Anwendungsorientierte Softwareentwicklung

Laborblatt 4: Spiel und Spaß mit der seriellen Schnittstelle

Lernziele

- Verständnis für serielle Kommunikation zwischen Mikrocontroller und Rechner entwickeln.
- Grundlagen der Programmiersprache Python anwenden.
- Ein bißchen `git`

Schritt 1: Repository Forken

Ziel: Ab jetzt beginnen wir schrittweise mit der Entwicklung eines Softwareprojekts. Dabei werden wir Schritt für Schritt vorgehen – und es wird sicher auch mal etwas schiefgehen. Aber genau dafür haben wir git – ein Werkzeug zur Versionsverwaltung.

1. Öffnet im Webbrowser die folgende URL: http://gitlab.uni-ulm.de/mlehn/asp-ulm
2. Loggt euch mit eurem kiz-Account ein.
3. Klickt oben rechts auf **Fork**.
4. Wählt bei **Select a namespace** euren kiz-Benutzernamen aus. Klickt anschließend unten auf **Fork project**.
5. Ihr landet nun auf der Projektseite eures eigenen Forks. Klickt dort auf **Code** und kopiert den Link unter **Clone with HTTPS**.
   Der Link hat die Form: https://gitlab.uni-ulm.de/**BENUTZERNAME**/asp-ulm
6. Wie gewohnt arbeiten wir mit zwei Terminals nebeneinander – links zum Editieren, rechts zum Ausführen von Kommandos.

Linkes Terminal (Editieren)

A) **Euer Git-Repository klonen**
   Klont euren Fork mit:
   `git clone https://gitlab.uni-ulm.de/BENUTZERNAME/asp-ulm`
   Wechselt anschließend ins Verzeichnis mit
   `cd asp-ulm`

B) **Zum Test eine Datei ändern**
   Öffnet die Datei `README.md` und schreibt in eine neue Zeile: `Jetzt geht's los!`

Rechtes Terminal (Kommandos ausführen)

A) Wechselt ebenfalls ins Projektverzeichnis mit `cd asp-ulm`
   Versucht nun, die Änderung zu committen mit `git commit -a`
   Dabei erscheint zunächst eine Fehlermeldung, etwa: **"Please tell me who you are…"**

B) Führt dann folgende zwei Kommandos aus (mit eurem eigenen Namen und E-Mail):
   `git config user.email "dein.name@uni-ulm.de"`
   `git config user.name "Dein Name"`

C) Führt jetzt erneut den Commit aus:
   `git commit -a`
   Es öffnet sich nano. Gebt eine kurze Nachricht ein, z. B.: `Mein erster Commit`
   Beendet nano mit Control + X und bestätigt mit y.

D) Übertragt euren Commit auf den Server :
   `git push`
   Dabei werdet ihr nach eurem kiz-Benutzernamen und Passwort gefragt.
   Achtet darauf: **Das Passwort wird beim Tippen nicht angezeigt**, nicht einmal als Sterne – das ist normal. Tippt es einfach blind ein und drückt Enter.

E) Wechselt zurück in den Browser, ladet die Seite neu und überprüft, ob eure Änderung übernommen wurde.

Schritt 2: Mikrocontroller programmieren mit dem Arduino Framework

Vorbereitung: Wechselt in beiden Terminals ins Unterverzeichnis `controller`.

Linkes Terminal: Öffnet im linken Terminal die Datei `controller.ino` mit `nano`. Dort steht folgendes Programm:

```
 1 void setup()
 2 {
 3     Serial.begin(9600);
 4     Serial.println("Hallo! Gleich kommen viele A's");
 5 }
 6
 7 void loop()
 8 {
 9     Serial.print("A");
10     delay(1000);
11 }
```

Rechtes Terminal

1. **Schließt den Mikrocontroller an.**
   Achtung: Unter WSL müsst ihr jedes Mal, wenn ihr einen Mikrocontroller anschließt, das Kommando `usb_check` ausführen, damit der USB-Port in WSL sichtbar wird.
2. **Programm kompilieren und hochladen:** `make upload`
3. **Seriellen Monitor starten:** `make monitor`
   Es sollte zunächst eine Zeile erscheinen mit: `Hallo! Gleich kommen viele A's` Und danach – wie versprochen – viele `A`s. :-)
   Den seriellen Monitor könnt ihr mit `Control-X` beenden.

