

CE31 Embedded Systems

Laborprojekt I²C-Kommunikation

Gruppe 6

Kevin Hübner | 570746
Tobit Zander | 582020

Inhaltsverzeichnis

1 Projekt Idee	3
2 Bauteilliste	3
3 Umsetzung	4
3.1 Temperatur- und Luftfeuchtigkeitssensor (Tobit Zander)	4
3.1.1 Sensor verstehen	4
3.1.2 Sensor auslesen	5
3.1.3 Erfassung von Sensordaten	5
3.1.4 Messen der <i>High</i> Zeitenlängen	7
3.1.5 Timeouts erkennen	9
3.1.6 Masterseitiges I ^C	10
3.1.7 Projekt Implementierung	12
3.2 4-stellige 7-Segment-Anzeige (Kevin Hübner)	21
3.2.1 Grundlagen der 7-Segment-Anzeige	21
3.2.2 Erste Inbetriebnahme der 7-Segment-Anzeige	23
3.2.3 I ^C Anpassungen	26
3.2.4 Zusammenführung von Display und I ^C	27
3.3 Zusammenführung beider Systeme	29
4 Probleme	30
5 Fazit	30

Abbildungsverzeichnis

1 Übertragungsprotokoll des AM2302 (ONE-WIRE)	5
2 Senden eines Startsignals und Werte vom Sensor	5
3 Erfassen der Zeitenlängen; Pufferausschnitt	7
4 1. Beispiel gemessener Zeiten, Logic Analyzer	7
5 2. Beispiel gemessener Zeiten, Logic Analyzer	7
6 3. Beispiel gemessener Zeiten, Logic Analyzer	7
7 Messen der High Zeitenlängen; Pufferausschnitt	8
8 Flankenerkennung des Nucleos	9
9 Timer Interrupt	10
10 Messung der Anstiegsdauer bei 2KOhm PullUp Widerstand	11
11 Testaufbau I ^C	11
12 Beispielübertragung einer 0 und des Arrays <code>test []={123, 112, 168, 300};</code>	12
13 Flussdiagramm: Hauptschleife	14
14 Flussdiagramm: fertiger Datensatz	15
15 Flussdiagramm: Fehlerbehandlungen	16
16 Flussdiagramm: Interrupt-Servie-Routinen des Timers und des GPIOs	17
17 Zyklus beim Überlauf	19
18 Datenübertragung bei der 7-Segment-Anzeige	21
19 7-Segment-Anzeige	22
20 Start-Befehl über die Flanken des Nucleo-Boards	23
21 Stop-Befehl über die Flanken des Nucleo-Boards	24
22 CLK-Cyklus	25
23 Logic Analyzer für das anschalten des Displays	25
24 Angeschaltetes Display	26

25	I2C Ergebnis der Kommunikation über Polling (erster Datensatz)	27
26	I2C Ergebnis der Kommunikation über Interrupt	27
27	Erfolgreiches Verbinden beider Nucleos	29
28	I2C Clock wird vom Slave auf <i>Low</i> gezogen	30
29	I2C Slave reagiert nicht	30

Tabellenverzeichnis

Liste von Code-Ausschnitten

1	Der <code>data_t</code> Datentyp	18
2	Nutzung des <code>data_t</code> Datentyps	18
3	Blockieren des Startsignals	19
4	Funktion zum Überprüfen des Acknowledgements	24
5	Double Dabble Funktion	28

1 Projekt Idee

In diesem Labor werden zwei Nucleo-Boards miteinander verbunden, um Daten über das I²C-Protokoll (oder auch I2C) auszutauschen.

Ein Board fungiert dabei als Master und das andere Board als Slave.

Das Master-Board ist mit einem Temperatur- und Luftfeuchtigkeitssensor (AM2302) ausgestattet. Dabei misst dieses Board in einem 5 Sekundentakt die aktuellen Werte ¹ und sendet abwechselnd Luftfeuchtigkeit und Temperatur über die I²C-Schnittstelle zum Slave.

Das andere Nucleo-Board ist mit einem Modul ausgestattet, das aus vier sieben-Segment-Anzeigen (TM1637) besteht. Diese Anzeigen dienen der Darstellung der empfangenen Daten. Das Board stellt dabei entweder Luftfeuchtigkeit oder Temperatur dar, jenachdem welchen Wert es über das I2C-Protokoll vom ersten Board erhält.

In diesem Labor wird Schritt für Schritt durch den Entwicklungsprozess geführt und dabei aufgetretene Probleme näher erläutert.

2 Bauteilliste

Folgende Bauteile kamen für dieses Projekt zum Einsatz:

Auf Seiten der Darstellung:

- Nucleo-F042K6
- TM1637 ² (4-stellige 7-Segment-Anzeige)

Auf Seiten der Datenerhebung:

- Nucleo-L432KC
- AM2302 ³ (Temperatur- und Luftfeuchtigkeitssensor)

Zum Verbindungsaufbau zwischen den beiden Mikrocontrollern:

- 2x 2k Ohm (als PullUp-Widerstände)

Weiterhin wurden während des Entwicklungsprozesses zusätzliche Hilfsmittel eingesetzt:

- Logic Analyzer
- Arduino (hier ein NANO EVERY)
- Handprobe HP3 (digital Oszilloskop)

¹ In einem Datensatz liegen sowohl die Luftfeuchtigkeits- als auch die Temperaturwert. Es gibt keine Möglichkeit die Werte einzeln zuerfragen.

² TM1637 Datenblatt: https://cdn.shopify.com/s/files/1/1509/1638/files/4-Bit_7-Segment_LED_Display_Datenblatt_AZ-Delivery_Vertriebs_GmbH.pdf

³ AM2302 Datenblatt: https://files.seedstudio.com/wiki/Grove-Temperature_and_Humidity_Sensor_Pro/res/AM2302-EN.pdf

3 Umsetzung

Im folgenden wird nun die Umsetzung des Projektes detailliert beschrieben.

Das gesamme Projekt liegt dabei versionsverwaltet auf dem HTW eingenen GitLab-Server unter <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh>. Dort sind auch, unter der README-Datei im Root-Verzeichnis,⁴ alle Informationen zum Bauen der Projekte erhältlich.

3.1 Temperatur- und Luftfeuchtigkeitssensor (Tobit Zander)

3.1.1 Sensor verstehen

Bevor mit der Implementierung begonnen werden konnte, wurde zunächst das Datenblatt⁵ des Luftfeuchtigkeits- und Temperartursensor (AM2302) herausgesucht und evaluiert.

Der für dieses Projekt genutzte Sensor nutzt ein *ONE-WIRE* Protokoll zur Übertragung der Daten. Dieses Protokoll sieht dabei lediglich nur eine Datenverbindung vor. Um allerdings dennoch eine *bidi-rektionale* Verdingung zwischen Mikrocontroller und Sensor bereitstellen zu können, kommt das sog. *wired AND* zum Einsatz.

Das heißtt, alle Teilnehmner schalten die Datenpins *hochohmig*. Ein benötigter *Pull up-Widerstand* zieht dann den Pegel automatisch auf *High*, solange alle Beteiligten ihre Pins hochohmig gestellt haben.

Sobald eine Null (entspricht dem *Low*-Pegel) von einem der beiden (Mikrocontroller oder der Sensor) übertragen werden soll, kann die Busleitung aktiv auf *Low* gezogen werden. Eine Eins entsteht wieder von selbst, wenn derjenige den Datenpin auf hochohmig stellt und die PullUp-Widerstände die Leitung auf *High* ziehen.

Dieses Prinzip wird auch vom I²C Bus genutzt. Und deshalb die beiden 2k Ohm Widerstände aus der Bauteilliste.

Das Senden der Messwerte durch den Sensor wird durch das Senden eines *Startsignal* getriggert (siehe T_{be} in Abb. 1). Nach Freigabe des Busses (T_{go}) durch den Mikrocontroller folgt eine *Low-* (T_{rel}) und dann wieder eine *High-Phase*(T_{reh}) und daraufhin beginnt die eigentliche Datenübertragung durch den Sensor.

Die Datenwerte werden dabei *binär Kodiert*. Die *1* und *0* werden allerdings nicht durch direckte Spannungspegel definiert (1 entspricht nicht direkt High und 0 nicht automatisch Low), sondern durch die *Länge der High Zeit*. Ist also der Pegel für 22µs bis 30µs High, so entspricht dies einer 1 und für eine 0 muss der Pegel für 68 bis 75µs auf High sein.

⁴ <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/blob/main/README.md>

⁵ https://files seedu studio.com/wiki/Grove-Temperature_and_Humidity_Sensor_Pro/res/AM2302-EN.pdf

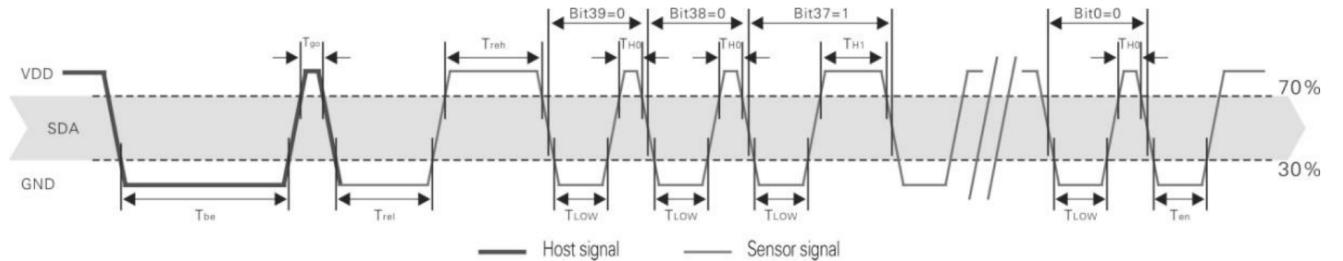


Abb. 1: Übertragungsprotokoll des AM2302 (ONE-WIRE)

Der gesamte Datensatz wird über diese Kodierung übertragen.

Ein Datensatz umfasst 40 Bits, die zwei oberen Bytes stellen dabei die Luftfeuchtigkeit dar, Byte 3 und 4 beschreibt die Temperatur und das 5 Byte wird für Parität genutzt; zur Gültigkeit des Datensatzes muss die Addition aus den ersten vier Bytes das Paritätsbyte ergeben.

Anhand dieser Erkenntnisse wurde zunächst ein Testprojekt erstellt, in dem grundlegende Operationen getestet wurden.

3.1.2 Sensor auslesen

Zu Beginn wurde zunächst ein Test durchgeführt, welcher dem Sensor ein einfaches Startsignal schickt, um das senden eines Datensatzes zu triggern.

Dazu wurde ein Pin auf *Low* gesetzt. Anschließend für 18ms gewartet und danach auf *Input* umgestellt. Die Umstellung auf *Input* bewirkt, dass der Pin in einen *Hochohmigen* Zustand gesetzt wird, in dem er neutral im Stromkreis ist. Durch den PullUp-Widerstand, der auf der Sensor-Platiene schon angebracht ist, ändert der Pegel auf *High*.

Mit Hilfe eines Logic Analyzer konnte dann der darauffolgende Datensatz (siehe Abb. 2) manuell entzifert und in Luftfeuchtigkeit und Temperatur umgerechnet werden, ebenso konnte auch das Paritätsbyte dieses Datensatzes verifiziert werden.

Hier der entsprechende Commit⁶.

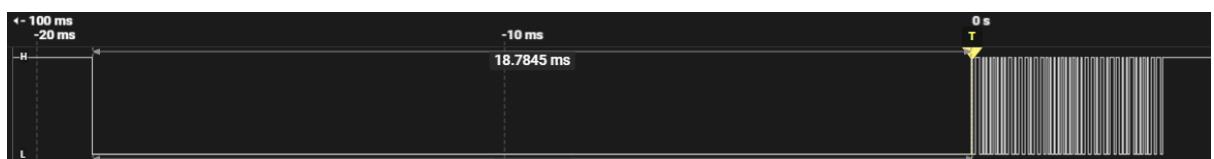


Abb. 2: Senden eines Startsignals und Werte vom Sensor

3.1.3 Erfassung von Sensordaten

Nach erfolgreicher Triggern der Datenbereitstellung durch den Sensors, wurde im anschließenden Schritt mit der Erfassung und Speicherung der Daten durch den Mikrocontroller begonnen.

