



Informatikwerkstatt, Foliensatz 5

Test mit Python

G. Kemnitz

Institut für Informatik, TU Clausthal (IW-F5)
19. November 2020



Inhalt:

Wiederholung
Schnellkurs Python
Echo-Test mit Python

Modultest vom PC aus
Modultest mit Python
Aufgaben

Interaktive Übungen:

- 1 Test mit Python-Skripten (python*).
- 2 Modularisierung und Modultest (com_pc)



Wiederholung



Vorzeichenbehaftete Zahlen

Statt durch Vorzeichen und Betrag Darstellung durch
»Stellenkomplement +1«. Mathematische Grundlage:

- Das Stellenkomplement zu einer Ziffer b_i ist die Differenz zur größten darstellbaren Ziffer mit dem Wert $B - 1$:

$$\bar{b}_i = B - 1 - b_i$$

(B – Basis des Zahlensystems, für Dezimalzahlen $B = 10$).

Beispiel: $\overline{437} = 562$

- Zahl plus Stellenkomplement gleich größte darstellbare Zahl.

Beispiel: $437 + \overline{437} = 437 + 562 = 999$

- plus Eins gleich kleinste nicht darstellbare Zahl:

$$Z + \bar{Z} + 1 = B^n$$

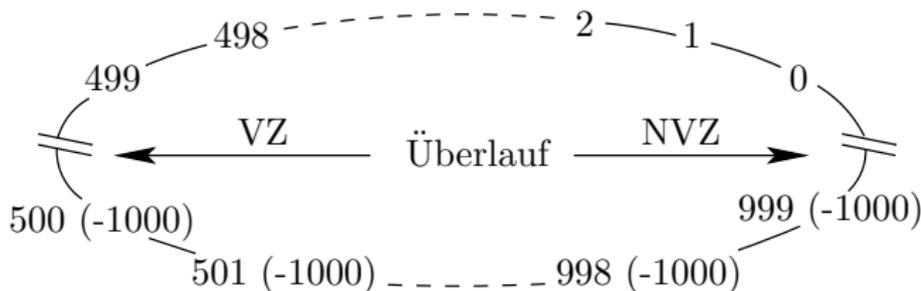
1. Wiederholung

$$Z + \bar{Z} + 1 = B^n$$

- Auflösung nach $-Z$:

$$-Z = \bar{Z} + 1 - \underbrace{\left[B^n \right]}_{*} \quad * \text{ nicht darstellbar}$$

- Die Zählreihenfolge bleibt, nur der Darstellungsbereich verschiebt sich:

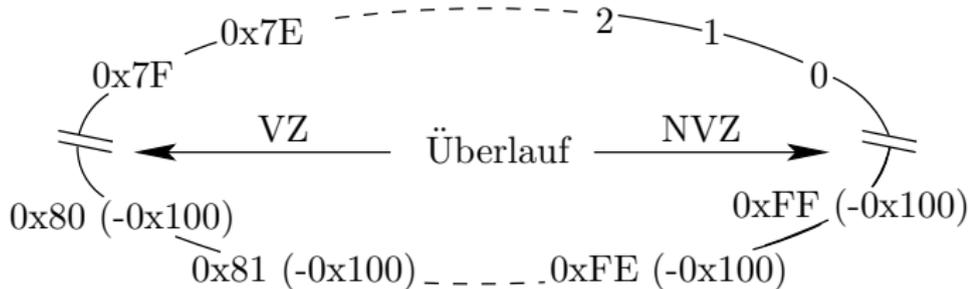


$$217 - 437 = 217 + \overline{437} + 1 = 217 + 562 + 1 = 778 - 1000 = -222$$



Zweierkomplement

- Basis: $B = 2$; Ziffern $\in \{0, 1\}$.
- Stellenkomplement: bitweise Negation.
- Das führende Bit ist das Vorzeichenbit.