Kurze Erklärung was passiert
In `controller.ino` steht eigentlich nur ein kleiner Ausschnitt des vollständigen C-Programms, das am Ende übersetzt wird. Die Funktionen `setup()` und `loop()` werden – wie hier gezeigt – intern in eine `main()`-Funktion eingebettet:

```
// Vereinfachte Darstellung – in Wirklichkeit stehen hier noch viele Includes
// und Initialisierungen

int main()
{
    // setup() wird genau einmal aufgerufen
    setup();

    // Danach wird loop() immer wieder aufgerufen
    while (1) {
        // Bei Arduino prüft eine interne Funktion z.B., ob serielle Daten
        // verfügbar sind:
        if (Serial.available()) { /* ... */}

        loop();
    }
}
```

Die Funktion `setup()` wird also genau einmal aufgerufen – in unserem Beispiel, um die serielle Schnittstelle zu initialisieren und eine Textzeile auszugeben.
Die Funktion `loop()` läuft dagegen in einer Endlosschleife: In jedem Durchlauf wird das Zeichen `A` ausgegeben und anschließend 1000 Millisekunden (also eine Sekunde) gewartet.

Was ihr euch für heute merken solltet:

- Mit `Serial.begin(9600)` initialisiert man die serielle Schnittstelle – hier mit einer Baudrate von 9600.
- Mit `Serial.print("foo")` wird die Zeichenkette `"foo"` ohne zusätzlichen Zeilenumbruch ausgegeben.
- Mit `Serial.println("foo")` wird die Zeichenkette `"foo"` mit anschließendem Zeilenumbruch (genauer: `\r\n`) ausgegeben.

Offene Fragen: Wie funktioniert eigentlich die serielle Schnittstelle genau?

Wenn `Serial.print("A");` ausgeführt wird, kann man auf Pin 1 (TX) mit dem Oszilloskop beobachten, wie das Zeichen `'A'` übertragen wird – als Folge von Spannungswechseln, die dem Bitmuster von `0x41` entsprechen.
Der Buchstabe `'A'` besitzt einen sogenannten ASCII-Wert und wird als 8-Bit-Zahl dargestellt: `01000001`. Diese Bits werden – ähnlich wie beim Morsen – nacheinander gesendet, indem der TX-Pin auf **HIGH** oder **LOW** gesetzt wird. Bei einer Baudrate von **9600** dauert jedes Bit genau 104 µs.
Aber: Die Daten werden nicht nur über **Pin 1 (TX)** ausgegeben, sondern **auch über das USB-Kabel übertragen** – mithilfe eines **seriell-zu-USB-Wandlers** (oft in den Mikrocontroller integriert oder im Board vorhanden). Deshalb kannst du das gesendete Zeichen auch im **seriellen Monitor auf dem Rechner** sehen, ganz ohne Oszilloskop.

Abschließend

Ändert Zeil 2 so, dass nur Hallo! mit anschliessendem Zeilenumbruch ausgegeben wird. Änderungen mit folgendem Befehl committen und ins Repository hochladen:

```
git commit -a -m "Schritt 2"
git push
```

Schritt 3: Spielerei mit dem Arduino Framework

Ändert die Datei `controller.ino` wie folgt:

```
 1 void setup()
 2 {
 3     Serial.begin(9600);
 4 }
 5
 6 int valueA0;
 7
 8 void loop()
 9 {
10     Serial.print("X ");
11     Serial.println(valueA0);
12     delay(100);
13     valueA0 += 1;
14 }
```

