Busse und Sensoren

Twix

Chaos Computer Club Cologne e.V.
http://koeln.ccc.de

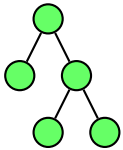
2017-10-30

Outline

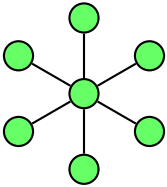
Was ist ein Bus?

- Wikipedia: Ein Bus ist ein System zur Datenübertragung zwischen mehreren Teilnehmern über einen gemeinsamen Übertragungsweg.
- Also ein Netzwerk?

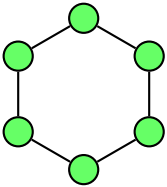
Netzwerktopologien



Baum



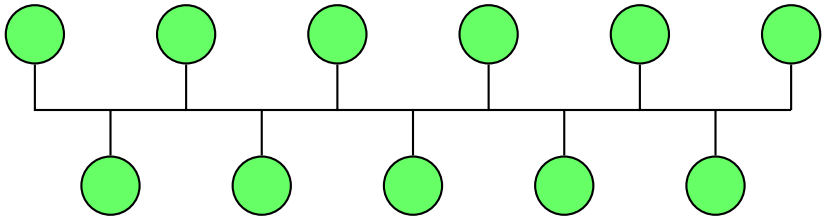
Stern



Ring

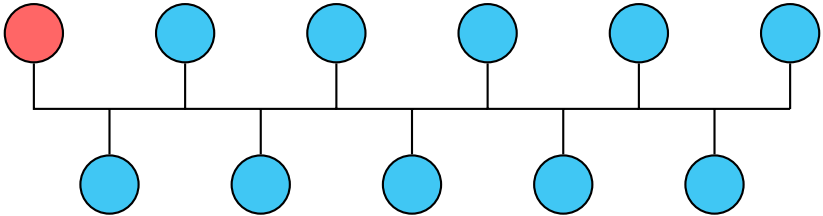
Nun, das sind keine Busse...

Bus Topologie



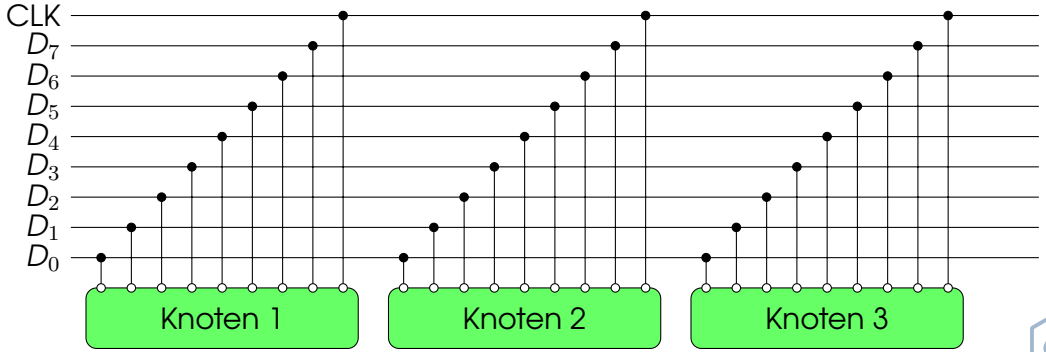
Ein Übertragungsweg für alle Knoten des Netzwerks

Master/Slave



Häufig wird die Kommunikation von einem Knoten koordiniert, dem Master.
Die anderen Knoten nennt man Slaves.

