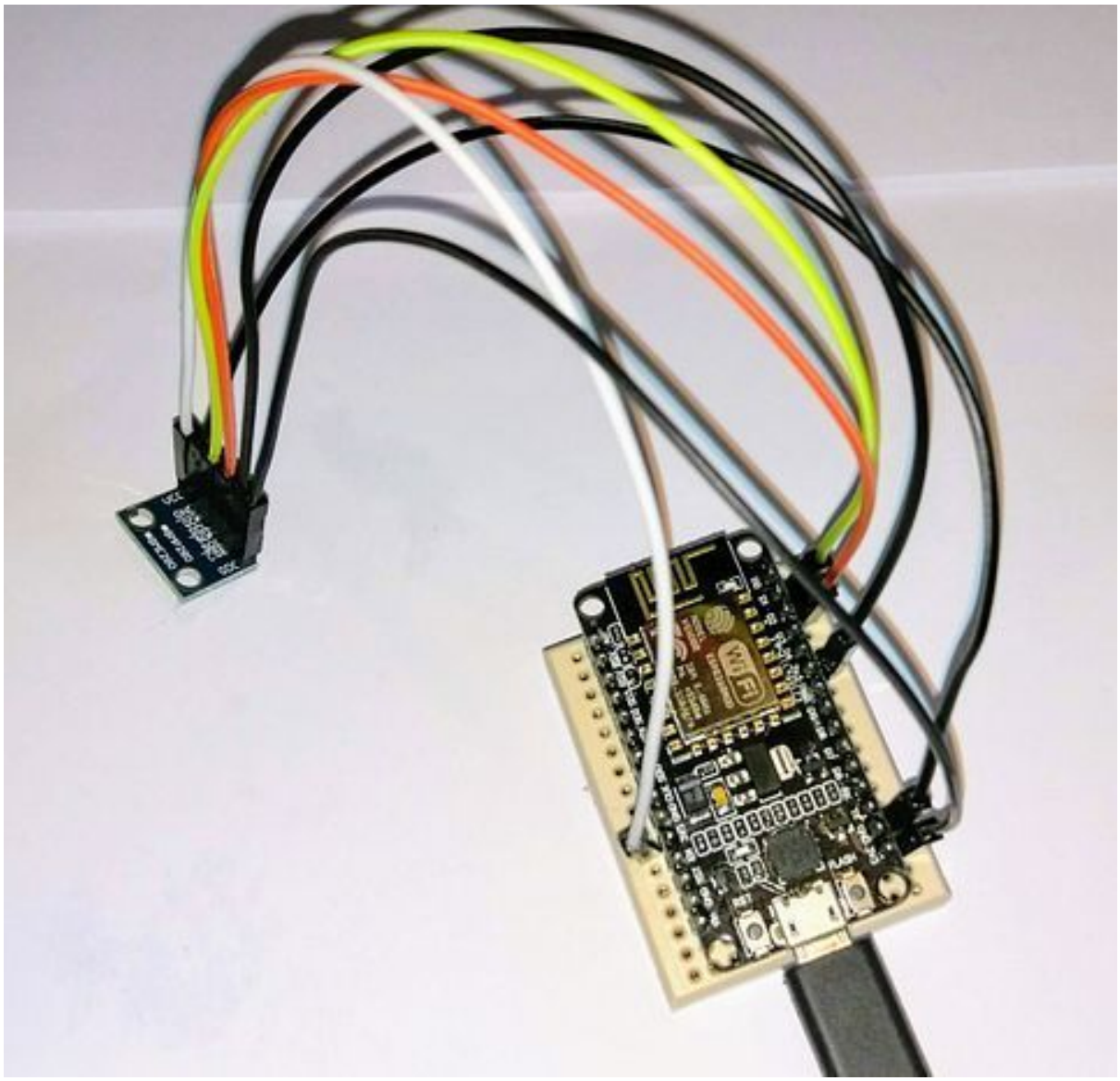


**Abb. 6.8: BMP280 Nodemcu**

Da die NodeMCU mehr 3.3 Volt-Anschlüsse hat, als der Raspberry, brauchen wir hier nichts zu basteln, um VCC, SDO und CSB auf “HIGH” Level zu legen.

Stecken Sie dann die NodeMCU wieder an ein USB Ladegerät oder den USB Port des Computers (direkte Verbindung ist jetzt nicht mehr zwingend nötig, da wir von nun an nur noch per WiFi mit der NodeMCU kommunizieren). Gehen Sie mit dem Webbrowser wieder auf dieselbe Seite wie vorhin und gehen Sie zum Tab “Hardware”. Vergewissern Sie sich, dass dort bei I2C Interface dieselben Pins angegeben sind, wo wir SLC und SDA eingesteckt hatten.

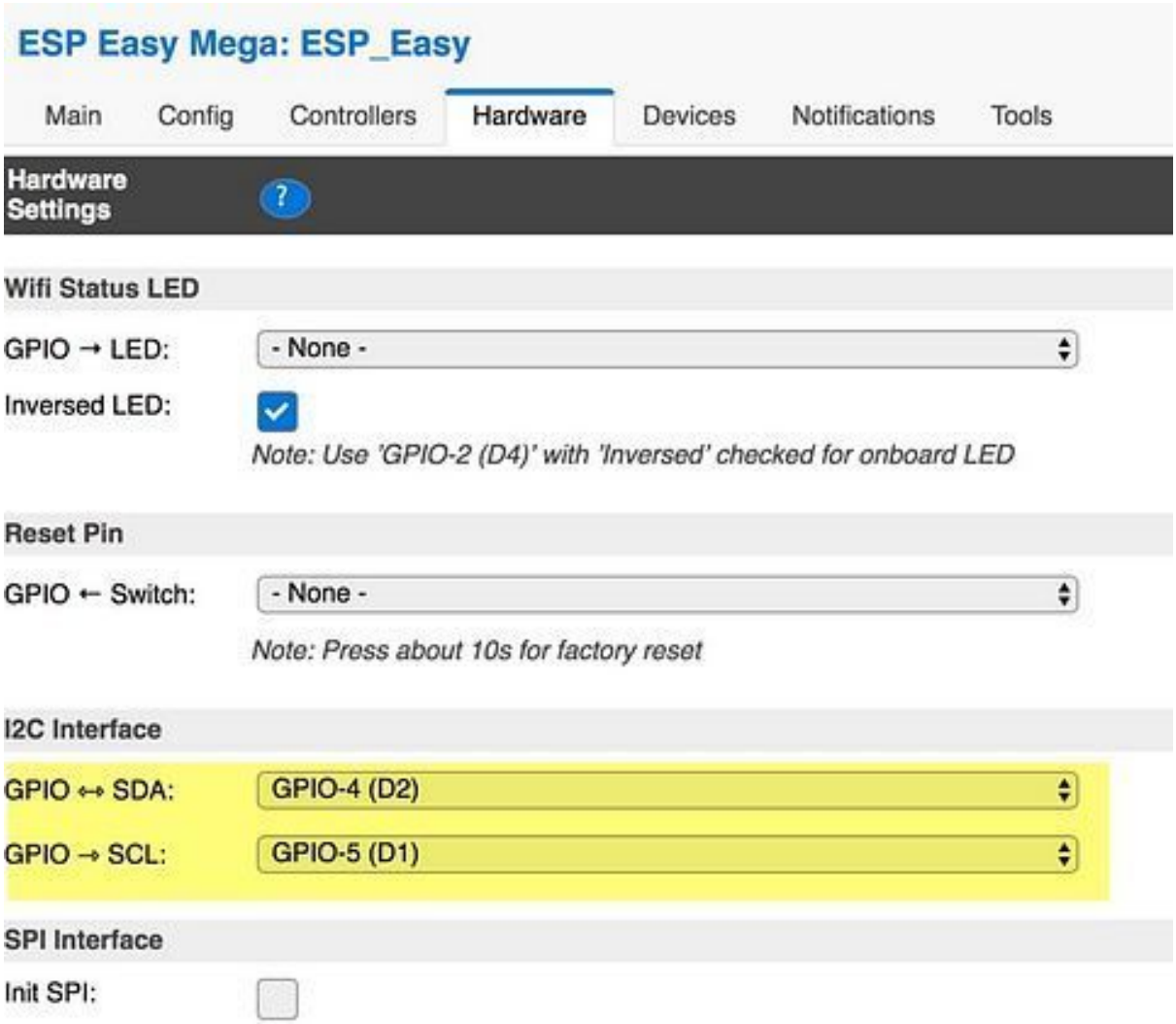
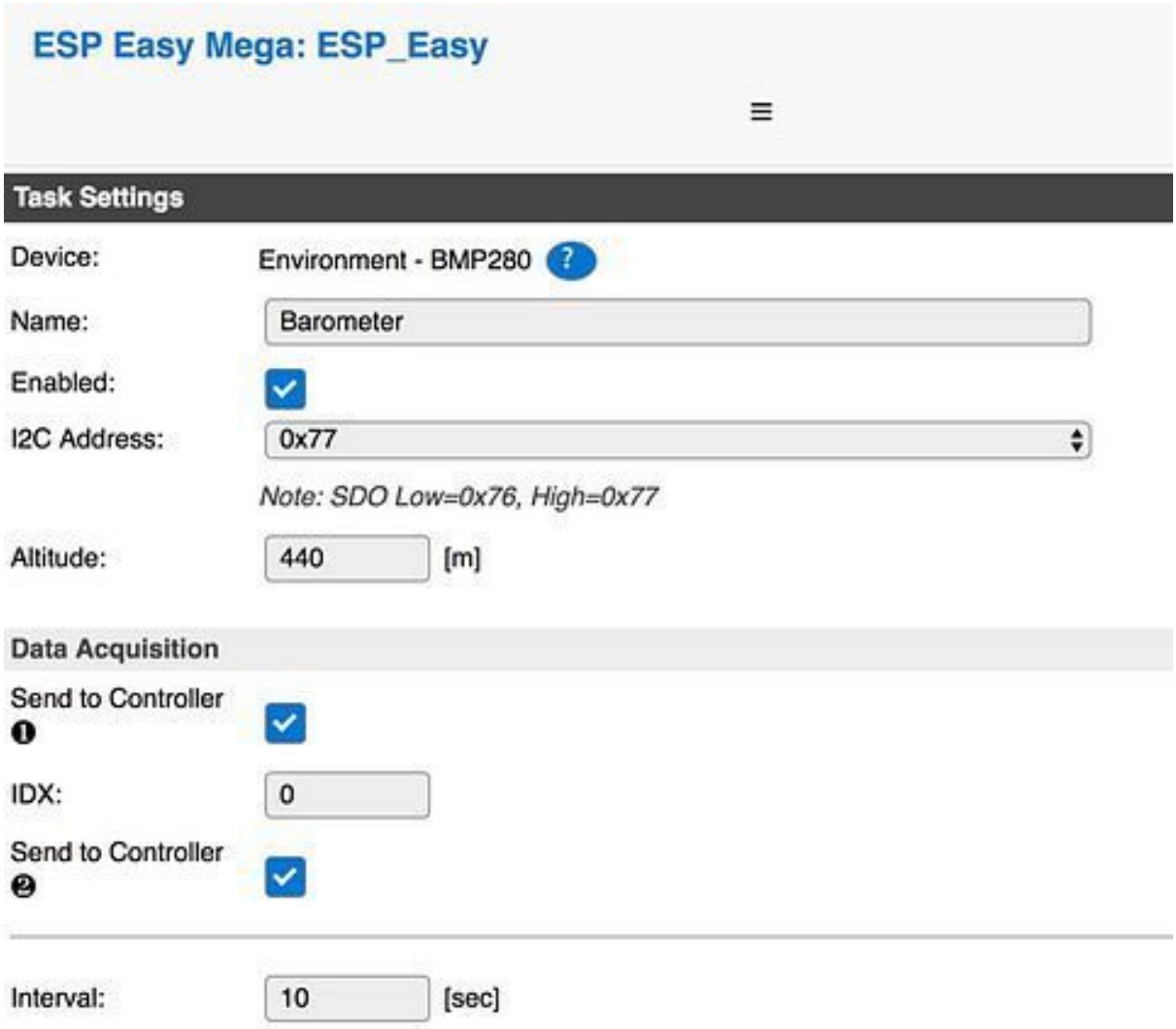



**Abb. 6.9: ESPEasy I2C Konfiguration**

Auf der Seite "Devices" erstellen Sie ein neues Device und suchen den Eintrag für BMP280 in der Liste:




)

Geben Sie die korrekten Daten ein, wie hier gezeigt.

