fhwedel
UNIVERSITY OF APPLIED SCIENCES

Projekt Microkontroller

Sensorgestützte LED-Kerze
Dokumentation

Guillaume Fournier-Mayer
Tinf-101922
Fachsemester 7
Verwaltungssemester 10

Wedel, den 1. September 2020

Inhaltsverzeichnis

Teil I
Problemstellung

In dem Projekt Mikrocontroller soll mit einem Mikrocontroller eine sensorgesteuerte LED-Kerze entwickelt werden die sich in zwei Betriebsmodi betreiben lässt. Dabei wird zwischen dem *Laternenmodus* und dem *Dekorationsmodus* unterschieden. Im ersteren soll die Kerze anfangen zu leuchten sobald sie Bewegt wird. Wurde keine Bewegung mehr detektiert, schaltet sich die Kerze nach Ablauf einer konfigurierten Abschaltverzögerung ab. Im zweiten Modus leuchtet die Kerze solange bis die Abschaltverzögerung abgelaufen ist.
Dabei soll die Kerze über eine Bewegungssteuerung und nicht über herkömmliche Knöpfe konfiguriert werden können. Die Konfigurationsmöglichkeiten beinhalten Abschaltverzögerung, Helligkeit und Farbe. Da die Kerze über einen Akku betrieben wird, soll zusätzlich dafür gesorgt werden, dass sie über Stromsparmaßnahmen verfügt.

Teil II
Benutzerhandbuch

1 Betriebsmodi

1.1 Laternenmodus

Im Laternenmodus schaltet sich die Kerze an, sobald diese bewegt wird. Detektiert die Kerze keine Bewegung mehr, schaltet sie sich automatisch, nach einer konfigurierten Zeit wieder aus. Um die Kerze manuell auszuschalten, muss sie um 180°, bzw. *über Kopf* gedreht werden.

1.2 Dekorationsmodus

Im Dekorationsmodus wird die Kerze durch Kippen nach hinten eingeschaltet. Nach einer konfigurierten Zeit schaltet sie sich automatisch wieder aus. Um die Kerze manuell auszuschalten, muss sie um 180°, bzw. *über Kopf* gedreht werden.

1.3 Konfigurationmodus

Um in den Konfigurationmodus zu wechseln, wird die Kerze nach vorne gekippt. Nun können alle Parameter der Kerze eingestellt werden (Siehe 2). Ein Wert wird über Links- bzw. Rechtskippen gewechselt. Soll der Wert bestätigt werden, wird die Kerze wieder nach vorne gekippt. Die Kerze fängt an zu blinken und signalisiert damit, dass der Wert eingestellt wurde. Wird die Kerze wieder zurück in ihre Ausgangsposition gestellt, hört sie auf zu blinken und ist bereit den nächsten Wert zu konfigurieren. Nach dem letzten Konfigurationsschritt schaltet sich die Kerze ab und kann nun in einen anderen Modus versetzt werden. Um den Konfigurationmodus zu verlassen, müssen alle Konfigurationsschritte bestätigt werden.

2 Konfiguration

Folgenden Abbildung zeigt das Konfigurationsmenü und seine Unterpunkte.

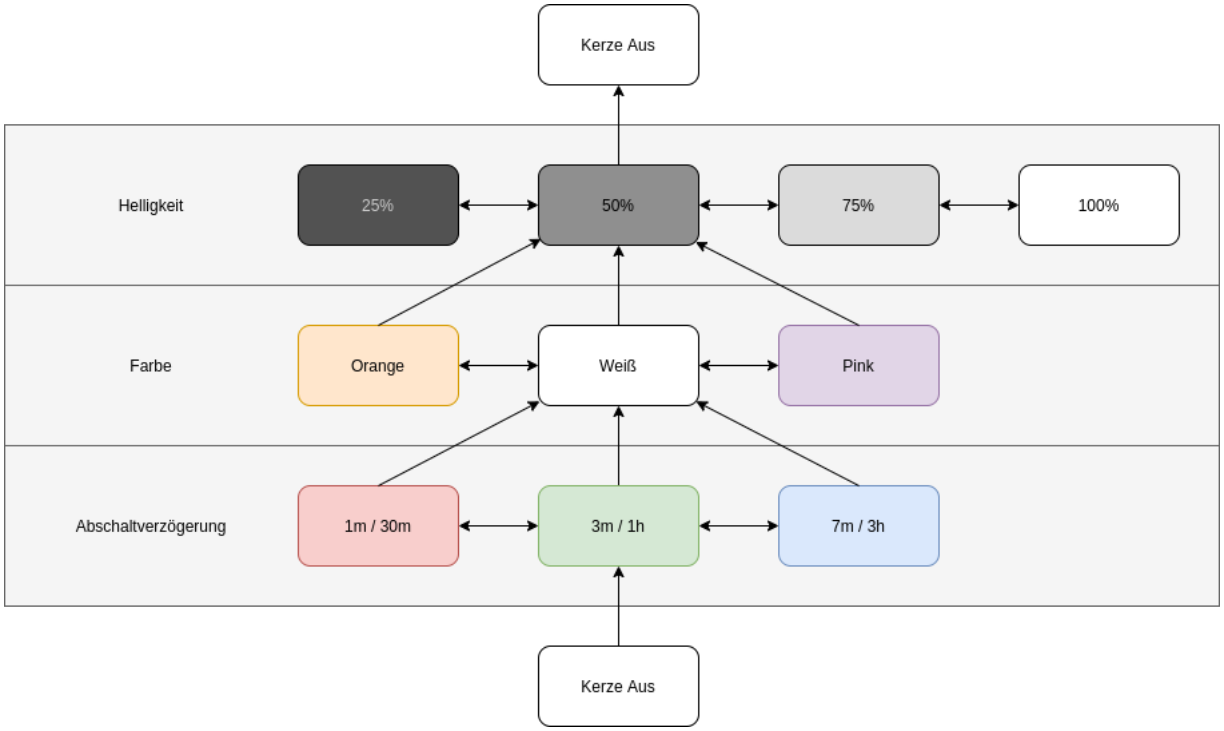

Abbildung 1: Konfigurationsmenü

2.1 Abschaltverzögerung

Die Abschaltverzögerung ist der erste Konfigurationsschritt und stellt drei Verzögerung zur Auswahl. Dabei wird zwischen den Abschaltverzögerung für den Laternenmodus und dem Dekorationsmodus unterschieden. Wird zum Beispiel *Rot* gewählt, schaltet sich die Kerze im Laternenmodus nach einer Minute und im Dekorationsmodus nach 30 Minuten ab. Dabei ist *Rot* die Standardabschaltverzögerung nachdem die Kerze gestartet wurde.