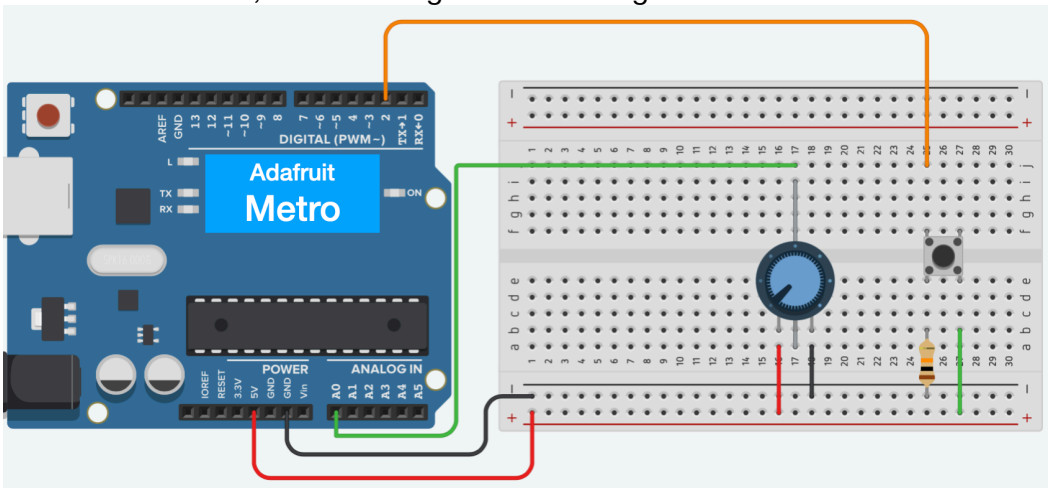
Erläuterung der Änderungen

- In der Funktion `setup()` wird jetzt ausschließlich die serielle Schnittstelle initialisiert.
- In Zeile 6 wird eine globale Variable `valueA0` vom Typ `int` deklariert. Da sie außerhalb einer Funktion definiert ist, handelt es sich um eine globale Variable. In C/C++ erhalten globale Variablen ohne Initialwert automatisch den Wert `0`.
- In Zeile 10 wird der Buchstabe `'X'` gefolgt von einem Leerzeichen ausgegeben. Dieses Leerzeichen ist später noch von Bedeutung.
- In Zeile 11 wird der aktuelle Wert der Variable `valueA0` ausgegeben, gefolgt von einem Zeilenumbruch durch `println`. Auch dieser Zeilenumbruch wird noch eine Rolle spielen.
- In Zeile 13 wird `valueA0` bei jedem Durchlauf der `loop()`-Funktion um eins erhöht.

Aufgabe: Das geänderte Programm übersetzen, den Mikrocontroller flashen und die Ausgabe im seriellen Monitor mit der erwarteten Ausgabe vergleichen.
Abschließend: `git commit -a -m "Schritt 3"` und `git push`

Schritt 4: Bastelei und noch mehr Spielerei

Schnappt euch **Breadboard, Kabelbrücken, Kabel, einen 10kΩ-Widerstand, einen Taster und ein Potentiometer**, und baut folgende Schaltung auf:



Wichtig:
- Das mittlere Beinchen des Potentiometers wird mit Pin A0 verbunden.
- Ein Bein des Tasters wird mit Pin 2 verbunden (das andere über den Widerstand nach GND).

Bearbeitet die Datei `controller.ino` wie folgt:

```
 1 void setup()
 2 {
 3     Serial.begin(9600);
 4     pinMode(2, INPUT);
 5 }
 6
 7 int valueA0;
 8
 9 void loop()
10 {
11     if (digitalRead(2)) {
12         Serial.println("A");
13     }
14
15     int analogValue = analogRead(A0);
16     if (valueA0 != analogValue) {
17         valueA0 = analogValue;
18         Serial.print("X ");
19         Serial.println(valueA0);
20     }
21     delay(100);
22 }
```

Aufgabe
Bevor das Programm im Detail erklärt wird, sollt ihr zunächst selbst beobachten, was es macht:
- Das Programm übersetzen, den Mikrocontroller flashen und den seriellen Monitor starten.
- Beobachten, was passiert, wenn ihr am Potentiometer dreht – insbesondere bei den maximalen Auslenkungen.
- Beobachten, was passiert, wenn ihr den Taster drückt.

