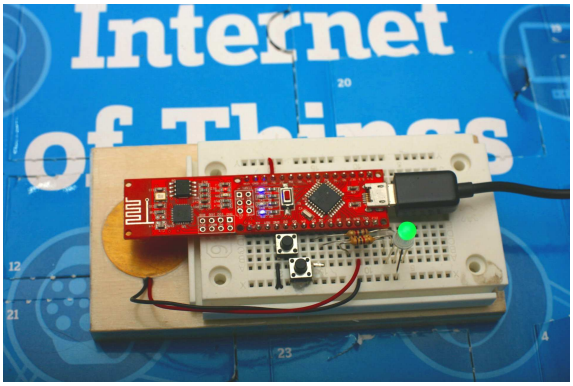
IoTeaTimer

Internet of Things Tea Timer



General Overnight

31. Januar 2016

Inhaltsverzeichnis

Inhalt iii

1 Projektvorstellung 1
1.1 Funktionsweise . . . 1
1.2 Anwendung . . . 2

2 Aufbau der Schaltung 3
2.1 Begründung der gewählten Anschlussbelegung . . . 3
2.2 Die Alarmtöne . . . 4
2.2.1 Die Tonauswahl . . . 4

3 Die Software 5
3.1 Software-Features . . . 5
3.1.1 Modularisierung . . . 5
3.1.2 Tests und Debugging . . . 6
3.1.3 Kommentare im Quelltext . . . 6

4 Geplante Weiterentwicklung 7

A Schaltpläne 9

B Quelltext-Listings 11

C Verwendete Bauteile 29

Kapitel 1

Projektvorstellung

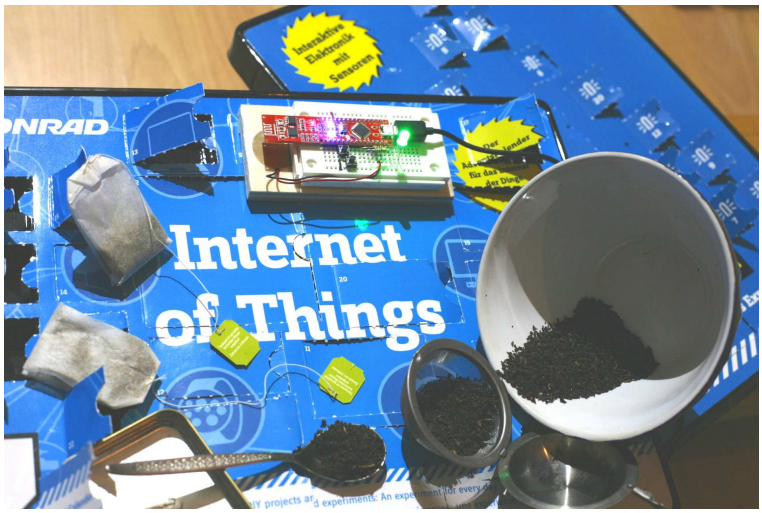


Abbildung 1.1: Internet of Tea

Der IoTeaTimer löst folgendes Problem. Je nach Art, muss ein echter oder Kräutertee drei bis 15 Minuten ziehen. Erfolgt die Teezubereitung nicht dort, wo er getrunken wird (etwa im Arbeitszimmer), sondern beispielsweise in der Küche, dann kann er leicht vergessen werden und viel zu lange ziehen.

Handelsübliche Kurzzeitmesser lösen das Problem nur unbefriedigend. Zuallererst müssen sie bei jedem Raumwechsel mitgenommen werden. Gerade bei modernen, elektronischen Geräten besteht dabei die Gefahr, dass die Zeiteinstellung versehentlich verändert wird, weil der Gerätedesigner die Bedienungselemente ausgerechnet in alle nutzbaren Griffflächen oder deren unmittelbare Nähe gelegt hat[1] Bei den im Handel erhältlichen Kurzzeitmessern ist darüber hinaus meist auch das Umstellen der Zeiten aufwändig und lästig, wenn verschiedene

[1]Die Bedienelemente in den Griffflächen sind ein ähnlich verbreitetes und „benutzerfreundliches“ Feature wie die streng monochrome Beschriftung (schwarz auf schwarz oder weiß auf weiß), auf die Douglas Adams im „Hitchhiker's Guide to the Galaxy“ angespielt hat und die auch auf dem Breadboard Syb-46 zu finden ist. Möglicherweise ist das eine Art Produktdarwinismus, der die Absatzmärkte für besonders hässliche, explizit als barrierefrei vermarktete Produkte sichern soll, anstatt alle benutzerfreundlich zu gestalten.

Teesorten im Wechsel zubereitet oder die Timer auch für andere Zwecke eingesetzt werden. Dabei sind eigentlich meist nur wenige, immer gleiche Zeiten nötig.

1.1 Funktionsweise

Um die beschriebenen Probleme zu lösen erlaubt der IoTeaTimer mehrere voreingestellte Alarmzeiten auf Tastendruck zu aktivieren. Weiterhin signalisiert er den Ablauf der Zeit nicht nur lokal, per Signalton und LED. Er stellt einen WLAN Access Point zur Verfügung, an dem sich andere Geräte anmelden können und dann über einen UDP-Broadcast benachrichtigt werden. Dadurch kann der IoTeaTimer (wie auch die im Haus oder der Wohnung verteilten Empfänger) immer an seinem Platz verbleiben. So sind die Geräte immer griffbereit und es wird kein Alarm übersehen bzw. überhört, weil gerade niemand im Raum ist.

Als Empfänger für die Alarmmeldungen eignet sich besonders die Geräte aus dem Schwesterprojekt IoTPager, das ebenfalls auf Basis des nanoESP realisiert ist. Dieser erzeugt die gleichen Alarmsignale wie der IoTeaTimer.

1.2 Anwendung

Nach dem Einschalten (dem Anschließen der Stromversorgung) oder einem Reset meldet sich der IoTeaTimer mit einem Dreiklang, sobald das Gerät betriebsbereit ist. Ab diesem Zeitpunkt kann über die beiden Taster ein Timer gestartet werden.

In der ersten Version aktiviert jeder der beiden Taster einen Timer unterschiedlicher Länge und es kann nur jeweils ein Timer laufen. Dies wird in der zweiten Version geändert (siehe Kapitel 4).

Die RGB-LED signalisiert durch ihre Farbe, welche voreingestellte Alarmzeit gewählt wurde (Rot: 5 min, Grün: 15 min). Sie wechselt dabei im Sekundentakt zwischen der vollen und einer reduzierten Helligkeit, die vom Verhältnis der Restlaufzeit zur Gesamtzeit abhängt. Das heißt, sie leuchtet zunächst praktisch stetig und beginnt dann immer stärker zu blinken.

