Übungen zu Grundlagen der systemnahen Programmierung in C (GSPiC) im Wintersemester 2018/19

2018-11-21

Alexander von der Haar
Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Beliebte Fehler

Sichtbarkeit & Lebensdauer

Sichtbarkeit und Lebensdauer	nicht `static`	`static`
lokale Variable	Sb Block Ld Block	Sb Block Ld Programm
globale Variable	Sb Programm Ld Programm	Sb Modul Ld Programm
Funktion	Sb Programm	Sb Modul

- Lokale Variable, nicht `static` = `auto` Variable

⇝ automatisch allokiert & freigegeben

- Funktionen als `static`, wenn kein Export notwendig

Globale Variablen

```
01 static uint8_t state; // global static
02 uint8_t event_counter; // global
03
04 void main(void) {
05     ...
06 }
07
08 static void f(uint8_t a) {
09     static uint8_t call_counter = 0; // local static
10     uint8_t num_leds; // local (auto)
11     ...
12 }
```

- Sichtbarkeit/Gültigkeit möglichst weit **einschränken**
- Globale `static` Variablen: Sichtbarkeit auf Modul beschränken

⇝ wo möglich, `static` für Funktionen und Variablen verwenden

Verwendung falscher Datentypen

- Die Größe von `int` ist in C nicht genau definiert
- zum Beispiel beim ATmega328PB: 16 bit
  - ⇒ Gerade auf µC führt dies zu Fehlern und/oder langsameren Code
- Für die Übung:
  - Verwendung von `int` ist ein “Fehler”
  - Stattdessen: Verwendung der in der `stdint.h` definierten Typen: `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, etc.
- Wertebereich:
  - `limits.h`: `INT8_MAX`, `INT8_MIN`, ...
- Speicherplatz ist sehr teuer auf µC

⇝ Nur so viel Speicher verwenden, wie tatsächlich benötigt wird!

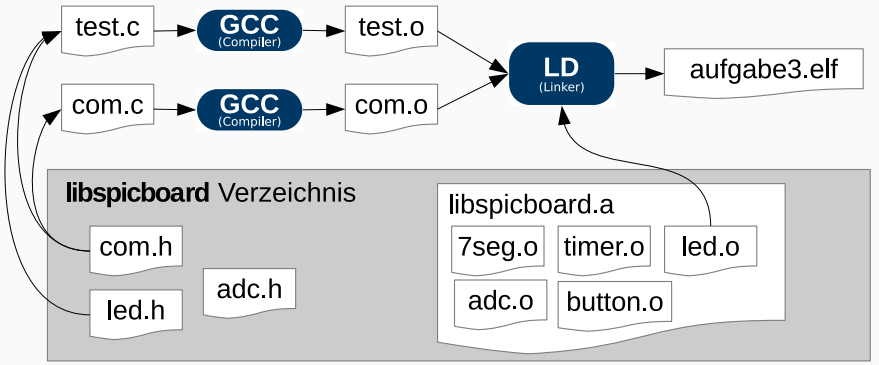
Typedefs & Enums

```
01 #define PB3 3
02 typedef enum { BUTTON0 = 4, BUTTON1 = 8 } BUTTON;
03 #define MAX_COUNTER 900
04 ...
05 void main(void) {
06     ...
07     PORTB |= (1 << PB3); // nicht (1 << 3)
08     ...
09     BUTTONSTATE old, new; // nicht uint8_t old, new;
10     ...
11     // Deklaration: BUTTONSTATE sb_button_getState(BUTTON btn);
12     old = sb_button_getState(BUTTON0);//nicht sb_button_getState
       ↪ (4)
13     ...
14 }
```

- Vordefinierte Typen verwenden
- Explizite Zahlenwerte nur verwenden, wenn notwendig

