

Blinkende LED – ein Blick unter die Haube

1 Ziel dieses Projektabschnitts

Viele arbeiten beim Programmieren eines Mikrocontrollers mit der Arduino-IDE. Dort reicht es, den C/C++-Code einzutippen und auf einen Button zu klicken – der Code wird kompiliert und direkt auf den Mikrocontroller übertragen (geflasht). Einfach und bequem.

In diesem Projektabschnitt wollen wir verstehen, was genau passiert, wenn man auf diesen Knopf drückt. Wir lassen die Arduino-IDE beiseite und schreiben den Maschinencode direkt selbst. Das heißt: Wir erstellen die Programmdatei per Hand – wie echte Hacker oder Ingenieure, die wissen wollen, was wirklich läuft.

Wir beschäftigen uns dabei zwar noch nicht mit Assembler – das kommt später. Statt dessen schreiben wir den Maschinencode direkt, also Byte für Byte. Das ist etwas, das man als Techniker oder Ingenieur einmal im Leben gemacht haben sollte – schon allein, um zu verstehen, was eine höhere Programmiersprache überhaupt leistet. Die eigentliche Herausforderung ist dabei nicht der Maschinencode selbst, sondern der Umgang mit dem Terminal:

- Wie schreibt man eine Datei im Editor `nano`?
- Wie organisiert man sich Verzeichnisse?
- Wie navigiert man im Terminal und lässt sich Dateien anzeigen?
- Und wie überträgt man schließlich mit `avrdude` den Maschinencode auf den Mikrocontroller?

All das sind wichtige Grundlagen – und nach diesem Abschnitt könnt ihr von euch behaupten: Ich habe ein Mikrocontrollerprogramm direkt in Maschinencode geschrieben und selbst geflasht.

Aber das ist noch nicht alles. Das praktische Ziel wird es sein, eine LED blinken zu lassen. Genauer gesagt soll sie abwechselnd 250 ns an und 250 ns aus sein. Mit dem bloßen Auge kann man das nicht mehr sehen – wir werden das deshalb mit dem Oszilloskop nachmessen. Die Periodendauer beträgt dabei

$$T = 250 \text{ ns} + 250 \text{ ns} = 500 \text{ ns},$$

was einer Frequenz von

$$f = \frac{1}{T} = \frac{1}{500 \text{ ns}} = 2 \text{ MHz}$$

entspricht.

Bei unserem ersten Experiment werden wir zwar sehen, dass wir ein Signal erzeugen, bei dem die LED an- und ausgeschaltet wird, es wird aber nicht symmetrisch sein. Das

ist aber nicht schlimm – wir werden bei der Gelegenheit eben doch etwas mehr darüber lernen, wie der Maschinencode ausgeführt wird.

Und weil das Schreiben von Maschinencode zwar nicht schwer, aber mühsam ist, wechseln wir anschließend doch auf eine höhere Sprache – aber noch nicht C/C++, sondern zunächst auf Assembler. Mit einem Assembler (**avr-as**) kann man den Code in Maschinencode übersetzen. Im Vergleich zu C/C++ sieht man hier sofort, dass man eigentlich exakt dasselbe wie im Maschinencode macht, nur komfortabler.¹

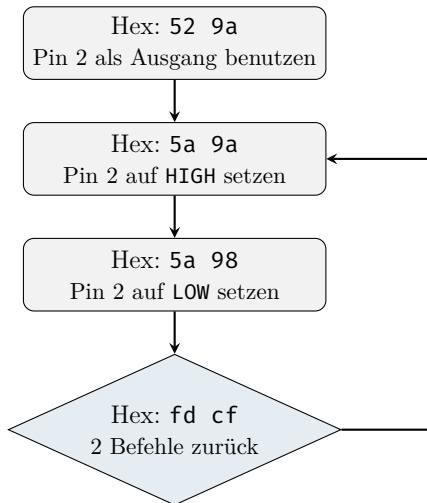
¹Analog ist aber auch C/C++ nur eine komfortable und portable Weise, um Assemblercode kompakt auszudrücken.

2 Einfaches Programm zur Ansteuerung einer LED

Um eine LED schnell blinken zu lassen, benötigen wir nur vier Maschinenbefehle:

- einen, um den gewünschten Pin des Mikroprozessors als Ausgang zu konfigurieren,
- einen, um diesen Pin auf **HIGH** zu setzen (also Spannung anzulegen),
- einen, um ihn auf **LOW** zu setzen (also die Spannung wieder wegzunehmen),
- und einen sogenannten *Sprungbefehl*, mit dem die Programmausführung an einer anderen Stelle fortgesetzt werden kann.

Jeder dieser Befehle besteht aus zwei Bytes und kann daher mit vier Hexadezimalziffern dargestellt werden. Die Befehle werden nacheinander (sequenziell) abgearbeitet – wie im folgenden Ablaufdiagramm dargestellt:



Schreibt man die Hexadezimalwerte der Befehle nacheinander auf, ergibt sich unser erstes vollständiges Programm in Maschinencode:

52 9a 5a 9a 5a 98 fd cf

Jede Hex-Ziffer steht dabei für genau vier Bits. Die vollständige Bitbedeutung kann man in [Tabelle 1](#) nachschlagen. In binärer Darstellung sieht unser Programm dann so aus:

```
01010010 10011010 01011010 10011010 01011010 10011000 11111101  
11001111
```

Das Programm ist nun vollständig entworfen. Es fehlt nur noch ein letzter Schritt: Es muss in den Programmspeicher des Mikroprozessors geschrieben werden – also in den sogenannten *Flash*-Speicher.

Im nächsten Abschnitt wird erklärt, wie man den Maschinencode in Hexadezimaldarstellung in eine Datei schreibt und mit dem Programm `avrdude` in den Flash-Speicher des Mikroprozessors überträgt.

Was heute mit einem einzigen Befehl erledigt ist, war bei historischen Computern und Mikroprozessoren der 1950er bis 1970er Jahre deutlich aufwändiger: Programme wurden

Tabelle 1: Darstellung aller 4-Bit-Zahlen in dezimaler, hexadezimaler und binärer Form

Dezimal	Hexadezimal	Binär
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

oft bitweise geladen, indem für jedes einzelne Bit (binary digit, also Binärziffer) ein Kippschalter umgelegt wurde.² Alternativ wurden Programme auf Lochkarten gestanzt und mechanisch eingelesen – was im Prinzip ebenfalls nichts anderes bedeutete, als viele Bits in Folge einzugeben, nur eben etwas bequemer. Man kann sich diese alten Verfahren heute als Gedankenmodell gut vorstellen, um besser zu verstehen, was das Programm `avrduude` im Hintergrund eigentlich für uns erledigt.

²Das Eingeben von Programmen per Kippschalter war in den 1970er Jahren ganz normal – vom Altair 8800 im Hobbykeller bis zu den legendären Supercomputern von Seymour Cray, dem Pionier des Hochleistungsrechnens. Beim Altair musste man den Bootloader Bit für Bit über das Frontpanel eintippen, LEDs zeigten den aktuellen Speicherinhalt. Auch Cray setzte bei seinen frühen Maschinen den ersten Code per Hand, bevor Magnetbandlaufwerke größere Programme nachluden.

3 Das Programm in den Mikroprozessor übertragen (den Mikroprozessor „flashen“)

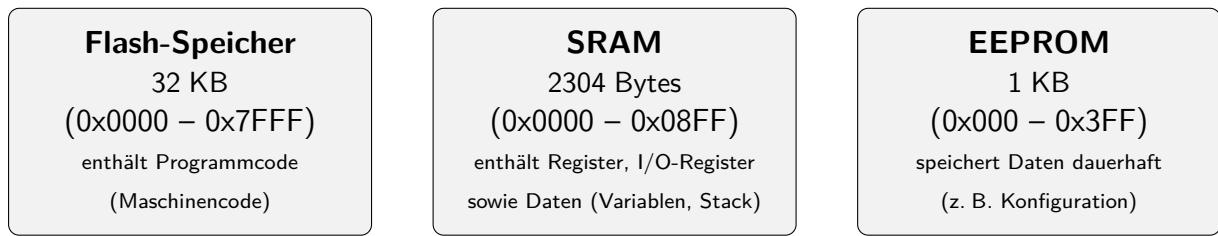


Abbildung 1: Speicheraufbau des ATmega328P: Der Flash enthält das Programm, der SRAM enthält Register, I/O-Bereiche und Daten. Der EEPROM speichert nicht-flüchtige Nutzdaten.

Unser Mikroprozessor verfügt über drei verschiedene Arten von Speicher ([Abbildung 1](#)):

- Einen Speicher für das Programm – den sogenannten *Flash*. Dort bleiben die Daten auch erhalten, wenn man den Stecker zieht.
- Einen *flüchtigen Speicher* (RAM), der während der Programmausführung zur Verfügung steht. Die Daten darin gehen verloren, sobald der Strom weg ist.
- Einen sogenannten *EEPROM* (Electrically Erasable Programmable Read-Only Memory), in dem man dauerhaft kleine Datenmengen speichern kann – z. B. Konfigurationen oder Zustände, die auch nach dem Ausschalten erhalten sollen.

Für uns ist zunächst nur der Flash-Speicher wichtig – in ihn muss unser Programm übertragen werden. Allen Speicherarten ist gemeinsam: Sie bestehen aus Arrays von Speicherzellen. In jeder Zelle kann genau ein Byte gespeichert werden, also 8 Bits. Diese 8 Bits lassen sich kompakt durch zwei Hexadezimalziffern darstellen. Die Speicherzellen sind durchnummiert – diese Nummer nennt man *Adresse*. Die erste Adresse hat immer den Wert 0. Unser Ziel ist es, dass der Programmspeicher nach dem Flashen so aussieht:

Adressen:	0	1	2	3	4	5	6	7	8	9	10	11
Speicherzellen:	52	9a	5a	9a	5a	98	fd	cf				

Beginnend bei Adresse 0 sollen die einzelnen Bytes also nacheinander im Speicher liegen. Sobald das gelungen ist, kann die Programmausführung starten – sie beginnt automatisch mit dem Befehl, der an Adresse 0 gespeichert ist.

Nachdem ein Befehl ausgeführt wurde, wird typischerweise der Befehl ausgeführt, der zwei Adressen weiter im Speicher liegt – denn bei unserem Mikroprozessor besteht jeder Befehl genau aus zwei Bytes. Eine Ausnahme bilden Sprungbefehle: Sie können explizit festlegen, dass die Ausführung an einer anderen Adresse fortgesetzt wird – entweder vorwärts (einige Befehle überspringend) oder rückwärts (zurückspringend).

3.1 Intel Hex Format

Ein Werkzeug, das den Maschinencode in den Flash-Speicher überträgt, muss mindestens zwei Dinge über die Daten wissen, die geschrieben werden sollen:

1. Wie viele Bytes insgesamt übertragen werden müssen.
2. An welche Adresse im Flash-Speicher das erste Byte geschrieben werden soll. Die folgenden Bytes werden dann automatisch an die nachfolgenden Adressen abgelegt.

Diese Informationen müssen in geeigneter Form bereitgestellt werden – typischerweise geschieht das durch eine spezielle Datei, in der der Maschinencode und die Speicheradressen kodiert sind. Ein gängiges Format dafür ist das sogenannte *Intel Hex Format*, das von vielen Programmen – darunter auch **avrdude** – verstanden wird.

