Arbeitsauftrag

Fach:	AIIT 5JG Höhere Elektrotechnik
Thema:	CAN Kommunikation
Lernziel:	Inbetriebnahme CAN am STM32 Datenaustausch zwischen CAN Knoten Temperaturmesswert übertragen
Zeitdauer:	3 x 2 Einheiten

Aufgabenstellung

Kommunikation zwischen zwei STM32 Boards mit CAN.

- Eine CAN Nachricht wird gesendet wenn der User Button betätigt wird.
- Zyklische Überprüfung ob eine CAN Nachricht angezeigt wird (Polling)

Inhalt des Telegramms / Frames:

Datenbyte 1: 0xAF
Datenbyte 2: Vorwärtszähler
Datenbyte 3: Reserviert
Datenbyte 4: Reserviert

Gesendete und empfangene Nachrichten werden am Display angezeigt.

```
Send-Cnt:      105
Recv-Cnt:      104
Send-Data:     100 af 01 00 00       // Identifier + Daten
Recv-Data:     101 ef bc             // Identifier + Daten
```

In einem weiteren Schritt werden Temperaturmesswerte über CAN übertragen, siehe tempsensor.c
Funktionen: tempSensorInit(), tempSensorGetTemperature()

Anleitung

- Aufruf der Funktionen canInit(), canSendTask() und canReceiveTask() aus main.c
- Die Funktionen canSendTask() und canReceiveTask() implementieren
- Verbindung mit CAN Transceiver Board (STM32 verwendet PB9 als CAN-TX und PB8 als CAN-RX)
- Test der CAN Verbindung
- Temperatursensor in Betrieb nehmen und Temperaturwerte in CAN Frame übertragen

Initialisierung

Die Initialisierung und das Verwalten der CAN Peripherie wird in einer Bibliothek durchgeführt - siehe src\stm32_ub_can1.c

Einstellung der CAN Baudrate in stm32_ub_can1.c

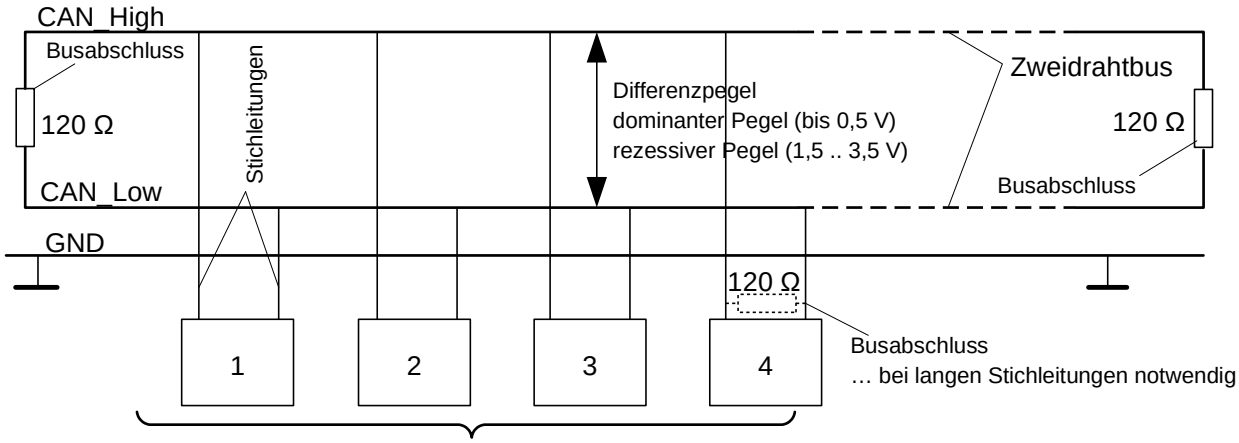
theoretischer Hintergrund siehe nächste Seite
CAN Konfiguration erfolgt in Funktion P1_init_CAN () im Modul stm32_ub_can1.c bzw. mit Define CAN1_CLOCK_PRESCALER in stm32_ub_can1.h

Filtereinstellungen

Die CAN Peripherie des STM32 kann erlaubt die Filterung der empfangenen Nachrichten (Filterung in Hardware!). Dies geschieht über Filter: Nur Nachrichten deren CAN IDs den Filtereinstellungen entsprechen, werden von der Peripherie entgegengenommen.
In der Vorlage ist die Filterung deaktiviert → Es werden alle Frames empfangen, siehe `P1_init_CAN()`