Wenn alles stimmt, werden Sie sofort von den ersten Messwerten belohnt:

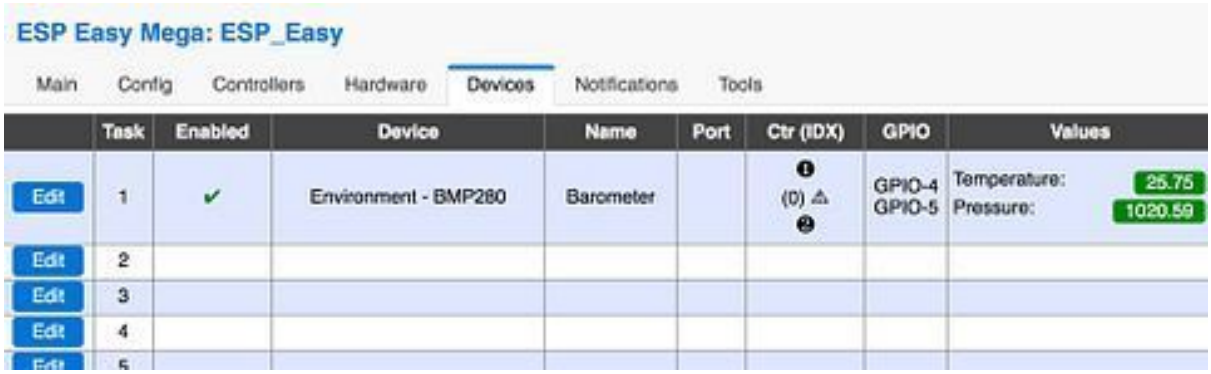



**Abb. 6.11: ESPEasy BMP280**

Wenn das der Fall ist, wollen wir nun diese Messwerte an unseren ioBroker übertragen. Installieren Sie zunächst auf der Adapter-Seite http://homeview.local:8081/#adapters den “MQTT Broker/Client”. MQTT hatte ich im Theorieteil schon kurz beschrieben. Als “Typ” bei der

Konfiguration geben Sie bitte ein “Server/Broker”, Port belassen Sie auf 1883. Den Rest lassen Sie am besten auf den Vorgaben. Nach dem Speichern wird der MQTT Broker starten, aber “gelb” bleiben. Das ist eine etwas unglückliche Art, anzuzeigen, dass er noch keine Clients hat. Lassen Sie sich davon also nicht irritieren. Gehen Sie stattdessen wieder auf die Kontroll-Seite des ESPEasy, und wählen Sie den Reiter “Controllers”. Klicken Sie bei 1 auf “Edit”, dort als Protokoll: OpenHAB MQTT. Die restlichen Angaben wie hier gezeigt:

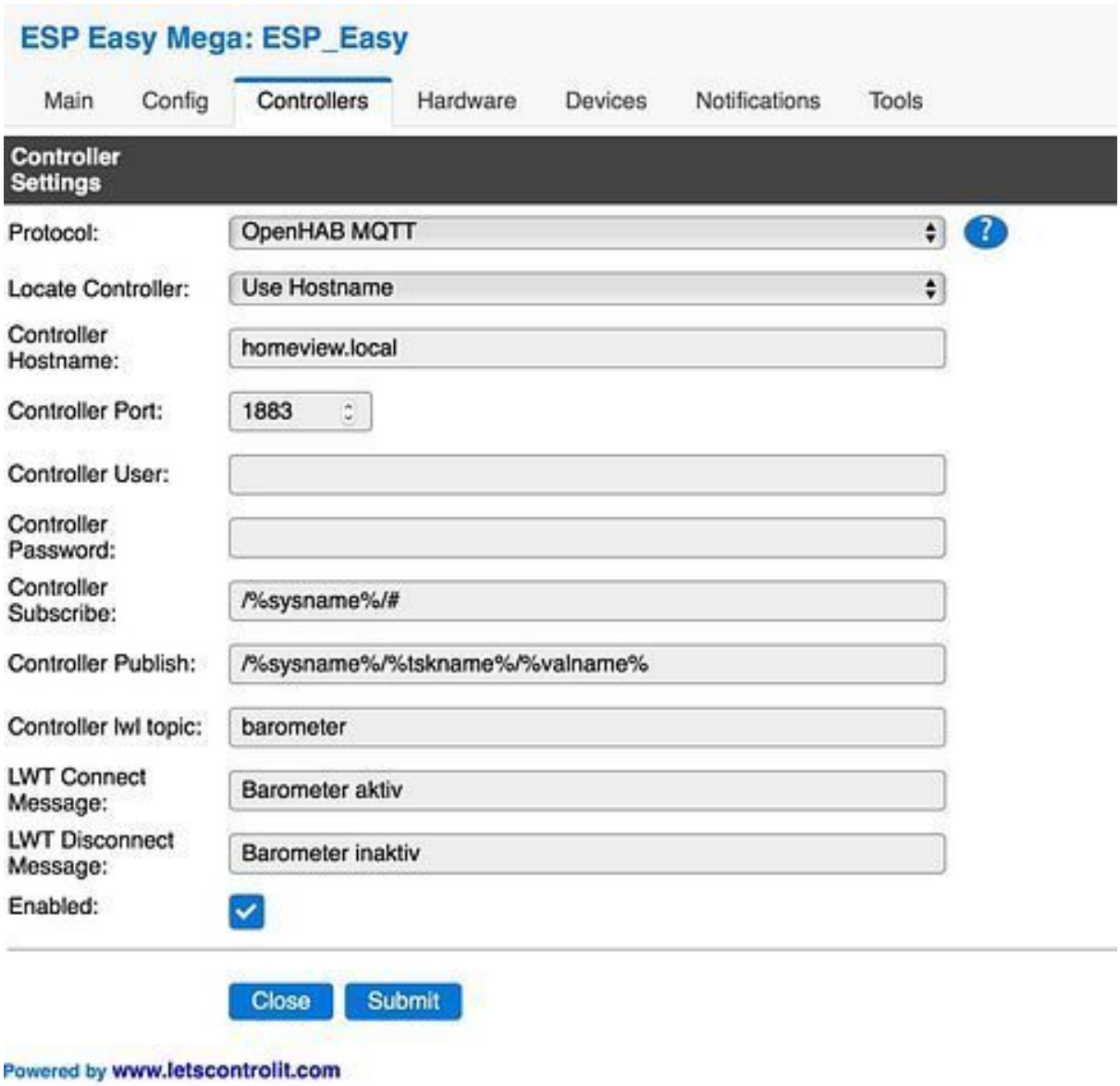



**Abb. 6.12: ESPEasy MQTT Konfiguration**

Beenden Sie mit “Submit” und schauen Sie wieder auf die ioBroker Seite. Unter “Instanzen” sollte mqtt.0 jetzt “grün” geworden sein oder bald werden. Wenn Sie unter “Objekte” den mqtt.0 Eintrag öffnen, sollten Sie jetzt ungefähr das hier sehen:

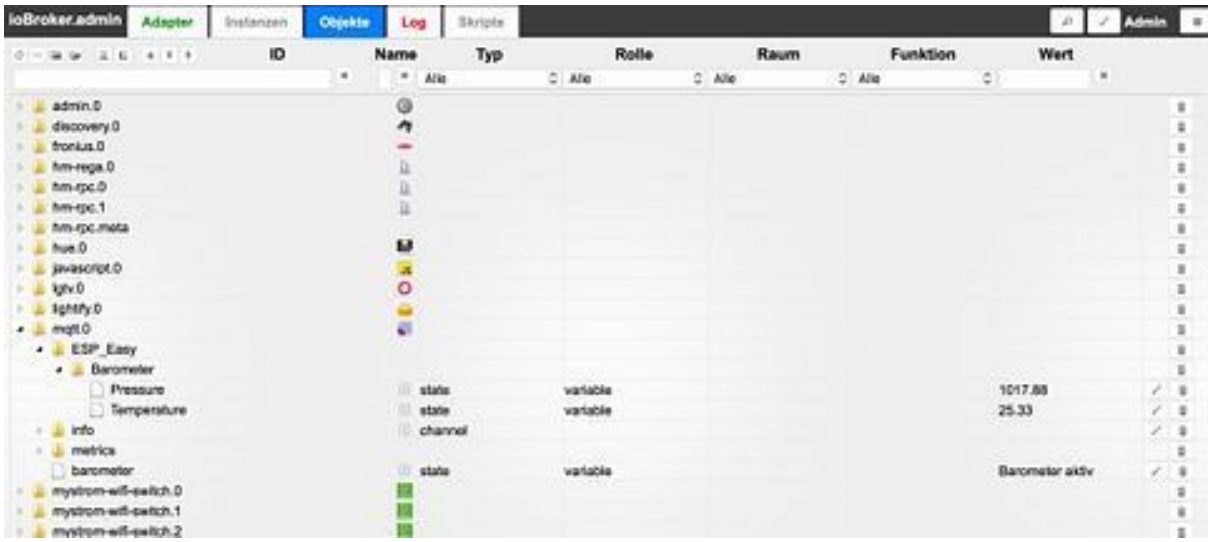



**Abb. 6.13: ESPEasy MQTT States**

Dieses mqtt Objekt in ioBroker können Sie genau so auslesen, wie alle anderen Objekte auch. Sie können auch irgendeinen MQTT Client, zum Beispiel MQTT Dash auf Ihrem Android Handy, oder Mqtt Buddy auf dem iPad oder iPhone installieren, das entsprechende Topic abonnieren und die Werte sehen. (Sofern das Smartphone im selben Netz wie der ioBroker eingebucht ist, natürlich).

Fazit: Wir haben mit minimalem Aufwand ein "intelligentes" Barometer an unsere Heimautomation angeschlossen. Dies war so einfach, weil ESPEasy von Haus aus Unterstützung sowohl für den BMP280, als auch für MQTT mitbringt. Können wir ähnlich einfach die Daten auch nach ioBroker bringen, wenn der Sensor direkt am Raspberry angeschlossen ist?

