Informatikwerkstatt, Foliensatz 3
PC-Kommunikation

G. Kemnitz

Institut für Informatik, TU Clausthal (IW3)

7. November 2016

Inhalt:

Interaktive Übungen:

1. Echoprogramm (echo)
2. Test mit Python-Scripten (python*).
3. Modularisierung und Modultest (com_pc)

Wiederholung

Wiederholungsaufgabe 3.1: PJ4 = PA0 ⊕ PA1

```
#include <avr/io.h>
int main(){
 DDRA =            // Init. als Eingänge
 DDRJ =            // Init. als Ausgänge
 ...      a;       // Variablenvereinbarungen
  while(...){      // Endlosschleife
  ...              // Lesen der Eingabe in a
                   // EXOR des gelesen mit dem
                   // nach rechtverschobenen gelesen
                   // Wert
                   // löschen der Bits 1 bis 7
                   // Ausgabe von Bit 0 auf LED4
}}
```

In der Endlosschleife soll an PJ4 die EXOR-Verknüpfung von PA0 PA1 ausgegeben werden.

Lösung

```
#include <avr/io.h>
int main(){
 DDRA =            //Init. als Eingänge
 DDRJ =            //Init. als Ausgänge
 uint8_t a;        //Variablenvereinbarungen
  while(1){        //
   a = PINA;       //Lesen der Eingabe in a
   a = (a>>1)^a;//EXOR des gelesen mit dem
                   //nach recht verschobenen Wert
   a = a & ~1;   //löschen der Bits 1 bis 7
   PORTJ = a<<4;//Ausgabe von Bit 0 auf LED4
  }
}
```

Wiederholungsaufgabe 3.2: Reengineering

Der Pseudo-Zufallsgenerator der letzten Aufgabe auf Foliensatz IW1 hat die Übergangsfunktion:

```
uint8_t z;
...
z =  (z>>1) ^ (z<<7) ^ ((z<<5)&0x80)
  ^ ((z<<4)&0x80) ^ ((z<<3)&0x80);
```

Welche Bitverknüpfungen der Istwerte werden den einzelnen Bits von z als Folgewerte zugewiesen?

$z_0 =$ $z_4 =$

$z_1 =$ $z_5 =$

$z_2 =$ $z_6 =$

$z_3 =$ $z_7 =$

Lösung

```
uint8_t z;
...
z =  (z>>1) ^ (z<<7) ^ ((z<<5)&0x80)
  ^ ((z<<4)&0x80) ^ ((z<<3)&0x80);
```

$$
\begin{aligned}
z_0 &= z_1 & \qquad z_4 &= z_5 \\
z_1 &= z_2 & \qquad z_5 &= z_6 \\
z_2 &= z_3 & \qquad z_6 &= z_7 \\
z_3 &= z_4 & \qquad z_7 &= z_0 \oplus z_2 \oplus z_3 \oplus z_4
\end{aligned}
$$

Wiederholungsaufgabe 3.3: Variablen

1. Was ist der Unterschied zwischen globalen und lokalen Variablen und wo im Programm werden sie vereinbart?
2. Was ist ein Zeiger?
3. Wie wird ein Zeiger »t« auf ein »uint16_t« vereinbart?
4. Vervollständigen Sie:

```
uint8_t z, *x;
...
x =         ;//Zuweisung der Adresse von z
z =         ;//Zuweisung des Wertes, auf den
             //x zeigt
```

Lösung

1. Globale Variablen werden außerhalb von Funktionen vereinbart, haben eine feste Adresse, die sie während der gesamten Laufzeit behalten. Lokale Variablen werden innerhalb eines in »{...}« eingeschlossenen Blocks vereinbart, existieren nur bis zum Verlassen des Blocks und erhalten eine Adresse auf dem Stack.
2. Ein Zeiger ist eine Variable für eine Adresse.
3. Zeiger »t« auf Variablen vom Typ »uint16_t«:

```
uint16_t *t;
```

4. Vervollständigtes Programm:

```
uint8_t z, *x;
...
x =  &z; //Zuweisung der Adresse von z
z =  *x; //Zuweisung des Wertes, auf den
         //x zeigt
```

Wiederholungsaufgabe 3.4: Zeiger & Felder

Der Inhalt von Feld »a« soll in Feld »b« kopiert werden.

```
//Vereinbarung der Felder a und b
                                        ;
//Zeigern pa und pb auf a[0] bzw. b[0]
                                        ;
uint8_t idx; //Schleifenzähler
int main(){
 // Wiederhole 5 mal
 for(                                ){
  //Inhalt von pa nach pb kopieren
  //und beide Zeiger erhöhen
                                        ;
                                        ;
 }
}
```

Var.	Adr.	AW
a[0]	200	'T'(54)
a[1]	201	'e'(65)
a[2]	202	'x'(78)
a[3]	203	't'(74)
a[4]	204	0
b[0]	205	0
b[1]	206	0
b[2]	207	0
b[3]	208	0
b[4]	209	0
b[5]	20A	0
pa.1	20B	2
pa.0	20C	0
pb.1	20D	2
pb.0	20E	5
idx	20F	0

Lösung und Fortsetzung

```
uint8_t a[]="Text", b[5];
uint8_t *pa=a, *pb=b;//oder *pa=&a[0],
uint8_t idx;
int main(){
 for(idx=0;idx<5;idx++){
  *pb = *pa; pa++; pb++;
 }
}
```

Var.	Adr.	AW
a[0]	200	'T'(54)
a[1]	201	'e'(65)
a[2]	202	'x'(78)
a[3]	203	't'(74)
a[4]	204	0
b[0]	205	0
b[1]	206	0
b[2]	207	0
b[3]	208	0
b[4]	209	0
b[5]	20A	0
pa.1	20B	2
pa.0	20C	0
pb.1	20D	2
pb.0	20E	5
idx	20F	0

Ändern der Schleife so, dass bis »(*pa)==0« wiederholt wird:

```
int main(){
 while(                          ){
  *pb = *pa; pa++; pb++;
 }
}
```

Lösung zur Fortsetzung

```
1  uint8_t a[]="Text", b[5];
2  uint8_t *pa=a, *pb=b;
3  uint8_t idx;
4  int main(){
5   while((*pa)!=0){
6     *pb = *pa; pa++; pb++;
7   }
8  }
```

- Werden mit der Programmlösung alle Zeichen aus Feld »a« in Feld »b« kopiert?

Var.	Adr.	AW
a[0]	200	'T'(54)
a[1]	201	'e'(65)
a[2]	202	'x'(78)
a[3]	203	't'(74)
a[4]	204	0
b[0]	205	0
b[1]	206	0
b[2]	207	0
b[3]	208	0
b[4]	209	0
b[5]	20A	0
pa.1	20B	2
pa.0	20C	0
pb.1	20D	2
pb.0	20E	5
idx	20F	0

Nein, die Schleife bricht vor dem Kopieren des abschließenden Leerzeichen ab. Um es auch zu kopieren, nach Zeile 7 einfügen:

```
*pb = *pa;
```

Wiederholungsaufgabe 3.5: Typecast

1. Was ist ein Typcast?
2. Welches Risko entsteht, wenn einer Variablen »a« vom Typ »int16_t« der Wert einer Variablen »b« vom Typ »uint32_t« zugewiesen wird.
3. Wie wird eine solche Zuweisung mit einem Typcast explizit erlaubt?

Lösung

1. Ein Typcast ist eine Typumwandlung (andere Bitanzahl, andere Wertedarstellung).
2. Wenn einer Variablen »a« vom Typ »int16_t« der Wert einer Variablen »b« vom Typ »uint32_t« zugewiesen wird, dann werden alle mit »b« darstellbaren und mit »a« nicht darstellbaren Wert (2^{15} bis $2^{32} - 1$) verfälscht.
3. Zuweisung explizit erlauben:

```
a = (int16_t)b; //Zuweisung mit Typecast
```

Wiederholungsaufgabe 3.6: Modultest

1. Wie wurden in Foliensatz IW2 Testbeispiele und Testsätze beschrieben?
2. Vereinbaren Sie einen Verbund »tb« für die Beschreibung eines Testbeispiels mit zwei »uint8_t« Eingabewerten »a« und »b« und einem »int16_t« Ausgabewert »y« sowie einen Testsatz als Feld für die Testbeispiele in der nachfolgenden Tabelle:

Testbeispiel	a	b	c
0	34	17	-2654
1	2	200	10025
2	78	4	-7546

3. Welche Aufgaben hat ein Testrahmen und was für einen Programmablauf hatten die bisher besprochenen Testrahmen?

Lösung

1. Testbeispiele wurden als Verbund von Eingabe- und Sollausgabewerten und Testsätze als initialisierte Felder von Testbeispielen beschrieben.
2. Verbund für die Beschreibung eines Testbeispiels mit zwei »uint8_t« Eingabewerten und einem »int16_t« Ausgabewert:

```
struc tb {uint8_t a,b; int16_t y;}
```

   Initalisiertes Feld mit den vorgegebenen Testbeispielen:

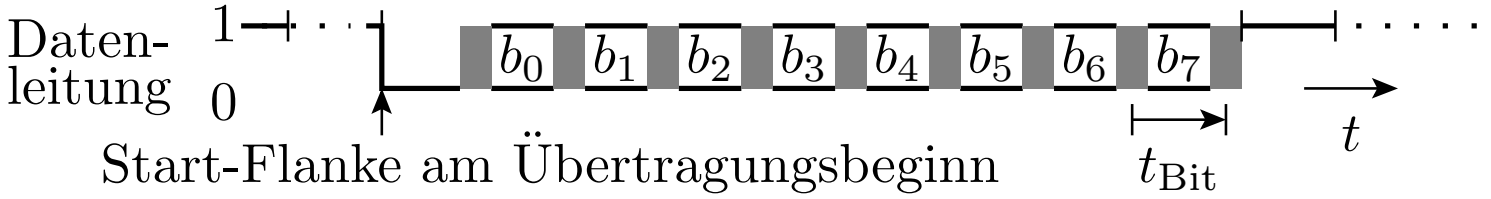
```
struc tb Testsatz[]={{ 34, 17,-2654},
        {2,200,10025},{ 78,  4,-7546}};
```

3. Der Testrahmen ist ein Hauptprogramm, das das Testobjekt in einer Schleife »wiederhole für alle Testbeispiele« abarbeitet, die Ergebnisse kontrolliert, Fehler zählt, ...