Module

Ablauf vom Quellcode zum laufenden Programm



1. Präprozessor
2. Compiler
3. Linker
4. Programmer/Flasher

Schnittstellenbeschreibung

- Header Dateien enthalten die Schnittstelle eines Moduls
  - Funktionsdeklarationen
  - Präprozessormakros
  - Typdefinitionen
- Header Dateien können u.U. mehrmals eingebunden werden
  - `led.h` bindet `avr/io.h` ein
  - `button.h` bindet `avr/io.h` ein
  - ⇝ Funktionen aus `avr/io.h` mehrmals deklariert
- Mehrfachinkludierung/Zyklen vermeiden ⇝ **Include-Guards**
  - Definition und Abfrage eines Präprozessormakros
  - Konvention: Makro hat den Namen der .h-Datei, '.' ersetzt durch '_'
  - z.B. für `button.h` ⇝ BUTTON_H
  - Inhalt nur einbinden, wenn das Makro noch nicht definiert ist
- Vorsicht: flacher Namensraum ⇝ möglichst eindeutige Namen

Schnittstellenbeschreibung

- Erstellen einer .h-Datei (Konvention: gleicher Name wie .c-Datei)

```
01 #ifndef COM_H
02 #define COM_H
03 /* fixed-width Datentypen einbinden (im Header verwendet) */
04 #include <stdint.h>
05
06 /* Datentypen */
07 typedef enum {
08     ERROR_NO_STOP_BIT, ERROR_PARITY,
09     ERROR_BUFFER_FULL, ERROR_INVALID_POINTER
10 } COM_ERROR_STATUS;
11
12 /* Funktionen */
13 void sb_com_sendByte(uint8_t data);
14 ...
15 #endif //COM_H
```


Initialisierung eines Moduls

- Module müssen Initialisierung durchführen
  - zum Beispiel Portkonfiguration
  - **Java:** mit Klassenkonstruktoren möglich
  - **C:** kennt kein solches Konzept
- *Workaround:* Modul muss bei erstem Aufruf einer seiner Funktionen ggf. die Initialisierung durchführen
  - muss sich merken, ob die Initialisierung schon erfolgt ist
  - Mehrfachinitialisierung vermeiden
- Anlegen einer Init-Variable
  - Aufruf der Init-Funktion bei jedem Funktionsaufruf
  - Init-Variable anfangs 0
  - Nach der Initialisierung auf 1 setzen

Initialisierung eines Moduls

- `initDone` ist initial `0`
- wird nach der Initialisierung auf 1 gesetzt
- ⇝ Initialisierung wird nur ein mal durchgeführt

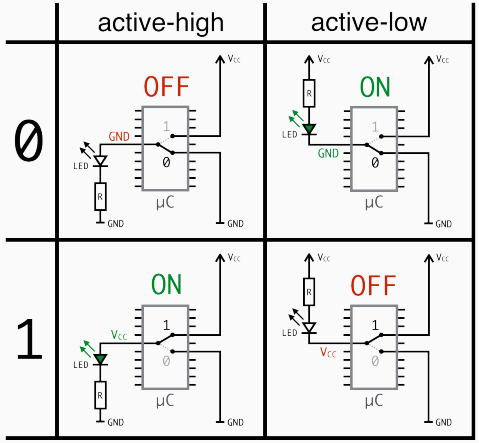
```
01 static void init(void){
02     static uint8_t initDone = 0;
03     if (initDone == 0) {
04       initDone = 1;
05       ...
06     }
07 }
08
09 void mod_func(void) {
10     init();
11     ...
```

Ein- & Ausgabe über Pins

Ausgang: active-high & active-low

Ausgang je nach Beschaltung:

active-high high-Pegel (logisch 1; V_{cc} am Pin) → LED leuchtet
active-low low-Pegel (logisch 0; *GND* am Pin) → LED leuchtet

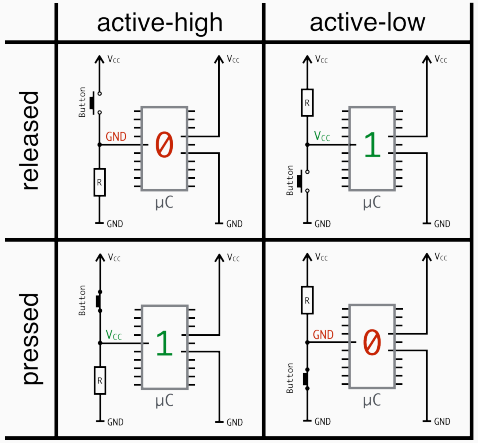


Eingang: active-high & active-low

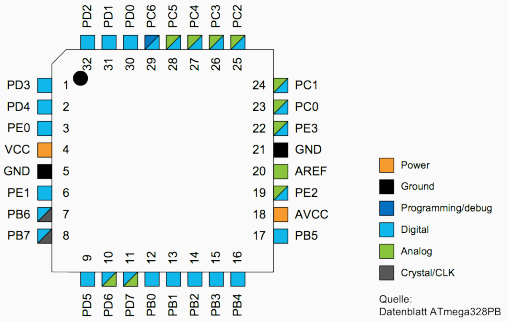
Eingang je nach Beschaltung:

active-high Button gedrückt → high-Pegel (logisch 1; V_{cc} am Pin)
active-low Button gedrückt → low-Pegel (logisch 0; *GND* am Pin)

interner pull-up-Widerstand (im ATmega328PB) konfigurierbar



Konfiguration der Pins



- jeder I/O-Port des AVR-µC wird durch drei 8-bit Register gesteuert:
  - Datenrichtungsregister (`DDRx` = data direction register)
  - Datenregister (`PORTx` = port output register)
  - Port Eingabe Register (`PINx` = port input register, nur-lesbar)
- jedem Anschluss-Pin ist ein Bit in jedem der 3 Register zugeordnet

I/O-Port-Register

DDRx hier konfiguriert man Pin i von Port x als Ein- oder Ausgang

- `Bit i = 1` → Pin i als Ausgang verwenden
- `Bit i = 0` → Pin i als Eingang verwenden

PORTx Auswirkung **abhängig von DDRx**:

- ist Pin i **als Ausgang konfiguriert**, so steuert Bit i im PORTx Register ob am Pin i ein high- oder ein low-Pegel erzeugt werden soll
  - `Bit i = 1` → high-Pegel an Pin i
  - `Bit i = 0` → low-Pegel an Pin i
- ist Pin i **als Eingang konfiguriert**, so kann man einen internen pull-up-Widerstand aktivieren
  - `Bit i = 1` → pull-up-Widerstand an Pin i (Pegel wird auf high gezogen)
  - `Bit i = 0` → Pin i als tri-state konfiguriert

PINx Bit i gibt aktuellen Wert des Pin i von Port x an (nur lesbar)

Beispiel: Initialisierung eines Ports

- Pin 3 von Port C (PC3) als Ausgang konfigurieren und PC3 auf Vcc schalten:

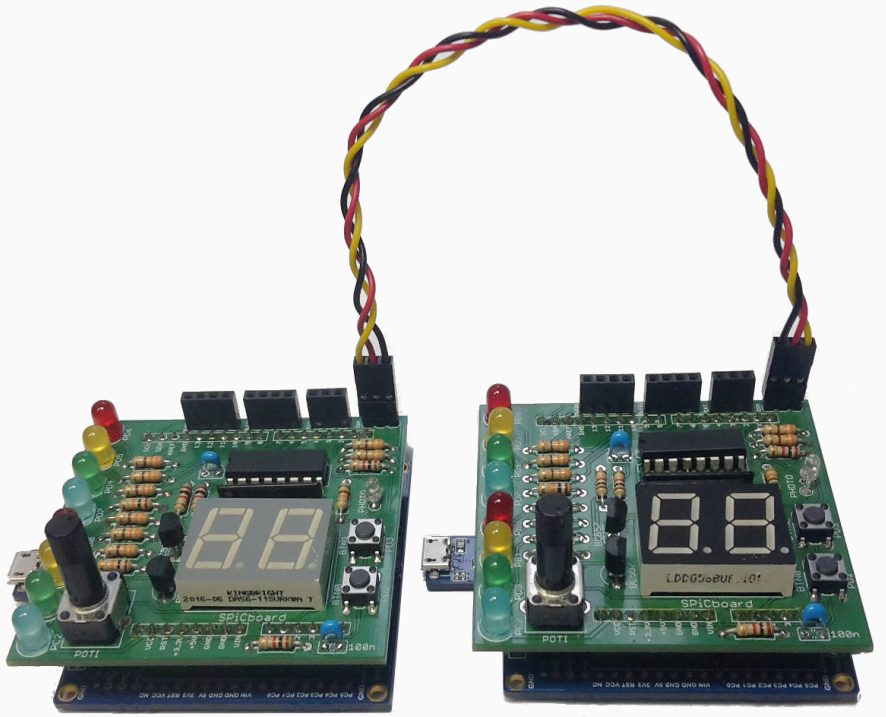
```
01 DDRC |= (1 << PC3); /* =0x08; PC3 als Ausgang nutzen... */
02 PORTC |= (1 << PC3); /* ...und auf 1 (=high) setzen */
```