Kurz vor dem Ablauf des Timers alarmiert der IoTeaTimer alle Geräte, die sich an seinem Access Point angemeldet haben und auf den entsprechenden Timerablauf warten. Die Benachrichtigungen werden per UDP an die Ports 1070 (roter Timer), 1071 (gelber Timer), 1072 (grüner Timer) oder 1073 (blauer Timer) gesendet. Anschließend spielt er für 20 Sekunden eine Tonsequenz ab und lässt die LED in der Farbe des ablaufenden Timers blinken. Das präzise Ende des Timeouts fällt mit dem Ende der Tonsequenz zusammen.

Tipp Ein versehentlich aktivierter Timer kann durch Drücken des Reset-Tasters auf dem Pretzelboard deaktiviert werden.

Kapitel 2

Aufbau der Schaltung

Der Zusammenbau des IoTeaTimers beginnt mit dem Einsetzen des nanoESP in das Breadboard. Dies geschieht in etwas anderer Weise als beim IoT-Adventskalender 2015. Anstatt in die Reihen D und H, werden die Reihen E und I genutzt. Das lässt oberhalb des Pretzelboards mehr Raum, um die Taster so zu platzieren, dass sie keine ungenutzten Eingänge schalten.

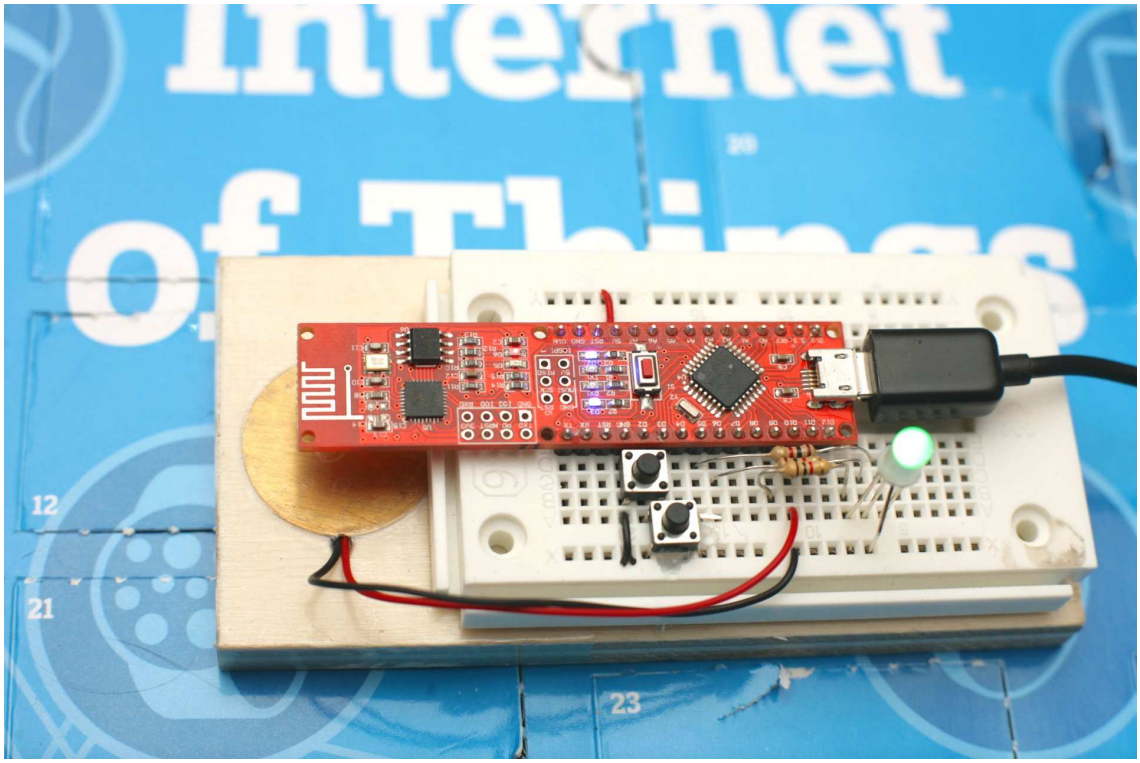


Abbildung 2.1: Aufbau des IoTeaTimers auf einem Breadboard

Die obere Verbindungsleiste wird über eine Drahtbrücke mit dem Masseanschluss an Pin 4 des Pretzelboards verbunden (Spalte 20, Reihe X und A).

Der erste Taster wird mit zwei verbunden Anschlüssen auf der rechten Seite in die Spalte 20

gesetzt (Reihe B und D), in der auch der Masse-Pin des Pretzelboards steckt. Die anderen beiden Anschlüsse stecken dann in der Spalte 17 (Pin D3).

Für den kollisionsfreien Einbau des zweiten Tasters muss ein Anschlussbeinchen hochgebogen werden. Er steckt mit einem Paar seiner verbundenen Anschlüsse in den Spalten 19 und 16 der Verbindungsleiste am oberen Rand des Breadboards (Masseanschluss). Das Beinchen links unten würde dadurch normalerweise mit dem Ausgang D5 des nanoESP verbunden werden (Spalte 16), der aber für die RGB-LED benötigt wird. Damit der Taster trotzdem gut im Breadboard hält, werden die Anschlüsse mit einer Zange flach gedrückt und der Taster auf dem Board mit einem Tropfen Heißkleber fixiert.

Die RGB-LED kann mit ihrem Masse-Anschluss direkt in der oberen Verbindungsschiene sitzen. Drei 1 kΩ-Widerstände verbinden schließlich die Ausgänge D5, D6 und D9 des Pretzelboards mit den Anoden der RGB-LED.

2.1 Begründung der gewählten Anschlussbelegung

Die Taster sind an die Pins D3 und D4 angeschlossen, da diese als einzige eine Interruptsteuerung ermöglichen. Das wird in der vorliegenden Version zwar noch nicht genutzt, soll aber bei einer späteren Erweiterung eine schnelle Reaktion auf Benutzereingaben sicherstellen. Durch diese Wahl geht zwar einer der knappen PWM-Ausgänge verloren, sie vermeidet aber auch Probleme durch Interferenzen zwischen PWM- und Tonausgabe, die laut Datenblatt beim Anschluss des Piezo-Summers an Pin D3 entstehen können.

RGB-LED und Piezo-Summer benötigen PWM-Ausgänge. Für ihren Anschluss bleiben also noch die Ausgänge D5 und D6 sowie D9 und D10. Die RGB-LED soll mit veränderlicher Helligkeit betrieben werden und muss daher über Pulsweitenmodulation (PWM) gesteuert werden. Die Wahl des Ausgangs D10 für den Summer führt zum Anschluss der LED über die aufeinanderfolgenden Ausgänge D5 (rot), D6 (grün) und D9 (blau).