0	1	1	0	0	1	0	1	$(-0 \cdot 2^8)$	Erweiterung zu vorzeichenbehaftete Zahlen
+1	0	0	0	0	0	0	1	$(-1 \cdot 2^8)$	
1	1	1	0	1 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	0	$(-1 \cdot 2^8)$	

$$0x65 + (0x83 - 0x100) = 0x65 - 0x7D = 0xE8 - 0x100 = -0x18$$



Hausaufgabe LFSR



Die nachfolgende Funktion »rand8()« zur Erzeugung von 8-Bit Pseudo-Zufallszahlen verwendet eine globale uint8_t-Zustandsvariable »rand_state«, bildet die Folgezustands- und Ergebnisbits 0 bis 6 durch Rechtsverschiebung und Bit 7 als EXOR-Summe der Bits 0, 2, 3, und 4 des alten Wertes der Zustandsvariablen.

0	0	1	1	0	0	0	1
0	0	0	1	1	0	0	0
0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	0
1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

```
uint8_t rand_state = 0x31; // Vereinbarung global
uint8_t lfsr(){//uint8_t x}{ Übergabe x Fehler in Aufgabe
    //Vereinb. Zustand, Anfangswert 0x31 *1
    #define x rand_state; // x anderer Name für rand_state
    return (x>>7) | (0x80 & ((x<<7)^(x<<5)^(x<<4)^(x<<3)));
}
```

*1: Vereinbarung als »static rand_state = 0x31;« denkbar, aber dann wird »rand_state« bei jedem Aufruf erst auf 0x31 gesetzt und immer der Folgezustand von 0x31, d.h. 0x18 zurückgegeben.



Simulation Modultest

Ein Testobjekt auf den bisherigen Foliensätzen war folgendes Unterprogramm zur Quadrierung:

```
uint32_t quad(int16_t a){ // Quadratberechnung
    return (uint32_t)a * a; // Warum mit Typcast?
};
```

Testbeispiele sind Tupel aus Eingaben und Sollausgaben. Man kann dafür einen neuen Datentyp definieren:

```
typedef struct{
    int16_t x;           //Eingabe
    uint32_t y;         //Sollausgabe
} test_t;
```

Ein »struct« ist eine Zusammenfassung aus bereits definierten Datentypen¹.

¹Grundregel der Programmierung: Alles was man benutzt (jeder Bezeichner, jede Variable, jeder Typ, ...), muss vorher im Programmtext definiert sein.



Testsatz

Eine Testsatz als Menge von Tests ist im einfachsten Fall ein initialisiertes Feld von Testbeispielen

```
test_t testsatz[] ={{<Tupel1>},{...},...};
```

Testbeispiele für die Quadratberechnung:

```
test_t testsatz[] ={           //Eingabe  Sollwert
    {0, 0},                    // 0x0000  0x00000000
    {1, 1},                    // 0x0001  0x00000001
    {9, 81},                   // 0x0009  0x00000051
    {-5, 25},                  // 0xFFFB  0x00000019
    {463, 214369},            // 0x01CF  0x00034561
    {0x7FFF, 1073676289}     // 0x7FFF  0x3FFF0001
};
```

Welche Testbeispiele führt das Programm falsch aus? Ausprobieren!



Testrahmen

Programm, das in einer Schleife alle Testbeispiele abarbeitet und die Ergebnisse kontrolliert oder zur Kontrolle ausgibt:

```
int main(){
    uint8_t idx, err_ct=0;
    uint32_t erg;
    for (idx=0; idx<6;idx++){
        erg = quad(testsatz[idx].x); //Istwert
        if (erg != testsatz[idx].y) //Soll/Ist-Vergl.
            err_ct++; //Fehlfkt. zählen
    }
}
```

Testdurchführung mit dem im Simulator im Debug-Modus:

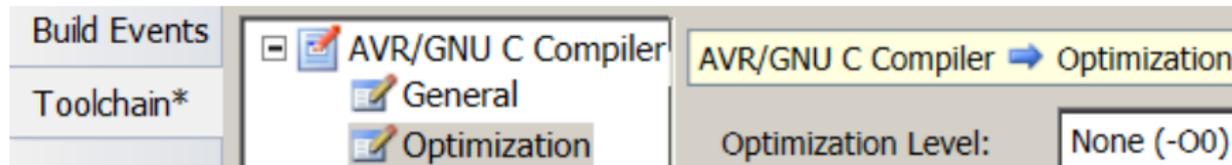
- Unterbrechungspunkt vor dem Fehlerzähler.
- Bei jeder Fehlfunktion Halt am Unterbrechungspunkt.