Ja, können wir. Allerdings ist ein wenig mehr "Handarbeit" nötig, daher wollte ich mit obigem Beispiel zunächst das Prinzip zeigen.

## Anbindung Raspberry mit MQTT an ioBroker

Kehren wir also wieder zurück zu unserem Johnny-five Programm im Raspberry Pi. Um die Messwerte des BMP280 via MQTT zu veröffentlichen, brauchen wir eine MQTT Client library. Davon gibt es (natürlich) einige auf npmjs.org. Wir wählen die MQTT client library von Matteo Collina. Zunächst müssen wir die Library in unser Projekt einbinden. Gehen Sie ins Verzeichnis des Johnny-Five Programs mit dem BMP280 Adapter und geben Sie ein[4]:

```
npm install --save mqtt
```

Die Verwendung im Programm selbst ist dann geradezu trivial:

[4]Wenn Sie wollen, können Sie das Programm natürlich auch auf Ihrem Entwicklungscomputer schreiben, und am Ende mit `scp bmp280.js pi@homeview.local:/home/pi/homeview/` auf den Raspi kopieren.

```
const five=require('johnny-five')
const Raspi=require('raspi-io');

// MQTT einbinden und mit dem MQTT Broker auf dem Raspberry verbinden.
const mqtt=require('mqtt').connect('mqtt://homeview.local')
const MQTT_TOPIC="/Wetter/Wohnzimmer/"

const board=new five.Board({
        io: new Raspi(),
  repl: false
})

board.on('ready',()=>{
        const bmp280=new five.Multi({
          controller: 'BME280',
        })
        bmp280.on('data',()=>{
                let temperature=bmp280.thermometer.celsius.toFixed(1);
                let pressure=bmp280.barometer.pressure.toFixed(1);
    // Statt Loggen auf die Konsole schicken wir die Resultate nun zum MQTT Broke\
r.
    mqtt.publish(MQTT_TOPIC+"Temperatur",temperature)
    mqtt.publish(MQTT_TOPIC+"Luftdruck",pressure*10)
        })
})
```

Da MQTT keine bestimmte Form der Nachrichten vorschreibt, hätten wir auch Temperatur und Druck in eine einzelne Nachricht packen und versenden können. Da wir die REPL nicht mehr brauchen, initialisieren wir das Board mit `repl: false`.

Kaum zu glauben, aber das ist (fast) alles! Sie können das Programm im "homeview" Verzeichnis auf dem Raspberry mit `sudo node bmp280.js` laufen lassen, und werden die Messwerte im ioBroker oder irgendeinem anderen MQTT Client sehen können.

Okay, aber wie bringen wir unseren Raspi nun dazu, das Barometer zusammen mit dem Homeview-Server laufen zu lassen? Einen separaten node-Prozess für dieses Prográmmchen zu starten, wäre ja overkill.

Glücklicherweise ist das sehr einfach:

Editieren Sie app.js (im homeview-Verzeichnis auf dem Raspi) mit `nano app.js`. Geben Sie ganz am Anfang ein: `require('./bmp280')` und verlassen Sie den Editor wieder mit CTRL-O,EINGABETASTE,CTRL-X.

Starten Sie dann den Server neu mit `sudo service homeview restart`.

Jetzt haben Sie das Barometer so eingebunden, dass es gleichzeitig mit dem Homeview-Server startet. In der ioBroker-Admin-Oberfläche (http://homeview.local:8081) finden Sie im Reiter "Objekte" nun einträge für "mqtt.0.Wetter.Wohnzimmer.Temperatur" und "mqtt.0.Wetter.Wohnzimmer.Luftdruck
Falls Sie nicht auf Meereshöhe leben, werden Sie allerdings höchstwahrscheinlich feststellen,

dass der Luftdruck nicht stimmt. Das liegt nicht daran. dass der BMP280 kaputt wäre, sondern daran, dass er den tatsächlichen Luftdruck liefert. Wir sind vom Wetterbericht aber gewöhnt, den normalisierten, also auf Meereshöhe umgerechneten, Luftdruck angezeigt zu bekommen.

Man kann den normalisierten Luftdruck vereinfacht so errechnen:

```
korrigiert = gemessen / ((1-hoehe_ueber_meer)/44330)^5.255)
```

Oder, in JavaScript formuliert:

```
korrigiert = gemessen / Math.pow(1 - hoehe_ueber_meer/44330, 5.255)
```

Falls Sie noch genauere Werte benötigen, können Sie auf die Dokumentation des Herstellers zum Kalibrieren und Auslesen des Sensors zurückgreifen. Bosch stellt auch einen einen Treiber[5] zur Verfügung, der alle Möglichkeiten des Chips zeigt und nutzt.

Eine einfacher verständliche Erläuterung sehen Sie hier: http://www.netzmafia.de/skripten/hardware/RasPi/Projekt-BMP280/index.html

Die endgültige Version des Programms sieht dann so aus:

```
const five=require('johnny-five')
const Raspi=require('raspi-io');
const altitude = 440; // unsere Höhe in MüM.

// MQTT einbinden und mit dem MQTT Broker auf dem Raspberry verbinden.
const mqtt=require('mqtt').connect('mqtt://localhost',{
  'clientId': 'wohnzimmer-barometer',
  'will': {
    'topic': 'info/connection/barometer',
    'payload': 'getrennt'

  }
})

mqtt.on("connect",()=>{
  mqtt.publish("info/connection/barometer","verbunden");
});

const MQTT_TOPIC="/Wetter/Wohnzimmer/"

const board=new five.Board({
  io: new Raspi(),
  repl: false
})
```

[5] https://github.com/BoschSensortec/BMP280_driver

```
// Korrektur auf aktuelle Höhe über Meeresspiegel, in mbar
const corr=(raw)=>{
  const corrected=raw/Math.pow((1-altitude/44330),5.255)
  return (10*corrected).toFixed(1)
}

board.on('ready',()=>{
        const bmp280=new five.Multi({
    controller: 'BME280',
    freq: 30000
        })
        bmp280.on('data',()=>{
                let temperature=bmp280.thermometer.celsius.toFixed(1);
                let pressure=corr(bmp280.barometer.pressure);
          mqtt.publish(MQTT_TOPIC+"Temperatur",temperature)
    mqtt.publish(MQTT_TOPIC+"Luftdruck",pressure)
        })
})
```

Nicht viel Neues: Bei der Initialisierung haben wir mit dem Attribut “will” ein “Testament” übergeben, also eine letzte Nachricht, die der Broker bei einem Verbindungsabbruch noch übermitteln soll. In `corr()` rechnen wir lokale Messwerte in kPa in normalisierte Werte in mbar um. Wir fragen alle 30 Sekunden die Werte ab und publizieren diese via MQTT.

Jetzt haben wir alles zusammen, um auch den Luftdruck in unserer WebApp anzuzeigen. Die nötigen Ergänzungen sollten Sie nun nicht mehr vor unlösbare Probleme stellen:

```
// In aurelia_project/environments/dev.ts und aurelia_project/environments/prod.t\
s:
  // ...
devices:{
  barometer: "mqtt.0.Wetter.Wohnzimmer.Luftdruck",
  aussen_temp: "hm-rpc.0.OEQ0088064.1.TEMPERATURE",
  // ...
}

// in config.ts:
// ...
barometer_cfg:{
    devices: env.devices.barometer,
    size: gauge_size,
    min:970,
    max:1045,
    message: "luftdruck",
    bands: [{"from":970,"to":995,color: "#42d4f4"},
      {from:995,to:1020,color:"green"},
      {from:1020,to:1045,color:"#42d4f4"}]
```

```
  }
// ...

// in app.ts:
// ...
private gauges = [configs.wohnzimmertemp_cfg, configs.aussentemp_cfg,
  configs.bad_oben_cfg,configs.dusche_cfg, configs.dachstock_cfg,
  configs.barometer_cfg,configs.powermeter_cfg]
// ...
```

Und in app.html:

```
<!-- ... -->
<div class="gauge">
    <circular-gauge cfg.bind="conf.barometer_cfg"></circular-gauge>
</div>
<!-- ... -->
```

Danach sollte es ungefähr so aussehen:

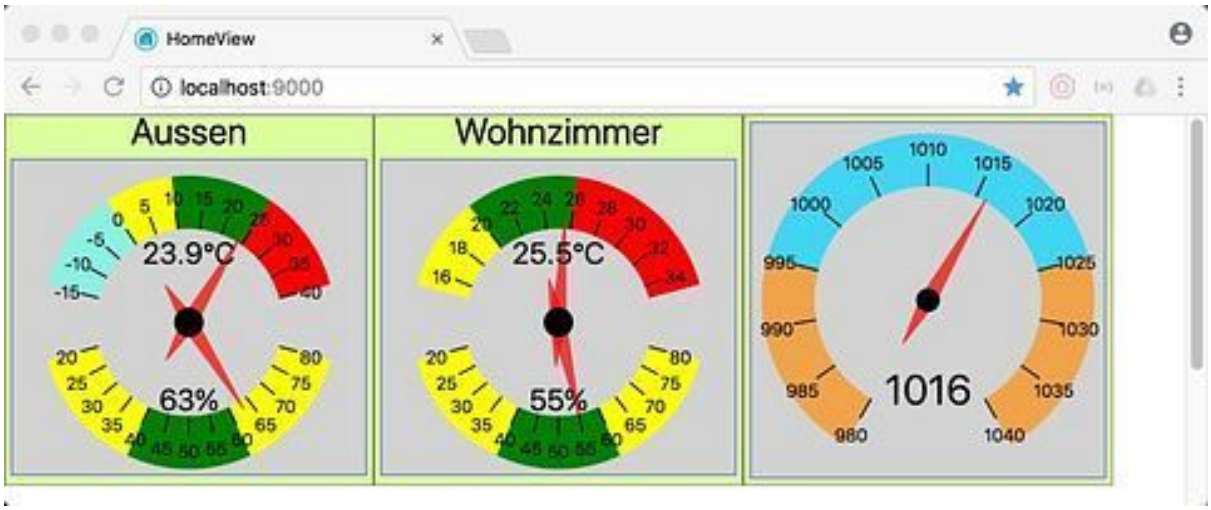



**Abb. 6.14: Barometer Anzeige**

# Motor

Natürlich kann man nicht nur Sensoren auslesen, sondern wir können mit IoT Geräten auch etwas *tun*. Zur Demonstration hier eine Schrittmotorsteuerung. Wir werden etwas Ähnliches später zur Regulierung des Boilers benötigen. Hier aber erst mal nur das Grundprinzip.

