

Den MicroController überwachen

Schaut man sich in den diversen Foren um, so ist ein Thema immer wieder ein großes Problem für viele Entwickler. Die Rede ist von der Ausfallsicherheit bei einem MicroController und wie man eine schnelle Abhilfe schaffen kann. Dieser Blogbeitrag soll dabei die Grundlage einer Steuerung näher erläutern und welche Möglichkeiten es mit dem Arduino oder dem beliebten NodeMCU gibt, um bei einem Sensorausfall trotzdem ein bestehendes Projekt sicher zu steuern. Dazu möchte ich Ihnen das Thema Watchdog und Heartbeat, welches auch in der Industrie ein weitverbreiteter Standard ist, näherbringen.

Grundprinzip einer Steuerung

Wer aus der Industrie kommt, der weiß wie in der Regel eine Speicherprogrammierbare Steuerung, kurz SPS, programmiert werden sollte. Dieses Grundprinzip sollte auch bei Ihren MicroController-Projekten zum Einsatz kommen. Denn Sie wollen sicherlich nicht, dass Sie z.B. bei der Gartenbewässerung durch Ausfall eines Sensors später eine Sumpflandschaft vorfinden, oder die Temperatur in einem Terrarium oder Aquarium falsch gesteuert wird.

Doch wie wird eine SPS oder in unseren Fall MicroController-Steuerung richtig initialisiert? Diese Frage soll Abbildung 1 genauer erläutern.

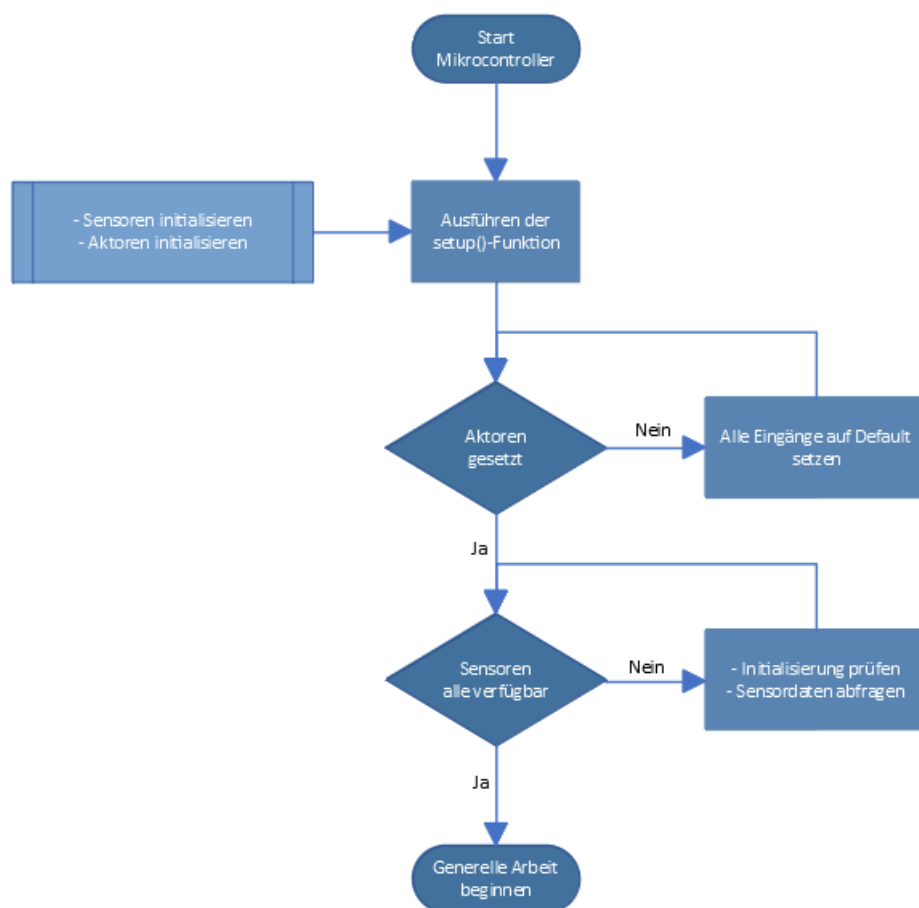


Abbildung 1: Grundprinzip der Initialisierung

Wenn Sie den MicroController starten, führt dieser zunächst die setup()-Funktion aus. In der Regel werden dort Sensoren, Aktoren und Anzeigeelemente initialisiert. Nachdem alle Sensoren initialisiert wurden, sollten zunächst alle Aktoren in ihre Grundposition gefahren werden. In der Regel ist das die Position, in der erst einmal nichts passieren kann. In dem Beispiel mit der Gartenbewässerung wäre dies dann der Motor, der den Wasserhahn erst einmal zudreht. Danach sollten dann die Sensoren

geprüft werden. Bei Sensoren, die eine feste Position haben, wie z.B. Wasserhahn voll aufgedreht oder voll geschlossen, ist das recht einfach. Hierzu wird kurzzeitig der entsprechende Aktor in die vorgegebene Referenzstellung gefahren und geprüft, ob der Sensor reagiert. Sollte ein Signal nicht erwartungsgemäß kommen, so kann man von der Steuerung davon ausgehen, dass ein Defekt vorliegt.

Es gibt aber auch Sensoren, wie z.B. den Temperatursensor DHT22, der eben keinen binären Wert liefert. Ein Drucksensor, welcher den Druck 0 bar für eine Wetterstation zurückliefert, könnte ggf. einen Defekt haben. Hier muss der Programmierer in der Steuerung etwas anders vorgehen. Viele Bibliotheken liefern bei einem Fehler ein „Nan“, was für Not a number steht, oder „Failed to read ...“ bei der Datenabfrage zurück. Aber dann gibt es auch unsaubere Bibliotheken, die genau dies nicht tun. Hier muss der Programmierer schon von Anfang an die Sicherheit mit einprogrammieren, welche Werte im Aufgabenbereich tatsächlich vorkommen könnten, dies wird mit steigender Erfahrung einfacher.

Sind alle Prüfungen abgeschlossen, so kann die eigentliche loop()-Funktion, also die generelle Arbeit vom MicroController, abgearbeitet werden.

Mit dieser einfachen Grundinitialisierung haben Sie schon einmal die erste Sicherheit in Ihre Steuerung gebracht und fangen schon beim Start des MicroControllers einen Defekt ab. Aber auch während des Betriebs kann es zu einem Defekt kommen und dieser sollte auch gefunden werden. Wie sieht da das Grundprinzip aus? Hierzu soll Abbildung 2 die nötige Erklärung liefern.

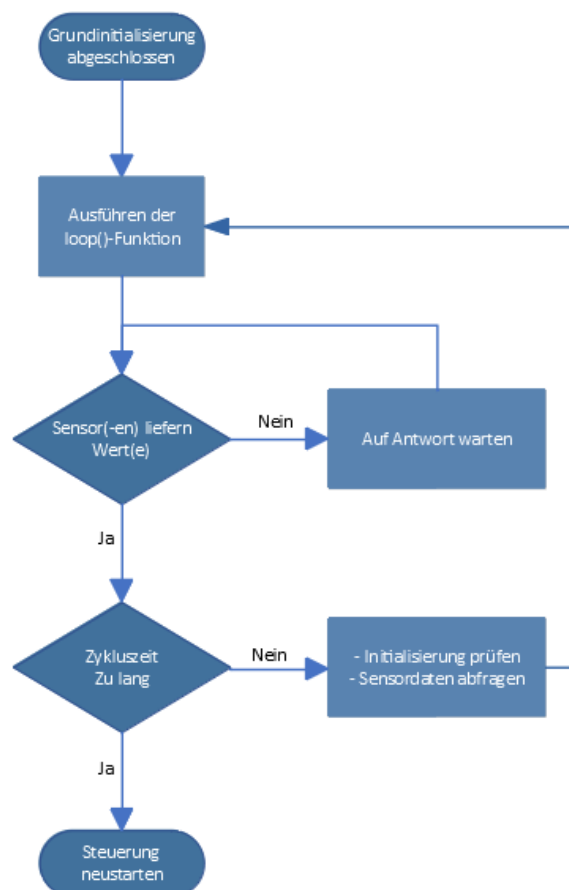


Abbildung 2: Grundprinzip normaler Steuerungsablauf

Ist die Grundinitialisierung abgeschlossen, springt das Programm in die loop()-Funktion unseres MicroControllers. Hier kann über Unterfunktionen oder direkt in der loop()-Funktion entsprechend die Sensorwertabfrage und das Setzen der Aktoren durchgeführt werden. Wichtig ist, dass Sie dem Sensor eine gewisse Zeit einräumen, bis diese die Werte geliefert haben. Interessant ist aber der

zweite Entscheidungsblock „Programmzyklus zu lang“. Wie Sie vielleicht wissen, arbeitet der Arduino oder der NodeMCU seinen Programmcode zyklisch ab. Das heißt, er arbeitet den Code Zeile für Zeile bis zum Ende ab, um dann wieder von Anfang der ersten Zeile der loop()-Funktion zu beginnen. Die Zeit für das Durcharbeiten eines kompletten Zyklus nennt man Zykluszeit. Ein gängiger Fehler wäre z.B., dass sie in einer while-Schleife vom Programm auf das High-Signal von einem Taster oder Sensor warten. Solange dieses Signal nicht kommt, kann der Programmcode nicht abgearbeitet werden. Bei einem defekten Sensor wird also die Abbruchbedingung einer Schleife nie erfüllt und das „Programm bleibt hängen“. Das wollen Sie natürlich vermeiden und prüfen daher die Zykluszeit, um einen solchen Fall abzufangen. In den meisten Fällen wird beim Überschreiten der Zykluszeit die Steuerung resettet, sodass mit der Grundinitialisierung begonnen wird und Sie werden ggf. vorher noch benachrichtigt. Wie diese Zykluszeit nun genau überwacht wird, erfahren Sie im nächsten Kapitel „Watchdog“ und im späteren Kapitel „Heartbeat“.

WARNUNG

An dieser Stelle gleich eine Warnung an Sie, sofern Sie gerne einmal experimentieren! Sollten Sie die Zeit vom Watchdog beim Arduino oder NodeMCU zu klein wählen, kann es dazu führen, dass Sie Ihren MicroController nicht mehr benutzen können, da er noch während des Bootens wieder neugestartet wird. Beim Arduino gibt es die Möglichkeit, über die ISP-Schnittstelle den Bootloader neu zu flashen und damit das Problem zu lösen. Beim NodeMCU findet sich eine solche Möglichkeit nicht, aber es scheint, dass man mit etwas Glück über den normalen Programmcodetransfer das Problem beheben kann.