- Pin 2 von Port D (PD2) als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
01 DDRD &= ~(1 << PD2); /* PD2 als Eingang nutzen... */
02 PORTD |= (1 << PD2); /* pull-up-Widerstand aktivieren */
03 if((PIND & (1 << PD2)) == 0){ /* den Zustand auslesen */
04   /* ein low Pegel liegt an, der Taster ist gedrückt */
05 }
```

- Die Initialisierung der Hardware wird in der Regel einmalig zum Programmstart durchgeführt

Kommunikation



SPiCboard verbinden

PD1 Ausgang (TX) wird mit RX des Kommunikationspartners verbunden

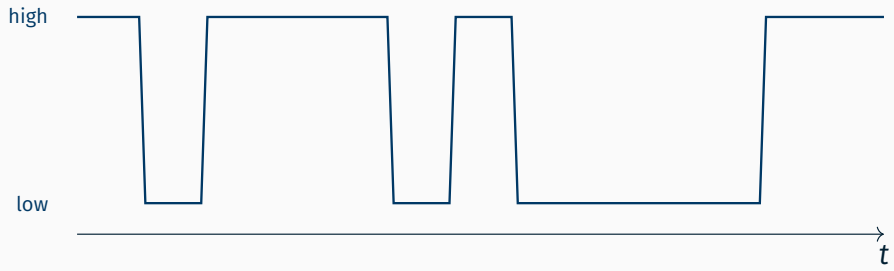
PD0 Eingang (RX) analog mit Ausgang (TX) verbinden

GND wird mit GND verbunden

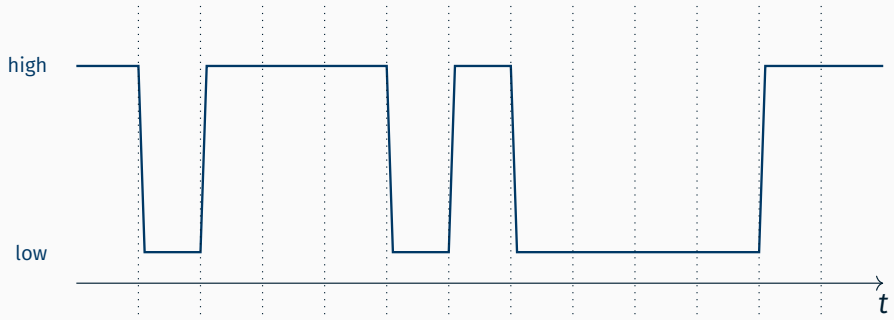
Voraussetzungen für den Datenaustausch

- gleiches Protokoll
  - wir nehmen `8-E-1`