Abschließend: `git commit -a -m "Schritt 4"` und `git push`

Erläuterung der Änderungen

- In **Zeile 4** wird mit `pinMode(2, INPUT)` der digitale Pin 2 als Eingang konfiguriert.
- In **Zeile 11 bis 13** seht ihr eine `if`-Anweisung. Die Bedingung prüft den Rückgabewert von `digitalRead(2)`. Diese Funktion liefert `1` (entspricht **HIGH** bzw. `true`), wenn an Pin 2 eine Spannung im Bereich von 5 V anliegt, und `0` (entspricht **LOW** bzw. `false`), wenn nahezu keine Spannung anliegt.
  Wenn der Rückgabewert 1 ist, wird in Zeile 12 das Zeichen `'A'` über die serielle Schnittstelle gesendet – inklusive Zeilenumbruch durch `Serial.println("A")`
- In **Zeile 15** wird eine lokale Variable `analogValue` vom Typ `int` deklariert und mit dem Rückgabewert von `analogRead(A0)` initialisiert. Pin A0 ist ein analoger Eingang. Eine Spannung zwischen 0 V und 5 V wird vom Mikrocontroller in einen digitalen Wert zwischen `0` und `1023` (also 10 Bit Auflösung) umgewandelt.
- Die `if`**-Anweisung in Zeile 16 bis 20** prüft, ob sich der neue analoge Wert `analogValue` vom bisherigen gespeicherten Wert `valueA0` unterscheidet. Nur wenn sich der Wert geändert hat, wird:
  - in **Zeile 17** der neue Wert in `valueA0` gespeichert,
  - in **Zeile 18–19** eine Ausgabe über die serielle Schnittstelle erzeugt (im Format `X <Wert>`).

  Bleibt der Wert unverändert, wird keine Ausgabe erzeugt. So wird unnötige Wiederholung vermieden.

Was ihr euch für heute merken solltet:

- Mit `pinMode(2, INPUT)` wird ein digitaler Pin als Eingang geschaltet.
- `digitalRead(2)` liefert `1`, wenn am Pin eine Spannung anliegt, und `0`, wenn nicht.
- Mit `analogRead(A0)` wird eine Spannung von 0 V bis 5 V in einen Wert von `0` bis `1023` umgewandelt.

Schritt 5: Finetuning

Das Verhalten des Mikrocontrollers soll verfeinert werden:

- **Wird der Taster gedrückt,** soll ein `'A'` über die serielle Schnittstelle gesendet werden.
- **Solange der Taster gedrückt bleibt,** darf kein weiteres Zeichen gesendet werden.
- **Wird der Taster losgelassen,** soll ein `'a'` gesendet werden.
- **Solange der Taster nicht gedrückt ist,** darf kein weiteres `'a'` gesendet werden.

Kurz gesagt: Die Zeichen `'A'` und `'a'` sollen nur bei einer Zustandsänderung gesendet werden – nicht beim Halten.

Aufgabe

1. Legt euch eine globale Variable `pressedA` vom Typ `bool` an. Damit sie global ist, muss sie **außerhalb einer Funktion** deklariert werden – z. B. nach Zeile 7:
   ```
   bool pressedA;
   ```
   Da globale Variablen in C/C++ automatisch mit dem Wert `0` initialisiert werden, ist `pressedA` anfangs `false`.
2. Ändert die `if`-Anweisung in `loop()` ab in

   ```
   if (!pressedA && digitalRead(2)) {
       Serial.println("A");
       pressedA = true;
   }
   ```
   Das `!` negiert den logischen Wert von `pressedA`, `&&` ist das logische Und. Mit dieser Abfrage wird nur dann ein `'A'` ausgegeben, wenn der Taster zuvor nicht gedrückt war.
   Gleichzeitig merken wir uns mit `pressedA = true`, dass der Taster jetzt gedrückt ist.
3. Übersetzt das Programm, flasht den Mikrocontroller und beobachtet die Ausgabe im seriellen Monitor. Mit dieser Änderung wird bereits ein Teil des gewünschten Verhaltens erreicht. Die Ausgabe von `'A'` beim Loslassen fehlt aber noch. Mit einer zweiten `if`-Anweisung und etwas Knobelei solltet ihr das Ziel vollständig erreichen.
4. **Wenn alles klappt:** `git commit -a -m "Schritt 5"` und `git push`