Sollten Sie etwas anderes als den Arduino Nano verwenden, so kann das Ausprobieren der Sketche dazu führen, dass der Nano nicht mehr funktioniert. Achten Sie gerade beim Arduino Nano darauf, dass der neuste Bootloader gebrannt wurde!

Der Watchdog beim Arduino

Watchdog heißt übersetzt Wachhund und genau diese Aufgabe übernimmt ein Watchdog. Er prüft, fast unabhängig von Ihrem Code, ob der MicroController normal arbeitet.

Damit Sie den Watchdog beim Arduino nutzen können, müssen Sie zunächst die AVT-Watchdog-Library in Ihr Projekt inkludieren. Diese Bibliothek müssen Sie nicht erst über die Bibliotheksverwaltung installieren, sondern ist direkt bei der Installation der Arduino IDE mit dabei. Für das Inkludieren fügen Sie einfach die unter Code 1 angegebenen Zeile am Anfang Ihres Projektes ein.

```
#include <avr/wdt.h>  
Code 1: Include der AVR-Watchdog-Library
```

Für den Watchdog brauchen Sie im Grunde drei Funktionen, die in Tabelle 1 genauer erklärt sind.

Funktion	Beschreibung
<code>wdt_enable(zeit)</code>	Aktiviert den Watchdog und setzt eine Zeit bis zum Auslösen. Die möglichen Zeitangaben sind Tabelle 2 zu entnehmen.
<code>wdt_disable()</code>	Deaktiviert den Watchdog wieder.
<code>wdt_reset()</code>	Setzt den internen Timer wieder zurück. Diese sollte regelmäßig, z.B. am Ende der <code>loop()</code> -Funktion geschehen, damit der Arduino nicht resettet.

Tabelle 1: Arduino Watchdog-Funktionen und -Beschreibungen

Die Watchdog-Timer können aus dem ATmega238-Datenblatt entnommen werden, der Einfachheit sind diese aber in Tabelle 2 aufgelistet. Bedenken Sie, dass gerade die ersten fünf Zeiten aus Tabelle 2 relativ knapp sind und es dabei zu den oben genannten Reset-Fehlern kommen kann.

Zeit	Parameter für <code>wdt_enable(zeit)</code>
16 ms	<code>WDTO_15MS</code>
32 ms	<code>WDTO_30MS</code>
64 ms	<code>WDTO_60MS</code>
0,125 s	<code>WDTO_120MS</code>
0,25 s	<code>WDTO_250MS</code>
0,5 s	<code>WDTO_500MS</code>
1,0s	<code>WDTO_1S</code>
2,0 s	<code>WDTO_2S</code>
4,0 s	<code>WDTO_4S</code>
8,0 s	<code>WDTO_8S</code>

Tabelle 2: Arduino-Watchdog Zeitkonstanten

Schaut man sich das nun mal in einem einfachen Beispiel an, siehe Code 2, kann man den Effekt vom Watchdog sehr gut sehen.

```
// Watchdog first example for Arduino  
// Autor: Joern Weise  
// License: GNU GPI 3.0  
// Created: 11. Sep 2020  
// Update: 11. Sep 2020  
//-----
```

```
#include <avr/wdt.h>
```

```
void setup() {
```

```

Serial.begin(9600);           //Activate serial com
pinMode(LED_BUILTIN, OUTPUT); //Activate included LED
digitalWrite(LED_BUILTIN, LOW); //Turn the LED off
Serial.println("Arduino Nano started....."); //First message for monitor
wdt_enable(WDTO_2S);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  Serial.println("Set LED to HIGH"); // Serial information
  delay(1000);                     // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
  Serial.println("Set LED to LOW"); // Serial information
  delay(1000);                     // wait for a second
  Serial.println("Reset watchdog"); // Serial information
  wdt_reset();                     // Reset watchdog
}

```

Code 2: Einfaches Arduino Beispiel für Watchdog

Im Grunde handelt es sich bei dem Code um eine modifizierte Version vom Beispiel „Blink“, nur dass noch der Watchdog implementiert wurde. Dieser soll die Zykluszeit vom Programm beobachten und bei einer Watchdog-Überschreitung von 2 Sekunden, den Arduino neu starten. Für die bessere Übersicht wurden noch ein paar Ausgaben an den seriellen Monitor hinzugefügt. Die Ausgabe auf dem seriellen Monitor können Sie in Abbildung 3 sehen.

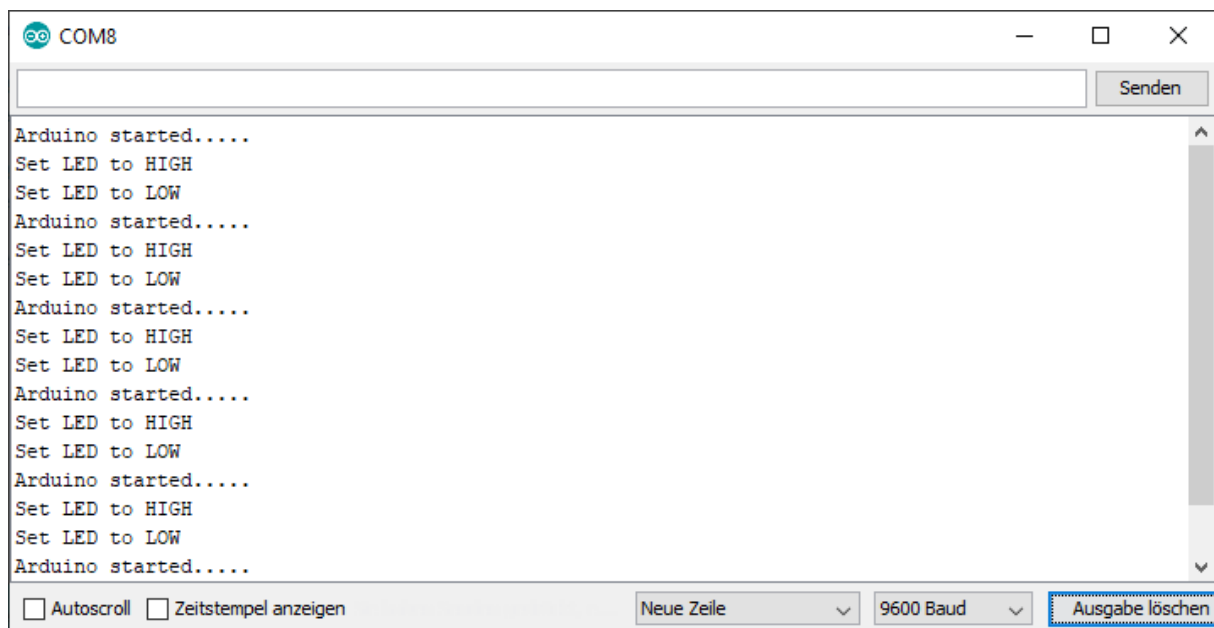


Abbildung 3: Ausgabe serieller Monitor Arduino Watchdog Beispiel 1

Unerwartet ist hier der komplette Neustart des Programms, denn die Ausgabe „Arduino started.....“ ist ja fester Teil des Codes in der Funktion setup(). Was also ist nun genau passiert? Sie sehen, dass der Arduino und die LED erst auf „HIGH“ und danach auf „LOW“ gesetzt wird. Jedoch erreicht der Code nie die Zeile, in der der Watchdog resettet wird, `wdt_reset()`. Dies ist der Fall, da die beiden Verzögerungen, die Funktion `delay(1000)`, die Zykluszeit auf über 2 Sekunden anwachsen lässt. Damit ist der Arduino durch den Watchdogtimer von 2 Sekunden gezwungen, einen kompletten Neustart durchzuführen und die Ausgabe wiederholt sich von neuem.

Ändern Sie die Verzögerung nur ein kleines bisschen, z.B. auf 300ms, siehe Code 3, so sieht die Ausgabe im seriellen Monitor wie in Abbildung 4 aus.

```
// Watchdog second example for Arduino
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 11. Sep 2020
// Update: 11. Sep 2020
//-----

#include <avr/wdt.h>

void setup() {
  Serial.begin(9600);          //Activate serial com
  pinMode(LED_BUILTIN, OUTPUT); //Activate included LED
  digitalWrite(LED_BUILTIN, LOW); //Turn the LED off
  Serial.println("Arduino started....."); //First message for monitor
  wdt_enable(WDTO_1S);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  Serial.println("Set LED to HIGH"); // Serial information
  delay(300); // wait for a second
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
  Serial.println("Set LED to LOW"); // Serial information
  delay(300); // wait for a second
  Serial.println("Reset watchdog"); // Serial information
  wdt_reset(); // Reset watchdog
}
```