⁶ https://gitlab.rz.hwt-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/blob/b6c98919b875442ec13493a7b8098ecf4a4c31c1/Tobit/test-pr/Core/Src/_main2.c

Da die Einsen und Nullen nicht über Spannungspiegel definiert sind, sondern über die zeitliche Länge des High-Pegels, wurde für diese Aufgabe ein *Hardware-Timer* eingesetzt.⁷ Dieser Timer (oder dt. Zähler) startet, sobald eine Flanke (zunächst steigend oder fallend) am Datenbus anliegt. Bei einer folgenden Flanke wird der Zählerwert in einen *Puffer* gespeichert und der Zähler zurück auf Null gesetzt.

Der Timer zählt mit einer Geschwindigkeit von 1MHz (Clock speed: 32 MHz und Prescaler: 32). Das Timer-Register hat 16 Bit, der Timer kann also etwa für 65ms Zählen, bis es zu einem Überlauf des Registers kommt. Die Größe des Registers ist somit außreichend groß.

An der Stelle, an welcher im [Abschnitt 3.1.2](#) das GPIO auf Input-Mode gestellt wurde, wurde in diesem Abschnitt ein Pin auf *External Interrupt Mode with Rising/Falling edge trigger detection* gestellt.

Damit können alle Flankenänderungen recourcensparend und zuverlässig behandelt werden. Ein Pin (oder GPIO) in solch einem Modus ist (wie ein Pin im Input-Modus) hochohmig und dadurch neutral im Stromkreis.

Nach Ausführung des Programms⁸ im *Debug-Modus* konnten die Werte aus dem Puffer (siehe [Abb. 3](#)) mit den Werten des Logic Analyzers (siehe [Abb. 4](#), [Abb. 5](#), [Abb. 6](#)) verglichen werden und folgende Schlüsse gezogen werden:

- Das erste und zweite Element (0 und 13) sind kein Bestandteil des eigentlichen Datensatzes; beide sind in jedem Datensatz enthalten der bisher getestet wurde.
- Die 13 (buf[1]) entsteht vermutlich daher, weil von dem Zeitpunktes des Timer-Start bis zur ersten steigenden Flanke schon Zeit verstrichen ist.
- Die gepufferten Zeiten sind immer etwas kleiner (4 bis 6µs), als die gemessenen Zeiten des Logic Analyzers:

Im Puffer steht z.B. der Wert 18, gemessen wurde allerdings 22µs (siehe [Abb. 4](#)) oder gesessen 78µs (siehe [Abb. 5](#)) und im Puffer steht 72µs oder es steht im Puffer 22µs aber es wurden 26µs ([Abb. 6](#)) gemessen.

Vermutlich entsteht dieses Phänomen durch den zeitlichen Versatz des Auslesens vom Timerwert und dem zurücksetzen des Wertes auf 0.

Allerdings ist dieses Problem ohne Bedeutung, da die Unterschiede zwischen gemessener und gepufferter Zeit noch gut in der Tolleranz liegen.

⁷ Alle benötigten Informationen zum Einrichten und Auslesen von Timern wurden aus Laboraufgaben aus früheren Semestern entnommen.

⁸ Hier der entsprechende Commit: <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/blob/7e12f31a90cf9912a9331d4b3186aa174c0ed062/Tobit/test-pr/Core/Src/main.c>

buf	uint8_t *[86]	0x200000f8 <buf>
> buf[0]	uint8_t *	0 (Decimal)
> buf[1]	uint8_t *	13 (Decimal)
> buf[2]	uint8_t *	18 (Decimal)
> buf[3]	uint8_t *	72 (Decimal)
> buf[4]	uint8_t *	76 (Decimal)
> buf[5]	uint8_t *	49 (Decimal)
> buf[6]	uint8_t *	23 (Decimal)
> buf[7]	uint8_t *	49 (Decimal)
> buf[8]	uint8_t *	22 (Decimal)
> buf[9]	uint8_t *	49 (Decimal)
> buf[10]	uint8_t *	23 (Decimal)
> buf[11]	uint8_t *	49 (Decimal)
> buf[12]	uint8_t *	22 (Decimal)
> buf[13]	uint8_t *	49 (Decimal)
> buf[14]	uint8_t *	23 (Decimal)
> buf[15]	uint8_t *	49 (Decimal)
> buf[16]	uint8_t *	22 (Decimal)
> buf[17]	uint8_t *	49 (Decimal)
> buf[18]	uint8_t *	23 (Decimal)
> buf[19]	uint8_t *	48 (Decimal)

Abb. 3: Erfassen der Zeiten; Pufferausschnitt

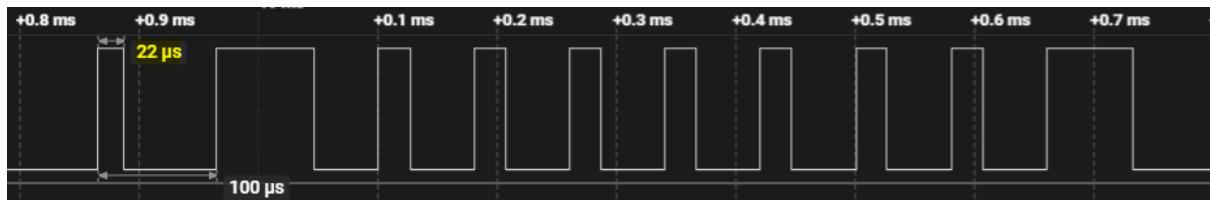


Abb. 4: 1. Beispiel gemessener Zeiten, Logic Analyzer

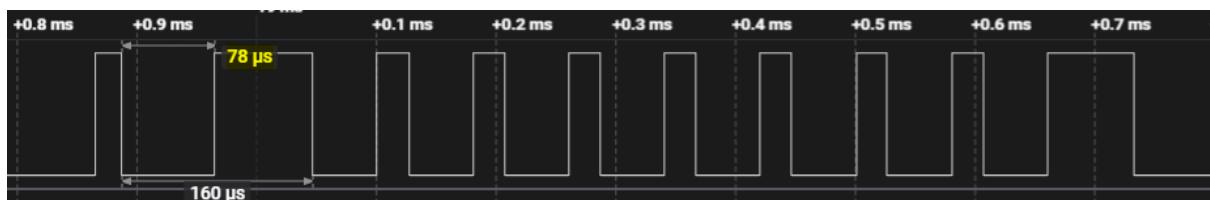


Abb. 5: 2. Beispiel gemessener Zeiten, Logic Analyzer

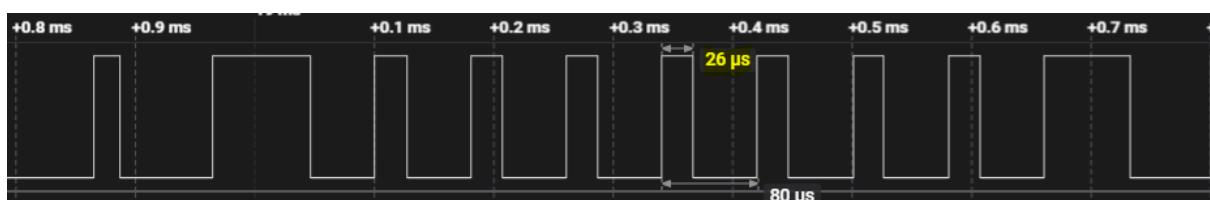


Abb. 6: 3. Beispiel gemessener Zeiten, Logic Analyzer

3.1.4 Messen der *High* Zeitlängen

Bisher wurden alle Zeitlängen, also Verschrichene Zeit im Low Pegel und im High Pegel, im Puffer gespeichert. Die Daten werden allerdings nur über die zeitliche Länge des High-Pegels definiert (Kurzer

High Pegel: 0, langer High Pegel: 1).

In diesem Schritt wurde zum herausfiltern der Low Zeiten, dazu die *Interrupt service routine (ISR)* angepasst.

Vorher wurde bei **jeder** Flanke die Zeit abgelesen, im Puffer gespeichert und der Timer zurückgesetzt, hier wird bei jeder **steigenden** Flanke der Zähler gestartet (Anfang einer High-Phase) und bei einer **fallenden** Flanke (High-Phase beendet) die verstrichene Zeit abgespeichert und der Timer auf 0 zurückgesetzt.

Um die jeweilige Flanke zu erkennen wird über *if-Abfragen* der Pegel getestet. Ist der anliegende Pegel High, so trat eine steigende Flanke auf und wenn der anliegende Pegel Low ist, lag eine fallend Flanke vor.⁹

Im folgenden der Pufferausschnitt, wenn alle Low-Phasen herausgefiltert sind (siehe Abb. 7).

▼ buf	uint8_t *[150]	0x200000f8 <buf>
▼ [0...99]	uint8_t *[100]	0x200000f8 <buf>
> buf[0]	uint8_t *	12 (Decimal)
> buf[1]	uint8_t *	77 (Decimal)
> buf[2]	uint8_t *	24 (Decimal)
> buf[3]	uint8_t *	23 (Decimal)
> buf[4]	uint8_t *	24 (Decimal)
> buf[5]	uint8_t *	23 (Decimal)
> buf[6]	uint8_t *	24 (Decimal)
> buf[7]	uint8_t *	23 (Decimal)
> buf[8]	uint8_t *	24 (Decimal)
> buf[9]	uint8_t *	69 (Decimal)
> buf[10]	uint8_t *	70 (Decimal)
> buf[11]	uint8_t *	70 (Decimal)
> buf[12]	uint8_t *	24 (Decimal)
> buf[13]	uint8_t *	70 (Decimal)
> buf[14]	uint8_t *	23 (Decimal)
> buf[15]	uint8_t *	24 (Decimal)
> buf[16]	uint8_t *	69 (Decimal)
> buf[17]	uint8_t *	23 (Decimal)

Abb. 7: Messen der High Zeitspannen; Pufferausschnitt

In diesem Schritt ist die führende Null nun nicht mehr vorhanden.

Das folgende Bild veranschaulicht, wie die 12µs Zustände kommen (siehe Abb. 8), obwohl diese nicht zum eigentlichen Datensatz gehören.

Nach dem der Timer startet, wird zunächst noch der Interrupt für das Pin konfiguriert. Anschließend wird die steigende Flanke erkannt, welche den Timer zurücksetzt und daraufhin die fallende Flanke, welche den Timerwert abspeichert. Diese 14.5µs entsprechen der 12, welche in den Puffer geschrieben wurde.

⁹ Hier der zugehörige Commit: <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/blob/14fc2910820f579252b00a05db28e5292ac06d0a/Tobit/test-pr/Core/Src/main.c>



Abb. 8: Flankenerkennung des Nucleos

Mit diesen Ergebnissen kann nun der erste Schritt, die Sensordaten auslesen und abspeichern, abgeschlossen werden.

Die Daten aus dem Puffer müssen nur noch zwischen lang (entspricht einer 1) und kurz (entspricht einer 0) aufgeteilt werden.

3.1.5 Timeouts erkennen

Mit dem vorangegangenen Abschnitt wurde das Lesen eines Datensatzes im *Happy-Day-Szenario* beendet, das heißt, das Lesen des Datensatz verläuft ohne Zwischenfälle.

In diesem Abschnitt wurde während des Entwicklungsprozesses der Fall betrachtet, wenn die Übertragung abbricht, das heißt: nicht alle 40 Bits sind vollständig.

Dazu wurde der Timer als ausschlaggebendes Kriterium genutzt.¹⁰ Grundsätzlich wird bei jeder steigenden Flanke der Timer auf Null zurückgesetzt. Das Zurücksetzen verhindert somit das Überlaufen des Timer-Registers. Die Idee: wenn der Timer überläuft, kam keine steigende Flanke, die den Timer zurückgesetzt hätte und so den Überlauf verhindert. Da allerdings der Timer im hypothetischen Fall übergelaufen ist, liegt ein Fehler vor.

Somit kann das Überlaufen als notwendiges Kriterium angesehen werden, Timeouts während der Übertragung festzustellen.

Einen Timerübergang wird am Besten mithilfe eines Interrupts ermittelt. Da allerdings der bisher genutzte Timer nicht den globalen Interrupt unterstützt hatte, wurde zu einem anderen Timer (nun TIM7) gewechselt. Abgesehen von der Interruptaktivierung wurde zusätzlich noch die Periodenlänge auf 200µs reduziert. Aufgrund dieser Reduzierung zählt der Timer nun nur noch bis maximal 200, wodurch es schneller zu einem Überlauf kommt. Der Vorteil davon ist, dass nun schneller auf das Timeout reagiert werden kann.