PC-Kommunikation

Kommunikationsprotokoll

Der Datenaustausch zwischen Rechnern erfolgt seriell[1] über USB, Ethernet, CAN-Bus, ... Unsere PC-Kommunikation nutzt USART2, Kommunikationsprotokoll[2] 8N1[3], 9600 Baud:



$b_i \in \{0,1\}$ Datenbits — Stoppbit (Wert 1)

$\cdots$ Übertragungspause

Byteübertragung: $10 \cdot t_{\text{Bit}}$

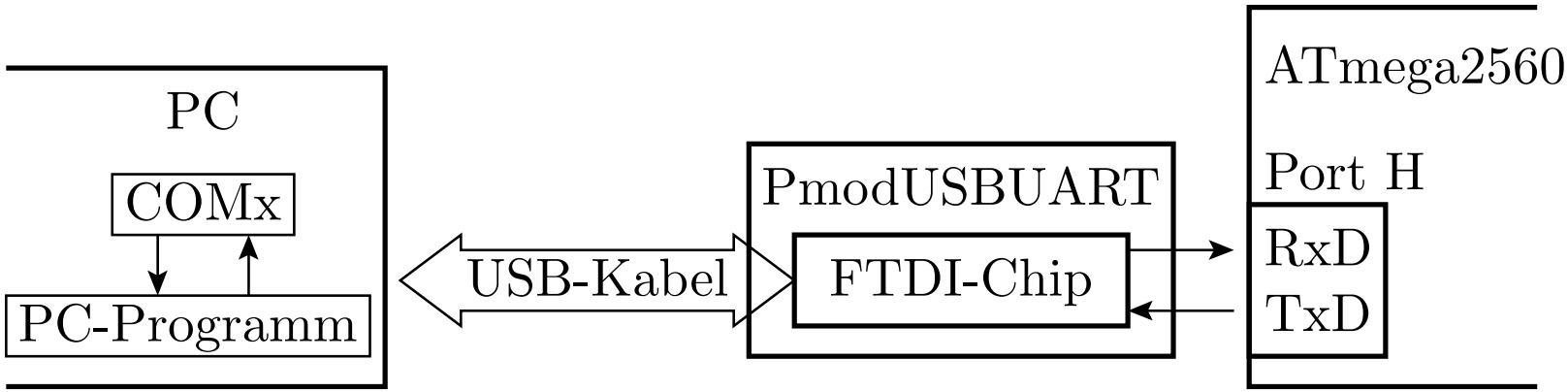
- 8N1: 8 Daten, kein Paritäts- und ein Stoppbit
- Bitzeit 1/9600s. (bis ca. 1000 Datenbytes pro s).

[1] Seriell: Hintereinander über eine, statt parallel über viele Leitungen.

[2] Kommunikationsprotokoll: Vereinbarung, nach der der Datenaustausch zwischen zwei oder mehr Teilnehmern erfolgt.

[3] Format 8N1: 8 Datenbits, kein Paritätsbit und 1 Stoppbit.

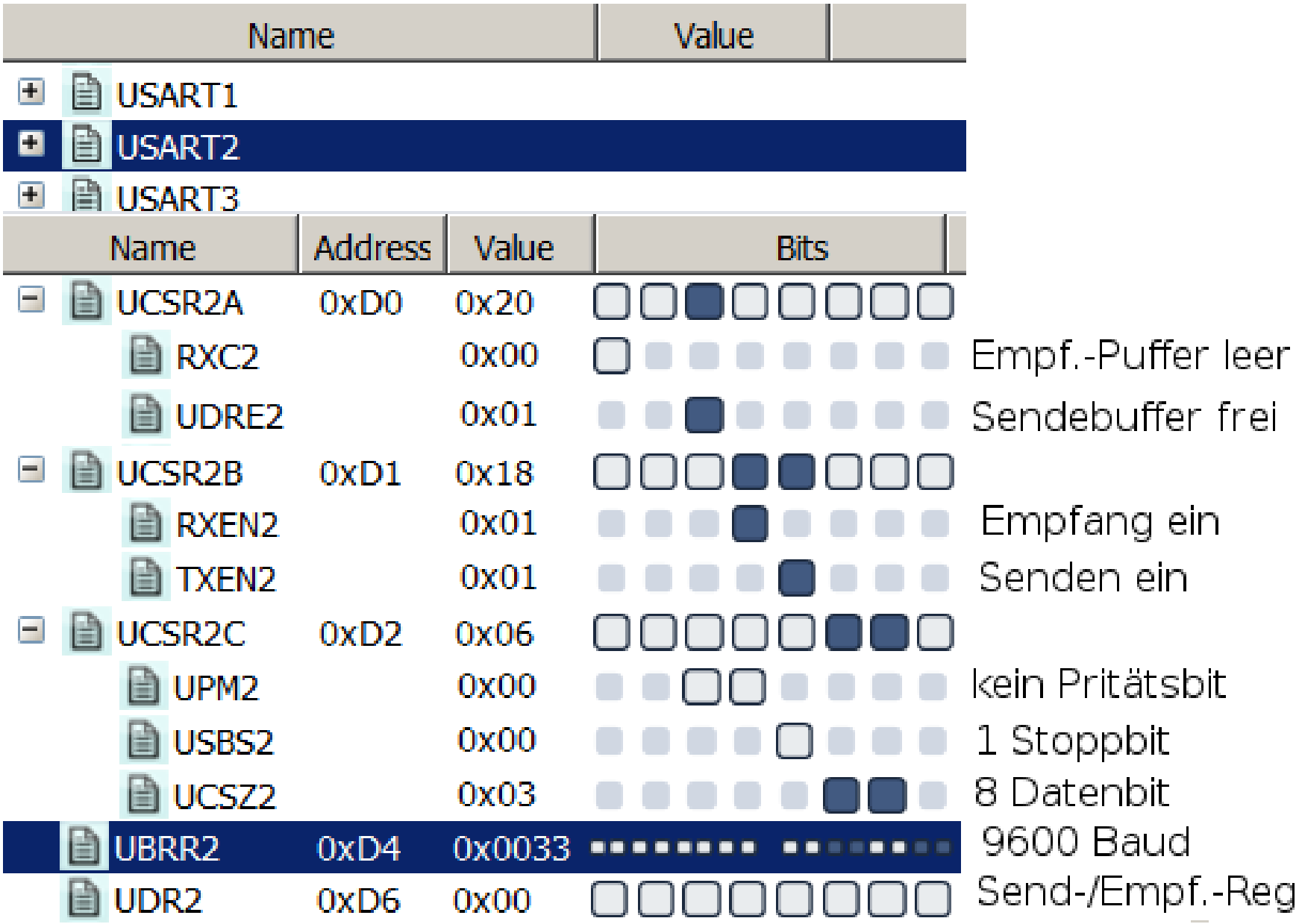
Kommunikationsfluss



- Der Mikrorechner kann zeitgleich je ein Bytes zum FTDI-Chip senden und vom FTDI-Chip empfangen.
- Der FTDI-Chip tauscht über USB Daten mit dem PC aus.
- Auf dem PC präsentiert der Treiber für den FTDI-Chip die Datenverbindung zum Mikrorechner als COM-Port.
- Jeder einmal über USB verbundene FTDI-Chip bekommt auf dem PC eine eigene COM-Port-Nummer.
- Das PC-Programm wird entweder HTerm oder ein selbst zu schreibendes Python-Programm sein.