Farbe	Laternenmodus	Dekorationsmodus
Rot	1m	30m
Grün	3m	1h
Blau	7m	3h

2.2 Farbe

Das Einstellen der Kerzenfarbe, ist der zweite Konfigurationsschritt und stellt Weiß, Orange und Pink zur Auswahl. Orange ist die Standardfarbe nachdem die Kerze gestartet wurde.

2.3 Helligkeit

Das Einstellen der kerzen Helligkeit ist der dritte und letzte Konfigurationsschritt. Dabei werden Helligkeiten von 25%, 50%, 75% und 100% zur Auswahl gestellt. 50% ist die Standardhelligkeit nachdem die Kerze gestartet wurde.

2.4 Zurücksetzen

Um alle Werte wieder auf Werkszustand zurück zu setzen, wird die Kerze für einen kurzen Moment stromlos geschaltet. Nach dem Wiedereinsetzen des Akkus sind die Werte zurückgesetzt.

Teil III
Programmierhandbuch

3 Entwicklungskonfiguration

3.1 Hardware

Zweck	Hardware
Microkontroller	Atmega168PA
Sensor	MPU6050
RGB-LED	HV-5RGBXX
UART	FTDI FT232RL
Programmer	MPLAB SNAP

3.2 Software

Zweck	Software
IDE	MPLAB IDE v5.40

3.3 Inbetriebnahme

3.3.1 Schaltplan

Um die Kerze in Betrieb zu nehmen muss zunächst die Hardware aufgebaut werden. Dazu dient der folgende Schaltplan.

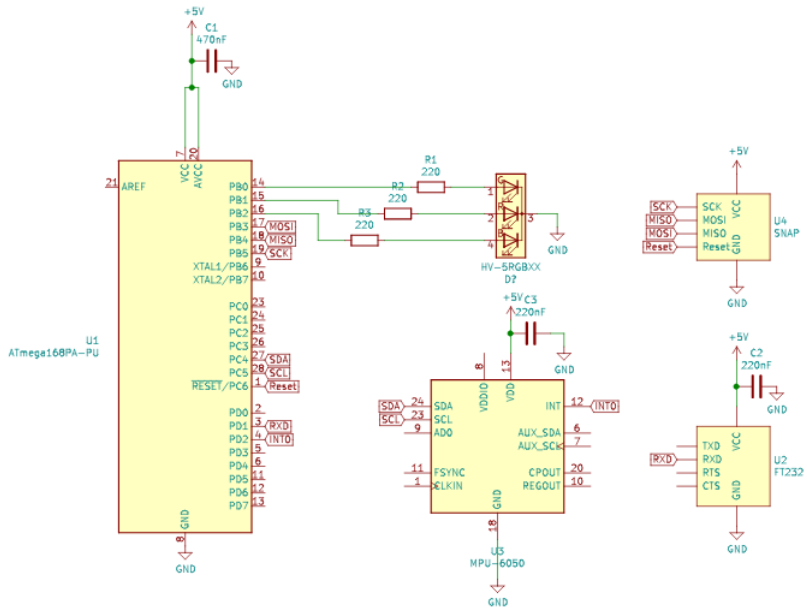

Abbildung 2: Schaltplan

Hierbei ist zu beachten, dass der MPU6050 nicht durch seine Pins im Breakboard befestigt wird, sondern waagerecht aufgeklebt wird (Siehe 5.3.2).

3.3.2 Kompilieren und Hochladen

Sobald die Hardware vollständig aufgebaut wurde, kann der *MPLAB SNAP* Programmierer und das *FT232* Modul, über Mikro-USB Kabel, an einen Computer angeschlossen werden und die *MPLAB IDE* gestartet werden. Als nächstes muss das Projekt mit der *MPLAB IDE* geöffnet werden. Der Programmer sollte erkannt werden und nach einem Klick auf *Run* wird der Code kompiliert und auf die Hardware geladen.

War dies erfolgreich, kann die Kerze wie im Benutzerhandbuch (Siehe II) beschrieben, genutzt werden.

4 Problemanalyse

4.1 Betriebsmodi

4.1.1 Laternenmodus

Sobald die LED Kerze bewegt wird, wird in den Laternenmodus geschaltet werden. Die Kerze leuchtet solange bis keine Bewegung mehr detektiert wurde und die Abschaltverzögerung abgelaufen ist. Es muss jedoch auch manuell möglich sein, die Kerze abzuschalten.

4.1.2 Dekorationsmodus

Die Kerze wird manuell in den Dekorationsmodus geschaltet. In diesem Modus leuchtet die LED durchgehend bis die Abschaltverzögerung die Kerze abschaltet. Wie im Laternenmodus soll es möglich sein, die Kerze manuell auszuschalten.

4.2 Konfiguration

Da die Kerze ausschließlich über Bewegungen konfigurierbar sein soll, wird eine Bewegungssteuerung benötigt. Dabei wird zwischen Bewegungen entlang einer Achse und Bewegungen um eine Achse unterschieden.
Ein Accelerometer kann dabei nur Bewegungen entlang einer Achse, wie zum Beispiel das verschieben eines Körpers, messen. Dagegen kann ein Gyroskop nur Bewegungen um eine Achse, wie zum Beispiel das Rotieren eines Körpers, messen. Durch die Kombination der beiden Sensoren erhält man eine inertiale Messeinheit und kann somit jede Bewegung eines Körpers detektieren. Falls die Startposition sowie Startorientierung bekannt sind, kann damit die Position sowie die Orientierung des Körpers im Raum bestimmt werden. Somit können Bewegungen durch einen Sensor erkannt werden und als Steuerung der Kerze benutzt werden.

4.3 Abschaltverzögerung

Damit es möglich ist, dass die Kerze sich automatisch nach einer konfigurierten Abschaltverzögerung abschaltet, ist es erforderlich, dass die vergangene Zeit gemessen werden kann.

4.4 Helligkeit und Farbe

Über Bewegungssteuerung soll es möglich sein, unter anderem die Helligkeit und Farbe einzustellen. Dafür wird ein puls-weiten-modulations Verfahren benötigt um die einzelnen Farbkanäle der RGB-LED anzusteuern.

