

Übungen zu Systemnahe Programmierung in C (SPiC) – Wintersemester 2018/19

Übung 3

Rebecca Felsheim
Benedict Herzog

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



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

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



- Die Größe von `int` ist in C nicht genau definiert
 - zum Beispiel beim ATmega328PB: 16 bit
 - ⇒ Gerade auf μ C 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 μ C
- ~ Nur so viel Speicher verwenden, wie tatsächlich benötigt wird!

```

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
13     //      ↪ (4)
14     ...
15 }

```

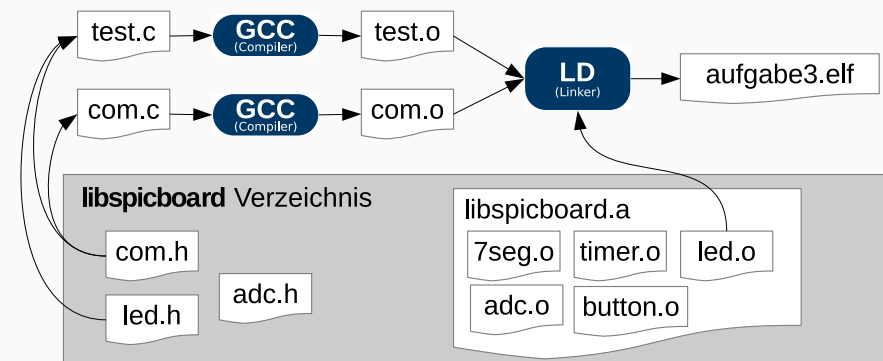
- Vordefinierte Typen verwenden
- Explizite Zahlenwerte nur verwenden, wenn notwendig

3

4

Module

Ablauf vom Quellcode zum laufenden Programm



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



- 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

6



- 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

```

7



- 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

8



- 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     ...

```

9



Zeiger & Felder

- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```
01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;
```

⋮	Stack ↓
...	0x0911
	0x0910
	0x090f
	0x090e
	0x090d
	0x090c
	0x090b
⋮	

10

Wiederholung: Zeiger



Wiederholung: Zeiger



- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```
01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;
```

⋮	Stack ↓
...	0x0911
a 23	0x0910
	0x090f
	0x090e
	0x090d
	0x090c
	0x090b
⋮	

- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```
01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;
```

⋮	Stack ↓
...	0x0911
a 23	0x0910
b 42	0x090f
	0x090e
	0x090d
	0x090c
	0x090b
⋮	

Achtung: Die genaue Anordnung der Variablen auf dem Stack ist abhängig vom Übersetzer und den gewählten Optimierungen!

Achtung: Die genaue Anordnung der Variablen auf dem Stack ist abhängig vom Übersetzer und den gewählten Optimierungen!



- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```

01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;

```

	:	Stack ↓
	...	0x0911
a	23	0x0910
b	42	0x090f
p	0x0910	0x090e
p	0x0910	0x090d
		0x090c
		0x090b
	:	

Achtung: ATmega328PB hat 8-bit Register und 16-bit Adressen

10

- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```

01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;

```

	:	Stack ↓
	...	0x0911
a	66	0x0910
b	42	0x090f
p	0x0910	0x090e
p	0x0910	0x090d
		0x090c
		0x090b
	:	

Achtung: ATmega328PB hat 8-bit Register und 16-bit Adressen

10



- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```

01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;

```

	:	Stack ↓
	...	0x0911
a	66	0x0910
b	42	0x090f
p	0x090f	0x090e
p	0x090f	0x090d
		0x090c
		0x090b
	:	

Achtung: ATmega328PB hat 8-bit Register und 16-bit Adressen

10

- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```

01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;

```

	:	Stack ↓
	...	0x0911
a	66	0x0910
b	2	0x090f
p	0x090f	0x090e
p	0x090f	0x090d
		0x090c
		0x090b
	:	

Achtung: ATmega328PB hat 8-bit Register und 16-bit Adressen

10



- Variable: `uint8_t x`
- Zeiger: `uint8_t *y`
- Adressoperator: `&x`
- Verweisoperator: `*y`

