

CAN-Lab White Paper

Kurzfassung, Motivation und Testaufbau

blog.andrelademann.de

Zwei ESP32-C3-Boards mit SN65HVD230-Transceivern übertragen bei 500 kbit/s stabil **4.303 CAN-Frames pro Sekunde** mit je 8 Datenbytes. Im Messzeitraum traten **keine Zählerlücken, keine Überläufe der Empfangswarteschlange und keine Busfehler** auf. Das sind rund 116 Bitzeiten pro Frame: Der Bus war praktisch voll, begrenzt hat die Leitung, nicht die Hardware.

MOTIVATION

Ich baue die Elektrik meines Peugeot 304 S neu auf: verteilte ESP32-Knoten mit Sensoren, die über CAN kommunizieren, und ein Raspberry Pi 5 als zentrale Recheneinheit. Bevor dieser Aufbau ins Auto wandert, wollte ich wissen, wie sich der Bus auf günstiger Maker-Hardware verhält, und zwar unter Last statt nur im Leerlauf.

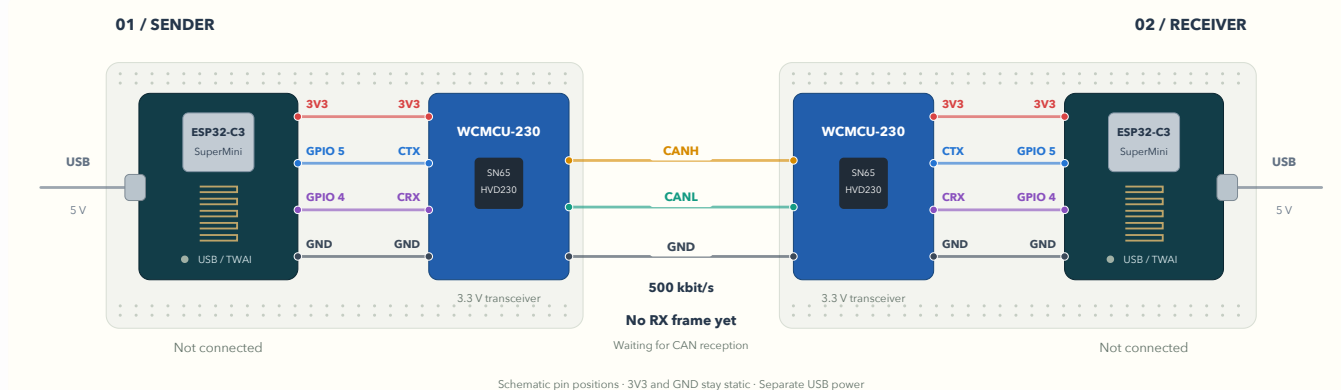
Dafür ist **CAN-Lab** entstanden: eine Web-Serial-Konsole im Browser, die zwei Boards gleichzeitig ausliest, den Sender steuert und Durchsatz sowie Fehlerzähler live darstellt. Der Code ist Open Source unter GPL-3.0 auf [GitHub](https://github.com).

STÜCKLISTE

Komponente	Anzahl	Rolle
ESP32-C3 SuperMini	2	Sender und Empfänger, CAN-Controller (TWAI) integriert
SN65HVD230-Transceiver-Modul, 3,3 V	2	Wandelt Logikpegel in CAN_H/CAN_L
Steckbrett und Jumper-Kabel	1 Satz	Verdrahtung
USB-Kabel (Daten, nicht nur Laden)	2	Stromversorgung und Web Serial

Die Boards und Module, die ich verwendet habe, sind im [Blogartikel](#) verlinkt. Ein ESP8266 hat keinen CAN-Controller und eignet sich nicht.

SO KOMMUNIZIEREN DIE BEIDEN KNOTEN



VERDRÄHTUNG, AUF BEIDEN SEITEN IDENTISCH

ESP32-C3	Transceiver	Zwischen den Modulen
3V3	3V3	CANH an CANH
GND	GND	CANL an CANL
GPIO 5	CTX	GND an GND
GPIO 4	CRX	Jedes Board mit eigener USB-Versorgung

Busparameter: 500 kbit/s, Standard-ID 0x123 (11 Bit), 8 Datenbytes pro Frame (4 Byte Zähler, 4 Byte Test-ID, Little Endian). Sende-Warteschlange 32 Frames, Empfangswarteschlange 128 Frames.