4.5 Debugging

Es muss möglich sein das Programm zu Debuggen. Dazu wird ein Modul benötigt, dass Debugausgaben an einem Computer senden kann.

5 Realisation

5.1 Modularisierung

Der Programmcode wurde modulweise erstellt, sodass Themengebiete voneinander abgekapselt sind. Durch geschicktes Einbinden kann somit Codeverdoppelung vermieden werden.

5.1.1 Modulübersicht

twi.h Das *Two Wire Interface (TWI)* Modul stellt Funktionen zu Verfügung um über ein *TWI*-Bus-System zu kommunizieren. Dabei wurde das Modul sehr *Low-Level* gehalten. Das bedeutet unter anderem, dass der Klient dieses Modules selber dafür verantwortlich ist *Start* und *Stop* Konditionen auf den Bus zu setzen. Dabei ist zu beachten, dass dieses Modul blockierend arbeitet.

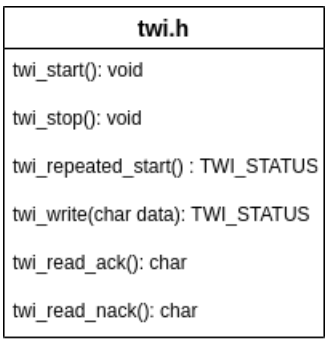

Abbildung 3: twi.h

led.h Das LED Modul stellt Funktionen rund um die LED Steuerung zu Verfügung. Hiermit lässt sich die die LED ein- und ausschalten sowie die Farbe konfigurieren und das Blinkverhalten einstellen.

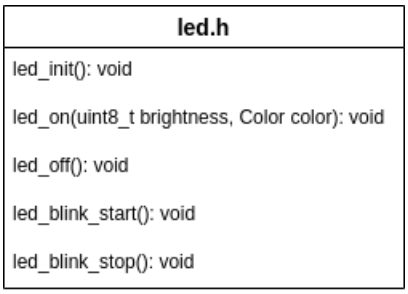

Abbildung 4: led.h

timer.h In dem Timer Modul finden sich Funktionen um den Timer zu steuern, der die Abschaltverzögerung implementiert.

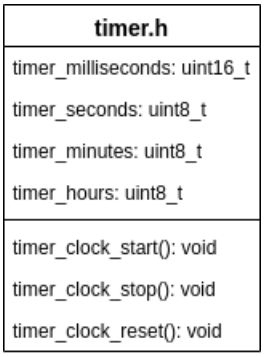

Abbildung 5: timer.h

mpu6050.h Dieses Modul bietet *High-Level* Funktionen um die Daten des Accelerometers zu erhalten sowie Konfigurationsmöglichkeiten.

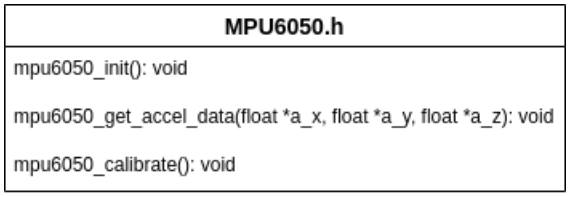

Abbildung 6: mpu6050.h

config.h Das config Modul bietet Konstanten zur Konfiguration der Kerze. So können alle Voreinstellungen bearbeitet werden und an die entsprechenden Anforderungen angepasst werden.

types.h Das types Modul beinhaltet alle Datenstrukturen die für das Programm benötigt werden (Siehe 5.2).

debug.h Das Debug Modul wird zu debug Zwecken benötigt und implementiert somit nur Funktionen um über die *USART* Schnittstelle Daten zu senden, jedoch nicht um Daten zu empfangen. Die Daten können dann von einer seriellen Schnittstelle an einem Computer betrachtet werden. Das Modul arbeitet hierbei blockierend.

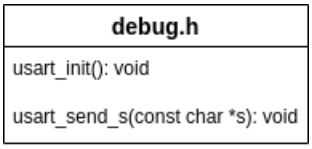

Abbildung 7: debug.h

5.1.2 Programmorganisationsplan

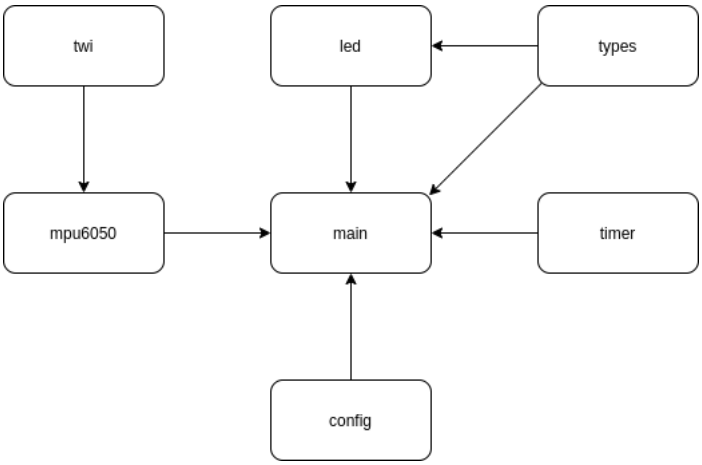

Abbildung 8: Programmorganisationsplan

5.2 Datenstrukturen

Alle Datenstrukturen die für das Programm benötigt werden, befinden sich in der *types.h* Datei.

5.2.1 State

Die *State* Datenstruktur repräsentiert den aktuellen Zustand der Zustandsmaschine (Siehe 5.7.1).

Listing 1: State Enum

```
1  /**
2   * Holds all the states of the statemachine.
3   */
4  typedef enum {
5      LED_OFF,
6      LED_ON_LANTERN,
7      LED_ON_DECORATION,
8      SETUP_DELAY,
9      SETUP_COLOR,
10     SETUP_BRIGHTNESS
11 } State;
```

5.2.2 Delay

Die *Delay* Datenstruktur beschreibt die Abschaltverzögerung und speichert die Minuten sowie die Stunden nachdem die Kerze sich abschalten soll.

Listing 2: Delay Struct

```
1  /**
2   * Represents the delay by storing minutes and hours.
3   */
4  typedef struct {
5      uint8_t minutes;
6      uint8_t hours;
7  } Delay;
```

5.2.3 Color

Die *Color* Datenstruktur steht für die LED Farbe und speichert die drei Farbkanäle Rot, Grün und Blau. Der Maximalwert eines Farbkanals ist 100.