2.2 Die Alarmtöne

Ohne einen Resonanzkörper erzeugt ein Piezo-Signalgeber nur extrem leise Töne und eignet sich in dieser Form nicht für eine praktische Anwendung im IoTeaTimer. Eine deutliche Steigerung der Lautstärke lässt sich schon durch eine Befestigung mit Klebeband auf einer ebenen Fläche erreichen. Eine weitere Verbesserung bringt die Montage des Summers über einem Hohlraum. Die optimale Befestigung erfolgt laut Datenblatt eines Herstellers im Zentrum der kreisförmigen Scheibe. Praktikabler ist die zweitgünstigste Montageart, das Befestigen am äußersten Rand.

Für den Einbau des Summers in den IoTeaTimer wird mit einem 26 mm Forstnerbohrer ein etwa 5 mm tiefes Loch in die Grundplatte aus 10 mm Sperrholz gebohrt (siehe Bild 2.2). Da der Piezo-Summer aus dem IoT-Adventskalender einen Durchmesser von 27 mm hat, ergibt sich ein Überstand von einem halben Millimeter. Die vom Forstnerbohrer erzeugte Zentrierbohrung wird mit einem 3 mm Bohrer durchgebohrt. Im praktischen Versuch zeigt das einen etwas besseren Klang. Das kleine Loch lässt sich für einen Vergleich sehr leicht

schließen und öffnen. Zum Testen erfolgt die Befestigung des Summers zunächst mit einem breiten, transparenten Klebestreifen (siehe Bild 2.3).

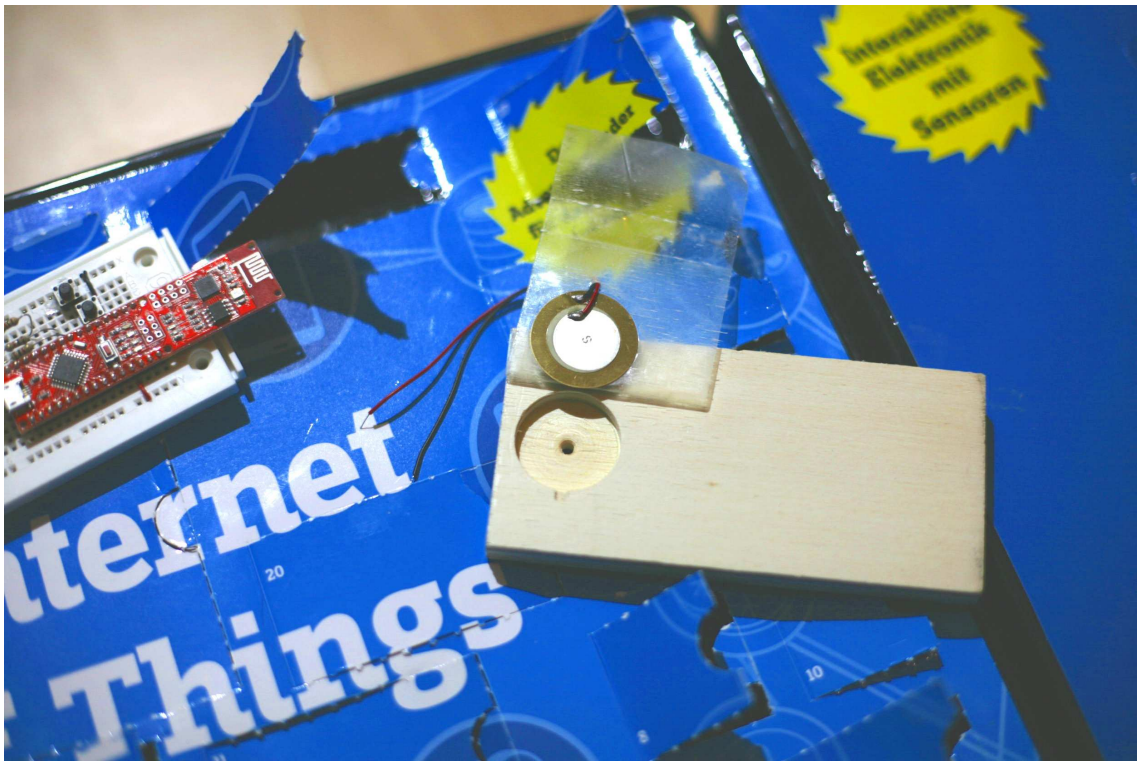


Abbildung 2.2: Die Grundplatte für den IoTeaTimer

2.2.1 Die Tonauswahl

Im Vergleich mit einem Lautsprecher bleibt der Frequenzgang des Summers auch nach dem Einbau in ein Gehäuse sehr ungleichmäßig. Er hängt zudem stark vom verwendeten Modell und von der Art des Einbaus ab. Daher sollte die Alarmtonsequenz schon unabhängig vom persönlichen Geschmack individuell angepasst werden. Um das zu vereinfachen sind in der Datei `mytones.h` Makros für den brauchbaren Frequenzbereich enthalten, die eine Ausgabe von Tönen anhand der Namen aus der Tonleiter ermöglichen.

