Mikrocontroller-technik

Assembler
mit
ATMEL®-
AVR®-Mikrocontrollern

Guy WEILER

www.weigu.lu

Copyright ©

Mikrocontrollertechnik

MODUL

A

Inhaltsverzeichnis MODUL A

Neue Befehle in den einzelnen Kapiteln:

A2: `rjmp, clr, ldi, out`

A3: `ser, sbi, cbi, sbis, sbic, in, com, mov, andi, and, brne, ori, or, eor`

A4: `nop, breq, cpi, subi, add, sub, sts, lds, st, ld, inc, dec, st+, ldd, adiw, lpm`

A5: `sbiw, sbci`

A6: `rcall, ret, push, pop, tst`

A0 Vorwort

Das folgende Lehrbuch für Mikrocontrollertechnik soll das Selbststudium ermöglichen, dient aber auch als Kurs für eine Abschlussklasse in der Techniker-Ausbildung in Luxemburg.

Das Lehrbuch gliedert sich in drei Module (A, B und C), die jeweils aufeinander aufbauen. Ein viertes Modul wird Projekte enthalten. Es werden abwechselnd theoretische und praktische Kapitel behandelt, wobei der Schwerpunkt auf den praktischen Programmieraufgaben liegt. Die theoretischen Kapitel dienen als Lernstoff zur Wissenskontrolle. Einige Kapitel sind recht ausführlich und dienen dann als eher Referenz (z.B. Adressiermodi). Aus diesen sind dann nur die zum jeweiligen Zeitpunkt benötigten Teile zu erlernen.

Im praktischen Teil sind Aufgaben und Text eng verwoben, so dass der größte Teil der Aufgaben gelöst werden muss um den Stoff zu erschließen. Es wird das Prinzip "***learning by doing***" angewendet, da dies die Kompetenz zum Problemlösen verbessert und nur selbst Erarbeitetes wirklich über einen längeren Zeitraum behalten wird.

Dies ist auch der Grund, warum keine Lösungen zu den Aufgaben veröffentlicht werden. Die Gefahr ist zu groß, dass der Lernende glaubt, alleine durch Lesen der Lösung könnte er den Stoff verinnerlichen.

Ein umfangreicher Anhang, der beim Programmieren immer griffbereit sein sollte, schließt den Kurs ab.

Im Kurs wird der Baustein ATmega32 bzw. ATmega32A[1] aus der 8-Bit-AVR®-Familie der Firma ATMEL® verwendet. Allerdings sollten auch alle Programme auf dem kleineren Controller ATmega8A funktionieren. Eventuell sind die Ports anzupassen[2]. Der ATmega16A ist pinkompatibel mit dem ATmega32A und kann natürlich auch verwendet werden. Der ältere AT90S8535 (bzw. die neuere Version ATmega8535) ist auch pinkompatibel und er kann für viele der Aufgaben eingesetzt werden. Allerdings sind einige Register geändert worden (z.B. Timer) so dass leichte Änderungen im Code nötig sind.

In den ersten Kapiteln werden die Bausteine mit dem internen Takt von 1 MHz betrieben, so wie sie auch vom Werk aus programmiert wurden. In einem späteren Kapitel wird dann ein externer Quarz mit 16 MHz verwendet.

Für die Erstellung des Kurses wurden hauptsächlich die Datenblätter von ATMEL®, die freie Enzyklopädie Wikipedia und die folgenden Bücher zu Rate gezogen:

G. Schmit	Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie, 2 Auflage, Oldenburg, ISBN 3-486-58016
R. Walter	AVR Mikrocontroller Lehrbuch, 2 Auflage, www.rowalt.de
W. Tampert	AVR-RISC-Mikrocontroller, 2 Auflage, Franzis, ISBN 3-7723-5476-9

1 Bei den Bausteinen mit dem Zusatz A handelt es sich um die neuere stromsparendere Variante. Diese sind vollständig kompatibel zu den älteren Varianten ohne A.

2 Aufpassen muss man auch, wenn man nicht mit der Definitionsdatei arbeitet, da zum Beispiel die Interruptvektortabellen unterschiedlich sind (gilt auch für den ATmega16)!

Mit Sicherheit befinden sich "Fähler" ;-) in diesem Kurs. Mit der Zeit werden es hoffentlich immer weniger. Sollte Ihnen ein Fehler auffallen so würde ich mich über eine Benachrichtigung freuen. Meine E-Mail-Adresse lautet: "weigu@weigu.lu". Vielen Dank im Voraus.

Lernen macht Spaß!

Auch wenn nicht jeder mit dieser Erkenntnis einverstanden ist, so war es doch die treibende Kraft, die diesen Kurs entstehen ließ.

Genau diesen Spaß wünsche ich allen Lesern die diesen Kurs erarbeiten.

Guy WEILER
(www.weigu.lu)

Darstellung

Für die Darstellung von Hexadezimalzahlen wird die in C übliche Schreibweise mit vorgestelltem **0x** verwendet (Bsp.: **0x5A**)[3]. Zahlen ohne Zusatz sind Dezimalzahlen. Für Binärzahlen wird die Schreibweise mit vorgestelltem **0b** (Bsp.: **0b01110111**) verwendet. ASCII-Zeichen und Strings werden zwischen Hochkomma gesetzt (Bsp.: **'A'**).

Englische Worte werden kursiv (italic) dargestellt. Assembler ist grau unterlegt. **Wichtiger Text** ist ohne Serifen und fett (*bold*) dargestellt. **Arbeitsregister**, **SF-Register**, **Flags** (einzelne Bits in SF-Registern) und **Pins** des Controller sind in unterschiedlichen Farben dargestellt.

Es werden die neueren auf den Zweierpotenzen beruhenden Binärpräfixe "kibi", "mebi", "gibi", "tebi" für binäre Vielfache von Einheiten verwendet. Die vorgestellten Symbole sind Ki, Mi, Gi und Ti.

Beispiel: Ein Kibibit wird mit dem Symbol Kibit oder manchmal Kib bezeichnet (Achtung: Großbuchstabe K, Kleinbuchstabe b für bit) und sind 1024 Bit.
Ein Gigibyte wird mit dem Symbol GiB bezeichnet (Achtung: Großbuchstabe G, Großbuchstabe B für Byte) und sind 2^{30} Byte.

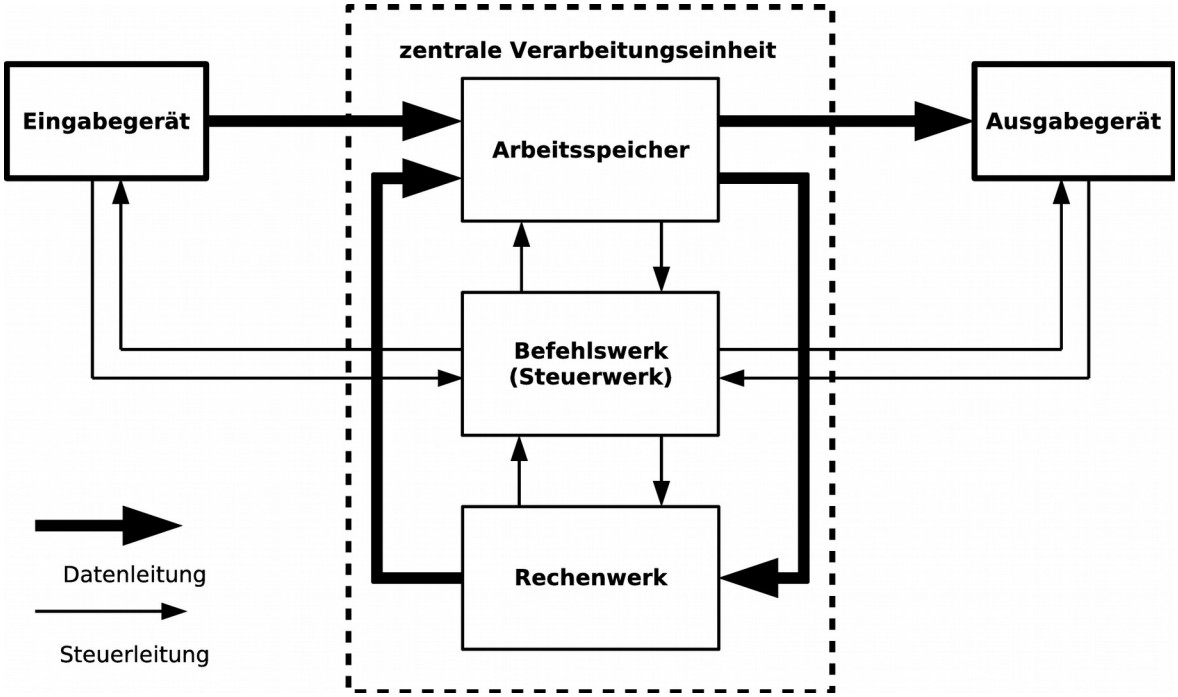
Name	Symbol	Wert
kibi	Ki	2^{10} = 1.024
mebi	Mi	2^{20} = 1.048.576
gibi	Gi	2^{30} = 1.073.741.824
tebi	Ti	2^{40} = 1.099.511.627.776

Weitere Informationen auf: http://de.wikipedia.org/wiki/Bit und
http://de.wikipedia.org/wiki/Byte

3 In PASCAL und BASIC-Programmen wird für Hexzahlen meist ein Dollarzeichen vorgestellt (Bsp.: $5A). Im Text wird bei Hexzahlen auch manchmal statt einem vorgestellten Zeichen ein "h" nachgestellt (Bsp.: 5Ah)

A1 Einführung

Eine Datenverarbeitungsanlage besteht allgemein aus einem oder mehreren Eingabegeräten (Messfühler (Sensoren), Tastatur, Schalter, mobiler Datenspeicher (USB-Stick, Diskette)...), einer zentralen Verarbeitungseinheit (Computer, Mikrocontroller) und einer Ausgabeeinheit (Drucker, Bildschirm, Lautsprecher, LED ...).



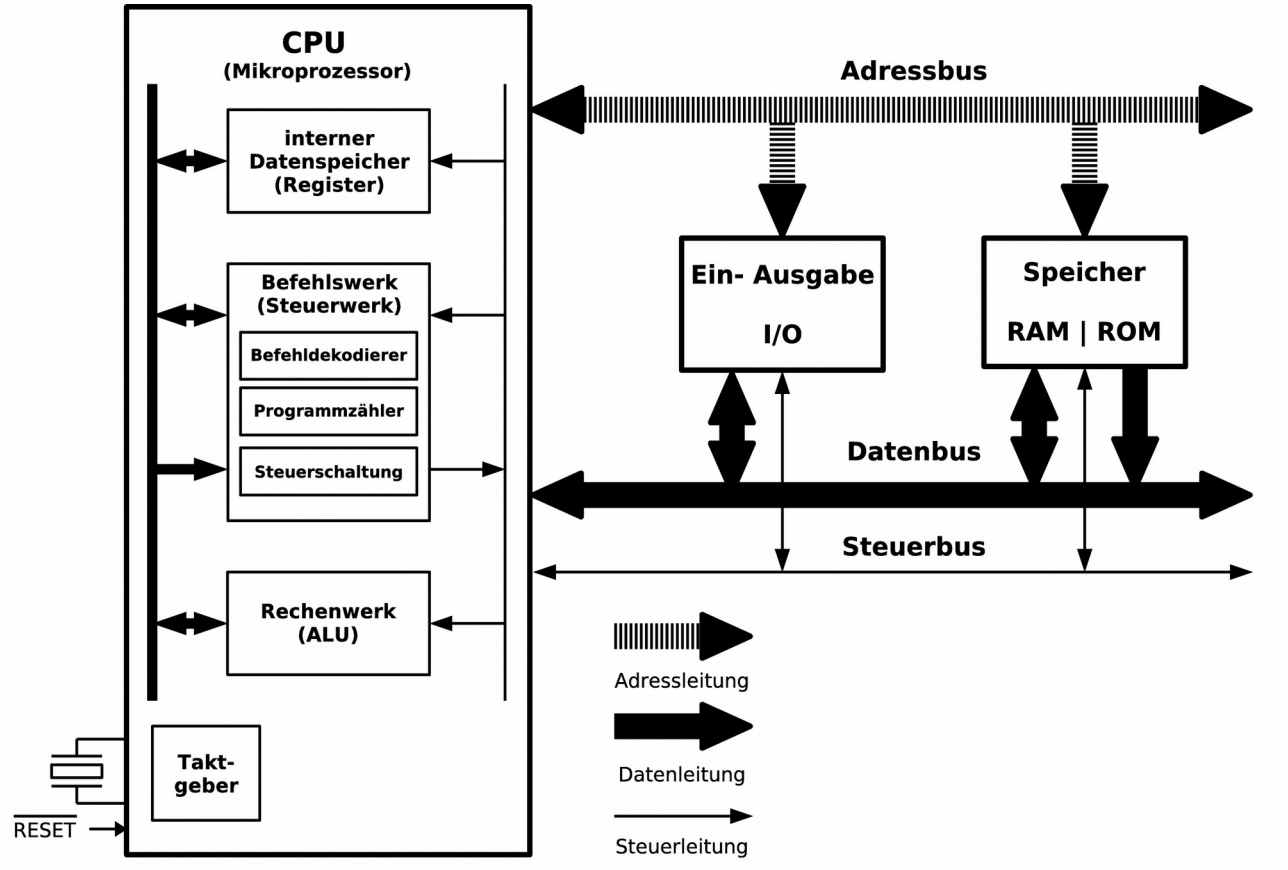
Die **Verarbeitungseinheit oder Zentraleinheit** ist für die Verarbeitung der Daten mittels eines Programms, für die Ein- und Ausgabe der Daten und für die Programmsteuerung verantwortlich. Sie besteht aus:

- **Rechenwerk** (Arithmetische und logische Einheit, ALU). Hier werden alle die Daten betreffenden Operationen ausgeführt.
- **Arbeitsspeicher** (Zentralspeicher, Hauptspeicher). Die Daten erhält das Rechenwerk aus dem Arbeitsspeicher und speichert sie auch wieder dahin zurück.
- **Befehlswerk** (Leitwerk, Steuerwerk). Das Befehlswerk steuert alle Arbeitsvorgänge und bewirkt das richtige Zusammenspiel dieser Vorgänge. Es führt alle von den Befehlen vorgegebenen Operationen aus.

Aufbau eines Mikroprozessorsystems

Das Herzstück eines heutigen Mikroprozessorsystems (Computer, Mikrocomputer) ist der Mikroprozessor (**CPU**[4], ***C**entral **P**rocessing **U**nit*, Zentraleinheit), der das Rechenwerk und das Befehlswerk enthält.

Um funktionsfähig zu sein benötigt der Mikroprozessor zusätzlich noch Speicher und Ein-/ Ausgabe-Bausteine.



Ein Mikroprozessorsystem (Mikrocomputer) ist eine Datenverarbeitungs- und Steuereinheit. Sie besteht aus:

- **Mikroprozessor (CPU, *Central Processing Unit*), Zentraleinheit)**
- **Speicher (*Memory* (*Read Only, Random Access*), Zentralspeicher)**
- **Ein-/Ausgabe-Bausteinen (*Input/Output, I/O*)**

4 Genau genommen wird die CPU nur dann Mikroprozessor genannt, wenn sie in einem einzigen Chip integriert wurde. Dies ist heute fast immer der Fall.

Der Mikroprozessor (µP)

Der Mikroprozessor ist der Chef (*master*) im System. Er steuert den gesamten Ablauf der Datenverarbeitung und bearbeitet die Daten.

Beispiel: **Der 8080 von Intel**

(Quellen: Datenblatt Intel, Wikipedia http://de.wikipedia.org/wiki/Intel_8080)

Der Intel 8080 ist ein 1974 eingeführter 8-Bit-Mikroprozessor von Intel. Er wird allgemein als erster vollwertiger Mikroprozessor angesehen. Als Nachfolger des Intel 8008 war sein Befehlssatz zu diesem kompatibel.

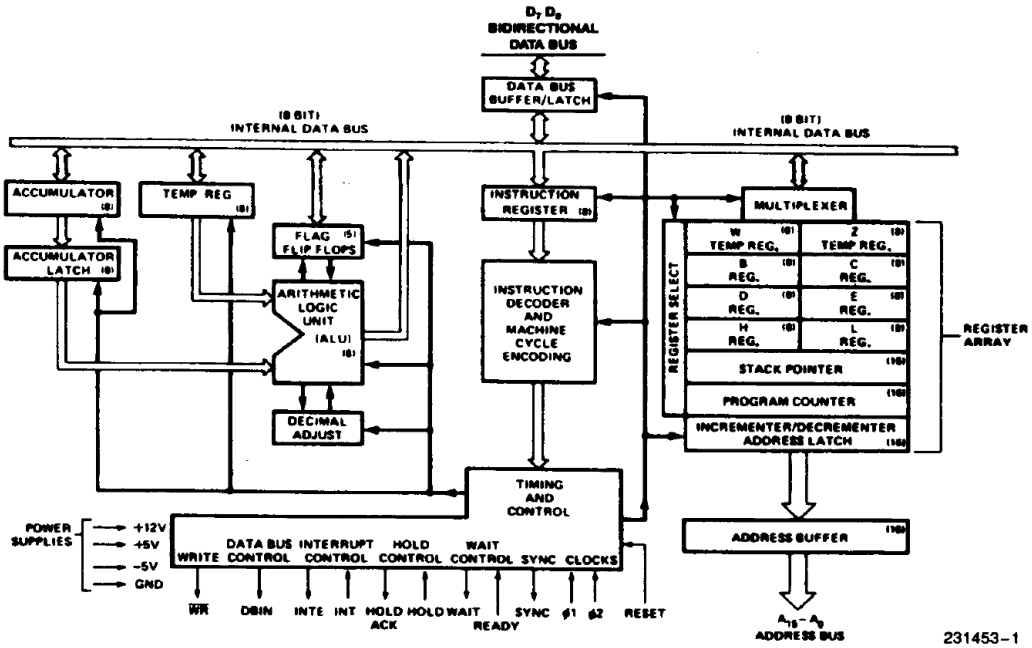


Figure 1. Block Diagram

Das großzügige Design mit 40 Anschlussstiften (Pins) ermöglichte einen Adressbus mit 16 Bit, so dass der 8080 64 KiB Speicher adressieren konnte. Außerdem konnten 256 vom Adressbereich unabhängige Ein-/Ausgabe-Register angeschlossen werden.
Er verfügte über sieben 8-Bit-Register, von denen sechs zu drei 16-Bit-Registern kombiniert werden konnten, sowie einen Stapelzeiger (*Stackpointer*) und einen Programmzeiger (*Program Counter PC*) mit je 16 Bit. Das Betriebssystem CP/M wurde für den 8080 entwickelt und stellte ein knappes Jahrzehnt lang das vorherrschende System für Mikrocomputer dar, ähnlich wie später MS-DOS.
Eingesetzt wurde der 8080 in Steuergeräten (z.B. Marschflugkörpern) und den ersten PCs (u.a. Micral, IMSAI und Altair 8800).
Neben dem eigentlichen Prozessor benötigte der 8080 noch zwei weitere Bausteine: einen separaten Taktgenerator und einen Buscontroller.
1976 brachte die Firma Zilog mit dem Z80 einen 8080-kompatiblen, aber stark erweiterten Prozessor heraus.

Technische Daten:
- Taktfrequenz: 2 MHz
- Anzahl Transistoren: 6.000 bei 6 µ
- Datenbus: 8 Bit
- Adressbus: 16 Bit
- Adressierbarer Speicher: 64 KiB

Auch moderne Prozessoren basieren im Grundaufbau noch immer auf der beim 8080 verwendeten Architektur (siehe Von-Neumann-Architektur).

Moderne Prozessoren arbeiten allerdings mit 64-Bit-Daten und besitzen sogar mehrere interne Kerne, so dass eine echte parallele Verarbeitung der Daten möglich ist, was ihre Verarbeitungsgeschwindigkeit vervielfacht.

Die Arbeitsgeschwindigkeit einer CPU hängt aber auch vom verwendeten rechteckförmigen Taktsignal ab, das alle Abläufe innerhalb des Mikrocomputers steuert. Dieses Signal wird meist von einem in der CPU integriertem Quarzgenerator erzeugt. Der Quarz wird extern angeschlossen. Mit unterschiedlichen Quarzen sind sogar oft unterschiedliche Verarbeitungsgeschwindigkeiten möglich.

Der Speicher (Zentralspeicher)

Der externe Speicher dient der Datenspeicherung, muss schnell sein, und wird direkt von der CPU angesteuert. Langsamere periphere Speicher die eher der Aufbewahrung von Daten dienen (Festplatten, DVDs, Speichersticks ...) werden über spezielle Ein-/Ausgabebausteine (Controller) angesteuert.

RAM (*Random Access Memory*)

Der größte Teil des Speichers ist schneller **flüchtiger Speicher**. Dieser **Schreib-Lese-Speicher** besteht meist aus hoch integriertem dynamischen RAM und verliert beim Abschalten seinen Inhalt. Er wird zum Zwischenspeichern der Daten während der Verarbeitung benötigt.

ROM (*Read Only Memory*)

Für Daten, die erhalten werden müssen (Programme, Einstellungsparameter, ...) wird **nichtflüchtiger Speicher** benötigt. Früher war dieser nicht veränderbar (*read only*). Heutige Flash-Speicher lassen sich aber relativ leicht auch beschreiben, was zum Beispiel ein verändern der Firmware[5] eines Gerätes ermöglicht.

Ein-/Ausgabe-Bausteine

Sie sind die Schnittstellen zur Außenwelt. Oft werden sie als Input/Output-Ports (I/O-Ports), Tore (Türen, Pforten) zur Ein- und Ausgabe von Daten, bezeichnet. An ihnen wird die Peripherie (externe Geräte) angeschlossen. Die meisten dieser Bausteine übernehmen einen Teil der Verarbeitung der Daten und entlasten so die CPU. Sie steigern dadurch die Arbeitsgeschwindigkeit des Systems.

Wichtige Ein/Ausgabe-Bausteine sind zum Beispiel die seriellen (SIO) und parallelen Schnittstellen (PIO), die Bausteine zur Steuerung der Festplattenspeicher, der Kartenleser, des Monitors usw..

5 **Firmware** ist die Betriebssystem-Software, die der Hersteller in seinem elektronische Geräten unterbringt.

Adress-, Daten- und Steuerbus

Da Daten und Adressen im Mikroprozessorsystem und auch in der CPU parallel übertragen werden sind viele Leitungen nötig um dies zu erreichen.

Die Gesamtheit solcher parallelen Leitungen wird als Bus (lat.: omnibus = alle) bezeichnet.

An einen solchen Bus werden alle Baugruppen des Mikroprozessorsystem (CPU, Speicher und die I/O-Bausteine) parallel angeschlossen, d.h. dass jede Busleitung mit jedem Baustein verbunden wird. Die Leitungen werden dabei von Null an durchnummeriert. Ein 64-Bit Datenbus besitzt also die Anschlüsse D0-D63.

Auf dem **Datenbus** werden die Daten übertragen.

Über den **Adressbus** wird der Baustein ausgewählt, an den die Daten übermittelt werden sollen. Gleichzeitig wird dabei die Adresse übermittelt, wo die Daten z.B. abgespeichert werden sollen.

Der **Steuerbus** gibt dabei z.B. an ob es sich um eine Lese- oder Schreib-Operation handelt.

Adress- und Steuerbus dienen also zusammen der Steuerung des Informationstransports, da auf den Datenbus immer nur ein Sender und ein Empfänger zugreifen dürfen.

Bemerkung: Ein Bus verbindet im Gegensatz zur Schnittstelle mehr als zwei Teilnehmer.

Vom Mikroprozessor zum Mikrocontroller

Da die Integrationsdichte von integrierten Bausteinen sich immer weiter erhöht hat, wurde es auf einmal möglich fast alle Komponenten eines Mikrocomputers auf einem Chip zu vereinen.

Sind die CPU, der Speicher und einige Ein-/Ausgabe-Bausteine in einem Chip vereint, so spricht man von einem Mikrocontroller (µC, *embedded computer*).

Mikrocontroller haben einen geringen Platzbedarf, kommen bei geringen Taktfrequenzen mit sehr wenig Energie aus und können in hohen Stückzahlen für wenig Geld produziert werden.

Sie können vielfältige Aufgaben mit unterschiedlicher Komplexität übernehmen und sind heute in fast allen elektronischen Geräten, oft sogar mehrfach, zu finden.

Es existieren Mikrocontroller in vielen Varianten. Auf modernen Mikrocontrollern befinden sich neben den üblichen Pheripheriebausteinen wie z.B. Timer, Taktgeneratoren, EEPROM-Speicher oder serielle Schnittstelle auch oft speziellere Peripherieblöcke mit USB- (*Universal Serial Bus*), I²C- (*Inter-Integrated Circuit*), SPI- (*Serial Peripheral Interface*), CAN- (*Controller Area Network*), LIN- (*Local Interconnect Network*), oder Ethernet-Schnittstellen. Auch LCD-Controller und -Treiber, PWM-Ausgänge (Puls-Weiten-Modulation) oder hochauflösende Analog-Digital-Wandler (mit 8 bis 24 Bit Auflösung und bis zu 16 Kanälen) sind oft verfügbar.

CISC und RISC

CISC (*Complex Instruction Set Computing*[6])

CISC bezeichnet die klassischen Befehlssätze von Mikroprozessoren. Ein CISC-Befehlssatz zeichnet sich durch viele verhältnismäßig leistungsfähige Einzelbefehle unterschiedlicher Länge aus. CPUs mit CISC-Befehlssatz sind in der Regel mikroprogrammiert, das heißt, dass das im Steuerwerk enthaltene Mikroprogrammwerk eine Menge von Mikroprogrammen (Mikrocode, Liste von Steuersignalen) enthält. In diesen Mikroprogrammen ist verschlüsselt wie der Prozessor die Befehle ausführt.

Die Entwicklung von Mikroprogrammen ist zeitaufwändig und fehleranfällig. Sie bot früher aber einen Zeitgewinn bei der Entwicklung von kürzerem Maschinencode. Auch konnte so Zentralspeicher eingespart werden, der zur Anfangszeit der Computertechnik sehr teuer war.

Wird bei modernen CPUs Mikrocode verwendet, so ist dieser oft veränderbar, wodurch der Hersteller Bugfixes einspielen kann.

Bei dem CISC Modell hat die CPU üblicherweise wenige Arbeitsregister und viele Adressierungsarten. Klassische CISC-Prozessoren sind zum Beispiel die 808x und 80x86-Serie von Intel, die 680x0 von Motorola sowie der Z80 von Zilog. Ab den Prozessoren 80486, Pentium und 68060 wurden Elemente der RISC-Prozessoren mit eingebaut.

Neue Prozessoren sind kaum noch mikroprogrammiert. Es hat sich herausgestellt, dass Programmierer und Compiler den umfangreichen Befehlssatz nur zu einem kleinen Teil nutzen. Auch billiger schneller Speicher mit großer Kapazität macht eine Reduktion des Code nicht mehr notwendig.

Ab dem Pentium Pro verfügen die Intel-Prozessoren über eine vorgeschaltete Funktionseinheit, welche komplexe Befehle in RISC-Befehle übersetzt.

RISC-Prozessoren (*Reduced Instruction Set Computing*[7])

Bei RISC-Prozessoren werden die Befehle meist direkt durch die Hardware implementiert. Dies ermöglicht eine hohe Ausführungsgeschwindigkeit da der Decodierungsaufwand auf Seiten der CPU gering ist. Man versucht auf komplexe Befehle konsequent zu verzichten, und somit wird kein Mikrocode verwendet.

Schnelle kurze Befehle erlauben es dem Prozessor schneller auf Unterbrechungen (i*nterrupts*) zu reagieren, und so zum Beispiel schneller auf externe Signale (oft Sensoren) zu reagieren. Da dies die Hauptaufgabe von Mikrocontrollern ist, werden diese daher meist als RISC-Prozessor ausgeführt.

Wie erwähnt sind die Befehle beim RISC-Prozessor fest verdrahtet. Das ist der Grund warum erste RISC-Prozessoren einen stark reduzierten Befehlssatz enthielten. Bei neuen Controllern werden allerdings auch komplexere Befehle integriert, was den Befehlssatz vervielfacht (Bsp.: PIC 33 Befehle, ATmega8A 130 Befehle).

6 Rechnen mit komplexem Befehlssatz

7 Rechnen mit reduziertem Befehlssatz

Eigenschaften von RISC-Prozessoren (http://de.wikipedia.org/wiki/Reduced_Instruction_Set_Computing):

- Nur die Befehle LOAD und STORE greifen auf den Speicher zu, die restlichen Befehle arbeiten mit Registeroperanden.
- Dadurch, dass ein Speicherzugriff immer einen expliziten Befehl benötigt, erhöht ein großer Registersatz, d. h. viele allgemeine Register (*general purpose register*, Arbeitsregister), die Effizienz, da Zwischenergebnisse dann lokal vorgehalten werden können.
- RISC-CPUs sind so ausgelegt, dass sie meist nur einen Takt benötigen um einen Befehl abzuarbeiten. Dies wird unter anderem durch „*pipelining*“ erreicht. Abweichungen gibt es nur bei LOAD, STORE, und Verzweigungsbefehlen.
- RISC-CPUs stellen weniger Befehle zur Verfügung als CISC-CPUs. Diese Befehle besitzen in der Regel alle die gleiche Codebreite und sind registerorientiert.

Bemerkung: CISC-Prozessoren benutzen oft Elemente des RISC-Design. Umgekehrt gibt es auch viele RISC-Prozessoren, welche zusätzlich komplexe Befehle integrieren (z.B. der Multiplikationsbefehl bei AVR®-Controllern). Geht es um Geschwindigkeit, ist der RISC-Design erste Wahl.

Die "Von Neumann-Architektur" und die "Harvard-Architektur"

Von Neumann-Architektur (http://de.wikipedia.org/wiki/Von-Neumann-Architektur)

In den ersten Rechnern war ein festes Programm entweder hardwaremäßig verdrahtet, oder es konnte mit Lochkarten eingelesen werden. Der Mathematiker John von Neumann begründete 1945 die nach ihm benannte Architektur, mit der es möglich wurde, Änderungen an Programmen sehr schnell durchzuführen und ganz verschiedene Programme ablaufen zu lassen, ohne Veränderungen an der Hardware vornehmen zu müssen.

Dieser universelle Von-Neumann-Rechner blieb bis heute zum größten Teil erhalten und basiert auf den im Kapitel "Aufbau eines Mikroprozessorsystems" gesehenen Komponenten:

- **Rechenwerk** (ALU, CPU).
- **Steuerwerk** (*control unit*, Taktgeber, Befehlszähler)
- **Speicher** (*memory*) speichert sowohl Programme als auch Daten
- **Eingabe-/Ausgabe-Einheit**
- **Bus-System**

Bei der Von-Neumann-Architektur werden im Speicher sowohl Programme wie auch Daten abgelegt.

Der Von-Neumann-Rechner arbeitet sequentiell d.h. der nächste Befehl kann erst verarbeitet werden, wenn die Verarbeitung des ersten Befehls abgeschlossen ist. Er arbeitet nach folgenden Regeln:

Prinzip des gespeicherten Programms:

- Bei der Von-Neumann-Architektur sind die Befehle in einem RAM-Speicher mit durchgehendem (linearem, eindimensionalem) Adressraum abgelegt.
- Ein Befehls-Adressregister (Befehlszähler, Programmzähler) zeigt auf den momentan auszuführenden Befehl.
- Befehle können so geändert werden wie Daten.

Prinzip der sequentiellen Programmausführung:

- Befehle werden aus einer Zelle des RAM-Speichers gelesen, dekodiert und dann ausgeführt (*fetch*, *decode*, *execute*).
- Erfolgt keine Verzweigung, so wird der Inhalt des Befehlszählers um den Wert Eins erhöht (inkrementiert).
- Es existieren ein oder mehrere Sprung-Befehle, die den Inhalt des Befehlszählers um einen anderen Wert als +1 verändern.
- Es existieren ein oder mehrere Verzweigungsbefehle, die in Abhängigkeit vom Wert eines Entscheidungsbits den Befehlszähler inkrementieren oder einen Sprung-Befehl ausführen.