Listing 3: Color Struct

```
1  /**
2   * Represents a RGB color.
3   */
```

```
4 typedef struct {
5     uint8_t red;
6     uint8_t green;
7     uint8_t blue;
8 } Color;
```

5.2.4 Config

Die *Config* Datenstruktur repräsentiert alle Konfigurationsparameter der Kerze und ist aus den anderen oben genannten Datenstrukturen kompositioniert. Dabei wird zwischen der Laternen- und der Dekorationsabschaltverzögerung unterschieden. Zusätzlich hält die Datenstruktur noch die Helligkeit deren Maximalwert auch hier 100 beträgt.

Listing 4: Config Struct

```
1 /**
2  * Holds all needed config data.
3  */
4 typedef struct {
5     uint8_t brightness;
6     Color color;
7     Delay lantern_delay;
8     Delay deco_delay;
9 } Config;
```

5.2.5 setup_colors

Das *setup_colors* Array speichert die Farbe die die Kerze im *SETUP_DELAY* Zustand annehmen soll.

Listing 5: setup_colors Array

```
1 /**
2  * An Array that holds all the setup colors presets
3  */
4 const Color setup_colors[SETUP_MAX_INDEX + 1] = {
5     {.red = DELAY_1_COLOR_RED, .green = DELAY_1_COLOR_GREEN, .blue =
          DELAY_1_COLOR_BLUE},
6     {.red = DELAY_2_COLOR_RED, .green = DELAY_2_COLOR_GREEN, .blue =
          DELAY_2_COLOR_BLUE},
7     {.red = DELAY_3_COLOR_RED, .green = DELAY_3_COLOR_GREEN, .blue =
          DELAY_3_COLOR_BLUE},
8 };
```

5.2.6 lantern_delays

Das *lantern_delays* Array speichert alle Abschaltverzögerungen für den Laternenmodus zwischen denen der Benutzer wählen kann.

Listing 6: deco_delays Array

```
/**
 * An Array that holds all the lantern switch-off delays
 */
const Delay lantern_delays[SETUP_MAX_INDEX + 1] = {
    {.minutes = DELAY_WALK_1_MINUTES, .hours = DELAY_WALK_1_HOURS},
    {.minutes = DELAY_WALK_2_MINUTES, .hours = DELAY_WALK_2_HOURS},
    {.minutes = DELAY_WALK_3_MINUTES, .hours = DELAY_WALK_3_HOURS},
};
```

5.2.7 deco_delays

Das *deco_delays* Array speichert alle Abschaltverzögerungen für den Dekorationsmodus zwischen denen der Benutzer wählen kann.

Listing 7: deco_delays Array

```
/**
 * An Array that holds all the decoration switch-off delays
 */
const Delay deco_delays[SETUP_MAX_INDEX + 1] = {
    {.minutes = DELAY_DECO_1_MINUTES, .hours = DELAY_DECO_1_HOURS},
    {.minutes = DELAY_DECO_2_MINUTES, .hours = DELAY_DECO_2_HOURS},
    {.minutes = DELAY_DECO_3_MINUTES, .hours = DELAY_DECO_3_HOURS},
};
```

5.2.8 led_colors

Das *led_colors* Array speichert alle Farben zwischen denen der Benutzer wählen kann.

Listing 8: led_colors Array

```
/**
 * An Array that holds all the led color presets
 */
const Color led_colors[SETUP_MAX_INDEX + 1] = {
    {.red = COLOR_1_RED, .green = COLOR_1_GREEN, .blue = COLOR_1_BLUE},
    {.red = COLOR_2_RED, .green = COLOR_2_GREEN, .blue = COLOR_2_BLUE},
    {.red = COLOR_3_RED, .green = COLOR_3_GREEN, .blue = COLOR_3_BLUE},
};
```

5.3 Bewegungssteuerung

Zum detektieren von Bewegungen wird ein *MPU6050* eingesetzt. Dieser Sensor beinhaltet ein drei-Achsen Gyroskop, ein drei-Achsen Accelerometer und ein Temperatursensor.

5.3.1 Sensorwahl

Da die konstante Erdbeschleunigung von $1g = 9,81\frac{m}{s}$ die Kerze kontinuierlich in Richtung Erdmittelpunkt beschleunigt, reicht das drei-Achsen Accelerometer aus, um die genaue Neigung der Kerze zu bestimmen. Zu beachten ist jedoch, dass nur Neigungen um die X- und Y-Achsen detektiert werden können. Rotationen um die Z-Achse werden von dem Accelerometer nicht wahrgenommen da die Z-Achse auf den beiden anderen Achsen steht.

Alternativ könnte auch das Gyroskop verwendet werden. Darüber lassen sich Drehwinkel um die Achsen messen. Da jedoch ein Gyroskop $\frac{Winkel}{Zeit}$ als Einheit hat, müsste integriert und die Werte aufsummiert werden. Die wahrscheinlichkeit, dass der Sensorwert dabei driftet ist sehr hoch und wird somit nicht als Lösung betrachtet.

5.3.2 Achsenorentierung

Zu beachten ist, dass der Sensor nicht über seine Pins im Breakout Board befestigt wird sondern flach auf dem Board festgeklebt wird. Daraus ergibt sich folgende Orientierung der Achsen.

Z-Achse Senkrechte Achse

Y-Achse Horizontale Achse

X-Achse Senkrecht auf beiden anderen Achsen

5.3.3 Kommunikation

Die Sensordaten werden über ein *TWI* Bus an den Mikrocontroller übertragen.

Senden Als Beispiel eines Sendevorgangs wird die Konfiguration des Accelerometers benutzt.

Listing 9: TWI Sendevorgang

```
1 void mpu6050_init_accel()
2 {
3   TWI_STATUS status;
4   // Configure Accelerometer to +-2g
5   debug_print("MPU Init: Configure Accelerometer\r\n");
6   twi_start(MPU_WRITE_ADDRESS);
7   status = twi_write(ACCEL_CONFIG);
8   if (status != TWI_ACK)
9     debug_print("MPU Init-Error: Set accelconfig address\r\n");
```

```
10    status = twi_write(0x00);
11    if (status != TWI_ACK)
12      debug_print("MPU Init-Error: Set accelconfig value\r\n");
13    twi_stop();
14  }
```

Zunächst wird das Startsignal auf den Bus gelegt und gewartet bis der Bus frei ist. Sobald die Kondition eintritt, wird die Schreibadresse des *MPU6050* gesetzt. Als nächstes wird die Registeradresse in die die Daten geschrieben werden sollen auf den Bus geschrieben und auf das *ACK* gewartet. War dies erfolgreich, können die eigentlichen Daten auf den Bus geschrieben werden und das Stoppsignal gesendet werden.