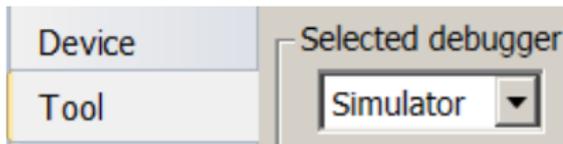
Projekt mtest_quad« ausprobieren



- Projekt »mtest_quad« öffnen.
- Compiler-Optimierung ausschalten (in -O0 ändern):
Project > mtest_quad Properties... (Alt+F7)



- Auswahl des Simulators als »Debugger«:



- Übersetzen.
- Debugger starten:



1. Wiederholung

Test und Fehlerlokalisierung im Schrittbetrieb:

```
struct{
    int16_t x;
    uint32_t y;
} testsatz[]={
    {0, 0}, {1, 1}, {9, 81},
    {-5, 25}, {463, 214369},
    {0x7FFF, 1073676289}
};

int main(){
    uint8_t idx, err_ct=0;
    uint32_t erg;
    for (idx=0; idx<6;idx++){
        erg = quad(testsatz[idx].y);
        if (erg != testsatz[idx].y)
            err_ct++;
    }
}
```

Watch 1	
Name	Value
<input type="checkbox"/> testsatz	{struct [6]{data}@0x0200}
<input checked="" type="checkbox"/> [0]	{struct {data}@0x0200}
<input checked="" type="checkbox"/> [1]	{struct {data}@0x0206}
<input type="checkbox"/> [2]	{struct {data}@0x020c}
<input checked="" type="checkbox"/> x	0x0009
<input checked="" type="checkbox"/> y	0x00000051
<input checked="" type="checkbox"/> [3]	{struct {data}@0x0212}
<input checked="" type="checkbox"/> [4]	{struct {data}@0x0218}
<input checked="" type="checkbox"/> [5]	{struct {data}@0x021e}
<input checked="" type="checkbox"/> idx	0x02
<input checked="" type="checkbox"/> erg	0x000019a1





Alternative Test vom PC

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
}
```



1. Wiederholung

Testobjekt:

```
uint32_t quad(int16_t a){ // Quadratberechnung
    return (uint32_t)a * a; // Warum mit Typcast?
};
```

Testrahmen:

```
#include "com_pc.h"
void main(){
    com_pc_init();
    while (1){
        int16_t a = (get_byte() << 8) | get_byte();
        uint32_t p = quad(a);
        for (uint8_t idx=0; idx<4; idx++){
            sendByte(p>>24);
            p = p << 8;
        }
    }
}
```

Header com_pc.h:

```
void com_pc_init();
uint8_t getByte();
void sendByte(uint8_t dat);
```

Testdurchführung

Hardware vorbereiten:

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



Software vorbereiten:

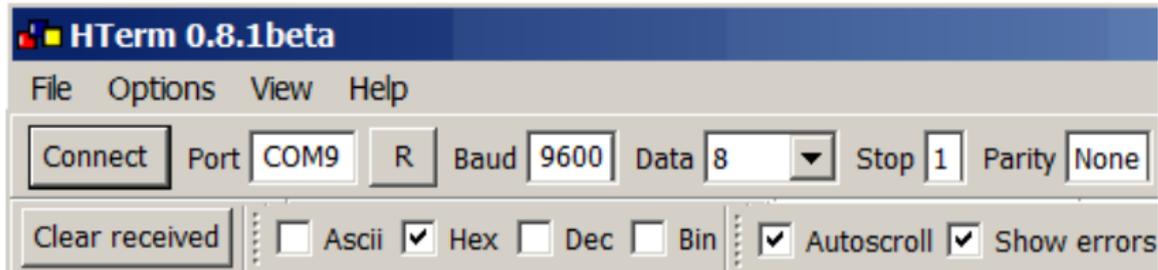
- Projekt anlegen, Programm eingeben.
- »Dragon« und Compileroptimierung »-O0« auswählen.
- Übersetzen und im Debugger starten.



Verbindung mit HTerm herstellen



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



- Beispiele aus Eingaben und Sollausgaben vorbereiten.
- Eingabe byteweise eintippen und Ausgaben byteweise mit Sollwerten vergleichen.
- Testautomatisierung auch bei PC-Test wünschenswert.



Schnellkurs Python



Python-Installation für die Übungen²

Die Übung nutzt Python3 unter Windows:

- Von »www.python.org« für die neueste »stable« Version (aktuell 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".

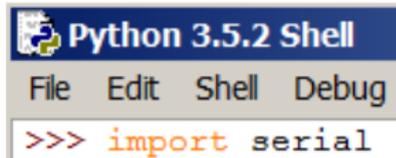
```
pip install pyserial  
pip install matplotlib
```

²Im 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 > Python 3.5 > IDLE (Python 3.5 ...)



Eine der wichtigsten Funktion zum Probieren:

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

Aufruf mit einer kommaseparierten Liste von Objekten. Ausgabe der Textdarstellungen, getrennt durch Leerzeichen.



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>)`
- Verketteten von Texten: `Text1 + Text 2`

Zum Ausprobieren auf der Konsole:

```
>> a=5; b=37.7; s='Hallo'  
>> >print(str(a) + str(b) + s) #print(a, 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.mf" – 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 geladen³. Die nachfolgende Eingabeanweisung braucht man vor allem, um Python-Scripte auf dem PC anzuhalten:

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

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
```

³Grund: Tests werden mehrfach wiederholt und das mehrfache Eingeben derselben Werte ist lästig und fehleranfällig.



Sequenzobjekte: Zeichenketten, ...

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"
```

Wiederhole für alle Zeichen einer Zeichenkette:

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



Tupel: Zusammenfassung beliebiger Objekte.

```
>> t = ('abc', 3, 4.2)
>> print(t, t[1], t[1:2])
>> (('abc', 3, 4.2), 3, (3,))
>> for x in t: # für alle Elemente in t
...     print(x) # Einrücktiefe erhöhen
...         # Einrücktiefe zurücksetzen
('abc', 3, 4.2)
3
(3,)
```

Liste: wie Tupel 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]
```



2. Schnellkurs Python

Iteratoren: Algorithmische Beschreibung von Zahlenfolgen zur Nachbildung der Zählschleifen in Sprachen wie c:

```
>> print(range(3))
range(0, 3)
>> for i in range(3):
...     print(i)     # Einrücktiefe erhöhen
...                 # Einrücktiefe zurücksetzen
0
1
2
```

Anfangswert ungleich null und Schrittweite ungleich 1:

```
>> for i in range(5, -4, 3):
...     print(i)     # Einrücktiefe erhöhen
...                 # Einrücktiefe zurücksetzen
-4
-1
2
```

Nach »:« beginnt ein neuer Anweisungsblock. Anweisungsblöcke, in c »{...}«, werden in Python durch »gleiche Einrücktiefe« markiert.



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 Einrücktiefe zurücksetzen.



Schleife mit Fallunterscheidung:

```
>> i=0;
>> for c in s:
...   if c=='l': # Einrücktiefe erhöhen
...     print(i) # Einrücktiefe erhöhen
...     i=i+1   # Einrücktiefe zurücksetzen
...           # Einrücktiefe zurücksetzen
2
3
```



Programm als Datei

In der »Idle«:

File > New File (Ctrl+N)

Programmeingabe:

```
s='Hallo'; i=0;
for c in s:
    if c='l': # Einrücktiefe erhöhen
        print(i) # Einrücktiefe erhöhen
    i=i+1;    # Einrücktiefe zurücksetzen
```

Programm Speichern

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

und starten

Run > Run Modul (F5).

Alternativ können Python-Programme mit jedem Text-Editor geschrieben und auf einem Terminal gestartet werden (siehe später »Echo-Test mit Python«).



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 Testbeispielausgabe ist dann »False«:

```
>> import add # alle in add.py definierte Obj.
>> print(add.add(3,5)) # <Modulname>.<Objektname>
8
>> from add import add # nur Unterprogramm add
>> print(add(2,9)) # nur Objektname
11
```

Bei Ausführungsfehlern (Division durch null, serielle Schnittstelle lässt sich nicht öffnen, ...) beendet sich ein Python-Programm mit einer Fehlermeldung. Alternative Fehlerbehandlung:

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



Anwendung von »try-catch« zur Ermittlung verfügbarer COM-Ports unter Windows

```
def get_COMs():
    import serial
    port_list = []
    for pnr in range(1, 255):
        port = 'COM%i'%(pnr)
        try:
            s = serial.Serial(port)
            s.close()
            port_list.append(port)
        except (OSError, serial.SerialException):
            pass
    return port_list

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

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



Echo-Test mit Python



Python als Programmiersprache für Tests

- Definition der Testbeispiele als Tupel
- Abarbeitung in einer Schleife
 - Sende Eingaben
 - Warte auf Ausgaben.
 - Kontrolle der Ausgaben
 - Protokollieren erkennbarer fehlerhafter Werte
- Messung der Antwortzeit
- graphische Darstellung von Ergebnissen
- ...

Experimente:

- 1 Test des Echo-Programms mit einem Python-Script.
- 2 Untersuchung der Übertragungsdauer.
- 3 Modultest mit Python-Skript auf dem PC.



3. Echo-Test mit Python

Echoprogramm vorbereiten

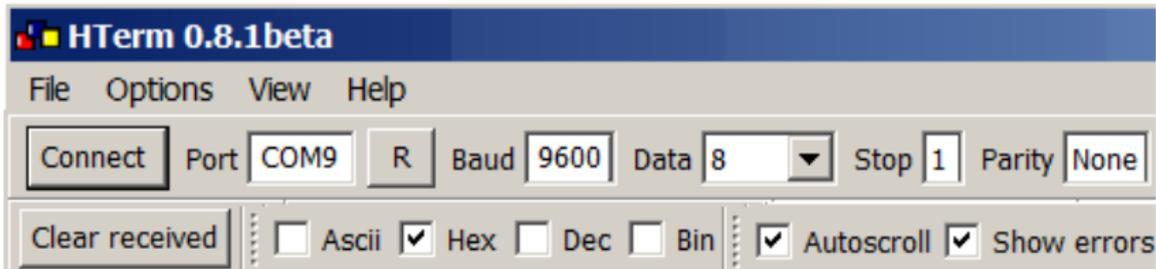
Hardware vorbereiten: (PModUSBUSART an JH oben, ... (Folie 15)

Software vorbereiten:

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

Verbindung mit HTerm herstellen:

- COM-Port auswählen, 9600 Baud, 8 Daten-, 1 Stopp- und kein Paritätsbit
- Verbindung herstellen (Connect).





3. Echo-Test mit Python

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



Clear received Ascii Hex Dec Bin

Received Data

1	2	3	4	5	6	7	8	9	10	11	12	13
89	45	23										

empfangene Zahlen

Input control

Input options

Clear transmitted Ascii Hex Dec Bin

Type **HEX** Hex.-Zahlen eingeben + Enter

Transmitted data

1	2	3	4	5	6	7	8	9	10	11	12	13	14
89	45	23											

gesendete Zahlen

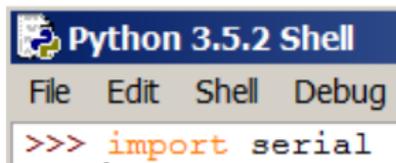
- Kontrolle, dass alle versendeten Zahlen zurückgesendet werden.
- Verbindung mit HTerm schließen (Disconnect)



Test des Echoprogramms mit Python

Nachdem Kommunikation mit HTerm getestet ist⁴ und Verbindung geschlossen⁵ ist:

- Start der Programmierkonsole »Idle« von 
 - > Start > Python 3.5 > IDLE (Python 3.5 ...)
- Import des Moduls für die serielle Kommunikation:



```
Python 3.5.2 Shell
File Edit Shell Debug
>>> import serial
```

⁴Wichtig, damit man weiß, dass der Mikrorechner verbunden ist und das Echo-Programm läuft.