Unser Maschinencode kann wie folgt in eine Datei **led.hex** verpackt werden, die dieses Format benutzt:

```
:08000000529A5A9A5A98FDCF5A  
:00000001FF
```

Listing 1: Datei *led_pin2.hex*

Die Datei **led.hex** enthält in diesem Fall zwei sogenannte *Datensätze*, jeweils als eine Zeile dargestellt. Jeder Datensatz beginnt mit einem Doppelpunkt (:) und besteht aus einer Reihe von Hexadezimalziffern, die in Gruppen interpretiert werden:

Datenzeile ist ein sogenannter *Daten-Datensatz*, der die eigentlichen Maschinencode-Bytes enthält festlegt, an welcher Adresse im Speicher sie abgelegt werden sollen.

:	08	0000	00	529A5A9A5A98FDCF	5A
Länge der Datenbytes:	08				
Startadresse im Speicher:	0000				
Record-Typ:	00 (Daten)				
Datenbytes:	52 9A 5A 9A 5A 98 FD CF				
Prüfsumme:	5A				

Hier also noch einmal die Bedeutung der einzelnen Abschnitte im Detail:

- **08** gibt die Anzahl der Datenbytes in diesem Datensatz an (hier: 8 Bytes).
- **0000** ist die Startadresse im Flash-Speicher, an der die folgenden Daten abgelegt werden sollen (in diesem Fall Adresse 0).
- **00** ist der Typ des Datensatzes. **00** bedeutet „normaler Datenblock“.
- **52 9A 5A 9A 5A 98 FD CF** sind die 8 Bytes Maschinencode, die geschrieben werden sollen.
- **5A** ist eine Prüfsumme (Checksumme) zur Fehlererkennung.

End-of-File-Zeile ist ein sogenannter *End-of-File-Datensatz*. Er signalisiert dem Flash-Werkzeug, dass keine weiteren Daten folgen. Die Struktur ist ähnlich wie beim vorherigen Datensatz:

```
: 00 0000 01 FF
```

Länge der Datenbytes:	00
Startadresse im Speicher:	0000 (nicht relevant)
Record-Typ:	01 (Ende)
Datenbytes:	52 9A 5A 9A 5A 98 FD CF
Prüfsumme:	5A

Die Bedeutung der einzelnen Teile zusammengefasst:

- **00** — Es gibt keine Datenbytes in diesem Datensatz.
- **0000** — Startadresse (nicht von Bedeutung, da keine Daten folgen).
- **01** — Kennzeichnet diesen Datensatz als End-of-File.
- **FF** — Die Prüfsumme, die auch hier zur Kontrolle dient.

Dieses Format ist sehr kompakt und für Maschinen leicht zu verarbeiten. Programme wie `avrdude` lesen diese Datei und schreiben die enthaltenen Daten automatisch an die angegebenen Speicheradressen im Flash-Speicher des Mikroprozessors.

3.2 Intel Hex Format: Prüfsumme berechnen

Die Prüfsumme dient der Fehlererkennung. Der Mikroprozessor (oder das Flash-Werkzeug) erhält pro Datensatz zunächst die Information, wie viele Datenbytes enthalten sind, gefolgt von den eigentlichen Bytes – und am Ende die Prüfsumme. Nach dem Empfang kann die Prüfsumme erneut berechnet und mit der übertragenen verglichen werden. Stimmt sie nicht überein, ist klar: Bei der Übertragung ist ein Fehler aufgetreten – entweder bei den Daten selbst oder bei der Prüfsumme.

Stimmt die Prüfsumme hingegen mit der berechneten überein, bedeutet das *nicht*, dass garantiert kein Fehler aufgetreten ist – nur, dass keiner entdeckt wurde. Die Prüfsumme reduziert also lediglich die Wahrscheinlichkeit, dass ein Übertragungsfehler unbemerkt bleibt.

Wie bei allen Prüfsummenverfahren muss man hier einen Kompromiss eingehen:

- Soll die Wahrscheinlichkeit für unerkannte Fehler möglichst gering sein, dann wird die Prüfsumme meist komplexer (z. B. durch längere oder kryptographische Verfahren).
- Soll die Prüfsumme hingegen besonders einfach und schnell berechnet werden (wie im Intel-Hex-Format), dann steigt die Wahrscheinlichkeit, dass bestimmte Fehler nicht erkannt werden – etwa wenn sich zwei fehlerhafte Bytes gerade gegenseitig „ausgleichen“.

Die Prüfsumme im Intel-Hex-Format ist ein guter Kompromiss: sehr einfach zu berechnen, aber ausreichend zuverlässig für typische Flash-Vorgänge über stabile Verbindungen³.

Berechnung der Prüfsumme im Intel-Hex-Format

Beim Intel-Hex-Format wird eine sogenannte *Zweierkomplement-Prüfsumme* verwendet. Einfach ausgedrückt: Zunächst werden alle Bytes (bis auf die Prüfsumme selbst) addiert, und zwar mit einem 8-Bit-Addierer – das heißt, es bleiben nur die unteren 8 Bits der Summe erhalten. Die Prüfsumme ist dann genau die Zahl, die man zu dieser Summe addieren müsste, damit das Ergebnis (bei 8 Bit gerechnet) wieder 0 ist.

Wer mit Formalismus prahlen möchte: Für $a, b \in B = \{0, \dots, 2^8 - 1\} = \{0, \dots, 255\}$ definiert man die 8-Bit-Addition durch $a \oplus b := (a + b) \bmod 2^8$. Für die n Bytes b_1, \dots, b_n berechnet man dann zunächst

$$s = b_1 \oplus b_2 \oplus \dots \oplus b_n.$$

Bezüglich \oplus ist die Menge B eine endliche abelsche Gruppe. Daher existiert zu jedem $s \in B$ ein inverses Element $-s$, und dieses $-s$ ist die gesuchte Prüfsumme p . Damit gilt:

$$p = -s \Leftrightarrow p \oplus s = 0,$$

also ist p genau die Zahl, die man zur Summe s addieren muss, um (modulo 2^8) wieder 0 zu erhalten.

Bei einer Beispielrechnung von Hand kann man direkt in der Hexadezimaldarstellung rechnen. Aus einer 8-stelligen Binärrechnung wird dadurch eine deutlich kompaktere, 2-stellige Hexadezimalrechnung.

Um Formulierungen wie etwa „die Hexadezimaldarstellung **19**“ zu vereinfachen, verwenden wir ab jetzt folgende Konvention: Ein Präfix **0x** kennzeichnet, dass die darauf folgenden Ziffern im Hexadezimalsystem zu lesen sind.

Also bedeutet **0x19** die Zahl 25 im Dezimalsystem, denn:

$$0x19 = 1 \cdot 16 + 9 = 25.$$

Angenommen, wir wollen für die drei Bytes

$$\begin{array}{ccc} 0xD8 & 0xB5 & 0x19 \end{array}$$

die Prüfsumme berechnen, dann gehen wir wie folgt vor:

1. Zunächst berechnen wir die Summe der drei Bytes mit 8-Bit-Addition:

$$s = (0xD8 \oplus 0xB5) \oplus 0x19$$

Die Addition erfolgt schrittweise. Zuerst:

$$\begin{array}{r} 0xD8 \\ + 0xB5 \\ \hline 0x18D \end{array}$$

³Für Hochsicherheitsübertragungen (z. B. im Netzwerk oder in Dateisystemen) werden statt einfacher Prüfsummen häufig CRCs (zyklische Redundanzprüfungen) oder kryptografische Hashes verwendet.

Da nur mit einem 8-Bit-Addierer gerechnet wird, schneiden wir das Ergebnis auf die unteren 8 Bit ab:

$$0xD8 \oplus 0xB5 = (0xD8 + 0xB5) \bmod 2^8 = 0x18D \bmod 2^8 = 0x8D$$

Dann:

$$\begin{array}{r} 0x8D \\ + 0x19 \\ \hline 0xA6 \end{array}$$

Insgesamt ergibt sich also $s = 0xA6$.

2. Die Prüfsumme p erhält man, indem man s von $2^8 = 0x100$ subtrahiert und das Ergebnis gegebenenfalls auf zwei Stellen (d. h. 8 Bit) kürzt. Das Abschneiden ist nur im Sonderfall $s = 0$ relevant, denn dann ergibt die Rechnung $p = 0x100$, aber korrekt ist natürlich $0x00$.

$$\begin{array}{r} 0x100 \\ - 0xA6 \\ \hline 0x5A \end{array}$$

Diese Berechnung von $p = (2^8 - s) \bmod 2^8$ entspricht exakt dem bekannten Verfahren zur Bildung des *Zweierkomplements*: Man invertiert das Bitmuster von s und addiert anschließend 1.

Auch das lässt sich für unser Beispiel nicht nur gut nachvollziehen, sondern sogar herleiten: Wir versuchen die Zahl zu bestimmen, die man auf $0xA6$ addieren muss, um $0x100$ zu erhalten. Dabei führen wir auch ganz nebenbei das Präfix **b** ein, um eine Binärdarstellung zu kennzeichnen. Es gilt:

$$s = 0xA6 = b10100110 \Rightarrow \bar{s} = b01011001 = 0x59$$

Dabei gilt stets $s + \bar{s} = 2^8 - 1 = 0xFF$, wie man in diesem Fall leicht nachrechnen kann:

$$\begin{array}{r} b10100110 \\ + b01011001 \\ \hline b11111111 \end{array}$$

Daraus ergibt sich:

$$s + \bar{s} = 2^8 - 1 \Rightarrow s + (\bar{s} + 1) = 2^8 \Rightarrow s \oplus (\bar{s} \oplus 1) = 0$$

Mit anderen Worten: $\bar{s} \oplus 1$ ist genau die Zahl, die man bei 8-Bit-Rechnung zu s addieren muss, um 0 zu erhalten – und damit ist

$$p = \bar{s} \oplus 1 = 0x59 \oplus 1 = 0x5A.$$

3.3 Mikrocontroller flashen mit **avrdude**

Mit dem Befehl `ls /dev/ttyUSB*` kann man unter Linux-Systemen die Geräte auflisten, die als serielle Schnittstellen erkannt werden – etwa ein über USB angeschlossener Mikrocontroller. Der gefundene Gerätename wird später beim Einsatz von **avrdude** benötigt, um den Zielpunkt korrekt anzugeben. Typischerweise erhält man eine Ausgabe wie:

```
/dev/ttyUSB0
```

Hinweis

Wenn mehrere Geräte angeschlossen sind, kann es hilfreich sein, den Befehl `ls /dev/ttyUSB*` einmal vor und einmal nach dem Anstecken des Arduino auszuführen, um den richtigen Gerätenamen zu identifizieren.

Mit dem Programm **avrdude** kann man anschließend das in `led.hex` gespeicherte Maschinenprogramm in den Flash-Speicher des Mikrocontrollers übertragen. Der vollständige Befehl sieht etwa so aus:

```
avrdude -c arduino -P /dev/ttyUSB0 -p atmega328p -U flash:w:led.hex:i
```

Dabei stehen die Optionen für:

- **-c arduino**

Gibt an, welcher Programmer verwendet wird. In diesem Fall wird der Arduino-Bootloader angesprochen, der über die serielle Schnittstelle programmiert werden kann.

- **-P /dev/ttyUSB0**

Legt fest, über welches Gerät die Kommunikation erfolgt. Dies ist der Name der seriellen Schnittstelle, die zuvor mit `ls /dev/ttyUSB*` ermittelt wurde.

- **-p atmega328p**

Gibt den Ziel-Prozessor an, hier der ATmega328P (wie er auf einem Arduino Uno verbaut ist).

- **-U flash:w:led.hex:i**

Bedeutet: Der Flash-Speicher (`flash`) soll beschrieben (`w` = write) werden, und zwar mit den Daten aus der Datei `led.hex`. Das `i` am Ende steht für das Dateiformat – hier: Intel Hex.