Zunächst der fliegend verdrahtete Versuchsaufbau:

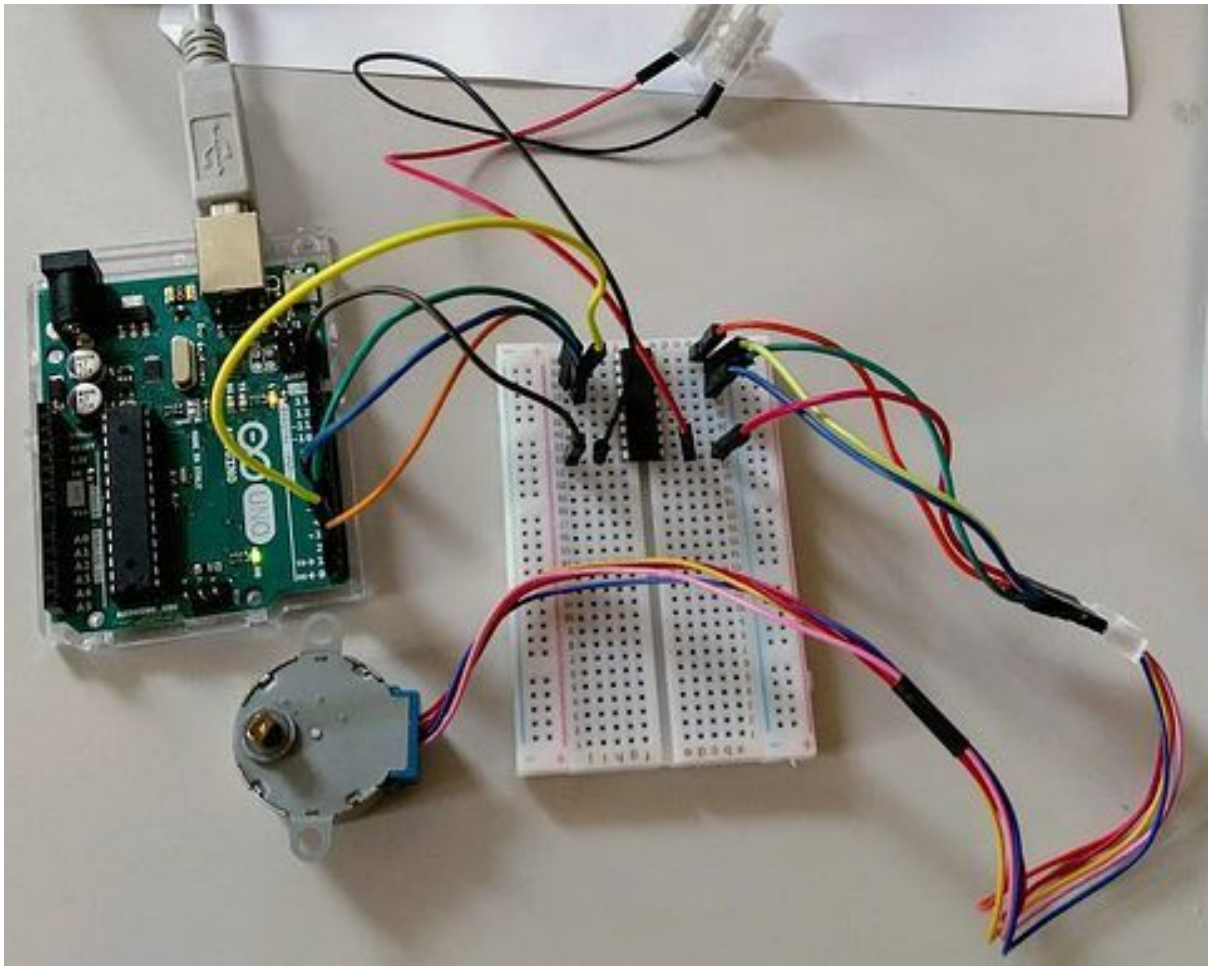


**Abb. 6.15: Schrittmotor fliegend verdrahtet**

Für erste Experimente verwende ich immer den Arduino, weil er schnell und einfach aufzusetzen ist, und weil er Fehler eher verzeiht, als Raspberry oder ESP8266. Wenn der Aufbau auf dem Arduino funktioniert, kann man ihn leicht auf eines der anderen Geräte portieren.

Oben rechts im Bild sehen Sie zwei Drähte aus einer Lüsterklemme kommen, das ist die externe Stromversorgung für den Motor. Der Arduino (und erst recht der Raspberry) ist zu schwach, um den Motor direkt anzutreiben. Auf dem Breadboard sitzt ein ULN2083 IC, welches folgende Anchlussbelegung hat:

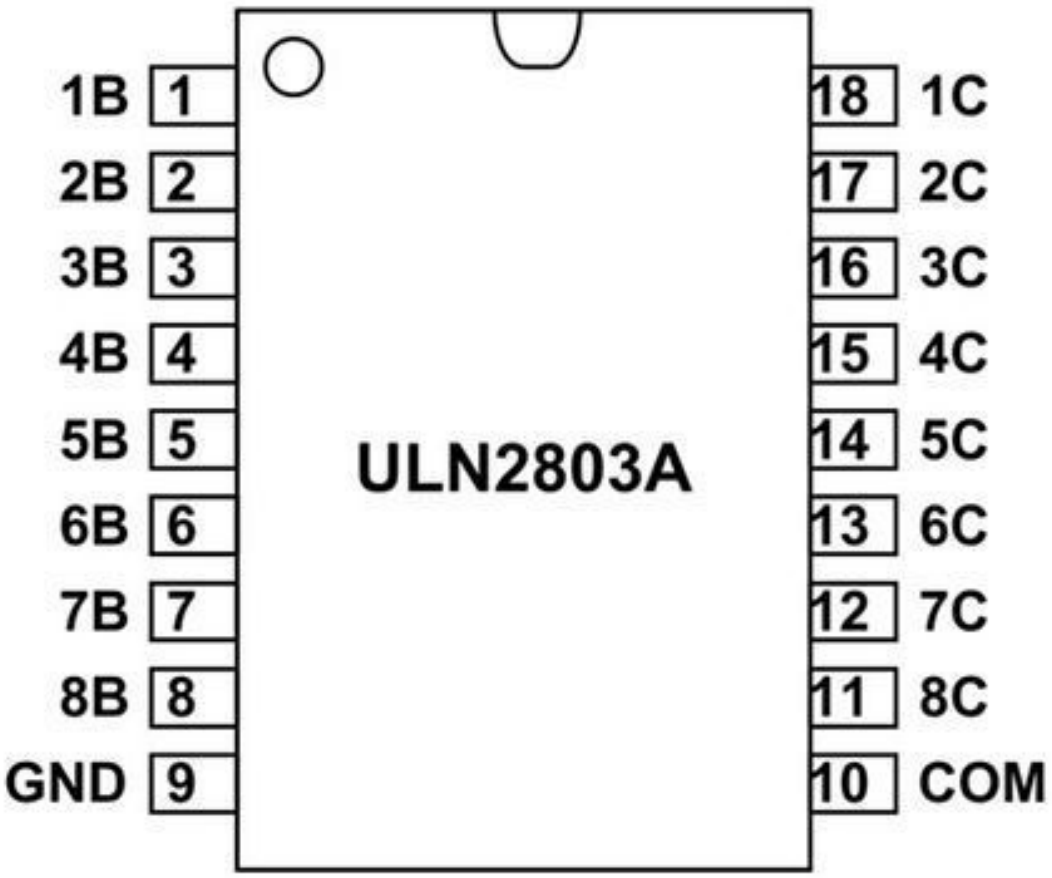



**Abb. 6.16: ULN 2803 Verstärker IC**

Wie es innerlich aussieht, braucht uns nicht zu interessieren (es ist ein 8-fach Darlington Array, falls Sie es doch wissen wollen). Wichtiger ist, was es tut: Es macht einen schwachen Eingangsstrom zu einem viel stärkeren Ausgangsstrom. Und das an 8 unabhängigen Leitungen. Die Beschaltung ist sehr einfach: Der Eingang 1B wird auf den Ausgang 1C umgesetzt und so weiter. bei GND steckt man den gemeinsamen Minuspol ein (und der muss auch zum Arduino geleitet werden), bei COM kommt der Pluspol für die Leistungsseite, das kann je nach Motor eine Spannung von 3 bis 20 Volt sein und sollte die vom Motor benötigte Leistung liefern können, also mindestens 500mA. Man kann dieses IC durchaus nicht nur für Motorsteuerungen verwenden, sondern immer dann, wenn man einen Stromverstärker braucht. Allerdings muss man beachten, dass die Darlington Schaltung bei positivem Eingangssignal eine Emitter-Schaltung öffnet. Oder andersherum ausgedrückt: Wenn man "1" an den Eingang anlegt, dann kann man den Ausgang als Minuspol des geschalteten Stroms benutzen, nicht als Pluspol. Das passt hier glücklicherweise ganz gut, wie Sie gleich sehen werden.

Als Motor verwenden wir einen der meist gebauten (und darum billigen) Schrittmotoren überhaupt, den 28BYJ-48. Den gibt es in 5 Volt und 12 Volt Ausführungen. Ich habe hier 5 Volt genommen. Dieser Motor ist in einer Vielzahl von Geräten verbaut. Vielleicht müssen Sie ihn gar nicht kaufen, sondern können ihn aus einem alten Drucker, Scanner, CD-Laufwerk, Klimaanlage, Lüfter etc. ausbauen. Wenn Sie ihn kaufen, wird er auch kein allzu tiefes Loch in die Haushaltskasse reißen, mit 5-10 Euro sind Sie dabei. Das Verstärker IC wird noch rund

20-40 Cent zusätzlich kosten.

Der 28BYJ-48 ist ein sogenannter unipolarer Schrittmotor. Das bedeutet, dass der Strom immer in dieselbe Richtung fließt. Die Drehrichtung legt man nicht durch Umpolen, sondern durch geeignete Reihenfolge der Aktivierung der Spulen fest. Daher braucht man auch keine spezielle Motorsteuerung (H-Bridge), sondern ein simples Universal-Verstärker-IC genügt.

Ein Schrittmotor unterscheidet sich von einen "gewöhnlichen" Elektromotor dadurch, dass er nicht einfach ein- oder ausgeschaltet wird, sondern dass man ihn schrittweise (sic!) um einen bestimmten Betrag drehen kann. Das erreicht man, indem man die Spulen in geeigneter Weise und Reihenfolge mit Strom versorgt. Der 28BYJ-48 hat 5 Drähte. Einer davon (idR der Rote) ist der gemeinsame Pluspol, der ins Zentrum jeder der beiden Spulen geht, die anderen 4 sind Minuspole für jedes Ende der Spulen. Sie sehen, das passt wunderbar zu unserem Darlington-IC, das diese Minuspole durchschalten oder sperren kann.

Bewaffnet mit diesem Wissen können wir den Motor nun verschalten: Der rote Draht kommt zum Pluspol der externen Stromversorgung, die auch zu Pin 10 des IC geht.

Den orangen Draht verbinden wir mit Pin 18 des IC, und dessen Steuerleitung auf Pin 1 des IC geht zu GPIO 3 des Arduino. In analoger Weise verbinden wir den gelben Draht mit GPIO 4, den pinken mit GPIO 5 und den blauen mit GPIO 5. Zu guter Letzt verbinden wir noch einen der GND Pins des Arduino mit Pin 9 des IC, den wir auch mit dem Minuspol der externen Stromversorgung verbinden. Fertig.