Code 3: Angepasstes Arduino Beispiel für Watchdog

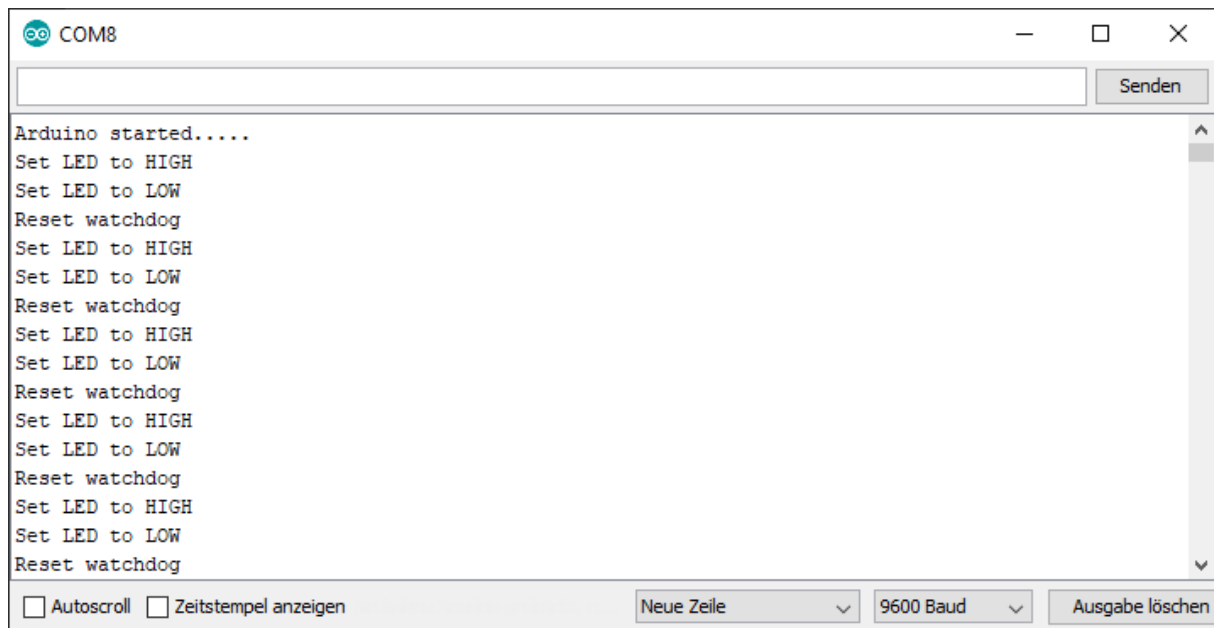


Abbildung 4: Ausgabe vom angepassten Arduino Beispiel

Was hat sich also grundlegend geändert, es wurde doch „nur“ die Zeitverzögerung verändert? Die Frage ist in diesem Fall auch gleichzeitig die Antwort. Das veränderte delay bewirkt, dass die Zykluszeit unter einer Sekunde bleibt und der Code komplett abgearbeitet werden kann. Dadurch wird die Funktion `wdt_reset()` zum Resetten des Watchdog-Timers ausgeführt und der Arduino arbeitet fleißig Zyklus für Zyklus seinen Code in der `loop()`-Funktion ab.

Das hier vorgestellte Beispiel ist nun nicht sehr praxisnah, dient aber erst einmal als Grundlage, um den Watchdog überhaupt zu verstehen. Um nun einen Praxisbezug zu bekommen, soll ein Code generiert werden, der einer kleinen alltäglichen Schaltung nahekommt.

Wir haben einen Taster, welcher am Arduino mit Pin 3 und GND verbunden ist. Dieser soll in regelmäßigen Abständen ein High-Signal liefern, ähnlich wie es ein Sensor macht. Bei Pin 3 wird der interne Pullup-Widerstand aktiviert, weswegen Pin 3 auf Masse gezogen werden muss, siehe Abbildung 5. Sind die Sensordaten 10 mal empfangen worden, wird der Watchdog deaktiviert.

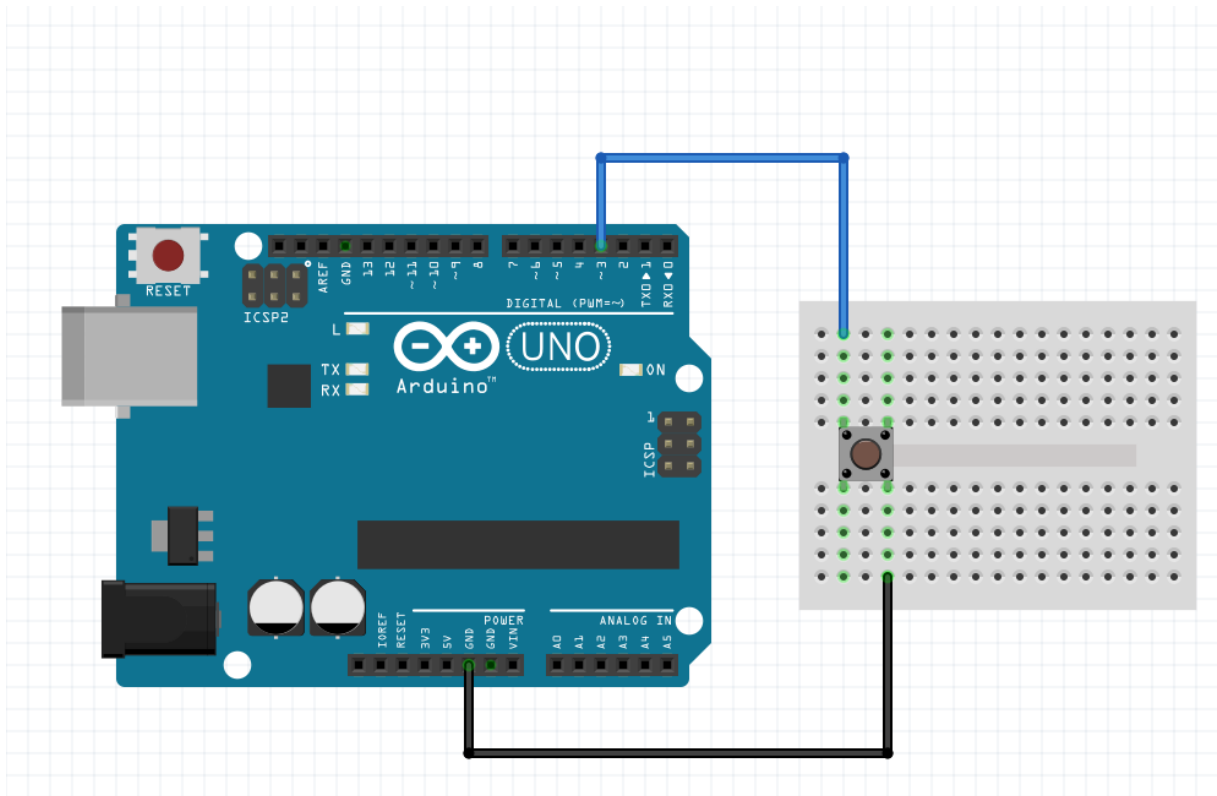


Abbildung 5: Praxisbeispiel für Arduino Watchdog

Der Code sieht entsprechend dem unten aufgezeigten Code 4 aus.

```
// Watchdog third example for Arduino
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 11. Sep 2020
// Update: 11. Sep 2020
//-----

#include <avr/wdt.h> //Load Watchdog-Library

int iCount = 0; //Global var for monitoring

//Init arduino
void setup()
{
  pinMode(3, INPUT); // Digital-Pin 3 as input
  digitalWrite(3, HIGH); // Activate pullup-resistor mode
  Serial.begin(9600); // Active serial com with baudrate 9600
  Serial.println("Arduino started ..."); //Say hello to the world
  wdt_enable(WDTO_4S); // Set watchdog timer to 4 seconds
}

//Simple loop-function
void loop()
{
  Serial.print(iCount); //Show current counter
```



```

Serial.println(": waiting for sensor ...");
GetSensorData();
wdt_reset();
iCount++;
if (iCount == 10) //After 10 cycles disable watchdog
{
    wdt_disable(); // Disable watchdog
    Serial.println("WD disabled ..."); //Write information to serial monitor
}
}

//Check current sensor data
void GetSensorData()
{
    while (digitalRead(3) == HIGH) //Loop till button pressed
    {
        delay(500);
    }
    Serial.println("Sensordata received ...");
    delay(1000);
}

```

Code 4: Code für Arduino Watchdog-Beispiel 3

Die Ausgabe bei einem normal funktionieren Sensor sieht wie in Abbildung 6 aus. Interessant an dieser Stelle zu sehen ist, dass nach der 10 Abfrage vom Sensor der Watchdog mit „WD disabled ...“ deaktiviert wurde.

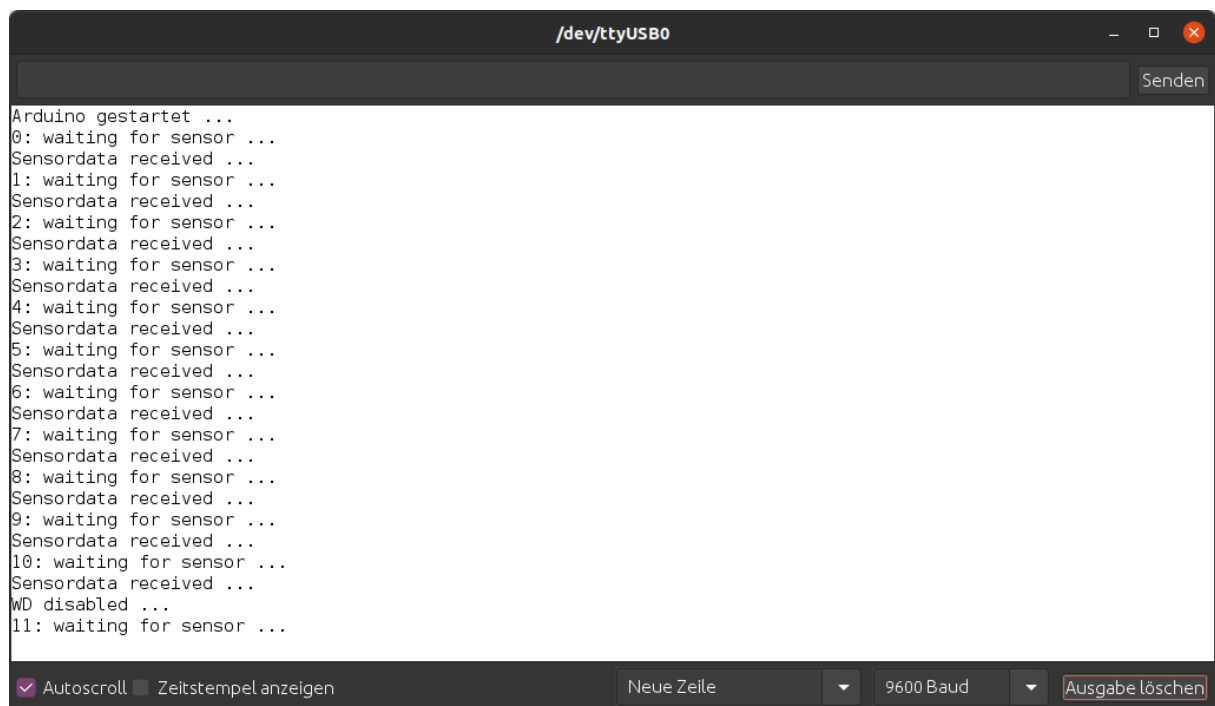


Abbildung 6: Ausgabe serieller Monitor für Arduino Watchdog-Beispiel 3

Bei einem defekten Sensor wird, sofern nicht die vorherigen 10 Prüfungen bestanden wurde, der Arduino neu gestartet, siehe Abbildung 7.