⁵Das Betriebssystem auf dem PC erlaubt je COM-Port nur einem Programm, sich mit diesem zu verbinden.



Kommandos zum Ausprobieren

- Kommunikationsverbindung öffnen⁶:

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



- Zeichenfolge an Variable zuweisen:

```
send = "Hallo_ Welt!"
```

- Anschauen von Typ, Wert und Länge der Zeichenkette:

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

⁶Denselben COM-Port wie im HTerm benutzen. Das Programm P05\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.



3. Echo-Test mit Python

- 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 Wertes »`w`«.



Fortsetzung Dienstag

17.11.2020



Lösung Test 1

Aufgabe 1:

```
uint8_t get_byte(){ // Empfangen eines Bytes
    while (!(UCSR2A & (1<<RXC2))); // warte auf Empfang
    return UDR2; // Byte zurückgeben
}
```

Aufgabe 2:

```
uint8_t byte_ct = 0;
uint8_t get_byte(){ // Empfangen eines Bytes
    while (!(UCSR2A & (1<<RXC2))); // warte auf Empfang
    return UDR2; // Byte zurückgeben
    byte_ct ++;
}
```



3. Echo-Test mit Python

Aufgabe 3:

```
// nicht blockierender Empfang
uint8_t get_byte(uint8_t ptr){
    if (UCSR2A & (1<<RXC2)){           // wenn Empfang
        *ptr = UDR2;                   // Byte übergeben
        return 1;                       // Rückgabe 1
    }
    return 0;                           // Rückgabe 0
}
```

Aufgabe 4 und 5

```
typedef struct{
    uint8_t a, b;
    uint16_t y;
} test_t;
test_t Testsatz[] = {{45, 47, 2367}, {40, 519(*), 50896}};
```

(*) WB-Fehler bei Zuweisung konstanter Werte bemerkt der Compiler

Teilnehmer, bei denen im Test kaum was richtig war, bitte am Ende der Vorlesung im bbb-Raum bleiben.



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 \Leftrightarrow Byte-Vektor.
- Zeilen eines Blocks **müssen** gleiche Einrücktiefe haben!

Programmdatei in der »Idle« öffnen:

File > Open > H:\~\Informatikwerkstatt\P05\Python\scom.py

Programmstart mit Ausgabe auf der »Idle«:

Run > Run Module (F5)



Start auf der Konsole



- Konsole öffnen (»Windows-Taste + R«, im sich öffnenden Feld »cmd« eingeben, Enter).
- Wechsel in das Verzeichnis mit dem Python-Programm, im Bild »H:\Informatikwerkstatt\Python«. Eingabe Programmname + Enter:

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

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



3. Echo-Test mit Python

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
      + "_dt=" +str(dt*1E3)+"_ms")
ser.close()
```

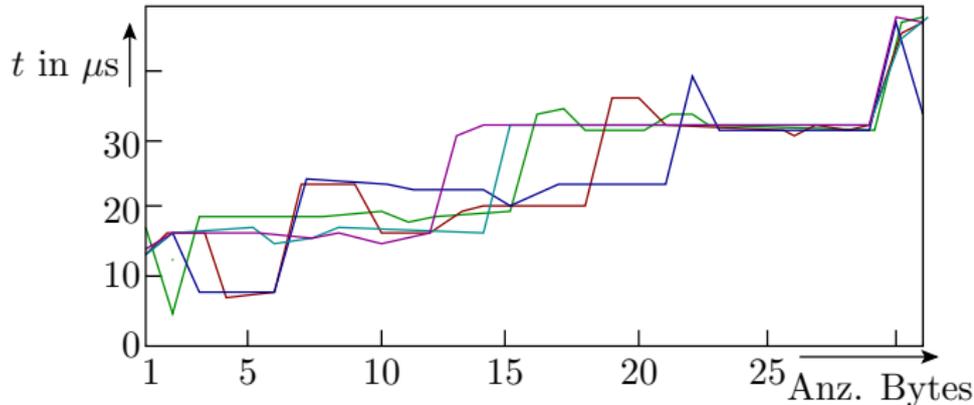
Programmstart und Ausgabe auf der Konsole:

```
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 Gesamtlänge
    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, "dt=" +
          +str(dt*1000)+"ms")
    l = l+1; #Ende des Schleifenkoerpers
ser.close() #danach Einruecktiefe ruecksetzen
```

- Anweisungen für graphische Ausgabe siehe übernächste Folie.



3. Echo-Test mit Python

Test von »scom_txy.py« auf der Konsole



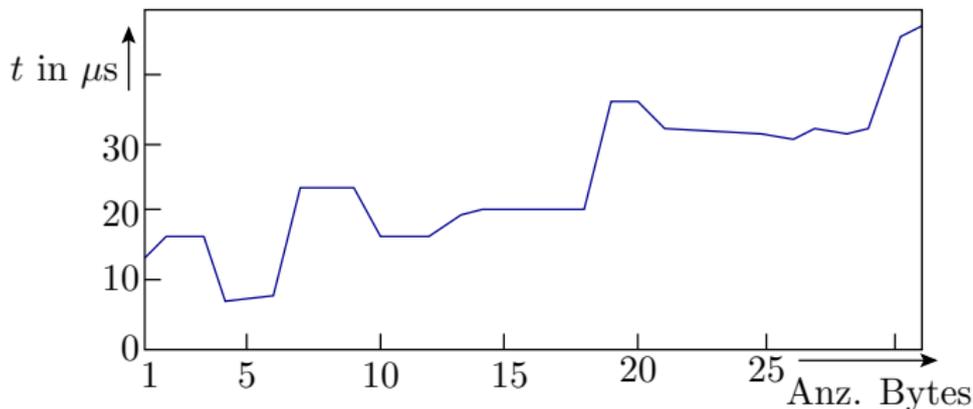
```
H:\Informatikwerkstatt\Progr_IW\Python>scom_txy.py
Empfangene Daten: D dt= 13.2549278489 ms
Empfangene Daten: Da dt= 14.921187855 ms
Empfangene Daten: Das dt= 14.4043923316 ms
Empfangene Daten: Das dt= 14.4265738178 ms
Empfangene Daten: Das i dt= 14.3878389836 ms
Empfangene Daten: Das is dt= 14.2295889771 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
```



3. Echo-Test mit Python

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  $\mu$ s")      #Achsenbeschrift.  
plt.show()                    #anzeigen
```

