Anwendungsorientierte Softwareentwicklung
Laborblatt 3: Blinkende LED - Bare-Metal-C

Lernziele

- Einführung in die Programmierung mit C/C++ statt Assembler
- Interpretation und Analyse des durch den C-Compiler erzeugten Assemblercodes
- Nutzung des Oszilloskops zur Analyse digitaler Ausgangssignale

Schritt 1: Arbeiten mit dem Terminal und Versuchsaufbau

Ziel: Anknüpfen an das letzte Labor: Wir rekapitulieren, wie wir Programme bearbeiten, kompilieren und auf den Mikrocontroller übertragen. Außerdem überprüfen wir die Blinkfrequenz der LED mit dem Oszilloskop.

Wie gewohnt arbeiten wir mit zwei Terminals nebeneinander – links zum Editieren, rechts zum Ausführen von Kommandos.

Linkes Terminal (Editieren)

A) **Git-Repository klonen**
Ladet die vorbereitete Code-Basis mit folgendem Befehl:
`git clone https://gitlab.uni-ulm.de/mlehn/avr-bare-metal-2.git`
Das Repository enthält u. a. die Datei `led_pin2_delay4.s` ,ein Assemblerprogramm, das eine an Pin 2 angeschlossene LED mit 1 Hz blinken lässt.

B) **Ins Unterverzeichnis wechseln und Datei in nano öffnen**
Wechselt in das Verzeichnis mit dem Assemblercode:
`cd avr-bare-metal-2/asm` und öffnet `led_pin2_delay4.s` in `nano`.

Rechtes Terminal (Kompilieren & Flashen)

A) **Ins selbe Verzeichnis wechseln und flashen**
Auch hier zunächst in Unterverzeichnis wechseln:
`cd avr-bare-metal-2/asm`
Schließt euren Mikrocontroller an und macht ihn mit `usb_check` für WSL sichtbar. Dann übersetzt und übertragt ihr den Code mit:
`make upload HEX=led_pin2_delay4.hex`

Aufgabe

1. Schließt eine LED mit geeignetem Vorwiderstand an Pin 2 an und beobachtet das Blinken.
2. Messt mit dem Oszilloskop, mit welcher Frequenz die LED tatsächlich blinkt.
3. Erklärt und demonstriert mindestens folgende Punkte:
   (a) Wie erreicht das Assemblerprogramm, dass Pin 2 mit 1 Hz oszilliert?
   (b) Mit welchem Werkzeug wird der Assemblercode in Maschinencode übersetzt?
   (c) Wie sieht der Maschinencode aus, der geflasht wird?
   (d) Mit welchem Werkzeug wird der Mikrocontroller geflasht?

Schritt 2: Modifikationen im Assemblerprogramm

Untersuchen, wie unterschiedliche Assembler-Instruktionen sich auf das Verhalten des Programms auswirken – insbesondere auf die Frequenz an Pin 2.
Ihr sollt das Programm schrittweise verändern und jeweils mit dem Oszilloskop überprüfen, ob die LED weiterhin mit 1 Hz blinkt.

Warum das wichtig ist:

- **Offiziell:** Ihr lernt weitere Instruktionen der AVR-Architektur kennen.

- **Inoffiziell:** Ihr erzeugt Varianten von Assemblercode, wie sie auch ein C-Compiler produzieren könnte. So seht ihr später mit eigenen Augen:
  - **C-Code kann in unterschiedlichen Assemblercode übersetzt werden.**
  - **Assembler wird in Maschinencode übersetzt** – und das eindeutig.
  - Fazit: C ist bequemer, aber weniger präzise als Assembler.

```
1          ldi     r19,    4           ; für Pin 2 (4 = 2^2)
2          out     0x0a,   r19
3          ldi     r18,    0
4  loop:   eor     r18,    r19         ; 1 Takt
5          out     0x0b,   r18         ; 1 Takt
6          ldi     r24,    0x54        ; 1 Takt
7          ldi     r25,    0x58        ; 1 Takt
8          ldi     r26,    0x14        ; 1 Takt
9          ldi     r27,    0x00        ; 1 Takt
10 delay:  subi    r24,    1           ; 1 Takt
11         sbci    r25,    0           ; 1 Takt
12         sbci    r26,    0           ; 1 Takt
13         sbci    r27,    0           ; 1 Takt
14         brne    delay               ; 2 Takte bei Sprung, 1 Takt sonst
15         rjmp    loop                ; 2 Takte
```