```

01 uint8_t a = 23;
02 uint8_t b = 42;
03 uint8_t * p = &a;
04 *p = 66;
05 p = &b;
06 *p -= 40;
07 uint8_t c = *p;

```

	:	Stack ↓
	...	0x0911
a	66	0x0910
b	2	0x090f
p	0x090f	0x090e
p	0x090f	0x090d
c	2	0x090c
	:	0x090b

Achtung: ATmega328PB hat 8-bit Register und 16-bit Adressen

10

- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	:	Stack ↓
	...	0x090a
		0x0909
		0x0908
		0x0907
		0x0906
		0x0905
		0x0904
		0x0903
		0x0902
	:	

11



- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	:	Stack ↓
	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908
x[1]	4	0x0907
x[0]	2	0x0906
		0x0905
		0x0904
		0x0903
		0x0902
	:	

11

- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	:	Stack ↓
	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908
x[1]	4	0x0907
x[0]	2	0x0906
y	0x0906	0x0905
y	0x0906	0x0904
		0x0903
		0x0902
	:	

11



- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908
x[1]	4	0x0907
x[0]	2	0x0906
y	0x0906	0x0905
y	0x0906	0x0904
z	4	0x0903
		0x0902

11

- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908
x[1]	4	0x0907
x[0]	2	0x0906
y	0x0906	0x0905
y	0x0906	0x0904
z	2	0x0903
		0x0902

11



- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908
x[1]	4	0x0907
x[0]	2	0x0906
y	0x0908	0x0905
y	0x0908	0x0904
z	2	0x0903
		0x0902

11

- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