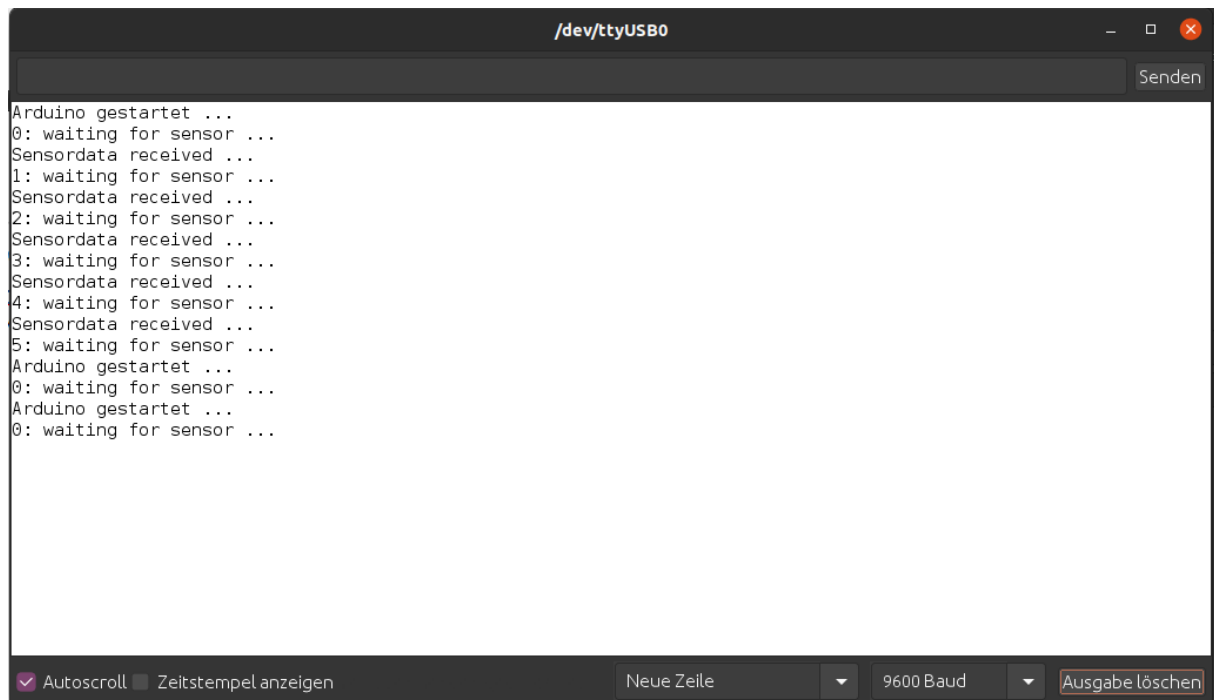


Abbildung 7: Ausgabe serieller Monitor für Arduino Watchdog-Beispiel 3 defekter Sensor

Beim Starten des Arduino funktionierte der Sensor zunächst, aber lieferte während der Prüfung dann kein Signal mehr. Dadurch wurde der Arduino, durch den abgelaufenen Watchdog-Timer, wieder neu gestartet, was bei der weiteren Prüfung vom Sensor zu einem Neustart führt.

Aufgrund der Tabelle 2 werden Sie sich vielleicht die Frage stellen, was Sie machen, wenn ein Programmzyklus mehr als 8 Sekunden dauert. Bisher war immer die Annahme, dass ein Programmzyklus nur wenige Millisekunden in Anspruch nimmt. Daher wurde der Reset vom Watchdog immer am Ende der loop()-Schleife durchgeführt. Sie müssen also bei längeren Zykluszeiten an entsprechender Stelle einen Reset vom Watchdog setzen, siehe dazu Code 5.

```
// Watchdog fourth example for Arduino
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 11. Sep 2020
// Update: 11. Sep 2020
//-----

#include <avr/wdt.h> //Load Watchdog-Library
#define LOOP_COUNT 20
int iCount = 0; //Global var for monitoring

//Init arduino
void setup()
{
  Serial.begin(9600); // Active serial com with baudrate 9600
  Serial.println("Arduino started ..."); //Say hello to the world
  wdt_enable(WDTO_4S); // Set watchdog timer to 4 seconds
}
```

```
//Simple loop-function
void loop()
{
  for(iCount = 0; iCount < LOOP_COUNT; iCount++)
  {
    delay(1000);
    Serial.print("Seconds: ");
    Serial.print(iCount);
    Serial.println(" - Resetting Watchdog");
    wdt_reset();
  }
  Serial.println("End of loop");
}
```

Code 5: Watchdog-Timer nicht am Ende von loop()-Funktion

Hier wird eine for-Schleife 20 mal durchlaufen, mit einer Verzögerung von einer Sekunde zwischen jedem weiteren Schleifendurchlauf. Würden der Watchdog-Timer in jedem Schleifendurchlauf nicht mittels `wdt_reset()` zurückgesetzt werden, würde nach vier Sekunden und somit vier Schleifen der Arduino neustarten. Die Ausgabe, siehe Abbildung 8, zeigt, dass durchaus Zeiten über den gesetzten Watchdog-Timer, hier vier Sekunden, möglich sind, Sie müssen nur an passender Stelle einen Reset vom Watchdog-Timer durchführen.

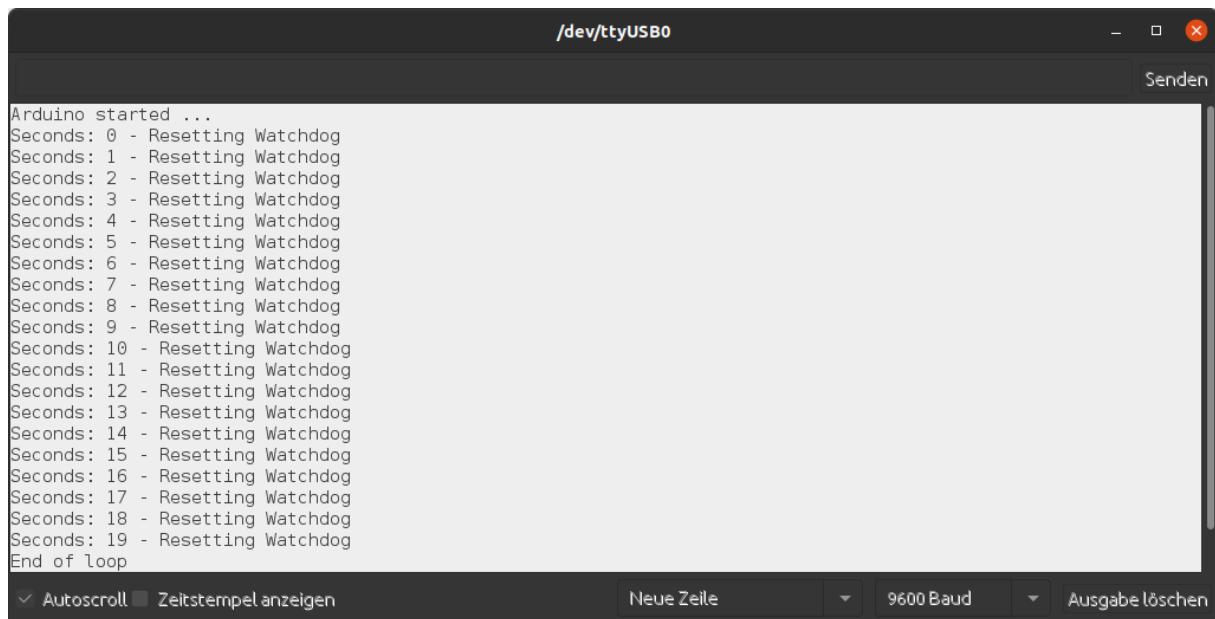


Abbildung 8: Ausgabe serieller Monitor für Arduino Watchdog-Beispiel 4

Sie können auch in einer Unterfunktion einen Reset vom Watchdog-Timer ausführen. Hier müssen Sie wissen oder besser entscheiden, wo das in Ihrem Projekt sinnvoll ist.

