Anfängerkurs zum Erlernen der Assemblersprache von ATMEL-AVR-Mikroprozessoren

Gerhard Schmidt
http://www.avr-asm-tutorial.net
November 2022

Historie:

Hinzugefügt: Weitere zwei Beispiele mit Festkommazahlen und Hinweis auf die Besonderheit der Z-Flagge bei SBC/SBCI/CPCI (November 2022)

Hinzugefügte Kapitel zu Fließkommazahlen und Speicherzugriffen vom September 2021

Ergänzte Fassung vom September 2010

Vollständig überarbeitete Neufassung Dezember 2008 / Januar 2009

Originalversion vom September 2003

1. Inhaltsverzeichnis

1 Warum Assembler lernen?

Assembler oder Hochsprache, das ist hier die Frage. Warum soll man noch eine neue Sprache lernen, wenn man schon welche kann? Das beste Argument: Wer in Frankreich lebt und nur Englisch kann, kann sich zwar durchschlagen, aber so richtig heimisch und unkompliziert ist das Leben dann nicht. Mit verquasten Sprachkonstruktionen kann man sich zwar durchschlagen, aber elegant hört sich das meistens nicht an. Und wenn es schnell gehen muss, geht es eben öfter schief.

1.1 In der Kürze liegt die Würze

Assemblerinstruktionen übersetzen sich 1 zu 1 in Maschineninstruktionen. Auf diese Weise macht der Prozessor wirklich nur das, was für den angepeilten Zweck tatsächlich erforderlich ist und was der Programmierer auch gerade will. Keine extra Schleifen und nicht benötigten Features stören und blasen den ausgeführten Code auf. Wenn es bei begrenztem Programmspeicher und komplexerem Programm auf jedes Byte ankommt, dann ist Assembler sowieso Pflicht. Kürzere Programme lassen sich wegen schlankerem Maschinencode leichter entwanzen, weil jeder einzelne Schritt Sinn macht und zu Aufmerksamkeit zwingt.

1.2 Schnell wie Hund

Da kein unnötiger Code ausgeführt wird, sind Assembler-Programme maximal schnell. Jeder Schritt ist von voraussehbarer Dauer. Bei zeitkritischen Anwendungen, wie z.B. bei Zeitmessungen ohne Hardware-Timer, die bis an die Grenzen der Leistungsfähigkeit des Prozessors gehen sollen, ist Assembler ebenfalls zwingend. Soll es gemütlich zugehen, können Sie programmieren wie Sie wollen.

1.3 Assembler ist leicht erlernbar

Es stimmt nicht, dass Assembler komplizierter und schwerer erlernbar ist als Hochsprachen. Das Erlernen einer einzigen Assemblersprache macht Sie mit den wichtigsten Grundkonzepten vertraut, das Erlernen von anderen Assembler-Dialekten ist dann ein Leichtes. Der erste Code sieht nicht sehr elegant aus, mit jedem Hunderter an Quellcode sieht das schon schöner aus. Schönheitspreise kriegt man erst ab einigen Tausend Zeilen Quellcode. Da viele Features prozessorabhängig sind, ist Optimierung eine reine Übungsangelegenheit und nur von der Vertrautheit mit der Hardware und dem Dialekt abhängig. Die ersten Schritte fallen in jeder neu erlernten Sprache nicht leicht und nach wenigen Wochen lächelt man über die Holprigkeit und Umständlichkeit seiner ersten Gehversuche. Manche Assembler-Instruktionen lernt man eben erst nach Monaten richtig nutzen.

1.4 AVR sind ideal zum Lernen

Assemblerprogramme sind gnadenlos, weil sie davon ausgehen, dass der Programmierer jeden Schritt mit Absicht so und nicht anders macht. Alle Schutzmechanismen muss man sich selber ausdenken und auch programmieren, die Maschine macht bedenkenlos jeden Unsinn mit. Kein Fensterchen warnt vor ominösen Schutzverletzungen, es sei denn man hat das Fenster selber programmiert. Denkfehler beim Konstruieren sind aber genauso schwer aufzudecken wie bei Hochsprachen. Das Ausprobieren ist bei den ATMEL-AVR aber sehr leicht, da der Code rasch um einige wenige Diagnostikzeilen ergänzt und mal eben in den Chip programmiert werden kann. Vorbei die Zeiten mit EPROM löschen, programmieren, einsetzen, versagen und wieder von vorne nachdenken. Änderungen sind schnell gemacht, kompiliert und entweder im Studio simuliert, auf dem STK-Board ausprobiert oder in der realen Schaltung einprogrammiert, ohne dass sich ein IC-Fuß verbogen oder die UV-Lampe gerade im letzten Moment vor der großen Erleuchtung den Geist aufgegeben hat.

1.5 Ausprobieren

Nur Mut bei den ersten Schritten. Wenn Sie schon eine Programmiersprache können, vergessen Sie sie erst mal gründlich, weil sonst die allerersten Schritte schwerfallen. Hinter jeder Assemblersprache steckt auch ein Prozessorkonzept, und große Teile der erlernten Hochsprachenkonzepte machen in Assembler sowieso keinen Sinn. Die ersten fünf Instruktionen gehen schwer, dann geht es exponentiell leichter. Nach den ersten 10 Zeilen nehmen Sie den ausgedruckten Instruction Set Summary mal für eine Stunde mit in die Badewanne und wundern sich ein wenig, was es so alles zu Programmieren und zum Merken gibt. Versuchen Sie zu Anfang keine Mega-Maschine zu programmieren, das geht in jeder Sprache gründlich schief. Heben Sie erfolgreich programmierte Codezeilen gut dokumentiert auf, Sie brauchen sie sowieso bald wieder.

1.6 Aufbau der Darstellung

Das vorliegende Werk muss man nicht unbedingt von vorne nach hinten und komplett lesen, einige Kapitel können auch schlicht als Nachschlagewerk dienen. Konsultieren Sie ab und an das Inhaltsverzeichnis auf Kapitel, die Sie noch nicht absolviert haben und die Sie vielleicht zur Lösung einer bestimmten Aufgabe brauchen könnten. Bedingt durch diesen Aufbau sind leider manche Doppelungen nicht zu vermeiden. Und es ist auch nicht zu vermeiden, dass in manchen Kapiteln schon Dinge verwendet werden, die erst später ausführlicher behandelt werden.

Im Gegensatz zu einem speziellen, sehr teuren Buch verzichte ich darauf, den kompletten Befehlssatz in tabellarischer Form abzuhandeln. Mir kommt es mehr auf den Kontext an, in dem Instruktionen einen Sinn machen. Wer den Befehlssatz ganz braucht, sollte sich das Device Databook seines Lieblingsprozessors daneben legen.

Viel Lernerfolg.

2 Das Konzept hinter der Sprache Assembler

Achtung! Bei dieser Seite geht es um die Programmierung von Mikrocontrollern, nicht um PCs mit Linux- oder Windows-Betriebssystem und ähnliche Elefanten, sondern um kleine Mäuse. Es geht auch nicht um die Programmierung von Ethernet-Megamaschinen, sondern um die Frage, warum man bei der Programmierung von Mikrocontrollern als Anfänger eher mit Assembler beginnen sollte als mit einer Hochsprache. Sie erläutert, was das Konzept hinter Assembler ist, was Hochsprachenprogrammierer vergessen müssen, um Assembler zu lernen und warum Assembler manchmal fälschlich als "Maschinensprache" bezeichnet wird.

2.1 Die Hardware von Mikrocontrollern

Was hat die Hardware von Mikrocontrollern mit Assembler zu tun? Viel, wie aus dem Folgenden hervorgeht. Das Konzept bei Assembler ist, die Ressourcen des Prozessors unmittelbar zugänglich zu machen. Unter Ressourcen sind dabei alle Hardwarebestandteile zu verstehen, also

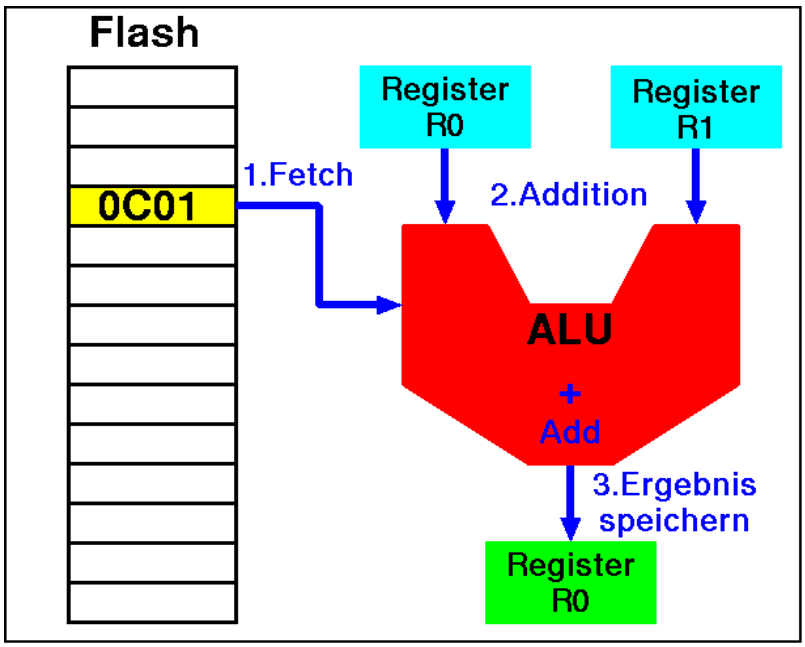
- die zentrale Steuer- und Recheneinheit (CPU) und deren Rechenknecht, die Arithmetische und Logik-Einheit (ALU),
- die diversen Speichereinheiten (interne oder externe RAM, EEPROM-Speicher),
- die Ports, die das Verhalten von Portbits ebenso beeinflussen wie auch Zeitgeber (Timer), AD-Wandler und andere Geräte.

2.2 Die Arbeitsweise der CPU

Am Wichtigsten ist die Fähigkeit der zentralen Steuereinheit, Instruktionen aus dem Programmspeicher (Flash) zu holen (Instruction fetch), in auszuführende Schritte zu zerlegen und diese Schritte dann auszuführen. Die Instruktionen stehen dabei in den AVR als 16-bittige Zahlenwerte im Flash-Speicher und werden von dort schrittweise abgeholt. Der Zahlenwert übersetzt sich dann z. B. in die Addition der beiden Register R0 und R1 und das Speichern des Ergebnisses im Register R0. Register sind dabei Speicherzellen, die 8 Bits enthalten und direkt gelesen und beschrieben werden können. An einigen Beispielen soll das genauer gezeigt werden.

CPU-Operation	Kode (binär)	Kode(Hex)
CPU schlafen legen	1001.0101.1000.1000	9588
Register R1 zu Register R0 addieren	0000.1100.0000.0001	0C01
Register R1 von Register R0 subtrahieren	0001.1000.0000.0001	1801
Konstante 170 in Register R16 schreiben	1110.1010.0000.1010	EA0A
Multipliziere Register R3 mit Register R2, Ergebnis in R1 und R0	1001.1100.0011.0010	9C32

Wenn die CPU also hexadezimal 9588 aus dem Flashspeicher liest, stellt sie ihre Tätigkeit ein und holt keine weiteren Instuktionen mehr aus dem Speicher. Keine Angst, da ist noch ein weiterer Schutzmechanismus davorgeschaltet, bevor sie das tatsächlich ausführt. Und man kann sie auch wieder aus diesem Zustand aufwecken. Liest sie hexadezimal 0C01, addiert sie R0 und R1 und schreibt das Ergebnis in das Register R0. Das läuft dann etwa so ab:



Im ersten Schritt wird das Instruktionswort geholt und in Ausführungsschritte der Arithmetic Logical Unit (ALU) zerlegt.

Im zweiten Schritt werden die beiden ausgewählten Register an die ALU-Eingänge angelegt und mit den Inhalten die Operation Addition ausgeführt.

Im dritten Schritt wird das Zielregister R0 an den Ergebnisausgang der ALU angeschlossen und das Resultat in das Register geschrieben.

Liest die Ausführungsmaschine 9C23 aus dem Flash, dann multipliziert sie die beiden Register R3 und R2 und schreibt das Ergebnis nach R1 (oberes Byte) und R0 (unteres Byte). Hat die ALU gar keine Hardware zum Multiplizieren (z. B. in einem ATtiny13, dann bewirkt 9C23 rein gar nix. Im Prinzip kann die CPU also 65.536 verschiedene Operationen ausführen. Weil aber zum Beispiel nicht nur 170 in das Register R16 geladen werden können soll, sondern jede beliebige Konstante zwischen 0 und 255 in jedes Register zwischen R16 und R31, verbraucht alleine diese Ladeinstruktion 256*16 = 4.096 der 65.536 theoretisch möglichen Instruktionen. Die Direktladeinstruktionen für die Konstante c in die Register R16..R31 (r4:r3:r2:r1:r0, r4 ist dabei immer 1) bauen sich dabei folgendermaßen auf:

Bit	***15***	***14***	***13***	***12***	***11***	***10***	***09***	***08***	***07***	***06***	***05***	***04***	***03***	***02***	***01***	***00***
Inhalt	1	1	1	0	c7	c6	c5	c4	r3	r2	r1	r0	c3	c2	c1	c0

Warum diese Bits so gemischt platziert sind, bleibt das Geheimnis von ATMEL. Die Addition und die Subtraktion verbraten je 32*32 = 1.024 Kombinationen und sind mit den Zielregistern R0..R31 (z4..z0) und den Quellregistern R0..R31 (q4..q0) so aufgebaut:

Bit	***15***	***14***	***13***	***12***	***11***	***10***	***09***	***08***	***07***	***06***	***05***	***04***	***03***	***02***	***01***	***00***
Inhalt Addieren	0	0	0	0	1	1	q4	z4	z3	z2	z1	z0	q3	q2	q1	q0
Inhalt Subtrahieren	0	0	0	1	1	0	q4	z4	z3	z2	z1	z0	q3	q2	q1	q0

Auch hier kümmern wir uns nicht weiter um die Platzierung, weil wir uns die auch nicht weiter merken müs-

sen. Es geht hier nur darum zu verstehen, wie die CPU in ihrem Innersten tickt.

2.3 Instruktionen in Assembler

Damit sich der Programmierer keine 16-bittigen Zahlen und deren verrückte Platzierung merken muss, sind in Assembler dafür verständliche Abkürzungen an deren Stelle gesetzt, sogenannte Mnemonics (frei übersetzt Gedächtnisstützen). So lautet dann die Entsprechung der Instruktion, die den Schlafmodus bewirkt, einfach "SLEEP". Liest der Assembler "SLEEP", macht er daraus hex 9588. Das SLEEP kann sich im Gegensatz zu 9588 hexidezimal auch jemand wie ich merken, der schon bei der eigenen Telefonnummer Schwierigkeiten hat.

Addieren heißt einfach "ADD". Um die beiden Register auch noch gleich in einer leicht lesbaren Form mit anzugeben, lautet die gesamte Instruktion "ADD R0,R1". Der gesamte Ausdruck übersetzt sich in ein einziges binäres 16-Bit-Wort, nämlich 0C01. Die Übersetzung erledigt der Assembler.

Die Formulierung übersetzt der Assembler in exakt das 16-Bit-Wort, das dann im Flash-Speicher steht, von der CPU dort abgeholt, in ihre interne Ablauffolge umgewandelt und ausgeführt wird. Für jede Instruktion, die die CPU versteht, gibt es ein entsprechendes Mnemonic. Umgekehrt: es gibt keine Mnemonics, denen nicht exakt ein bestimmter Ablauf einer CPU-Instruktion entspricht. Die Fähigkeit der CPU bestimmt bei Assembler also den Sprachumfang. Die "Sprache" der CPU selbst ist also bestimmend, die Mnemonics sind bloß repräsentative Statthalter für die Fähigkeiten der CPU selbst.

2.4 Unterschied zu Hochsprachen

Hier ist für Hochsprachler noch ein Hinweis nötig.

In Hochsprachen sind die Sprachkonstrukte weder von der Hardware noch von den Fähigkeiten der CPU abhängig. Die Sprachkonstrukte laufen auf einer Vielzahl von Prozessoren, sofern es für die Sprache einen Compiler gibt, der die erdachten Schlüsselwörter in CPU-Operationen übersetzen kann. Ein GOTO in Basic mag so ähnlich aussehen wie ein JMP in Assembler, dahinter steckt aber ein ganz anderes Konzept.

Die Sache mit der Übertragbarkeit auf andere Prozessoren funktioniert natürlich nur, solange der Controller und seine CPU das mitmacht. Letztlich bestimmt dann die Hochsprache, welcher Prozessor geht und welcher nicht. Ein ärmlicher Kompromiss, weil er unter Umständen ganze Welten mit ihren vielfältigen Möglichkeiten als ungeeignet ausschließt.

Deutlich wird das bei der oben gezeigten Instruktion "MUL". In Assembler kann die CPU das Multiplizieren entweder selbst erledigen oder der Assemblerprogrammierer muss sich selbst eine Multiplikationsroutine ausdenken, die das erledigt. In einem für einen ATtiny13 geschriebenen Assembler-Quellcode MUL zu verwenden, gibt einfach nur eine Fehlermeldung (Illegal instruction). In einer Hochsprache wird der Compiler bei einer Multiplikation feststellen, ob eine Multiplikationseinheit vorhanden ist, dann erzeugt er den Code entsprechend. Ist keine da, fügt er entsprechenden Code ein, ohne dass der Programmierer das bemerkt. Und vielleicht auch noch gleich eine Integer-, Longword- und andere Multiplikationsroutinen, ein ganzes Paket eben.

Und schon reicht das Flash eines Tiny nicht mehr aus und es muss unbedingt ein Mega mit Dutzenden unbeschalteter Pins sein damit der Elefant nicht benutzter Routinen reinpasst.

2.5 Assembler und Maschinensprache

Assembler wird wegen seiner Nähe zur Hardware auch oft als Maschinensprache bezeichnet. Das ist nicht ganz exakt, weil die CPU selbst nur 16-bittige Worte, aber nicht die Zeichenfolge "ADD R0,R1" versteht.

Der Ausdruck "Maschinensprache" ist daher eher so zu verstehen, dass der Sprachumfang exakt dem der Maschine entspricht und der Assembler nur menschenverständliche Kürzel in maschinenverständliche Binärworte verwandelt.

2.6 Interpreterkonzept und Assembler

Bei einem Interpreter übersetzt die CPU selbst den menschenverständlichen Code erst in ihr verständliche Binärworte. Der Interpreter würde die Zeichenfolge "A = A + B" einlesen (neun Zeichen), würde die Leerzeichen darin herausschälen, die Parameter A und B identifizieren, würde das "+" in eine Addition übersetzen und letztlich vielleicht das Gleiche ausführen wie die Assemblerformulierung weiter oben.

Im Unterschied zum Assembler oben, bei der die CPU nach dem Assemblieren sofort ihr Lieblingsfutter, 16-bittige Instruktionsworte, vorgeworfen bekommt, ist die CPU beim Interpreterkonzept überwiegend erst mit Übersetzungsarbeiten beschäftigt. Da zum Übersetzen vielleicht 20 oder 200 CPU-Schritte erforderlich sind, bevor die eigentliche Arbeit geleistet werden kann, ist ihre Ausführungsgeschwindigkeit nur als lahm zu bezeichnen.

Was bei schnellen Prozessoren noch verschmerzt werden kann, ist bei zeitkritischen Abläufen prohibitiv: niemand weiss, was die CPU wann macht und wie lange sie dafür tatsächlich braucht. Die Vereinfachung, sich nicht mit irgendwelchen Ausführungszeiten befassen zu müssen, bewirkt, dass der Programmierer sich damit gar nicht befassen kann, weil ihm die nötigen Informationen darüber auch noch vorenthalten werden.

2.7 Hochsprachenkonzepte und Assembler

Hochsprachen ziehen zwischen die CPU und den Quelltext noch weitere undurchaubare Ebenen ein. Ein durchgängiges Konzept sind z. B. sogenannte Variablen. Das sind Speicherplätze, die für eine Zahl, einen Text oder auch nur einen einzigen Wahrheitswert vorbereitet werden und im Quelltext der Hochsprache für den betreffenden Speicherplatz stehen.

Vergessen Sie für das Erlernen von Assembler zu allererst das ganze Variablenkonzept, es führt Sie nur an der Nase herum und hält Sie davon ab, das Konzept zu durchblicken. Assembler kennt nur Bits und Bytes, Variablen sind ihm völlig fremd.

Hochsprachen fordern zum Beispiel, dass Variablen vor ihrer Verwendung zu deklarieren sind, also z. B. als Byte (8-Bit) oder Double-Word (16-Bit-Wort). Compiler platzieren solche Variablen dann je nach Geschmack irgendwo in den Speicherraum oder in die im AVR reichlich vorhandenen 32 Register. Ob er den Speicherort, wie der Assemblerprogrammierer, mit Übersicht und Nachdenken entscheidet, ist von den Kosten für den Compiler abhängig. Der Programmierer kann nur noch versuchen herauszufinden, wo die Variable denn letztlich steckt. Die Verfügungsgewalt darüber hat er dem Compiler überlassen.

Die Instruktion "A = A + B" ist jetzt auch noch typgeprüft: ist A ein Zeichen (Char) und B eine Zahl (1), dann wird die Formulierung so nicht akzeptiert, weil man Zeichencodes angeblich nicht zum Addieren benutzen kann. Der Schutzeffekt dieses Verbots ist ungefähr Null, aber Hochsprachenprogrammierer glauben dass sie dadurch vor Unsinn bewahrt werden. In Assembler hindert nichts und niemand daran, zu einem Byte z. B. sieben dazu zu zählen oder 48 abzuziehen, egal ob es sich bei dem Inhalt des Bytes um ein Zeichen oder eine Zahl handelt. Was in dem Register R0 drin steht, entscheidet der Programmierer, nicht der Compiler. Ob eine Operation mit dem Inhalt Sinn macht, entscheidet der Programmierer und nicht der Compiler. Ob vier Register einen 32-Bit-Wert repräsentieren oder vier ASCII-Zeichen, ob die in auf- oder absteigender Reihenfolge oder völlig durcheinander liegen, kann sich der Assemblerprogrammierer auch aussuchen, der Hochsprachenprogrammierer nicht. Typen gibt es in Assembler nicht, alles besteht aus beliebig vielen Bits und Bytes an beliebigen Orten im verfügbaren weitläufigen Speicherraum.

Von ähnlichem Effekt sind auch die anderen Regeln, denen sich der Programmierer in Hochsprachen zu unterwerfen hat. Angeblich ist es sicherer und übersichtlicher, alles in Unterprogrammen zu programmieren, außer nach festgelegten Regeln nicht im Programmablauf herum zu springen und immer Werte als Parameter zu übergeben und Resultate auch so entgegen zu nehmen. Vergessen Sie einstweilen die meisten dieser Regeln, Assembler braucht auch welche, aber ganz andere, die Sie sich noch dazu alle selbst ausdenken müssen.

Hochsprachenprogrammierer haben noch ein Konzept verinnerlicht, das beim Erlernen von Assembler nur hinderlich ist: die Trennung in diverse Ebenen, in Hardware, Treiber und andere Interfaces. In Assembler ist das alles nicht getrennt, eine Trennung macht auch keinen sonderlichen Sinn, weil Sie die Trennung dauernd wieder durchlöchern müssen, um Ihr Problem optimal lösen zu können. Da viele Hochsprachenregeln in Mikrocontrollern keinen Sinn machen und das alles so puristisch auch gar nicht geht, erfindet der Hochsprachenprogrammierer gerne allerlei Möglichkeiten, um bei Bedarf diese Restriktionen zu umgehen. In Assembler stellen sich diese Fragen erst gar nicht. Alles ist im direkten Zugriff, jede Speicherzelle verfügbar, nix ist gegen Zugriff abgeschirmt, alles kann beliebig verstellt und kaputt gemacht werden. Die Verantwortung dafür bleibt alleine beim Programmierer, der seinen Grips anstrengen muss, um drohende Konflikte bei Zugriffen zu vermeiden.

Dem fehlenden Schutz steht die totale Freiheit gegenüber. Lösen Sie sich einstweilen von den Fesseln, Sie werden sich andere sinnvolle Fesseln angewöhnen, die Sie vor Unheil bewahren.

2.8 Was ist nun genau warum einfacher?

Alles was der Assembler-Programmierer an Worten und Begriffen braucht, steht im Datenblatt des Prozessors: die Instruktions- und die Porttabelle. Fertig. Mit diesen Begrifflichkeiten kann er alles erschlagen. Keine andere Dokumentation ist vonnöten.

Wie der Timer gestartet wird (ist "Timer.Start(8)" nun wirklich leichter zu verstehen als "LDI R16,0x02 und OUT TCCR0,R16"), wie er angehalten wird (CLR R16 und OUT TCCR0,R16), wie er auf Null gesetzt wird (CLR R16 und OUT TCNT0,R16), steht nicht in der Dokumentation irgendeiner Hochsprache mit irgendeiner mehr oder weniger eingängiger Spezialbegrifflichkeit, sondern im Device Databook allein. Ob der Timer 8- oder 16-bittig ist, entscheidet der Programmierer in Assembler selbst, bei Basic der Compiler und bei C vielleicht auch der Programmierer, wenn er genügend mutig ist.

Was ist dann an der Hochsprache leichter, wenn man statt "A = A * B" schreiben muss "MUL R16,R17"? Kaum etwas. Außer A und B sind nicht als Byte definiert. Dann muss der Assemblerprogrammierer überlegen, weil sein Hardware-MUL nur Bytes multipliziert. Wie man 24-bittige Zahlen mit 16-bittigen Zahlen mit und ohne Hardware-Multiplikator multipliziert, kann man lernen (siehe späteres Kapitel) oder aus dem Internet abkupfern. Jedenfalls kein Grund, in eine Hochsprache mit ihren Bibliotheken zu flüchten, nur weil man zu faul zum Lernen ist.

Mit Assembler lernt der Programmierer die Hardware kennen, weil er ohne die Ersatzkrücke Compiler auskommen muss, die ihm zwar alles abnimmt, aber damit auch alles aus der Hand nimmt. Als Belohnung für seine intensive Beschäftigung mit der Hardware hat er die auch beliebig im Griff und kann, wenn er will, auch mit einer Baudrate von 45,45 bpm über die serielle Schnittstelle senden und empfangen. Eine Geschwindigkeit, die kein Windowsrechner zulässt, weil die Krücke Treiber halt nun mal nur ganzzahlige Vielfache von 75 kennt, weil frühere Fernschreiber mit einem Getriebe das auch schon hatten und flugs von 75 auf 300 umgeschaltet werden konnten.

Wer Assembler kann, hat ein Gefühl dafür, was der Prozessor kann. Wer dann bei komplexen Aufgabenstellungen zu einer Hochsprache übergeht, hat die Entscheidung rationaler gefällt als wenn er mangels Können erst gar nix anderes kennt und sich mit Work-Arounds für nicht in der Hochsprache implementierte Features die Nacht um die Ohren schlagen muss. Der Assemblerprogrammierer hat zwar viel mehr selbst

zu machen, weil ihm die ganzen famosen Bibliotheken fehlen, kann aber über so viel abseitiges Herumgeeiere nur lächeln, weil ihm die ganze Welt des Prozessors unterwürfig zur Verfügung steht.

3 Hardware für die AVR-Assembler-Programmierung

Damit es beim Lernen von Assembler nicht zu trocken zugeht, braucht es etwas Hardware zum Ausprobieren. Gerade wenn man die ersten Schritte macht, muss der Lernerfolg schnell sichtbar sein.

Hier werden mit wenigen einfachen Schaltungen im Eigenbau die ersten Hardware-Grundlagen beschrieben. Um es vorweg zu nehmen: es gibt von der Hardware her nichts einfacheres als einen AVR mit den eigenen Ideen zu bestücken. Dafür wird ein Programmiergerät beschrieben, das einfacher und billiger nicht sein kann. Wer dann Größeres vorhat, kann die einfache Schaltung stückweise erweitern.

Wer sich mit Löten nicht herumschlagen will und nicht jeden Euro umdrehen muss, kann ein fertiges Programmier-Board erstehen. Die Eigenschaften solcher Boards werden hier ebenfalls beschrieben.

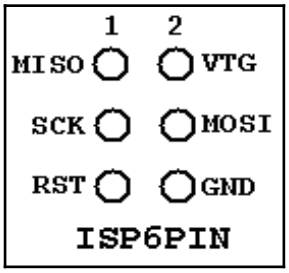
3.1 Das ISP-Interface der AVR-Prozessoren

Bevor es ins Praktische geht, zunächst ein paar grundlegende Informationen zum Programmieren der Prozessoren. Nein, man braucht keine drei verschiedenen Spannungen, um das Flash-EEPROM eines AVR zu beschreiben und zu lesen. Nein, man braucht keinen weiteren Mikroprozessor, um ein Programmiergerät für einfache Zwecke zu bauen. Nein, man braucht keine 10 I/O-Ports, um so einem Chip zu sagen, was man von ihm will. Nein, man muss den Chip nicht aus der Schaltung auslöten, in eine andere Fassung stecken, ihn dann dort programmieren und alles wieder rückwärts. Geht alles viel einfacher.

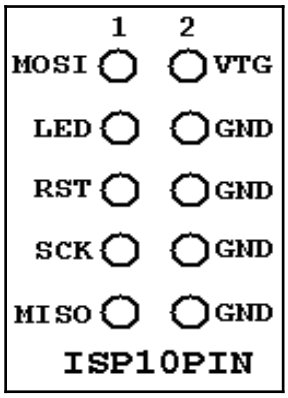
Für all das sorgt ein in allen Chips eingebautes Interface, über das der Inhalt des Flash-Programmspeichers sowie des eingebauten EEPROM's beschrieben und gelesen werden kann. Das Interface arbeitet seriell und braucht genau drei Leitungen:

- SCK: Ein Taktsignal, das die zu schreibenden Bits in ein Schieberegister im AVR eintaktet und zu lesende Bits aus einem weiteren Schieberegister austaktet,
- MOSI: Das Datensignal, das die einzutaktenden Bits vorgibt,
- MISO: Das Datensignal, das die auszutaktenden Bits zum Lesen durch die Programmiersoftware ausgibt.

Damit die drei Pins nicht nur zum Programmieren genutzt werden können, wechseln sie nur dann in den Programmiermodus, wenn das RESET-Signal am AVR (auch: RST oder Restart genannt) auf logisch Null liegt. Ist das nicht der Fall, können die drei Pins als beliebige I/O-Signalleitungen dienen. Wer die drei Pins mit dieser Doppelbedeutung benutzen möchte und das Programmieren des AVR in der Schaltung selbst vornehmen möchte, muss z.B. einen Multiplexer verwenden oder Schaltung und Programmieranschluss durch Widerstände voneinander entkoppeln. Was nötig ist, richtet sich nach dem, was die wilden Programmierimpulse mit dem Rest der Schaltung anstellen können.



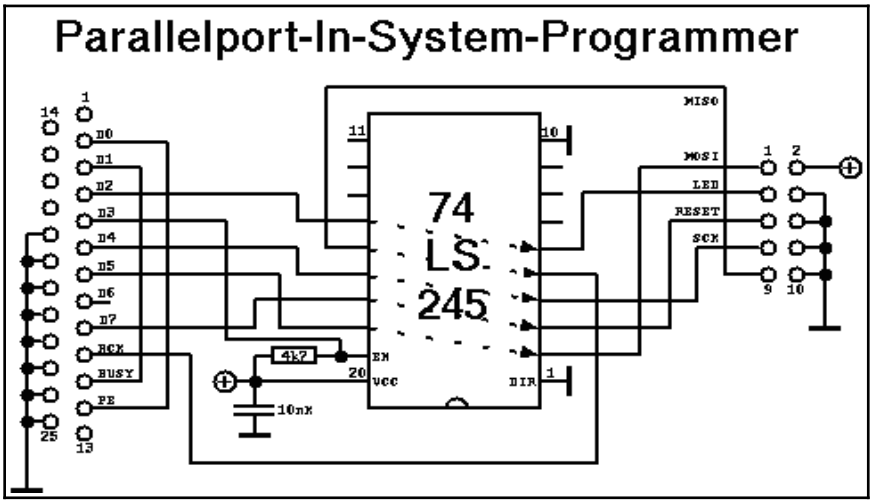
Nicht notwendig, aber bequem ist es, die Versorgungsspannung von Schaltung und Programmier-Interface gemeinsam zu beziehen und dafür zwei weitere Leitungen vorzusehen. GND versteht sich von selbst, VTG bedeutet Voltage Target und ist die Betriebsspannung des Zielsystems. Damit wären wir bei der 6-Draht-ISP-Programmierleitung. Die ISP6-Verbinder haben die nebenstehende, von ATMEL standardisierte Pinbelegung.

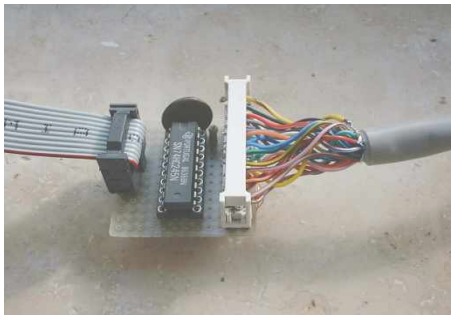


Und wie das so ist mit Standards: immer gab es schon welche, die früher da waren, die alle verwenden und an die sich immer noch (fast) alle halten. Hier ist das der 10-polige ISP-Steckverbinder. Er hat noch zusätzlich einen LED-Anschluss, über den die Programmiersoftware mitteilen kann, dass sie fertig mit dem Programmieren ist. Auch nicht schlecht, mit einer roten LED über einen Widerstand gegen die Versorgungsspannung ein deutliches Zeichen dafür zu setzen, dass die Programmiersoftware ihren Dienst versieht.

3.2 Programmierer für den PC-Parallel-Port

So, Lötkolben anwerfen und ein Programmiergerät bauen. Es ist denkbar einfach und dürfte mit Standardteilen aus der gut sortierten Bastelkiste schnell aufgebaut sein.





Ja, das ist alles, was es zum Programmieren braucht. Den 25-poligen Stecker steckt man in den Parallelport des PC's, den 10-poligen ISP-Stecker an die AVR-Experimentierschaltung. Wer gerade keinen 74LS245 zur Hand hat, kann auch einen 74HC245 verwenden. Allerdings sollten dann die unbenutzten Eingänge an Pin 11, 12 und 13 einem definierten Pegel zugeführt werden, damit sie nicht herumklappern, unnütz Strom verbraten und HF erzeugen.

Die Schaltung ist fliegend aufgebaut. Das 10-adrige Flachbandkabel führt zu einem 10-Pin-ISP-Stecker, das Rundkabel zum Printerport.

Den Rest erledigt die alte ISP-Software von ATMEL, PonyProg2000 oder andere Software. Allerdings ist bei der Brennsoftware auf die Unterstützung neuerer AVR-Typen zu achten.

Wer eine serielle Schnittstelle hat (oder einen USB-Seriell-Wandler) kann sich zum Programmieren aus dem Studio oder anderer Brenner-Software auch den kleinen, süßen AVR910-Programmer bauen (Bauanleitung und Schaltbild siehe die Webseite von Klaus Leidinger:

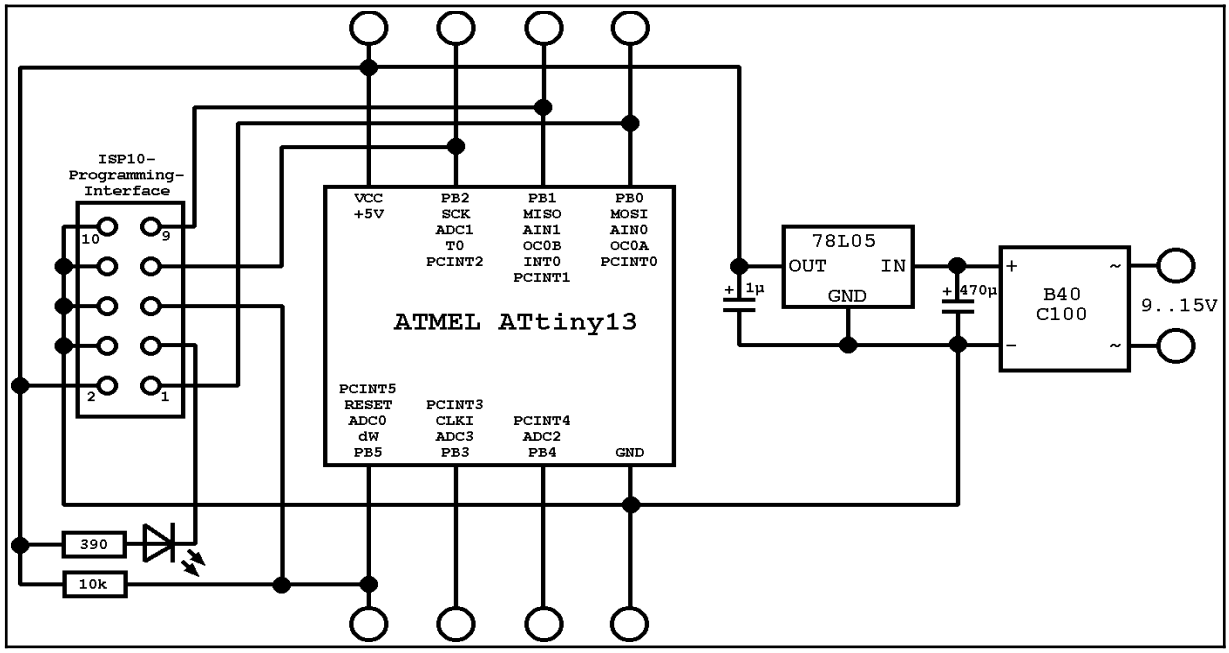
http://www.klaus-leidinger.de/mp/Mikrocontroller/AVR-Prog/AVR-Programmer.html

3.3 Experimentierschaltung mit ATtiny13

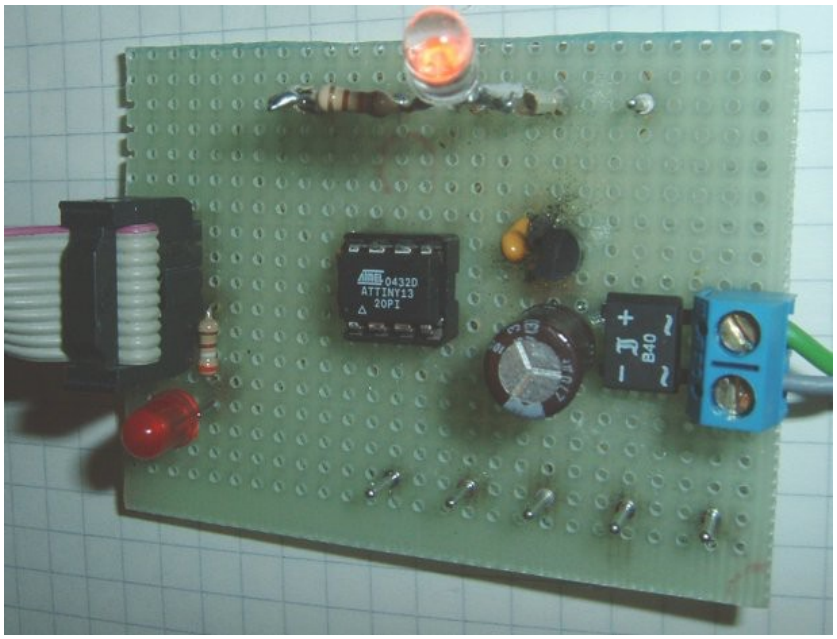
Dies ist ein sehr kleines Experimentierboard, das Tests mit den vielseitigen Innereien des ATtiny13 ermöglicht.

Das Bild zeigt

- das ISP10-Programmier-Interface auf der linken Seite, mit einer Programmier-LED, die über einen Widerstand von 390 Ω an die Betriebsspannung angeschlossen ist,den ATtiny13, dessen Reset-Eingang an Pin 1 mit einem Widerstand von 10 kΩ an die Betriebsspannung führt,
- den Stromversorgungsteil mit einem Brückengleichrichter, der mit 9..15 V aus einem ungeregelten Netzteil oder einem Trafo gespeist werden kann, und einem kleinen 5 V-Spannungsregler.



Der ATtiny13 braucht keinen externen Quarz oder Taktgenerator, weil er einen internen 9,6 MHz-RC-Oszillator hat und von der Werksausstattung her mit einem Vorteiler von 8 bei 1,2 MHz arbeitet.

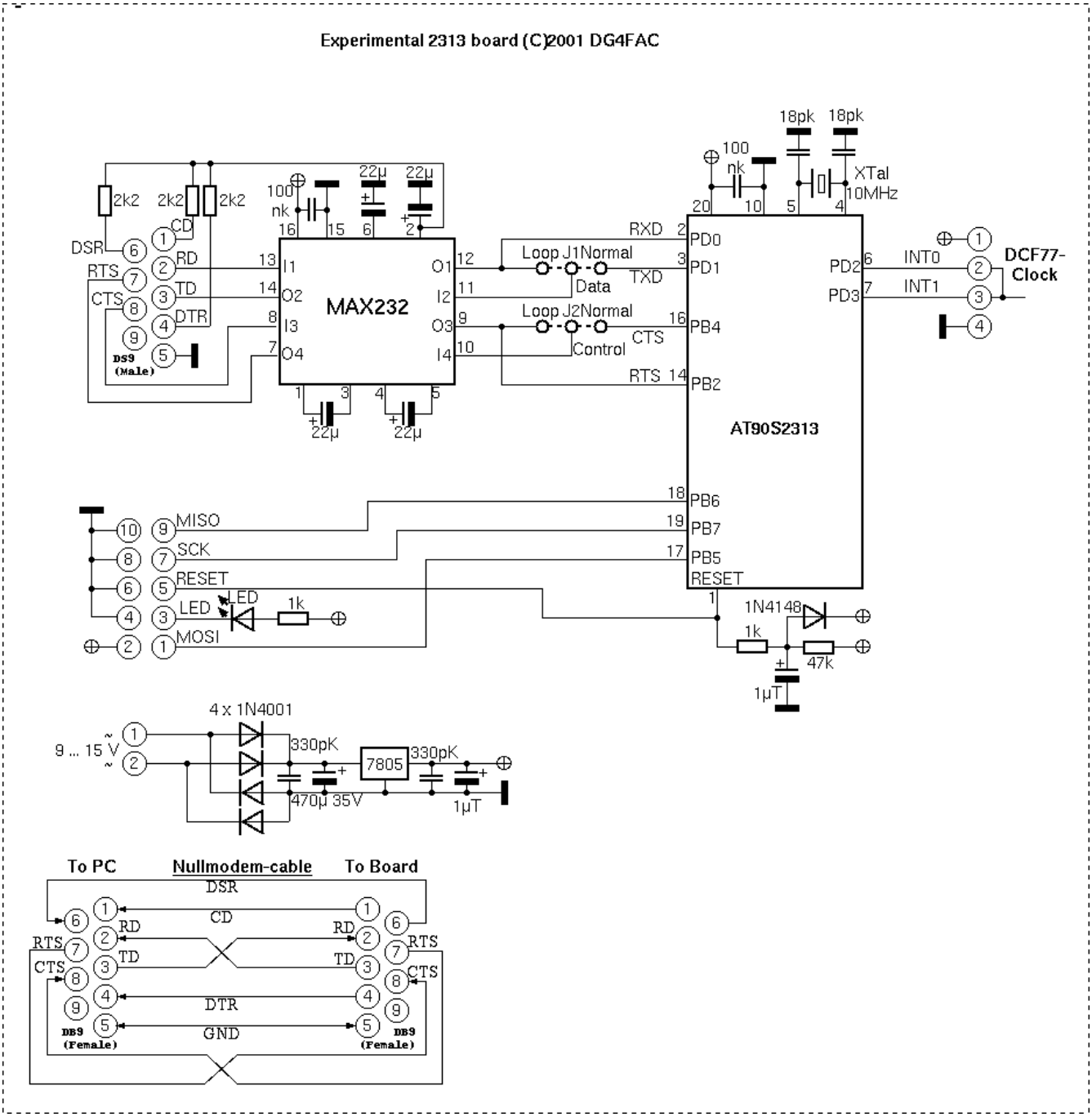


Die Hardware kann auf einem kleinen Experimentierboard aufgebaut werden, wie auf dem Bild zu sehen. Alle Pins des ATtiny13 sind hier über Lötnägel zugänglich und können mit einfachen Steckverbindern mit externer Hardware verbunden werden (im Bild zum Beispiel eine LED mit Vorwiderstand).

Das Board erlaubt das einfache Testen aller Hardware-Komponenten wie z. B. der Ports, Timer, AD-Wandler.

3.4 Experimentalschaltung mit AT90S2313/ATtiny2313

Damit es noch mehr zum Programmieren gibt, hier eine einfache Schaltung mit einem schon etwas größeren AVR-Typ. Verwendbar ist der veraltete AT90S2313 oder sein neuerer Ersatztyp ATtiny2313.



Im Bild ist der Prozessorteil mit dem ISP10-Anschluss rechts zu sehen, die restliche Hardware wird über die Steckpins angeschlossen.

Die Schaltung hat ein kleines geregeltes Netzteil für den Trafoanschluss (für künftige Experimente mit einem 1A-Regler ausgestattet), einen Quarz-Taktgenerator (hier mit einem 10 MHz-Quarz, es gehen aber auch langsamere), die Teile für einen sicheren Reset beim Einschalten, das ISP-Programmier-Interface (hier mit einem ISP10PIN-Anschluss).

Damit kann man im Prinzip loslegen und an die vielen freien I/O-Pins des 2313 jede Menge Peripherie dranstricken.

Das einfachste Ausgabegerät dürfte für den Anfang eine LED sein, die über einen Widerstand gegen die Versorgungsspannung geschaltet wird und die man an einem Portbit zum Blinken animieren kann.

3.5 Fertige Programmierboards für die AVR-Familie

Wer nicht selber löten will oder gerade einige Euros übrig hat und nicht weiß, was er damit anstellen soll, kauft sich ein fertiges Programmierboard.

STK500

Leicht erhältlich ist das STK500 von ATMEL. Es bietet u.a.:

- Sockel für die Programmierung der meisten AVR-Typen,
- serielle und parallele Low- und High-Voltage Programmierung,
- ISP6PIN- und ISP10PIN-Anschluss für externe Programmierung,
- programmierbare Oszillatorfrequenz und Versorgungsspannungen,
- steckbare Tasten und LEDs,
- einen steckbaren RS232C-Anschluss (UART),
- ein serielles Flash-EEPROM,
- Zugang zu allen Ports über 10-polige Pfostenstecker.

Die Experimente können mit dem mitgelieferten AT90S8515 oder ATmega8515 sofort beginnen. Das Board wird über eine serielle Schnittstelle (COMx) an den Rechner gekoppelt und von den neueren Versionen des Studio's von ATMEL bedient. Für die Programmierung externer Schaltungen besitzt das Board einen ISP6-Anschluss. Damit dürften alle Hardware-Bedürfnisse für den Anfang abgedeckt sein.

Für den Anschluss an eine USB-Schnittstelle am PC braucht man noch einen handelsüblichen USB-Seriell-Wandler. Für eine gute automatische Erkennung durch das Studio ist im Gerätemanager eine der Schnittstellen COM2 bis COM9 für den Wandler und eine Geschwindigkeit von 115 kBaud einzustellen.

AVR Dragon

Wer an seinem PC oder Notebook keine RS232-Schnittstelle mehr hat, ist mit dem preiswerten AVR Dragon gut bedient. An das kleine schmucke Board kann man eine ISP6- oder ISP10-Schnittstelle anbringen und damit externe Hardware programmieren. Das Board hat auch Schnittstellen und Support für die Hochspannungsprogrammierung.

Andere Boards

Es gibt eine Vielzahl an Boards. Eine Einführung in alle verfügbare Boards kann hier nicht gegeben werden. Wichtige Kriterien für die Auswahl eines Boards sind:

- Lassen sich neben dem On-Board-Prozessor auch externe Prozessoren programmieren (z. B. über eine ISP6- oder ISP10-Schnittstelle)?

- Wenn ja, wie erfolgt langfristig der Support (Update) für neuere AVR-Typen?
- Sind ausreichend Schnittstellen vorhanden, an denen bei Bedarf externe Komponenten angeschlossen werden können?
- Ist der On-Board-Prozessor ein Typ, für den es weiterhin Support von ATMEL gibt?
- Ist für die Programmierung proprietäre Herstellersoftware nötig?

3.6 Wie plane ich ein Projekt in Assembler?

Hier wird erklärt, wie man ein einfaches Projekt plant, das in Assembler programmiert werden soll. Weil die verwendeten Hardwarekomponenten eines Prozessors sehr vieles vorweg bestimmen, was und wie die Hard- und Software aufgebaut werden muss, zunächst die Überlegungen zur Hardware.

3.6.1 Überlegungen zur Hardware

In die Entscheidung, welchen AVR-Typ man verwendet, gehen eine Vielzahl an Anforderungen ein. Hier einige häufiger vorkommende Forderungen zur Auswahl:

1. Welche festen Portanschlüsse werden gebraucht? Feste Portanschlüsse sind Ein- oder Ausgänge von internen Komponenten, die an ganz bestimmten Anschlüssen liegen müssen und nicht frei wählbar sind. Sie werden zuerst zugeordnet. Komponenten und Anschlüsse dieser Art sind:
    a) Soll der Prozessor in der Schaltung programmiert werden können (ISP-Interface), dann werden die Anschlüsse SCK, MOSI und MISO diesem Zweck fest zugeordnet. Bei entsprechender Hardwaregestaltung können diese als Eingänge (SCK, MOSI) oder als Ausgang doppelt verwendet werden (Entkopplung über Widerstände oder Multiplexer empfohlen).
    b) Wird eine serielle Schnittstelle benötigt, sind RXD und TXD dafür zu reservieren. Soll zusätzlich das RTS/CTS-Hardware-Protokoll implementiert werden, sind zusätzlich zwei weitere Portbits dafür zu reservieren, die aber frei platziert werden können.
    c) Soll der Analogkomparator verwendet werden, dann sind AIN0 und AIN1 dafür zu reservieren.
    d) Sollen externe Signale auf Flanken überwacht werden, dann sind INT0 bzw. INT1 dafür zu reservieren.
    e) Sollen AD-Wandler verwendet werden, müssen entsprechend der Anzahl benötigter Kanäle die Eingänge dafür vorgesehen werden. Verfügt der AD-Wandler über die externen Anschlüsse AVCC und AREF, sollten sie entsprechend extern beschaltet werden.
    f) Sollen externe Impulse gezählt werden, sind dafür die Timer-Input-Anschlüsse T0, T1 bzw. T2 zu reservieren. Soll die Dauer externer Impulse exakt gemessen werden, ist dafür der ICP-Eingang zu verwenden. Sollen Timer-Ausgangsimpulse mit definierter Pulsdauer ausgegeben werden, sind die entsprechenden OCx-Ausgänge dafür zu reservieren.
    g) Wird externer SRAM-Speicher benötigt, müssen alle nötigen Address- und Datenports sowie ALE, RD und WR dafür reserviert werden.
    h) Soll der Takt des Prozessors aus einem externen Oszillatorsignal bezogen werden, ist XTAL1 dafür zu reservieren. Soll ein externer Quarz den Takt bestimmen, sind die Anschlüsse XTAL1 und XTAL2 dafür zu verwenden.
2. Welche zusammenhängenden Portanschlüsse werden gebraucht? Zusammenhängende Portanschlüsse sind solche, bei denen zwei oder mehr Bits in einer bestimmten Reihenfolge angeordnet sein sollten, um die Software zu vereinfachen.
    a) Erfordert die Ansteuerung eines externen Gerätes das Schreiben oder Lesen von mehr als einem Bit gleichzeitig, z.B. eine vier- oder achtbittige LCD-Anzeige, sollten die nötigen Portbits in dieser Reihenfolge platziert werden. Ist das z.B. bei achtbittigen Interfaces nicht

möglich, können auch zwei vierbittige Interfaces vorgesehen werden. Die Software wird vereinfacht, wenn diese 4-Bit-Interfaces links- bzw. rechtsbündig im Port angeordnet sind.

b) Werden zwei oder mehr ADC-Kanäle benötigt, sollten diese in einer direkten Abfolge (z.B. ADC2/ADC3/ADC4) platziert werden, um die Ansteuerungssoftware zu vereinfachen.

3. Welche frei platzierbaren Portbits werden noch gebraucht? Jetzt wird alles zugeordnet, was keinen bestimmten Platz braucht.
4. Wenn es jetzt wegen eines einzigen Portbits eng wird, kann der RESET-Pin bei einigen Typen als Eingang verwendet werden, indem die entsprechende Fuse gesetzt wird. Da der Chip anschließend nur noch über Hochvolt-Programmierung zugänglich ist, ist dies für fertig getestete Software akzeptabel. Für Prototypen in der Testphase ist für ISP ein Hochvolt-Programmier-Interface am umdefinierten RESET-Pin und eine Schutzschaltung aus Widerstand und Zenerdiode gegenüber der Signalquelle vonnöten (beim HV-Programmieren treten +12 Volt am RESET-Pin auf).

In die Entscheidung, welcher Prozessortyp für die Aufgabe geeignet ist, gehen ferner noch ein:

- Wieviele und welche Timer werden benötigt?
- Welche Werte sind beim Abschalten des Prozessors zu erhalten (EEPROM dafür vorsehen)?
- Wieviel Speicher wird im laufenden Betrieb benötigt (entsprechend SRAM dafür vorsehen)?
- Platzbedarf? Bei manchen Projekten mag der Platzbedarf des Prozessors ein wichtiges Entscheidungskriterium sein.
- Strombedarf? Bei Batterie-/Akku-betriebenen Projekten sollte der Strombedarf ein wichtiges Auswahlkriterium sein.
- Preis? Spielt nur bei Großprojekten eine wichtige Rolle. Ansonsten sind Preisrelationen recht kurzlebig und nicht unbedingt von der Prozessorausstattung abhängig.
- Verfügbarkeit? Wer heute noch ein Projekt mit dem AT90S1200 startet, hat ihn vielleicht als Restposten aus der Grabbelecke billig gekriegt. Nachhaltig ist so eine Entscheidung nicht. Es macht wesentlich mehr Mühe, so ein Projekt auf einen Tiny- oder Mega-Typen zu portieren als der Preisvorteil heute wert ist. Das "Portieren" endet daher meist mit einer kompletten Neuentwicklung, die dann auch schöner aussieht, besser funktioniert und mit einem Bruchteil des Codes auskommt.

Mit dem Simulator avr_sim (siehe http://www.avr-asm-tutorial.net/avr_sim/index_de.html) kann man sich das Auswählen des Prozessors ganz komfortabel erleichtern.

3.6.2 Überlegungen zum Timing

Geht ein Projekt darüber hinaus, ein Portbit abzufragen und daraus abgeleitet irgendwas anderes zu veranlassen, dann sind Überlegungen zum Timing des Projektes zwingend. Timing

- beginnt mit der Wahl der Taktfrequenz des Prozessors,
- geht weiter mit der Frage, was wie häufig und mit welcher Präzision vom Prozesser erledigt werden muss,
- über die Frage, welche Timing-Steuerungsmöglichkeiten bestehen, bis zu
- wie diese kombiniert werden können.

Wahl der Taktfrequenz des Prozessors

Die oberste Grundfrage ist die nach der nötigen Präzision des Taktgebers.

Reicht es in der Anwendung, wenn die Zeiten im Prozentbereich ungenau sind, dann ist der interne RC-Oszillator in vielen AVR-Typen völlig ausreichend. Bei den neueren ATtiny und ATmega hat sich die selbsttätige Oszillator-Kalibration eingebürgert, so dass die Abweichungen vom Nominalwert des RC-Oszillators nicht mehr so arg ins Gewicht fallen.

Wenn allerdings die Betriebsspannung stark schwankt, kann der Fehler zu groß sein. Wem der interne RC-Takt zu langsam oder zu schnell ist, kann bei einigen neueren Typen (z.B. dem ATtiny13) einen Vorteiler bemühen. Der Vorteiler geht beim Anlegen der Betriebsspannung auf einen voreingestellten Wert (z.B. auf 8) und teilt den internen RC-Oszillator entsprechend vor. Entweder per Fuse-Einstellung oder auch per Software-Einstellung kann der Vorteiler auf niedrigere Teilerverhältnisse (höhere Taktfrequenz) oder auf einen höheren Wert umgestellt werden (z.B. 128), um durch niedrigere Taktfrequenz z.B. verstärkt Strom zu sparen.

Obacht bei V-Typen! Wer dem Prozessor einen zu hohen Takt zumutet, hat es nicht anders verdient als dass der Prozessor nicht tut, was er soll. Wem der interne RC-Oszillator zu ungenau ist, kann per Fuse eine externe RC-Kombination, einen externen Oszillator, einen Quarz oder einen Keramikschwinger auswählen. Gegen falsch gesetzte Fuses ist kein Kraut gewachsen, es muss dann schon ein Board mit eigenem Oszillator sein, um doch noch was zu retten.

Die Höhe der Taktfrequenz sollte der Aufgabe angemessen sein. Dazu kann grob die Wiederholfrequenz der Tätigkeiten dienen. Wenn eine Taste z.B. alle 2 ms abgefragt werden soll und nach 20 Mal als entprellt ausgeführt werden soll, dann ist bei einer Taktfrequenz von 1 MHz für 2000 Takte zwischen zwei Abfragen Platz, 20.000 Takte für die wiederholte Ausführung des Tastenbefehls. Weit ausreichend für eine gemütliche Abfrage und Ausführung.

Eng wird es, wenn eine Pulsweitenmodulation mit hoher Auflösung und hoher PWM-Taktfrequenz erreicht werden soll. Bei einer PWM-Taktfrequenz von 10 kHz und 8 Bit Auflösung sind 2,56 MHz schon zu langsam für eine software-gesteuerte Lösung. Wenn das ein Timer mit Interrupt-Steuerung übernimmt, um so besser.

4 Werkzeuge für die AVR-Assembler-Programmierung

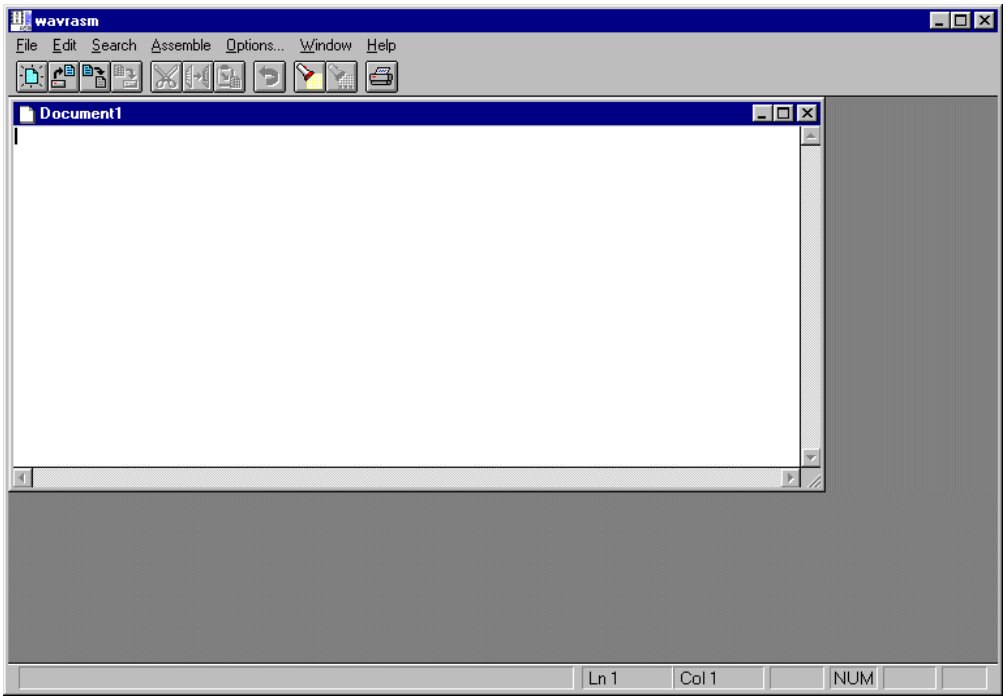
In diesem Abschnitt werden die Werkzeuge vorgestellt, die zum Assembler-Programmieren nötig sind. Dabei werden zunächst Werkzeuge besprochen, die jeden Arbeitsschritt separat erledigen, damit die zugrunde liegenden Schritte einzeln nachvollzogen werden können. Erst danach wird eine integrierte Werkzeugumgebung vorgestellt.

Für die Programmierung werden vier Teilschritte und Werkzeuge benötigt. Im einzelnen handelt es sich um

1. den Editor,
2. den Assembler,
3. das Programmier-Interface, und
4. den Simulator.

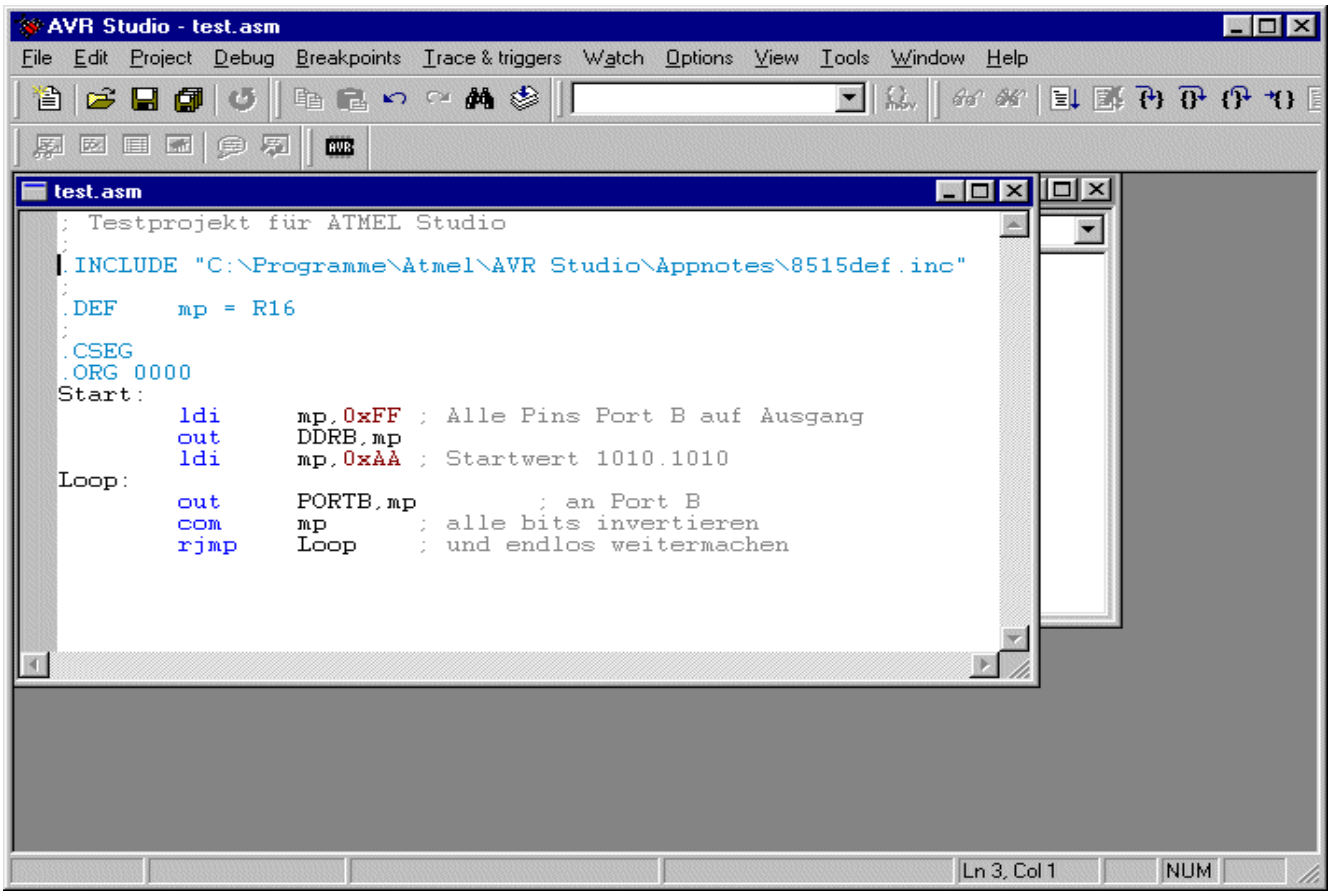
4.1 Der Editor

Assemblerprogramme schreibt man mit einem Editor. Der braucht im Prinzip nicht mehr können als ASCII-Zeichen zu schreiben und zu speichern. Im Prinzip täte es ein sehr einfaches Schreibgerät. Wir zeigen hier ein etwas veraltetes Gerät, den WAVRASM von ATMEL. Der WAVRASM sieht nach der Installation und nach dem Start eines neuen Projektes etwa so aus:



Im Editor schreiben wir einfach die Assemblerinstruktionen und Instruktionszeilen drauf los, gespickt mit Kommentaren, die alle mit einem Semikolon beginnen. Das sollte dann etwa so aussehen.