Der Nachteil des "Von-Neumann-Rechners" ist der Flaschenhals für die Daten, der zwischen CPU und Speicher auftritt. Seit Mitte der 90er Jahre des 20. Jahrhunderts stieg die Geschwindigkeit der CPUs schneller als die der verwendeten Speicherbausteine (RAM) und der übertragenden Busse. Die CPU wurde nicht mehr ausreichend schnell mit Daten versorgt. In der Praxis versucht man diesen Effekt durch die Nutzung von Datencaches abzuschwächen.

Harvard-Architektur (http://de.wikipedia.org/wiki/Harvard-Architektur)

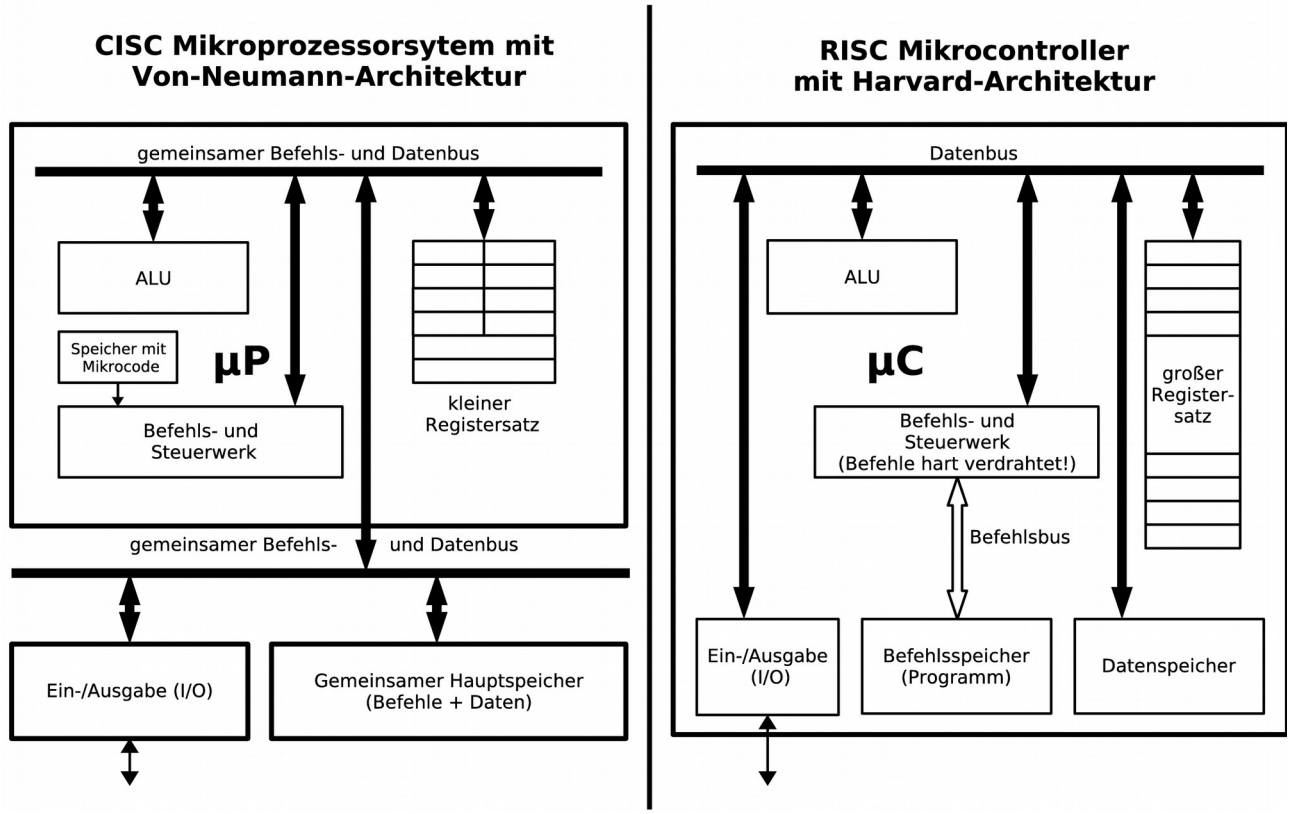
Bei der Harvard-Architektur sind Befehlsspeicher und Datenspeicher physisch voneinander getrennt und beide werden über getrennte Busse angesteuert.

Vorteile der Harvard-Architektur:

- Befehle und Daten können gleichzeitig geladen bzw. geschrieben werden. Bei der Von-Neumann-Architektur sind dazu mindestens zwei aufeinander folgende Buszyklen notwendig. Mit der Harvard-Architektur lassen sich besonders schnelle CPUs, Mikrocontroller und Signalprozessoren realisieren.
- Moderne Prozessoren in Harvard-Architektur sind in der Lage, parallel mehrere Rechenwerke gleichzeitig mit Daten und Befehlen zu füllen.
- Durch die Trennung von Befehl- und Datenspeicher kann die Datenwortbreite und Befehlswortbreite unabhängig festgelegt werden. Dadurch kann z.B. die Effizienz des Programmspeicherbedarfs in Mikrocontroller-Systemen verbessert werden, da sie nicht direkt von den Datenbusbreiten abhängig ist, sondern ausschließlich vom Befehlssatz.
- Die gefürchteten Pufferüberläufe, die für die meisten Sicherheitslöcher in modernen Systemen verantwortlich sind, werden bei stärkerer Trennung von Befehlen und Daten besser beherrschbar.

Die Harvard-Architektur wird überwiegend in RISC-Prozessoren konsequent umgesetzt.

Bemerkung: Die Bezeichnung Harvard-Architektur wird auch auf Prozessoren in herkömmlichen von-Neumann-Systemen angewendet, deren Prozessoren zwar einen gemeinsamen Hauptspeicher für Code und Daten verwenden, diese jedoch in unterschiedlichen Caches platzieren.



Aufbau eines Mikrocontrollersystems

Außer dem Mikrocontroller werden im Mikrocontrollersystem nicht unbedingt weitere Komponenten benötigt. Reicht der interne Oszillator nicht aus, so kann ein externer Quarz angeschlossen werden. Auch ist es natürlich möglich den Speicher durch externe Bausteine zu vergrößern oder zusätzliche Schnittstellenbausteine zum Anschließen der Peripherie zu verwenden.

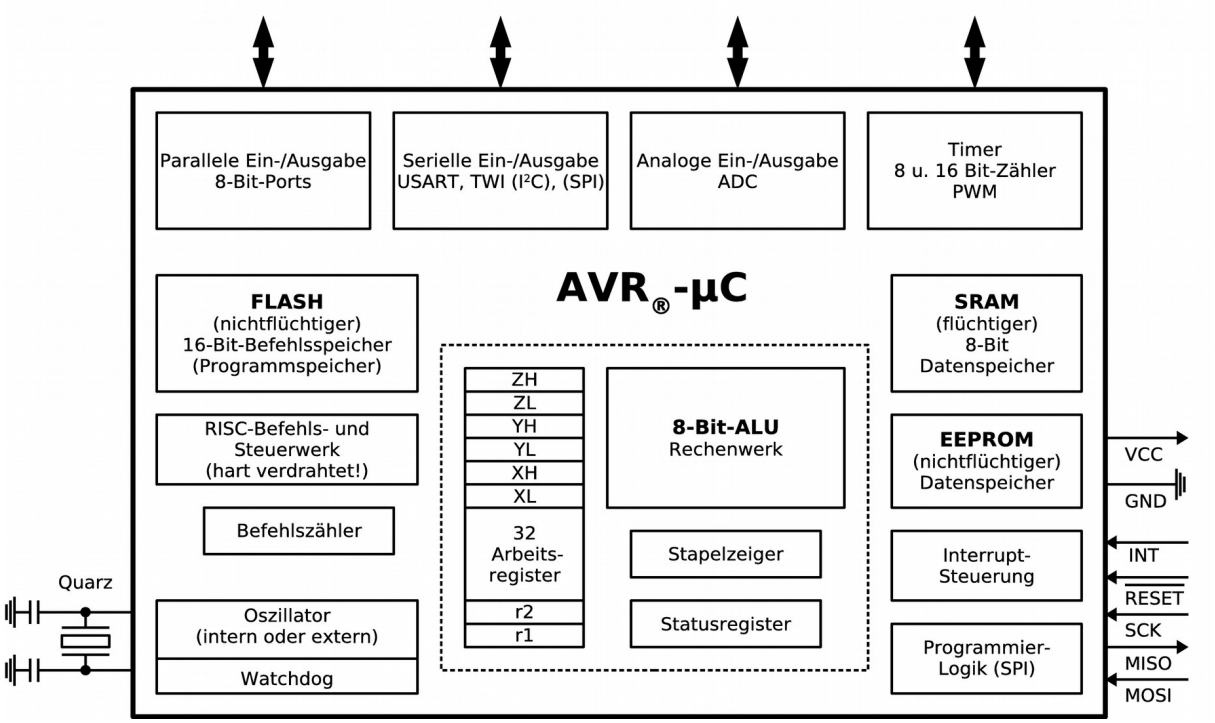
Die meisten Mikrocontroller können im System, d.h. ohne den Baustein aus seiner Schaltung zu entfernen, programmiert werden (ISP[8]). Dazu wird meist ein externes Programmiergerät verwendet, das mit der parallelen, seriellen oder USB-Schnittstelle eines PC verbunden wird.

Ein Teil des internen Mikrocontrollerspeichers (Flash) kann auch für ein so genanntes "Bootloader-Programm" verwendet werden. Das Programmiergerät ist dann nicht mehr unbedingt erforderlich. Der Mikrocontroller kann dann über einen Schnittstellenbaustein direkt mit der Schnittstelle des PC verbunden werden. Die neueren Chips von ATMEL® sind mit einem solchen Bootloader versehen.

8 In System Programming

Der Mikrocontroller (µC)

Der klassische Mikrocontroller besitzt ein schnelles RISC-Design und ist in der Harvard-Architektur aufgebaut. Dies trifft auch auf die 8-Bit-AVR®-Familie der Firma ATMEL® zu.



Im Gegensatz zum Computer sind beim Mikrocontrollersystem alle Blöcke der zentralen Verarbeitungsanlage in einem IC integriert.

Im folgenden soll der Aufbau eines Controllers anhand von Bausteinen der AVR®-Familie untersucht werden. Im Anhang befinden sich die beiden Blockschaltbilder des ATmega8A und des ATmega32A.

Wir unterscheiden folgende Blöcke:

Speicher: Der Speicher teilt sich in nichtflüchtigen[9] FLASH-Befehlsspeicher (16-Bit) und EEPROM-Datenspeicher (8-Bit) sowie in flüchtigen[10] SRAM-Datenspeicher (8-Bit) auf. Im nächsten Kapitel wird detailliert auf den Speicher eingegangen.

CPU: Sie besteht aus dem 8-Bit-Rechenwerk (ALU) mit Statusregister, dem Stapelzeiger und 32 8-Bit Arbeitsregistern (Arbeitsspeicher). Sechs dieser Register (r26-r31) können als Doppelregister (16-Bit) X, Y und Z adressiert werden. Die Arbeitsregister befinden sich im flüchtigen SRAM; ihr Inhalt bleibt also nicht erhalten.

9 Der Inhalt des Speichers bleibt nach dem Ausschalten erhalten. Nichtflüchtiger Speicher kann mittels des Programmiergeräts beschrieben werden.

10 Der Inhalt ist nach dem Einschalten unbestimmt! Das bedeutet, dass jede Speicherzelle nach dem Einschalten einen beliebigen Wert zwischen `0` und `0xFF` enthalten kann.

Befehlswerk: Das RISC-Befehls- und Steuerwerk mit dem Befehlszähler (Programmzähler PC) arbeitet die meist 16-Bit breiten Befehle oft in einem einzigen Takt ab. Das bedeutet, dass mit einem 16 MHz-Quarz fast 16 MIPS (*Million Instructions per Second*) verarbeitet werden können. Das ist ungefähr soviel wie ein Intel 80386 Prozessor um 1990 schaffte.

Ein-/Ausgabe: Die Peripherie wird über die Ein-/Ausgabe-Einheiten angesteuert. Je nach Größe des Controllers sind mehr oder weniger der folgenden Funktionen enthalten:

- 1-4 parallele digitale Ein- und Ausgänge (8-Bit-Ports)
- (mehrere) 8- und 16-Bit-Timer
- RTC (*Real Time Clock*) mit eigenem Quarz
- mehrere PWM (*Pulse Wide Modulation*)-Ausgänge
- serielle asynchrone Schnittstelle (EIA232 (RS232) bzw. V.24) mit dem UART (*Universal Asynchronus Receiver and Transmitter*)
- serielle synchrone Schnittstelle mit dem USART (*Universal Synchronous and Asynchronous serial Receiver and Transmitter*)
- eine serielle Master/Slave-Schnittstelle SPI (*Serial Peripheral Interface*) die oft zur Programmierung des FLASH benutzt wird
- eine serielle Schnittstelle TWI (*Two-Wire serial Interface*) die einer I^2C-Schnittstelle entspricht, aber wahrscheinlich aus rechtlichen Gründen anders genannt wird.
- mehrkanalige (8) 10-Bit Analog/Digital-Wandler

Oszillator: Der Controller kann ohne externen Quarz mit seinem internen kalibrierten RC-Oszillator mit 1, 2, 4 oder 8 MHz betrieben werden. Mit externem Quarz sind 16MHz möglich. Sollen Timer nicht mit der Betriebsfrequenz laufen, so kann ein zusätzlicher Quarz für die Timerfunktionen angeschlossen werden. Ein programmierbarer Watchdog-Timer (Wachhund) mit eigenem integriertem RC-Oszillator verhindert auf Wunsch, dass ein Hängenbleiben des Programms nicht erkannt wird.

Interrupts: Eine vielfältige Interruptsteuerung mit externen und internen Interrupts ist möglich. Auch kann der Controller in bis zu sechs unterschiedliche Schlafmodi versetzt werden.

Prog.-Logik: Über eine sechs adrige SPI-Schnittstelle kann man den Controller "*in-system*[11]" elektrisch programmieren und wieder löschen (FLASH, EEPROM, *Fuse- und Lock-Bits*[12]).

Alle Mikrocontroller der AVR®-Familie sind gleich aufgebaut. Sie unterscheiden sich hauptsächlich in der Größe des Speichers und ihrer Ausstattung (Anzahl der Ports, Anzahl der Timer, ADC, RTC ...).

Hier eine kleine Auswahl von Mikrocontrollern der Firma ATMEL.

11 "Im System": Ohne dass der Chip aus der Schaltung ausgebaut werden muss!

12 Mit diesen programmierbaren Bits kann zum Beispiel der Controller fürs Auslesen gesperrt werden oder der externe Quarzoszillator aktiviert werden (siehe später).

Device	Flash (KiB)	EEPROM (KiB)	SRAM (Byte)	Max I/O Pins	F.max (MHz)	16-bit Timers	8-bit Timer	PWM (ch.)	RTC	UART	I2C	ISP	10-bit A/D (ch.)	Analog Comp.	Watch-dog	On Chip Osc.	Inter-rupts	Ext. Int.	Self Progr.
ATtiny11	1	--	--	6	6	--	1	--	--	--	--	--	--	Yes	Yes	Yes	4	1	--
ATmega8A	8	0.5	1024	23	16	1	2	3	Yes	1	Yes	Yes	8	Yes	Yes	Yes	18	2	Yes
ATmega8535	8	0.5	512	32	16	1	2	4	--	1	Yes	Yes	8	Yes	Yes	Yes	20	3	Yes
ATmega16A	16	0.5	1024	32	16	1	2	4	Yes	1	Yes	Yes	8	Yes	Yes	Yes	20	3	Yes
ATmega32A	32	1	2048	32	16	1	2	4	Yes	1	Yes	Yes	8	Yes	Yes	Yes	19	3	Yes
ATmega64	64	2	4096	54	16	2	2	8	Yes	2	Yes	Yes	8	Yes	Yes	Yes	34	8	Yes
ATmega128	128	4	4096	53	16	2	2	8	Yes	2	Yes	Yes	8	Yes	Yes	Yes	34	8	Yes

Weitere Eigenschaften von Controllern der AVR®-Mikrocontroller Familie:

- Systemtakt bis 20 MHz
- Betriebsspannungsbereich von 1,8 V bis 5,5 V
- Sechs unterschiedliche Gehäuseformen
- Erweiterter Befehlssatz mit Hardware- Multiplikationsbefehlen
- Neu- und Umprogrammierung während des Betriebs (Bootloader)
- JTAG-Interface zum Anschluss von Testsystemen (Debuggen)

Bemerkung: Anfang 2008 erschienen die neue ATxmegas der AVR®-Familie. Vorteile der ATxmegas ist die höhere Taktfrequenz bis 32 MHz, der geringere Stromverbrauch, *Direct Memory Access* (DMA), eine Crypto-Engine für AES und DES und ein neues *Event*-System. Es gibt allerdings keine ATxmegas in DIL-Bauform und die maximale Spannung liegt bei 3,6 V.

Der interner Speicher des ATmega32A

Der Programm- oder Befehlspeicher (ROM, Flash-EPROM[13])

Der Flash-EPROM (Flash) lässt sich vom Anwender mittels eines Programmiergeräts "*in-system*"[14] elektrisch programmieren und wieder löschen. Der Inhalt des Flash bleibt nach dem Abschalten erhalten und so stehen die programmierten Befehlssequenzen und die im Programm enthaltenen konstanten Daten gleich nach dem Einschalten zur Verfügung. ATMEL® garantiert, dass der Flash mehr als 10000 mal beschrieben werden kann.

13 *Erasable Programmable Read-Only Memory*

14 Die Programmierschnittstelle wird als ISP-Schnittstelle bezeichnet. Die Abkürzung ISP steht für *In-System Programming*

Die Befehle sind alle 2 Byte (1 Wort) groß[15] und werden meist in einem einzigen Taktzyklus abgearbeitet.

Der Programmspeicher (ROM, Flash) wird wortweise (2 Byte) adressiert! Die Flash-Adressen sind also Wort-Adressen!

Nach dem Einschalten beginnt das Programm mit dem ersten Befehl auf der Adresse **`0x0000`**. Hier befindet sich meist ein Sprungbefehl zur eigentlichen ersten Adresse des Programms, da sich ab **`0x0000`** die Einsprungadressen für Interrupt-Routinen befinden.

Konstante Programmdaten können auch im Flash abgelegt werden und hier ist dann sogar eine byteweise Adressierung über Indexregister möglich.

Beim ATmega32A beträgt der Flash 32 KiB, also 16Ki-Worte!
Beim ATmega8A sind es 8 KiB also nur 4 Ki-Worte.

⌂ **A100**

a) Wie viel Bit muss der Adresszähler besitzen um die 32 KiB des ATmega32A zu adressieren? Berechne die höchste Adresse im Speicher (Bezeichnung "**`FLASHEND`**"[16]).

b) Wie viel Bit muss der Adresszähler besitzen um die 8 KiB des ATmega8A zu adressieren? Berechne die höchste Adresse im Speicher (**`FLASHEND`**).

Der Arbeits- oder Datenspeicher

Der Arbeitsspeicher teilt sich in einen flüchtigen schnellen SRAM-Speicher und in einen nichtflüchtigen EEPROM[17]-Speicher auf. Im EEPROM bleiben die Daten auch nach dem Abschalten erhalten. Beide werden im Gegensatz zum Programmspeicher byteweise adressiert.

Der Datenspeicher (RAM, SRAM) wird byteweise adressiert! Die RAM-Adressen sind also Byte-Adressen!

Programmspeicher (Flash)
wortweise Adressierung

Adresse:		
0x0106		
0x0105		
0x0104		
0x0103		
0x0102	0xBA	0x98
0x0101	0x76	0x54
0x0100	0x32	0x10

Datenspeicher (SRAM)
byteweise Adressierung

Adresse:	
0x0106	
0x0105	0xBA
0x0104	0x98
0x0103	0x76
0x0102	0x54
0x0101	0x32
0x0100	0x10

15 Einige Ausnahmen benötigen mehr als 16 Bit und werden auf mehrere Adressen aufgeteilt.

16 Da man nicht alle wichtigen Adressen als Zahlenwert behalten kann, und diese Werte auch noch von Controller zu Controller unterschiedlich sind wurden sie mit Namen versehen. Die Zuweisung Name = Adresse erfolgt in der jeweiligen Definitionsdatei "**`*.inc`**" des Controller (siehe später).

17 *Electrically Erasable Programmable Read-Only Memory*

Werden Worte (2 Byte) im RAM abgelegt, so steht das niederwertige Byte (LByte) an der niedrigen Adresse, das höherwertige Byte (HByte) an der höheren Adresse.

Diese Art der Adressierung wird auch Little-Endian (Klein-Endender zuerst) oder Intel-Format genannt.

Das statische SRAM (Static Random Access Memory)

Der statische Schreib/Lese-Speicher ist flüchtig. Nach dem Abschalten sind die Daten verloren. Beim Einschalten ist sein Inhalt undefiniert! Er dient dazu die veränderlichen Daten während dem Betrieb aufzunehmen.

Die Arbeitsregister (`r0-r31`)

Auf den ersten 32 Adressen (**`0x00-0x1F`**) des SRAM liegen die **32 Arbeitsregister** (**`r0-r31`**). Mit ihnen lassen sich alle arithmetische und logische Operationen über die ALU[18] durchführen. Register 0 bis 15 (**`r0-r15`**) erlauben leider keine unmittelbare Adressierung und werden deshalb weniger oft verwendet als die Register 16 bis 31 (**`r16-r31`**). Register 24 bis 31 lassen sich zusätzlich auch als 16-Bit-Register (Doppelregister) verwenden. Register 26 bis 31 können als Adresszeiger (*Pointer*) bei der indirekten Adressierung eingesetzt werden und werden mit **`X`** (**`r27`**:**`r26`**), **`Y`** (**`r29`**:**`r28`**) und **`Z`** (**`r31`**:**`r30`**) bezeichnet.

Die SF-Register (special function register, Sonderfunktions-Register)

Dann folgen die **64 "*Special-Funktion*"-Register** (SF-Register). Sie dienen dazu die interne und externe Peripherie anzusteuern (zum Beispiel: **`PORTD`**) und zu initialisieren. Einzelne SF-Register sind dabei Datenregistern (I/O-Register), andere dienen als Steuerregister[19]. Sie befinden sich auf den Adressen **`0x20-0x5F`** (siehe Anhang). Um die SF-Register zu adressieren benutzt man am einfachsten die von ATMEL® in den Definitionsdateien angegebenen Bezeichner.

Der Stapelspeicher

Auf der höchsten Adresse des SRAM-Speichers (**`RAMEND`** in den Definitionsdateien) wird der Stapel zur Rettung von Variablen und Rücksprungadressen beim Aufruf von Unterprogramm- und Interrupt-Routinen initialisiert (Kapitel Stapelspeicher). Der Stapel läuft dann zu niedrigeren Adressen hin und darf die anderen Daten nicht überschreiben.

Beim ATmega32A hat der SRAM-Speicher eine Größe von 2 KiB (ohne die 32 Arbeitsregister und die 64 SF-Register!!)!
Beim ATmega8A ist es 1 KiB.

⌂ **A101** Berechne die höchste SRAM-Adresse (**`RAMEND`**) für den ATMEGA32A und für den ATmega8A.

18 *Arithmetic Logic Unit* (Arithmetisch-Logische-Einheit)

19 Leider werden im Datenblatt öfters alle SF-Register als I/O-Register bezeichnet, was zu Verwirrungen führen kann.

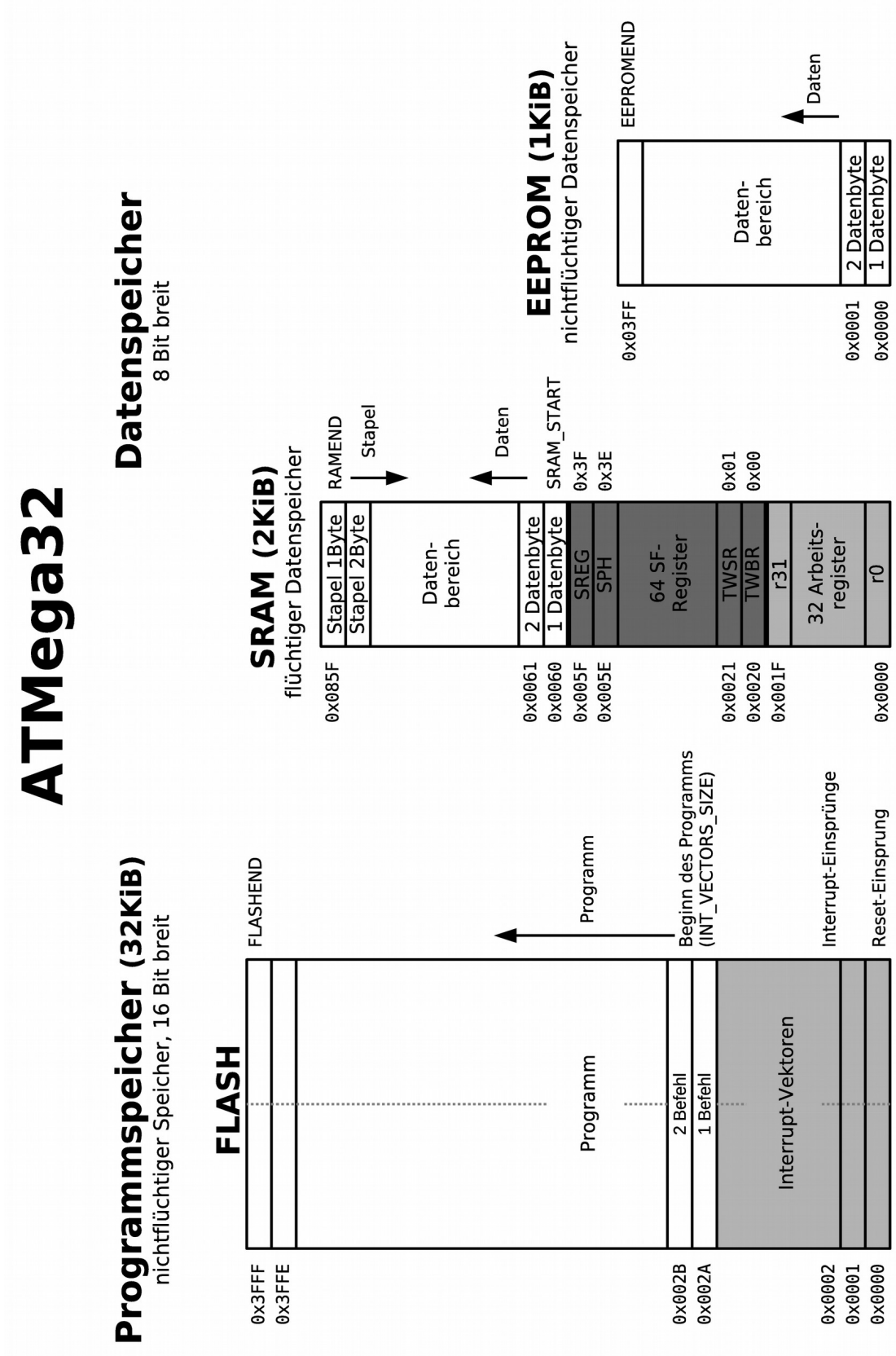
Der nichtflüchtige Speicher (EEPROM)

Das EEPROM lässt sich entweder über das Programmiergerät oder während des laufenden Betriebs programmieren. Es kann mindestens 100000mal beschrieben werden. Im Betrieb geschieht dies über eine Steuereinheit und ist dementsprechend langsam. Der EEPROM-Speicher eignet sich also nicht als Arbeitsspeicher. In ihm werden feste Steuerwerte und Parameter aufbewahrt.

Beim ATmega32A hat der EEPROM-Speicher eine Größe von 1 KiB!
Beim ATmega8A sind es 512Byte.

⌂ **A102** Berechne die höchste EEPROM-Adresse (**`EEPROMEND`**) für den ATMEGA32A und für den ATmega8A.

Zusammenfassung:



A2 Assemblerprogrammierung

In diesem Kapitel sollen einige Grundlagen zur Assemblerprogrammierung erläutert werden, und es soll eine Vorlage erstellt werden, die für alle zukünftige Programme verwendet werden kann.

Assembler ist eine Computersprache und gleichzeitig auch die Bezeichnung für ein Übersetzungsprogramm, das diese Sprache in den Maschinencode eines Controllers oder Prozessors überführt. Für jede Prozessor- oder Controller-Familie gibt es eigene Assembler-Übersetzungsprogramme, die meist vom Hersteller bereitgestellt werden.

Assembler ist keine Hochsprache wie Pascal, Fortran, Basic oder C sondern eine sehr maschinennahe Sprache. Sie kennt praktisch nur einzelne Maschinenbefehle, ermöglicht es aber mit Hilfe von Unterprogrammen, Makros, bedingter Assemblierung und Include-Dateien strukturierten und übersichtlichen Code zu gestalten. Assembler ist besonders gut geeignet um den Aufbau und die Funktionsweise eines Prozessors oder Controllers kennen zu lernen und zu verstehen.

Um den Kurs übersichtlich zu halten werden im folgenden allerdings weder Makros noch bedingte Assemblierung verwendet. Eine Ausnahme sind die fertigen Bibliotheken, die auf der Homepage (http://weigu.lu/a/asm) bereitstehen.

Zum besseren Verständnis der Programme werden Flussdiagramme zu den Programmen erstellt.

Dabei ist das Erstellen der Flussdiagramme immer der erste Schritt vor dem Schreiben des Programmcodes.

⌂ **A200** Arbeite das Kapitel "**Das Flussdiagramm**" im Anhang durch.

Ein Assemblerprogramm ist eine reine Textdatei (Quelle). Diese kann mit jedem beliebigen Texteditor erstellt werden (Notepad, Word, Kate...). Natürlich kann man auch den im Studio 4 von ATMEL® enthaltenen Editor verwenden (Kapitel "Das erste Programm" im Anhang).

Der AVR-Assembler unterscheidet nicht zwischen Klein- und Großschrift.

```
        ldi     Tmp1,0xFF       ;DDRD = 11111111b
        out     DDRD,Tmp1       ;alle Bits im Datenrichtungsregister auf Eins
;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   ser     Tmp1            ;PORTD = 11111111b
        out     PORTD,Tmp1      ;Alle PortD-Pins auf High setzen (LEDs ein)
```

Der Assembler überprüft die Syntax und übersetzt den Quellcode, wenn kein Fehler gefunden wurde, in die Maschinensprache (Hexcode). Zusätzlich können Mapdateien (**.map**), Listdateien (**.lst**) und Objectdateien (**.obj**) erstellt werden.

Die Listdatei und die Mapdatei sind Textdateien.

Die **Listdatei** listet alle Assembler-Befehle auf, wie sie auf den Controller geladen werden.

```
44: 00002A   EF0F  ldi     Tmp1,0xFF       ;DDRD = 11111111b
45: 00002B   BB01  out     DDRD,Tmp1       ;alle Bits im Datenrichtungsreg. auf 1
46: 00002C   EF0F  ser     Tmp1            ;PORTD = 11111111b
47: 00002D   BB02  out     PORTD,Tmp1      ;Alle PortD-Pins auf High setzen
```

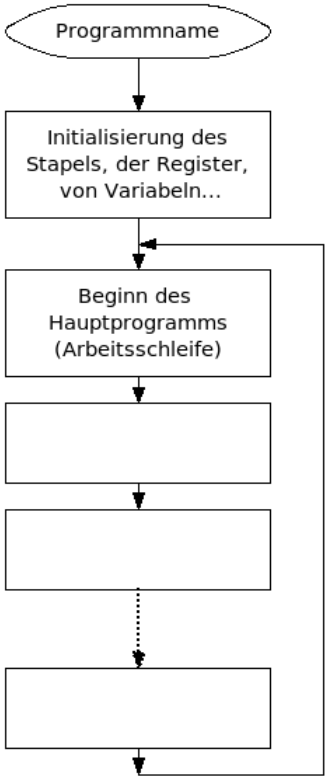
Links steht nach einer Zeilennummer die Adresse, der Maschinencode des Befehls und danach der Maschinencode in Assembler-Darstellung.