Nun zum Programm für den Arduino. Wählen Sie in der Arduino IDE den richtigen Chip (bei mir Arduino/Genuino Uno) und den richtigen Port aus, und geben Sie das Programm ein:

```
/**
 * Steuerung eines 28BYJ-48 Stepper Motors mit einem 2803APG Darlington Array.
 * Die rote Leitung ist gemeinsamer Pluspol und kommt an (externe) 5V Leitung.
 * Die anderen Pins sind nach der Farbe der Anschlüsse des Motors benannt.
 */
#define orange 2
#define yellow 3
#define pink 4
#define blue 5

// Wieviele Schritte für eine ganze Umdrehung (Nicht bei allen Modellen gleich)
const int steps360=512;
// Dauer der Pause zwischen zwei Schritten in Mikrosekunden. Nicht zu klein wähle\
n.
const int pause = 10000;

// Reihenfolge der Spulen. Wir wählen hier das 8-Schritt Verfahren.
String steps[] = {"0111", "0011", "1011", "1001", "1101", "1100", "1110", "0110"}\
;

void setup() {
  pinMode(orange, OUTPUT);
```

```
  pinMode(yellow, OUTPUT);
  pinMode(pink, OUTPUT);
  pinMode(blue, OUTPUT);
}

/**
 *  Drehung im Gegenuhrzeigersinn (Counterclockwise) um "count" Schritte.
 */
void ccw(int count) {
  for (int i = 0; i < count; i++) {
    for (int j = 0; j < 8; j++) {
      setPins(steps[j]);
      delayMicroseconds(pause);
    }
  }
}

/**
 *  Drehung im Uhrzeigersinn (clockwise) um "count" Schritte.
 */
void cw(int count) {
  for (int i = 0; i < count; i++) {
    for (int j = 7; j >= 0; j--) {
      setPins(steps[j]);
      delayMicroseconds(pause);
    }
  }
}

void setPins(String step) {
  digitalWrite(orange, step[0] == '1' ? HIGH : LOW);
  digitalWrite(yellow, step[1] == '1' ? HIGH : LOW);
  digitalWrite(pink, step[2] == '1' ? HIGH : LOW);
  digitalWrite(blue, step[3] == '1' ? HIGH : LOW);

}

void loop() {
  cw(steps360);
  delay(100);
  ccw(steps360);
  delay(1000);
}
```

Das Programm sollte selbsterklärend sein. Die Pause (`delayMicroseconds()`) dient dazu, der Achse des Motors Zeit zu geben, in die eingestellte Position zu kommen. Wenn diese Zeit zu klein

gewählt ist, wird der Motor einzelne Schritte "verlieren" oder sich gar nicht mehr bewegen. Je weniger Strom die Spannungsquelle liefert, desto länger muss diese Zeit sein. Für ein Standard-USB Netzteil sind Werte um die 1000 Mikrosekunden okay. Es spricht aber natürlich nichts dagegen, den Motor z.B. mit 20000 langsamer und dafür sicherer zu bewegen.

Dieses Programm macht für jeden Schritt 8 Teilschritte. Das Array steps[] gibt für jeden Teilschritt an, durch welche Spulen Strom fliesst ("1"), und welche nicht ("0"). Etwas weiter hinten werde ich noch ein Verfahren mit 4 Schritten vorstellen.

Es scheint verschiedene Getriebevarianten bei diesem Motor zu geben; jedenfalls ist die Zahl der Schritte für eine Umdrehung nicht bei jedem gleich. Bei meinem sind es 512(*8) Schritte, was in der Konstanten steps360 festgehalten wird.

Wenn Sie dieses Programm hochladen und die Stromversorgung anschliessen, sollte der Motor immer eine Umdrehung im Uhrzeigersinn und dann wieder eine Umdrehung im Gegenuhrzeigersinn machen (von der Achse aus gesehen).

Wenn das klappt, können Sie als Fingerübung den Motor an die NodeMCU anschliessen.

Wir machen jetzt aber weiter mit einer mechanischen Steuerung für den Boiler:

# Elektroboiler mechanisch steuern

Es gibt eine eiserne Grundregel für Hobby-Maker: Finger weg vom Stromnetz. Wenn wir am Raspberry ein Kabel falsch stecken, kostet es schlimmstenfalls den Raspberry. Wenn wir an der Hauselektrik etwas falsch machen, gibt es schlimmstenfalls ein Feuer oder einen lebensgefährlichen Stromschlag. Arbeiten an Spannungen von mehr als 50 Volt bei eine Stärke von mehr als 500mA sind dem Elektriker vorbehalten.

Aber wir wollen trotzdem den Elektroboiler steuern. Warmwasser ist ein hervorragender und vergleichsweise billiger Energiespeicher. Überschlägig gilt folgende Rechnung: Um ein Gramm Wasser um ein Grad aufzuheizen, braucht man 1 Kalorie. Oder andersherum ausgedrückt: Warmwasser enthält pro Liter und Grad eine Kilokalorie an Energie. Wenn wir unseren 400-Liter-Boiler mit Sonnenstrom von 50 auf 70 Grad aufheizen, dann stecken wir 8000 Kilokalorien hinein, das sind fast 10 kWh. Diese 10 kWh sparen wir an Netzstrom, wenn wir Heisswasser entnehmen, bis der Boiler wieder abgekühlt ist. Und diese gespeicherte Energie steht auch nachts zur Verfügung, wenn die Sonne nicht scheint, Ganz ohne teuren Akku.

Wir wollen also den Boiler je nach Sonnenkraft unterschiedlich weit aufheizen. In einer realen Anwendung sollten Sie aber darauf achten, dass er mindestens einmal pro Woche mindestens 65°C erreicht, um Legionellen abzutöten, die sonst sehr ernste Erkrankungen auslösen könnten! Und natürlich sollten Sie ihn nicht zu nah an 100° bringen, um keinen Wärmeverlust oder gar Gefahr durch Überdruck zu provozieren. Last but not least gilt auch: Je höher die Wassertemperatur, desto aggressiver die Korrosion. Am besten sehen Sie zu, dass die Temperatur sich zwischen 45 und 75°C bewegt. Natürlich kann man eine solche Steuerung automatisieren. Wir haben längst das Rüstzeug, um das zu programmieren.

Schwieriger ist die Hardware-Seite: Wenn wir die Stromanschlüsse nicht berühren dürfen, wie können wir den Boiler dann regeln?

- Wir können einen modernen Boiler kaufen, der von Haus aus Smarthome-fähig ist. Ich finde es aber schade, einen funktionierenden Boiler zu entsorgen. Zumal, seien wir ehrlich, die Investition sich erst nach mehreren Jahren rentieren würde. Strom ist einfach noch zu billig.
- Wir können einen Elektriker beauftragen, uns ein fernsteuerbares Relais in die Stromzuleitung zum Boiler einzubauen. Hier gilt Ähnliches: Ein Elektriker müsste herkommen und die Installation machen. Das würde leicht mehrere hundert Euro kosten.
- Wir können den Boiler pseudo-manuell schalten. Also quasi eine Hand simulieren. Spätestens seit der Useless box[6] ist das Bedienen von Schaltern durch Robot-Bauteile ins allgemeine Bewusstsein gelangt. Und da dies ein Buch übers Selbermachen ist, wählen wir natürlich diesen Weg.

Der Boiler hat einen Knopf zum Temperaturregeln:

[6] https://www.youtube.com/watch?v=aqAUmgE3WyM

**Abb.6.17: Originalknopf**

Wenn man den abzieht, kommt die Achse eines Potentiometers zum Vorschein:

**Abb. 6.18: Potentiometer-Achse**

Dort können wir eine mit dem 3D-Drucker auf Mass angefertigte Halterung anbringen:

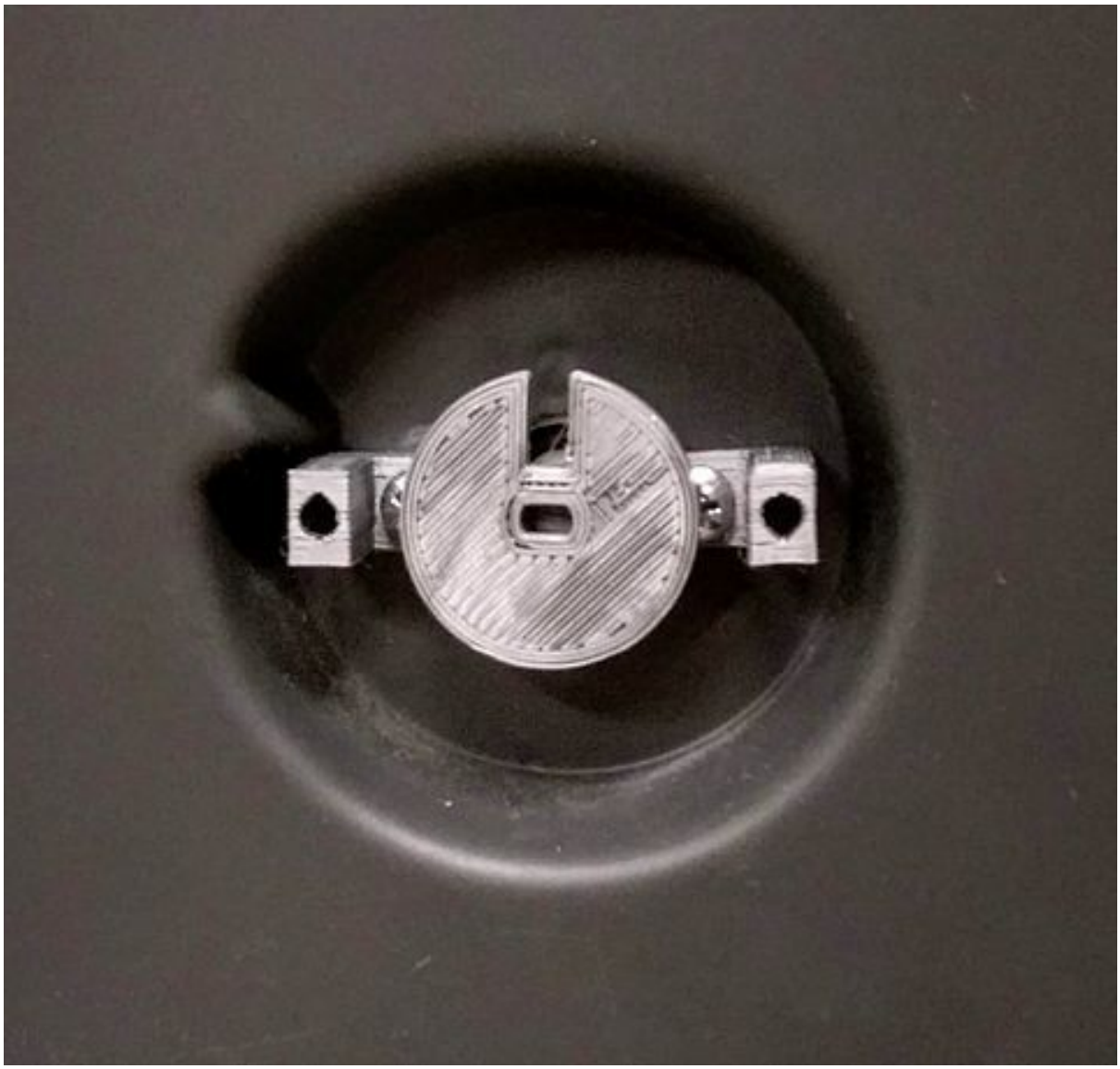


**Abb. 6.19: Motor-Halterung**

Und an dieser wiederum unseren Schrittmotor aus dem letzten Kapitel montieren.