Nun wird das Assembler-Programm mit dem File-Menue in irgendein Verzeichnis, am besten in ein eigens dazu errichtetes, abgespeichert. Fertig ist der Assembler-Quellcode.
Manche Editoren erkennen Instruktionen und Symbole und färben diese Worte entsprechend ein, so dass man Tippfehler schnell erkennt und ausbessern kann (Syntax-Highlighting genannt). In einem solchen Editor sieht unser Programm wie im Bild aus.

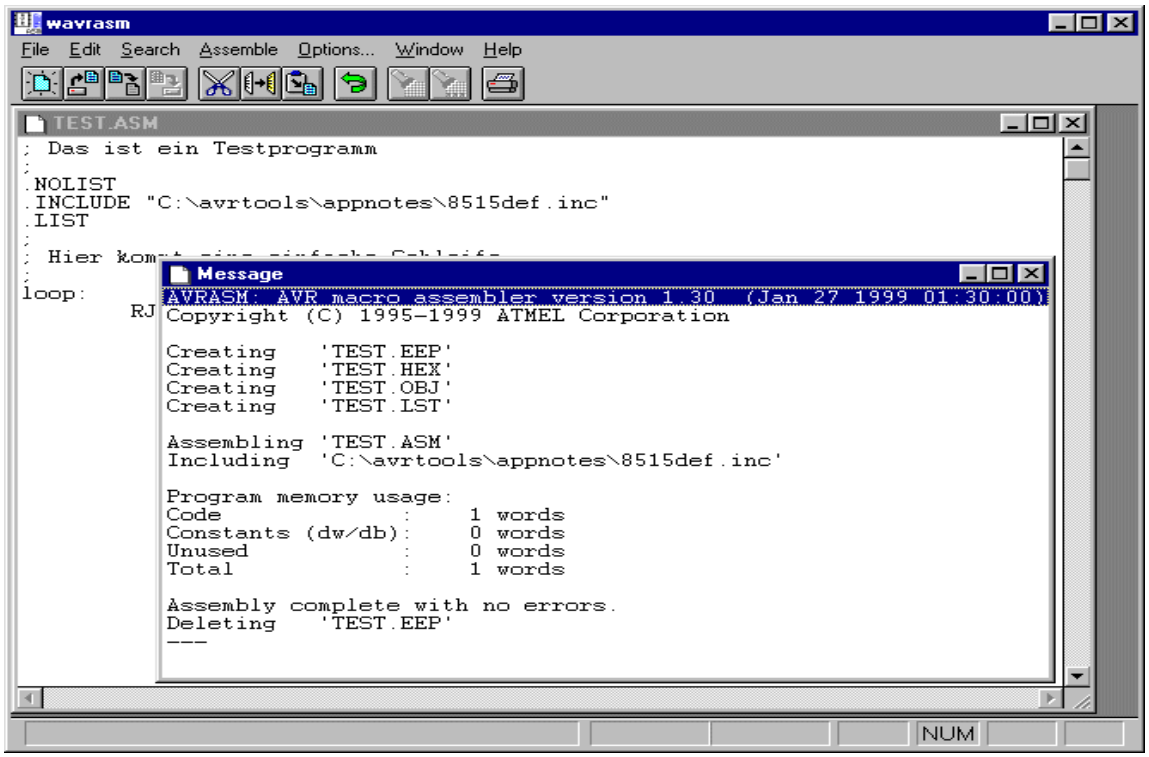


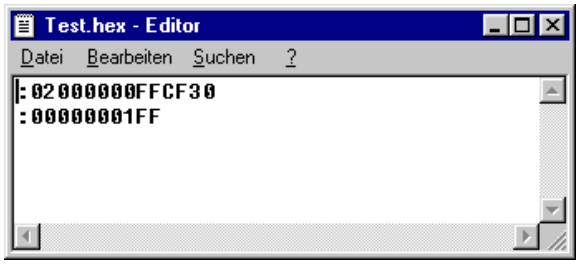
Auch wenn die im Editor eingegebenen Worte noch ein wenig kryptisch aussehen, sie sind von Menschen lesbar und für den eigentlichen Mikroprozessor noch völlig unbrauchbar.

4.2 Der Assembler

Nun muss das ganze Programm von der Textform in die Maschinen-sprachliche Form gebracht werden. Den Vorgang heißt man Assemblieren, was in etwa "Zusammenbauen", "Auftürmen" oder auch "zusammen schrauben" bedeutet. Das erledigt ein Programm, das auch so heißt: Assembler. Derer gibt es sehr viele. Für AVR heißen die zum Beispiel "AvrAssembler" oder "AvrAssembler2" (von ATMEL, Bestandteil der Studio-Entwicklungsumgebung), "TAvrAsm" (Tom's AVR Assembler) oder mein eigener "GAvrAsm" (Gerd's AVR Assembler). Welchen man nimmt, ist weitgehend egal, jeder hat da so seine Stärken, Schwächen und Besonderheiten.

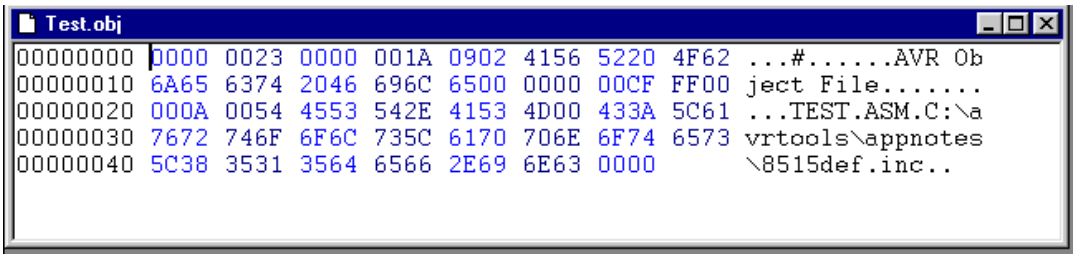
Beim WAVRASM klickt man dazu einfach auf den Menuepunkt mit der Aufschrift "Assemble". Das Ergebnis ist in diesem Bild zu sehen. Der Assembler geruht uns damit mitzuteilen, dass er das Programm übersetzt hat. Andernfalls würfe er mit dem Schlagwort „Error“ um sich. Immerhin ein Wort Code ist dabei erzeugt worden. Und er hat aus unserer einfachen Textdatei gleich vier neue Dateien erzeugt. In der ersten der vier neuen Dateien, TEST.EEP, befindet sich der Inhalt, der in das EEPROM geschrieben werden soll. Er ist hier ziemlich uninteressant, weil wir nichts ins EEPROM programmieren wollten. Hat er gemerkt und die Datei auch gleich wieder gelöscht.



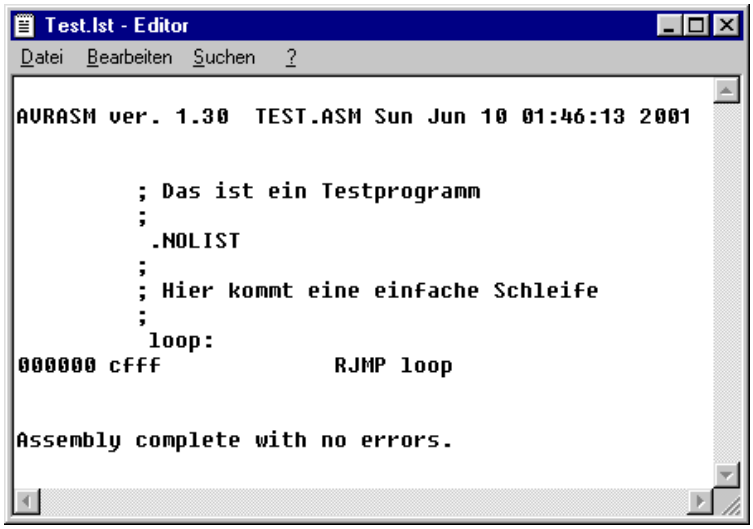


Die zweite Datei, TEST.HEX, ist schon wichtiger, weil hier die Instruktionsworte untergebracht sind. Diese Datei brauchen wir zum Programmieren des Prozessors. Sie enthält die nebenstehenden Hieroglyphen. Die hexadezimalen Zahlen sind als ASCII-Zeichen aufgelöst und werden mit Adressangaben und Prüfsummen zusammen abgelegt. Dieses Format heißt Intel-Hex-Format und ist uralt. Jedenfalls versteht diesen Salat jede Programmier-Software recht gut.

Die dritte Datei, TEST.OBJ, kriegen wir später, sie wird zum Simulieren gebraucht. Ihr Format ist hexadezimal und von ATMEL speziell zu diesem Zweck definiert. Sie sieht im Hex-Editor wie abgebildet aus. Merke: Diese Datei wird vom Programmiergerät nicht verstanden!



Die vierte Datei, TEST.LST, können wir uns mit einem Editor anschauen.

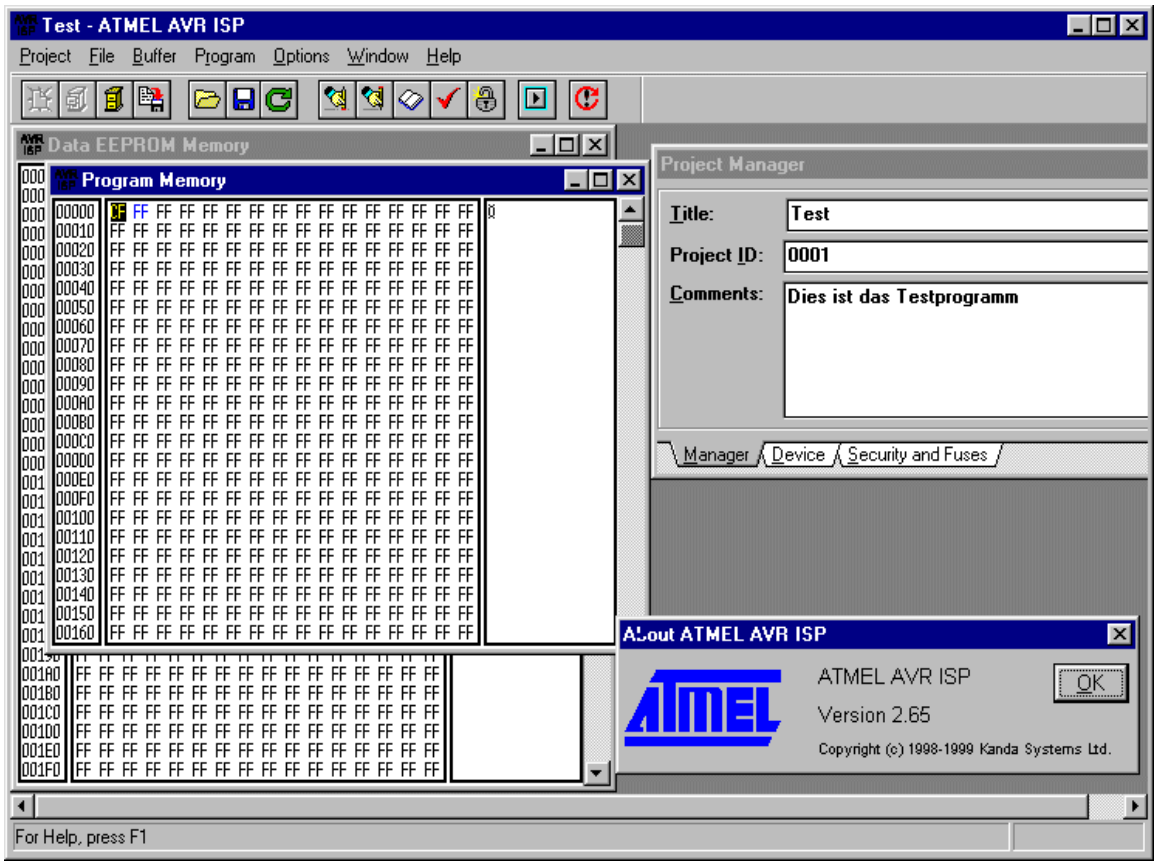


Sie enthält das Nebenstehende. Wir sehen das Programm mit allen Adressen (hier: "000000"), Maschineninstruktionen (hier: "cfff") und Fehlermeldungen (hier: keine) des Assemblers. Die List-Datei braucht man selten, aber gelegentlich.

4.3 Das Programmieren des Chips

Nun muss der in der Hex-Datei abgelegte Inhalt dem AVR-Chip beigebracht werden. Das erledigt Brenner-Software. Üblich sind die Brenn-Tools im Studio von ATMEL, das vielseitige Pony-Prog 2000 und andere mehr (konsultiere die Lieblings-Suchmaschine).

Das Brennen wird hier am Beispiel des ISP gezeigt. Das gibt es nicht mehr, arbeitet aber sehr viel anschaulicher als moderne Software, weil es einen Blick in das Flash-Memory des AVR ermöglicht.
Zu sehen ist der Inhalt des Flash- Speichers nach dem Löschen: lauter Einsen!



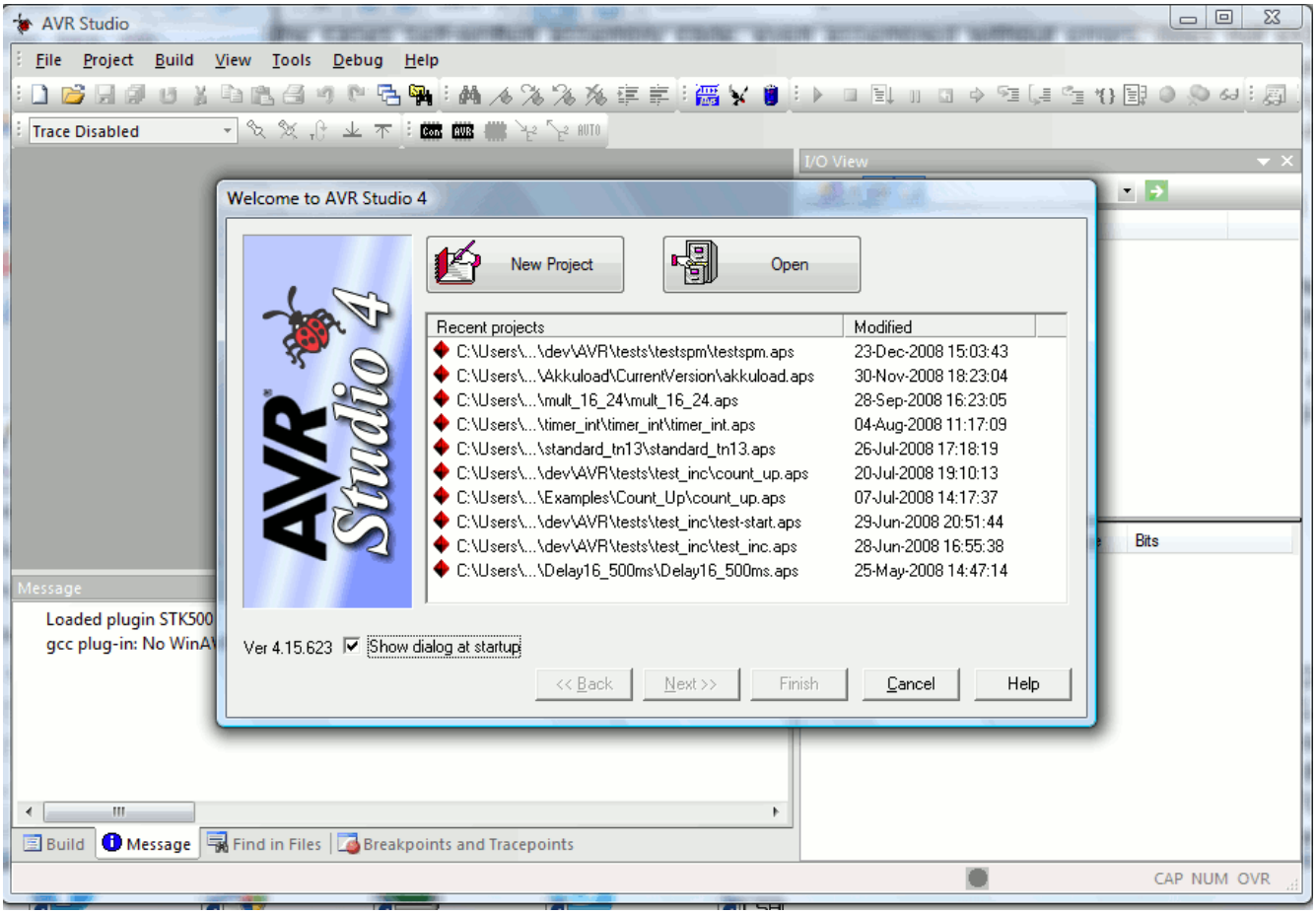
Wir starten das Programm ISP, erzeugen ein neues Projekt und laden die gerade erzeugte Hex-Datei mit LOAD PROGRAM. Das sieht dann wie im Bild aus. Wenn wir nun mit dem Menü „Program“ den Chip programmieren, dann legt dieser gleich los. Beim Brennen gibt eine Reihe von weiteren Voraussetzungen (richtige Schnittstelle auswählen, Adapter an Schnittstelle angeschlossen, Chip auf dem Programmierboard vorhanden, Stromversorgung auf dem Board eingeschaltet, ...), ohne die das natürlich nicht geht.

4.4 Das Simulieren im Studio

In einigen Fällen hat selbst geschriebener Code nicht bloß Tippfehler, sondern hartnäckige logische Fehler. Die Software macht einfach nicht das, was sie soll, wenn der Chip damit gebrannt wird. Tests auf dem Chip selbst können kompliziert sein, speziell wenn die Hardware aus einem Minimum besteht und keine Möglichkeit besteht, Zwischenergebnisse auszugeben oder wenigstens Hardware-Signale zur Fehlersuche zu benutzen. In diesen Fällen hat das Studio-Software-Paket von ATMEL einen Simulator, der für die Entwanzung ideale Möglichkeiten bietet. Das Programm kann Schritt für Schritt abgearbeitet werden, die Zwischenergebnisse in Prozessorregistern sind überwachbar, etc.

Die folgenden Bilder sind der Version 4 entnommen, die ältere Version 3 sieht anders aus, macht aber in etwa dasselbe. Die Studio Software enthält alles, was man zur Entwicklung, zur Fehlersuche, zur Simulation, zum Brennen der Programme und an Hilfen braucht. Nach der Installation und dem Start des Riesenpakets (z. Zt. ca. 100 MB) sieht das erste Bild wie folgt aus: Der erste Dialog fragt, ob wir ein neues oder bereits bestehendes Projekt öffnen wollen. Im Falle einer Erstinstallation ist "New Project" die korrekte Ant-

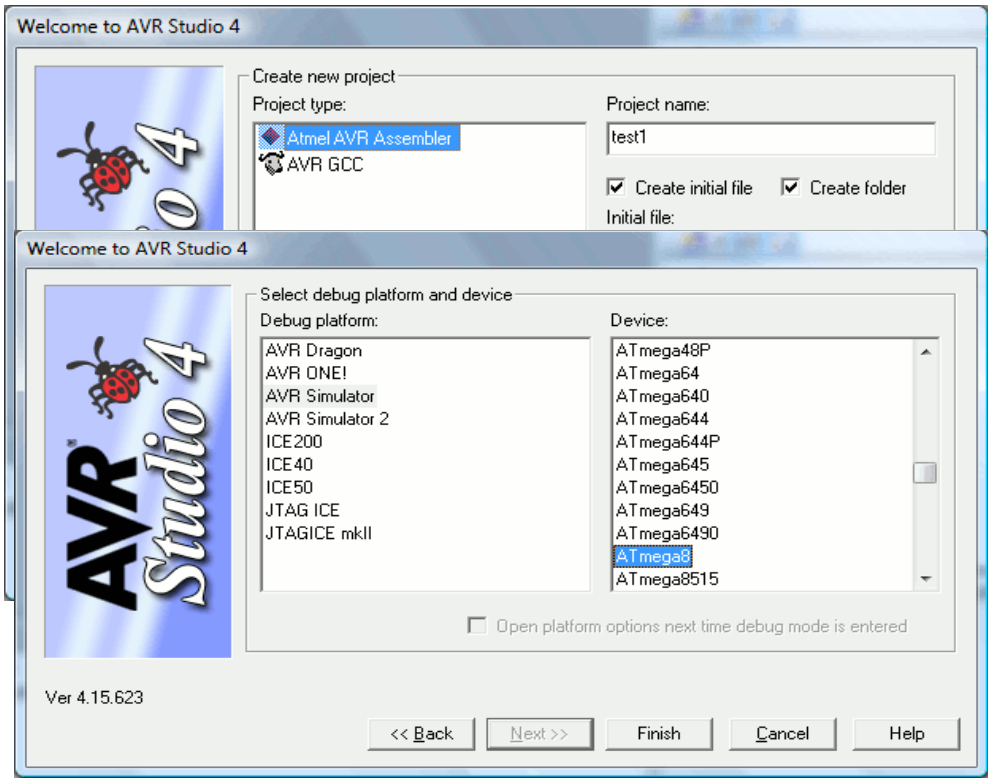
wort.



Der Knopf "Next>>" bringt uns zum Einstellungsdialog für das neue Projekt:

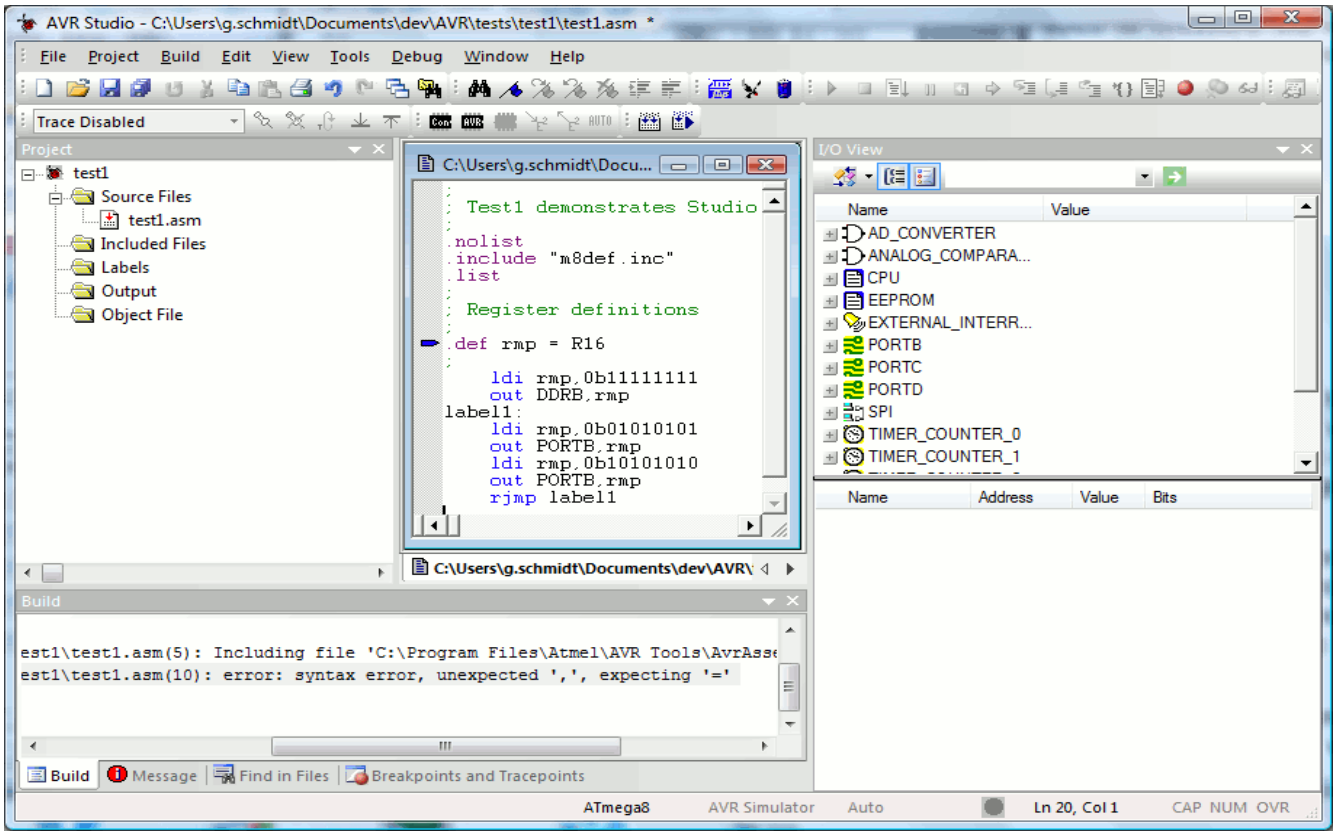
In diesem Dialog wird „Assembler“ ausgewählt, das Projekt kriegt einen aussagekräftigen Namen („test1“), die erste Quellcode-Datei lassen wir erzeugen und auch gleich einen neuen Ordner dazu. Das Ganze geht in einen Ordner auf der Festplatte, zu dem wir Schreibzugriff haben und wird mit „Next“ beendet.

Hier wird die Plattform "Simulator" ausgewählt. Ferner wird hier der Prozessor-Zieltyp (hier: ATmega8) ausgewählt. Der Dialog wird mit "Finish" abgeschlossen.



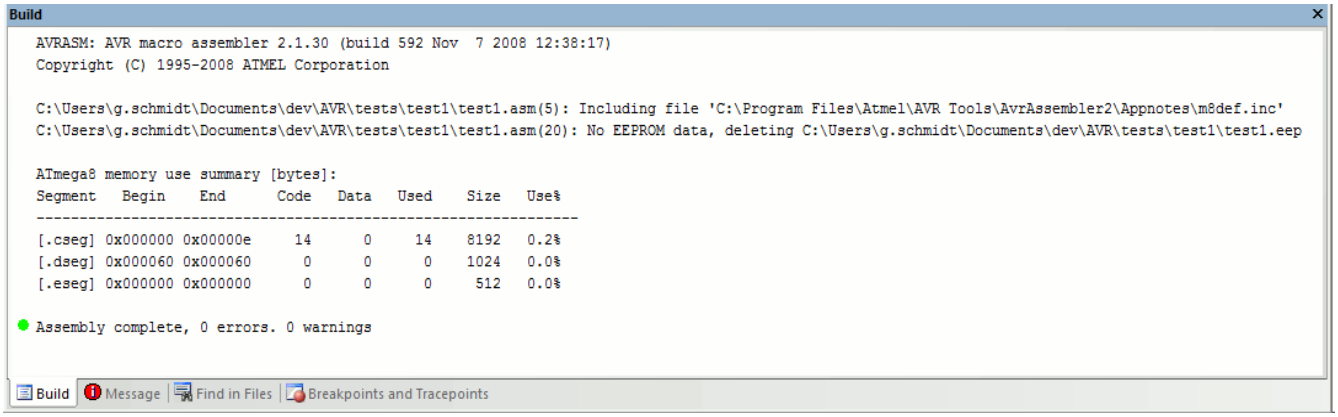
Das öffnet ein ziemlich großes Fenster mit ziemlich vielen Bestandteilen:

- links oben das Fenster mit der Projektverwaltung, in dem Ein- und Ausgabedateien verwaltet werden können,
- in der Mitte oben ist das Editorfenster zu sehen, das den Inhalt der Datei "test1.asm" anzeigt und in das der Quelltext eingegeben werden kann,
- rechts oben die Hardware-Ansicht des Zielprozessors (dazu später mehr),
- links unten das "Build"-Fensters, in dem Diagnose-Ausgaben erscheinen, und
- rechts unten ein weiteres Feld, in dem diverse Werte beim Entwanzen angezeigt werden.



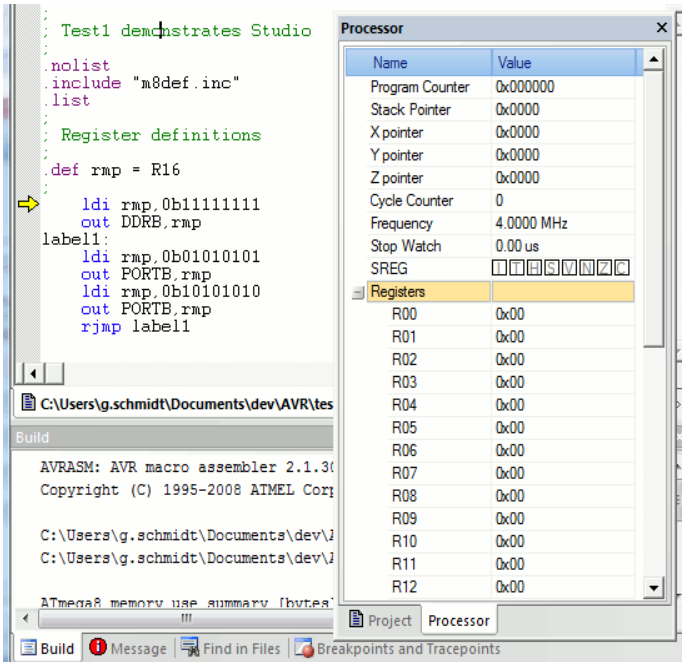
Alle Fenster sind in Größe und Lage verschiebbar.

Für den Nachvollzug der nächsten Schritte im Studio ist es notwendig, das im Editorfenster sichtbare Programm abzutippen (zu den Bestandteilen des Programmes später mehr) und mit "Build" und "Build" zu übersetzen. Das bringt die Ausgabe im unteren Build-Fenster zu folgender Ausgabe:

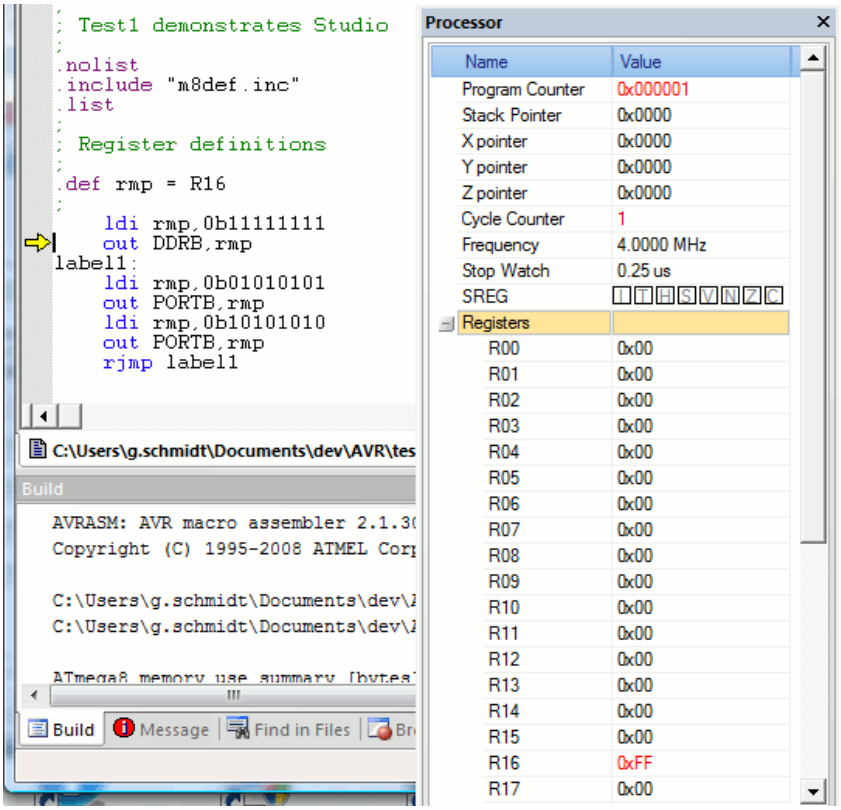


Dieses Fenster teilt uns mit, dass das Programm fehlerfrei übersetzt wurde und wieviel Code dabei erzeugt wurde (14 Worte, 0,2% des verfügbaren Speicherraums. Nun wechseln wir in das Menü "Debug",

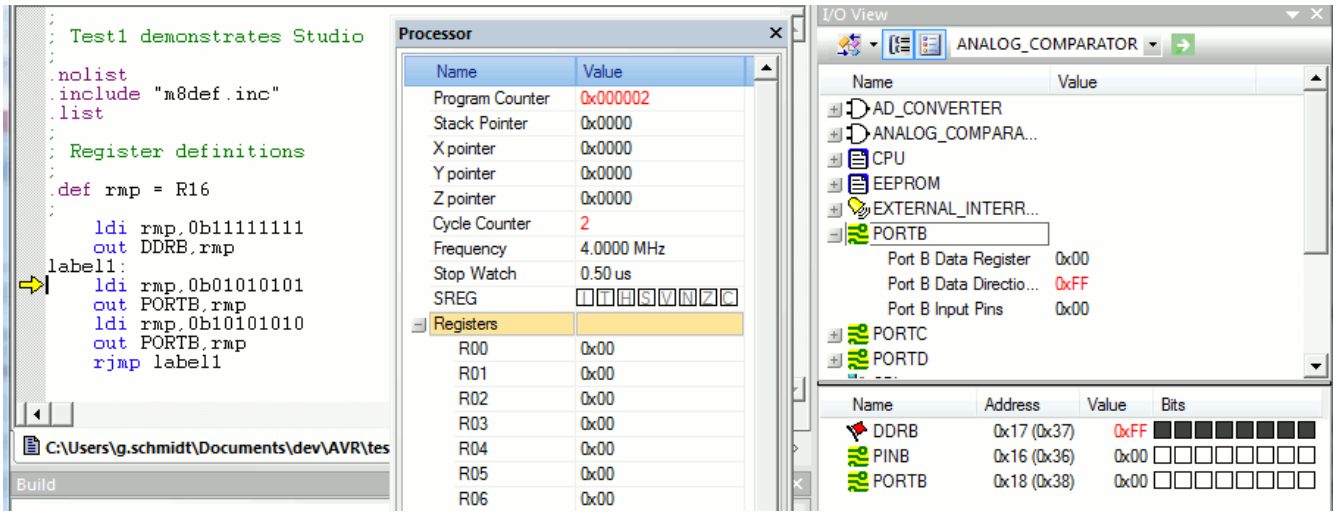
was unsere Fensterlandschaft ein wenig verändert.



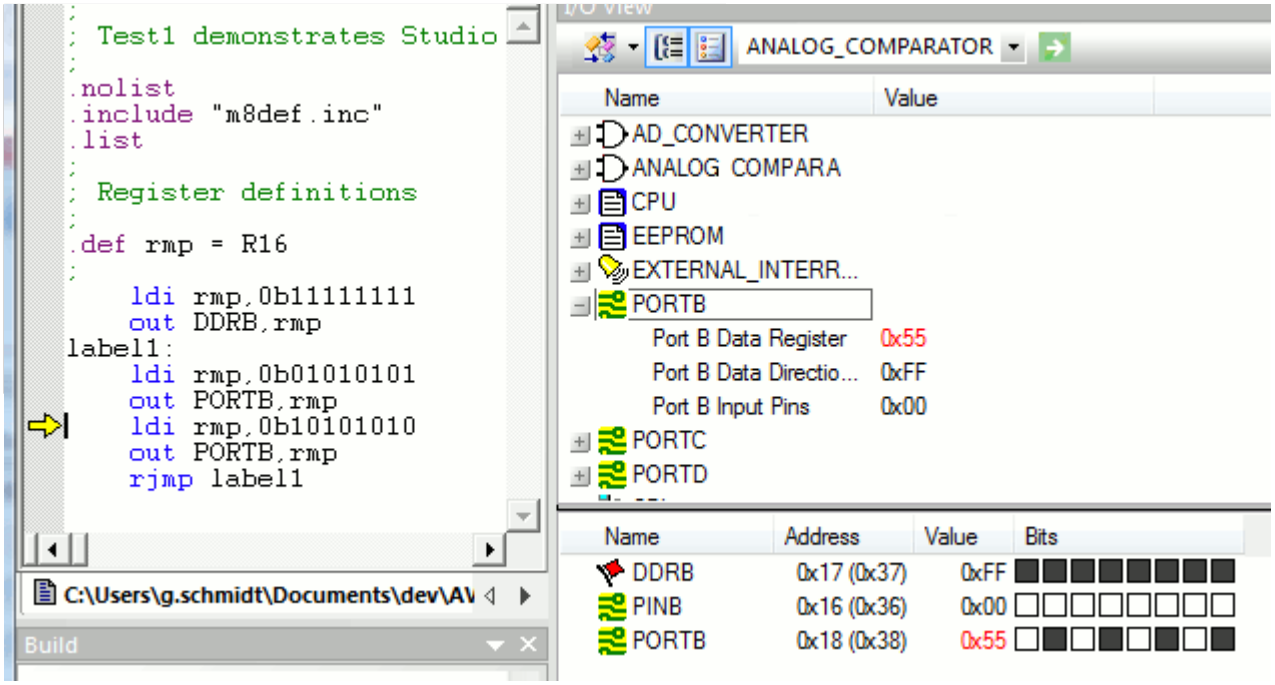
Im linken Editor-Fenster taucht nun ein gelber Pfeil auf, der auf die erste ausführbare Instruktion des Programmes zeigt. Mit "View", "Toolbars" und "Processor" bringt man das Prozessorfenster rechts zur Anzeige. Es bringt uns Informationen über die aktuelle Ausführungsadresse, die Anzahl der verarbeiteten Instruktionen, die verstrichene Zeit und nach dem Klicken auf das kleine "+" neben "Registers" den Inhalt der Register.



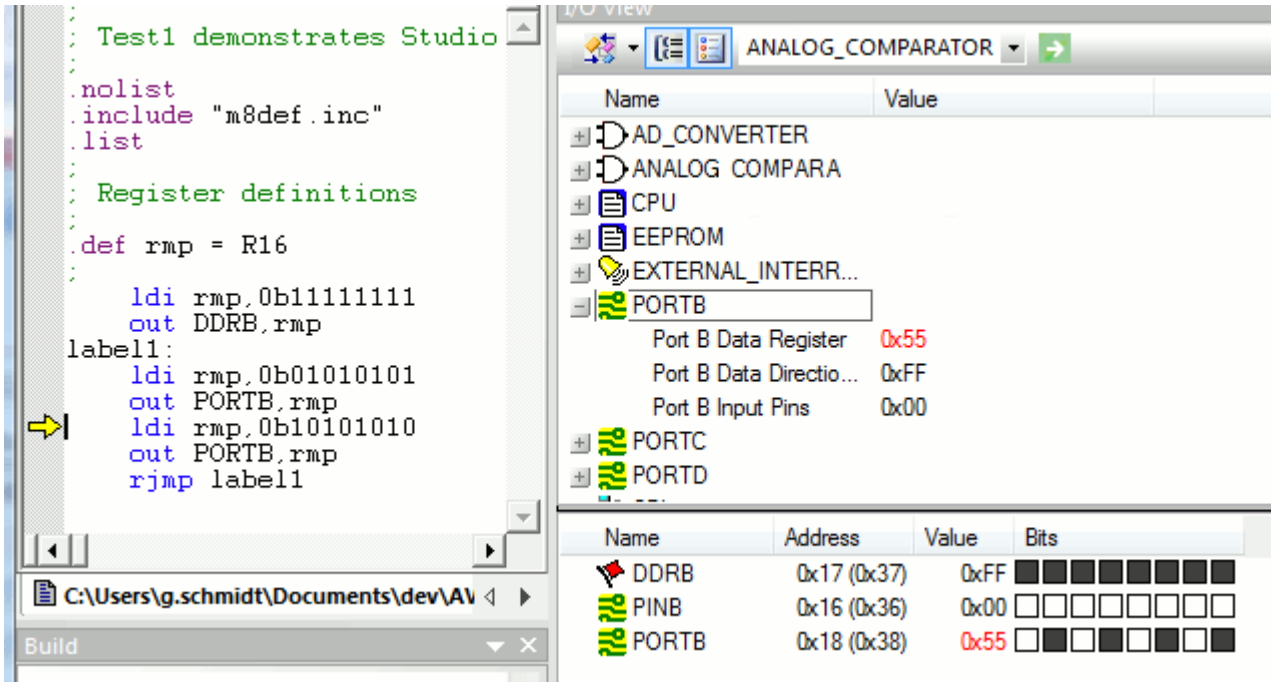
Mit "Debug" und "Step into" oder der Taste F11 wird nun ein Einzelschritt vorgenommen.
Der Programmzähler hat sich nun verändert, er steht auf "0x000001". Der Schrittzähler hat einen Zyklus gezählt und im Register R16 steht nun hexadezimal "0xFF" oder dezimal 255. LDI lädt also einen Hexadezimalwert in ein Register. Nach einem weiteren Schritt mit F11 und Aufklappen von PORTB im Hardware-Fenster "I/O-View" ist die Wirkung der gerade abgearbeiteten Instruktion "out PORTB,rmp" zu sehen:
Das Richtungs-Port-Register Data Direction PortB (DDRB) hat nun 0xFF und in der Bitanzeige unten acht schwarze Kästchen. Mit zwei weiteren Einzelschritten (F11) hat dann der Ausgabeport PORTB den Hexadezimalwert "0x55".



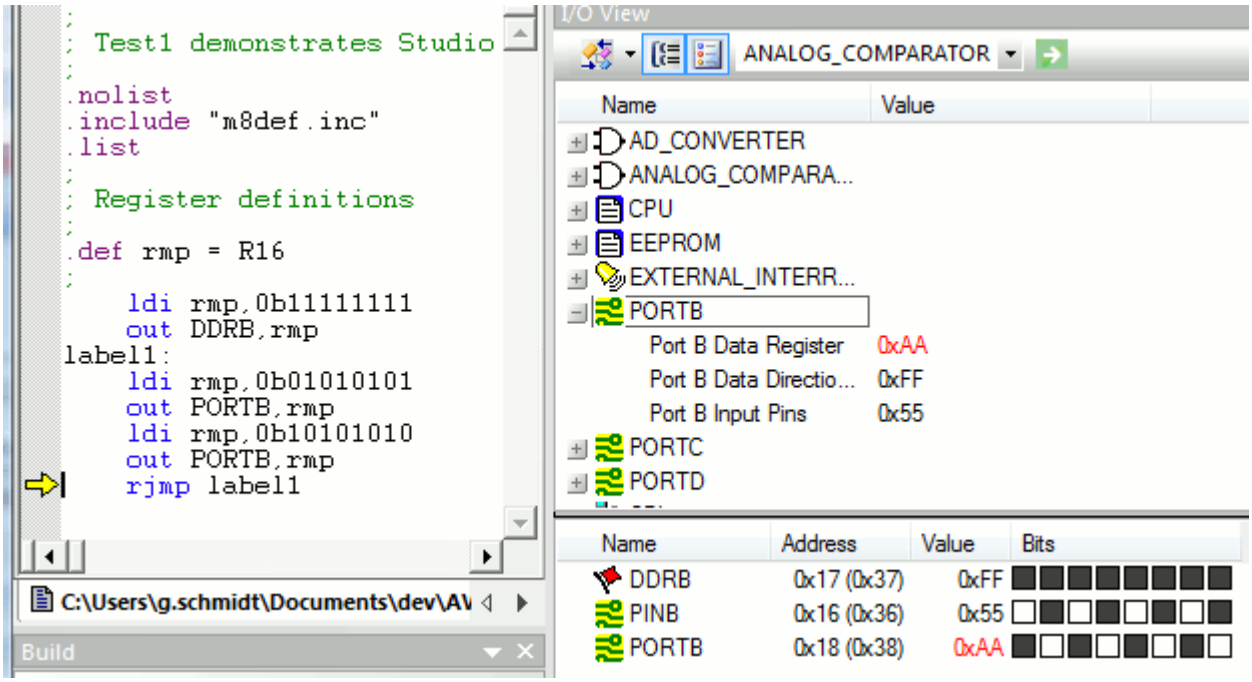
Das Richtungs-Port-Register Data Direction PortB (DDRB) hat nun 0xFF und in der Bitanzeige unten acht schwarze Kästchen.



Mit zwei weiteren Einzelschritten (F11) hat dann der Ausgabeport PORTB den Hexadezimalwert "0x55".



Zwei weitere Schritte klappen die vier schwarzen und die vier weißen Kästchen um.



Erstaunlicherweise ist nun der Port PINB dem vorhergehenden Bitmuster gefolgt. Aber dazu dann später im Abschnitt über Ports.

Soweit dieser kleine Ausflug in die Welt des Simulators. Er kann noch viel mehr, deshalb sollte dieses Werkzeug oft und insbesondere bei allen hartnäckigen Fällen von Fehlern verwendet werden. Klicken Sie sich mal durch seine Menüs, es gibt viel zu entdecken.

5 Struktur von AVR-Assembler-Programmen

In diesem Abschnitt werden die Strukturen vorgestellt, die für Assembler-Programme typisch sind und sich immer wieder wiederholen. Dazu gehören Kommentare, die Angaben im Kopf des Programmes, der Code zu Beginn des eigentlichen Programmes und der Aufbau von Programmen.

5.1 Kommentare

Das wichtigste an Assemblerprogrammen sind die Kommentare. Ohne Kommentierung des geschriebenen Codes blickt man schon nach wenigen Tagen oft nicht mehr durch, wofür das Programm gut war oder was an dieser Stelle des Programmes eigentlich warum getan wird.

Man kann natürlich auch ohne Kommentare Programme schreiben, vor allem wenn man sie vor anderen und vor sich selbst geheim halten will. Ein Kommentar beginnt mit einem Semikolon. Alles, was danach in dieser Zeile folgt, wird vom Übersetzungsprogramm, dem Assembler, einfach ignoriert. Wenn man mehrere Zeilen lange Kommentare schreiben möchte, muss man eben jede weitere Zeile mit einem Semikolon beginnen.

So sieht dann z.B. der Anfang eines Assemblerprogrammes z.B. so aus:

```
;
; Klick.asm, Programm zum Ein- und Ausschalten eines Relais alle zwei Sekunden
; Geschrieben von G.Schmidt, letzte Änderung am 6.10.2001
;
```

Kommentieren kann und soll man aber auch einzelne Abschnitte eines Programmes, wie z.B. eine abgeschlossene Routine oder eine Tabelle. Randbedingungen wie z.B. die dabei verwendeten Register, ihre erwarteten Inhalte oder das Ergebnis nach der Bearbeitung des Teilabschnittes erleichtern das spätere Aufsuchen von Fehlern und vereinfachen nachträgliche Änderungen. Man kann aber auch einzelne Zeilen mit Befehlen kommentieren, indem man den Rest der Zeile mit einem Semikolon vor dem Assembler abschirmt und dahinter alles mögliche anmerkt:

```
LDI R16,0x0A ; Hier wird was geladen
MOV R17,R16 ; und woanders hinkopiert
```

5.2 Angaben im Kopf des Programmes

Den Sinn und Zweck des Programmes, sein Autor, der Revisionsstand und andere Kommentare haben wir schon als Bestandteil des Kopfes identifiziert. Weitere Angaben, die hier hin gehören, sind der Prozessortyp, für den die Software geschrieben ist, die wichtigsten Konstanten (zur übersichtlichen Änderung) und die Festlegung von errinnerungsfördernden Registernamen.

Der Prozessortyp hat dabei eine besondere Bedeutung. Programme laufen nicht ohne Änderungen auf jedem Prozessortyp. Nicht alle Prozessoren haben den gleichen Befehlssatz, jeder Typ hat seine typische Menge an EEPROM und SRAM, usw. Alle diese Besonderheiten werden in einer besonderen Kopfdatei (header file) festgelegt, die in den Code importiert wird. Diese Dateien heißen je nach Typ z. B. 2323def.inc, 8515def.inc, etc. und werden vom Hersteller zur Verfügung gestellt.

Es ist guter Stil, mit dieser Datei sofort nach dem Kommentar im Kopf zu beginnen. Sie wird folgendermaßen eingelesen:

```
.NOLIST ; Damit wird das Auflisten der Datei abgestellt
.INCLUDE "C:\avrtools\appnotes\8515def.inc" ; Import der Datei
```

```
.LIST ; Auflisten wieder anschalten
```

Der Pfad, an dem sich die Header-Datei befindet, kann natürlich weggelassen werden, wenn sie sich im gleichen Verzeichnis wie die Assemblerdatei befindet. Andernfalls ist der Pfad entsprechend den eigenen Verhältnissen anzupassen.

Das Auflisten der Datei beim Übersetzen kann nervig sein, weil solche Header-Dateien sehr lang sind, beim Auflisten des übersetzten Codes (entstehende .lst-Datei) entsprechend lange Listen von meist uninteressanten (weil trivialen) Informationen produzieren. Das Abschalten vor dem Einlesen der Header-Datei spart jede Menge Papier beim Ausdrucken der List-Datei. Es lohnt sich aber, einen kurzen Blick in die Include-Datei zu werfen. Zu Beginn der Datei wird mit

```
.DEVICE AT90S8515 ; Festlegung des Zieldevices
```

der Zielchip definiert. Das wiederum bewirkt, dass Instruktionen, die auf dem Zielchip nicht definiert sind, vom Assembler mit einer Fehlermeldung zurückgewiesen werden. Die Device-Anweisung an den Assembler braucht also beim Einlesen der Header-Datei nicht noch einmal in den Quellcode eingegeben werden (ergäbe eine Fehlermeldung).

Hier sind z.B. auch die Register XH, XL, YH, YL, ZH und ZL definiert, die beim byteweisen Zugriff auf die Doppelregister X, Y und Z benötigt werden. Ferner sind darin alle Port-Speicherstellen definiert, z. B. erfolgt hier die Übersetzung von PORTB in hex 18. Schließlich sind hier auch alle Port-Bits mit denjenigen Namen registriert, die im Datenblatt des jeweiligen Chips verwendet werden. So wird hier z.B. das Portbit 3 beim Einlesen von Port B als PINB3 übersetzt, exakt so wie es auch im Datenblatt heißt.

Mit anderen Worten: vergisst man die Einbindung der Include-Datei des Chips zu Beginn des Programmes, dann hagelt es Fehlermeldungen, weil der Assembler nur Bahnhof versteht. Die resultierenden Fehlermeldungen sind nicht immer sehr aussagekräftig, weil fehlende Labels und Konstanten von manchem ATMEL-Assembler nicht mit einer Fehlermeldung quittiert werden. Stattdessen nimmt der Assembler einfach an, die fehlende Konstante sei Null und übersetzt einfach weiter. Man kann sich leicht vorstellen, welches Chaos dabei herauskommt. Der arglose Programmierer denkt: alles in Ordnung. In Wirklichkeit wird ein ziemlicher Käse im Chip ausgeführt.

In den Kopf des Programmes gehören insbesondere auch die Register-Definitionen, also z.B.

```
.DEF mpr = R16 ; Das Register R16 mit einem Namen belegen
```

Das hat den Vorteil, dass man eine vollständige Liste der Register erhält und sofort sehen kann, welche Register verwendet werden und welche noch frei sind. Das Umbenennen vermeidet nicht nur Verwendungskonflikte, die Namen sind auch aussagekräftiger.

Ferner gehört in den Kopf die Definition von Konstanten, die den gesamten Programmablauf beeinflussen können. So eine Konstante wäre z.B. die Oszillatorfrequenz des Chips, wenn im Programm später die serielle Schnittstelle verwendet werden soll. Mit

```
.EQU fq = 4000000 ; Quarzfrequenz festlegen
```

zu Beginn der Assemblerdatei sieht man sofort, für welchen Takt das Programm geschrieben ist. Beim Umschreiben auf eine andere Frequenz muss nur diese Zahl geändert werden und man braucht nicht den gesamten Quelltext nach dem Auftauchen von 4000000 zu durchsuchen.

5.3 Angaben zum Programmbeginn

Nach dem Kopf sollte der Programmcode beginnen. Am Beginn jedes Codes stehen die Reset- und Interrupt-Vektoren (später dazu mehr). Da diese relative Sprünge enthalten müssen, folgen darauf am besten die Interrupt-Service-Routinen. Danach ist ein guter Platz für abgeschlossene Unterprogramme. Danach sollte das Hauptprogramm stehen. Das Hauptprogramm beginnt mit immer mit dem Einstellen der Register-Startwerte, dem Initialisieren des Stackpointers und der verwendeten Hardware. Danach geht es programmspezifisch weiter.

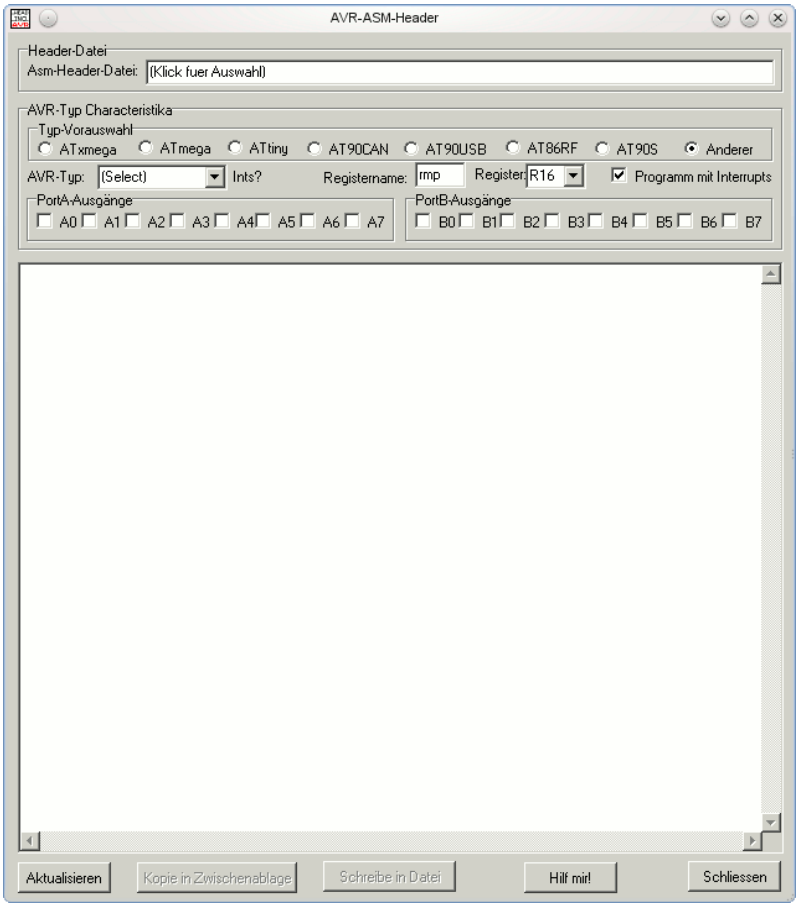
5.4 Beispielgliederung

Hier ist eine beispielhafte Gliederung eines Assemblerprogrammes abgebildet. So oder ähnlich sollten alle Programme aussehen.

```
;
; ******************************************
; * [Projekttitel hier einfuegen]          *
; * [Mehr Infos zur Software-Version hier] *
; * (C)20xx by [Copyright Info hier]       *
; ******************************************
;
; Einlesen des Headerfiles fuer den Zieltyp
.NOLIST
.INCLUDE "tn13def.inc" ; Header fuer ATTINY13
.LIST
;
; ============================================
;   H A R D W A R E   I N F O R M A T I O N
; ============================================
;
; [Alle Informationen ueber die Zielhardware
;  hier einfuegen]
;
; ============================================
;      P O R T S   U N D   P I N S
; ============================================
;
; [Namen fuer die Hardwareports und -pins hier]
; Format: .EQU Controlportout = PORTA
;         .EQU Controlportin = PINA
;         .EQU LedOutputPin = PORTA2
;
; ============================================
;  A E N D E R B A R E   K O N S T A N T E N
; ============================================
;
; [Alle Konstanten, deren Aenderung Sinn macht,
;  sollten hier definiert werden]
; Format: .EQU const = $ABCD
;
; ============================================
```

```
;      F E S T E + A B G E L E I T E T E
;           K O N S T A N T E N
; ============================================
;
; [Feste Konstanten, deren Aenderung keinen
;  Sinn macht, sowie aus anderen Konstanten
;  abgeleitete Groessen hier definieren]
; Format: .EQU const = $ABCD
;
; ============================================
;   R E G I S T E R   D E F I N I T I O N E N
; ============================================
;
; [All Registernamen hier definieren, freie
;  Register und die Verwendung unbenannter
;  Register hier ebenfalls eintragen]
; Format: .DEF rmp = R16
; Verwendet: R1 bis R5 fuer Berechnungen
.DEF rmp = R16 ; Multipurpose Register
; Frei: R17 bis 31
;
; ============================================
;       S R A M   D E F I N I T I O N E N
; ============================================
;
.DSEG
.ORG  0X0060
; Format:
;   Label: .BYTE N ; Reserviere N Bytes ab Label:
;
; ============================================
;  R E S E T   U N D   I N T   V E K T O R E N
; ============================================
;
.CSEG
.ORG $0000
        rjmp Main ; Reset Vektor
        reti ; Int Vektor 1
        reti ; Int Vektor 2
        reti ; Int Vektor 3
        reti ; Int Vektor 4
        reti ; Int Vektor 5
        reti ; Int Vektor 6
        reti ; Int Vektor 7
        reti ; Int Vektor 8
        reti ; Int Vektor 9
;
; ============================================
;      I N T E R R U P T   S E R V I C E
; ============================================
;
```

```
; [Alle Interrupt Service Routinen hier]
;
; ===========================================
;   H A U P T - P R O G R A M M   I N I T
; ===========================================
;
Main:
; Initiiere Stapel
        ldi rmp, LOW(RAMEND) ; Init LSB Stapel
        out SPL,rmp
; Initiiere Port B
        ldi rmp,0 ; Richtung Port B
        out DDRB,rmp
; [Alle anderen Initiierungen hier]
        ldi rmp,1<<SE ; Enable sleep mode
        out MCUCR,rmp
        sei ; schalte Ints ein;
; ===========================================
;      P R O G R A M M – S C H L E I F E
; ===========================================
;
Loop:
        sleep ; schlafen legen
         nop ; Dummy fuer Auf-
wachen
        rjmp loop ; zurueck in
die Schlafposition
;
; Ende Quellcode
;
.db „(C)201x by mich!“ ; men-
schenlesbar
.db „C(2)10 xybm ci!h“ ; wort-
weise lesbar
;
```



5.5 Ein Programm zur Erzeugung strukturierter Quellcode-Dateien

Das Programm für die komfortable Erzeugung solcher Dateien gibt es unter

http://www.avr-asm-tutorial.net/avr_de/avr_head/avr_head.html

zum kostenlosen Download.

6 Register

6.1 Was ist ein Register?

Register sind besondere Speicher mit je 8 Bit Kapazität. Sie sehen bitmäßig daher etwa so aus:

7	6	5	4	3	2	1	0

Man merke sich die Nummerierung der Bits: sie beginnt immer bei Null. In einen solchen Speicher passen entweder

- Zahlen von 0 bis 255 (Ganzzahl ohne Vorzeichen),
- Zahlen von -128 bis +127 (Ganzzahl mit Vorzeichen in Bit 7),
- ein Acht-Bit-ASCII-Zeichen wie z.B. 'A' oder auch
- acht einzelne Bits, die sonst nix miteinander zu tun haben (z.B. einzelne Flaggen oder Flags).

Das Besondere an diesen Registern (im Gegensatz zu anderen Speichern) ist, dass sie

- direkt in Instruktionen verwendet werden können, da sie direkt an das Rechenwerk, den Akkumulator, angeschlossen sind,
- Operationen mit ihrem Inhalt mit nur einem Instruktionswort ausgeführt werden können,
- sowohl Quelle von Daten als auch Ziel des Ergebnisses der Operation sein können.

Es gibt 32 davon in jedem AVR. Auch der kleinste AVR hat schon so viele. Diese Eigenschaft macht die AVR ziemlich einzigartig, weil dadurch viele Kopieraktionen und der langsamere Zugriff auf andere Speicherarten oft nicht nötig ist. Die Register werden mit R0 bis R31 bezeichnet, man kann ihnen mit einer Assemblerdirektive aber auch einen etwas wohlklingenderen Namen verpassen, wie z.B.

```
.DEF MeinLieblingsregister = R16
```

Assemblerdirektiven gibt es einige (siehe die Tabelle im Anhang), sie stellen Regie-Anweisungen an den Assembler dar und erzeugen selbst keinen ausführbaren Code. Sie beginnen immer mit einem Punkt.

Statt des Registernamens R16 wird dann fürderhin immer der neue Name verwendet. Das könnte also ein schreibintensives Programm werden.

Mit der Instruktion

```
LDI MeinLieblingsRegister, 150
```

was in etwa bedeutet: Lade die Zahl 150 in das Register R16, aber hurtig, (in englisch: LoaD Immediate) wird ein fester Wert oder eine Konstante in mein Lieblingsregister geladen. Nach dem Übersetzen (Assemblieren) ergibt das im Programmspeicher etwa folgendes Bild:

```
000000 E906
```

In E906 steckt sowohl die Load-Instruktion als auch das Zielregister (R16) als auch die Konstante 150, auch wenn man das auf den ersten Blick nicht sieht. Auch dies macht Assembler bei den AVR zu einer höchst effektiven Angelegenheit: Instruktion und Konstante in einem einzigen Instruktionswort – schnell und effektiv ausgeführt. Zum Glück müssen wir uns um diese Übersetzung nicht kümmern, das macht der Assembler für uns.

In einer Instruktion können auch zwei Register vorkommen. Die einfachste Instruktion dieser Art ist die KopierInstruktion MOV. Sie kopiert den Inhalt des einen Registers in ein anderes Register. Also etwa so:

```
.DEF MeinLieblingsregister = R16
.DEF NochEinRegister = R15
        LDI MeinLieblingsregister, 150
        MOV NochEinRegister, MeinLieblingsregister
```

Die ersten beiden Zeilen dieses großartigen Programmes sind Direktiven, die ausschließlich dem Assembler mitteilen, dass wir anstelle der beiden Registernamen R16 und R15 andere Benennungen zu verwenden wünschen. Sie erzeugen keinen Code! Die beiden Programmzeilen mit LDI und MOV erzeugen Code, nämlich:

```
000000 E906
000001 2F01
```

Die zweite Instruktion schiebt die 150 im Register R16 in das Rechenwerk und kopiert dessen Inhalt in das Zielregister R15. MERKE:

Das erstgenannte Register in der Assemblerinstruktion ist immer das Zielregister, das das Ergebnis aufnimmt.

(Also so ziemlich umgekehrt wie man erwarten würde und wie man es ausspricht. Deshalb sagen viele, Assembler sei schwer zu erlernen!)

6.2 Unterschiede der Register

Schlaumeier würden das obige Programm vielleicht eher so schreiben:

```
.DEF NochEinRegister = R0
        LDI NochEinRegister, 150
```

Und sind reingefallen: Nur die Register R16 bis R31 lassen sich hurtig mit einer Konstante laden, die Register R0 bis R15 nicht! Diese Einschränkung ist ärgerlich, ließ sich aber bei der Konstruktion der Assemblersprache für die AVRs wohl kaum vermeiden.

Es gibt eine Ausnahme, das ist das Nullsetzen eines Registers. Diese Instruktion

```
        CLR MeinLieblingsRegister
```

ist für alle Register zulässig (aber nur, weil weil es eigentlich korrekt XOR MeinLieblingsregister,MeinLieblingsregister heißen müsste und gar kein echtes und ganz anders als LDI MeinLieblingsRegister,0 funktioniert).