STECKBRETT-AUFBAU

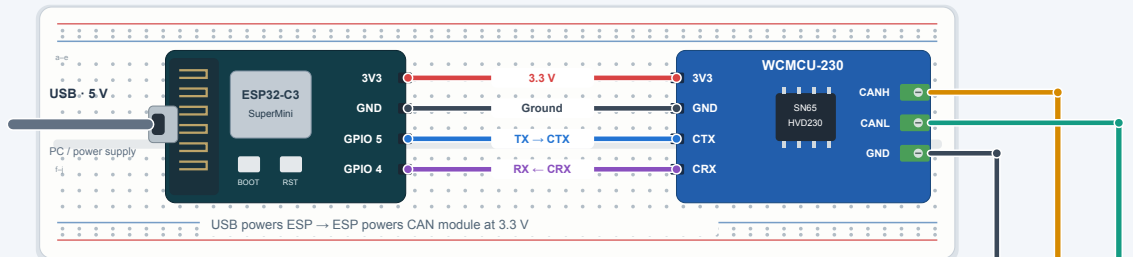
Two boards. One CAN bus.

ESP32-C3 SuperMini + WCMCU-230 / SN65HVD230 · 500 kbit/s

Wiring diagram: components and pin positions are schematic. Follow the pin labels on the actual board.

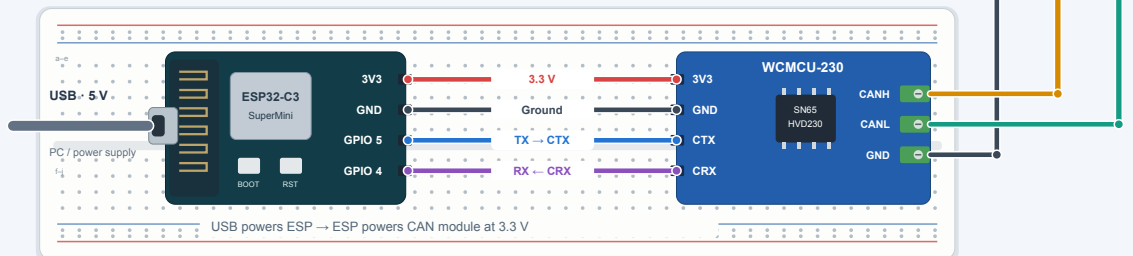
01 SENDER

sender.ino · sends counter with ID 0x123



02 RECEIVER

receiver.ino · receives frames and blinks



How to connect

CANH ↔ CANH · CANL ↔ CANL · GND ↔ GND

Twist CANH and CANL for longer connections.

Power both ESPs separately via USB; no 5V/3V3 jumper.

Termination & checks

WCMCU-230 modules have built-in bus termination.

No additional external resistors are needed.

With power off, measure CANH to CANL: total ≈ 60 Ω.

Source: repository README / GPIO configuration · SN65HVD230: TI datasheet · Power CAN modules at 3.3 V only.