**Abb. 6.20: Motor montiert**

Auf diese Weise berühren wir keine stromführenden Teile und sind stets auf der sicheren Seite.

Der Rest ist Software. Die eigentliche Steuerung haben wir ja schon im vorherigen Kapitel abgehandelt. Die Frage, auf welchem Gerät die Steuersoftware laufen soll, ist eigentlich nur eine Frage der örtlichen Begebenheiten und der persönlichen Präferenzen. Bei uns sollte die Anbindung per WLAN erfolgen, daher fiel der Arduino weg. Der ESP8266 hatte vom Keller aus Probleme, sich mit dem WLAN zu verbinden, daher entschied ich mich für einen Pi Zero W als Boilersteuerung. Damit kehren wir zurück zu Johnny-Five.

```
/**********************************************************************
 * Johnny Five driver for unipolar stepper motor (e.g. 28BYJ-48)
 **********************************************************************/
const five = require('johnny-five')
// Dismal nur 4 Schritte
const seq = ["0011", "0110", "1100", "1001"]

/**
 * Constructor arguments: 4 Pins for motor coils, delay in milliseconds to wait a\
```

```
fter
 * each step (allow the motor to reach new position).
 */
class Stepper {

  constructor(orange, yellow, pink, blue, motor_power, delay) {
    this.in1 = new five.Pin(orange, { mode: five.Pin.OUTPUT });
    this.in2 = new five.Pin(yellow, { mode: five.Pin.OUTPUT });
    this.in3 = new five.Pin(pink, { mode: five.Pin.OUTPUT });
    this.in4 = new five.Pin(blue, { mode: five.Pin.OUTPUT });

    this.speed = delay;
    this.coils=[this.in1,this.in2,this.in3,this.in4]
  }

  set_step(step, dir) {
    let lcoils=dir ? this.coils : this.coils.reverse()
    for(let i=0;i<step.length;i++){
      lcoils[i].write(parseInt(step[i]))
    }
  }

  /**
   * rotate clockwise
   */
  async cw(steps) {
    for (let i = 0; i < steps; i++) {
      for (let j = seq.length - 1; j >= 0; j--) {
        let step = seq[j]
        this.set_step(step,false);
        await this.sleep(this.speed);
      }
    }
  }

  /**
   * rotate counterclockwise
   */
   async ccw(steps) {
    for (let i = 0; i < steps; i++) {
      for (let j = 0; j < seq.length; j++) {
        let step = seq[j]
        this.set_step(step,true);
        await this.sleep(this.speed);
      }
    }
```

```
  }


  sleep(ms) {
    return new Promise(resolve => {
      setTimeout(resolve, ms)
    })
  }
}


exports.Stepper = Stepper
```

Man sieht sofort, dass dieses Node-Modul sehr ähnlich ist, wie das Arduino-Programm im vorherigen Kapitel. Nur habe ich diesmal nur vier Teilschritte pro Step definiert, die dafür etwas weiter sind.

Es wird recht simpel in ein Node-Programm eingebunden:

```
const Raspi = require('raspi-io')
const five = require('johnny-five')
const Stepper=require('./stepper').Stepper
const board = five.Board({
    io: new Raspi()
  }
);
const orange = "GPIO26"
const yellow = "GPIO19"
const pink = "GPIO13"
const blue = "GPIO6"
let speed = 6;

let stepper;

board.on('ready', async () => {
  stepper = new Stepper(orange, yellow, pink, blue, speed)
  await stepper.forward(50);
  await stepper.backward(50);

  board.repl.inject({
   stp:stepper
  })
  function setSpeed(sp) {
    stepper.speed = sp;
  }

})
```

Die Pin-Nummern entsprechen dem Setup in meiner experimentellen Pi-Zero Verbidung:

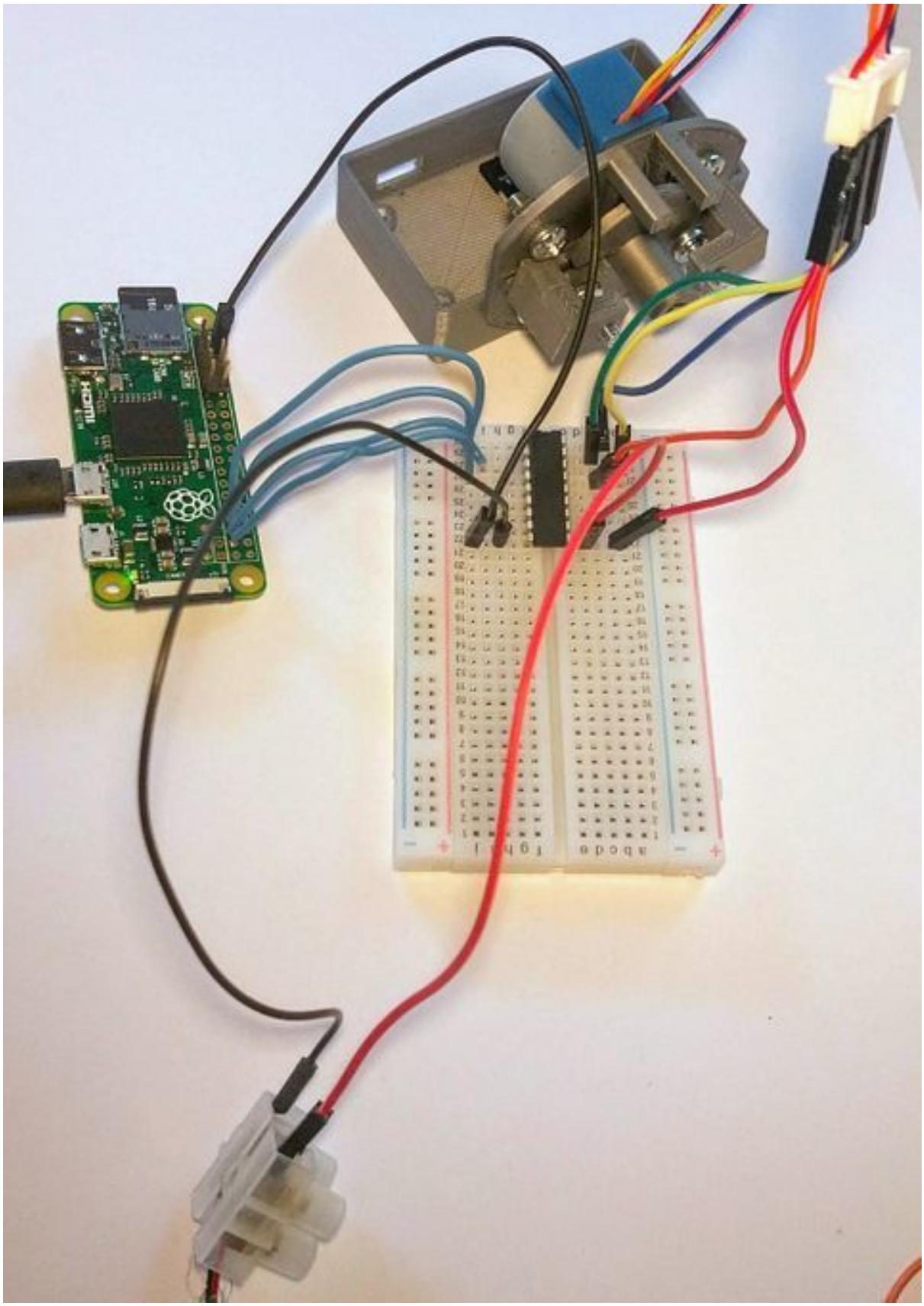


**Abb. 6.21: Frei fliegende Raspi-Motor Verdrahtung**

Nach dem Start mit `sudo node index.js`wird der Motor zuerst 50 Schritte im Uhrzeigersinn, dann 50 Schritte im Gegenuhrzeigersinn drehen, und dann auf Befehle warten.

Die Bedeutung des `board.repl.inject` Ausdrucks ist, den Stepper für interaktive Bedienung in der REPL freizugeben. Sie können nach dem Start des Programms in der Kommandozeile zum Beispiel eingeben: `stp.cw(100)`, um den Motor "manuell" 100 Schritte im Uhrzeigersinn drehen zu lassen. Wenn Sie das nicht mehr benötigen, können Sie den Programmteil einfach auskommentieren und das Board mit `repl: false` initialiseren, wie wir es schon heim Barometer gemacht haben.

Doch wie geben wir dem Raspi Kommandos übers Netz? Wir werden ihn ja nicht immer an der Konsole haben. Da NodeJS ohnehin schon läuft, könnten wir einen REST-Service mit Express aufbauen, um Kommandos entgegenzunehmen und Rückmeldungen zu liefern. Ich habe mich aber für eine einfachere und flexiblere Lösung entschieden: Wir werden den Raspi über MQTT steuern, da wir ohnehin schon eine MQTT Infrastruktur fürs Barometer aufgebaut haben. Wie wir dort gesehen haben, ist MQTT sehr einfach einzurichten, und MQTT wird unseren schwachen Pi Zero auch weniger ans Limit bringen, als ein REST Server.

Spätestens an diesem Punkt müssen wir uns allerdings noch einmal ernsthafte Gedanken über die Absicherung unserer Hausautomation machen. Wenn jemand unerlaubt ein paar Lichter ein- und ausschaltet, ist das eine Sache. Wenn jemand mit dem Boiler herumspielt, ist das schon etwas ganz anderes. Das Heimnetz soll nicht von aussen erreichbar sein; eine Firewall ist Pflicht. Aber auch Gäste oder Freunde des Nachwuchses, die ganz legal ins Gäste-WLAN dürfen, sollen nicht an der Boilersteuerung herumspielen können. Der Zugang zum MQTT Broker muss somit mit einem Passwort gesichert sein, und der Datenverkehr muss verschlüsselt erfolgen. Für beides hat der MQTT-Adapter des ioBroker glücklicherweise schon Vorkehrungen; sie müssen es auf der Einstellungsseite der MQTT Instanz nur noch aktivieren.

Folgende MQTT Topics sollen unterstützt werden:

- Boiler/setTemp - Boiler auf x Grad einstellen
- Boiler/calibrate - Motor Kalibrieren
- Boiler/temp -Temperatur auslesen

## Komponente zur Temperaturregelung

Als erstes implementieren wir eine Komponente, mit der wir manuell von unserer Web-Oberfläche aus die Temperatur einstellen können.