Diese zwei Klassen von Registern gibt es außer bei LDI noch bei folgenden Instruktionen:

- ANDI Rx,K ; Bit-Und eines Registers Rx mit einer Konstante K,
- CBR Rx,M ; Lösche alle Bits im Register Rx, die in der Maske M (eine Konstante) gesetzt sind,
- CPI Rx,K ; Vergleiche das Register Rx mit der Konstante K,
- SBCI Rx,K ; Subtrahiere die Konstante K und das Carry-Flag vom Wert des Registers Rx und speichere das Ergebnis im Register Rx,

- SBR Rx,M ; Setze alle Bits im Register Rx, die auch in der Maske M (eine Konstante) gesetzt sind,
- SER Rx ; Setze alle Bits im Register Rx (entspricht LDI Rx,255),
- SBI Rx,K ; Subtrahiere die Konstante K vom Inhalt des Registers Rx und speichere das Ergebnis in Register Rx.

Rx muss bei diesen Instruktionen ein Register zwischen R16 und R31 sein! Wer also vorhat, solche Instruktionen zu verwenden, sollte ein Register oberhalb von R15 dafür auswählen. Das programmiert sich dann leichter. Noch ein Grund, die Register mittels .DEF umzubenennen: in größeren Programmen wechselt sich dann leichter ein Register, wenn man ihm einen besonderen Namen gegeben hat.

6.3 Pointer-Register

Noch wichtigere Sonderrollen spielen die Registerpaare R26/R27, R28/R29 und R30/R31. Diese Pärchen sind so wichtig, dass man ihnen in der AVR-Assemblersprache extra Namen gegeben hat: X, Y und Z. Diese Doppelregister sind als 16-Bit-Pointerregister definiert. Sie werden gerne bei Adressierungen für internes oder externes RAM verwendet (X, Y und Z) oder als Zeiger in den Programmspeicher (Z).

Bei den 16-Bit-Pointern befindet sich das niedrigere Byte der Adresse im niedrigeren Register, das höherwertige Byte im höheren Register. Die beiden Teile haben wieder eigene Namen, nämlich ZH (höherwertig, R31) und ZL (niederwertig, R30). Die Aufteilung in High und Low geht dann etwa folgendermaßen:

```
.EQU Adresse = RAMEND ; In RAMEND steht die höchste SRAM-Adresse des Chips
        LDI YH,HIGH(Adresse)
        LDI YL,LOW(Adresse)
```

Für die Pointerzugriffe selbst gibt es eine Reihe von Spezial-Zugriffs-Instruktionen zum Lesen(LD=Load) und Schreiben (ST=Store), hier am Beispiel des X-Zeigers:

Zeiger	***Vorgang***	***Beispiele***
X	Lese/Schreibe von der Adresse X und lasse den Zeiger unverändert	LD R1,X ST X,R1
X+	Lese/Schreibe von der Adresse X und erhöhe den Zeiger anschließend um Eins	LD R1,X+ ST X+,R1
-X	Vermindere den Zeiger um Eins und lese/schreibe dann erst von der neuen Adresse	LD R1,-X ST -X,R1

Analog geht das mit Y und Z ebenso.

Für das Lesen aus dem Programmspeicher gibt es nur den Zeiger Z und die Instruktion LPM. Sie lädt das Byte an der Adresse Z in das Register R0. Da im Programmspeicher jeweils Worte, also zwei Bytes stehen, wird die Adresse mit zwei multipliziert und das unterste Bit gibt jeweils an, ob das untere oder obere Byte des Wortes im Programmspeicher geladen werden soll. Also etwa so:

```
        LDI ZH,HIGH(2*Adresse)
        LDI ZL,LOW(2*Adresse)
        LPM
```

Nach Erhöhen des Zeigers um Eins wird das niedrigste Bit Eins und mit LPM das zweite (höhere) Byte des Wortes im Programmspeicher gelesen. Da die Erhöhung des 16-Bit-Speichers um Eins auch oft vorkommt, gibt es auch hierfür eine Spezialinstruktion für Zeiger:

```
ADIW ZL,1
LPM
```

ADIW heißt soviel wie ADdiere Immediate Word und kann bis maximal 63 zu dem Wort addieren. Als Register wird dabei immer das untere Zeigerregister angegeben (hier: ZL). Die analoge Instruktion zum Zeiger vermindern heißt SBIW (SuBtract Immediate Word). Anwendbar sind die beiden Instruktionen auf die Registerpaare X, Y und Z sowie auf das Doppelregister R24/R25, das keinen eigenen Namen hat und auch keinen Zugriff auf RAM- oder sonstige Speicher ermöglicht. Es kann als 16-Bit-Wert optimal verwendet werden.

Wie bekommt man aber nun die Werte, die ausgelesen werden sollen, in den Programmspeicher? Dazu gibt es die DB- und DW-Direktiven für den Assembler. Ausnahmsweise erzeugen diese beiden Direktiven Code, und zwar die eingetippten Konstanten. Byte-weise Listen werden so erzeugt:

```
Liste:
.DB 123,45,67,78 ; eine Liste mit vier Bytes
.DB "Das ist ein Text. " ; eine Liste mit einem Text
```

Auf jeden Fall ist bei .DB darauf achten, dass die Anzahl der einzufügenden Bytes pro Zeile geradzahlig sein muss. Sonst fügt der Assembler in seiner Not noch ein Nullbyte am Ende hinzu, das vielleicht gar nicht erwünscht ist und alles durcheinander bringt.

Das Problem gibt es bei wortweise organisierten Tabellen nicht. Die sehen so aus:

```
.DW 12345,6789 ; zwei Worte
```

Statt der Konstanten können selbstverständlich auch Labels (Sprungadressen) eingefügt werden, also z.B. so:

```
Label1:
   [... hier kommen irgendwelche Instruktionen...]
Label2:
   [... hier kommen noch irgendwelche Instruktionen...]
Sprungtabelle:
.DW Label1,Label2
```

Beim Lesen per LPM erscheint immer das niedrigere Byte der 16-Bit-Zahl zuerst!

Und noch was für Exoten, die gerne von hinten durch die Brust ins Auge programmieren: Die Register sind auch mit Zeigern lesbar und beschreibbar. Sie liegen an der Adresse 0000 bis 001F. Das kann man nur gebrauchen, wenn man auf einen Rutsch eine Reihe von Registern in das RAM kopieren will oder aus dem RAM laden will. Lohnt sich aber erst ab 5 Registern.

6.4 Empfehlungen zur Registerwahl

- Register immer mit der .DEF-Anweisung festlegen, nie direkt verwenden. Das ist praktischer, weil Verwechslungen vermieden und Doppelverwendungen erkannt werden. Wer es noch praktischer haben

will, setzt vor den Namen ein kleines r, also z. B. rZaehler.

- Werden Pointer-Register für RAM u.a. benötigt, R26 bis R31 dafür reservieren.
- 16-Bit-Zähler realisiert man am besten im Registerpaar R24/R25, weil es ADIW und SBIW und nicht als Pointer verwendet werden kann.
- Soll aus dem Programmspeicher gelesen werden, Z (R30/31) und R0 dafür reservieren.
- Werden oft konstante Werte oder Zugriffe auf einzelne Bits in einem Register verwendet, dann die Register R16 bis R23 dafür vorzugsweise reservieren.
- Muss bei Interrupts der Inhalt des Flaggenregisters gesichert werden, vorzugsweise R15 für diesen Zweck verwenden.
- Für alle anderen Anwendungsfälle vorzugsweise R1 bis R14 verwenden.

7 Ports

7.1 Was ist ein Port?

Ports sind eigentlich ein Sammelsurium verschiedener Speicher. In der Regel dienen sie der Kommunikation mit irgendeiner internen Gerätschaft wie z.B. den Timern oder der Seriellen Schnittstelle oder der Bedienung von äußeren Anschlüssen wie den Parallel-Schnittstellen des AVR. Der wichtigste Port wird weiter unten besprochen: das Status-Register, das an das wichtigste interne Gerät, nämlich den Akkumulator, angeschlossen ist.

7.2 Port-Organisation

Es gibt insgesamt 64 direkt adressierbare Ports, die aber nicht bei allen AVR-Typen auch tatsächlich physikalisch vorhanden sind. Bei größeren ATmega gibt es neben den direkt adressierbaren Ports noch indirekt adressierbare Ports, doch dazu später mehr. Je nach Größe und Ausstattung des Typs sind eine Reihe von Ports sinnvoll ansprechbar. Welche der Ports in welchem Typ tatsächlich vorhanden sind, ist letztlich aus den Datenblättern zu erfahren. Hier ein Ausschnitt aus den Ports des ATmega8 (von ATMEL zur allgemeinen Verwirrung auch "Register" genannt:

Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C
0x3E (0x5E)	SPH	–	–	–	–	–	SP10	SP9	SP8
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x3C (0x5C)	Reserved								
0x3B (0x5B)	GICR	INT1	INT0	–	–	–	–	IVSEL	IVCE
0x3A (0x5A)	GIFR	INTF1	INTF0	–	–	–	–	–	–
0x39 (0x59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0
0x38 (0x58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	–	TOV0
0x37 (0x57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
0x36 (0x56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
0x35 (0x55)	MCUCR	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
0x34 (0x54)	MCUCSR	–	–	–	–	WDRF	BORF	EXTRF	PORF
0x33 (0x53)	TCCR0	–	–	–	–	–	CS02	CS01	CS00
0x32 (0x52)	TCNT0	Timer/Counter0 (8 Bits)							
0x31 (0x51)	OSCCAL	Oscillator Calibration Register							
0x30 (0x50)	SFIOR	–	–	–	–	ACME	PUD	PSR2	PSR10
0x2F (0x4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
0x2E (0x4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
0x2D (0x4D)	TCNT1H	Timer/Counter1 – Counter Register High byte							
0x2C (0x4C)	TCNT1L	Timer/Counter1 – Counter Register Low byte							

Ports haben eine feste Adresse (in dem oben gezeigten Ausschnitt z. B. 0x3F für den Port SREG, 0x steht dabei für hexadezimal!), über die sie angesprochen werden können. Die Adresse gilt teilweise unabhängig vom AVR-Typ, teilweise aber auch nicht. So befindet sich das Statusregister SREG immer an der Adresse 0x3F, der Ausgabeport der Parallelschnittstelle B immer an der Portadresse 0x18.

Ports nehmen oft ganze Zahlen auf (z. B. TCCR0 im Ausschnitt oben), sie können aber auch aus einer Reihe einzelner Steuerbits bestehen. Diese einzelnen Bits haben dann eigene Namen, so dass sie mit Instruktionen zur Bitmanipulation angesteuert werden können. In der Portliste hat auch jedes Bit in dem jeweiligen Port seinen speziellen symbolischen Namen, z. B. hat das Bit 7 im Port "GICR" den symbolischen

Namen "INT1".

7.3 Port-Symbole, Include

Diese Adressen und Bit-Nummern muss man sich aber nicht merken. In den Include-Dateien zu den einzelnen AVR-Typen, die der Hersteller zur Verfügung stellt, sind die jeweiligen verfügbaren Ports und ihre Bits mit wohlklingenden Namen belegt. So ist in den Include-Dateien die Assemblerdirektive

```
.EQU PORTB, 0x18
```

angegeben und wir müssen uns fürderhin nur noch merken, dass der Port B PORTB heißt. Für das Bit INT1 im Port GICR ist in der Include-Datei definiert:

```
.EQU INT1, 7
```

Die Include-Datei des AVR ATmega8515 heißt "m8515def.inc" und kommt mit folgender Direktive in die Quellcode-Datei:

```
.INCLUDE "m8515def.inc"
```

oder, wenn man nicht mit dem Studio arbeitet,

```
.INCLUDE "C:\PfadNachIrgendwo\m8515def.inc
```

und alle für diesen Typ bekannten Portregister und Steuerbits sind jetzt mit ihren symbolischen Alias-Namen leichter ansprechbar.

7.4 Ports setzen

In Ports kann und muss man Werte schreiben, um die betreffende Hardware zur Mitarbeit zu bewegen. So enthält z.B. das MCU General Control Register, genannt MCUCR, eine Reihe von Steuerbits, die das generelle Verhalten des Chips beeinflussen (siehe im Ausschnitt oben bzw. die Beschreibung des MCUCR später im Detail). MCUCR ist ein mit Einzelbits vollgepackter Port, in dem jedes Bit noch mal einen eigenen Namen hat (ISC00, ISC01, ...). Wer den Port benötigt, um den AVR in den Tiefschlaf zu versetzen, muss sich im Typenblatt die Wirkung dieser Sleep-Bits heraussuchen und durch eine Folge von Instruktionen die entsprechende einschläfernde Wirkung programmieren, für den ATmega8 also z.B. so: ...

```
.DEF MeinLieblingsregister = R16

    LDI MeinLieblingsregister, 0b10000000

    OUT MCUCR, MeinLieblingsregister

    SLEEP
```

Die Out-Instruktion bringt den Inhalt meines Lieblingsregisters, nämlich ein gesetztes Sleep-Enable-Bit SE, zum Port MCUCR und versetzt den AVR gleich und sofort in den Schlaf, wenn er im ausgeführten Code auf eine SLEEP-Instruktion trifft. Da gleichzeitig alle anderen Bits mitgesetzt werden und mit Sleep-Mode SM=0 als Modus der Halbschlaf eingestellt wurde, geht der Chip nicht völlig auf Tauchstation. In diesem Zustand wird die Instruktionsausführung eingestellt, die Timer und andere Quellen von Interrupts bleiben aber aktiv und können den Halbschlaf jederzeit unterbrechen, wenn sich was Wichtiges tut.

7.5 Ports transparenter setzen

Weil "LDI MeinLieblingsregister, 0b10000000" eine ziemlich intransparente Angelegenheit ist, weil zum Verständnis dafür, was eigentlich hier gemacht wird, ein Blick in die Portbits im Datenblatt nötig ist, schreibt man dies besser so:

```
LDI MeinLieblingsRegister, 1<<SE
```

"1<<SE" nimmt eine binäre Eins (= 0b00000001) und schiebt diese SE mal links (<<). SE ist im Falle des ATmega8 als 7 definiert, also die 1 sieben mal links schieben. Das Ergebnis (= 0b10000000) ist gleichbedeutend mit dem oben eingefügten Bitmuster und setzt das Bit SE im Port MCUCR auf Eins.

Wenn wir gleichzeitig auch noch das Bit SM0 auf 1 setzen wollen (was einen der acht möglichen Tiefschlafmodi einschaltet, dann wird das so formuliert:

```
LDI MeinLieblingsRegister, (1<<SE) | (1<<SM0)
```

Der vertikale Strich zwischen beiden Teilergebnissen ist ein binäres ODER, d. h. dass alle auf Eins gesetzten Bits in einem der beiden Teilausdrücke in Klammern Eins werden, also sowohl das Bit SE (= 7) als auch das Bit SM0 (= 4) gesetzt sind, woraus sich 0b10010000 ergibt.

Noch mal zum Merken: die Linksschieberei wird nicht im Prozessor vollzogen, nur beim Assemblieren. Die Instruktion LDI wird auch nicht anders übersetzt, wenn wir die Schieberei verwenden, und ist auch im Prozessor nur eine einzige Instruktion. Von den Symbolen SE und SM0 hat der Prozessor selbst sowieso keine Ahnung, das ist alles nur für den Quellcode-Schreiber von Bedeutung.

Die Linksschieberei hat den immensen Vorteil, dass nun für jeden aufgeklärten Menschen sofort erkennbar ist, dass hier die Bits SE und SM0 manipuliert werden. Nicht dass jeder sofort wüsste, dass damit der Schlafmodus eingestellt wird, aber es liegt wenigstens in der Assoziation näher als 0b10010000.

Noch ein Vorteil: das SE-Bit liegt bei anderen Prozessoren woanders im Port MCUCR als in Bit 7, z. B. im AT90S8515 lag es in Bit 5. SM0 ist bei diesem Typ gar nicht bekannt, bei einer Portierung unseres Quellcodes für den ATmega8 auf diesen Typ kriegen wir dann eine Fehlermeldung (SM0 ist dort nicht definiert) und wir wissen dann sofort, wo wir suchen und nacharbeiten müssen.

7.6 Ports lesen

Umgekehrt lassen sich die Portinhalte mit der IN-Instruktion in beliebige Register einlesen und dort weiterverarbeiten. So lädt

```
.DEF MeinLieblingsregister = R16

    IN MeinLieblingsregister, MCUCR
```

den lesbaren Teil des Ports MCUCR in das Register R16. Den lesbaren Teil deswegen, weil es bei vielen Ports auch nicht belegte Bits gibt, die dann immer als Null eingelesen werden.

Oft will man nur bestimmte Portbits setzen und die Porteinstellung ansonsten so lassen. Das geht mit Lesen-Ändern-Schreiben (Read-Modify-Write) z. B. So:

```
    IN MeinLieblingsregister, MCUCR

    SBR MeinLieblingsRegister, (1<<SE) | (1<<SM0)

    OUT MCUCR,MeinLieblingsRegister
```

Braucht aber halt drei Instruktionen.

7.7 Auf Portbits reagieren

Noch öfter als ganze Ports einlesen muss man auf Änderungen bestimmter Bits der Ports prüfen. Dazu muss nicht der ganze Port gelesen und verarbeitet werden. Es gibt hierfür spezielle Sprunginstruktionen, die aber im Kapitel über das Springen vorgestellt werden. Umgekehrt kommt es oft vor, dass ein bestimmtes Portbit gesetzt oder rückgesetzt werden muss. Auch dazu braucht man nicht den ganzen Port lesen und nach der Änderung im Register dann den neuen Wert wieder zurückschreiben. Die beiden Instruktionen heißen SBI (Set Bit I/O-Register) und CBI (Clear Bit I/O-Register). Ihre Anwendung geht z.B. so:

```
.EQU Aktivbit=0 ; Das zu manipulierende Bit des Ports

    SBI PortB, Aktivbit ; Das Bit wird Eins

    CBI PortB, Aktivbit ; Das Bit wird Null
```

Die beiden Instruktionen haben einen gravierenden Nachteil: sie lassen sich nur auf Ports bis zur Adresse 0x1F anwenden, für Ports darüber sind sie leider unzulässig. Für Ports oberhalb des mit IN und OUT (bis 0x3F) beschreibbaren Adressraums geht das natürlich schon gar nicht.

7.8 Porteinblendung im Speicherraum

Für den Zugang zu den Ports, die im nicht direkt zugänglichen Adressraum liegen (z. B. bei einigen großen ATmega und ATxmega) und für den Exotenprogrammierer gibt es wie bei den Registern auch hier die Möglichkeit, die Ports wie ein SRAM zu lesen und zu schreiben, also mit dem LD- bzw. der ST-Instruktion. Da die ersten 32 Adressen im SRAM-Speicherraum schon mit den Registern belegt sind, werden die Ports mit ihrer um 32 erhöhten Adresse angesprochen, wie z.B. bei

```
.DEF MeinLieblingsregister = R16

    LDI  ZH,HIGH(PORTB+32)

    LDI  ZL,LOW(PORTB+32)

    LD  MeinLieblingsregister,Z
```

Das macht nur im Ausnahmefall einen Sinn, geht aber halt auch. Es ist der Grund dafür, warum das SRAM erst ab Adresse 0x60 beginnt (0x20 für die Register, 0x40 für die Ports reserviert), bei großen ATmega erst bei 0x100.

7.9 Details wichtiger Ports in den AVR

Die folgende Tabelle kann als Nachschlagewerk für die wichtigsten gebräuchlichsten Ports im AT90S8515 dienen. Sie enthält nicht alle möglichen Ports. Insbesondere die Ports der MEGA-Typen und der AT90S4434/8535 sind der Übersichtlichkeit halber nicht darin enthalten! Bei Zweifeln immer die Originaldokumentation befragen!

Gerät	*Kurz*	*Port*	*Name*
Akkumulator	SREG	Status Register	SREG
Stack	SPL/SPH	Stackpointer	SPL/SPH

Gerät	Kurz	Port	Name
Ext.SRAM/Ext.Interrupt	MCUCR	MCU General Control Register	MCUCR
Ext.Int.	INT	Interrupt Mask Register	GIMSK
		Flag Register	GIFR
Timer Interrupts	Timer Int.	Timer Int Mask Register	TIMSK
		Timer Interrupt Flag Register	TIFR
Timer 0	Timer 0	Timer/Counter 0 Control Register	TCCR0
		Timer/Counter 0	TCNT0
Timer 1	Timer 1	Timer/Counter Control Register 1 A	TCCR1A
		Timer/Counter Control Register 1 B	TCCR1B
		Timer/Counter 1	TCNT1
		Output Compare Register 1 A	OCR1A
		Output Compare Register 1 B	OCR1B
		Input Capture Register	ICR1L/H
Watchdog Timer	WDT	Watchdog Timer Control Register	WDTCR
EEPROM	EEPROM	EEPROM Adress Register	EEAR
		EEPROM Data Register	EEDR
		EEPROM Control Register	EECR
SPI	SPI	Serial Peripheral Control Register	SPCR
		Serial Peripheral Status Register	SPSR
		Serial Peripheral Data Register	SPDR
UART	UART	UART Data Register	UDR
		UART Status Register	USR
		UART Control Register	UCR
		UART Baud Rate Register	UBRR
Analog Comparator	ANALOG	Analog Comparator Control and Status Register	ACSR
I/O-Ports, A/B/C/D/...	IO-Ports	Portpin-Ausgabe/-Richtung/-Eingabe	PORTA DDRA PINA

7.10 Das Statusregister als wichtigster Port

Der am häufigste verwendete Port für den Assemblerprogrammierer ist das Statusregister mit den darin enthaltenen acht Bits. In der Regel wird auf diese Bits vom Programm aus nur lesend/auswertend zugegriffen, selten werden Bits explizit gesetzt (mit der Assembler-Instruktion SEx) oder zurückgesetzt (mit der Instruktion CLx). Die meisten Statusbits werden von Bit-Test, Vergleichs- und Rechenoperationen gesetzt

oder rückgesetzt und anschließend für Entscheidungen und Verzweigungen im Programm verwendet.

Die folgende Tabelle enthält eine Liste der Assembler-Instruktionen, die die jeweiligen Status-Bits beeinflussen.

Bit	Rechnen	Logik	Vergleich	Bits	Schieben	Sonstige
Z	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR Z, BSET Z, CLZ, SEZ, TST	ASR, LSL, LSR, ROL, ROR	CLR
C	ADD, ADC, ADIW, SUB, SUBI, SBC, SBCI, SBIW	COM, NEG	CP, CPC, CPI	BCLR C, BSET C, CLC, SEC	ASR, LSL, LSR, ROL, ROR	-
N	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR N, BSET N, CLN, SEN, TST	ASR, LSL, LSR, ROL, ROR	CLR
V	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR V, BSET V, CLV, SEV, TST	ASR, LSL, LSR, ROL, ROR	CLR
S	SBIW	-	-	BCLR S, BSET S, CLS, SES	-	-
H	ADD, ADC, SUB, SUBI, SBC, SBCI	NEG	CP, CPC, CPI	BCLR H, BSET H, CLH, SEH	-	-
T	-	-	-	BCLR T, BSET T, BST, CLT, SET	-	-
I	-	-	-	BCLR I, BSET I, CLI, SEI	-	RETI

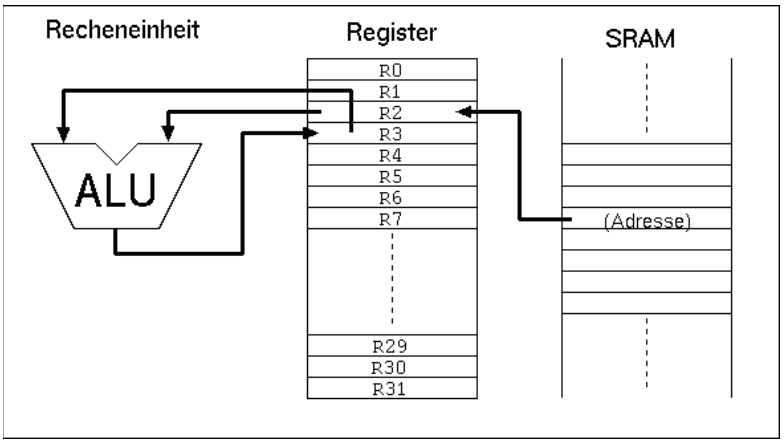
8 SRAM

8.1 Verwendung von SRAM in AVR Assembler

Alle AT90S-AVR-Typen verfügen in gewissem Umfang über statisches RAM (SRAM) an Bord. Bei sehr einfachen Assemblerprogrammen kann man es sich im allgemeinen leisten, auf die Verwendung dieser Hardware zu verzichten und alles in Registern unterzubringen. Wenn es aber eng wird im Registerbereich, dann sollte man die folgenden Kenntnisse haben, um einen Ausweg aus der Speicherenge zu nehmen.

8.2 Was ist SRAM?

SRAM sind Speicherstellen, die im Gegensatz zu Registern nicht direkt in die Recheneinheit (Arithmetic and Logical Unit ALU, manchmal aus historischen Gründen auch Akkumulator genannt) geladen und verarbeitet werden können. Ihre Verwendung ist daher auf den Umweg über ein Register angewiesen. Im dargestellten Beispiel wird ein Wert von der Adresse im SRAM in das Register R2 geholt (1.Instruktion), irgendwie mit dem Inhalt von Register R3 verknüpft und das Ergebnis in Register R3 gespeichert (Instruktion 2). Im letzten Schritt kann der geänderte Wert auch wieder in das SRAM geschrieben werden (3.Instruktion).



Es ist daher klar, dass SRAM-Daten langsamer zu verarbeiten sind als Daten in Registern. Dafür besitzt schon der zweitkleinste AVR immerhin 128 Bytes an SRAM-Speicher. Da passt schon einiges mehr rein als in 32 popelige Register.

Die größeren AVR ab AT90S8515 aufwärts bieten neben den eingebauten 512 Bytes zusätzlich die Möglichkeit, noch externes SRAM anzuschließen. Die Ansteuerung in Assembler erfolgt dabei in identischer Weise wie internes RAM. Allerdings belegt das externe SRAM eine Vielzahl von Portpins für Adress- und Datenleitungen und hebt den Vorteil der internen Bus-Gestaltung bei den AVR wieder auf.

8.3 Wozu kann man SRAM verwenden?

SRAM bietet über das reine Speichern von Bytes an festen Speicherplätzen noch ein wenig mehr. Der Zugriff kann nicht nur mit festen Adressen, sondern auch mit Zeigervariablen erfolgen, so dass eine fließende Adressierung der Zellen möglich ist. So können z.B. Ringpuffer zur Zwischenspeicherung oder berechnete Tabellen verwendet werden. Das geht mit Registern nicht, weil die immer eine fest angegebene Adresse benötigen.

Noch relativer ist die Speicherung über einen Offset. Dabei steht die Adresse in einem Pointerregister, es wird aber noch ein konstanter Wert zu dieser Adresse addiert und dann erst gespeichert oder gelesen. Damit lassen sich Tabellen noch raffinierter verwenden.

Die wichtigste Anwendung für SRAM ist aber der sogenannte Stack oder Stapel, auf dem man Werte zeitweise ablegen kann, seien es Rücksprungadressen beim Aufruf von Unterprogrammen, bei der Unterbre-

chung des Programmablaufes mittels Interrupt oder irgendwelche Zwischenwerte, die man später wieder braucht und für die ein extra Register zu schade ist.

8.4 Wie verwendet man SRAM?

Schreiben und Lesen von Speicherzellen

Um einen Wert in eine Speicherstelle im SRAM abzulegen, muss man seine Adresse festlegen. Das verwendbare SRAM reicht von Adresse 0x0060 bis zum jeweiligen Ende des SRAM-Speichers (beim AT90S8515 ist das ohne externes SRAM z.B. 0x025F). Mit der Instruktion

```
STS 0x0060, R1
```

wird der Inhalt des Registers R1 in die Speicherzelle im SRAM kopiert. Mit

```
LDS R1, 0x0060
```

wird vom SRAM in das Register kopiert. Das ist der direkte Weg mit einer festen Adresse, die vom Programmierer festgelegt wird.

Um das Hantieren mit festen Adressen und deren möglicherweisen späteren Veränderung bei fortgeschrittener Programmierkunst sowie das Merken der Adresse zu erleichtern empfiehlt der erfahrene Programmierer wieder die Namensvergabe, wie im folgenden Beispiel:

```
.EQU MeineLieblingsSpeicherzelle = 0x0060
        STS MeineLieblingsSpeicherzelle, R1
```

Aber auch das ist noch nicht allgemein genug. Mit

```
.EQU MeineLieblingsSpeicherzelle = SRAM_START
.EQU MeineZweiteLieblingsSpeicherzelle = SRAM_START + 1
```

ist die in der Include-Datei eingetragene Adresse der SRAM-Speicherzellen noch allgemeingültiger angegeben.

Zugegeben, kürzer ist das alles nicht, aber viel leichter zu merken.

Organisation als Datensegment

Bei etwas komplexeren Datenstrukturen empfiehlt sich das Anlegen in einem Datensegment. Eine solche Struktur sieht dann z. B. so aus:

```
.DSEG ; das ist der Beginn des Datensegments, die folgenden Einträge organisieren SRAM
.ORG SRAM_START ; an den Beginn des SRAM legen.
;
EinByte: ; ein Label als Symbol für die Adresse einer Speicherzelle
.BYTE 1 ; ein Byte dafuer reservieren
;
ZweiBytes: ; ein Label als Symbol für die Adresse zweier aufeinander folgender Speicherzellen
.BYTE 2 ; zwei Bytes reservieren
;
```

.EQU Pufferlaenge = 32 ; definiert die Laenge eines Datenpuffers
Buffer_Start: ; ein Label fuer den Anfang des Datenpuffers
.BYTE Pufferlaenge ; die folgenden 32 Speicherzellen als Datenpuffer reservieren
Buffer_End: ; ein Label fuer das Ende des Datenpuffers
;
.CSEG ; Ende des Datensegments, Beginn des Programmsegments

Das Datensegment enthält also ausschließlich Labels, über die die entsprechenden reservierten Speicherzellen später adressiert werden können, und .BYTE-Direktiven, die die Anzahl der zu reservierenden Zellen angeben. Ziel der ganzen Angelegenheit ist das Anlegen einer flexibel änderbaren Struktur für den Zugriff über Adresssymbole. Inhalte für diese Speicherzellen werden dabei nicht erzeugt, es gibt auch keine Möglichkeit, den Inhalt von SRAM-Speicherzellen beim Brennen des Chips zu manipulieren.

Zugriffe auf das SRAM über Pointer

Eine weitere Adressierungsart für SRAM-Zugriffe ist die Verwendung von Pointern, auch Zeiger genannt. Dazu braucht es zwei Register, die die 16-Bit-Adresse enthalten. Wie bereits in der Pointer-Register-Abteilung erläutert sind das die Registerpaare X mit XL/XH (R26, R27), Y mit YL/YH (R28, R29) und Z mit ZL und ZH (R30, R31). Sie erlauben den Zugriff auf die jeweils adressierte Speicherzelle direkt (z.B. ST X, R1), nach vorherigem Vermindern der Adresse um Eins (z.B. ST -X,R1) oder mit anschließendem Erhöhen um Eins (z.B. ST X+, R1). Ein vollständiger Zugriff auf drei Zellen sieht also etwa so aus:

```
.EQU MeineLieblingsZelle = 0x0060
.DEF MeinLieblingsRegister = R1
.DEF NochEinRegister = R2
.DEF UndNochEinRegister = R3
        LDI  XH, HIGH(MeineLieblingszelle)
        LDI  XL, LOW(MeineLieblingszelle)
        LD MeinLieblingsregister, X+
        LD NochEinRegister, X+
        LD UndNochEinRegister, X
```

Sehr einfach zu bedienen, diese Pointer. Und nach meinem Dafürhalten genauso einfach (oder schwer) zu verstehen wie die Konstruktion mit dem Dach in gewissen Hochsprachen.

Indizierte Zugriffe über Pointer

Die dritte Konstruktion ist etwas exotischer und nur erfahrene Programmierer greifen in ihrer unermesslichen Not danach. Nehmen wir mal an, wir müssen sehr oft und an verschiedenen Stellen eines Programmes auf die drei Positionen im SRAM zugreifen, weil dort irgendwelche wertvollen Informationen stehen. Nehmen wir ferner an, wir hätten gerade eines der Pointer-Register so frei, dass wir es dauerhaft für diesen Zweck opfern könnten. Dann bleibt bei dem Zugriff nach ST/LD-Muster immer noch das Problem, dass wir das Pointer-Register immer anpassen und nach dem Zugriff wieder in einen definierten Zustand versetzen müssten. Das ist eklig.

Zur Vermeidung (und zur Verwirrung von Anfängern) hat man sich den Zugriff mit Offset einfallen lassen

(deutsch etwa: Ablage). Bei diesem Zugriff wird das eigentliche Zeiger-Register nicht verändert, der Zugriff erfolgt mit temporärer Addition eines festen Wertes. Im obigen Beispiel würde also folgende Konstruktion beim Zugriff auf die Speicherzelle 0x0062 erfolgen. Zuerst wäre irgendwann das Pointer-Register zu setzen:

```
.EQU MeineLieblingsZelle = 0x0060

.DEF MeinLieblingsRegister = R1

    LDI YH, HIGH(MeineLieblingszelle)

    LDI YL, LOW(MeineLieblingszelle)
```

Irgendwo später im Programm will ich dann auf Zelle 0x0062 zugreifen:

```
    STD Y+2, MeinLieblingsRegister
```

Obacht! Die „plus zwei“ werden nur für den Zugriff addiert, der Registerinhalt von Y wird dabei nicht verändert. Zur weiteren Verwirrung des Publikums geht diese Konstruktion nur mit dem Y- und dem Z-Pointer, nicht aber mit dem X-Pointer!

Die korrespondierende Instruktion für das indizierte Lesen eines SRAM-Bytes

```
    LDD MeinLieblingsRegister, Y+2
```

ist ebenfalls vorhanden.

Das war es schon mit dem SRAM, wenn da nicht der Stack noch wäre.

8.5 Verwendung von SRAM als Stack

Die häufigste und bequemste Art der Nutzung des SRAM ist der Stapel, englisch *stack* genannt. Der Stapel ist ein Türmchen aus Holzklötzen. Jedes neu aufgelegte Klötzchen kommt auf den schon vorhandenen Stapel obenauf, jede Entnahme vom Turm kann immer nur auf das jeweils oberste Klötzchen zugreifen, weil sonst der ganze schöne Stapel hin wäre. Das kluge Wort für diese Struktur ist *Last-In-First-Out (LIFO)* oder schlichter: *die Letzten werden die Ersten sein*.

Zur Verwirrung des Publikums wächst der Stapel bei fast allen Mikroprozessoren aber nicht von der niedrigsten zu höheren Adressen hin, sondern genau umgekehrt. Wir könnten sonst am Anfang des SRAMs unsere schönen Datenstrukturen nicht anlegen.

Einrichten des Stapels

Um vorhandenes SRAM für die Anwendung als Stapel herzurichten ist zu allererst der Stapelzeiger einzurichten. Der Stapelzeiger ist ein 16-Bit-Zeiger, der als Port ansprechbar ist. Das Doppelregister heißt SPH:SPL. SPH nimmt das obere Byte der Adresse, SPL das niederwertige Byte auf. Das gilt aber nur dann, wenn der Chip über mehr als 256 Byte SRAM verfügt. Andernfalls fehlt SPH und kann/muss nicht verwendet werden. Wir tun im nächsten Beispiel so, als ob wir mehr als 256 Bytes SRAM haben.

Zum Einrichten des Stapels wird der Stapelzeiger mit der höchsten verfügbaren SRAM-Adresse bestückt.

```
.DEF MeinLieblingsRegister = R16

    LDI MeinLieblingsRegister, HIGH(RAMEND) ; Oberes Byte
```

OUT SPH,MeinLieblingsRegister ; an Stapelzeiger

LDI MeinLieblingsRegister, LOW(RAMEND) ; Unteres Byte

OUT SPL,MeinLieblingsRegister ; an Stapelzeiger

Die Größe RAMEND ist natürlich prozessorspezifisch und steht in der Include-Datei für den Prozessor. So steht z.B. in der Datei 8515def.inc die Zeile

.equ RAMEND =$25F ;Last On-Chip SRAM Location

Die Datei 8515def.inc kommt wieder mit der Assembler-Direktive

.INCLUDE "8515def.inc"

irgendwo am Anfang des Assemblerprogrammes hinzu.

Damit ist der Stapelzeiger eingerichtet und wir brauchen uns im weiteren nicht mehr weiter um diesen Zeiger kümmern, weil er ziemlich automatisch manipuliert wird.

Verwendung des Stapels

Die Verwendung des Stapels ist unproblematisch. So lassen sich Werte von Registern auf den Stapel legen:

PUSH MeinLieblingsregister ; Ablegen des Wertes

Wo der Registerinhalt abgelegt wird, interessiert uns nicht weiter. Dass dabei der Zeiger automatisch erniedrigt wird, interessiert uns auch nicht weiter. Wenn wir den abgelegten Wert wieder brauchen, geht das einfach mit:

POP MeinLieblingsregister ; Rücklesen des Wertes

Mit POP kriegen wir natürlich immer nur den Wert, der als letztes mit PUSH auf den Stapel abgelegt wurde. Wichtig: Selbst wenn der Wert vielleicht gar nicht mehr benötigt wird, muss er mit Pop wieder vom Stapel! Das Ablegen des Registers auf den Stapel lohnt also programm-technisch immer nur dann, wenn

- der Wert in Kürze, d.h. ein paar Instruktionen weiter im Ablauf, wieder gebraucht wird,
- alle Register in Benutzung sind und,
- keine Möglichkeit zur Zwischenspeicherung woanders besteht.

Wenn diese Bedingungen nicht vorliegen, dann ist die Verwendung des Stapels ziemlich nutzlos und verschwendet bloß Zeit.

Stapel zum Ablegen von Rücksprungadressen

Noch wertvoller ist der Stapel bei Sprüngen in Unterprogramme, nach deren Abarbeitung wieder exakt an die aufrufende Stelle im Programm zurück gesprungen werden soll. Dann wird beim Aufruf des Unterprogrammes die Rücksprungadresse auf den Stapel abgelegt, nach Beendigung wieder vom Stapel geholt und in den Programmzähler bugsiert. Dazu dient die Konstruktion mit der Instruktion

RCALL irgendwas ; Springe in das UP irgendwas

[...] hier geht es normal weiter im Programm

Hier landet der Sprung zum Label irgendwas irgendwo im Programm,

irgendwas: ; das hier ist das Sprungziel

[...] Hier wird zwischendurch irgendwas getan

[...] und jetzt kommt der Rücksprung an den Aufrufort im Programm:

RET

Beim RCALL wird der Programmzähler, eine 16-Bit-Adresse, auf dem Stapel abgelegt. Das sind zwei mal PUSH, dann sind die 16 Bits auf dem Stapel. Beim Erreichen der Instruktion RET wird der Programmzähler mit zwei POPs wieder hergestellt und die Ausführung des Programmes geht an der Stelle weiter, die auf den RCALL folgt.

Damit braucht man sich weiter um die Adresse keine Sorgen zu machen, an der der Programmzähler abgelegt wurde, weil der Stapel automatisch manipuliert wird. Selbst das vielfache Verschachteln solcher Aufrufe ist möglich, weil jedes Unterprogramm, das von einem Unterprogramm aufgerufen wurde, zuoberst auf dem Stapel die richtige Rücksprungadresse findet.

Unverzichtbar ist der Stapel bei der Verwendung von Interrupts. Das sind Unterbrechungen des Programmes aufgrund von äußeren Ereignissen, z.B. Signale von der Hardware. Damit nach Bearbeitung dieser äußeren "Störung" der Programmablauf wieder an der Stelle vor der Unterbrechung fortgesetzt werden kann, muss die Rücksprungadresse bei der Unterbrechung auf den Stapel. Interrupts ohne Stapel sind also schlicht nicht möglich.

Fehlermöglichkeiten beim (Hoch-)Stapeln

Für den Anfang gibt es reichlich Möglichkeiten, mit dem Stapeln üble Bugs zu produzieren.

Sehr beliebt ist die Verwendung des Stapels ohne vorheriges Setzen des Stapelzeigers. Da der Zeiger zu Beginn bei Null steht, klappt aber auch rein gar nix, wenn man den ersten Schritt vergisst.

Beliebt ist auch, irgendwelche Werte auf dem Stapel liegen zu lassen, weil die Anzahl der POPs nicht exakt der Anzahl der PUSHs entspricht. Das ist aber schon seltener. Es kommt vorzugsweise dann vor, wenn zwischendurch ein bedingter Sprung nach woanders vollführt wurde und dort beim Programmieren vergessen wird, dass der Stapel noch was in Petto hat.

Noch seltener ist ein Überlaufen des Stapels, wenn zuviele Werte abgelegt werden und der Stapelzeiger sich bedrohlich auf andere, am Anfang des SRAM abgelegten Werte zubewegt oder noch niedriger wird und in den Bereich der Ports und der Register gerät. Das hat ein lustiges Verhalten des Chips, auch äußerlich, zur Folge. Kommt aber meistens fast nie vor, nur bei vollgestopftem SRAM.

9 Steuerung des Programmablaufes in AVR Assembler

Hier werden alle Vorgänge erläutert, die mit dem Programmablauf zu tun haben und diesen beeinflussen, also die Vorgänge beim Starten des Prozessors, Sprünge und Verzweigungen, Unterbrechungen, etc.

9.1 Was passiert beim Reset?

Beim Anlegen der Betriebsspannung, also beim Start des Prozessors, wird über die Hardware des Prozessors ein sogenannter Reset ausgelöst. Dabei wird der Zähler für die Programmschritte auf Null gesetzt. An dieser Stelle des Programmes wird die Verarbeitung also immer begonnen. Ab hier muss der Programm-Code stehen, der ausgeführt werden soll. Aber nicht nur beim Start wird der Zähler auf diese Adresse zurückgesetzt, sondern auch bei

- externem Rücksetzen am Eingangs-Pin Reset durch die Hardware,
- Ablauf der Wachhund-Zeit (Watchdog-Reset), einer internen Überwachungsuhr,
- direkten Sprüngen an die Adresse Null (Sprünge siehe unten).

Die dritte Möglichkeit ist aber gar kein richtiger Reset, denn das beim Reset automatisch ablaufende Rücksetzen von Register- und Port-Werten auf den jeweils definierten Standardwert (Default) wird hierbei nicht durchgeführt. Vergessen wir also besser die 3.Möglichkeit, sie ist unzuverlässig.

Die zweite Möglichkeit, nämlich der eingebaute Wachhund, muss erst explizit von der Leine gelassen werden, durch Setzen seiner entsprechenden Port-Bits. Wenn er dann nicht gelegentlich mit Hilfe der Instruktion

```
WDR ; Watchdog Reset
```

zurückgepfiffen wird, dann geht er davon aus, dass Herrchen AVR eingeschlafen ist und weckt ihn mit einem brutalen Reset.

Ab dem Reset wird also der an Adresse 0x000000 stehende Code wortweise in die Ausführungsmimik des Prozessors geladen und ausgeführt. Während der Ausführung wird die Adresse um 1 erhöht und schon mal die nächste Instruktion aus dem Programmspeicher geholt (Fetch during Execution). Wenn die erste Instruktion keine Verzweigung des Programmes auslöst, kann die zweite Instruktion also direkt nach der ersten ausgeführt werden. Jeder Taktzyklus am Takteingang des Prozessors entspricht daher einem ausgeführten Instruktion (wenn nichts dazwischenkommt).

Die erste Instruktion des ausführbaren Programmes muss immer bei Adresse 0000 stehen. Um dem Assembler mitzuteilen, dass er nach irgendwelchen Vorwörtern wie Definitionen oder Konstanten nun bitte mit der ersten Zeile Programmcode beginnen möge, gibt es folgende Direktiven:

```
.CSEG
.ORG 0000
```

Die erste Direktive teilt dem Assembler mit, dass ab jetzt in das Code-Segment zu assemblieren ist. Ein anderes Segment wäre z.B. das EEPROM, das ebenfalls im Assembler-Quelltext auf bestimmte Werte eingestellt werden könnte. Die entsprechende Assembler-Direktive hierfür würde dann natürlich lauten:

```
.ESEG
```

Die zweite Assembler-Direktive oben stellt den Programmzähler beim Assemblieren auf die Adresse 0000 ein. ORG ist die Abkürzung für Origin, also Ursprung. Wir könnten auch bei 0100 mit dem Programm beginnen, aber das wäre ziemlich sinnlos (siehe oben). Da die beiden Angaben CSEG und ORG trivial sind,

können sie auch weggelassen werden und der Assembler macht das dann automatisch so. Wer aber den Beginn des Codes nach zwei Seiten Konstantendefinitionen eindeutig markieren will, kann das mit den beiden Direktiven tun.

Auf das ominöse erste Wort Programmcode folgen fast ebenso wichtige Spezialinstruktionen, die Interrupt-Vektoren. Dies sind festgelegte Stellen mit bestimmten Adressen. Diese Adressen werden bei Auslösung eines Interrupts durch die Hardware angesprungen, wobei der normale Programmablauf unterbrochen wird. Diese Vektoren sind spezifisch für jeden Prozessortyp (siehe unten bei der Erläuterung der Interrupts). Die Instruktionen, mit denen auf eine Unterbrechung reagiert werden soll, müssen an dieser Stelle stehen. Werden also Interrupts verwendet, dann kann die erste Instruktion nur eine Sprunginstruktion sein, damit diese ominösen Vektoren übersprungen werden. Der typische Programmablauf nach dem Reset sieht also so aus:

```
.CSEG

.ORG 0000

        RJMP Start

        [...] hier kommen dann die Interrupt-Vektoren

        [...] und danach nach Belieben noch alles mögliche andere Zeug

Start:  ; Das hier ist der Programmbeginn

        [...] Hier geht das Hauptprogramm dann erst richtig los.
```

Die Instruktion RJMP bewirkt einen Sprung an die Stelle im Programm mit der Kennzeichnung Start:, auch ein Label genannt. Die Stelle Start: folgt weiter unten im Programm und ist dem Assembler hiermit entsprechend mitgeteilt: Labels beginnen immer in Spalte 1 einer Zeile und enden mit einem Doppelpunkt. Labels, die diese beiden Bedingungen nicht einhalten, werden vom Assembler nicht ernstgenommen. Fehlende Labels aber lassen den Assembler sinnierend innehalten und mit einer "Undefined label"-Fehlermeldung die weitere Zusammenarbeit einstellen.

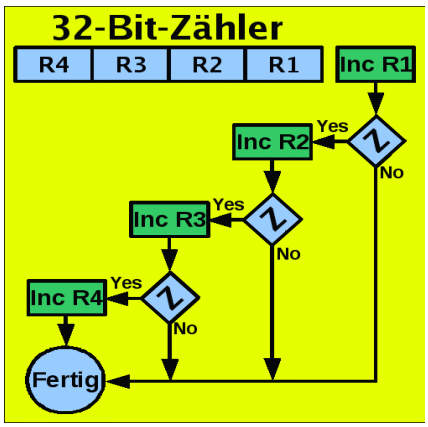
9.2 Linearer Programmablauf und Verzweigungen

Nachdem die ersten Schritte im Programm gemacht sind, noch etwas triviales: Programme laufen linear ab. Das heißt: Instruktion für Instruktion wird nacheinander aus dem Programmspeicher geholt und abgearbeitet. Dabei zählt der Programmzähler immer eins hoch. Ausnahmen von dieser Regel werden nur vom Programmierer durch gewollte Verzweigungen oder mittels Interrupt herbeigeführt. Da sind Prozessoren ganz stur immer geradeaus, es sei denn, sie werden gezwungen.

Und zwingen geht am einfachsten folgendermaßen. Oft kommt es vor, dass in Abhängigkeit von bestimmten Bedingungen gesprungen werden soll. Dann benötigen wir bedingte Verzweigungen. Nehmen wir an, wir wollen einen 32-Bit-Zähler mit den Registern R1, R2, R3 und R4 realisieren. Dann muss das niederwertigste Byte in R1 um Eins erhöht werden. Wenn es dabei überläuft, muss auch R2 um Eins erhöht werden usw. bis R4.

Die Erhöhung um Eins wird mit der Instruktion INC erledigt. Wenn dabei ein Überlauf auftritt, also 255 zu 0 wird, dann ist das im Anschluss an die Durchführung am gesetzten Z-Bit im Statusregister zu bemerken. Das eigentliche Übertragsbit C des Statusregisters wird bei der INC-Instruktion übrigens nicht verändert, weil es woanders dringender gebraucht wird und das Z-Bit völlig ausreicht. Wenn der Übertrag nicht auftritt, also das Z-Bit nicht gesetzt ist, können wir die Erhöherei beenden. Wenn aber das Z-Bit gesetzt ist, darf die Erhöherei beim nächsten Register weitergehen. Wir müssen also springen, wenn das Z-Bit nicht

gesetzt ist. Die entsprechende Sprunginstruktion heißt aber nicht BRNZ (BRanch on Not Zero), sondern BRNE (BRanch if Not Equal). Na ja, Geschmackssache. Das ganze Räderwerk des 32-Bit langen Zählers sieht damit so aus:




```
    INC R1
    BRNE Weiter
    INC R2
    BRNE Weiter
    INC R3
    BRNE Weiter
    INC R4
Weiter:
```

Das war es schon. Eine einfache Sache. Das Gegenteil von BRNE ist übrigens BREQ oder BRanch EQual.

Welche der Statusbits durch welche Instruktionen und Bedingungen gesetzt oder rückgesetzt werden (auch Prozessorflags genannt), geht aus den einzelnen Instruktionsbeschreibungen in der Instruktionsliste hervor. Entsprechend kann mit den bedingten Sprunginstruktionen

BRCC/BRCS ; Carry-Flag 0 oder gesetzt

BRSH ; Gleich oder größer

BRLO ; Kleiner

BRMI ; Minus

BRPL ; Plus

BRGE ; Größer oder gleich (mit Vorzeichen)

BRLT ; Kleiner (mit Vorzeichen)

BRHC/BRHS ; Halbübertrag 0 oder 1

BRTC/BRTS ; T-Bit 0 oder 1

BRVC/BRVS ; Zweierkomplement-Übertrag 0 oder 1

BRIE/BRID ; Interrupt an- oder abgeschaltet

auf die verschiedenen Bedingungen reagiert werden. Gesprungen wird immer dann, wenn die entsprechende Bedingung erfüllt ist. Keine Angst, die meisten dieser Instruktionen braucht man sehr selten. Nur Zero und Carry sind für den Anfang wichtig.

9.3 Zeitzusammenhänge beim Programmablauf

Wie oben schon erwähnt entspricht die Zeitdauer zur Bearbeitung einer Instruktion in der Regel exakt einem Prozessortakt. Läuft der Prozessor mit 4 MHz Takt, dann dauert die Bearbeitung einer Instruktion 1/4 µs oder 250 ns, bei 10 MHz Takt nur 100 ns. Die Dauer ist quarzgenau vorhersagbar und Anwendungen, die ein genaues Timing erfordern, sind durch entsprechend exakte Gestaltung des Programmablaufes erreichbar. Es gibt aber eine Reihe von Instruktionen, z.B. die Sprunginstruktionen oder die Lese- und Schreibinstruktionen für das SRAM, die zwei oder mehr Taktzyklen erfordern. Hier hilft nur die Instruktionstabelle weiter.

Um genaue Zeitbeziehungen herzustellen, muss man manchmal den Programmablauf um eine bestimmte Anzahl Taktzyklen verzögern. Dazu gibt es die sinnloseste Instruktion des Prozessors, die Tu-nix-Instruktion:

```
NOP
```

Diese Instruktion heißt "No Operation" und tut nichts außer Prozessorzeit zu verschwenden. Bei 4 MHz Takt braucht es genau vier solcher NOPs, um eine exakte Verzögerung um 1 µs zu erreichen. Zu viel anderem ist diese Instruktion nicht zu gebrauchen. Für einen Rechteckgenerator von 1 kHz müssen aber nicht 1000 solcher NOPs kopiert werden, das geht auch anders! Dazu braucht es die Sprunginstruktionen. Mit ihrer Hilfe können Schleifen programmiert werden, die für eine festgelegte Anzahl von Läufen den Programmablauf aufhalten und verzögern. Das können 8-Bit-Register sein, die mit der DEC-Instruktion heruntergezählt werden, wie z.B. bei

```
    CLR R1
Zaehl:
    DEC R1
    BRNE Zaehl
```

Es können aber auch 16-Bit-Zähler sein, wie z.B. bei

```
    LDI ZH,HIGH(65535)
    LDI ZL,LOW(65535)
Zaehl:
    SBIW ZL,1
    BRNE Zaehl
```

Mit mehr Registern lassen sich mehr Verzögerungen erreichen. Jeder dieser Verzögerer kann auf den jeweiligen Bedarf eingestellt werden und funktioniert dann quarzgenau, auch ohne Hardware-Timer. Ein späteres Kapitel gibt mehr Beispiele für Zeitschleifen und zeigt, wie man sie berechnet.

9.4 Makros im Programmablauf

Es kommt vor, dass in einem Programm identische oder ähnliche Code-Sequenzen an mehreren Stellen benötigt werden. Will oder kann man den Programmteil nicht mittels Unterprogrammen bewältigen, dann kommt für diese Aufgabe auch ein Makro in Frage. Makros sind Codesequenzen, die nur einmal entworfen werden und durch Aufruf des Makronamens in den Programmablauf eingefügt werden. Anstelle des Ma-

kronamens setzt der Assembler dann die Sequenz im Makro.

Nehmen wir an, es soll die folgende Codesequenz (Verzögerung um 1 µs bei 4 MHz Takt) an mehreren Programmstellen benötigt werden. Dann erfolgt irgendwo im Quellcode die Definition des folgenden Makros:

```
.MACRO Delay1
        NOP
        NOP
        NOP
        NOP
.ENDMACRO
```

Diese Definition des Makros erzeugt keinen Code, es werden also keine vier NOPs in den Code eingefügt. Erst der Aufruf des Makros

```
[...] irgendwo im Code
        Delay1
[...] weiter mit Code
```

bewirkt, dass an dieser Stelle vier NOPs eingebaut werden. Der zweimalige Aufruf des Makros baut acht NOPs in den Code ein.

Handelt es sich um größere Codesequenzen, hat man etwas Zeit im Programm oder ist man knapp mit Programmspeicher, sollte man auf den Code-extensiven Einsatz von Makros verzichten und lieber Unterprogramme verwenden.

9.5 Unterprogramme

Im Gegensatz zum Makro geht ein Unterprogramm sparsamer mit dem Programmspeicher um. Irgendwo im Code steht die Sequenz ein einziges Mal herum und kann von verschiedenen Programmteilen her aufgerufen werden. Damit die Verarbeitung des Programmes wieder an der aufrufenden Stelle weitergeht, folgt am Ende des Unterprogrammes ein Return. Das sieht dann für 10 Takte Verzögerung so aus:

```
Delay10:
        NOP
        NOP
        NOP
        RET
```

Unterprogramme beginnen immer mit einem Label, hier Delay10:, weil sie sonst nicht angesprungen werden könnten. Es folgen drei Nix-Tun-Instruktionen und der abschließenden Return-Instruktion. Schlaumeier könnten jetzt einwenden, das seien ja bloß 7 Takte (RET braucht 4) und keine 10. Gemach, es kömmt noch der Aufruf dazu und der schlägt mit 3 Takten zu Buche:

```
[...] irgendwo im Programm:
```

RCALL Delay10

[...] weiter im Programm

RCALL ist eine Verzweigung zu einer relativen Adresse, nämlich zum Unterprogramm. Mit RET wird wieder der lineare Programmablauf abgebrochen und zu der Instruktion nach dem RCALL verzweigt. Es sei noch dringend daran erinnert, dass die Verwendung von Unterprogrammen das vorherige Setzen des Stackpointers oder Stapelzeigers voraussetzt (siehe Stapel), weil die Rücksprungadresse beim RCALL auf dem Stapel abgelegt wird.

Wer das obige Beispiel ohne Stapel programmieren möchte, verwendet die absolute Sprunginstruktion:

[...] irgendwo im Programm

RJMP Delay10

Zurueck:

[...] weiter im Programm

Jetzt muss das "Unterprogramm" allerdings anstelle des RET natürlich ebenfalls eine RJMP-Instruktion bekommen und zum Label Zurueck: verzweigen. Da der Programmcode dann aber nicht mehr von anderen Stellen im Programmcode aufgerufen werden kann (die Rückkehr funktioniert jetzt nicht mehr), ist die Springerei ziemlich sinnlos. Auf jeden Fall haben wir jetzt auch das relative Springen mit RJMP gelernt.

RCALL und RJMP sind auf jeden Fall unbedingte Verzweigungen. Ob sie ausgeführt werden hängt nicht von irgendwelchen weiteren Bedingungen ab. Zum bedingten Ausführen eines Unterprogrammes muss das Unterprogramm mit einer bedingten Verzweigungsinstruktion kombiniert werden, der unter bestimmten Bedingungen das Unterprogramm ausführt oder den Aufruf eben überspringt. Für solche bedingten Verzweigungen zu einem Unterprogramm eignen sich die beiden folgenden Instruktionen ideal. Soll in Abhängigkeit vom Zustand eines Bits in einem Register zu einem Unterprogramm oder einem anderen Programmteil verzweigt werden, dann geht das so:

SBRC R1,7 ; Überspringe die folgende Instruktion wenn Bit 7 im Register R1 Null ist

RCALL UpLabel ; Rufe Unterprogramm auf

Hier wird der RCALL zum Unterprogramm UpLabel: nur ausgeführt, wenn das Bit 7 im Register R1 eine Eins ist, weil die Instruktion bei einer Null (Clear) übersprungen wird. Will man auf die umgekehrte Bedingung hin ausführen, so kommt statt SBRC die analoge Instruktion SBRS zum Einsatz. Die auf SBRC/ SBRS folgende Instruktion kann sowohl eine Ein-Wort- als auch eine Zwei-Wort-Instruktion sein, der Prozessor weiß über die Länge der Instruktionen korrekt Bescheid und überspringt dann eben um die richtige Anzahl Instruktionsworte. Zum Überspringen von mehr als einer Instruktion kann dieser bedingte Sprung natürlich nicht benutzt werden.

Soll ein Überspringen nur dann erfolgen, wenn zwei Registerinhalte gleich sind, dann bietet sich die etwas exotische Instruktion

CPSE R1,R2 ; Vergleiche R1 und R2, springe wenn gleich

RCALL UpIrgendwas ; Rufe irgendwas

Der wird selten gebraucht und ist ein Mauerblümchen.

Man kann aber auch in Abhängigkeit von bestimmten Port-Bits die nächste Instruktion überspringen. Die entsprechenden Instruktionen heißen SBIC und SBIS, also etwa so:

```
    SBIC PINB,0 ; Überspringe wenn Bit 0 Null ist
    RJMP EinZiel ; Springe zum Label EinZiel
```

Hier wird die RJMP-Instruktion nur übersprungen, wenn das Bit 0 im Eingabeport PINB gerade Null ist. Also wird das Programm an dieser Stelle nur dann zum Programmteil EinZiel: verzweigen, wenn das Portbit 0 Eins ist. Das ist etwas verwirrend, da die Kombination von SBIC und RJMP etwas umgekehrt wirkt als sprachlich naheliegend ist. Das umgekehrte Sprungverhalten kann anstelle von SBIC mit SBIS erreicht werden. Leider kommen als Ports bei den beiden Instruktionen nur die unteren 32 infrage, für die oberen 32 Ports können diese Instruktionen nicht verwendet werden.

Nun noch die Exotenanwendung für den absoluten Spezialisten. Nehmen wir mal an, sie hätten vier Eingänge am AVR, an denen ein Bitklavier angeschlossen sei (vier kleine Schalterchen). Je nachdem, welche der vier Schalter eingestellt sei, soll der AVR 16 verschiedene Tätigkeiten vollführen. Nun könnten Sie selbstverständlich den Porteingang lesen und mit jede Menge Branch-Anweisungen schon herausfinden, welches der 16 Programmteile denn heute gewünscht wird. Sie können aber auch eine Tabelle mit je einer Sprunginstruktion auf die sechzehn Routinen machen, etwa so:

```
MeinTab:
    RJMP Routine1
    RJMP Routine2
    [...]
    RJMP Routine16
```

Jetzt laden Sie den Anfang der Tabelle in das Z-Register mit

```
    LDI  ZH,HIGH(MeinTab)
    LDI  ZL,LOW(MeinTab)
```

und addieren den heutigen Pegelstand am Portklavier in R16 zu dieser Adresse.

```
    IN R16,PINB ; Lese Portklavier von Port B
    ANDI R16,0x0F ; Loesche die oberen Bits
    ADD ZL,R16 ; Addiere Portklavierstand
    BRCC NixUeberlauf ; Kein Ueberlauf
    INC ZH ; Ueberlauf in MSB
NixUeberlauf:
```

Jetzt können Sie nach Herzenslust und voller Wucht in die Tabelle springen, entweder nur mal als Unterprogramm mit

```
    ICALL ; Rufe Unterprogramm auf, auf das ZH:ZL zeigt
```

oder auf Nimmerwiederkehr mit

```
    IJMP ; Springe zur Adresse, auf die ZH:ZL zeigt
```

Der Prozessor lädt daraufhin den Inhalt seines Z-Registerpaares in den Programmzähler und macht dort weiter. Manche finden das eleganter als unendlich verschachtelte bedingte Sprünge. Mag sein.

9.6 Interrupts im Programmablauf

Sehr oft kommt es vor, dass in Abhängigkeit von irgendwelchen Änderungen in der Hardware oder zu bestimmten Gelegenheiten auf dieses Ereignis reagiert wird. Ein Beispiel ist die Pegeländerung an einem Eingang. Man kann das lösen, indem das Programm im Kreis herum läuft und immer den Eingang befragt, ob er sich nun geändert hat. Die Methode heisst Polling, weil es wie bei die Bienchen darum geht, jede Blüte immer mal wieder zu besuchen und deren neue Pollen einzusammeln.

Gibt es außer diesem Eingang noch anderes wichtiges für den Prozessor zu tun, dann kann das dauernde zwischendurch Anfliegen der Blüte ganz schön nerven und sogar versagen: Kurze Pulse werden nicht erkannt, wenn Bienchen nicht gerade vorbei kam und nachsah, der Pegel aber schon wieder weg ist. Für diesen Fall hat man den Interrupt erfunden.

Der Interrupt ist eine von der Hardware ausgelöste Unterbrechung des Programmablaufes. Dazu kriegt das Gerät zunächst mitgeteilt, dass es unterbrechen darf. Dazu ist bei dem entsprechenden Gerät ein oder mehr Bits in bestimmten Ports zu setzen. Dem Prozessor ist durch ein gesetztes Interrupt Enable Bit in seinem Status-Register mitzuteilen, dass er unterbrochen werden darf. Irgendwann nach den Anfangsfeierlichkeiten des Programmes muss dazu das I-Flag im Statusregister gesetzt werden, sonst macht der Prozessor beim Unterbrechen nicht mit und der Interrupt verhungert.

```
SEI ; Setze Int-Enable
```

Wenn jetzt die Bedingung eintritt, also z.B. ein Pegel von Null auf Eins wechselt, dann legt der Prozessor seine aktuelle Programmadresse erst mal auf den Stapel ab (Obacht! Der muss vorher eingerichtet sein!). Er muss ja das unterbrochene Programm exakt an der Stelle wieder aufnehmen, an dem es unterbrochen wurde, dafür der Stapel. Nun springt der Prozessor an eine vordefinierte Stelle des Programmes und setzt die Verarbeitung erst mal dort fort. Die Stelle nennt man Interrupt Vektor. Sie ist prozessorspezifisch festgelegt. Da es viele Geräte gibt, die auf unterschiedliche Ereignisse reagieren können sollen, gibt es auch viele solcher Vektoren. So hat jede Unterbrechung ihre bestimmte Stelle im Programm, die angesprungen wird. Die entsprechenden Programmstellen einiger älterer Prozessoren sind in der folgenden Tabelle aufgelistet. (Der erste Vektor ist übrigens gar kein Int-Vektor, weil er keine Rücksprung-Adresse auf dem Stapel ablegt!)