Hier gab es allerdings das Problem, dass immer direkt nach dem Starten des Timer-Interrupts der Interrupt auslöste. Auf Stackoverflow wurde dafür jedoch eine Lösung gefunden.¹¹

Wie im folgenden Bild (Abb. 9) ersichtlich, scheint diese Methode ein aussagekräftiges Kriterium darzustellen, um ein unerwartetes Beenden der Übertragung festzustellen. Denn bei jeder steigenden Flanke wird der Timer auf Null zurückgesetzt (entspricht einem Toggeln des orangenen Pegels). Nach der letzten steigenden Flanke des Datensatzes tritt nach etwas mehr als 200µs eine Pegeländerung (im Bild

¹⁰ Hier der zugehörige Commit: <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/blob/802b78e46e3046979759920bed217532faf7c814/Tobit/test-pr/Core/Src/main.c>

¹¹ <https://stackoverflow.com/questions/71099885/why-hal-tim-periodelapsedback-call-backs-is-called-immediately-after-hal-tim-base-sta>

orange dargestellt) auf, diese ist dem Auslösen des Interruptes zuzuscheiben (siehe Quellcode). Im Falle eines tatsächlichen Abbruchs würden an dieser Stelle Maßnahmen ergriffen.

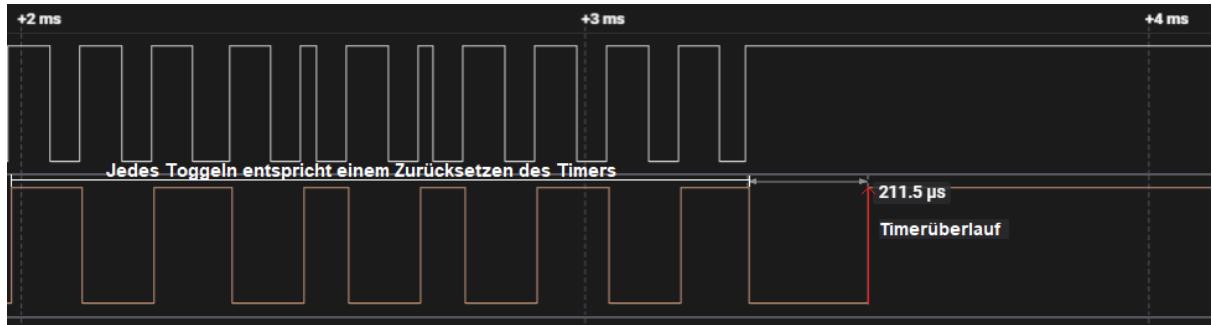


Abb. 9: Timer Interrupt

oben: Übertragung des Datensatzes, unten: jedes zurücksetzen des Timers ist ein Toggeln des Pegels

3.1.6 Masterseitiges I²C

¹² Ein letzter wichtiger Punkt, der noch zu testen gilt, stellt die Übertragung der Daten auf das zweite Nucleo-Board dar.

In diesem Projekt wurde eine I²C basierte Kommunikation gewählt. Die masterseitige I²C Kommunikation wurde in diesem Abschnitt des Erarbeitungsprozesses ausgetestet.

Um die Übertragung aussagekräftig Überprüfen zu können, wurde an dieser Stelle ein weiterer Mikrocontroller benötigt, um die Slave-Seite zu simulieren.

Da dem Entwickler allerdings kein zweites Nucleo-Board verfügbar war, wurde als Alternative hier ein Arduino NANO EVERY eingesetzt.

Auf Seiten des Arduinos wurde die Standardsoftware *slave_receiver* aus der *wire-Library* genutzt, um alle ankommenden Daten über den *Serial Monitor* auszugeben.

Für den Nucleo wurde im Codegerüstgenerator das I2C1-Interface aktiviert.

An diesen Punkt wurde sich mit dem Kollegen (siehe [Abschnitt 3.2](#)) auf folgende Schnittstelle geeinigt: Standart-Geschwindigkeit (100 KHz) und 7-Bit Adressen. ¹³

Als PullUp Widerstände werden zwei 2KOhm Widerstände eingesetzt, die Anstiegsflanke wurde mittels Oszilloskop gemessen (siehe [Abb. 10](#)).

Anhand des Bildes scheint es, dass 2kOhm ein guter Kompromiss zwischen Anstiegszeit (etwa 1µs) und Stromverbrauch ($I = \frac{3v_3}{2\text{k}Ohm} \approx 1\text{mA}$) ist.

¹² Im folgenden der entsprechende Commit: <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embed-gruppe-6-tz-kh/-/blob/51545de28ded9a42c2e88143d0d02d6806328ec1/Tobit/test-pr/Core/Src/main.c>

¹³ Eine detailliertere Schnittstellenbeschreibung unter: <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embed-gruppe-6-tz-kh/-/blob/main/README.md>

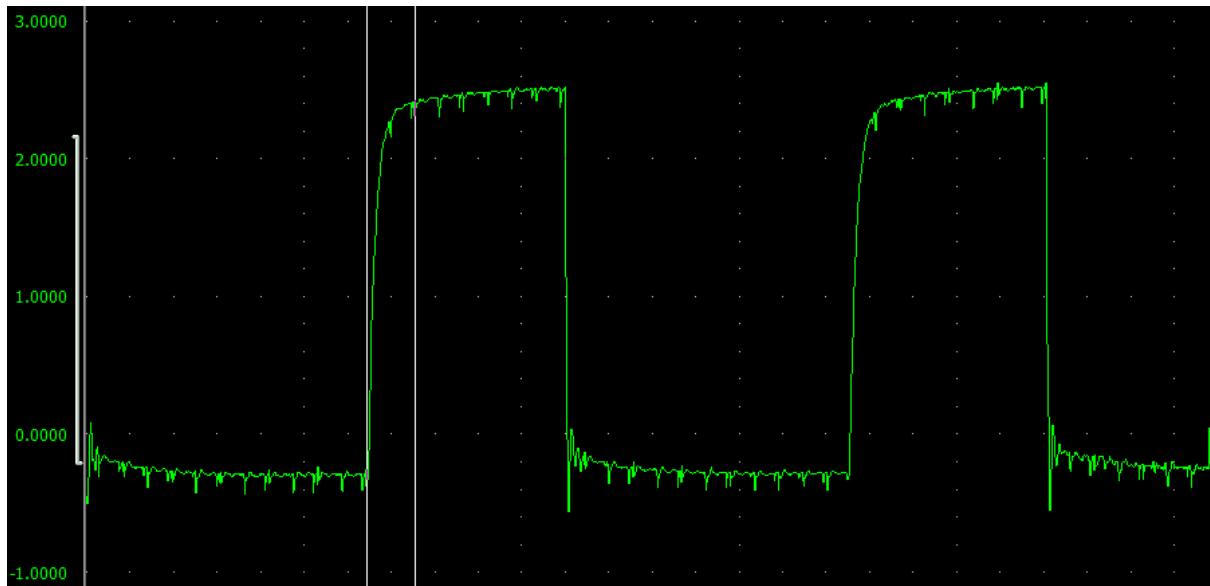


Abb. 10: Messung der Anstiegsdauer bei 2KOhm PullUp Widerstand
zwischen den beiden Cursor liegt 1μs

Im folgenden der I²C-Testaufbau (siehe Abb. 11).

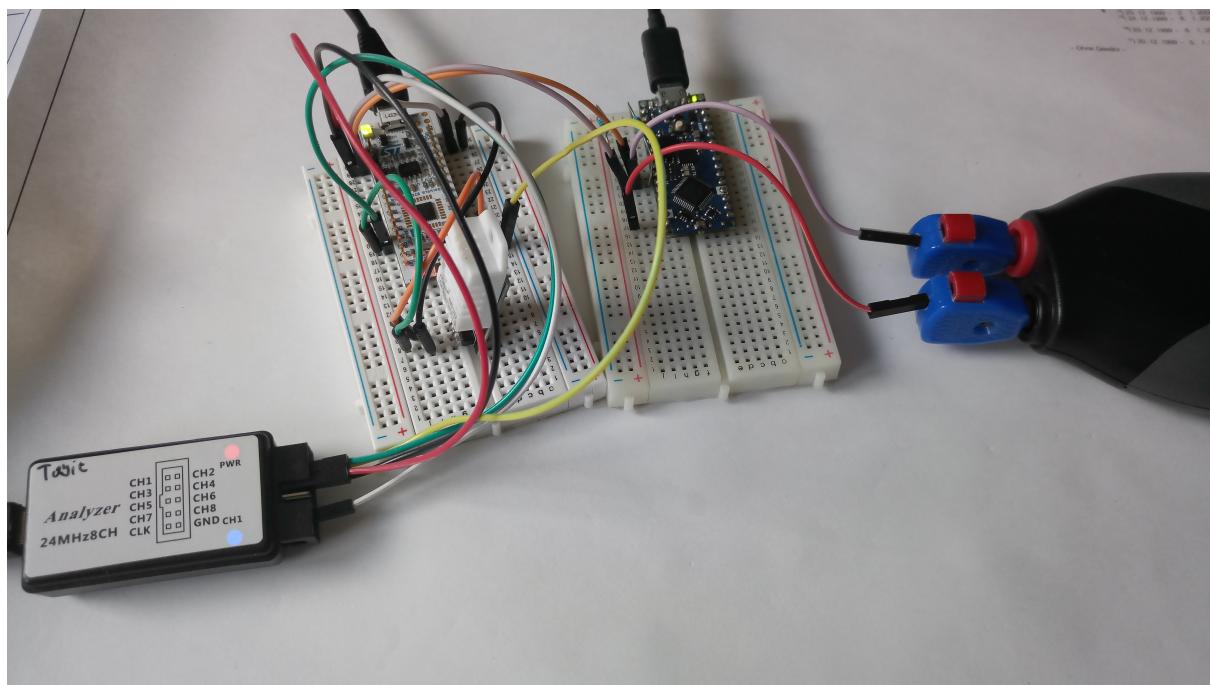


Abb. 11: Testaufbau I²C
links: Nucleo als Master, rechts: Arduino als Slave

Als I²C-Test wurde unter anderem ein Array aus 4 *uint16_t* Elementen übertragen (siehe Abb. 12).

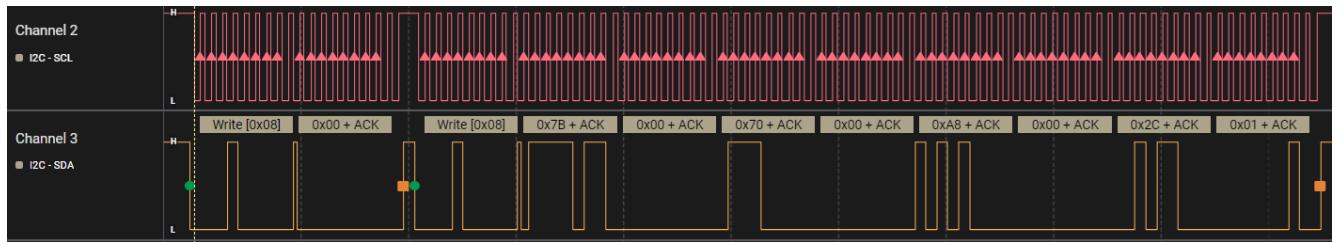


Abb. 12: Beispielübertragung einer 0 und des Arrays `test [] = {123, 112, 168, 300};`

Ein Problem trat allerdings dabei auf: Um den Slave mit der Adresse 8 anzusprechen, war es nötig 16 als Adresse anzugeben, also das Doppelte bzw. ein Bitschift nach links.

Mit dem erfolgreichen Senden von Beispieldaten konnten die vorbereitenden Test abgeschlossen und die finale Entwicklung gestartet werden.

3.1.7 Projekt Implementierung

Um eine strukturierte und logische Implementierung anfertigen zu können, wurde zunächst ein *Flussdiagramm* angefertigt, welches im Laufe der Erarbeitung an einigen Stellen nachgebessert wurde.

Die Grundstruktur ist eine Zustandsmaschine mit sechs Zuständen: *ready*, *measuring*, *finished*, *processing*, *invalid* und *error*. Der Initialzustand ist dabei *ready*.