```

08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7];

```

	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908
x[1]	4	0x0907
x[0]	2	0x0906
y	0x0908	0x0905
y	0x0908	0x0904
z	8	0x0903
		0x0902

11



- Konstanter Zeiger: `uint8_t a[]`
- Variabler Zeiger: `uint8_t *b`
- Aktuelles Element: `*b`
- x-te Element: `b[x]`
- x-te Element: `*(b+x)`

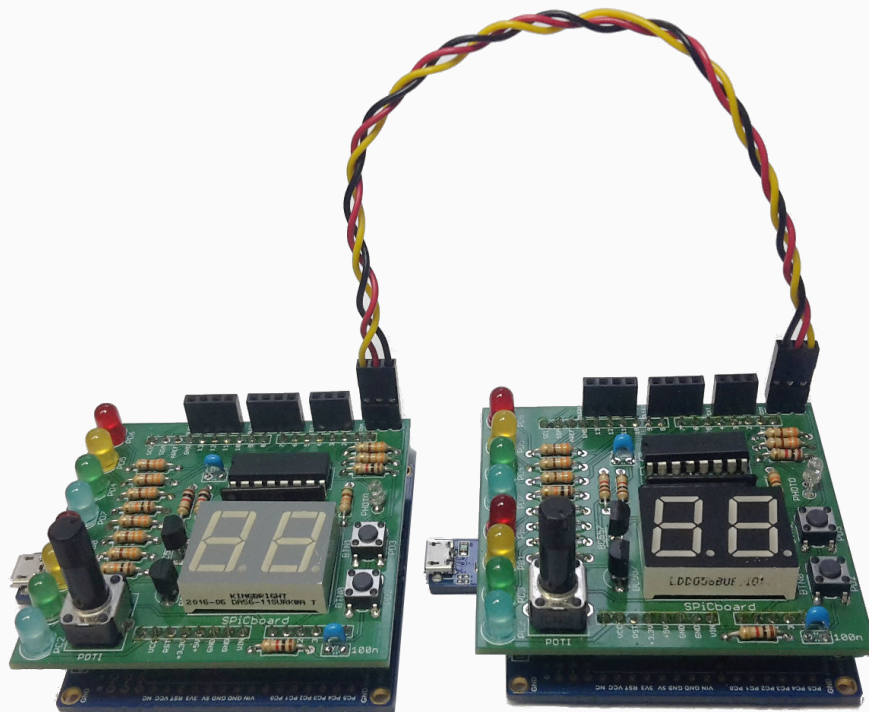
```
08 uint8_t x[] = {2,4,8,16};
09 uint8_t *y = x;
10 uint8_t z = x[1];
11 z = *y;
12 y = y+2;
13 z = *y;
14 z = x[7]; // ???
```

	:	Stack ↓
	...	0x090a
x[3]	16	0x0909
x[2]	8	0x0908 ←
x[1]	4	0x0907
x[0]	2	0x0906
y	0x0908	0x0905
y	0x0908	0x0904
z	???	0x0903
		0x0902
	:	

11

Kommunikation

SPiCboard verbinden



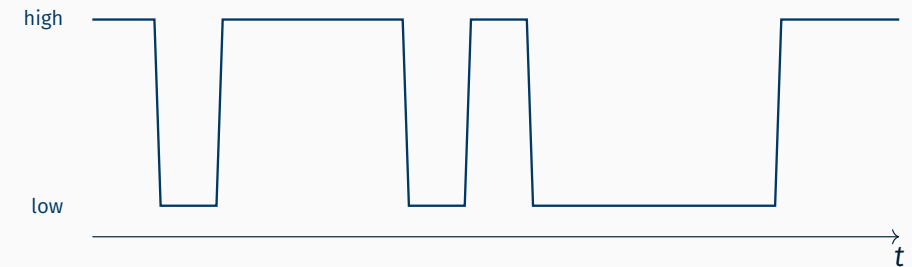