Der Watchdog der ESP32-Familie

Was für den Arduino gilt, gilt auch für die Produktfamilie der ESP32-MicroController. Auch hier gibt es einen Watchdog, der entsprechend aktiviert werden kann, um eine Ausfallsicherheit zu haben. Um den Watchdog für die ESP32-MicroController zu nutzen, müssen Sie die entsprechende Watchdog-Library in der Arduino IDE einbinden, siehe Code 6.

```
#include <esp_task_wdt.h>
```

```
#include <esp_task_wdt.h> //Load Watchdog-Library
#define WDT_TIMEOUT_SECONDS 3
```

```

int iCount = 0; //Global var for monitoring
//Init ESP32
void setup()
{
  Serial.begin(115200); // Active serial com with baudrate 9600
  Serial.println("ESP started ..."); //Say hello to the world
  pinMode(0, INPUT_PULLUP); // Digital-Pin 0 as input
  esp_task_wdt_init(WDT_TIMEOUT_SECONDS,true); //Init Watchdog with 3 seconds timeout and
  panicmode
  esp_task_wdt_add(NULL); //No special task needed
}

//Simple loop-function
void loop()
{
  Serial.print(iCount); //Show current counter
  Serial.println(": waiting for sensor ...");

  GetSensorData();

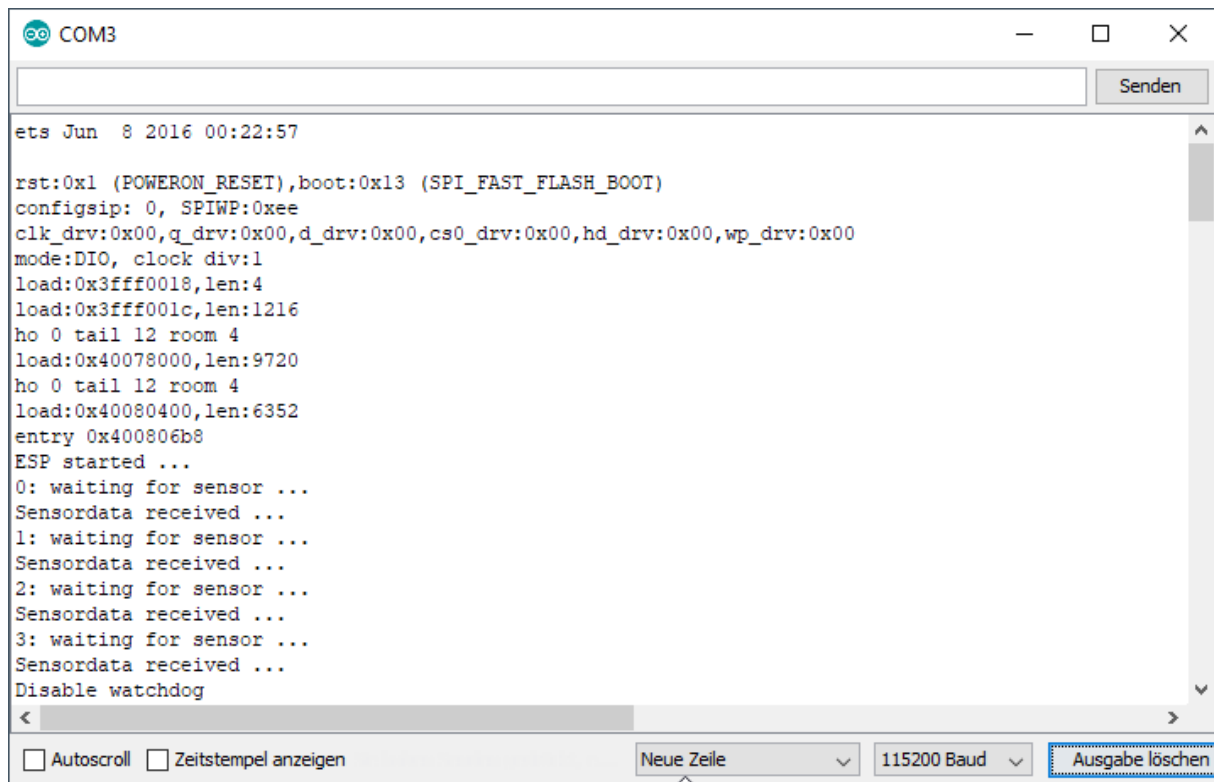
  esp_task_wdt_reset();
  if(iCount == 3) //After 3 cyclus disable watchdog
  {
    Serial.println("Disable watchdog"); //Write information to serial monitor
    esp_task_wdt_deinit(); // Disable watchdog
  }
  iCount++; //Increase counter
}

//Check current sensor data
void GetSensorData()
{
  while(digitalRead(0) == HIGH) //Loop till button pressed
  {
    delay(500);
  }
  Serial.println("Sensordata received ...");
  delay(1000);
}

```

Code 7: Code für ESP Watchdog-Beispiel 1

Eine erfolgreiche Ausgabe sehen Sie in Abbildung 10.

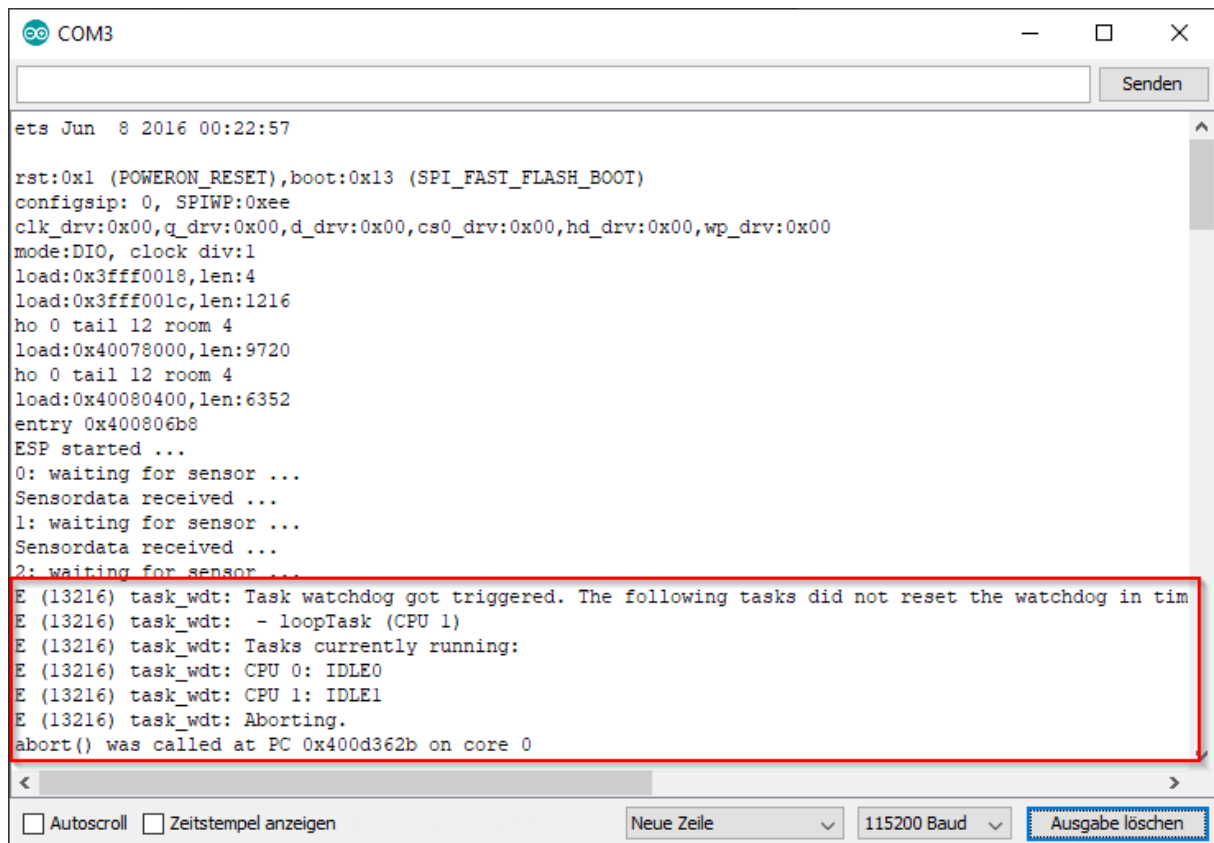


```
ets Jun 8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
config: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
ESP started ...
0: waiting for sensor ...
Sensordata received ...
1: waiting for sensor ...
Sensordata received ...
2: waiting for sensor ...
Sensordata received ...
3: waiting for sensor ...
Sensordata received ...
Disable watchdog
```

Abbildung 10: Ausgabe serieller Monitor für ESP32 Watchdog-Beispiel 1

Soweit ist die Ausgabe erst einmal unspektakulär, da der Sensor erst einmal ordnungsgemäß funktioniert. Der ESP32 startet normal und Sie erhalten die üblichen Anzeigen beim Start des ESP32 auf dem seriellen Monitor angezeigt. Nach den drei Durchläufen des Sensors wird, wie es im Quellcode hinterlegt ist, der Watchdog deaktiviert und die Ausgabe „Disable watchdog“ ausgegeben. Interessanter dagegen ist die Ausgabe bei einem Ablauf vom Watchdog-Timer des ESP32, siehe Abbildung 11.



```
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
ESP started ...
0: waiting for sensor ...
Sensordata received ...
1: waiting for sensor ...
Sensordata received ...
2: waiting for sensor ...
E (13216) task_wdt: Task watchdog got triggered. The following tasks did not reset the watchdog in time
E (13216) task_wdt: - loopTask (CPU 1)
E (13216) task_wdt: Tasks currently running:
E (13216) task_wdt: CPU 0: IDLE0
E (13216) task_wdt: CPU 1: IDLE1
E (13216) task_wdt: Aborting.
abort() was called at PC 0x400d362b on core 0
```

Abbildung 11: Ausgabe serieller Monitor für ESP32 Watchdog-Beispiel 1 defekter Sensor

Schaut man sich die rot umrandete Ausgabe genauer an, siehe Abbildung 11, liefert einem diese Ausgabe eine Menge Informationen. Zunächst sehen wir, dass der Timer vom Watchdog abgelaufen ist bzw. nicht resettet wurde. Direkt danach sehen Sie, dass ein Prozess noch läuft und der laufende Prozess abgebrochen wird. Zuletzt erhalten Sie noch die Information, dass die Funktion `abort()` ausgeführt wird, die den Neustart vom ESP32 durchführt.

Zwischenfazit

Bis zu diesem Punkt haben Sie die Grundidee einer simplen Steuerung kennengelernt und worauf es bei der Initialisierung und Überwachung eines einzelnen MicroControllers ankommt. Durch den Watchdog wurde Ihnen nähergebracht, wie Sie den Watchdog für den Arduino bzw. einen MicroController der ESP32-Familie in Ihren Quellcode einbinden können. Zusätzlich wurde erklärt, wie Sie beim Arduino auch Watchdog-Timer größer 8 Sekunden implementieren können und dass ein Reset des Watchdog-Timers sogar in Unterfunktionen erfolgen kann.

Der Heartbeat

Wie schon erwähnt, haben Sie die Grundlagen für die Überwachung von genau einem MicroController kennengelernt. Doch was ist, wenn Sie nun ein größeres Projekt haben, wie z.B. eine komplexe Gartenüberwachungs- und -steuerung, welche mehr als einen MicroController benötigt? Gerade wenn die verschiedenen MicroController parallel Aufgaben übernehmen sollen und ggf. auch Informationen von anderen MicroControllern benötigen, brauchen Sie eine zentrale Überwachung. Hierfür gibt es die Methode vom „Heartbeat“, was zu Deutsch Herzschlag bedeutet. Das Prinzip dahinter ist, dass ein zentraler MicroController, der sogenannte Master, mit allen weiteren MicroControllern, den sogenannten Slaves, ein zyklisches Signal austauscht, um zu kontrollieren, ob jeder Slave noch aktiv arbeitet. Hierbei gibt es zwei Varianten, wie dies umgesetzt werden kann; im

Fehlerfall reagieren die Master jedoch gleich und setzen ein LOW-Signal auf den RESET-Pin vom Slave-Board. Bei den Arduinos ist dieser durch „RESET“ oder „RST“, bei den MicroControllern der ESP32-Familie mit „EN“ beschriftet. Empfehlenswert ist immer ein Blick in die Pinout-Beschreibung vom jeweiligen MicroController.