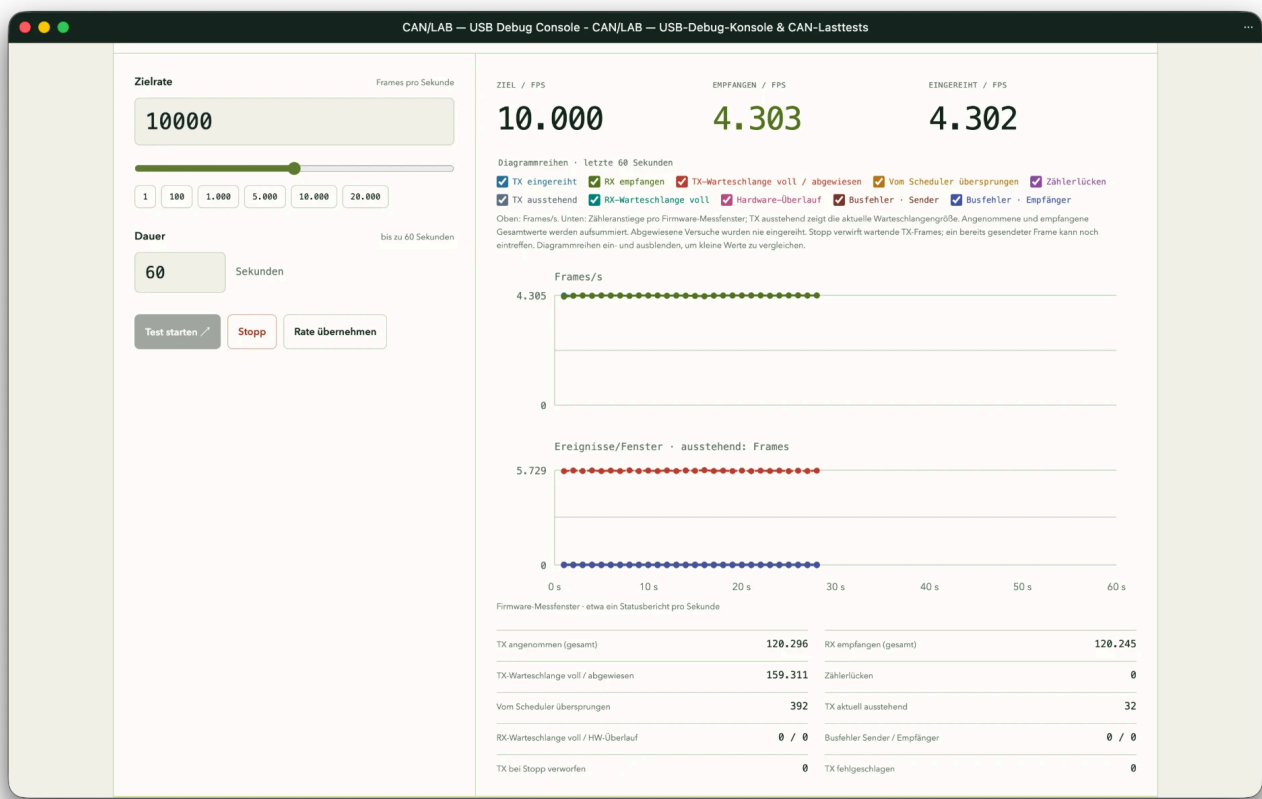
Auch als [interaktives Steckbrett bei Wokwi](#) – eine Verdrahtungshilfe, keine elektrische CAN-Simulation.

MESSMETHODIK

- 1. Handshake.** Beide Boards melden Rolle, Protokollversion und Bitrate (LAB HELLO). Weichen die Bitraten ab, startet kein Test.
- 2. Scharfschalten.** Zuerst wird der Empfänger vorbereitet; der Sender startet erst, wenn der Empfänger seine Bereitschaft bestätigt.
- 3. Last erzeugen.** Der Sender versucht, die Zielrate einzureihen. Was die Sende-Warteschlange nicht annimmt, zählt als abgewiesen, ohne den Nutzlastzähler zu erhöhen. Abgewiesene Versuche erzeugen also keine künstlichen Lücken.
- 4. Messen.** Beide Boards melden etwa einmal pro Sekunde ihre Zähler aus einem eigenen Firmware-Messfenster, nicht aus der Ankunftszeit der USB-Logs.
- 5. Lücken erkennen.** Der Empfänger prüft die fortlaufenden Zählerwerte. Fehlt ein Wert, zählt er eine Lücke.

Die Zielrate ist angebotene Last und darf die Kapazität des Busses übersteigen. Genau das ist der Sinn dieses Tests.

LASTTEST: 10.000 FRAMES/S ANGEBOTEN, 60 S GEPLANT



Aufgenommen während des laufenden Tests, nach etwa 28 Sekunden.