Name	Int Vector Adresse			ausgelöst durch ...
	2313	2323	8515	
RESET	0000	0000	0000	Hardware Reset, Power-On Reset, Watchdog Reset
INT0	0001	0001	0001	Pegelwechsel am externen INT0-Pin
INT1	0002	-	0002	Pegelwechsel am externen INT1-Pin
TIMER1 CAPT	0003	-	0003	Fangschaltung am Timer 1
TIMER1 COMPA	-	-	0004	Timer1 = Compare A
TIMER1 COMPB	-	-	0005	Timer1 = Compare B
TIMER1 COMP1	0004	-	-	Timer1 = Compare 1

Name	Int Vector Adresse			ausgelöst durch ...
TIMER1 OVF	0005	-	0006	Timer1 Überlauf
TIMER0 OVF	0006	0002	0007	Timer0 Überlauf
SPI STC	-	-	0008	Serielle Übertragung komplett
UART RX	0007	-	0009	UART Zeichen im Empfangspuffer
UART UDRE	0008	-	000A	UART Senderegister leer
UART TX	0009	-	000B	UART Alles gesendet
ANA_COMP	-	-	000C	Analog Comparator

Man erkennt, dass die Fähigkeit, Interrupts auszulösen, sehr unterschiedlich ausgeprägt ist. Sie entspricht der Ausstattung der Chips mit Hardware. Die Adressen folgen aufeinander, sind aber für den Typ spezifisch durchnummeriert. Die Reihenfolge hat einen weiteren Sinn: Je höher in der Liste, desto wichtiger. Wenn also zwei Geräte gleichzeitig die Unterbrechung auslösen, gewinnt die jeweils obere in der Liste. Die niedrigerwertige kommt erst dran, wenn die höherwertige fertig bearbeitet ist. Damit das funktioniert, schaltet der erfolgreiche Interrupt das Interrupt-Status-Bit aus. Dann kann der niederwertige erst mal verhungern. Erst wenn das Interrupt-Status-Bit wieder angeschaltet wird, kann die nächste anstehende Unterbrechung ihren Lauf nehmen.

Für das Setzen des Statusbits kann die Unterbrechungsroutine, ein Unterprogramm, zwei Wege einschlagen. Es kann am Ende mit der Instruktion

```
RETI
```

abschließen. Diese Instruktion stellt den ursprünglichen Instruktionszähler wieder her, der vor der Unterbrechung gerade bearbeitet werden sollte und setzt das Interrupt-Status-Bit wieder auf Eins.

Der zweite Weg ist das Setzen des Status-Bits per Instruktion

```
SEI ; Setze Interrupt Enabled

RET ; Kehre zurück
```

und das Abschließen der Unterbrechungsroutine mit einem normalen RET.

Aber Obacht, das ist nicht identisch im Verhalten! Im zweiten Fall tritt die noch immer anstehende Unterbrechung schon ein, bevor die anschließende Return-Instruktion bearbeitet wird, weil das Status-Bit schon um eine Instruktion früher wieder Ints zulässt. Das führt zu einem Zustand, der das überragende Stapelkonzept so richtig hervorhebt: ein verschachtelter Interrupt. Die Unterbrechung während der Unterbrechung packt wieder die Rücksprung-Adresse auf den Stapel (jetzt liegen dort zwei Adressen herum), bearbeitet die Unterbrechungsroutine für den zweiten Interrupt und kehrt an die oberste Adresse auf dem Stapel zurück. Die zeigt auf die noch verbliebenen RET-Instruktion, der jetzt erst bearbeitet wird. Dieser nimmt nun auch die noch verbliebene Rücksprung-Adresse vom Stapel und kehrt zur Originaladresse zurück, die zur Zeit der Unterbrechung zur Bearbeitung gerade anstand.

Durch die LIFO-Stapelei ist das Verschachteln solcher Aufrufe über mehrere Ebenen kein Problem, solange der Stapelzeiger noch richtiges SRAM zum Ablegen vorfindet. Folgen allerdings zu viele Interrupts, bevor die vorherigen richtig beendet wurden, dann kann der Stapel auf dem SRAM überlaufen. Aber das muss man schon mit Absicht programmieren, weil es doch sehr selten vorkommt.

Klar ist auch, dass an der Adresse des Int-Vektors nur für eine Ein-Wort-Instruktion Platz ist. Das ist in der Regel eine RJMP-Instruktion an die Adresse des Interrupt- Unterprogrammes. Es kann auch einfach eine RETI-Instruktion sein, wenn dieser Vektor gar nicht benötigt wird. Bei sehr großen ATmega ist der Speicherraum mit der Ein-Wort-Instruktion RJMP nicht mehr ganz erreichbar. Bei diesen haben die Vektoren deshalb zwei Worte. Diese MÜSSEN mit JMP (Zwei-Wort-Instruktion) bzw. mit RETI, gefolgt von einem NOP, konstruiert werden, damit die Adressen stimmen.

Wegen der Notwendigkeit, die Interrupts möglichst rasch wieder freizugeben, damit der nächste anstehende Int nicht völlig aushungert, sollten Interrupt-Service-Routinen nicht allzu lang sein. Lange Prozeduren sollten vermieden und durch ein anderes Bearbeitungsschema ersetzt werden (Bearbeitung der zeitkritischen Teile innerhalb des Interrupts, Setzen einer Flagge, weitere Bearbeitung außerhalb der Servicerouti-ne).

Eine ganz, ganz wichtige Regel sollte in jedem Fall eingehalten werden: Die erste Instruktion einer Interrupt-Service-Routine ist die Rettung des Statusregisters in ein Register. Die letzte Instruktion der Routine ist die Wiederherstellung desselben in den Originalzustand vor der Unterbrechung. Jede Instruktion in der Unterbrechungsroutine, der irgendeines der Flags (außer dem I-Bit) verändert, würde unter Umständen verheerende Nebenfolgen auslösen, weil im unterbrochenen Programmablauf plötzlich irgendeins der verwendeten Flags anders wird als an dieser Stelle im Programmablauf vorgesehen, wenn die Interrupt-Service Routine zwischendurch zugeschlagen hat. Das ist auch der Grund, warum alle verwendeten Register in Unterbrechungsroutinen entweder gesichert und am Ende der Routine wiederhergestellt werden müssen oder aber speziell nur für die Int-Routinen reserviert sein müssen. Sonst wäre nach einer irgendwann auftretenden Unterbrechung für nichts mehr zu garantieren. Es ist nichts mehr so wie es war vor der Unterbrechung.

Weil das alles so wichtig ist, hier eine ausführliche beispielhafte Unterbrechungsroutine.

```
.CSEG ; Hier beginnt das Code-Segment
.ORG 0000 ; Die Adresse auf Null
    RJMP Start ; Der Reset-Vektor an die Adresse 0000
    RJMP IService ; 0001: erster Interrupt-Vektor, INT0 service routine
    [...] hier eventuell weitere Int-Vektoren
Start: ; Hier beginnt das Hauptprogramm
    [...] hier kann alles mögliche stehen
IService: ; Hier beginnt die Interrupt-Service-Routine
    IN R15,SREG ; Statusregister Zustand einlesen, R15 exklusiv
    [...] Hier macht die Int-Service-Routine irgendwas
    OUT SREG,R15 ; und Statusregister wiederherstellen
    RETI ; und zurückkehren
```

Das klingt alles ziemlich umständlich, ist aber wirklich lebenswichtig für korrekt arbeitende Programme.

Das war für den Anfang alles wirklich wichtige über Interrupts. Es gäbe noch eine Reihe von Tips, aber das würde für den Anfang etwas zu verwirrend. In Kapitel 11 gibt es mehr über Interrupts und wie sie den gesamten Programmablauf verändern.

AVR-ASM-Tutorial

10 Schleifen, Zeitverzögerungen und Co.

Hier werden Zeitbeziehungen in Schleifen vermittelt.

10.1 8-Bit-Schleifen

Code einer 8-Bit-Schleife

Eine Zeitschleife mit einem 8-Bit-Register sieht in Assembler folgendermaßen aus:

```
.equ c1 = 200 ; Anzahl Durchläufe der Schleife
        ldi R16,c1 ; Lade Register mit Schleifenwert
Loop: ; Schleifenbeginn
        dec R16 ; Registerwert um Eins verringern
        brne Loop ; wenn nicht Null dann Schleifenbeginn
```

Praktischerweise gibt die Konstante c1 die Anzahl Schleifendurchläufe direkt an. Mit ihr kann die Anzahl der Durchläufe und damit die Verzögerung variiert werden. Da in ein 8-Bit-Register nur Zahlen bis 255 passen, ist die Leistungsfähigkeit arg eng beschränkt.

Prozessortakte

Für die Zeitverzögerung mit einer solchen Schleife sind zwei Größen maßgebend:

- die Anzahl Prozessortakte, die die Schleife benötigt, und
- die Zeitdauer eines Prozessortakts.

Die Anzahl Prozessortakte, die die einzelnen Instruktionen benötigen, stehen in den Datenbüchern der AVRs. Und zwar ziemlich weit hinten, unter "Instruction Set Summary", in der Spalte "#Clocks". Demnach ergibt sich folgendes Bild:

```
.equ c1 = 200 ; 0 Takte, macht der Assembler alleine
        ldi R16,c1 ; 1 Takt
Loop: ; Schleifenbeginn
        dec R16 ; 1 Takt
        brne Loop ; 2 Takte wenn nicht Null, 1 Takt bei Null
```

Damit setzt sich die Anzahl Takte folgendermaßen zusammen:

1. Laden: 1 Takt, Anzahl Durchläufe: 1 mal
2. Schleife mit Verzweigung an den Schleifenbeginn: 3 Takte, Anzahl Durchläufe: c1 - 1 mal
3. Schleife ohne Verzweigung an den Schleifenbeginn: 2 Takte, Anzahl Durchläufe: 1 mal

Damit ergibt sich die Anzahl Takte nt zu: nt = 1 + 3*(c1-1) + 2 oder mit aufgelöster Klammer zu: nt = 1 + 3*c1 - 3 + 2 oder halt noch einfacher zu: nt = 3*c1 Jetzt haben wir die Anzahl Takte.

Taktfrequenz des AVR

Die Dauer eines Takts ergibt sich aus der Taktfrequenz des AVR. Die ist gar nicht so einfach zu ermitteln, weil es so viele Möglichkeiten gibt, sie auszuwählen oder zu verstellen:

- AVRs ohne interne Oszillatoren: diese brauchen entweder
    - einen externen Quarz,

    - einen externen Keramikresonator, oder
    - einen externen Oszillator

  Die Taktfrequenz wird in diesem Fall von den externen Komponenten bestimmt.
- AVRs mit einem oder mehreren internen Oszillatoren: diese haben eine voreingestellte Taktfrequenz, die im Kapitel "System Clock and Clock Options" bzw. im Unterkapitel "Default Clock Source" steht. Jeder AVR-Typ hat da so seine besondere Vorliebe.
- AVRs mit internen RC-Oszillatoren bieten ferner die Möglichkeit, diesen zu verstellen. Dazu werden Werte in den Port OSCCAL geschrieben. Bei älteren AVR geschieht das durch den Programmierer, bei moderneren schreibt ein automatischer Mechanismus den OSCCAL-Wert zu Beginn in den Port. Auch die Automatik benötigt den Handeingriff, wenn höhere Genauigkeit gefordert ist und der AVR abseits der Spannung oder Betriebstemperatur betrieben wird, für die er kalibriert ist (beim ATtiny13 z.B. 3 V und 25°C).
- Im Kapitel "System Clock and Clock Options" wird auch erklärt, wie man durch Verstellen von Fuses
    - zwischen internen und externen Taktquellen auswählt,
    - einen internen Vorteiler zwischen Taktquelle und Prozessortakt schalten kann (Clock Prescaler, siehe unten),
    - eine Wartezeit beim Start des Prozessors einfügt, bis der Oszillator stabil schwingt ("Delay from Reset").

  Die Fuses werden mittels Programmiergerät eingestellt. Obacht! Beim Verstellen von Fuses, z.B. auf eine externe Taktquelle, muss diese auch angeschlossen sein. Sonst verabschiedet sich der Prozessor absolut taktlos vom Programmiergerät und gibt diesem keine sinnvollen Antworten mehr.
- Um das Publikum zu verwirren, haben einige AVR noch einen Vorteiler für den Prozessortakt (Clock Prescaler). Dieser lässt sich entweder per Fuse (siehe oben) auf 1 oder 8 einstellen, bei manchen AVR aber auch per Software auf Teilerwerte von 1, 2, 4, 8, bis 256 einstellen (Port CLKPR).

Haben Sie aus all dem Durcheinander jetzt herausgefunden, mit wieviel MHz der AVR nun eigentlich läuft (fc), dann ergibt sich die Dauer eines Prozessortakts (tc) zu: tc [µs] = 1 / fc [MHz] Bei 1,2 MHz Takt (ATtiny13, interner RC-Oszillator mit 9,6 MHz, CLKPR = 8) sind das z.B. 0,83 µs. Die restlichen Stellen können Sie getrost vergessen, der interne RC-Oszillator ist so arg ungenau, dass weitere Stellen eher dem Glauben an Zauberei ähneln.

Zeitverzögerung

Mit der Anzahl Prozessortakte von oben (nc=3*c1, c1=200, nc=600) und 1,2 MHz Takt ergibt sich eine Verzögerung von 500 µs. Mit maximal 640 µs ist die Maximalbegrenzung dieser Konstruktion erreicht.

Verlängerung!

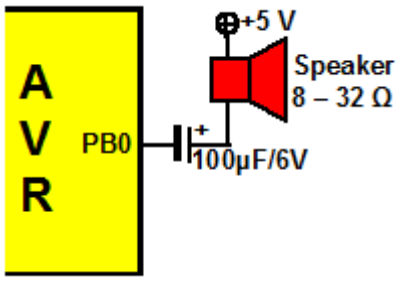
Mit folgendem Trick können Sie noch etwas Verlängerung herausholen:

```
.equ c1 = 200 ; 0 Takte, macht der Assembler alleine
        ldi R16,c1 ; 1 Takt
Loop: ; Schleifenbeginn
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
```

```
        dec R16 ; 1 Takt
        brne Loop ; 2 Takte wenn nicht Null, 1 Takt bei Null
```

Jetzt braucht jeder Schleifendurchlauf (bis auf den letzten) acht Takte und es gelten folgende Formeln: nc = 1 + 8*(c1 - 1) + 7 oder nc = 8 * c1 Das verlängert die 8-Bit-Registerschleife immerhin auf 256 * 8 = 2048 Takte oder - bei 1,2 MHz - auf ganze 1,7 ms.

10.2 Das Summprogramm



Immer noch nicht für die Blink-LED geeignet, aber immerhin für ein angenehmes Brummen mit 586 Hz im Lautsprecher. Schließen Sie einen Lautsprecher an Port B, Bit 0, an, fairerweise über einen Elko von einigen µF, und lassen Sie das folgende Programm auf den ATtiny13 los:

```
.inc "tn13def.inc" ; für einen ATtiny13
.equ c1 = 0 ; Bestimmt die Tonhöhe
        sbi DDRB,0 ; Portbit ist Ausgang
Loop:
        sbi PORTB,0 ; Portbit auf high
        ldi R16,c1
Loop1:
        nop
        nop
        nop
        nop
        nop
        dec R16
        brne Loop1
        cbi PORTB,0 ; Portbit auf low
        ldi R16,c1
Loop2:
        nop
        nop
        nop
        nop
        nop
        dec R16
        brne Loop2
        rjmp Loop
```

Gut brumm!

10.3 16-Bit-Schleife

Hier wird eine 16-Bit-Zeitschleife mit einem Doppelregister erklärt. Mit dieser können Verzögerungen um

ca. eine halbe Sekunde realisiert werden. Damit wird eine Blinkschaltung mit einer LED realisiert.

Code einer 16-Bit-Schleife

Eine Zeitschleife mit einem 16-Bit-Doppelregister sieht in Assembler folgendermaßen aus:

```
.equ c1 = 50000 ; Anzahl Durchläufe der Schleife
        ldi R25,HIGH(c1) ; Lade MSB-Register mit Schleifenwert
        ldi R24,LOW(c1) ; Lade LSB-Register mit Schleifenwert
Loop: ; Schleifenbeginn
        sbiw R24,1 ; Doppelregisterwert um Eins verringern
        brne Loop ; wenn nicht Null dann wieder Schleifenbeginn
```

Die Konstante c1 gibt wieder die Anzahl Schleifendurchläufe direkt an. Da in ein 16-Bit-Register Zahlen bis 65535 passen, ist die Schleife 256 mal leistungsfähiger als ein einzelnes 8-Bit-Register. Die Instruktion "SBIW R24,1" verringert das Doppelregister wortweise, d.h. nicht nur der Inhalt von R24 wird um eins verringert, bei einem Unterlauf von R24 wird auch das nächsthöhere Register R25 um Eins verringert. Das Doppelregister aus R24 und R25 (besser als R25:R24 benannt) eignet sich für solche Schleifen besonders gut, weil die Doppelregister X (R27:R26), Y (R29:R28) und Z (R31:R30) neben den 16-Bit-Operationen ADIW und SBIW auch noch andere Instruktionen kennen, und daher für den schnöden Zweck einer Schleife zu schade sind und R25:R24 eben nur ADIW und SBIW können.

Prozessortakte

Die Anzahl Prozessortakte, die die einzelnen Instruktionen benötigen, steht wieder in den Datenbüchern der AVRs. Demnach ergibt sich für die 16-Bit-Schleife folgendes Bild:

```
.equ c1 = 50000 ; 0 Takte, macht der Assembler alleine
        ldi R25,HIGH(c1) ; 1 Takt
        ldi R24,LOW(c1) ; 1 Takt
Loop: ; Schleifenbeginn
        sbiw R24,1 ; 2 Takte
        brne Loop ; 2 Takte wenn nicht Null, 1 Takt bei Null
```

Damit setzt sich die Anzahl Takte folgendermaßen zusammen:

1. Laden: 2 Takte, Anzahl Durchläufe: 1 mal
2. Schleife mit Verzweigung an den Schleifenbeginn: 4 Takte, Anzahl Durchläufe: c1 - 1 mal
3. Schleife ohne Verzweigung an den Schleifenbeginn: 3 Takte, Anzahl Durchläufe: 1 mal

Damit ergibt sich die Anzahl Takte nt zu: nt = 2 + 4*(c1-1) + 3 oder mit aufgelöster Klammer zu: nt = 2 + 4*c1 - 4 + 3 oder noch einfacher zu: nt = 4*c1 + 1.

Zeitverzögerung

Die Maximalzahl an Takten ergibt sich, wenn c1 zu Beginn auf 0 gesetzt wird, dann wird die Schleife 65536 mal durchlaufen. Maximal sind also 4*65536+1 = 262145 Takte Verzögerung möglich. Mit der Anzahl Prozessortakte von oben (c1=50000, nc=4*c1+1) und 1,2 MHz Takt ergibt sich eine Verzögerung von 166,7 ms. Variable Zeitverzögerungen unterschiedlicher Länge sind manchmal nötig. Dies kann z.B. der Fall sein, wenn die gleiche Software bei verschiedenen Taktfrequenzen laufen soll. Dann sollte die Software so flexibel gestrickt sein, dass nur die Taktfrequenz geändert wird und sich die Zeitschleifen automatisch anpassen. So ein Stück Software ist im Folgenden gezeigt. Zwei verschiedene Zeitverzögerungen

sind als Rechenbeispiele angegeben (1 ms, 100 ms).

```
;
; Verzoegerung 16-Bit mit variabler Dauer
;
.include "tn13def.inc"
;
; Hardware-abhaengige Konstante
;
.equ fc = 1200000 ; Prozessortakt (default)
;
; Meine Konstante
;
.equ fck = fc / 1000 ; Taktfrequenz in kHz
;
; Verzoegerungsroutine
;
Delay1ms: ; 1 ms Routine
.equ c1ms = (1000*fck)/4000 - 1 ; Konstante fuer 1 ms
        ldi R25,HIGH(c1ms) ; lade Zaehler
        ldi R24,LOW(c1ms)
        rjmp delay
;
Delay100ms: ; 100 ms Routine
.equ c100ms = (100*fck)/4 - 1 ; Konstante fuer 100 ms
        ldi R25,HIGH(c100ms) ; lade Zaehler
        ldi R24,LOW(c100ms)
        rjmp delay
;
; Verzoederungsschleife, erwartet Konstante in R25:R24
;
Delay:
        sbiw R24,1 ; herunter zaehlen
        brne Delay ; zaehle bis Null
        nop ; zusaetzliche Verzoegerung
```

Hinweise: Die Konstanten c1ms und c100ms werden auf unterschiedliche Weise berechnet, um bei der Ganzzahlen-Verarbeitung durch den Assembler einerseits zu große Rundungsfehler und andererseits Überläufe zu vermeiden.

Verlängerung!

Mit folgendem Trick können Sie wieder etwas Verlängerung herausholen:

```
.equ c1 = 0 ; 0 Takte, macht der Assembler alleine
        ldi R25,HIGH(c1) ; 1 Takt
        ldi R24,LOW(c1) ; 1 Takt
Loop: ; Schleifenbeginn
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
```

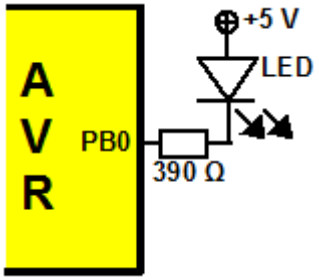
```
        nop ; tue nichts, 1 Takt
        nop ; tue nichts, 1 Takt
        sbiw R24 ; 2 Takte
        brne Loop ; 2 Takte wenn nicht Null, 1 Takt bei Null
```

Jeder Schleifendurchlauf (bis auf den letzten) braucht jetzt zehn Takte und es gelten folgende Formeln: nc = 2 + 10*(c1 - 1) + 9 oder nc = 10 * c1 + 1

Maximal sind jetzt 655361 Takte Verzögerung möglich.

10.4 Das Blinkprogramm

Damit sind wir beim beliebten "Hello World" für AVRs: der im Sekundenrhytmus blinkenden LED an einem AVR-Port. Die Hardware, die es dazu braucht, ist bescheiden. Die Software steht da:




```
.inc "tn13def.inc" ; für einen ATtiny13
.equ c1 = 60000 ; Bestimmt die Blinkfrequenz
        sbi DDRB,0 ; Portbit ist Ausgang
Loop:
        sbi PORTB,0 ; Portbit auf high
        ldi R25,HIGH(c1)
        ldi R24,LOW(c1)
Loop1:
        nop
        nop
        nop
        nop
        nop
        sbiw R24,1
        brne Loop1
        cbi PORTB,0 ; Portbit auf low
        ldi R25,HIGH(c1)
        ldi R24,LOW(c1)
Loop2:
        nop
        nop
        nop
        nop
        nop
        sbiw R24,1
        brne Loop2
        rjmp Loop
```

Gut blink!

11 Mehr über Interrupts

Bevor man seine ersten Programme mit Interrupt-Steuerung programmiert, sollte man das hier alles gelesen, verstanden und verinnerlicht haben. Auch wenn das einigen Aufwand bedeutet: es zahlt sich schnell aus, weil es viel Frust vermeidet.

11.1 Überlegungen zum Interrupt-Betrieb

Einfachste Projekte kommen ohne Interrupt aus. Wenn allerdings der Strombedarf minimiert werden soll, dann auch dann nicht. Es ist daher bei fast allen Projekten die Regel, dass eine Interruptsteuerung nötig ist. Und die will sorgfältig geplant sein.

Grundanforderungen des Interrupt-Betriebs

Falls es nicht mehr parat ist, hier ein paar Grundregeln:

- Interrupts ermöglichen
    - Interruptbetrieb erfordert einen eingerichteten SRAM-Stapel! ==> SPH:SPL sind zu Beginn auf RAMEND gesetzt, der obere Teil des SRAM (je nach Komplexität und anderweitigem Stapeleinsatz 8 bis x Byte) bleibt dafür freigehalten!
    - Jede Komponente (z.B. Timer) und jede Bedingung (z.B. ein Overflow), die einen Interrupt auslösen soll, wird durch Setzen des entsprechenden Interrupt-Enable-Bits in seinen Steuerregistern dazu ermutigt, dies zu tun.
    - Das I-Flag im Statusregister SREG wird zu Beginn gesetzt und bleibt möglichst während des gesamten Betriebs gesetzt. Ist es bei einer Operation nötig, Interrupts zu unterbinden, wird das I-Flag kurz zurückgesetzt und binnen weniger Befehlsworte wieder gesetzt.
- Interrupt-Service-Tabelle:
    - Jeder Komponente und jeder gesetzten Interruptbedingung ist eine Interrupt-Service-Routine zugeordnet, die an einer ganz bestimmten Addresse im Flash-Speicher beginnt. Die Speicherstelle erfordert an dieser Stelle einen Ein-Wort-Sprung in die eigentliche Service-Routine (RJMP; bei sehr großen ATmega sind Zwei-Wort-Sprünge - JMP - vorgesehen).
    - Die ISR-Addressen sind prozessortyp-spezifisch angeordnet! Beim Portieren zu einem anderen Prozessortyp sind diese Adressen entsprechend anzupassen.
    - Jede im Programm nicht verwendete ISR-Adresse wird mit einem RETI abgeschlossen, damit versehentlich eingeschaltete Enable-Bits von Komponenten definiert abgeschlossen sind und keinen Schaden anrichten könen. Die Verwendung der .ORG-Direktive zum Einstellen der ISR-Addresse ist KEIN definierter Abschluss!
    - Beim Vorliegen der Interrupt-Bedingung wird in den Steuerregistern der entsprechenden Komponente ein Flag gesetzt, das im Allgemeinen nach dem Anspringen der Interrupt-Service-Tabelle automatisch wieder gelöscht wird. In wenigen Ausnahmefällen kann es nötig sein (z.B. beim TX-Buffer-Empty-Interrupt der SIO, wenn kein weiteres Zeichen gesendet werden soll), den Interrupt-Enable abzuschalten und das bereits erneut gesetzte Flag zu löschen.
    - Bei gleichzeitig eintreffenden Interrupt-Service-Anforderungen sind die ISR-Addressen nach Prioritäten geordnet: die ISR mit der niedrigsten Addresse in der Tabelle wird bevorzugt ausgeführt.
- Interrupt-Service-Routinen:
    - Jede Service-Routine beginnt mit der Sicherung des Prozessor- Statusregisters in einem für diesen Zweck reservierten Register und endet mit der Wiederherstellung des Status. Da die Unterbrechung zu jeder Zeit erfolgen kann, - also auch zu einer Zeit, in der der Prozes-

sor mit Routinen des Hauptprogramms beschäftigt ist, die das Statusregister verwenden, - kann eine Störung dieses Registers unvorhersehbare Folgen haben.
    - Beim Ansprung der ISR wird die Rücksprungaddresse auf dem Stapel abgelegt, der dadurch nach niedrigeren Addressen hin wächst. Der Interrupt und das Anspringen einer Interrupt-Service-Tabelle schaltet die Ausführung weiterer anstehender Interrupts zunächst ab. Jede Interrupt-Service-Routine endet daher mit der Instruktion RETI, die den Stapel wieder in Ordnung bringt und die Interrupts wieder zulässt.
    - Da jede Interrupt-Service-Routine anstehende weitere Interrupts solange blockiert, wie sie selbst zu ihrer Ausführung benötigt, hat jede Interrupt-Service-Routine so kurz wie nur irgend möglich zu sein und sich auf die zeitkritischen Operationen zu beschränken.
    - Da auch Interrupts mit höherer Priorität blockiert werden, sollte bei zeitkritischen Operationen niedriger prioritäre Ints besonders kurz sein.
    - Da eine erneute Unterbrechung während der Verarbeitung einer Service-Routine nicht vorkommen kann, können in den verschiedenen ISR-Routinen die gleichen temporären Register verwendet werden.
- Schnittstelle Interrupt-Routine und Hauptprogramm:
    - Die Kommunikation zwischen der Interruptroutine und dem Hauptprogramm erfolgt über einzelne Flaggen, die in der ISR gesetzt und im Hauptprogramm wieder zurückgesetzt werden. Zum Rücksetzen der Flaggen kommen ausschließlich Ein-Wort-Instruktionen zum Einsatz oder Interrupts werden vorübergehend blockiert, damit während des Rücksetzvorgangs diese oder andere Flaggen im Register bzw. im SRAM nicht fälschlich überschrieben werden.
    - Werte aus der Service-Routine werden in dezidierten Registern oder SRAM-Speicherzellen übergeben. Jede Änderung von Register- oder SRAM-Werten innerhalb der Service-Routine, die außerhalb des Interrupts weiterverarbeitet werden, ist daraufhin zu prüfen, ob bei der Übergabe durch weitere Interrupts Fehler möglich sind. Die Übergabe und Weiterverarbeitung von Ein-Byte-Werten ist unproblematisch, bei Übergabe von zwei und mehr Bytes ist ein eindeutiger Übergabemechanismus zwingend (Interrupt Disable beim Kopieren der Daten in der Hauptprogramm-Schleife, Flag-Setzen/-Auswerten/-Rücksetzen, o.ä.)! Als Beispiel sei der Übergabemechanismus eines 16-Bit-Wertes von einem Timer/Counter an die Auswerteroutine beschrieben. Der Timer schreibt die beiden Bytes in zwei Register und setzt ein Flag in einem anderen Register, dass Werte zur Weiterverarbeitung bereitstehen. Im Hauptprogramm wird dieses Flag ausgewertet, zurückgesetzt und die Übergabe des Wertes gestartet. Wenn nun das erste Byte kopiert ist und erneut ein Interrupt des Timers/Counters zuschlägt, gehören anschließend Byte 1 und Byte 2 nicht zum gleichen Wertepaar. Das gilt es durch definierte Übergabemechanismen zu verhindern!
- Die Hauptprogramm-Routinen:
    - Im Hauptprogramm legt ein Loop den Prozessor schlafen, wobei der Schlafmodus "Idle" eingestellt sein muss. Jeder Interrupt weckt den Prozessor auf, verzweigt zur Service-Routine und setzt nach deren Beendigung die Verarbeitung fort. Es macht Sinn, nun die Flags darauf zu überprüfen, ob eine oder mehrere der Service-Routinen Bedarf an Weiterverarbeitung signalisiert hat. Wenn ja, wird entsprechend dorthin verzweigt. Nachdem alle Wünsche der ISRs erfüllt sind, wird der Prozessor wieder schlafen gelegt.

11.2 Die Interrupt-Vektoren-Tabelle

Hier kommt alles über die Reset- und Interrupt-Vektoren-Tabelle, und was man bei ihr richtig und falsch machen kann.

Vergessen Sie für eine Weile mal die Worte Vektor und Tabelle, sie haben hier erst mal nix zu sagen und

dienen nur der Verwirrung des unkundigen Publikums. Wir kommen auf die zwei Worte aber noch ausführlich zurück.

Stellen Sie sich einfach vor, dass jedes Gerät im AVR für jede Aufgabe, die einer Unterbrechung des Prozessors würdig ist, eine feste Adresse hat, zu der sie den Prozessor zwingt hinzuspringen, wenn das entsprechende Ereignis eintritt (also z.B. der Pegel an einem INT0-Eingang wechselt oder wenn der Timer gerade eben übergelaufen ist). Der Sprung an genau diese eine Adresse ist in jedem AVR für jedes Gerät und jedes Ereignis festgenagelt. Welche Adresse mit welchem Gerät und Ereignis verknüpft ist, steht im Handbuch für den jeweiligen AVR-Typ. Oder ist in den Tabellen am Ende dieses Kapitels aufgelistet.

Alle Interrupt-Sprungziele sind am Anfang des Programmspeichers, beginnend bei der Adresse 0000 (mit dem Reset), einfach aufgereiht. Es sieht fast aus wie eine Tabelle, es ist aber gar keine wirkliche. Eine Tabelle wäre, wenn an dieser Adresse tatsächlich das eigentliche Sprungziel selbst stünde, z.B. eine Adresse, zu der jetzt weiter verzweigt werden soll. Der Prozessor täte diese dort abholen und den aus der Tabelle ausgelesenen Wert in seinen Programmzähler laden. Wir hätten dann eine echte Liste mit Adressen vor uns, eine echte Tabelle.

Beim AVR gibt es aber gar keine solche Tabelle mit Adresswerten. Stattdessen stehen in unserer sogenannten Tabelle Sprungbefehle wie RJMP herum. Der AVR ist also noch viel einfacher gestrickt: wenn ein INT0-Interrupt auftritt, lädt er einfach die Adresse 0001 in seinen Programmspeicher und führt die dort stehende Instruktion aus. Und das MUSS dann eben ein Sprungbefehl an die echte Adresse sein, an der auf den Interrupt reagiert wird, die Interrupt-Service-Routine (ISR). Also nix Tabelle, sondern Auflistung der Sprunginstruktionen zu den Serviceroutinen.

Jetzt haben wir noch die Sache mit dem Vektor zu klären. Auch dieses Wort ist Käse und hat mit den AVRs rein gar nix zu tun. Als man noch Riesen-Mikrocomputer baute mit etlichen 40-poligen ICs, die als Timer, UARTs und I/O-Ports nach außen hin die Schnittstellenarbeit verrichteten, fand man es eine gute Idee, wenn diese selbst darüber bestimmen könnten, wo ihre Service-Routine im Programmspeicher zu finden ist. Sie gaben dazu bei einem Interrupt einen Wert an den Prozessor, der dann diesen zu einer Tabellen-Anfangsadresse addierte und die Adresse der Service-Routine von dieser Adresse holte. Das war sehr flexibel, weil man sowohl die Tabelle als auch die vom Schnittstellenbaustein zurückgegebenen Werte jederzeit im Programm manipulieren konnte und so flugs die ganze Tabelle oder die Interrupt-Service-Routine wechseln konnte. Der zu der Tabellenadresse zu zählende Wert wurde als Vektor oder Displacement (Verschiebung) bezeichnet. Und jetzt wissen wir, wie es zu dem Wort Vektortabelle kam, und dass es mit den AVR rein gar nix zu tun hat, weil wir es weder mit einer Tabelle noch mit Vektoren zu tun haben.

Unsere Sprungziele sind im AVR fest verdrahtet, es wird auch nix zu einer Anfangsadresse einer Tabelle addiert und schon gar nicht sind irgendwelche Service-Routinen austauschbar. Warum verwenden wir diese Worte überhaupt? Gute Frage. Weil es alle tun, weil es sich doll anhört und weil es Anfänger eine Weile davon abhält zu kapieren worum es wirklich geht.

Aussehen bei kleinen AVR

Eine Reset- und Interrupt-Vektor-Tabelle sieht bei einem kleineren AVR folgendermaßen aus:

```
rjmp Main ; Reset, Sprung zur Initiierung
rjmp Int0Sr ; Externer Interrupt an INT0, Sprung zur Service-Routine
reti ; Irgendein anderer Interrupt, nicht benutzt
rjmp IntTc0OvflwSr ; Überlauf Timer 0, Behandlungsroutine
reti ; Irgendwelche anderen Interrupts, nicht benutzt
reti ; Und noch mehr Interrupts, auch nicht benutzt
```

Merke:

- Die Anzahl der Instruktionen (RJMP, RETI) hat exakt der Anzahl der im jeweiligen AVR-Typ möglichen Interrupts zu entsprechen.
- Alle benutzten Interrupts verzweigen mit RJMP zu einer spezifischen Interrupt-Service-Routine.
- Alle nicht benutzten Interrupt-Sprungadressen werden mit der Instruktion RETI abgeschlossen.

Alles andere hat in dieser Sprungliste rein gar nix zu suchen. Das hat damit zu tun, dass die Sprungadressen dann genau stimmen, kein Vertun beim Springen erfolgt und die korrekte Anzahl und Abfolge mit dem Handbuch verglichen werden kann. Die Instruktion RETI sorgt dafür, dass der Stapel in Ordnung gebracht wird und Interrupts auf jeden Fall wieder zugelassen werden.

Schlaumeier glauben, sie könnten auf die lästigen RETI-Instruktionen verzichten. Z.B. indem sie das oben stehende wie folgt formulieren:

```
   rjmp Main ; Reset, Sprung zur Initiierung
.org $0001
   rjmp Int0Sr ; Externer Interrupt an INT0, Sprung zur Service-Routine
.org $0003
   rjmp IntTc0OvflwSr ; Überlauf Timer 0, Behandlungsroutine
```

Das geht, alle Sprungbefehle stehen an der korrekten Position. Es funktioniert auch, solange nicht absichtlich oder aus Versehen ein weiterer Interrupt zugelassen wird. Der ungeplante Interrupt findet an der mit .org übersprungenen Stelle den auszuführenden Opcode $FFFF vor, da alle unprogrammierten Speicherstellen des Programmspeichers beim Löschen mit diesem befüllt werden. Dieser Opcode ist nirgendwo definiert, er macht auch nix, er bewirkt nichts und die Bearbeitung wird einfach an der nächsten Stelle im Speicher fortgesetzt. Wo der versehentliche Interrupt landet, ist dann recht zufällig und jedenfalls ungeplant.

Folgt auf die Einsprungstelle irgendwo noch ein weiterer Sprung zu einer anderen Serviceroutine, dann wird eben die fälschlich ausgeführt. Folgt keine mehr, dann läuft das Programm in die nächstfolgende Unterroutine. Das wäre fatal, weil die garantiert nicht mit RETI endet und daher die Interrupts nie wieder zugelassen werden. Oder, wenn keine Unterroutinen zwischen der Sprungtabelle und dem Hauptprogramm stehen, der weitere Ablauf läuft in das Hauptprogramm, und alles wird wieder von vorne initiiert.

Ein weiteres Scheinargument, damit liefe die Software auf jedem anderen AVR-Typ auch korrekt, ist ebenfalls Käse. Ein Blick auf die Tabellen mit den Interrupt-Sprungzielen zeigt, dass sich ATMEL mitnichten immer an irgendwelche Reihenfolgen gehalten hat und dass es munter durcheinander geht. Die Scheinkompatibilitäet kann also zur Nachlässigkeit verführen, eine fatale Fehlerquelle. Daher sollte gelten:

- In einer Sprungtabelle haben .org-Direktiven nix verloren.
- Jeder (noch) nicht verwendete Eintrag in der Sprungtabelle wird mit RETI abgeschlossen.
- Die Länge der Sprungtabelle entspricht exakt der für den Prozessor definierten Anzahl Interruptsprünge.

Aufbau bei großen AVR

Große AVR haben einen Adressraum, der mit relativen Sprungbefehlen nicht mehr ganz zugänglich ist. Bei diesen hat die Sprungliste Einträge mit jeweils zwei Worten Länge. Etwa so:

```
   jmp Main ; Reset, Sprung zur Initiierung
   jmp Int0Sr ; Externer Interrupt an INT0, Sprung zur Service-Routine
   reti ; Irgendein anderer Interrupt, nicht benutzt
   nop
```

```
jmp IntTc0OvflwSr ; Überlauf Timer 0, Behandlungsroutine
reti ; Irgendwelche anderen Interrupts, nicht benutzt
nop
reti ; Und noch mehr Interrupts, auch nicht benutzt
nop
```

Bei Ein-Wort-Einträgen (z.B. RETI) sind die NOP-Instruktionen eingefügt, um ein weiteres Wort zu ergänzen, damit die Adressierung der nachfolgenden Sprungziel-Einträge wieder stimmt.

11.3 Das Interruptkonzept

Hier wird erläutert, wie interrupt-gesteuerte Programme grundsätzlich ablaufen müssen. Um das Prinzip zu verdeutlichen, wird zuerst ein einfacher Ablauf demonstriert. Dann wird gezeigt, was sich bei mehreren Interrupts ändert.

Für alle, die bisher nur in Hochsprachen oder, in Assembler, nur schleifengesteuerte Programme gebaut haben, hier eine wichtige Warnung. Das Folgende wirft einiges Gewohntes über Programmierung über den Haufen. Mit den gewohnten Schemata im Kopf ist das Folgende schlicht nicht zu kapieren. Lösen Sie sich von dem Gewohnten und stellen Sie ausschließlich die Interrupts in den Vordergrund. Um diese dreht sich bei dieser Programmierung nun alles. Das heißt, der Interrupt ist unser Ausgangspunkt für so ziemlich alles:

- mit ihm beginnt jeder Ablauf,
- er steuert alle Folgeprozesse,
- sein Wohlergehen (Funktionieren, Eintreten, rechtzeitige Bearbeitung, etc.) ist alleroberste Maxime,
- alle anderen Programmteile horchen auf seinen Eintritt und sind reine Sklaven und Zulieferanten des großen Meisters Interrupt.

Wenn Sie Schwierigkeiten und Unwohlsein beim Kapieren des Konzepts verspüren, versuchen Sie aus dem ungewohnten Dilemma auch nicht den Ausweg, in alte Strukturen zu verfallen. Ich garantiere Ihnen, dass eine Mischung aus Interrupts und gewohntem linearen Programmieren am Ende viel komplizierter ist als das neue Konzept in seiner Reinform anzuwenden: sie verheddern sich in diversen Schleifen, der nächste Interrupt kommt bestimmt, aber ob Ihr Hauptprogramm auf ihn hört, bleibt ein Rätsel.

Wenn Sie das Interrupt-Konzept verinnerlicht haben, geht alles viel leichter und eleganter. Und garantiert noch schneller und eleganter als in der Hochsprache, die solche netten Erfindungen vor Ihnen verheimlicht, sie in vorgefertigte und oft unpassende Korsette einschnürt, so dass Sie die wahre Vielfalt des Möglichen gar nicht erst zu Gesicht bekommen.

11.4 Ablauf eines interruptgesteuerten Programms

Den Standardablauf eines interrupt-gesteuerten Propgrammes in allgemeiner Form zeigt das folgende Bild.

Reset, Init

Nach dem Einschalten der Betriebsspannung beginnt der Prozessorstart immer bei der Adresse 0000. An dieser Adresse MUSS ein Sprungbefehl an die Adresse des Hauptprogrammes stehen (in der Regel RJMP, bei großen ATmega kann auch JMP verwendet werden).

Das Hauptprogramm setzt zu allererst die Stapeladresse in den bzw. die Stapelzeiger, weil alle interrupt-

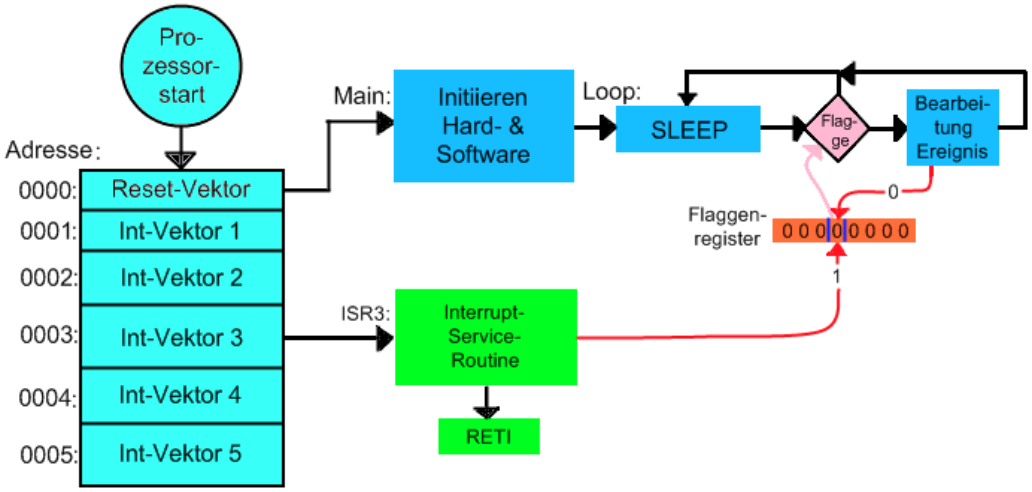
gesteuerten Programme den Stapel ZWINGEND benötigen.

Dann initiiert das Hauptprogramm die Hardware, schaltet also Timer, AD-Wandler, Serielle Schnittstelle und was auch immer gebraucht wird ein und ermöglicht die entsprechenden Interrupts.

Außerdem werden dann noch Register mit ihren richtigen Startwerten zu laden, damit alles seinen geregelten Gang geht. Dann ist noch wichtig, den Schlafmodus so zu setzen, dass der Prozessor bei Interrupts auch wieder aufwacht.

Am Ende wird das Interrupt-Flag des Prozessors gesetzt, damit er auch auf ausgelöste Interrupts reagieren kann. Damit ist alles erledigt und der Prozessor wird mit der Instruktion SLEEP schlafen gelegt.

Interrupt



Irgendwann schlägt nun Interrupt 3 zu. Der Prozessor

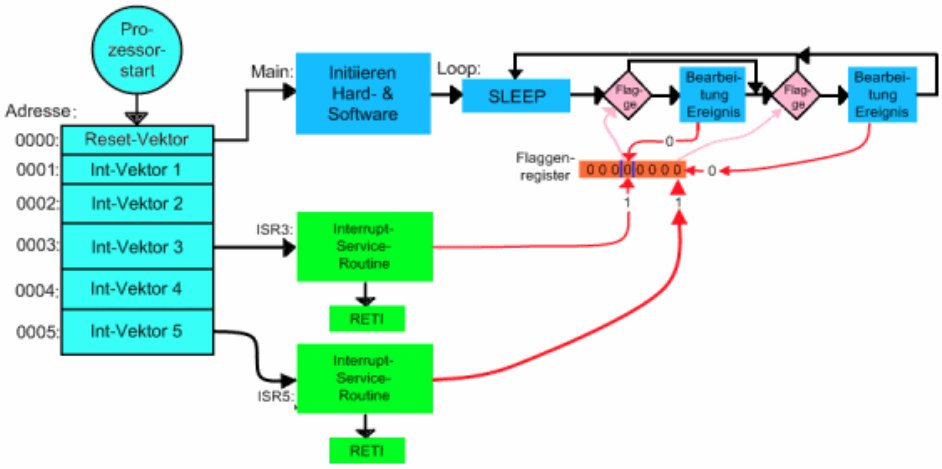
- wacht auf,
- schaltet die Interrupts ab,
- legt den derzeitigen Wert des Programmzählers (die nächste Instruktion hinter SLEEP) auf dem Stapel ab,
- schreibt die Adresse 0003 in den Programmzähler, und
- setzt die Verarbeitung an dieser Adresse fort,
- dort steht ein Sprungbefehl zur Interrupt-Service-Routine an Adresse ISR3:,
- ISR3: wird bearbeitet und signalisiert durch Setzen von Bit 4 in einem Flaggenregister, dass eine Weiterverarbeitung des Ereignisses notwendig wird,
- ISR3 wird beendet, indem mit der Instruktion RETI die Ausgangsadresse vom Stapel geholt und in den Programmzähler geladen wird und schließlich die Interrupts wieder zugelassen werden,
- der nun aufgeweckte Prozessor setzt die Verarbeitung an der Instruktion hinter SLEEP fort,
- falls ISR3 das Flaggenbit gesetzt hat, kann jetzt die Nachbearbeitung des Ereignisses erfolgen und das Flaggenbit wieder auf Null gesetzt werden,
- ohne oder mit Nachbearbeitung wird der Prozessor auf jeden Fall danach wieder schlafen gelegt.

Der Ablauf zeigt, wie die Interrupt-Service-Routine mit dem Rest des Programmablaufs kommuniziert: über das Flaggenbit wird mitgeteilt, dass etwas Weiteres zu tun ist, was nicht innerhalb der Service-Routine erledigt wurde (z.B. weil es zu lange dauern würde, zu viele Ressourcen benötigt, etc.).

11.5 Ablauf bei mehreren interruptgesteuerten Abläufen

Sind mehrere Interrupts aktiv, beginnt der Programmablauf genauso und auch alle weiteren Abläufe sind ganz ähnlich.

Durch die Wahl der Reihenfolge der Nachbearbeitung lässt sich steuern, welche zuerst fertig ist.



Das Schema lässt sich leicht um weitere Interrupts und deren Nachbearbeitung erweitern. Das Ganze funktioniert nur, wenn die Interrupt-Service-Routinen kurz sind und sich nicht gegenseitig blockieren und behindern. Bei zunehmendem Zeitbedarf und vielen rasch aufeinander folgenden Interrupts sind sorgfältige Überlegungen zum Timing wichtig.

Folgen zwei Setzvorgänge im Flaggenregister zu rasch aufeinander und dauert die Nachbearbeitung zu lange, würde ein Ereignis verpasst. Man muss dann andere Wege versuchen, die Lasten bei der Bearbeitung zu verteilen. Man erkennt hier nebenbei die Zentrierung des gesamten Programmablaufes auf die Interrupts. Sie sind die zentrale Steuerung.

11.6 Interrupts und Ressourcen

Interrupts haben den Vorteil, dass sie nur dann auftreten und bearbeitet werden müssen, wenn ein bestimmtes Ereignis tatsächlich eintritt. Selten ist vorhersagbar, wann das Ereignis genau eintreten wird. Sogar mehrere unterschiedliche Ereignisse können gleichzeitig oder kurz hintereinander eintreten. Zum Beispiel kann der Timer gerade einen Overflow haben und der AD-Wandler ist gerade mit einer Messumwandlung fertig.

Diese Eigenart, dass sie jederzeit auftreten können, macht einige besondere Überlegungen beim Programmieren erforderlich, die in Hochsprachen entweder nicht vorkommen oder vom Compiler eigenständig erledigt werden. In Assembler müssen wir selber denken, und werden dafür mit einem schlanken, sauschnellen und zuverlässigen Programm belohnt.

Die folgenden Überlegungen nimmt uns Assembler-Programmierern keiner ab.

Klare Ressourcenzuordnung

Das Statusregister als sensible Ressource

Das Zuschlagen eines Interrupts kann jederzeit und überall, nur nicht innerhalb von Interrupt-Service-Routinen erfolgen. Erfolgt innerhalb der Service-Routine eine Beeinflussung von Flaggen des Statusregisters, was nahezu immer der Fall ist, dann kann das im Hauptprogramm-Loop üble Folgen haben. Plötzlich ist das Zero- oder Carry-Flag im Statusregister gesetzt, nur weil zwischendurch der Interrupt zuschlug. Nichts funktioniert mehr so, wie man es erwartet hat. Aber das nur machmal und nicht immer. Ein solches Programm verhält sich eher wie ein Zufallsgenerator als wie ein zuverlässiges Stück Software.

Nr.	Sicherung in	Code	Zeitbedarf Takte	Vorteile	Nachteile
1	Register	in R15,SREG [...] out SREG,R15	2	Schnell	Registerverbrauch
2	Stapel	push R0 in R0,SREG […] out SREG,R0 pop R0	6	Kein Registerver-brauch	Lahm
3	SRAM	sts $0060,R0 in R0,SREG […] out SREG,R0 lds R0,$0060	6		

AVR-Prozessoren haben keinen zweiten Satz Statusregister, auf den sie bei einem Interrupt umschalten können. Deshalb muss das Statusregister vor einer Veränderung in der Interrupt-Service-Routine gesichert und vor deren Beendigung wieder in seinen Originalzustand versetzt werden. Dazu gibt es drei Möglichkeiten (Tabelle). Damit ist die Auswahl und Priorität klar. Alles entscheidend ist, ob man Register satt verfügbar hat oder verfügbar machen kann.

Es gibt eine böse Fußangel. Oft verwendet man für eine einzelne Flagge gerne das T-Bit im Statusregister. Da gibt es bei allen drei Methoden einen üblen Fallstrick: Jede Veränderung am T-Bit wird wieder überschrieben, wenn am Ende der Routine der Originalstatus des Registers wieder hergestellt wird. Bei Methode 1 muss also Bit 7 des R15 gesetzt werden, damit das T-Bit nach der Beendigung der Routine als gesetzt resultiert. Obacht: Kann einige Stunden Fehlersuche verursachen!

Verwendung von Registern

Angenommen, eine Interrupt-Service-Routine mache nichts anderes als das Herunterzählen eines Zählers. Dann ist klar, dass das Register, in dem der Zähler untergebracht ist (z.B. mit .DEF rCnt = R17 definiert), zu nichts anderem eingesetzt werden kann. Jede andere Verwendung von R17 würde den Zählrhythmus empfindlich stören. Dabei hülfe es bei einer Verwendung in der Hauptprogramm-Schleife auch nicht, wenn wir den Inhalt des Registers mit PUSH rCnt auf dem Stapel ablegen würden und nach seiner Verwendung den alten Zustand mit POP rCnt wieder herstellen würden. Irgendwann schlägt der Interrupt genau zwischen dem PUSH und dem POP zu, und dann haben wir den Salat. Das passiert vielleicht nur alle paar Minuten mal, ein schwer zu diagnostizierender Fehler.

Wir müssten dann schon vor der Ablage auf dem Stapel alle Interrupts mit CLI abschalten und nach der Verwendung und Wiederherstellung des Registers die Interrupts wieder mit SEI zulassen. Liegen zwischen Ab- und Anschalten der Interrupts viele Hundert Instruktionen, dann stauen sich die zwischenzeitlich aufgelaufenen Interrupts und können verhungern. Wenn zwischendurch der Zähler zwei mal übergelaufen ist, dann geht unsere Uhr danach etwas verkehrt, weil aus den zwei Ereignissen nur ein Interrupt geworden ist.

Innerhalb einer anderen Interrupt-Service-Routine können wir rCnt auf diese Weise (also mit PUSH und

POP) verwenden, weil diese nicht durch andere Interrupts unterbrochen werden kann. Allerdings sollte man sich bewusst sein, dass jedes PUSH- und POP-Pärchen vier weitere Taktimpulse verschwendet. Wer also satt Zeit hat, hat auch Register en masse. Wer keine Register übrig hat, kommt um so was vielleicht nicht herum.

Manchmal MUSS auf ein Register oder auf einen Teil eines Registers (z.B. ein Bit) sowohl von innerhalb als auch von außerhalb einer Interrupt-Service-Routine zugegriffen werden, z.B. weil die ISR dem Hauptprogramm-Loop etwas mitzuteilen hat. In diesen Fällen muss man sich ein klares Bild vom Ablauf verschaffen. Es muss klar sein, dass Schreibzugriffe sich nicht gegenseitig stören oder blockieren.

Es muss dann auch klar sein, dass dasselbe Register bei zwei nacheinder folgenden Lesevorgängen bereits von einem weiteren Interrupt verändert worden sein kann. Liegen die zwei Zeitpunkte längere Zeit auseinander, dann ist es je nach dem Timing des Gesamtablaufes fast zwingend, dass irgendwann ein Konflikt auftritt.

An einem Beispiel: sowohl der Timer als auch der ADC haben dem Hauptprogramm etwas mitzuteilen und setzen jeweils ein Bit eines Flaggenregisters rFlag. Also etwa so:

```
Isr1:
        [...]
        sbr rFlag,1<<bitA ; setze Bearbeitungsflagge A
        [...]
Isr2:
        [...]
        sbr rFlag,1<<bitB ; setze Bearbeitungsflagge B
        [...]
```

Wenn jetzt in der Hauptprogrammschleife die beiden Bits nacheinander abgefragt und Behandlungsroutinen aufgerufen werden, kann entweder bitA oder bitB oder sowohl bitA als auch bitB gesetzt sein. Auf jeden Fall müssen die gesetzten Flaggen so schnell wie möglich wieder auf Null gesetzt werden, denn der nächste Interrupt kann schon bald wieder zuschlagen. Auf keinen Fall dürfen wir beide Flaggen erst nach einer aufwändigen Bearbeitung löschen, weil wir dann vielleicht die nächste Bearbeitungsanforderung verpassen würden.

Es lohnt sich dann, mit der in schnellerer Folge auftretenden Anforderung zu beginnen und dann die etwas gemütlicher werkelnde zweite Anforderung zu bearbeiten. Wenn A häufiger ist als B, z.B. so:

```
Loop:
        sleep ; schlafen legen
        nop ; Dummy nach Aufwachen
        sbrc rFlag,bitA ; frage erst bitA ab
        rcall BearbA ; bearbeite Anforderung A
        sbrc rFlag,bitB ; frage dann bitB ab
        rcall BearbB ; bearbeite dann Anforderung B
        sbrc rFlag,bitA ; frage zur Sicherheit noch mal bitA ab
        rcall BearbA ; bearbeite weitere Anforderung A, falls nötig
        rjmp Loop ; gehe wieder schlafen
;
BearbA: ; Bearbeite Anforderung A
        cbr rFlag,1<<bitA ; lösche vor der Bearbeitung die gesetzte Flagge
        [...]
        ret
;
```

```
BearbB: ; Bearbeite Anforderung B
        cbr rFlag,1<<bitB ; lösche vor der Bearbeitung die gesetzte Flagge
        [...]
        ret
```

Man beachte, dass in beiden Fällen immer nur ein Bit im Register rFlag zurückgesetzt werden darf, also auf keinen Fall CLR rFlag, weil es könnten ja in seltenen Fällen auch beide gleichzeitig gesetzt worden sein.

Ist eine der Behandlungsroutinen ein längliches Monster, z.B. der Update einer LCD-Anzeige, 16 Schreibvorgänge im EEPROM oder das Leeren eines Pufferspeichers mit 80 Zeichen im SRAM, dann kann das zum Verhungern der jeweils anderen Bearbeitungsroutine führen. In diesen Fällen ist Obacht geboten. Dann muss man eben das EEPROM auch interrupt-gesteuert befüllen (mit dem EE_RDY-Interrupt), auch wenn das einigen Programmieraufwand mehr erfordert. Oder sich eine andere Lösung ausdenken.

Betrachten Sie die genannten Konflikte der Bearbeitung von Signalen als anspruchsvolle Denksportaufgabe, dann haben Sie den richtigen Ansatz.

Die oben beschriebene Zugriffssteuerung über eine Interrupt-Blockade mit CLI/SEI ist auf jeden Fall dann zwingend, wenn ein Doppelregister außerhalb von Interrupt-Service-Routinen auf einen neuen Wert gesetzt werden muss und innerhalb von Service-Routinen verwendet wird. Zwischen dem Setzen des einen Bytes des Doppelregisters und des zweiten könnte ein Interrupt zuschlagen und einen völlig verqueren Wert vorfinden. Solche Fehler sind teuflisch, weil es in der Regel korrekt funktioniert und nur alle paar Stunden ein Mal dieser Fall eintritt. So einen eingebauten Bug findet man garantiert nicht, weil er sich so selten in freier Wildbahn in der Schaltung zeigt.

Daher sollten folgende Regeln gelten:

- Register möglichst klar der alleinigen Verwendung innerhalb und außerhalb von Interrupt-Service-Routinen zuordnen.
- Jede gemeinsame Nutzung von Registern intensiv auf mögliche Konflikte hin durchdenken. Vorher denken vermeidet die Fehlersuche hinterher.
- Vorsicht bei der gemeinsamen Nutzung von Doppelregistern. Interrupts blockieren! Und hinterher nicht vergessen, sie wieder zuzulassen.

11.7 Quellen von Interrupts

Generell gilt:

- Jeder AVR-Typ verfügt über unterschiedliche Hardware und kann daher auch unterschiedliche Arten an Interrupts auslösen.
- Ob und welche Interrupts implementiert sind, ist aus dem jeweiligen Datenbuch für den AVR-Typ zu entnehmen.
- Die Interruptquellen bei jedem Typ sind priorisiert, d.h. bei gleichzeitig auftretender Interruptanforderung wird der Interrupt mit der höheren Priorität zuerst bearbeitet.

11.7.1 Arten Hardware-Interrupts

Bei den folgenden Tabellen bedeutet ein kleines **n** eine Ziffer, die Zählung beginnt immer mit **0**. Die Benennung der Interrupts in den Handbüchern sind uneinheitlich, die Abkürzungen für die Benennung der Interrupts wurden für eine übersichtliche Darstellung gewählt und sind nicht offiziell. Folgende hauptsächlichen Interrupts sind zu unterscheiden. Aufgelistet ist die Hardware, wann ein Interrupt erfolgt, Erläuterun-

gen dazu, die Verfügbarkeit der Interruptart bei den AVR-Typen und der Name, unter dem der Interrupt in den folgenden Tabellen gefunden wird.

Gerät	Interrupt bei...	Erläuterung	Verfügbarkeit	Name(n)
Externe Interrupts	Pegelwechseln an einem bestimmten Portanschluss (INTn)	Wählbar, ob jeder Pegelwechsel, nur solche von Low auf high oder nur solche von High auf Low einen Interrupt auslösen können (ISC-Bits).	INT0 bei allen, viele haben INT1, wenige INT2	INTn
Port Interrupts	Pegelwechsel an einem Port (PCIn)	Wählbar mit Maske, welche Bits bei Pegelwechsel einen Interrupt auslösen	bei vielen tiny/mega	PCIn
Timer Interrupts	Überlauf	Timer überschreitet seinen Maximalwert und beginnt bei Null; bei 8-Bit-Timern 256, bei 16-Bit-Timern 65536, bei CTC auch der im Compare-Register A eingestellte Maximalwert	bei allen Timern	TCnO
	Vergleichswert	bei Timern mit nur einem Vergleichswert: Übereinstimmung mit COMP-Wert erreicht	viele	TCnC
	Vergleichswerte A, B, C	bei Timern mit mehreren Vergleichswerten: Übereinstimmung mit COMP-Wert erreicht	A,B: viele, C wenige	TCnA...TCnC
	Fangwert	bei Timer im Zählmodus: Übereinstimmung der gezählten Impulse mit CAPT-Wert erreicht	viele	TCnP
UART	Zeichen empfangen	ein vollständiges Zeichen wurde empfangen und liegt im Eingangspuffer	viele	UnRX
	Senderegister frei	Zeichen in Sendepuffer übernommen, frei für nächstes Zeichen	viele	UnUD
	Senden beendet	letztes Zeichen fertig ausgesendet, Senden beendet	viele	UnTX
EEPROM	EEPROM ready	Schreibvorgang im EEPROM ist beendet, der nächste Schreibvorgang kann begonnen werden	Fast alle	EERY
ADC	Wandlung komplett	eine Umwandlung der Eingangsspannung ist erfolgt, der Ergebniswert kann vom Datenregister abgeholt werden	Alle mit ADC	ADC
AnalogKomparator	Wechsel bei Vergleich	je nach Einstellung der ACIS-Bits erfolgt ein Interrupt bei jedem Polaritätätswechsel oder nur bei positiven bzw. negativen Flanken	alle	ANA ANAn
Weitere	(diverse)	(weitere Interrupt-Arten sind den Handbüchern zu entnehmen)	einige	

Jeder dieser Interrupts ist per Software ein- und ausschaltbar, auch während des Programmablaufs. Die Voreinstellung beim RESET ist, dass keiner dieser Interrupts ermöglicht ist.

11.7.2 AVR-Typen mit Ein-Wort-Vektoren

Bedingt durch die unterschiedliche Ausstattung der AVR-Typen mit internem SRAM ist bei den AVR-Typen die Interrupt-Vektoren-Tabelle entweder auf Ein-Wort- (RJMP) oder Zwei-Wort-Instruktionen (JMP) ausgelegt. AVR-Typen mit wenig SRAM kommen mit einem relativen Sprung aus und benötigen daher ein Wort pro Vektor. Jedes Programmwort in der Tabelle entspricht genau einem Interrupt. In der folgenden Tabelle sind die AVR-Typen mit ihren Interrupts aufgelistet. Die Adressen geben an, an welcher Stelle der Tabelle der Vektor liegt. Sie geben gleichzeitig die Priorität an: je niedriger die Adresse desto höher der Vorrang des Interrupts. Leere Kästen geben an, dass dieser Vektor nicht verfügbar ist.