4 Nachmessen, was das Programm macht

Sobald das Programm in den Flash-Speicher geschrieben wurde, beginnt der Mikrocontroller automatisch mit der Ausführung.

Schließt man eine LED an Pin 2 an, leuchtet sie – allerdings ist mit bloßem Auge nicht zu erkennen, dass sie extrem schnell blinkt. Selbst ein minimales Flackern oder eine wahrnehmbare Helligkeitsveränderung ist nicht sichtbar.

Um dennoch zu überprüfen, was tatsächlich geschieht, verwenden wir im Labor ein Oszilloskop. Abbildung 2 zeigt die Messung des Signals an Pin 2.

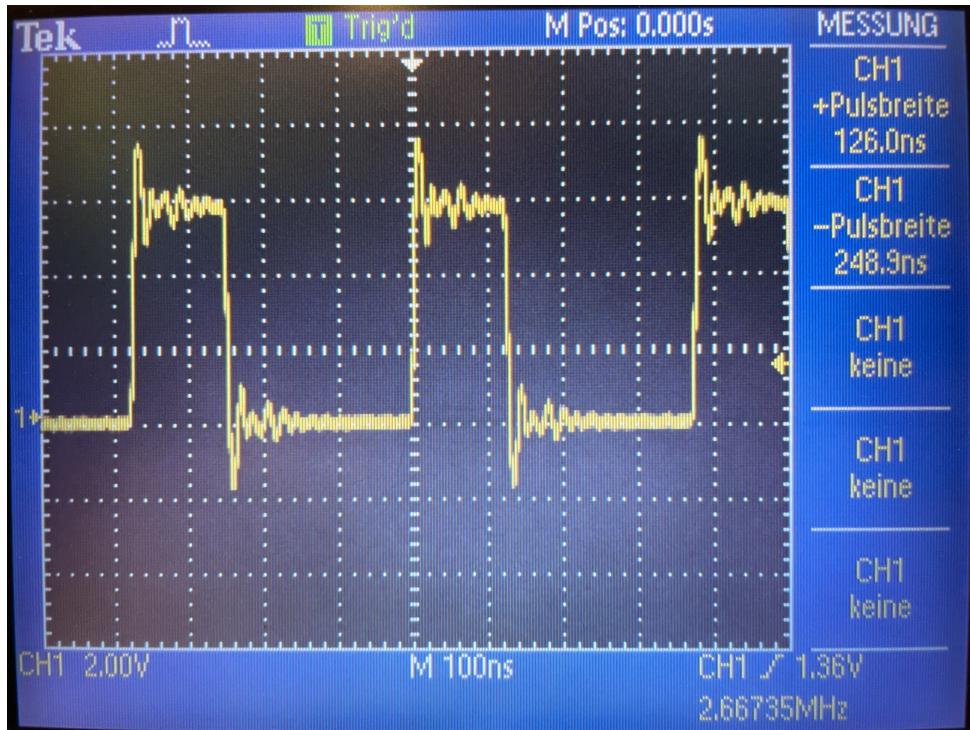


Abbildung 2: Oszilloskop-Messung an Pin 2 während der Ausführung von led.hex. Die HIGH-Phase dauert etwa 125 ns, die LOW-Phase etwa 250 ns – das Signal ist somit nicht symmetrisch.

Zunächst die gute Nachricht: Unser Programm funktioniert wie erwartet – der Pin wird abwechselnd auf **HIGH** und **LOW** gesetzt.

Allerdings zeigt die Messung auch: Das Signal ist nicht symmetrisch. Die HIGH-Phase dauert nur etwa 125 ns, während die LOW-Phase etwa doppelt so lang ist.

Warum das so ist, analysieren wir im nächsten Abschnitt – anhand der internen Taktzyklen, die jede Instruktion benötigt.

5 Analyse des Programmlaufs

Der im Labor eingesetzte Mikrocontroller (ATmega328P) arbeitet mit einer Taktfrequenz von 16 MHz. Damit dauert ein einzelner CPU-Takt:

$$\frac{1}{16 \text{ MHz}} = \frac{1}{16 \cdot 10^6} \text{ s} = \frac{1000}{16} \cdot 10^{-9} \text{ s} = 62.5 \text{ ns.}$$

Im Datenblatt des ATmega328P kann man nachlesen, wie viele Taktzyklen einzelne Maschinenbefehle benötigen. Für unser Beispiel ergibt sich eine erfreuliche Regelmäßigkeit: Jeder der verwendeten Befehle benötigt genau zwei Takte zur Ausführung – einschließlich des Sprungbefehls. Wir ergänzen daher unser Ablaufdiagramm um die jeweilige Taktanzahl:

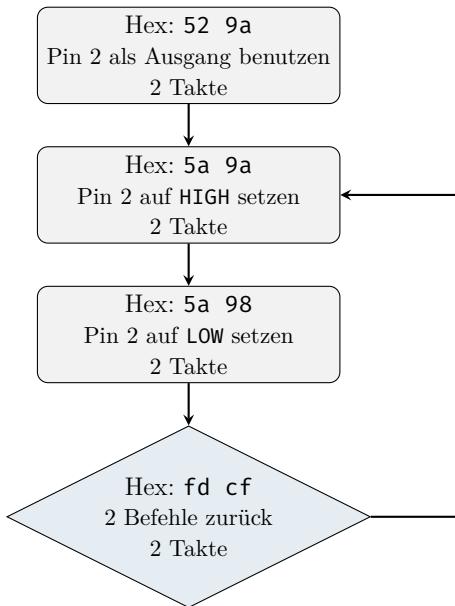


Abbildung 3 zeigt, wie sich der Zustand von Pin 2 im Verlauf der Programmausführung verändert. Beachtet man, dass zwei CPU-Takte 125 ns und vier Takte 250 ns entsprechen, erklärt sich damit nicht nur die in **Abbildung 2** gemessene Signalform. Es zeigt sich auch, dass man durch genaue Kenntnis der Taktanzahl pro Befehl sehr präzise vorhersagen kann, wie sich das Signal verhalten wird – ganz ohne zu messen.

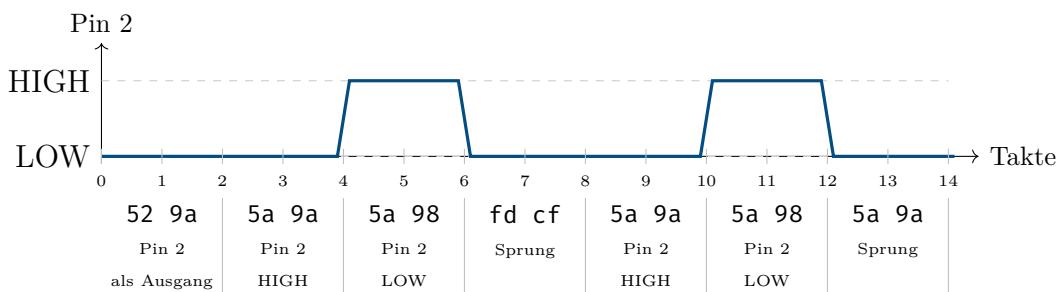


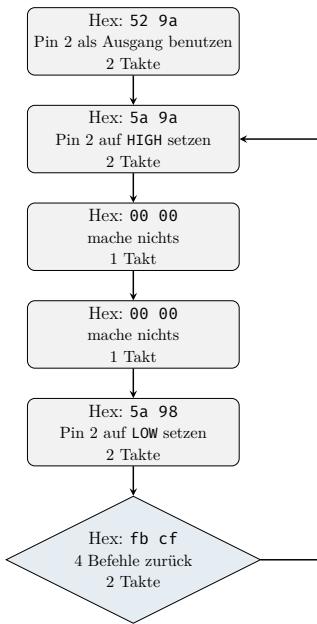
Abbildung 3: Signalverlauf an Pin 2 in Abhängigkeit der CPU-Takte. Die HIGH- und LOW-Zustände folgen aus den verwendeten Maschinenbefehlen, die jeweils zwei Takte benötigen.

Im nächsten Abschnitt werden wir das Programm so modifizieren, dass ein symmetrisches Signal entsteht.

6 Programmanpassung für ein symmetrisches Signal

Um ein symmetrisches Signal zu erzeugen, müssen wir lediglich dafür sorgen, dass Pin 2 für zwei zusätzliche Takte auf **HIGH** bleibt. Glücklicherweise stellt der ATmega328P – wie nahezu jeder Prozessor – einen sogenannten **NOP**-Befehl bereit („No Operation“), der genau das tut: nichts. Laut Datenblatt benötigt dieser Befehl genau einen CPU-Takt.

Wir können unser bestehendes Programm daher leicht anpassen, indem wir zwei **NOP**-Befehle direkt nach dem Setzen von Pin 2 auf **HIGH** einfügen. Dadurch bleibt der Pin für insgesamt vier Takte (statt bisher zwei) auf **HIGH** und wir erhalten ein symmetrisches Signal mit gleicher Dauer der **HIGH**- und **LOW**-Phasen. Das neue Ablaufdiagramm sieht wie folgt aus:



Schreibt man die Hexadezimalwerte der Befehle nacheinander auf, lautet das modifizierte Programm:

```
52 9a 5a 9a 00 00 00 00 5a 98 fb cf
```

Zum Abschluss stellt sich nun die Frage, wie das modifizierte Programm in das Intel-Hex-Format überführt werden kann. Die Herausforderung besteht darin, die neuen Befehle korrekt in einer Hex-Datei darzustellen, sodass sie von **avrdude** fehlerfrei in den Flash-Speicher geschrieben werden können.

Übung

Erstelle für das modifizierte Programm eine gültige Intel-Hex-Datei. Achte dabei auf die korrekte Adressierung und die richtige Berechnung der Prüfsumme. Anschließend kannst du das neue Programm mit **avrdude** auf den Mikrocontroller übertragen und mit dem Oszilloskop überprüfen, ob nun ein symmetrisches Signal erzeugt wird.

Wie ihr in diesem Fall die Prüfsumme im Labor unter Gefechtsbedingungen schnell berechnen könnt, erfahrt ihr auf Laborblatt 1.

7 Assembler statt Maschinencode

Das bisherige Programm zur Ansteuerung der LED haben wir direkt im Maschinencode geschrieben – also Byte für Byte festgelegt, welche Instruktionen ausgeführt werden sollen. Das war eine lehrreiche Erfahrung, aber auch mühsam. Zum Glück gibt es Werkzeuge, die uns diese Arbeit abnehmen. Ein sogenannter *Assembler* übersetzt Programme, die in einer menschenlesbaren Symbolsprache geschrieben sind (sogenannter *Assemblercode*), automatisch in Maschinencode. Ein Assemblerprogramm, das exakt dasselbe Verhalten wie unser modifiziertes LED-Programm erzeugt, könnte folgendermaßen aussehen:

```
sbi    0x0a, 2 ; Pin 2 als Ausgang konfigurieren

loop: sbi    0x0b, 2 ; Pin 2 auf HIGH setzen
      nop          ; Nichts machen
      nop          ; Nichts machen
      cbi    0x0b, 2 ; Pin 2 auf LOW setzen
      rjmp   loop    ; Zurück zur Marke 'loop (d.h. 4 Befehle zurück)
```

Listing 2: Datei led_pin2.s

Dabei gilt:

- **sbi** steht für „Set Bit“ – ein bestimmtes Bit in einem I/O-Register wird auf 1 gesetzt.
- **cbi** steht für „Clear Bit“ – ein bestimmtes Bit wird auf 0 gesetzt.
- **nop** ist eine sogenannte No-Operation – der Befehl macht nichts, belegt aber einen CPU-Takt.
- **rjmp** ist ein relativer Sprungbefehl – er springt um eine bestimmte Anzahl an Befehlen zurück (hier: zur Marke **loop**).