Aufgaben (nach jedem Schritt mit dem Oszilloskop nachmessen)

1. Ersetzt die folgenden zwei Zeilen:
   ```
   10 delay:  subi    r24,    1           ; 1 Takt
   11         sbci    r25,    0           ; 1 Takt
   ```
   durch diese eine Zeile:
   ```
   10         sbiw    r24,    1           ; 2 Takte
   ```
   Die Instruktion `sbiw` *(Subtract Immediate from Word)* behandelt den Registerverbund r25:r24 als 16-Bit-Wert und subtrahiert hier 1. Obwohl sie nicht schneller ist (ebenfalls 2 Takte), ist der Code dadurch 2 Byte kleiner – denn sbiw belegt nur eine Instruktion (2 Byte) statt zwei (2 × 2 Byte).

2. Ersetzt die folgenden zwei Zeilen:
   ```
   12         sbci    r26,    0           ; 1 Takt
   13         sbci    r27,    0           ; 1 Takt
   ```
   durch:
   ```
   12         sbc     r26,    r1          ; 1 Takt
   13         sbc     r27,    r1          ; 1 Takt
   ```
   Die Instruktion `sbc` *(Subtract with Carry)* funktioniert analog zu `sbci` *(Subtract Immediate with Carry)* nur dass hier der zweite Operand ein Register ist statt eines unmittelbaren Werts. Das Register `r1` wird bei jedem Reset automatisch mit 0 initialisiert. Solange man es nicht selbst verändert, enthält es daher immer den Wert 0.
   Fazit: Funktional und hinsichtlich Länge und Taktzyklen ist der Code identisch, aber ihr lernt eine alternative Möglichkeit kennen, das Carry-Bit zu subtrahieren.

3. Ersetzt die folgenden drei Zeilen:
   ```
   1          ldi     r19,    4           ; für Pin 2 (4 = 2^2)
   2          out     0x0a,   r19
   3          ldi     r18,    0
   ```
   durch:
   ```
   1          ldi     r24,    4           ; für Pin 2 (4 = 2^2)
   2          out     0x0a,   r24
   3          ldi     r18,    0
   4          ldi     r19,    4
   ```
   Das Programm funktioniert wie vorher – erkläre wieso diese Änderung aber eigentlich keine gute Idee ist.

Schritt 3: Bare-Metal-C

Im Unterverzeichnis `c` des Repositories findet ihr C-Programme für den Mikrocontroller.
Um vom aktuellen Verzeichnis `asm` dorthin zu gelangen, müsst ihr „ein Verzeichnis hoch und dann eines runter“:
`cd ..`
`cd c`
Oder direkt in einem Rutsch mit: `cd ../c`

Öffnet im linken Terminal die Datei `led_pin2_simple.c`:

```
 1 static void setPinModes(unsigned char mode)
 2 {
 3     *(volatile unsigned char *)(0x2A) = mode;
 4 }
 5
 6 static void digitalWriteAll(unsigned char val)
 7 {
 8     *(volatile unsigned char *)(0x2B) = val;
 9 }
10
11 static void delay_cycles(unsigned long count)
12 {
13     do {
14         // Compiler soll für count register verwenden
15         asm volatile("" : "+r"(count));
16         count = count - 1;
17     } while (count != 0);
18 }
19
20 int main()
21 {
22     setPinModes(4);
23     digitalWriteAll(4);
24     while (1) {
25     }
26 }
```

Ein paar Worte zu dem, was ihr da seht:

- Im oberen Teil (Zeilen 1 bis 18) werden drei Funktionen definiert:
  `setPinModes`, `digitalWriteAll` und `delay_cycles`.
  Die Implementierung dieser Funktionen müsst ihr nicht im Detail verstehen – sie wirkt auf den ersten Blick vielleicht sogar abschreckend, selbst für hartgesottene C-Programmierer. Aber natürlich erkläre ich euch die Details gern, wenn ihr neugierig seid.
- Im unteren Teil (Zeilen 20 bis 26) seht ihr die Funktion `main`. Wenn das Programm startet, werden die Anweisungen in den geschweiften Klammern der Reihe nach von oben nach unten ausgeführt.
  Hier werden zum Beispiel `setPinModes(4)` und `digitalWriteAll(4)` aufgerufen – Funktionen, mit denen man die Pins 0 bis 7 manipulieren kann.
  Was konkret passiert:
  - `setPinModes(4)` setzt das Datenrichtungsregister so, dass nur Pin 2 als Ausgang konfiguriert wird. (4 entspricht `00000100` in binärer Darstellung, d. h. nur Bit 2 ist gesetzt.)
  - `digitalWriteAll(4)` setzt dann Pin 2 auf HIGH, weil auch hier nur Bit 2 im Argument gesetzt ist. Die anderen Pins bleiben auf LOW.