Typ/Adr	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011	0012	0013	0014
AT90S1200	RES	INT0	TC0O	ANA																	
AT90S2313	RES	INT0	INT1	TC1P	TC1C	TC1O	TC0O	U0RX	U0UD	U0TX	ANA										
AT90S2323	RES	INT0	TC0O																		
AT90S2343	RES	INT0	TC0O																		
AT90S4433	RES	INT0	INT1	TC1P	TC1C	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA							
AT90S4434	RES	INT0	INT1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA				
AT90S8515	RES	INT0	INT1	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ANA								
AT90S8535	RES	INT0	INT1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA				
ATmega8	RES	INT0	INT1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA	TWI	SPMR		
ATmega8515	RES	INT0	INT1	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	INT2	TC0C	EERY	SPMR				
ATmega8535	RES	INT0	INT1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA	TWI	INT2	TC0C	SPMR
ATtiny11	RES	INT0	PCI0																		
ATtiny12	RES	INT0	PCI0																		
ATtiny13	RES	INT0	PCI0	TC0O	EERY	ANA	TC0A	TC0B	WDT	ADC											
ATtiny15L	RES	INT0	PCI0	TC1A	TC1O	TC0O	EERY	ANA	ADC												
ATtiny22	RES	INT0	TC0O																		
ATtiny24	RES	INT0	PCI0	PCI1	WDT	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B	TC0O	ANA	ADC	EERY	USIS	USIO				

Typ/Adr	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011	0012	0013	0014
ATtiny25	RES	INT0	PCI0	TC1A	TC1O	TC0O	EERY	ANA	ADC	TC1B	TC0A	TC0B	WDT	USIS	USIO						
ATtiny26	RES	INT0	PCI0	TC1A	TC1B	TC1O	TC0O	USIS	USIO	EERY	ANA	ADC									
ATtiny28	RES	INT0	INT1	PCI0	TC0O	ANA															
ATtiny44	RES	INT0	PCI0	PCI1	WDT	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B	TC0O	ANA	ADC	EERY	USIS	USIO				
ATtiny45	RES	INT0	PCI0	TC1A	TC1O	TC0O	EERY	ANA	ADC	TC1B	TC0A	TC0B	WDT	USIS	USIO						
ATtiny84	RES	INT0	PCI0	PCI1	WDT	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B	TC0O	ANA	ADC	EERY	USIS	USIO				
ATtiny85	RES	INT0	PCI0	TC1A	TC1O	TC0O	EERY	ANA	ADC	TC1B	TC0A	TC0B	WDT	USIS	USIO						
ATtiny261	RES	INT0	PCI0	TC1A	TC1B	TC1O	TC0O	USIS	USIO	EERY	ANA	ADC	WDT	INT1	TC0A	TC0B	TC0P	TC1D	FAUL		
ATtiny461	RES	INT0	PCI0	TC1A	TC1B	TC1O	TC0O	USIS	USIO	EERY	ANA	ADC	WDT	INT1	TC0A	TC0B	TC0P	TC1D	FAUL		
ATtiny861	RES	INT0	PCI0	TC1A	TC1B	TC1O	TC0O	USIS	USIO	EERY	ANA	ADC	WDT	INT1	TC0A	TC0B	TC0P	TC1D	FAUL		
ATtiny2313	RES	INT0	INT1	TC1P	TC1A	TC1O	TC0O	U0RX	U0UD	U0TX	ANA	PCI0	TC1B	TC0A	TC0B	USIS	USIO	EERY	WDTO		

Man beachte, dass

- Arten und Reihenfolgen der Interrupts nur bei einigen wenigen Typen gleich sind,
- bei Umstellung von einem auf einen anderen Prozessor in der Regel die gesamte Vektortabelle umgestellt werden muss.

11.7.3 AVR-Typen mit Zwei-Wort-Vektoren

Bei AVR-Typen mit größerem SRAM kann ein Teil des verfügbaren Programmraums mit relativen Sprungbefehlen wegen der begrenzten Sprungdistanz nicht mehr erreicht werden. Diese Typen verwenden daher pro Vektor zwei Speicherworte, so dass der absolute Sprungbefehl JMP verwendet werden kann, der zwei Worte einnimmt.
Die folgenden Tabellen listen die bei diesen AVR-Typen verfügbaren Interrupts auf. Da die Anzahl an Vektoren sehr groß ist, ist die Tabelle in je-weils 16 Vektoren unterteilt.

Interrupts mit den Vektoren 0000 bis 001E

AVR-Typ/Adr	0000	0002	0004	0006	0008	000A	000C	000E	0010	0012	0014	0016	0018	001A	001C	001E
AT90CAN32	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	TC2C	TC2O	TC1P	TC1A	TC1B	TC1C	TC1O
AT90CAN64	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	TC2C	TC2O	TC1P	TC1A	TC1B	TC1C	TC1O
AT90CAN128	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	TC2C	TC2O	TC1P	TC1A	TC1B	TC1C	TC1O
ATmega16	RES	INT0	INT1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY
ATmega32	RES	INT0	INT1	INT2	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega48	RES	INT0	INT1	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B
ATmega64	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C
ATmega88	RES	INT0	INT1	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B
ATmega103	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C
ATmega128	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C
ATmega161	RES	INT0	INT1	INT2	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U1RX	U0UD
ATmega162	RES	INT0	INT1	INT2	PCI0	PCI1	TC3P	TC3A	TC3B	TC3O	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O
ATmega163	RES	INT0	INT1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY
ATmega164	RES	INT0	INT1	INT2	PCI0	PCI1	PCI2	PCI3	WDT	TC2A	TC2B	TC2O	TC1P	TC1A	TC1B	TC1O
ATmega165	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega168	RES	INT0	INT1	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B
ATmega169	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega323	RES	INT0	INT1	INT2	TC2C	TC20	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega324	RES	INT0	INT1	INT2	PCI0	PCI1	PCI2	PCI3	WDT	TC2A	TC2B	TC2O	TC1P	TC1A	TC1B	TC1O
ATmega325	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega329	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega406	RES	BATI	INT0	INT1	INT2	INT3	PCI0	PCI1	WDT	WAKE	TC1C	TC1O	TC0A	TC0B	TC0O	TWBU
ATmega640	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O
ATmega644	RES	INT0	INT1	INT2	PCI0	PCI1	PCI2	PCI3	WDT	TC2A	TC2B	TC2O	TC1P	TC1A	TC1B	TC1O
ATmega645	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega649	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega1280	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O
ATmega1281	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O

AVR-Typ/Adr	0000	0002	0004	0006	0008	000A	000C	000E	0010	0012	0014	0016	0018	001A	001C	001E
ATmega2560	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O
ATmega2561	RES	INT0	INT1	INT2	INT3	INT4	INT5	INT6	INT7	PCI0	PCI1	PCI2	WDT	TC2A	TC2B	TC2O
ATmega3250	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega3290	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega6450	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX
ATmega6490	RES	INT0	PCI0	PCI1	TC2C	TC2O	TC1P	TC1A	TC1B	TC1O	TC0C	TC0O	SPIS	U0RX	U0UD	U0TX

Interrupts mit den Vektoren von 0020 bis 003E

AVR-Typ/Adr	0020	0022	0024	0026	0028	002A	002C	002E	0030	0032	0034	0036	0038	003A	003C	003E
AT90CAN32	TC0C	TC0O	CANI	OVRI	SPIS	U0RX	U0UD	U0TX	ANAC	ADC	EERY	TC3P	TC3A	TC3B	TC3C	TC3O
AT90CAN64	TC0C	TC0O	CANI	OVRI	SPIS	U0RX	U0UD	U0TX	ANAC	ADC	EERY	TC3P	TC3A	TC3B	TC3C	TC3O
AT90CAN128	TC0C	TC0O	CANI	OVRI	SPIS	U0RX	U0UD	U0TX	ANAC	ADC	EERY	TC3P	TC3A	TC3B	TC3C	TC3O
ATmega16	ANA	TWI	INT2	TC0C	SPMR											
ATmega32	ADC	EERY	ANA	TWI	SPMR											
ATmega48	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA	TWI	SPMR						
ATmega64	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA	TC1C	TC3P	TC3A	TC3B	TC3C	TC3O	U1RX	U1UD
ATmega88	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA	TWI	SPMR						
ATmega103	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA								
ATmega128	TC0O	SPIS	U0TX	U0UD	U0TX	ADC	EERY	ANA	TC1C	TC3P	TC3A	TC3B	TC3C	TC3O	U1RX	U1UD
ATmega161	U1UD	U0TX	U1TX	EERY	ANA											
ATmega162	TC0C	TC0O	SPIS	U0RX	U1RX	U0UD	U1UD	U0TX	U1TX	EERY	ANA	SPMR				
ATmega163	ANA	TWI														
ATmega164	TC0A	TC0B	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	ADC	EERY	TWI					
ATmega165	USIS	USIO	ANA	ADC	EERY	SPMR										
ATmega168	TC0O	SPIS	U0RX	U0UD	U0TX	ADC	EERY	ANA	TWI	SPMR						
ATmega169	USIS	USIO	ANA	ADC	EERY	SPMR	LCD									
ATmega323	ADC	EERY	ANA	TWSI												
ATmega324	TC0A	TC0B	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	ADC	EERY	TWI					
ATmega325	USIS	USIO	ANA	ADC	EERY	SPMR										

AVR-Typ/Adr	0020	0022	0024	0026	0028	002A	002C	002E	0030	0032	0034	0036	0038	003A	003C	003E
ATmega329	USIS	USIO	ANA	ADC	EERY	SPMR	LCD									
ATmega406	TWI	VADC	CADC	CADR	EERY	SPMR										
ATmega640	TC1P	TC1A	TC1B	TC1C	TC1O	TC0A	TC0B	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	ADC	EERY	TC3P
ATmega644	TC0A	TC0B	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	ADC	EERY	TWI					
ATmega645	USIS	USIO	ANA	ADC	EERY	SPMR										
ATmega649	USIS	USIO	ANA	ADC	EERY	SPMR	LCD									
ATmega1280	TC1P	TC1A	TC1B	TC1C	TC1O	TC0A	TC0B	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	ADC	EERY	TC3P
ATmega1281	TC1P	TC1A	TC1B	TC1C	TC1O	TC0A	TC0B	TC0O	SPIS	U0TX	U0UD	U0TX	ANA	ADC	EERY	TC3P
ATmega2560	TC1P	TC1A	TC1B	TC1C	TC1O	TC0A	TC0B	TC0O	SPIS	U0RX	U0UD	U0TX	ANA	ADC	EERY	TC3P
ATmega2561	TC1P	TC1A	TC1B	TC1C	TC1O	TC0A	TC0B	TC0O	SPIS	U0TX	U0UD	U0TX	ANA	ADC	EERY	TC3P
ATmega3250	USIS	USIO	ANA	ADC	EERY	SPMR		PCI2	PCI3							
ATmega3290	USIS	USIO	ANA	ADC	EERY	SPMR	LCD	PCI2	PCI3							
ATmega6450	USIS	USIO	ANA	ADC	EERY	SPMR		PCI2	PCI3							
ATmega6490	USIS	USIO	ANA	ADC	EERY	SPMR	LCD	PCI2	PCI2							

Interrupts mit den Vektoren 0040 bis 005F

AVR-Typ/Adr	0040	0042	0044	0046	0048	004A	004C	004E	0050	0052	0054	0056	0058	005A	005C	005E
AT90CAN32	U1RX	U1UD	U1TX	TWI	SPMR											
AT90CAN64	U1RX	U1UD	U1TX	TWI	SPMR											
AT90CAN128	U1RX	U1UD	U1TX	TWI	SPMR											
ATmega64	U1TX	TWI	SPMR													
ATmega128	U1TX	TWI	SPMR													
ATmega1280	TC3A	TC3B	TC3C	TC3O	U1RX	U1UD	U1TX	TWI	SPMR	TC4P	TC4A	TC4B	TC4C	TC4O	TC5P	TC5A
ATmega1281	TC3A	TC3B	TC3C	TC3O	U1RX	U1UD	U1TX	TWI	SPMR	TC4P	TC4A	TC4B	TC4C	TC4O	TC5P	TC5A
ATmega2560	TC3A	TC3B	TC3C	TC3O	U1RX	U1UD	U1TX	TWI	SPMR	TC4P	TC4A	TC4B	TC4C	TC4O	TC5P	TC5A
ATmega2561	TC3A	TC3B	TC3C	TC3O	U1RX	U1UD	U1TX	TWI	SPMR	TC4P	TC4A	TC4B	TC4C	TC4O	TC5P	TC5A

Interrupts mit den Vektoren 0060 bis 0070

AVR-Typ/Adr	0060	0062	0064	0066	0068	006A	006C	006E	0070

ATmega1280	TC5B	TC5C	TC5O	U2RX	U2UD	U2TX	U3RX	U3UD	U3TX
ATmega1281	TC5B	TC5C	TC5O						
ATmega2560	TC5B	TC5C	TC5O	U2RX	U2UD	U2TX	U3RX	U3UD	U3TX
ATmega2561	TC5B	TC5C	TC5O						

12 Rechnen in Assemblersprache

Hier gibt es alles Wichtige zum Rechnen in Assembler. Dazu gehören die gebräuchlichen Zahlensysteme, das Setzen und Rücksetzen von Bits, das Schieben und Rotieren, das Addieren/Subtrahieren/Vergleichen und die Umwandlung von Zahlen.

12.1 Zahlenarten in Assembler

An Darstellungsarten für Zahlen in Assembler kommen infrage:

- Positive Ganzzahlen,
- Vorzeichenbehaftete ganze Zahlen,
- Binary Coded Digit, BCD,
- Gepackte BCD,
- ASCII-Format.

12.1.1 Positive Ganzzahlen

Die kleinste handhabbare positive Ganzzahl in Assembler ist das Byte zu je acht Bits. Damit sind Zahlen zwischen 0 und 255 darstellbar. Sie passen genau in ein Register des Prozessors. Alle größeren Ganzzahlen müssen auf dieser Einheit aufbauen und sich aus mehreren solcher Einheiten zusammensetzen. So bilden zwei Bytes ein Wort (Bereich 0 .. 65.535), drei Bytes ein längeres Wort (Bereich 0 .. 16.777.215) und vier Bytes ein Doppelwort (Bereich 0 .. 4.294.967.295).

Die einzelnen Bytes eines Wortes oder Doppelwortes können über Register verstreut liegen, da zur Manipulation ohnehin jedes einzelne Register in seiner Lage angegeben sein muss. Damit wir den Überblick nicht verlieren, könnte z.B. ein Doppelwort so angeordnet sein:

```
.DEF dw0 = r16
.DEF dw1 = r17
.DEF dw2 = r18
.DEF dw3 = r19
```

dw0 bis dw3 liegen jetzt in einer Registerreihe. Soll dieses Doppelwort z.B. zu Programmbeginn auf einen festen Wert (hier: 4.000.000) gesetzt werden, dann sieht das in Assembler so aus:

```
.EQU dwi = 4000000 ; Definieren der Konstanten
    LDI dw0,LOW(dwi) ; Die untersten 8 Bits in R16
    LDI dw1,BYTE2(dwi) ; Bits 8 .. 15 in R17
    LDI dw2,BYTE3(dwi) ; Bits 16 .. 23 in R18
    LDI dw3,BYTE4(dwi) ; Bits 24 .. 31 in R19
```

Damit ist die Zahl in verdauliche Brocken auf die Register aufgeteilt und es darf mit dieser Zahl gerechnet werden.

12.1.2 Vorzeichenbehaftete Zahlen

Manchmal, aber sehr selten, braucht man auch negative Zahlen zum Rechnen. Die kriegt man definiert, indem das höchstwertigste Bit eines Bytes als Vorzeichen interpretiert wird. Ist es Eins, dann ist die Zahl negativ. In diesem Fall werden alle Zahlenbits mit ihrem invertierten Wert dargestellt. Invertiert heißt, dass

-1 zu binär 1111.1111 wird, die 1 also als von binär 1.0000.0000 abgezogen dargestellt wird. Das vorderste Bit ist dabei aber das Vorzeichen, das signalisiert, dass die Zahl negativ ist und die folgenden Bits die Zahl invertiert darstellen. Einstweilen genügt es zu verstehen, dass beim binären Addieren von +1 (0000.0001) und -1 (1111.1111) ziemlich exakt Null herauskommt, wenn man von dem gesetzten Übertragsbit Carry mal absieht.

In einem Byte sind mit dieser Methode die Ganzzahlen von +127 (binär: 0111.1111) bis -128 (binär: 1000.000) darstellbar. In Hochsprachen spricht man von Short-Integer.

Benötigt man größere Zahlenbereiche, dann kann man weitere Bytes hinzufügen. Dabei enthält nur das jeweils höchstwertigste Byte das Vorzeichenbit für die gesamte Zahl. Mit zwei Bytes ist damit der Wertebereich von +32767 bis -32768 (Hochsprachen: Integer), mit vier Bytes der Wertebereich von +2.147.483.647 bis -2.147.483.648 darstellbar (Hochsprachen: LongInt).

12.1.3 Binary Coded Digit, BCD

Die beiden vorgenannten Zahlenarten nutzen die Bits der Register optimal aus, indem sie die Zahlen in binärer Form behandeln. Man kann Zahlen aber auch so darstellen, dass auf ein Byte nur jeweils eine dezimale Ziffer kommt. Eine dezimale Ziffer wird dazu in binärer Form gespeichert. Da die Ziffern von 0 bis 9 mit vier Bits darstellbar sind und selbst in den vier Bits noch Luft ist (in vier Bits würde dezimal 0 .. 15 passen), bleibt dabei ziemlich viel Raum leer. Für das Speichern der Zahl 250 werden schon drei Register gebraucht, also z.B. so:

Bit	***7***	***6***	***5***	***4***	***3***	***2***	***1***	***0***
Wertigkeit	***128***	***64***	***32***	***16***	***8***	***4***	***2***	***1***
R16, Ziffer 1	0	0	0	0	0	0	1	0
R17, Ziffer 2	0	0	0	0	0	1	0	1
R18, Ziffer 3	0	0	0	0	0	0	0	0

Instruktionen zum Setzen:

```
LDI R16,2
LDI R17,5
LDI R18,0
```

Auch mit solchen Zahlenformaten lässt sich rechnen, nur ist es aufwendiger als bei den binären Formen. Der Vorteil ist, dass solche Zahlen mit fast beliebiger Größe (soweit das SRAM reicht ...) gehandhabt werden können und dass sie leicht in Zeichenketten umwandelbar sind.

12.1.4 Gepackte BCD-Ziffern

Nicht ganz so verschwenderisch geht gepacktes BCD mit den Ziffern um. Hier wird jede binär kodierte Ziffer in jeweils vier Bits eines Bytes gepackt, so dass ein Byte zwei Ziffern aufnehmen kann. Die beiden Teile des Bytes werden oberes und unteres Nibble genannt. Packt man die höherwertige Ziffer in die oberen vier Bits des Bytes (Bit 4 bis 7), dann hat das beim Rechnen Vorteile (es gibt spezielle Einrichtungen im AVR zum Rechnen mit gepackten BCD-Ziffern). Die schon erwähnte Zahl 250 würde im gepackten BCD-Format folgendermaßen aussehen:

Byte	Ziffern	Wert	8	4	2	1	8	4	2	1
2	4 und 3	02	0	0	0	0	0	0	1	0
1	2 und 1	50	0	1	0	1	0	0	0	0

Instruktionen zum Setzen:

```
LDI R17,0x02 ; Oberes Byte
LDI R16,0x50 ; Unteres Byte
```

Zum Setzen ist nun die binäre (0b...) oder die headezimale (0x...) Schreibweise erforderlich, damit die Bits an die richtigen Stellen im oberen Nibble kommen.

Das Rechnen mit gepackten BCD ist etwas umständlicher im Vergleich zum Binär-Format, die Zahlenumwandlung in darstellbare Zeichenketten aber fast so einfach wie im ungepackten BCD-Format. Auch hier lassen sich fast beliebig lange Zahlen handhaben.

12.1.5 Zahlen im ASCII-Format

Sehr eng verwandt mit dem ungepackten BCD-Format ist die Speicherung von Zahlen im ASCII-Format. Dabei werden die Ziffern 0 bis 9 mit ihrer ASCII-Kodierung gespeichert (ASCII = American Standard Code for Information Interchange). Das ASCII ist ein uraltes, aus Zeiten des mechanischen Fernschreibers stammendes, sehr umständliches, äußerst beschränktes und von höchst innovativen Betriebssystem-Herstellern in das Computer-Zeitalter herüber gerettetes sieben-bittiges Format, mit dem zusätzlich zu irgendwelchen Instruktionen der Übertragungssteuerung beim Fernschreiber (z.B. EOT = End Of Transmission) auch Buchstaben und Zahlen darstellbar sind. Es wird in seiner Altertümlichkeit nur noch durch den (ähnlichen) fünfbittigen Baudot-Code für deutsche Fernschreiber und durch den Morse-Code für ergraute Marinefunker übertroffen.

In diesem Code-System wird die 0 durch die dezimale Ziffer 48 (hexadezimal: 30, binär: 0011.0000), die 9 durch die dezimale Ziffer 57 (hexadezimal: 39, binär: 0011.1001) repräsentiert. Auf die Idee, diese Ziffern ganz vorne im ASCII hinzulegen, hätte man schon kommen können, aber da waren schon die wichtigen Verkehrs-Steuerzeichen für den Fernschreiber. So müssen wir uns noch immer damit herumschlagen, 48 zu einer BCD-kodierten Ziffer hinzu zu zählen oder die Bits 4 und 5 auf eins zu setzen, wenn wir den ASCII-Code über die serielle Schnittstelle senden wollen. Zur Platzverschwendung gilt das schon zu BCD geschriebene. Zum Laden der Zahl 250 kommt diesmal der folgende Quelltext zum Tragen:

```
LDI R18,'2'
LDI R17,'5'
LDI R16,'0'
```

Das speichert direkt die ASCII-Kodierung in das jeweilige Register.

12.2 Bitmanipulationen

Um eine BCD-kodierte Ziffer in ihr ASCII-Pendant zu verwandeln, müssen die Bits 4 und 5 der Zahl auf Eins gesetzt werden. Das verlangt nach einer binären Oder-Verknüpfung und ist eine leichte Aufgabe. Die geht so (ORI geht nur mit Registern ab R16 aufwärts):

```
ORI R16,0x30
```

Steht ein Register zur Verfügung, in dem bereits 0x30 steht, hier R2, dann kann man das Oder auch mit diesem Register durchführen:

```
OR R16,R2
```

Zurück ist es schon schwieriger, weil die naheliegende, umgekehrt wirkende Instruktion,

```
ANDI R16,0x0F
```

die die oberen vier Bits des Registers auf Null setzt und die unteren vier Bits beibehält, nur für Register oberhalb R15 möglich ist. Eventuell also in einem solchen Register durchführen!

Hat man die 0x0F schon in Register R2, kann man mit diesem Register Und-verknüpfen:

```
AND R1,R2
```

Auch die beiden anderen Instruktionen zur Bitmanipulation, CBR und SBR, lassen sich nur in Registern oberhalb von R15 durchführen. Sie würden entsprechend korrekt lauten:

```
SBR R16,0b00110000 ; Bits 4 und 5 setzen

CBR R16,0b00110000 ; Bits 4 und 5 löschen
```

Sollen eins oder mehrere Bits einer Zahl umgekehrt werden, bedient man sich gerne des Exklusiv-Oder-Verfahrens, das nur für Register, nicht für Konstanten, zulässig ist:

```
LDI R16,0b10101010 ; Invertieren aller geraden Bits

EOR R1,R16 ; in Register R1 und speichern in R1
```

Sollen alle Bits eines Bytes umgedreht werden, kommt das Einer-Komplement ins Spiel. Mit

```
COM R1
```

wird der Inhalt eines Registers bitweise invertiert, aus Einsen werden Nullen und umgekehrt. So wird aus 1 die Zahl 254, aus 2 wird 253, usw. Anders ist die Erzeugung einer negativen Zahl aus einer Positiven. Hierbei wird das Vorzeichenbit (Bit 7) umgedreht bzw. der Inhalt von Null subtrahiert. Dieses erledigt die Instruktion

```
NEG R1
```

So wird aus +1 (dezimal: 1) -1 (binär 1111.1111), aus +2 wird -2 (binär 1111.1110), usw.

Neben der Manipulation gleich mehrerer Bits in einem Register gibt es das Kopieren eines einzelnen Bits aus dem eigens für diesen Zweck eingerichteten T-Bit des Status-Registers. Mit

```
BLD R1,0
```

wird das T-Bit im Status-Register in das Bit 0 des Registers R1 kopiert und das dortige Bit überschrieben. Das T-Bit kann vorher auf Null oder Eins gesetzt oder aus einem beliebigen anderen Bit-Lagerplatz in einem Register geladen werden:

```
CLT ; T-Bit auf Null setzen, oder

SET ; T-Bit auf Eins setzen, oder

BST R2,2 ; T-Bit aus Register R2, Bit 2, laden
```

12.3 Schieben und Rotieren

Das Schieben von binären Zahlen entspricht dem Multiplizieren und Dividieren mit 2. Beim Schieben gibt es unterschiedliche Mechanismen.

Die Multiplikation einer Zahl mit 2 geht einfach so vor sich, dass alle Bits einer binären Zahl um eine Stelle nach links geschoben werden. In das freie Bit 0 kommt eine Null. Das überzählige ehemalige Bit 7 wird dabei in das Carry-Bit im Status-Register abgeschoben. Der Vorgang wird logisches Links-Schieben genannt.

```
LSL R1
```

Das umgekehrte Dividieren durch 2 heißt Dividieren oder logisches Rechts-Schieben.

```
LSR R1
```

Dabei wird das frei werdende Bit 7 mit einer 0 gefüllt, während das Bit 0 in das Carry geschoben wird. Dieses Carry kann dann zum Runden der Zahl verwendet werden. Als Beispiel wird eine Zahl durch vier dividiert und dabei gerundet.

```
LSR R1 ; Division durch 2

BRCC Div2 ; Springe wenn kein Runden

INC R1 ; Aufrunden

Div2:

LSR R1 ; Noch mal durch 2

BRCC DivE ; Springe wenn kein Runden

INC R1 ; Aufrunden

DivE:
```

Teilen ist also eine einfache Angelegenheit bei Binärzahlen (aber nicht durch 3 oder 5)!

Bei Vorzeichen-behafteten Zahlen würde das Rechtsschieben das Vorzeichen in Bit 7 übel verändern. Das darf nicht sein. Desdewegen gibt es neben dem logischen Rechtsschieben auch das arithmetische Rechtsschieben. Dabei bleibt das Vorzeichenbit 7 erhalten und die Null wird in Bit 6 eingeschoben.

```
ASR R1
```

Wie beim logischen Schieben landet das Bit 0 im Carry.

Wie nun, wenn wir 16-Bit-Zahlen mit 2 multiplizieren wollen? Dann muss das links aus dem untersten Byte herausgeschobene Bit von rechts in das oberste Byte hineingeschoben werden. Das erledigt man durch Rollen. Dabei landet keine Null im Bit 0 des verschobenen Registers, sondern der Inhalt des Carry-Bits.

```
LSL R1 ; Logische Schieben unteres Byte

ROL R2 ; Linksrollen des oberen Bytes
```

Bei der ersten Instruktion gelangt Bit 7 des unteren Bytes in das Carry-Bit, bei der zweiten Instruktion dann in Bit 0 des oberen Bytes. Nach der zweiten Instruktion hängt Bit 7 des oberen Bytes im Carry-Bit herum und wir könnten es ins dritte Byte schieben, usw. Natürlich gibt es das Rollen auch nach rechts, zum Dividieren von 16-Bit-Zahlen gut geeignet. Hier nun alles Rückwärts:

```
LSR R2 ; Oberes Byte durch 2, Bit 0 ins Carry

ROR R1 ; Carry in unteres Byte und dabei rollen
```

So einfach ist das mit dem Dividieren bei großen Zahlen. Man sieht sofort, dass Assembler-Dividieren viel schwieriger zu erlernen ist als Hochsprachen-Dividieren, oder?

Gleich vier mal Spezial-schieben kommt jetzt. Es geht um die Nibble gepackter BCD-Zahlen. Wenn man nun mal das obere Nibble anstelle des unteren Nibble braucht, dann kommt man um vier mal Rollen

```
ROR R1

ROR R1
```

```
ROR R1
ROR R1
```

mit einem einzigen

```
SWAP R1
```

herum. Das vertauscht mal eben die oberen vier mit den unteren vier Bits.

12.4 Addition, Subtraktion und Vergleich

Ungeheuer schwierig in Assembler ist Addieren, Dividieren und Vergleichen. Zart-besaitete Anfänger sollten sich an dieses Kapitel nicht herantrauen. Wer es trotzdem liest, ist übermütig und jedenfalls selbst schuld.

Um es gleich ganz schwierig zu machen, addieren wir die 16-Bit-Zahlen in den Registern R1:R2 zu den Inhalten von R3:R4 (Das „:“ heißt nicht Division! Das erste Register gibt das High-Byte, das zweite nach dem „:“ das Low-Byte an).

```
ADD R2,R4 ; zuerst die beiden Low-Bytes
ADC R1,R3 ; dann die beiden High-Bytes
```

Anstelle von ADD wird beim zweiten Addieren ADC verwendet. Das addiert auch noch das Carry-Bit dazu, falls beim ersten Addieren ein Übertrag stattgefunden hat. Sind sie schon dem Herzkasper nahe?

Wenn nicht, dann kommt jetzt das Subtrahieren. Also alles wieder rückwärts und R3:R4 von R1:R2 subtrahiert.

```
SUB R2,R4 ; Zuerst das Low-Byte
SBC R1,R3 ; dann das High-Byte
```

Wieder derselbe Trick: Anstelle des SUB das SBC, das zusätzlich zum Register R3 auch gleich noch das Carry-Bit von R1 abzieht. Kriegen Sie noch Luft? Wenn ja, dann leisten sie sich das folgende: Abziehen ohne Ernst!

Jetzt kommt es knüppeldick: Ist die Zahl in R1:R2 nun größer als die in R3:R4 oder nicht? Also nicht SUB, sondern CP (für ComPare), und nicht SBC, sondern CPC:

```
CP R2,R4 ; Vergleiche untere Bytes
CPC R1,R3 ; Vergleiche obere Bytes
```

Wenn jetzt das Carry-Flag gesetzt ist, kann das nur heißen, dass R3:R4 größer ist als R1:R2. Wenn nicht, dann eben nicht.

Jetzt setzen wir noch einen drauf! Wir vergleichen Register R1 und eine Konstante miteinander: Ist in Register R16 ein binäres Wechselbad gespeichert?

```
CPI R16,0xAA
```

Wenn jetzt das Z-Bit gesetzt ist, dann ist das aber so was von gleich!

Und jetzt kommt die Sockenauszieher-Hammer-Instruktion! Wir vergleichen, ob das Register R1 kleiner oder gleich Null ist.

```
TST R1
```

Wenn jetzt das Z-Flag gesetzt ist, ist das Register ziemlich leer und wir können mit BREQ, BRNE, BRMI, BRPL, BRLO, BRSH, BRGE, BRLT, BRVC oder auch BRVS ziemlich lustig springen.

Sie sind ja immer noch dabei! Assembler ist schwer, gelle? Na dann, kriegen sie noch ein wenig gepacktes BCD-Rechnen draufgepackt.

Beim Addieren von gepackten BCD's kann sowohl die unterste der beiden Ziffern als auch die oberste überlaufen. Addieren wir im Geiste die BCD-Zahlen 49 (=hex 49) und 99 (=hex 99).

ADDITION	Oberes Nibble	Unteres Nibble
Packed BCD Zahl 1	4	9
Packed BCD Zahl 2	9	9
Ergebnis	E	2

Beim Addieren in hex kommt hex E2 heraus und es kommt kein Byte-Überlauf zustande. Die untersten beiden Ziffern sind beim Addieren übergelaufen (9+9=18 = hex 12). Folglich ist die oberste Ziffer korrekt um eins erhöht worden, aber die unterste stimmt nicht, sie müsste 8 statt 2 lauten. Also könnten wir unten 6 addieren, dann stimmt es wieder.

Unteres Nibble korrigieren	Oberes Nibble	Unteres Nibble
Ergebnis	E	2
Korrektur	0	6
Ergebnis	E	8

Die oberste Ziffer stimmt überhaupt nicht, weil hex E keine zulässige BCD-Ziffer ist. Sie müsste richtigerweise 4 lauten (4+9+1=14) und ein Überlauf sollte auftreten. Also, wenn zu E noch 6 addiert werden, kommt dezimal 20 bzw. hex 14 heraus. Alles ganz easy: Einfach zum Ergebnis noch hex 66 addieren und schon stimmt alles.

Oberes Nibble korrigieren	Unteres Nibble oberes Byte	Oberes Nibble	Unteres Nibble
Ergebnis		E	8
Korrektur		6	0
Ergebnis	+ 1	4	8

Warum dann nicht gleich hex 66 addieren?

Aber gemach! Das wäre nur korrekt, wenn bei der hintersten Ziffer, wie in unserem Fall, entweder schon beim ersten Addieren oder später beim Addieren der 6 tatsächlich ein Überlauf in die nächste Ziffer auftrat. Wenn das nicht so ist, dann darf die 6 nicht addiert werden. Woran ist zu merken, ob dabei ein Übertrag von der niedrigeren in die höhere Ziffer auftrat? Am Halbübertrags-Bit im Status-Register. Dieses H-Bit zeigt für einige Instruktionen an, ob ein solcher Übertrag aus dem unteren in das obere Nibble auftrat. Dasselbe gilt analog für das obere Nibble, nur zeigt hier das Carry-Bit den Überlauf an. Die folgenden Tabellen zeigen die verschiedenen Möglichkeiten an.

ADDR1,R2 (Half)Carry-Bit	ADDNibble,6 (Half)Carry-Bit	Korrektur
0	0	6 wieder abziehen
1	0	keine
0	1	keine
1	1	(geht gar nicht!)

Nehmen wir an, die beiden gepackten BCD-Zahlen seien in R2 und R3 gespeichert, R1 soll den Überlauf aufnehmen und R16 und R17 stehen zur freien Verfügung. R16 soll zur Addition von 0x66 dienen (das Register R2 kann keine Konstanten addieren), R17 zur Subtraktion der Korrekturen am Ergebnis. Dann geht das Addieren von R2 und R3 so:

```
        LDI R16,0x66
        LDI R17,0x66
        ADD R2,R3
        BRCC NoCy1
        INC R1
        ANDI R17,0x0F
NoCy1:
        BRHC NoHc1
        ANDI R17,0xF0
NoHc1:
        ADD R2,R16
        BRCC NoCy2
        INC R1
        ANDI R17,0x0F
NoCy2:
        BRHC NoHc2
        ANDI R17,0x0F
NoHc2:
        SUB R2,R17
```

Die einzelnen Schritte: Im ersten Schritt werden die beiden Zahlen addiert. Tritt dabei schon ein Carry auf, dann wird das Ergebnisregister R1 erhöht und eine Korrektur des oberen Nibbles ist nicht nötig (die obere 6 im Korrekturspeicher R16 wird gelöscht). INC und ANDI beeinflussen das H-Bit nicht. War nach der ersten Addition das H-Bit schon gesetzt, dann kann auch die Korrektur des unteren Nibble entfallen (das untere Nibble wird Null gesetzt). Dann wird 0x66 addiert. Tritt dabei nun ein Carry auf, dann wird wie oben verfahren. Trat dabei ein Half-Carry auf, dann wird ebenfalls wie oben verfahren. Schließlich wird das Korrektur-Register vom Ergebnis abgezogen und die Berechnung ist fertig.

Kürzer geht es so.

```
        LDI R16,0x66
        ADD R2,R16
        ADD R2,R3
        BRCC NoCy
        INC R1
        ANDI R16,0x0F
```

```
NoCy:
        BRHC NoHc
        ANDI R16,0xF0
NoCy:
        SUB R2,R16
```

Ich überlasse es dem Leser zu ergründen, warum das auch so kurz geht.

12.5 Umwandlung von Zahlen - generell

Alle Zahlenformate sind natürlich umwandelbar. Die Umwandlung von BCD in ASCII und zurück war oben schon besprochen (Bitmanipulationen).

Die Umwandlung von gepackten BCD's ist auch nicht schwer. Zuerst ist die gepackte BCD mit einem SWAP umzuwandeln, so dass das oberste Digit ganz rechts liegt. Mit einem ANDI kann dann das obere (ehemals untere) Nibble gelöscht werden, so dass das obere Digit als reine BCD blank liegt. Das zweite Digit ist direkt zugänglich, es ist nur das obere Nibble zu löschen. Von einer BCD zu einer gepackten BCD kommt man, indem man die höherwertige BCD mit SWAP in das obere Nibble verfrachtet und anschließend die niederwertige BCD damit verODERt.

Etwas schwieriger ist die Umwandlung von BCD-Zahlen in eine Binärzahl. Dazu macht man zuerst die benötigten Bits im Ergebnisspeicher frei. Man beginnt mit der niedrigstwertigen BCD-Ziffer. Bevor man diese zum Ergebnis addiert, wird das Ergebnis erstmal mit 10 multipliziert. Dazu speichert man das Ergebnis irgendwo zwischen, multipliziert es mit 4 (zweimal links schieben/rotieren), addiert das alte Ergebnis (mal 5) und schiebt noch einmal nach links (mal 10). Jetzt erst wird die BCD-Ziffer addiert. Das macht man mit allen höherwertigen BCD-Ziffern genauso weiter. Tritt bei irgendeinem Schieben oder Addieren des obersten Bytes ein Carry auf, dann passt die BCD-Zahl nicht in die verfügbaren binären Bytes.

Die Verwandlung einer Binärzahl in BCD-Ziffern ist noch etwas schwieriger. Handelt es sich um 16-Bit-Zahlen, kann man solange 10.000 subtrahieren, bis ein Überlauf auftritt, das ergibt die erste BCD-Ziffer. Anschließend subtrahiert man 1.000 bis zum Überlauf und erhält die zweite Ziffer, usw. bis 10. Der Rest ist die letzte Ziffer.Die Ziffern 10.000 bis 10 kann man im Programmspeicher in einer wortweise organisierten Tabelle verewigen, z.B. so

```
DezTab:

.DW 10000, 1000, 100, 10
```

und wortweise mit der LPM-Instruktion aus der Tabelle herauslesen in zwei Register.

Eine Alternative ist eine Tabelle mit der Wertigkeit jedes einzelnen Bits einer 16-Bit-Zahl, also z.B.

```
.DB 0,3,2,7,6,8

.DB 0,1,6,3,8,4

.DB 0,0,8,1,9,2

.DB 0,0,4,0,9,6

.DB 0,0,2,0,4,8 ; und so weiter bis

.DB 0,0,0,0,0,1
```

Dann wären die einzelnen Bits der 16-Bit-Zahl nach links herauszuschieben und, wenn es sich um eine 1 handelt, der entsprechende Tabellenwert per LPM zu lesen und zu den Ergebnisbytes zu addieren. Ein vergleichsweise aufwendigeres und langsameres Verfahren.

Eine dritte Möglichkeit wäre es, die fünf zu addierenden BCD-Ziffern beginnend mit 00001 durch Multiplikation mit 2 bei jedem Durchlauf zu erzeugen und mit dem Schieben der umzuwandelnden Zahl beim untersten Bit zu beginnen.

Es gibt viele Wege nach Rom, und manche sind mühsamer.

Weitere, ausgearbeitete Software zur Zahlenumwandlung gibt es im Kapitel 11.9.

12.6 Multiplikation

Dezimales Multiplizieren

Zwei 8-Bit-Zahlen sollen multipliziert werden. Man erinnere sich, wie das mit Dezimalzahlen geht:

```
    1234 * 567 = ?
--------------------------
    1234 *   7 =    8638
+  12340 *  6  =   74040
+ 123400 * 5   =  617000
--------------------------
    1234 * 567 =  699678
==========================
```

Also in Einzelschritten dezimal:

1. Wir nehmen die Zahl mit der kleinsten Ziffer mal und addieren sie zum Ergebnis.
2. Wir nehmen die Zahl mit 10 mal, dann mit der nächsthöheren Ziffer, und addieren sie zum Ergebnis.
3. Wir nehmen die Zahl mit 100 mal, dann mit der dritthöchsten Ziffer, und addieren sie zum Ergebnis.

Binäres Multiplizieren

Jetzt in Binär: Das Malnehmen mit den Ziffern entfällt, weil es ja nur Null oder Eins gibt, also entweder wird die Zahl addiert (1) oder eben nicht (0). Das Malnehmen mit 10 wird in binär zum Malnehmen mit 2, weil wir ja nicht mit Basis 10 sondern mit Basis 2 rechnen. Malnehmen mit 2 ist aber ganz einfach: man kann die Zahl einfach mit sich selbst addieren oder binär nach links schieben und rechts eine binäre Null dazu schreiben. Man sieht schon, dass das binäre Multiplizieren viel einfacher ist als das dezimale Multiplizieren. Man fragt sich, warum die Menschheit so ein schrecklich kompliziertes Dezimalsystem erfinden musste und nicht beim binären System verweilt ist.

AVR-Assemblerprogramm

Die rechentechnische Umsetzung in AVR-Assembler-Sprache zeigt der Quelltext.

```
; Mult8.asm multipliziert zwei 8-Bit-Zahlen
; zu einem 16-Bit-Ergebnis
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
; Ablauf des Multiplizierens:
; 1.Multiplikator 2 wird bitweise nach rechts in das Carry-Bit geschoben.
;   Wenn es eine Eins ist, dann wird die Zahl in Multiplikator 1 zum Ergebnis dazu gezählt,
;   wenn es eine Null ist, dann nicht.
; 2.Nach dem Addieren wird Multiplikator 1 durch Links schieben mit 2 multipliziert.
; 3.Wenn Multiplikator 2 nach dem Schieben nicht Null ist, dann wird wie oben weiter
;   gemacht. Wenn er Null ist, ist die Multiplikation beendet.
```

```
; Benutzte Register
.DEF rm1 = R0 ; Multiplikator 1 (8 Bit)
.DEF rmh = R1 ; Hilfsregister für Multiplikation
.DEF rm2 = R2 ; Multiplikator 2 (8 Bit)
.DEF rel = R3 ; Ergebnis, LSB (16 Bit)
.DEF reh = R4 ; Ergebnis, MSB
.DEF rmp = R16 ; Hilfsregister zum Laden
;
.CSEG
.ORG 0000
        rjmp START
START:
        ldi rmp,0xAA ; Beispielzahl 1010.1010
        mov rm1,rmp  ; in erstes Multiplikationsregister
        ldi rmp,0x55 ; Beispielzahl 0101.0101
        mov rm2,rmp  ; in zweites Multiplikationsregister
; Hier beginnt die Multiplikation der beiden Zahlen
; in rm1 und rm2, das Ergebnis ist in reh:rel (16 Bit)
MULT8:
; Anfangswerte auf Null setzen
        clr rmh ; Hilfsregister leeren
        clr rel ; Ergebnis auf Null setzen
        clr reh
; Hier beginnt die Multiplikationsschleife
MULT8a:
; Erster Schritt: Niedrigstes Bit von Multplikator 2
; in das Carry-Bit schieben (von links Nullen nachschieben)
        clc ; Carry-Bit auf Null setzen
        ror rm2 ; Null links rein, alle Bits eins rechts,
                ; niedrigstes Bit in Carry schieben
; Zweiter Schritt: Verzweigen je nachdem ob eine Null oder
; eine Eins im Carry steht
        brcc MULT8b ; springe, wenn niedrigstes Bit eine
                    ; Null ist, über das Addieren hinweg
; Dritter Schritt: Addiere 16 Bits in rmh:rm1 zum Ergebnis
; in reh:rel (mit Überlauf der unteren 8 Bits!)
        add rel,rm1 ; addiere LSB von rm1 zum Ergebnis
        adc reh,rmh ; addiere Carry und MSB von rm1
MULT8b:
; Vierter Schritt: Multipliziere Multiplikator rmh:rm1 mit
; Zwei (16-Bit, links schieben)
        clc ; Carry bit auf Null setzen
        rol rm1 ; LSB links schieben (multiplik. mit 2)
        rol rmh ; Carry in MSB und MSB links schieben
; Fünfter Schritt: Prüfen, ob noch Einsen im Multi-
; plikator 2 enthalten sind, wenn ja, dann weitermachen
        tst rm2 ; alle bits Null?
        brne MULT8a ; wenn nicht null, dann weitermachen
; Ende der Multiplikation, Ergebnis in reh:rel
; Endlosschleife
LOOP:
        rjmp loop
```

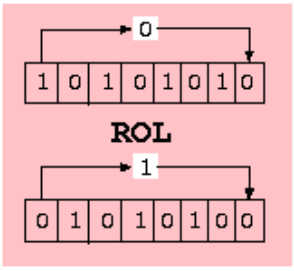
Zu Beginn der Berechnung liegen die folgenden Bedingungen vor:

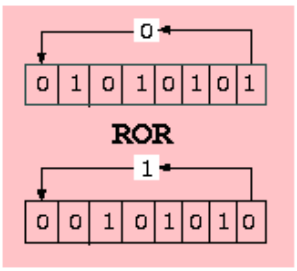
	rmh = R1 = 0x00								***rm1 = R0 = 0xAA***							
Z1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
*****									***rm2 = R2 = 0x55***							
Z2									0	1	0	1	0	1	0	1
=	***reh = R4 = 0x00***								***rel = R3 = 0x00***							

Erg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

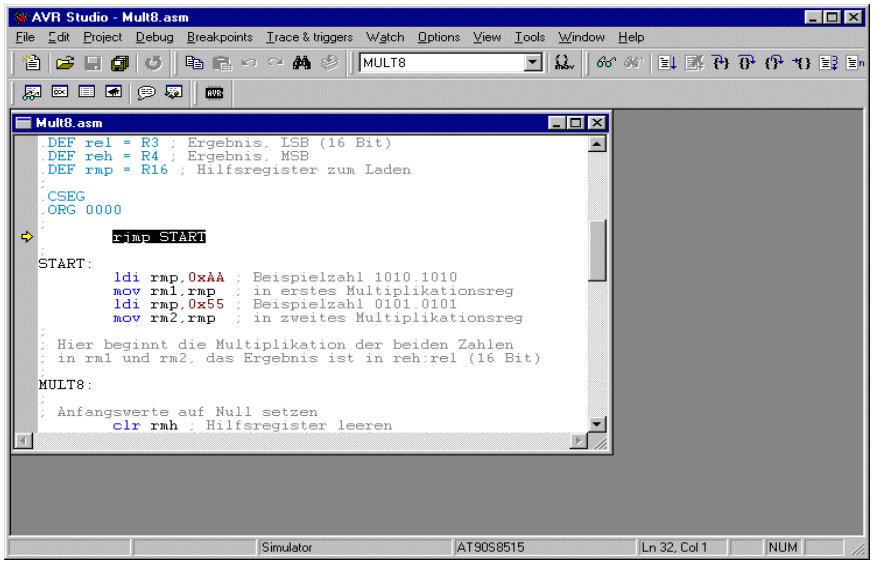
Binäres Rotieren

Für das Verständnis der Berechnung ist die Kenntnis der Assembler-Instruktionen ROL bzw ROR wichtig. Die Instruktion verschiebt die Bits eines Registers nach links (ROL) bzw. rechts (ROR), schiebt das Carry-Bit aus dem Statusregister in die leer werdende Position im Register und schiebt dafür das beim Rotieren herausfallende Bit in das Carry-Flag. Dieser Vorgang wird für das Linksschieben mit dem Inhalt des Registers von 0xAA, für das Rechtsschieben mit 0x55 gezeigt:



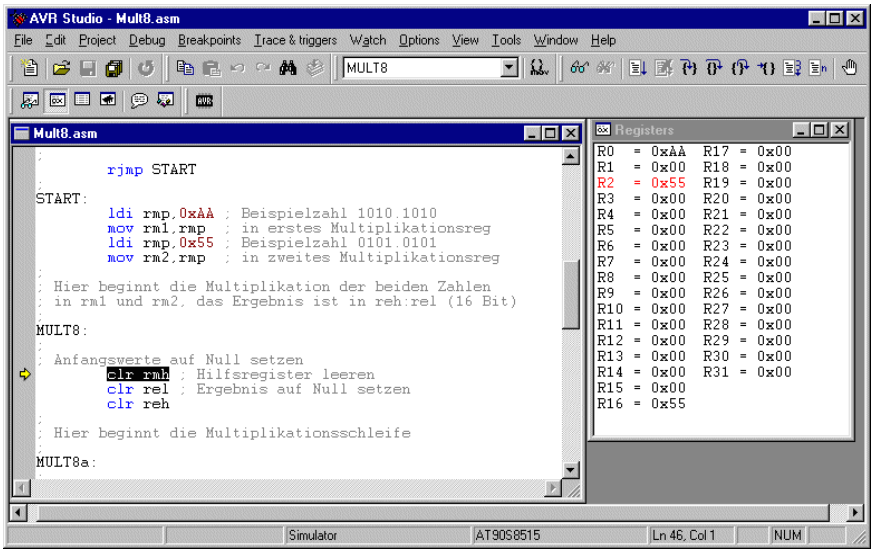


Multiplikation im Studio

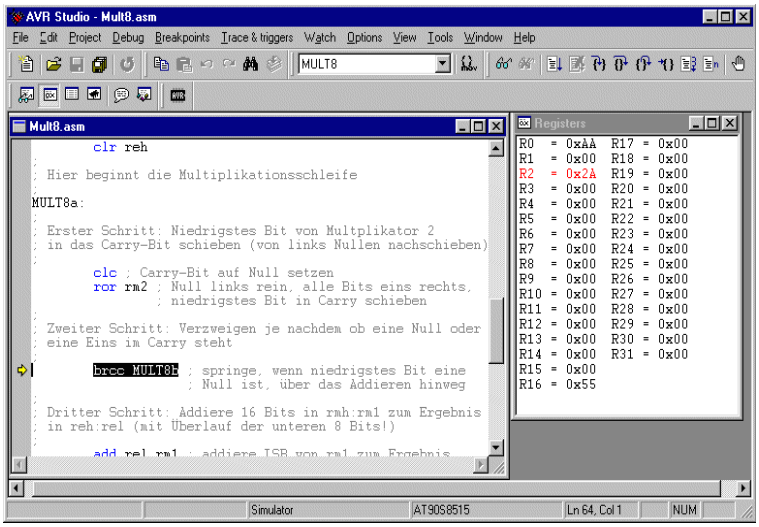


Die folgenden Bilder zeigen die einzelnen Schritte im Studio:

Der Object-Code ist gestartet, der Cursor steht auf der ersten Instruktion. Mit F11 machen wir Einzelschritte.

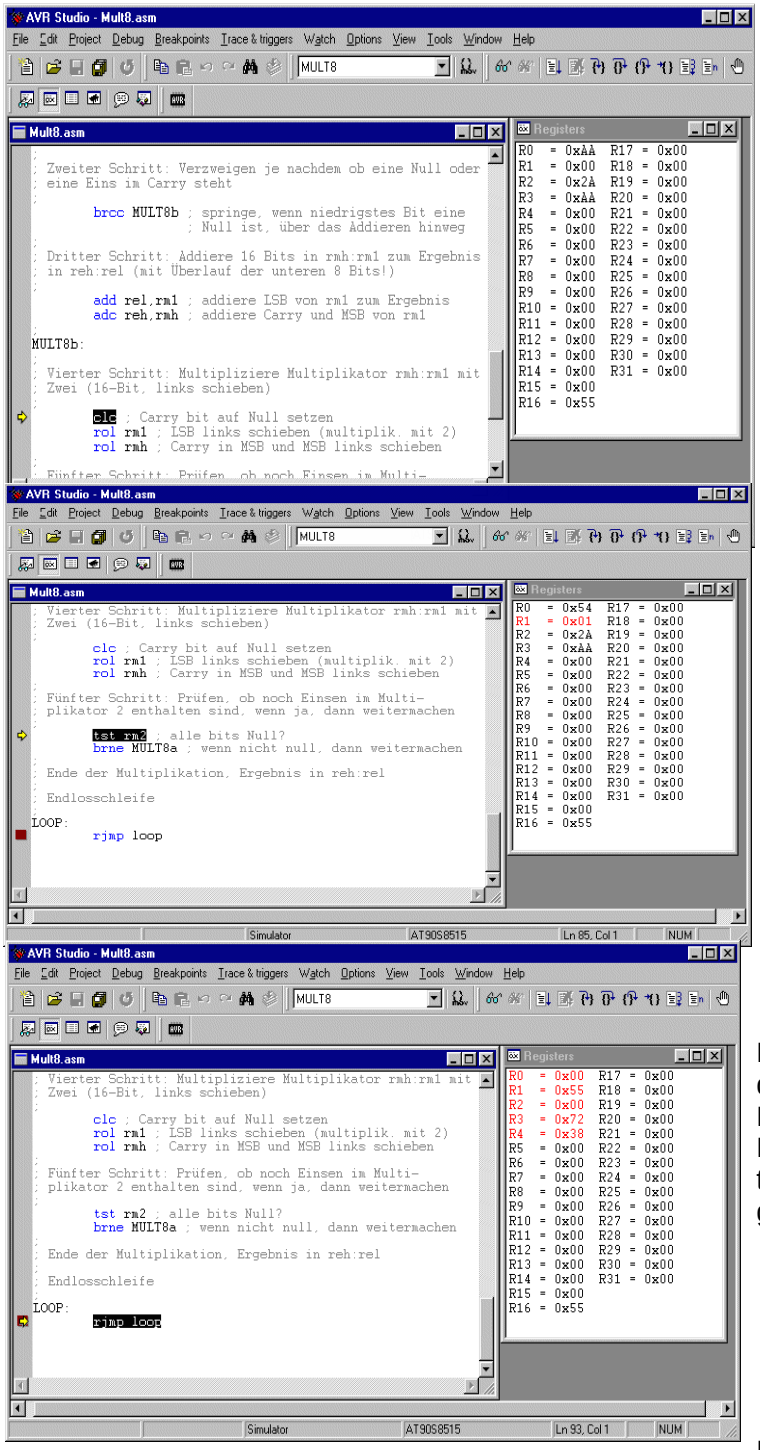


In die Register R0 und R2 werden die beiden Werte AA und 55 geschrieben, die wir multiplizieren wollen.



R2 wurde nach rechts geschoben, um das niedrigste Bit in das Carry-Bit zu schieben. Aus 55 (0101.0101) ist 2A geworden (0010.1010), die rechteste 1 liegt im Carry-Bit des Status-Registers herum.

Weil im Carry eine Eins war, wird 00AA in den Registern R1:R0 zu dem (leeren) Registerpaar R4:R3 hinzu addiert. 00AA steht nun auch dort herum.



Jetzt muss das Registerpaar R1:R0 mit sich selbst addiert oder in unserem Fall einmal nach links geschoben werden. Aus 00AA (0000.0000.1010.1010) wird jetzt 0154 (0000.0001.0101.0100), weil wir von rechts eine Null hinein geschoben haben.

Die gesamte Multiplikation geht solange weiter, bis alle Einsen in R2 durch das Rechtsschieben herausgeflogen sind. Die Zahl ist dann fertig multipliziert, wenn nur noch Nullen in R2 stehen. Die weiteren Schritte sind nicht mehr abgebildet.

Mit F5 haben wir im Simulator den Rest durchlaufen lassen und sind am Ausgang der Multiplikationsroutine angelangt, wo wir einen Breakpoint gesetzt hatten. Das Ergebnisregister R4:R3 enthält nun den Wert 3872, das Ergebnis der Multiplikation von AA mit 55.

Das war sicherlich nicht schwer, wenn man sich die Rechengänge von Dezimal in Binär übersetzt. Binär ist viel einfacher als Dezimal!

12.7 Division

Dezimales Dividieren

Zunächst wieder eine dezimale Division, damit das besser verständlich wird. Nehmen wir an, wir wollen 5678 durch 12 teilen. Das geht dann etwa so:

```
             5678 : 12 = ?
--------------------------
- 4 * 1200 = 4800
            ------
=              878
- 7 *  120 =  840
             -----
=               38
- 3 *   12 =   36
              ---
=                2
Ergebnis: 5678 : 12 = 473 Rest 2
================================
```

Binäres Dividieren

Binär entfällt wieder das Multiplizieren des Divisors (4 * 1200, etc.), weil es nur Einsen und Nullen gibt. Dafür haben binäre Zahlen leider sehr viel mehr Stellen als dezimale. Die direkte Analogie wäre in diesem Fall etwas aufwändig, weshalb die rechentechnische Lösung etwas anders aussieht.

Die Division einer 16-Bit-Zahl durch eine 8-Bit-Zahl in AVR Assembler ist hier als Quellcode gezeigt.

Assembler Quellcode der Division

```
; Div8 dividiert eine 16-Bit-Zahl durch eine 8-Bit-Zahl
; Test: 16-Bit-Zahl: 0xAAAA, 8-Bit-Zahl: 0x55
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
; Registers
.DEF rd1l = R0 ; LSB Divident
.DEF rd1h = R1 ; MSB Divident
.DEF rd1u = R2 ; Hifsregister
.DEF rd2  = R3 ; Divisor
.DEF rel  = R4 ; LSB Ergebnis
.DEF reh  = R5 ; MSB Ergebnis
.DEF rmp  = R16; Hilfsregister zum Laden
.CSEG
.ORG 0
        rjmp start
start:
; Vorbelegen mit den Testzahlen
        ldi rmp,0xAA ; 0xAAAA in Divident
        mov rd1h,rmp
        mov rd1l,rmp
        ldi rmp,0x55 ; 0x55 in Divisor
        mov rd2,rmp
; Divieren von rd1h:rd1l durch rd2
div8:
        clr rd1u ; Leere Hilfsregister
        clr reh  ; Leere Ergebnisregister
        clr rel  ; (Ergebnisregister dient auch als
        inc rel  ; Zähler bis 16! Bit 1 auf 1 setzen)
; Hier beginnt die Divisionsschleife
div8a:
        clc      ; Carry-Bit leeren
        rol rd1l ; nächsthöheres Bit des Dividenten
        rol rd1h ; in das Hilfsregister rotieren
        rol rd1u ; (entspricht Multipliklation mit 2)
        brcs div8b ; Eine 1 ist herausgerollt, ziehe ab
        cp rd1u,rd2 ; Divisionsergebnis 1 oder 0?
```

```
        brcs div8c  ; Überspringe Subtraktion, wenn kleiner
div8b:
        sub rd1u,rd2; Subtrahiere Divisor
        sec      ; Setze carry-bit, Ergebnis ist eine 1
        rjmp div8d  ; zum Schieben des Ergebnisses
div8c:
        clc      ; Lösche carry-bit, Ergebnis ist eine 0
div8d:
        rol rel  ; Rotiere carry-bit in das Ergebnis
        rol reh
        brcc div8a  ; solange Nullen aus dem Ergebnis
                    ; rotieren: weitermachen
; Ende der Division erreicht
stop:
        rjmp stop   ; Endlosschleife
```

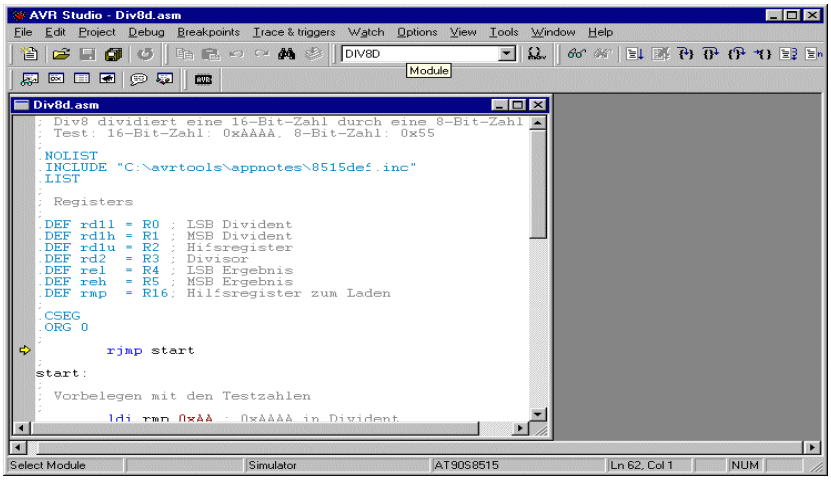
Programmschritte beim Dividieren

Das Programm gliedert sich in folgende Teilschritte:

- Definieren und Vorbelegen der Register mit den Testzahlen,
- das Vorbelegen von Hilfsregistern (die beiden Ergebnisregister werden mit 0x0001 vorbelegt, um das Ende der Division nach 16 Schritten elegant zu markieren!),
- die 16-Bit-Zahl in rd1h:rd1l wird bitweise in ein Hilfsregister rd1u geschoben (mit 2 multipliziert), rollt dabei eine 1 links heraus, dann wird auf jeden Fall zur Subtraktion im vierten Schritt verzweigt,
- der Inhalt des Hilfsregisters wird mit der 8-Bit-Zahl in rd2 verglichen, ist das Hilfsregister größer wird die 8-Bit-Zahl subtrahiert und eine Eins in das Carry-Bit gepackt, ist es kleiner dann wird nicht subtrahiert und eine Null in das Carry-Bit gepackt,
- der Inhalt des Carry-Bit wird in die Ergebnisregister reh:rel von rechts einrotiert,
- rotiert aus dem Ergebnisregister eine Null links heraus, dann muss weiter dividiert werden und ab Schritt 3 wird wiederholt (insgesamt 16 mal), rollt eine 1 heraus, dann sind wir fertig.

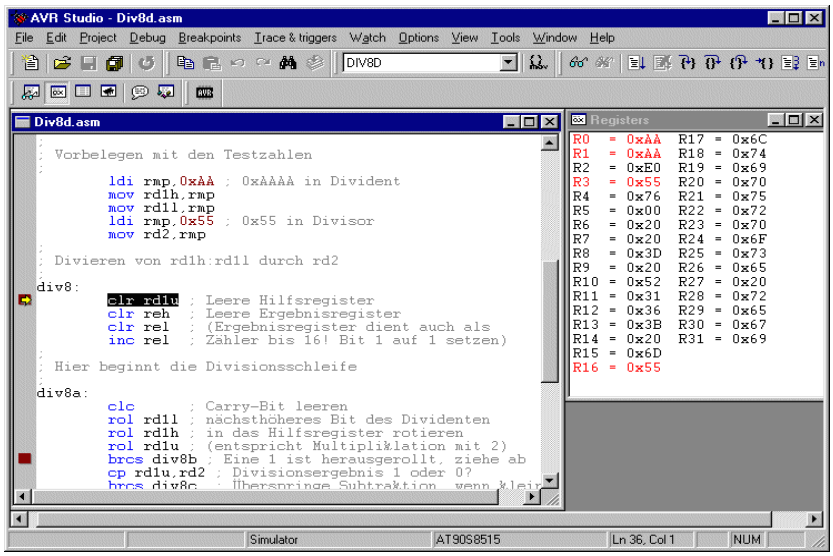
Für das Rotieren gilt das oben dargestellte Procedere (siehe oben, zum Nachlesen).

Das Dividieren im Simulator

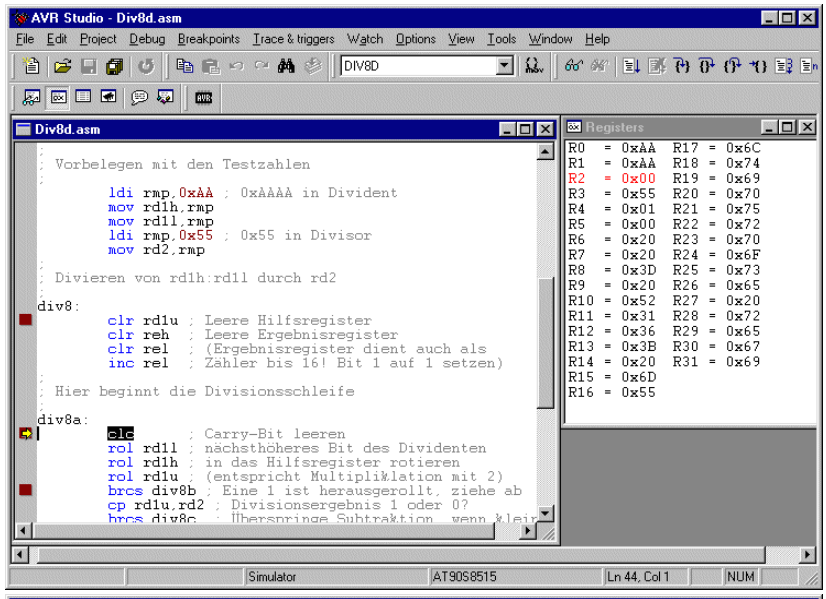


Die folgenden Bilder zeigen die Vorgänge im Simulator, dem Studio. Dazu wird der Quellcode assembliert und die Object-Datei in das Studio geladen.