Byte-Empfang und Senden im Mikrorechner

```
int main(){
  // Variablenvereinbarung und Initialisierung
  <USART2 Protokoll 8N1, 9600 Baud>
  <Sender und Empfänger ein>
  while(1){ // Endlosschleife
    ...
    // Byteempfang
    <Warte, bis Byte da ist>
    <Lesen und Verarbeiten des Bytes>
    ...
    // Byte versenden
    <Warte, bis Sendepuffer frei>
    <Byte in Sendepuffer schreiben>
    ...
  }
}
```


Name
Value
USART1
USART2
USART3
Name
Address
Value
Bits
UCSR2A 0xD0 0x20
RXC2 0x00
Empf.-Puffer leer
UDRE2 0x01
Sendebuffer frei
UCSR2B 0xD1 0x18
RXEN2 0x01
Empfang ein
TXEN2 0x01
Senden ein
UCSR2C 0xD2 0x06
UPM2 0x00
kein Pritätsbit
USBS2 0x00
1 Stoppbit
UCSZ2 0x03
8 Datenbit
UBRR2 0xD4 0x0033
9600 Baud
UDR2 0xD6 0x00
Send-/Empf.-Reg

Echoprogramm

Das Echoprogramm für nachfolgende Experimente

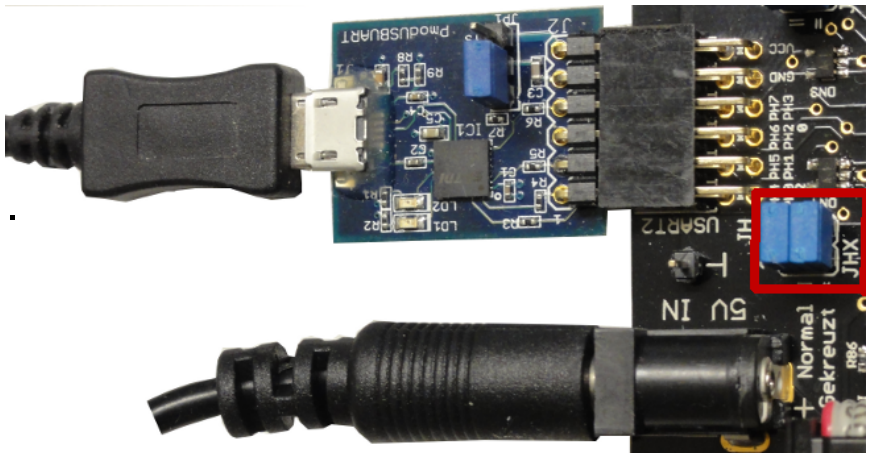
```
#include <avr/io.h>                    //Projekt: F3-echo
uint8_t daten;                         //Datei: echo.c
int main(void){
 UCSR2C=0b110;                         //Format 8N1
 UBRR2=51;                             //9600 Baud
 UCSR2B=(1<<RXEN2)|(1<<TXEN2);         //Empf. + Sender ein
 DDRJ = 0xFF;                          //LEDs als Ausgänge
 PORTJ= 0;                             //LED-Ausgabe 0x00
 while(1){
  while(!(UCSR2A &(1<<RXC2)));         //warte auf Byte
  daten = UDR2;                        //Byte übernehmen
  while(!(UCSR2A&(1<<UDRE2)));         //warte Puffer frei
  UDR2 = daten;                        //Byte übergeben
  PORTJ++;                             //LED-Ausgabe erhöhen
 }
}
```

Zurücksenden und zählen aller empfangenen Bytes.

Test des Echo-Programms

Hardware vorbereiten:

- Spannung ausschalten.
- PModUSBUSART Kontrolle, Jumper wie im Bild, und und an JH oben stecken.
- Jumper JHX »gekreuzt (=)«.
- PModUSBUSART mit PC verbinden. Spannung einschalten.

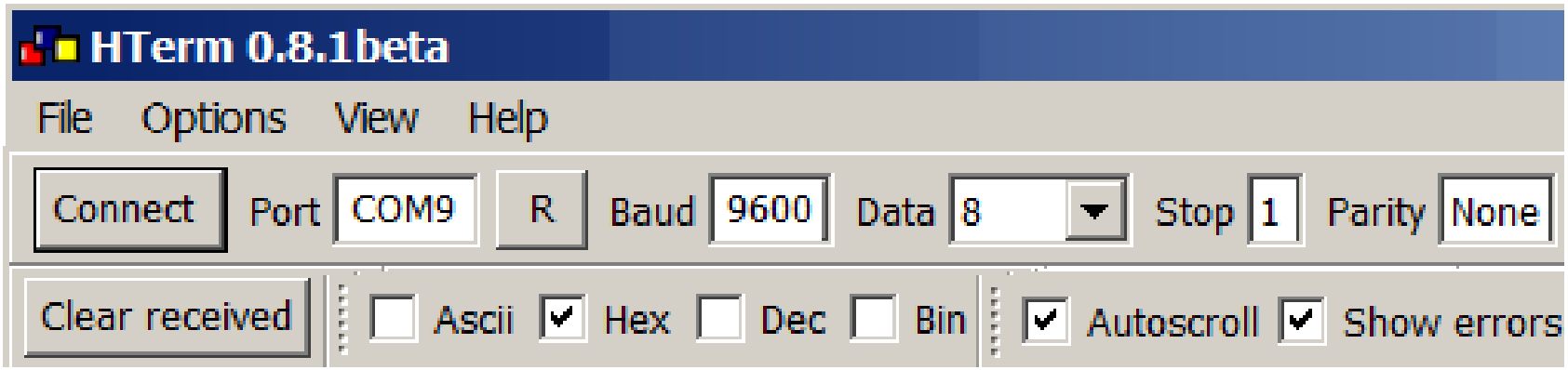


Software vorbereiten:

- Projekt Echo öffnen.
- »Dragon« und Compileroptimierung »-O0« auswählen.
- Übersetzen und im Debugger starten.

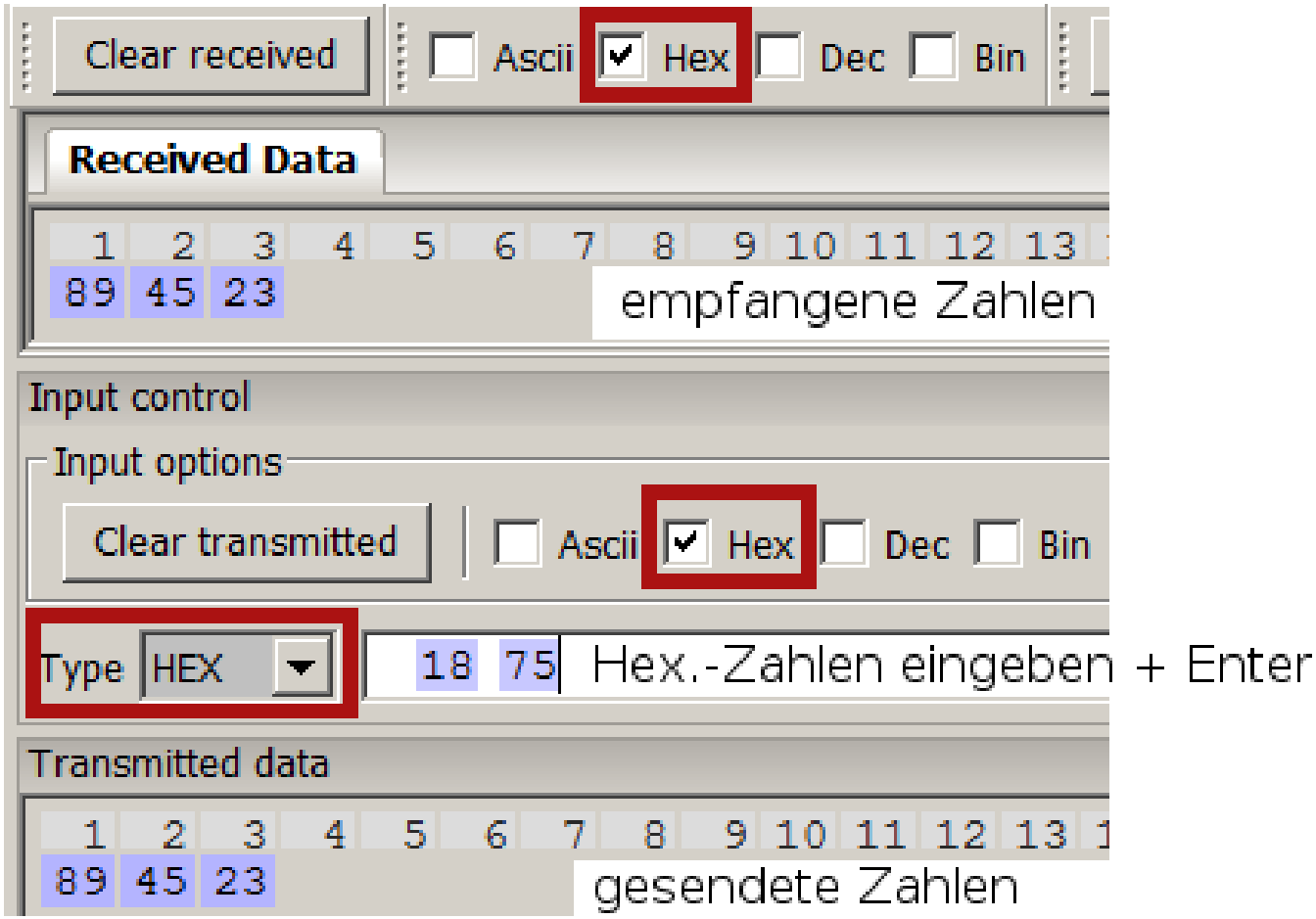
Verbindung mit HTerm herstellen

- Auf dem PC HTerm starten.
- COM-Port auswählen[4].
- 9600 Baud, 8 Daten, 1 Stopp- und kein Paritätsbit einstellen.
- Verbindung herstellen (Connect).



[4] Die COM-Schnittstelle, die nach Anstecken des USB-Kabels vom PmodUSBURT und »R« (Refresh Comport List) als neuer Port erscheint.

Für die Eingabe »HEX« auswählen. Für die Darstellung der Sende- und Empfangsdaten nur bei »Hex« $\surd$ setzen.



- Alle versendeten Zahlen werden zurückgesendet.