Das System beginnt mit einem StartUp (siehe Abb. 13), in welchem unter anderem die I²C Verbindung überprüft wird. Der Master-Nucleo sendet dabei einen Konstanten Wert an den Slave-Nucleo, welcher wiederum mit einem vordefinierten Wert reagiert. Damit wird sowohl eine bestehende Verbindung verifiziert, aber es dient auch zur Authentifizierung des Slaves.

Kommt das System durch den StartUp, folgt die *Hauptschleife*.

In der *Hauptschleife* rotiert das System dauerhaft (ohne Abbruch) und überprüft, auf welchen Zustand das System gesetzt wurde. Abhängig vom Zustand wird dann die entsprechende Funktion aufgerufen.

Der **ready**-Zustand (siehe Abb. 13) gibt die Freigabe um eine Messung durchzuführen. Hierbei wird zunächst überprüft, ob die vorangegangene Messung älter als 2 Sekunden ist, da sonst der Sensor fehlerhafte Daten ausgibt.

Ist die vorangegangene Messung tatsächlich älter als 2 Sekunden, wird der Status auf **measuring** gesetzt. Nun folgt das Senden eines Startsignals, wie es zuvor im Test-Projekt geschehen war (Abschnitt 3.1.2). Das Aktivieren der Interrupts (Timer und GPIO) und das Setzen des Status *measuring*, sind die Türöffner für den eigentlichen Prozess des Einlesens des Datensatzes.¹⁴ (siehe Abb. 16).

Potentielle Fehlerfälle (Überschreiten von Zeittolleranzen oder die Anzahl der Bits) werden mit dem entsprechenden Statuswechsel signalisiert und in der nächsten Hauptschleife bearbeitet.

Beispielsweise sorgt das Verbleiben einer Datenflanke während eines Datensatzes das Überlaufen des TIM7-Registers. Aus diesem Ereignis lässt sich schließen, dass wahrscheinlich keine Verbindung zum Sensor besteht (siehe Abschnitt 3.1.5), weswegen die Statusvariable auf Error 2 gesetzt wird. Dieser Error-Code steht für "Keine Verbindung vom Sensor".

¹⁴ Dieser Prozess wurde bereits im Abschnitt Abschnitt 3.1.4 erläutert.

Bei erfolgreichem Lesen und Abspeichern des empfangenen Datensatzes wird schließlich auch die Statusvariable auf *finished* umgestellt.

Mit dem Status **finished** (siehe Abb. 14) werden die jüngst empfangenen Zeitwerte in Bits umgerechnet (siehe Abschnitt 3.1.4) und validiert. Anders als im Zustand *measuring*, werden hier die Zeilängen der einzelnen Bits und das Paritätsbyte überprüft. Im Fehlerfall wird der Zustand auf *invalid* gesetzt.

Nach erfolgreicher Validierung blockiert das System nun und wartet solange, bis 5 Sekunden seit der letzten Datenübertragung zum Slave vergangen sind.

Anschließend folgt das Senden der Daten; es wird abwechselnd die Luftfeuchtigkeit oder die Temperatur gesendet. Ebenso wird der *invalid-Zähler*, welcher die Anzahl der bisher aufgetretenen invalid-Zustände speichert, zurückgesetzt und der Zustand wieder auf *ready* gesetzt, um den nächsten Datensatz zu erfassen.

Trat der Zustand **invalid** (siehe Abb. 15) auf, wird zunächst über das Eintreten des invalid-Zustands mit der *Statusled* informiert. Der *invalid-Zähler* wird inkrementiert. Trat dieser Zustand schon zuhäufig auf, wird der Fehlercode 3 über I²C gesendet und auf der Sieben-Segment-Anzeige dargestellt. Der invalid-Zähler wird erst bei einem erfolgreichen *processing* zurückgesetzt. Am Ende dieses Zustands wird die Statusvariable auf *valid* gesetzt, um erneut eine Messung zu versuchen.

Auch im **error**-Zustand wird der entsprechende Fehlercode über die *Statusled* angezeigt, allerdings über ein anderes Blinksignal. Wenn eine Verbindung zum Nucleo mit Sieben-Segment-Anzeige besteht, wird auch hier der Fehlercode übertragen. Da es sich bei Fehlercode 1 und 2 ¹⁵ um Fehler handelt, die nicht unbedingt durch das erneute versuchen einer Messung gelöst werden können, wird beim Error-Status der Nucleo *rebootet*.

¹⁵ Code 1 steht für keine Verbindung zum I²C-Slave und Code 2 steht für keine Verbindung zum Sensor.