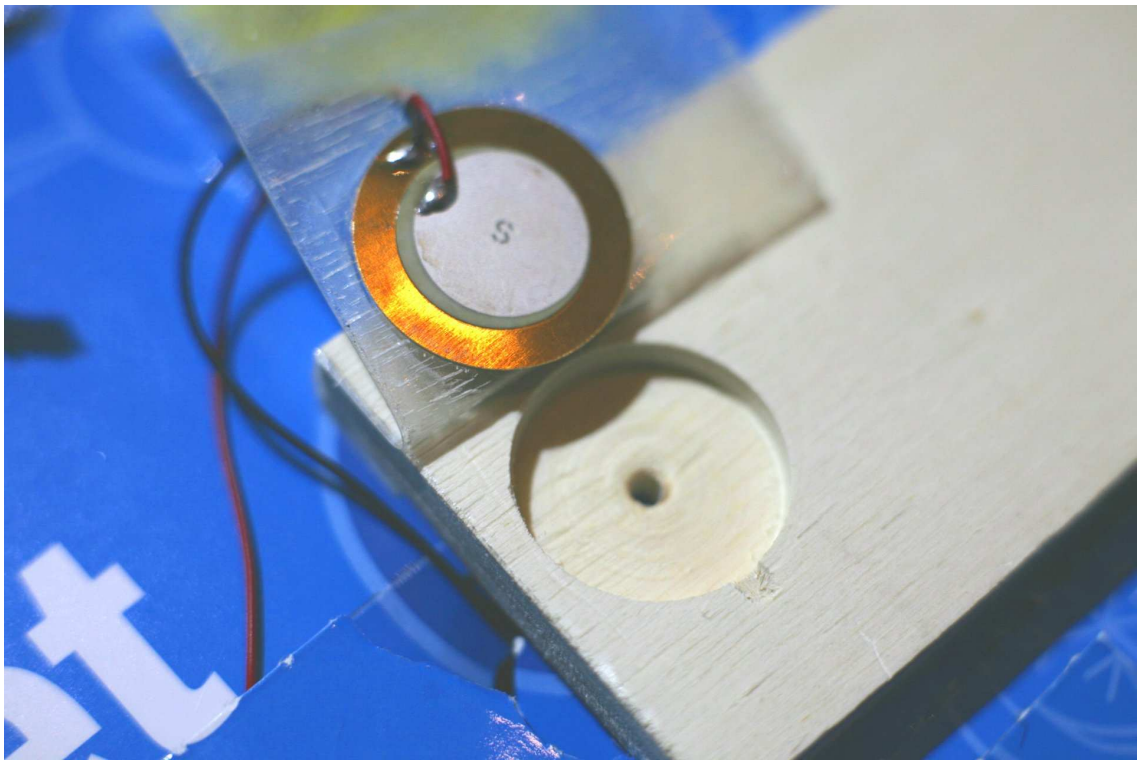


Abbildung 2.3: Einbau des Piezo-Summers

Kapitel 3

Die Software

Die Implementation des IoTeaTimers nutzt einen Ansatz, der die Anpassung der Software und auch die Entwicklung ganz neuer Projekte auf einfache Art erleichtert, ohne eigene Bibliotheken zu definieren. Dazu sind in IoTeaTimer.ino (siehe Listing B.1)

1. zunächst alle Code-Bestandteile an den Dateianfang gerückt, die lokal anzupassen oder häufig zu ändern sind.

2. folgen danach projektspezifische Funktionen, in denen die Funktionalitäten gekapselt sind, die aus der Hauptschleife (void loop() {...}) aufgerufen werden. So bleiben diese generischen Funktionen kurz und lassen sich leicht anpassen, wenn der Quelltext für ein geändertes oder neues Projekt übernommen wird.

3. sind immer gleiche Code-Teile, die üblicherweise von Projekt zu Projekt mitgenommen werden, in eigene Dateien ausgelagert, die über Präprozessor-Direktiven (#include " ... ") eingebunden werden[1].

3.1 Software-Features

3.1.1 Modularisierung

Beim Aufbau der Experimente aus dem IoT-Adventskalender ist aufgefallen, dass große Teile des Codes unverändert in das neue Projekt übernommen werden mussten. Die beim IoTeaTimer genutzte Modularisierung vereinfacht das, da die gleichbleibenden Code-Teile in separate Dateien ausgelagert sind, die einfach kopiert werden können. Im Idealfall muss dann lediglich die Hauptdatei des Projekts angepasst werden. Dazu wurde außerdem eine möglichst universelle Belegung der I/O-Pins entworfen.

[1] In spitzen Klammern notierte Namen innerhalb von Include-Direktiven beziehen sich auf Header-Dateien in Systemverzeichnissen. Dagegen sucht der Präprozessor in Gänsefüßchen notierte Dateinamen im aktuellen Verzeichnis, d.h. dort, wo auch die Hauptdatei <Projektname>.ino liegt.

3.1.2 Tests und Debugging

Folgende Features erleichtern beim IoTeaTimer das Testen und die Fehlersuche (Debugging).

- Debug-Meldungen können über Syslog-kompatible Level nach ihrem Wichtigkeitsgrad gekennzeichnet und damit selektiv ein- und ausgeschaltet werden.
- Durch Definition des Makros NDEBUG können alle Debugausgaben komplett abgeschaltet werden.
- Die Funktion zur Ausgabe von Debug-Meldungen ersetzt Passwörter und andere sensitive Zeichenketten durch gleichlange Strings, die nur aus dem Buchstaben „X“bestehen.
- Für Testzwecke kann über das Makro TEST_ACTIONS eine reduzierte Minutenlänge eingestellt werden.

Bei der Kodierung der Präprozessor-Direktiven wurde darauf geachtet, dass deaktivierter Code möglichst komplett aus dem kompilierten Quelltext verschwindet, um die knappen Ressourcen des Pretzelboards nicht unnötig zu strapazieren.

```
1 if (DEBUG) { ... }
```

Das Makro DEBUG wird hier erst zur Laufzeit abgefragt, als jedesmal, wenn das Programm an der entsprechenden Stelle ausgeführt wird. Der Wert des Makros wird aber beim Kompilieren festgelegt und ändert sich nit im lauffähigen Programm.

```
1 #if DEBUG
2   ...
3 #endif
```

Bei dieser Notation alles zwischen dem #if DEBUG und dem #endif bereits vom Präprozessor, d.h. vor dem eigentlichen Kompilieren aus dem Quelltext entfernt, wenn DEBUG den Wert false hat. Er benötigt daher weder Speicherplatz noch andere Ressourcen während des Programmablaufs.

3.1.3 Kommentare im Quelltext

Um die Übersichtlichkeit des Quellcodes zu verbessern, sind Kommentare die der Dokumentation dienen im C-Stil notiert (/* Kommentartext */). Der C++-Stil (// Kommentartext) wird dagegen für das (vorübergehende) Auskommentieren, d.h. Deaktivieren von Code-Zeilen genutzt.

Kapitel 4

Geplante Weiterentwicklung

- Erweiterung auf vier vordefinierte Alarmzeiten (rot, gelb, grün, blau), die mit Taster 1 ausgewählt und mit Taster 2 aktiviert werden.
- Unterstützung für den unabhängigen Ablauf mehrerer, verschiedener Timer.
- Einbau des Bewegungsschalters aus dem Elektronik-Adventskalender, sodass ein laufender Alarm durch Bewegen des Geräts bestätigt und damit beendet werden kann.