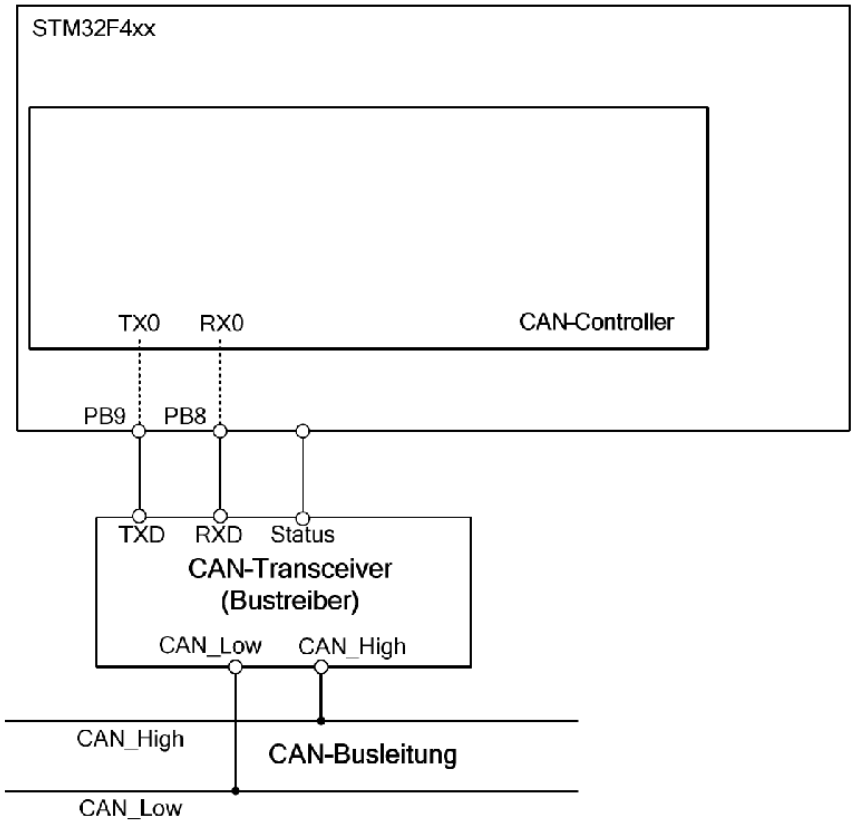
Grundlagen CAN-Bus

Topologie

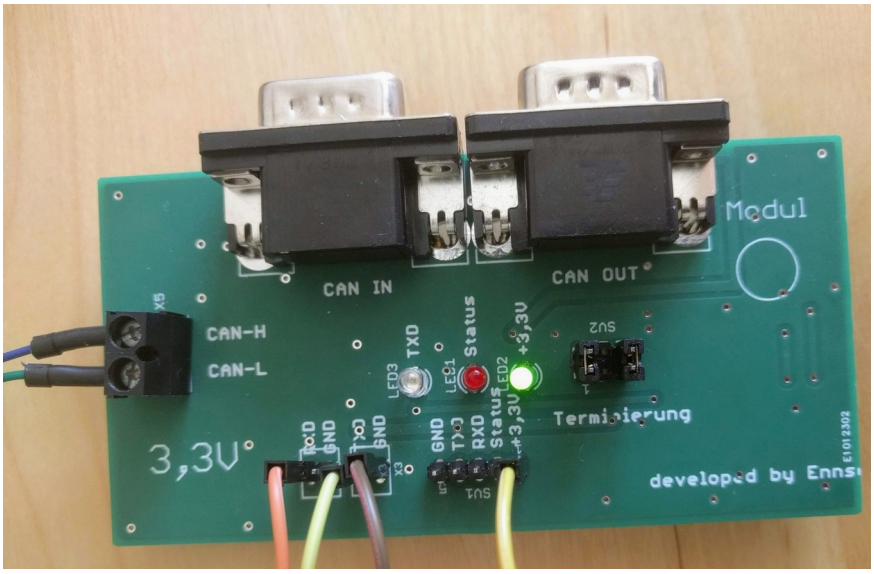


Alternativbezeichnung: Device, Station, Knoten, Node, Steuergerät

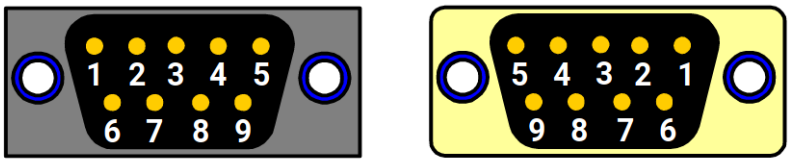
CAN am STM32F4xx



CAN-Transceiver



Anschlüsse zum µC: +3,3 V, GND, CAN_RX, CAN_TX mit Steckdrähte (Jumper Wire)
Anschlüsse zum CAN-Bus: D-Sub 9polig bzw. Schraubklemmen



Male Female

Pin	Signal	Description
1	Reserved	No connection, only pass-through
2	CAN_L	CAN Low bus line (dominant low)
3	CAN_GND	CAN Ground (see grounding section)
4	Reserved	No connection, only pass-through
5	(CAN_SHLD)	No connection, only pass-through
6	(GND)	CAN Ground (see grounding section)
7	CAN_H	CAN High bus line (dominant high)
8	Reserved	No connection, only pass-through
9	(CAN_V+)	No connection, only pass-through

Hinweis:
Bei der Versorgung von zwei STM32F4xxDiscovery-Boards und zwei CAN-Transceiver mit einem USB-Anschluss können Probleme auftreten. In diesem Fall wird eine externe Spannung empfohlen