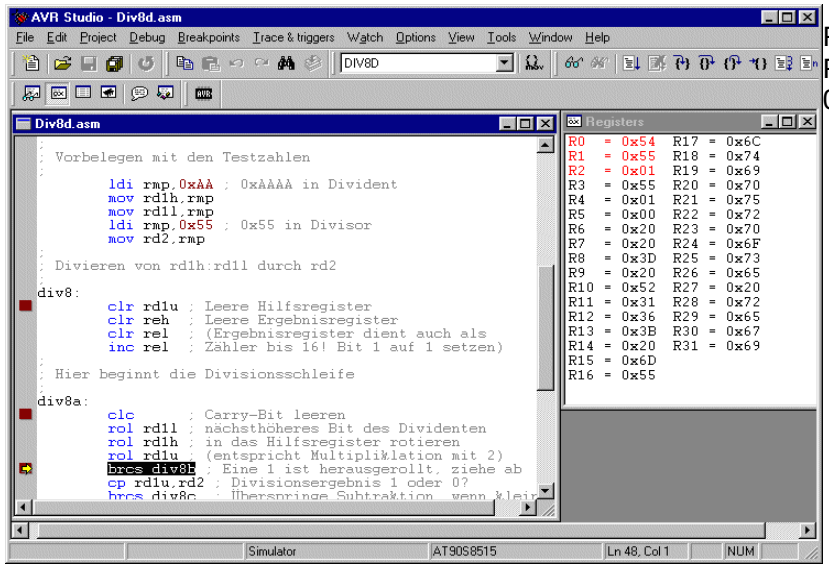
Der Object-Code ist gestartet, der Cursor steht auf der ersten Instruktion. Mit F11 machen wir Einzelschritte.



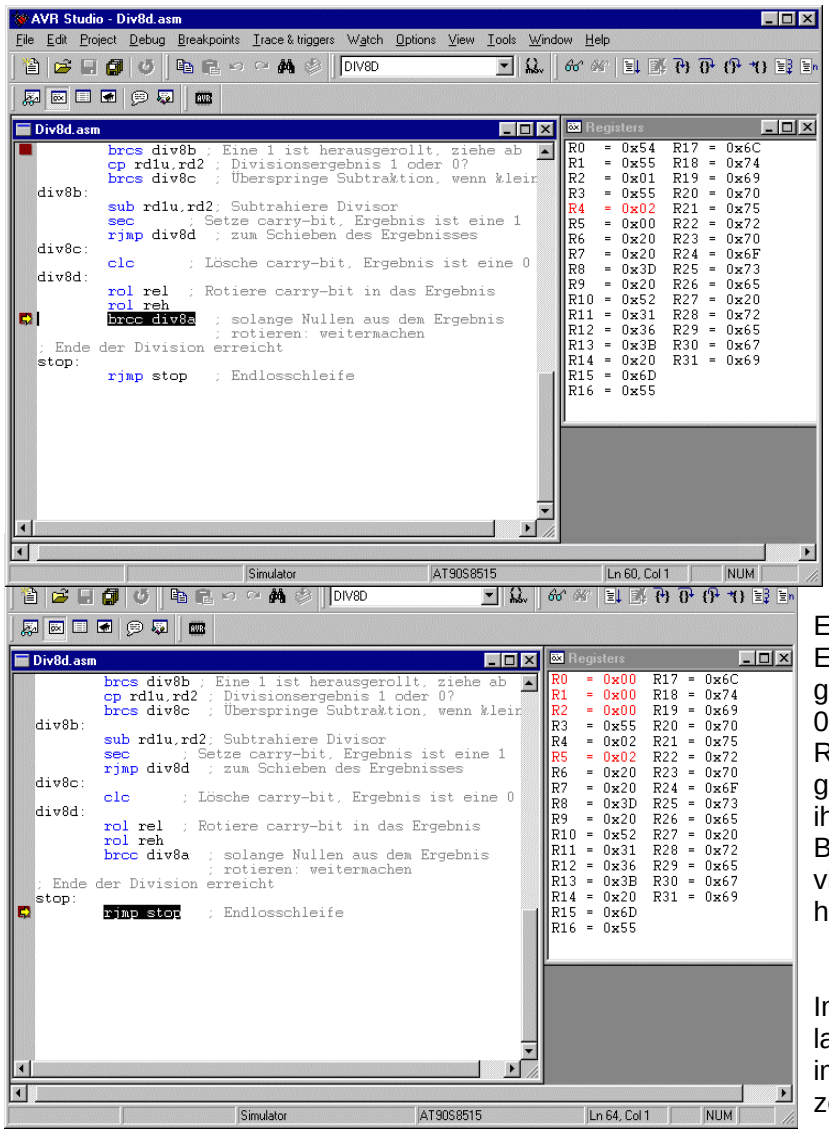
In die Register R0, R1 und R3 werden die beiden Werte 0xAAAA und 0x55 geschrieben, die wir dividieren wollen.



Die Startwerte für das Hilfsregister werden gesetzt, das Ergebnisregisterpaar wurde auf 0x0001 gesetzt.



R1:R0 wurde nach links in Hilfsregister R2 geschoben, aus 0xAAAA ist 0x015554 entstanden.



Weil 0x01 in R2 kleiner als 0x55 in R3 ist, wurde das Subtrahieren übersprungen, eine Null in das Carry gepackt und in R5:R4 geschoben. Aus der ursprünglichen 1 im Ergebnisregister R5:R4 ist durch das Linksrotieren 0x0002 geworden. Da eine Null in das Carry herausgeschoben wurde, geht die nächste Instruktion (BRCC) mit einem Sprung zur Marke div8a und die Schleife wird wiederholt.

Nach dem 16-maligen Durchlaufen der Schleife gelangen wir schließlich an die Endlosschleife am Ende der Division. Im Ergebnisregister R5:R4 steht nun das Ergebnis der Division von 0xAAAA durch 0x55, nämlich 0x0202. Die Register R2:R1:R0 sind leer, es ist also kein Rest geblieben. Bliebe ein Rest, könnten wir ihn mit zwei multiplizieren und mit der 8-Bit-Zahl vergleichen, um das Ergebnis vielleicht noch aufzurunden. Aber das ist hier nicht programmiert.

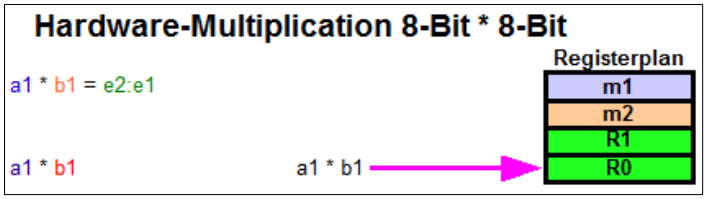
Im Prozessor-View sehen wir nach Ablauf des gesamten Divisionsprozesses immerhin 60 Mikrosekunden Prozessorzeit verbraucht.

12.8 Binäre Hardware-Multiplikation

Alle ATmega, AT90CAN and AT90PWM haben einen Multiplikator an Bord, der 8- mal 8-Bit-Multiplikationen in nur zwei Taktzyklen durchführt. Wenn also Multiplikationen durchgeführt werden müssen und man sicher ist, dass die Software niemals in einem AT90S- oder ATtiny-Prozessor laufen werden muss, sollte man diese schnelle Hardware-Möglichkeit nutzen. Diese folgenden Seiten zeigen, wie man das macht.

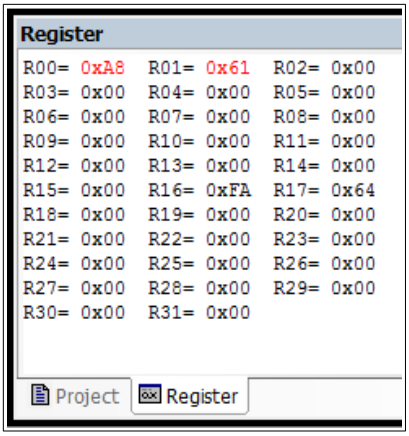
Hardware Multiplikation von 8-mal-8-Bit Binärzahlen

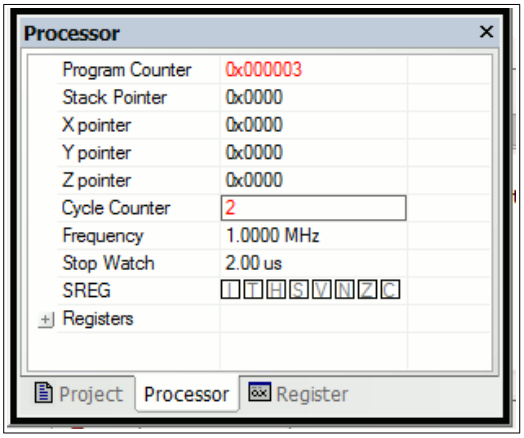
Diese Aufgabe ist einfach und direkt. Wenn die zwei Binärzahlen in den Registern R16 und R17 stehen, dann muss man nur folgende Instruktion eingeben: mul R16,R17. Weil das Ergebnis der Multiplikation bis zu 16 Bit lange Zahlen ergibt, ist das Ergebnis in den Registern R1 (höherwertiges Byte) und R0 (niedrigerwertiges Byte) untergebracht. Das war es auch schon.



```
; Tests 8-by-8-bit hardware multiplication with ATmega8
;
; Define Registers
;
.def ResL = R0
.def ResM = R1
.def m1 = R16
.def m2 = R17
;
; Load multiplicators
;
    ldi m1,250
    ldi m2,100
;
; Perform multiplication
;
    mul m1,m2
;
; 16-bit result is in R1:R0
;
```

Das Programm demonstriert die Simulation im Studio. Es multipliziert dezimal 250 (hex FA) mit dezimal 100 (hex 64) in den Registern R16 und R17.





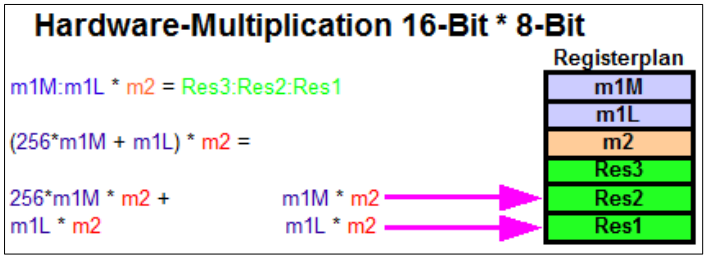
Die Register R0 (LSB) und R1 (MSB) enthalten das Ergebnis hex 61A8 oder dezimal 25,000.

Und: ja, die Multiplikation braucht bloß zwei Zyklen, oder 2 Mikrosekunden bei einem Takt von einem MHz.

Hardware Multiplikation von 16- mit 8-Bit-Zahl

Sind größere Zahlen zu multiplizieren? Die Hardware ist auf 8 Bit beschränkt, also müssen wir ersatzweise einige geniale Ideen anwenden. Das ist an diesem Beispiel 16 mal 8 gezeigt. Verstehen des Konzepts hilft bei der Anwendung der Methode und es ist ein leichtes, das später auf die 32- mit 64-Bit-Multiplikation zu übertragen.

Zuerst die Mathematik: eine 16-Bit-Zahl lässt sich in zwei 8-Bit-Zahlen auftrennen, wobei die höherwertige der beiden Zahlen mit dezimal 256 oder hex 100 mal genommen wird. Für die, die eine Erinnerung brauchen: die Dezimalzahl 1234 kann man auch als die Summe aus (12*100) und 34 betrachten, oder auch als die Summe aus (1*1000), (2*100), (2*10) und 4. Die 16-Bit-Binärzahl m1 ist also gleich 256*m1M plus m1L, wobei



m1M das MSB und m1L das LSB ist.

Multiplikation dieser Zahl mit der 8-Bit-Zahl m2 ist also mathematisch ausgedrückt:

$$m1 * m2 = (256*m1M + m1L) * m2 = 256*m1M*m2 + m1L*m2$$

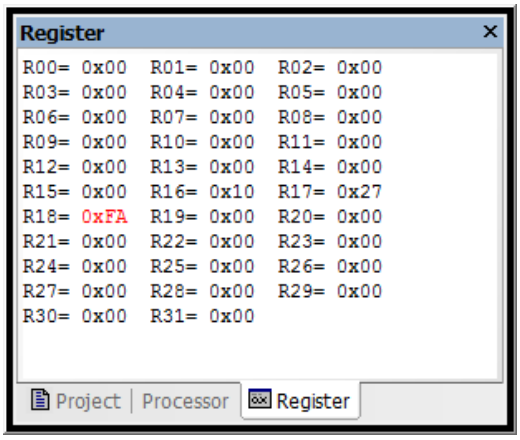
Wir müssen also lediglich zwei Multiplikationen durchführen und die Ergebnisse addieren. Zwei Multiplikationen? Es sind doch drei * zu sehen! Für die Multiplikation mit 256 braucht man in der Binärwelt keinen Hardware-Multiplikator, weil es ausreicht, die Zahl einfach um ein Byte nach links zu rücken. Genauso wie in der Dezimalwelt eine Multiplikation mit 10 einfach ein Linksrücken um eine Stelle ist und die frei werdende leere Ziffer mit Null gefüllt wird. Beginnen wir mit einem praktischen Beispiel. Zuerst brauchen wir einige Register, um

1. die Zahlen m1 und m2 zu laden,
2. für das Ergebnis Raum zu haben, das bis zu 24 Bit lang werden kann.

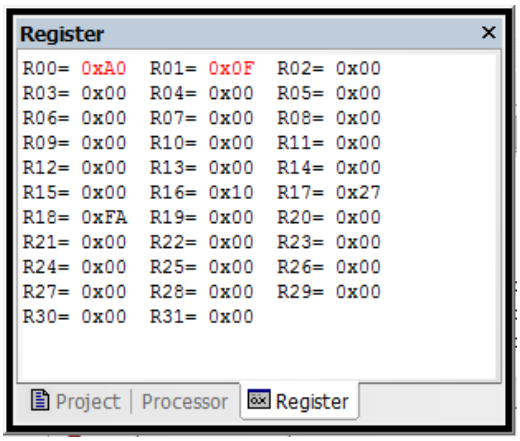
```
;
; Testet Hardware Multiplikation 16-mit-8-Bit
;
; Register Definitionen:
;
.def Res1 = R2 ; Byte 1 des Ergebnisses
.def Res2 = R3 ; Byte 2 des Ergebnisses
.def Res3 = R4 ; Byte 3 des Ergebnisses
.def m1L = R16 ; LSB der Zahl m1
.def m1M = R17 ; MSB der Zahl m1
.def m2 = R18 ; die Zahl m2
```

Zuerst werden die Zahlen in die Register geladen:

```
;
; Lade Register
;
.equ m1 = 10000
;
        ldi m1M,HIGH(m1) ; obere 8 Bits von m1 in m1M
        ldi m1L,LOW(m1) ; niedrigere 8 Bits von m1 in m1L
        ldi m2,250 ; 8-Bit Konstante in m2
```



Die beiden Zahlen sind in R17:R16 (dez 10000 = hex 2710) und R18 (dez 250 = hex FA) geladen.



Dann multiplizieren wir zuerst das niedrigerwertige Byte:

```
;
; Multiplikation
;
        mul m1L,m2 ; Multipliziere LSB
        mov Res1,R0 ; kopiere Ergebnis in Ergebnisregister
```

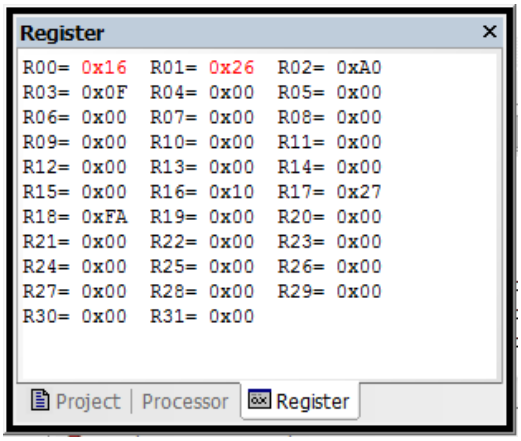
```
        mov Res2,R1
```

Die LSB-Multiplikation von hex 27 mit hex FA ergibt hex 0F0A, das in die Register R0 (LSB, hex A0) und R1 (MSB, hex 0F) geschrieben wurde. Das Ergebnis wird in die beiden untersten Bytes der Ergebnisregister, R3:R2, kopiert.

Nun folgt die Multiplikation des MSB mit m2:

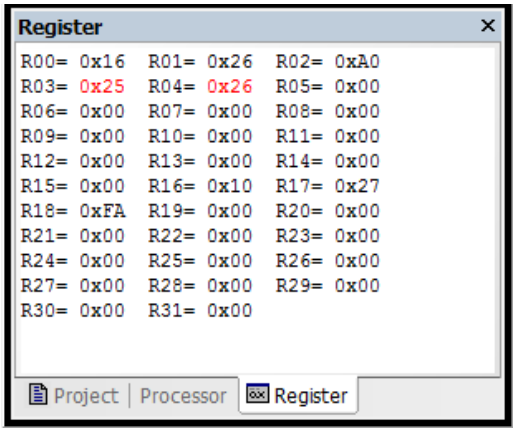
```
        mul m1M,m2 ; Multipliziere MSB
```

Die Multiplikation des MSB von m1, hex 10, mit m2, hex FA, ergibt hex 2616 in R1:R0.

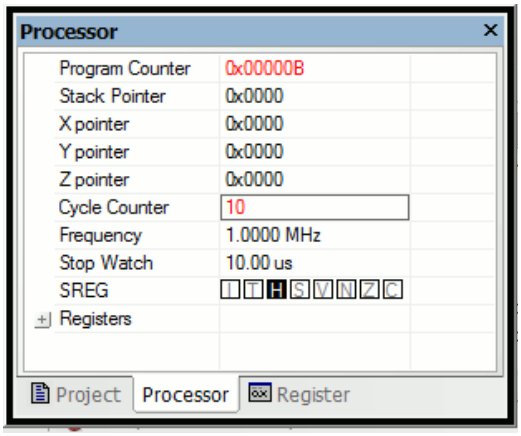


Nun werden zwei Schritte auf einmal gemacht: die Multiplikation mit 256 und die Addition des Ergebnisses zum bisherigen Ergebnis. Das wird erledigt durch Addition von R1:R0 zu Res3:Res2 anstelle von Res2:Res1. R1 wird zunächst schlicht nach Res3 kopiert. Dann wird R0 zu Res2 addiert. Falls dabei das Übertragsflag Carry nach der Addition gesetzt ist, muss noch das nächsthöhere Byte Res3 um Eins erhöht werden.

```
   mov Res3,R1 ; Kopiere MSB Ergebnis zum Byte 3
   add Res2,R0 ; Addiere LSB Ergebnis zum Byte 2
   brcc NoInc ; Wenn Carry nicht gesetzt, springe
   inc Res3 ; erhoehe Ergebnis-Byte 3
NoInc:
```



Das Ergebnis in R4:R3:R2 ist hex 2625A9, was dezimal 2.500.000 entspricht (wie jeder sofort weiß), und das ist korrekt.



Der Zykluszähler zeigt nach der Multiplikation auf 10, bei 1 MHz Takt sind gerade mal 10 Mikrosekunden vergangen. Sehr viel schneller als die Software-Multiplikation!

Hardware Multiplikation einer 16- mit einer 16-bit-Binärzahl

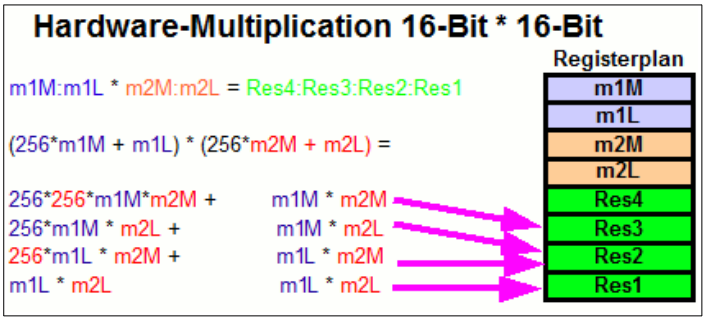
Nun, da wir das Prinzip verstanden haben, sollte es einfach sein die 16-mal-16-Multiplikation zu erledigen. Das Ergebnis benötigt jetzt vier Bytes (Res4:Res3:Res2:Res1, untergebracht in R5:R4:R3:R2). Die Formel lautet:

$$m1 * m2 =$$

$$(256*m1M + m1L) * (256*m2M + m2L) =$$

$$65536*m1M*m2M + 256*m1M*m2L + 256*m1L*m2M + m1L*m2L$$

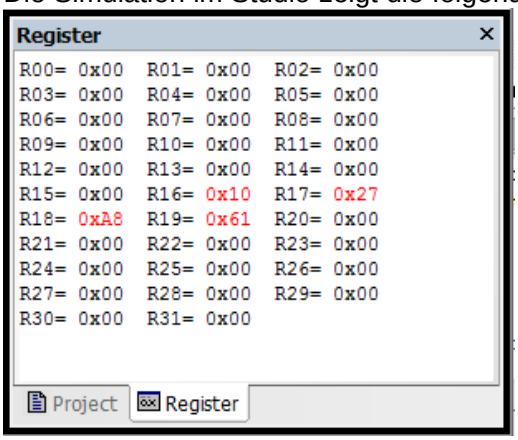
Offensichtlich sind jetzt vier Multiplikationen zu erledigen. Wir beginnen mit den beiden einfachsten, der ersten und der letzten: ihre Ergebnisse können einfach in die Ergebnisregister kopiert werden. Die beiden mittleren Multiplikationen in der Formel müssen zu den beiden mittleren Ergebnisregistern addiert werden. Mögliche Überläufe müssen in das Ergebnisregister Res4 übertragen werden, wofür hier ein einfacher Trick angewendet wird, der einfach zu verstehen sein dürfte. Die Software:



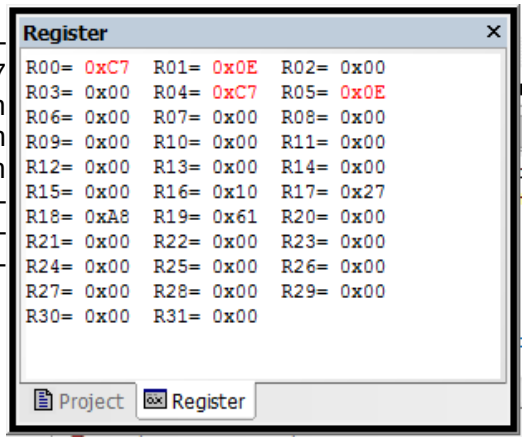

```
;
; Test Hardware Multiplikation 16 mal 16
;
; Definiere Register
;
.def Res1 = R2
.def Res2 = R3
.def Res3 = R4
.def Res4 = R5
.def m1L = R16
.def m1M = R17
.def m2L = R18
.def m2M = R19
.def tmp = R20
;
; Lade Startwerte
;
.equ m1 = 10000
.equ m2 = 25000
;
        ldi m1M,HIGH(m1)
        ldi m1L,LOW(m1)
        ldi m2M,HIGH(m2)
        ldi m2L,LOW(m2)
;
; Multipliziere
;
        clr R20 ; leer, fuer Uebertraege
        mul m1M,m2M ; Multipliziere MSBs
        mov Res3,R0 ; Kopie in obere Ergebnisregister
        mov Res4,R1
        mul m1L,m2L ; Multipliziere LSBs
        mov Res1,R0 ; Kopie in untere Ergebnisregister
        mov Res2,R1
        mul m1M,m2L ; Multipliziere 1M mit 2L
        add Res2,R0 ; Addiere zum Ergebnis
        adc Res3,R1
```

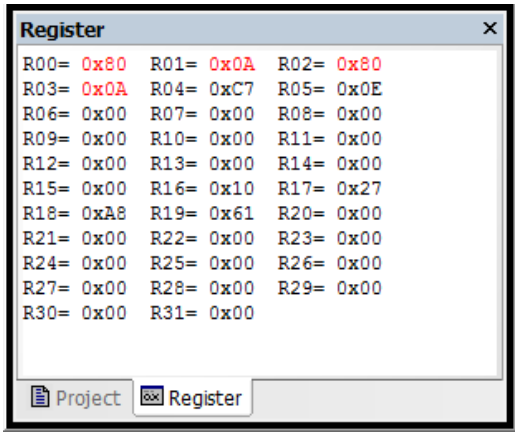
```
        adc Res4,tmp ; Addiere Uebertrag
        mul m1L,m2M ; Multipliziere 1L mit 2M
        add Res2,R0 ; Addiere zum Ergebnis
        adc Res3,R1
        adc Res4,tmp
;
; Multiplikation fertig
;
```

Die Simulation im Studio zeigt die folgenden Schritte. Das Laden der beiden Konstanten 10000 (hex 2710) und 25000 (hex 61A8) in die Register im oberen Registerraum ...

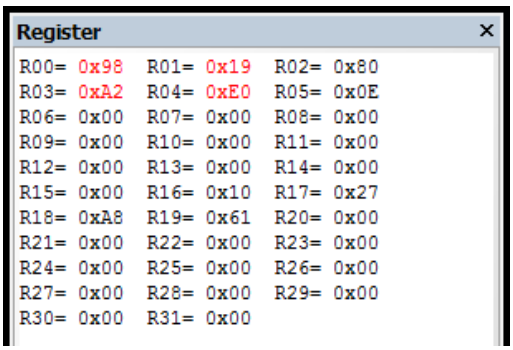


Multiplikation der beiden MSBs (hex 27 und 61) und kopieren des Ergebnisses in R1:R0 in die beiden oberen Ergebnisregister R5:R4 (Multiplikation mit 65536 eingeschlossen) ...



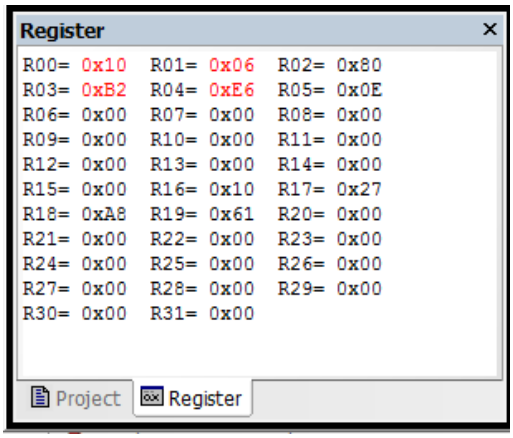


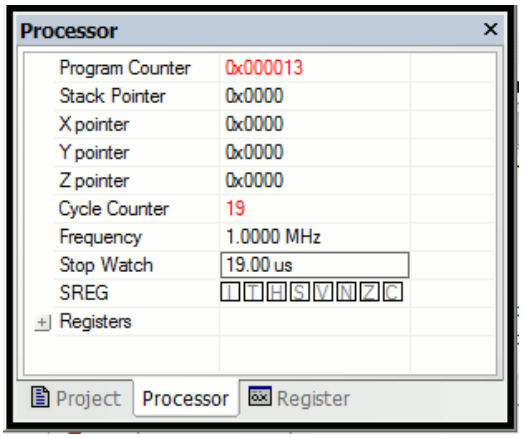
Multiplikation der beiden LSBs (hex 10 und A8) und kopieren des Ergebnisses in R1:R0 in die beiden niedrigeren Ergebnisregister R3:R2 ...



Multiplikation des MSB von m1 mit dem LSB von m2, und Addition des Ergebnisses in R1:R0 zu den beiden mittleren Ergebnisregistern, kein Übertrag ist erfolgt ...

Multiplikation des LSB von m1 mit dem MSB von m2, sowie Addition des Ergebnisses in R1:R0 mit den beiden mittleren Ergebnisbytes, kein Übertrag. Das Ergebnis ist hex 0EE6B280, was dezimal 250.000.000 ist und offenbar korrekt ...



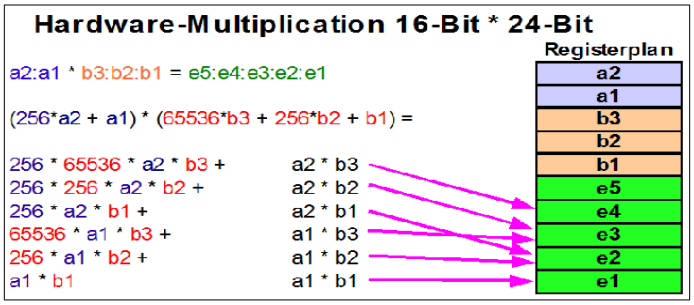


Die Multiplikation hat 19 Taktzyklen gebraucht, das ist sehr viel schneller als die Software-Multiplikation.

Ein weiterer Vorteil: die benötigte Zeit ist IMMER genau 19 Taktzyklen lang, nicht abhängig von den Zahlenwerten (wie es bei der Software-Multiplikation der Fall ist) und vom Auftreten von Überläufen (deshalb der Trick mit der Addition von Null mit Carry). Darauf kann man sich verlassen ...

Hardware Multiplikation einer 16- mit einer 24-Bit-Binärzahl

Die Multiplikation einer 16-Bit-Binärzahl "a" mit einer 24-Bit-Binärzahl "b" führt im Ergebnis zu einem maximal 40 Bit breiten Ergebnis. Das Multiplizier-Schema erfordert sechs 8-mit-8-Bit-Multiplikationen und das Addieren der Ergebnisse an der richtigen Position zum Gesamtergebnis. Der Assembler-Code dafür:




```
; Hardware Multiplikation 16 mit 24 Bit
.include "m8def.inc"
;
; Register Definitionen
.def a1 = R2 ; definiere 16-bit Register
.def a2 = R3
.def b1 = R4 ; definiere 24-bit Register
.def b2 = R5
.def b3 = R6
.def e1 = R7 ; definiere 40-bit Ergebnisregister
.def e2 = R8
.def e3 = R9
.def e4 = R10
.def e5 = R11
.def c0 = R12 ; Hilfsregister fuer Additionen
.def rl = R16 ; Laderegister
;
; Load constants
.equ a = 10000 ; Multiplikator a, hex 2710
.equ b = 1000000 ; Multiplikator b, hex 0F4240
        ldi rl,BYTE1(a) ; lade a
        mov a1,rl
        ldi rl,BYTE2(a)
        mov a2,rl
        ldi rl,BYTE1(b) ; lade b
        mov b1,rl
        ldi rl,BYTE2(b)
        mov b2,rl
        ldi rl,BYTE3(b)
        mov b3,rl
```

```
;
; Loesche Registers
        clr e1 ; Loesche Ergebnisregister
        clr e2
        clr e3
        clr e4
        clr e5
        clr c0 ; loesche Hilfsregister
;
; Multipliziere
        mul a2,b3 ; Term 1
        add e4,R0 ; addiere zum Ergebnis
        adc e5,R1
        mul a2,b2 ; Term 2
        add e3,R0
        adc e4,R1
        adc e5,c0 ; (addiere moeglichen Ueberlauf)
        mul a2,b1 ; Term 3
        add e2,R0
        adc e3,R1
        adc e4,c0
        adc e5,c0
        mul a1,b3 ; Term 4
        add e3,R0
        adc e4,R1
        adc e5,c0
        mul a1,b2 ; Term 5
        add e2,R0
        adc e3,R1
        adc e4,c0
        adc e5,c0
        mul a1,b1 ; Term 6
        add e1,R0
        adc e2,R1
        adc e3,c0
        adc e4,c0
        adc e5,c0
;
; fertig.
        nop
; Ergebnis sollte sein: hex 02540BE400
```

Die vollständige Abarbeitung braucht

- 10 Taktzyklen für das Laden der Konstanten,
- 6 Taktzyklen für das Löschen der Register, und
- 33 Taktzyklen für die Multiplikation.

13 Zahlenumwandlung in AVR-Assembler

Das Umwandeln von Zahlen kommt in Assembler recht häufig vor, weil der Prozessor am liebsten (und schnellsten) in binär rechnet, der dumme Mensch aber nur das Zehnersystem kann. Wer also aus einem Assemblerprogramm heraus mit Menschen an einer Tastatur kommunizieren möchte, kommt um Zahlenumwandlung nicht herum. Diese Seite befasst sich daher mit diesen Wandlungen zwischen den Zahlenwelten, und zwar etwas detaillierter und genauer.

Wer gleich in die Vollen gehen möchte, kann sich direkt in den üppig kommentierten Quelltext stürzen.

13.1 Allgemeine Bedingungen der Zahlenumwandlung

Die hier behandelten Zahlensysteme sind:

- Dezimal: Jedes Byte zu je acht Bit enthält eine Ziffer, die in ASCII formatiert ist. So repräsentiert der dezimale Wert 48, in binär $30, die Ziffer Null, 49 die Eins, usw. bis 57 die Neun. Die anderen Zahlen, mit denen ein Byte gefüllt werden kann, also 0 bis 47 und 58 bis 255, sind keine gültigen Dezimalziffern. (Warum gerade 48 die Null ist, hat mit amerikanischen Militärfernschreibern zu tun, aber das ist eine andere lange Geschichte.)
- BCD-Zahlen: BCD bedeutet Binary Coded Decimal. Es ist ähnlich wie dezimale ASCII-Zahlen, nur entspricht die BCD-dezimale Null jetzt tatsächlich dem Zahlenwert Null (und nicht 48). Dementsprechend gehen die BCDs von 0 bis 9. Alles weitere, was noch in ein Byte passen würde (10 .. 255) ist keine gültige Ziffer und gehört sich bei BCDs verboten.
- Binärzahlen: Hier gibt es nur die Ziffern 0 und 1. Von hinten gelesen besitzen sie jeweils die Wertigkeit der Potenzen von 2, also ist die Binärzahl 1011 soviel wie 1* (2 hoch 0) + 1*(2 hoch 1) + 0*(2 hoch 2) + 1*(2 hoch 3), so ähnlich wie dezimal 1234 gleich 4*(10 hoch 0) + 3*(10 hoch 1) + 2*(10 hoch 2) + 1*(10 hoch 3) ist (jede Zahl hoch 0 ist übrigens 1, nur nicht 0 hoch 0, da weiss man es nicht so genau!). Binärzahlen werden in Paketen zu je acht (als Byte bezeichnet) oder 16 (als Wort bezeichnet) Binärziffern gehandhabt, weil die einzelnen Bits kaum was wert sind.
- Hexadezimal: Hexadezimalzahlen sind eigentlich Viererpäckchen von Bits, denen man zur Vereinfachung die Ziffern 0 bis 9 und A bis F (oder a bis f) gibt und als solche meistens ASCII-verschlüsselt. A bis F deswegen, weil vier Bits Zahlen von 0 bis 15 sein können. So ist binär 1010 soviel wie 1*(2 hoch 3) + 1*(2 hoch 1), also dezimal 10, kriegt dafür den Buchstaben A. Der Buchstabe A liegt aber in der ASCII-Codetabelle beim dezimalen Wert 65, als Kleinbuchstabe sogar bei 97. Das alles ist beim Umwandeln wichtig und beim Codieren zu bedenken.

Soweit der Zahlenformatsalat. Die zum Umwandeln geschriebene Software soll einigermaßen brauchbar für verschiedene Zwecke sein. Es lohnt sich daher, vor dem Schreiben und Verwenden ein wenig Grütze zu investieren. Ich habe daher folgende Regeln ausgedacht und beim Schreiben eingehalten:

- Binärzahlen: Alle Binärzahlen sind auf 16 Bit ausgelegt (Wertebereich 0..65.535). Sie sind in den beiden Registern rBin1H (obere 8 Bit, MSB) und rBin1L (untere 8 Bit, LSB) untergebracht. Dieses binäre Wort wird mit rBin1H:L abgekürzt. Bevorzugter Ort beim AVR für die beiden ist z.B. R1 und R2, die Reihenfolge ist dabei egal. Für manche Umwandlungen wird ein zweites binäres Registerpaar gebraucht, das hier als rBin2H:L bezeichnet wird. Es kann z.B. in R3 und R4 gelegt werden. Es wird nach dem Gebrauch wieder in den Normalzustand versetzt, deshalb kann es unbesehen auch noch für andere Zwecke dienen.
- BCD- und ASCII-Zahlen: Diese Zahlen werden generell mit dem Zeiger Z angesteuert (Zeigerregister ZH:ZL oder R31:R30), weil solche Zahlen meistens irgendwo im SRAM-Speicher herumstehen. Sie sind so angeordnet, dass die höherwertigste Ziffer die niedrigste Adresse hat. Die Zahl 12345 würde also im SRAM so stehen: $0060: 1, $0061: 2, $0062: 3, usw. Der Zeiger Z kann aber auch in den Bereich der Register gestellt werden, also z.B. auf $0005. Dann läge die 1 in R5, die 2 in R6, die 3 in R7, usw. Die Software kann also die Dezimalzahlen sowohl aus dem SRAM als auch aus den Registern verarbeiten. Aber Obacht: es wird im Registerraum unbesehen alles überschrieben, was dort herumste-

hen könnte. Sorgfältige Planung der Register ist dann heftig angesagt!

- Paketeinteilung: Weil man nicht immer alle Umwandlungsroutinen braucht, ist das Gesamtpaket in vier Teilpakete eingeteilt. Die beiden letzten Pakete braucht man nur für Hexzahlen, die beiden ersten zur Umrechnung von ASCII- oder BCD- zu Binär bzw. von Binär in ASCII- und BCD. Jedes Paket ist mit den darin befindlichen Unterprogrammen separat lauffähig, es braucht nicht alles eingebunden werden. Beim Entfernen von Teilen der Pakete die Aufrufe untereinander beachten! Das Gesamtpaket umfasst 217 Worte Programm.
- Fehler: Tritt bei den Zahlenumwandlungen ein Fehler auf, dann wird bei allen fehlerträchtigen Routinen das T-Flag gesetzt. Das kann mit BRTS oder BRTC bequem abgefragt werden, um Fehler in der Zahl zu erkennen, abzufangen und zu behandeln. Wer das T-Flag im Statusregister SREG noch für andere Zwecke braucht, muss alle "set"-, "clt"-, "brts"- und "brtc"-Anweisungen auf ein anderes Bit in irgendeinem Register umkodieren. Bei Fehlern bleibt der Zeiger Z einheitlich auf der Ziffer stehen, bei der der Fehler auftrat.
- Weiteres: Weitere Bedingungen gibt es im allgemeinen Teil des Quelltextes. Dort gibt es auch einen Überblick zur Zahlenumwandlung, der alle Funktionen, Aufrufbedingungen und Fehlerarten enthält.

13.2 Von ASCII nach Binär

Die Routine AscToBin2 eignet sich besonders gut für die Ermittlung von Zahlen aus Puffern. Sie überliest beliebig viele führende Leerzeichen und Nullen und bricht die Zahlenumwandlung beim ersten Zeichen ab, das keine gültige Dezimalziffer repräsentiert. Die Zahlenlänge muss deshalb nicht vorher bekannt sein, der Zahlenwert darf nur den 16-Bit-Wertebereich der Binärzahl nicht überschreiten. Auch dieser Fehler wird erkannt und mit dem T-Flag signalisiert.

Soll die Länge der Zahl exakt fünf Zeichen ASCII umfassen, kann die Routine Asc5ToBin2 verwendet werden. Hier wird jedes ungültige Zeichen, bis auf führende Nullen und Leerzeichen, angemahnt.

Die Umrechnung erfolgt von links nach rechts, d.h. bei jeder weiteren Stelle der Zahl wird das bisherige Ergebnis mit 10 multipliziert und die dazu kommende Stelle hinzugezählt. Das geht etwas langsam, weil die Multiplikation mit 10 etwas rechenaufwendig ist. Es gibt sicher schnellere Arten der Wandlung.

13.3 Von BCD zu Binär

Die Umwandlung von BCD zu Binär funktioniert ähnlich. Auf eine eingehende Beschreibung der Eigenschaften dieses Quellcodes wird daher verzichtet.

13.4 Binärzahl mit 10 multiplizieren

Diese Routine Bin1Mul10 nimmt eine 16-Bit-Binärzahl mit 10 mal, indem sie diese kopiert und durch Additionen vervielfacht. Aufwändig daran ist, dass praktisch bei jedem Addieren ein Überlauf denkbar ist und abgefangen werden muss. Wer keine zu langen Zahlen zulässt oder wem es egal ist, ob Unsinn rauskommt, kann die Branch-Anweisungen alle rauswerfen und das Verfahren damit etwas beschleunigen.

13.5 Von binär nach ASCII

Die Wandlung von Binär nach ASCII erfolgt in der Routine Bin2ToAsc5. Das Ergebnis ist generell fünfstellig. Die eigentliche Umwandlung erfolgt durch Aufruf von Bin2ToBcd5. Wie das funktioniert, wird weiter unten beschrieben.

Wird statt Bin2ToAsc5 die Routine Bin2ToAsc aufgerufen, kriegt man den Zeiger Z auf die erste Nicht-Null in der Zahl gesetzt und die Anzahl der Stellen im Register rBin2L zurück. Das ist bequem, wenn man das Ergebnis über die serielle Schnittstelle senden will und unnötigen Leerzeichen-Verkehr vermeiden will.

13.6 Von binär nach BCD

Bin2ToBcd5 rechnet die Binärzahl in rBin1H:L in dezimal um. Auch dieses Verfahren ist etwas zeitaufwändig, aber sicher leicht zu verstehen. Die Umwandlung erfolgt durch fortgesetztes Abziehen immer kleiner werdender Binärzahlen, die die dezimalen Stellen repräsentieren, also 10.000, 1.000, 100 und 10. Nur bei den Einern wird von diesem Schema abgewichen, hi.

13.7 Von binär nach Hex

Die Routine Bin2ToHex4 produziert aus der Binärzahl eine vierstellige Hex-ASCII-Zahl, die etwas leichter zu lesen ist als das Original in binär. Oder mit welcher Wahrscheinlichkeit verschreiben Sie sich beim Abtippen der Zahl 1011011001011110? Da ist B65E doch etwas bequemer und leichter zu memorieren, fast so wie 46686 in dezimal.

Die Routine produziert die Hex-Ziffern A bis F in Großbuchstaben. Wer es lieber in a .. f mag: bitte schön, Quelltext ändern.

13.8 Von Hex nach Binär

Mit Hex4ToBin2 geht es den umgekehrten Weg. Da hier das Problem der falschen Ziffern auftreten kann, ist wieder jede Menge Fehlerbehandlungscode zusätzlich nötig.

13.9 Quellcode

```
; ******************************************************************************
; * Routinen zur Zahlumwandlung, V 0.1 Januar 2002 , (C)2002 info avr-asm-tutorial.net *
; ******************************************************************************
; Die folgenden Regeln gelten für alle Routinen zur Zahlumwandlung:
; - Fehler während der Umwandlung werden durch ein gesetztes T-Bit im Status-Register
;   signalisiert.
; - Der Z Zeiger zeigt entweder in das SRAM (Adresse >=$0060) oder auf einen
;   Registerbereich (Adressen $0000  bis $001D), die Register R0, R16 und R30/31 dürfen
;   nicht in dem benutzten Bereich liegen!
; - ASCII- und BCD-kodierte mehrstellige Zahlen sind absteigend geordnet, d.h. Die
;   höherwertigen Ziffern haben die niedrigerwertigen Adressen.
; - 16-bit-Binärzahlen sind generell in den Registern  rBin1H:rBin1L lokalisiert, bei
;   einigen Routinen wird zusätzlich rBin2H:rBin2L verwendet. Diese müssen im
;   Hauptprogramm definiert werden.
; - Bei Binärzahlen ist die Lage im Registerbereich nicht maßgebend, sie können auf- oder
;   absteigend geordnet sein oder getrennt im Registerraum liegen. Zu vermeiden ist eine
;   Zuordnung zu R0, rmp, ZH oder ZL.
; - Register rmp (Bereich: R16..R29) wird innerhalb der Rechenroutinen benutzt, sein
;   Inhalt ist nach Rückkehr nicht definiert.
; - Das Registerpaar Z wird innerhalb von Routinen verwendet. Bei der Rückkehr ist sein
;   Inhalt abhängig vom Fehlerstatus definiert.
; - Einige Routinen verwenden zeitweise Register R0. Sein Inhalt wird vor der Rückkehr
;   wieder hergestellt.
; - Wegen der Verwendung des Z-Registers ist in jedem Fall die Headerdatei des Prozessors
;   einzubinden oder ZL (R30) und ZH (R31) sind manuell zu definieren. Wird die
;   Headerdatei oder die manuelle Definition nicht  vorgenommen, gibt es beim Assemblieren
;   eine Fehlermeldung oder es geschehen rätselhafte Dinge.
; - Wegen der Verwendung von Unterroutinen muss der Stapelzeiger (SPH:SPL bzw. SPL bei
;   mehr als 256 Byte SRAM) initiiert sein.
;
;
;
; ************* Überblick über die Routinen **********************
; Routine    Aufruf       Bedingungen          Rückkehr,Fehler
```

```
; ------------------------------------------------------------
; AscToBin2  Z zeigt auf  Beendet beim ersten  16-bit-Bin
;            erstes       Zeichen, das nicht   rBin1H:L,
;            ASCII-       einer Dezimalziffer  Überlauf-
;            Zeichen      entspricht, über-    fehler
;                         liest Leerzeichen
;                         und führende Nullen
; Asc5ToBin2 Z zeigt auf  Benötigt exakt 5     16-bit-Bin
;            erstes       gültige Ziffern,     rBin1H:L,
;            ASCII-       überliest Leerzei-   Überlauf
;            Zeichen      chen und Nullen      oder ungül-
;                                              tige Ziffern
; Bcd5ToBin2 Z zeigt auf  Benötigt exakt 5     16-bit-Bin
;            5-stellige   gültige Ziffern      rBin1H:L
;            BCD-Zahl                          Überlauf
;                                              oder ungül-
;                                              tige Ziffern
; Bin2ToBcd5 16-bit-Bin   Z zeigt auf erste    5-digit-BCD
;            in rBin1H:L  BCD-Ziffer (auch     ab Z, keine
;                         nach Rückkehr)       Fehler
; Bin2ToHex4 16-bit-Bin   Z zeigt auf erste    4-stellige
;            in rBin1H:L  Hex-ASCII-Stelle,    Hex-Zahl ab
;                         Ausgabe A...F        Z, keine
;                                              Fehler
; Hex4ToBin2 4-digit-Hex  Benötigt exakt vier  16-bit-Bin
;            Z zeigt auf  Stellen Hex-ASCII,   rBin1H:L,
;            erste Stelle akzeptiert A...F und ungültige
;                         a...f                Hex-Ziffer
; ******************* Umwandlungscode **********************
;
; Paket I: Von ASCII bzw. BCD nach Binär
; AscToBin2
; =========
; wandelt eine ASCII-kodierte Zahl in eine 2-Byte-/16-Bit- Binärzahl um.
; Aufruf: Z zeigt auf erste Stelle der umzuwandelnden Zahl, die Umwandlung wird bei der
;         ersten nicht dezimalen Ziffer beendet.
; Stellen: Zulässig sind alle Zahlen, die innerhalb des Wertebereiches von 16 Bit binär
;         liegen (0..65535).
; Rückkehr: Gesetztes T-Flag im Statusregister zeigt Fehler an.
;   T=0: Zahl in rBin1H:L ist gültig, Z zeigt auf erstes Zeichen, das keiner Dezimalziffer
;        entsprach
;   T=1: Überlauffehler (Zahl zu groß), rBin1H:L undefiniert, Z zeigt auf Zeichen, bei
;        dessen Verarbeitung der Fehler auftrat.
; Benötigte Register: rBin1H:L (Ergebnis), rBin2H:L (wieder  hergestellt), rmp
; Benötigte Unterroutinen: Bin1Mul10
AscToBin2:
        clr rBin1H ; Ergebnis auf Null setzen
        clr rBin1L
        clt ; Fehlerflagge zurücksetzen
AscToBin2a:
        ld rmp,Z+ ; lese Zeichen
        cpi rmp,' ' ; ignoriere führende Leerzeichen ...
        breq AscToBin2a
        cpi rmp,'0' ; ... und Nullen
        breq AscToBin2a
AscToBin2b:
        subi rmp,'0' ; Subtrahiere ASCII-Null
        brcs AscToBin2d ; Ende der Zahl erkannt
        cpi rmp,10 ; prüfe Ziffer
        brcc AscToBin2d ; Ende der Umwandlung
        rcall Bin1Mul10 ; Binärzahl mit 10 malnehmen
```

```
        brts AscToBin2c ; Überlauf, gesetztes T-Flag
        add rBin1L,rmp ; Addiere Ziffer zur Binärzahl
        ld rmp,Z+ ; Lese schon mal nächstes Zeichen
        brcc AscToBin2b ; Kein Überlauf ins MSB
        inc rBin1H ; Überlauf ins nächste Byte
        brne AscToBin2b ; Kein Überlauf ins nächste Byte
        set ; Setze Überlauffehler-Flagge
AscToBin2c:
        sbiw ZL,1 ; Überlauf trat bei letztem Zeichen auf
AscToBin2d:
        ret ; fertig, Rückkehr
;
; Asc5ToBin2
; ==========
; wandelt eine fünfstellige ASCII-kodierte Zahl in 2-Byte-Binärzahl um.
; Aufruf: Z zeigt auf erste Stelle der ASCII-kodierten Zahl, führende Leerzeichen und
;         Nullen sind erlaubt.
; Stellen: Die Zahl muss exakt 5 gültige Stellen haben.
; Rückkehr: T-Flag zeigt Fehlerbedingung an:
;   T=0: Binärzahl in rBin1H:L ist gültig, Z zeigt auf erste Stelle der ASCII-kodierten
;        Zahl.
;   T=1: Fehler bei der Umwandlung. Entweder war die Zahl zu groß (0..65535, Z zeigt auf
;        die Ziffer, bei der der Überlauf auftrat) oder sie enthilt ein ungültiges Zeichen
;        (Z zeigt auf das ungültige Zeichen).
; Benötigte Register: rBin1H:L (Ergebnis), R0 (wiederhergestellt), rBin2H:L (wieder
;   hergestellt), rmp
; Aufgerufene Unterroutinen: Bin1Mul10
;
Asc5ToBin2:
        push R0 ; R0 wird als Zähler verwendet, retten
        ldi rmp,6 ; Fünf Ziffern, einer zu viel
        mov R0,rmp ; in Zählerregister R0
        clr rBin1H ; Ergebnis auf Null setzen
        clr rBin1L
        clt ; Fehlerflagge T-Bit zurücksetzen
Asc5ToBin2a:
        dec R0 ; Alle Zeichen leer oder Null?
        breq Asc5ToBin2d ; Ja, beenden
        ld rmp,Z+ ; Lese nächstes Zeichen
        cpi rmp,' ' ; überlese Leerzeichen
        breq Asc5ToBin2a ; geh zum nächsten Zeichen
        cpi rmp,'0' ; überlese führende Nullen
        breq Asc5ToBin2a ; geh zum nächsten Zeichen
Asc5ToBin2b:
        subi rmp,'0' ; Behandle Ziffer, ziehe ASCII-0 ab
        brcs Asc5ToBin2e ; Ziffer ist ungültig, raus
        cpi rmp,10 ; Ziffer größer als neun?
        brcc Asc5ToBin2e ; Ziffer ist ungültig, raus
        rcall Bin1Mul10 ; Multipliziere Binärzahl mit 10
        brts Asc5ToBin2e ; Überlauf, raus
        add rBin1L,rmp ; addiere die Ziffer zur Binärzahl
        ld rmp,z+ ; lese schon mal das nächste Zeichen
        brcc Asc5ToBin2c ; Kein Überlauf in das nächste Byte
        inc rBin1H ; Überlauf in das nächste Byte
        breq Asc5ToBin2e ; Überlauf auch ins übernächste Byte
Asc5ToBin2c:
        dec R0 ; Verringere Zähler für Anzahl Zeichen
        brne Asc5ToBin2b ; Wandle weitere Zeichen um
Asc5ToBin2d: ; Ende der ASCII-kodierten Zahl erreicht
        sbiw ZL,5 ; Stelle die Startposition in Z wieder her
        pop R0 ; Stelle Register R0 wieder her
```

```
        ret ; Kehre zurück
Asc5ToBin2e: ; Letztes Zeichen war ungültig
        sbiw ZL,1 ; Zeige mit Z auf ungültiges Zeichen
        pop R0 ; Stelle Register R0 wieder her
        set ; Setze T-Flag für Fehler
        ret ; und kehre zurück
; Bcd5ToBin2
; ==========
; wandelt eine 5-bit-BCD-Zahl in eine 16-Bit-Binärzahl um
; Aufruf: Z zeigt auf erste Stelle der BCD-kodierten Zahl
; Stellen: Die Zahl muss exakt 5 gültige Stellen haben.
; Rückkehr: T-Flag zeigt Fehlerbedingung an:
;   T=0: Binärzahl in rBin1H:L ist gültig, Z zeigt auf erste Stelle der BCD-kodierten
         Zahl.
;   T=1: Fehler bei der Umwandlung. Entweder war die Zahl zu groß (0..65535, Z zeigt auf
;        die Ziffer, bei der der Überlauf auftrat) oder sie enthielt ein ungültiges
;        Zeichen (Z zeigt auf das ungültige Zeichen).
; Benötigte Register: rBin1H:L (Ergebnis), R0 (wiederhergestellt), rBin2H:L (wieder
;        hergestellt), rmp
; Aufgerufene Unterroutinen: Bin1Mul10
;
Bcd5ToBin2:
        push R0 ; Rette Register R0
        clr rBin1H ; Setze Ergebnis Null
        clr rBin1L
        ldi rmp,5 ; Setze Zähler auf 5 Ziffern
        mov R0,rmp ; R0 ist Zähler
        clt ; Setze Fehlerflagge zurück
Bcd5ToBin2a:
        ld rmp,Z+ ; Lese BCD-Ziffer
        cpi rmp,10 ; prüfe ob Ziffer korrekt
        brcc Bcd5ToBin2c ; ungültige BCD-Ziffer
        rcall Bin1Mul10 ; Multipliziere Ergebnis mit 10
        brts Bcd5ToBin2c ; Überlauf aufgetreten
        add rBin1L,rmp ; Addiere Ziffer
        brcc Bcd5ToBin2b ; Kein Überlauf ins nächste Byte
        inc rBin1H ; Überlauf ins nächste Byte
        breq Bcd5ToBin2c ; Überlauf ins übernächste Byte
Bcd5ToBin2b:
        dec R0 ; weitere Ziffer?
        brne Bcd5ToBin2a ; Ja
        pop R0 ; Stelle Register wieder her
        sbiw ZL,5 ; Setze Zeiger auf erste Stelle der BCD-Zahl
        ret ; Kehre zurück
Bcd5ToBin2c:
        sbiw ZL,1 ; Eine Ziffer zurück
        pop R0 ; Stelle Register wieder her
        set ; Setze T-flag, Fehler
        ret ; Kehre zurück
;
; Bin1Mul10
; =========
; Multipliziert die 16-Bit-Binärzahl mit 10
; Unterroutine benutzt von AscToBin2, Asc5ToBin2, Bcd5ToBin2
; Aufruf: 16-Bit-Binärzahl in rBin1H:L
; Rückkehr: T-Flag zeigt gültiges Ergebnis an.
;   T=0: rBin1H:L enthält gültiges Ergebnis.
;   T=1: Überlauf bei der Multiplikation, rBin1H:L undefiniert
; Benutzte Register: rBin1H:L (Ergebnis), rBin2H:L (wird wieder  hergestellt)
Bin1Mul10:
        push rBin2H ; Rette die Register rBin2H:L
```

```
        push rBin2L
        mov rBin2H,rBin1H ; Kopiere die Zahl dort hin
        mov rBin2L,rBin1L
        add rBin1L,rBin1L ; Multipliziere Zahl mit 2
        adc rBin1H,rBin1H
        brcs Bin1Mul10b ; Überlauf, raus hier!
Bin1Mul10a:
        add rBin1L,rbin1L ; Noch mal mit 2 malnehmen (=4*Zahl)
        adc rBin1H,rBin1H
        brcs Bin1Mul10b ; Überlauf, raus hier!
        add rBin1L,rBin2L ; Addiere die Kopie (=5*Zahl)
        adc rBin1H,rBin2H
        brcs Bin1Mul10b ;Überlauf, raus hier!
        add rBin1L,rBin1L ; Noch mal mit 2 malnehmen (=10*Zahl)
        adc rBin1H,rBin1H
        brcc Bin1Mul10c ; Kein Überlauf, überspringe
Bin1Mul10b:
        set ; Überlauf, setze T-Flag
Bin1Mul10c:
        pop rBin2L ; Stelle die geretteten Register wieder her
        pop rBin2H
        ret ; Kehre zurück
; ********************************************
; Paket II: Von Binär nach ASCII bzw. BCD
; Bin2ToAsc5
; ==========
; wandelt eine 16-Bit-Binärzahl in eine fünfstellige ASCII-kodierte Dezimalzahl um
; Aufruf: 16-Bit-Binärzahl in rBin1H:L, Z zeigt auf Anfang der Zahl
; Rückkehr: Z zeigt auf Anfang der Zahl, führende Nullen sind  mit Leerzeichen
;    überschrieben
; Benutzte Register: rBin1H:L (bleibt erhalten), rBin2H:L (wird überschrieben), rmp
; Aufgerufene Unterroutinen: Bin2ToBcd5
;
Bin2ToAsc5:
        rcall Bin2ToBcd5 ; wandle Binärzahl in BCD um
        ldi rmp,4 ; Zähler auf 4
        mov rBin2L,rmp
Bin2ToAsc5a:
        ld rmp,z ; Lese eine BCD-Ziffer
        tst rmp ; prüfe ob Null
        brne Bin2ToAsc5b ; Nein, erste Ziffer ungleich 0 gefunden
        ldi rmp,' ' ; mit Leerzeichen überschreiben
        st z+,rmp ; und ablegen
        dec rBin2L ; Zähler um eins senken
        brne Bin2ToAsc5a ; weitere führende Leerzeichen
        ld rmp,z ; Lese das letzte Zeichen
Bin2ToAsc5b:
        inc rBin2L ; Ein Zeichen mehr
Bin2ToAsc5c:
        subi rmp,-'0' ; Addiere ASCII-0
        st z+,rmp ; und speichere ab, erhöhe Zeiger
        ld rmp,z ; nächstes Zeichen lesen
        dec rBin2L ; noch Zeichen behandeln?
        brne Bin2ToAsc5c ; ja, weitermachen
        sbiw ZL,5 ; Zeiger an Anfang
        ret ; fertig
; Bin2ToAsc
; =========
; wandelt eine 16-Bit-Binärzahl in eine fünfstellige ASCII-kodierte Dezimalzahl um, Zeiger
;   zeigt auf die erste signifikante Ziffer der Zahl, und gibt Anzahl der Ziffern zurück
; Aufruf: 16-Bit-Binärzahl in rBin1H:L, Z zeigt auf Anfang der Zahl (5 Stellen
```

```
;         erforderlich, auch bei kleineren Zahlen!)
; Rückkehr: Z zeigt auf erste signifikante Ziffer der ASCII-kodierten Zahl, rBin2L enthält
;           Länge der Zahl (1..5)
; Benutzte Register: rBin1H:L (bleibt erhalten), rBin2H:L (wird überschrieben), rmp
; Aufgerufene Unterroutinen: Bin2ToBcd5, Bin2ToAsc5
;
Bin2ToAsc:
        rcall Bin2ToAsc5 ; Wandle Binärzahl in ASCII
        ldi rmp,6 ; Zähler auf 6
        mov rBin2L,rmp
Bin2ToAsca:
        dec rBin2L ; verringere Zähler
        ld rmp,z+ ; Lese Zeichen und erhöhe Zeiger
        cpi rmp,' ' ; war Leerzeichen?
        breq Bin2ToAsca ; Nein, war nicht
        sbiw ZL,1 ; ein Zeichen rückwärts
        ret ; fertig
; Bin2ToBcd5
; ==========
; wandelt 16-Bit-Binärzahl in 5-stellige BCD-Zahl um
; Aufruf: 16-Bit-Binärzahl in rBin1H:L, Z zeigt auf die erste Stelle der BCD-kodierten
;         Resultats
; Stellen: Die BCD-Zahl hat exakt 5 gültige Stellen.
; Rückkehr: Z zeigt auf die höchste BCD-Stelle
; Benötigte Register: rBin1H:L (wird erhalten), rBin2H:L (wird nicht erhalten), rmp
; Aufgerufene Unterroutinen: Bin2ToDigit
;
Bin2ToBcd5:
        push rBin1H ; Rette Inhalt der Register rBin1H:L
        push rBin1L
        ldi rmp,HIGH(10000) ; Lade 10.000 in rBin2H:L
        mov rBin2H,rmp
        ldi rmp,LOW(10000)
        mov rBin2L,rmp
        rcall Bin2ToDigit ; Ermittle 5.Stelle durch Abziehen
        ldi rmp,HIGH(1000) ; Lade 1.000 in rBin2H:L
        mov rBin2H,rmp
        ldi rmp,LOW(1000)
        mov rBin2L,rmp
        rcall Bin2ToDigit ; Ermittle 4.Stelle durch Abziehen
        ldi rmp,HIGH(100) ; Lade 100 in rBin2H:L
        mov rBin2H,rmp
        ldi rmp,LOW(100)
        mov rBin2L,rmp
        rcall Bin2ToDigit ; Ermittle 3.Stelle durch Abziehen
        ldi rmp,HIGH(10) ; Lade 10 in rBin2H:L
        mov rBin2H,rmp
        ldi rmp,LOW(10)
        mov rBin2L,rmp
        rcall Bin2ToDigit ; Ermittle 2.Stelle durch Abziehen
        st z,rBin1L ; Rest sind Einer
        sbiw ZL,4 ; Setze Zeiger Z auf 5.Stelle (erste Ziffer)
        pop rBin1L ; Stelle den Originalwert wieder her
        pop rBin1H
        ret ; und kehre zurück
; Bin2ToDigit
; ===========
; ermittelt eine dezimale Ziffer durch fortgesetztes Abziehen einer binär kodierten
; Dezimalstelle
; Unterroutine benutzt von: Bin2ToBcd5, Bin2ToAsc5, Bin2ToAsc
; Aufruf: Binärzahl in rBin1H:L, binär kodierte Dezimalzahl  in rBin2H:L, Z zeigt auf
```

```
;         bearbeitete BCD-Ziffer
; Rückkehr: Ergebis in Z (bei Aufruf), Z um eine Stelle erhöht, keine Fehlerbehandlung
; Benutzte Register: rBin1H:L (enthält Rest der Binärzahl),  rBin2H (bleibt erhalten), rmp
; Aufgerufene Unterroutinen: -
;
Bin2ToDigit:
        clr rmp ; Zähler auf Null
Bin2ToDigita:
        cp rBin1H,rBin2H ; Vergleiche MSBs miteinander
        brcs Bin2ToDigitc ; MSB Binärzahl kleiner, fertig
        brne Bin2ToDigitb ; MSB Binärzahl größer, subtrahiere
        cp rBin1L,rBin2L ; MSB gleich, vergleiche LSBs
        brcs Bin2ToDigitc ; LSB Binärzahl kleiner, fertig
Bin2ToDigitb:
        sub rBin1L,rBin2L ; Subtrahiere LSB Dezimalzahl
        sbc rBin1H,rBin2H ; Subtrahiere Carry und MSB
        inc rmp ; Erhöhe den Zähler
        rjmp Bin2ToDigita ; Weiter vergleichen/subtrahieren
Bin2ToDigitc:
        st z+,rmp ; Speichere das Ergebnis und erhöhe Zeiger
        ret ; zurück
; *********************************************
; Paket III: Von Binär nach Hex-ASCII
;
; Bin2ToHex4
; ==========
; wandelt eine 16-Bit-Binärzahl in Hex-ASCII
; Aufruf: Binärzahl in rBin1H:L, Z zeigt auf erste Position des vierstelligen ASCII-Hex
; Rückkehr: Z zeigt auf erste Position des vierstelligen ASCII-Hex, ASCII-Ziffern A..F in
;           Großbuchstaben
; Benutzte Register: rBin1H:L (bleibt erhalten), rmp
; Aufgerufene Unterroutinen: Bin1ToHex2, Bin1ToHex1
;
Bin2ToHex4:
        mov rmp,rBin1H ; MSB in rmp kopieren
        rcall Bin1ToHex2 ; in Hex-ASCII umwandeln
        mov rmp,rBin1L ; LSB in rmp kopieren
        rcall Bin1ToHex2 ; in Hex-ASCII umwandeln
        sbiw ZL,4 ; Zeiger auf Anfang Hex-ASCII
        ret ; fertig
; Bin1ToHex2 wandelt die 8-Bit-Binärzahl in rmp in Hex-ASCII
; gehört zu: Bin2ToHex4
Bin1ToHex2:
        push rmp ; Rette Byte auf dem Stapel
        swap rmp ; Vertausche die oberen und unteren 4 Bit
        rcall Bin1ToHex1 ; wandle untere 4 Bits in Hex-ASCII
        pop rmp ; Stelle das Byte wieder her
Bin1ToHex1:
        andi rmp,$0F ; Maskiere die oberen vier Bits
        subi rmp,-'0' ; Addiere ASCII-0
        cpi rmp,'9'+1 ; Ziffern A..F?
        brcs Bin1ToHex1a ; Nein
        subi rmp,-7 ; Addiere 7 für A..F
Bin1ToHex1a:
        st z+,rmp ; abspeichern und Zeiger erhöhen
        ret ; fertig
; *********************************************
; Paket IV: Von Hex-ASCII nach Binär
; Hex4ToBin2
; ==========
; wandelt eine vierstellige Hex-ASCII-Zahl in eine 16-Bit- Binärzahl um
```

```
; Aufruf: Z zeigt auf die erste Stelle der Hex-ASCII-Zahl
; Rückkehr: T-Flag zeigt Fehler an:
;   T=0: rBin1H:L enthält die 16-Bit-Binärzahl, Z zeigt  auf die erste Hex-ASCII-Ziffer
;        wie beim Aufruf
;   T=1: ungültige Hex-ASCII-Ziffer, Z zeigt auf ungültige Ziffer
; Benutzte Register: rBin1H:L (enthält Ergebnis), R0 (wieder hergestellt), rmp
; Aufgerufene Unterroutinen: Hex2ToBin1, Hex1ToBin1
;
Hex4ToBin2:
        clt ; Lösche Fehlerflag
        rcall Hex2ToBin1 ; Wandle zwei Hex-ASCII-Ziffern
        brts Hex4ToBin2a ; Fehler, beende hier
        mov rBin1H,rmp ; kopiere nach MSB Ergebnis
        rcall Hex2ToBin1 ; Wandle zwei Hex-ASCII-Ziffern
        brts Hex4ToBin2a ; Fehler, beende hier
        mov rBin1L,rmp ; kopiere nach LSB Ergebnis
        sbiw ZL,4 ; Ergebis ok, Zeiger auf Anfang
Hex4ToBin2a:
        ret ; zurück
; Hex2ToBin1 wandelt 2-stellig-Hex-ASCII nach 8-Bit-Binär
Hex2ToBin1:
        push R0 ; Rette Register R0
        rcall Hex1ToBin1 ; Wandle nächstes Zeichen in Byte
        brts Hex2ToBin1a ; Fehler, stop hier
        swap rmp; untere vier Bits in obere vier Bits
        mov R0,rmp ; zwischenspeichern
        rcall Hex1ToBin1 ; Nächstes Zeichen umwandeln
        brts Hex2ToBin1a ; Fehler, raus hier
        or rmp,R0 ; untere und obere vier Bits zusammen
Hex2ToBin1a:
        pop R0 ; Stelle R0 wieder her
        ret ; zurück
; Hex1ToBin1 liest ein Zeichen und wandelt es in Binär um
Hex1ToBin1:
        ld rmp,z+ ; Lese Zeichen
        subi rmp,'0' ; Ziehe ASCII-0 ab
        brcs Hex1ToBin1b ; Fehler, kleiner als 0
        cpi rmp,10 ; A..F ; Ziffer größer als 9?
        brcs Hex1ToBin1c ; nein, fertig
        cpi rmp,$30 ; Kleinbuchstaben?
        brcs Hex1ToBin1a ; Nein
        subi rmp,$20 ; Klein- in Grossbuchstaben
Hex1ToBin1a:
        subi rmp,7 ; Ziehe 7 ab, A..F ergibt $0A..$0F
        cpi rmp,10 ; Ziffer kleiner $0A?
        brcs Hex1ToBin1b ; Ja, Fehler
        cpi rmp,16 ; Ziffer größer als $0F
        brcs Hex1ToBin1c ; Nein, Ziffer in Ordnung
Hex1ToBin1b: ; Error
        sbiw ZL,1 ; Ein Zeichen zurück
        set ; Setze Fehlerflagge
Hex1ToBin1c:
        ret ; Zurück
```

14 Fließkommazahlenberechnungen in Assemblersprache

14.1 Fließkommazahlen, bitte schön, wenn es unbedingt sein muss

Wer sich das Leben schwerer machen möchte als unbedingt sein muss: es gibt neben Ganzzahlen, vor-

zeichenbehafteten Ganzzahlen und Festkomma-Zahlen auch Fließkommazahlen. Was in Hochsprachen so einfach daherkommt wie 1,234567 ist in Assembler aber ziemlich kompliziert. Und das kommt so.