Textdarstellung

Zeichendarstellung im ASCII-Code

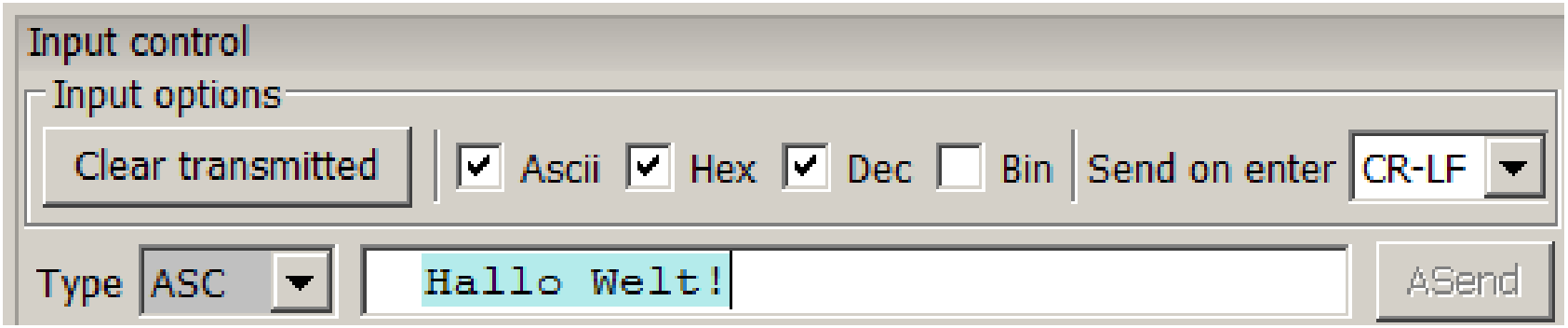
Buchstaben, Ziffern und andere Zeichen werden als Bytes und Texte als Felder von Bytes dargestellt. ASCII-Code:

hex	bin	dez	ASCII	hex	bin	dez	ASCII
0x0a	0b000 1010	10	LF	0x41	0b100 0001	65	A
0x0d	0b000 1101	13	CR	⋮	⋮	⋮	⋮
0x20	0b010 0000	32	␣	0x50	0b101 0000	80	P
0x21	0b010 0001	33	!	⋮	⋮	⋮	⋮
0x2E	0b010 1110	46	.	0x5A	0b101 1010	90	Z
0x30	0b011 0000	48	0	0x61	0b110 0001	97	a
0x31	0b011 0001	49	1	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	0x70	0b111 0000	112	p
0x39	0b011 1001	57	9	⋮	⋮	⋮	⋮
0x3F	0b011 1111	63	?	0x7A	0b111 1010	122	z

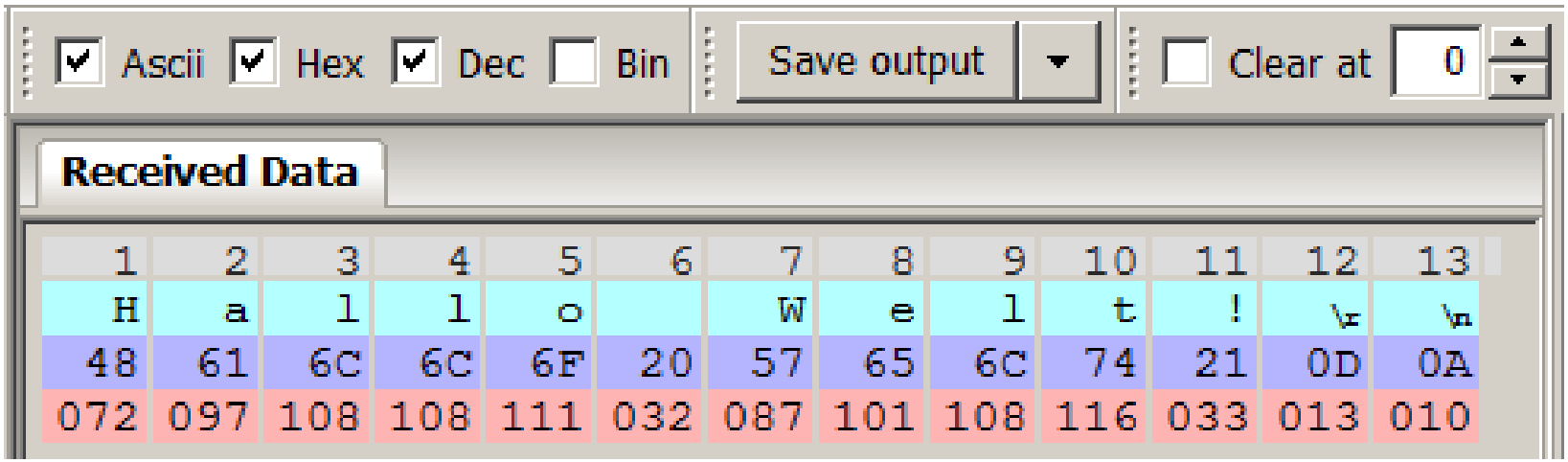
LF – Zeilenvorschub; CR – Wagenrücklauf; ␣ – Leerzeichen

Senden und Empfang von Texten

Das HTerm kann ASCII-Zeichenketten + CR+LF senden:



Empfangene Daten als Zeichen- und Zahlenfolge:



1	2	3	4	5	6	7	8	9	10	11	12	13
H	a	l	l	o		W	e	l	t	!	\r	\n
48	61	6C	6C	6F	20	57	65	6C	74	21	0D	0A
072	097	108	108	111	032	087	101	108	116	033	013	010

Kontrollieren Sie auch, dass sich bei jedem Senden der LED-Ausgabewert an LED1 bis LED8 um die Anzahl der gesendeten Zeichen erhöht.

Test mit Python

Python als Programmiersprache für Tests

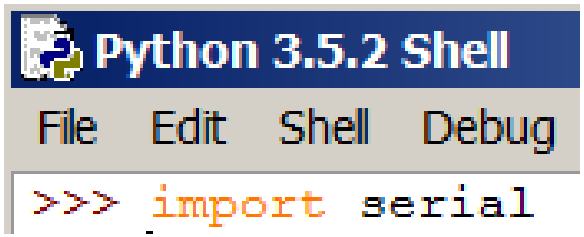
Python: Interpretersprache vorzugsweise zum Experimentieren und für Tests, einfach zu erlernen, gut zu debuggen, ... Installation für zu Hause und Kurzeinführung siehe später Zusatzteil.

Test des Echoprogramms mit Python:

- Verbindung mit HTerm schließen (Disconnect)
- Start der Programmierkonsole »Idle« von Start:

```
> All Programs > Python 3.5 > IDLE (Python 3.5 ...)
```

- Import des Moduls für die serielle Kommunikation:



- Kommunikationsverbindung öffnen[5]:

```
ser = serial.Serial("COM9")
```

- Zeichenfolge an Variable zuweisen:

```
send = "Hallo Welt!"
```

- Anschauen von Typ, Wert und Länge der Zeichenkette:

`>>> type(send)` `<type 'str'>`	Funktion zur Bestimmung des Datentyps. Der Datentyp ist "Zeichenkette (string)".
`>>> send` `'Hallo Welt!'`	Bei Eingabe des Variablennamens wird der Wert angezeigt.
`>>> len(send)` `11`	Die Funktion "len()" liefert die Anzahl der Elemente der Zeichenkette.

[5] Denselben COM-Port wie im HTerm benutzen. Das Programm P3\Python\list_com_ports.py listet alle COM-Ports, die sich öffnen lassen. 8N1, 9600 Baud ist der Standardwert und muss deshalb nicht eingestellt werden.

- Zeichenkette senden:

```
ser.write(send.encode("ascii"))
```

- Auf 11 Zeichen warten und diese lesen:

```
y = ser.read(11)
```

- Anzeige von Typ, Wert und Länge der empfangenen Daten mit print(<Zeichenkette>):

```
print("type(y)__________:_" + str(type(y)))
a = y.decode()
print(a, type(a)); print(y, type(y))
print("Empfangene_Daten:_" + a)
print("len(a)___________:_%i" %len(a))
```

a + b	Verkettung der Zeichenketten a und b.
str(x)	Konvertierung von »x« in eine Textdarstellung.
"... %i"%(<w>)	Formatierte Ausgabe des Werte »w«.

Zusammenfassen zum Programm »scom.py«

```
import serial
ser = serial.Serial("COM9")
send = "Hallo Welt!".encode("ascii")
ser.write(send)
y = ser.read(len(send))
print("Empfangene Daten: " + y.decode())
ser.close()
```

- encode/decode: Umwandlung Zeichenkette ⇔ Byte-Vektor.
- Zeilen eines Blocks **müssen** gleiche Einrücktiefe haben!

Programmdatei in der »Idle« öffnen:

```
File > Open > H:~\Informatikwerkstatt\P03\Python\scom.py
```

Programmstart mit Ausgabe auf der »Idle«:

```
Run > Run Module (F5)
```

Start auf der Konsole

- Konsole öffnen (Eingabe von »cmd + Enter« im Suchfeld über »Start«):

- Wechsel in das Programmverzeichnis. Eingabe Programmname + Enter:

```
C:\Windows\system32\cmd.exe
C:\>H:
H:\>cd Informatikwerkstatt\Python
H:\Informatikwerkstatt\Python>scom.py
Empfangene Daten: Hallo Welt!
```

- Die Programmausgabe »Empfangene Daten: Hallo Welt!« erfolgt auf der Konsole.