    - **`8`** Anzahl der Datenbits
    - **`E`** gerades (*even*) Paritätsbit
    - **`1`** Stopbit
  - sowie (implizit) ein Startbit
  - das Datenbit mit dem niedrigsten Stellenwert wird zuerst übertragen (aufsteigend)
  - hoher Pegel entspricht einer logischen `1`, niedriger Pegel einer `0`
- gleiche Geschwindigkeit, bei uns `1200` Bd
  (`1` Baud entspricht ein Symbol pro Sekunde)

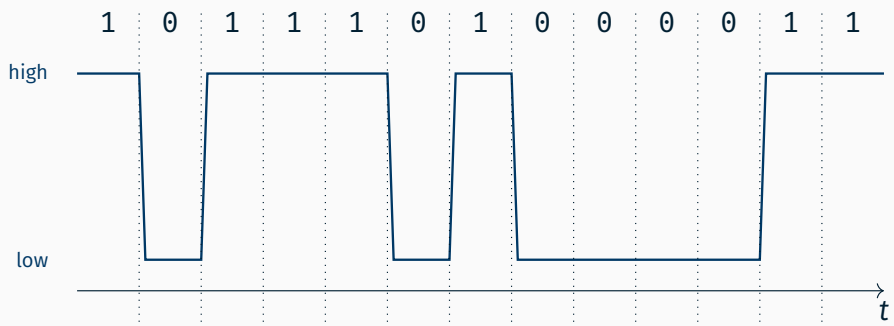
Beispiel



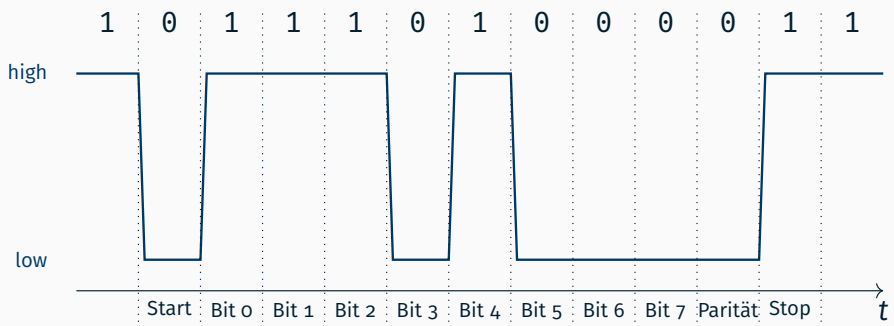
Beispiel



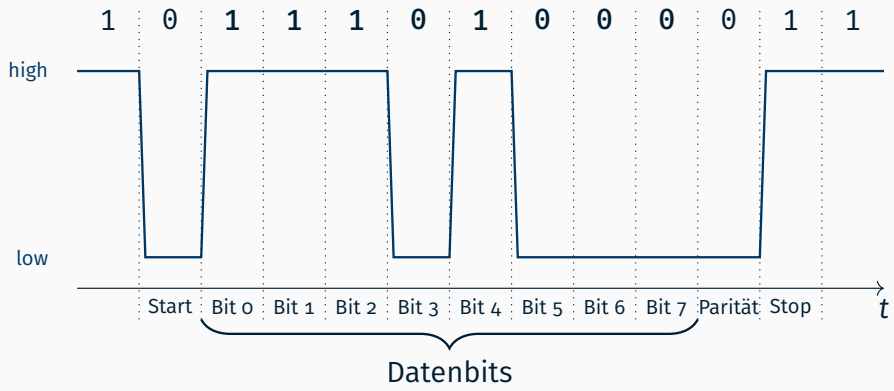
Beispiel



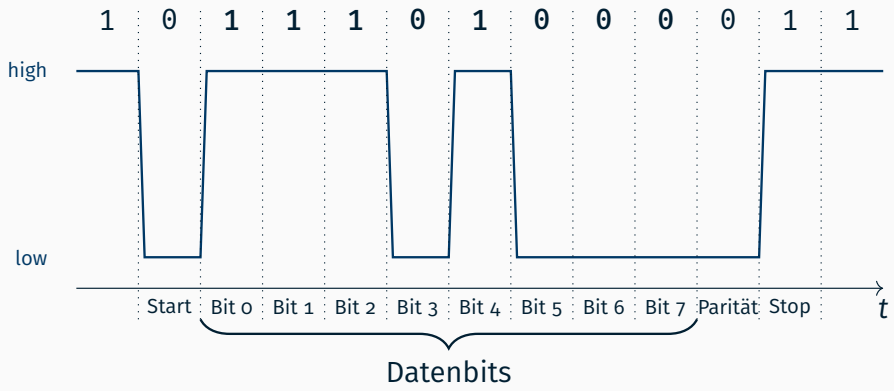
Beispiel



Beispiel

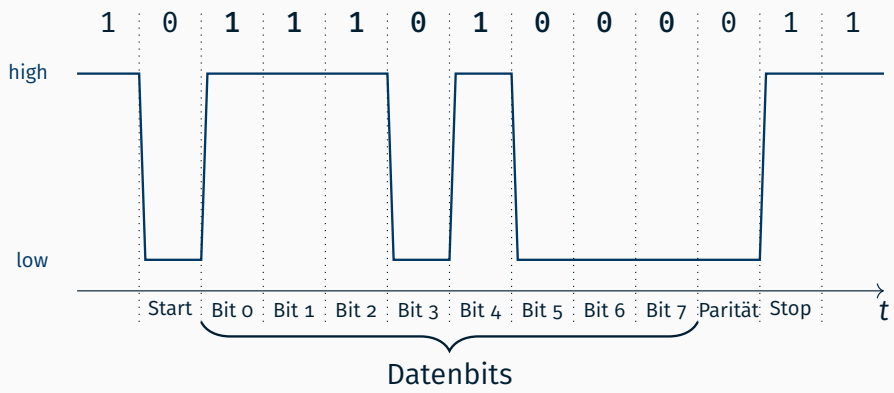


Beispiel



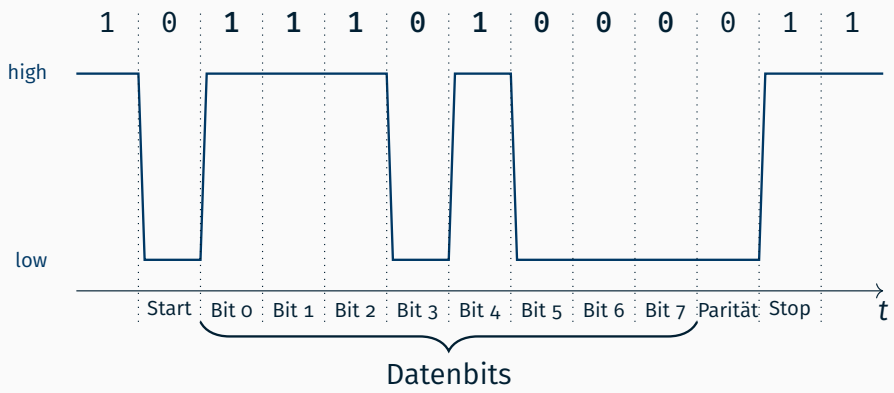
- Empfangenes Byte (rückwärts gelesen): $00010111_2 = 0x17 = 23$

Beispiel