Messwert	Wert	Messwert	Wert
Zielrate	10.000 Frames/s	Zählerlücken	0
Empfangen	4.303 Frames/s	RX-Warteschlange voll / HW-Überlauf	0 / 0
Eingereiht	4.302 Frames/s	Busfehler Sender / Empfänger	0 / 0
TX angenommen (gesamt)	120.296	TX fehlgeschlagen	0
RX empfangen (gesamt)	120.245	TX aktuell ausstehend	32
TX-Warteschlange voll / abgewiesen	159.311	Vom Scheduler übersprungen	392

Die Kurve der empfangenen Frames liegt über die gesamte Messdauer flach bei rund 4.300 Frames/s, ohne Einbrüche und ohne Abfall über die Zeit.

DER BUS WAR VOLL

Ein klassischer CAN-Frame mit 11-Bit-ID und 8 Datenbytes belegt 108 Bit, mit dem Pausenfeld zwischen zwei Frames 111 Bit. Durch Bit-Stuffing können bis zu 24 weitere Bit hinzukommen. Daraus ergibt sich eine theoretische Obergrenze:

Bitrate	Ohne Stuffing	Maximales Stuffing
500 kbit/s	ca. 4.505 Frames/s	ca. 3.704 Frames/s
1 Mbit/s	ca. 9.009 Frames/s	ca. 7.407 Frames/s

500.000 Bit/s geteilt durch 4.303 Frames/s ergibt rund 116 Bitzeiten pro Frame: 111 Bit plus durchschnittlich etwa 5 Stuff-Bits. Der gemessene Wert liegt im erwarteten Bereich, und der Bus hatte praktisch keine freie Bitzeit mehr.

WAS DIE ZÄHLER ZEIGEN

Der Sender war nicht der Engpass. Angenommene und abgewiesene Versuche ergeben zusammen 279.607 – ziemlich genau 28 Sekunden bei 10.000 Versuchen pro Sekunde. Die Warteschlange lief voll, weil der Bus die Frames nicht schneller abnehmen konnte.

Kein Datenverlust. Null Zählerlücken: Jeder eingereihte Frame kam in der richtigen Reihenfolge an. Die Differenz von 51 Frames zwischen den Summen erklärt sich plausibel durch die 32 noch ausstehenden Frames und die nicht synchronisierten Messfenster der beiden Boards.

GRENZEN DIESES TESTS

Zwei Knoten, eine ID. Arbitrierung zwischen mehreren Sendern wurde nicht getestet.

Kurze Leitung auf dem Steckbrett. Leitungslänge, Stichleitungen und Störungen wie im Fahrzeug spielen hier keine Rolle.

Momentaufnahme. Ein Lauf, aufgenommen nach etwa 28 von 60 Sekunden. Für belastbare Aussagen braucht es wiederholte und längere Läufe.

Volllast ist kein Betriebsziel. In echten Fahrzeugnetzen plant man mit deutlichem Puffer, damit hochpriorie Nachrichten jederzeit durchkommen.

Fehlende letzte Frames bleiben unbemerkt. Gehen die letzten Frames eines Tests verloren, erkennt der Zähler das allein nicht.

CAN-Lab White Paper

Bauanleitung, Teil 1: Hardware und Firmware

blog.andrelademann.de

1. HARDWARE BESORGEN

Zwei ESP32-C3-Boards, zwei SN65HVD230-Module, ein Steckbrett, Jumper-Kabel und zwei USB-Datenkabel (siehe Stückliste). Achte darauf, dass das Board einen **ESP32-C3** trägt: Die Sketches sind für den C3 SuperMini geschrieben. Andere ESP32-Varianten mit TWAI funktionieren grundsätzlich auch, brauchen aber angepasste Pins.

2. VERDRAHTEN

Löte die Stiftleisten an, stecke Boards und Module auf und verdrahte sie gemäß der Verdrahtungstabelle. Miss danach ohne Stromversorgung den Widerstand zwischen CANH und CANL: **etwa 60 Ω** bedeuten, dass beide 120- Ω -Abschlusswiderstände auf den Modulen vorhanden sind. Rund 120 Ω heißt, an einem Ende fehlt die Terminierung.

Verbinde niemals 3V3 mit 3V3 zwischen den Boards und betreibe die Transceiver nie mit 5 V.