Paralleler Bus - Beispiel Schaltplan

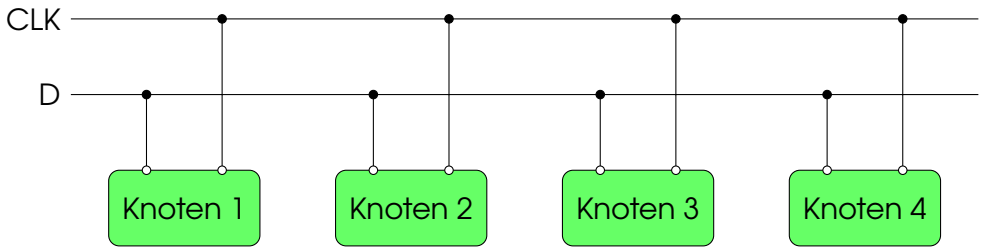


Paralleler Bus

- Mehrere Datenleitungen
- Pro Taktzyklus werden mehrere Bits übertragen
- Manchmal Addressleitungen und zusätzliche Steuerleitungen

Wir werden uns nicht weiter mit parallelen Bussen beschäftigen.

Serieller Bus



Serieller Bus

- In der Regel eine Datenleitung, bzw. eine Datenleitung pro Richtung
- Pro Taktzyklus wird nur ein Bit übertragen
- Um mehrere Bits oder gar Bytes zu Übertragen, sind also mehrere Taktzyklen nötig

Typische Serielle Busse

Ethernet	Ursprünglich ein Bus, heute fast nur noch Punkt zu Punkt Verbindungen
CAN-Bus	Controller Area Network, Feldbus z.B. im Auto

Für uns interessant sind insbesondere:

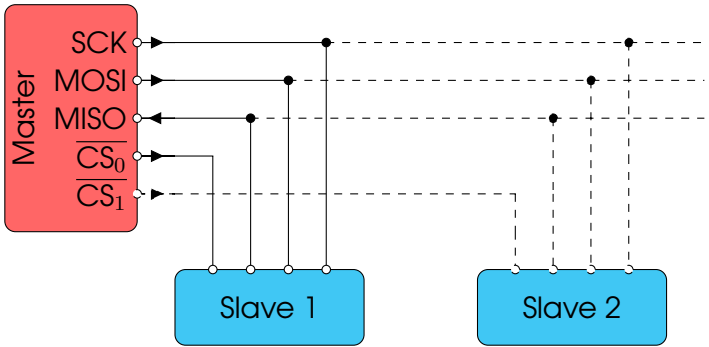
SPI	Serial Peripheral Interface
I2C	Inter-Integrated Circuit
1-Wire	auch One-Wire, Eindraht Bus

SPI Eigenschaften

- Voll Duplex, es wird gleichzeitig gesendet und empfangen
- 3 Leitugen für beidseitige Kommunikation

  SCK Takt, vom Master vorgegeben
  MOSI Master Out Slave In, Daten vom Master zum Slave
  MISO Master In Slave Out, Daten vom Slave zum Master
- In der Regel eine weitere Leitung pro Slave, häufig CS(Chip Select) oder SS(Slave Select) genannt, mit welcher der angesprochene Slave ausgewählt wird
- Grundsätzlich sehr lockerer Standard, mit mehreren Varianten

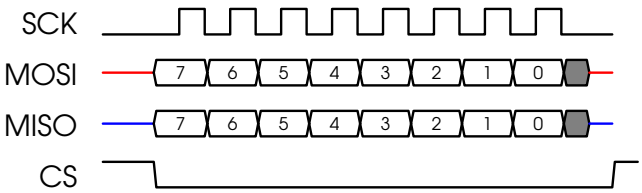
SPI Aufbau


Master
SCK
MOSI
MISO
$\overline{CS_0}$
$\overline{CS_1}$
Slave 1
Slave 2

Übertragung eines Bytes

1. CS wird Low, signalisiert damit die Aktivierung des Chips
2. SCK Taktet 8 mal
3. Master und Slave “schieben” während jedes Taktzyklusses ein Bit aus ihrem Ausgang heraus
4. CS wird wieder High, signalisiert damit das Ende der Übertragung

SPI Timing



Übertragung eines Bytes, MSB zu erst, CPOL=0(Clock Polarität), CPHA=0(Clock Phase)

SPI mit dem STM32 HAL

1. Definition eines Handles
   - eventuell als globale Variable
2. Clocks der verwendeten Peripherie einschalten
3. GPIOs konfigurieren
4. SPI konfigurieren