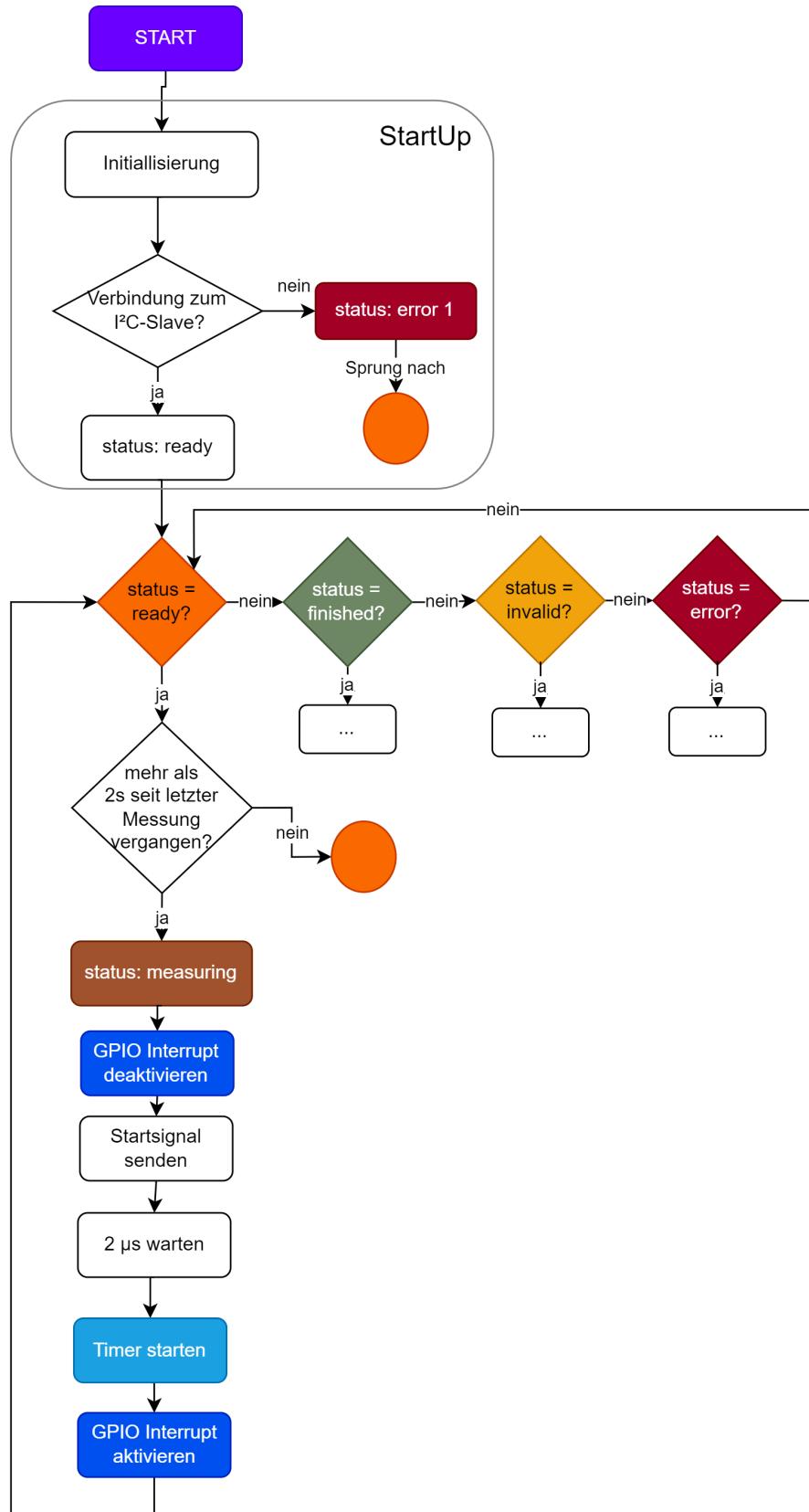


Abb. 13: Flussdiagramm: Hauptschleife

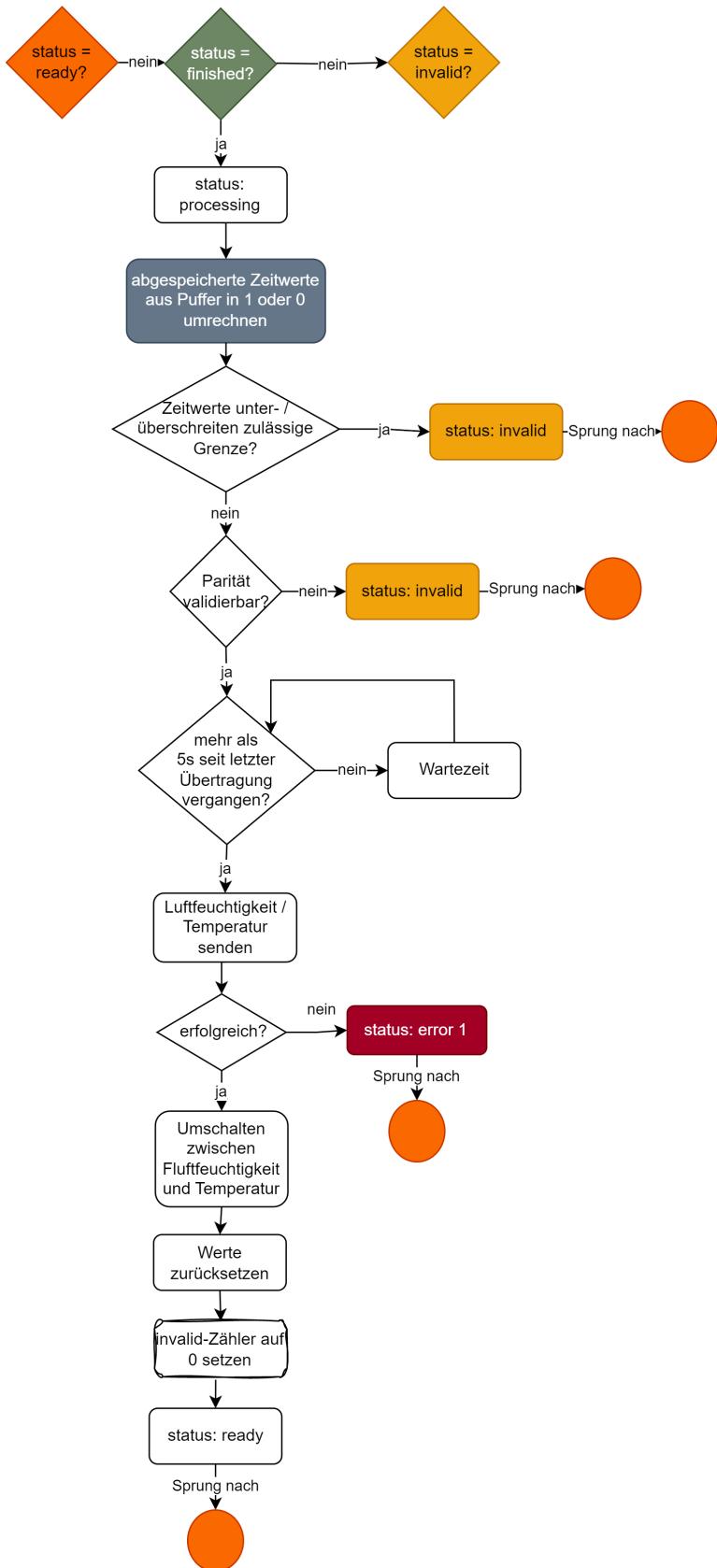


Abb. 14: Flussdiagramm: fertiger Datensatz

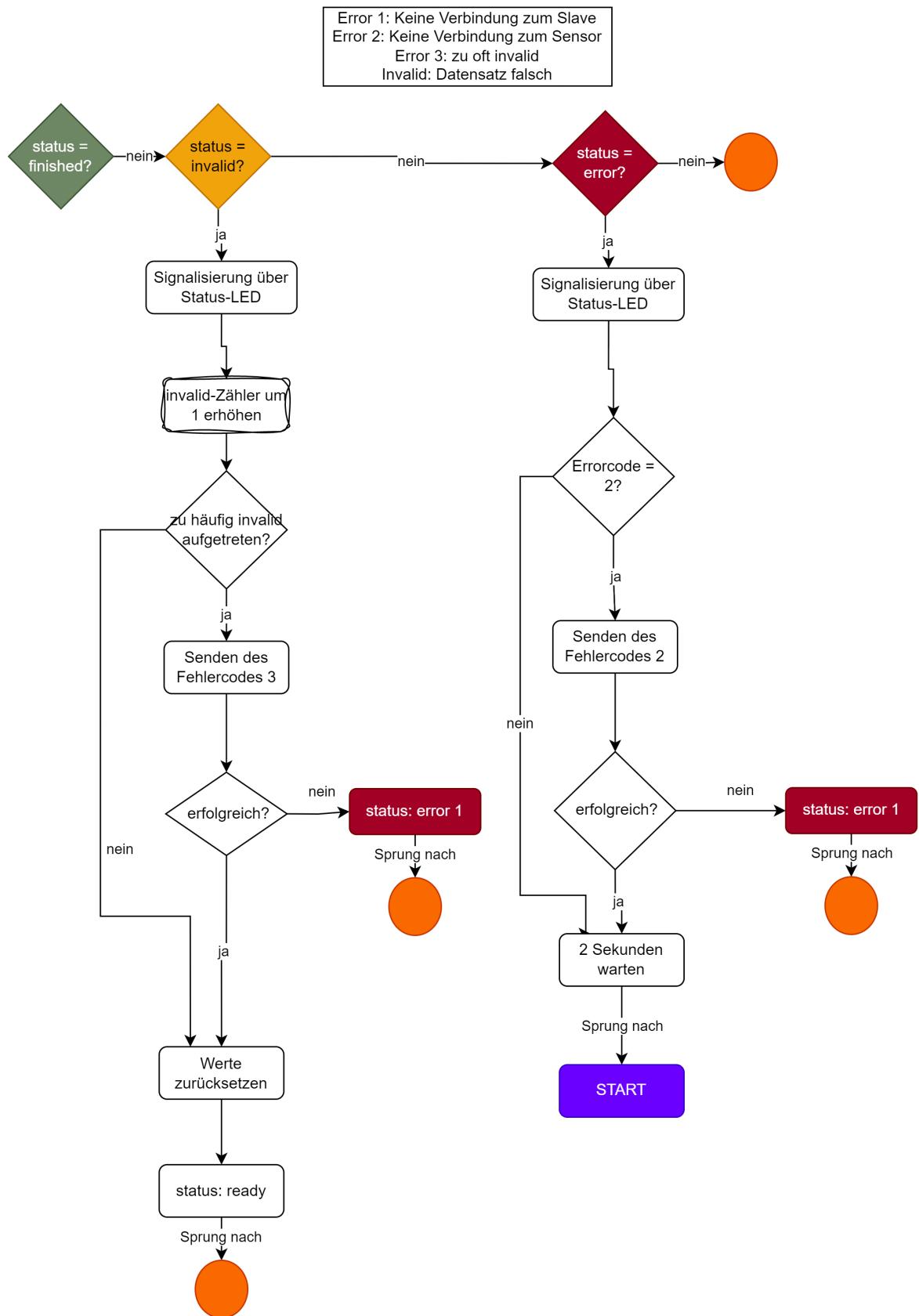


Abb. 15: Flussdiagramm: Fehlerbehandlungen

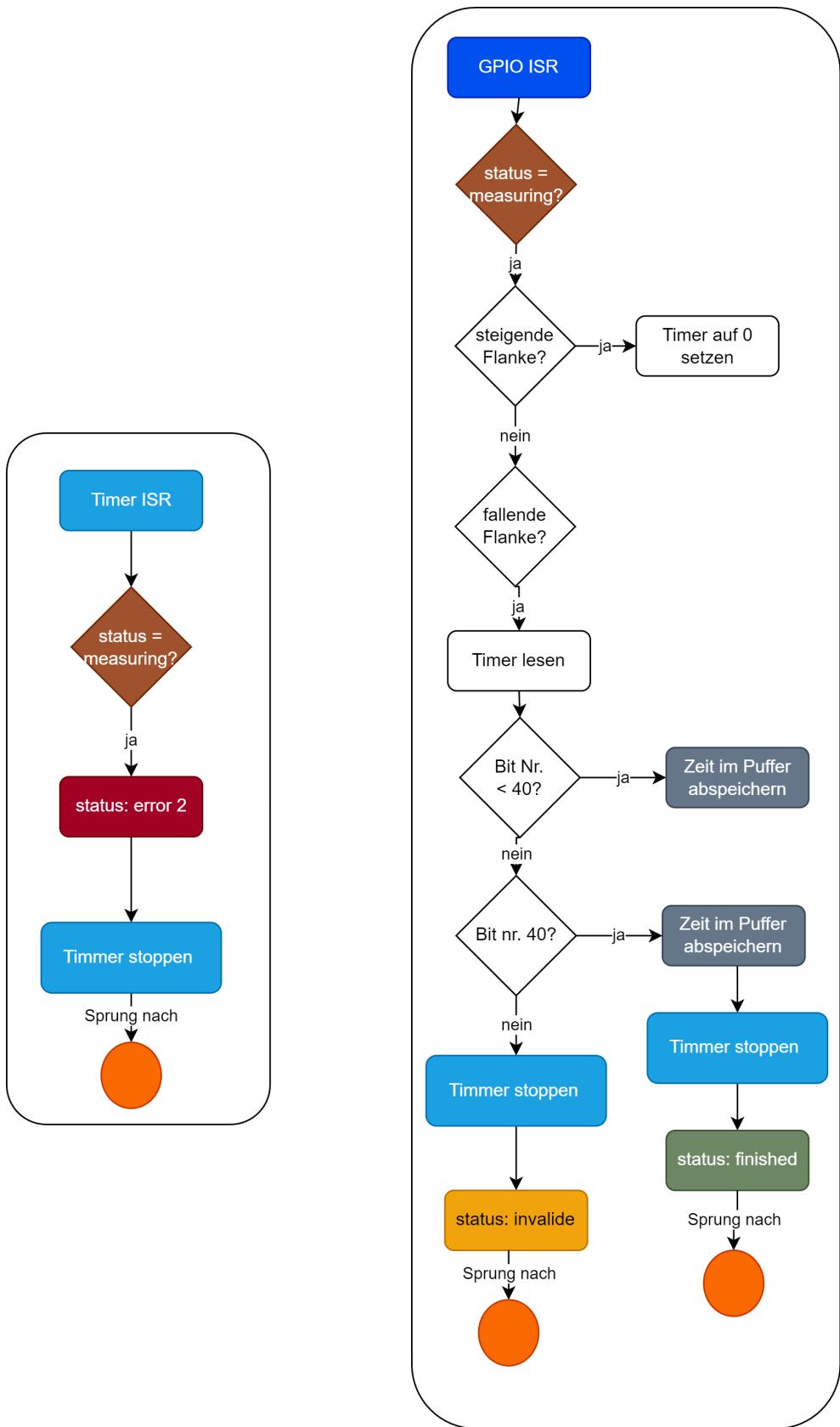


Abb. 16: Flussdiagramm: Interrupt-Servive-Routinen des Timers und des GPIOs

Auf Grundlage dieses Flussdiagramms und der vorangegangenen Tests, konnte mit dem eigentlichen Coden begonnen werden.

Zunächst wurde ein neues Projekt mit allen benötigten Komponenten (Timer, I²C und GPIO mit Interruptfunktion) angelegt. Zusätzlich wurden ein paar Kleinigkeiten bezüglich des Test-Projektes umgeändert.¹⁶

Der eigentliche Coding-Prozess soll an dieser Stelle eher unerwähnt bleiben, da das Flussdiagramm zu großen Teilen eins zu eins in C-Code übertragen werden konnte. Auch die dabei aufgetretenen Fehler waren zunächst eher belanglos.

Allerdings hier nun noch ein paar Elemente, die dennoch Erwähnung finden sollten.

1. Die Datensätze, welche vom Sensor gelesen werden bestehen aus 5 Bytes, die ersten zwei entsprechen der Luftfeuchtigkeit, die zweiten Bytes für die Temperatur und das letzte Byte ist die Parität. Da nun die Luftfeuchtigkeit und die Temperatur zwei Bytes umfassen bietet sich der Datentyp `uint16_t` an. Zur Validierung muss allerdings die Summe der einzelnen Bytes das Paritätsbyte ergeben.

Die Lösungsstrategie, welche hier zum Einsatz kommt, ist ein `union`.

Ein `union` ermöglicht das Überlagern von Datentypen auf dem selben Speicher. Übertragen auf dieses Problem, werden zwei `uint8_t` Elemente auf dem selben Speicher, wie ein `uint16_t` gespeichert (siehe [Code-Ausschnitt 1](#)). In [Code-Ausschnitt 2](#) ist die Umsetzung dokumentiert.

Mit `.Uint16_t_1x` wird die Variable als `uint16_t` interpretiert und mit `.Uint8_t_2x.firstByte` bzw. `.Uint8_t_2x.secondByte` als zwei einzelne Bytes.

Code-Ausschnitt 1: Der `data_t` Datentyp

```
1 struct halfData {
2     uint8_t firstByte;
3     uint8_t secondByte;
4 };
5
6 //used to interpret one uint16_t as two uint8_t
7 typedef union {
8     uint16_t Uint16_t_1x;
9     struct halfData Uint8_t_2x;
10 } data_t;
```

Code-Ausschnitt 2: Nutzung des `data_t` Datentyps

```
1 dataset[humidity].Uint16_t_1x = dataset[humidity].Uint16_t_1x | 1 << (17 -
element);
2 //...
3 sum = dataset[humidity].Uint8_t_2x.firstByte;
4 sum += dataset[humidity].Uint8_t_2x.secondByte;
5 sum += dataset[temperature].Uint8_t_2x.firstByte;
6 sum += dataset[temperature].Uint8_t_2x.secondByte;
7 if (parity != sum) {
8     currentState = invalid;
9     return;
10 }
```

2. Ein anderer Punkt stellt das Blockieren des *measuring* Zustands dar. Laut Datenblatt darf der Sensor maximal alle 2 Sekunden ausgelesen werden, da sonst die Daten unbrauchbar sind.

¹⁶ Beispielsweise wurde am Ende die Wartezeit zwischen senden des Startsignals und dem Freigeben der Leitung von 18ms auf 10ms reduziert.

Hierfür wurde eine Sperre eingebaut, welche das Senden des Startsignals verhindert (siehe [Code-Ausschnitt 3](#)). Die Sperre basiert auf dem System-Tick und einem Zeitstempel `tick_blockMeasurement`, der den Zeitpunkte der letzten Messung darstellt.

Im [Code-Ausschnitt 3](#) wird zunächst `HAL_GetTick()` von `tick_blockMeasurement` abgezogen. Ist die letzte Messung älter als 2 Sekunden, ist die Differenz größer als 2000 und es kann eine neue Messung durchgeführt werden.

Läuft das System allerdings sehr lange (etwa 50 Tage), kommt es zu einem Überlauf der `systick`-Variable.¹⁷ In diesem Fall würde `HAL_GetTick() - tick_blockMeasurement` nicht mehr die korrekte vergangene Zeit angeben, denn die Differenz würde negativ werden. Auf Grund des zyklischen Verhaltens, kann allerdings die negative Differenz in eine äquivalente positive Differenz umgerechnet werden (siehe [Abb. 17](#)). Diese äquivalente Zahl wird dann überprüft, ob sie größer als 2000 ist.