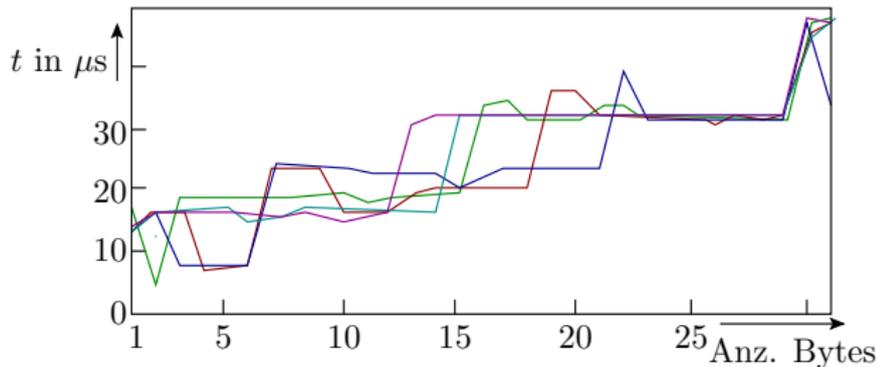




3. Echo-Test mit Python

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(); ...
```





3. Echo-Test mit Python

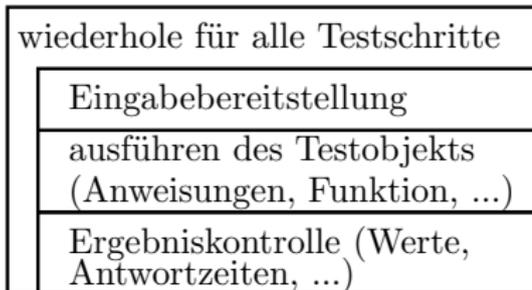
- Bei 9600 Bitzeiten pro s, 1 Startbit + 8 Datenbits + 1 Stoppbit dauert eine Byteübertragung ≥ 1 ms.
- Für 21 bis 28 Byte große Pakete werden etwa 30 ms benötigt, d.h. fast max. Durchsatz.
- Auf anderen Rechnern, zeitgleichen Übertragungen über USB, Nutzung von Bluetooth, ... kann die Übertragung auch 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



Testrahmen



Die Basisfunktionen für den Test vom PC:

- Übergabe von Eingabe-Bytes und die
 - Rückgabe von Ergebnis-Bytes zur Auswertung.
- sind im Echo-Programm enthalten.



Modularisierung des Echoprogramms

Aufteilung des Echoprogramms »echo.c« vom vorherigen Foliensatz in nachnutzbare Module für den Test von Programmbeisteinen:

```
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_ */
```

- Der Header wird in die C-Dateien, die die Funktionen definieren und nutzen eingefügt.
- In nutzenden Dateien macht das die Aufrufchnittstelle bekannt.
- In der definierenden Datei dient das zur Kontrolle, dass Definition und Implementierung der Aufrufchnittstellen gleich sind.
- #ifndef ... #define ... #endif – Precompiler-Anweisungen zur Verhinderung einer Mehrfacheinbindung.



Echoprogramm aus Funktionsbausteinen

```
#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: »avr/io.h« ist bereits in »com_pc.h« eingefügt. Ohne dem »Precompiler-Konstrukt« auf Folie zuvor Mehrfacheinfügung.



Modultest vom PC – 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.



4. Modultest vom PC aus

```
#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 »F4-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.

Transmitted data				Received Data							
1	2	3	4	1	2	3	4	5	6	7	8
47	0C			47	0C	53	3B	05	0B	03	54
071	012			071	012	083	059	005	011	003	084

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



Modultest mit Python



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.



5. Modultest mit Python

```
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
```

Die Tests zu den Ausgabezeilen mit »False« haben versagt. Wo liegt die Fehlerursache:

- im Testobjekt
- im Testrahmen auf dem PC oder
- im Python-Programm?

Typfehler im Testobjekt: »uint8_t d« kann keine negativen Differenzen darstellen.



Aufgaben



Aufgabe 5.1: Ausprobieren

- 1 Ausprobieren der Anweisungsbeispiele aus dem Python-Schnellkurs.
- 2 Ausprobieren der Tests für das Echoprogramm.
- 3 Ausprobieren des Modultests mit Python.



Aufgabe 5.2: Modultest Vorzeichenzahlen

Ändern Sie das Mikrorechnerprogramm Folie 57 und das Python-Programm Folie 61 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))
```

Hinweise:

- Im Web suchen, wie in Python Vorzeichenzahlen in Byte-Strings umgewandelt werden.
- Entwicklung von μ P-Funktionen für das Senden und den Empfang von Vorzeichenzahlen.



Aufgabe 5.3: Test einer 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, das dieses Programm mit zehn zufällig ausgewählten Beispielen testet.

Kontrolle für Testbeispiele mit Sollergebnis ungleich Ist-Ergebnis:

- Fehler im Python-Programm?
- Im Mikrorechner ankommende Daten falsch?
- Rechnet der Mikrorechner falsch?
- Interpretiert Python den Rückgabewert falsch?

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 22):

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

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