Anhang A

Schaltpläne

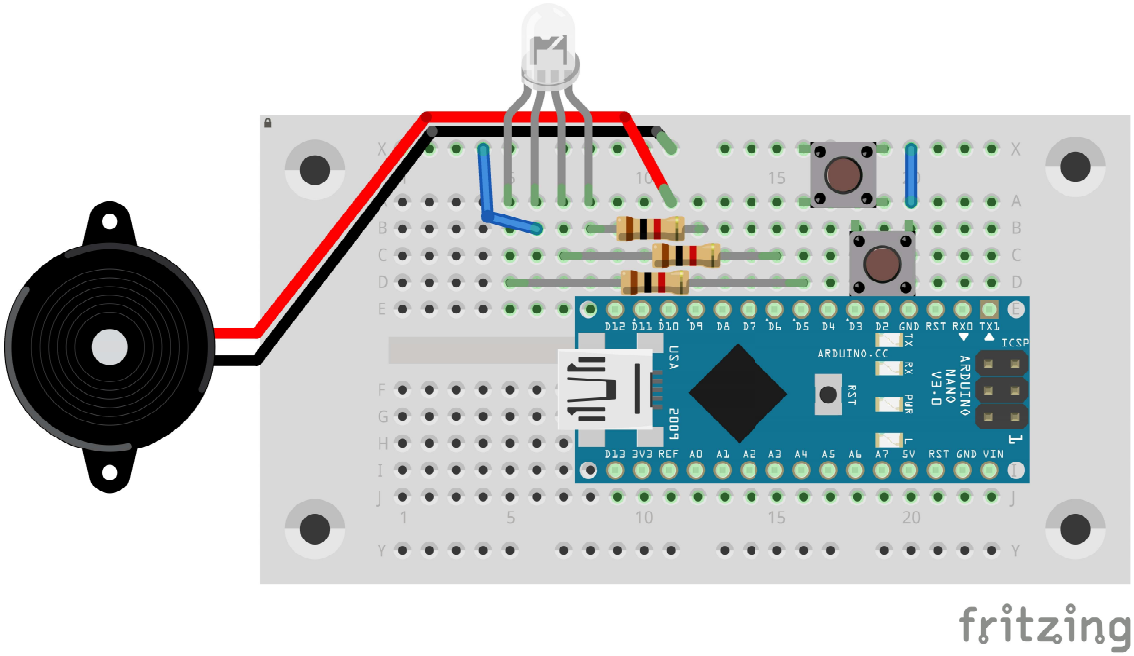


Abbildung A.1: IoTeaTimer Breadboard Layout

Anhang B

Quelltext-Listings

Listing B.1: Hauptquelltext IoTeaTimer.ino

```
1  /*
2   * IoT Tea Timer - set an alarm on a nanoESP, receive it via inet (WiFi).
3   *
4   * (C) 2016 General Overnight <generalovernight.wordpress.com>
5   *
6   * This is Free Software available under the GNU General Public License (GPL).
7   *
8   * $Id: IoTeaTimer.ino,v 1.6 2016-01-31 08:55:30 manolo Exp $
9   */
10
11 /* Include Arduino.h before any local header files as ist provides the
12  * standard types used therein (like boolean or String).
13  */
14 #include <Arduino.h>
15
16 #include "mytones.h"
17 #include "logdebug.h"
18 #include "wlansetup.h"
19 #include "serialcomm.h"
20
21 /* ********************** */
22 /* *** Debug settings *** */
23 /* ********************** */
24
25 /* NOTE: uncomment the following line for fully silent operation
26  * regardless of the DEBUG setting below (production code).
27  */
28 #define NDEBUG
29
30 /* We use a subset of the standard syslog levels for classifying debug
31  * output here. These are defined in "logdebug.h" and include can be
32  * specified with the macros LVL_EROR, LVL_WARN, LVL_NOTE, LVL_INFO and
33  * LVL_DBUG.
34  */
35
```

```
36 /* Set the minimum level of debug messages to be printed (defaults to
37  * LVL_WARN if undefined)
38  */
39 #define DEBUG LVL_INFO
40
41 #ifndef NDEBUG
42 /* This flag enables short action timeouts so you don't have to wait a
43  * long time when testing the corresponding functions.
44  */
45 # define TEST_ACTIONS
46 /* NOTE: comment out the following line to enable TEST_ACTIONS */
47 # undef TEST_ACTIONS
48 #endif
49
50 /* ********************************** */
51 /* *** End section: Debug settings *** */
52 /* ********************************** */
53
54 /* ********************************** */
55 /* *** Custom Settings - Edit here! *** */
56 /* ********************************** */
57
58 /* Read in the SSID, PSK and IP settings from secrets.h (that file has
59  * to be created locally, see example-secrets.h)
60  */
61 #include "secrets.h"
62
63 /* Default timeout for AT commands */
64 #define DEFTO 6000
65
66 #ifdef TEST_ACTIONS
67 /* Reduced length of a minute to speed up testing (this should be
68  * greater than ADJTIME + ALRMLEN)
69  */
70 # define MINUTE_LEN 12
71 #else
72 # define MINUTE_LEN REAL_MINUTE
73 #endif
74
75 /* Adjust the timeout by this many seconds to account for the time spent
76  * sending the alarm over the wireless network.
77  */
78 #define ADJTIME 18
79
80 /* The duration of the alarm sound will be subtracted from the timeout
81  * so that the end of the sound output marks the precise end of the
82  * timeout.
83  */
84 #define ALRMLEN 20
85
86 /* Blink the onboard LED D3 (pin 13) to indicate activity if true, else
87  * the RGB-LED.
88  */
```

```
89  //#define USE_ONBOARD_LED true
90  #define USE_ONBOARD_LED false
91
92  /* Red timer: 5 min */
93  #define TIME_R 5
94  /* Green timer 15 min */
95  #define TIME_G 15
96
97  /* ** Destination UDP Port numbers, defined as strings for convenience ** */
98  /* Red alarm */
99  #define PORT_R "1070"
100 /* Green alarm */
101 #define PORT_Y "1071"
102 /* Blue alarm */
103 #define PORT_G "1072"
104 /* Yellow alarm */
105 #define PORT_B "1073"
106
107 /* Define (a standard) I/O pin setup */
108 #define BTN0 2 /* Interrupt */
109 #define BTN1 3 /* Interrupt (PWM also, but tone output may interfere) */
110 #define LED_COM 8
111 #define LED_R 5 /* PWM */
112 #define LED_G 6 /* PWM */
113 #define LED_B 9 /* PWM */
114 #define BUZZ 10 /* PWM */
115
116 /* ************************** */
117 /* *** End custom Settings *** */
118 /* ************************** */
119
120 /* number of seconds in a real (wall clock) minute */
121 #define REAL_MINUTE 60
122
123 /* maximum value for PWM outputs */
124 #define MAX_PWM 255
125
126 /* Set up the serial connction between Atmega MCU and ESP WLAN-Module */
127 #include <SoftwareSerial.h>
128 SoftwareSerial esp8266(11, 12);
129
130 /* ********************************** */
131 /* *** Custom Functions: Edit here *** */
132 /* ********************************** */
133
134 /* *** Event handler *** */
135
136 int handleButton(int btn)
137 {
138   int rc = 0;
139   int led = -1;
140   int timeout = 0;
141   String port = "-";
```

```
142   unsigned long start = millis();
143
144   switch (btn)
145   {
146     case BTN0:
147       led = LED_R;
148       port = PORT_R;
149       timeout = TIME_R * MINUTE_LEN - ADJTIME - ALRMLEN;
150       break;
151     case BTN1:
152       led = LED_G;
153       port = PORT_G;
154       timeout = TIME_G * MINUTE_LEN - ADJTIME - ALRMLEN;
155       break;
156     default:
157       /* software error (cf. sysexits.h) */
158       rc = 70;
159       dbg(LVL_EROR, "Software error: invalid button ID: " + String(btn));
160   }
161
162   if (0 == rc)
163   {
164     dbg(LVL_INFO, "Button activated, timeout = " + String(timeout));
165
166     digitalWrite(led, HIGH);
167     /* Who is connected? */
168     dbg(LVL_INFO, runATcmd("+CWLIF", 1));
169     /* Spend the time until the scheduled timeout in a loop blinking an LED */
170     for (int i = timeout; i--> 0;)
171     {
172 #if USE_ONBOARD_LED
173       digitalWrite(LED_BUILTIN, i % 2);
174 #else
175       rgbWrite(led, (unsigned int)((float)MAX_PWM * (float)i / (float)timeout));
176 #endif
177       delay(999);
178     }
179     /* Timeout! Send an alarm over WiFi first ... */
180     if (configUDP(port))
181     {
182       tone(BUZZ, TONE_a2, 200);
183       delay(200);
184       /* Someone connected? */
185       dbg(runATcmd("+CWLIF", 1));
186       // Alas, ESP chokes on this:
187       //sendUDP(DSTIP, port, "ALRM");
188       sendUDP("ALRM");
189       tone(BUZZ, TONE_c1, 200);
190       delay(200);
191       runATcmd("+CIPCLOSE", "OK");
192       tone(BUZZ, TONE_c1, 200);
193       delay(200);