In der ersten Variante wechselt der Slave zyklisch das Heartbeat-Signal von LOW zu HIGH und HIGH zu LOW. Der Master empfängt diesen Signalwechsel und erwartet in einem bestimmten Zeitfenster eine Änderung vom Signal. Wird diese nicht festgestellt, setzt der Master das Signal zum Neustart des Slave-MicroControllers bzw. für alle Slaves. Die Schaltung ist dabei recht einfach, siehe Abbildung 12.

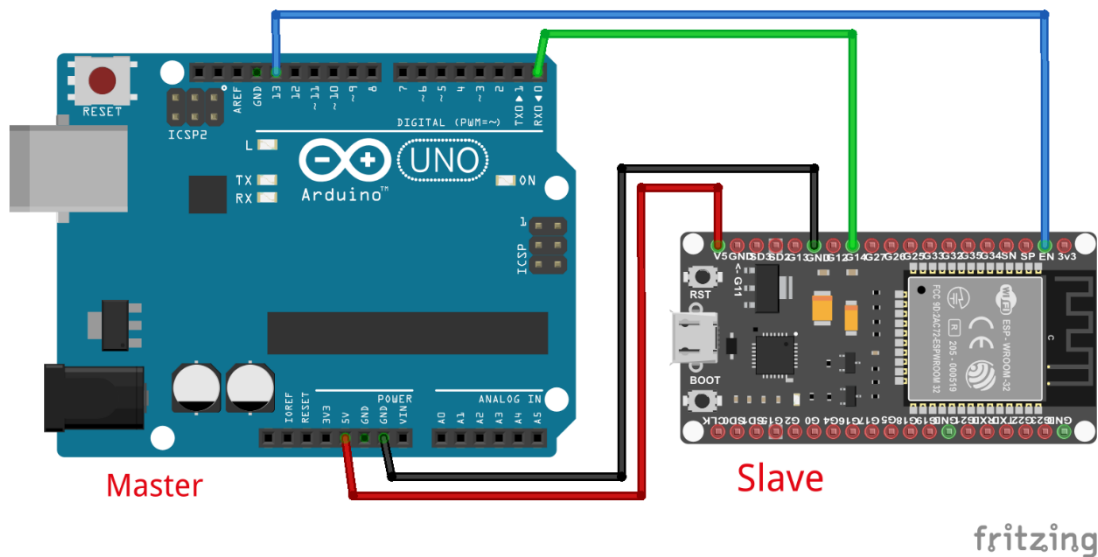


Abbildung 12: Grundsaltung für Heartbeat-Beispiel 1

Der Code für den Master, der Arduino Uno, behandelt nur die Abfrage vom Heartbeat des Slaves, siehe Code 8.

```
// Heartbeat example 1 for master (Arduino Uno)
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 15. Sep 2020
// Update: 15. Sep 2020
//-----

#define HEARTBEAT_SHORT_INTERVAL 2000 //Maximum allowed time for normal change
#define HEARTBEAT_LONG_INTERVAL 5000 //Needed time for rebooting slave

int iCount = 0; //Global var for monitoring
unsigned long previousMillis = 0; //Stores current millis to check reset needed
int iCurrentInterval = 0; //Var to set correct interval, see define
int iLastHeartbeatStatus = LOW; //Last know status from slave
int iSlavePin = 2; //Pin to receive slave IO
bool bSlaveStarting = true; //Bool to switch during "normal" mode and slave is (re-)starting
//Init Arduino
void setup()
{
  Serial.begin(115200); // Active serial com with baudrate 115200
```



```

Serial.println("Arduino (master) started ..."); //Say hello to the world
pinMode(iSlavePin, INPUT); // Digital-Pin 0 as input}
pinMode(13, OUTPUT); //Output to reset slave
digitalWrite(13, HIGH); //Needs per default set to HIGH
iCurrentInterval = HEARTBEAT_LONG_INTERVAL; //Lets wait 5 sec during startup
}

//Simple loop-function
void loop()
{
  delay(200); // just wait a few seconds before starting next loop
  unsigned long currentMillis = millis(); //Get current millis
  if(!bSlaveStarting) //Normal mode 2 sec
  {
    //Check if status from slave change from LOW to HIGH
    if(iLastHeartbeatStatus == LOW && digitalRead(iSlavePin) == HIGH)
    {
      previousMillis = currentMillis;
      iLastHeartbeatStatus = HIGH;
      Serial.print("Change: "); //Write loop
      Serial.print(iCount); //Show current counter
      Serial.println(" - Got heartbeat HIGH");
      iCount++;
    }

    //Check if status from slave change from HIGH to LOW
    if(iLastHeartbeatStatus == HIGH && digitalRead(iSlavePin) == LOW)
    {
      previousMillis = currentMillis;
      iLastHeartbeatStatus = LOW;
      Serial.print("Change: "); //Write loop
      Serial.print(iCount); //Show current counter
      Serial.println(" - Got heartbeat LOW");
      iCount++;
    }
  }
  else //(Re-)starting mode 5 sec
  {
    if(digitalRead(iSlavePin) == HIGH) //So there must be something that sends signal
    {
      Serial.println("Slave online");
      iLastHeartbeatStatus = HIGH;
      bSlaveStarting = false; //Switch to normal mode
      previousMillis = millis(); //Overwrite previousMillis
      iCurrentInterval = HEARTBEAT_SHORT_INTERVAL; // 2 seconds wait for switching
    }
  }

  //Check if reboot from slave is needed or not

```

```

if(currentMillis - previousMillis >= iCurrentInterval)
{
  ResetSlave(); //Reset slave and overwrite some vars
}

}

//Function to reset slave
void ResetSlave()
{
  Serial.println("Set RESET-Pin slave to LOW");
  digitalWrite(13, LOW); //Reset slave
  delay(500); //Wait 500ms, so slave restarts
  Serial.println("Set RESET-Pin slave to HIGH");
  digitalWrite(13, HIGH); //Lets start slave up
  previousMillis = millis(); //Overwrite previousMillis
  iCurrentInterval = HEARTBEAT_LONG_INTERVAL; //Saftey for waiting slave restart
  bSlaveStarting = true; //Slave is restarting so switch to (re-)start mode
}

```

Code 8: Code Master Heartbeat-Beispiel 1

Da der Slave in diesem Beispiel, der NodeMCU, hier keine weitere Funktion hat, ist nur der Code für den zyklischen Heartbeat in Code 9 zu sehen.

```

// Heartbeat example 1 for slave (ESP32)
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 15. Sep 2020
// Update: 15. Sep 2020
//-----

#define HEARTBEAT_INTERVAL 1000

int iCount = 0; //Global var for monitoring
unsigned long previousMillis = 0;
int iLastHeartbeatStatus = LOW;
//Init ESP32
void setup()
{
  Serial.begin(115200); // Active serial com with baudrate 9600
  Serial.println("ESP started ..."); //Say hello to the world
  pinMode(14, OUTPUT); // Digital-Pin 14 as output
  digitalWrite(14, LOW); //Init as with LOW SIGNAL
}

//Simple loop-function
void loop()
{
  unsigned long currentMillis = millis(); //Get current millis

```

```

if(currentMillis - previousMillis >= HEARTBEAT_INTERVAL) //Check if interval is equal or over 1
second
{
  Serial.print("Loop: "); //Write loop
  Serial.println(iCount); //Show current counter
  previousMillis = currentMillis; //Overwrite stored previous millis
  if(iLastHeartbeatStatus == LOW) //Check if last signal was HIGH or LOW
  {
    Serial.println("Set Heartbeat to HIGH"); //Some message for serial monitor
    iLastHeartbeatStatus = HIGH; //Set signal to HIGH
  }
  else
  {
    Serial.println("Set Heartbeat to LOW"); //Some message for serial monitor
    iLastHeartbeatStatus = LOW; //Set signal to LOW
  }
  digitalWrite(14, iLastHeartbeatStatus); // Write new status to digital pin
  iCount++; //Increase counter
}
delay(200); // just wait a few milliseconds before starting next loop
}

```

Code 9: Code Slave Heartbeat-Beispiel 1

Damit Sie genau sehen, was der Master und der Slave machen, sind in beiden Codes reichlich Ausgaben an den seriellen Monitor hinzugefügt worden. Für einen normalen Durchlauf sieht die Ausgabe auf dem seriellen Monitor recht simpel aus, siehe Abbildung 13.



Abbildung 13: Ausgabe serieller Monitor für Master Heartbeat-Variante 1

Der Master wird gestartet und wartet zunächst auf den Slave, bis das erste HIGH-Signal empfangen wird. In diesem Fall wird die Ausgabe „Slave online“ erzeugt. Ab diesen Zeitpunkt wechselt der Modus im Quellcode und es wird spätestens alle 2 Sekunden ein Signalwechsel erwartet. Im Fall der Zeitstempel der seriellen Ausgabe sieht man deutlich, dass fast alle Sekunde der Slave einen Heartbeat sendet. Interessant hingegen ist Abbildung 14, da Sie hier 3 Signalwechsel und danach einen Neustart vom Slave sehen „Set RESET-Pin.....“.