PD1 Ausgang (TX) wird mit RX des Kommunikationspartners verbunden

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

GND wird mit GND verbunden

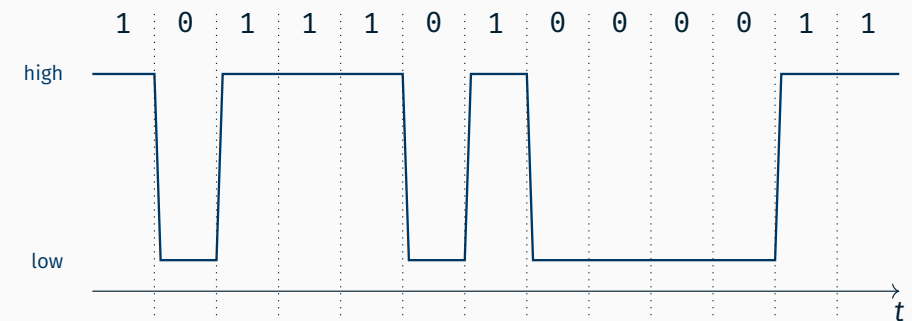
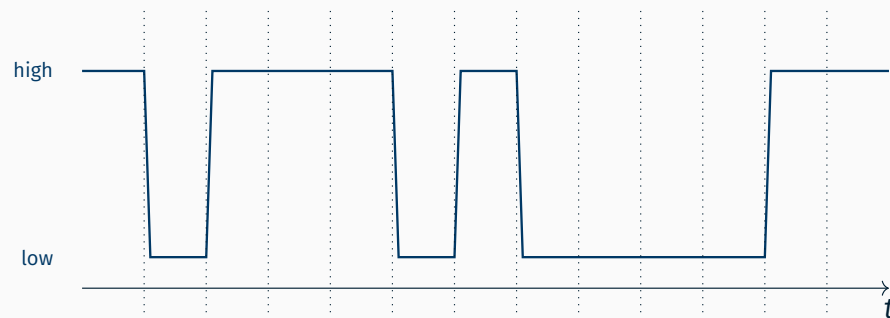


- gleiches Protokoll
 - wir nehmen 8-E-1
 - 8 Anzahl der Datenbits
 - E gerades (even) Paritätsbit
 - 1 Stopbit
 - sowie (implizit) ein Startbit
 - das Datenbit mit dem niedrigsten Stellenwert wird zuerst übertragen (aufsteigend)
 - hoher Pegel entspricht einer logischen 1, niedriger Pegel einer 0
-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



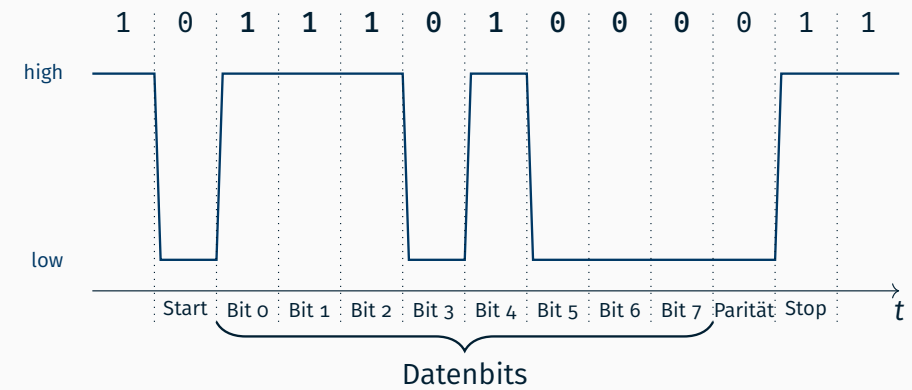
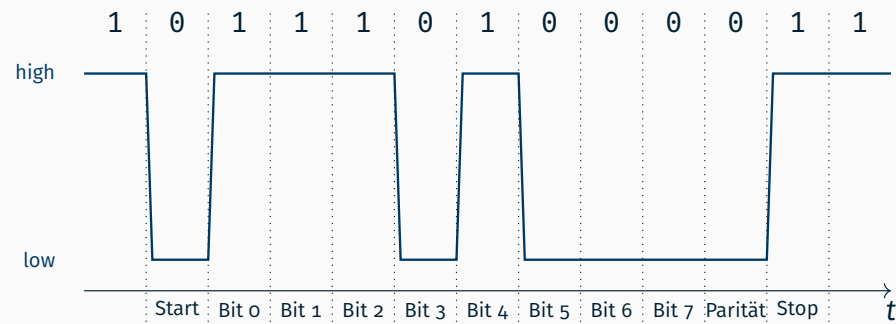
14

15



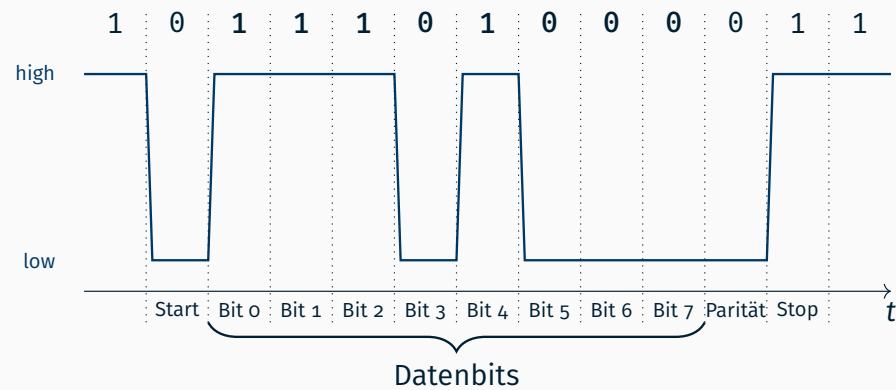
15

15

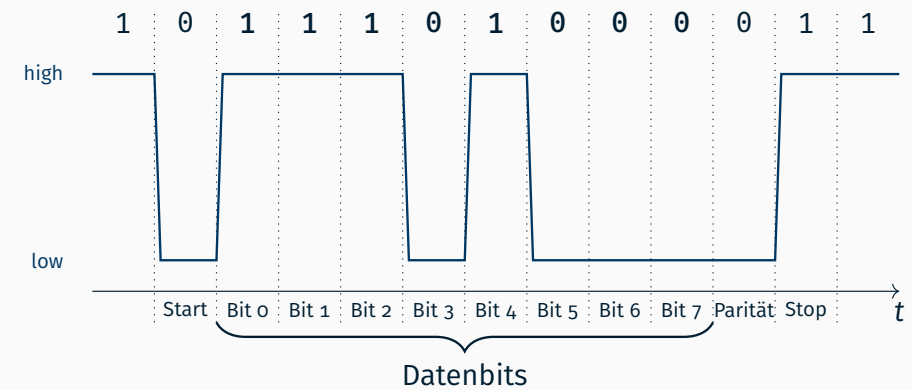