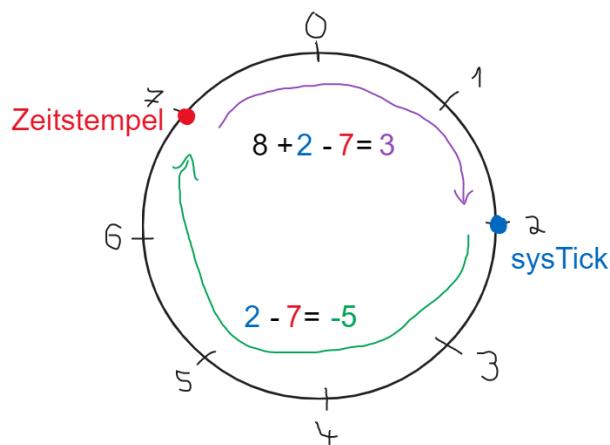


Abb. 17: Zyclus beim Überlauf
Der Kreis entspricht dem Typ `uint8`, die -5 ist equivalent zur 3

Der Initialwert von `tick_blockMeasurement` ist, anders als wie evtl. angenommen nicht 0, sonst würde das System, bei der ersten Messung, 2 Sekunden warten. Um dies zu verhindern wird das zyklische Verhalten ausgenutzt; `tick_blockMeasurement` wird dabei mit "höchstmögliche 32-Bit Zahl minus 2000 Millisekunden" initialisiert. Dadurch ist die Differenz vom Start an, ≥ 2000 . Samot kann die Messung ohne Blockierung starten.

Code-Ausschnitt 3: Blockieren des Startsignals

```

1 int64_t difference = HAL_GetTick() - tick_blockMeasurement;
2 //check for a systick overflow:
3 if (difference < 0) {
4     difference = uint32NumValues + difference;
5 }
6 if (difference >= 2000) {
7     //if more than 2 seconds have passed
8     tick_blockMeasurement = HAL_GetTick();
9     statusReady();
10 }
```

Damit ist die Bearbeitung der Sensor-Seite abgeschlossen.

Das System wurde anschließend erneut mit dem Arduino getestet¹⁸. Dabei konnte allerdings nicht der

¹⁷ Die Systick Variable besteht aus 32 Bit.

¹⁸ Hier liegt die Software: <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz->

invalid Zustand getestet werden.

Das Verbinden beider Systeme (Sensor-Seite und Darstellungs-Seite) sollte, Aufgrund der definierten Schnittstellen zwischen beiden Nucleos, reibungslos verlaufen.

3.2 4-stellige 7-Segment-Anzeige (Kevin Hübner)

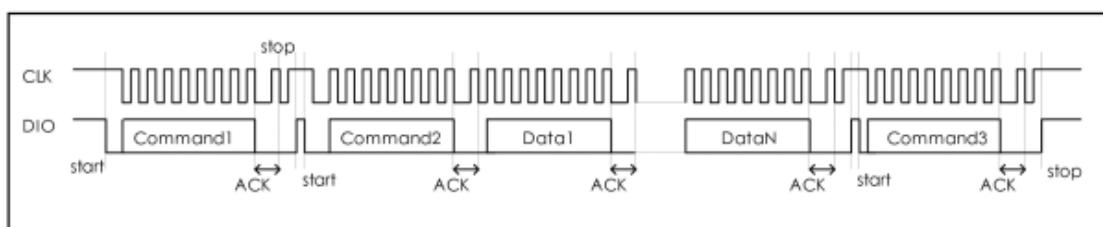
3.2.1 Grundlagen der 7-Segment-Anzeige

Um die gemessenen Daten des Temperatur- und Luftfeuchtigkeitssensors anzeigen zu können wird eine 4-Stellige-7-Segment-Anzeige genutzt. Vor der Implementierung wurde zuerst das Datenblatt¹⁹ studiert. Zudem hilft ein Github-Projekt²⁰ bei der Implementierung, dieses stellt wichtige Funktionen bereit, die unabdingbar funktionieren müssen.

Die 7-Segment-Anzeige bietet im Allgemeinen nur 4 Pins, welche mit dem Nucleo-Board oder anderen Boards verbunden werden. Dabei gibt es einen Pin der mit GND verbunden wird und einen zweiten Pin der mit der Spannungsversorgung von 3.3 V des Nucleo-Boards verbunden wird. Die anderen beiden Pins sind für die Datenübertragung (DIO-Pin) und den Clock-Input (CLK-Pin) zuständig. Dafür werden am Nucleo-Board zwei GPIO-Pins im Output-Mode benötigt. Auf Basis dieser Einstellungen wurde das Grundgerüst²¹ erstellt.

Im Datenblatt findet sich folgende Abbildung Abb. 18:

2. Write SRAM data in address auto increment 1 mode.



Command1: Set data

Command2: Set address

Data1~N: Transfer display data

Command3: Control display

Abb. 18: Datenübertragung bei der 7-Segment-Anzeige

Darin ist ersichtlich, wie die Anzeige angesprochen wird. Wichtige Information die hier mitgenommen werden können sind die Commands, die Start-Befehle, der Stop-Befehl und ACK. Die Commands werden benötigt um das Displayregister zu beschreiben.

¹⁹ <http://www.datasheet-pdf.com/PDF/TM1637-Datasheet-TitanMicro-788613>

²⁰ <https://github.com/UsefulElectronics/stm32-tm1637-library/>

²¹ <https://gitlab.rz.hwt-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/commit/58021275a10379d7487f0d2c0f1618022f65e426>

Der Beispielablauf um etwas auf dem Display anzeigen zu lassen wäre also, laut Abbildung, wie folgt:

Schritt 1: Start-Befehl (Zum starten der Datenübertragung auf den Chip der Anzeige)

Schritt 2: Command 1 (Zum Festlegen ob gelesen oder geschrieben werden soll)

Schritt 3: Command 2 (Zum Festlegen der Adresse)

Schritt 4: Datensatz 1 (Zum Festlegen der aktiven Segmente an der ersten Stelle)

Schritt 5: Datensatz 2 (Zum Festlegen der aktiven Segmente an der zweiten Stelle)

Schritt 6: Datensatz 3 (Zum Festlegen der aktiven Segmente an der dritten Stelle)

Schritt 7: Datensatz 4 (Zum Festlegen der aktiven Segmente an der vierten Stelle)

Schritt 8: Command 3 (Zum Festlegen ob das Display an oder ausgeschaltet sein soll)

Schritt 9: Stop-Befehl (Zum stoppen der Datenübertragung)

Diese Schritte können sehr leicht runtergebrochen werden, da fast alle benötigten Informationen im Datenblatt in der gleichen Reihenfolge zu finden sind.

Das Erste was einem Auffällt ist, dass jeder Command bzw. jeder Data-Befehl mit einem ACK endet. Das heißt es wird auf eine Rückmeldung vom Mikrocontroller der 7-Segment-Anzeige gewartet und ist die einzige Input-Aktion, welche beachtet werden muss. Die Befehle für den Start bzw. Stop sind abhängig von der CLK/DIO-Flanke und diese müssen in exakter Reihenfolge auf *High* bzw. *Low* geschaltet werden.

Für die Commands gibt es immer einen Start-Befehl, aber nur für den ersten Command gibt es einen Stop-Befehl, jedoch kommt nach dem dritten Command ebenfalls ein Stop-Befehl um die Datenübertragung abzuschließen. Die Datensätze sind unabhängig voneinander und beschreiben das Register der 7-Segment-Anzeige mit den Werten, welche an der jeweiligen Stelle dargestellt werden soll. Jede Stelle der Anzeige wird über ein 8-Bit (1 Byte) langes Register geschrieben. Dabei ist zu beachten, dass die Binäre-Darstellung einer Zahl nicht gleich der Darstellung auf der Anzeige entspricht. Jedes einzelne Segment entspricht einem Bit aus den 8-Bit des Registers d.h. um beispielsweise eine 1 auf der Anzeige darzustellen, müssen zwei Bit im Register auf 1 gesetzt werden, das Segment *b* sowie das Segment *c* (siehe Abb. 19).

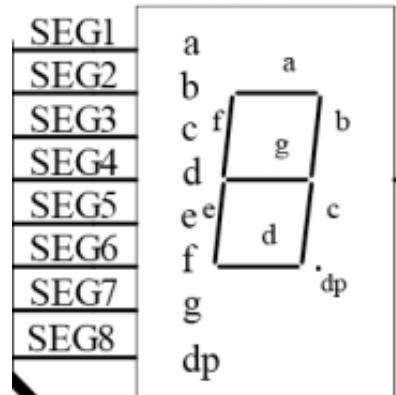


Abb. 19: 7-Segment-Anzeige

Jeder Command und jede 7-Segment-Stelle wird über 9 CLKs abgearbeitet (8 für die Daten und 1 für den ACK). Insgesamt sind das bei den 4 Stellen der 7-Segment-Anzeige mit den 3 Commands, 7 dieser 9 CLK-Cyclen.

3.2.2 Erste Inbetriebnahme der 7-Segment-Anzeige

Die Pins, welche benötigt werden um die 7-Segment-Anzeige in Betrieb zu nehmen (DIO-Pin und CLK-Pin), werden über die Software STM32 CubeMX (Pins PA6 | A5 und PA7 | A6 bei diesem Nucleo-Board) auf Output gestellt. Beim Debuggen kann man noch den Pin PB3 des Nucleo-Boards als GPIO-Output setzen. Der Pin kann dann genutzt werden, um eine Led zu steuern und dient als debug-Hilfe.

Da später Sensordaten von dem ersten Nucleo über I²C empfangen werden, wird zusätzlich I²C aktiviert. Wichtig dabei ist noch die Aktivierung des I²C-Interrupthandlers und das Setzen der Slave-Adresse (hier 0x20) damit wir von einem Board, welches als Master agiert, die Daten empfangen können.

Als Grundstein der Implementierung wird die Funktion HAL_GPIO_WritePin benötigt. Mit dieser können die Pins, welche die Pegel für den CLK-Pin und DIO-Pin der Anzeige definieren, auf *High* bzw. *Low* gestellt werden. Durch das Ändern der Pegel wird dem Microkontroller der 7-Segment-Anzeige alle nötigen Informationen mitgeteilt. Die Informationen liegen dabei als Bits, je nach Pegelzustand 0 oder 1, vor und werden an die Anzeige übergeben. Es muss dabei beachtet werden, zu welchem Zeitpunkt ein bestimmter Wert an die Anzeige geschickt wird.

Als ersten Schritt wurde der Start- bzw. Stop-Befehl implementiert. Über die vorher erwähnte Funktion HAL_GPIO_WritePin lassen sich beide Befehle leicht implementieren. Es muss nur darauf geachtet werden, dass der richtige Pin zur richtigen Zeit *Low* bzw. *High* gesetzt wird. Für den Start-Befehl wird die CLK auf *High* gesetzt und während die CLK in diesem Zustand ist, wird der DIO-Pin von *High* auf *Low* gesetzt (siehe Abbildung 20). Für den Stop-Befehl wird der CLK-Pin ebenfalls auf *High* gesetzt, jedoch wird diesmal der DIO-Pin von *Low* auf *High* gezogen (siehe Abbildung 21).

Mithilfe eines Logic Analyzers wurde anschließend getestet, ob die Flanken der Stop- bzw. Start-Funktion zum richtigen Zeitpunkt eintreffen.

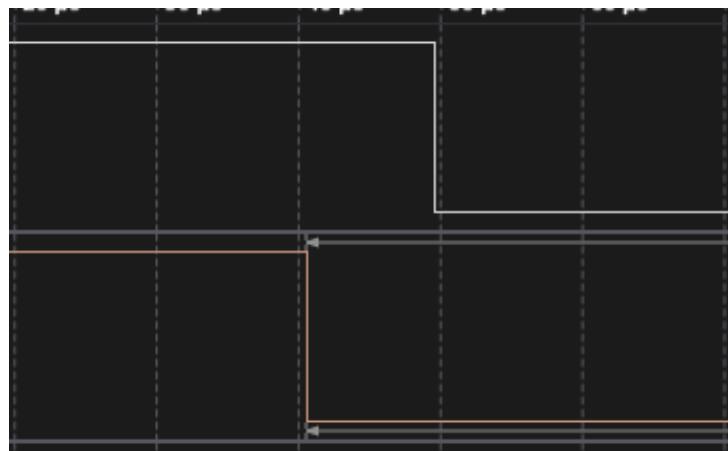


Abb. 20: Start-Befehl über die Flanken des Nucleo-Boards
oben: CLK, unten: DIO

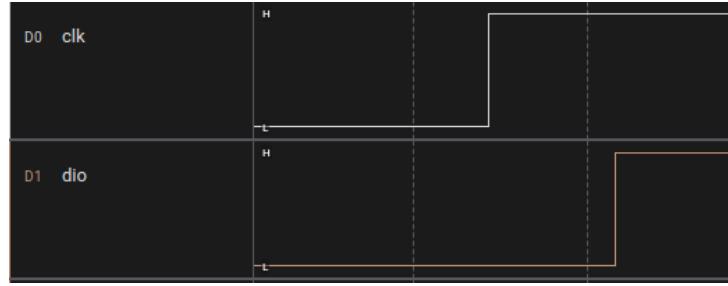


Abb. 21: Stop-Befehl über die Flanken des Nucleo-Boards

Ebenfalls wichtig für das Projekt ist der Acknowledgment-Check (ACK-Check). Nach jedem abgearbeiteten 9-CLK-Cycle wird eine Antwort (ACK) erwartet. Da jedoch beide GPIO-Pins auf Output gestellt sind, kann keiner der beiden Pins ein ACK entgegennehmen. Hier ist es wichtig den CLK-Pin kurzzeitig auf Input zu setzen und auf diesen ACK zu warten. Um einen Pin als Input zu deklarieren, wird dieser per `GPIO_InitStruct.Mode` auf `GPIO_MODE_INPUT` gesetzt. Für die ACK-Check-Funktion muss der Pin-Zustand geändert werden (Output -> Input -> Output wenn ACK erfolgt ist) und es muss auf den ACK gewartet werden. Das wurde, für dieses Projekt, in der Funktion `display_check_acknowledge` (Abbildung [Code-Ausschnitt 4](#)) realisiert.