Messung der Übertragungsdauer mit »scom_t.py«

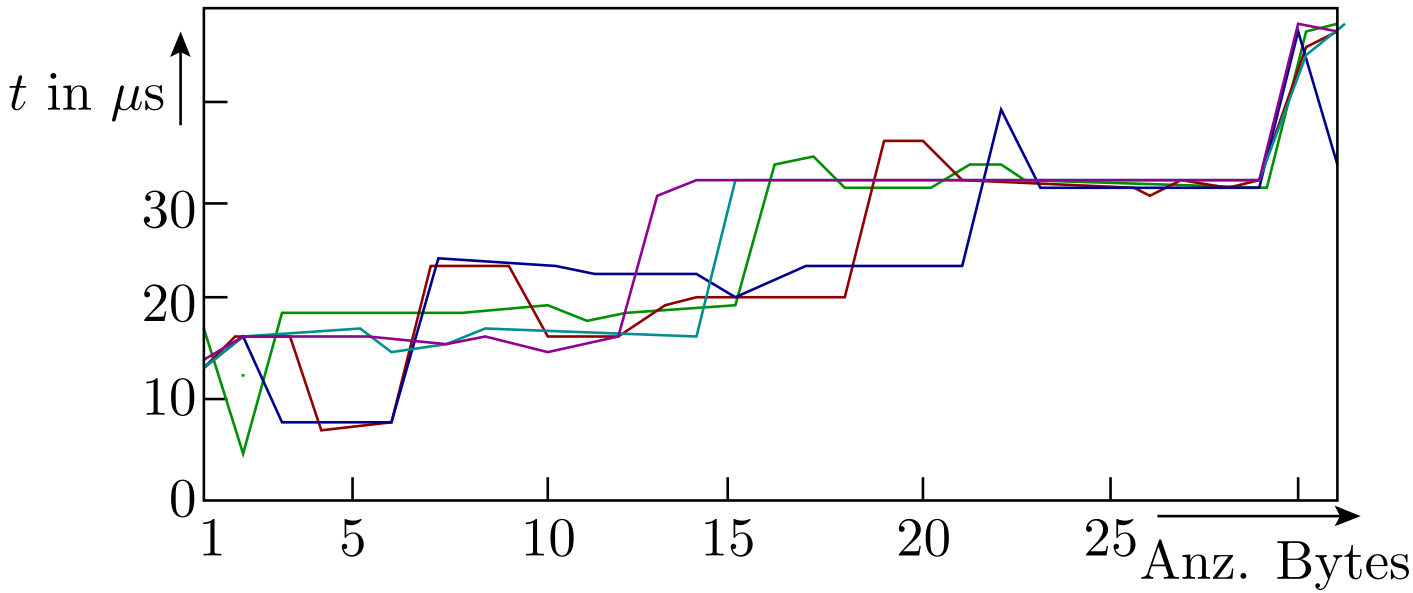
```
import serial
# Funktion clock() aus Modul "time"
from time import clock
ser = serial.Serial("COM9")
send = "Hallo Welt!".encode("ascii")
ts = clock();      #Startzeit in Sekunden nach ...
ser.write(send)
y = ser.read(len(send)).decode()
dt = clock()-ts    #Zeitdifferenz zur Startzeit
print("Empfangene Daten:"+y.decode()
            +" dt= "+str(dt*1E3)+" ms")
ser.close()
```

Programmstart und Ausgabe auf der Konsole:

```
H:\Informatikwerkstatt\Progr_IW\Python>scom_t.py
Empfangene Daten: Hallo Welt! dt= 13.6730654185 ms
```

Übertragungszeit und Paketgröße

Die Übertragung wird über USB und später auch über Bluetooth getunnelt und erfolgt in Paketen aus mehreren Bytes. Dauer abhängig von der Paketgröße und nicht deterministisch.



Das nachfolgende Programm scom_txy.py bestimmt für Bytefolgen der Länge 1 bis 31 die Übertragungsdauer und scom_txy5.py wiederholt das 5-mal.

Zeitmessung für Paketgröße 1 bis 30 »scom_txy.py«

```
import serial
from time import clock
ser = serial.Serial("COM9", timeout=1)
send = "Das ist ein sehr langer String!".encode("ascii")
l = 1; dt_list=[];    #leere Liste fuer dt-Werte
while l<=len(send):   #Wiederhole bis Gesamtlaenge
  ts = clock();       #neuer Block => einruecken
  ser.write(send[:l])#Sende die ersten l Zeichen
  y = ser.read(l).decode()#Warte auf l Zeichen
  dt = clock()-ts     #Zeitdifferenz zur Startzeit
  dt_list.append(dt) #Differenzzeiten an Liste haengen
  print("Empf. Daten: "+y.decode(), "dt= "
                      +str(dt*1000)+" ms")
  l = l+1;            #Ende des Schleifenkoerpers
ser.close()           #danach Einruecktiefe ruecksetzen
```

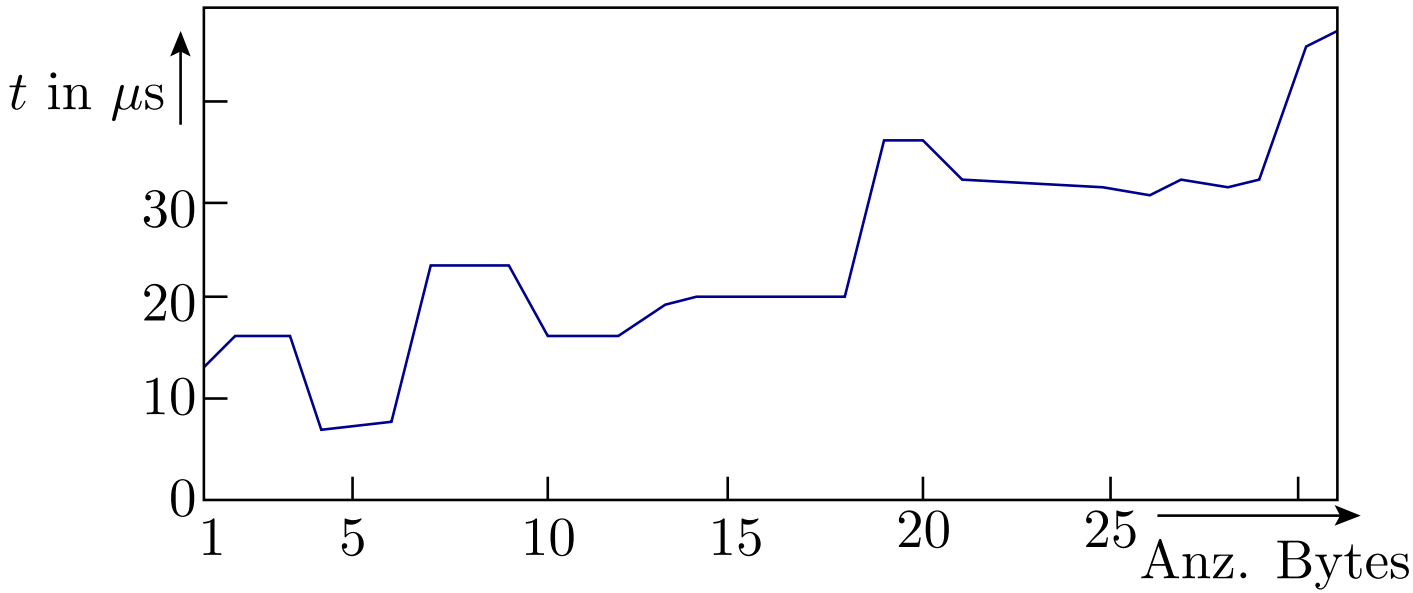
- Anweisungen für graphische Ausgabe siehe übernächste Folie.

Test von »scom_txy.py« auf der Konsole

```
H:\Informatikwerkstatt\Progr_IW\Python>scom_txy.py
Empfangene Daten: D dt= 13.2549278489 ms
Empfangene Daten: Da dt= 14.921187855 ms
Empfangene Daten: Das dt= 14.40439233 16 ms
Empfangene Daten: Das  dt= 14.4265738178 ms
Empfangene Daten: Das i dt= 14.3878389836 ms
Empfangene Daten: Das is dt= 14.2295889771 ms
Empfangene Daten: Das ist dt= 14.368306033 ms
Empfangene Daten: Das ist  dt= 14.3656574973 ms
Empfangene Daten: Das ist e dt= 14.2249540396 ms
Empfangene Daten: Das ist ei dt= 14.1339106258 ms
Empfangene Daten: Das ist ein dt= 14.1051078004 ms
Empfangene Daten: Das ist ein  dt= 14.4951046784 ms
Empfangene Daten: Das ist ein s dt= 30.4876252137 ms
Empfangene Daten: Das ist ein se dt= 29.8907114861 ms
Empfangene Daten: Das ist ein seh dt= 30.3879740589 ms
Empfangene Daten: Das ist ein sehr dt= 30.0079091897 ms
Empfangene Daten: Das ist ein sehr  dt= 30.3273888054 ms
Empfangene Daten: Das ist ein sehr l dt= 30.1208030228 ms
Empfangene Daten: Das ist ein sehr la dt= 30.1522543839 ms
Empfangene Daten: Das ist ein sehr lan dt= 30.1446398438 ms
Empfangene Daten: Das ist ein sehr lang dt= 30.2389939272 ms
Empfangene Daten: Das ist ein sehr lange dt= 30.3604955013 ms
```