Eine **Mapdatei** gibt Auskunft darüber, an welchen Adressen Code und Objekte landen.

```
EQU  INT_VECTORS_SIZE 0000002a
CSEG RESET        00000000
CSEG INIT         0000002a
DEF  Tmp1         r16
CSEG MAIN         0000002c
CSEG END          0000002e
```

Die Datei die für den Programmer benötigt wird ist die **Hexdatei** (**.hex**), in der die Maschinensprache enthalten ist. Diese Datei wird durch den Programmer in den Speicher des ATmega-Controllers geladen.

```
:020000020000FC
:0200000029C015
:0A0054000FEF01BB0FEF02BBFFCF5F
:00000001FF
```



Das nebenstehende Flussdiagramm zeigt den prinzipiellen Ablauf eines Assemblerprogramms. Nach dem Einschalten der Versorgungsspannung beginnt das Programm mit dem ersten Befehl, der sich auf der Adresse **0x0000** (**0x** bedeutet, dass wir das hexadezimale Zahlensystem verwenden) im Programmspeicher (Flash) befindet. Als erstes müssen diverse Initialisierungen vorgenommen werden. Dann folgt die Arbeitsschleife (**MAIN**), welche natürlich auch Verzweigungen, Schleifen und Unterprogramme enthalten kann.

Die Assembler-Programmiervorlage

(A2_template.asm)

Um das Assembler-Grundgerüst nicht immer neu schreiben zu müssen wird hier eine Vorlage bereitgestellt, welche für alle eigenen Programme verwendet werden kann.

Schreibt man ein neues Programm, so wird zuerst die Vorlage geöffnet, und sofort wieder unter dem Namen des zu schreibenden Programms abgespeichert. Nicht benötigte Teile in der Vorlage werden einfach gelöscht.

Die Vorlage reduziert die Tipparbeit, und ermöglicht eine einheitliche Struktur aller Programme. Die vorgegebene Position für Programmteile vermeidet Fehler.

Anhand der Vorlage sollen einige der wichtigsten Assemblereigenheiten erklärt werden:

Eine Eingabezeile im Assembler besteht aus mehreren Feldern. Felder werden durch mindestens ein Leerzeichen getrennt. Es erhöht allerdings die Übersichtlichkeit, wenn statt Leerzeichen Tabulatoren verwendet werden.

```
[Label:]        Direktive oder Befehl [Operanden]      [;Kommentar]
```

Die in eckigen Klammern stehenden Teile können entfallen. Natürlich kann eine Zeile auch ausschließlich aus Kommentar oder einem Label bestehen.

- **Label** sind vom Anwender bestimmte **Namen für Adressen**. Diese **symbolischen Namen** erleichtern das Verständnis des Programms, da sie aussagekräftiger als reine Adressen sind. Der Assembler ersetzt beim Assemblieren die Labels durch die entsprechenden Adressen.

- **Direktiven** sind **Assembleranweisungen**, die dem Assembler mitteilen was dieser beim Assemblieren tun soll. Sie sind nicht zu verwechseln mit den Befehlen.

- **Befehle** mit ihren **Operanden** teilen dem Mikrocontroller mit was dieser tun soll. Der Befehlssatz eines Mikrocontrollers (siehe Anhang) zeigt welche Aktionen dieser Mikrocontroller beherrscht. Ein Befehl besteht aus dem Opcode und den Operanden. Es gibt Befehle ohne Operand, welche wo nur ein Operand benötigt wird und Befehle mit zwei Operanden. Bei zwei Operanden steht immer zuerst das Ziel (links) und dann die Quelle (rechts)!!

  Befehl Ziel,Quelle

  Beispiel: `ldi r16,0x0A ;lade 10 dez. ins Arbeitsregister 16`

- **Kommentare** sind wichtige Ergänzungen die zum Verständnis des Programms beitragen. Sie beginnen mit einem Strichpunkt!

Besonders die **Assembler-Direktiven** ermöglichen uns übersichtliche und verständliche Assemblerprogramme zu schreiben.

Direktiven werden nicht in Maschinencode übersetzt (Pseudo-Befehle!), sondern dienen nur der Steuerung des Übersetzungsvorgangs. Sie beginnen mit einem Punkt und stehen meist gleich am Anfang der Zeile.

Folgende Direktiven werden verwendet:

Direktive	Operand	Beschreibung
.LIST		Listing-Ausgabe einschalten (*default*). Der produzierte Code wird von Menschen lesbar in einer ***.LST**-Datei ausgegeben.
.NOLIST		Listing-Ausgabe ausschalten.
.INCLUDE	"**Textdatei**" oder <**Textdatei**>	Fügt eine externe Textdatei ein (als ob deren Inhalt an dieser Stelle stünde). Es kann sich hierbei z.B. um einen Programmteil in Assembler, ein Unterprogramm oder eine Definitionsdatei (Header-Datei) mit zusätzliche Direktiven sein. Beispiel: **.INCLUDE "m32def.inc"**
.DEVICE	Bausteintyp	Definiert den Bausteintyp. Wird nicht benötigt, da schon in der Definitionsdatei enthalten.
.DEF	Name = Register	Definiere (*define*) einen Namen für ein Arbeitsregister (Variable!, r0 bis r31). Beispiel: **.DEF Tmp1 = r16**
.EQU	Name = Ausdruck	Definiert eine Konstante mit Namen. Dieser Name ist dann nicht mehr veränderbar. Beispiel: **.EQU Clock = 1000**
.SET	Name = Ausdruck	Definiert eine Konstante mit Namen. Dieser Name ist innerhalb des Programms (Übersetzungsvorgangs) veränderbar. Beispiel: **.SET Value = 500 ;alter Wert** **.SET Value = 200 ;neuer Wert**
.ORG	Adresse	Legt eine Anfangsadresse fest ab der der folgende Code abgespeichert wird. Hiermit kann der Speicher organisiert werden. Beispiel: **.ORG 0xA00**
.EXIT		Ende des Quelltextes

Weiter Direktiven für die Organisation des Speicher-Bereichs:

Direktive	Operand	Beschreibung
.CSEG		Beginn des Codesegmentes. Alles Folgende wird als Code übersetzt und im Flash gespeichert. Mit den **.DB** und **.DW**-Direktiven können Variablen im **FLASH** abgelegt werden.
.DSEG		Beginn des Datensegmentes. Mit der **.BYTE**-Direktive und eventuell symbolischen Namen (Label) wird hier der **SRAM**-Speicher organisiert.
.ESEG		Beginn des EEPROM-Segments. Mit der **.DB** und **.DW**-Direktive werden Variablen im **EEPROM** abgelegt.
.DB	`Liste mit Bytekonstanten`	(engl.: „define Byte“) Fügt konstante Bytes ein. Dabei ist es die Bedeutung der Bytes egal (Zahl von 0..255, ASCII-Zeichen 'b', eine Zeichenkette "Hallo"; alle Bytes werden durch Kommas getrennt). Im Flash muss eine gerade Zahl von Bytes eingefügt werden (16 Bit-Worte), sonst hängt der Assembler ein Nullbyte an. Beispiel: **`.DB 5,0xA0,'H',0b11101110,"T3EC"`**
.DW	`Liste mit Wortkonstanten`	(engl.: „define Word“) Fügt konstantes binäres Wort (16 Bit) ein. Im EEPROM und Flash zuerst das niederwertige Byte, dann das höherwertige Byte.
.BYTE	`Anzahl`	Reserviert Speicherplatz (Bytes) im SRAM (Datensegment). Beispiel: **`.BYTE 5`**

Um die Übersichtlichkeit weiter zu erhöhen sollen hier einige **Abmachungen** getroffen werden, an die man sich für diesen Kurs weit möglichst halten soll, auch wenn Assembler nicht zwischen Klein- und Großschrift unterscheiden.

- Für Labels und Direktiven werden Großbuchstaben verwendet. Für die Labels verwenden wir englische Abkürzungen. Ein Label muss mit einem Buchstaben beginnen.
- Für Befehle werden nur Kleinbuchstaben verwendet.
- Variablen (Register) oder Konstanten werden öfter mit einem aussagekräftigem Namen versehen (mittels **.DEF**, **.EQU** oder **.SET**). Dieser Name soll mit einem Großbuchstaben beginnen. Er kann auch Kleinbuchstaben enthalten. Vorzugsweise sollen englische Bezeichner verwenden.
- Es sollen keine Sonderzeichen verwendet werden. Auch nicht in den Kommentaren, da öfter Probleme mit dem Zeichensatz auftreten, besonders wenn man mit unterschiedlichen Betriebssystemen arbeitet.

Zuerst der **Programmkopf** der Vorlage. Dieser besteht aus Kommentaren und dient der Dokumentation des Programms. Jede Kommentarzeile beginnt mit einem Semikolon (Strichpunkt).

Es ist im ureigenen Interesse des Programmierers Programme ausführlich zu dokumentieren. Der Mensch ist äußerst vergesslich und schon nach wenigen Wochen können Programmteile, welche heute sonnenklar erscheinen schon wieder unverständlich sein. Auch ermöglicht erst eine detaillierte Dokumentation anderen Programmierern den Code eines Programms ohne allzu viel Aufwand verstehen zu können.

```
;*****************************************************************************
;*
;*      Titel:  Programmiervorlage (A2_template.asm)
;*      Datum:  08/01/08        Version:        0.4
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor: ATmega32                     Quarzfrequenz:
;*      Eingaenge:
;*      Ausgaenge:
;*
;*      Informationen zur Funktionsweise:
;*
;*****************************************************************************
```

Als nächstes folgt das Einbinden einer **Definitionsdatei** zum verwendeten Baustein des Herstellers:

```
;-----------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten
```

Die **.NOLIST**-Direktive verhindert, dass die ***.LST**-Datei durch den langen Text der Definitionsdatei unübersichtlich wird. Diese Direktive ist natürlich nicht zwingend notwendig. Die **.LIST**-Direktive schaltet nach der den List-Output nach der **.INCLUDE** -Direktive wieder ein.

Mit **.INCLUDE** kann eine Textdatei eingebunden werden. Befindet sich die Datei nicht im gleichen Verzeichnis wie das Assemblerprogramm, so muss der Pfad mit angegeben werden.

Die hier verwendete Definitionsdatei "**m32def.inc**" wird von ATMEL geliefert (befindet sich im Verzeichnis C:\Program Files\Atmel\AVR Tools\AvrAssembler2\Appnotes) und erleichtert durch viele **.EQU**-Direktiven die Programmierung erheblich, da zum Beispiel für die SF-Register keine hexadezimale Adressen mehr benötigt werden, sondern die im Datenblatt verwendeten Abkürzungen (für den ATmega 8 wird die Datei "**m8def.inc**" benötigt).

Hier ein kleiner Auszug aus der verwendeten Datei:

```
; ***** SPECIFY DEVICE ************************************************
.device ATmega32
...
; ***** I/O REGISTER DEFINITIONS ***************************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ    SREG    = 0x3f
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
.equ    OCR0    = 0x3c
.equ    GICR    = 0x3b
.equ    GIFR    = 0x3a
.equ    TIMSK   = 0x39
...
```

⌂ **A201** Schau die Datei "**m32def.inc**" mit einem Texteditor an.

Weiter mit der Vorlage:

```
;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
```

```
;-------------------------------------------------------------------------------
;.DSEG                          ;was ab hier folgt kommt in den SRAM-Speicher
;TAB1:  .BYTE  100              ;100 Byte grosse Tabelle im Datensegment
```

Wird der SRAM (Datenspeicher) zum Abspeichern von Variablen oder Tabellen genutzt, so sollte dieser gleich am Anfang des Assemblerprogramms organisiert werden. Dies erhöht die Übersichtlichkeit des Programms (falls diese Zeilen benötigt werden, muss der Strichpunkt natürlich gelöscht werden).

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                          ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                 ;Programm beginnt an der FLASH-Adresse 0x0000
RESET1:  rjmp    INIT          ;springe nach INIT (ueberspringe ISR Vektoren)
```

Ab der **.CSEG**-Direktive erfolgt der Programmcode.

Beim Anlegen der Betriebsspannung an den Controller, oder nach Ablauf der "Watchdog"-Zeit wird ein **Reset** ausgelöst. Dabei wird der Programmzähler auf Null gesetzt. Ab dieser Adresse wird mit der Verarbeitung von Programmcode begonnen. Mit "**.ORG 0x0000**" wird die Startadresse festgelegt (Label "**RESET1:**").

Danach werden mehrere Adressen übersprungen, die für Interruptvektoren vorgesehen sind (Kapitel "Interrupt- und Systemsteuerung"). Dies geschieht mit dem Assemblerbefehl **rjmp**.[20]

rjmp k

Unbedingter (ohne Bedingung) relativer Sprung zu einer Adresse (*relative jump*).
(k steht für die Adresskonstante, 1 Wort-Befehl (2 Byte))

1	1	0	0	k	k	k	k	k	k	k	k	k	k	k	k

Relative Sprungbefehle können nur 2 KiWorte vor und rückwärts springen, was meist ausreicht. Sie sind schneller als nicht-relative Sprünge (Bsp.: jmp, 2 Worte, 3 Taktzyklen).
Beeinflusste Flags: keine Taktzyklen: 2

Der relative Sprungbefehl bewirkt, dass erst mit dem Code ab Label "**INIT**:" weitergearbeitet wird. Wird mit Interrupts gearbeitet, so werden in diesem Zwischenbereich die Sprungadressen zu den Interrupt-Behandlungsroutinen organisiert (siehe kommentierte Zeilen).

```
;-------------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-------------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
;.ORG   INT0addr                ;interner Vektor für INT0 (alt.: .ORG 0x0002)
;       rjmp    ISR_I0          ;Springe zur ISR von INT0

;-------------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-------------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
```

Da der reservierte Platz für Interruptvektoren für jeden Controller verschieden ist, wird hier der Name **INT_VECTORS_SIZE** (Definitionsdatei) benutzt um die reservierten Adressen zu überspringen. Dazu wird vor dem Label **INIT:** eine neue Anfangsadresse festgelegt (mit **.ORG**). Der Label **INIT:** entspricht dann dieser Adresse.

20 Erscheint ein neuer Befehl im Text, so wird dieser mit einem Rahmen kurz vorgestellt. Es ist sinnvoll sich die englischen Bezeichnung der Befehle zu merken.

⌂ **A202** Finde den Ausdruck **INT_VECTORS_SIZE** in der Definitionsdatei für den ATmega8 und den ATmega32. Wie lauten die beiden hexadezimalen Adressen bei denen das eigentliche Programm beginnen kann? Vergleiche mit der Speicherorganisation im Anhang.

```
;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 15 wird zum Rechnen benoetigt
        clr    r15              ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister
.DEF    W = r24

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
        out     SPH,Tmp1                ;datei festgelegt
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1
```

Im Initialisierungs- und Definitionsteil werden mehreren Registern Namen zugewiesen um die Übersichtlichkeit in den Programmen zu erhöhen.

Zum Rechnen wird öfter ein Register mit Null als Inhalt benötigt. **r15** wird dazu mit dem Namen **Zero** versehen und dann mit dem „**clr**“-Befehl auf Null gesetzt.

clr Rd

Lösche Arbeitsregister Rd (*clear register*).

0	0	1	0	0	1	d	d	d	d	d	d	d	d	d	d

Das Register wird mit sich selbst **Exklusiv-ODER (EOR, XOR)** verknüpft und dadurch gelöscht (Rd = 0). Anders als beim Befehl **ldi Rd,0** werden hier die Flags beeinflusst!!
Beeinflusste Flags: S(0), V(0), N(0), Z(1)
Taktzyklen: 1

Zwei Arbeitsregister (**r16** und **r17**) werden als Zwischenspeicher (**Tmp1**, **Tmp2**) reserviert, da oft Zwischenspeicher im Programm benötigt werden, um zum Beispiel die SF-Register mit Konstanten zu laden. Da Zwischenspeicher meist unmittelbar (*immediate*) adressiert werden (Bsp.: **ldi Tmp1,0xAA**) verwendet man keines der ersten sechzehn Arbeitsregister, da diese nicht unmittelbar adressierbar sind (siehe später). Ein Register (**r18**) wird für einen Zähler (**Cnt1**) reserviert. Register **r24** und **r25** können zusammen als universelles Doppelregister verwendet werden. Sie dienen aber auch zur Parameterübergabe bei Unterprogrammen.

Dann erfolgt noch die Initialisierung des Stapels, die immer benötigt wird, sobald man mit Unterprogrammen oder Interrupt-Routinen arbeitet.

Wir verwenden dazu die zwei Assembler-Funktionen (keine Befehle!) **LOW()** und **HIGH()**, welche das niederwertige und das höherwertige Byte der Adresse **RAMEND** liefert. **RAMEND** ist ein Name für die höchste Adresse des SRAM-Speichers. Auf diese wird der Stapelzeiger initialisiert (Kapitel "Stapelspeicher"). Da unterschiedliche Controller auch unterschiedliche Größen von SRAM-Speicherplatz besitzen befindet sich diese Adresse auch wieder in der Definitionsdatei.

Mit dem Befehl zur unmittelbaren Adressierung "**ldi**" (*load immediate*) wird ein Byte der Adresse in den Zwischenspeicher geladen, und dann mit dem "**out**"-Befehl in den Stapelzeiger (2 SF-Register (**SPL**, **SPH**) im SRAM) geschrieben.

ldi Rd,K

Lade unmittelbar eine Konstante K (8 Bit) in ein Arbeitsregister Rd (*load immediate*).

1	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K

Für die unmittelbare Adressierung können nur die **Arbeitsregister 16-31** verwendet werden!! (das d bei Rd steht für "*destination*" (Ziel)).
Beeinflusste Flags: keine Taktzyklen: 1

Der Transferbefehl "**ldi Tmp1,HIGH(RAMEND)**" lädt eine Konstante in das Arbeitsregister "**Tmp1**". Die **Quelle** bei Befehlen steht immer **rechts** und das **Ziel links**!

Der Befehl "**out SPH,Tmp1**" lädt den Inhalt des Arbeitsregisters in das SF-Register **SPH**. Dieses Ein/Ausgaberegister befindet sich auf der Adresse **0x005E** im SRAM, wird jedoch mit seiner eigenen Adresse **0x3E** angesprochen. In Assembler werden jedoch die Abkürzungen aus der Definitionsdatei verwendet, was das Programm wesentlich übersichtlicher gestaltet und die Programmierarbeit vereinfacht.

out P,Rr

Kopiere Inhalt (8 Bit) eines Arbeitsregisters Rr in ein SF-Register (*store register to SF-register*).

1	0	1	1	1	P	P	r	r	r	r	r	P	P	P	P

Die 64 SF-Register (manchmal als *Port* oder *I/O Space* bezeichnet) können nicht unmittelbar angesprochen werden. SF-Register können nur über den Umweg eines der 32 Arbeitsregister beschrieben werden (das r bei Rr steht für "*source*" (Quelle)). .
Beeinflusste Flags: keine Taktzyklen: 1

⌂ **A203** Finde die Ausdrücke **RAMEND**, **SPH** und **SPL** in der Definitionsdatei für den ATmega32. Beschreibe mit diesen Werten wie der Stapel initialisiert wird.

Der Schluss unserer Programmvorlage bietet Platz für das Hauptprogramm sowie das Einbinden von Unterprogrammen, Interrupt-Behandlungsroutinen und Tabellen.

```
;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:
        rjmp    MAIN    ;Endlosschleife

        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
;END:   rjmp    END             ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
;.INCLUDE "lib/SR_TIME_16M.asm"      ;Zeitschleifenbibliothek einbinden

;-----------------------------------------------------------------------------
;       Tabellen im Programmspeicher (Flash)
;-----------------------------------------------------------------------------
;TAB:  .DB    "Hallo"

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

Die Hauptschleife ist üblicherweise eine Endlosschleife. Sollte dies nicht der Fall sein, so kann man noch eine **END**-Zeile an den Schluss setzen.

Werden einige der auskommentierten Teile der Vorlage benötigt, so wird der Strichpunkt vor dem Code gelöscht.

Nicht im Programm benötigt Teile werden vollständig gelöscht.

Hier nochmal die gesamte Programmvorlage:

```
;*****************************************************************************
;*
;*      Titel:  Programmiervorlage (A2_template.asm)
;*      Datum:  08/01/08        Version: 0.4 (27/12/12)
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor: ATmega32                     Quarzfrequenz:
;*      Eingaenge:
;*      Ausgaenge:
;*
;*      Informationen zur Funktionsweise:
;*
;*****************************************************************************

;-----------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;-----------------------------------------------------------------------------
;.DSEG                          ;was ab hier folgt kommt in den SRAM-Speicher
;TAB1:  .BYTE  100              ;100 Byte grosse Tabelle im Datensegment

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET:  rjmp    INIT            ;springe nach INIT (ueberspringe ISR Vektoren)

;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
;.ORG   INT0addr                ;interner Vektor für INT0 (alt.: .ORG 0x0002)
;       rjmp    ISR_I0          ;Springe zur ISR von INT0

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 15 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister
.DEF    W = r24

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
```

```
        ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
        out     SPH,Tmp1                ;datei festgelegt
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:
        rjmp    MAIN    ;Endlosschleife

        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
;END:   rjmp    END             ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
;.INCLUDE "lib/SR_TIME_16M.asm"   ;Zeitschleifenbibliothek einbinden

;-----------------------------------------------------------------------------
;       Tabellen im Programmspeicher (Flash)
;-----------------------------------------------------------------------------
;TAB:  .DB    "Hallo"

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

Bemerkung: Für Einführungskurse oder kleine Programme existiert eine abgespeckte Version der Vorlage mit dem Namen "**A2_template_light.asm**" (http://weigu.lu/a/asm).

A3 Digitale Ein- und Ausgabe

In diesem Kapitel wird die Grundfunktion der I/O-Anschlussstifte (I/O-Pins), die digitale Ein- und Ausgabe, behandelt.

Bemerkungen: Für die folgenden Programme wird meist die parallele 8-Bit-Schnittstelle Port D verwendet, da sie sowohl beim ATmega8A wie auch beim ATmega32A zur Verfügung steht.

Fast alle Pins der Ausgangsports sind doppelt belegt. Zum Beispiel werden beim ATmega32A vier Pins des Port B für die ISP-Schnittstelle (*In System Programming*) benötigt. Benutzt man Port B jetzt zum Beispiel für die Ansteuerung eines LCD-Displays, so können bei der Programmierung mit angeschlossenem Display Probleme auftreten. Das Display muss zur Programmierung abgeklemmt werden. Man muss also immer genau überlegen welche Pins man benutzen will. Dies ist natürlich auch vom Chip abhängig.

⌂ **A300** Ermittle anhand der Datenblätter welche zusätzlichen Funktionen zur digitalen Ein- und Ausgabe die Pins des ATmega32A und des ATmega8A besitzen.

Digitale Daten ausgeben

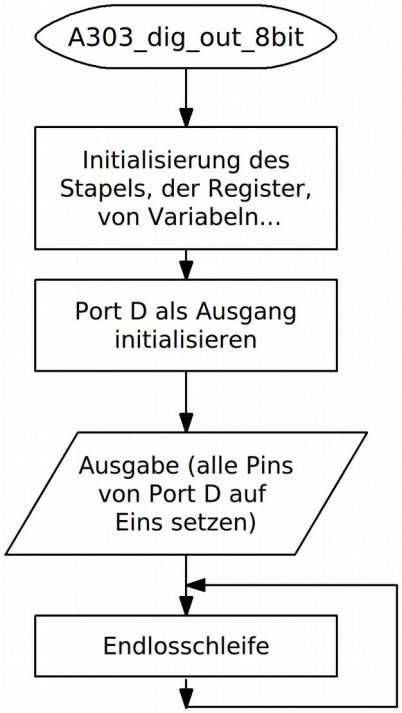
Das erste Programm `(A303_dig_out_8bit.asm)`

Das erste Programm soll acht LEDs gleichzeitig einschalten. Dazu sind acht LEDs (mit entsprechenden Vorwiderständen) mit den acht Pins von Port D zu verbinden.

⌂ **A301**
a) Ermittle aus dem Datenblatt des ATmega32A wie hoch der vom Ausgang gelieferte Strom maximal sein darf?
b) Berechne die Vorwiderstände für eine rote LED, eine gelbe LED und eine grüne LED (Datenblatt oder Produktkatalog!), wenn diese jeweils mit 15 mA betrieben werden sollen. Welche Spannung liefert der ATMega32A bei 15 mA (siehe Datenblatt)?
c) Berechne den Vorwiderstand für eine rote "*low current*" LED, die nur 2 mA benötigt. Achte auf die Spannung (Datenblatt ATmega32A)!

Bemerkung: Die Befehle **`ldi`**, **`out`** und **`rjmp`** wurden schon im vorigen Kapitel besprochen.

Das entsprechende Flussdiagramm sieht folgendermaßen aus:



Folgende Programmzeilen müssen im Initialisierungsteil in der Vorlage hinzugefügt werden:

```
        ;alle Pins von PortD als Ausgang initialisieren
        ldi     Tmp1,0xFF       ;alle Bits im Zwischenspeicher auf Eins
        out     DDRD,Tmp1       ;DDRD = 0b11111111; alle Pins Ausgang
```

Im Datenrichtungsregister für Port D (**DDRD**, ***D**ata **D**irection **R**egister **D***) wird festgelegt ob die einzelnen Pins als Ausgänge oder als Eingänge funktionieren. **Eine Eins bedeutet Ausgang, eine Null Eingang.**

Das Hauptprogramm sieht folgendermaßen aus:

```
;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   ser     Tmp1            ;alle Bits im Zwischenspeicher auf Eins
        out     PORTD,Tmp1      ;PORTD = 0b11111111
                                ;Alle PortD-Pins auf High setzen (8 LEDs ein)
END:    rjmp    END             ;Endlosschleife
```

Im Hauptprogramm werden die Ausgangspins von Port D auf High-Pegel (+5V) gezogen. Dazu müssen alle Bits von Port D auf Eins gesetzt werden.

Der "**ser**"-Befehl (*set register*) setzt das Arbeitsregister auf **0xFF**. Mit dem "**out**"-Befehl wird der Inhalt des Arbeitsregisters dann an das SF-Register **PORTD** gesendet.

ser Rd

Setze alle Bits des Register Rd auf Eins (*set all bits in register*).

1	1	1	0	1	1	1	1	d	d	d	d	1	1	1	1

Entspricht dem Befehl **`ldi Rd,0xFF`**.
Beeinflusste Flags: keine Taktzyklen: 1

⌂ **A302** Wird die Codezeile mit dem "**`ser`**"-Befehl zwingend benötigt? Erkläre!?

⌂ **A303**
a) Erstelle das Programm indem du die Vorlage mit den obigen Zeilen ergänzt.
b) Speichere das Programm unter dem Namen "**`A303_dig_out_8bit.asm`**" ab. Teste dann das Programm einmal mit deiner Hardware und simuliere das Programm ebenfalls im Studio 4 (Anhang "Das erste Programm"!).
c) Miss alle Spannungen und den Strom an einer Diode!

Bemerkungen: An oberster Stelle soll immer die Übersichtlichkeit und Flexibilität eines Programms stehen. Hier wurde der "**`ser`**"-Befehl eingeführt, da auch Ziel dieses Kurses ist einen größtmöglichen Teil der Befehle kennen zu lernen. Es kann aber aussagekräftiger und flexibler sein den Befehl **`ldi Rd,0xFF`** zu verwenden.

Die Codezeile mit dem erneuten Setzen der Bits im Zwischenspeicher (A302) vor der Ausgabe könnte man natürlich weg rationalisieren. Dies ist aber nicht anzuraten, da es die Übersichtlichkeit verschlechtert und schon bei leichten Änderung des Programms (nochmalige Verwendung des Zwischenspeichers für andere Aufgaben) zu schwer auffindbaren Fehlern führen kann.

Im Kurs wird oft die hexadezimale Darstellung gewählt um diese durch häufige Anwendung zu trainieren. Natürlich kann auch die übersichtlichere binäre Darstellung (z.B. **`ldi Rd,0b11111111`**) verwendet werden.

Das gesamte Programm könnte folgendermaßen aussehen:

```
;***********************************************************************
;*
;*      Titel:  Mein erstes Programm (A303_dig_out_8bit.asm)
;*      Datum:  20/02/08        Version:        0.2
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32A       Quarzfrequenz:  intern 1MHz
;*      Eingaenge:      keine
;*      Ausgaenge:      Es sollen acht LEDs mit den acht Pins des PORTD
;*                      verbunden werden (Vorwiderstaende!).
;*
;*      Informationen zur Funktionsweise:
;*      PORTD als Ausgang. Alle LEDs sollen gleichzeitig eingeschaltet werden.
;*
;***********************************************************************

;-----------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------
```

```
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;***************************************************************************
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;***************************************************************************
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET1:  rjmp    INIT           ;springe nach INIT (ueberspringe ISR Vektoren)

;---------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;---------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher

        ;alle Pins von PortD als Ausgang initialisieren
        ldi     Tmp1,0xFF       ;alle Bits im Zwischenspeicher auf Eins
                                ;(alternativer Befehl: ser Tmp1)
        out     DDRD,Tmp1       ;DDRD = 0b11111111; alle Pins Ausgang

;---------------------------------------------------------------------------
;       Hauptprogramm
;---------------------------------------------------------------------------
MAIN:   ser     Tmp1            ;alle Bits im Zwischenspeicher auf Eins
        out     PORTD,Tmp1      ;PORTD = 0b11111111
                                ;Alle PortD-Pins auf High setzen (8 LEDs ein)
END:    rjmp    END             ;Endlosschleife

;***************************************************************************
.EXIT                           ;Ende des Quelltextes
```

⌂ **A304** Schreibe das Programm so um, dass nur die LED an Pin 5 aufleuchtet (neuer Befehl "**sbi**"). Dabei soll auch die Initialisierung so verändert werden, dass alle nicht benötigten Pins als Eingänge konfiguriert sind.
Nenne das Programm "**A304_dig_out_1bit.asm**".

sbi P,b

Setze Bit b im SF-Register auf Eins (*set bit in SF-register*).

1	0	0	1	1	0	1	0	P	P	P	P	P	b	b	b

0≤b≤7. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit). Bei den oberen 32 Registern muss eine Maskierung eingesetzt werden.
Beeinflusste Flags: keine Taktzyklen: 2

cbi P,b

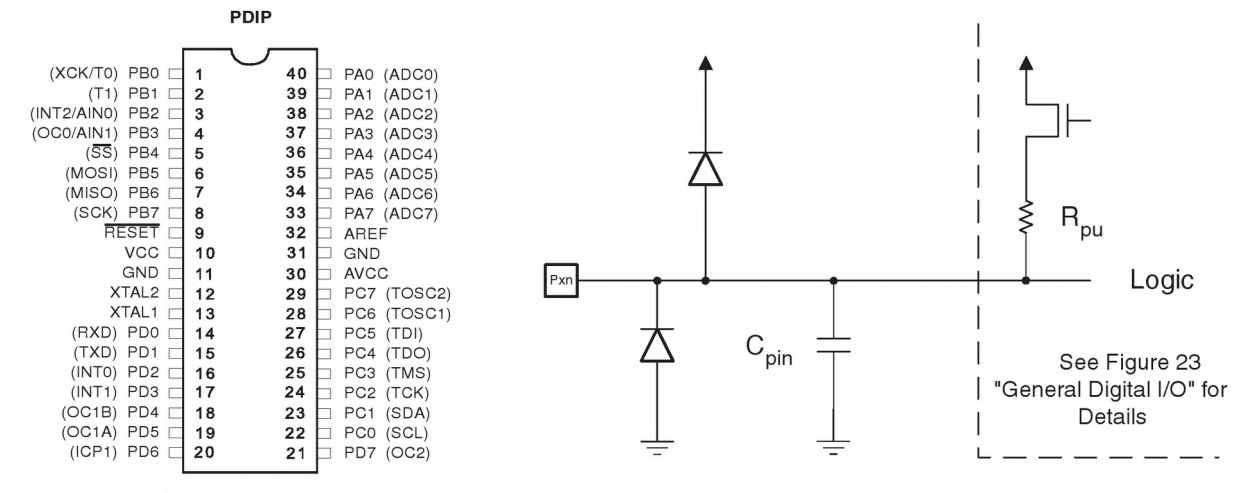
Lösche Bit b im SF-Register (clear *bit in SF-register*).

1	0	0	1	1	0	0	0	P	P	P	P	P	b	b	b

0≤b≤7. Setzt Bit auf Null. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit).
Beeinflusste Flags: keine **Taktzyklen: 2**

Die 4 Ein-/Ausgabeports des ATmega32

Der ATmega32A besitzt 4 Ein-/Ausgabeports. Das macht insgesamt $4 \cdot 8 = 32$ **individuell ansteuerbare** Pins. Jeder dieser Pins kann als Eingang oder Ausgang konfiguriert werden. Als Eingang kann ein interner Pull-Up-Widerstand dazugeschaltet werden. Jeder als Ausgang geschaltete Pin kann genug Strom liefern um eine LED direkt anzusteuern. Die Pins sind mit zwei Dioden gegen Masse und VCC geschützt. Pins werden mit **PORTxn** oder **Pxn** bezeichnet, wobei **x** der Portname und **n** die Pinnummer ist (beginnend mit Null!). Das sechste Pin von Port D heißt also **PORTD5** (oder **PD5**)!



Quellen: Atmel Datenblatt ATmega32A

Für die Programmierung eines Pins stehen drei Bits in drei unterschiedlichen SF-Registern zur Verfügung:

- Im **Datenrichtungsregister** **DDRx** (***D**ata **D**irection **R**egister*) wird (meist im Initialisierungsteil) festgelegt ob ein Pin als Eingang oder Ausgang funktionieren soll. Das Register ist lese- und schreibbar. Das erwünschte Bit (Pin) wird mit **DDxn** bezeichnet.
  DDxn = 1: Ausgang
  DDxn = 0: Eingang

Nicht beschaltete Pins sind immer als Eingang zu initialisieren.

Beispiele:

```
        ldi     r16,0xF0        ;DDRD = 11110000b
        out     DDRD,Tmp1       ;obere 4 Bit als Ausgang, untere 4 Bit unbenutzt
        sbi     DDRA,3          ;Pin 3 (PORTA3,PA3) = Ausgang
        cbi     DDRA,4          ;Pin 4 (PORTA4,PA4) = Eingang
```

- Über das **Datenausgaberegister** **PORTx** (***PORTx*** *data register*) werden die Daten an die einzelnen Pins ausgegeben. Es ist lese- und schreibbar.
  Ist ein Pin als Eingang initialisiert, besitzt dieses Register eine andere Funktion! Mit einer Eins kann ein interner Pull-Up-Widerstand zum Eingangspin zugeschaltet werden.

  Beispiele:

```
        ldi     Tmp1,'A'        ;ASCII-Code von 'A' ueber die LEDs ausgeben
        out     PORTD,Tmp1      ;
        sbi     PORTB,7         ;oberste LED PB7 einschalten

        cbi     DDRD,0          ;PD0 = Eingang
        sbi     PORTD,0         ;internen Pull-Up-Widerstand zuschalten
```

- Über das **Dateneingaberegister** **PINx** (***P****ortx* ***IN****put pins adress*) werden Daten eingelesen. Es ist nur lesbar (an sich ist **PINx** kein klassisches Register, denn die Bits folgen augenblicklich dem Zustand der Pins).

  Beispiele:

```
        in      Tmp1,PINA       ;PINA (8 Bit) einlesen
        andi    Tmp1,0x80       ;maskiere oberstes Bit
        breq    SW_ON           ;falls Null, dann springe nach SW_ON
        ....
        ....
SW_ON:  ....