Der Vorteil: Der Code ist deutlich lesbarer als eine Folge von Bytes wie **52 9a ...**. Man erkennt sofort, was gemeint ist – und das Risiko von Fehlern ist deutlich geringer.

7.1 Vom Assemblercode zur Intel-Hex-Datei

Ein Assemblerprogramm wie das oben gezeigte kann mit dem Tool **avr-as** in Maschinencode übersetzt werden. In den guten alten Tagen war das oft schon alles: Der erzeugte Maschinencode konnte direkt ausgeführt oder als Intel-Hex-Datei gespeichert werden – fertig. Die Welt war einfach gestrickt, und die Werkzeuge ebenso.

Heute sind die Systeme komplexer – und entsprechend auch die Toolchains. Um mit dieser Komplexität umzugehen, werden die Aufgaben in kleinere Schritte unterteilt:

- **avr-as** erzeugt zunächst eine sogenannte *Objektdatei* (**.o**) mit Maschinencode, aber noch ohne vollständige Adressinformationen.
- Diese Datei wird vom *Linker* **avr-ld** weiterverarbeitet. Der Linker ersetzt symbolische Sprungziele – wie zum Beispiel die Marke **loop** in unserem Assemblerprogramm – durch konkrete Adressen. Dadurch entsteht eine ausführbare Datei im sogenannten *ELF-Format* (Executable and Linkable Format), einem modernen Standard unter Linux.

- Da der ATmega328P das ELF-Format natürlich nicht versteht (er stammt ja schließlich aus den guten alten Tagen und er erwartet rohen Maschinencode, wie es sich gehört), muss der Maschinencode mit **avr-objcopy** aus der ELF-Datei extrahiert und in das *Intel-Hex-Format* überführt werden – ein Format, das sich zum Flashen des Mikrocontrollers eignet.

Die komplette Toolchain sieht also so aus:

```
avr-as -o led_pin2.o led_pin2.s
avr-ld -mavr5 -o led_pin2.elf led_pin2.o
avr-objcopy -O ihex -R .eeprom led_pin2.elf led_pin2.hex
```

Bevor man den Mikrocontroller flasht, kann man sich den Inhalt der Datei `led_pin2.hex` ansehen – er sollte exakt mit dem von Hand erzeugten Intel-Hex-File aus [Abschnitt 6](#) übereinstimmen. Anschließend kann der Mikrocontroller – wie bereits in [Unterabschnitt 3.3](#) gezeigt – mit **avrdude** geflasht werden.

7.2 Ein Makefile für Linux und macOS

Um die Arbeit mit mehreren Assemblerdateien zu erleichtern, verwenden wir das in [Listing 4](#) gezeigte **Makefile**. Dieses sorgt dafür, dass zu jeder `.s`-Datei automatisch eine passende Intel-Hex-Datei erzeugt wird. Das Makefile sollte sich dafür im gleichen Verzeichnis wie die Assemblerdateien befinden. Zum Übersetzen genügt der Aufruf:

```
make
```

Zusätzlich kann man mit

```
make upload HEX=led_pin2.hex
```

gezielt eine Hex-Datei – wie hier `led_pin2.hex` – auf den Mikrocontroller übertragen. Im Prinzip verbirgt sich hinter dem „Verify“-Knopf in der Arduino-IDE ein Aufruf von `make`, und hinter dem „Upload“-Knopf ein Aufruf von `make upload HEX=....`

Für die Übersetzung kommt die zuvor beschriebene Toolchain aus **avr-as**, **avr-ld**, **avr-objcopy** und **avrdude** zum Einsatz – jedoch in sinnvoller Kombination. Genauer gesagt:

- Mit **avr-as** wird `led_pin2.o` nur dann erzeugt, wenn sich `led_pin2.s` verändert hat – also wenn die Quelldatei neuer ist als das Objektfile oder letzteres noch fehlt.
- Mit **avr-ld** wird ein neues `led_pin2.elf` nur erzeugt, wenn sich `led_pin2.o` verändert hat.
- Mit **avr-objcopy** wird ein neues `led_pin2.hex` nur dann erzeugt, wenn sich `led_pin2.elf` verändert hat.

Das spart unnötige Arbeit – und genau darin liegt die Stärke von `make`. Wird hingegen direkt mit

```
make upload HEX=led_pin2.hex
```

Listing 4 Datei `Makefile` zur Übersetzung von Assemblerprogrammen unter Linux und macOS. Diese Version verarbeitet ausschließlich `.s`-Dateien.

```
1 # Diese Version verarbeitet nur Assemblerprogramme (*.s)
2 # Später wird ein erweitertes Makefile auch C unterstützen.
3
4 SOURCES := $(wildcard *.s)
5 OBJECTS := $(SOURCES:.s=.o)
6 ELFS   := $(SOURCES:.s=.elf)
7 HEXS   := $(SOURCES:.s=.hex)
8
9 all: $(HEXS)
10
11 %.o: %.s
12     avr-as -o $@ $<
13
14 %.elf: %.o
15     avr-ld -mavr5 -o $@ $<
16
17 %.hex: %.elf
18     avr-objcopy -O ihex -R .eeprom $< $@
19
20 clean:
21     $(RM) $(OBJECTS) $(ELFS) $(HEXS)
22
23 ifeq ($(MAKECMDGOALS),upload)
24 ifeq ($(shell uname), Linux)
25 PORT := $(shell ls /dev/ttyUSB* 2>/dev/null | head -n 1)
26 else
27 PORT := $(shell ls /dev/cu.usb* 2>/dev/null | head -n 1)
28 endif # uname
29 endif # MAKECMDGOALS == upload
30
31 upload: $(HEX)
32     @if [ -z "$(HEX)" ]; then \
33         echo "Usage: make upload HEX=<filename.hex>"; \
34         false; \
35     fi
36     @if [ -z "$(PORT)" ]; then \
37         echo "no USB device found"; \
38         false; \
39     fi
40     avrdude -c arduino -P $(PORT) -p atmega328p -U flash:w:$(HEX):i
```

eine Datei hochgeladen, dann prüft `make` automatisch, ob die angegebene Datei aktuell ist – also ob eventuell zuvor noch Schritte wie die Übersetzung aus `.s` nach `.o`, das Linken zur `.elf`-Datei oder das Erzeugen der `.hex`-Datei erforderlich sind. Auch diese Abhängigkeiten werden zuverlässig erkannt und in der richtigen Reihenfolge ausgeführt – ganz ohne weiteres Zutun.

Auch wenn man es auf den ersten Blick nicht vermutet: Was man im **Makefile** sieht, ist tatsächlich eine Programmiersprache. Über deren Syntax und Benutzerfreundlichkeit lässt sich sicher streiten – aber funktional ist sie allemal: Sie erkennt automatisch Abhängigkeiten zwischen Dateien, entscheidet, welche Zwischenschritte notwendig sind, und führt sie in der richtigen Reihenfolge aus.

Allerdings: Manche Dinge tun beim Hinschauen weh – etwa der Block in Zeile 23–29. Dass dort die `ifeq`-Konstrukte nicht eingerückt sind, ist kein Versehen, sondern leider notwendiger Gehorsam: `make` akzeptiert Direktiven wie `ifeq`, `endif`, `define` usw. nur am Zeilenanfang – genau wie auch einfache Zuweisungen wie `PORT := ...`. Eine visuelle Einrückung zur besseren Lesbarkeit ist schlicht nicht erlaubt. Gerade wenn man – wie der Autor – allergisch auf uneinheitliches Einrücken in C ist und es für einen Segen hält, dass Python Einrückung erzwingt, wirkt die Syntax von `make` mitunter wie eine kleine Zeitreise in die 1980er.

Das Makefile in [Listing 4](#) wird euch im Labor zur Verfügung gestellt. Wie man es erweitert, sodass auch C-Programme verarbeitet werden können, ist später Teil einer Laboraufgabe. Dazu ist es nicht notwendig, jedes Detail über Makefiles zu kennen. Es reicht, wenn ihr zunächst nachvollzieht, wie die oben beschriebenen Abhängigkeiten im Makefile umgesetzt sind.

In Zeile 11 steht zum Beispiel die Regel, wie man aus einer `.s`-Datei eine `.o`-Datei erzeugt – sofern eine Aktualisierung notwendig ist (d. h. die `.o`-Datei fehlt oder älter ist als die zugehörige `.s`-Datei). In Zeile 12 findet sich mit `avr-as -o $@ $<` das zugehörige Kommando wobei `$@` automatisch durch den Namen der Zielfile (also der `.o`-Datei) ersetzt wird und `$<` durch den Namen der Quelldatei (also der `.s`-Datei).

Insgesamt beschreibt das Makefile somit einen gerichteten Graphen von Abhängigkeiten, wie er in [Abbildung 5](#) visualisiert ist: Jeder Schritt baut auf dem vorhergehenden auf, und `make` sorgt dafür, dass alle nötigen Befehle in der richtigen Reihenfolge ausgeführt werden – aber auch nur dann, wenn sie wirklich erforderlich sind.

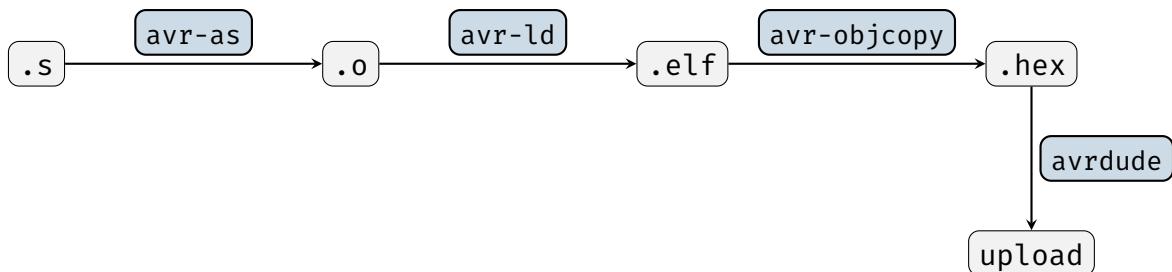


Abbildung 5: Abhängigkeitsgraph für die Erzeugung einer `.hex`-Datei und den anschließenden Upload. Die Werkzeuge `avr-as`, `avr-ld` und `avr-objcopy` erzeugen die `.hex`-Datei, die schließlich von `avrdude` auf den Mikrocontroller übertragen wird.

7.3 Eleganter blinken mit exklusivem Oder

Das folgende Programm lässt – wie zuvor – eine LED blinken, und zwar so, dass sie abwechselnd 250 ns an und 250 ns aus ist. Es entsteht also wieder ein sauberes, symmetrisches Signal.

Der Unterschied liegt im Code: Diesmal kommen Register zum Einsatz – und vor allem eine neue Instruktion, nämlich ein exklusives Oder. Diese logische Operation wird häufig mit **XOR** abgekürzt (für *exclusive OR*). Beim ATmega328P heißt die entsprechende Maschineninstruktion jedoch **EOR** – wie in Zeile 5 des folgenden Programms.