Implementierung CAN Bus am STM32F4xx

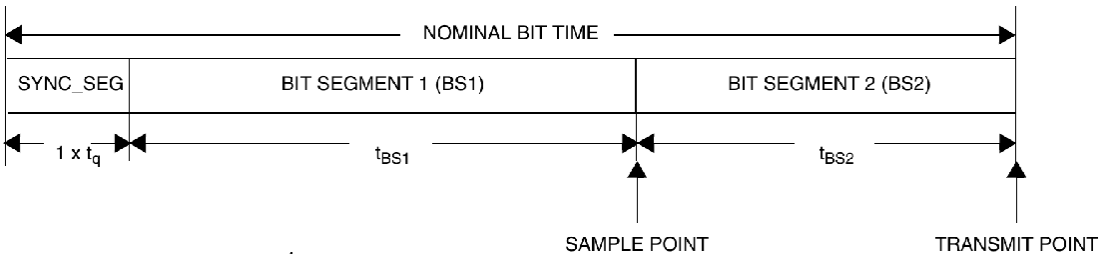
Exemplarischer Code für Senden / Empfangen:

```
// Struktur CAN1_TX_FRAME_t siehe stm32_ub_can1.h
typedef struct {
  uint32_t can_id;      // STD=11bit EXT=29bit
  uint8_t nr_bytes;     // Anzahl der Daten [0...8]
  uint8_t data[8];      // Datenbytes
} CAN1_TX_FRAME_t;
...
typedef struct {
  CAN1_FRAME_MODE_t frame_mode;
  uint32_t can_id;      // STD=11bit EXT=29bit
  uint8_t nr_bytes;     // Anzahl der Daten [0...8]
  uint8_t data[8];      // Datenbytes
} CAN1_RX_FRAME_t;

CAN1_TX_FRAME_t myTXFrame1; // Puffer für TX-Daten
CAN1_RX_FRAME_t myRXFrame1; // Puffer für RX-Daten

myTXFrame1.can_id = 0x100;
myTXFrame1.anz_bytes = 2;       // zwei Datenbytes werden gesendet
myTXFrame1.data[0] = 0xff;      // Datenwert 1
myTXFrame1.data[1] = sendCnt1;  // Datenwert 2

// Frame senden: Frame wird auf die Transmit Mailbox geschrieben
if (UB_CAN1_send_std_data(myTXFrame1) == SUCCESS) {
  ...
}

ret = UB_CAN1_receive (&myRXFrame1);
if(ret == CAN1_RX_READY) {
  // Empfangstelegramm verarbeiten
  ...
}
```

Einstellung CAN Baudrate

Allgemein: Die Baudrate wird mit dem Bit Timing und einem Prescaler (Define CAN1_CLOCK_PRESCALER) eingestellt.

Bit Timing: Mit dem Bit-Timing wird der Abtastzeitpunkt festgelegt.



Ein CAN-Bit wird aufgeteilt in synchronization segment (SYNC_SEG), bit segment 1 (BS1) und bit segment 2 (BS2). Der Abtastzeitpunkt liegt zwischen BS1 und BS2 und sollte bei ca. 75% liegen.

Die Länge der Segmente wird mit Zeitquanten (time quanta, t_q) bestimmt. SYNC_SEG ist konstant 1 t_q , BS1 kann eine Länge von 1-16 t_q und BS2 eine Länge von 1-8 t_q haben.

Die Bitzeit ergibt sich damit zu:

$$BitTime = 1 \cdot t_q + t_{BS1} + t_{BS2} \qquad BaudRate = \frac{1}{BitTime}$$

Ziel Abtastung bei ~75% → $BitTime = 1 \cdot t_q + 15 \cdot t_q + 6 \cdot t_q = 22 \cdot t_q$

Die Zeit t_q wird durch einen Prescaler aus dem Bustakt abgeleitet:

$$t_q = Prescaler * t_{APB1}$$

$$Baudrate = \frac{1}{22 t_q} = \frac{1}{22 \cdot Prescaler \cdot t_{APB1}} = \frac{f_{APB1}}{22 \cdot Prescaler}$$

Beispiel:
APB1-Takt: 45 MHz

```
#define CAN1_CLOCK_PRESCALER  16
```

→ $Baudrate = \frac{45\,MHz}{16 \cdot (1+15+6)} \approx 128\,kBaud/s$

Implementierung

Der Takt für den CAN-Bus wird ausgehend vom APB1-Takt mit dem Define CAN1_CLOCK_PRESCALER vorgegeben - siehe stm32f4xx_can.h und stm32_ub_can1.c .

```
CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
CAN_InitStructure.CAN_BS1 = CAN_BS1_15tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_6tq;
CAN_InitStructure.CAN_Prescaler = CAN1_CLOCK_PRESCALER
...
CAN_Init(CAN1, &CAN_InitStructure);
```

Hinweise:

- Grundsätzlich ist eine geeignete Terminierung notwendig (Pin 1 und 2 und Pin 5 und 6 vom SV2 am CAN-Transceiver verbinden). Für kleine Baudraten (bis ca. 50 kHz) kann die Terminierung auch weggelassen werden.
- Für den APB1-Takt 45 MHz ergeben sich keine exakten Werte für die Baudraten. Dazu müsste der PLL-Clock angepasst werden.