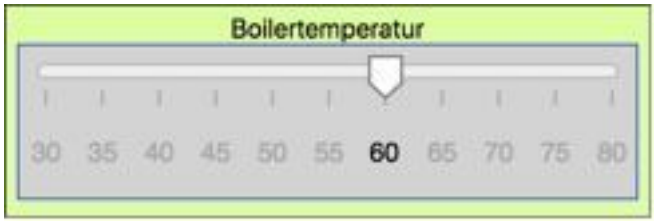



**Abb. 6.22: Boilersteuerungs-Instrument**

Das Programm dazu wird Ihnen inzwischen trivial vorkommen; dank der schon geleisteten Vorarbeiten sind nur recht wenige Codezeilen nötig:

**components/setpoint.ts**

```
import {sliderHorizontal} from 'd3-simple-slider'
import { bindable, noView, autoinject } from 'aurelia-framework'
import {Helper, Component, eaMessage} from './helper'

@autoinject
@noView
export class Setpoint implements Component{
  @bindable cfg
  body: any;
  component_name="Setpoint";
  private slider

  constructor(private hlp:Helper, public element:Element){}

  attached(){
    this.hlp.initialize(this, {
      minValue: 0,
      maxValue: 100,
      step: 1,
      width: 250,
      height: 50,
      displayValue: true,
      message: "setpoint"
    })
  }
  configure() {}
  render() {
    let dim=this.hlp.defaultFrame(this)
    let debounced=Util.debounce((val)=>{          // (1)
      this.hlp.ea.publish(this.cfg.message,val)
    },500,this)

    this.slider=sliderHorizontal()                // (2)
      .min(this.cfg.minValue)
      .max(this.cfg.maxValue)
      .step(this.cfg.step)
      .width(dim.w-2*Helper.BORDER-2*this.cfg.offset)
      .displayValue(this.cfg.displayValue)
      .on('onchange',debounced)                    // (3)

    this.body.append("g")                         // (4)
      .attr("width",dim.w)
      .attr("height",this.cfg.height)
      .attr("transform",`translate(${dim.x+Helper.BORDER+this.cfg.offset},${dim.y\
+Helper.BORDER+this.cfg.offset})`)
```

```
    .call(this.slider)
  }

  update(value:eaMessage){
    this.slider.value(value.data)
  }

}
```

Wir verwenden eine Fertigkomponente (d3-simple-slider von John Walley), die wir in unsere Standard-Komponentenhülle einzubauen. Bei (2) konfigurieren wir den Slider, bei (4) wird er eingebunden. Neu ist vielleicht die Funktion bei (1). Hier lassen wir uns von der Util - Komponente eine Funktion erstellen, die eine "entprellte" Version unserer gewohnten ea.publish-Routine darstellt. Diese Funktion übergeben wir bei (3) dem onChange-Event des Sliders. Die Bedeutung dieses etwas umständlich erscheinenden Vorgehens ist: Wenn der Anwender am Schieber zieht, dann liefert dieser für jede kleine Änderung einen "onchange" Event. Wir würden also in kürzester Zeit von Hunderten von Events überschüttet. Unser "debounce" führt dazu, dass this.hlp.ea.publish erst dann stattfindet, wenn für mindestens 500 Millisekunden kein neuer Event eintraf. Dann wird der letzte Wert (und nur dieser) weitergeleitet.

Der dazugehörige Eintrag in config.ts ist:

**config.ts**

```
"boiler_cfg":{
    "minValue": 30,
    "maxValue": 80,
    "step": 5,
    "width": gauge_size,
    "height": switch_size ,
    "caption": "Boilertemperatur",
    "offset": 5
  }
```

und in app.html:

**app.html**

```
<require from="components/setpoint"></require>
<!-- ... -->
<div class="gauge">
  <setpoint cfg.bind="conf.boiler_cfg"></setpoint>
</div>
```

Das genügt, um den Slider anzuzeigen und bedienbar zu machen. Allerdings tut er noch nichts. Wie üblich haben wir die Aurelia-Komponente so allgemein formuliert, dass sie keine spezifische Aufgabe übernehmen kann. Dadurch können wir diesen Slider bei Befarf auch für andere

Aufgaben ensetzen. Den Boiler-spezifischen Code könnten wir wie bei den Thermometern und den Schaltern in app.ts unterbringen, allerdings ist diese Klasse mittlerweile recht groß und damit unübersichtlich geworden. Jede Klasse sollte eine einzige Aufgabe erledigen, und diese Regel haben wir hier verletzt - was in der Experimentierphase verzeihlich ist. Aber die endgültige Anwendung sollte robuster und leichter wartbar sein. Es kann gut sein, dass man nach einem Jahr etwas verändern muss, und glauben Sie mir: Nach einem Jahr werden Sie sich in Spaghetticode nicht mehr zurechtfinden.

Zeit für ein Refactoring.

Sie erhalten diesen Stand des Projekts mit

```
git checkout -f origin/teil_20
git clean -f
npm install
```

Unser Projekt geht in eine neue Versionsstufe, das sollten wir auch an der Versionsnummer deutlich machen. Geben Sie ein:

```
npm version minor
```

Damit setzt npm die Versionsnummer von 0.1.0 auf 0.2.0. `npm version major` würde die erste Ziffer hochzählen und `npm version rev` die letzte.

Beim Aufbau des bisherigen Codes haben wir ja gesehen, dass es drei verschiedene Typen von Instrumenten gibt:

- “Gauges”, die Werte anzeigen können
- “Switches”, die Zustände anzeigen und modifizieren können
- “Charts”, die Charts anzeigen können.

Der Plan ist nun, die Steuerung jedes Typs in eine eigene Klasse auszulagern und in app.js nur noch diese Klassen einzubinden.

Erstellen Sie also ein neues Unterverzeichnis *devices* in *src*.

Dort beginnen wir mit der Steuerklasse für die Gauges:

**devices/gauges.ts**

```
import { autoinject } from "aurelia-framework";
import { FetchService } from "../services/fetchservice";
import { EventAggregator } from "aurelia-event-aggregator";


export type ScaleDef = {
  from: number,
  to: number,
  color: string
```

```
}

export interface GaugeDef {
  devices: Array<string>
  size: number
  upper: [ScaleDef]
  lower: [ScaleDef]
  message: [string]
  caption?: string
  visible?: boolean
  interval?: number
}

@autoinject
export class Gauges {
  private gauges: Array<GaugeDef> = []

  constructor(private fetcher: FetchService, private ea: EventAggregator) { }

  run(gaugelist: Array<any>) {
    this.gauges = gaugelist
    this.gauges.forEach(gauge => {
      this.getValue(gauge)
      const interval = Math.round((gauge.interval || 1000) + 5000 * Math.random())
      setInterval(() => { this.getValue(gauge) }, interval)
    })
  }

  getValue(gauge: GaugeDef) {
    gauge.devices.forEach(async (device, index) => {
      try {
        const result = await this.fetcher.getIobrokerValue(device)
        this.ea.publish(gauge.message[index], parseFloat(result))
      } catch (err) {
        console.log("An error occured: " + err)
        this.ea.publish(gauge.message[index], "?")
      }

    })
  }
}
```

Neu ist hier zuerst mal die Definition eines “types” namens ScaleDef. Types sind gewissermassen das, was TypeScript ausmacht: Definierbare Typen. Hier sagen wir, dass ein Objekt, dass sich “ScaleDef” nennen will genau die Variablen from, to und color haben muss, wobei from und to Zahlen sein müssen, und color eine Zeichenkette. Der Compiler wird sich beschweren, wenn wir

irgendwo etwas übergeben, wo ein ScaleDef erwartet wird, das diese Kriterien nicht erfüllt.

Gleich darunter folgt ein "interface", das ist eine weitere Möglichkeit, definierbare Typen darzustellen, die Sie vielleicht schon aus anderen Programmiersprachen kennen. Interfaces sind ein wenig mächtiger, als types, funktionieren aber ganz ähnlich. Sie sehen, dass das Interface GaugeDef unter anderem auch auf unsere vorhin definierte ScaleDef zurückgreift. Ein ? am Ende des Attributnamens erklärt dieses Attribut für optional.

Diese beiden Definitionen sind streng genommen nicht notwendig. Sie werden bei Übersetzen in JavaScript sowieso wegtranspiliert. Aber sie bewahren uns vor einer ganzen Klasse von Programmierfehlern: Vergessene Attribute oder Tippfehler in Attributnamen. TypeScript kann bereits beim Transpilieren testen, ob die Typen korrekt sind, so dass nicht erst beim Programmlauf eine Exception geworfen wird, wie das bei reinem JavaScript bei solchen Fehlern der Fall wäre.

Dann folgt die eigentliche Klassendefinition, die wohl nichts Überraschendes enthält: Es geschieht ungefähr dasselbe, wie in der entsprechenden Funktion in app.ts. Nur eines habe ich noch geändert: Man kann das Intervall zwischen zwei Updates optional in der Konfiguration jedes Instruments separat einstellen. Es ist ja nicht bei allen Anzeigen gleich sinnvoll, jede Sekunde einen neuen Wert abzulesen. Es wird also für jedes Instrument ein eigener setInterval() Aufruf abgesetzt. Mit Random() sorgen wir dafür, dass die Aufrufe ein wenig über die Zeit verteilt werden, auch wenn das Interval auf denselben Wert gesetzt ist.

Vielleicht wundern Sie sich über den Aufruf von *this.getValue(gauge)* in *run()*, kurz bevor *this.getValue()* sowieso im setInterval aufgerufen wird. Das mache ich deswegen weil setValue sonst erst nach erstem Ablauf des Inervalls aufgerufen würde. So lange würde man nur einen Null-Wert sehen. Auf diese Weise erhält die Anzeige gleich von Anfang an einen korrekten Wert.

Damit verwandt die Frage: Wieso setzt man die Abfrage für den Startwert nicht gleich im constructor ab, sondern macht dafür eine spezielle run() Funktion? Nun, zum Zeitpunkt des Konstruktors existiert die visuelle Komponente noch nicht. Der Versuch, den Zeiger zu verstellen, würde darum zu diesem Zeitpunkt ins Leere laufen und mit einer Fehlermeldung enden. Die run() Funktion rufen wir im Rahmen von attached() in app.ts auf (s. etwas weiter unten), und das findet ja nach der Konstruktion des DOM statt.

In gleicher Weise habe ich Klassen für Switches und Charts erstellt und im devices-Ordner abgelegt. Ich spare mir jetzt hier das Listing, Sie sehen die Klassen ja im Quellcode-Archiv.

Jetzt wird app.ts sehr viel übersichtlicher:

**src/app.ts**

```
1  import { autoinject } from 'aurelia-framework'
2  import configs from './config'
3  import { Boiler } from "./routes/boiler";
4  import { Gauges } from './devices/gauges';
5  import { Switches } from './devices/switches';
6  import { Charts } from './devices/charts';
7
8  @autoinject
9  export class App {
10   conf = configs
```

```
11
12    constructor(
13      private gauges: Gauges, private switches: Switches,
14      private charts: Charts) { }
15
16    attached() {
17      this.gauges.run([configs.wohnzimmertemp_cfg, configs.aussentemp_cfg,
18        configs.bad_oben_cfg, configs.dusche_cfg, configs.dachstock_cfg,
19        configs.barometer_cfg, configs.powermeter_cfg])
20
21      this.switches.run([configs.fernsehlicht_cfg, configs.carloader_cfg,
22        configs.mediacenter_cfg, configs.extender_cfg, configs.e14_cfg,
23        configs.aussenlicht_cfg])
24
25      this.charts.run([configs.aussen_chart_cfg, configs.barometer_chart_cfg])
26
27    }
28
29    switchGauge(hide, show) {
30      this.conf[hide].visible = false
31      this.conf[show].visible = true
32    }
33  }
```

Im *constructor()* werden die Klassen instanziiert (Falls Ihnen das nicht mehr klar ist, lesen Sie bitte noch einmal im Kapitel über das Aurelia Modulkonzept nach), und in *attached()* wird die jeweilge *run()* Funktion aufgerufen, die die Intervalle erstellt und die Aufrufe des EventAggregators regelt. Mehr benötigen wir nicht mehr in app.ts.