194     } else {
```

```
195         dbg("Error opening UDP connection");
196         tone(BUZZ, TONE_dis2, 200);
197         delay(200);
198       }
199       dbg(LVL_NOTE,
200           "Network alarm finished after " + String((millis() - start) / 1000)
201           + " seconds.");
202       /* sound an alarm */
203   #if USE_ONBOARD_LED
204       digitalWrite(LED_BUILTIN, HIGH);
205   #endif
206       for (int i = 5; i--> 0;)
207       {
208         rgbWrite(led, LOW);
209         tone(BUZZ, TONE_c1);
210         /* tone returns immediately, use delay to determine its duration */
211         delay(1000);
212         rgbWrite(led, LOW);
213         tone(BUZZ, TONE_e1);
214         delay(1000);
215         rgbWrite(led, LOW);
216         tone(BUZZ, TONE_g1);
217         delay(1000);
218         rgbWrite(led, LOW);
219         tone(BUZZ, TONE_ais2, 1000);
220         delay(1000);
221         noTone(BUZZ);
222       }
223       dbg(LVL_NOTE,
224           "Timer finished after " + String((millis() - start) / 1000)
225           + " seconds.");
226       /* ... finally reset the timer LED */
227       digitalWrite(led, LOW);
228     }
229   }
230
231
232   /* Lights up the RGB LED alternating between full on and a dimmed mode
233    * specified by the brightness parameter.
234    *
235    * TODO: The colour parameter currently only supports red and green
236    * blue. It expects the pin number as its value which is to be changed.
237    *
238    */
239   void rgbWrite(unsigned int colour, unsigned int brightness)
240   {
241     /* A variable declared static keeps its value across multiple
242      * invocations and is initialized only once, when the function is
243      * first run.
244      */
245     static boolean toggle = false;
246
247   #ifndef NDEBUG
```

```
248   /* Use a standard C prinf function to format the string to be output
249    * with Serial.println into a C character array of appropriate length
250    * (80 characters default terminal line length).
251    */
252   char msg[82] = "";
253   /* One extra byte at the end for the ASCII-NUL terminating the
254    * C-string and an additional one as a safety net.
255    */
256   snprintf(msg, sizeof(msg)-2,
257            "rgbWrite(colour=%u, brightness=%u)", colour, brightness);
258   Serial.println(msg);
259 #endif
260   /* sanity check the brightness value to guard against programming errors */
261   if (brightness > MAX_PWM) { brightness = MAX_PWM; }
262
263   /* Reset all LED outputs first */
264   digitalWrite(LED_R, LOW);
265   digitalWrite(LED_G, LOW);
266   digitalWrite(LED_B, LOW);
267   if (toggle)
268   {
269     /* Light up the LED with the specified colour in full brightness */
270     switch (colour) {
271       case LED_R:
272         digitalWrite(LED_R, HIGH);
273         break;
274       case LED_G:
275         digitalWrite(LED_G, HIGH);
276         break;
277     }
278   }
279   else
280   {
281     /* Light up the LED with the specified colour in reduced brightness */
282     switch (colour) {
283       case LED_R:
284         analogWrite(LED_R, brightness);
285         break;
286       case LED_G:
287         analogWrite(LED_G, brightness);
288         break;
289     }
290   }
291   /* toggle the value of toggle for the next run */
292   toggle = ! toggle;
293 }
294
295
296 /* ************************** */
297 /* *** End custom functions *** */
298 /* ************************** */
299
300
```

```
301 /* ==== Main ================================================================ */
302
303 void setup() {
304   int rc = 0;
305   Serial.begin(19200);
306   esp8266.begin(19200);
307
308   tone(BUZZ, TONE_h2, 500);
309
310   if (0 != (rc = espConfig())) {
311     tone(BUZZ, TONE_h2, 1500);
312     for (int i = rc; i> 0; i>>=1) {
313       /* tone returns immediately so a delay is necessary as
314        * long as the beep duration at least (second arg to tone).
315        */
316       delay(1000);
317       tone(BUZZ, TONE_h, 500);
318     }
319     serialDebug();
320   } else {
321     tone(BUZZ, TONE_c1, 500);
322   }
323
324   /* configure I/O */
325   pinMode(LED_R, OUTPUT);
326   pinMode(LED_G, OUTPUT);
327   pinMode(LED_B, OUTPUT);
328   pinMode(BTN0, INPUT_PULLUP);
329   pinMode(BTN1, INPUT_PULLUP);
330
331   /* Play a triad (Dreiklang) when ready The tone duration is not
332    * specified but limited by the following commands.
333    */
334   tone(BUZZ, TONE_c1);
335   delay(500);
336   tone(BUZZ, TONE_dis1);
337   delay(500);
338   tone(BUZZ, TONE_gis1, 500);
339   delay(500);
340 }
341
342
343 void loop() {
344   boolean sw0 = !digitalRead(BTN0), sw1 = !digitalRead(BTN1);
345 #if DEBUG > 1
346   static unsigned long timestamp = millis();
347   unsigned long now = millis();
348   unsigned long runtime = now - timestamp;
349   if (runtime) {
350     dbg( 7, "Program Uptime: " + String(now / 1000.0)
351          + " s;  Main loop: previous run "
352          + String(runtime) + " ms ago; Buttons: "
353          + String(sw0) + "," + String(sw1));
```

```
354     } else {
355       dbg(7, "Program Uptime: " + String(now / 1000.0)
356           + "s.  Main loop: first run; Buttons: "
357           + String(sw0) + "," + String(sw1));
358     }
359     timestamp = now;
360   #endif
361
362     if (sw0 | sw1)
363     {
364       if (sw0)
365       {
366         /* sw1 has been pressed */
367         handleButton(BTN0);
368       }
369       else /* sw1 must have been pressed, else we had not entered this loop */
370       {
371         handleButton(BTN1);
372       }
373     } else {
374       /* Check the serial Atmel-ESP link for pending data, process it, if
375        * any or wait a quarter of a second before the next loop run.
376        */
377       if (!serialIO()) { delay(250); }
378     }
379   }
380
381
382   /* ======= End Main ============================================================ */
383
384   // Local Variables:
385   // mode: c++
386   // End:
```