14.2 Das Format von Fließkommazahlen

Binäre Fließkommazahlen bestehen aus zwei Bestandteilen:

1. einer Mantisse, und
2. einem Exponenten.

In der dezimalen Welt gibt die Mantisse den Zahlenbestandteil an, bei 1,234567 würde sie lauten: 1,234567E+00. Der Exponent gibt an, wie oft die Mantisse mit 10 malgenommen werden muss, in unserem Beispiel wäre das keinmal, also eine Null. Mantissen beginnen also immer mit einer Ziffer ungleich Null vor dem Komma. Ist die Zahl größer als Eins, wird sie solange durch 10 geteilt, bis sie mit irgendeiner einzelnen Ziffer anfängt. Dabei wird der Exponent immer um eins nach oben gezählt.

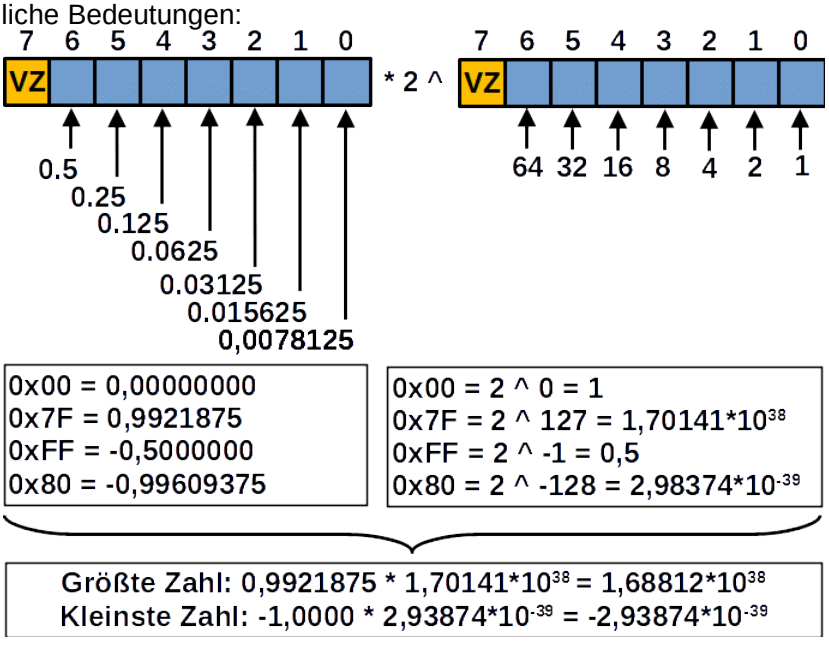
Umgekehrt: ist die Zahl kleiner als 1 und ist die erste Ziffer vor dem Komma Null, wird sie solange mit 10 multipliziert, bis die erste Ziffer vor dem Komma nicht mehr Null ist. Jetzt wird der Exponent abwärts gezählt, bei 0,01234567 zweimal, er wird also minus Zwei. Der Exponent muss bei solchen Zahlenwerten daher ein Vorzeichen haben, damit er negativ werden kann.

Den Vorgang, dass der Exponent so lange verschoben wird, bis die erste Ziffer vor dem Komma nicht größ oder gleich 10 und nicht Null wird, bezeichnet man als Normalisieren. Das werden wir im Folgenden und beim Umwandeln von binären Fließkommazahlen in dezimales Zahlenformat noch oft sehen.

Zahlen selbst können aber selbst auch negativ werden, wie -1,234567. Da sich durch Malnehmen und Teilen das Vorzeichen der Zahl nicht ändert, braucht auch die Mantisse ein Voreichen, wenn negative Zahlen verarbeitet werden sollen, z. B. -183°C (der Siedepunkt des Sauerstoffs). Normalisiert würde diese Zahl **1,83E+02** lauten.

Übertragen in die binäre Welt brauchen Fließkommazahlen also wenistens zwei Bytes: eines für die Mantisse und eines für den Exponenten. Beide haben ein Vorzeichen. Allerdings hat jedes Bit in der Mantisse und im Exponenten gänzlich unterschiedliche Bedeutungen:

1. In der Mantisse ist jedes Bit, vom Komma oder vom höchsten her begonnen, eine um Zwei abnehmende Bedeutung. So ist das erste Bit dezimal 0,5 wert, das zweite Bit 0,25, etc. etc.
2. Beim Exponenten ist jedes Bit, von hinten her betrachtet, immer das Doppelte des dahinterstehenden wert. Der Exponent kann daher von 0 bis 127 reichen (binär 0x00 bis 0x7F), wenn er positiv ist, und von -1 bis -128 wenn er negativ ist (0xFF für -1 und 0x80 für -128).

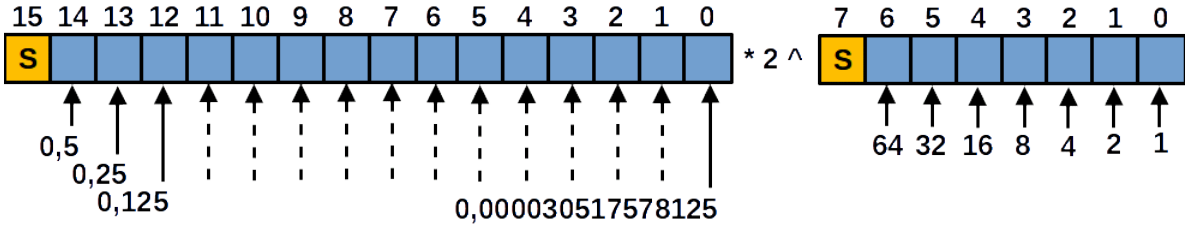


Da der Exponent mit der Basis Zwei hochgenommen wird (* 2 ^), ist er dezimal allerdings sehr viel mehr wert als die Mantisse. So ist 2^127 schon 1,7-mal-10-hoch-38 oder $*10^{38}$ oder auch E38. Umgekehrt können negative Exponenten auch sehr kleine Zahlenwerte sein, so ist 2^-128 dezimal 2,9E-39. Mit nur acht Bit Expo-

nent lässt sich daher schon der mächtige Zahlenbereich zwischen 2,9E-39 bis 1,7E+38 abbilden. Außer für astronomische Berechnungen kommen diese acht Bits schon gut hin.

Richtig mau sieht es aber bei der Mantisse aus: mit ihr lassen sich Zahlen nur mit zwei Dezimalziffern Genauigkeit abbilden. Schon mit 0,995 wären unsere acht Bits überfordert, da schon bei 0,992 Schluss ist.

Macht nix, nehmen wir halt zwei Bytes für die Mantisse. Das letzte Bit der Mantisse ist jetzt 0,0000305 wert. Damit lassen sich jetzt knapp fünf Dezimalstellen korrekt abbilden. Das ist für Genauigkeitsfanatiker noch nicht sehr befriedigend, die brauchen ein weiteres Byte (von einfacher Genauigkeit nach doppelter),



0x0000 = 0,000000000000000	0x00 = 2 ^ 0 = 1
0x0001 = 0,000030517578125	0x01 = 2 ^ 1 = 2
0x7FFF = 0,999969482421875	0x7F = 2 ^ 127 = $1{,}70141 \cdot 10^{38}$
0x8000 = - 0x7FFF = -0,999969482421875	0x80 = 2 ^ -128 = $2{,}93874 \cdot 10^{-39}$
0xFFFF = - 0x0001 = -0,500000000000000	0xFF = 2 ^ -1 = 0,5

und sind auch damit wahrscheinlich noch nicht zufrieden.

Deswegen wird beim Fließkomma mit jedem Mantissenbit gegeizt. Das führt dann zu allerlei verwirrenden Zusatzregeln. Da nach dem Normalisieren (das ist das Links- oder Rechtsschieben der Mantisse mit Dekrementieren bzw. Inkrementieren des Exponenten) das erste Mantissen-Bit immer eine Eins ist, kann man diese Eins auch weglassen und stattdessen in den 16 Bits ein weiteres Mantissenbit unterbringen. Mit diesem Bit-Geiz-Gefriemel verkompliziert man dann die Normen um eine weitere Variante.

Wer nicht glaubt, dass Fließkommazahlen Ressourcenfresser sind, der ziehe sich mal das nächste Kapitel rein. Dass man für diese einfache Übung ganze 400 Zeilen Assembler-Quellcode braucht wird aber noch davon getoppt, dass diese Wandlung mehr als 4 ms Zeit braucht, in der der AVR mit nix anderem beschäftigt ist als mit Links- und Rechts-Schieben, mit Addieren und mit Bytes-Verschieben.

Ähnlich aufwändig ist die Umwandlung von Dezimalzahlen in binäre Fließkommazahlen hier. Die ist mit 527 Assembler-Quellcode-Zeilen sogar noch geringfügig aufwändiger.

Einen Vorteil haben diese Fließkomma-Orgien immerhin: sie vereinfachen das Malnehmen und Teilen zweier Zahlen. Wenn wir zwei Zahlen mit ihren Mantissen M1 und M2 und ihren Exponenten E1 und E2 (M1*2^E1, M2*2^E2) miteinander malnehmen, brauchen wir nur die Mantissen malnehmen: M = M1 * M2, die beiden Exponenten werden dann einfach addiert E = E1 + E2. Beim Dividieren werden die Mantissen geteilt und die Exponenten einfach subtrahiert.

Dafür wird das Addieren und Subtrahieren komplizierter: zuerst müssen die beiden Exponenten in Übereinstimmung gebracht werden (durch Rechtsschieben der kleineren Zahl), erst dann können beide Mantissen addiert bzw. subtrahiert werden.

14.3 Fazit:

Wer schlau ist und die Exponentenwichserei bis 10^{38} gar nicht braucht, bleibt bei Ganz- und Festkommazahlen (Pseudo-Fließkomma), gerne auch mit Vorzeichen, und vermeidet Fließkomma solange es irgend geht. Wer mehr Genauigkeit braucht, ist mit Ganzzahlen eh besser bedient, weil sich damit erhöhte Genauigkeit mit wenigen Bytes zusätzlich erreichen lässt, ohne dass man zu 64-Bit-Monster-Zahlen greifen muss.

14.4 Umwandlung von Fließkommazahlen in Dezimal

Um binär-kodierte Fließkommazahlen in das Dezimalformat umzuwandeln und als ASCII-Zeichenkette ausgeben zu können, brauchen wir einen gehörigen Schatz an binärer und exponentieller Mathetik. Wenn Du in beiden Disziplinen arg schwächelst und zu faul bist, Dir dieses Elend reinzuziehen, dann nimm lieber einen PC-Prozessor mit fertiger Fließkommazahlen-Sub-Maschine oder eine vorgefertigte AVR-Bibliothek in C. Für alle anderen: es ist nicht so arg schwer zu verstehen, wie Fließkommazahlen ticken. Nur für den Prozessor ist es etwas aufwändiger, aber ansonsten besteht es nur aus Malnehmen und Teilen mit Zwei, aus ein wenig Links- und Rechtsschieben sowie aus Addieren ohne und mit Übertrag. Also nix Wildes.

14.4.1 Speichern von Zahlen

Das Problem hat der C-Programmierer schon mal gar nicht, das entscheidet sein Compiler schon für ihn, er wird da gar nicht erst damit behelligt.

Wie aus der den vorherigen Seiten hervorgeht, besteht eine 24-Bit-Binärzahl aus 16 Bits Mantisse und 8 Bits Exponent. Beide Teile haben in ihrem obersten Bit ein Vorzeichen. Die drei Bytes passen prima in normale Register.

Die Auflösung einer solchen Zahl in Dezimalformat umfasst 4 1/2 Ziffern für die Mantisse, den beiden Vorzeichen, einem **E** und dem zweistelligen Exponenten. Wenn wir noch eine oder zwei weitere Stellen für die Ausgabe vorsehen, sind wir schon bei 12 Zeichen. Wir tun also gut daran, dieses Monster in das SRAM zu stecken. In Registern wäre die Rechnerei schneller, aber dafür bliebe kaum Platz im unteren Rechenwerk für anderes.

Eine 15-Bit-Mantisse im Binärformat korrespondiert mit fünf dezimalen Ziffern (2^{15} = 32.768). Das Format ist daher folgendermaßen aufgebaut:

0	1	2	3	4	5	6	7	8	9	10	11
	1	,	0	6	6	6	5	E	+	0	1

Wir brauchen das Vorzeichen der Mantisse (wenn positiv: ein Leerzeichen), die normalisierte erste Stelle, dann das Dezimalkomma, vier signifikante und eine insignifikante Stelle, dann das **E**, das Vorzeichen des Exponenten (+ oder -) und zwei Exponenten-Ziffern. Insgesamt sind das 12 Zeichen Resultat.

Das ist der Platz, der für das Ergebnis reserviert werden sollte. Um diesen Platz zu reservieren, schreiben wir:

```
.dseg
.org SRAM_START
DecAsc:
  .byte 12 ; Reserviere 12 bytes fuer Resultat
```

Die Umwandlung braucht das Addieren von

- einer Auflösung von sechs Ziffern, damit wir etwas mehr an Genauigkeit haben,
- einen zusätzlichen Platz für weitere drei Ziffern, damit wir am Ende genauer runden können,
- einer zusätzlichen Ziffer links, um Überläufe behandeln zu können.

Daher sind wir insgesamt bei einer zehnziffrigen Zahl. Wir brauchen davon zwei Puffer: einen für die Berechnung des Mantissenwerts und einen für die Berechnung des Addierwertes für jedes Binär-Bit.

Um das Lesen der Werte im Simulator zu vereinfachen, habe ich noch etwas freien Platz im SRAM hinzugefügt. Der ist aber im Ernstfall unnötig.

```
.dseg
```

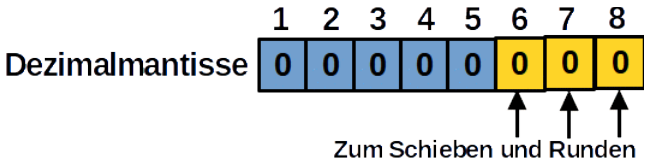
```
.org SRAM_START
.equ MantLength = 10
sMant:
  .byte MantLength
sMantEnd:
  .byte 16-MantLength
sAdder:
  .byte MantLength
sAdderEnd:
  .byte 16-MantLength
sDecAsc:
  .byte 12
```

Die zwei **End:**-Labels, die auf das Ende jedes Puffers zeigen, werden gebraucht, um für den Fall, dass wir von hinten nach vorne (z. B. beim Addieren) vorgehen müssen, einen Zeiger auf das Ende der Zahl zu haben.

Eine Grundentscheidung hier ist es, BCD-kodierte Ziffern zu verwenden. Möglich gewesen wäre es auch, gepacktes BCD-Format zu verwenden (verringert die Anzahl Instruktionen beim Addieren, muss aber die H-Flagge für Überläufe auswerten) oder man kann direkt ASCII-kodierte Ziffern verwenden (erspart die abschließende ASCII-Wandlung, braucht aber beim Addieren länger).

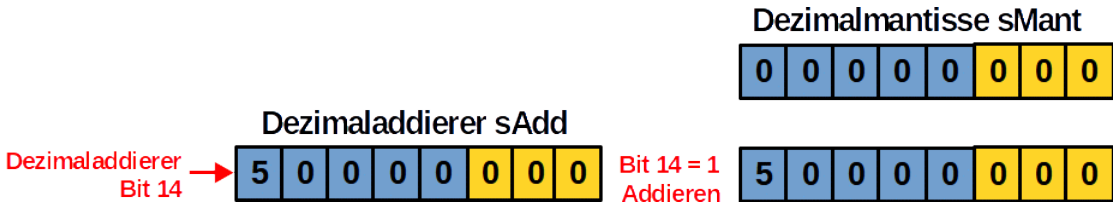
Im ersten Schritt des Programms müssen wir den Stapelzeiger setzen, denn es werden zum einfacheren Verständnis Unterroutinen verwendet.

Im zweiten Schritt werden wir das Vorzeichenbit der Mantisse los. Ist es negativ (Bit 7 des MSB's ist Eins), müssen wir die Mantisse von 0000 abziehen, was in diesem Fall ein Invertieren der beiden Mantissenbytes bedeutet. Das macht aus der negativen eine positive Mantisse. Die Dezimalmantisse sieht dann so aus.



14.4.2 Umwandlung der Mantisse in dezimal

Die Mantissenumwandlung beginnt mit Bit 14 in der binären Mantisse. Weil Bit 14 wegen der Normalisierung immer 1 ist, können wir das Vorgehen vereinfachen und dafür immer 0.50000000 einsetzen. In Assembler formulieren wir das dann so:

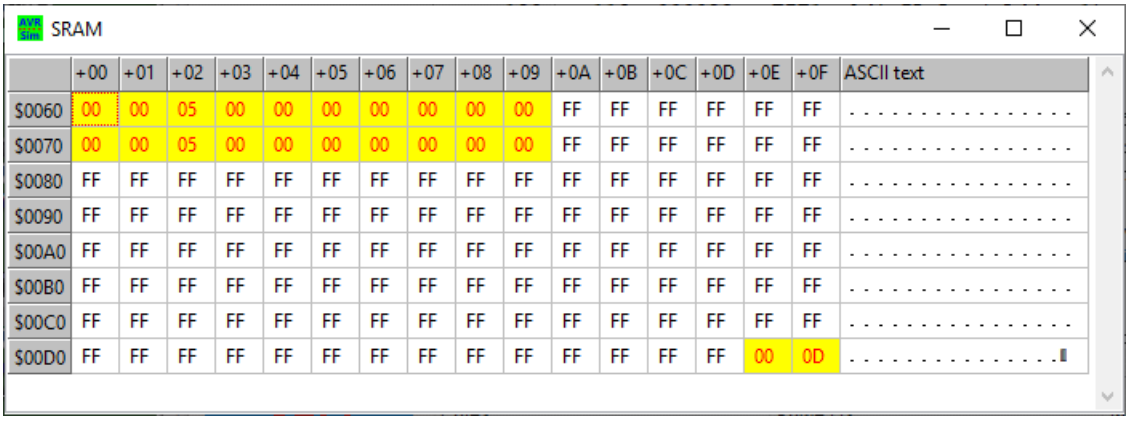



```
;
; Initiiere Dezimalmantisse und Addierer
InitMant:
  ldi ZH,High(sMant) ; Zeiger Z auf Mantissenanfang, MSB
  ldi ZL,Low(sMant) ; dto., LSB
  clr rmp ; Loeschen Bytes
  ldi rCnt,MantLength ; Laenge Mantisse
InitMant1:
  std Z+dAddMant,rmp ; Addierer loeschen
  st Z+,rmp ; Mantisse loeschen
  dec rCnt ; Am Ende?
```

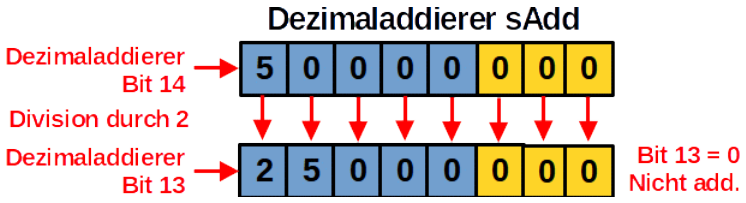
```
  brne InitMant1 ; Nein, weiter
  ldi rmp,5 ; Starte mit Bit 15
  sts sMant+2,rmp ; Setze Startwert, Mantissa
  sts sAdder+2,rmp ; und Addierer
  ret
```

Bitte dabei beachten, dass wir beide Puffer gleichzeitig beschreiben: der Befehl **STD** befüllt den Addierer im Abstand von dAddMant = sAdd - sMant gleich mit. Die Reihenfolge von **STD** und **ST** ist nicht egal, weil **ST** auch gleich noch den Zeiger um Eins erhöht.

Der Init-Vorgang sieht im Simulator avr_sim so aus: beide Zahlen sind auf 0,5 gesetzt.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	00	00	05	00	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF	. . . . .
$0070	00	00	05	00	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	0D	. . . . .



Der nächste Schritt teilt den Addierer durch 2, um die Wertigkeit des Bit 13 zu kriegen. Diese Prozedur beginnt mit dem Teilen der **5** im Addierpuffer und setzt sich über den gesamten Puffer fort. Wenn das Teilen einer Ziffer durch 2 einen Rest ergibt (nach LSR ist das Bit 0 im Carry Eins, ist bei 5 der Fall), dann muss 10 zur nachfolgenden Ziffer addiert werden. Ist das Carry-Bit nach der letzten Ziffer noch gesetzt, muss der Addierer in der letzten Stelle um Eins erhöht werden (Aufrunden des Addierers). Da das einen Überlauf in die vorletzte Ziffer ergeben könnte, muss dies solange in den höheren Ziffern erfolgen, bis kein Überlauf mehr erfolgt.

Weil Bit 13 in der Testmantisse 0x5555 Null ist, entfällt das Addieren der geteilten Zahl zur Dezimalmantisse.

Das Teilen der Zahl geht in Assembler so:

```
;
; Dividiere Addierer durch zwei
DivideBy2:
  ldi ZH,High(sAdder+1) ; Zeige auf Adderer, MSB
  ldi ZL,Low(sAdder+1)
  clc ; Loesche Carry fuer Ueberlauf
  ldi rCnt,MantLength-2 ; Mantissenlaenge minus Eins in Zaehler
DivideBy2a:
  ld rmp,Z ; Lese Byte aus Addierer
  brcc DivideBy2b ; Carry nicht gesetzt, nicht 10 addieren
  subi rmp,-10 ; Addiere 10
DivideBy2b:
```

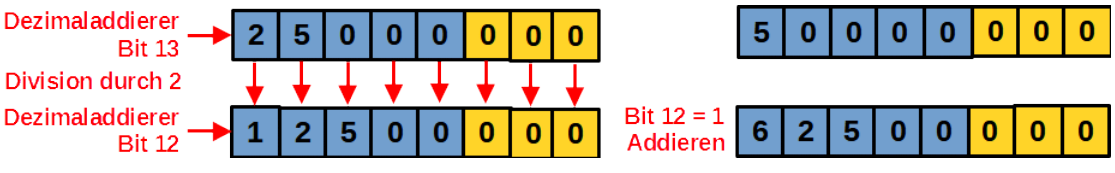
```
  lsr rmp ; Divididiere durch zwei
  st Z+,rmp ; Speichere Divisionsergebnis
  dec rCnt ; Abwaerts zaehlen
  brne DivideBy2a ; noch nicht fertig
  ld rmp,Z ; Lese letztes Byte des Addierers
  lsr rmp ; Teile durch 2
  st Z,rmp
  brcc Divideby2e
  inc rmp ; Letztes Byte aufrunden
Divideby2c:
  st Z,rmp ; Korriegiere letzte Ziffer
  subi rmp,10 ; Ziffer > 10?
  brcs DivideBy2e ; Nein
DivideBy2d:
  st Z,rmp ; Korrigiere letzte Ziffer
  ld rmp,-Z ; Lese vorletzte Ziffer
  inc rmp ; Erhoehe Ziffer
  st Z,rmp ; und speichere
  subi rmp,10 ; Ziffer >= 10?
  brcc DivideBy2d ; Ja, wiederhole
DivideBy2e:
  ret
```

Der Teil am Ende ab **Divide2c:** bzw. eine Zeile davor ist für das Aufrunden der letzten Ziffer verantwortlich. Dabei wird nach dem Abziehen von 10 die letzte Ziffer erneut abgespeichert und mit **LD rmp,-Z** die vorletzte Ziffer gelesen und ebenfalls um Eins erhöht, wenn das Carry gesetzt war. Dies wird solange wiederholt, bis das Carry beim Abziehen von 10 gesetzt ist.

SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	00	00	06	02	05	00	00	00	00	00	FF	FF	FF	FF	FF	FF	. . . . .
$0070	00	00	01	02	05	00	00	00	00	00	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	15	. . . . .

Das Teilen von 0,25 durch 2 (Zeile 2 der Simulation) ist hier gezeigt. Weil in der Testzahl 0x5555 das 12-te Bit eine Eins ist, wird dieser Addierer anschließend zur Dezimalmantisse hinzuaddiert. Beim Addieren zur Dezimalmantisse wird von hinten begonnen und nach links hin bearbeitet. Jede Ziffer, die 10 erreicht oder überschreitet, wird um zehn vermindert und eine Eins zur nächsten Ziffer hinzugezählt. Wenn das binäre Mantissenbit nicht Eins ist, erfolgt direkt das nächste Teilen durch zwei und der Addierer wird nicht addiert.

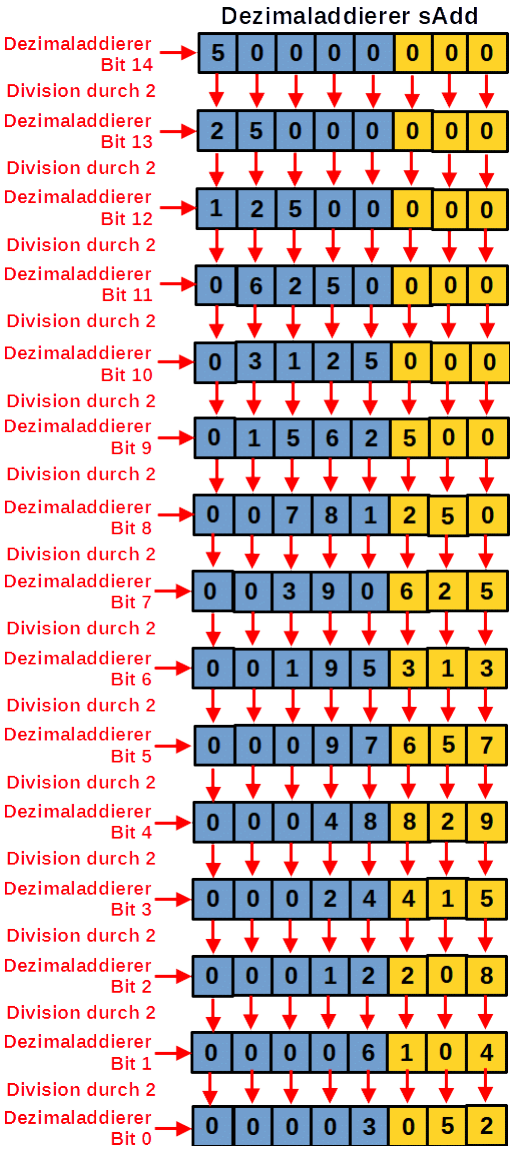


Der Quellcode für das Addieren ist hier:

```
;
; Addiere den Addierer zur Dezimalmantisse
MantAdd:
  ldi ZH,High(sMantEnd) ; Zeiger Z auf Dezimalmantisse Ende, MSB
  ldi ZL,Low(sMantEnd) ; dto., LSB
  ldi rCnt,MantLength-1 ; Mantissenlaenge
  clc ; Beginne mit Carry null
MantAdd1:
  ld rmp,-Z ; Lese letztes Mantissenbyte
  ldd rmp2,Z+dAddMant ; Lese zugehoeriges Addiererbyte
  adc rmp,rmp2 ; Addiere beide mit Carry
  st Z,rmp ; Speichere in SRAM
  subi rmp,10 ; Subtrahiere 10
  brcs MantAdd2 ; Carry gesetzt, kleiner als 10
  st Z,rmp ; Ueberschreibe Ziffer
  sec ; Setze Carry fuer naechste Ziffer
  rjmp MantAdd3 ; Zum Abwaertszaehlen
MantAdd2:
  clc ; Loesche Carry fuer naechstes addieren
MantAdd3:
  dec rCnt ; Abwaerts zaehlen
  brne MantAdd1 ; Noch nicht fertig
  ret
```

So wird jedes weitere binäre Mantissenbit ebenso behandelt, bis alle verarbeitet sind. Hier das letzte Teilen und Addieren bei einer Binärmantisse von 0x5555.

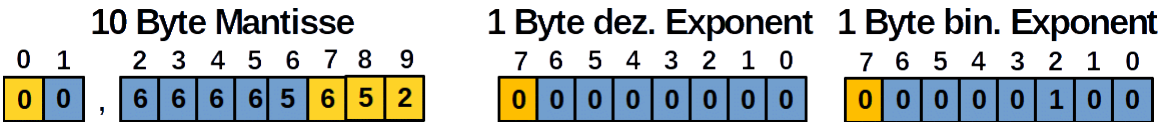
Das hier sind alle 15 Dezimaladdierer im Überblick. Man sieht, dass der letzte Addierer in der fünften Stelle noch mal 3,1 addiert. Um diese ist die fünfte Stelle ungenau. Hätten wir ein weiteres Bit gehabt (weil das 15.te Bit eigentlich überflüssig ist), wäre diese letzte Stelle mit 1,5 noch etwas genauer geworden.



14.4.3 Umwandeln der Exponenten-Bits

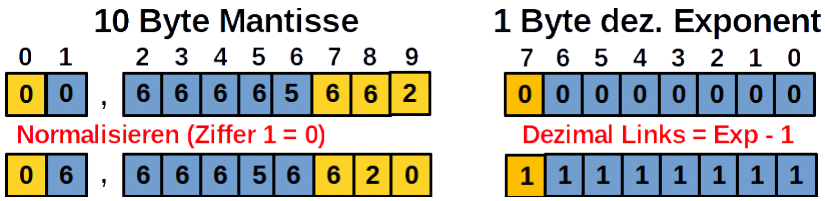
Dies sind nun alle relevanten Parameter, wenn es an das Umwandeln des Exponenten geht. Die drei Komponenten sind:

1. die dezimale Mantisse, wie sie aus der vorhergehenden Berechnung resultiert,
2. der dezimale Exponent, der mit Null beginnt und erhöht oder erniedrigt wird, und
3. der binäre Exponent, der nun auf die Mantisse losgelassen werden muss. Im Beispielfall ist das +4, er kann aber zwischen -128 und +127 liegen.



Im ersten Schritt muss die dezimale Mantisse normalisiert werden, denn ihre erste Ziffer (in der zweistelligen Zahl die zweite Ziffer) ist Null.

Dazu werden alle dezimalen Mantissenziffern um eine Position nach links geschoben. Es beginnt mit der linkesten Ziffer und geht bis zur vorletzten Ziffer. Die letzte Ziffer wird Null gesetzt.



Das bedeutet, die dezimale Mantisse wird mit zehn multipliziert. Entsprechend muss der dezimale Exponent um Eins vermindert werden. Aus der Null wird nun -1 oder 0xFF.

Ist die zweite Ziffer nach einer Normalisierung noch immer nicht Null, muss die Normalisierung weitere Male erfolgen, bis das der Fall ist (tritt in unserem Fall aber nicht auf).

Umgekehrt würde beim Normalisieren vorgegangen, wenn die allererste Ziffer nicht Null wäre (was beim Multiplizieren mit zwei nachfolgend mehrfach passieren wird). Dann müsste die ganze Zahl, von ganz rechts aus beginnend, um eine Position nach rechts geschoben werden. In die linkeste Position kommt dann eine Null. Der dezimale Exponent wird dann um Eins zu erhöhen sein.

Der Quellcode für die Normalisierung:

```
;
; Normalisieren der Dezimalmantisse
Normalize:
  lds rmp,sMant ; Lese Mantissen-Ueberlauf
  tst rmp ; Nicht Null?
  brne NormalizeRight ; Schiebe nach rechts
Normalize1:
  lds rmp,sMant+1 ; Erste Ziffer lesen
  tst rmp ; Null?
  breq NormalizeLeft ; Ja, links schieben
  ret ; Keine Normalisierung noetig
  ; Schiebe Mantisse eine Position links
NormalizeLeft:
  ldi ZH,High(sMant+1) ; Zeige auf erste Ziffer, MSB
  ldi ZL,Low(sMant+1) ; dto., LSB
  ldi rCnt,MantLength-2 ; Schieberzaehler
NormalizeLeft1:
  ldd rmp,Z+1 ; Lese naechstes Byte
  st Z+,rmp ; Kopiere in aktuelle Position
```

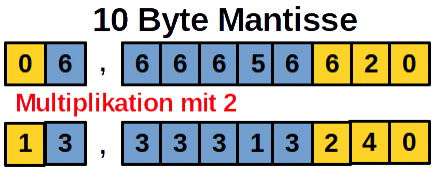
```
  dec rCnt ; Abwaerts zaehlen
  brne NormalizeLeft1 ; Weitere Schiebereien
  clr rmp ; Loesche letzte Ziffer
  st Z,rmp ; in der Pufferposition
  dec rDecExp ; Vermindere Dezimalexponent
  rjmp Normalize1 ; Weitere Schiebereinen erforderlich?
  ; Schiebe Zahl nach rechts
NormalizeRight:
  ldi ZH,High(sMantEnd-1) ; Z auf Ende Mantisse, MSB
  ldi ZL,Low(sMantEnd-1) ; dto., LSB
  ldi rCnt,MantLength-1 ; Zaehler
NormalizeRight1:
  ld rmp,-Z ; Lese Ziffer links
  std Z+1,rmp ; Speichere eine Position rechts
  dec rCnt ; Abwaertszaehlen
  brne NormalizeRight1 ; Weitere Ziffern
  clr rmp ; Erste Ziffer loeschen
  st Z,rmp ; in Mantissen-Ueberlauf
  inc rDecExp ; Erhoehe Dezimalexponent
  ret
```

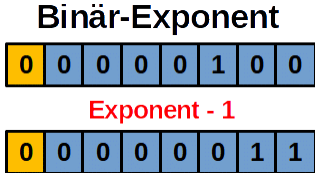
SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	00	06	06	06	06	05	06	03	00	00	FF	FF	FF	FF	FF	FF	. . . . .
$0070	00	00	00	00	00	00	03	00	05	00	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	55	00	17	. . . . . U . .

Das war nun die erste Normalisierung: die zweite Ziffer ist jetzt nicht mehr Null, die erste Ziffer ist Null geblieben. Am Ende der Zahl ist eine Null hinzugefügt worden.

Das Normalisieren hat am binären Exponenten nichts verändert: er ist weiterhin



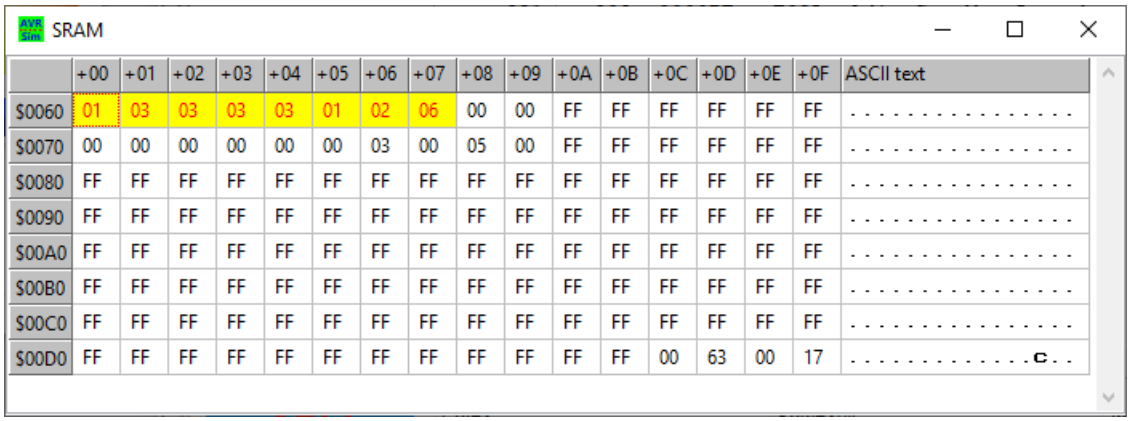


gleich vier. Das muss sich nun ändern: ist er positiv, muss es abwärts gehen, ist er negativ muss es aufwärts gehen. Zielwert ist in beiden Fällen die Null, dann sind wir fertig.

Damit es abwärts geht, muss die Mantisse mit zwei malgenommen werden. Damit es aufwärts geht, muss sie durch zwei geteilt werden.

Hier wird die dezimale Mantisse mit zwei malgenommen. Wie man sieht, erscheint in der linkesten Ziffer ein Überlauf, weil die erste Mantissenziffer bei Verdoppelung größer als 9 wird. Das muss dann anschließend wieder fein weg-normalisiert werden.

Der Binär-Exponent wird jetzt um Eins niedriger.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	01	03	03	03	03	01	02	06	00	00	FF	FF	FF	FF	FF	FF	. . . . .
$0070	00	00	00	00	00	00	03	00	05	00	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	63	00	17	. . . . . c . .

Der Quellcode der Multiplikation mit zwei:

```
;
; Multipliziere Zahl mit 2
Multiply2:
  ldi ZH,High(sMantEnd) ; Z auf Ende Mantisse, MSB
  ldi ZL,Low(sMantEnd) ; dto., LSB
  ldi rCnt,MantLength ; Ueber gesamte Laenge
  clc ; Carry = 0 zu Beginn
Multiply2a:
  ld rmp,-Z ; Lese letzte Ziffer
  rol rmp ; Multipliziere mit 2 und addiere Carry
  st Z,rmp ; Ueberschreibe Ziffer
  subi rmp,10 ; Subtrahiere 10
  brcs Multiply2b ; Carry gesetzt, kleiner als 10
  st Z,rmp ; Ueberschreibe letzte Ziffer
  sec ; Setze Carry fuer naechsthoehere Ziffer
  rjmp Multiply2c ; Zum Abwaertszaehlen
Multiply2b:
  clc ; Loesche Carry fuer naechste Ziffer
Multiply2c:
  dec rCnt ; Abwaertszaehlen
  brne Multiply2a ; Weitere Ziffern behandeln
  ret
```

Die simulierte Multiplikation mit zwei ist oben zu sehen. Die Übertragsziffer ist nun Eins geworden, sie muss durch anschließenden Aufruf von **Normalize:** nach rechtsgeschoben werden. Dadurch erhöht sich der dezimale Exponent wieder um Eins (war vorher -1, wird jetzt Null.

Dasselbe passiert, wenn der Binärexponent negativ ist, nur muss dann nicht mit zwei multipliziert sondern dividiert werden. Führt die Division zu einer Null in der ersten Ziffer, muss die wieder nach links wegnomalisiert werden.

Das Dividieren geht im Quellcode so:

```
;
; Dividiere Zahl durch 2
Divide2:
  ldi ZH,High(sMant) ; Zeiger auf Mantisse, MSB
  ldi ZL,Low(sMant) ; dto., LSB
  ldi rCnt,MantLength ; Alle Ziffern behandeln
```

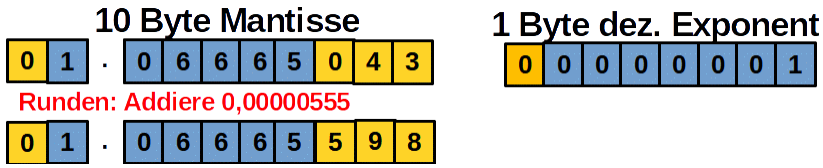
```
  clc ; Carry loeschen zu Beginn
Divide2a:
  ld rmp,Z ; Ziffer lesen
  brcc Divide2b ; Wenn carry gesetzt 10 addieren
  subi rmp,-10 ; 10 addieren
Divide2b:
  lsr rmp ; Mit 2 malnehmen
  st Z+,rmp ; und speichern
  dec rCnt ; Abwaertszaehlen
  brne Divide2a ; Weitere Ziffern dividieren
  brcc Divide2d ; Carry ist clear, nicht aufrunden
Divide2c:
  inc rmp ; Aufrunden
  st Z,rmp ; Speichern
  subi rmp,10 ; Ueberlauf?
  brcs Divide2d ; Nein
  ld rmp,-Z ; Lese naechsthoehere Ziffer
  rjmp Divide2c ; Erhoehe weiter
Divide2d:
  ret
```

Diese Schritte mit Multiplikation/Division wiederholen sich so lange, bis der binäre Exponent Null erreicht. Das gleiche Spiel wiederholt sich nun drei weitere Male bis der binäre Exponent Null geworden ist.

Beim vierten Verdoppeln tritt wieder ein Überlauf ein, der wieder durch Normalisieren bereinigt werden muss. Der dezimale Exponent geht dann von Null auf Eins. Bei noch größerem binären Exponenten tritt so ein Überlauf etwa alle vier Mal auf, so dass wir maximal zirka Exponenten von 128 / 4 = 32 kriegen.

14.4.4 Runden der Dezimalmantisse

Die letzten drei Dezimalstellen hatten wir zum Schieben und Runden mitgeschleppt. Das wird nun ausgeführt, indem wir zum Ergebnis 0.00000555 hinzuaddieren. Das sollte zu einem vollen Rundungserfolg führen, so dass unsere fünfte Stelle nach dem Komma etwas genauer wird.

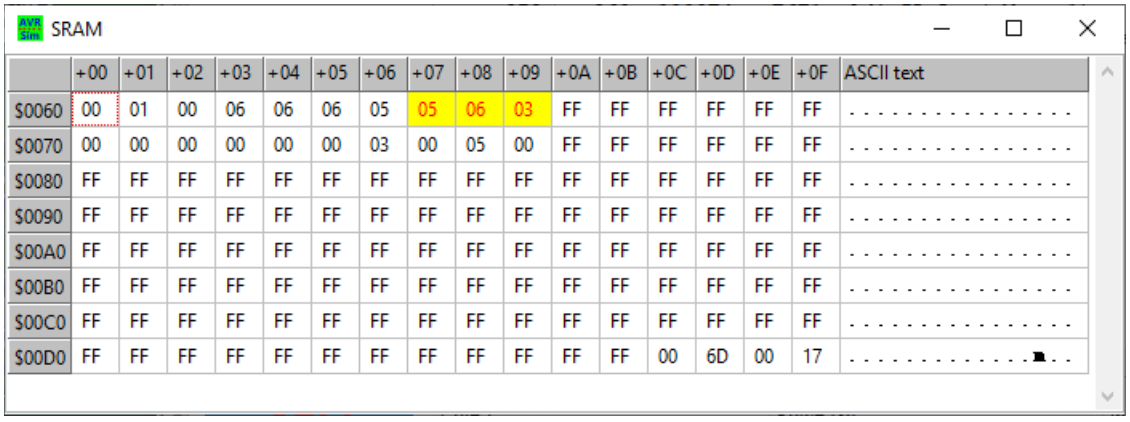


Runden geht in Assembler dann so:

```
;
; Aufrunden der Mantisse
RoundUp:
```

```
  ldi ZH,High(sMantEnd) ; Zeiger auf Ende Mantisse, MSB
  ldi ZL,Low(sMantEnd) ; dto., MSB
  ldi rmp2,5 ; Fuenfen addieren
  ldi rCnt,3 ; Drei mal
  clc ; Carry loeschen
RoundUp1:
  ld rmp,-Z ; Letzte Ziffer lesen
  adc rmp,rmp2 ; Fuenf mit Carry addieren
  st Z,rmp ; Speichern
  subi rmp,10 ; Zehn abziehen
  brcs RoundUp2 ; Nicht groesser als 9
  st Z,rmp ; Ziffer - 10 speichern
  sec ; Carry-Flagge setzen
  rjmp RoundUp3 ; Zum Zaehlen
RoundUp2:
  clc ; Carry-Flagge loeschen
RoundUp3:
  dec rCnt ; Abwaertszaehlen
  brne RoundUp1 ; Weitere Ziffern
  ldi rmp2,0 ; Nullen addieren
  ldi rCnt,MantLength-3 ; Anzahl weitere Ziffern
RoundUp4:
  ld rmp,-Z ; Letzte Ziffer lesen
  adc rmp,rmp2 ; Addieren von Null und Carry
  st Z,rmp ; speichern
  subi rmp,10 ; Zehn abziehen
  brcs RoundUpRet ; Carry gesetzt, Ende
  dec rCnt ; Abwaerts zaehlen
  brne RoundUp4 ; Weiter erhoehen
  rcall Normalize ; Am Ende noch mal normalisieren
RoundUpRet:
  ret
```

In den ersten drei Durchläufen von hinten wird fünf zu jeder Ziffer hinzuaddiert, bei allen weiteren Durchläufen wird nur der mögliche Überlauf addiert. Treten Überläufe bis hin zur ersten Ziffer auf, muss wieder normalisiert werden.

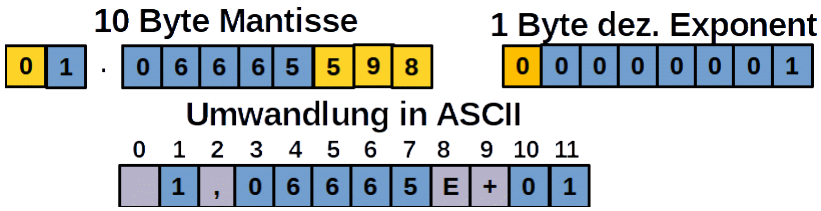


	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	00	01	00	06	06	06	05	05	06	03	FF	FF	FF	FF	FF	FF	. . . . .
$0070	00	00	00	00	00	00	03	00	05	00	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	6D	00	17	. . . . .

In diesem Fall trat bei der Addition von 555 kein Übertrag in die letzte Stelle statt.

Die Dezimalmantisse und der Dezimalexponent sind jetzt komplett für die Umwandlung in die ASCII-Zahl.

14.4.5 Wandlung von BCD in ASCII



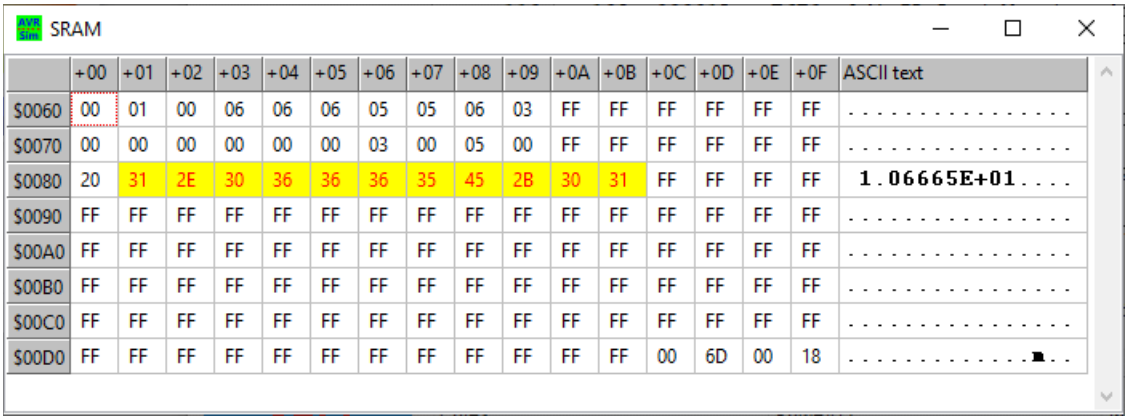
Alle Ziffern sind BCD-Zahlen. Wir müssen daher 0x30 hinzu addieren (oder -'0' subtrahieren), um ASCII-Ziffern zu erhalten. Natürlich müssen wir der Zeichenkette noch folgendes hinzufügen:

1. das Vorzeichen der Dezimalmantisse, wenn die Zahl negativ war (wenn nicht: ein Leerzeichen),
2. den Dezimalpunkt,
3. wenn der Dezimalexponent Null ist, vier Leerzeichen. Wenn nicht: **E** und das Vorzeichen des Exponenten und den Exponenten als zweiziffrige Zahl (zehn abziehen bis zum Unterlauf und Anzahl minus Eins als Zehner schreiben, Rest sind dann die Einer).

Der Quellcode hierfür:

```
;
; In Ascii-Zeichenkette umwandeln
ToAsc:
  ldi ZH,High(sDecAsc) ; Zeiger auf ASCII Zeichenkette, MSB
  ldi ZL,Low(sDecAsc) ; dto., LSB
  ldi XH,High(cMantissa) ; Mantissenvorzeichen lesen
  ldi rmp,' ' ; Positiv
  lsl XH ; Vorzeichen in Carry
  brcc ToAsc1 ; Nicht negativ
  ldi rmp,'-' ; Vorzeichen negativ
ToAsc1:
  st Z+,rmp ; Vorzeichen in Zeichenkette
  ldi XH,High(sMant+1) ; X als Zeiger auf Dezimalmantisse, MSB
  ldi XL,Low(sMant+1) ; dto., LSB
  ld rmp,X+ ; Lese erste Mantissenziffer
  subi rmp,-'0' ; in ASCII
  st Z+,rmp ; und schreibe in Zeichenkette
  ldi rmp,',' ; Dezimalkomma
  st Z+,rmp ; in Zeichenkette
  ldi rCnt,MantLength-5
ToAsc2:
  ld rmp,X+ ; Lese Ziffer
  subi rmp,-'0' ; in ASCII
  st Z+,rmp ; in Zeichenkette
  dec rCnt ; Abwaerts zaehlen
  brne ToAsc2 ; Weitere Ziffern kopieren
  tst rDecExp ; Dezimalexponent = 0?
  breq ToAscExpBlk ; Vier Leerzeichen ausgeben
  ldi rmp,'E' ; Exponentenzeichen
  st Z+,rmp ; in Zeichenkette
  ldi rmp,'+' ; Vorzeichen Exponent
  sbrs rDecExp,7 ; Bei negativ ueberspringen
  rjmp ToAsc3 ; Exponent positiv
  neg rDecExp ; 0 minus Exponent
  ldi rmp,'-' ; Minuszeichen
ToAsc3:
```

```
  st Z+,rmp ; Vorzeichen ausgeben
  ldi rmp,'0'-1 ; Exponentenzehner
ToAsc4:
  inc rmp ; Naechster Zehner
  subi rDecExp,10 ; 10 abziehen
  brcc ToAsc4
  st Z+,rmp ; Zehner ausgeben
  ldi rmp,'0'+10 ; Einer
  add rmp,rDecExp ; in ASCII
  st Z,rmp ; und in Zeichenkette
  ret
ToAscExpBlk:
  ldi rmp,' ' ; Leerzeichen
  ldi rCnt,4 ; Vier Stueck
ToAscExpBlk1:
  st Z+,rmp ; in Zeichenkette
  dec rCnt ; Abwaerts zaehlen
  brne ToAscExpBlk1 ; Weitere Zeichen
  ret
```



Das sieht nun ganz so aus, als könnte und wollte es auf eine LCD geschrieben werden.

14.4.6 Ausführungszeiten

Wenn Dein Programm es eilig hat und den Prozessor eh schon ziemlich auslastet, dann ist die Umwandlung von Fließkommazahlen in Dezimal genau der Sargnagel, den Du gebrauchen kannst, um ihm den Rest zu geben.

Mantisse	***Exponent***	***Dauer***
0x4000	0x00	448 µs
	0x01	668 µs
	0x02	816 µs
	0x10	2,15 ms
	0x7F	23,2 ms
0xFFFF	0x00	449 µs
0x5555		2,88 ms
0x7FFF		3,87 ms

Die Fälle mit negativen Mantissen und Exponenten brauchen nur unwesentlich mehr Zeit (zwei zusätzliche Instruktionen NEG und einmal COM).

Wenn Du den Assembler-Quellcode für eigene Experimente oder zum Erweitern auf 32, 40 oder 64 Bits brauchst (380 Zeilen), dann ist er hier fliesskomma24.asm zugänglich.

14.4.7 Schnellere Methode: eine binäre 40-Bit-Fließkommazahl in dezimal umwandeln

Die oben gezeigte Methode ist nicht sehr effektiv, weil sie eine Menge SRAM verwendet und jede Ziffer in ein eigenes Byte packt. Würden wir die 16-Bit-Mantisse auf eine 32-Bit-Mantisse erweitern, wie sie 40-Bit-Fließkommazahlen verwenden, dann würden wir eine langwierige Umwandlungsroutine kriegen, die 50 ms und länger dauert. Wir brauchen daher eine optimierte Variante, und die macht es dann so:

1. Zuerst wird die 32-Bit-Mantisse in einen Integer-Wert umgewandelt. Da ein 32-Bit-Wert Zahlen bis 2^{33}-1 = 8.589.934.591 = 0x01FFFFFFF annehmen kann, brauchen wir fünf Bytes für diese dezimale Ganzzahl. Wir packen links noch ein Byte hinzu, für den Fall, dass bei einer Multiplikation mit zehn ein Überlauf auftreten sollte. Diese Zahl heißt im Quellcode rAdd5:4:3:2:1:0. Für jedes der 32 Mantissen-Bits wird dessen dezimale Wertigkeit gebildet, indem ausgehend von 50.000.000.000 oder hexadezimal 0B.A4.3B.74.00 dieses jeweils durch zwei geteilt wird (natürlich unter Aufrunden der jeweils letzten Teilung) diese Wertigkeiten jeweils dann zusammengezählt werden, wenn das jeweilige Mantissenbit eine Eins ist.
2. Dann wird der binäre Exponent verarbeitet. Ist der positiv, dann wird die Zahl in rAdd5:4:3:2:1:0 solange mit zwei multipliziert wie der Exponent es sagt. Tritt bei dieser Multiplikation ein Überlauf nach rAdd5 auf, dann wird der dezimale Exponent um eins erhöht und die Zahl durch 10 geteilt. Ist der Exponent negativ, wird die Zahl so lange durch zwei geteilt bis der negative Exponent abgearbeitet ist. Tritt beim Zweiteilen der Fall ein, dass das höchste Byte rAdd4 Null wird, dann wird der dezimale Exponent um Eins vermindert und die Zahl mit 10 malgenommen (zweimal Linksschieben, Addieren von sich selbst, einmal Linksschieben).
3. Jetzt wird die Zahl in rAdd5:4:3:2:1:0 normiert. Ist sie größer als 100.000.000.000 (wenn entweder rAdd5 grßer Null ist oder wenn rAdd4:3:2:1:0 größer oder gleich 100.000.000.000), dann wird die Zahl durch zehn geteilt und der dezimale Exponent um Eins erhöht. Wurde diese Teilung nicht vorgenommen, dann wird überprüft, ob die Zahl kleiner ist als 1.000.000.000. Falls dies der Fall ist, wird die Zahl solange mit 10 malgenommen (unter Verminderung des dezimalen Exponenten), bis das der Fall ist.
4. Diese Ganzzahl wird nun in dezimale Ziffern umgewandelt, indem zuerst solange 1.000.000.000 abgezogen wird, bis ein Unterlauf auftritt. Das ergibt die erste Ziffer. Auf diese Ziffer folgt das Dezimalkomma (im Quellcode ist das ein Dezimalpunkt). Dann geht es mit 100.000.000 genauso weiter und bis herunter nach 10. Die letzte Ziffer ist der Rest der Zahl. Ist der dezimale Exponent nicht Null, wird er dann mit einem E, dem + oder - und nachfolgend als zweistellige Dezimalzahl angehängt.

Man beachte, dass das Dividieren durch 10 es erforderlich macht, alle sechs Bytes bzw. 48 Bits einzeln nach links zu schieben, zu prüfen, ob 10 erreicht sind und, falls ja, eine Eins, bzw. eine Null in das Ergebnis hereinzuschieben ist. Diese umfangreiche Schieberei ist zeitaufwändig und verlängert die Ausführungsdauer in Fällen, bei denen der positive binäre Exponent groß ist.

Die Tabelle rechts zeigt, dass für binäre Exponenten von 0x7F lange Zeitdauern zu Buche schlagen. Hier lohnt es sich, über Optimierungen wie die Division durch 100 oder 1000 nachzudenken. Alle anderen Fälle sind unkritisch.

Beim Vergleich der Zeiten mit der obigen Methode ist zu berücksichtigen, dass wir hier die doppelte Anzahl an Mantissenbits mit der doppelten Genauigkeit verarbeiten. Dafür sind die Zeiten recht kurz.

Binary		Decimal	Execution time
Mantissa	Exponent	Result	at 1MHz, ms
0x40000000	0x00	5.00000000000E-01	0.800
0x40000000	0x01	6.25000004800E-02	1.678
0x40000000	0x7F	8.50705917304E+37	40.145
0x40000000	0xFF	1.56250001200E-02	1.639
0x40000000	0x80	1.4936794650E-39	6.810
0x7FFFFFFF	0x00	9.9999999963E-01	2.239
0x7FFFFFFF	0x7F	1.701411833989E+38	41.840
0xBFFFFFFF	0x00	-5.00000000240E-01	1.480
0xBFFFFFFF	0x01	-1.00000000048	1.396
0x80000000	0x00	-9.99999999870E-01	2.310
0xBFFFFFFF	0x7F	-8.50705917716E+37	40.832
0xBFFFFFFF	0x80	-1.46936794650E-39	6.808

Der Quellcode in Assembler-Format ist hier zugänglich. Ich habe ihn nicht in Deutsch übersetzt, da er kaum mit Kommentaren glänzt und auch so verständlich sein sollte. Er arbeitet daher in englischer Zahlennotation. Für seriöse Anwendungen könnte es sich lohnen, ein weiteres Byte hinten anzuhängen, um ein Mehr an Genauigkeit zu erreichen. Register dafür sind noch satt vorhanden und verfügbar. Auch der erforderliche Umbau, um ein um ein Bit komprimiertes Format von Fließkommazahlen (mit unterdrücktem Bit 31) zu verarbeiten, ist sehr einfach.

14.4.8 Schlussfolgerung

Halte Dich von Fließkomma-Zahlen so fern als möglich. Sie fressen Deine Performance auf, blasen Deinen Code mit unnützem Beiwerk auf und sind, bis auf ganz wenige Ausnahmen, komplett unnötiger Müll. Mache Zeugs, für das Du unbedingt Fließkomma brauchst, auf einem PC, der kann das besser als ein AVR, weil er einen eigenen Fließkomma-Prozessor hat.

14.5 Umwandlung von Fließkommazahlen von Dezimal in Binär in Assemblersprache

Nachdem nun die Fließkommazahlen in ihrem Aufbau hier geklärt sind und die Umwandlung von binären Fließkommazahlen in dezimales Format hier gezeigt worden ist, fehlt nun noch das Gegenstück: die Umwandlung aus dezimalen Zahlenformaten in binäre Fließkommazahlen. Und das geht so.

14.5.1 Dezimale Zahlenformate

Es gibt ganz verschiedene dezimale Zahlenformate:

- **1** oder **123**: dezimale Ganzzahlen ohne Komma,
- **12,3**: dezimale Fließkommazahlen mit Dezimalkomma,
- **-1,234**: negative dezimale Fließkommazahlen mit Dezimalkomma,
- **1,2345E+2, 1,2345E+12**: dezimale Fließkommazahlen mit ein- oder zweistelligem positiven Dezimalexponenten,
- **1,23456E-12**: dezimale Fließkommazahlen mit negativen Dezimalexponenten,
- **-1,234567E-13**: negative dezimale Fließkommazahlen mit negativen Dezimalexponenten.

Um alle diese verschiedenen Arten von Dezimalzahlen korrekt in das binäre Fließkommazahlenformat verwandeln zu können, muss die Software

1. feststellen, ob die Dezimalzahl negativ ist (beginnt mit "-"),

2. die vorzeichenlose dezimale Mantisse in das binäre Zahlenformat umwandeln,
3. den dezimalen Exponenten ermitteln (Anzahl der Ziffern vor dem Dezimalkomma plus negativer oder positiver Exponent nach "E") und die binäre Mantisse um diese Anzahl mit 10 multiplizieren (Exponent größer Null) bzw. durch 10 teilen (Exponent kleiner als Null),
4. die binäre Mantisse normalisieren (höchstes Mantissenbit = 1), und
5. bei negativer Mantisse das Vorzeichenbit und die restliche Mantisse invertieren.

14.5.2 Assembler-Software für das Umwandeln

Im Assembler-Quellcode hier, fliesskomma40_d2b, ist die umzuwandelnde Zahl als Tabelle im ASCII-Zeichen-Format vorgegeben. Diese wird zuerst in das SRAM kopiert, damit beim weiteren Rechengang Doppelzugriffe auf zwei oder mehr Orte im Flash gleichzeitig vermieden werden können. Wer die Dezimalzahl schon im SRAM vorhanden hat, kann diese dorthin kopieren und mit einer binären Null abschließen (als null-terminierte Zeichenkette).