Nun bauen wir auch für den Boiler eine eigene Steuerklasse in “devices”. Hier gibt es nur einen.

**devices/boiler.ts**

```
import { FetchService } from '../services/fetchservice'
import { autoinject } from 'aurelia-framework'
import { EventAggregator } from 'aurelia-event-aggregator'
import configs from '../config'

@autoinject
export class Boiler {

  constructor(private fetcher: FetchService,
              private ea: EventAggregator) {
     this.ea.subscribe(configs.boiler_cfg.message, (msg => {
      this.fetcher.setIoBrokerValue("mqtt.0.Boiler.setTemp", msg)
    }))
  }
```

```
  run() {
    this.getValue()
    setInterval(() => this.getValue(), 10000)
  }
  getValue() {
    this.fetcher.getIobrokerValue("mqtt.0.Boiler.setTemp", undefined)
      .then(temp => {
        this.ea.publish(configs.boiler_cfg.message, temp)
      })
  }
}
```

Sie sehen, dass auch das nicht mehr schwierig ist: Ich setze eine EventAggregator-Subscription auf die message des Boilers und mache daraus einen Schreibzugriff auf den entsprechenden state des mqtt Brokers, der wiederum daraus eine MQTT Nachricht macht. Allerdings müssen wir noch dafür sorgen, dass dieses Teilprogramm zusammen mit dem Hauptprogramm gestartet wird. Wir ergänzen also den constructor von app.ts:

```
constructor(
    private gauges: Gauges, private switches: Switches,
    private charts: Charts private boiler:Boiler) { }
```

Durch das Injizieren des Boilers wird dieser instanziiert. Jetzt können Sie das Programm starten, und es sollte funktionieren.

Wenn Sie einen MQTT Cient, zum Beispiel MQTT Dash auf dem Android-Handy, oder MQTT Tool auf iOS Geräten, auf Ihren ioBroker richten und "/Boiler/setTemp" abonnieren, werden Sie jede Änderung am Schieber sofort auch auf den Handy sehen. Falls der MQTT Server auf dem ioBroker noch keinen entsprechenden State hat, genügt es, von irgendeinem MQTT Client, der mit dem ioBroker verbunden ist, eine entsprechende Nachricht abzusetzen. Der Broker akzeptiert ja alle topics und erstellt für jedes neue Topic umgehend einen ioBroker-State, den wir wiederum wie gewohnt programatisch oder auch über das Admin-UI unter "Objekte" lesen und verändern können.

Die andere Richtung funktioniert allerdings noch nicht: Wenn von woanders her eine Nachricht an /Boiler/setTemp geschickt wird, lässt das den Slider kalt. Wir müssen ihn noch lauschen lassen.

Ergänzen Sie Boiler.ts so:

**routes/boiler.ts**

```
  // constructor...{}

 run() {
    this.getValue()
    setInterval(() => this.getValue(), 10000)
  }
  getValue() {
    this.fetcher.getIobrokerValue("mqtt.0.Boiler.setTemp", undefined)
      .then(temp => {
        this.ea.publish(configs.boiler_cfg.message, temp)
       })
  }
```

und app.ts so:

**src/app.ts**

```
attached() {
    this.gauges.run([configs.wohnzimmertemp_cfg, configs.aussentemp_cfg,
                     configs.powermeter_cfg])

    this.switches.run([configs.fernsehlicht_cfg, configs.carloader_cfg,
      configs.aussenlicht_cfg])

    this.charts.run([configs.aussen_chart_cfg, configs.barometer_chart_cfg])
    this.boiler.run()
  }
```

Wir starten also auch in boiler.ts eine run() Funktion. Diese liest alle 10 Sekunden den aktuellen Wert des ioBroker States für das MQTT Topic /Boiler/setTemp aus und sendet eine entsprechende Nachricht an die setpoint - Komponente (oder an wen auch immer, der die entsprechende Nachricht abonniert hat). Wenn Sie das Programm erneut starten, werden Sie sehen, dass der Slider sofort auf den aktuellen Wert geht. Wenn Sie den Slider verstellen, können Sie am Handy den neuen Wert sehen. Wenn Sie am Handy einen neuen Wert eingeben, wird der Slider nach einigen Sekunden ebenfalls auf diesen neuen Wert gehen. Wir haben jetzt also einen bidirektionalen Informationsfluss. Da die Boilerregelung keine Komponente ist, bei der es auf Geschwindigkeit ankommt, ist die kleine Verzögerung nicht schlimm.

## Motorsteuerung

Bisher haben wir virtuelle Messwerte vom PC über den ioBroker auf ein Handy geschickt und zurück, aber noch keine wirkliche Funktionalität. Wir müssen die Software mit unserem Motor verbinden. Wir hatten dort ja schon eine Motorsteuerung mit Johnny-Five gebaut, die auf Kommando im Uhrzeiger- oder Gegenuhrzeigersinn drehen kann. Alles was noch fehlt, ist ein

MQTT Client in diesem Programm, und eine Möglichkeit, als °C formulierte MQTT Nachrichten in Steps für den Motor umzusetzen.

Dazu bohren wir stepper.js ein wenig auf:

**stepper.js**

```
class Stepper{

  // ...
  setScale(steps, start, end, domainFrom, domainTo, setOrigin) { // (1)
    this.steps360 = steps;
    this.position = start;
    this.calc = val => {
      return five.Fn.map(val, domainFrom, domainTo, start, end)
    }
    this.actPos = 0
    if (setOrigin) {
      this.ccw(steps)
    }
  }

  async goto(pos) {         // (2)
    if (!this.running) {
      this.running = true
      let actPos = this.actPos
      let newPos = this.calc(pos)   // (3)
      this.actPos = newPos
      this.set_step("1001",true)    // (4)
      tis.sleep(10)
      this.run(newPos - actPos).then(() => {  // (5)
        this.sleep(10)
        this.set_step("0000",true)  // (6)
        this.running = false
      })
    }
  }

  async run(steps) {        // (7)
    if (steps < 0) {
      return this.ccw(steps * -1)
    } else {
      return this.cw(steps)
    }
  }
  // ...
}
```

Bei (1) tun wir etwas, was wir so ähnlich schon bei unseren allerersten Anzeigeinstrumenten getan haben: Wir setzen einen Wertebereich (domain) auf einen Drehwinkel (range) um. Nur benutzen wir diesmal für die Rechenarbeit keine d3js-Scale (wir haben hier auf dem Boiler-Controller ja kein D3js installiert und wollen das auch nicht unbedingt), sondern die Funktion Fn.map fon Johnny-Five, die ungefähr dasselbe tut. Unsere Domain sind die Boilertemperaturen von 30-100°C, die Range ist der Drehwinkel des Potentiometers von 0-240°. Zu jeder Boilertemperatur gehört ein bestimmter Winkel, und den können wir errechnen, wenn wir wissen, wieviele Schritte für eine 360° Drehung nötig sind.

Bei (2) wenden wir das Wissen an: Zunächst schauen wir nach, ob der Motor gerade am Drehen ist. Wenn ja, akzeptieren wir keine neuen Kommandos. Dann setzen wir das Flag, welches anzeigt, dass er nun beschäftigt ist. Bei (3) errechnen wir die neue Winkelposition. Dann kommt etwas Seltsames bei (4), auf das ich in wenigen Sekunden zurückkommen werde, bei (5) drehen wir den Motor um die Differenz zwischen alter und neuer Position. Danach warten wir einige Millisekunden, um die Drehachse und die Zahnräder zur Ruhe kommen zu lassen, dann schalten wir bei (6) alle Spulen aus. Der Motor hat nämlich ein recht hohes Haltemoment und saugt rund 300mA, nur um seine momentane Position zu halten. Das ist hier aber gar nicht nötig (Das Potentiometer wird sich sowieso nicht von allein bewegen), und es führt zu einer starken Erwärmung des Motors und zu erhöhtem Stromverbrauch. Daher schalten wir am Ende jeder Einstellungsbedwegung alle Spulen aus. Sie werden feststellen, dass der Motor jetzt völlig kühl bleibt. Allerdings sind die Spulen jetzt nicht mehr in dem Zustand, in dem sie am End des Schritts sein sollten. Daher setzen wir sie am Anfang der Drehfunktion bei (4) auf die Konstellation, die sie natürlicherweise am Ende des vorherigen Schrittes hatten.

(7) ist lediglich eine gewisse Vereinfachung. Wenn die Differenz aus neuer und alter Temperatur positiv ist, wird im Uhrzeigersinn gedreht, sonst im Gegenuhrzeigersinn.

Auch index.js des Motortreibers wurde ein wenig geändert und mit MQTT verbunden:

```
const Raspi = require('raspi-io')
const five = require('johnny-five')
const Stepper=require('./stepper').Stepper
const mqtt=require('mqtt').connect('mqtt://homepi.local',{
  clientId: "Boiler-Motor",
  will:{
    topic: "/Boiler/ackn",
    payload: "motor disconnected"
  }
})

const board = five.Board({
    io: new Raspi(),
    repl: false
  }
);
const range=240
const stepsPer360=512
```

```
const orange = "GPIO26"
const yellow = "GPIO19"
const pink = "GPIO13"
const blue = "GPIO6"
let speed = 15;

let stepper;

mqtt.on("connect",()=>{
  mqtt.subscribe("/Boiler/setTemp")
  mqtt.publish("/Boiler/ackn","motor ready")
})

mqtt.on('message',(topic,message)=>{                // (1)
  if(topic==="/Boiler/setTemp"){
    stepper.goto(parseInt(message)).then(()=>{
    })
  }
})

board.on('ready', async () => {                     // (2)
  stepper = new Stepper(orange, yellow, pink, blue, speed)
  stepper.setScale(stepsPer360,0,240,20,100,true)
  function setSpeed(sp) {
    stepper.speed = sp;
  }
})
```

Neu ist bei (1), dass wir “setTemp”-Nachrichten von MQTT abfangen, un diese an die eben diskutierte goto-Funktion des Steppers weiterleiten. Bei (2) initialisieren wir den Controller, indem wir die Verbindungen deklarieren und die Scale setzen. Der letzte Parameter “true” bewirkt, wenn Sie nochmal kurz bei stepper.js nachsehen, das der Motor um eine volle Drehung im Gegenuhrzeigersinn bewegt wird. Das heisst, er wird ziemlich sicher am linken Anschlag “anstossen”. Nun, das schadet einem Schrittmotor nichts. Er dreht einfach nicht mehr weiter, wenn er am Anschlag ist. Aber auf diese Weise haben wir einen definierten Ausgangspunkt: Wir wissen jetzt mit Sicherheit, dass das Potentiometer ganz nach links gedreht ist.