Empfangen Als Beispiel für das Empfangen von Daten wird das auslesen des Accelerometers benutzt.

Listing 10: TWI Empfangsvorgang

```
1   int16_t mpu6050_get_data(uint8_t reg)
2   {
3
4     debug_print("MPU Getdata\r\n");
5     TWI_STATUS status;
6
7     twi_start(MPU_WRITE_ADDRESS);
8     status = twi_write(reg);
9     if (status != TWI_ACK)
10      debug_print("MPU Getdata-Error: Set getdata address\r\n");
11    status = twi_repeated_start(MPU_READ_ADDRESS);
12    if (status != TWI_ACK)
13      debug_print("MPU Getdata-Error: Set repeated start address\r\n");
14
15    char data_msb = twi_read_ack();
16    char data_lsb = twi_read_nack();
17    twi_stop();
18
19    return ((int16_t)data_msb << 8) | data_lsb;
20  }
```

Zunächst wird wie beim Senden das Startsignal auf den Bus gelegt, die Schreibadresse des *MPU6050* gesetzt und das zu lesenden Register gesendet. Nun erfolgt ein wiederholter Start mit der Leseadresse des *MPU6050*. Durch `twi_read_ack` wird das erste Byte ausgelesen. Der *MPU6050* erhöht nach dem Empfangen des *ACK* automatisch die Registeradresse, sodass mit `twi_read_nack` das zweite Byte ausgelesen werden kann. Nach dem zweiten Byte wird jedoch ein *NACK* gesendet, welches dem *MPU6050* signalisiert, dass keine Daten mehr erwartet werden. Nach dem Stoppsignal ist die Kommunikation beendet.

5.3.4 Kalibrierung

Durch die Kalibrierung wird der *null Fehler* kompensiert und damit gewährleistet, dass der Sensor kalibrierte Daten liefert.
Ein *null Fehler* entsteht dadurch, dass ein Sensor einen Wert liefert der ungleich null ist, obwohl der Sensor sich in Ruhelage befindet. Da der Fehler sich selbst bei baugleichen Sensoren unterscheidet, muss für jeden Sensor eine einmalige Kalibrierung erfolgen.

Über die `mpu6050_calibrate` Funktion werden die Werte gemittelt und dann über die `debug_print` Funktion ausgegeben. Dazu wird zunächst der Sensor waagerecht auf eine plane Fläche gelegt, sodass die Z-Achse nach oben Zeigt und die Funktion ausgeführt. Nun wir der Z-Achsenwert notiert, das Verfahren für alle Achsen wiederholt und der entsprechende Achsenwert notiert.
Um den eigentlichen Versatz zu berechnen werden folgende Formeln verwendet.

$$Versatz_X = 0 - Achsenwert_X$$

$$Versatz_Y = 0 - Achsenwert_Y$$

$$Versatz_Z = 1 - Achsenwert_Z$$

Für den benutzen Sensor ergeben sich somit folgende Versätze.

Achse	Versatz
X	−0.035939
Y	0.010786
Z	0.187407

Der Versatz muss nun auf jeden entsprechenden Wert hinzuaddiert werden um den *null Fehler* zu kompensieren.

Listing 11: Nullfehler Kompension

```
1  *a_x = mpu6050_get_data(MPU_ACCEL_X) / ACCEL_SCALE_FACTOR + A_X_OFFSET;
2  *a_y = mpu6050_get_data(MPU_ACCEL_Y) / ACCEL_SCALE_FACTOR + A_Y_OFFSET;
3  *a_z = mpu6050_get_data(MPU_ACCEL_Z) / ACCEL_SCALE_FACTOR + A_Z_OFFSET;
```

5.3.5 Digitale Tiefpassfilterung

Um dem Rauschen des Sensors entgegen zu wirken, wird die interne digitale Tiefpassfilterung des *MPU6050* benutzt. Dabei wird der Tiefpassfilter auf $5Hz$ gestellt um hochfrequentes Rauschen zu filtern.

5.4 Abschaltverzögerung

Um die Abschaltverzögerung zu realisieren wird der Hardware Timer *Timer0* im CTC-Modus des *Atmega168* benutzt. Dieser soll jede Millisekunde eine *Interrupt Service Routine (ISR)* ausführen die Millisekunden, Sekunden und Stunden verwaltet.

Listing 12: Abschaltverzögerung

```
1  ISR(TIMER0_COMPA_vect) {
2      timer_milliseconds++;
3      if (timer_milliseconds == 1000) {
4          timer_milliseconds = 0;
5          timer_seconds++;
6          if (timer_seconds == 60) {
7              timer_seconds = 0;
8              timer_minutes++;
9              if (timer_minutes == 60) {
10                 timer_hours++;
11                 if (timer_hours == 24) {
12                     timer_hours = 0;
13                 }
14             }
15         }
16     }
17 }
```

Um das Vergleichsregister einzustellen, sodass die *ISR* jede Millisekunde ausgeführt wird, wird folgende Formel benutzt.

$$OCR0A = \frac{CPU_{freq}}{Vorteiler \cdot Timer_{freq}}$$

Da die CPU Frequenz bei 1MHz liegt, für den Vorteiler acht gewählt wurde und die gewünschte Timer Frequenz bei 1000Hz liegt, ergibt sich 125 für das vergleichs Register.

$$125 = \frac{1MHz}{8 \cdot 1Khz}$$

Da jedoch keine Millisekunden genaue Auflösung benötigt wird, könnte auch ein anderer Wert für das Vergleichsregister und den Vorteiler gewählt werden, sodass die *ISR* weniger oft ausgeführt wird und somit die CPU Auslastung geringer wird. Beispielsweise würde das Wählen des Vergleichsregisters von 250 dazu führen, dass die *ISR* nur noch mit einer Frequenz von 500Hz ausgeführt wird. Jedoch müsste dann `timer_milliseconds` um zwei inkrementiert werden.

5.5 LED

Als LED kommt eine *HV-5RGBXX* zum Einsatz. Diese wird über drei Pins mit jeweiligen 220Ohm Widerstand an den *Atmega168* angeschlossen (Siehe 3.3.1).