Werfen wir zuerst einen Blick auf den Code, bevor wir im Detail nachvollziehen, wie er funktioniert und welche Rolle Register und andere Besonderheiten dabei spielen:

```
1      ldi    r19,  4          ; r19 = Maske für Pin 2 (Bem.: 4 = 2^2)
2      out    0x0a, r19        ; Pin 2 als Ausgang
3      ldi    r18,  0          ; r18 = 0
4 loop:
5      eor    r18,  r19        ; r18 = r18 XOR r19 ("toggle")
6      out    0x0b, r18        ; Pin 2 auf HIGH oder LOW setzen
7      rjmp   loop
```

Listing 3: Datei `led_pin2_xor.s`

Bevor wir den neuen Code Schritt für Schritt analysieren, sollen zunächst einige wichtige Begriffe und Konzepte geklärt werden:

- Was sind Register – und worin unterscheiden sie sich von normalen Speicherzellen?
- Was genau sind I/O-Register – und wie kann man über sie die Pins des Mikrocontrollers ansteuern?
- Wie lässt sich ein Pin als Ein- oder Ausgang konfigurieren?
- Und wie kann man ihn gezielt auf **HIGH** oder **LOW** setzen?

Die folgenden Abschnitte beantworten all diese Fragen. Sie helfen nicht nur, den neuen Code besser zu verstehen, sondern erklären rückblickend auch viele Details unseres allerersten Assemblerprogramms ([Abschnitt 7](#)).

Im Labor könnt ihr übrigens gerne zuerst einfach ausprobieren, ob dieses neue Programm tatsächlich wieder ein 250 ns–250 ns–Blinksignal erzeugt – ganz so wie das vorherige. Wenn das Oszilloskop ein vertrautes Muster zeigt, ist das schon mal ein gutes Zeichen. Und wie genau dieses Programm funktioniert, sehen wir uns jetzt im Detail an.

Register: Rechnen nur mit Register Register sind spezielle Speicherzellen, die der Prozessor direkt für Rechenoperationen verwenden kann. Beim ATmega328P gibt es davon 32 Stück – jeweils 8 Bit groß – mit den Namen **r0**, **r1**, **r2**, ..., **r31**. Technisch gesehen liegen Register beim ATmega328P im gleichen Datenraum wie der SRAM ([Abbildung 1](#)), der auch für lokale und globale Variablen verwendet wird – *die genaue Struktur dieses Speicherbereichs zeigt Abbildung 6*.

Die Besonderheit ist: Rechenoperationen wie Addition, Vergleich oder logische Verknüpfungen können ausschließlich mit Registern durchgeführt werden. Wenn man mit Variablen arbeiten möchte, die im normalen SRAM liegen, muss man sie zunächst in ein Register laden – und nach der Berechnung ggf. zurückschreiben.

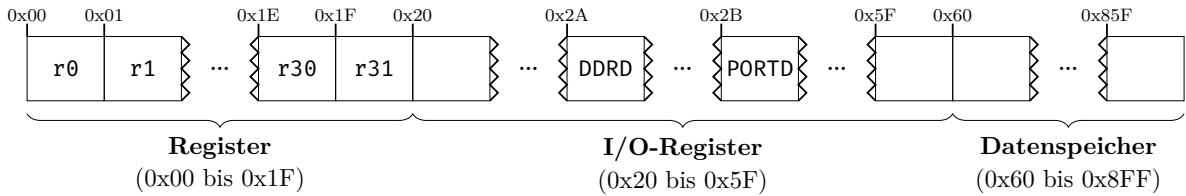


Abbildung 6: Schematische Darstellung (nicht maßstabsgetreu) des SRAM-Bereichs des ATmega328P. Gezeigt sind die Speicherzellen für die Register, sowie die Speicheradressen, mit denen sich die Richtung der Pins festlegen lässt (z.B. DDRD) oder die Ausgänge direkt gesetzt werden (z.B. PORTD).

Um überhaupt mit Registern rechnen zu können, müssen sie zunächst mit einem Wert belegt werden. Eine einfache Möglichkeit dafür ist der Befehl `ldi`, der für „load immediate“ steht – auf Deutsch etwa: „direkt laden“. Damit kann man eine Konstante unmittelbar in ein Register schreiben:

```
ldi    r19, 4 ; r19 erhält den Wert 4
ldi    r18, 0 ; r18 erhält den Wert 0
```

In diesem Beispiel wird das Register `r19` mit dem Wert 4 vorbelegt – also mit einer Bitmaske, bei der nur Bit 2 gesetzt ist. Das Register `r18` erhält den Wert 0.

Nun kommt eine typische Rechenoperation ins Spiel: die sogenannte *exklusive Oder*-Verknüpfung (XOR), die im Assembler des ATmega328P durch den Befehl `eor` realisiert wird. Sie funktioniert bitweise und liefert nur dann eine 1, wenn die beiden Operanden an der jeweiligen Stelle unterschiedliche Bits besitzen. Der Befehl

```
eor    r18, r19
```

führt dazu, dass der Inhalt von `r18` durch `r18 XOR r19` ersetzt wird. Da `r18` zuvor den Wert 0 hatte, und `r19` den Wert 4 hat, ergibt sich $r18 = 0 \text{ XOR } 4 = 4$. Führt man anschließend erneut die Instruktion

```
eor    r18, r19
```

aus, dann ist $r18 = 4 \text{ XOR } 4 = 0$. Der ursprüngliche Wert wurde also wieder gelöscht. Wiederholte Anwendung dieser Instruktion toggelt den Inhalt von `r18` somit zwischen 0 und 4 – genau dieses Verhalten nutzen wir aus, um den Pin-Zustand wechselweise auf HIGH und LOW zu setzen:

$$\begin{array}{ll}
 \begin{array}{ll}
 00000100 & (\text{r19} = 4) \\
 \text{XOR } 00000000 & (\text{r18} = 0) \\
 \hline
 00000100 & (\text{neu: r19} = 4)
 \end{array} &
 \begin{array}{ll}
 00000100 & (\text{r19} = 4) \\
 \text{XOR } 00000100 & (\text{r18} = 4) \\
 \hline
 00000000 & (\text{neu: r19} = 0)
 \end{array}
 \end{array}$$

I/O-Register: Schnittstelle zu den Pins I/O-Register (Input/Output-Register) sind spezielle Speicherzellen im SRAM-Bereich, über die sich die Pins des Mikrocontrollers steuern lassen. Für die Pins 0–7 unseres Mikroprozessors sind vor allem die Adressen `0x2A` und `0x2B` relevant:

- `0x2A` entspricht dem sogenannten `DDRD`-Register. Wenn dort das Bit mit Index 2 gesetzt wird, wird Pin 2 als Ausgang konfiguriert.

- **0x2B** entspricht dem PORTD-Register. Wenn dort das Bit mit Index 2 gesetzt wird, liegt an Pin 2 ein HIGH-Pegel an – andernfalls ist er LOW.

Interessant ist: In unserem ersten Assemblerprogramm (siehe [Abschnitt 7](#)) haben wir bereits diese Register verwendet – allerdings mit anderen Adressen. Dort hieß es zum Beispiel:

```
sbi 0x0a, 2 ; Pin 2 als Ausgang festlegen
sbi 0x0b, 2 ; Pin 2 auf HIGH setzen
cbi 0x0b, 2 ; Pin 2 auf LOW setzen
```

Hier wurde die Adresse **0x0a** anstelle von **0x2A** und **0x0b** anstelle von **0x2B** verwendet. Der Grund: Die Instruktionen **sbi** (set bit) und **cbi** (clear bit) sind speziell dafür gedacht, direkt auf I/O-Register zuzugreifen. Sie operieren nur auf Adressen im Bereich **0x00** bis **0x1F**, was den ersten 32 I/O-Registern entspricht. Intern wird bei der Ausführung dieser Befehle automatisch ein Offset von **0x20** hinzugerechnet. So spart man im Maschinenbefehl wertvolle Bits, da nur ein kleiner Adressbereich codiert werden muss.

Ein I/O-Register vollständig überschreiben Im bisherigen Beispiel wurden einzelne Bits in den I/O-Registern gezielt gesetzt oder gelöscht – z. B. mithilfe der Instruktionen **sbi** (set bit) oder **cbi** (clear bit). Eine alternative Möglichkeit besteht darin, gleich das gesamte Register auf einmal zu überschreiben. Das ist nicht nur kompakter und übersichtlicher, sondern auch effizienter: Während **sbi** und **cbi** jeweils zwei CPU-Takte benötigen, lässt sich ein vollständiger Schreibzugriff mit der Instruktion **out** in nur einem einzigen Takt erledigen.

Zur Erinnerung:

- Das Register **DDRD** (Adresse **0x2A**) legt fest, welche Pins als Ausgang verwendet werden. Um nur Pin 2 als Ausgang zu aktivieren, schreibt man die Zahl **4** ($= 2^2$) in das **DDRD**-Register – also genau das Bit mit Index 2.
- Das Register **PORTD** (Adresse **0x2B**) bestimmt, ob ein Ausgangspin **HIGH** oder **LOW** ist. Um Pin 2 auf **HIGH** zu setzen, schreibt man ebenfalls den Wert **4** hinein. Um ihn auf **LOW** zu setzen, genügt der Wert **0**.

Dazu verwendet man die Instruktion **out**, die zwei Angaben verlangt:

- die Zieladresse – z. B. **0x2A** für **DDRD** oder **0x2B** für **PORTD**
- ein Register, das den zu schreibenden Wert enthält

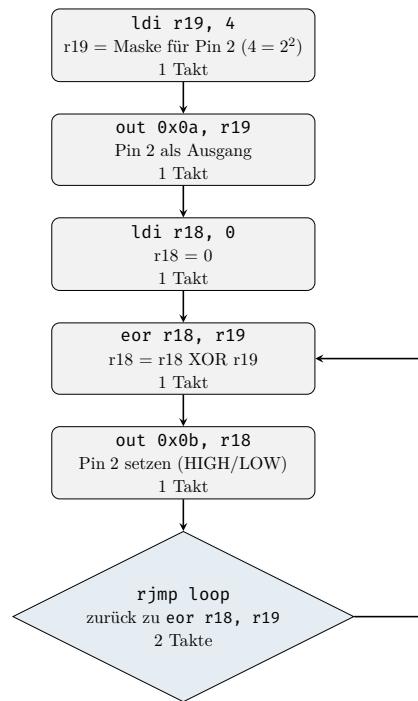
Im folgenden Beispiel wird genau das umgesetzt:

```
ldi r19, 4      ; Bit 2 setzen
out 0x0A, r19  ; Pin 2 als Ausgang festlegen (DDRD)
out 0x0B, r19  ; Pin 2 auf HIGH setzen (PORTD)
```

Wichtig: Auch bei **out** wird – wie bei **sbi** und **cbi** – intern ein Offset von **0x20** zur angegebenen Adresse addiert. In obigem Beispiel wird also tatsächlich an Adresse **0x2A** bzw. an Adresse **0x2B** geschrieben, was genau der physikalischen Speicheradresse von **DDRD** entspricht (siehe [Abbildung 6](#)).

Damit haben wir nun jede Instruktion aus [Listing 3](#) verstanden. Wenn wir jetzt noch den Ablauf des Programms analysieren, wird deutlich, warum das Signal am Pin 2 erneut

symmetrisch ist – also wie beim ersten Programm jeweils 250 ns HIGH und 250 ns LOW. Der folgende Ablaufplan zeigt die Taktanzahl und erklärt, wie sich der Pin-Zustand über die Zeit ändert:



8 Einfaches Delay mit einem Register

Unser nächstes Ziel ist es, dass Pin 2 etwas langsamer zwischen HIGH und LOW wechselt. Am Ende soll es schließlich möglich sein, dort eine LED anzuschließen, die dann für das Auge wahrnehmbar mit einer vorgegebenen Frequenz blinkt.

Kleiner Spoiler: Mit dem aktuellen Ansatz würde die LED mit etwa 20 000 Hertz blinken – das wäre für das Auge immer noch nicht wahrnehmbar (selbst wenn es technisch möglich wäre, eine LED mit so hoher Frequenz ein- und auszuschalten).