3. FIRMWARE FLASHEN

```
git clone https://github.com/vergissberlin/example-canbus.git
cd example-canbus

# Sender flashen (erstes Board angeschlossen)
pio run -e sender -t upload

# Empfänger flashen (zweites Board angeschlossen)
pio run -c receiver.ini -e receiver -t upload
```

Voraussetzung ist [PlatformIO](#). Sind beide Boards gleichzeitig angeschlossen, wählst du den Port mit `--upload-port`.

DER KERN DER SENDER-KONFIGURATION

```
#include "driver/twai.h"

constexpr gpio_num_t CAN_TX_PIN = GPIO_NUM_5;
constexpr gpio_num_t CAN_RX_PIN = GPIO_NUM_4;

void setup() {
    twai_general_config_t g = TWAI_GENERAL_CONFIG_DEFAULT(
        CAN_TX_PIN, CAN_RX_PIN, TWAI_MODE_NORMAL);
    g.tx_queue_len = 32;
    twai_timing_config_t t = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f = TWAI_FILTER_CONFIG_ACCEPT_ALL();
    twai_driver_install(&g, &t, &f);
    twai_start();
}
```

Pins, Sende-Warteschlange, 500 kbit/s und ein Filter, der alle IDs annimmt.

CAN-Lab White Paper

Bauanleitung, Teil 2: CAN-Lab und Lasttest

blog.andrelademann.de

4. CAN-LAB VERBINDEN

1. Öffne canlab.andrelademann.de in **Chrome** oder **Edge**. Safari unterstützt Web Serial nicht.
2. Schließe zuerst den Empfänger an und klicke auf **Connect USB**, danach den Sender.
3. Keine Ausgabe? Drücke kurz **RESET** (nicht BOOT) am betroffenen Board.

Die USB-Daten bleiben in deinem Browser. CAN-Lab lässt sich als App installieren und funktioniert danach auch offline.

5. VERBINDUNG PRÜFEN

Im Normalbetrieb sendet der Sender einen Frame pro Sekunde. Steigt der Zähler **RX ID 0x123** beim Empfänger, funktioniert die CAN-Kommunikation. Ein **TX ... ok** beim Sender bestätigt nur das Einreihen, nicht den Empfang.

6. LASTTEST FAHREN

1. Starte vorsichtig: **100 Frames/s für 10 Sekunden**.
2. Läuft das ohne Lücken und Fehler, erhöhe auf 1.000, dann 5.000 und schließlich 10.000 Frames/s.
3. Wähle als Dauer 60 Sekunden und beobachte, wo die empfangene Rate abflacht. Rund 4.300 Frames/s bei 500 kbit/s ohne Lücken heißt: Dein Aufbau läuft am Limit des Busses.

7. FEHLERSUCHE

Keine Frames beim Empfänger: Fehlt die gemeinsame Masse? Das ist die häufigste Ursache.

Busfehler steigen sofort: CANH und CANL vertauscht oder Terminierung fehlt – Widerstand messen.

Kein serieller Port: Ladekabel ohne Datenleitungen, oder ein anderer serieller Monitor hält den Port.

Lücken bei hoher Rate: Fehlende USB-Logzeilen können Lücken vortäuschen. Maßgeblich sind die Firmware-Zähler.

AUSBLICK

Als Nächstes teste ich 1 Mbit/s, mehrere Sender mit unterschiedlichen IDs und längere verdrehte Leitungen. Danach zieht der Aufbau auf das Raspberry Pi 5 mit CAN-HAT und SocketCAN um, und dann geht es ins Auto. Wenn du den Test nachbaust, schick mir deine Werte.

QUELLEN

CAN-Lab
<https://canlab.andrelademann.de/>
vergissberlin/example-canbus – Sketches und CAN-Lab-Quellcode, GPL-3.0
<https://github.com/vergissberlin/example-canbus>
History of CAN technology – CAN in Automation
<https://www.can-cia.org/can-knowledge/history-of-can-technology>
CAN bus – Wikipedia, Frame-Format und Bit-Stuffing
https://en.wikipedia.org/wiki/CAN_bus