LOOP:   sbic    PINA,7          ;ueberspringe naechste Zeile falls PA7=0
        rjmp    LOOP            ;(warte also bis PA7=0)
SW_ON2: ....
```

Zusätzlich besteht die Möglichkeit über das Flag (Bit) **PUD** (***P****ull-****U****p* ***D****isable*) im **SFIOR** (***S****pecial* ***F****unction* ***I/O*** ***R****egister*) alle Pull-Up-Widerstände für alle Pins abzuschalten.

Damit erhalten wir folgende Tabelle für **PORTD5**:

	DDD5	PORTD5	Pull-UP	Bemerkung
Eingang	**0**	**0**	Nein	Hochohmiger Eingang (Tri-State, Hi-Z). Daten in **PIND** Bit5.
	0	**1**	Ja	Der Pull-Up-Widerstand ist nur eingeschaltet, wenn **PUD** im **SFIOR** Null ist (*default*). Pull-Up-Widerstände ziehen Strom. Nur Einschalten wenn nötig! Daten in **PIND** Bit5.
Ausgang	**1**	**0**	Nein	Gegentakt-Ausgang (Push-Pull). Unterer Transistor legt den Ausgang auf Low-Potential (0).
	1	**1**	Nein	Gegentakt-Ausgang (Push-Pull). Oberer Transistor legt den Ausgang auf High-Potential (1)

Die SF-Register zur parallelen digitalen Ein- und Ausgabe

Die 4 Datenrichtungsregister DDRx

DDRA-Register: SF-Register-Adresse **0x1A** (SRAM-Adresse **0x003A**)
DDRB-Register: SF-Register-Adresse **0x17** (SRAM-Adresse **0x0037**)
DDRC-Register: SF-Register-Adresse **0x14** (SRAM-Adresse **0x0034**)
DDRD-Register: SF-Register-Adresse **0x11** (SRAM-Adresse **0x0031**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

DDRx = Data Direction Register PORTx

Bit	7	6	5	4	3	2	1	0
DDRx	**DDx7**	**DDx6**	**DDx5**	**DDx4**	**DDx3**	**DDx2**	**DDx1**	**DDx0**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

DDxn ***Data Direction Bit n (Port x)***

0 Das betreffende Pin als Eingang aktiv. Um Kurzschlüsse zu vermeiden sollten alle nicht benötigten Pins auf Eingang geschaltet werden (Startwert).

1 Das betreffende Pin ist als Ausgang beschaltet.

Die 4 Datenausgaberegister PORTx

PORTA-Register: SF-Register-Adresse **0x1B** (SRAM-Adresse **0x003B**)
PORTB-Register: SF-Register-Adresse **0x18** (SRAM-Adresse **0x0038**)

PORTC-Register: SF-Register-Adresse **0x15** (SRAM-Adresse **0x0035**)
PORTD-Register: SF-Register-Adresse **0x12** (SRAM-Adresse **0x0032**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

PORTx PORTx Data Register

Bit	7	6	5	4	3	2	1	0
PORTx	**PORTx7**	**PORTx6**	**PORTx5**	**PORTx4**	**PORTx3**	**PORTx2**	**PORTx1**	**PORTx0**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

PORTxn ***PORTx Data Pin n***

a) **Das Pin ist als Ausgang konfiguriert DDRxn = 1**
0 Das Pin liegt auf Masse. Der Strom kann zum Pin fließen (Stromsenke).
1 Das Pin liegt auf VCC und kann als Quelle einen Strom liefern.
b) **Das Pin ist als Eingang konfiguriert DDRxn = 1**
0 Das Pin liegt ist hochohmig (Tri-State).
1 Ist das **PUD**-Bit (*pull-up disable*)im Register **SFIOR** gelöscht (Startwert) so wird intern ein Pull-Up-Widerstand gegen **VCC** geschaltet. Dadurch ist es möglich auf externe Pull-Up-Widerstände zu verzichten, wenn das Pin als Eingang benutzt wird. Bei gesetztem **PUD**-Bit ist
der Ausgang hochohmig.

Die 4 Dateneingaberegister PINx

PINA-Register: SF-Register-Adresse **0x19** (SRAM-Adresse **0x0039**)
PINB-Register: SF-Register-Adresse **0x16** (SRAM-Adresse **0x0036**)
PINC-Register: SF-Register-Adresse **0x13** (SRAM-Adresse **0x0033**)
PIND-Register: SF-Register-Adresse **0x10** (SRAM-Adresse **0x0030**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

PINx = PORTx INput Pins Adress

Bit	7	6	5	4	3	2	1	0
PINx	**PINx7**	**PINx6**	**PINx5**	**PINx4**	**PINx3**	**PINx2**	**PINx1**	**PINx0**
Startwert	-	-	-	-	-	-	-	-
Read/Write	R	R	R	R	R	R	R	R

PINxn ***PORTx INput Pin n***
Unabhängig vom Datenrichtungsregister kann über die **PINx** Adresse der Zustand eines Pins eingelesen werden. Das Pin sollte dabei immer einen definierten Zustand haben (Pull-Up oder Pull-Down-Widerstand zuschalten!).

Bemerkungen: **Um mögliche Kurzschlüsse zu vermeiden sollen nicht benutzte Pins immer als Eingang initialisiert werden.**

Es sollen immer nur die wirklich benötigten Pins oder Bits angesprochen werden und nicht das ganze Port global, um Fehler wie zum Beispiel das unbeabsichtigte Abschalten eines Pull-Up-Widerstandes zu vermeiden. Zur

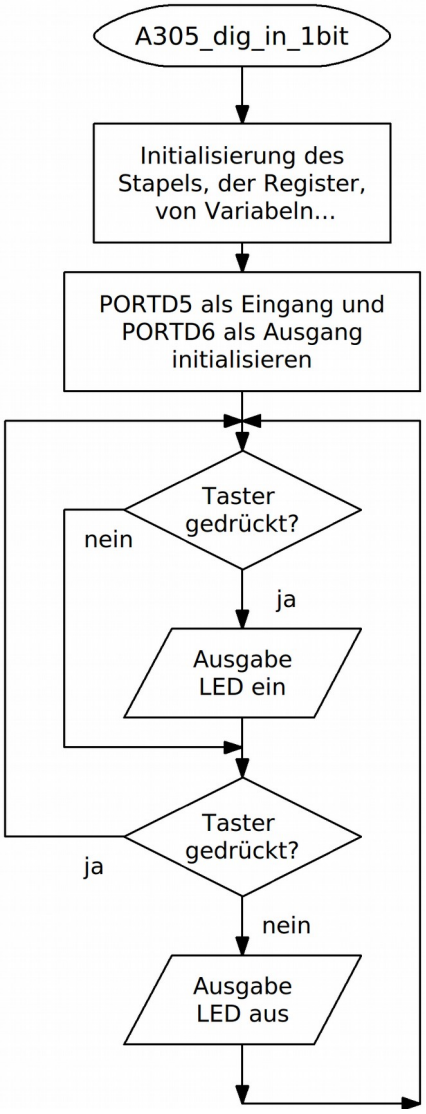
individuellen Ansteuerung sind die Befehle "**sbi**" und "**cbi**" bzw. "**sbis**" und "**sbic**" besonders gut geeignet.

Digitale Daten einlesen

Das dritte Programm (A305_dig_in_1bit.asm)

In unserem nächsten Programm soll mit Hilfe eines Tasters (nicht entprellter Taster an **PORTD5** gegen Masse![21]) eine LED (an **PORTD6**) eingeschaltet werden Die LED soll nur so lange leuchten wie der Taster gedrückt wurde.

Das entsprechende Flussdiagramm sieht folgendermaßen aus:



21 Da bei den ATmega-Controllern interne Pull-Up-Widerstände zur Verfügung stehen wird meist mit der negativen Logik gearbeitet! Ein betätigter Taster gibt also eine Null zurück!

Eine mögliche Version des Programms kann so aussehen:

```
.EQU    PButt = 5               ;Taster (Push-Button) an PORTD5
.EQU    LED   = 6               ;LED an PORTD6

        ;nur Pin6 von PortD als Ausgang initialisieren
        ldi     Tmp1,0x40       ;DDRD = 0b01000000
        out     DDRD,Tmp1       ;PORTD6 auf 1 (Ausgang) alle anderen 0 (Eingang)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   sbis    PIND,PButt      ;Ueberspringe naechste Zeile, wenn Bit gesetzt
                                ;Taster nicht gedrueckt (Pin PORTD5 = 1)
        sbi     PORTD,LED       ;Schalte LED ein (PORTD6 = 1)
        sbic    PIND,PButt      ;Ueberspringe naechste Zeile, wenn Bit geloescht
                                ;Taster gedrueckt (Pin PORTD5 = 0)
        cbi     PORTD,LED       ;Schalte LED aus (PORTD6 = 0)
        rjmp    Main            ;Endlosschleife
```

Diese Zeilen werden in die Vorlage eingefügt. Das Programm arbeitet mit einigen neuen sehr praktischen Befehlen um mit SF-Registern (z.B. **PORTx** oder. **PINx**)zu arbeiten: **sbis** (*skip if bit set*) und **sbic** (*skip if bit cleared*) ermöglichen es ein einzelnes Bit in einem Ein-/Ausgaberegister zu überprüfen. Wenn die Bedingung erfüllt ist wird die nächste Zeile übersprungen. Mit **sbi** (*set bit*) und **cbi** (*clear bit*) können einzelne Bits in Ein-Ausgaberegistern gesetzt oder gelöscht werden.

Mit klassischen Befehlen hätte dieses Programmstück wegen der notwendigen Maskierung (siehe nächstes Kapitel) mehr Zeilen benötigt und wäre komplizierter ausgefallen:

sbis P,b

Überspringe nächste Zeile, wenn Bit b im SF-Register gesetzt (Eins) ist (*skip if bit in SF-register is set*).

1	0	0	1	1	0	1	1	P	P	P	P	P	b	b	b

0≤b≤7. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit).
Beeinflusste Flags: keine Taktzyklen:
1 (kein Sprung), 2 oder 3 (Sprung 1 oder 2 Worte)

sbic P,b

Überspringe nächste Zeile, wenn Bit b im SF-Register gelöscht (Null) ist (*skip if bit in SF-register is cleared*).

1	0	0	1	1	0	0	1	P	P	P	P	P	b	b	b

0≤b≤7. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit).
Beeinflusste Flags: keine Taktzyklen:
1 (kein Sprung), 2 oder 3 (Sprung 1 oder 2 Worte)

```
(MAIN:  in      Tmp1,PIND       ;PIND einlesen
        andi    Tmp1,0x20       ;Maskierung von PD5 (0b00100000) mit AND
        breq    ON              ;Springe falls Null (PD5 = 0) nach ON
        in      Tmp1,PORTD      ;PORTD einlesen
        andi    Tmp1,0xBF       ;AND-Maske mit 0b10111111
        out     PORTD,Tmp1      ;LED mittels AND-Maskierung ausschalten, PD6 = 0
        rjmp    MAIN            ;Verbleibe in der Endlosschleife
ON:     in      Tmp1,PORTD      ;PORTD einlesen
        ori     Tmp1,0x40       ;ODER-Maske mit 0b01000000
        out     PORTD,Tmp1      ;LED mittels ODER-Maskierung einschalten, PD6 = 1
        rjmp    MAIN            ;Verbleibe in der Endlosschleife )
```

Die Befehle `sbis, sbic, sbi` und `cbi` können nur bei den untersten 32 SF-Registern eingesetzt werden!.

Soll das ganze Port eingelesen werden, so geschieht dies mit dem "**`in`**"-Befehl und dem Dateneingaberegister **`PINx`**. Beispielsweise zum Einlesen von Port B:

```
        in      Tmp1,PINB       ;ganzes PORTB (8 Bit) einlesen
```

Achtung!
Man muss zur Abfrage der Eingänge das Eingangsregister `PINx` benutzen!

Irrt man sich und verwendet **`PORTx`** statt **`PINx`** so liest man den Zustand der Pull-Up-Widerstände ein und nicht den Zustand des Eingangssignals (siehe später).

`in Rd,P`

Lade den Inhalt (8 Bit) eines SF-Register in das Arbeitsreg. Rd (load *SF-register to register*).

1	0	1	1	0	P	P	d	d	d	d	d	P	P	P	P

Beeinflusste Flags: keine Taktzyklen: 1

⌂ **A305** Erstelle das Programm indem du die Vorlage mit den obigen Zeilen ergänzt. Speichere das Programm unter dem Namen "**`A305_dig_in_1bit.asm`**" ab. Teste dann das Programm einmal mit deiner Hardware und simuliere das Programm ebenfalls im Studio 4 (Anhang "Das erste Programm!).

Bemerkung: Leider funktioniert das erstellte Programm nicht einwandfrei! Der digitale Eingang hängt, wenn der Taster nicht gedrückt wird, in der Luft. Der Eingang ist nicht definiert und wirkt so als Antenne. Die Helligkeit der LED ändert sich, wenn man mit der Hand in die Nähe des Tasters kommt. Je nach Störpegel kann die LED sogar dauernd eingeschaltet bleiben.

Pull-Up Pull-Down

Damit Aufgabe 305 zufriedenstellend funktionieren kann wird ein Pull-Up-Widerstand (da der Taster gegen Masse liegt!) benötigt. Man kann dazu den internen Pull-Up-Widerstand einschalten. Dazu wird im Ausgaberegister der dem Taster entsprechende Pin auf Eins gesetzt:

```
        sbi     PORTD,PButt     ;internen Pull-Up-Widerstand aktivieren
```

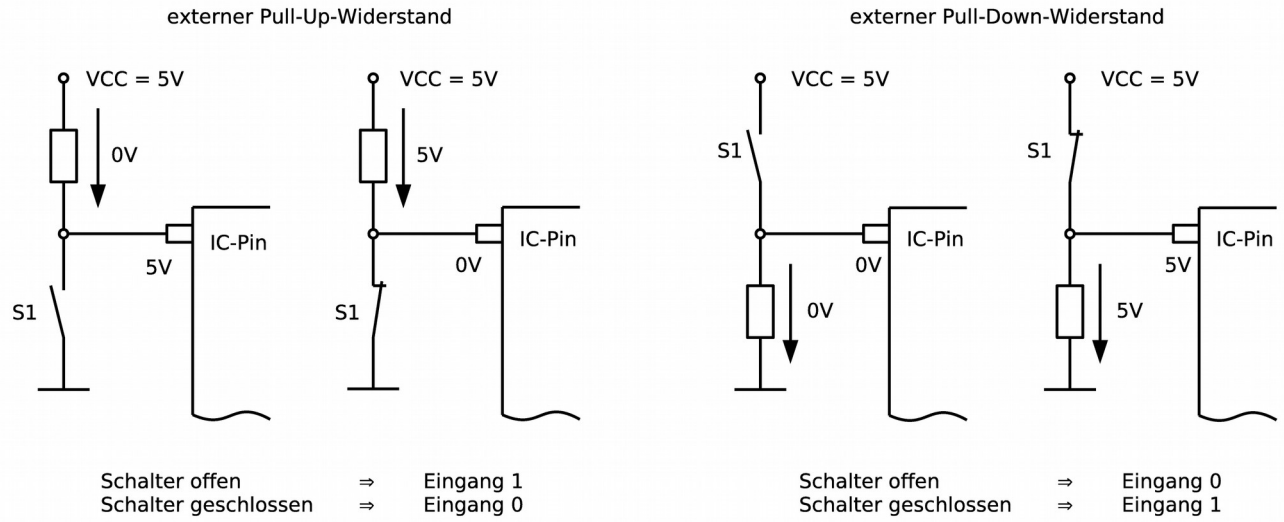
Es besteht natürlich auch die Möglichkeit einen externen Widerstand zu verwenden.

Hier eine kleine Wiederholung wie Taster an einen Eingang angeschlossen werden. Taster gegen Masse benötigen Pull-Up-Widerstände. Taster gegen Betriebsspannung benötigen Pull-Down-Widerstände.

Achtung:
Die Logik ändert sich je nachdem welche Schaltungsvariante benutzt wird.

Bei Pull-Up-Widerständen entsteht eine negative Logik (Taster betätigt entspricht 0V).

Mit Pull-Down- Widerständen entsteht eine positive Logik (Taster betätigt entspricht 5V)!



Pull-Up- und Pull-Down-Widerstände werden benötigt um zu verhindern, dass ein digitaler Eingang einen undefinierten Zustand annehmen kann!

Bemerkungen: Mit einer Invertierung der Eingänge nach dem Einlesen kann natürlich ohne weiteres die Logik verändert werden. Dabei kann mit dem "**com**"-Befehl (Einerkomplement) entweder das ganze Register invertiert werden oder besser mit einer Exklusiv-Oder-Verknüpfung nur das benötigte Pin (Maskierung siehe weiter unten).

com Rd

Einerkomplement Reg. Rd (*one's complement*).

1 0 0 1	0 1 0 d	d d d d	0 0 0 0

Invertiert gesamtes Register Rd.
Beeinflusste Flags: S, V(0), N, Z, C(1) Taktzyklen: 1

Um die Spannungsquelle des Controller nicht unnötig zu belasten sollen nur die wirklich benötigten Pull-Up-Widerstände aktiviert werden!

Nach dem Zuschalten eines Pull-Up Widerstandes in der Initialisierung kann es bei ungünstigem Layout bis zu 10 Taktzyklen (bei 16 MHz) dauern bis der Pull-Up wirklich aktiv ist. Es sollte also nicht gleich nach dem Zuschalten des Pull-Ups eingelesen werden. Alternativ können einige NOP-Befehle oder eine Zeitschleife von ungefähr 1µs eingefügt werden.

⌂ **A306**

a) Ändere das Programm so um, dass es den internen Pull-Up-Widerstand verwendet und nenne es "**`A306_dig_out_1bit_pull_up.asm`**".
b) Miss an **PD4** den Strom der von Masse in die Schaltung fließt. Stimmt dieser Wert mit dem Datenblatt überein? Welcher Wert hat dann der interne Pull-Up-Widerstand?

⌂ **A307**

Ändere das Programm jetzt so um, dass der Zustand der LED bei jedem Drücken des Tasters verändert wird (dies nennt man auch noch "*toggeln*" des Zustandes).
a) Erstelle zuerst! das entsprechende Flussdiagramm.
b) Speichere das Programm als "**`A307_dig_in_1bit_toggle.asm`**" ab.

Tipp:
Mit Hilfe einer Exklusiv-Oder-Funktion lässt sich der Zustand eines einzelnen Bits bzw. Pins invertieren (Befehl **`eor`**). Bei Schwierigkeiten kann man zuerst das nächste Unterkapitel zur Maskierung durchlesen!

Bemerkung: Wenn ein normaler Taster benutzt wird, wird das Programm noch immer nicht ganz richtig funktionieren, da der Taster prellt. In einem nächsten Kapitel (Modul B) wird erklärt wie man dieses Prellen softwaremäßig beseitigen kann. Wenn die Möglichkeit besteht, dann soll das Programm auch einmal mit einem hardwaremäßig entprellten Taster oder Schalter getestet werden.

Die Maskierung von Daten

Bei den unteren 32 SF-Registern bietet uns die ATmega-Serie bequeme Befehle um sogar einzelne Bits zu adressieren. Auf andere Register (zum Beispiel die oberen 32 SF-Register) trifft dies allerdings nicht zu. Hier muss klassisch mit einer Maskierung des Datenbytes, zwecks Löschen, Setzen oder Toggeln eines oder mehrerer Bits, gearbeitet werden.

Unter der Maskierung von Bits versteht man das zwangsweise Setzen eines Bits auf logisch Null (0) oder logisch Eins (1) durch Verwendung einer geeigneten Bitmaske und einer Booleschen Grundverknüpfung (logische Funktion: AND, OR, NOT), ohne die anderen Bits zu verändern.

Die AND-Verknüpfung:

Eine AND-Verknüpfung mit logisch 0 setzt den Wert einer Bitstelle zwangsweise auf logisch 0!

alter Wert des Bits		Maske		neuer Wert der Bits
0	AND	0	=	0
1	AND	0	=	0

Eine AND-Verknüpfung mit logisch 1 ändert den Wert eines Bit nicht!

alter Wert des Bits		Maske		neuer Wert der Bits
0	AND	1	=	0
1	AND	1	=	1

Die zwangsweise Ausmaskierung einer Bitstelle auf den Wert logisch 0 (Löschen eines Bits) erfolgt durch eine Bitmaske in der an der entsprechenden Stelle eine logische 0 steht und an allen anderen Stellen eine logische 1. Natürlich können auch mehrere Bits auf einmal ausmaskiert werden.

AND: Null in der Maske löscht das Bit (logisch 0)
Eins in der Maske hat keinen Einfluss

Beispiel: In Register `r20` wird mit Hilfe von Zustandsbits (Flags) der Zustand von acht Werkzeugmaschinen protokolliert. Bei einer Null ist die Maschine aus, bei einer Eins ist sie eingeschaltet.
Es soll überprüft werden ob die dritte und die sechste Maschine wirklich

ausgeschaltet sind. Ermittle die zu verwendende Maske.

Bit 2^2 und Bit 2^5 sollen unverändert eingelesen werden. Die Zustände der anderen Bits (Maschinen) interessieren uns nicht, und sie werden durch die Maske gelöscht. Wir benötigen also folgende Maske:

Maschine	8	7	6	5	4	3	2	1
Bit	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Maske	0	0	1	0	0	1	0	0

Ein Programmcode könnte dann beispielsweise so aussehen:

```
mov     Tmp1,r20     ;Register 20 in den Zwischenspeicher
                     ;kopieren. Noetig, da Register sonst durch
                     ;die logische Operation veraendert wird!
andi    Tmp1,0x24    ;Maskierung von Bit 2^2 und 2^5(00x00x00b)
                     ;mit AND
brne    WARNING      ;Springe falls eine der beiden Maschinen
                     ;nicht aus (oder beide; Z=1) zum Label WARNING
```

`mov Rd,Rr`

Kopiere den Inhalt von Arbeitsregister Rr (Quelle, *source*) zum Arbeitsregister Rd (Ziel, *destination*) (*copy register*).

0	0	1	0	1	1	r	d	d	d	d	d	r	r	r	r

Klassischer Transferbefehl zum Kopieren von **Arbeits-registern**. Der Inhalt wird nicht verschoben sondern kopiert. Die Quelle Rr (*source*) bleibt unverändert.
Beeinflusste Flags: keine Taktzyklen: 1

`andi Rd,K`

Log. UND-Verknüpfung des Registers Rd mit der Konstanten K (*logical and with immediate*).

0	1	1	1	K	K	K	K	d	d	d	d	K	K	K	K

Unmittelbare Verknüpfung. Resultat in Rd. Nur Reg. r16-r31.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

`and Rd,Rr`

Logische UND-Verknüpfung des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*logical and*).

0	0	1	0	0	0	r	d	d	d	d	d	r	r	r	r

Resultat in Rd. Bei Maskierung Maske in Rr.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

`brne k`

Bedingter relativer Sprung falls nicht gleich (*branch if not equal*).(k = Adresskonstante)

1	1	1	1	0	1	k	k	k	k	k	k	k	0	0	1

Sprünge die an eine Bedingung geknüpft sind werden als Verzweigung bezeichnet (*branch*). Der Befehl wird meist gleich nach einer arithmetischen oder logischen Operation eingesetzt. **Der Sprung erfolgt falls das Resultat der Operation nicht Null wurde** (**Z**-Flag = 0). War diese Operation ein Vergleich (Subtraktion, siehe *compare*-Befehle) so waren beide Operanden nicht gleich.
Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

Die OR-Verknüpfung:

Eine OR-Verknüpfung mit logisch 0 ändert den Wert eines Bit nicht!

alter Wert des Bits		Maske		neuer Wert der Bits
0	OR	0	=	0
1	OR	0	=	1

Eine OR-Verknüpfung mit logisch 1 setzt den Wert einer Bitstelle zwangsweise auf logisch 1!

alter Wert des Bits		Maske		neuer Wert der Bits
0	OR	1	=	1
1	OR	1	=	1

Das zwangsweise Setzen einer Bitstelle auf den Wert logisch 1 erfolgt durch eine Bitmaske in der, an der entsprechenden Stelle eine logische 1 steht, und an allen anderen Stellen eine logische 0. Natürlich können auch mehrere Bits auf einmal gesetzt werden.

OR: **Eins** in der Maske **setzt** das Bit (logisch '1')
Null in der Maske hat keinen Einfluss

⌂ **A308** Die Werkzeugmaschinen 8, 7 und 2 aus dem vorigen Beispiel wurden eingeschaltet. Dies soll jetzt im Register **`r20`** dokumentiert werden. Berechne die Maske und schreibe die entsprechenden Programmzeilen.

`ori Rd,K`

Log. ODER-Verknüpfung des Registers Rd mit der Konstanten K (*logical or with immediate*).

0	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K

Unmittelbare Verknüpfung. Resultat in Rd. Nur Reg. r16-r31.
Beeinflusste Flags: S, V(0), N, Z **Taktzyklen: 1**

or Rd,Rr

Log. ODER-Verknüpfung des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*logical or*).

0	0	1	0	1	0	r	d	d	d	d	d	r	r	r	r

Resultat in Rd. Bei Maskierung Maske in Rr.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

Die XOR-Verknüpfung:

Neben dem Löschen und Setzen einzelner Bits ist es manchmal nötig einzelne Bits zu invertieren (umzuschalten, zu toggeln). Hierzu eignet sich die XOR-Verknüpfung.

Bemerkung: XOR steht für Exklusiv-Oder (*exclusive or*). Andere Bezeichnung sind EOR (siehe Befehl), Antivalenz (Kontravalenz), und ENTWEDER-ODER. Es handelt sich auch um eine Modulo-2-Addition (siehe Halbaddierer)

Eine XOR-Verknüpfung mit logisch 0 ändert den Wert eines Bit nicht!

alter Wert des Bits		Maske		neuer Wert der Bits
0	XOR	0	=	0
1	XOR	0	=	1

Eine XOR-Verknüpfung mit logisch 1 invertiert den Wert einer Bitstelle!

alter Wert des Bits		Maske		neuer Wert der Bits
0	XOR	1	=	1
1	XOR	1	=	0

Das zwangsweise Invertieren einer Bitstelle erfolgt durch eine Bitmaske in der an der entsprechenden Stelle eine logische 1 steht und an allen anderen Stellen eine logische 0. Natürlich können auch mehrere Bits auf einmal invertiert werden.

XOR: Eins in der Maske invertiert das Bit.
Null in der Maske hat keinen Einfluss.

⌂ **A309** Der Inhalt von Register **r21** soll mit der **XOR**-Funktion invertiert werden. Schreibe die dazu nötigen Programmzeilen. Welchen Befehl könnte man hier noch verwenden?

eor Rd,Rr

Logische EXKLUSIV-ODER-Verknüpfung des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*exclusive or*).

0	0	1	0	0	1	r	d	d	d	d	d	r	r	r	r

Resultat in Rd. Bei Maskierung Maske in Rr. Es existiert kein unmittelbarer Befehl für die EOR-Verknüpfung.
Beeinflusste Flags: S,V(0),N,Z Taktzyklen: 1

A4 Befehle und Adressierung

Verschiedene Befehle aus unterschiedlichen Befehlsgruppen wurden im vorigen Kapitel schon verwendet. In diesem Kapitel sollen alle Befehlsgruppen vorgestellt werden und auch die unterschiedlichen Adressierungsarten. Die Beeinflussung der Bedingungsbits (Flags, Zustandsbits) durch Befehle soll ebenfalls untersucht werden sowie die damit verbundenen bedingten Sprünge.

Die Befehle der ATmega-Mikrocontroller

Beim ATmega-Controller stehen mehr Befehle zur Verfügung als bei einem klassischen 8-Bit-Prozessor. Erstens besitzt der Controller wesentlich mehr Arbeitsregister. Zweitens beschränken sich arithmetische Befehle nicht nur auf ein Akkumulator-Register. Es können alle 32 Arbeitsregister für Berechnungen herangezogen werden.

Insgesamt stehen beim ATmega32A 131 unterschiedliche Befehle (ATmega8A 130 Befehle) zur Verfügung.

Ein 150-seitiges Dokument zum Befehlssatz von ATMEL kann zum detaillierten Verständnis bei Bedarf an der folgenden Adresse bezogen werden:

www.atmel.com/atmel/acrobat/doc0856.pdf

Ein Befehl besteht aus dem Opcode und den Operanden. Es gibt Befehle ohne Operand, welche wo nur ein Operand benötigt wird und Befehle mit zwei Operanden.

Beispiele:

	Opcode	**Operand**
kein Operand:	**`nop`**	
1 Operand:	**`inc`**	**`r16`**
2 Operanden:	**`ldi`**	**`r16,0x08`**

Bei zwei Operanden steht immer zuerst das Ziel und dann die Quelle!

Für die meisten Befehle wird nur ein Wort benötigt (16 Bit). Ausnahmen sind:
"**`lds Rd,k16`**", "**`sts k16,Rd`**", "**`call k`**" und "**`jmp k`**" welche jeweils 2 Worte benötigen.

Im Befehlssatz werden für Operanden folgende Abkürzungen verwendet:

`Rd` Meist Ziel-Arbeitsregister (*destination*, bei einigen Befehlen auch Quelle)
`r0-r31` (bei unmittelbaren (*immediate)* Befehlen nur **`r16-r31`**)
`Rdl` Niederwertiges Byte (LByte) eines 16 Bit Ziel-Arbeitsregister
`Rr` Quell- oder Sende-Arbeitsregister (**`r0-r31`**, *source*)
`K` Daten-Konstante 8 Bit (0-255)

k Adress-Konstante für Operationen mit dem "*program counter* PC".
(Bsp.: Label für einen Sprung)
b Bitkonstante 3 Bit (0-7), zum Auswählen eines Bits in einem Arbeits- oder SF-Registers
P Adresse eines SF-Registers 6 Bit (0-63)
s Bitkonstante 3 Bit (0-7), zum Auswählen eines Bits im Statusregister
X,Y,Z Doppelregister (Pointer, Adresszeiger) zur direkten Adressierung
X ⇒ **r27:r26**; **Y** ⇒ **r29:r28**; **Z** ⇒ **r31:r30**

Wie aus dem Befehlssatz ersichtlich unterscheidet man folgende Befehlsgruppen:

Datentransferbefehle (Datentransportbefehle)

Transfer- oder Transportbefehle dienen dem Transport von Daten zwischen Arbeitsregistern, Arbeitsregistern und SF-Registern sowie Arbeitsregistern und dem SRAM. Auch werden sie benötigt um Konstanten in die Register, Registerpaare oder in den Speicher zu schreiben.
Datentransferbefehle beeinflussen die Zustandsbits (Flags) nicht.

Beispiele für Transferbefehle: **mov Rd,Rr**; **in Rd,P**; **push Rr**; **sts k,Rr**; **LPM**

Bemerkung: Bei **mov**-Befehlen wird keine Verschiebung im eigentlichen Sinne durchgeführt, sondern der Inhalt wird kopiert! Der Inhalt des Quellregisters bleibt also erhalten!

Arithmetische und logische Operationen (Befehle)

Arithmetik-Befehle sind Befehle zur Anwendung der Grundrechenarten. Logikbefehle umfassen die Boolschen Funktionen UND, ODER, NICHT sowie Exklusiv-ODER. Sie erlauben ein gezieltes Maskieren, Löschen oder Setzen von Bits in Datenworten.

Arithmetische und logische Operationen werden von der ALU ausgeführt und arbeiten nur mit den Arbeitsregistern!

Arithmetische und logische Operationen beeinflussen die Zustandsbits.

Beispiele für arithmetische Operationen: **add Rd,Rr**; **sbiw Rdl,K**; **inc Rd**; **mul Rd,Rr**

Beispiele für logische Operationen: **or Rd,Rr**; **adiw Rdl,K**; **com Rd**; **tst Rd**

Bitorientierte Befehle

Bitorientierte Befehle dienen dazu Zustandsbits (Flags) zu beeinflussen, Programmunterbrechung (Interrupts) zu erlauben bzw. zu verbieten, einzelne Bits in den Arbeitsregistern bzw. SF-Registern zu setzen oder zu löschen oder um Register zu rotieren.

Beispiele für bitorientierte Befehle: **sec**; **cli**; **sbi P,b**; **lsl Rd**

Sprungbefehle (jump), Verzweigungsbefehle (branch) und Unterprogrammbefehle (call)

Sprungbefehle werden benötigt um Verzweigungen im Programm zu erreichen und beeinflussen die Zustandsbits nicht. Man unterscheidet unbedingte Sprünge (*jump*) und bedingte (an eine Bedingung verknüpfte) Sprünge welche im englischen mit "verzweigen" (*to branch*) bezeichnet werden. Bedingte Sprünge überprüfen meist einzelne Bits im Zustandsregister (Flags), welche durch arithmetische oder logische Operationen beeinflusst wurden (Bsp.: **`tst Rd`**). Hierzu können auch die drei vergleichende Befehle **`cp`**, **`cpc`** und **`cpi`** verwendet werden, die man den arithmetischen Befehlen zurechnen könnte.

Beim ATmega-Controller gibt es noch fünf sehr praktische "Skip"-Befehle, die bei erfüllter Bedingung eine Befehlszeile überspringen. Vier davon (**`sbrc`**; **`sbrs`**; **`sbic`**; **`sbis`**) testen einzelne Bits in Arbeits- bzw. den untersten 32 SF-Registern um bei erfüllter Bedingung die folgende Zeile zu überspringen. Ein Befehl überspringt die folgende Zeile wenn beide Register gleich sind (**`cpse`**).

Unterprogrammbefehle ähneln den Sprungbefehlen und werden oft diesen zugerechnet. Man unterscheidet Programmaufrufbefehle (Unterprogrammsprungbefehle) und Rücksprungbefehle.

Beispiele für unbedingte Sprungbefehle: **`rjmp`**, **`jmp`**

Beispiele für bedingte Sprungbefehle: **`breq k`**; **`brbs s,k`**; **`sbrc Rr,b`**; **`sbis P,b`**

Beispiele für Unterprogrammbefehle: **`rcall`**; **`call`**; **`ret`**; **`reti`**

Sonstige Befehle

Es existieren noch 4 Befehle die nicht in die obigen Kategorien eingeordnet werden können. Interessant ist der Leerbefehl (**`nop`**), der nur einen Taktzyklus an Zeit verbraucht. Er wird öfter als Platzhalter für späteren Code eingesetzt. Auch kann man mit ihm zum Beispiel Zeitschleifen präzise abstimmen.

`nop`

Leerbefehl (*no operation*).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tut nichts. Verbraucht nur einen Taktzyklus.
Beeinflusste Flags: keine Taktzyklen: 1

Das Zustands- oder Statusregister SREG

Das Rechenwerk des Controllers addiert und subtrahiert nur Bitmuster. Es interpretiert die Resultate nicht. Um arithmetische Operationen richtig bewerten zu können benötigen wir ein Zustands- oder Statusregister (Flagregister). Seine Bits (0-5) kennzeichnen das Ergebnis einer Operation die in der

ALU durchgeführt wurde. Diese Zustandsbits werden auch noch als **Flags** (Signal-Fahnen) bezeichnet.

Es sind also nur die arithmetischen Befehle (inkl. Vergleichsbefehle), logische Befehle, Rotationsbefehle und Befehle zur Bit-Beeinflussung im Statusregister die das Statusregister beeinflussen. Zusätzlich sind im Zustandsregister noch ein Bit enthalten mit dem global Interrupts zugelassen oder gesperrt werden können (**I**-Flag) und ein Bit (**T**-Flag), das dazu dient dem Anwender das Zwischenspeichern eines Zustandes (Bits, Flags) zu vereinfachen.

Für jedes Flag existiert ein spezieller Befehl um das Flag zu setzen (*set*) bzw. zu löschen (*clear*).

Die Flags des Statusregister SREG

SREG-Register: SF-Register-Adresse **0x3F** (SRAM-Adresse **0x005F**)
Befehle: **in**, **out**, **sec**, **clc**, **sez**, **clz**, **sen**, **cln**, **sev**, **clv**, **ses**, **cls**, **seh**, **clh**, **set**, **clt**, **sei**, **cli** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

SREG = Status Register

Bit	7	6	5	4	3	2	1	0
SREG 0x3F	**I** Interrupt	**T** Transfer	**H** Halfcarry	**S** Sign	**V** Overflow	**N** Negative	**Z** Zero	**C** Carry
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

I Das **"Global Interrupt Enable/Disable"**-Flag **I** wird vom Benutzer gesetzt (1) oder rückgesetzt (0) um global Unterbrechungen zu erlauben bzw. zu verbieten (Befehle: **sei**, **cli**).

T Das **"Transfer"**-Flag **T** erlaubt mit Hilfe der Befehle **bld** und **bst** ein einzelnes Bit aus einem Arbeitsregister abzuspeichern (retten) und wieder zu laden (wiederherstellen). Es kann mit den Befehlen **set** und **clt** auch einfach gelöscht oder gesetzt werden.

H Das **"Halfcarry"**-Flag **H** (Auxiliary-Carry) kennzeichnet einen Übertrag von Bit 3 auf Bit 4 und wird bei Berechnungen mit Nibbles (4 Bit) benötigt. Es kann mit den Befehlen **seh** und **clh** auch einfach gelöscht oder gesetzt werden.

S Das **"Sign"**-Flag **S** wird bei der Berechnung mit vorzeichenbehafteten Zahlen eingesetzt. Es entspricht der Exklusiv-Oder Verknüpfung des Negative- und des Overflow-Flags: $S = N \; xor \; V = (\overline{N} \wedge V) \vee (N \wedge \overline{V})$. Es kann mit den Befehlen **ses** und **cls** auch einfach gelöscht oder gesetzt werden.

V Das **"Overflow"**-Flag **V** zeigt einen Überlauf bei der Zweierkomplement-Berechnung, also bei vorzeichenbehafteten Zahlen an. Es kann mit den Befehlen **sev** und **clv** auch einfach gelöscht oder gesetzt werden

N Das **"Negative"**-Flag **N** entspricht dem höchstwertigen Bit des Resultats.

8 Bit: $N = r7$

16 Bit: $N = r15$.

Es kann mit den Befehlen **sen** und **cln** auch einfach gelöscht oder gesetzt werden.

Z Das **"Zero"**-Flag **Z** wird auf Eins gesetzt wenn das Ergebnis einer Operation Null ist. Es kann mit den Befehlen **sez** und **clz** auch einfach gelöscht oder gesetzt werden

C Das **"Carry"**-Flag **C** wird Eins, wenn ein Übertrag an der höchsten Stelle entsteht. Es kann mit den

Befehlen **sec** und **clc** auch einfach gelöscht oder gesetzt werden.

Der Inhalt der Zustandsbit entscheidet über den weiteren Programmablauf bei Programmverzweigungen. Programmverzweigungen werden über bedingte Sprungbefehle (Verzweigung) verwirklicht.

Beispiel: Beim Befehl **breq** (*branch if equal*, springe wenn Ergebnis eines Vergleichs (Subtraktion) gleich Null) fragt die Ablaufsteuerung den Zustand des Zero-Flags ab. Bei **Z** = 1 wird das Programm zu der im Sprungbefehl enthaltenen Adresse (**k**) verzweigt, bei **Z** = 0 an der alten Adresse fortgesetzt. Solche bedingte Springbefehle werden meist gleich nach einer arithmetischen oder logischen Operation eingesetzt.

Entsprechend bedingte Sprungbefehle gibt es auch für die übrigen Flags. Wegen dieser Bedeutung der Zustandsbit muss der Programmierer genau wissen, welche Befehle Auswirkungen auf den Inhalt bestimmter Zustandsbit haben. Dies ist im Befehlssatz vermerkt.

Mit der Step-Into-Funktion des Studio 4 Programms im Debug-Modus lassen sich bequem einzelne Befehle eines Programms ausführen. Am Bildschirm kann anschließend die Wirkung der Befehle auf das Zustandsregister ausgewertet werden (siehe folgende Aufgaben).

breq k

Bedingter relativer Sprung falls gleich (*branch if equal*).(k = Adresskonstante)

1	1	1	1	0	0	k	k	k	k	k	k	k	0	0	1

Der Sprung erfolgt falls das Resultat einer vorigen Operation Null wurde (**Z**-Flag = 1). War diese Operation ein Vergleich (Subtraktion, siehe *compare*-Befehle) so waren beide Operanden gleich.
Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

Adressierungsarten

Wir unterscheiden:

1. Die **unmittelbare Adressierung** bei der die **Operanden Bestandteil des Befehls** sind.
2. Die **direkte Adressierung** bei der die **unveränderbare Adresse im Befehl enthalten** ist.
3. Die **indirekte Adressierung** wo die **Adresse in einem speziellen 16-Bit Adressregister** (Doppelregister **X**, **Y** oder **Z**) abgespeichert wird und somit dynamisch **veränderbar** ist. Das Doppelregister wird als Adresszeiger, Indexregister oder Pointer bezeichnet.

In der bei Controllern verwendeten Harvard-Struktur wird der Programm- und der Datenspeicher getrennt. Bei der Adressierung werden wir diese auch getrennt betrachten.

Alle Adressierungsarten lassen sich auf den SRAM-Datenspeicher anwenden. Der nichtflüchtige EEPROM-Datenspeicher kann nicht intern adressiert werden. Dazu stehen keine Befehle zur Verfügung. Seine Adressierung über die SF-Register wird später behandelt.

Da der Programmspeicher vorrangig nicht zur Verwaltung von Daten gedacht ist sind seine Adressierungsarten stark eingeschränkt.

Adressierung des Datenbereichs:

Die **direkte Adressierung** kann man unterteilen in die:

- **Direkte Registeradressierung**
  (Arbeiten mit wenigen Variablen (32 Arbeitsregister))
- **Direkte Adressierung der SF-Register**
  (Ein- und Ausgabe über die Peripherie)
- **Direkte Adressierung des SRAM-Datenspeichers**
  (Arbeiten mit vielen Variablen mit konstanten Adressen)

Bei der **indirekte Adressierung des SRAM-Speichers** unterscheidet man:

- **Indirekte Adressierung**
  (Viele Variablen mit variablen Adressen)
- **Indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers**
- **Indirekte Adressierung mit festem (konstantem) Abstand**
- **Indirekte Adressierung mit "`push`" und "`pop`"**

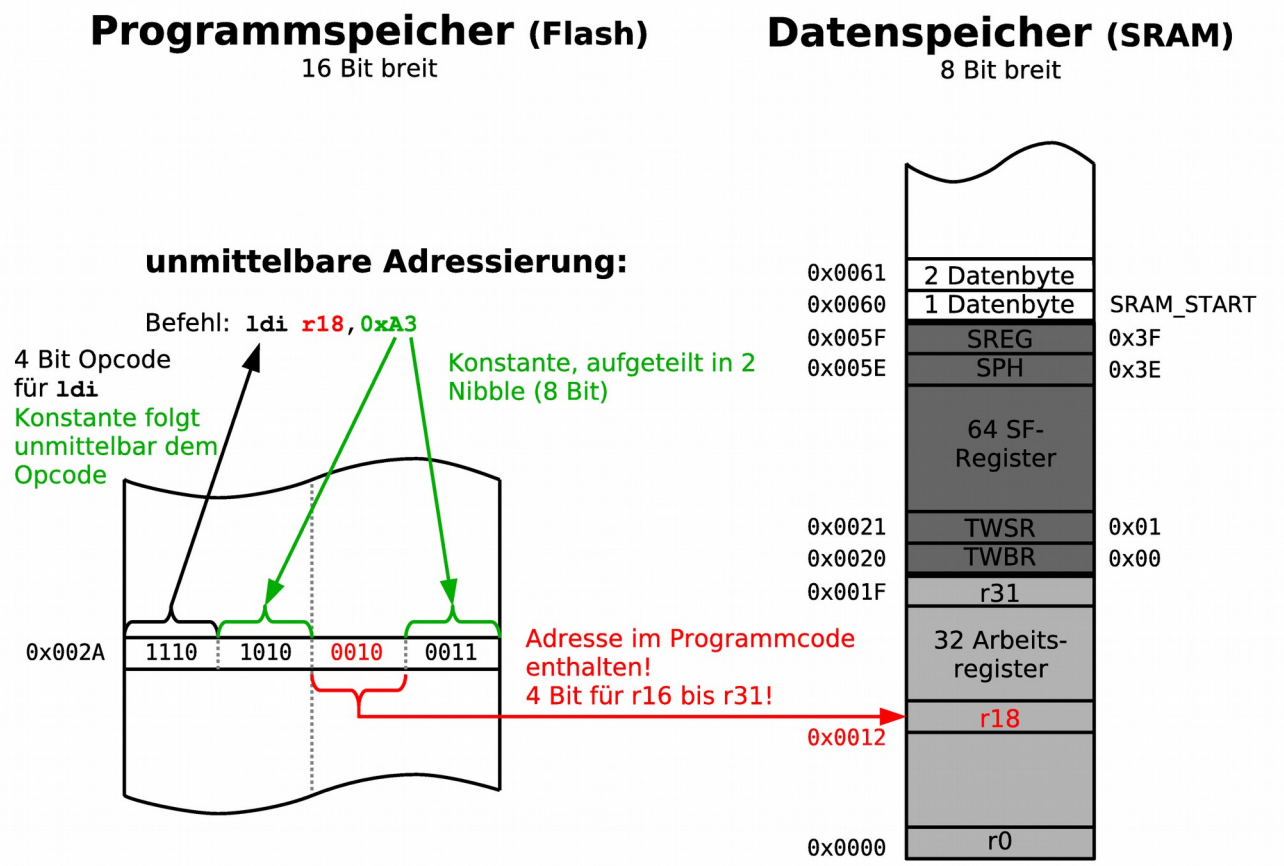
Die unmittelbare Adressierung (`r15-r31`)

Die unmittelbare Adressierung dient zum Arbeiten mit konstanten Werten. Die Konstante wird mit dem betroffenen Register unmittelbar hinter dem Opcode angegeben und befindet sich als Wert also unveränderbar im Flash.

Befehle für die unmittelbare Adressierung erkennt man am Buchstaben „**`i`**“ für das englische „***immediate***“.

Beispiele für eine unmittelbare Adressierung:

```
        ldi   r18,0xA3
        andi  Tmp1,0xFF
        subi  Tmp2,0x01
```

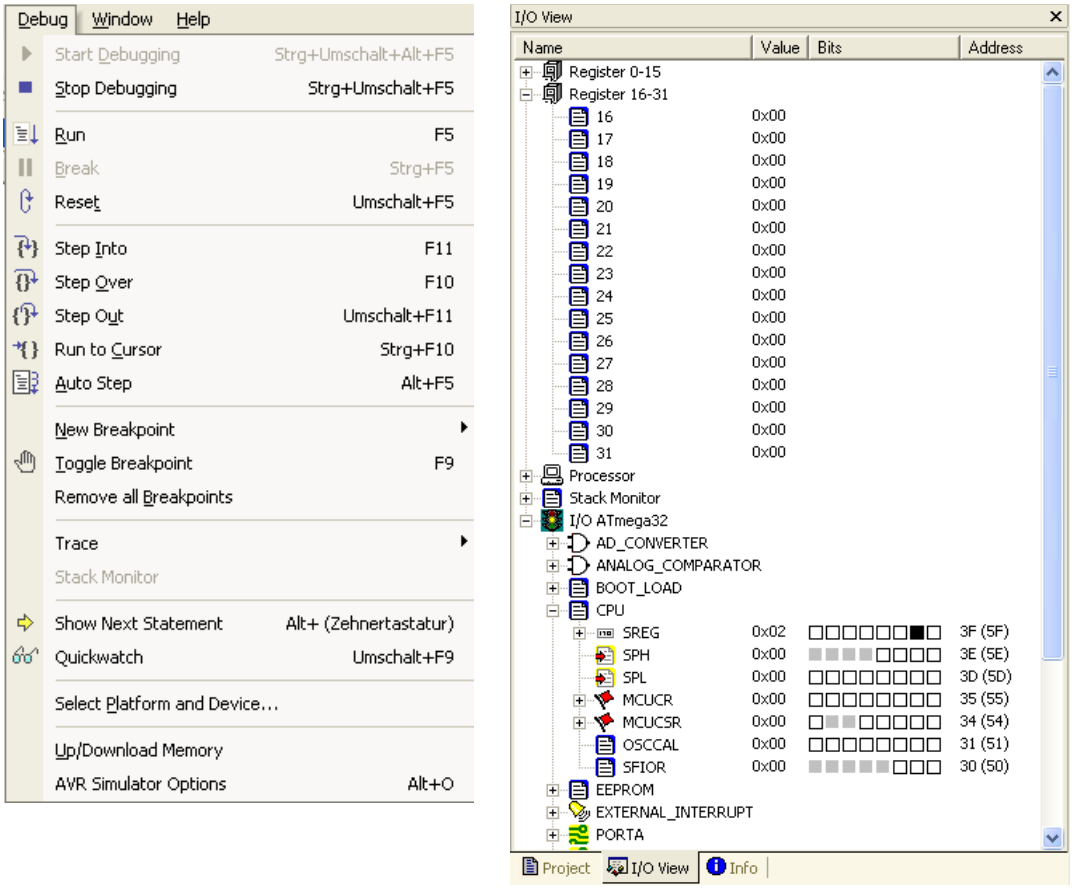


⌂ **A400** Zeichne ein Flussdiagramm und schreibe ein Programm das folgende logische Verknüpfungen durchführt:
Zuerst wird unmittelbar die Dezimalzahl **85** in die Variable (Register) **Tmp1** geladen. Dann wird eine logische UND-Verknüpfung mit der Zahl **170** durchführt (**andi**). Das Resultat wird dann noch mit der Zahl **128** ODER-Verknüpft (**ori**). Gib dem Programm den Namen "**A400_immediate.asm**".

a) Führe die Berechnung zuerst von Hand aus.
b) Teste das Programm im Studio 4 ohne den Baustein zu programmieren (Nach dem "Assemble (Build F7)"-Befehl den Debugging-Modus einschalten
mit "Start Debug" im Menu "Debug". Dann Einzelschritt mit "Step Into (F11)").
c) In welchem Register steht jeweils das Resultat?
d) Betrachte das Statusregister (Flagregister) **SREG** und die Resultate der Berechnungen im "I/O-View"-Fenster und interpretiere die Ergebnisse.

⌂ **A401** Schreibe ein Programm, das folgende Berechnungen mit Dezimalzahlen durchführt: **100 – 50, 100 – 100, 100 – 200** (Befehl: **subi**). Danach soll das Programm nochmals die gleichen Berechnungen durchführen, allerdings mit dem Vergleichsbefehl **cpi** statt **subi**.
Gib dem Programm den Namen "**A401_subi_cpi.asm**".

a) Führe die Subtraktionen zuerst von Hand aus (Addition des Zweier komplement!).
b) Teste das Programm im Studio 4 im Step-Modus
c) Betrachte das Statusregister (Flagregister) **SREG** und die Resultate der



Berechnungen im "I/O-View"-Fenster und interpretiere die Ergebnisse. Es fällt auf, dass das Carry und das Halfcarry-Flag anders als bei der Berechnung von Hand interpetiert werden. Erlärungen finden sich im Dokument „AVR Instruction Set“ bei den Befehlen **`subi`** und **`cpi`**.

`subi Rd,K`

Subtrahiert Konstante K (8 Bit) von Register Rd (*subtract immediate*).

0	1	0	1	K	K	K	K	d	d	d	d	K	K	K	K

Unmittelbare Verknüpfung. Resultat in Rd. Nur Reg. r16-r31.
Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

cpi Rd,K

Vergleicht Konstante K mit Inhalt von Register Rd (*compare with immediate*).

0	0	1	1	K	K	K	K	d	d	d	d	K	K	K	K

Unmittelbare Verknüpfung. Nur Reg. r16-r31. Entspricht einer Subtraktion, allerdings wird der Inhalt von Rd nicht verändert! Wird meist vor bedingten Sprüngen eingesetzt. **Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1**

Bemerkung: Wie schon oben erwähnt, kann **keine unmittelbare Adressierung** mit den Arbeitsregistern **`r0`** **bis** **`r15`** durchgeführt werden.

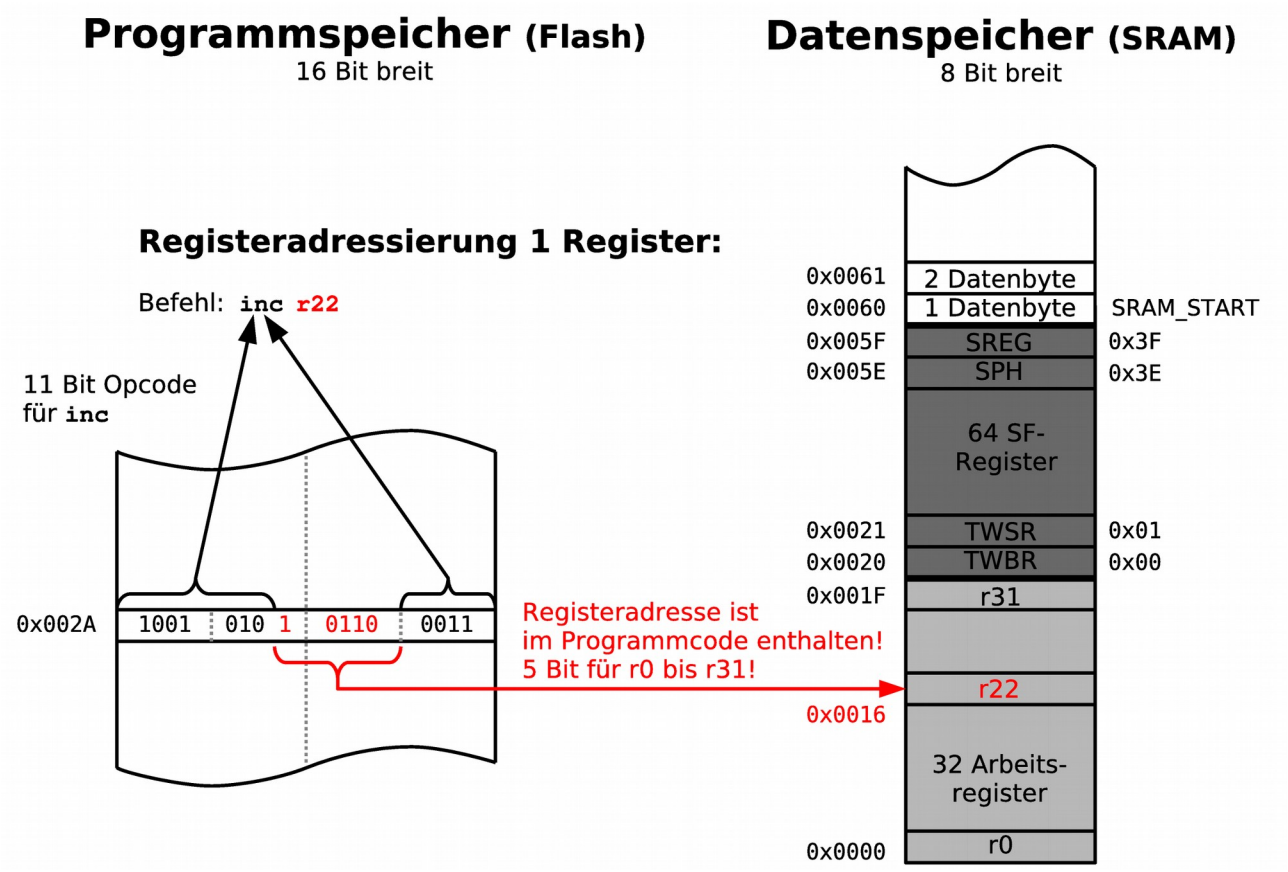
Die direkte Registeradressierung

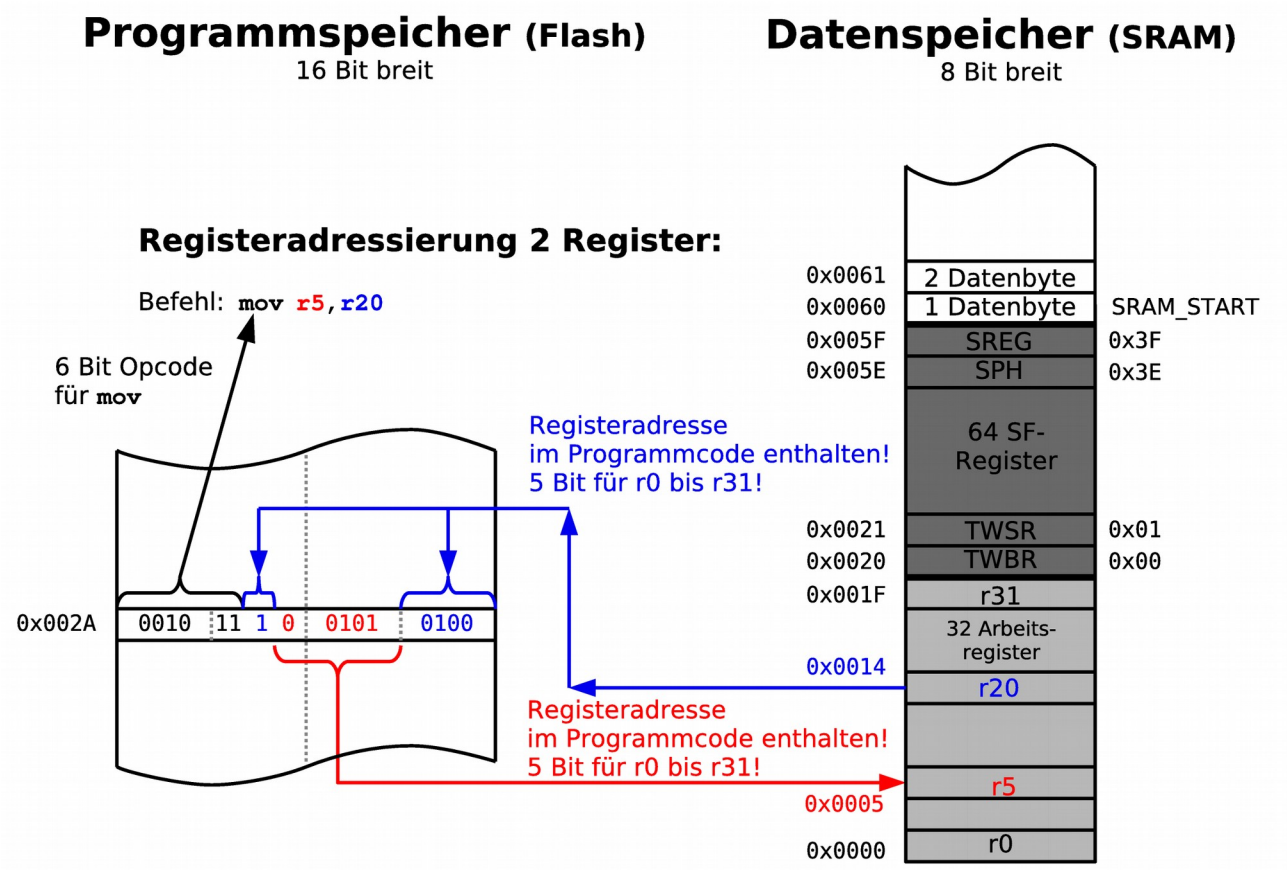
Bei der Registeradressierung beziehen sich alle Operanden eines Befehls auf die Arbeitsregister. Es gibt Befehle die nur ein einzelnes Register benötigen, andere arbeiten mit zwei Registern. Manchmal wird die Registeradressierung auch als direkte Registeradressierung (siehe direkte Adressierung) bezeichnet, da die Adresse des Arbeitsregisters direkt im Operanden enthalten ist.

Befehle für die direkte Registeradressierung erkennt man daran, dass ausschließlich Arbeitsregister als Operanden fungieren (Ausnahme "**`sbr`**" und "**`cbr`**").

Beispiele für eine Registeradressierung:

```
        inc     r23
        com     Tmp1
        mov     r5,r20
        sub     r5,r20
        cp      Tmp2,r22
```


Programmspeicher (Flash)
16 Bit breit
Datenspeicher (SRAM)
8 Bit breit
Registeradressierung 1 Register:
Befehl: inc r22
11 Bit Opcode
für inc
0x002A
1001 010 1 0110 0011
Registeradresse ist
im Programmcode enthalten!
5 Bit für r0 bis r31!
0x0061 2 Datenbyte
0x0060 1 Datenbyte SRAM_START
0x005F SREG 0x3F
0x005E SPH 0x3E
64 SF-
Register
0x0021 TWSR 0x01
0x0020 TWBR 0x00
0x001F r31
r22
0x0016
32 Arbeits-
register
0x0000 r0


Programmspeicher (Flash)
16 Bit breit
Datenspeicher (SRAM)
8 Bit breit
Registeradressierung 2 Register:
Befehl: mov r5,r20
6 Bit Opcode
für mov
0x002A
0010 11 1 0 0101 0100
Registeradresse
im Programmcode enthalten!
5 Bit für r0 bis r31!
Registeradresse
im Programmcode enthalten!
5 Bit für r0 bis r31!
0x0061 2 Datenbyte
0x0060 1 Datenbyte SRAM_START
0x005F SREG 0x3F
0x005E SPH 0x3E
64 SF-
Register
0x0021 TWSR 0x01
0x0020 TWBR 0x00
0x001F r31
32 Arbeits-
register
0x0014
r20
r5
0x0005
0x0000 r0

⌂ **A402**

a) Zeichne ein Flussdiagramm für ein Programm das folgende Berechnungen durchführt:
Zuerst wird die Dezimalzahl **100** in die Variable (Register) **Tmp1** geladen. Dann wird das Register invertiert. Als zweite Zahl wird **90** addiert. Vom Resultat wird **55** mittels Registeradressierung subtrahiert um dann schlussendlich noch einmal **90** zu addieren.

b) Führe die Berechnung zuerst von Hand aus (Subtraktion durch Addition des Zweierkomplements!).

b) Schreibe das Programm und teste es im Studio 4 im Step-Modus.

Gib dem Programm den Namen "**A402_register.asm**".

c) Betrachte das Statusregister (Flagregister) **SREG** und die Resultate der Berechnungen im "I/O-View"-Fenster und interpretiere die Ergebnisse.

add Rd,Rr

Addition des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*add without carry*).

0	0	0	0	1	1	r	d	d	d	d	d	r	r	r	r

Resultat in Rd. Carry-Flag wird nicht dazuaddiert (siehe adc)
Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

sub Rd,Rr

Subtrahiere das Arbeitsregisters Rr vom Arbeitsregister Rd (*subtract without carry*).

0	0	0	1	1	0	r	d	d	d	d	d	r	r	r	r

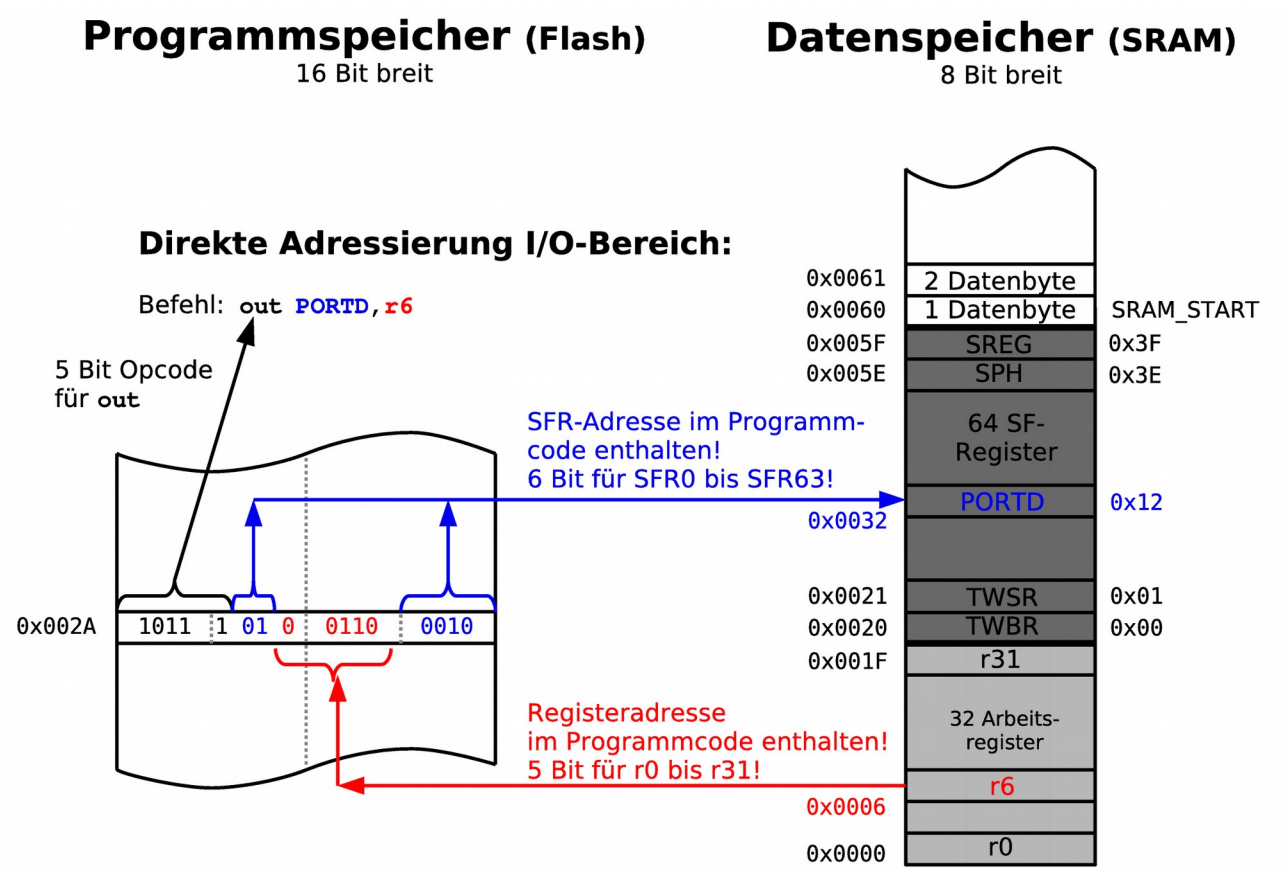
Resultat in Rd. Carry-Flag wird nicht subtrahiert (siehe sbc)
Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

Die direkte Adressierung der SF-Register (SonderFunktions-Register)

Es gibt sechs Befehle für die direkte Adressierung der SF-Register:

```
        in    r19,0x10        ;Adresse von PIND (Definitionsdatei)
        out   PORTD,Tmp1
        sbi   PORTD,4
        cbi   0x18,PB7        ;Adresse von PORTB (Definitionsdatei)
        sbic  PORTA,2
        sbis  0x18,PB3
```

Die Befehle "`sbi`", "`cbi`", "`sbic`" und "`sbis`" können nur die unteren 32 SF-Register (0-31) ansteuern. Bitmanipulationen der oberen 32 Register müssen also mit Maskierungen erfolgen.



Die direkte Adressierung des Datenspeichers (SRAM)

Für die direkte Adressierung des Datenspeichers existieren nur 2 Befehle ("**`lds`**" und "**`sts`**"). Es sind die einzigen Befehle die 2 Befehlsworte (32 Bit) benötigen. Es sind zwei Taktzyklen nötig. Das höherwertige Befehlswort enthält den Opcode und die Adresse des Arbeitsregisters. Das niederwertige Befehlswort enthält die SRAM-Adresse, die also unveränderbar im Programmcode (Flash) enthalten ist.

Es gibt zwei Befehle für die direkte Adressierung des SRAM:

```
        sts   0x60,Tmp1       ;Speichere den Inhalt des Registers Tmp1
                              ;in den Datenspeicher auf die
                              ;Anfangsadresse 0x0060 (engl. store)
        lds   r16,0x0AAA      ;Lade den Inhalt der SRAM-Adresse 0x0AAA
                              ;in das Arbeitsregister r16 (engl. load)
```

Der ganze Adressbereich des SRAM kann adressiert werden. Es ist also auch möglich die Arbeitsregister und die SF-Register so zu adressieren. Wegen der größeren und langsameren Befehle macht das aber wenig Sinn, da bessere Befehle zur Verfügung stehen.

Mit zusätzlichem externen Speicher ist eine Adressierung bis 64 KiB möglich (**`0xFFFF`**).

`sts k,Rr`

Kopiert den Registerinhalt von Rr (8 Bit) direkt ins SRAM (*store direct to data space (sram)*).

1	0	0	1	0	0	1	r	r	r	r	r	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

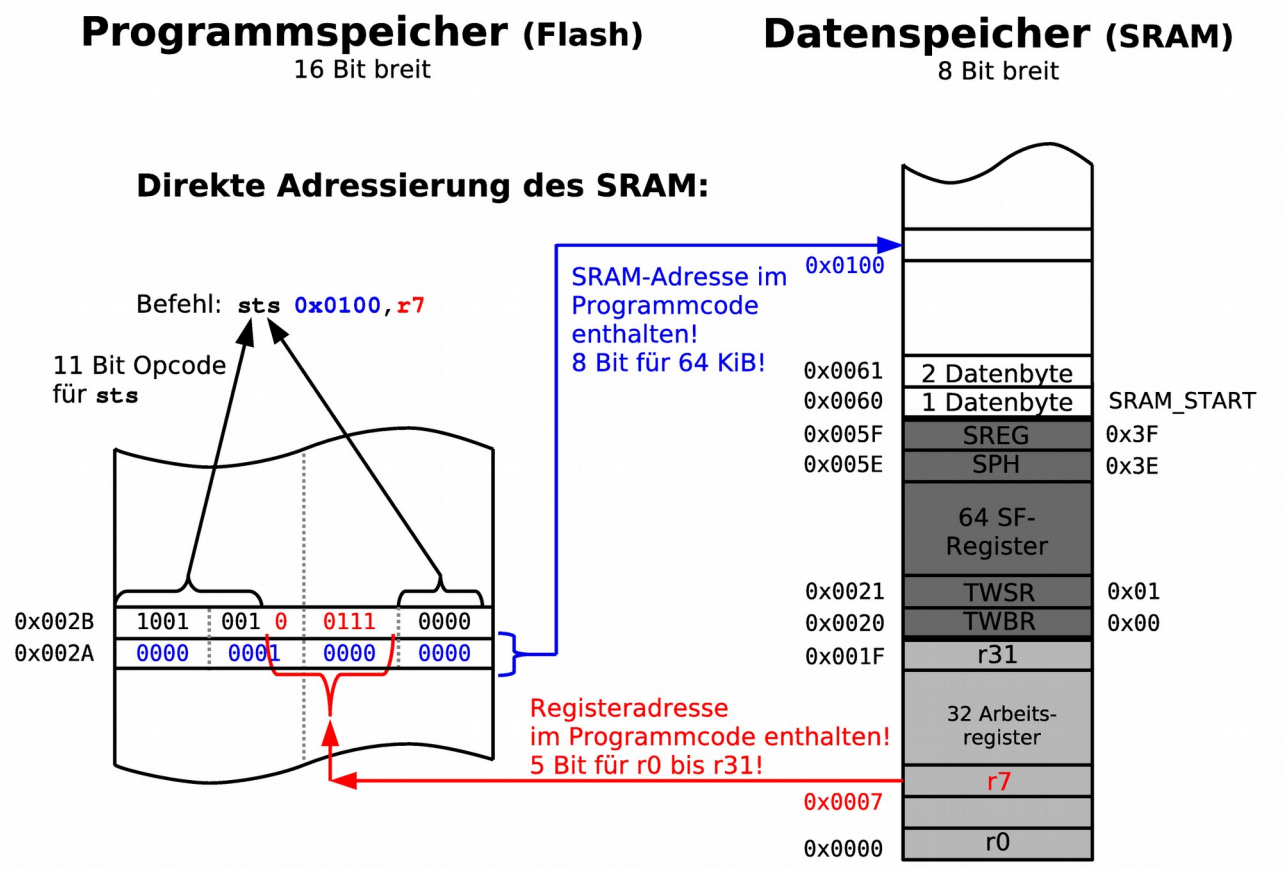
Direkte Adressierung! Die 16 Bit-Adresse ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).
Beeinflusste Flags: keine **Taktzyklen: 2**

`lds Rd,k`

Kopiert eine Speicherzeile (8 Bit) aus dem SRAM ins Register Rd (*load direct from data space (sram)*).

1	0	0	1	0	0	0	d	d	d	d	d	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

Direkte Adressierung! Die 16 Bit-Adresse ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).
Beeinflusste Flags: keine **Taktzyklen: 2**



Bei der direkten Adressierung verwendet man meist symbolische Adressen (Label). Beim Assemblieren werden diese Label dann durch die eigentliche Adresse ersetzt (siehe Modul B).

Die direkte Adressierung des Datenspeichers ist unflexibel, da die Adresse nicht durch das Programm verändert werden kann. Bei der Bearbeitung mehrerer Daten ist für jeden Zugriff eine Zeile Programmcode nötig. Für die Adressierung größerer Datenbestände wird man die indirekte Adressierung verwenden.

⌂ **A403**

a) Schreibe das vorige Programm so um, dass zuerst alle vier Zahlen in den Datenspeicher geladen werden (ab Adresse **0x0100**). Auch sollen alle Resultate der Berechnungen im Datenspeicher abgelegt werden. Es sollen nur zwei Arbeitsregister verwendet werden. Versuche mit einem Minimum an Programmzeilen auszukommen.
Gib dem Programm den Namen "**A403_sram_direct.asm**".

b) Teste das Programm im Studio 4 im Step-Modus. Beobachte die Veränderungen im Speicher indem du das Speicherfenster einschaltest ("View/Memory" oder "Alt+4").

Die indirekte Adressierung

Bei der indirekten Adressierung ist die Adresse des Operanden nicht direkt im Befehl enthalten sondern in einem der drei Registerpaare **X**, **Y** oder **Z**, welche als Adresszeiger (Indexregister, Pointer) dienen.

Dies bietet den großen Vorteil, dass die Adresse veränderbar ist und somit zum Beispiel in einer Schleife erhöht werden kann um große Datensätze oder Tabellen zu adressieren. Da die Adresse 16 Bit groß ist, werden für sie zwei Arbeitsregister benötigt. Hierfür sind die Arbeitsregister **r26**-**r31** vorgesehen welche als Doppelregister **X** (**r27**:**r26**), **Y** (**r29**:**r28**) und **Z** (**r31**:**r30**) angesprochen werden können.

Vor der indirekten Adressierung muss der Adresszeiger initialisiert werden!

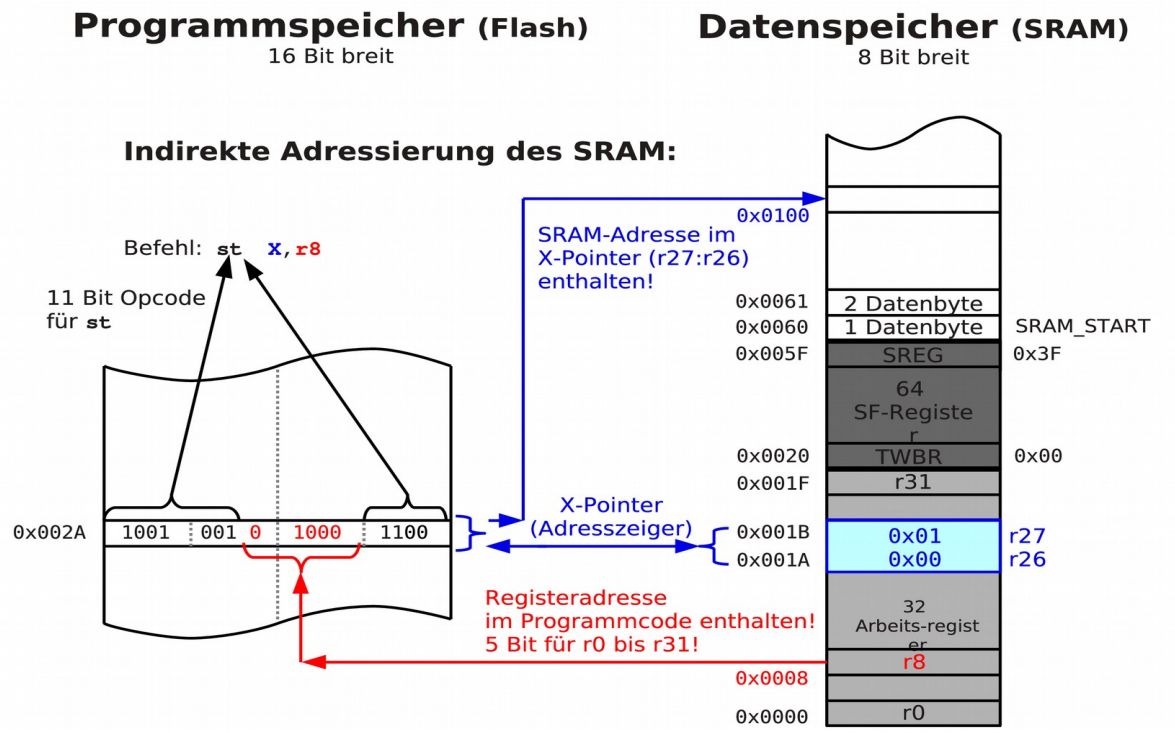
Beispiel:

```
        ldi  XL,0x00 ;Adresszeiger X mit 0x0100 initialisieren
        ldi  XH,0x01 ;
```

Mit der indirekten Adressierung kann ebenfalls der gesamte SRAM-Bereich adressiert werden, also im Notfall auch die Arbeits- und SF-Register sowie auch eventuell vorhandener externer Speicher (64 KiB).

Einige Befehle für die indirekte Adressierung des SRAM (es existieren je ein "**ld**" (*load*) und ein "**st**" (*store*) Befehl pro Doppelregister (6 mögliche Befehle)):

```
        st   X,Tmp1       ;Speichere den Inhalt des Registers Tmp1 in
                          ;den Datenspeicher. Die Adresse befindet sich
                          ;im Doppelregister X (engl. store)
        ld   r16,Y        ;Lade den Inhalt der SRAM-Adresse die sich im
                          ;Doppelregister Y befindet in das Arbeits-
                          ;register r16 (engl. load)
```



⌂ **A404**

a) Entwickle das Flussdiagramm eines Programms, das den SRAM-Speicher ab der Adresse **0x0100** mit den Dezimalzahlen **0** bis **255** auffüllt.

b) Schreibe das Assemblerprogramm. Benutze das **X**-Register um den Adresszähler zu initialisieren (Das niederwertige Byte kann mit "**XL**" adressiert werden, das höherwertige Byte mit "**XH**" (siehe Definitionsdatei)). Nenne das Programm "**A404_sram_indirect_1.asm**".

c) Teste das Programm im Studio 4 zuerst im Step-Modus. Lass das Programm dann ganz durchlaufen ("Run" ("F5") dann "Break" ("Ctrl+F5")) und
beobachte das Resultat im Speicherfenster ("View/Memory" oder "Alt+4").

st X,Rr

Kopiert den Registerinhalt von Rr (8 Bit) indirekt mittels Adresszeiger X ins SRAM (*store indirect from register to data space (sram) using index X*).

1 0 0 1	0 0 1 r	r r r r	1 1 0 0

Indirekte Adressierung! Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).

Beeinflusste Flags: keine Taktzyklen: 2

`ld Rd,X`

Kopiert eine Speicherzelle (8 Bit) aus dem SRAM indirekt mittels Adresszeiger X ins Register Rd (*load indirect from data space (sram) to register using index X*).

1	0	0	1	0	0	0	d	d	d	d	d	1	1	0	0

Indirekte Adressierung! Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).
Beeinflusste Flags: keine Taktzyklen: 2

`inc Rd`

Inkrementiere Rd (*increment*).

1	0	0	1	0	1	0	d	d	d	d	d	0	0	1	1

Rd ← Rd + 1. Resultat in Rd. Das Carry-Flag wird nicht beeinflusst!
Beeinflusste Flags: S, V, N, Z Taktzyklen: 1

`dec Rd`

Dekrementiere Rd (*decrement*).

1	0	0	1	0	1	0	d	d	d	d	d	1	0	1	0

Rd ← Rd - 1. Resultat in Rd. Das Carry-Flag wird nicht beeinflusst!
Beeinflusste Flags: S, V, N, Z Taktzyklen: 1

Bemerkung: Bei der indirekten Adressierung muss praktisch immer der Adresszeiger laufend erhöht oder erniedrigt werden. Werden mehr als 256 Speicherzeilen adressiert, so reicht der einfache Inkrement bzw. Dekrement-Befehl der ein Byte adressiert auch nicht mehr aus und es muss mit 16 Bit gerechnet werden (Befehle "**`adiw`**" bzw. "**`sbiw`**"). Um dies zu vereinfachen bietet die ATmega-Serie die indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers.

Indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers

Einige Befehle für die indirekte Adressierung des SRAM mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers (aus 12 möglichen Befehlen):

```
        st   Y+,Tmp1      ;Speichere den Inhalt des Registers Tmp1 in
                          ;den Datenspeicher. Die Adresse befindet sich
                          ;im Doppelregister Y. Erhoehe danach den
                          ;Adresszeiger um Eins (Y=Y+1)
        st   -Z,Tmp1      ;Erniedrige zuerst den Adresszeiger um Eins
                          ;(Z=Z-1). Speichere danach den Inhalt des
                          ;Registers Tmp1 in den Datenspeicher (Adresse
                          ;in Z)
        ld   r16,Z+       ;Lade den Inhalt der SRAM-Adresse die sich im
                          ;Doppelregister Z befindet in das Arbeits-
                          ;register r16 und erhoehe dann den
                          ;Adresszeiger um Eins (Z = Z+1)
        ld   r16,-X       ;Erniedrige zuerst den Adresszeiger um Eins
                          ;(X=X-1) und lade dann den Inhalt der SRAM-
                          ;Adresse die sich im Doppelregister X befindet
                          ;in das Arbeitsregister r16.
```

Beim Inkrementieren (Post-Inkrement) wird immer zuerst gespeichert bzw. geladen und dann erst inkrementiert. Inkrementiert wird **nach der Operation! Im Befehl befindet sich das Pluszeichen hinter dem Adresszeiger.**
Beim Dekrementieren ist es umgekehrt (Pre-Dekrement)! Der Adresszeiger wird zuerst dekrementiert ehe die Daten gespeichert oder geladen werden. Dekrementiert wird **vor der Operation! Im Befehl befindet sich das Minuszeichen immer vor dem Adresszeiger.**

Bemerkung: Der folgende Post-Inkrement-Befehl:

```
        st   Y+,Tmp1
```

spart eine Befehlszeile und 2 Taktzyklen. Er entspricht den beiden Befehlen:

```
        st   Y,Tmp1
        adiw YL,1
```

Der folgende Pre-Dekrement-Befehl:

```
        st   -Z,Tmp1
```

spart ebenfalls eine Befehlszeile und 2 Taktzyklen. Er entspricht den beiden Befehlen:

```
        sbiw ZL,1
        st   Z,Tmp1
```

Es existieren keine Post-Dekrement oder Pre-Inkrement-Befehle.

⌂ **A405** Ändere das vorige Programm so um, dass der Speicher ab der Adresse **0x0060** bis zur Adresse **0x03FF** mit dem Buchstaben '**B**' auffüllt wird (siehe ASCII-Tabelle im Anhang). Es soll ein Post-Inkrement-Befehl mit dem **Y**-Register verwendet werden.

Nenne das Programm: "**A405_sram_indirect_postincrement.asm**".
Teste das Programm im Studio 4.

st Y+,Rr

Kopiert den Registerinhalt von Rr (8 Bit) indirekt mittels Adresszeiger Y (Post-inkrementiert) ins SRAM (*store indirect from register to data space (sram) using index Y*).

1	0	0	1	0	0	1	r	r	r	r	r	1	0	0	1

Indirekte Adressierung! **Nach dem Kopieren wird der Adresszeiger automatisch erhöht!** Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).
Beeinflusste Flags: keine Taktzyklen: 2

Indirekte Adressierung mit festem (konstantem) Abstand

Die ATmega-Serie bietet eine weitere Annehmlichkeit bei der indirekten Adressierung. Es besteht die Möglichkeit mehrere Datenbytes mit einem festen Abstand zur momentanen Adresse (Adresszeiger) zu adressieren. Dies bietet eine Vereinfachung bei der Adressierung von Tabellen mit festen Datensätzen.

Es können nur die Adresszeiger **Y** und **Z** benutzt werden! Der Abstand (***displacement***) kann 5 Bit betragen (0-63). Der Abstand wird für die Operation zum Adresszeiger dazu addiert, wobei dessen Inhalt nicht verändert wird.

Es gibt vier Befehle für die indirekte Adressierung des SRAM mit festem Abstand:

```
std  Y+q,Tmp1     ;Speichere den Inhalt des Registers Tmp1 in
                  ;den Datenspeicher. Die Adresse ist die Summe
                  ;aus dem Doppelregister Y und dem Abstand q
std  Z+q,r20      ;Speichere den Inhalt des Registers r20 in
                  ;den Datenspeicher. Die Adresse ist die Summe
              ;aus dem Doppelregister Z und dem Abstand q
ldd  r16,Y+q     ;Lade den Inhalt der SRAM-Adresse (Y+q) in das
                  ;Arbeitsregister r16
ldd  Tmp1,Z+q   ;Lade den Inhalt der SRAM-Adresse (Z+q) in das
                   ;Arbeitsregister Tmp1
```

⌂ **A406** Ein Datenloggerprogramm hat im SRAM des ATmega-Controllers ab der Adresse **0x0100** jede Stunde 8 Mittelwerte von 8 Sensoren (Sensor 1 bis Sensor 8) abgespeichert. Jeweils nach einer Woche (168 Stunden) sollen die Werte der Sensoren drei, eins und fünf in dieser Reihenfolge auf den LEDs (**PORTD**) ausgegeben werden.
a) Zeichne das Flussdiagramm und schreibe ein Programm um diese Aufgabe

zu erledigen.
Nenne das Programm: "**A406_sram_indirect_displacement.asm**".

b) Teste das Programm im Studio 4. Gib dazu vor dem schrittweisen "Debugging" manuell 20 Werte im Speicherfenster ab Adresse **0x0100** ein. Beobachte die Adresszeiger, die Register und Port D im Ein-/Ausgabefenster (I/O-View).

ldd Rd,Z+q

Kopiert eine Speicherzelle (8 Bit) aus dem SRAM indirekt mittels Adresszeiger Z (+ Abstand) ins Register Rd (*load indirect from data space (sram) to register using index Z*).

1	0	q	0	q	q	0	d	d	d	d	0	q	q	q

Indirekte Adressierung! **Zum Adresszeiger wird ein fester (konstanter) Abstand q vor der Adressierung hinzu addiert.** Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren.
Beeinflusste Flags: keine Taktzyklen: 2

adiw Rdl,K

Addiert die Konstante K (0-63) zu einem Doppelregister (*add immediate to word*).

1	0	0	1	0	1	1	0	K	K	d	d	K	K	K	K

Resultat im Doppelregister. Neben den Doppelregistern X,Y,Z kann auch das Doppelregister r25:r24 verwendet werden. Obschon das Doppelregister adressiert wird ist im Befehl nur das niederwertige Register anzugeben Bsp.: adiw YL,10. (andere Schreibweisen sind je nach Assembler möglich). Die Konstante kann maximal 63 betragen (6 Bit).
Beeinflusste Flags: S, V, N, Z, C Taktzyklen: 2

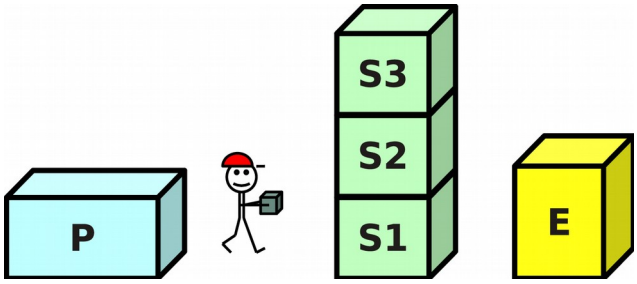
Indirekte Adressierung mit "push" und "pop"

Eine Sonderform der indirekten Adressierung erfolgt mit den Befehlen "**push**" und "**pop**" in Kombination mit dem Stapelzeiger SP als Adresszeiger. In einem späteren Kapitel wird diese Adressierung behandelt.

Wiederholung

Um die Adressierung besser zu verstehen, wollen wir die Adressierungsarten noch mal anhand eines Beispiels mit Transferbefehlen erläutern. Zu jedem Unterpunkt soll noch einmal die entsprechende Speicher-Grafik betrachtet werden.

Das kleine Märchen vom Laufburschen.



Ein Laufbursche soll Pakete austragen. Auf dem Gelände der Firma befinden sich drei Gebäude. Das Direktionsgebäude P (Programm), das Verwaltungsgebäude S (SRAM) und ein Lagerraum E (EEPROM).
Die Befehle erhält der Laufbursche im Direktionsgebäude P.
Das Verwaltungsgebäude hat drei Stockwerke: Im ersten Stock des Gebäudes befinden sich die Büros der Administration für gängige Arbeiten (Arbeitsregister). Im zweiten Stock die Büros für Im- und Export (SF-Register). Im dritten Stock das Archiv (Speicher).
Der Lagerraum E kann nur mit einer speziellen Genehmigung betreten werden.

- Bei einer **unmittelbaren Adressierung** erhält der Laufbursche sein Paket (**0xA3**) bereits im Direktionsgebäude und liefert es einfach an ein Administrationsbüro (**r18**) im Gebäude S (erster Stock S1) aus (**ldi r18,0xA3**).

- Bei den **direkten Adressierungen** holt er das Paket (**0xA3**) in einem Administrationsbüro (**r20**) im Gebäude S (erster Stock S1) ab, und trägt es in ein anderes Büro oder in das Archiv im selben Gebäude. Ziel- und Quelladresse der Büros (bzw. des Archivs) hat er zuvor im Direktionsgebäude erfragt. Je nach Stockwerk unterscheiden wir die:

  Registeradressierung: Er trägt das Paket in ein anderes Administrationsbüro (**r5**) im gleichen Stockwerk (**mov r5,r20**).

  Adressierung des Ein- Ausgabebereichs: Er trägt das Paket vom ersten (S1) in das zweite Stockwerk (S2), in ein Büro (**PORTD**) der Abteilung Im- und Export (**out PORTD,r20**).

  Adressierung des SRAM-Speichers: Er trägt das Paket vom ersten in das dritte Stockwerk (S3) ins Archiv. Hier wird eine größere Adresse (16 Bit) benötigt, da das Archiv sehr viele Kisten zum Archivieren besitzt (**sts 0x0AAA,r20**)

- Bei der **indirekten Adressierung** kennt der Laufbursche die Adresse nicht, aber im Direktionsgebäude (P) teilt man ihm mit in welchem der drei Großraumbüros **X**, **Y** oder **Z** er die Adresse zuvor abholen soll. Erst dann kann er das Paket ausliefern (**st X,r20**). Meist ist es eine große Adresse für das Archiv (S3).

  Pre-Dekrement: Hierbei kann es vorkommen, dass er dem Beamten im Großraumbüro mitteilen muss, dieser soll die Adresse um Eins erniedrigen, bevor er sie dem Laufburschen aushändigt (**st -X,r20**).
  Post-Inkrement: Es kann auch vorkommen, dass der Laufbursche die Adresse entgegen nimmt und dann dem Beamten mitteilt, er soll die im

Großraumbüro aufbewahrte Adresse um Eins erhöhen (**`st X+,r20`**).
Displacement: Eine dritte Möglichkeit ist die, dass man dem Laufburschen im Direktionsgebäude eine Zahl (0-63) nennt, welche er zur Adresse (nachdem er diese im Großraumbüro erhalten hat) addieren soll (**`std X+3,r20`**).

⌂ **A407** a) Zeichne ein Flussdiagramm eines Programms, das den Speicherbereich von
der Adresse **`0x0060`** bis **`0x01FF`** (erste Tabelle) nach **`0x0300`** (zweite Tabelle) verschiebt (Cut and Paste). Anstelle der verschobenen Daten soll nachher das Null-Byte (siehe ASCII-Tabelle) in der ersten Tabelle stehen.
b) Schreibe das entsprechende Programm und gib dem Programm den Namen
"**`A407_sram_indirect_cut_paste.asm`**".
c) Teste das Programm im Studio 4. Gib dazu vor dem schrittweisen "Debugging" manuell 20 Werte im Speicherfenster ab Adresse **`0x0060`** ein. Beobachte die Adresszeiger und Register im Ein-/Ausgabefenster (I/O-View).

Adressierung des Programmbereichs

Beim Programmbereich unterscheiden wir folgende Adressierungsarten:

- **Relative Adressierung des Programmspeichers**
(Befehle "**`rjmp`**" und "**`rcall`**")
- **Direkte Adressierung des Programmspeichers**
(Befehle "**`jmp`**" und "**`call`**")
- **Indirekte Adressierung des Programmspeichers**
(Befehle "**`ijmp`**" und "**`icall`**")
- **Indirekte Adressierung von Konstanten im Programmspeicher**
(Befehle "**`lpm`**")

Relative Adressierung des Programmspeichers mit "`rjmp`" und "`rcall`"

Meist werden relative Sprünge benutzt. Hierbei wird ein relativer positiver (Vorwärtssprung) oder negativer (Rückwärtssprung) Abstand **`k`** zur momentanen Adresse hinzu addiert. Der Assembler berechnet diesen Abstand mit Hilfe der Labels. Hierbei ist zu beachten, dass nach der Addition des Abstandes zum Programmzähler (*Program Counter* PC) dieser nochmals um Eins erhöht wird (PC = PC+k+1).

Bei relativen Sprüngen stehen im Opcode für den Abstand 12 Bit zur Verfügung. Es kann also maximal über 2k Worte vorwärts und rückwärts gesprungen werden. Soll im Programm noch weiter gesprungen werden, so muss die direkte Adressierung verwendet werden. Der Assembler überwacht ob die Grenze überschritten wird und meldet in diesem Fall einen Fehler.

Die relative Adressierung ist schneller und Platz sparender als die direkte Adressierung. Man soll also, falls möglich, diese Adressierung verwenden.

Hier ein kurzer Ausschnitt aus einer List-Datei:

```
00002e 9985      MAIN:  sbic    PIND,PButt    ;
00002f cffe             rjmp    MAIN
000030 e420             ldi     Mask,0x40     ;
000031 2702             eor     Tmp1,Mask     ;
000032 bb02             out     PORTD,Tmp1    ;
000033 cffa             rjmp    MAIN          ;
```

Der erste "**rjmp**"-Befehl (Opcode: **0xc**) springt ein Wort rückwärts. Der Abstand muss also -2 oder **0xfffffe** (Zweierkomplement!) betragen, da der PC nach der Addition noch um Eins erhöht wird!. Der zweite "**rjmp**"-Befehl springt 5 Worte rückwärts (**k** = -6 entspricht **0xfffffa**) .

Direkte Adressierung des Programmspeichers mit "jmp" und "call"

Bei der direkten Adressierung des Programmspeichers stehen 22 Bit! (4M Worte) für eine feste Adresse zur Verfügung. Die Befehle benötigen zwei Worte und drei oder vier Taktzyklen!

Indirekte Adressierung des Programmspeichers mit "ijmp" und "icall"

Für spezielle Anwendungen (z.B. Umsetzung von "switch"-Konstruktionen in C) kann eine indirekte Adressierung des Programmspeichers benutzt werden. Die Sprungadresse muss dazu zuerst im **Z**-Adresszeiger initialisiert werden.

Indirekte Adressierung von Konstanten im Programmspeicher mit "lpm"

Der Ladebefehl "**lpm**" (***load program memory***) ermöglicht es ein beliebiges Datenbyte aus dem Programmspeicher (Flash) in das Arbeitsregister **r0** zu laden. Die indirekte Adresse muss sich dazu im **Z**-Pointer (Adresszeiger) befinden.

Bei der ATmega-Familie ist es besser die beiden erweiterten "**lpm**" Befehle benutzen, da diese alle Arbeitsregister adressieren können und die Syntax mitteilt welches Register als Adresszeiger benutzt wird.

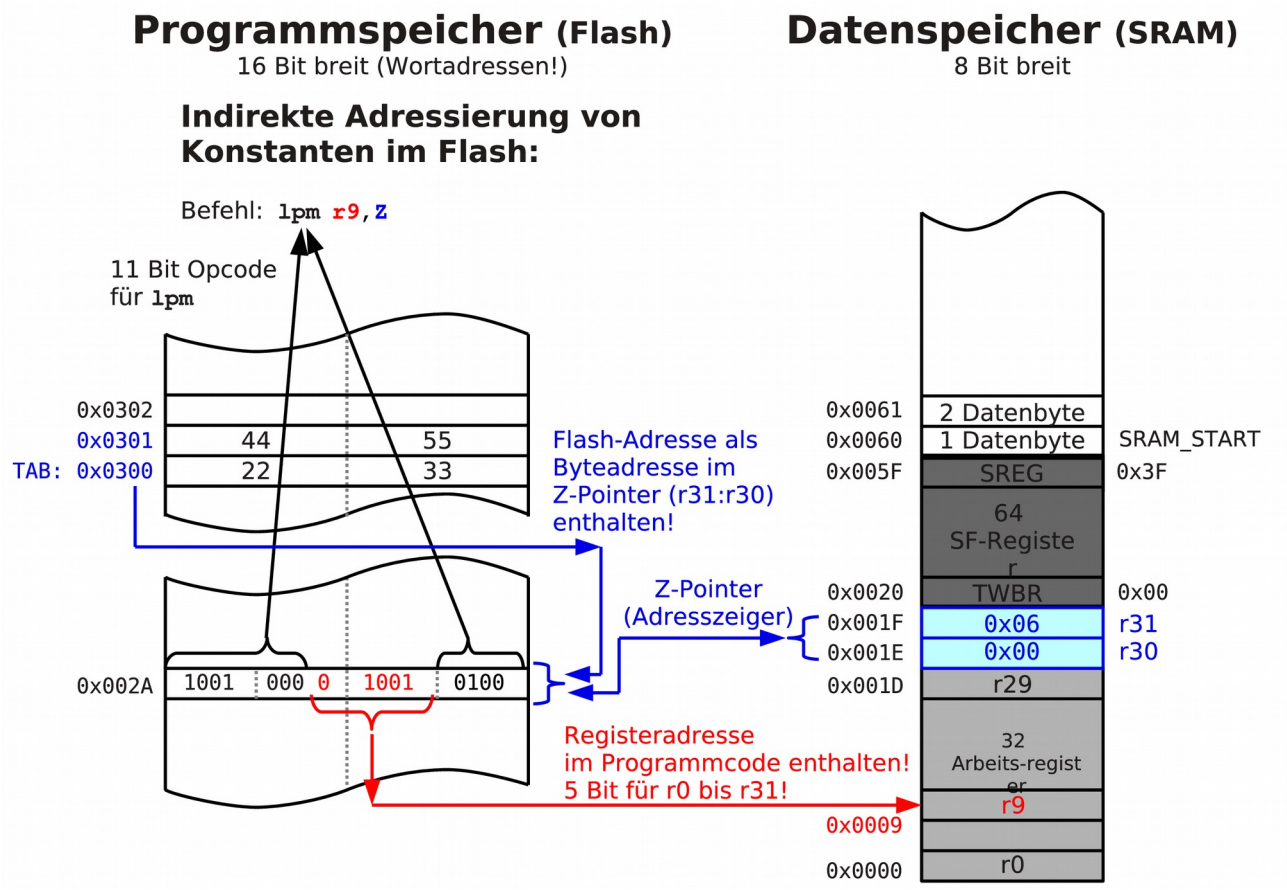
```
   lpm     r20,Z   ;Speichere den Inhalt der Programmzeile deren
                   ;Adresse im Z-Pointer steht in das Arbeits-
                   ;register r20.

   lpm     r20,Z+  ;Speichere den Inhalt der Programmzeile deren
                   ;Adresse im Z-Pointer steht in das Arbeits-
                   ;register r20. Inkrementiere danach den Adresszeiger Z
```

`lpm Rd,Z`

Kopiert eine Speicherzelle (8 Bit) aus dem Flash indirekt mittels Adresszeiger Z ins Register Rd (*load program memory*).

1	0	0	1	0	0	0	d	d	d	d	d	0	1	0	0

Indirekte Adressierung! Der gesamte Programmspeicher (Flash) des ATmega32A kann mittels Adresszeiger Z adressiert werden. Achtung! In Z muss die Byteadresse und nicht die Wortadresse enthalten sein (Byteadresse = 2 * Wortadresse).
Beeinflusste Flags: keine Taktzyklen: 3



Diese Adressierungsart wird meist in Kombination mit der **`.DB`** (bzw. **`.DW`**) und eventuell der **`.ORG`** Direktive verwendet. Damit können Tabellen mit Konstanten (zum Beispiel Zeichenketten (Strings)) gleich beim Programmieren im Programmspeicher abgelegt werden. Die recht aufwändige Programmierung des EEPROM kann so vermieden werden.

.ORG	Adresse	Legt eine Anfangsadresse fest ab der der folgende Code abgespeichert wird. Hiermit kann der Speicher organisiert werden. Beispiel: **.ORG = 0xA00**
.DB	Liste mit Bytekonstanten	(engl.: „define Byte“) Fügt konstante Bytes ein. Dabei ist es die Bedeutung der Bytes egal (Zahl von 0..255, ASCII-Zeichen 'b', eine Zeichenkette "Hallo"; alle Bytes werden durch Kommas getrennt). Im Flash muss eine gerade Zahl von Bytes eingefügt werden (16 Bit-Worte), sonst hängt der Assembler ein Nullbyte an.
.DW	Liste mit Wortkonstanten	(engl.: „define Word“) Fügt konstantes binäres Wort (16 Bit) ein. Im EEPROM und Flash zuerst das niederwertige Byte, dann das höherwertige Byte.

Beispiel für die Programmierung einer Tabelle:

```
;-------------------------------------------------------------------------
;       Tabelle      (Konstantenbereich im Flash (hinter dem Programm))
;-------------------------------------------------------------------------
.ORG    0x0300                  ;ab Adresse 0x0300 im Programmspeicher
TAB:
.DB     0x22,0x33,0x44,0x55     ;Tabelle mit 4 Byte
```

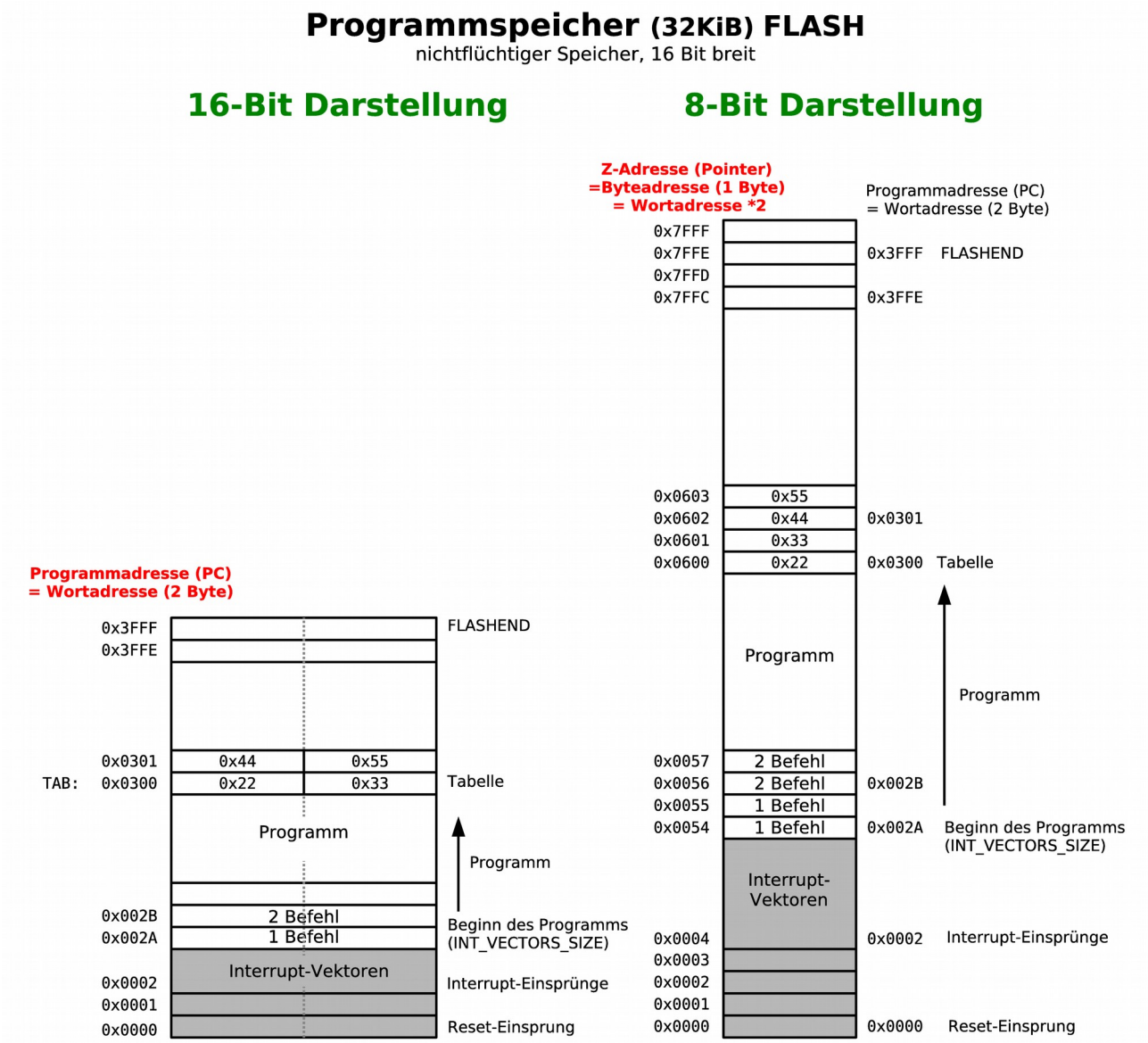
Auf diese Tabellen kann dann natürlich nur Lesend (load) zugegriffen werden.

Bemerkung: Die Speicherorganisation mit der der **.ORG** Direktive kann Programme vereinfachen. Bei großen Projekten mit unterschiedlichen Bibliotheken (bzw. Unterprogrammen) kann es allerdings zu Überschneidungen und damit zu Fehlern führen. Hier ist es besser die Speicherorganisation ausschließlich mit symbolischen Adressen (Labeln) vorzunehmen.

Da der Flash-Speicher in Worten (2 Byte) organisiert ist und intern in Worten adressiert wird, ist es nötig die Wortadresse mit dem Faktor zwei zu multiplizieren, da der Adresszeiger (hier **Z**) mit einer byteweisen Adressierung arbeitet[22].

Bei der Initialisierung des Z-Adresszeigers muss in unserem Beispiel also **0x0600** (Byteadresse) statt **0x0300** (Wortadresse) als Anfangsadresse verwendet werden.

22 Eine wortweise Adressierung macht keinen Sinn, da der Ladebefehl nur Werte in 1 Byte großen Arbeitsregistern ablegen kann. Jedes zweite Byte könnte so nicht genutzt werden.



Die Berechnung kann man getrost dem Assembler überlassen.

```
        ldi     ZL,LOW(TAB*2)   ;Adresszeiger mit der Adresse der Tabelle
        ldi     ZH,HIGH(TAB*2)  ;* 2 (Worte statt Bytes) initialisieren
```

Der verwendete Label **Tab** steht dabei für die durch Anfangsadresse der Tabelle.

Im Folgenden ein Beispielprogramm zur Verwendung von Tabellen im Programmspeicher.

```
;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
        ;Adresszeiger initialisieren
        ldi     ZL,LOW(TXTSTR*2) ;Adresszeiger mit der Adresse der Texttabelle
        ldi     ZH,HIGH(TXTSTR*2);* 2 (Worte statt Bytes) initialisieren
        ;alle Pins von PortD als Ausgang initialisieren
        ldi     Tmp1,0xFF       ;DDRD = 11111111b
        out     DDRD,Tmp1       ;alle Bits im Datenrichtungsregister auf Eins
```

```
;-------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------
MAIN:   lpm     Tmp1,Z+         ;Speichere das Zeichen der Speicherzeile deren
                                ;Adresse im Z-Pointer steht in das Arbeits-
                                ;register Tmp1. Inkrementiere danach den
                                ;Adresszeiger Z
        tst     Tmp1            ;Letztes Zeichen (Nullbyte) erreicht?
        breq    END             ;Falls ja Ende
        out     PortD,Tmp1      ;Gib das Zeichen an PortD aus
        rjmp    MAIN            ;Hole nächstes Zeichen

        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
END:    rjmp    END             ;Endlosschleife

;-------------------------------------------------------------------------
;       Tabellen im Programmspeicher (Flash)
;-------------------------------------------------------------------------
.ORG    0x0300                  ;ab Adresse 0x0300
TXTSTR: .DB     "MICEL",13,0    ;Text mit "carriage return" sowie
                                ;abschliessendem Nullbyte (siehe ASCII-Tab.)

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

Bemerkung: Die Tabelle im Beispiel besteht aus acht ASCII-Zeichen. Nach den fünf ASCII Buchstaben "MICEL", kommen drei ASCII-Steuerzeichen, die einfach durch Komma getrennt in Dezimal angeschrieben sind. Das erste Steuerzeichen ist ein Wagenrücklauf und das zweite Steuerzeichen ein Zeilenvorschub (siehe ASCII-Tabelle im Anhang). Das letzte Steuerzeichen (Nullbyte) wird verwendet um das Ende der Tabelle zu erkennen.

⌂ **A408**

a) Teste das obige Programm im Studio 4. Stell das Speicherfenster beim Betrachten des Programmspeichers auf 2 Kolonnen ein (wortweise Adressierung, 16 Bit).
Nenne das Programm: "**A408_flash_indirect_textstring.asm**".

b) Was ist die Aufgabe des Programms?

⌂ **A409**

a) Zu welcher Befehlsgruppe gehören die folgenden Befehle?

b) Nenne ebenfalls die jeweilige Adressierungsart:

```
nop             .....
lsl   r5        .....
ldd   X+3,r20   .....
lpm             .....
swap  r20       .....
rjmp  NOLOOP    .....
sbic  0x11,3    .....
adiw  r28,4     .....
com   Tmp1      .....
ld    r22,-Z    .....
```

```
movw  r16,r18   ........................................
icall           ........................................
and   r1,r2     ........................................
```

A5 Zeitschleifen

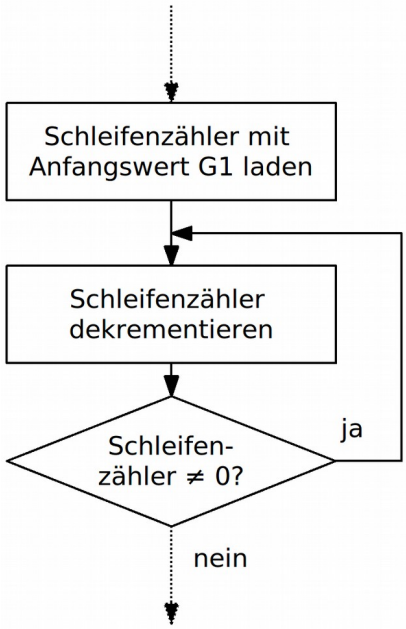
Als nächstes soll ein Blinklicht programmiert werden. Der Controller ist aber für diese Aufgabe viel zu schnell. Es muss also zwischen dem Ein- und Ausschalten der LED eine Verzögerung einbaut werden, so dass unser Auge dem Blinken folgen kann. Dies passiert mit einer so genannten Zeitschleife. Natürlich hat der ATmega-Chip auch interne Timer mit denen so etwas bewerkstelligt werden kann, ohne dass der Controller Zeit tot schlägt.

Für einfache Anwendungen werden dennoch immer wieder Zeitschleifen benötigt.

Zeitschleifen sind Programme, deren Aufgabe es ist Zeit zu verbrauchen.

8-Bit-Zeitschleife

Die erste einfachste Zeitschleife soll eine Zehntelsekunde verbrauchen. Um dies zu bewerkstelligen wird ein Schleifenzähler (zum Beispiel "**TCnt1**"[23]) mit seinem **Anfangswert G_1** initialisiert und dann auf Null abgezählt.



Für jeden Befehl in der Zeitschleife muss bestimmt werden, wie viele Takte erforderlich sind, um diesen Befehl auszuführen. Ist die Taktfrequenz des Controllers bekannt, so erhält man die Dauer der Zeitschleife.

Die folgenden Befehle sollen verwendet werden:

```
        ldi     TCnt1,0xFF      ;Schleifenzaehler TCnt1 = G1 (hier 255)
LOOP:   dec     TCnt1           ;Dekrementiere den Schleifenzaehler
        brne    LOOP            ;Verlasse die Schleife bei TCnt1=0
```

23 Zuweisung mittels Assembleranweisung (Bsp: **.DEF TCnt1 = r19**)

⌂ **A500**

a) Im Anhang befindet sich der Befehlssatz des ATmega32A. In diesem ist unter "#Clocks" angegeben wie viele Takte jeder einzelne Befehl benötigt. Ermittle anhand des Befehlssatzes die Anzahl der benötigten Takte für die benutzen Befehle und trage sie in die Tabelle ein.
b) Wieso werden beim Sprungbefehl wohl zwei Werte angegeben?
c) Vervollständige die folgende Tabelle und stelle daraus ein Formel auf zur Berechnung der (totgeschlagenen) **Zeit t** in Funktion des **Anfangswertes G_1**. Benutze die Abkürzung t_T für die **Zeitdauer eines Taktes** (entspricht der Periodendauer T eines Rechtecksignals).
d) Stelle die Formel nach G_1 um.

Befehle	Takte	Durchläufe
`ldi TCnt1,255`		
`LOOP: dec TCnt1`		
`brne LOOP`		

Formeln zur Berechnung der Zeit **t** bzw. des Anfangwertes G_1 einer **8-Bit-Zeitschleife**:

$t =$

$G_1 =$

⌂ **A501**

a) Ermittle die maximale Zeit die mit dieser Zeitschleife verbraucht werden kann einmal für die interne Frequenz von 1 MHz und einmal für eine externe Frequenz von 16 MHz. Die Zeitdauer eines Taktes t_T lässt sich aus der Frequenz errechnen mit $t_T = T = 1/f$.
b) Kann damit eine Zehntelsekunde abgedeckt werden?
c) Was passiert wenn man als Anfangswert Null in das Zählerregister setzt?

16-Bit-Zeitschleife

Leider reicht die Zeit die man mit einer 8-Bit-Zeitschleife erreichen kann nicht aus um das Blinken sichtbar zu machen.

sbiw Rdl,K

Subtrahiert die Konstante K (0-63) von einem Doppelregister (*subtract immediate from word*)

1	0	0	1	0	1	1	1	K	K	d	d	K	K	K	K

Resultat im Doppelregister. Neben den Doppelregistern X,Y,Z kann auch das Doppelregister r25:r24 verwendet werden. Obschon das Doppelregister adressiert wird ist im Befehl nur das niederwertige Register anzugeben Bsp.: adiw YL,10. (andere Schreibweisen sind je nach Assembler möglich). Die Konstante kann maximal 63 betragen (6 Bit).
Beeinflusste Flags: S, V, N, Z, C Taktzyklen: 2

⌂ **A502**

a) Schreibe das obige Programm so um, dass statt einem 8-Bit-Register ein Doppelregister verwendet wird. Nutze hier das Doppelregister **X**. Dieses Doppelregister wird mit **XL** und **XH** angesprochen. Zum Dekrementieren kann der Befehl "**sbiw XL,1**" verwendet werden.
(!Achtung! Obschon in der Syntax **XL** angegeben wird bezieht sich die Subtraktion auf das gesamte Doppelregister **X**).

b) Stelle die Formel zur Berechnung der Zeit **t** in Funktion des Anfangswertes G_1 auf.

c) Die Formel lässt sich Vereinfachen. Wie groß ist der maximale Fehler, der mit der vereinfachten Formel in Kauf genommen wird?

d) Ist es zulässig die vereinfachte Formel zu verwenden?

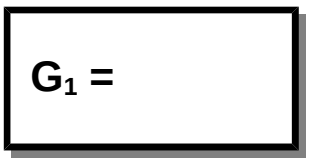
e) Stelle die (vereinfachte) Formel nach G_1 um.

f) Ermittle die maximale Zeit die mit dieser Zeitschleife verbraucht werden kann einmal für die interne Frequenz von 1 MHz und einmal für die externe Frequenz von 16 MHz wenn G_1 auf dezimal **65535** (**0xFFFF**) gesetzt wird.

g) Ermittle die Anfangswerte des Doppelregisters damit bei beiden Frequenzen eine Zehntelsekunde erreicht wird.

Befehle	Takte	Durchläufe
LOOP:		

Formeln zur Berechnung der Zeit **t** bzw. des Anfangswertes G_1 einer **16-Bit-Zeitschleife**:



⌂ **A503**

a) Ändere das Toggle-Programm aus dem Kapitel A3 so um, dass die LED blinkt (natürlich ohne Schalter). Dazu wird die vorhin erstellte 16-Bit-Zeitschleife eingebaut. Beginne mit dem maximalen Wert für den Schleifenzähler. Speichere das Programm als "**A503_16bit_loop.asm**".
b) Berechne die Blinkfrequenz und miss sie mit dem Oszilloskop.
c) Setze den Anfangswert so, dass die Schleife eine Zehntelsekunde benötigt. Wie hoch ist dann die Blinkfrequenz?
d) Ändere den Schleifenzähler so lange, bis das Blinken so eben noch sichtbar ist. Berechne die Blinkfrequenz.
e) Miss die Blinkfrequenz mit dem Oszilloskop und vergleiche sie mit dem errechneten Wert. Sind Pausendauer und Impulsdauer gleich?

Bemerkung: Statt mit dem "**sbiw**"-Befehl kann eine 16-Bit Subtraktion auch in zwei Schritten mit den beiden Befehlen "**subi**" und "**sbci**" durchgeführt werden. Mit "**subi**" werden die **niederwertigen Register** subtrahiert. Der eventuell auftretende Übertrag (carry) durch Borgen wird vom "**sbci**"-Befehl der die beiden **hochwertigen Register** subtrahiert zusätzlich abgezogen. Die Formel zur Berechnung der Zeitschleife bleibt gleich, da beide Befehle zusammen auch nur zwei Taktzyklen benötigen.

```
LOOP:   subi    XL,1            ;Dekrementiere den Schleifenzaehler
        sbci    XH,0
        brne    LOOP            ;Verlasse die Schleife bei Schleifenzaehler X=0
```

subi Rd,K

Subtrahiere die Konstante K (8 Bit) vom Register Rd (*subtract immediate*).

0	1	0	1	K	K	K	K	d	d	d	d	K	K	K	K

Resultat in Rd. Nur r16-r31!
Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

sbci Rd,K

Subtrahiere die Konstante K (8 Bit) und das Carry-Flag vom Register Rd (*subtract immediate with carry*).

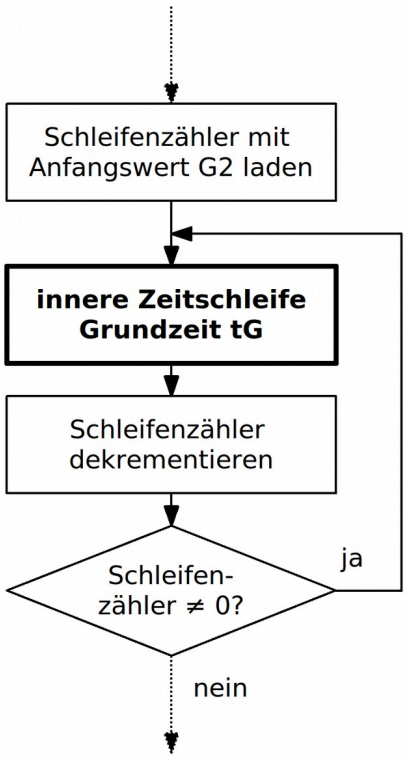
0	1	0	0	K	K	K	K	d	d	d	d	K	K	K	K

Resultat in Rd (Rd = Rd-K-C). Nur r16-r31!
Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

Verschachtelte Zeitschleifen.

Um längere Zeiten zu erreichen, können die Zeitschleifen verschachtelt werden. In einer äußeren Zeitschleife wird dann eine innere Schleife G_2 mal ausgeführt. Die innere Schleife erhält eine feste Grundzeit t_G von zum Beispiel einer Millisekunde. Die Gesamtzeit ist dann ein Vielfaches dieser Grundzeit $t = G_2 \cdot t_G$.
(Bsp.: t_G = 1ms; G_2 = 1000; t = 100·1ms = 1 s).

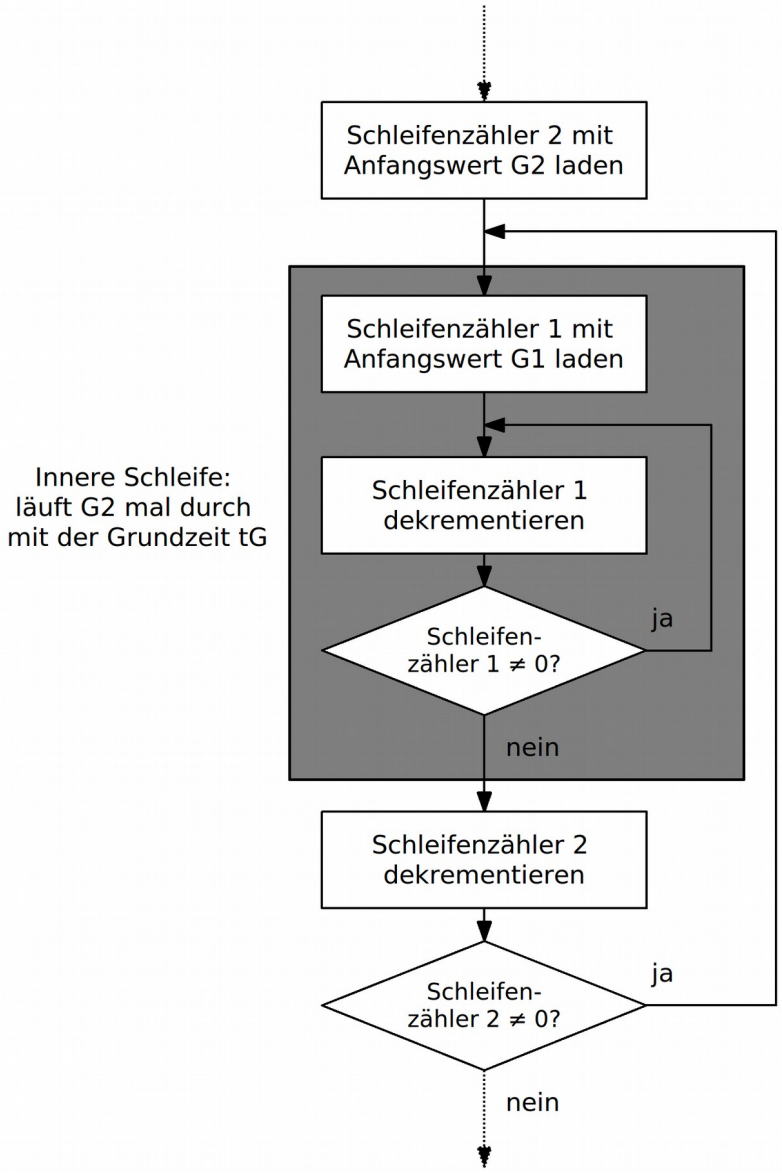
Prinzip der verschachtelten Zeitschleife:



Wenn die Grundzeit t_G der inneren Schleife viel größer ist als die Zeit, die für die Befehle der äußeren Schleife benötigt werden, (hier "**sbiw**" und "**brne**", max. 4 Takte) kann die Gesamtzeit mit $t = G_2 \cdot t_G$ angenommen werden:

$$t = G_2 \cdot t_G$$

Gesamtes Flussdiagramm:



⌂ **A504** Ersetze die einfache 16-Bit-Zeitschleife im vorigen Programm durch eine verschachtelte 16-Bit-Zeitschleife. Verwende das Doppelregister `r25:r24` als Schleifenzähler 2. Dieses Doppelregister soll mit `WL` und `WH` angesprochen werden (siehe **.DEF**-Anweisungen in der Vorlage). Zum Dekrementieren kann der Befehl "`sbiw W,1`"[24] verwendet werden.
Die Grundzeit soll eine Millisekunde betragen. Die LED soll mit einer Frequenz von 1 Hz aufleuchten. Speichere das Programm als "**A504_16bit_nested_loop_1.asm**".

24 Beim Befehl `sbiw r24,1` wird das niederwertigste Register (LBYTE) angegeben obschon sich die Subtraktion auf das gesamte Doppelregister (`r25:r24`) bezieht.

Bemerkung: Eine Zeitschleife kann natürlich auch mehrfach verschachtelt werden.

Externer Quarz mit 16 MHz

Bis jetzt wurde mit dem internen RC-Oszillator von 1 MHz gearbeitet, für den der ATmega32A bei der Auslieferung programmiert wurde. Es soll für alle weiteren Versuche jedoch ein präziserer externer Quarz mit 16 Mhz verwendet werden. Hierzu müssen die Fuse-Bits des ATmega32A umprogrammiert werden. Das Programmieren der Fuse-Bits wird im Anhang beschrieben und sollte jetzt vorgenommen werden.

⌂ **A505** Ändere das vorige Programm so um, dass es trotz der veränderten Quarzfrequenz genau das gleiche tut wie in der vorigen Aufgabe. Speichere das Programm als "**A505_16bit_nested_loop_2.asm**".

⌂ **A506** Zeichne ein Flussdiagramm und schreibe ein Programm, welches ein Rechtecksignal mit einer Frequenz von 100 Hz ausgeben kann. Das Tastverhältnis (Tastgrad, engl. duty cycle, Verhältnis der Länge des eingeschalteten Zustands (Impulsdauer) zur Periodendauer bei einem Rechtecksignal) soll 33 % betragen. Kontrolliere das Ergebnis mit dem Oszilloskop und kommentiere die Genauigkeit.
Speichere das Programm als "**A506_frequency_generator_1.asm**".