GPIOs konfigurieren

```
// Turn GPIOA Clock on
__HAL_RCC_GPIOA_CLK_ENABLE();
// Init SCK
HAL_GPIO_Init(GPIOA, &(GPIO_InitTypeDef){
        .Pin = GPIO_PIN_5, // Pin 5
        .Mode = GPIO_MODE_AF_PP, // Alternative Function(SPI)
        .Pull = GPIO_NOPULL, // Pull-Ups/Downs disabled
        .Speed = GPIO_SPEED_HIGH // Pin in High Speed mode
});
```

MISO auf Pin 6 und MOSI auf Pin 7 werden auf die selbe Art und Weise initialisiert. CS muss als Ausgang konfiguriert werden

SPI Hardware konfigurieren

```
SPI_HandleTypeDef hspi1; // Define the handle for later use
hspi1.Instance = SPI1; // We are going to use SPI1
hspi1.Init = (SPI_InitTypeDef){
.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_8, // SPI Clock
.Direction = SPI_DIRECTION_2LINES, // Full duplex needs 2 lines
.DataSize = SPI_DATASIZE_8BIT, // We are going to transfer Bytes
.CRCCalculation = SPI_CRCCALCULATION_DISABLED,
.CLKPhase = SPI_PHASE_1EDGE, // Sampling at first Clock edge
.CLKPolarity = SPI_POLARITY_LOW, // Clock is Low when idle
.Mode = SPI_MODE_MASTER, // We are going to be master
.NSS = SPI_NSS_SOFT, // We want to handle our CS by ourselfs
.TIMode = SPI_TIMODE_DISABLED,
.FirstBit = SPI_FIRSTBIT_MSB };
```

Initialisierung der SPI Hardware

```
// Turn SPI1 Clock on
__HAL_RCC_SPI1_CLK_ENABLE();

// Initialize SPI1
if (HAL_SPI_Init(&hspi1) != HAL_OK)
{
        // Error Handling
}
```

Senden und Empfangen eines Bytes

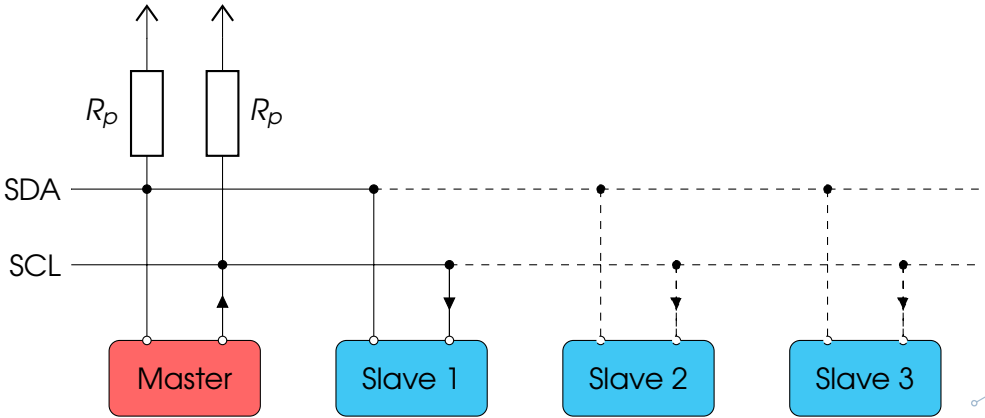
```
uint8_t result;

// Enable Chip Select
HAL_GPIO_WritePin(GPIOA,GPIO_PIN_4,GPIO_PIN_RESET);
// Transmit byte in outval and receive byte to result
HAL_SPI_TransmitReceive(&hspi1,&outval,&result,1,0xff);
// Disable Chip Select
HAL_GPIO_WritePin(GPIOA,GPIO_PIN_4,GPIO_PIN_SET);
```