Listing B.2: Tondefinitionen in mytones.h

```
1  #ifndef MYTONES_H
2  #define MYTONES_H
3
4  // Define some named pitches
5  #define TONE_h     247
6  #define TONE_a1    440
7  #define TONE_h1    494
8  #define TONE_c1    523
9  #define TONE_cis1  554
10 #define TONE_d     587
11 #define TONE_dis1  622
12 #define TONE_e1    659
13 #define TONE_eis1  698
14 #define TONE_fis1  740
15 #define TONE_g1    784
16 #define TONE_gis1  830
17 #define TONE_dis2  1174
18 #define TONE_g2    1568
```

```
19 #define TONE_gis2 1661
20 #define TONE_a2   1760
21 #define TONE_ais2 1865
22 //                  ++
23 #define TONE_h2   1975
24 #define TONE_c3   2093
25 //                  +
26 #define TONE_cis3 2217
27 #define TONE_Gis3 3322
28 #define TONE_a3   3520
29 #define TONE_h3   3729
30 #define TONE_c3   3951
31 //#define TONE_
32
33
34 #endif
```

Listing B.3: Logging- und Debug-Funktionen in logdebug.ino

```
1  /*
2   * Common logging and debugging functions.
3   *
4   * (C) 2016 General Overnight <generalovernight.wordpress.com>
5   *
6   * This is Free Software available under the GNU General Public License (GPL).
7   *
8   * $Id: logdebug.ino,v 1.1 2016-01-31 08:55:30 manolo Exp $
9   */
10
11 void serialDebug(void) {
12   while (true)
13   {
14     if (esp8266.available()) { Serial.write(esp8266.read()); }
15     if (Serial.available()) { esp8266.write(Serial.read()); }
16   }
17 }
18
19
20 void dbg(String msg)
21 {
22 #ifdef DEBUG
23   const String secrets[] = { PSK, SSID, };
24   /* Replace any instances of the secrets defined above in the message
25    * with a equal length string of Xs.
26    *
27    * sizeof(secrets) returns the number of bytes occupied by the secrets array.
28    * This is divided by the space necessary to hold a String item to get
29    * the number actual number of items in the array.
30    *
31    * We start from the end of the array for convenience.
32    */
33   for (int i = sizeof(secrets) / sizeof(String); i--> 0;) {
34     /* try to find the string at position i in secrets in the msg
35      * String.
```

```
36        */
37       int start = msg.indexOf(secrets[i]);
38       if (-1 < start) {
39         String newmsg = msg.substring(0, start);
40         for (int k = secrets[i].length(); k--> 0;) {
41           newmsg += "X";
42         }
43         msg = newmsg + msg.substring(start + secrets[i].length());
44       }
45     }
46     /* Finally output the sanitized message */#
47     Serial.println(msg);
48 #endif
49 }
50
51 void dbg(int level, String msg)
52 {
53 #ifndef NDEBUG
54 #ifdef DEBUG
55     if (level <= DEBUG) {
56 #else
57     if (level <= LVL_WARN) {
58 #endif
59       dbg(msg);
60     }
61 #endif
62 }
63
64
65 // Local Variables:
66 // mode: c++
67 // End:
```

Listing B.4: Headerdatei für logdebug.ino

```
1  /*
2   * Common logging and debugging functions.
3   *
4   * (C) 2016 General Overnight <generalovernight.wordpress.com>
5   *
6   * This is Free Software available under the GNU General Public License (GPL).
7   *
8   * $Id: logdebug.h,v 1.1 2016-01-31 08:55:30 manolo Exp $
9   */
10
11 #ifndef LOGDEBUG_H
12 #define LOGDEBUG_H
13
14 /* We use a subset of the standard syslog levels for classifying debug
15  * output here.
16  */
17 #define LVL_EROR 3 /* error messages */
18 #define LVL_WARN 4 /* warnings */
19 #define LVL_NOTE 5 /* normal but notable information */
```

```
#define LVL_INFO 6 /* things which may be interesting to a user but
                      don't require action */
#define LVL_DBUG 7 /* debugging messages */


void dbg(int level, String msg);

void dbg(String msg);

void serialDebug(void);

#endif
```

Listing B.5: Setup-Funktionen für das ESP-Modul in wlansetup.ino

```
/*
 * Functions for setting up the ESP8266 WLAN module.
 *
 * (C) 2016 General Overnight <generalovernight.wordpress.com>
 *
 * This is Free Software available under the GNU General Public License (GPL).
 *
 * $Id: wlansetup.ino,v 1.1 2016-01-31 08:55:31 manolo Exp $
 */

int espConfig()
{
  int rc = 0;

  /* increase the timeout for resetting the ESP */
  esp8266.setTimeout(DEFTO * 3);
  /* send a reset command to the ESP8266 and wait for the string "ready" */
  rc |= runATcmd("+RST", "ready");

  /* configure as a WLAN access point */
  if (!configAP(SSID, PSK)) {
    esp8266.setTimeout(DEFTO);
    dbg(LVL_NOTE, "WiFi Access point successfully set up with config:");
    dbg(LVL_NOTE, runATcmd("+CIFSR", 3));
  } else {
    rc |= 0x2;
  }

  rc |= (runATcmd("+CIPMODE=0", "OK")) << 3;
  rc |= (runATcmd("+CIPMUX=0", "OK")) << 2;
  esp8266.setTimeout(DEFTO);

  /* finally light up the onboard LED if everything went OK */
  if (0 == rc) {
    digitalWrite(LED_BUILTIN, HIGH);
  } else {
    dbg("ESP config error: " + String(rc));
  }

```

```
40   return rc;
41 }
42
43
44 int configAP(String ssid, String psk)
45 {
46   int rc = 0;
47
48   /* increase the timeout, AP setup commands may take significantly more
49    * time to finish.
50    */
51   esp8266.setTimeout(DEFTO * 2);
52   rc |= (runATcmd("+CWMODE=2", "OK"));
53   /* Set up WiFi AP with parameters (in RAM only):
54    *   SSID, PSK, channel number (1-14), encryption index (0-4)
55    */
56   rc |= (runATcmd("+CWSAP_CUR=\""+String(ssid)+"\",\""+String(psk)+"\",1,3",
57                   "OK"));
58   /* Set the IP address for the AP (in RAM only) */
59   rc |= (runATcmd("+CIPAP_CUR=\"" SRCIP "\"", "OK"));
60   /* Reset the timeout value */
61   esp8266.setTimeout(DEFTO);
62
63   return rc;
64 }
65
66 int configUDP(String port)
67 {
68   int rc = 0;
69
70   esp8266.setTimeout(DEFTO + DEFTO / 2);
71   rc |= !runATcmd("+CIPMODE=0", "OK");
72   rc |= !runATcmd("+CIPMUX=0", "OK");
73   /* Start an UDP Broadcast server */
74   rc |= !runATcmd("+CIPSTART=\"UDP\",\"" DSTIP "\"," + port, "OK");
75   esp8266.setTimeout(DEFTO);
76
77   return rc;
78 }
79
80
81 // Local Variables:
82 // mode: c++
83 // End:
```