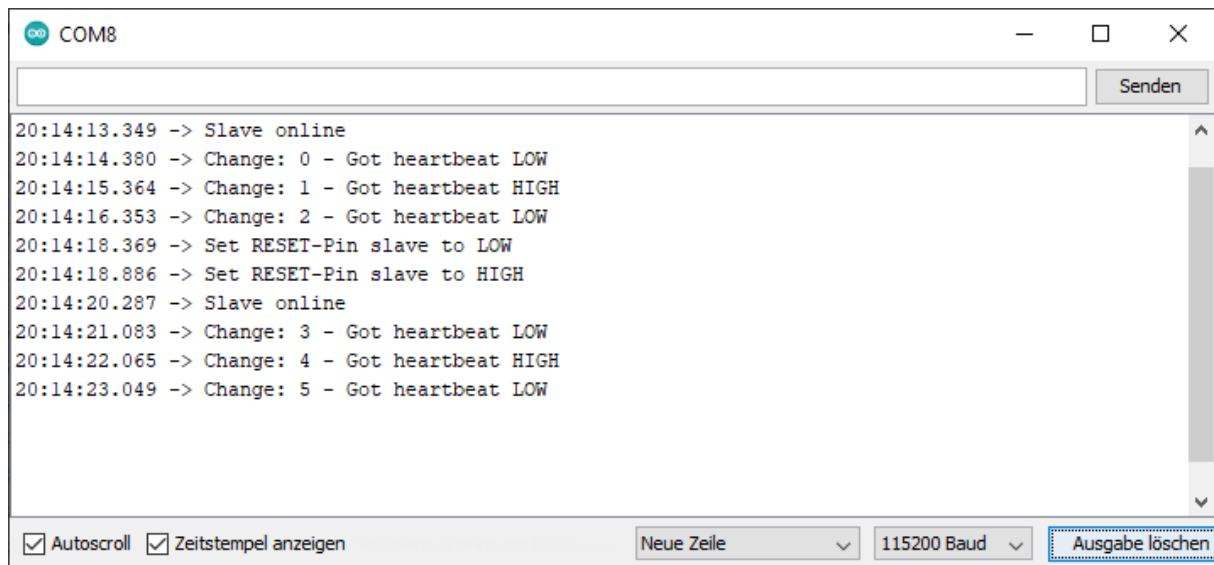


Abbildung 14: Ausgabe serieller Monitor für Master Heartbeat-Variante 1 mit Neustart vom Slave

Diesen Fall können Sie leicht erzeugen, wenn Sie für mehr als zwei Sekunden den Pin 2 vom Master oder Pin G14 am Slave lösen. Vor dem Neustart des Slaves sehen Sie, dass der letzte Signalwechsel zwei Sekunden zurückliegt und damit das definierte Intervall überschreitet. Der Master setzt Pin 13 auf LOW, was beim Slave einen Neustart bewirkt. Nach knapp 500ms wird der Pin 13 vom Master wieder auf HIGH gesetzt, damit der Slave auch wieder starten kann. Kurz danach ist der Slave wieder online, wenn Sie die Pin Verbindung wieder hergestellt haben, und der normale Heartbeat geht weiter.

Die zweite Variante vom Heartbeat, setzt eine weitere Verbindung zwischen Master und Slave voraus, siehe Abbildung 15 die graue Verbindung. Hier wird Pin 3 vom Master mit Pin G27 vom Slave verbunden.

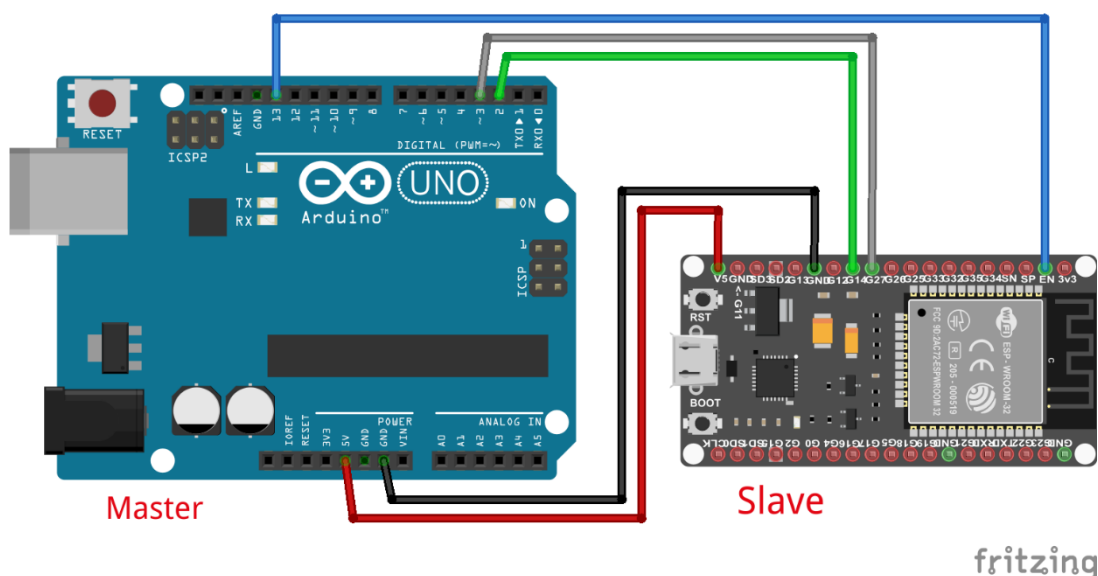


Abbildung 15: Grundschtaltung für Heartbeat-Beispiel 2

Das Prinzip der zweiten Heartbeat-Variante besteht darin, dass nicht der Slave zyklisch ein Signal sendet, wie in Heartbeat-Variante 1, sondern der Master ein Signal vorgibt und der Slave dieses beantwortet. In den meisten Fällen wird einfach das gesendete Signal vom Master gespiegelt und

zurückgesendet. Der Code für den Master, Arduino Uno, ist in // Heartbeat example 2 for master (Arduino Uno)

// Autor: Joern Weise

// License: GNU GPI 3.0

// Created: 15. Sep 2020

// Update: 15. Sep 2020

//-----

#define HEARTBEAT_RECEIVE_INTERVAL 2000 //Maximum allowed time for normal change

#define HEARTBEAT_CHANGE_INTERVAL 3000 //Needed time for rebooting slave

int iCount = 0; //Global var for monitoring

unsigned long previousMillis = 0; //Stores current millis to switch Heartbeat-Signal

int iLastHeartbeatStatus = HIGH; //Last know status from slave

int iSlavePin = 2; //Pin to receive slave IO

int iSignalPin = 3; //Pin to set new flag

bool bSlaveStarting = true; //Bool to switch during "normal" mode and slave is (re-)starting

bool bCheckNewSignal = false;

//Init Arduino

void setup()

{

Serial.begin(115200); // Active serial com with baudrate 115200

Serial.println("Arduino (master) started ..."); //Say hello to the world

pinMode(iSlavePin, INPUT); // Digital-Pin 2 as input}

pinMode(iSignalPin, OUTPUT); //Digital-Pin 3 as output

pinMode(13, OUTPUT); //Output to reset slave

digitalWrite(iSignalPin, iLastHeartbeatStatus); //Needs per default set to HIGH

}

//Simple loop-function

void loop()

{

delay(200); // just wait a few seconds before starting next loop

unsigned long currentMillis = millis(); //Get current millis

if(!bSlaveStarting) //Slave is started

{

if(bCheckNewSignal) //Check if master need to get status from slave

{

if(digitalRead(iSlavePin) == iLastHeartbeatStatus)

{

Serial.println("Change: " + String(iCount) + " - Receive signal " + bool(iLastHeartbeatStatus));

bCheckNewSignal = false;

iCount++;

}

}

}

else //(Re-)starting mode from slave

{

if(digitalRead(iSlavePin) == iLastHeartbeatStatus) //So there must be something that sends signal

{

Serial.println("Slave online");

bSlaveStarting = false; //Switch to normal mode

previousMillis = millis(); //Overwrite previousMillis

```

    }
}

//Check if Master needs to change state and write to SignalPin
if(currentMillis - previousMillis >= int(HEARTBEAT_CHANGE_INTERVAL))
{
    if(!bSlaveStarting)
    {
        if(iLastHeartbeatStatus == LOW)
            iLastHeartbeatStatus = HIGH;
        else
            iLastHeartbeatStatus = LOW;

        digitalWrite(iSignalPin,iLastHeartbeatStatus);
        Serial.println("Change master signal to: " + String(bool(iLastHeartbeatStatus)));
        bCheckNewSignal = true;
    }
    previousMillis = currentMillis;
}

//Check if reboot from slave is needed or not
if(currentMillis - previousMillis >= int(HEARTBEAT_RECEIVE_INTERVAL) && bCheckNewSignal)
    ResetSlave(); //Reset slave and overwrite some vars
}

//Function to reset slave
void ResetSlave()
{
    Serial.println("Set RESET-Pin slave to LOW");
    digitalWrite(13, LOW); //Reset slave
    delay(500); //Wait 500ms, so slave restarts
    Serial.println("Set RESET-Pin slave to HIGH");
    digitalWrite(13, HIGH); //Lets start slave up
    previousMillis = millis(); //Overwrite previousMillisSlave
    bSlaveStarting = true; //Slave is restarting so switch to (re-)start mode
    iLastHeartbeatStatus = HIGH;
    bCheckNewSignal = false;
}

```

Code 10 zu sehen

```

// Heartbeat example 2 for master (Arduino Uno)
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 15. Sep 2020
// Update: 15. Sep 2020
//-----

#define HEARTBEAT_RECEIVE_INTERVAL 2000 //Maximum allowed time for normal change
#define HEARTBEAT_CHANGE_INTERVAL 3000 //Needed time for rebooting slave

int iCount = 0; //Global var for monitoring
unsigned long previousMillis = 0; //Stores current millis to switch Heartbeat-Signal

```

```

int iLastHeartbeatStatus = HIGH; //Last know status from slave
int iSlavePin = 2; //Pin to receive slave IO
int iSignalPin = 3; //Pin to set new flag
bool bSlaveStarting = true; //Bool to switch during "normal" mode and slave is (re-)starting
bool bCheckNewSignal = false;
//Init Arduino
void setup()
{
    Serial.begin(115200); // Active serial com with baudrate 115200
    Serial.println("Arduino (master) started ..."); //Say hello to the world
    pinMode(iSlavePin, INPUT); // Digital-Pin 2 as input}
    pinMode(iSignalPin, OUTPUT); //Digital-Pin 3 as output
    pinMode(13, OUTPUT); //Output to reset slave
    digitalWrite(iSignalPin, iLastHeartbeatStatus); //Needs per default set to HIGH
}

//Simple loop-function
void loop()
{
    delay(200); // just wait a few seconds before starting next loop
    unsigned long currentMillis = millis(); //Get current millis
    if(!bSlaveStarting) //Slave is started
    {
        if(bCheckNewSignal) //Check if master need to get status from slave
        {
            if(digitalRead(iSlavePin) == iLastHeartbeatStatus)
            {
                Serial.println("Change: " + String(iCount) + " - Receive signal " + bool(iLastHeartbeatStatus));
                bCheckNewSignal = false;
                iCount++;
            }
        }
    }
    else //(Re-)starting mode from slave
    {
        if(digitalRead(iSlavePin) == iLastHeartbeatStatus) //So there must be something that sends signal
        {
            Serial.println("Slave online");
            bSlaveStarting = false; //Switch to normal mode
            previousMillis = millis(); //Overwrite previousMillis
        }
    }

    //Check if Master needs to change state and write to SignalPin
    if(currentMillis - previousMillis >= int(HEARTBEAT_CHANGE_INTERVAL))
    {
        if(!bSlaveStarting)
        {
            if(iLastHeartbeatStatus == LOW)
                iLastHeartbeatStatus = HIGH;
            else
                iLastHeartbeatStatus = LOW;
        }
    }
}