I2C Eigenschaften

- Halb Duplex, es kann nur gesendet oder empfangen werden
- Zwei Leitungen
  - SCL Takt, wird vom Master vorgegeben
  - SDA Datenleitung
    - Alle Bus Teilnehmer schreiben auf diese eine Datenleitung
- Üblicherweise 112 Geräte adressierbar

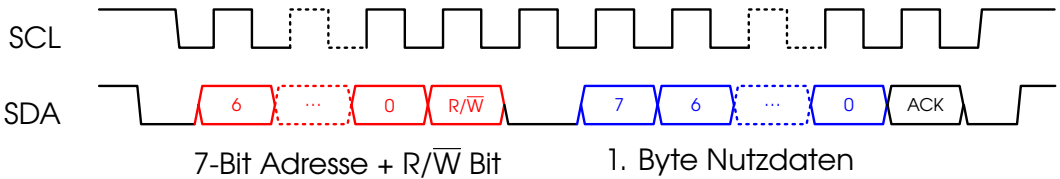
I2C Aufbau



Elektrische Eigenschaften

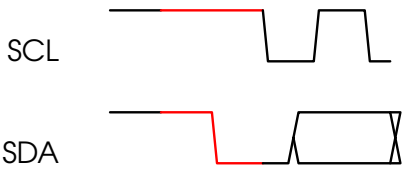
- Positive Logik, d.h. $> 0,7 \cdot V_{dd}$ entspricht High, $< 0,3 \cdot V_{dd}$ entspricht Low
- Beide Leitungen werden mit einem Pull-Up Wiederstand auf High gezogen
- Alle Geräte haben Open-Collector Ausgänge
  - Aktiv wird also nur der Low-Pegel erzeugt
  - Es kann nicht festgestellt werden, welches Gerät den Low-Pegel verursacht
- Änderung von SDA nur zulässig, wenn SCL Low ist
- Für ACK-Bits wird Low Pegel als ACK und High Pegel als NACK interpretiert

I2C Timing Diagramm



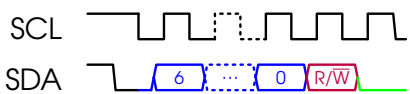
Ablauf einer I2C Kommunikation

Start Condition



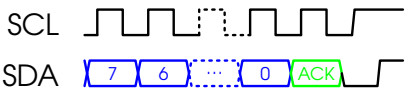
- Kommunikation beginnt mit der Start-Condition
- SDA wird Low während SCL High ist

Addressierung des Slaves



- Nach der Start-Condition werden 7 Adressbits(blau) übertragen, sowie das R/$\overline{W}$-Bit(lila)
  - Das R/$\overline{W}$-Bit gibt an, ob von dem Gerät gelesen oder auf das Gerät geschrieben werden soll
- Slave bestätigt(ACK) in dem für die Dauer eines Bits SDA auf Low gezogen wird(grün)

Übermittlung von Daten



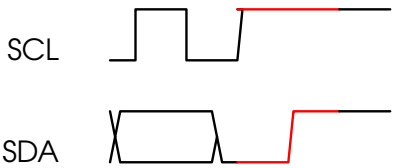
Lesend($R/\overline{W} = H$):

- Adressierter Slave sendet ein Byte(blau)
- Master quittiert mit ACK, wenn weitere Bytes gelesen werden sollen(grün)
- Sonst: Master quittiert mit NACK

Schreibend($R/\overline{W} = L$):

- Master sendet Byte(blau)
- Adressierter Slave quittiert mit ACK(grün)

Stop Condition



- Das Ende der Übertragung signalisiert der Master mittels Stop-Condition
- SDA wird High, während SCL High ist(rot)

Praxis mit dem STM32 HAL

Grundsätzlich: Ähnliche Vorgehensweise wie bei der SPI Hardware.
Unterschiede:

- Die GPIOs werden nicht als Push-Pull, sondern als Open-Drain konfiguriert
- Es wird direkt die gewünschte Bitrate eingestellt
- Insgesamt weniger Einstellmöglichkeiten

GPIOs konfigurieren

```
// Turn GPIOB Clock on
__HAL_RCC_GPIOB_CLK_ENABLE();

// Configure PB6 for SCL
HAL_GPIO_Init(GPIOB, &(GPIO_InitTypeDef){
        .Mode = GPIO_MODE_AF_OD, // AF Open Drain
        .Pin = GPIO_PIN_6, // Pin 6
        .Pull = GPIO_NOPULL, // Pull-Ups/Downs disabled
        .Speed = GPIO_SPEED_HIGH // Pin in High Speed mode
});
```

SDA auf Pin 7 wird auf die selbe Art initialisiert.