5.5.1 Farbe und Helligkeit

Durch die drei Pins die die Grundfarben Rot, Grün und Blau repräsentieren, lassen sich über additive Farbmischung unterschiedliche Farben einstellen.
Um Intensitäten einstellen zu können wird ein puls-weiten-modulations (PWM) Signal pro Pin erzeugt. Hierfür wird der Hardware Timer *TIMER2* des *Atmega168* verwendet. Da der Timer jedoch nur zwei Ausgänge bestitz, kann kein Hardware PWM Signal, wie zum Beispiel *Fast PWM* verwendet werden, sodass ein *Soft-PWM* implementiert wurde.

Listing 13: led_on

```
1  void led_on(uint8_t brightness, Color color) {
2
3      //Calculate the pulse with for all color channles
4      pulse_width_red = (uint8_t) ((brightness * color.red) / (COLOR_MAX *
           PWM_PERIOD));
5      pulse_width_green = (uint8_t) ((brightness * color.green) / (COLOR_MAX
           * PWM_PERIOD));
6      pulse_width_blue = (uint8_t) ((brightness * color.blue) / (COLOR_MAX *
           PWM_PERIOD));
7
8      // CTC Modus
9      TCCR2A = (1 << WGM21);
10     // 256 Prescaler
11     TCCR2B = (1 << CS22) | (1 << CS21);
12     // Set compare value
13     OCR2A = 1;
14     // Enable Compare Interrupt
15     TIMSK2 |= (1 << OCIE2A);
16 }
```

Um die LED einzuschalten wird die `led_on` Funktion mit den jeweiligen Parametern aufgerufen. Zunächst werden die Pulsweiten für die einzelnen Farbkanäle berechnet. Diese sagen aus, wie lange der Farbkanal pro Periode eingeschaltet ist. In der Formel wird direkt die Helligkeit mit der Farbintensität verrechnet.

$$Pulsweite = \frac{Helligkeit \cdot Farbintensität}{Farbintensität_{max} \cdot Periodenschritte_{max}}$$

Des weiteren wird der Timer gestartet, sodass dieser mit einer Frequenz von $3906Hz$ bzw. alle $256\mu s$ die *ISR* ausführt.

Listing 14: PWM ISR

```
/**
 * Turns the color channels of the led on and off to emulate a PWM signal
 */
ISR(TIMER2_COMPA_vect) {

    static uint8_t pwm_step = 0;

    // Turn the led off if blinkstate is false
    if (!led_blink_state) {
        PORTB &= ~((1 << PORTB0) | (1 << PORTB1) | (1 << PORTB2));
    } else {
        // Check if Green channel should be turned on or off
        if (pwm_step < pulse_width_green) {
            PORTB |= (1 << PORTB0);
        } else {
            PORTB &= ~(1 << PORTB0);
        }

        // Check if red channel should be turned on or off
        if (pwm_step < pulse_width_red) {
            PORTB |= (1 << PORTB1);
        } else {
            PORTB &= ~(1 << PORTB1);
        }

        // Check if blue channel should be turned on or off
        if (pwm_step < pulse_width_blue) {
            PORTB |= (1 << PORTB2);
        } else {
            PORTB &= ~(1 << PORTB2);
        }

        // Increment the PWM Step
        pwm_step++;

        // Reset the pwm counter if needed
        if (pwm_step >= PWM_PERIOD) {
            pwm_step = 0;
        }
    }
}
```

In dieser *ISR* wird der Zähler `pwm_step` pro Aufruf inkrementiert und mit den jeweiligen Pulsweiten der Farbkanäle verglichen. Liegt der Zähler unterhalb einer Pulsweite,

wird der entsprechende Farbkanal eingeschaltet. Umgekehrt wird der Farbkanal ausgeschaltet sobald der Zähler oberhalb der Pulsweite liegt. Sobald der Zähler die maximalen Periodenschritte erreicht hat, wird dieser auf null zurückgesetzt. Als maximal Periodenschritte wurde zehn gewählt. Daraus folgt eine Periodendauer von ca. 2ms, welches zu schnell für das menschliche Auge ist und somit als kontinuierliches Licht wahrgenommen wird.

Auch wenn die *TIMER2_COMPA_vect ISR* nicht rechenintensiv ist, könnte sie verkürzt werden, sodass die CPU weniger blockiert. Dazu könnte die Logik in die Hauptschleife ausgelagert werden und `OCR2A` auf die maximale Periodenschritte gesetzt werden. Ein vergleich des Periodenschrittes mit `TCNT2` wäre dann äquivalent. Der Vorteil wäre hierbei, dass die zeitlich kürzere *ISR* die CPU weniger blockiert. Ein jedoch großer Nachteil könnte sein, dass die Farben bei hoher CPU Auslastung driften, da nicht genug CPU Zeit in der Hauptschleife vorhanden ist um die PWM Signale stabil zu halten.
Ein weiterer Ansatz wäre es, zwei hardware Timer zu benutzen und *Fast-PWM* einzusetzen. Dabei würde zum Beispiel ein Timer zwei Farbkanäle und der andere einen Farbkanal steuern. Der Vorteil an dieser Lösung wäre, dass weniger Overhead durch das aufrufen der *ISR* auftreten würde. Jedoch werden auch zwei der drei hardware Timer des *Atmega168* dafür beansprucht.

5.5.2 Blinken

Um ein Blinken der LED zu realisieren wird der hardware Timer *TIMER1* eingesetzt und so konfiguriert, dass seine *TIMER1_COMPA_vect ISR* ca. mit 4Hz, also vier mal in einer Sekunde blinkt.

Listing 15: Blink ISR

```
1 /**
2  * Toogles the blink state
3  */
4 ISR(TIMER1_COMPA_vect) {
5     led_blink_state = !led_blink_state;
6 }
```

Dabei toggelt die *ISR* das `led_blink_state` Flag und manipuliert somit die *TIMER2_COMPA_vect ISR*, sodass diese die LED komplett ausschaltet wenn das Flag nicht gesetzt ist.

Listing 16: Blinkbedingung

```
1     // Turn the led off if blinkstate is false
2     if (!led_blink_state) {
3         PORTB &= ~((1 << PORTB0) | (1 << PORTB1) | (1 << PORTB2));
4     } else {
```

5.6 Stromsparmaßnahmen

Da die LED Kerze über einen Akku betrieben werden soll, soll möglichst viel Strom gespart werden. Dabei muss abgewogen werden, was wirklich Sinnvoll ist und was in Relation zu anderen großen Stromverbrauchern wie der LED nur marginale Verbesserungen mit sich bringt.

