Leistungskurs Praktische Q3 Hessen

Skript

Shamsher Singh Kalsi

Berufliches Gymnasium — Ferdinand-Braun Schule
Kursleiter: Herr Sebastian Stolz

9. September 2025

Inhaltsverzeichnis

1 Einführung **2**

1 Einführung

05.09.2025

Aufgabe 1.1: Serielle Kommunikation

1. Beantworte folgende Fragen schriftlich:
   - An welcher Stelle spielt die serielle Kommunikation heutzutage eine Rolle?
   - Erkläre die Begriffe: Startbit, Datenbit, Stoppbit.
   - Ein PC sendet den Buchstaben ’A’ mit 1 Startbit, 8 Datenbits, 1 Stoppbit. Skizziere das resultierende Bitmuster
   - Welche Parameter müssen Sender und Empfänger bei RS232 vorab gemeinsam einstellen?
   - Warum reicht ein einziger Draht für die Übertragung?
2. Aufgabe 2 – Serielle Kommunikation (RS232) in Java mit Hilfe eines Emulators
   - Warum brauchen Sender und Empfänger die gleiche Baudrate?
   - Was passiert, wenn der Sender schneller schreibt als der Empfänger lesen kann? Welche Lösungen gibt es für dieses Problem?

Lösung 1.1: Serielle Kommunikation

1. Beantworte folgende Fragen schriftlich:

   - Serielle Kommunikation wird heute noch in eingebetteten Systemen, Industrieanlagen, Messgeräten und auch beim Serverzugriff (Konsolenport) genutzt, da sie einfach, robust und für kurze Distanzen ausreichend ist.
   - **Startbit:** signalisiert Beginn eines Zeichens (logisch 0) und dient zur Synchronisation. **Datenbits:** eigentliche Nutzinformation, meist 8 Bit, LSB zuerst. **Stoppbit:** beendet die Übertragung (logisch 1), Leitung geht in Idle-Zustand.
   - Beispiel: Der Buchstabe A hat den ASCII-Wert $0\mathrm{x}41 = 01000001$. Übertragung (LSB zuerst) mit 1 Startbit und 1 Stopbit:

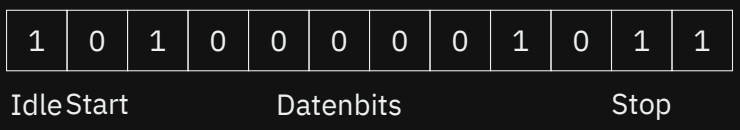
     $$\underbrace{1}_{\text{Idle}}\ \underbrace{0}_{\text{Start}}\ 10000010\ \underbrace{1}_{\text{Stop}}\ 1$$

   - Sender und Empfänger müssen sich bei RS232 vorab einigen auf: Baudrate, Datenbits, Parität, Stoppbits, sowie ggf. Art der Flusskontrolle.
   - Ein einzelner Draht genügt pro Richtung, weil Bits nacheinander mit fester Baudrate gesendet werden; Start- und Stoppbits übernehmen die Synchronisation. Für echte Vollduplex-Kommunikation sind jedoch zwei Leitungen (Tx/Rx) plus Masse üblich.

2. Aufgabe 2 – Serielle Kommunikation (RS232) in Java mit Hilfe eines Emulators

   - Beide Seiten brauchen dieselbe Baudrate, da es kein separates Taktsignal gibt. Unterschiedliche Baudraten führen zu falscher Bitinterpretation.
   - Wenn der Sender schneller schreibt als der Empfänger liest, läuft dessen Puffer über und Daten gehen verloren. Lösungen: Hardware-Flowcontrol (RTS/CTS), Software-Flowcontrol (XON/XOFF), größere FIFO-Puffer oder Protokolle mit Bestätigung (ACK/NACK).

```java
import com.fazecast.jSerialComm.SerialPort;
public class Sender {
  public static void main(String[] args) throws Exception {
    SerialPort sp = SerialPort.getCommPort("COM5");
    sp.setBaudRate(9600);
    sp.openPort();
    sp.getOutputStream().write("Hallo, COM6\n".getBytes());
    sp.closePort();
  }
}
```