Aber die Idee, die wir hier entwickeln und umsetzen, lässt sich leicht verbessern, um im nächsten Abschnitt unser Ziel zu erreichen. Der in [Listing 4](#) gezeigte Code enthält im Vergleich zu [Listing 3](#) drei zusätzliche Instruktionen, die in den Zeilen 7 bis 10 hinzugekommen sind:

- Die Instruktion in Zeile 7 initialisiert das Register **r24** mit dem Wert 2. Dieses Register dient als Zähler für einen Countdown und kann natürlich auch mit anderen Werten belegt werden. Da Register bei unserem Mikrocontroller jedoch nur 8 Bit breit sind, sind nur Werte von 0 bis 255 (bzw. in Hexadezimaldarstellung von **0x00** bis **0xFF**) möglich.
- Die Instruktion in Zeile 9 dekrementiert das Register **r24** um 1. Technisch wird dabei der Wert 1 von **r24** subtrahiert und das Ergebnis wieder in **r24** gespeichert. Wichtig ist hierbei, dass bei dieser Subtraktion nicht nur das Rechenergebnis, sondern auch sogenannte *Statusflags* berechnet werden. Eines davon ist das sogenannte *Zero-Flag*, das genau dann gesetzt wird, wenn das Ergebnis der Subtraktion gleich 0 ist.
- Die Instruktion in Zeile 10 ist ein bedingter Sprung. **brne** steht für „Branch if Not Equal“ und bedeutet: Der Sprung zur Marke **delay** erfolgt nur, wenn das Zero-Flag *nicht* gesetzt ist – also genau dann, wenn **r24** nach der letzten Subtraktion noch nicht 0 ist. Ist das Zero-Flag hingegen gesetzt (d. h. **r24** ist jetzt 0), wird mit der nächsten Instruktion in Zeile 12 fortgefahren.

[Abbildung 7](#) zeigt ein Ablaufdiagramm für das Programm in [Listing 4](#). Hier wird deutlich, dass mit den neu eingefügten Zeilen eine innere Schleife realisiert wird, in der das Register **r24** heruntergezählt wird. Diese Schleife ist eingebettet in eine äußere Schleife, mit der Pin 2 abwechselnd auf **HIGH** und **LOW** gesetzt wird.

Je größer der initiale Wert von **r24** ist, desto mehr Takte werden benötigt, um die innere Schleife zu durchlaufen, bevor erneut der Zustand von Pin 2 geändert wird. Die Dauer, die Pin 2 in einem bestimmten Zustand (also **HIGH** oder **LOW**) verweilt, lässt sich somit gezielt beeinflussen.

8.1 Taktanzahl pro Schleifendurchlauf

Wir möchten nun berechnen, wie viele Takte zwischen zwei aufeinanderfolgenden Ausführungen der **eor**-Instruktion in Zeile 5 vergehen – also wie lange es dauert, bis sich der Zustand von Pin 2 wieder ändert.

Entscheidend ist dabei, wie oft die innere Schleife durchlaufen wird. Wenn das Register **r24** mit einem Wert n initialisiert wird, dann wird die Schleife insgesamt n Mal durchlaufen. Dabei erfolgt der Rücksprung zur Marke **delay** genau $n - 1$ Mal. Beim n -ten Mal ist das Ergebnis der Subtraktion null, das Zero-Flag ist gesetzt, und es erfolgt kein Sprung mehr.

```

1      ldi    r19,  4          ; 1 Takt
2      out    0x0a, r19        ; 1 Takt
3      ldi    r18,  0          ; 1 takt
4  loop:
5      eor    r18,  r19        ; 1 Takt
6      out    0x0b, r18        ; 1 Takt
7      ldi    r24,  0x2        ; 1 Takt (Zähler für Countdown setzen)
8  delay:
9      subi   r24,  1          ; 1 Takt
10     brne   delay           ; 2 Takte bei Sprung, 1 Takt sonst
11
12     rjmp   loop            ; 2 Takte

```

Listing 4 led_pin2_delay1.s – Pin 2 wird mit Hilfe einer Countdown-Schleife in **r24** verzögert getoggelt

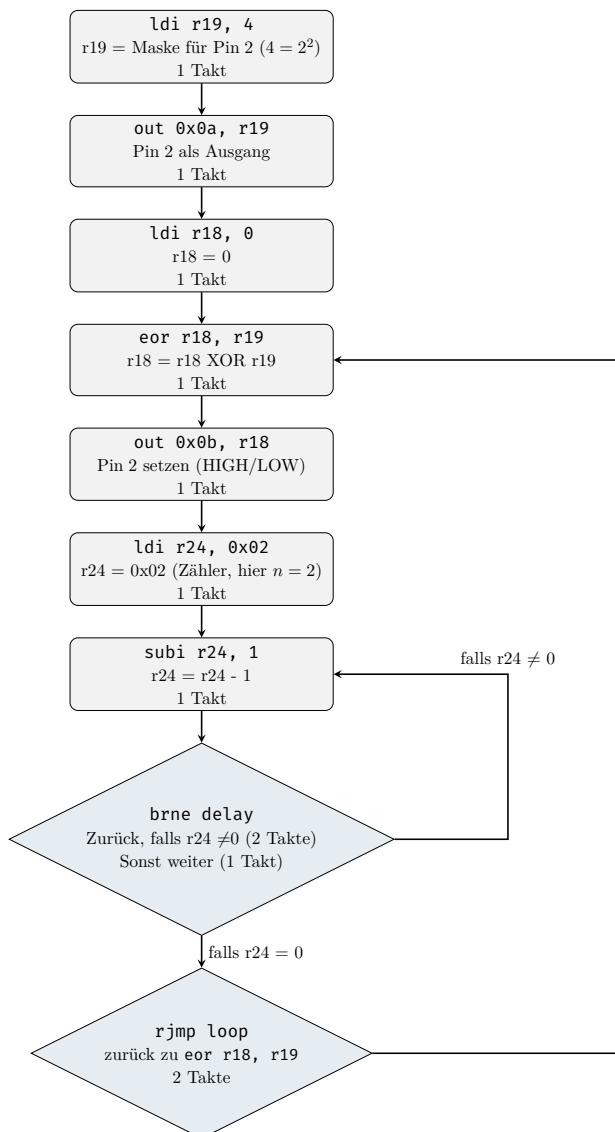


Abbildung 7: Ablaufdiagramm für das Programm aus [Listing 4](#) mit Verzögerungsschleife über **r24**. Die äußere Schleife ändert den Pin-Zustand, die innere Schleife erzeugt eine variable Pause.

Laut Datenblatt benötigt die **subi**-Instruktion (Zeile 9) stets 1 Takt. Die **brne**-Instruktion (Zeile 10) benötigt:

- 2 Takte, wenn der Sprung ausgeführt wird ($n - 1$ Mal),
- 1 Takt, wenn kein Sprung erfolgt (1 Mal).

Die innere Schleife (Zeilen 9–10) benötigt damit insgesamt:

$$(n - 1) \cdot (1 + 2) + 1 + 1 = 3(n - 1) + 2 \quad \text{Takte}$$

Zusätzlich werden vor und nach dieser Schleife folgende Instruktionen ausgeführt:

- Zeile 6: **ldi r24, n** → 1 Takt
- Zeile 5: **out 0x0b, r18** → 1 Takt
- Zeile 4: **eor r18, r19** → 1 Takt
- Zeile 11: **rjmp loop** → 2 Takte

Das ergibt insgesamt:

$$3(n - 1) + 2 + 3 + 2 = 3n + 4 \quad \text{Takte pro Schleifendurchlauf.}$$

Je größer also der Initialwert n , desto länger dauert es, bis sich der Zustand an Pin 2 wieder ändert.

8.2 Nachmessen der Verzögerung

Die folgende Tabelle zeigt, wie viele Takte für einen Durchlauf der äußeren Schleife benötigt werden, abhängig vom Initialwert n im Register **r24**. Die Anzahl der Takte ergibt sich gemäß der vorherigen Herleitung zu:

$$T(n) = 3(n - 1) + 7 = 3n + 4 \tag{1}$$

Ein Takt dauert bei einem Mikrocontroller mit 16 MHz genau 62,5 ns. [Tabelle 2](#) zeigt für verschiedene Werte von n , wie viele Takte und wie viel Zeit ein kompletter Durchlauf der äußeren Schleife benötigt. Natürlich werden wir das im Labor nachmessen. Dabei ist es hilfreich, zunächst für kleine Werte von n zu überprüfen, wie lange Pin 2 jeweils auf HIGH bzw. LOW ist, um so die Formel aus (1) zu validieren.

Initialwert n	Taktanzahl	Zeit in μs	Zeit in ns
1	7	0,4375	437,5
2	10	0,6250	625
3	13	0,8125	812,5
4	16	1,0000	1000
255	769	48,0625	48062,5

Tabelle 2: Verzögerung pro Schleifendurchlauf für verschiedene Initialwerte

9 Längeres Delay mit vier Registern

```
1      ldi    r19,  4          ; für Pin 2 (4 = 2^2)
2      out    0x0a, r19
3      ldi    r18,  0
4 loop:
5      eor    r18,  r19        ; 1 Takt
6      out    0x0b, r18        ; 1 Takt
7
8      ldi    r24,  0x54        ; 1 Takt
9      ldi    r25,  0x58        ; 1 Takt
10     ldi   r26,  0x14        ; 1 Takt
11     ldi   r27,  0x00        ; 1 Takt
12 delay:
13     subi  r24,  1          ; 1 Takt
14     sbci  r25,  0          ; 1 Takt
15     sbci  r26,  0          ; 1 Takt
16     sbci  r27,  0          ; 1 Takt
17     brne  delay           ; 2 Takte bei Sprung, 1 Takt sonst
18
19     rjmp  loop             ; 2 Takte
```

Listing 5 Programm `led_pin2_delay4.s`: Vier Register (`r24` bis `r27`) werden als 32-Bit-Zähler verwendet, um ein deutlich längeres Delay zu erzeugen. Der Pin 2 wird dabei weiterhin mit jeder Iteration der äußeren Schleife getoggelt.

Um ein längeres Delay zu erzielen, benötigen wir mehr als 8 Bit für den Countdown-Zähler. Das in [Listing 5](#) gezeigte Programm verwendet daher einen 32-Bit-Zähler, der über die vier Register `r24` bis `r27` realisiert wird. Diese werden als Registerverbund betrachtet und gemeinsam heruntergezählt, um eine deutlich längere Verzögerung zu erreichen als mit nur einem einzelnen Register.

Ein *Registerverbund* bezeichnet die Interpretation mehrerer aufeinanderfolgender 8-Bit-Register als ein gemeinsames größeres Register, in unserem Fall ein 32-Bit-Zähler. Jedes Register speichert dabei ein Byte des Gesamtwerts, wobei `r24` das niederwertigste Byte und `r27` das höchstwertige Byte enthält. Zum Beispiel entspricht die folgende Belegung

$$r24 = 0x12, \quad r25 = 0x34, \quad r26 = 0x56, \quad r27 = 0x78$$

dem zusammengesetzten Wert

$$n = r27:r26:r25:r24 = 0x78563412.$$

Im Listing [Listing 5](#) wird der Zähler in den Zeilen 8 bis 11 mit vier `ldi`-Instruktionen initialisiert. Dort ergibt sich

$$n = r27:r26:r25:r24 = 0x00000200 = 512,$$

was bedeutet, dass die Schleife 512-mal durchlaufen wird, bevor der Zustand von Pin 2 erneut geändert wird.