Schritt 6: Vom Hallo-Welt-Programm zur seriellen Schnittstelle

Wechselt in beiden Terminals ins Verzeichnis `python`. Das Verzeichnis liegt parallel zum aktuellen Verzeichnis `controller` – ihr gelangt dorthin mit:
`cd ..`
`cd python`
oder in einem Zug:
`cd ../python`

Dort findet ihr die Datei `serial-reader.py` mit folgendem Inhalt:

```
print("Hallo Welt")
```

Das ist noch ein klassisches Hallo-Welt-Programm. Führt es aus mit:

```
python serial-reader.py
```

Die Ausgabe sollte – wenig überraschend – einfach lauten: `Hallo Welt` (mit Zeilenumbruch).

Jetzt wagen wir einen gewaltigen Schritt nach vorne: Wir ändern das Python-Programm so ab, dass es die serielle Schnittstelle ausliest!

Ganz nebenbei lernen wir dabei (learning by doing) auch ein paar grundlegende Spracheigenschaften von Python kennen.

Ändert das Programm `serial-reader.py` wie folgt ab:

```
 1 import serial
 2 import time
 3
 4 ser = serial.Serial("/dev/ttyUSB0", 9600)
 5
 6 print("Warte auf serielle Eingaben...")
 7
 8 while True:
 9     while ser.in_waiting > 0:
10         ch = ser.read(1)
11         print(ch)
12     time.sleep(0.1)
```

Einrückung ist nicht verhandelbar

Python ist bei falscher Einrückung **genauso wenig tolerant wie ich** – nämlich **gar nicht**.

Im Gegensatz zu vielen anderen Programmiersprachen gibt es in Python **keine geschweiften Klammern**, um Schleifen- oder Funktionsblöcke abzugrenzen. Stattdessen erkennt der Python-Übersetzter – genau wie der Programmierer – **allein anhand der Einrückung**, welche Anweisungen zu welchem Block gehören.
In unserem Beispiel:
- Die äußere `while True:`-Schleife beginnt in **Zeile 8** und umfasst **Zeile 9 bis 12**.
- Die innere Schleife `while ser.in_waiting > 0:` beginnt in **Zeile 9** und umfasst nur die **Zeilen 10 und 11**.
- `time.sleep(0.1)` gehört nicht mehr zur inneren Schleife – das erkennt man allein daran, dass es auf gleicher Höhe wie die innere `while` eingerückt ist.

Aufgabe

Führt das Programm aus mit `serial-reader.py`

Ihr solltet eine Ausgabe sehen, die etwa so aussieht (notfalls am Potentiometer drehen oder den Taster drücken):

```
b'X'
b' '
b'4'
b'0'
b'4'
b'\r'
b'\n'
```

Beendet das Programm mit `Control-C`.
Aktualisiert euer Repository mit `git commit -a -m "Schritt 6"` und `git push`

Erklärung der Ausgabe

- Das Programm gibt jeweils ein empfangenes Byte pro Zeile aus.
- Die Daten stammen direkt von der seriellen Schnittstelle des Mikrocontrollers.
- In Python werden solche Bytes als sogenannte Byte-Objekte dargestellt – daran erkennt man das Präfix `b'...'`.
- Im Unterschied zu normalen Zeichenketten (Strings) enthält ein Byte-Objekt exakt ein Byte, wie zum Beispiel:
  - `b'X'` steht für das Byte mit dem ASCII-Wert `0x58`, also den Buchstaben `X`
  - `b' '` ist das Leerzeichen (`0x20`)
  - `b'\r'` ist das Steuerzeichen Carriage Return (`0x0D`)
  - `b'\n'` ist der Line Feed (`0x0A`)

  Diese beiden letzten Zeichen entstehen, wenn im Arduino-Programm `Serial.println(...)` verwendet wird – denn `println` sendet automatisch `\r\n` am Ende.