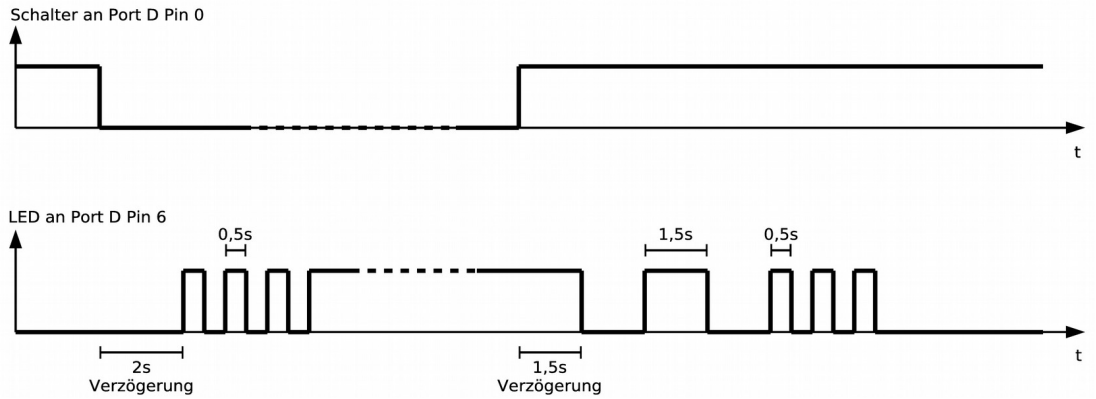
⌂ **A507** a) Zeichne ein Flussdiagramm und schreibe ein Programm, welches 4 verschiedene Rechtecksignale ausgeben kann. Die Frequenz wird über zwei Schalter nach folgender Vorgabe eingestellt:

0b00 → 1 Hz
0b01 → 10 Hz,
0b10 → 100 Hz,
0b11 → 500 Hz

Die Grundzeit der Zeitschleife soll 0,2 ms betragen. Kontrolliere das Ergebnis mit dem Oszilloskop und kommentiere die Genauigkeit. Speichere das Programm als "**A507_frequency_generator_2.asm**".

b) Für Fleißige:
Bei hohen Frequenzen kann die zur Schalterabfrage benötigte Zeit die Genauigkeit der Frequenz beeinflussen. Ändere das Programm so um, dass die Schalter nur ein mal pro Sekunde abgefragt werden. Speichere das Programm als "**A507_frequency_generator_3.asm**".

⌂ **A508** Schreibe ein Assemblerprogramm, das eine LED nach folgendem Muster ein- bzw. ausschaltet. Speichere das Programm als "**A508_pulse_delay_1.asm**"


Schalter an Port D Pin 0
t
LED an Port D Pin 6
0,5s
1,5s
0,5s
2s
Verzögerung
1,5s
Verzögerung
t

A6 Unterprogramme

Die letzte Aufgabe hat gezeigt, dass es nicht sehr praktisch ist gewisse Programmteile im Code mehrfach zu wiederholen. Diese Programmteile lassen sich sehr günstig in einem Unterprogramm[25] (*subroutine*) zusammenfassen.

Unterprogramme sind Programmteile, die von einem Hauptprogramm aus (oder aus einem anderen Unterprogramm) aufgerufen werden und nach der Ausführung wieder an dieselbe Stelle des Hauptprogramms zurückspringen.

Häufig stellt man bei der Analyse einer neuen, umfangreichen Programmieraufgabe fest, dass bestimmte Schrittfolgen mehrfach in derselben Weise vorkommen. Hier setzt die Unterprogramm-Technik ein.

Unterprogramme sind also Hilfsprogramme, die für Sonderaufgaben eingesetzt werden.

Eigenschaften von Unterprogrammen:

- Unterprogramme unterteilen die Aufgabe in Teilprobleme, die sich einzeln besser programmieren und testen lassen. Oft ist es von Vorteil, zuerst die Unterprogramme zu entwerfen und zu testen, bevor mit der Programmierung des Hauptprogramms begonnen wird.
- Unterprogramme liegen oft bereits als fertige Lösungen in einer Bibliothek vor oder können der Literatur entnommen werden.
- Unterprogramme verkürzen das Hauptprogramm da gleiche Programmteile nicht mehrmals wiederholt werden müssen.
- Unterprogramme sind vergleichbar mit Funktionen und Prozeduren in Programmier-Hochsprachen.

Unterprogramme werden mit dem "**rcall**"-Befehl aus einem Programm heraus aufgerufen! Im Unterprogramm selbst ist der einzige spezifische Befehl der "**ret**"-Befehl (Rücksprung), der bewirkt, dass der Programmablauf an der Stelle gleich hinter den Aufruf (**rcall**) weitergeführt wird. Dazu muss eine Rücksprungadresse auf dem Stapel (*stack*), einem reservierter Speicherbereich im SRAM, abgelegt werden. In einem späteren Kapitel wird im Modul B genauer auf diesen Stapel eingegangen.

rcall k

Relativer Aufruf eines Unterprogramms
(Adresskonstante k) (*relative call to subroutine*).

1	1	0	1	k	k	k	k	k	k	k	k	k	k	k	k

Relative Aufruf (max. 2 KiWorte vor und rückwärts). Die Rücksprungadresse wird auf dem Stapel abgelegt.
Beeinflusste Flags: keine Taktzyklen: 3

Neben dem relativen "**rcall**"-Befehl existiert wie bei den Sprüngen auch der direkte "**call**"-Befehl. Beim relativen "**rcall**"-Befehl kann maximal über 2Ki Worte vorwärts und rückwärts gesprungen werden. Der Assembler überwacht beim "**rcall**"-Befehl ob die Grenze überschritten

25 Ein Synonym für Unterprogramme ist die Bezeichnung **Routine**.

wird und meldet in diesem Fall einen Fehler. Soll im Programm weiter gesprungen werden, so wird der direkte "**call**"-Befehl verwendet.

Der relative Befehl ist schneller und Platz sparender als der direkte Befehl. Man soll also, falls möglich, den relativen Befehl verwenden.

ret

Rücksprung aus einem Unterprogramm (*return from subroutine*).

1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0

Die Rücksprungadresse wird vom Stapel genommen.
Beeinflusste Flags: keine Taktzyklen: 4

Der Vollständigkeit halber soll auch noch der indirekte "**icall**"-Befehl erwähnt werden, mit dem ein Unterprogramm indirekt aufgerufen werden kann (Adresse in **Z**).

Damit Unterprogramm-Aufrufe funktionieren können, muss der Stapel wie in der Assemblervorlage (Kapitel A2) initialisiert werden!

```
;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
out     SPH,Tmp1                ;datei festgelegt
ldi     Tmp1,LOW(RAMEND)
out     SPL,Tmp1
```

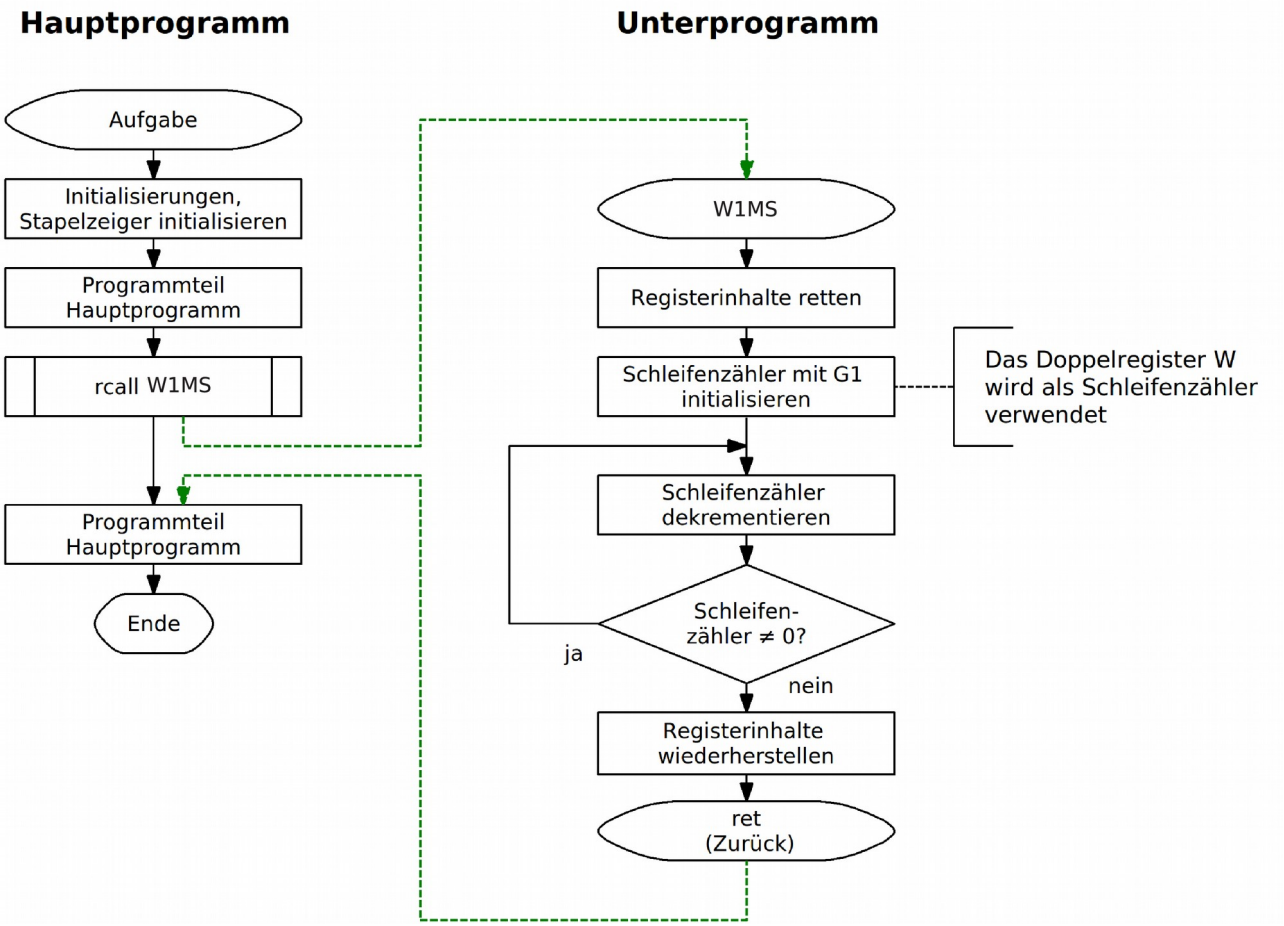
!Achtung!: Unterprogramme können nur verwendet werden, wenn der Stapel vorher initialisiert wurde!

Das Unterprogramm wird mit seinem Namen (symbolische Adresse bzw. Label) aufgerufen. Der Label soll zur besseren Unterscheidung zu den Befehlen mit großen Buchstaben geschrieben werden und sinnvoll sein, also einen Hinweis auf die Funktion des Unterprogramms liefern.

Beispiel:

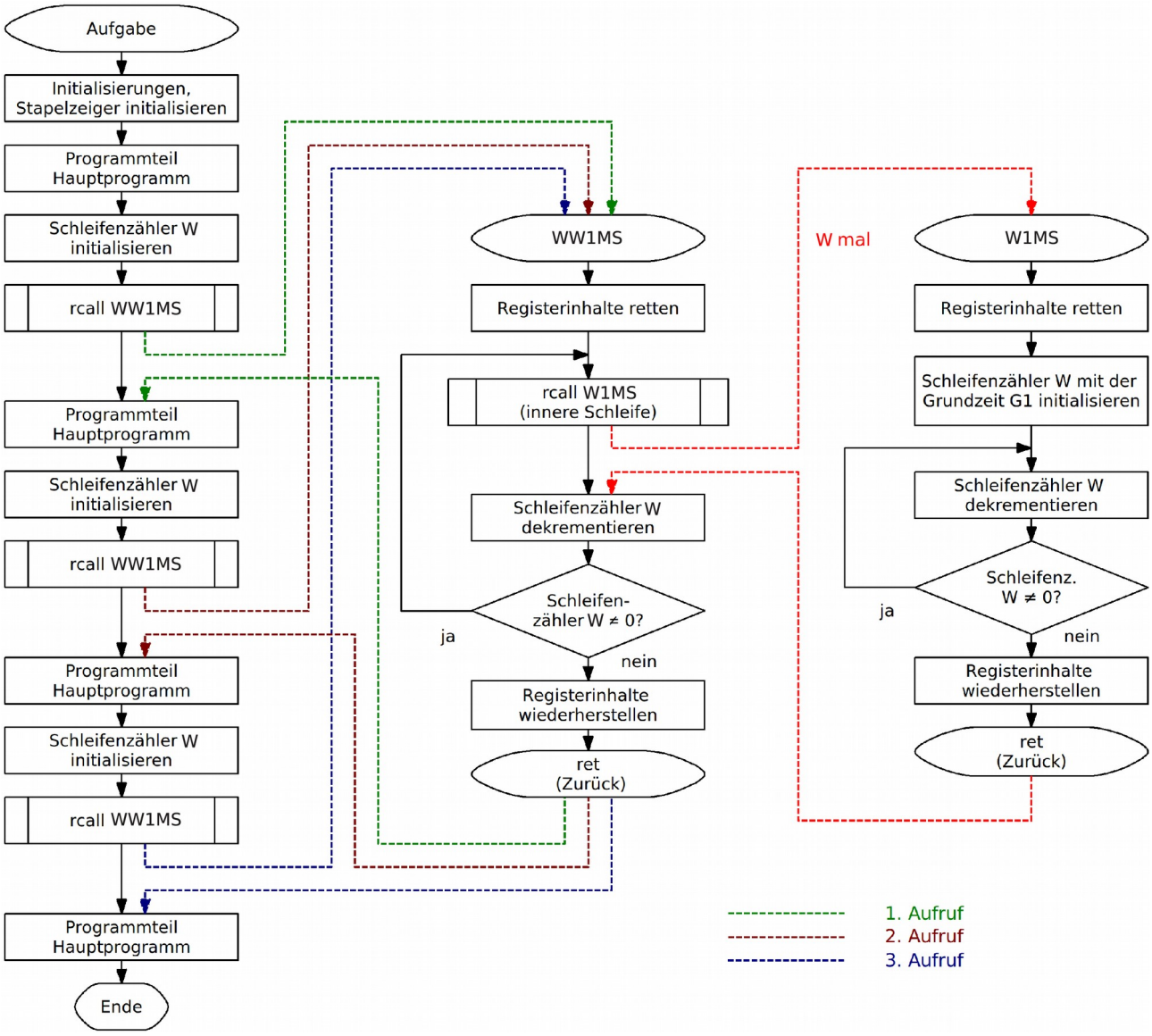
```
rcall   W1MS    ;rufe ein Zeitschleifenunterprogramm, das 1 MilliSekunde lang wartet
```

Das Verhalten eines Programms beim Aufruf von Unterprogrammen wird mit Hilfe folgender Flussdiagramme dargestellt (Ausnahmsweise wurden zum besseren Verständnis Befehle (**rcall** und **ret**) im Flussdiagramm eingetragen). Aus einem Hauptprogramm heraus wird eine Zeitschleife als Unterprogramm aufgerufen. Die gestrichelten grünen Linien deuten den Aufruf des Unterprogramms. Nach der Abarbeitung des Unterprogramms bewirkt der "**ret**"-Befehl, dass das Programm im Hauptprogramm gleich hinter dem "**rcall**"-Befehl weiterarbeitet.



Unterprogramme können auch aus Unterprogrammen heraus aufgerufen werden. Als Beispiel soll eine verschachtelte Zeitschleife dreimal aus dem Hauptprogramm heraus aufgerufen werden. Die innere Schleife der verschachtelten Zeitschleife wurde als eigenes Unterprogramm ausgekoppelt.

Bemerkung: Mit Sprungbefehlen wäre ein dreimaliger Aufruf eines Programmteils nicht zu bewerkstelligen, da man sonst mit einem Sprungbefehl drei unterschiedliche Labels anspringen müsste.



Globale Variablen

Zur Speicherung von Variablen werden meist die Arbeitsregister verwendet. Die Variablen sind dadurch im ganzen Programm, also auch in den Unterprogrammen sichtbar. Es kann von überall auf diese Variablen zugegriffen werden. Es ist also nicht unbedingt nötig den Unterprogrammen oder Interrupt-Routinen extra Parameter[26] zu übergeben. Das Unterprogramm muss nur wissen in welchem Arbeitsregister sich die Information befindet.

26 Parameter sind die Informationen (Daten) die zwischen Hauptprogramm und Unterprogramm ausgetauscht werden. Die Parameterübergabe kann natürlich auch statt über Register über den Speicher (Stapel, SRAM) geschehen, und das sogar auch indirekt indem nur die Adresse übermittelt wird.

Lokale Variablen

Unterprogramme benötigen zur Bewältigung ihrer Aufgabe oft interne Variablen (Register) die nur im Unterprogramm benötigt werden. Diese Variablen werden als lokale Variablen bezeichnet.

Da Unterprogramme flexibel eingesetzt werden, ist allerdings nicht immer bekannt, welche Register vom Hauptprogramm schon besetzt sind. Es ist deshalb wichtig, den Inhalt der Register, die zur Speicherung von lokalen Variablen benutzt werden sollen, zuerst in eine Speicherzelle zu retten und nach dem Abarbeiten des Unterprogramms wiederherzustellen. Es werden dazu Speicherzellen des Stapels verwendet. Das Retten und Wiederherstellen der Registerinhalte geschieht mit den beiden Befehlen "**push**" und "**pop**".

Retten und Wiederherstellen mit "push" und "pop"

Mit dem Befehle "**push r16**" wird der Inhalt vom Arbeitsregister **r16** auf dem Stapel abgelegt. Mit "**pop r16**" wird der Inhalt des Registers wiederhergestellt. Der Stapel arbeitet nach dem **LIFO-Prinzip** (**Last In First Out**), also wie ein richtiger Papierstapel. Das letzte Blatt das abgelegt wurde wird auch wieder als erstes entnommen.

push Rr

Kopiere Inhalt von Register Rr auf den Stapel (*push register on stack*).

1	0	0	1	0	0	1	r	r	r	r	r	1	1	1	1

Nach dem Kopieren wird der Stapelzeiger um 1 dekrementiert.
Beeinflusste Flags: keine Taktzyklen: 2

pop Rd

Kopiere Inhalt vom Stapel ins Register Rd (*pop register from stack*).

1	0	0	1	0	0	0	d	d	d	d	d	1	1	1	1

Vor dem Kopieren wird der Stapelzeiger um 1 inkrementiert.
Beeinflusste Flags: keine Taktzyklen: 2

Der Code für ein typisches Unterprogramm, das die Register **r16** bis **r18** als interne Variablen benutzt, könnte folgendermaßen aussehen:

```
;-------------------------------------------------------------------------
;       Unterprogramm mit drei internen Variablen
;-------------------------------------------------------------------------

SR_3WR: push    r16         ;Alle drei Variablen auf den Stapel retten
        push    r17
        push    r18
        ;
        ;Code des Unterprogramms
        ;
        pop     r18         ;Alle drei Variablen wiederherstellen
        pop     r17         ;Achtung !! Umgekehrte Reihenfolge !!
        pop     r16
        ret                 ;Ruecksprung ins Hauptprogramm
```

Bemerkungen: Innerhalb eines Unterprogramms sollen keine Definitionen (**.DEF**) oder Zuweisungen (**.EQU**) verwendet werden, da nicht bekannt ist welche Definitionen bzw. Zuweisungen das Hauptprogramm schon benutzt. Für interne Labels soll eine Erweiterung des Unterprogrammlabels zur Anwendung kommen um so doppelte Belegung von Labeln zu vermeiden (Bsp: Unterprogrammlabel: W1ms; interne Labels: W1ms_1, W1ms_2 ...).

Soll ein Unterprogramm zu jedem beliebigen Zeitpunkt aufgerufen werden können, so soll man auch das Statusregister **SREG** auf den Stapel retten. Dadurch werden im Hauptprogramm, durch den Aufruf des Unterprogramms, die Statusbits nicht verändert. Das Retten des SF-Register SREG muss über ein Arbeitsregister erfolgen!
Beispiel:

```
;-------------------------------------------------------------------------
;       Unterprogramm mit zwei internen Variablen, SREG wird auch gerettet
;-------------------------------------------------------------------------

SR_3WS: push    r16         ;r16 auf den Stapel retten
        in      r16,SREG    ;SREG in einen Zwischenspeicher laden
        push    r16         ;SREG auf den Stapel retten
        push    r17         ;r17 auf den Stapel retten
        ;
        ;Code des Unterprogramms
        ;
        pop     r17         ;r17 wiederherstellen ! Umgekehrte Reihenfolge !
        pop     r16         ;SREG wiederherstellen
        out     SREG,r16    ;
        pop     r16         ;r16 wiederherstellen
        ret                 ;Ruecksprung ins Hauptprogramm
```

Externe Unterprogramme Einbinden mit ".INCLUDE"

In Assembler gibt es keine so schöne kompakte Befehle wie in den Hochsprachen. Trotzdem wird Assembler-Programmieren fast genauso bequem, wenn man sich die entsprechenden Befehle mit Unterprogrammen nachbildet und diese in einer Bibliothek sammelt (oder bestehende Bibliotheken benutzt).

Zeitschleifen werden immer wieder benötigt, und im Folgen soll eine kleine Bibliothek mit unterschiedlichen Zeitschleifen erstellt werden. Die entsprechende Datei soll mit dem Namen "**SR_TIME_16M.asm**" versehen werden und einige Zeitschleifen-Unterprogramme enthalten die mit einem externen Quarz von 16 MHz funktionieren.

Um den Sachverhalt zu verdeutlichen soll im folgenden der Code für die Unterprogramme aus dem obigen Flussdiagramm dargestellt werden.

Zuerst die 16-Bit-Zeitschleife, hier mit einer Grundzeit von 1ms:

```
;*************************************************************************
;*
;*      Titel:  Bibliothek mit Zeitunterprogrammen (SR_TIME_16M.asm)
;*      Datum:  10/05/07        Version:        0.1
;*      Autor:
```

```
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32A       Quarzfrequenz:   extern 16MHz
;*
;*****************************************************************************

;-----------------------------------------------------------------------------
;       16-Bit-Zeitschleife wartet 1ms (Wait 1ms)
;-----------------------------------------------------------------------------
W1MS:   push    r24     ;rette verwendete Registerinhalte
        push    r25
        ldi     r24,0x9C ;Initialisiere den Schleifenzaehler W
        ldi     r25,0x0F ;G = t-tT/4tT = 1ms/4*62,5ns-1/4 = 4000-1/4
                         ;Korrektur: -15tT fuer rcall, 2 push, 2 pop, ret
                         ;1 Durchlauf der Schleife benoetigt 4tT
                         ;GKorrektur = G-1/4-15/4 = 4000-4 = 3996 = 0x0F9C
                         ;(Korrektur koennte man vernachlaessigen,Fehler 0,1%)
W1MSL:  sbiw    r24,1   ;Dekrementiere den Schleifenzaehler W
        brne    W1MSL   ;Verlasse die Schleife bei Schleifenzaehler W = 0
        pop     r25     ;Wiederherstellung der verwendeten Registerinhalte
        pop     r24
        ret             ;Ruecksprung ins Hauptprogramm
```

Dann die verschachtelte 16-Bit-Zeitschleife, bei der der Startwert des Schleifenzählers über das Doppelregister **X** übergeben wird:

```
;-----------------------------------------------------------------------------
;       Verschachtelte 16-Bit-Zeitschleife wartet W mal 1ms (Wait W*1ms)
;       Der Anfangswert von W (r25:r24) muss im Hauptprogramm gesetzt werden!
;       Der Fehler durch die zusaetzlichen Befehle kann hier vernachlässigt
;       werden (max Fehler bei einmaligem Aufruf < 0,12%).
;-----------------------------------------------------------------------------
WW1MS:  push    r24     ;rette verwendete Registerinhalte
        push    r25
WW1MSL: rcall   W1MS    ;Rufe innere Schleife (Unterprogramm W1ms)
        sbiw    r24,1   ;Dekrementiere den Schleifenzaehler W
        brne    WW1MSL  ;Verlasse die Schleife bei Schleifenzaehler W = 0
        pop     r25     ;Wiederherstellung der verwendeten Registerinhalte
        pop     r24
        ret             ;Ruecksprung ins Hauptprogramm
```

Diese Programmteile werden in der Datei "**SR_TIME_16M.asm**" abgespeichert. Um das ganze übersichtlicher zu gestalten soll unser Arbeitsverzeichnis ein Unterverzeichnis mit der Bezeichnung **lib** erhalten, in dem sich die erstellten Assemblerbibliotheken befinden. Eingebunden wird die Datei mit der "**.INCLUDE**"-Direktive (siehe Kapitel A2).

Ein einfaches Hauptprogramm bei dem eine LED im Sekundentakt aufleuchtet könnte dann folgendermaßen aussehen:

```
;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    Mask = r19              ;Register 19 dient zur Maskierung
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe
.DEF    W = r24

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
```

```
        ldi     Tmp1,LOW(RAMEND)        ;RAMEND (SRAM) ist in der Definitions-
        out     SPL,Tmp1                ;datei festgelegt
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1

        sbi     DDRD,0          ;nur PD0 als Ausgang initialisieren
        ldi     Mask,0x01       ;Setze Maske zur Invertierung (PD0)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   ldi     WL,LOW(500)     ;16-Bit-Schleifenzaehler mit
        ldi     WH,HIGH(500)    ;G2 = 500 t = 0,5s initialisieren
        rcall   WW1MS           ;Zeitschleife aufrufen
        ;Toggeln und Ausgabe
        in      Tmp1,PIND
        eor     Tmp1,Mask       ;Invertiere mit EXOR
        out     PORTD,Tmp1      ;LED (Pin PD0) wechselt Zustand
        rjmp    MAIN            ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
.INCLUDE "lib/SR_TIME_16M.asm"

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

1. Bemerkung: Ruft man die Zeitschleife **WW1MS** mit einem leeren **W**-Register (**W** = 0) auf, so wird diese nicht Null-Mal sondern **65536**-mal aufgerufen, da ja vor der Kontrolle dekrementiert wird. Dies kann unerwünschten Effekten führen, wenn zum Beispiel eine Midi-Datei (Musikdatei) abgespielt wird, wo die Notendauer auch Null sein kann! Es ist also sinnvoll das Unterprogramm um einige Zeilen zu erweitern:

```
WW1MS:  tst     r24      ;Falls W = 0 schnellstmoeglich zurueck
        brne    WW1MSP
        tst     r25
        brne    WW1MSP
        ret
WW1MSP: push    r24      ;rette verwendete Registerinhalte
        push    r25
WW1MSL: rcall   W1MS     ;Rufe innere Schleife (Unterprogramm W1MS)
        sbiw    r24,1    ;Dekrementiere den Schleifenzaehler W
        brne    WW1MSL   ;Verlasse die Schleife bei Schleifenzaehler W = 0
        pop     r25      ;Wiederherstellung der verwendeten Registerinhalte
        pop     r24
        ret              ;Ruecksprung ins Hauptprogramm
```

tst Rd

Teste ob das Register Rd Null oder Negativ ist (*test for zero or minus*).

0	0	1	0	0	0	d	d	d	d	d	d	d	d	d	d

Entspricht einer logischen Und-Verknüpfung des Registers mit sich selbst (AND Rd,Rd) .

Beeinflusste Flags: S, V (0), N, Z Taktzyklen: 1

2. Bemerkung: Es besteht auch die Möglichkeit eine vom Quarz unabhängige Zeitschleifenbibliothek zu programmieren. Dabei lässt man den Assembler den Anfangswert der Zeitschleifen beim Übersetzen errechnen. Im Hauptprogramm definiert man zum Beispiel die Konstante **Freq** und weist ihr die Quarz- oder Oszillatorfrequenz zu (Bsp.: **.EQU Freq = 8000000**) mit der dann in der Bibliothek umgerechnet wird. Zum Beispiel für die Zeitschleife mit einer Millisekunde:

```
        ldi     XL,LOW(Freq/4000-4)
        ldi     XH,HIGH(Freq/4000-4)
```

Leider sind die Rechenmöglichkeiten des Assembler recht eingeschränkt, so dass die Formel ein wenig umgestellt werden musste (Division vor Subtraktion). Ausgehend von der präzisen Formel und einer Korrektur von 15 Taktzyklen (**push**, **call**, ...) ergibt sich:

$$G = \frac{t - t_T}{4t_T} = \frac{t}{4t_T} - \frac{1}{4} \qquad \text{-Korrektur} \frac{15t_T}{4t_T \text{ pro Durchlauf}}:$$

$$G = \frac{t}{4t_T} - \frac{16}{4} = \frac{t}{4t_T} - 4 = \frac{t}{4\left(\frac{1}{Freq}\right)} = \frac{Freq \cdot t}{4} - 4$$

$$G = \frac{Freq \cdot 1\text{ms}}{4} - 4 = \frac{Freq \cdot \frac{1}{1000\text{s}}}{4} - 4 = \frac{Freq}{4000} - 4 \qquad \text{(Freq in Hertz!)}$$

⌂ **A600**

a) Ergänze die Zeitschleifen-Bibliothek "**SR_TIME_16M.asm**" um eine präzise Zeitschleife mit 1 Mikrosekunde (**W1US**; nutze dazu den "**NOP**"-Befehl).
b) Ergänze die Zeitschleifen-Bibliothek um folgende Zeitschleifen mit fester Zeit: **W10US**, **W100US**, **W10MS**, **W100MS**, **W500MS**, **W1S**, **W5S**, **W10S**, **W1MIN**, **W10MIN**.
c) Ergänze die Zeitschleifen-Bibliothek zusätzlich um folgende Zeitschleifen mit veränderbarer Zeit über das W-Register: **WW100US**, **WW10MS**, **WW100MS**, **WW1S**, **WW1MIN**.
d) Teste alle Zeitschleifen. Gib dazu ein Rechtecksignal an **PA0** aus und kontrolliere die Frequenz mit einem Zähler. Nenne das neue Programm "**A600_test_time_lib.asm**".

⌂ **A601**

Ändere das Programm "**A508_pulse_delay_1.asm**" aus dem vorigen Kapitel so um, dass es mit der erstellten Bibliothek arbeitet. Nenne das neue Programm "**A601_pulse_delay_2.asm**".

⌂ **A602**

Zeichne ein Flussdiagramm und schreibe ein Programm für einen 8-Bit-Vorwärtszähler, dessen Schrittweite an vier Schaltern (Port C, **PC0** bis **PC3**) während des Zählens eingestellt werden kann und dessen Stand auf acht Leuchtdioden (Port D) ausgegeben wird. Der Zähler soll mit einer Geschwindigkeit von einem Schritt pro Sekunde zählen. Der Anfangswert des Zählers beträgt **0**. Die Schalterabfrage und die Ausgabe an die LEDs soll als Unterprogramm realisiert werden.
Nenne das Programm "**A602_counter_1.asm**".

⌂ **A603** Zeichne ein Flussdiagramm und schreibe ein Programm das die Quersumme einer über acht Schalter (Port D) eingelesene Hex-Zahl bildet. Vor dem Rücksprung zum erneuten Einlesen einer Hex-Zahl, wird die Quersumme für drei Sekunden auf fünf Leuchtdioden (Port C, **PC0**-**PC4**) ausgegeben.

Beispiel: eingelesene Zahl: **0x87** = **1000 0111**
Quersumme: **0x08** + **0x07** = **0x0F**

Die Bildung der Quersumme geschieht in einem Unterprogramm. Die Arbeitsregister **r24** und **r25** dienen der Parameterübergabe. Das Unterprogramm befindet sich in der gleichen Datei wie das Hauptprogramm. Nenne das Programm "**A603_checksum.asm**".

⌂ **A604** Ergänze das Programm "**A602_counter_1.asm**" um die Fähigkeit vor- und rückwärts zu zählen. Die Umschaltung erfolgt durch einen zusätzlichen Schalter (**PC4**). Rückwärts wird bei betätigtem Schalter gezählt. Zusätzlich soll dieses neue Programm nur gerade Zahlen zum Zählen benutzen. Nenne das Programm "**A604_counter_2.asm**".