Bei unserem Mikroprozessor gibt es keine Instruktion, um direkt einen Registerverbund mit vier Registern zu dekrementieren⁴. Man kann eine solche Operation aber durch eine Kombination von **subi** (Subtrahiere Sofortwert) und **sbc i** (Subtrahiere mit Übertrag) nachbauen. Das geschieht in den Zeilen 13 bis 16 in Listing 5:

```
subi r24, 1
sbc i r25, 0
sbc i r26, 0
sbc i r27, 0
```

Vereinfacht gesagt haben diese vier Instruktionen den Effekt, dass der gesamte 32-Bit-Wert im Registerverbund **r27:r26:r25:r24** um eins dekrementiert wird. Sobald der Wert den Wert Null erreicht hat (d. h. alle vier Register enthalten den Wert **0x00**), wird das Zero-Flag im Statusregister gesetzt – andernfalls bleibt es gelöscht.

Etwas genauer: Die **subi**-Instruktion subtrahiert den Wert 1 von **r24** und setzt dabei die Flags im Statusregister, unter anderem das Carry-Flag (C), falls ein Übertrag stattgefunden hat. Zwei typische Fälle:

- **Fall 1:** $r24 = 0x01$ vor der Subtraktion.

Nach **subi r24, 1** ist $r24 = 0x00$, und es tritt kein Übertrag auf, das Carry-Flag bleibt gelöscht.

- **Fall 2:** $r24 = 0x00$ vor der Subtraktion.

Nach **subi r24, 1** ergibt sich $r24 = 0xFF$, da die Subtraktion unterläuft. In diesem Fall wird das Carry-Flag gesetzt.

Allgemein gilt: Das Carry-Flag wird bei **subi** genau dann gesetzt, wenn **r24** vor der Subtraktion den Wert **0x00** hatte, also ein Übertrag beim Subtrahieren von 1 auftritt.

Die nachfolgenden **sbc i**-Instruktionen subtrahieren dann jeweils den Wert 0 *unter Berücksichtigung* des Carry-Flags. Wenn es z. B. beim Dekrementieren von **r24** einen Übertrag gab (weil es vorher 0 war), wird dieser Übertrag auf **r25** übertragen, wodurch der Verbund korrekt um 1 dekrementiert wird. Die Verkettung dieser vier Instruktionen bildet damit ein korrektes 32-Bit-Dekrement nach. Das Zero-Flag wird nur dann gesetzt, wenn das Ergebnis der gesamten Operation null ist, d. h. alle vier Register enthalten **0x00** und es gab keinen Übertrag mehr nach der letzten **sbc i**-Instruktion.

9.1 Taktanzahl pro Schleifendurchlauf

Abbildung 8 zeigt ein Ablaufdiagramm für Listing 5. Obwohl wir deutlich mehr Instruktionen haben als beim Delay mit einem Register, ist die Struktur sehr ähnlich: Eine innere Schleife für die Verzögerung und eine äußere für das Toggeln von Pin 2.

Wir können also analog bestimmen, wie viele Takte vergehen, bis sich der Zustand von Pin 2 wieder ändert. Ist n der Wert des Register-Verbunds **r27:r26:r25:r24**, dann benötigt die innere Schleife

$$(n - 1) \cdot 6 + 5$$

Takte. Die äußere Schleife enthält weitere Instruktionen, die zusammen 8 Takte benötigen. Insgesamt ergibt sich damit:

$$T(n) = (n - 1) \cdot 6 + 5 + 8 = 6n + 7$$

⁴Für einen Registerverbund aus zwei Registern, z. B. **r25:r24**, gibt es die **SBIW**-Instruktion (Subtract Immediate from Word).

Für einige Beispielwerte lässt sich das überprüfen:

- $n = 1 \Rightarrow T(1) = 13$ Takte $\Rightarrow 13 \cdot 62,5 \text{ ns} = 812,5 \text{ ns}$
- $n = 2 \Rightarrow T(2) = 19$ Takte $\Rightarrow 19 \cdot 62,5 \text{ ns} = 1187,5 \text{ ns}$

Noch interessanter ist die Frage, welchen Wert n man wählen muss, damit Pin 2 mit einer Frequenz von 1 Hz blinkt, also jeweils 500 ms HIGH und 500 ms LOW ist. Dazu muss gelten:

$$(6n + 7) \cdot 62,5 \text{ ns} = 500 \text{ ms} = 500\,000\,000 \text{ ns}$$

Löst man diese Gleichung nach n auf, ergibt sich:

$$n = \frac{500\,000\,000 / 62,5 - 7}{6} = \frac{8\,000\,000 - 7}{6} = 1\,333\,332,166\dots$$

Da wir nur ganze Werte für n einsetzen können, wählen wir:

$$n = 1\,333\,332 = \mathbf{0x145854}$$

Dieser Wert liegt innerhalb unserer Messgenauigkeit und sorgt dafür, dass eine angeschlossene LED etwa mit 1 Hz blinkt.

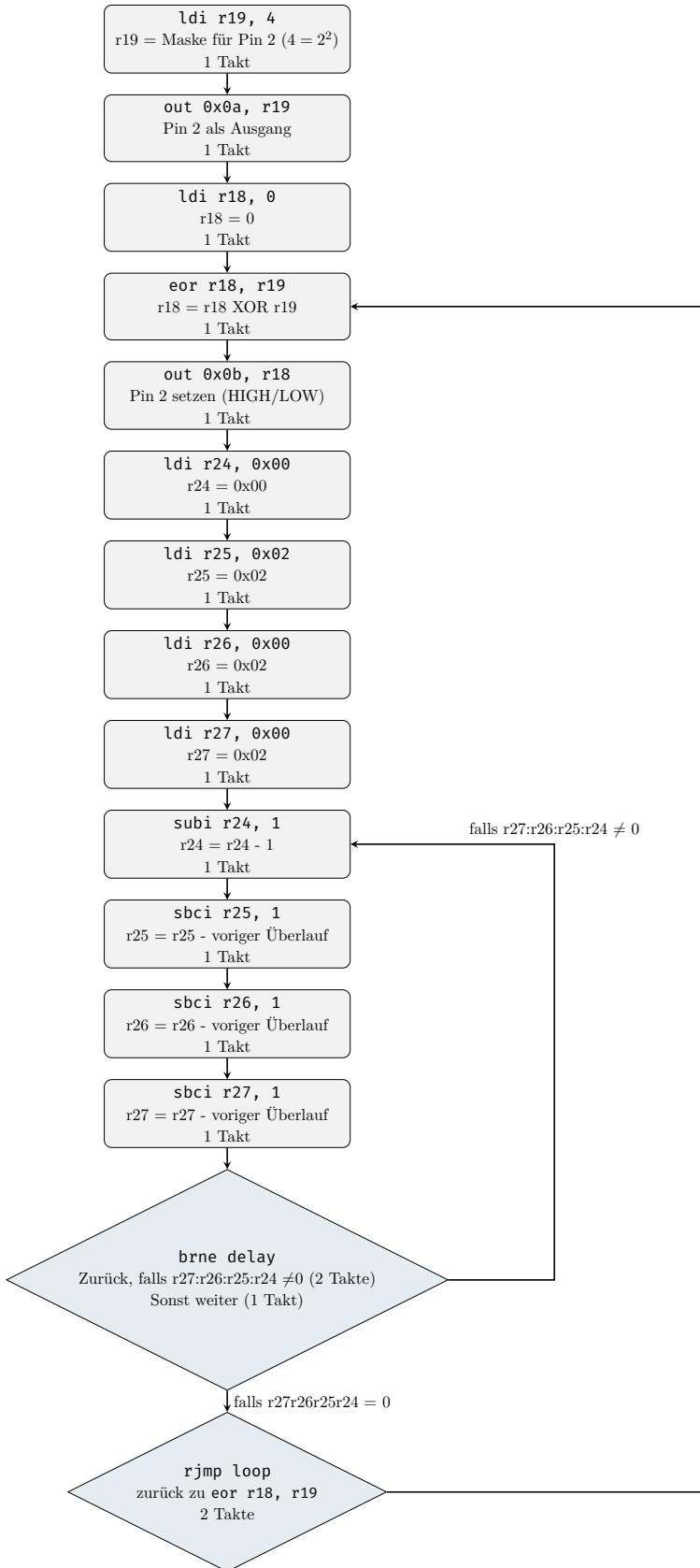


Abbildung 8: Ablaufdiagramm zum Programm `led_pin2_delay4.s`. Der Countdown erfolgt über einen 32-Bit-Registerverbund ($r27:r26:r25:r24$). Das Diagramm zeigt, wie durch die innere Schleife mit `subi/sbci/brne` ein längeres Delay erzeugt wird, bevor in der äußeren Schleife der Zustand von Pin 2 erneut gewechselt wird.

10 Bare-Metal-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     // 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 }
```

Listing 6 C-Programm `led_pin2_delay.c`, das nahezu identischen Assembler-Code erzeugt wie die handgeschriebene Variante in [Listing 5](#). Die LED an Pin 2 wird mit exakt demselben Delay getoggelt. Unterschiede im erzeugten Assembler-Code ([Listing 8](#)) sind minimal und betreffen nur irrelevante Details.

Was passiert eigentlich beim Klick auf „Upload“ in der Arduino-IDE? Mit dem Wissen aus diesem Projekt können wir diese Frage nun beantworten – und die einzelnen Schritte selbst ausführen. In diesem letzten Abschnitt steigen wir von Maschinencode zu C auf, bleiben aber ganz nah an der Hardware: ohne Bibliotheken, ohne Magie – bare metal.

10.1 C-Programm für eine blinkende LED

[Listing 6](#) zeigt ein C-Programm, mit dem eine LED, die an Pin 2 angeschlossen ist, mit einer Frequenz von 1 Hz blinken wird. Schaut man sich den Code an, dann sieht man, dass vielleicht doch etwas Magie dabei ist. Der Code in den Zeilen 1 bis 23 ist nichts für schwache Nerven und erfordert nicht nur ein etwas tieferes Verständnis der Programmiersprache C und wie ein C-Compiler daraus Maschinencode erzeugt, sondern auch Kenntnisse über den Mikroprozessor. Hier werden zum Beispiel die Adressen `0x2A` und `0x2B` verwendet, um Pin 2 als Ausgang zu konfigurieren bzw. ihn auf HIGH oder LOW zu setzen.

Aber der Witz ist: Ein Endnutzer muss nicht verstehen, wie der Code in den Zeilen 1 bis 18 funktioniert, um ihn verwenden zu können. Denn im Endeffekt werden damit einfach benutzbare Funktionen zur Verfügung gestellt:

- `setPinModes` – um Pins als Eingang oder Ausgang zu konfigurieren,
- `digitalWriteAll` – um Pins auf HIGH oder LOW zu setzen,
- `delay_cycles` – um eine gewisse Anzahl von Taktzyklen zu warten.

Die Implementierung dieser Funktionen kann man in einer Bibliothek verstecken; man muss dann nur wissen, was diese Funktionen tun und wie sie zu benutzen sind. Für Arduino gibt es dafür eine sehr gute Dokumentation.

Als Anwender muss man dann nur den Code in den Zeilen 20 bis 34 selbst schreiben. Und auch diesen Teil kann man noch vereinfachen.

Es geht an dieser Stelle also **nicht** darum, diesen speziellen C-Code im Detail zu verstehen – denn später werden wir ja die Arduino-Entwicklungsumgebung verwenden, und mit ihr ist das deutlich einfacher. Hier soll lediglich gezeigt werden, dass man grundsätzlich mit ein paar Zeilen C-Code genau das umsetzen kann, was wir zuvor in Assembler programmiert haben: eine blinkende LED, direkt auf der Hardware, ohne Bibliotheken, ganz nah am Metall.

10.2 Toolchain

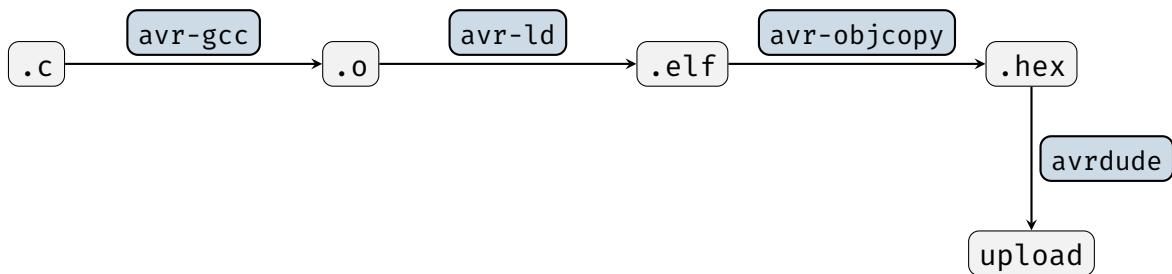


Abbildung 9: Abhängigkeitsgraph für die Erzeugung einer `.hex`-Datei und den anschließenden Upload. Die Werkzeuge `avr-gcc`, `avr-ld` und `avr-objcopy` erzeugen die `.hex`-Datei, die schließlich von `avrdude` auf den Mikrocontroller übertragen wird.

Die Toolchain hat sich nun leicht verändert: Der C-Compiler kann aus dem C-Code direkt Maschinencode in Form eines Object-Files erzeugen. Dieses `.o`-File wird dann wie zuvor vom Linker und weiteren Werkzeugen verarbeitet, bis am Ende eine `.hex`-Datei entsteht. Eine Übersicht über diesen Ablauf zeigt [Abbildung 9](#). Natürlich kann man ein Makefile

wie in [Listing 7](#) verwenden, um nicht jeden Befehl jedes Mal tippen zu müssen. Und damit ist eigentlich das Geheimnis schon gelüftet:

Was passiert beim Klick auf „Upload“?

Tippt man ein Programm wie in [Listing 6](#) ein und drückt auf den Upload-Button, dann wird intern ein Kommando wie
`make upload HEX=led_pin2_delay.hex`
ausgeführt, mit dem die komplette Toolchain automatisch abgearbeitet wird.

Wir können dieses Kommando auch manuell auf der Kommandozeile eingeben und dabei beobachten, was passiert:

```
MCL:c lehn$ make upload HEX=led_pin2_delay.hex
avr-gcc -mmcu=atmega328p -c -Os -o led_pin2_delay.o led_pin2_delay.c
avr-ld -mavr5 -o led_pin2_delay.elf led_pin2_delay.o
avr-objcopy -O ihex -R .eeprom led_pin2_delay.elf led_pin2_delay.hex
avrdude -c arduino -P /dev/cu.usbserial-01C5FB3D -p atmega328p -U flash:w:
    led_pin2_delay.hex:i
Reading 30 bytes for flash from input file led_pin2_delay.hex
Writing 30 bytes to flash
Writing | ##### | 100% 0.03 s
Reading | ##### | 100% 0.01 s
30 bytes of flash verified

Avrdude done. Thank you.
rm led_pin2_delay.elf led_pin2_delay.o
```

10.3 Lesbaren Assembler-Code statt Maschinencode erzeugen

Früher haben C-Compiler nur Assembler-Code erzeugt, der anschließend von einem Assembler in Maschinencode übersetzt wurde. Heutige Compiler überspringen diesen Schritt in der Regel und erzeugen direkt Maschinencode, da im Compiler praktisch ein Assembler bereits integriert ist.

Wenn man möchte, kann man sich den erzeugten Assembler-Code aber nach wie vor ausgeben lassen – und genau das ist in unserem Fall interessant. Unser Makefile hilft uns dabei, den Compiler mit den dafür notwendigen Optionen aufzurufen. Mit dem Kommando

```
MCL:c lehn$ make led_pin2_delay.s
avr-gcc -mmcu=atmega328p -S -Os -o led_pin2_delay.s led_pin2_delay.c
```

wird der C-Code `led_pin2_delay.c` in Assembler-Code (`led_pin2_delay.s`) übersetzt.

[Listing 8](#) zeigt den vom C-Compiler `avr-gcc` erzeugten Assembler-Code. Relevant sind hier nur die Zeilen 16 bis 32. Tatsächlich könnte man alle übrigen Zeilen löschen und aus den verbleibenden Zeilen ein `.hex`-File erzeugen, das sich in der Funktionalität nicht unterscheiden würde.

```

1 # Diese Version verarbeitet Assemblerprogramme (*.s)
2 # und C-Programme (*.c)
3
4 ASM_SOURCES := $(wildcard *.s)
5 C_SOURCES := $(wildcard *.c)
6 OBJECTS := $(ASM_SOURCES:.s=.o) $(C_SOURCES:.c=.o)
7 ELFS := $(OBJECTS:.o=.elf)
8 HEXS := $(ELFS:.elf=.hex)
9
10 all: $(HEXS)
11
12 %.o: %.c
13         avr-gcc -mmcu=atmega328p -c -Os -o $@ $<
14
15 %.s: %.c
16         avr-gcc -mmcu=atmega328p -S -Os -o $@ $<
17
18 %.o: %.s
19         avr-as -o $@ $<
20
21 %.elf: %.o
22         avr-ld -mavr5 -o $@ $<
23
24 %.hex: %.elf
25         avr-objcopy -O ihex -R .eeprom $< $@ 
26
27 clean:
28     $(RM) $(OBJECTS) $(ELFS) $(HEXS)
29
30 ifeq ($(MAKECMDGOALS),upload)
31 ifeq ($(shell uname), Linux)
32 PORT := $(shell ls /dev/ttyUSB* 2>/dev/null | head -n 1)
33 else
34 PORT := $(shell ls /dev/cu.usb* 2>/dev/null | head -n 1)
35 endif # uname
36 endif # MAKECMDGOALS == upload
37
38 upload: $(HEXS)
39     @if [ -z "$(HEXS)" ]; then \
40         echo "Usage: make upload HEX=<filename.hex>"; \
41         false; \
42     fi
43     @if [ -z "$(PORT)" ]; then \
44         echo "no USB device found"; \
45         false; \
46     fi
47     avrdude -c arduino -P $(PORT) -p atmega328p -U flash:w:$(HEXS):i

```

Listing 7 Makefile zur Übersetzung von C- und Assemblerprogrammen unter Linux und macOS. Diese Version verarbeitet sowohl .c-Dateien als auch .s-Dateien und erzeugt automatisch passende .hex-Dateien. Ein optionales Ziel `upload` erkennt die serielle Schnittstelle automatisch und überträgt das Programm per `avrdude`.

10.4 Vergleich mit unserem Assembler-Programm

```
1      .file "led_pin2.c"
2  __SP_H__ = 0x3e
3  __SP_L__ = 0x3d
4  __SREG__ = 0x3f
5  __tmp_reg__ = 0
6  __zero_reg__ = 1
7      .text
8      .section    .text.startup, "ax", @progbits
9  .global main
10     .type main, @function
11 main:
12 /* prologue: function */
13 /* frame size = 0 */
14 /* stack size = 0 */
15 .L_stack_usage = 0
16     ldi r18, lo8(4)
17     out 0xa, r18
18     ldi r18, 0
19     ldi r19, lo8(4)
20 .L3:
21     eor r18, r19
22     out 0xb, r18
23     ldi r24, lo8(84)
24     ldi r25, lo8(88)
25     ldi r26, lo8(20)
26     ldi r27, 0
27 .L2:
28     sbiw r24, 1
29     sbci r26, 0
30     sbci r27, 0
31     brne .L2
32     rjmp .L3
33     .size main, .-main
34     .ident "GCC: (Homebrew AVR GCC 14.2.0) 14.2.0"
```

Listing 8 Assemblercode `led_pin2_delay.s`, automatisch erzeugt von `avr-gcc` aus dem C-Programm in [Listing 6](#). Der Code unterscheidet sich nur in zwei irrelevanten Details von der handgeschriebenen Version in [Listing 5](#), insbesondere bleibt das Timing vollständig identisch.

Gehen wir die Unterschiede zu unserem handgeschriebenen Assembler-Programm schrittweise durch:

1. In den Zeilen 16 bis 19 steht:

```
ldi r18, lo8(4)
out 0xa, r18
ldi r18, 0
ldi r19, lo8(4)
```

Das ist äquivalent zu:

```
ldi r18,4  
out 0xa,r18  
ldi r18,0  
ldi r19,4
```

Der Compiler hat hier also eine unnötige Instruktion erzeugt. Cleverer (und kürzer) wäre:

```
ldi r19,4  
out 0xa,r19  
ldi r18,0
```

Und genau das haben wir in unserem Programm gemacht.

2. In den Zeilen 20 bis 22 steht:

```
.L3:  
eor r18,r19  
out 0xb,r18
```

Unsere Marke heißt **loop**, seine heißt **.L3**. Der Rest ist identisch. Wie man eine Marke nennt, spielt keine Rolle – Menschen bevorzugen aussagekräftige Namen, Compiler verwenden oft **.L** mit fortlaufender Nummer.

3. In den Zeilen 23 bis 26 steht:

```
ldi r24,lo8(84)  
ldi r25,lo8(88)  
ldi r26,lo8(20)  
ldi r27,0
```

Das ist gleichbedeutend mit:

```
ldi r24, 84  
ldi r25, 88  
ldi r26, 20  
ldi r27, 0
```

Schreibt man die Werte hexadezimal, wie wir es tun, ergibt sich:

```
ldi r24, 0x54  
ldi r25, 0x58  
ldi r26, 0x14  
ldi r27, 0x00
```

4. In den Zeilen 27 bis 32 steht:

```
.L2:  
sbiw r24, 1  
sbc i r26, 0  
sbc i r27, 0  
brne .L2  
rjmp .L3
```

Unsere Marke heißt **delay** statt **.L2**, da wir Menschen sind. Ein weiterer Unterschied ist die erste Instruktion:

```
sbiw r24, 1
```

Diese benötigt 2 Takte und dekrementiert das Registerpaar **r25:r24**. Sie hat also denselben Effekt wie:

```
subi r24, 1  
sbc i r25, 0
```

Der Compiler verwendet hier **sbiw**, weil diese Instruktion – obwohl sie genau so viele Takte benötigt wie unsere Variante mit **subi** und **sbc i** – die Codegröße reduziert. **sbiw** belegt nur 2 Bytes im Flash, während zwei separate Instruktionen insgesamt 4 Bytes benötigen würden. Gerade bei Mikrocontrollern mit begrenztem Programmspeicher ist das ein wichtiger Vorteil.

11 Weitere Spielereien mit dem C-Code

Wir haben gesehen, dass der C-Code funktional äquivalent zu unserem Assembler-Code ist. Jetzt kann man natürlich auch mit dem C-Code experimentieren. Zum Beispiel enthält [Listing 6](#) die Zeile:

```
delay_cycles((500000000 / 62.5 - 7) / 6);
```

Diese hat den Effekt, dass Pin 2 abwechselnd 500 ms lang HIGH und anschließend 500 ms lang LOW ist. Die Zahl **n = 500000000** gibt den gewünschten Delay in Nanosekunden an.

Hier kann man natürlich auch andere Werte wählen – auch sehr kleine wie **n = 1** oder **n = 2** – und mit dem Oszilloskop nachmessen, dass sich das Verhalten genauso verändert wie in unserem Assembler-Programm.