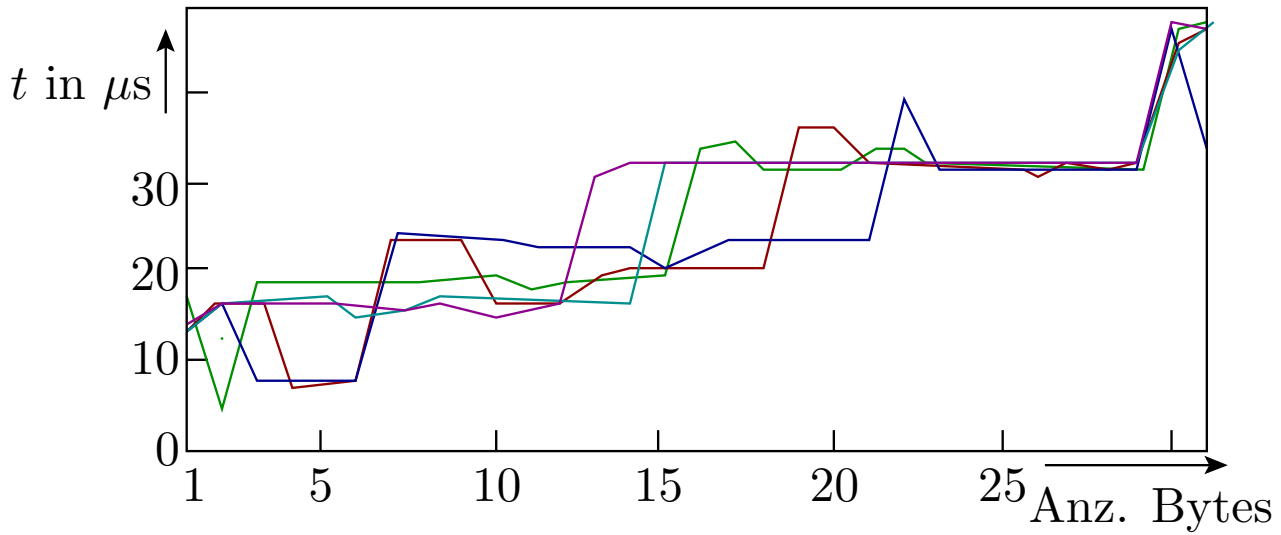
Graphische Ausgabe am Ende von »scom_txy.py«

```
# Import der Klasse "pyplot"
import matplotlib.pyplot as plt
plt.plot(range(1, len(dt_list+1)), dt_list)
plt.xlabel("Anzahl der Bytes") #plot erzeugen
plt.ylabel("dt in ms")         #Achsenbeschrift.
plt.show()                     #anzeigen
```



5-fache Wiederholung (Programm: scom_txy5.py)

```
...; plt.hold(True)
for idx in range(5):
 <Bestimme Übertragungsdauer für 1 bis 31 Byte>
 plt.plot(range(1,len(dt_list)+1), dt_list)
ser.close()
plt.show(); ...
```

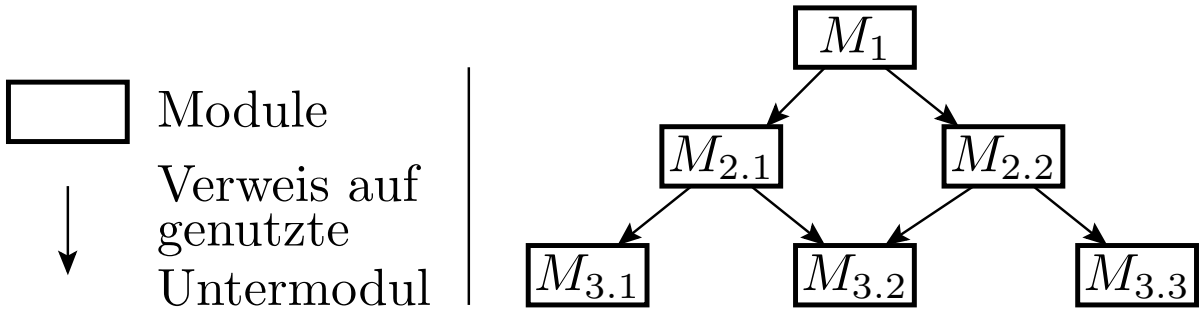


- Bei 9600 Bitzeiten pro s, 1 Startbit + 8 Datenbits + 1 Stoppbit dauert eine Byteübertragung $\geq$1 ms.
- Für 21 bis 28 Byte große Pakete werden etwa 30 ms benötigt, d.h. fast max. Durchsatz.
- Auf anderen Rechnern, zeitgleichen Übertragungen über USB, Nutzung von Bluetooth, ... kann die Übertragung auch so lange dauern, dass es stört.
- Falls ihre Zielanwendung später wegen zu großen Übertragungszeiten nicht funktioniert, kann das Übertragungsverhalten in der dargestellten Weise untersucht und so zielgerichtet nach alternativen Lösungen gesucht werden.

Modultest vom PC aus

Modularisierung

Software wird modulweise entwickelt und in Betrieb genommen:



In unserer Veranstaltung

- ist das oberste Modul »main()« (Hauptprogramm) und
- sind die genutzten Bausteine Unterprogramme.

Unterprogramme mit gemeinsamen privaten Datenobjekten (Variablen, EA-Register) werden je in einer eigenen c-Datei zusammengefasst. Aufrufschnittstellen für die von anderen Modulen nutzbaren Funktionen stehen in den Header-Dateien.

Testrahmen auf dem PC

Da Mikrorechner nicht über Tastaturen, Bildschirme und Dateisysteme verfügen, bietet sich der Test über eine serielle Schnittstelle an:

- PC-Programm, das an den Mikrorechner Testeingaben schickt und zurückgesendete Ergebnisse kontrolliert und
- ein Mikrorechnerhauptprogramm, das
  - auf Daten vom PC wartet,
  - die zu testende Funktion ausführt und
  - Berechnungsergebnisse zum PC sendet.

Die Grundfunktionen hierfür (USART initialisieren, Byte empfangen und Byte senden)

- sind im Echo-Programm enthalten und
- sollen als nachnutzbare Bausteine in eine eigene Datei ausgelagert werden.

Modularisierung des Echoprogramms

Aufteilung des Echoprogramms »echo.c« von Folie 23 in nachnutzbare Module und ein Hauptprogramm für den Test der Module:

```
int main(void){
// ---------- Initialisierung ----------------------
 UCSR2C=0b110;                        // Format 8N1
 UBRR2=51;                            // 9600 Baud
 UCSR2B=(1<<RXEN2)|(1<<TXEN2);// Empf. + Sender ein
// -------------------------------------------------
 while(1){
// ---------- Empfangen eines Bytes ---------------
  while (!(UCSR2A & (1<<RXC2)));//warte auf Byte
  daten = UDR2;                       //Byte übernehmen
// ---------- Versenden eines Bytes ---------------
  while (!(UCSR2A &(1<<UDRE2)));//warte Puffer frei
  UDR2 = daten;                       //Byte übergeben
 } // ------------------------------------------------
}
```

Funktionen für die PC-Kommunikation (com_pc.c)

```
#include <avr/io.h>
//Initialisierung von USART2 (8N1, 9600 Baud)
void com_pc_init(){
 UCSR2C=0b110;                    //Format 8N1
 UBRR2=51;                        //9600 Baud
 UCSR2B=(1<<RXEN2)|(1<<TXEN2); //E+S ein
}
//ein Byte empfangen
uint8_t getByte(){
 while (!(UCSR2A & (1<<RXC2)));//warte auf ein Byte
 return UDR2;                     //Byte zurueckgeben
}
//ein Byte versenden
void sendByte(uint8_t dat){
 while (!(UCSR2A & (1<<UDRE2)));//warte Puffer frei
 UDR2 = dat;                       //Byte uebernehmen
}
```

Header »com_pc.h« für den Export

```
#ifndef COM_PC_H_
#define COM_PC_H_

#include <avr/io.h>

 void com_pc_init();              //Init. USART2
 uint8_t getByte();               //Byte empfangen
 void sendByte(uint8_t dat); //Byte versenden

#endif /* COM_PC_H_ */
```

- Enthält die drei Funktionsdefinitionen ohne Anweisungen.
- #...: Precompiler-Anweisungen. Ausführung (Textverarbeitung) vor dem Compilieren.
- #ifndef ... #define ... #endif verhindern, dass Definitionen mehrfach in den zu übersetzenden Quelltext übernommen werden.

Testrahmen (Hauptprogramm)

```
#include <avr/io.h>    // Anmerkung *1
#include "com_pc.h"
uint8_t d;
int main(void){
 com_pc_init();        //Init. USART2
 while(1){             //Endlosschleife
  d = getByte();       //Byte empfangen
  sendByte(d);         //Byte zurücksenden
 }
}
```

*1: überflüssig, da »avr/io.h« in »com_pc.h« eingefügt wird, falls es vorher noch nicht eingefügt wurde.