5.6.1 MPU6050

Da von dem Sensor nur das Accelerometer verwendet wird, werden das Gyroskop sowie der Temperatursensor deaktiviert. Zusätzlich wird die Samplerate auf 32Hz runter gesetzt.

5.6.2 Atmega168

Sobald die LED nicht leuchtet, die Zustandsmaschine sich somit im *LED_OFF* Zustand befindet, wird der Mikrocontroller schlafen gelegt. Dabei wird dieser in den *Standbymode* versetzt, sodass die Stromaufnahe von ca. $8.5mA$ auf ca. $2.5mA$ des Gesamtsystems zurück geht. Dies ist eine Reduktion von 70% und wird die Akkulaufzeit verlängern. Dies geschieht über den Aufruf der `sleep` Funktion.

Listing 17: sleep

```
1  /**
2   * Sets the Cpu to sleep while eliminating any race conditions
3   */
4  void sleep() {
5      cli();
6      EIMSK |= (1 << INT0);
7      sleep_enable();
8      sei();
9      sleep_cpu();
10     sleep_disable();
11 }
```

Dabei werden zunächst alle Interrupts gesperrt, sodass es zu keinen *Race Conditions* kommen kann. Ein Beispiel für solch eine *Race Condition* wäre, dass die *ISR* die externen interrupts deaktiviert, die CPU sich aber im nächsten Moment schlafen legt und somit nicht wieder aufwachen kann.
Als nächstes wird das externe Interrupt maskiert, sodass der Mikrocontroller nur aufwacht, sobald am *INT0* Pin eine steigende Flanke anliegt. Durch das Ausführen der `sei` Funktion werden alle Interrupts wieder aktiviert. Zusätzlich wird der nächste Funktionsaufruf unterbrechungsfrei garantiert. Somit kann es zu keinen *Race Conditions* mehr kommen und der Mikrocontroller kann durch `sleep_cpu` schlafen gelegt werden.

Listing 18: Aufwach ISR

```
1  /**
2   * The external interrupt wakeup ISR
3   */
```

```
4 ISR(INT0_vect) {
5     //Deactivate INT0 external interrupt
6     EIMSK &= ~(1 << INT0);
7 }
```

Der *Standbymode* wurde dabei ausgewählt, da aus diesem der Mikrocontroller in nur sechs Taktzyklen wieder aufwachen kann. Das Aufwachen passiert dabei über das *INT0* Interrupt welches durch den *MPU6050* ausgelöst wird sobald diesem neue Daten zu Verfügung stehen. Nach dem Aufwachen wird der Code nach `sleep_disbale` weiter ausgeführt.

5.7 Hauptschleife

Die Hauptschleife ist der Kern des Programmes und verwaltet alle Variablen.

5.7.1 Zustandsmaschine

Um die verschiedenen Zustände der Kerze zu verwalten, wird eine Zustandsmaschine verwendet. Diese beinhaltet folgende Zustände, wobei die Kerze im *LED_OFF* Zustand startet. Bei jedem Zustandswechsel wird das `lock` Flag aktiviert und blockiert solange die Zustandsmaschine bis die Kerze ihre Ausgangsposition wieder eingenommen hat. Somit wird verhindert, dass eine Bewegung der Kerze mehrere Zustände überspringt. Die eigentliche Zustandsmaschine besteht aus dem `State` Enum sowie der *Switch-Anweisung* in der Hauptschleife.

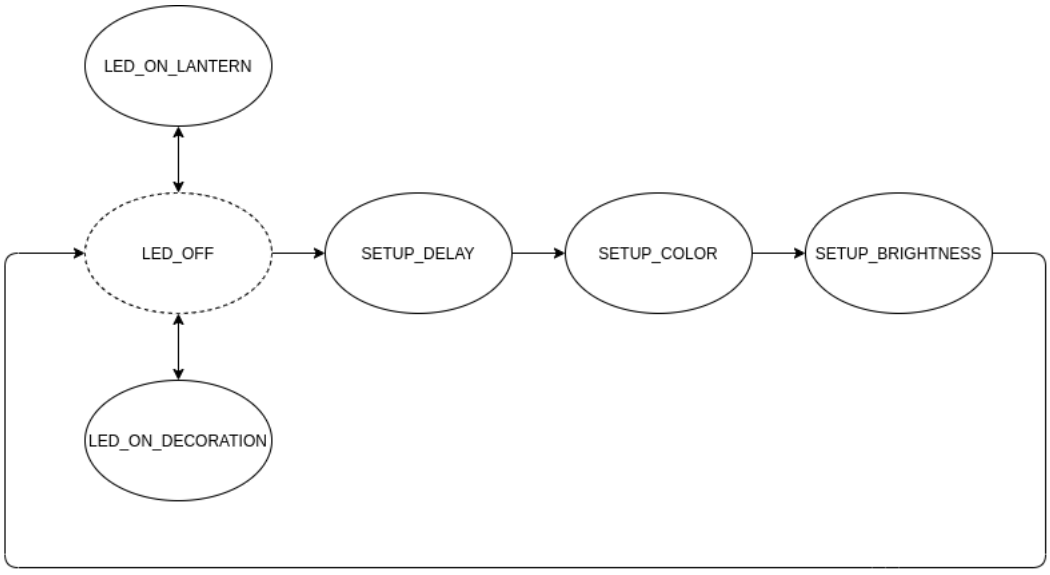

Abbildung 9: Zustandsmaschine

5.7.2 Konfiguration

In jedem *Setup* Zustand kann das entsprechende Array (Siehe 5.2) durchlaufen werden und somit die vordefinierten Werte in der `Config` Datenstruktur gespeichert werden. Der einsatz eines Arrays spart in diesem Fall mehrere Zustände, da nicht für jeden vordefinierten

Wert ein neuer Zustand erstellt werden muss. Der Nachteil dieser Lösung ist der höhere Ramverbrauch durch die Arrays, welcher jedoch bei einer Arraygröße von drei unerheblich ist. Beispielhaft für dieses Verfahren wird der *SETUP_COLOR* Zustand beschrieben.