- Empfangenes Byte (rückwärts gelesen): $00010111_2 = 0x17 = 23$
- Parität `0` (da Datenbits vier `1`er ⇒gerade Anzahl)

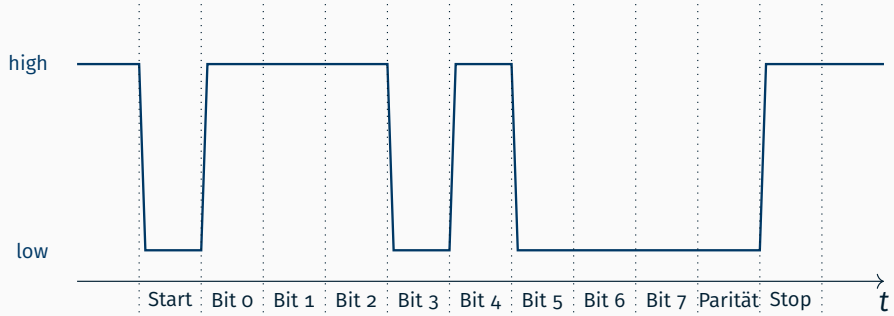
Beispiel



- Empfangenes Byte (rückwärts gelesen): $00010111_2 = 0x17 = 23$
- Parität `0` (da Datenbits vier `1`er ⇒gerade Anzahl)
- Startbit immer `0` und Stopbit immer `1`

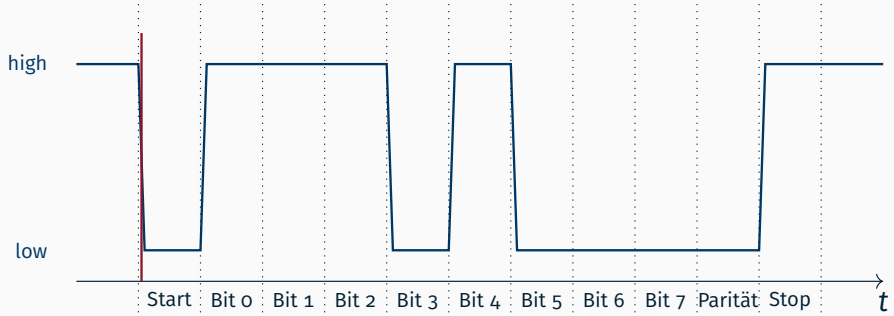
18

Abstastung



- Die *Fenstergröße* **T** ist uns bekannt

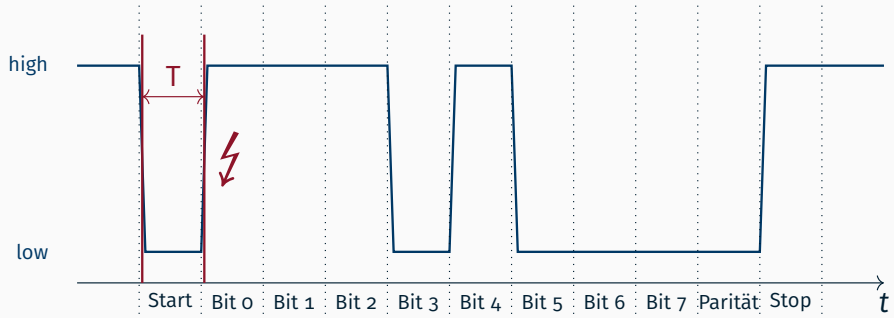
Abstastung



- Die *Fenstergröße* **T** ist uns bekannt
- Anfang ist einen Pegelwechsel von hoch auf niedrig

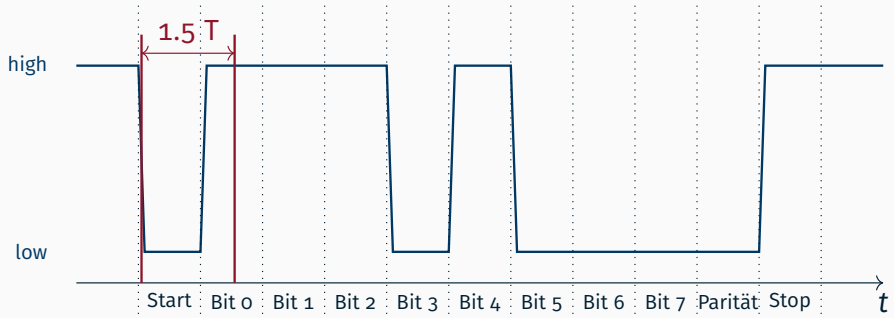
19

Abstastung



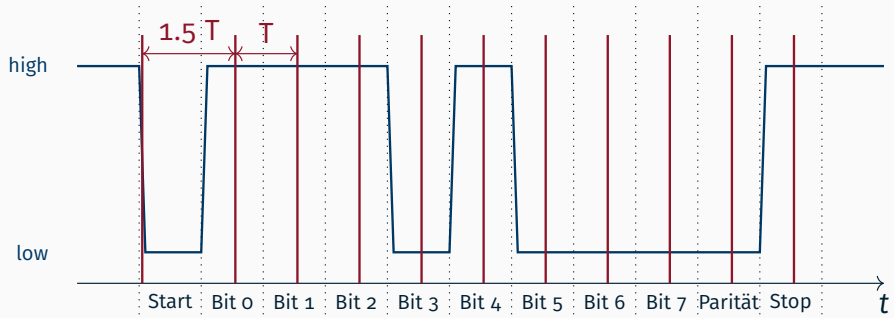
- Die *Fenstergröße* **T** ist uns bekannt
- Anfang ist einen Pegelwechsel von hoch auf niedrig

Abstastung



- Die *Fenstergröße* **T** ist uns bekannt
- Anfang ist einen Pegelwechsel von hoch auf niedrig
- warte danach **1.5 T** für ein eindeutiges Symbol

Abstastung



- Die *Fenstergröße* **T** ist uns bekannt
- Anfang ist einen Pegelwechsel von hoch auf niedrig
- warte danach **1.5 T** für ein eindeutiges Symbol
- für jedes weitere Symbol wieder alle **T** abtasten

Aufgabe: Eigenes Kommunikationsmodule

COM-Modul – Aufgabe

- COM-Modul der SPiCboard-Bibliothek selbst implementieren
  - Modulschnittstelle bietet Funktionen für:
    - Versenden eines Bytes
    - Empfangen eines Bytes
  - Schnittstellendatei (`com.h` liegt im pub Ordner)
- Gleiches Verhalten wie das Original
- Beschreibung in der Doku auf der Webseite:

`https://www4.cs.fau.de/Lehre/WS18/V_GSPIC/SPiCboard/libapi.shtml`