Ein Testobjekt

Testobjekt sei folgende Berechnungssequenz:

```
uint8_t a, b, s, d, q, r;
uint16_t p;
...
s = a + b;      // Summe
d = a - b;      // Differenz
p = a * b;      // Produkt
q = a/b;        // ganzzahliger Quotient
r = a%b;        // Divisionsrest
```

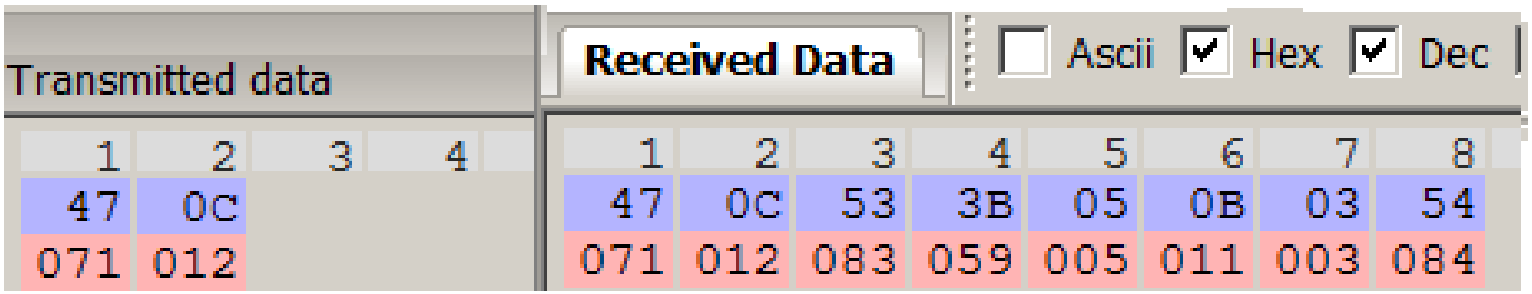
Darum soll ein Rahmenprogramm gelegt werden, das

- in einer Endlosschleife
- vom PC auf zwei Bytes für a und b wartet,
- die zu testenden Anweisungen ausführt und
- 8 Bytes (s, d, 2×p, q und r) zurückschickt.

```
#include <avr/io.h>          //test_com_pc.c
#include "com_pc.h"
uint8_t a, b, s, d, q, r; uint16_t p;
int main(){
 com_pc_init();
 while (1){
  a = getByte();  b = getByte();
  //-- zu testende Anweisungen -----
  s = a + b;      //Summe
  d = a - b;      //Differenz
  p = a * b;      //Produkt
  q = a/b;        //ganzzahliger Quotient
  r = a%b;        //Divisionsrest
  //--------------------------------
  sendByte(a);    sendByte(b);
  sendByte(s);    sendByte(d);
  sendByte(q);    sendByte(r);
  sendByte(p>>8); sendByte(p&0xFF);
 }
}
```

Test mit dem HTerm

- Projekt »F3-com_pc« öffnen.
- »Dragon« und Compiler-Optimierung -O0 auswählen.
- Übersetzen. Debugger starten. Programm freilaufend starten.
- HTerm öffnen. 9600 Baud, 8 Datenbit, 1 Stoppbit.
- COM-Port des angesteckten »PmodUSBUART«. »Connect«.
- 2 Byte senden und 8 Bytes empfangen.



a	b	$a+b$	$a-b$	a/b	$a \cdot b$
71	12	83	59	5 Rest 11	$3 \cdot 2^8 + 84 = 852$

Python-Programm für den Test vom PC

```
import serial                          #Programm: test_com_pc.py
ser = serial.Serial("COM9")#COM anpassen!
# Testbeispiele
i_tuple = ((27,87),(220,20),(110,4), (218, 219))
for inp in i_tuple:                    #fuer alle Testbeispiele
 ser.write(bytearray(inp)) #als Byte-Array versenden
 x = ser.read(8)                       #auf 8 Bytes warten
 a = x[0]; b = x[1]; s = x[2]   #Zusammensetzen der
 d = x[3]; q = x[4]; r = (x[5] #empfangen Bytes zu
 p = x[6] * 256 + x[7]                 #Datenobjekten
 print('a=%3i , b=%3i Summe: %i %s'%(a,b,s,str(s==a+b)))
 print(13*' '+'Differenz:%i %s'%(d, str(d==a-b)))
 print(13*' '+'Quotient: %i Rest: %i (%s)'%(q, r,
          ... str(a==q*b+r)))
 print(13*' '+'Produkt:  %i (%s)'%(p, str(p==a*b)))
ser.close()
```

(Details der Python-Programmierung siehe später Zusatzteil.)

Testdurchführung

- HTerm »Disconnect«.
- Auf dem Mikrorechner das Programm »test_com_pc« starten.
- Windows-Konsole (cmd.exe) starten. Im Verzeichnis H:\Informatikwerkstatt\Python das Programm test_com_pc.py starten. Programmausgabe kontrollieren:

```
H:\Informatikwerkstatt\Progr_IW\Python>test_com_pc.py
a= 27 b= 87 Summe: 114 True
            Differenz: 196 False
            Quotion: 0 Rest: 27 True
            Produkt: 2349 True
a=220 b= 20 Summe: 240 True
            Differenz: 200 True
            Quotion: 11 Rest: 0 True
            Produkt: 4400 True
a=110 b=  4 Summe: 114 True
            Differenz: 106 True
            Quotion: 27 Rest: 2 True
            Produkt: 440 True
a=218 b=219 Summe: 181 False
            Differenz: 255 False
            Quotion: 0 Rest: 218 True
            Produkt: 47742 True
```

Aufgaben

Aufgabe 3.1: Textdarstellung und -ausgabe

Durch welche Zahlenfolge wird der nachfolgende Text dargestellt:

"Informatikwerkstatt , Uebung3"

- Lösen Sie die Aufgabe mit der ASCII-Tabelle auf Folie 28.
- Kontrollieren Sie das Ergebnis, in dem Sie die Zeichenkette mit dem HTerm versenden und zusätzlich als ASCII-Folge anzeigen lassen.
- Kontrollieren Sie das Ergebnis mit folgendem Programm:

```
#include "com_pc.h"
uint8_t text[] = "Informatikwerkstatt ...";
int main(){
  com_pc_init();
  uint8_t i;
  for (i=0;i<28; i++) sendByte(text[i]);
}
```

Aufgabe 3.2: Wiederhole bis zum »Nullbyte«

Wenn C wie in der nachfolgenden Programmzeile

```
uint8_t text[] = "Informatikwerkstatt, Uebung3";
```

eine Zeichenkette initialisiert, hängt es ein Byte mit dem Wert null an.

- Kontrollieren Sie das mit dem Debugger.
- Schreiben Sie das Programm aus der vorherigen Aufgabe so um, dass es nicht genau 28 Zeichen ausgibt, sondern alle Zeichen bis vor dem Zeichenwert null.

  Hinweis: Man nutzt hierfür eine Schleife, »wiederhole, solange ein Zeiger »ptr« nicht auf den Wert null zeigt:

  ```
  while (*ptr!=0){<Anweisungsfolge>}
  ```

Aufgabe 3.3: Textausgabe

Ergänzen Sie in der Funktionssammlung »com_pc.c« eine Funktion zur Textausgabe mit der Aufrufschnittstelle:

```
void sendString(uint8_t *strg);
```

Als Testbeispiel soll das nachfolgende Hauptprogramm:

```
#include <avr/io.h>
#include "com_pc.h"
int main(){
  com_pc_init();
  sendString("Das ist ein Text.");
}
```

den Text »Das ist ein Text.« an den PC schicken.

Aufgabe 3.4: Zeitmessung Warteschleife

- Schreiben Sie ein c-Programm, dass auf ein Byte vom PC wartet, die folgende Warteschleife

```
uint32_t ct;
...
for (ct=0; ct<500000; ct++);
```

  abarbeitet und das Byte zurücksendet.
- Testen Sie das Programm mit HTerm und schätzen Sie die Zeit vom Versenden bis zum Empfang. (Man kann in HTerm empfangene Daten mit Zeitstempel aufzeichnen.)
- Schreiben Sie ein Python-Programm, das zehn mal die Zeit vom Versenden bis zum Empfang bestimmt und die Einzelwerte und den Mittelwert der gemessenen Zeiten ausgibt.

Aufgabe 3.5: Modultest vom PC aus

- Schreiben Sie ein Programm, das in einer Endlosschleife immer auf zwei Bytes wartet, diese nach der Vorschrift

```
int16_t wert = (int16_t)(b1<<8|b2);
```

  (b1, b2 – erstes bzw. zweites empfangenes Byte) zu einer 16-Bit vorzeichenbehafteten Zahl zusammenfasst, diese negiert und verdoppelt und das Ergebnis zurücksendet.
- Testen Sie das Programm mit der Eingabe 0x45A im HTerm.
- Programmieren Sie in Python einen Testrahmen, der zehn zufällig zu wählende Testbeispiele abarbeitet und für jedes Testbeispiel den Eingabewert, den Ausgabewert sowie den logischen Vergleichswert von Ist- und Sollausgabe ausgibt.

Aufgabe 3.6: Test eine 2-Byte-Multiplikation

- Schreiben Sie ein Programm, das zwei 2-Byte vorzeichenbehaftete Faktoren empfängt, multipliziert und ein 4-Byte-Produkt zurücksendet.
- Schreiben Sie ein Python-Programm, dass dieses Programm mit zehn zufällig ausgewählten Beispielen testet.

Kontrollieren Sie für alle Testbeispielen, bei denen Soll- und Istwert abweichen:

- ob der Fehler im Python-Programm steckt,
- die im Mikrorechner ankommenden Daten falsch sind,
- der Mikrorechner falsch rechnet oder
- die zurückgesendeten Byte-Werte von Python falsch interpretiert werden.

Hinweise für die Testdurchführung siehe nächste Folie.

Hinweise zur Fehlersuche

Der Test von Mikrorechnerprogrammen mit serieller Kommunikation im Schrittbetrieb ist dahingehend problematisch, dass

- Mikrorechner- und PC-Programm beim Anhalten Daten verlieren können und
- nach Fortsetzung auf die verlorenen Daten warten.