Listing B.6: Headerdatei für wlansetup.ino

```
1 /*
2  * Functions for setting up the ESP8266 WLAN module.
3  *
4  * (C) 2016 General Overnight <generalovernight.wordpress.com>
5  *
6  * This is Free Software available under the GNU General Public License (GPL).
7  *
```

```
 * $Id: wlansetup.h,v 1.1 2016-01-31 08:55:31 manolo Exp $
 */

#ifndef WLANSETUP_H
#define WLANSETUP_H

int espConfig();

int configAP(String ssid, String psk);

int configUDP(String port);

#endif

// Local Variables:
// mode: c++
// End:
```

Listing B.7: Funktionen für serielle Kommunikation zwischen MCU und WLAN-Modul serialcomm.ino

```
/*
 * Functions pertaining to serial I/O between the Atmega MCU and the ESP.
 *
 * (C) 2016 General Overnight <generalovernight.wordpress.com>
 *
 * This is Free Software available under the GNU General Public License (GPL).
 *
 * $Id: serialcomm.ino,v 1.1 2016-01-31 08:55:30 manolo Exp $
 */

#include "serialcomm.h"


boolean sendUDP(String addr, String port, String data)
{
  boolean rc = 0;

  esp8266.setTimeout(DEFTO);
  rc = runATcmd("+CIPSEND=" + String(data.length() + 2) + "," + addr + "," + port,
                data, "OK");

  return rc;
}


/*
 * Send the specified data via UDP.
 *
 * Returns true (1) on error.
 */
boolean sendUDP(String data)
{
```

```
33    boolean rc = 0;
34
35    esp8266.setTimeout(DEFTO);
36    rc = runATcmd("+CIPSEND=" + String(data.length() + 2), data, "OK");
37
38    return rc;
39  }
40
41
42  /*
43   * Run the specified AT command, returning the response
44   */
45  String runATcmd(String command)
46  {
47    String resp = "", cmd = "AT" + command;
48    int start;
49
50    esp8266.println(cmd);
51    resp = esp8266.readString();
52    dbg(LVL_DBUG, "# " + cmd + " = " + resp);
53    while (-1 < (start = resp.indexOf(command))) {
54      resp.remove(start, 1 + command.length());
55    }
56
57    return(resp);
58  }
59
60
61  /*
62   * Run the specified AT command, returning the response but remove at
63   * most stripLevel instances of the command name beforehand.
64   */
65  String runATcmd(String command, int stripLevel)
66  {
67    String resp = "", cmd = "AT" + command;
68    int start;
69
70    esp8266.println(cmd);
71    resp = esp8266.readString();
72    dbg(LVL_DBUG, "# " + cmd + " = " + resp);
73    while (stripLevel--> 0 && -1 < (start = resp.indexOf(command))) {
74      resp.remove(start, 1 + command.length());
75    }
76
77    return(resp);
78  }
79
80
81  /*
82   * Run specified AT command, waiting for an expected response string.
83   *
84   * Returns true (1) on error
85   */
```

```
86  boolean runATcmd(String command, char expect[])
87  {
88    return(runATcmd(command, "", expect));
89  }
90
91
92  /*
93   * Run specified AT command with optional data to send.
94   *
95   * Returns true (1) on error
96   */
97  boolean runATcmd(String command, String data, char expect[])
98  {
99    boolean rc = false, found = false, withdata = boolean(data.length());
100   String response = "", cmd = "AT" + command;
101
102   esp8266.println(cmd);
103
104   while (!(found || rc)) {
105     response = esp8266.readStringUntil('\n');
106     if (1 + response.indexOf("ERROR")) {
107       rc = true;
108     } else if (1 + response.indexOf(withdata ? ">" : expect)) {
109       found = true;
110     }
111     dbg(">>> " + response);
112   }
113
114   if (withdata) {
115     dbg("... sending data: '" + data + "'");
116     esp8266.println(data);
117     rc |= !esp8266.findUntil(expect, "ERROR");
118   }
119
120   dbg(LVL_DBUG, "# AT+" + cmd + " = " + response);
121
122   if (rc)
123   {
124     dbg(LVL_EROR, "Error sending command: " + cmd + ": " + response);
125   } else {
126     dbg(LVL_INFO, "Command successful: " + cmd);
127   }
128   return(rc);
129 }
130
131
132 /*
133  * Returns:
134  *   false: when no data were available,
135  *   true: when data were exchanged
136  */
137 boolean serialIO()
138 {
```

```
139   boolean rc = false;
140   unsigned long time;
141
142   /* Read data from the ESP if available, but don't lock out other
143    * program parts and stop once data were read.
144    */
145   while (!rc && esp8266.available()) {
146     Serial.write(esp8266.read());
147     rc = true;
148   }
149
150   while (!rc && Serial.available()) {
151     esp8266.write(Serial.read());
152     rc = true;
153   }
154
155   return(rc);
156 }
157
158
159 // Local Variables:
160 // mode: c++
161 // End:
```

Listing B.8: Headerdatei für serialcomm.ino

```
1  /*
2   * Functions pertaining to serial I/O between the Atmega MCU and the ESP.
3   *
4   * (C) 2016 General Overnight <generalovernight.wordpress.com>
5   *
6   * This is Free Software available under the GNU General Public License (GPL).
7   *
8   * $Id: serialcomm.h,v 1.1 2016-01-31 08:55:30 manolo Exp $
9   */
10
11 #ifndef SERIALCOMM_H
12 #define SERIALCOMM_H
13
14 boolean sendUDP(String addr, String port, String data);
15
16 boolean sendUDP(String data);
17
18 String runATcmd(String command);
19
20 String runATcmd(String command, int stripLevel);
21
22 boolean runATcmd(String command, char expect[]);
23
24 boolean runATcmd(String command, String data, char expect[]);
25
26 boolean serialIO(void);
27
28
```

```
29 #endif
```

Anhang C

Verwendete Bauteile

Der IoTeaTimer setzt sich aus folgenden Teilen zusammen, die im IoT- oder im Elektronik-Adventskalender 2015 enthalten waren.

1. Breadboard Syb-46
2. nanoESP (Pretzelboard)
3. ca. 15 mm Kupferdraht (isoliert)
4. 3 Widerstände 1kΩ
5. 2 Taster
6. RGB-LED
7. Piezo-Summer

Um die Tonausgabe des Summers und die Handhabbarkeit des Aufbaus zu verbessern wird außerdem eine aus Sperrholz (Länge 120 mm, Breite ca. 50–60 mm, Stärke ca. 10 mm) benötigt, die durch weitere Brettchen zu einem geschlossenen Gehäuse ergänzt werden kann.