Listing 19: SETUP_COLOR Zustand

```
1               case SETUP_COLOR:
2               {
3                   if (FORWARD) { // Enter brightness setup menu
4                       lock = true;
5                       config.color = led_colors[setup_index];
6                       led_blink_start(LED_BLINK);
7                       setup_index = SETUP_DEFAULT_INDEX;
8                       state = SETUP_BRIGHTNESS;
9                   } else if (RIGHT && setup_index < SETUP_MAX_INDEX) {
10                      // Get the next color by incrementing the index
11                      lock = true;
12                      setup_index++;
13                      led_on(config.brightness, led_colors[setup_index]);
14                  } else if (LEFT && setup_index > 0) {
15                      // Get the previous color by decrementing the index
16                      lock = true;
17                      setup_index--;
18                      led_on(config.brightness, led_colors[setup_index]);
19                  }
20
21                  break;
22              }
```

Die `setup_index` Variable wird bei jedem Zustandswechsel auf `SETUP_DEFAULT_INDEX`, welches eins ist, gesetzt. Somit startet jeder *Setup* Zustand in der Mitte des Arrays und ermöglicht es, durch inkrementieren von `setup_index` einen höheren Wert, sowie durch das dekrementieren der Variable einen niedrigeren Wert zu wählen. Dabei muss zusätzlich darauf geachtet werden, dass es zu keinem Über- bzw. Unterlauf kommt.

5.8 Debug

Um das Programm zu debuggen wurde ein eigenes Modul entworfen, welches über die *USART* Schnittstelle des *Atmega168* mit einem Computer kommunizieren kann. Damit die Debugausgaben nicht händisch ein- bzw. auskommentiert werden müssen, wurde ein Makro erstellt welches über das präprozessor Flag `DEBUG` aktiviert bzw. deaktiviert wird. Dabei kann das Flag modulweise aktiviert werden und damit nur einzelne Module in den Debugmodus versetzt werden. Sollte das `DEBUG` Flag nicht gesetzt sein, optimiert der Kompilierer den Programmcode raus, sodass keine Performance- sowie Platzeinbußen hingenommen werden müssen.

Listing 20: Debug Makro

```
1 /**
2  * Debug makro that is used to send some data over the USART interface.
3  * If DEBUG is false, the code should be optimized by the compiler.
4  */
5 #define debug_print(...) \
6           if(DEBUG){\
7           char buffer[100];\
8           sprintf(buffer, __VA_ARGS__);\
9           usart_send_s(buffer);}
```

5.9 Voreinstellungen

Alle voreingestellten Werte, wie zum Beispiel die Farben der Kerze, werden in dem *config.h* Modul gehalten. Somit können alle Werte an einem Ort eingesehen und gegebenenfalls geändert werden.

Listing 21: Ausschnitt aus config.h

```
1  //Orange
2  #define COLOR_1_RED 100
3  #define COLOR_1_GREEN 50
4  #define COLOR_1_BLUE 0
5
6  //Warm White
7  #define COLOR_2_RED 100
8  #define COLOR_2_GREEN 85
9  #define COLOR_2_BLUE 50
10
11 //Purple
12 #define COLOR_3_RED 100
13 #define COLOR_3_GREEN 0
14 #define COLOR_3_BLUE 100
```

5.10 Erweiterbarkeit

Um das Programm mit neunen Werten zu erweitern, müssen zunächst neue Konstanten in der *config.h* definiert werden (Siehe 5.9). Dann muss das Array was die Voreinstellungen hält mit den neuen Werten erweitert werden (Siehe 5.2). Zusätzich muss die `SETUP_MAX_INDEX` Konstante um die Anzahl an hinzugefügten Werten addiert werden. Hierbei ist jedoch zu erwähnen, dass diese Konstante von allen Arrays verwendet wird. Somit müssen alle Arrays um Werte erweitert werden. Eine zu empfehlende Verbesserung wäre es, für jedes Array eine eigene Konstante anzulegen.

6 Programmtests

Aufgrund der niedrigen Komplexität des Programmes wurden keine automatischen Tests geschrieben. Die Tests wurden daher *per Hand* ausgeführt. Dabei wurden alle Funktionen sowohl einzeln als auch im Gesamten getestet und die Ergebnisse über die Debugausgabe überprüft. Zusätzlich wurde die Kerze von Dritten getestet um Betriebsblindheit auszuschließen.

6.1 Durchgeführte Tests

Alle ausgeführten Tests wurden mit einer zurückgesetzten Kerze getestet, sodass sich die Tests gegenseiten nicht beeinflussen.

Testfall	Erwartetes Ergebnis	Erzieltes Ergebnis
Starten des Laternenmodus durch Schütteln der Kerze.	Die Kerze sollte in der Standardfarbe (Orange), Standardhelligkeit (50%) leuchten und nach der Standardabschaltverzögerung (1 min) abschalten.	Die Kerze leuchtet mit ihren Standartwerten und schaltet sich nach der Standardabschaltverzögerung ab.
Starten des Dekorationsmodus durch nach hinten Kippen der Kerze.	Die Kerze sollte in der Standardfarbe (Orange), Standardhelligkeit (50%) leuchten und nach der Standardabschaltverzögerung (30 min) abschalten.	Die Kerze leuchtet mit ihren Standartwerten und schaltet sich nach der Standardabschaltverzögerung ab.
Zurücksetzen der Abschaltverzögerung durch Bewegung im Laternenmodus. Dafür wird die Kerze in Laternenmodus versetzt und nach 30 Sekunden geschüttelt.	Die Kerze sollte nach insgesamt einer Minute und 30 Sekunden nach Testbeginn sich ausschalten.	Die Kerze schaltet sich nach 1 Minute und 30 Sekunden aus.
Ausschalten der Kerze im Laternenmodus. Dafür wird die Kerze in den Laternenmodus versetzt und anschließend auf dem Kopf gedreht.	Die Kerze sollte sich ausschalten.	Die Kerze schaltet sich aus.

Ausschalten der Kerze im Dekorationsmodus. Dafür wird die Kerze in den Dekorationsmodus versetzt und anschließend auf dem Kopf gedreht.	Die Kerze sollte sich ausschalten.	Die Kerze schaltet sich aus.
Folgende Parameter werden durch den Konfigurationsmodus gesetzt. • Abschaltverzögerung: 3 min • Farbe: Pink • Helligkeit: 100%	Die Kerze soll in den konfigurierten Parametern leuchten und nach der konfigurierten Abschaltverzögerung sich abschalten.	Die Kerze leuchtet mit den konfigurierten Parametern und schaltet sich nach der konfigurierten Abschaltverzögerung ab.
Kombination der Tests. Die Tests werden ohne Zurücksetze hintereinander ausgeführt.	Die Kerze soll sich in den Tests konsistent und deterministisch verhalten	Die Kerze verhält sich in den Tests konsistent und deterministisch

Abbildungsverzeichnis

Listings