Code-Ausschnitt 4: Funktion zum Überprüfen des Acknowledgements

```

1 void display_check_acknowledge()
2 {
3     change_pin_mode(input);
4     display_clk_low();
5
6     while(HAL_GPIO_ReadPin(CLK_GPIO_Port, CLK_Pin))
7     {
8         HAL_GPIO_WritePin(led_GPIO_Port, led_Pin, SET);
9
10    }
11    HAL_GPIO_WritePin(led_GPIO_Port, led_Pin, RESET);
12    change_pin_mode(output);
13 }
```

Im Hauptteil des Programms fehlt nur noch eine Methode, um mithilfe von CLK und DIO die Anzeige so zu manipulieren, sodass die gewünschten Zeichen angezeigt werden.

Der zu übertragenden Command bzw. die zu übertragenden Daten einer 7-Segment-Stelle liegen dabei in Arrays vor. Das Ziel ist es nun mit jedem CLK-Schlag (*Low->High->Low*) ein Bit über den DIO-Pin an die Anzeige zu senden (Beispiel in [Abb. 22](#)). Das jeweilige Bit wird über Bitshifting aus dem entsprechenden Array herausgefiltert.

Jeder Command bzw. jeder Datensatz wird über 9-CLKs in einer Periode umgesetzt.

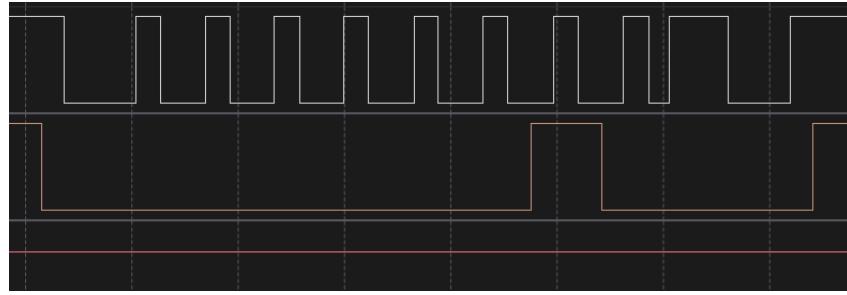


Abb. 22: CLK-Cyklus
oben: CLK, unten: DIO

Desweiteren ist zu beachten, dass jeder Command bzw. Datensatz mit einem Start-Befehl beginnt, jedoch nicht mit einem Stop-Befehl aufhört. Somit lohnt es sich zwei Funktionen zu schreiben, eine um die Commands ab zu arbeiten und eine andere um die Datensätze abzuarbeiten.

Für den Command 1 wird 0x40 (binär 0b0100 0000) zur Anzeige gesendet. Dieser Wert legt fest, dass in folgende Register geschieben wird, das die Adresse automatisch Inkrementiert wird und es aktiviert den normal Mode.

In Command 2 wird die Startadresse gewählt, welche beschrieben werden soll. Laut Datenblatt ist dies die Adresse C0H, mit dem 8-Bit Wert 0xC0 oder in binär 0b11000000.

Danach folgen die Datensätze, mit denen die 7-Segment-Stellen beschrieben werden.

Über Command 3 lässt sich die Anzeige an und ausschalten, aber auch die Helligkeit manipulieren. In diesem Projekt ist die Helligkeit irrelevant. Die Anzeige lässt sich mit 0x80 (0b10000000) ausschalten und mit 0x88 (0b10001000) einschalten.

In Abb. 23 wurden die drei Commands sowie vier mal der Datensatz 0x3F (0x3F entspricht einer 0) an das Display übertragen und mit einem Logic Analyzer untersucht.



Abb. 23: Logic Analyzer für das anschalten des Displays
oben: CLK, unten: DIO, Beispielübertragen stellt vier Nullen dar.

Als Beispiel²² wurde auf der Anzeige 0000 ausgegeben (siehe Abb. 24). Dabei blieb die Anzeige periodisch für 5 Sekunden (über die HAL_Delay-Funktion) eingeschaltet und für 2.5 Sekunden im ausgeschalteten Zustand.

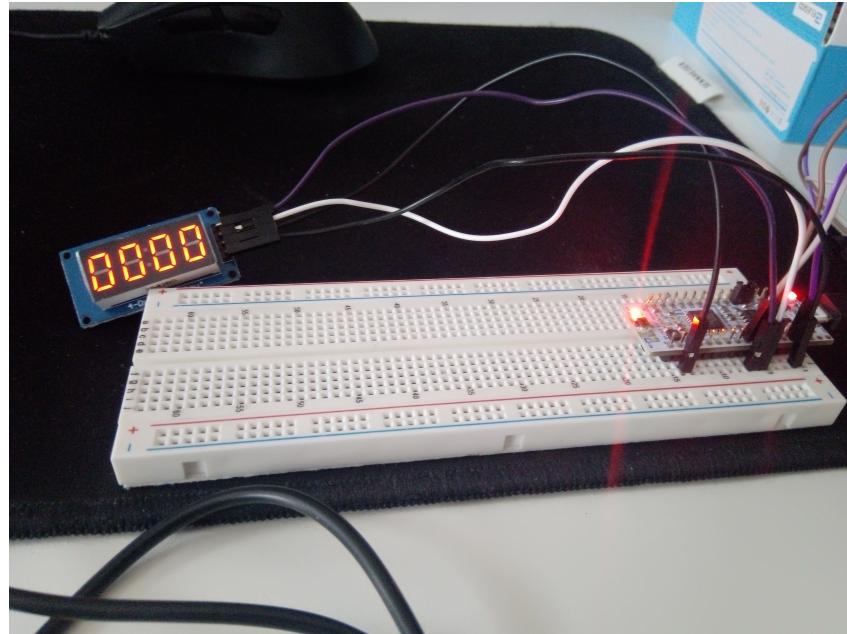


Abb. 24: Angeschaltetes Display

3.2.3 I2C Anpassungen

Nun ist die Implementierung der Anzeige so gut wie fertig: Daten können manuell auf das Display schreiben werden und das Display kann An bzw. Ausgeschalten werden. Da jedoch Daten von einem weiteren Nucleo, welche mit einem Sensor verbunden ist, empfangen werden, ist es notwendig die I2C-Kommunikation zwischen den Boards zutesten. Aus diesem Grund wurde ein neues Test-Projekt erstellt. Als Testen wird hier zunächst ein Arduino genutzt, welcher als Master Daten sendet.

Für dieses Testprojekt wird eine LED, sowie die I2C-Kommunikation aktiviert. Da dieses Nucleo-Board, welches die Darstellung der Luft- und Temperaturwerte übernimmt, als Slave arbeitet wird in den I2C-Einstellung noch die Slaveadresse 0x20 eingetragen.²³ Diese Adresse wird später vom Master adressiert. Des Weiteren wird für I2C der Interruptmodus eingeschaltet.

Um allerdings zwischen den Boards kommunizieren zu können, werden noch zwei PullUp-Widerstände für die Daten- und Clockverbindung hinzugefügt.

Mit den vorgenommenen Einstellungen und der LED kann überprüft werden, ob die Kommunikation zwischen beiden Boards funktioniert.

²² <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/commit/7f20b1f06ab9d345955305177ffee06c94a51b81>

²³ Für die Seite des I2C-Slaves wurde sich hieran orientiert: <https://community.st.com/s/article/how-to-create-an-i2c-slave-device-using-the-stm32cube-library> und hier: https://www.st.com/resource/en/user_manual/dm00173145-description-of-stm32l414-hal-and-lowlayer-drivers-stmicroelectronics.pdf

I2C kann in drei verschiedenen Modi genutzt werden: per Polling, per Interrupt und per DMA. Im ersten Test wird Polling genutzt, da es sich um die einfachere Methode handelt. Der Master sendet dabei den Wert 0x40FF und der Slave empfängt diesen über die Funktion `HAL_I2C_Slave_Receive` (Polling). Bei Polling wartet der Slave so lange auf eine Nachricht, bis ein Timeout überschritten wurde.

Der Versuch mit Polling ²⁴ sieht dann wie folgt aus: [Abb. 25](#).

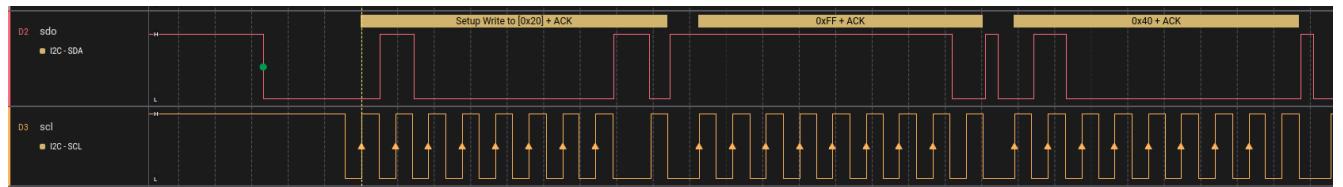


Abb. 25: I2C Ergebnis der Kommunikation über Polling (erster Datensatz)

Im zweiten Test wird der Interruptmodus von I2C ausgetestet.

²⁵ Dieser wird über die Funktion

`HAL_I2C_Slave_Receive_IT` (Interrupt) gestartet. Des Weiteren wird eine Callback-Funktion `HAL_I2C_SlaveRxCpltCallback` benötigt. Diese wird automatisch vom System aufgerufen, sobald der Slave einen vollständigen Datensatz erhalten hat.

Für diesen Test sendet der Master zunächst beliebige Werte mit der Länge von zwei Bytes.

In der Callback-Funktion wird die LED umgeschaltet, d.h. zwischen angeschaltet und ausgeschaltet gewechselt. Damit lässt sich visuell feststellen, ob der Aufruf der Callback-Funktion gelungen ist. Entspricht der empfangene Wert dem Wert 0xCAFE, so sendet der Slave mit der Funktion `HAL_I2C_Slave_Transmit` die Antwort 0x55 zum Master. (siehe [Abb. 26](#))

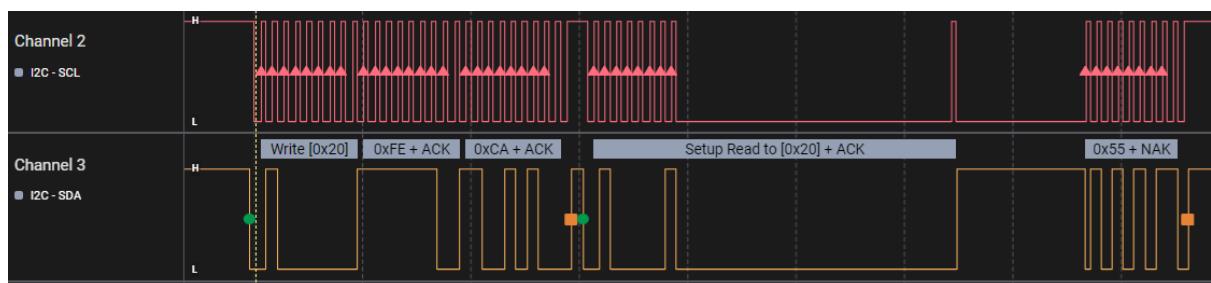


Abb. 26: I2C Ergebnis der Kommunikation über Interrupt

Somit ist die Kommunikation mit I2C für das Testprojekt ²⁶ gelungen und für das eigentliche Projekt nun beide Teile: Display und I2C zusammengeführt.