Vorschlag zur Problemvermeidung

- In beiden Programmen nach jedem versendeten und jedem empfangen Paket einen Unterbrechungspunkt setzen.
- Unterbrechnungspunkte in Python setzt man mit (siehe Zusatzteil Folie 70):

```
raw_input('Enter zur Programmfortsetzung:')
```

- Nach jedem Halt zuerst das empfangende und dann das sendende Programm starten.

Aufgabe 3.7: Modultest Vorzeichenzahlen

Ändern Sie das Mikrorechnerprogramm Folie 51 und das Python-Programm Folie 54 so, dass vorzeichenbehaftete Zahlen addiert, subtrahiert, dividiert und multipliziert werden.

Testbeispiele für das Python-Programm:

```
i_tuple = ((-23,45), (-89,-7), (0x7F, -17),
           (-58, -99))
```

Zusatzteil (Python)

Pythoninstallation für die Übungen[6]

Die Übung nutzt Python3 unter Windows:

- Von »www.python.org« für die neuste »stable« Version (aktuall 3.5.2) »Windows x86-64 executable installer« herunterladen und als Administrator ausführen.
- Haken bei »Add Python 3.5 to Path«.
- Customize installation, Next, Haken bei »Install for all users«.

Nachinstallation der in den Übungen benötigten Zusatzpakete:

- Windows-Taste +"R" > "cmd".
- rechte Maustaste, "Run as Administrator".

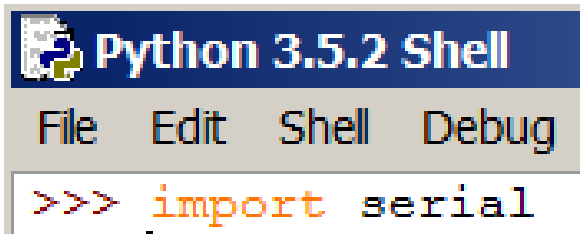
```
pip install pyserial
pip install matplotlib
```

[6] Im Labor sollte bereits alles richtig installiert sein.

Ausprobieren von Programmanweisungen

Python eignet sich u.a. deshalb gut zum Programmierenlernen und für Test-Scripte, weil sich Programmzeilen einzeln auf der Konsole testen lassen. Start der Programmierkonsole »Idle« von Start:

```
> All Programs > Python 3.5 > IDLE (Python 3.5 ...)
```



Eine der wichtigsten Funktion zum Probieren:

```
print(<Obj>{,Obj})
```

Sie gibt eine kommaseparierte Liste der Textdarstellungen aller aufgezählten Objekte aus.

Zum Ausprobieren auf der »idle«:

```
>>>s = 'Hallo'
>>>print(3, 5==7, s) #Zahl, Wahrheitswert, Text
(3, False, 'Hallo')
>>>print(type(s), len(s))
(<type 'str'>, 5)
```

Textverarbeitung

Beim Testen sind gut lesbare Textdarstellungen wichtig. :

- unformatierte Textkonvertierung: str(<OBj>)
- formatierte Textkonvertierung:
  '<Formatstring>'%(<Wertetupel>)
- Verketten von Texten: Text1 + Text 2

Zum Ausprobieren auf der Konsole:

```
>>>a=5; b=37.7; s='Hallo'
>>>print(str(a) + str(b) + s)
537.7Hallo
>>>print('a=%2i, b=%4.2f: %s'%(a,b,s))
a= 5, b=37.70: Hallo
```

("%s" – Text (string); "%ni" – ganze Zahl (int), Darstellung mit mindestens n Zeichen; "%$n.m$f" – Gleitkommazahl (float), Darstellung mit mindestens n Zeichen und m Nachkommastellen).

Interaktive Eingaben

In Testscripten werden Eingaben und Sollausgaben vorzugsweise als Konstanten (Tupel oder Listen siehe nächste Folie) vereinbart oder aus Dateien geladen[7]. Die nachfolgende Eingabeanweisung braucht man vor allem, um Python-Scripte auf dem PC anzuhalten:

```
e = raw_input('<Ausgabetext>')
```

Sie gibt einen Text aus, wartet auf eine Eingabe + <Enter> und hat als Rückgabewert die eingegebene Zeichenkette ohne »Enter«:

```
>>> e = raw_input('Warte auf Eingabe: ')
Warte auf Eingabe: text
>>> print('Eingabetext: ' + e)
Eingabetext: text
```

[7] Grund: Tests werden mehrfach wiederholt und das mehrfache Eingeben derselben Werte ist lästig und fehleranfällig.

Sequenzobjekte: Zeichenkette, Tupel

Zeichenkette: Sequenz von Zeichen bzw. Bytes. Auswahl von Elementen und Teilzeichenketten:

```
>>>s='Hallo Welt!'
>>>print('s[3]="%s", s[4:9]="%s"'%(s[3],s[4:9]))
s[3]="l", s[4:9]="o Wel"
```

Tupel: Sequenz beliebiger Objekte.

```
>>> t = ('abc', 3, 4.2)
>>> print(t, t[1], t[1:2])
(('abc', 3, 4.2), 3, (3,))
```

Sequenzobjekt: Liste

Liste: Sequenz beliebiger Objekte mit zusätzlichen Funktionen zum Einfügen, Löschen und Sortieren von Elementen:

```
>>>l = []; print(l) # leere Liste
[]
>>>l.append('E1'); print(l) # 1. Element anfügen
['E1']
>>>l.append(3==4); print(l) # 2. Element anfügen
['E1', False]
```

Listen von Zahlenfolgen werden mit range erzeugt:

```
>>>print(range(5))
[0, 1, 2, 3, 4]
>>>print(range(-4,2,1))
[-4, -3, -2, -1, 0, 1]
```

Kontrollstrukturen, Fallunterschiedungen

Python unterstützt alle gebräuchlichen Kontrollstrukturen: Fallunterscheidungen, Schleifen, Unterprogramme, ... und eine Spezialstruktur für die Fehlerbehandlung:

```
if <B1>:
 <Anweisungsfolge, wenn B1 erfüllt ist>
elif <B2>:
 <Anweisungsfolge, wenn B1 nicht und B2 erfüllt>
else:
 <Anweisungsfolge, wenn weder B1 noch B2 erfüllt>
<immer auszuführende Anweisungen>
```

- Bedingungen »B1« und »B2« müssen den Typ <type 'bool'> haben. Bildung durch Vergleich, z.B. »a==b«.
- Nach »:« Einrücktiefe erhöhen und bis Blockende beibehalten.
- Hinter der letzten bedingt auszuführenden Anweisung Einrücktief zurücksetzen.

Wiederholschleifen

Wiederholschleifen iterieren über Sequenzobjekte (Zeichenketten, Tupel, Listen):

```
>>>s='Hallo'
>>>for c in s: # für alle Zeichen in s
... print(c)   # Einrücktiefe erhöhen
...            # Einrücktiefe zurücksetzen
H
a
l
l
o
```

Auch hier nach »:« (für den Schleifenkörper) Einrücktiefe erhöhen und nach der Schleife zurücksetzen.

Zusätzliche Fallunterscheidung im Schleifenkörper:

```
>>>i=0;
>>>for c in s:
... if c=='l': # Einrücktiefe erhöhen
...  print(i)  # Einrücktiefe erhöhen
    i=i+       # Einrücktiefe zurücksetzen
...            # Einrücktiefe zurücksetzen
2
3
```

Programme besser als Dateien eingeben und Testen. In der »Idle«:

```
File > New File (Ctrl+N)
```

Programmeingabe:

```
for c in s:
 if c='l': # Einrücktiefe erhöhen
  print(i) # Einrücktiefe erhöhen
 i=i+      # Einrücktiefe zurücksetzen
```

Programm Speichern

```
File > Save (Ctrl+S), <Pfad+Dateinamen>
```

und starten

```
Run > Run Modul (F5).
```

Unterprogramme

```
def <Funktionsname>(<Liste  Aufrufparameter>):
 <Anweisungen> # Einrücktiefe erhöhen
 [return <Rückgabewert>]
                # Einrücktiefe zurücksetzen
```

Beispiel:

```
# Dateiname: add.py
def add(a, b):
 return a+b

if __name__=='__main__':   # nur bei Ausführung
 print(add(5, 7))          # nicht bei Import
```

Ausführen der Datei »add.py:

```
>>> add.py
12
```

Import, Fehlerbehandlungen

Mit »import« lassen sich Funktionen aus anderen Dateien (Modulen) nutzbar machen. Die Bedingung »__name__'=='__main__'« für die Abarbeitung der Testbeispielausge ist dann »False«:

```
>>> import add
>>> print(add.add(3,5))
8
```

Wenn sich ein Programm nicht bei einem Ausführungsfehler beenden soll:

```
try:
 <auszuprobierende Anweisungen>
except <abzufangende Fehlertypen>:
 <Anweisungen im Fehlerfall>
```

Ermittlung verfügbarer COM-Ports unter Windows

```
def get_COMs():
 import serial
 port_list = []
 for pnr in range(3):
  port = 'COM%s'%(pnr+1)
  try:
   s = serial.Serial(port)
   s.close()
   port_list.append(port)
  except (OSError,serial.SerialException):
   pass
 return port_list

if __name__=='__main__': # Testrahmen
 print(get_COMs())
```

Wie könnte man zusätzlich feststellen, an welchem COM-Port der programmierte Mikrorechner angeschlossen ist?