- Nach dem Setzen des Ausgangspins mit `digitalWriteAll(4)` soll der Mikrocontroller einfach nichts weiter tun – er soll den gesetzten Zustand dauerhaft beibehalten.
  Die Schleife

```
24      while (1) {
25      }
```

  sorgt dafür, dass das Programm nicht endet, sondern in einer Endlosschleife hängen bleibt. Das ist bei Mikrocontrollern üblich: Ein Programm läuft meist so lange, wie das Gerät mit Strom versorgt wird – und `main()` soll daher niemals zurückkehren, also nie beendet werden.

Aufgabe

1. **Kompilieren und Flashen**
   Auch in diesem Verzeichnis (c) befindet sich ein Makefile, das wie gewohnt verwendet werden kann. Führt im rechten Terminal folgendes Kommando aus, um das Programm zu übersetzen und auf den Mikrocontroller zu flashen: `make upload HEX=led_pin2_simple.hex`
   a) Schließt eine LED über einen geeigneten Vorwiderstand an Pin 2 an und überzeugt euch, dass sie leuchtet.
   b) Achtet auf die Ausgabe von `make`. Erklärt uns, was sich in der Toolchain geändert hat.
2. **Automatisch erzeugten Assembler analysieren**
   Übersetzt mit `make led_pin2_simple.s` das C-Programm in Assemblercode. Zeigt euch den Inhalt mit: `cat led_pin2_simple.s`. Die erzeugte Datei enthält viele Zusatzinformationen (z. B. Dateiname, Compiler-Version, Kommentare), die für uns nicht relevant sind. Relevant sind nur wenige Zeilen, etwa diese hier:

```
        ldi     r24,    lo(4)
        out     0xa,    r24
        out     0xb,    r24
.L2:
        rjmp    .L2
```

   (Das lo(8) steht hier einfach für die Konstante 8. Man kann auch direkt ldi r24, 8 schreiben.)
3. **Eigene Version schreiben und vergleichen**
   Erstellt im linken Terminal die Datei check.s mit folgendem Inhalt:

```
1         ldi     r24,    4
2         out     0xa,    r24
3         out     0xb,    r24
4 halt:   rjmp    halt
```

   Erzeugt daraus Maschinencode mit `make check.hex`
   a) Vergleicht den erzeugten Maschinencode mit dem aus dem C-Programm:
   `cat led_pin2_simple.hex`
   `cat check.hex`
   Oder nutzt das `diff`-Kommando: `diff check.hex led_pin2_simple.hex`
   Wenn keine Unterschiede gefunden werden, bleibt die Ausgabe leer – sonst zeigt `diff` die abweichenden Zeilen.
   b) Ändert in `check.s` die erste Zeile:

```
1         ldi     r24,    4
```

   mit

```
1         ldi     r24,    5
```

   Übersetzt erneut mit: `make check.hex` und vergleicht wieder mit dem `diff`-Kommando. Jetzt sollte ein Unterschied angezeigt werden – überprüft, an welcher Stelle sich die erzeugten .hex-Dateien unterscheiden.

Schritt 4: Eleganter blinken mit exklusivem Oder

Ändert die Funktion `main()` wie folgt ab:

```
20 int main()
21 {
22     setPinModes(4);
23     unsigned char pinValues = 4;
24     while (1) {
25         pinValues = pinValues ^ 4;
26         digitalWriteAll(pinValues);
27     }
28 }
```