```

```

    digitalWrite(iSignalPin,iLastHeartbeatStatus);
    Serial.println("Change master signal to: " + String(bool(iLastHeartbeatStatus)));
    bCheckNewSignal = true;
}
previousMillis = currentMillis;
}

//Check if reboot from slave is needed or not
if(currentMillis - previousMillis >= int(HEARTBEAT_RECEIVE_INTERVAL) && bCheckNewSignal)
    ResetSlave(); //Reset slave and overwrite some vars

}

//Function to reset slave
void ResetSlave()
{
    Serial.println("Set RESET-Pin slave to LOW");
    digitalWrite(13, LOW); //Reset slave
    delay(500); //Wait 500ms, so slave restarts
    Serial.println("Set RESET-Pin slave to HIGH");
    digitalWrite(13, HIGH); //Lets start slave up
    previousMillis = millis(); //Overwrite previousMillisSlave
    bSlaveStarting = true; //Slave is restarting so switch to (re-)start mode
    iLastHeartbeatStatus = HIGH;
    bCheckNewSignal = false;
}

```

Code 10: Code Master Heartbeat-Beispiel 2

In dieser Variante ist der Code vom Slave, der NodeMCU, recht schlank, siehe Code 11 und kann daher theoretisch bequem in eine eigene Funktion ausgelagert werden.

```

// Heartbeat example two for slave (ESP32)
// Autor: Joern Weise
// License: GNU GPI 3.0
// Created: 15. Sep 2020
// Update: 15. Sep 2020
//-----

int iCount = 0; //Global var for monitoring
int iLastHeartbeatStatus = LOW;
//Init ESP32
void setup()
{
    Serial.begin(115200); // Active serial com with baudrate 9600
    Serial.println("ESP started ..."); //Say hello to the world
    pinMode(14, OUTPUT); // Digital-Pin 14 as output
    pinMode(27, INPUT); // Pin 27 as input
    digitalWrite(14, LOW); //Init as with LOW SIGNAL
}

//Simple loop-function

```



```

void loop()
{
  if(digitalRead(27) != iLastHeartbeatStatus)
  {
    //Check if Input Pin 27 is different to iLastHeartbeatStatus
    iLastHeartbeatStatus = digitalRead(27); //Get new iLastHeartbeatStatus
    digitalWrite(14, iLastHeartbeatStatus); //Mirror and answer to master
    Serial.print(iCount);           //Print to SerialMonitor
    Serial.print(" - Set Heartbeat to ");
    Serial.println(iLastHeartbeatStatus);
    iCount++; //Increase counter
  }
  delay(200); //Short delay
}

```

Code 11: Code Slave Heartbeat-Beispiel 2

Auch in dieser Variante ist der Code mit reichlich Kommentaren und Ausgaben im seriellen Monitor geschrieben. Das Ergebnis für einen normalen Betrieb, aus Sicht vom Master, sehen Sie in Abbildung 16.

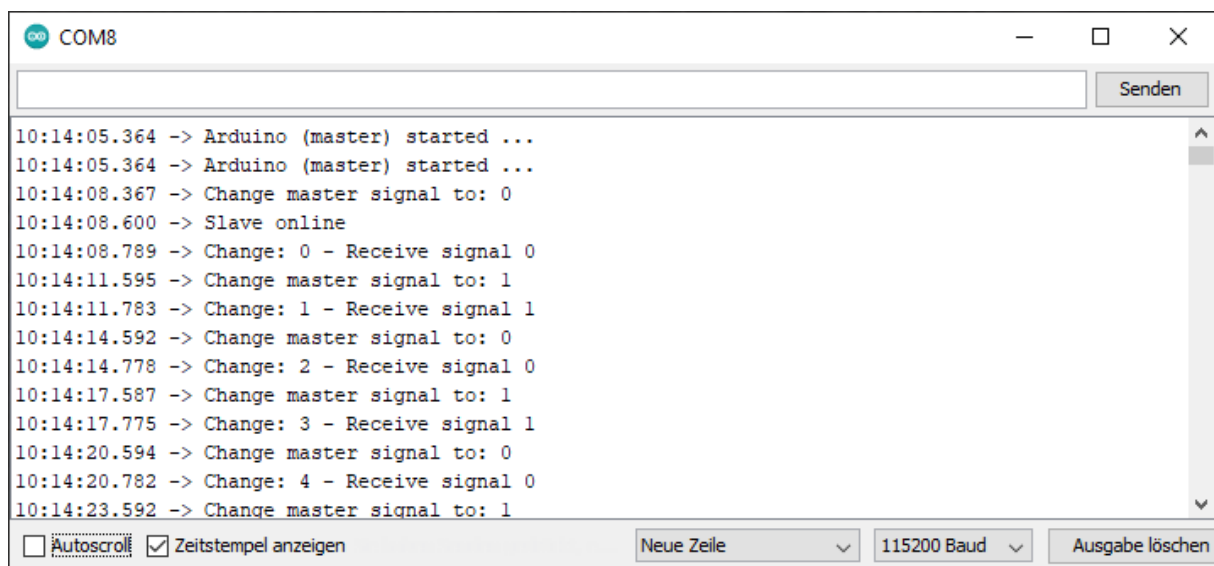


Abbildung 16: Ausgabe serieller Monitor für Master Heartbeat-Variante 2

Interessant ist aber auch wieder der Fehlerfall, siehe Abbildung 17, da der Fehler vom Master so leichter beschrieben werden kann.

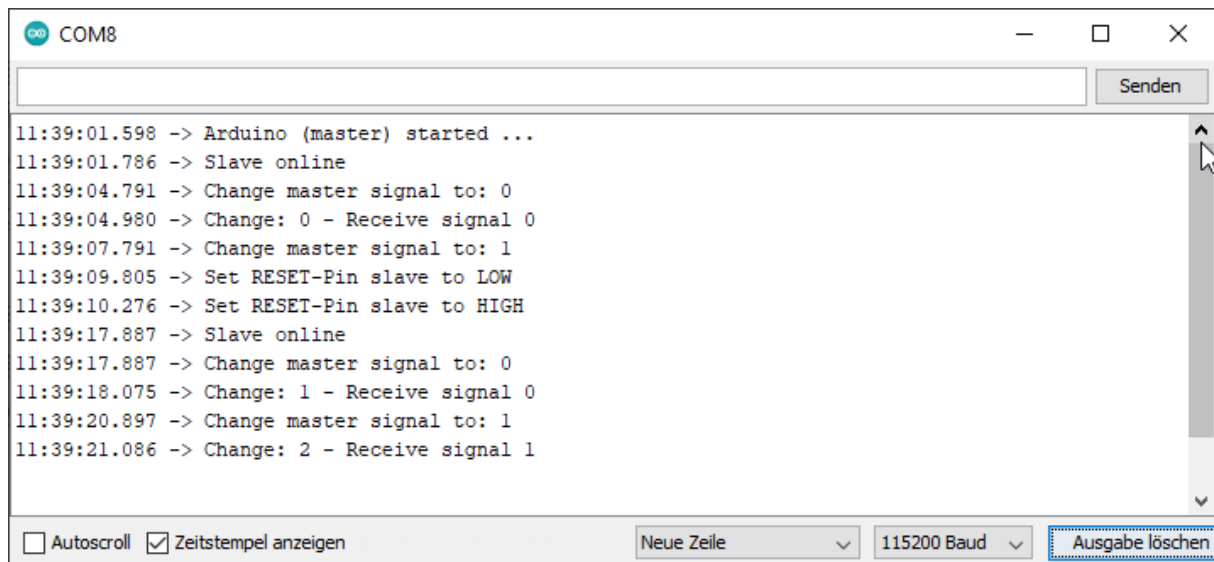


Abbildung 17: Ausgabe serieller Monitor für Master Heartbeat-Variante 2 mit Neustart vom Slave

Zunächst startet der Master, Arduino Uno, normal und setzt sein Signal auf HIGH. Dies ist wichtig, sollte es zu einem Defekt am Kabel kommen, wird das empfangene Signal immer LOW sein. Somit wartet der Master nun auf die korrekte Antwort vom Slave, dem NodeMCU. Da dieser das Signal vom Master auf den digitalen Pin 3 vom Master spiegelt, erhält der Master ein HIGH-Signal zurück, der Slave ist damit erreichbar und der normale Lauf beginnt. Der Master ändert nun in der vorgegebenen Zeit, hier alle 3 Sekunden, sein Signal und erwartet innerhalb von 2 Sekunden dasselbe Signal auf Pin3. Im Falle unseres Fehlers funktioniert dies beim ersten LOW-Signal noch. Beim Setzen vom HIGH-Signal seitens des Masters, kommt aber kein Signal zurück und der Master startet den Slave neu. Hierbei wird gleichzeitig das Master-Signal wieder auf HIGH gesetzt und der Timer zum Wechseln des Signals an den Slave ausgesetzt, bis der Slave wieder erreichbar ist. Nach 7 Sekunden ist der Slave wieder erreichbar und der normale Ablauf wird durchgeführt.

Fazit

Nachdem Sie nun den gesamten Blogbeitrag gelesen haben, sollten Sie ein Verständnis dafür bekommen haben, wie wichtig es sein kann, einen MicroController über einen Watchdog oder über den Heartbeat zu überwachen, um eine sichere Funktion zu gewährleisten. Natürlich ist nicht jede Variante für jedes Projekt geeignet und Sie als Programmierer müssen in beiden Fällen ermitteln, warum der MicroController nicht mehr ordnungsgemäß funktioniert. Je nach Aufgabe oder Umfang Ihres Projektes kann es sogar sein, dass Sie beide Varianten einsetzen können, das bleibt Ihnen aber überlassen. Sie sehen jedoch selbst, wie mit wenigen Zeilen Code eine Überwachung möglich ist. Im Falle vom Heartbeat müssen Sie nicht die hier vorggeführte Variante mit den digitalen Pins wählen, sondern können über die serielle Schnittstelle eine Art Protokoll entwickeln. Da dies aber den Rahmen vom Blogbeitrag deutlich gesprengt hätte, haben Sie die wohl einfachste Variante kennengelernt.

Weitere Projekte für AZ-Delivery von mir finden Sie unter <https://github.com/M3taKn1ght/Blog-Repo>.