Technischer Hintergrund

Die Variable `ser`, die durch `serial.Serial(...)` entsteht, steht für eine spezielle Datenstruktur, die den Zugriff auf die serielle Schnittstelle verwaltet.
Dabei wird auch ein interner **Puffer** angelegt, in dem empfangene Bytes zwischengespeichert werden.

- Immer wenn über die serielle Schnittstelle ein Byte empfangen wird, landet es zunächst in diesem Puffer.
- Die Variable `ser.in_waiting` enthält stets die Anzahl der aktuell im Puffer gespeicherten Bytes.
- Mit `ser.read(1)` kann jeweils **das älteste Byte** aus dem Puffer gelesen werden.

Dieser Puffer funktioniert nach dem Prinzip einer **FIFO-Datenstruktur** *(First In, First Out)*:
Das zuerst empfangene Byte wird auch als erstes wieder gelesen.

Was du dir für heute merken solltest:

- Mit `import serial` wird das Modul pyserial eingebunden – es muss installiert sein.
- `ser = serial.Serial(...)` öffnet die serielle Schnittstelle (hier: `/dev/ttyUSB0`) mit der angegebenen Baudrate.
  Die weitere Kommunikation mit der Schnittstelle erfolgt über die Variable `ser`.
- `ser.read(1)` liest ein Byte von der Schnittstelle und gibt es als Byte-Objekt zurück.
- Die Bedingung `ser.in_waiting > 0` prüft, ob bereits Daten empfangen wurden.
- Der Präfix `b'...'` bedeutet in Python: Das ist ein Byte, kein String.
- `import time` lädt das eingebaute Modul `time` (muss nicht separat installiert werden).
- Mit `time.sleep(0.1)` wird eine Pause von einer Zehntelsekunde eingelegt – das entlastet den Prozessor und verhindert unnötige Schleifenaktivität.

Schritt 7: Hausaufgabe!

Mit Python kann man richtig tolle Dinge programmieren – man muss sich nur mit ein paar wenigen Grundlagen vertraut machen. Damit ihr fit dafür seid, gibt's jetzt eine kleine Hausaufgabe:
Wir haben an der Uni Lizenzen für ein interaktives Online-Tutorial:

👉 Online-Python-Brückenkurs (OPB)

Aufgabe bis zum nächsten Termin (in zwei Wochen):
- **Kapitel 1 bis 11** durcharbeiten
- Die **Challenges „Kapitel 1–7"** und **„Kapitel 8–11"** bestehen

Die Kapitel sind **kurz und machbar** – je ca. 5–10 Minuten. Also keine Panik!

Danach habt ihr ein gutes Verständnis für zentrale Sprachelemente in Python, z. B.:
- **Variablen** (Kapitel 4)
- **Zahlen** (Kapitel 5)
- **Zeichenketten (Strings)** (Kapitel 6)
- **Booleans** (Kapitel 7)
- **If-Anweisungen** (Kapitel 8)
- **Listen** (Kapitel 9)
- **For-Schleifen** (Kapitel 10)
- **While-Schleifen** (Kapitel 11)

Wenn euch das Auslesen der seriellen Schnittstelle noch nicht genug Motivation war – wartet einfach den nächsten chill-out-Schritt ab 😉

🎮 Chill-out-Schritt: Breakout – fast fertig…

Im Verzeichnis `python` findet ihr auch das Python-Programm `breakout.py`. Das ist noch nicht das fertige Spiel – sondern genau das, was ihr beim nächsten Mal selbst programmieren könnt, wenn ihr den Python-Onlinekurs fleißig bearbeitet habt. 😉

Jetzt aber: Einfach mal ausprobieren! Führt das Programm aus: `python breakout.py`

Aber danach wird aufgeräumt
Entfernt `breakout.py` aus eurem Repository – denn selber schreiben macht mehr Spaß!

```
git rm breakout.py
git commit -a -m "Fertig!"
git push
```

Beim nächsten Mal baut ihr das Spiel selbst Schritt für Schritt auf –
und versteht dabei genau, wie Python mit Hardware spricht.