Aufgaben

1. **Test des geänderten Programms**
   Bevor wir genauer analysieren, warum das Programm funktioniert, testen wir zunächst ob es funktioniert:
   - Übersetzt das Programm wie gewohnt mit dem Makefile und Flasht den Mikrocontroller mit: `make upload HEX=led_pin2_simple.hex`
   - Beobachtet, was die LED an Pin 2 tut.
   - Messt mit dem Oszilloskop, wie sich das Signal an Pin 2 verhält.
2. **Eigene Version schreiben und vergleichen**
   Übersetzt das C-Programm mit `make led_pin2_simple.s` Dadurch wird der vom Compiler erzeugte Assemblercode generiert. Zeigt den Inhalt an mit: `cat led_pin2_simple.s`
   Der für uns relevante Teil des erzeugten Codes ist äquivalent zu:

   ```
   1         ldi     r24,    4
   2         out     0xa,    r24
   3         ldi     r25,    4
   4 loop:   eor     r24,    r25
   5         out     0xb,    r24
   6         rjmp    loop
   ```

   Erstellt im linken Terminal die Datei `check.s` mit genau diesem Inhalt. Übersetzt beide Programme mit: `make led_pin2_simple.hex check.hex` und vergleicht die erzeugten Maschinencodes mit `diff`.

Kurze Erklärung des C-Codes

In Zeile 23 wird mit

```
23     unsigned char pinValues = 4;
```

eine **lokale** Variable vom Typ `unsigned char` definiert und mit 4 initialisiert:

- Der Typ `unsigned char` legt fest, dass die Variable genau ein Byte belegt.
- Da pinValues innerhalb der Funktion `main` (ohne spezielle Zusatzschlüsselwörter) definiert wurde, handelt es sich um eine lokale Variable mit automatischer Lebensdauer: Sie existiert nur während der Ausführung der Funktion und ist nach deren Ende nicht mehr gültig.

  Der Compiler darf solche **lokalen** Variablen in Register legen, was besonders effizient ist. Wird eine Variable dagegen außerhalb aller Funktionen definiert, ist sie **global** und benötigt dauerhaft Speicherplatz im RAM.

In Zeile 25 wird mit

```
25          pinValues = pinValues ^ 4;
```

wird der aktuelle Wert von pinValues durch dessen Exklusive-Oder—Verknüpfung mit 4 ersetzt. Wie bekannt, hat das die Wirkung, dass Bit 2 umgeschaltet (geflippt) wird:

- Wenn Bit 2 zuvor 1 war → wird es 0,
- wenn es 0 war → wird es 1.

So entsteht der Blinkeffekt.

Schritt 5: Beruhigender Blinken mit 1Hz

Prima – wir haben jetzt wieder ein Programm, das **Pin 2 abwechselnd für ca. 250 ns auf HIGH und dann auf LOW** schaltet.
Das Programm ist diesmal jedoch in **C geschrieben**, wird vom **C-Compiler in Assemblercode** übersetzt und anschließend vom **Assembler in Maschinencode**. Auf umständliche Weise haben wir also das Gleiche erreicht wie zuvor – aber mit moderneren Mitteln.

Auch wenn der C-Code (selbst main() allein) für den Laien zunächst kryptisch aussieht, lässt sich erahnen: Es ist deutlich **einfacher, die Regeln von C zu lernen**, als direkt in Assembler oder gar Maschinencode zu programmieren – besonders bei größeren Projekten.

Im letzten Beispiel soll die LED wieder mit 1 Hz blinken. Dazu bauen wir an der richtigen Stelle eine Verzögerung in die Schleife ein, die jeweils eine halbe Sekunde wartet – so ergibt sich ein kompletter Zyklus von einer Sekunde (1 Hz). Die benötigte Anzahl an Taktzyklen lässt sich berechnen:

$$n = \frac{\frac{500000000}{62.5} - 7}{6} = 1333332 \text{ Takte}$$

Ändert im C-Programm die Funktion main() wie folgt ab:

```
20 int main()
21 {
22     // setup
23     unsigned char pin = 2;
24     unsigned char pinModes = 1 << pin;
25     setPinModes(pinModes);
26 
27     unsigned char pinValues = 0;
28     while (1) {
29         // loop
30         pinValues ^= 1 << pin;
31         digitalWriteAll(pinValues);
32         delay_cycles((500000000 / 62.5 - 7) / 6);
33     }
34 }
```

Aufgabe

1. Lasst die LED mit 1 Hz blinken.
2. Führt `diff led_pin2_simple.hex ../asm/led_pin2_delay4.hex` aus und überzeugt euch, dass der Maschinencode identisch ist mit dem, was wir in Schritt 2 in Assembler formuliert hatten.