1 0 1 0 0 0 0 0 1 0 1 1
Idle
Start
Datenbits
Stop

09.09.2025

Aufgabe 1.2: Steuerung eines Mikroprozessors mit der seriellen Schnittstelle

1. Schreibe ein (Python-)Programm, welches einen selbst gewählten Sensor auf einem Raspberry-PI oder einem Arduino steuert.
2. Erweitere das Programm so, dass es über eine serielle Schnittstelle angesprochen werden kann. Emuliere die serielle Schnittstelle mit Hilfe von Software oder nutze einen RS232/TTL Wandler mit MAX3232
3. Nutze die Klasse SSerialäus dem Moodle-Kurs, um von einem Laptop oder PC mit einem Java-Programm über die serielle Schnittstelle den Sensor zu steuern.

Lösung 1.2:

Aufgabe 1.3: Steuerung eines Mikroprozessors mit der seriellen Schnittstelle

1. Schreibe ein Python-Programm, das einen Sensor bzw. ein Aktor-Device auf einem Raspberry Pi steuert.

Listing 1: Raspberry Pi: LED-Steuerung + serielle Steuerung (pySerial + gpiozero)

```
# requirements: gpiozero, pyserial
# python3 script that controls an LED via gpiozero and listens on a serial port for commands
from gpiozero import LED
import serial
import time

LED_PIN = 17
SERIAL_PORT = '/dev/ttyUSB0' # auf RPi: USB-Serial oder /dev/ttyAMA0 etc.
BAUD = 9600

led = LED(LED_PIN)
ser = serial.Serial(SERIAL_PORT, BAUD, timeout=1) # pySerial: blocking read behaviour depends on timeout

def handle_line(line):
line = line.strip()
if line.upper() == 'LED ON':
led.on()
ser.write(b'OK\n')
elif line.upper() == 'LED OFF':
led.off()
ser.write(b'OK\n')
elif line.upper() == 'STATUS':
ser.write(('ON\n' if led.is_lit else 'OFF\n').encode())
else:
ser.write(b'ERR Unknown command\n')

try:
while True:
raw = ser.readline() # read a line terminated by newline; depends on timeout. (pySerial)
if raw:
handle_line(raw.decode('utf-8', errors='ignore'))
time.sleep(0.01)
finally:
ser.close()
led.off()
```

Dieses Python-Beispiel verwendet ‘gpiozero‘ zur einfachen GPIO-Abstraktion auf dem Raspberry Pi und ‘pySerial‘ für die serielle Schnittstelle; ‘pySerial‘’s Lese-/Timeout-Verhalten ist dokumentiert und beeinflusst, wie ‘readline()‘ blockiert. :contentReference[oaicite:1]index=1

2. Ein Arduino-Beispiel, das serielle Kommandos entgegennimmt und einen digitalen Pin steuert:

Listing 2: Arduino: Serial command handler

```
// Arduino sketch: listens on Serial, toggles digital pin, replies with status
const int LED_PIN = 13;

void setup() {
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(9600); // init serial
}

void loop() {
  if (Serial.available() > 0) { // number of bytes available
    String cmd = Serial.readStringUntil('\n'); // read until newline
    cmd.trim();
    if (cmd == "LED ON") {
      digitalWrite(LED_PIN, HIGH);
      Serial.println("OK");
    } else if (cmd == "LED OFF") {
      digitalWrite(LED_PIN, LOW);
      Serial.println("OK");
    } else if (cmd == "STATUS") {
      Serial.println(digitalRead(LED_PIN) ? "ON" : "OFF");
    } else {
      Serial.println("ERR");
    }
  }
}
```