```
01 void sb_com_sendByte(uint8_t data);
02 uint8_t sb_com_receiveByte(uint8_t *data);
```


COM-Modul – Aufgabe

- Hilfsfunktionen zum Setzen/Auslesen der Register
  - Setzen des Pegels für das Senderegister TX
  - Auslesen des Pegels für das Empfangsregister RX
- Hilfsfunktionen gehören nicht zur Schnittstelle
  - ⇝ Sichtbarkeit einschränken
- Analog: Hilfsfunktion für die Initialisierung der Register
  - ⇝ `sb_com_init()`

```
01 static void sb_com_setTx(uint8_t bit);
02 static uint8_t sb_com_getRx();
```


COM-Modul – Aufgabe

- Funktionen der Schnittstelle müssen überall sichtbar sein
  - ⇝ Sichtbarkeit nicht einschränken
- Kompatibilität von Schnittstelle und Implementierung durch Inkludieren des Headers gewährleistet
  - Fehlercodes (`COM_ERROR_STATUS`)/Makros ebenfalls enthalten
  - Warum sind die Fehlercodes/Makros nicht in der `.c` Datei?

```
01 #include <adc.h>
02 #include <com.h>
03
04 void main(void) {
05     while(1) {
06         sb_com_sendByte(sb_adc_read(POTI) / 4);
07     }
08 }
```


COM-Modul – Aufgabe

- Aufgabe ist in drei Teile unterteilt
- Teilaufgabe a: Initialisierung
  - Einmaliges Initialisieren der Empfangs- und Senderegister
  - Implementieren der Hilfsfunktion `sb_com_init()`
  - Wird bei jedem Sende- oder Empfangsvorgang aufgerufen
  - Implementieren der Hilfsfunktionen `sb_com_setTx()` und `sb_com_getRx()`
  - Können benutzt werden, um den RX bzw. TX Pin zu lesen/setzen

COM-Modul – Aufgabe

- Aufgabe ist in drei Teile unterteilt
- Teilaufgabe b: Sendefunktionalität
  - Unabhängig vom Empfangen implementieren
  - Implementieren der Funktion `sb_com_sendByte()`
  - Ablauf für das zu sendende Byte:
    - Senden des Startbits
    - Senden der Nutzdaten (LSB zuerst)
    - Senden des Paritätsbits
    - Senden des Stoppbits
  - Verwendung von `sb_com_wait(1.0)` um ein Symbol zu warten
  - Testen der Sendefunktionalität mit dem vorgebenen Programm `test_receiver.elf`

COM-Modul – Aufgabe

- Aufgabe ist in drei Teile unterteilt
- Teilaufgabe c: Empfangsfunktionalität
  - Unabhängig vom Senden implementieren
  - Implementieren der Funktion `sb_com_receiveByte()`
  - Ablauf für das zu empfangende Byte:
    - Warten auf fallende Flanke an RX
    - Empfangen des Startbits
    - Empfangen der Nutzdaten
    - Empfangen des Paritätsbits
    - Empfangen des Stoppbits
  - Treten Fehler während der Übertragung auf, wird der Empfangsvorgang bis zum Ende ausgeführt und der entsprechende Fehlercode zurückgegeben
  - Testen der Empfangsfunktionalität mit dem vorgebenen Programm `test_sender.elf`

COM-Modul – Aufgabe

- Testen der finalen Implementierung mit einer Anwendung
- Beispielanwendung `auf100`
  - 2-Personen-Spiel zum Testen des Kommunikationsmoduls
    Spielprinzip: Es wird mit einer zufälligen Zahl begonnen, die auf der 7-Segmentanzeige dargestellt wird. Jeder Spieler kann diese Zahl nun abwechselnd durch drehen des Potentiometers um 1 bis 8 erhöhen. Der Spieler, der die 100 erreicht hat gewonnen.