15

15



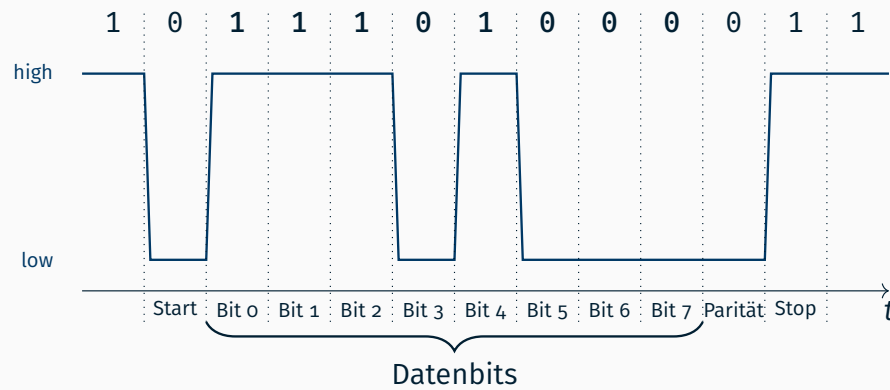
- Empfangenes Byte (rückwärts gelesen): $00010111_2 = 0 \times 17 = 23$



- Empfangenes Byte (rückwärts gelesen): $00010111_2 = 0 \times 17 = 23$
- Parität 0 (da Datenbits vier 1er $\Rightarrow$ gerade Anzahl)

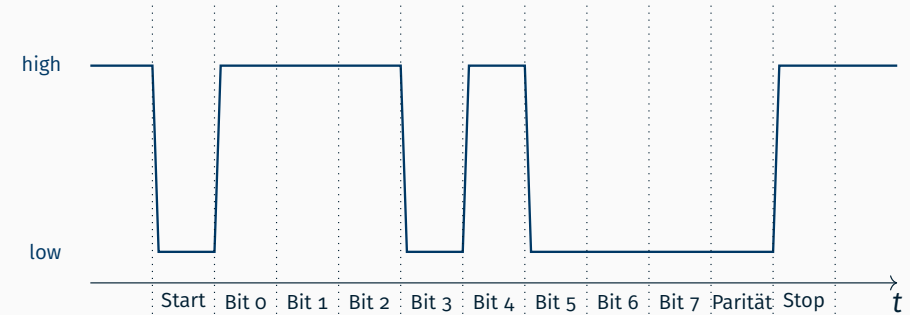
15

15



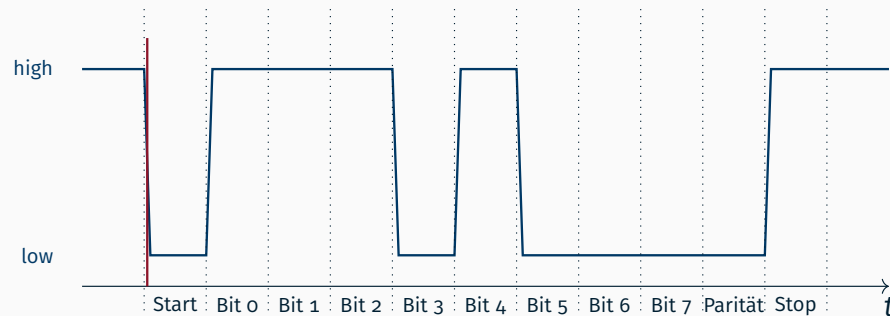
- Empfangenes Byte (rückwärts gelesen): $00010111_2 = 0 \times 17 = 23$
- Parität 0 (da Datenbits vier 1er $\Rightarrow$ gerade Anzahl)
- Startbit immer 0 und Stopbit immer 1

15

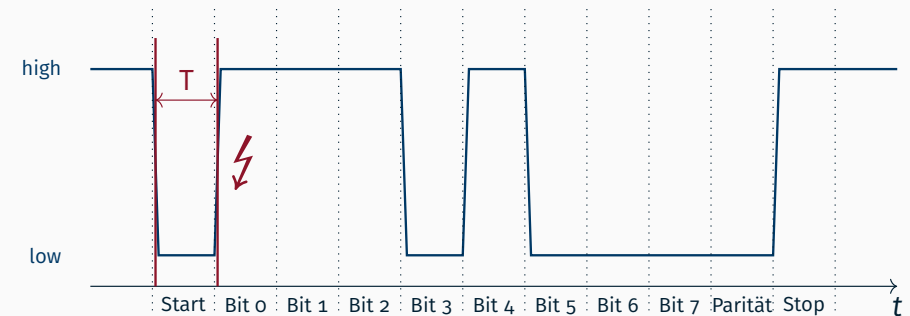


- Die Fenstergröße T ist uns bekannt

16



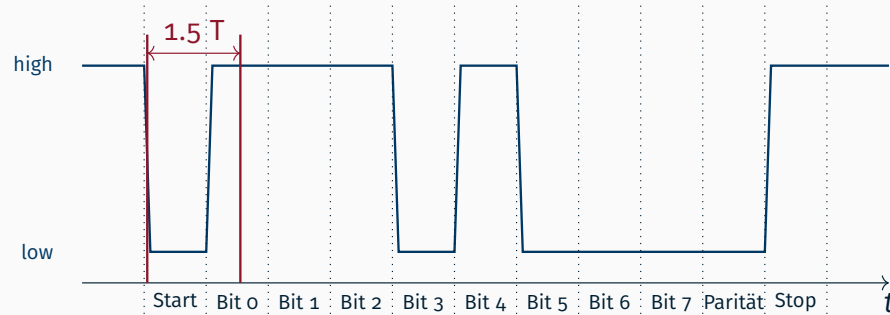
- Die Fenstergröße T ist uns bekannt
- Anfang ist einen Pegelwechsel von hoch auf niedrig



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

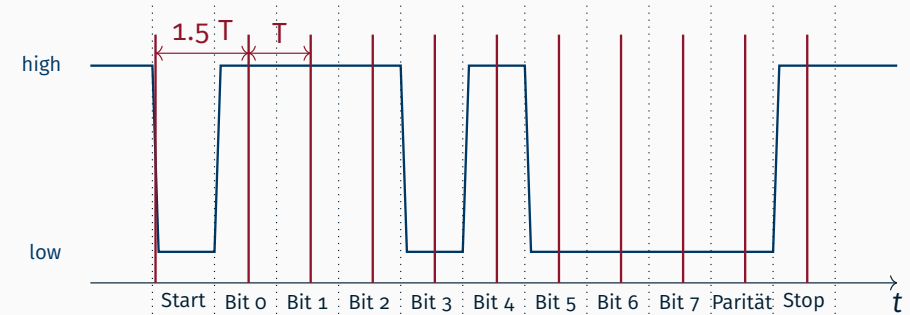
16

16



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

16



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

16

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_SPIC/SPiCboard/libapi.shtml

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

17



- 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();
```

18

- 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 }
```

19



- 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

20

- 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

20



- 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`

- 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.