Die Arduino-API stellt ‘Serial.available()‘ und ‘Serial.read()‘ / ‘readStringUntil()‘ bereit; ‘available()‘ gibt die Anzahl bereits empfangener Bytes an, ‘read()‘ liefert das nächste Byte oder -1, wenn nichts da ist — das ist der übliche Pattern für nicht-blockierende Abfragen auf Arduino. :contentReference[oaicite:2]index=2

3. Java-Client mit der Klasse Serial (wie in deinem Moodle-Skript beschrieben). Das Beispiel öffnet den Port, schickt einen String, liest eine Antwortzeile und parst eine Zahl mit Double.parseDouble(...):

   Listing 3: Java: Steuerprogramm (Nutzungsbeispiel der in der Aufgabenstellung beschriebenen Serial-Klasse)

```
// Beispiel: Java-Client (konform mit der Serial-Klassen-API aus der
    Aufgabenstellung)
public class SerialController {
  public static void main(String[] args) {
    Serial s = new Serial("COM5", 9600, 8, 1, 0); // parity 0 = none, falls so
        erwartet
    if (!s.open()) {
      System.err.println("Port konnte nicht geöffnet werden");
      return;
    }
    // Beispiel: sende Kommando, warte auf Antwort als Zeile
    s.write("STATUS\n");
    String reply = s.readLine(); // blockiert bis Zeile komplett
```

```
            System.out.println("Reply: " + reply);
            // parsing numeric reply example
            try {
              double val = Double.parseDouble(reply.trim());
              System.out.println("Parsed value: " + val);
            } catch (NumberFormatException e) {
              System.out.println("No numeric reply: " + reply);
            }
            s.close();
          }
        }
```

In Java wandelt ‘Double.parseDouble(String)‘ einen String in einen primitiven ‘double‘ um; das ist die standardisierte Methode in der Java-API. Für ereignisgesteuertes Lesen oder Low-Level-Buffers sind Bibliotheken wie jSerialComm nützlich, die sowohl blockierende als auch non-blocking Modi und Event-Listener unterstützen. :contentReference[oaicite:3]index=3

4. Hinweis zur Pegelwandlung und Verbindung: Wenn du echte RS-232-Signale anschließt (±V-Level) musst du einen Pegelwandler wie den MAX3232 verwenden; TTL-UART-Pins (3.3 V/5 V) dürfen nicht direkt an RS-232-Level angeschlossen werden. Der MAX3232 bietet Treiber/Empfänger und die nötigen Charge-Pump-Kondensatoren. :contentReference[oaicite:4]index=4

5. Bonus: Protokoll mit Längenpräfix und Erkennung unvollständiger Übertragung.

   Implementiere auf Senderseite: sende zuerst ASCII-Länge, dann “:“, dann die Nachricht (z. B. ‘12:Hello World¡), oder sende ein 2-Byte Binary-Length-Prefix gefolgt von Rohbytes. Auf Empfängerseite: lese zunächst bis “:“, parse die Länge, dann lies exakt diese Anzahl Bytes; falls weniger Bytes empfangen wurden, warte weiter oder melde „incomplete“ — mit ‘read(b,len)‘ kannst du gezielt eine bestimmte Anzahl von Bytes lesen und prüfen, wie viele tatsächlich geliefert wurden. Diese Technik verhindert das Vermischen von Nachrichten bei Stream-Orientierung. :contentReference[oaicite:5]index=5

Theorem 1.1: Fundamentaler Satz

Inhalt des Theorems ...

Beispiel 1.1: Erstes Beispiel

Dieses Beispiel illustriert den Satz.

Aufgabe 1.4: Rechenaufgabe

Bearbeite folgende Aufgabe ...

Lösung 1.3: zur Aufgabe

Hier die Lösungsschritte ...

Hinweis

Ein kurzer Hinweis.

Terminal: Beispielcode

```
echo ”Hallo Welt”ls -la
```