I2C Hardware konfigurieren

```
I2C_HandleTypeDef hi2c1; // Define the handle for later use
hi2c1.Instance = I2C1; // We are going to use I2C1
hi2c1.Init = (I2C_InitTypeDef){
        .ClockSpeed = 400000, // Bitrate of 400 kbit/s
        .DutyCycle = I2C_DUTYCYCLE_2,
         // We have 7 Bit adresses
        .AddressingMode = I2C_ADDRESSINGMODE_7BIT,
        .GeneralCallMode = I2C_GENERALCALL_DISABLE,
        .DualAddressMode = I2C_DUALADDRESS_DISABLE
};
```

Initialisierung der I2C Hardware

```
// Turn I2C1 clock on
__HAL_RCC_I2C1_CLK_ENABLE();

// Init I2C1
if (HAL_I2C_Init(&hi2c1) != HAL_OK)
{
        // Error handling
}
```

Daten versenden und empfangen

```
// Address of BMP280 is 0x76
#define BMP_ADDR (0x76<<1)
uint8_t id = 0;
// Write Byte 0xd0 to BMP280. Writing only one Byte will be
// interpreted as "read this register"
HAL_I2C_Master_Transmit(&hi2c1, BMP_ADDR, &(uint8_t){0xd0}, 1,
 ↪  0xff);
// Read one Byte from BMP280 to our variable id.
// It will be the content of Register 0xd0
HAL_I2C_Master_Receive(&hi2c1, BMP_ADDR, &id, 1, 0xff);
```

BMP280

- Temperatur- und Luftdruck-Sensor
- Kalibriert, Kalibrationsdaten liegen in einem ROM im IC(Ab Adresse 0x88)
- Möglichkeit auf Anfrage einen Messvorgagn zu starten oder in regelmäßigen Abständen
- Möglichkeit, schneller mit weniger Genauigkeit zu messen

Messvorgang(einzeln)

Vorbereitend:

- Kalibrationsdaten aus dem Rom auslesen(an Adresse 0x88)

Für jeden Messvorgang:

1. Messvorgang auslösen(in Register "ctrl_meas"(0xF4) schreiben)
2. Warten, das Ende des Messvogangs wird im Register "status"(0xF3) signalisiert
3. Messwerte aus dem Register temp(0xFA bis 0xFC) holen
4. Messwerte aus den Register press(0xF7 bis 0xF9) holen
5. Umrechnungsfunktionen anwenden(siehe Datenblatt)

lora-bone BSP

Das Board Support Package für den lora-bone enthält einige intialisierungs Funktionen.

```
#include "lora-bone.h"

int main(void)
{
        bone_initI2c1();
        ...
}
```

Den BSP findet ihr im Buildenv in middlewares/bsp/lora-bone

Ein paar hilfreiche Dokumente

- http://www.st.com/resource/en/user_manual/dm00154093.pdf
- https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMP280-DS001-18.pdf

Aufgaben

1. Lest das Feld id(0xd0) wie im letzem Kapitel gezeigt aus.
2. Schreibt in das Feld "ctrl_meas"(Adresse 0xF4) den Wert 0xA2, wartet ca. 200 ms und lest dann die Felder 0xFA bis 0xFC aus. Wenn die drei Bytes in einem Array `uint8_t buf[3];` liegen, könnt ihr einen Integer auf folgende Weise zusammen bauen:

```
uint32_t UT = (buf[0]<<12)|(buf[1]<<4)|(buf[2]>>4);
```