3.2.4 Zusammenführung von Display und I2C

In diesem Schritt wird die Darstellung auf dem Displays mit der Kommunikation zum anderen Nucleo-Board zusammengeführt.

²⁴ <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/commit/f0cb88f215dd76c01cb24b5267ed8661962d2888>

²⁵ <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/commit/8dc36db574aa064f33940e40cb78256e63d41787>

²⁶ <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/commit/8dc36db574aa064f33940e40cb78256e63d41787>

Dabei sind noch einige Dinge zu beachten:

Das erste Board, welches den Sensor ausliest, sendet abwechselnd Luftfeuchtigkeit und Temperatur. Zur Unterscheidung, wird in dem vorletzten Bit kodiert, ob es sich um Luftfeuchtigkeit oder Temperatur handelt. Luftfeuchtigkeit entspricht einer 0 und 1 ist Temperatur. Über eine Bitmaske und Bitweisen UND wird die entsprechende Größe herausgefiltert. auf dem Display wird die erste Stelle genutzt, um anzuzeigen, ob sich um eine Temperatur oder um Luftfeuchtigkeit handelt; Für die Temperatur wird ein *C* für Celsius dargestellt und für Luftfeuchtigkeit ein *H* (Humidity).

Das erste Bit beschreibt, im Falle der Temperatur, der Vorzeichen.

Bit gesetzt bedeutet eine negative Temperatur und in der zweiten Stelle der Anzeige wird ein '-' angezeigt, sond ist die Stelle leer.

Die beiden letzten Stellen der Anzeige nehmen die gemessenen Werte ein. Da nun alle vier Stellen belegt sind, wurde sich dazu entschieden, auf die Nachkommastellen zu verzichten.

Doch ohne Nachkommastellen müssen die Zahlen gerundet werden was zu einem weiteren Problem führt. Die Daten aus 2 Byte enthalten in den ersten zwei Stelle (MSB und MSB-1) die Codierung, welche entscheidet ob es sich um eine Temperatur oder eine Luftfeuchtigkeit handelt. Diese wird über eine Bitmaske entfernt, damit die Daten als Wert interpretiert werden können. Für die Anzeige ist es notwendig den Wert so vorzubereiten, dass die Zehner- und Einer-Stellen in jeweils 8 Bits binär vorliegen. Das Stichwort dazu lautet BCD-Code.

Beispielsweise die Zahl 13 sieht binär, in 16 Bits, so aus: 0b0000000000001101. Um die 13 auf der Anzeige darzustellen, muss die 13 in zwei verschiedene 8 Bits aufgeteilt werden. Die 1 (Zehner) wird in 0b00000010 und die 3 (Einer) in 0b00000011 umgewandelt. Der Algorithmus ²⁷ ist in der Double_Dabble-Funktion in [Code-Ausschnitt 5](#) implementiert.

Code-Ausschnitt 5: Double Dabble Funktion

```
1 void double_dabble(uint8_t data) {
2     union dd wandler;
3     wandler.num = 0x00;
4
5     wandler.raw.input_data = data;
6
7     for (int i = 0; i < 7; i++) {
8         wandler.num = wandler.num << 1;
9         if (wandler.raw.zehner >= 5) {
10             wandler.raw.zehner += 3;
11
12         }
13         if (wandler.raw.einer >= 5) {
14             wandler.raw.einer += 3;
15
16         }
17     }
18     wandler.num = wandler.num << 1;
19
20     dd_data = wandler;
21 }
```

²⁷ https://en.wikipedia.org/wiki/Double_dabble

Ein Fehlerfall wird auf das Anzeige über “ErrX” dargestellt, wobei X den Fehlercode darstellt. Im Datensatz des Masters wird Error über das setzen der obersten zwei Bits (MSB und MSB-1) kodiert.

Damit wäre alles notwendige implementiert.

Wie beim ersten Nucleo, basiert auch dieses System auf einer einfachen Zustandsmaschine. Während des StartUps wartet der Slave darauf, dass der Master 0xCAFE sendet, wodurch der Slave in den Zustand *active* versetzt wird. Das ist der Schlüssel, in die Hauptschleife des Systems zu gelangen. Einmal in die Hauptschleife gelangt, wartet das System nun ununterbrochen auf neue Datensätze, die dargestellt werden können.

Werden neue Datensätze empfangen, so wird der Status auf *work* gesetzt und der Datensatz wird auf der Anzeige dargestellt. Nach erfolgreichem Aktualisieren des Displays wird der Status auf *no-work* geändert, um in den IDLE-Modus zurück zu kehren.

Bleiben für mehr als 10 Sekunden neue Datensätze aus, so wird ein Fehler auf dem Display angezeigt.

Und im Fall, dass es keine Verbindung zum Display gibt, blinkt die LED auf dem Board.

Das Projekt ²⁸ für die 4-Stellen-7-Segment-Anzeige ist damit fertig gestellt.

3.3 Zusammenführung beider Systeme

Nach Beendigung beider Implementierungen, konnten beide Systeme tatsächlich problemlos verbunden werden (siehe Abb. 27).

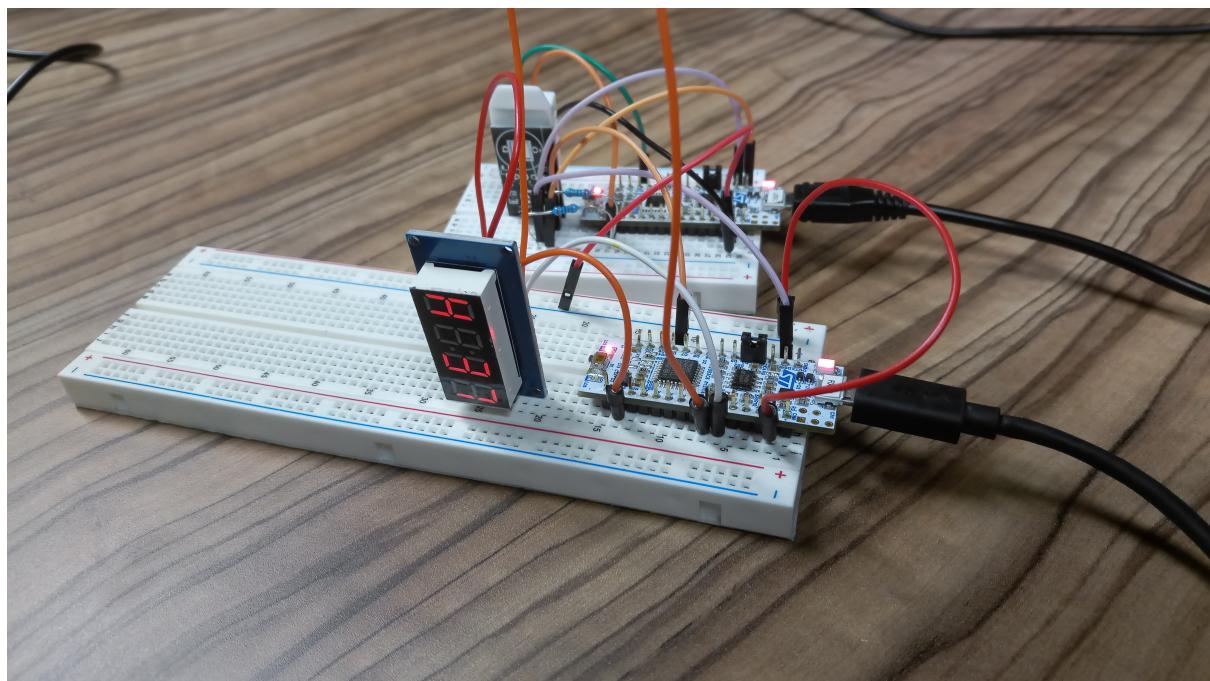


Abb. 27: Erfolgreiches Verbinden beider Nucleos

²⁸ <https://gitlab.rz.htw-berlin.de/s0582020/ce31-embd-gruppe-6-tz-kh/-/commit/bbc1329171e7bb3871fb4049f9456aa00d5a023f>

4 Probleme

Während der Tests der I2C-Kommunikation kam es zwischenzeitig zu Problemen: 1. Wurde der CLK-Pegel nach Polling-Timeouts vom Slave dauerhaft auf *Low* gezogen (siehe Abb. 28). Das Problem konnte durch die Einstellung *Clock No Stretch Mode* behoben werden. Dabei war diese Einstellung im Reiter Connectivity auf Disable gestellt, jedoch muss diese auf Enable gestellt werden.

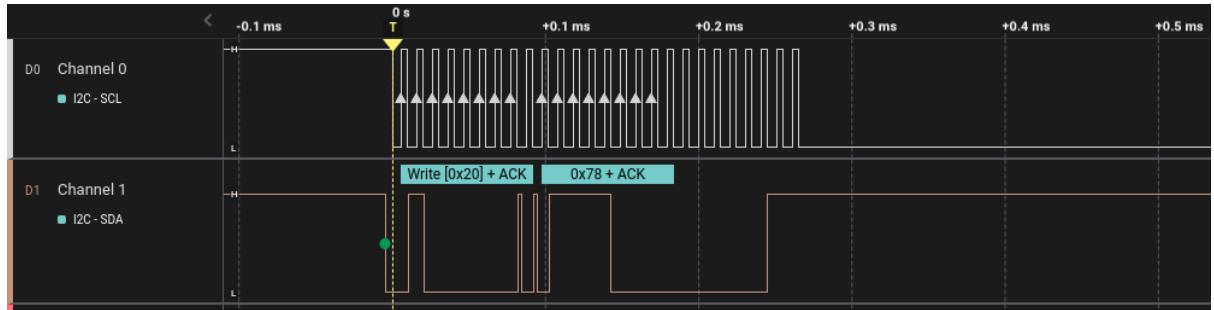


Abb. 28: I2C Clock wird vom Slave auf *Low* gezogen

2. Es wurde fälschlicher Weise während der Tests die Funktion `HAL_I2C_EnableListen_IT` in den Code hinzugefügt, allerdings trat damit eher das Gegenteil auf als wie der Name vermuten lässt. Der Slave reagierte damit gar nicht mehr (siehe Abb. 29). Nach herausnehmen dieser Funktion lief das Verhalten die geplant.

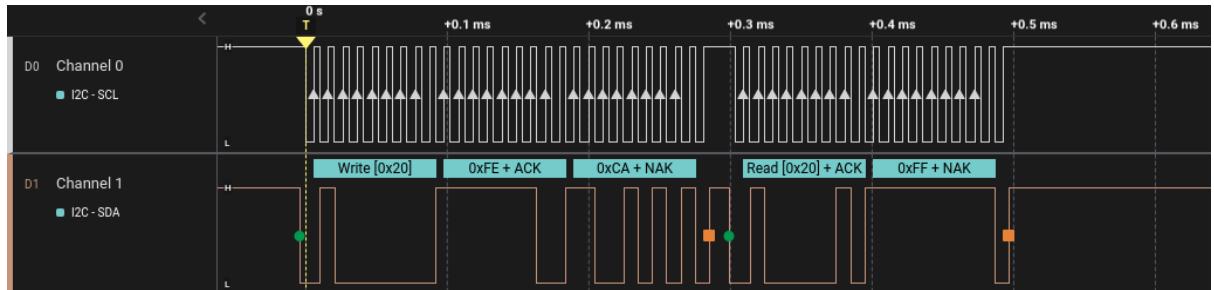


Abb. 29: I2C Slave reagiert nicht

5 Fazit

Das Projekt lief im Großen und Ganzen recht flüssig, auch wenn es ein paar größere Hürden zu überwinden galt.

Aufgrund von anderen zeitnahen Abgaben in anderen Modulen, war es notwendig das Projekt schnellstmöglich fertigzustellen, dies ging leider auf Kosten der Dokumentationsqualität.

Als Weiterentwicklung könnten sämtliche `HAL_Delay`-Funktionen ersetzt werden durch eine ökonomischere Methode; so könnte sich der Mikrocontroller schlafen legen und über einen Timerinterrupt nach entsprechender Zeit wieder geweckt werden.