Die Software ist für binäre Mantissen bis 40 Bit Länge geschrieben und für 8 Bit breite Binär-Exponenten. Das entspricht 48-Bit-Binär-Zahlen. Wer weniger Genauigkeit braucht, kann die Software entsprechend um ein oder zwei Bytes kürzen und spart damit Verarbeitungszeit.

14.5.3 Negatives Vorzeichen feststellen

Der Zeiger X (XH:XL = R27:R26) zeigt auf den Beginn dieser Zeichenkette und liest das erste ASCII-Zeichen. Ist dies ein Minuszeichen, wird die Flagge **bMneg** gesetzt. Die Verarbeitung eines negativen Mantissen-Vorzeichens erfolgt dann später.

14.5.4 Dezimalmantisse einlesen und ins Binärformat umwandeln

Die Software beginnt dann das Einlesen der Mantisse. Das Dezimal-Trennzeichen (hier auf englisches Zahlenformat eingestellt: ".") werden dabei überlesen und erst im nächsten Arbeitsgang separat verarbeitet. Der Nullterminator 0x00 oder ein "E" beenden das Einlesen der Mantisse. Andere Zeichen als ASCII-Nullen bis ASCII-Neun führen dazu, dass die Software in die **ErrorLoop**-Schleife abtaucht. Das Register rmp (=R16) enthält dann ein ASCII-Zeichen, die den Grund dafür angeben:

- "0": Ziffer kleiner als Null,
- "9": Ziffer größer als Neun,
- "E": Exponent größer/kleiner als +/-39,
- "b": Binär-Exponent unterschreitet -128,
- "B": Binär-Exponent überschreitet 127.

Die einzelnen gelesenen Digits werden, beginnend mit 10.000.000.000 (0x02540BE400), malgenommen (durch fortgesetztes Addieren der fünf Bytes) und danach jeweils die nächstniedrige Zehnerpotenz zugrunde gelegt. Das Einlesen der Zehnerpotenzen aus der Tabelle vermeidet das Teilen durch 10 und sorgt daher für eine sehr flotte Umwandlung in das Binärformat. Ist die Tabelle fertig ausgelesen (erstes führendes Byte nicht Null), wird das weitere Einlesen von Ziffern eingestellt.

14.5.5 Binärmantisse berechnen

Um aus dieser eingelesenen Binärzahl nun das binäre Mantissenmuster zu erzeugen, werden alle Mantissenbits des Ergebnisses Null gesetzt. Bis das letzte Bit (Bit 0 in **rR0**): dieses wird Eins gesetzt, um das Ende der Umwandlung zu bewirken.

Beginnend mit der Dezimalzahl 1.000.000.000.000 (0xE8D4A51000) wird diese jeweils durch zwei geteilt und geprüft, ob die eingegebene Zahl größer oder gleich ist. Ist dies der Fall, wird die Dezimalzahl abgezogen und eine Eins in die Ergebnisregister eingeschoben. Wenn nicht, wird nicht abgezogen und eine Null eingeschoben. Wenn nach dem Einschieben eine Null in das Carry rotiert ist, wird weiter dividiert und abgezogen. Wenn nicht, ist die binäre Mantisse fertig umgewandelt.

14.5.6 Exponenten ermitteln und umrechnen

Nun geht es an die dezimalen Exponenten. Zunächst wird die Stellung des Dezimaltrenners in der Mantisse ermittelt, indem diese gelesen wird, das Minuszeichen überlesen und für jede Dezimalziffer vor dem Trenner der dezimale Exponent **rDExp** um Eins erhöht wird. Wird die Ende-Null 0x00 erreicht, ist die Ermittlung des Dezimalexponenten fertig, wenn stattdessen "E" erreicht wird, beginnt das Einlesen des Dezimalexponenten.

Wird ein "-" erkannt, wird die Flagge **bEneg** gesetzt. Plus-Zeichen werden überlesen, das Null-Zeichen 0x00 schließt das Einlesen des Exponenten ab. Jede gelesene Ziffer wird auf 0..9 geprüft, schon eingelesene Ziffern mit zehn multipliziert und die nächste Ziffer von ASCII in binär gewandelt und hinzuaddiert. Erreicht oder überschreitet der Dezimalexponent 40, gibt es eine Verzeigung zur Fehlerschleife.

Zuletzt wird der eingelesene Exponent zu dem Exponenten aus der Mantissenauswertung hinzuaddiert. Ist der Dezimalexponent positiv, wird die Mantisse solange mit 10 multipliziert wie es der Exponent angibt. Die Multiplikation erfolgt in der Unterroutine **Mult10**. Zuerst wird geprüft, ob das oberste Register **rR4** größer oder gleich 25 ist. Falls ja, wird die Mantisse solange durch zwei geteilt, bis dies der Fall ist. Dabei wird der Binär-Exponent um Eins erhöht, Überschreitungen von 127 werden durch Verzweigen in die Fehlerschleife abgefangen. Erst dann erfolgt das Multiplizieren mit 10 (Kopieren der Mantisse, Mantisse mal zwei, Mantisse mal zwei, Addieren der Kopie, Mantisse mal zwei).

Bei negativem Dezimalexponenten wird die Mantisse durch 10 geteilt. In der Software sind hierfür gleich zwei Routinen im Unterprogramm **Div10** enthalten:

1. die klassische Methode: 40-maliges Herausschieben eines Bits, Abziehen von 10, falls kein Carry: Einschieben einer Eins, falls Carry: Rücknahme des Abzugs, Einschieben einer Null ins Resultat.
2. die beschleunigte Methode: die bisherige Mantisse wird kopiert, zum Aufrunden wird fünf zur Mantisse hinzuaddiert, dann wird die Kopie durch zwei geteilt und van der Mantisse abgezogen, in einer Schleife wird dann jeweils die Kopie durch zwei geteilt und zweimal wird die geteilte Kopie abgezogen und zweimal nicht abgezogen. Das wird solange wiederholt bis die Kopie Null geworden ist.

Die Verwendung der beschleunigten Methode lohnt sich, wenn viele Teilungen durch 10 erfolgen müssen. Statt 14 ms für einen Dezimalexponenten von 1E-30 braucht die klassische Methode 24,55 ms, also fast das Doppelte an Zeit.

Die beschleunigte Methode ist hier für beliebige Teiler und hier für das Teilen durch 10 näher beschrieben.

14.5.7 Normalisierung und Verarbeitung von negativen Mantissen

Abschließend wird die binäre Mantisse normalisiert. Diese wird entweder nach rechts geschoben (wenn Bit 39 gesetzt sein sollte) bzw. solange nach links geschoben, bis Bit 38 der Mantisse eins ist. Natürlich erfolgt beides unter Anpassung des Binärexponenten. Wer die Normalisierung so braucht, dass das oberste Bit der Mantisse Eins ist und verschwindet, kann hier noch ein weiteres Mal linksschieben.

War die Flagge **rDneg** gesetzt, dann wird noch das gesamte Fünferpack invertiert.

Nach allen diesen Operationen ist das Ergebnis (binäre Mantisse in **rR4:rR3:rR2:rR1:rR0**, binärer Exponent in **rBExp**) nun komplett.

14.5.8 Ergebnisse

Die Tabelle rechts zeigt die Ergebnisse solcher Umwandlungen von dezimal in binär in ausgewählten Fällen. Dargestellt ist die Ausgangszahl, das binäre Ergebnis (Mantisse und Exponent), die dezimale Entsprechung dieser Binärzahl sowie die Ausführungszeiten in Millisekunden bei einem Takt von 1 MHz. In allen Fällen wurde mit der beschleunigten Divisions-Variante gerechnet.

	Converted to		Decimal	Time
Decimal	Mantissa	Exponent	Effective	ms @1MHz
„1“	0x40000000B5	0x02	1.000007630	1.198
„-1“	0xBFFFFFFF4B	0x02	-1.000007630	1.234
„12“	0x60000000EF	0x05	12.00006104	1.361
„123“	0x7B000000CA	0x08	123.0007324	1.526
„1.23“	0x4EB851EC0B	0x02	1.230064850	1.364
„.12345678901“	0x7E6B74DE18	0xFE	0.123456551	2.076
„1.2345678901“	0x4F03290ACF	0x02	1.234510670	2.122
„1.2345678901E+01“	0x62C3F34D7F	0x05	12.34564839	2.291
„1.23456678901E+30“	0x7CA8D4FDBC	0x65	1.2346090E+30	5.016
„1.2345678901E-30“	0x6428F8D0D8	0x9E	1.2346130E-30	14.228

Man erkennt, dass die rückgerechneten Zahlen ab der fünften bis sechsten Dezimalstelle von der Ausgangszahl abweichen. So ist 0,123456551 ab der zweiten 5 in der siebten Dezimalstelle unkorrekt, was beim Runden die sechste Dezimalstelle falsch werden ließe. Das entspricht der Erwartung, denn bei 40 Bits beträgt die Genauigkeit LOG2(40) ca. 5 1/2 Dezimalstellen.

14.5.9 Fazit

Wer den Herrn Professor mit extensiven Bitschieberen beschäftigen und vom Wesentlichen abweichen lassen will, nehme Fließkommazahlen. So eignet sich die Umwandlung von Dezimalzahlen in binäre Fließkommazahlen ganz hervorragend als Ersatz für Verzögerungsroutinen im Millisekunden-Bereich. Und das Schönste daran: keiner erkennt die eigentliche Primitivschleife dahinter. Besonders negative dezimale Exponenten wie 1E-36 sind bei abgeschaltetem Beschleunigungsschalter dankbare Objekte und sehr langzeitwirksam.

Viel Spaß beim Spielen mit der Software.

15 Umgang mit Festkommazahlen in AVR Assembler

15.1 Sinn und Unsinn von Fließkommazahlen

Oberster Grundsatz: Verwende keine Fließkommazahlen, es sei denn, Du brauchst sie wirklich. Fließkommazahlen sind beim AVR Ressourcenfresser, lahme Enten und brauchen wahnsinnige Verarbeitungszeiten. So oder ähnlich geht es einem, der glaubt, Assembler sei schwierig und macht lieber mit Basic und seinen höheren Genossen C und Pascal herum.

Nicht so in Assembler. Hier kriegst Du gezeigt, wie man bei 4 MHz Takt in gerade mal weniger als 60 Mikrosekunden, im günstigsten Fall in 18 Mikrosekunden, eine Multiplikation einer Kommazahl abziehen kann. Ohne extra Fließkommazahlen-Prozessor oder ähnlichem Schnickschnack für Denkfaule.

Wie das geht? Zurück zu den Wurzeln! Die meisten Aufgaben mit Fließkommazahlen sind eigentlich auch mit Festkommazahlen gut zu erledigen. Und die kann man nämlich mit einfachen Ganzzahlen erledigen. Und die sind wiederum in Assembler leicht zu programmieren und sauschnell zu verarbeiten. Das Komma denkt sich der Programmierer einfach dazu und schmuggelt es an einem festem Platz einfach in die Strom von Ganzzahlen-Ziffern rein. Und keiner merkt, dass hier eigentlich gemogelt wird.

15.2 Lineare Umrechnungen

Als Beispiel folgende Aufgabe: ein 8-Bit-AD-Wandler misst ein Eingangssignal von 0,00 bis 2,55 Volt und liefert als Ergebnis eine Binärzahl zwischen $00 und $FF ab. Das Ergebnis, die Spannung, soll aber als eine ASCII-Zeichenfolge auf einem LC-Display angezeigt werden. Doofes Beispiel, weil es so einfach ist: Die Hexzahl wird in eine BCD-kodierte Dezimalzahl zwischen 000 und 255 umgewandelt und nach der ersten Ziffer einfach das Komma eingeschmuggelt. Fertig.

Leider ist die Elektronikwelt manchmal nicht so einfach und stellt schwerere Aufgaben. Der AD-Wandler tut uns nicht den Gefallen und liefert für Eingangsspannungen zwischen 0,00 und 5,00 Volt die 8-Bit-Hexzahlen $00 bis $FF. Jetzt stehen wir blöd da und wissen nicht weiter, weil wir die Hexzahl eigentlich mit 500/255, also mit 1,9608 malnehmen müssten. Das ist eine doofe Zahl, weil sie fast zwei ist, aber nicht so ganz. Und so ungenau wollten wir es halt doch nicht haben, wenn schon der AD-Wandler mit einem Viertel Prozent Genauigkeit glänzt. Immerhin zwei Prozent Ungenauigkeit beim höchsten Wert ist da doch zuviel des Guten.

Um uns aus der Affäre zu ziehen, multiplizieren wir das Ganze dann eben mit 500*256/255 oder 501,96 und teilen es anschließend wieder durch 256. Wenn wir jetzt anstelle 501,96 mit aufgerundet 502 multiplizieren (es lebe die Ganzzahl!), beträgt der Fehler noch ganze 0,008%. Mit dem können wir einstweilen leben. Und das Teilen durch 256 ist dann auch ganz einfach, weil es eine bekannte Potenz von Zwei ist, und weil der AVR sich beim Teilen durch Potenzen von Zwei so richtig pudelwohl fühlt und abgeht wie Hund. Beim Teilen einer Ganzzahl durch 256 geht er noch schneller ab, weil wir dann nämlich einfach das letzte Byte der Binärzahl weglassen können. Nix mit Schieben und Rotieren wie beim richtigen Teilen.

Die Multiplikation einer 8-Bit-Zahl mit der 9-Bit-Zahl 502 (hex 01F6) kann natürlich ein Ergebnis haben, das nicht mehr in 16 Bit passt. Hier müssen deshalb 3 Bytes oder 24 Bits für das Ergebnis vorbereitet sein, damit nix überläuft. Während der Multiplikation der 8-Bit-Zahl wird die 9-Bit-Zahl 502 immer eine Stelle nach links geschoben (mit 2 multipliziert), passt also auch nicht mehr in 2 Bytes und braucht also auch drei Bytes. Damit erhalten wir als Beispiel folgende Belegung von Registern beim Multiplizieren:

Zahl	*Wert (Beispiel)*	*Register*
Eingangswert	255	R1
Multiplikator	502	R4:R3:R2
Ergebnis	128010	R7:R6:R5

Nach der Vorbelegung von R4:R3:R2 mit dem Wert 502 (Hex 00.01.F6) und dem Leeren der Ergebnisregister R7:R6:R5 geht die Multiplikation jetzt nach folgendem Schema ab:

1. Testen, ob die Zahl schon Null ist. Wenn ja, sind wir fertig mit der Multiplikation.
2. Wenn nein, ein Bit aus der 8-Bit-Zahl nach rechts heraus in das Carry-Flag schieben und gleichzeitig eine Null von links hereinschieben. Die Instruktion heißt Logical-Shift-Right oder LSR.
3. Wenn das herausgeschobene Bit im Carry eine Eins ist, wird der Inhalt des Multiplikators (beim ersten Schritt 502) zum Ergebnis hinzu addiert. Beim Addieren auf Überläufe achten (Addition von R5 mit R2 mit ADD, von R6 mit R3 sowie von R7 mit R4 mit ADC!). Ist es eine Null, unterlassen wir das Addieren und gehen sofort zum nächsten Schritt.
4. Jetzt wird der Multiplikator mit zwei multipliziert, denn das nächste Bit der Zahl ist doppelt so viel wert. Also R2 mit LSL links schieben (Bit 7 in das Carry herausschieben, eine Null in Bit 0 hineinschieben), dann das Carry mit ROL in R3 hineinrotieren (Bit 7 von R3 rutscht jetzt in das Carry), und dieses Carry mit ROL in R4 rotieren.
5. Jetzt ist eine binäre Stelle der Zahl erledigt und es geht bei Schritt 1 weiter.

Das Ergebnis der Multiplikation mit 502 steht dann in den Ergebnisregistern R7:R6:R5. Wenn wir jetzt das Register R5 ignorieren, steht in R7:R6 das Ergebnis der Division durch 256. Zur Verbesserung der Genauigkeit können wir noch das Bit 7 von R5 dazu heranziehen, um die Zahl in R7:R6 zu runden. Und unser Endergebnis in binärer Form braucht dann nur noch in dezimale ASCII-Form umgewandelt werden (siehe Umwandlung binär zu Dezimal-ASCII). Setzen wir dem Ganzen dann noch einen Schuss Komma an der richtigen Stelle zu, ist die Spannung fertig für die Anzeige im Display.

Der gesamte Vorgang von der Ausgangszahl bis zum Endergebnis in ASCII dauert zwischen 79 und 228 Taktzyklen, je nachdem wieviel Nullen und Einsen die Ausgangszahl aufweist. Wer das mit der Fließkommaroutine einer Hochsprache schlagen will, soll sich bei mir melden.

15.3 Beispiel: 8-Bit-AD-Wandler mit Festkommaausgabe

Assembler Quelltext der Umwandlung einer 8-Bit-Zahl in eine dreistellige Festkommazahl

```
; Demonstriert Fließkomma-Umwandlung in
; Assembler, (C)2003 www.avr-asm-tutorial.net
; Die Aufgabe: Ein 8-Bit-Analog-Wandler-Signal in Binärform wird eingelesen, die Zahlen
;    reichen von hex 00 bis FF. Diese Zahlen sind umzurechnen in eine Fließkommazahl
;    zwischen 0,00 bis 5,00 Volt.
; Der Programmablauf:
;    1. Multiplikation mit 502 (hex 01F6).
;       Dieser Schritt multipliziert die Zahl mit den Faktoren 500 und 256 und dividiert
;       mit 255 in einem Schritt.
;    2. Das Ergebnis wird gerundet und das letzte Byte abgeschnitten.
;       Dieser Schritt dividiert durch 256, indem das letzte Byte des Ergebnisses
;       ignoriert wird. Davor wird das Bit 7 des Ergebnisses  abgefragt und zum Runden des
;       Ergebnisses  verwendet.
;    3. Die erhaltene Zahl wird in ASCII-Zeichen umgewandelt und mit einem Dezimalpunkt
;       versehen.
;       Das Ergebnis, eine Ganzzahl zwischen 0 und und 500 wird in eine
Dezimalzahl ;       ;       verwandelt und in der Form 5,00 als Fließkommazahl in ASCII-
Zeichen dargestellt.
; Die verwendeten Register:
;    Die Routinen benutzen die Register R8 bis R1, ohne diese vorher zu sichern.
;    Zusätzlich wird das Vielzweckregister rmp verwendet, das in der oberen Registerhälfte
;    liegen muss. Bitte beachten, dass diese verwendeten Register nicht mit anderen
;    Verwendungen in Konflikt kommen können.
;    Zu Beginn wird die 8-Bit-Zahl im Register R1 erwartet.
;    Die Multiplikation verwendet R4:R3:R2 zur Speicherung des Multiplikators 502 (der
;    bei der Berechnung maximal 8 mal links geschoben wird - Multiplikation mit 2). Das
;    Ergebnis der Multiplikation wird in den Registern R7:R6:R5 berechnet.
```

```
;    Das Ergebnis der sogenannten Division durch 256 durch Ignorieren von R5 des
;    Ergebnisses ist in R7:R6 gespeichert. Abhängig von Bit 7 in R5 wird zum Registerpaar
;    R7:R6 eine Eins addiert. Das Ergebnis in R7:R6 wird  in das Registerpaar R2:R1
;    kopiert.
;    Die Umwandlung der Binärzahl in R2:R1 in eine ASCII-Zeichenfolge verwendet das Paar
;    R4:R3 als Divisor für die Umwandlung. Das Ergebnis, der ASCII-String, ist in den Re-
;    gistern R5:R6:R7:R8 abgelegt.
; Weitere Bedingungen:
;   Die Umwandlung benutzt Unterprogramme, daher  muss der Stapel funktionieren. Es werden
;   maximal drei Ebenen Unterprogrammaufrufe verschachtelt (benötigt sechs Byte SRAM).
; Umwandlungszeiten:
;   Die gesamte Umwandlung benötigt 228 Taktzyklen  maximal (Umwandlung von $FF) bzw. 79
;   Taktzyklen (Umwandlung von $00). Bei 4 MHz Takt entspricht dies 56,75 Mikrosekunden
;   bzw. 17,75 Mikrosekunden.
; Definitionen:
;   Register
.DEF rmp = R16 ; als Vielzweckregister verwendet
;   AVR-Typ
;   Getestet für den Typ AT90S8515, die Angabe wird nur für den Stapel benötigt. Die
;   Routinen arbeiten auf anderen AVR-Prozessoren genauso gut.
.NOLIST
.INCLUDE "8515def.inc"
.LIST
; Start des Testprogramms
; Schreibt eine Zahl in R1 und startet die Wandlungsroutine, nur für Testzwecke.
.CSEG
.ORG $0000
        rjmp main
main:
        ldi rmp,HIGH(RAMEND) ; Richte den Stapel ein
        out SPH,rmp
        ldi rmp,LOW(RAMEND)
        out SPL,rmp
        ldi rmp,$FF ; Wandle FF um
        mov R1,rmp
        rcall fpconv8 ; Rufe die Umwandlungsroutine
no_end:   ; unendliche Schleife, wenn fertig
        rjmp no_end
; Ablaufssteuerung der Umwandlungsroutine, ruft die verschiedenen Teilschritte auf
fpconv8:
        rcall fpconv8m ; Multipliziere mit 502
        rcall fpconv8r ; Runden und Division mit 256
        rcall fpconv8a ; Umwandlung in ASCII-String
        ldi rmp,'.' ; Setze Dezimalpunkt
        mov R6,rmp
        ret ; Alles fertig
;
; Unterprogramm Multiplikation mit 502
; Startbedingung:
; +----+
; | R1 |  Eingabezahl, im Beispiel $FF
; | FF |
; +----+
; +----+----+----+
; | R4 | R3 | R2 | Multiplikant 502 = $00 01 F6
; | 00 | 01 | F6 |
; +----+----+----+
;
;
; +----+----+----+
; | R7 | R6 | R5 | Resultat, im Beispiel 128.010
```

```
; | 01 | F4 | 0A |
; +----+----+----+
fpconv8m:
        clr R4 ; Setze den Multiplikant auf 502
        ldi rmp,$01
        mov R3,rmp
        ldi rmp,$F6
        mov R2,rmp
        clr R7 ; leere Ergebnisregister
        clr R6
        clr R5
fpconv8m1:
        or r1,R1 ; Prüfe ob noch Bits 1 sind
        brne fpconv8m2 ; Noch Einsen, mach weiter
        ret ; fertig, kehre zurück
fpconv8m2:
        lsr R1 ; Schiebe Zahl nach rechts (teilen durch 2)
        brcc fpconv8m3 ; Wenn das niedrigste Bit eine 0 war, dann überspringe den
Additionsschritt
        add R5,R2 ; Addiere die Zahl in R4:R3:R2 zum Ergebnis
        adc R6,R3
        adc R7,R4
fpconv8m3:
        lsl R2 ; Multipliziere R4:R3:R2 mit 2
        rol R3
        rol R4
        rjmp fpconv8m1 ; Wiederhole für das nächste Bit
; Runde die Zahl in R7:R6 mit dem Wert von Bit 7 von R5
fpconv8r:
        clr rmp ; Null nach rmp
        lsl R5 ; Rotiere Bit 7 ins Carry
        adc R6,rmp ; Addiere LSB mit Übertrag
        adc R7,rmp ; Addiere MSB mit Übertrag
        mov R2,R7 ; Kopiere den Wert nach R2:R1 (durch 256 teilen)
        mov R1,R6
        ret
; Wandle das Wort in R2:R1 in einen ASCII-Text in R5:R6:R7:R8
; +----+----+
; + R2 | R1 | Eingangswert 0..500
; +----+----+
; +----+----+
; | R4 | R3 | Dezimalteiler
; +----+----+
; +----+----+----+----+
; | R5 | R6 | R7 | R8 | Ergebnistext (für Eingangswert 500)
; | '5'| '.'| '0'| '0'|
; +----+----+----+----+
fpconv8a:
        clr R4 ; Setze Dezimalteiler auf 100
        ldi rmp,100
        mov R3,rmp
        rcall fpconv8d ; Hole ASCII-Ziffer durch wiederholtes Abziehen
        mov R5,rmp ; Setze Hunderter Zeichen
        ldi rmp,10 ; Setze Dezimalteiler auf 10
        mov R3,rmp
        rcall fpconv8d ; Hole die nächste Ziffer
        mov R7,rmp
        ldi rmp,'0' ; Wandle Rest in ASCII-Ziffer um
        add rmp,R1
        mov R8,rmp ; Setze Einer Zeichen
        ret
```

```
; Wandle Binärwort in R2:R1 in eine Dezimalziffer durch fortgesetztes  Abziehen des
; Dezimalteilers
;    in R4:R3 (100, 10)
fpconv8d:
        ldi rmp,'0' ; Beginne mit ASCII-0
fpconv8d1:
        cp R1,R3 ; Vergleiche Wort mit Teiler
        cpc R2,R4
        brcc fpconv8d2 ; Carry nicht gesetzt, subtrahiere Teiler
        ret ; fertig
fpconv8d2:
        sub R1,R3 ; Subtrahiere Teilerwert
        sbc R2,R4
        inc rmp ; Ziffer um eins erhöhen
        rjmp fpconv8d1 ; und noch einmal von vorne
; Ende der Fließkomma-Umwandlungsroutinen
; Ende des Umwandlungstestprogramms
```

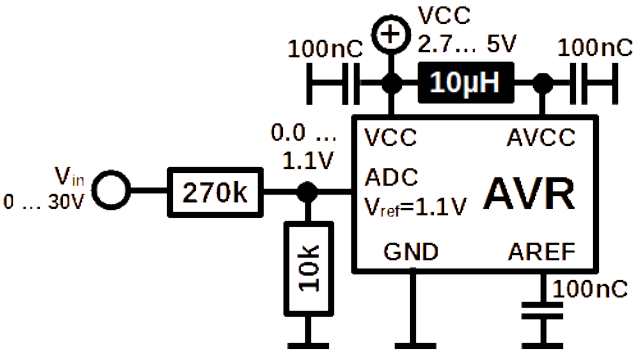
15.4 Beispiel: Spannungsmessung 0 … 30 V mit 10-Bit-AD-Wandler

Nehmen wir noch ein etwas komplizierteres Beispiel:

- Ein Spannungsteiler teilt eine Eingangsspannung von bis zu 30 Volt auf niedrigeres Niveau herab.
- Der AD-Wandler ist auf die interne Spannungsreferenz von 1,1 Volt eingestellt. Hat der 2,56 V, muss der Spannungsteiler halt anders werden.
- Die Spannung soll mit einer Anzeige von 0,00V bis 30,00V erfolgen.

15.4.1 Spannungsteiler

Zunächst der Spannungsteiler. Er muss die Eingangsspannung von 30 V auf die 1,1 V herabteilen. Die beiden Widerstände R1 (auf Null Volt) und R2 (auf 30 V) müssen daher einen Messstrom von I_{mess} = 1,1 V / R1 = (30 - 1,1) V / R2 haben. Wenn wir also R1 fest vorgeben, muss R2 = (30 - 1,1) * R1 / 1,1V sein. Mit R1 = 10 kΩ wären das R2 (kΩ) = (30 - 1,1) / 1,1 * 10 kΩ = 262,7 kΩ. Das läuft in der E12-Reihe auf 270 kΩ hinaus, damit wir auch die 30 V sicher gemessen und angezeigt kriegen.



Mit 10 kΩ und 270 kΩ produziert unser Spannungsteiler einen Messstrom von I_{mess} = 1,1 / 10 kΩ = 0,11 mA, was an dem 270 kΩ eine Spannung von 29,7 V abfallen lässt. Zusammen mit der AD-Wandler-Eingangsspannung von 1,1 V macht das 30,8 V Vollausschlag.

15.4.2 Umrechnung in eine Spannung

Liegt also am AD-Wandler-Eingang eine Spannung von 1,1 Volt an, dann sollte das Ergebnis der Wandlung bei einem 10-Bit-ADC 1.024 sein (genau genommen ein Digit weniger, nämlich 1.023, wenn statt 1,1 V nur 1,1 - 1,1/1024 am Eingang anlägen), angezeigt werden sollte aber 30,80V. Löschen wir für eine Weile das Komma (das erst später wieder eingeschmuggelt werden wird), dann sollte aus 1.024 die 3.080 herauskommen. Wir müssen die 1.024 vom ADC also mit 3,0078 malnehmen. Wir könnten das AD-Resultat nun zwei Mal zu sich selbst addieren und den 0,0078 einen Guten Tag sagen.

Wer es aber auf 1% genau nehmen will (so genau kriegen wir das mit zwei 1%-Widerständen schon noch hin), multipliziert die 3,0078 mit 256 und kriegt recht genau 770 heraus. Mit der (anstelle von 3 * 256 =

768) malgenommen ist das Resultat nun besser als 1% genau.

Vin (V)	Vadc (V)	ADC	770*ADC	Hex	Rd+Hex4	Dezimal	Anzeige
0,9	0,0321	30	23.100	005A3C	005A	90	0,90V
1	0,0357	33	25.410	006342	0063	99	0,99V
1,1	0,0393	37	28.490	006F4A	006F	111	1,11V
5	0,1786	166	127.820	01F34C	01F3	499	4,99V
9	0,3214	299	230.230	038356	0383	899	8,99V
12	0,4286	399	307.230	04B01E	04B0	1200	12,00V
15	0,5357	499	384.230	05DCE6	05DD	1501	15,01V
18	0,6429	598	460.460	0706AC	0707	1799	17,99V
24	0,8571	798	614.460	09603C	0960	2400	24,00V
30	1,0714	997	767.690	0BB6CA	0BB7	2999	29,99V

Die Tabelle zeigt die Spannungsberechnung. Je nach der Eingangsspannung liest der ADC seine durch die Spannungsreferenz (1,1 Volt) geteilte Spannung ein. Die Multiplikation mit 770 bringt eine 24-Bit-Binärzahl zustande, deren Wert in einer weiteren Tabellenspalte in Hexadezimal-Format angezeigt wird. Die beiden letzten Hexadezimalziffern dieser Zahl werden zum Runden verwendet: ab 0x7F wird die Zahl aufgerundet und Eins zu den vier oberen Hex-Ziffern hinzu addiert. Die Umwandlung dieser 16-Bit-Zahl in dezimales Format ergibt die Spannung, aber noch ohne Komma. Das wird dann an der richtigen Stelle noch hinzugefügt und ergibt zusammen mit einem angefügten V das Anzeige-Format.

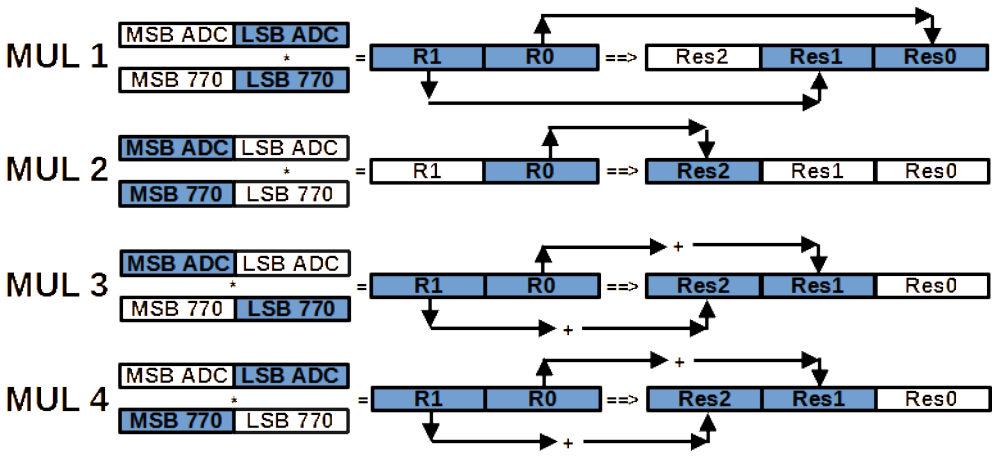
Die Tabelle zeigt, dass von den vier Stellen der Zahl nur die letzte Ziffer schwankt und mal 9 oder 1 ist, wenn sie eigentlich Null sein sollte. Das bestätigt die besser-als-1%-ige Toleranz der Messergebnisse.

Unglücklicherweise müssen wir nun eine 16-Bit- mal 16-Bit-Multiplikation machen, weil beide Zahlen etwas größer sind als 255. Das Resultat braucht aber nur 24 Bit lang sein (maximal 3.080 * 256), weil größere Ergebnisse schlicht nicht vorkommen können.

15.4.3 Multiplikation mit dem Hardware-Multiplizierer

Wer nun einen ATmega hat, kriegt das Multiplizieren mit dem eingebauten Multiplizierer schnell gebacken. Beides sind zwei-bytige Zahlen, mit jeweils einem LSB und einem MSB. Das Multiplizieren von (256*MSB1 + LSB1) mit (256*MSB2 + LSB2) ergibt den Term 65536*MSB1*MSB2 + 256*MSB1*LSB2 + 256*MSB2*LSB1 + LSB1*LSB2, braucht also vier Multiplikationen. Die Faktoren 65.536 und 256 werden dabei dadurch gebildet, indem die jeweils 16 Bit breiten Multiplikationsresultate jeweils in verschiedene höhere Bytes des Ergebnisses hinzuaddiert werden. Kann das MSB gar nicht Eins oder mehr werden, wie beim Multiplizieren der beiden MSBs, dann muss nur das LSB addiert werden.

Das Bild zeigt, wie es geht.



1. Als erstes werden die beiden LSBs mit der Instruktion MUL behandelt. Sie liefert das 16-Bit-Multiplikationsergebnis in den beiden Registern R1:R0. Diese werden in die beiden untersten Register des 24-Bit-Ergebnisses kopiert.
2. Dann werden die beiden MSBs miteinander multipliziert. Das LSB des Multiplikationsergebnisses wird in das höchste der drei Ergebnisregister kopiert (und damit mit 65.536 multipliziert). Das MSB im R1 ist Null und kann daher entfallen.
3. Die beiden nächsten Schritte multiplizieren jeweils das MSB der einen Zahl mit dem LSB der anderen. Beide Ergebnisse in R1:R0 werden zu den beiden Ergebnis-Registern 1 und 2 hinzuaddiert (R0 mit ADD, R1 mit ADC) und damit gleichzeitig mit 256 malgenommen.

So sieht der Quellcode dann aus:

```
; Konstante
.equ cMult = 770
; Register
.def rAdcL = R2 ; Geladen mit dem LSB aus dem ADC
.def rAdcH = R3 ; Geladen mit dem MSB aus dem ADC
.def rMultL = R4 ; Geladen mit der Konstante, LSB
.def rMultH = R5 ; dto., MSB
.def rRes0 = R6 ; Ergebnis-Byte 0, wird zum Runden verwendet
.def rRes1 = R7 ; dto., 1, LSB Ergebnis
.def rRes2 = R8 ; dto., 2, MSB Ergebnis
;
; Laden der Konstante
  ldi R16,Low(cMult) ; Lade Konstante, LSB
  mov rMultL,R16
  ldi R16,High(cMult) ; dto., MSB
  mov rMultH,R16
;
; Multiplikation der beiden LSB
  mul rAdcL,rMultL ; Multipliziere die LSBs
  mov rRes0,R0 ; Kopie in Resultat, LSB
  mov rRes1,R1 ; dto., MSB
; Multiplikation der beiden MSB
  mul rAdcH,rMultH ; Multpliziere die MSBs
  mov rRes2,R0 ; Kopie mal 65.536 in Resultat, nur LSB
; Multiplikation von LSB mit MSB
  mul rAdcL,rMultH ; Multipliziere LSB mit MSB
  add rRes1,R0 ; Addiere das 256-fache zum Resultat, LSB
  adc rRes2,R1 ; dto., MSB
; Multiplikation von MSB mit LSB
  mul rAdcH,rMultL ; Multipliziere MSB mit LSB
  add rRes1,R0 ; Addiere das 256-fache zum Resultat, LSB
  adc rRes2,R1 ; dto., MSB
  ldi rmp,0x7F ; Runden Resultat
  add rRes0,rmp
  brcc ToSram
  ldi rmp,0
  adc rRes1,rmp
  adc rRes2,rmp
ToSram:
```

Das Ganze braucht bei 1 MHz Takt nur 25µs lang, ist also verdammt schnell.

15.4.4 Multiplizieren ohne Hardware-Multiplizierer

Das Multiplizieren des AD-Resultats mit 770 ist eine schlicht langweilige Angelegenheit: es ist binär 0b0011.0000.0010 oder halt 3 * AD-Resultat mal 256 plus 2 * AD-Resultat.

Wir brauchen also eigentlich nicht die ganze Schieberei der Nullen, da nur drei Additionen fällig werden: eine mit dem Zweifachen und zwei mit dem 65536- und dem 65537-Fachen. Da ist also die Möglichkeit, sich die ganze Schieberei zu ersparen. Mal sehen, ob diese Optimierung schneller ist als die sowieso schon recht schnelle Hardware-Multiplikation mit ihren 25µs.

Der Quellcode geht so:

```
  mov rRes0,rAdcL
  mov rRes1,rAdcH
  add rRes1,rAdcL
  mov rRes2,rAdcH
  ldi rmp,0
  adc rRes2,rmp
  lsl rRes0
  rol rRes1
  rol rRes2
  add rRes1,rAdcL
  adc rRes2,rAdcH
```

Das Ganze braucht mit dem anschließenden Runden nur schlappe 17µs, ist also noch deutlich schneller als mit dem Hardware-Multiplikator. Dafür kann es jeder AVR-Typ, nicht nur ein ATmega. Fazit: Manchmal ist das Bit-Zählen und -Jonglieren aus der Abteilung Spezialmultiplikation schneller als jeder Hardware-Multiplikator.

Wer es allgemeingültiger haben will, so dass es nicht nur mit 770 als Multiplikator funktioniert, muss den mühsamen Schiebemechanismus durch-ixen. Hier der Quellcode:

```
  clr R0
  clr rRes0
  clr rRes1
  clr rRes2
Shift:
  lsr rMultH
  ror rMultL
  brcc Shift1
  add rRes0,rAdcL
  adc rRes1,rAdcH
  adc rRes2,R0
Shift1:
  lsl rAdcL
  rol rAdcH
  rol R0
  tst rMultL
  brne Shift
  tst rMultH
  brne Shift
```

Was so kurz aussieht, braucht jetzt aber ganz viel länger, nämlich 121µs. Hier lohnt sich dann mal der Einsatz des Hardware-Multiplikators.

15.4.5 Umwandlung in die ASCII-Anzeige

Nun stehen in beiden Fällen die beiden Ergebnis-Bytes in rRes2:rRes1 in binärem Format herum. Aus 0x0C05 möchte aber jetzt gerne mal der Text 30,77V werden.

Das SRAM ist ein guter Platz für solche Zeichenketten. Dazu brauchen wir einen Zeiger in das SRAM. Es ginge auch ohne, aber so ist es ein bisschen bequemer.

Nun läuft immer dasselbe ab: zuerst subtrahieren wir solange 1.000 bis das Ergebnis unterläuft. Und zählen dabei, wie oft das geht (hier: drei Mal). Die 1.000 zählen wir am Ende wieder hinzu, weil wir damit den Unterlauf korrigieren. Das Ergebnis prüfen wir auf ASCII-Null: die erste Ziffer wird ein Leerzeichen, wenn

eine Null herauskommt.

Danach ziehen wir, weiter in 16-Bit-Manier, immer 100 ab, bis der Unterlauf eintritt. Nach dem Schreiben dieser Ziffer schmuggeln wir das Dezimalkomma hier ein.

Die 10-er-Phase ist etwas kürzer, weil wir auf 8-Bit-Abziehen umstellen können.

Noch einfacher ist die letzte Ziffer: hier muss nur noch die ASCII-Null zum verbliebenen Zahlenrest addiert werden. Und danach kommt noch ein V hinzu.

```
ToSram:
  ldi ZH,High(sDecVtg)
  ldi ZL,Low(sDecVtg)
  ldi rmp,Low(1000)
  mov R0,rmp
  ldi rmp,High(1000)
  mov R1,rmp
  ldi rmp,'0'-1
ToSram1:
  inc rmp
  sub rRes1,R0
  sbc rRes2,R1
  brcc ToSram1
  add rRes1,R0
  adc rRes2,R1
  cpi rmp,'0'
  brne ToSram2
  ldi rmp,' '
ToSram2:
  st Z+,rmp ; Write
  ldi rmp,Low(100)
  mov R0,rmp
  ldi rmp,High(100)
  mov R1,rmp
  ldi rmp,'0'-1
ToSram3:
  inc rmp
  sub rRes1,R0
  sbc rRes2,R1
  brcc ToSram3
  add rRes1,R0
  adc rRes2,R1
  st Z+,rmp
  ldi rmp,'.'
  st Z+,rmp
  ldi rmp,10
  mov R0,rmp
  ldi rmp,'0'-1
ToSram4:
  inc rmp
  sub rRes1,R0
  brcc ToSram4
  st Z+,rmp
  add rRes1,R0
  ldi rmp,'0'
  add rmp,rRes1
```

```
st Z+,rmp
ldi rmp,'V'
st Z,rmp
```

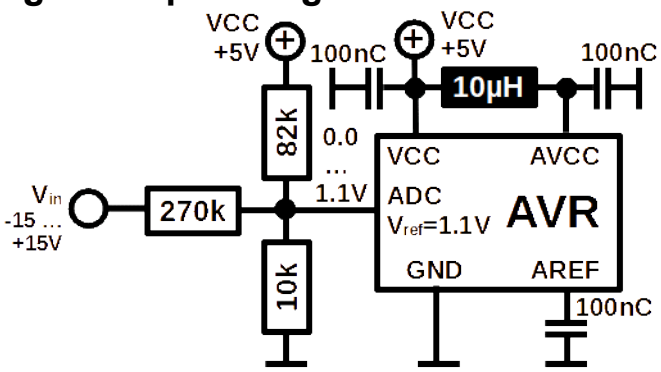
Diese ganze Wandlung dauert bei 1 MHz Takt 93µs, also auch nicht gerade üppig lange.

Die Software ist im Quellcode hier verfügbar.

15.5 Beispiel 4: Ein 10-Bit-AD-Wandler bei positiven und negativen Eingangsspannungen

Und noch ein Beispiel aus der Praxis: der AD-Wandler soll sowohl positive als auch negative Spannungen messen.

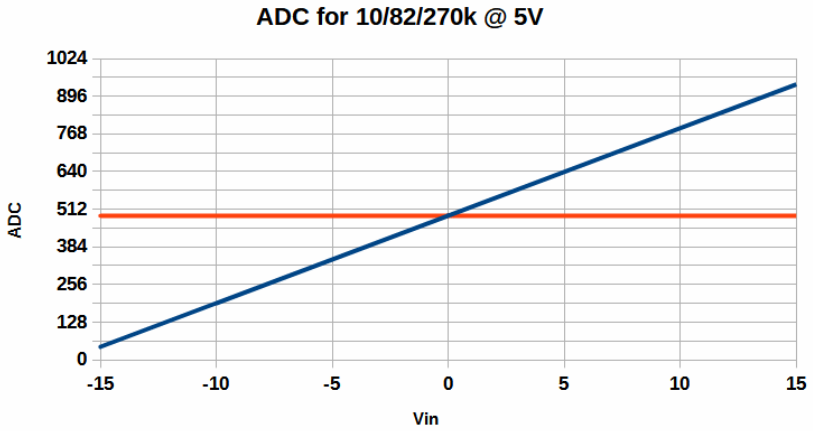
15.5.1 Der Vorteiler für positive und negative Spannungen



So sieht nun der Vorteiler aus: wieder teilt ein Spannungsteiler mit 270k und 10k die Eingangsspannung herunter, aber nun mischt sich ein 82k ein. Der zieht die ADC-Eingangsspannung in den positiven Bereich, indem er die Betriebsspannung von 5 V, ebenfalls abgebremst, zu der Messspannung hinzu addiert. Das führt dazu, dass negative Eingangsspannungen zu positiven Spannungen am ADC-Eingang werden.

Der Nachteil der Schaltung ist, dass sie nun eine geregelte Betriebsspannung braucht: bei +3,3 V müsste der 82k kleiner sein, bei +3,0 V noch kleiner. Das ist nun nicht mehr variabel und sorgfältig zu fixieren. Keinesfalls darf die ADC-Eingangsspannung negativ werden, das würde den Pin nachhaltig zerstören.

So sieht nun die Spannung aus, die der ADC bei Eingangsspannungen zwischen -15 V und +15 V produziert. Man erkennt, dass er bei 0 V Eingangsspannung nicht bei der Häfte (512), sondern bei nur 490 ankommt. Das kommt daher, weil ja bei Null Volt Eingangsspannung der 270k die ADC-Spannung ein wenig herunterzieht. Und das macht eben die 22 Unterschied aus. Die rote Linie zeigt eben gerade nicht auf 512.



Wer andere Eingangsspannungen, Betriebsspannungen und R-Kombinationen erproben möchte, findet hier den entsprechenden Rechensheet für eigene Experimente.

15.5.2 Umrechnung der ADC-Werte in Spannungen

Von den gemessenen ADC-Werten muss nun 490 subtrahiert werden. Kommt hierbei

- Null heraus, kommt in die Spannungszeichenkette schon mal „0,00“,
- kein Carry heraus, dann ist die Differenz mit 1.500 / ADC = 3,3632287 malzunehmen, um zu den angezeigten 15,00 V zu kommen. Da das nicht sehr gut geht, nehmen wir es mit 256 * 3,3632287 = 861 mal. Genau genommen wären das 860,987, aber die Abweichung von 861 ist nur 0,00156% mehr. Und das ist 1.000 mal weniger Abweichung als die 1%-Widerstände machen. Aus dem 24-Bit-Ergebnis verwenden wir das niedrigste Byte wieder zum Runden, wandeln es in Dezimal um, schmuggeln wieder das Komma rein und haben das Ergebnis im Format „xx,xxV“,
- ein Carry vor, dann ist das Ergebnis schon mal eine negative Spannung und beginnt mit „-“. Mit der Instruktion NEG werden das LSB und das MSB der Differenz dann positiv gemacht und mit dem gleichen Faktor multipliziert. Auch hier wird das niedrigste Byte wieder gestrichen und das dezimale Ergebnis, aber diesmal mit dem negativen Vorzeichen ausgegeben.

Das ist schon einfach, wenn man es kann. Nur ein Widerstand mehr, ein Jonglieren mit dem Vorzeichen und schon kann man auch negative Spannungen messen.

15.6 Fazit

Wer dieses Ganze nun mit Fließkomma-Mathematik bewältigen möchte, mache sich stattdessen auf Ausführungszeiten im Zehner-Millisekunden-Bereich gefasst, also 100-fach länger und braucht dazu auch noch jede Menge weitere Resourcen (jede Menge an Flash - in einen Tiny passt das bei Weitem nicht mehr rein -, Register, SRAM-Speicher). Zeit und Argumente, den Nachdenk-Turbo reinzulegen und das Ganze mit Pseudo-Komma-Zahlen wie hier gezeigt zu erledigen.

16 Adressierung von Speicherzellen in AVRs

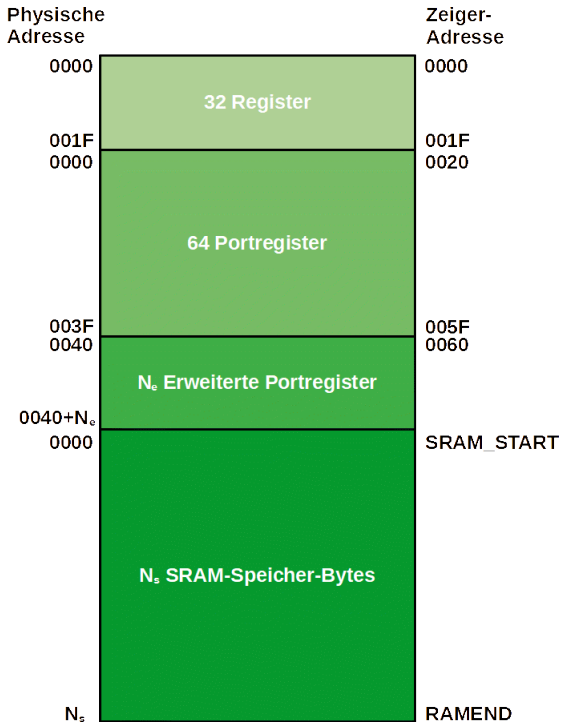
Speicherzellen zum Adressieren gibt es in AVRs an folgenden Orten:

1. im statischen RAM (SRAM)
2. in Portregistern
3. im Dauerspeicher (EEPROM)
4. im Programmspeicher (flash)

16.1 Adressieren von SRAM/Registern/Portregistern

Das allerwichtigste, wenn es um das Adressieren von Speicherzellen geht, ist schon mal der Unterschied in Adresstypen. Es gibt zwei davon:

1. Physische Adressen, und
2. Zeigeradressen.



Nur in Ausnahmefällen sind beide identisch, meistens sind sie unterschiedlich. Im Falle des statischen RAMs haben beide Adressen die folgenden Werte:

- Die physische Adresse des SRAM beginnt bei 0x0000. Wenn der AVR-Typ 1 kB SRAM hat, endet seine physische Adresse bei 0x03FF.
- Die Zeigeradresse beginnt bei der Adresse **SRAM_START**, welche als Konstante in der def.inc-Datei des AVR-Typs verewigt ist. Wenn der AVR-Typ nicht über erweiterte Portregister verfügt, ist das die Adresse 0x0060, in anderen Fällen kann es aber auch 0x0100 oder sogar noch höher sein.

Beim Zugriff auf SRAM wird immer nur die Zeigeradresse verwendet, die physische Adresse spielt hingegen keine Rolle. Das unterscheidet SRAM von Portregistern, bei denen beide Adressierungen eine Rolle spielen.

In der Konsequenz bedeutet das, dass beim ersten Umschalten des Assemblers mit der Direktive **.DSEG** auf das SRAM-Segment automatisch die Adresse SRAM_START als Segmentzeiger geladen wird. Woran merkt man das? Nun, man kann die def.inc-Datei nach SRAM_START durchsuchen. Man kann aber auch, etwas komfortabler, nach dem **.DSEG** ein Label setzen. Label sind dabei Marker, die immer eine Adresse festsetzen. Label-Symbole enden immer mit einem Doppelpunkt. Aufgabe der Umschaltung auf das Datensegment ist einzig das Setzen solcher Marker.

Wenn man jetzt mit meinem Assembler gavrasm mit gesetzter S-Option assembliert oder auch mit dem Simulator avr_sim assembliert, z. B. diesen Quellcode:

```
.nolist
.include "m324PBdef.inc" ; Definiere Typ ATmega324PB
.list
.dseg
DieErsteSramSpeicherzelle:
.cseg
```

dann findet sich im erzeugten Assembler-Listing am Ende diese Tabelle, aus der man bequem die Label-Adresse 0x0100 dieses AVR-Typs herauslesen kann. (Andere Assembler produzieren solche Tabellen übrigens nicht!)

```
List of symbols:
Type nDef nUsed          Decimalval          Hexval Name
  T     1     1                 165              A5 ATMEGA324PB
  L     1     0                 256            0100 DIEERSTESRAMSPEICHERZELLE
```

Damit man bequem auch das RAM-Ende herausfinden kann, geht man ein bisschen anders vor: mit **.equ MeinRamEnde = RAMEND** kopiert man die Konstante RAMEND in eine eigene Konstante und kann deren Wert dann aus der Symboltabelle (jetzt unter: Type C) herauslesen. Natürlich produziert das Zahlen, die für jeden AVR-Typ spezifisch sind.

Einfach die Direktive **.NOLIST** vor der def.inc-Include zu entfernen, kann nicht empfohlen werden, weil

man dann mit Hunderten Einträgen vollgemüllt wird, nach denen man gar nicht suchen will.

Wer mit dem Simulator avr_sim unterwegs ist, kann das alles noch bequemer haben: einfach unter "View" und "Symbols" mit dem Suchbegriff "RAM" suchen und schon taucht unter anderem RAMEND in der kurzen Liste auf.

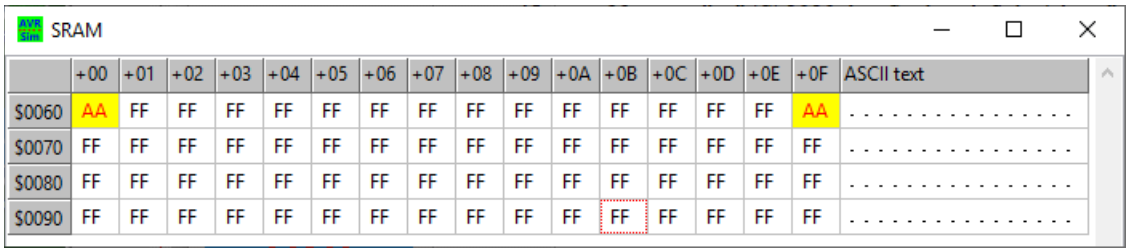
16.1.1 Adressierung von SRAM mit fester Adresse

SRAM kann eine physische Größe von bis zu 32.768 Bytes haben. Da die zu RAMEND umgeformte Zeigeradresse noch etwas höher ist, sind solche Adressen nicht nur 15 Bits lang sondern ganze 16 Bits. Zeigeradressen haben daher stets 16 Bits Länge.

Solche Zeigeradressen können direkt mit den Instruktionen **STS 16-Bit-Adresse,Register** oder **LDS Register,16-Bit-Adresse** beschrieben und gelesen werden. Die 16-Bit-Adresse folgt dabei auf das Instruktionswort, so dass beide Instruktionen aus zwei Befehlsworten bestehen. Als **Register** kann jedes zwischen R0 und R31 verwendet werden.

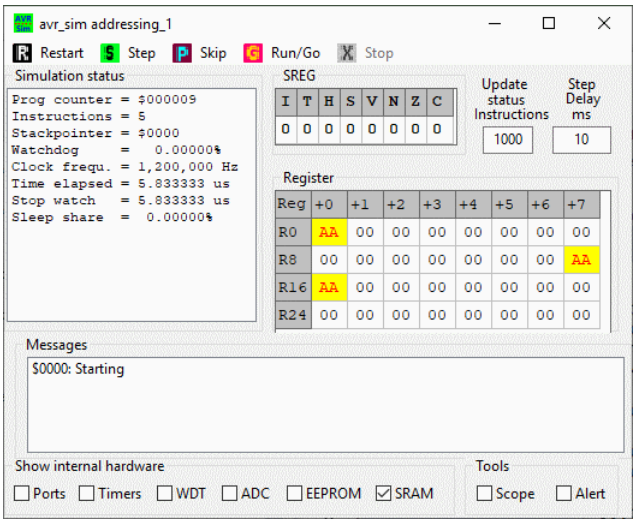
Der folgende Quellcode schreibt 0xAA (oder binär 0b10101010) in die erste und die 16-te Speicherzelle des SRAM. Um die Sinnhaftigkeit der Zeigeradressierung zu demonstrieren, wird dann dieses Byte auch noch in die Register R0 und R15 geschrieben.

```
.dseg
ErsteSramZelle: ; Ein label mit dieser Adresse
;
.cseg
  ldi R16,0xAA
  sts ErsteSramZelle,R16 ; Schreibe in die erste SRAM-Zelle
  sts ErsteSramZelle+15,R16 ; Und auch in die 16-te Zelle
  sts 0,R16 ; Und in das Register R0
  sts 15,R16 ; Sowie in das Register R15
```



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	AA	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	AA	. . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .

Oben sehen wir die beiden in das SRAM geschriebenen Bytes, rechts die in die Register geschriebenen Bytes.



Wenn wir nicht schreiben, sondern lesen wollen, geht das so:

```
  lds R16,ErsteSramZelle ; Lesen aus der ersten SRAM-Zelle
  lds R17,ErsteSramZelle+15 ; Lesen aus der 16-ten SRAM-Zelle
  lds R18,0 ; Lesen aus dem Register R0
  lds R19,15 ; Lesen aus dem Register R15
```

Wenn Du Deinen Controller gerne mit Unsinn beschäftigen willst: **lds R16,16** oder **sts 16,R16** sind dafür wunderschöne Beispiele: der Controller macht jeden Unsinn mit, kein Fensterchen warnt vor unnötigen und unsinnigen Instruktionen.

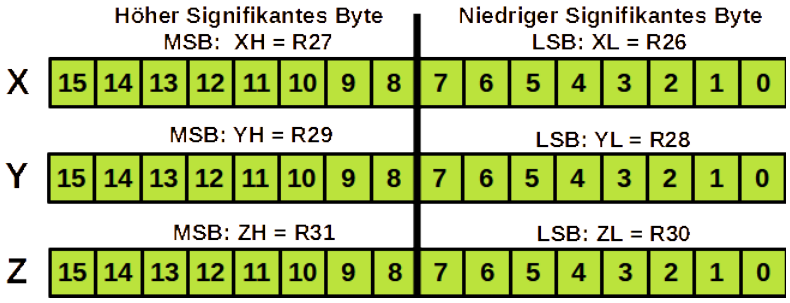
```
15: 000000   9100  lds R16,ErsteSramZelle ; Lesen aus der ersten SRAM-Zelle
    000001   0100
16: 000002   9110  lds R17,ErsteSramZelle+15 ; Lesen aus der 16-ten SRAM-Zelle
    000003   010F
17: 000004   9120  lds R18,0 ; Lesen aus dem Register R0
    000005   0000
18: 000006   9130  lds R19,15 ; Lesen aus dem Register R15
    000007   000F
```

Alle vier Lese-Instruktionen verwenden feste Adressen, die den LDS-Instruktionen einfach als weiteres Wort angefügt sind.

16.1.2 Zugriff auf SRAM-Zellen mit Zeigern

Um nicht nur auf einzelne Zellen des SRAM's zugreifen zu können, sondern auf ganze, mehr oder weniger große Bereiche des SRAM's zugreifen zu können, braucht es 16-Bit-Zeiger. Die AVR's haben gleich drei solcher Zeiger, nämlich Doppelregister, im Angebot:

- X = XH:XL = R27:R26,
- Y = YH:YL = R29:R28,
- Z = ZH:ZL = R31:R30.



Um das Doppelregister X auf die erste SRAM-Zelle zeigen zu lassen, brauchen wir die zwei folgenden Instruktionen:

```
.dseg
ErsteSramZelle: ; Platzieren eines Labels an diese Adresse
.cseg
  ldi XH,High(ErsteSramZelle) ; Setze das MSB der Adresse in Register XH
  ldi XL,Low(ErsteSramZelle) ; Setze das LSB der Adresse in Register XL
```

X zeigt nun als Zeiger auf diese Adresse. Um 0xAA an diese Adresse zu schreiben, fügen wir folgende Instruktionen hinzu:

```
  ldi R16,0xAA ; Schreibe AA in Register R16
  st X,R16 ; und in die erste SRAM-Zelle
```

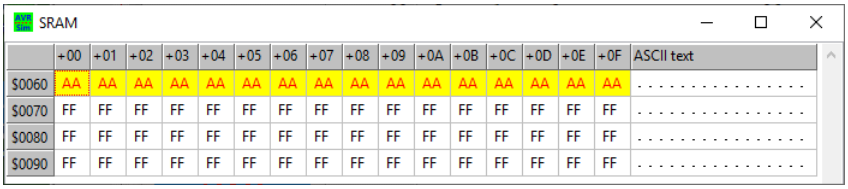
Selbiges könnten wir in gleicher Weise auch mit Y=YH:YL und Z=ZH:ZL veranstalten.

Das ist nun noch nicht so sehr fortgeschritten, da wir hier mit vier Instruktionen nur das machen, was wir mit zwei Instruktionen schon oben mit festen Adressen gemacht haben. Aber gemach, Zeiger können noch viel mehr.

16.1.3 Adressierung von SRAM-Zellen mit erhöhendem Zeiger

Nun könnten wir die ersten 16 Bytes des SRAMs mit 0xAA befüllen, indem wir die Zeigeradresse immerzu um Eins erhöhen:

```
.dseg
ErsteSramZelle: ; Platzieren eines Labels an diese Adresse
.cseg
  ldi XH,High(ErsteSramZelle) ; Setze das MSB der Adresse
  ldi XL,Low(ErsteSramZelle) ; Setze das LSB der Adresse
  ldi R16,0xAA ; Schreibe AA in das Register R16
Fuellschleife:
  st X+,R16 ; und in die SRAM-Zelle und erhoehe dann die Zeigeradresse
  cpi XL,Low(ErsteSramZelle+16) ; Pruefe ob das Ende des Bereichs erreicht ist
  brne Fuellschleife ; Nein, befuelle weiterhin
```



Dies hier zeigt die Bedingungen beim Start:

- Das Zeigerregister X (in R27:R26) zeigt auf die erste Zelle im SRAM. MSB und LSB sind entsprechend so eingestellt.
- Das Register R16 ist auf 0xAA eingestellt.

Der erste Schritt, ein Durchlauf durch die Schleife, ist gemacht. Das **ST** hat den Inhalt von R16 in die erste Zelle geschrieben. Das Plus-Zeichen hinter X hat bewirkt, dass der Zeiger X danach um Eins erhöht wurde. Das Plus bewirkt, dass aus eigentlich zwei Instruktionen

1. **ST X,R16**, und
2. **ADIW XL,1**,

eine einzige wird. Das Plus nennt sich auch Auto-Inkrement. Anstelle der vier Takte, die diese beiden Instruktionen benötigen würden, braucht **ST X+** aber nur zwei: plus oder nicht plus braucht dieselben Takte.

Im letzten Schritt wurde geprüft, ob der LSB des Zeigers schon auf jenseits des Füllbereichs zeigt. Die letzte Zelle des Füllbereichs ist dann überschritten, wenn das LSB des Zeigers auf **LetzteSramZelle-Plus1:** zeigt. Dann wären wir fertig. Weil wir nur das untere Byte überprüfen, können wir damit nicht mehr als 256 Speicherzellen vollschreiben. Wer mehr als 256 Bytes schreiben will, muss auch das MSB überprüfen, wie hier:

```
.dseg
ErsteSramZelle: ; Platzieren Label
  .byte 512 ; Definiere die Laenge des Bereichs
LetzeSramZellePlus1:
;
.cseg
  ldi XH,High(ErsteSramZelle) ; Setze das MSB
  ldi XL,Low(ErsteSramZelle) ; Setze das LSB
  ldi R16,0xAA ; Schreibe AA in Register R16
Fuellschleife:
  st X+,R16 ; in die SRAM-Zelle und Zeiger erhoehen
  cpi XH,High(LetzteSramZellePlus1) ; Pruefe MSB
  brne Fuellschleife
  cpi XL,Low(LetzteSramZellePlus1) ; Pruefe LSB
  brne Fuellschleife
```

Nun kann unser Bereich so lange sein wie er will, er kriegt in Gänze seinen Festwert.

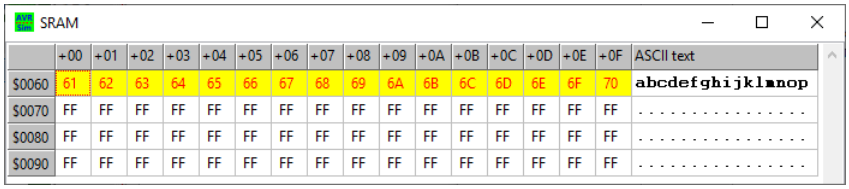
16.1.4 Adressieren von SRAM-Zellen mit Rückwärtszeiger

Automatisches Erhöhen kennen wir jetzt, aber kann man auch von hinten nach vorne adressieren?

Als ein etwas eigenwilliges, aber anschauliches Beispiel dafür kopieren wir einen Text rückwärts, also spiegelverkehrt, von einem SRAM-Bereich in einen anderen.

Zuallererst machen wir einen Text in das SRAM. Das geht so:

```
.dseg
  Textbereich:
  .byte 16
  TextbereichEnde:
.cseg
  ldi XH,High(Textbereich)
  ldi XL,Low(Textbereich)
  ldi R16,'a'
Fuellschleife:
  st X+,R16
  inc R16
  cpi XL,Low(TextbereichEnde)
  brne Fuellschleife
```



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	abcdefghijklmnop
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

Das produziert einen schönen Text. mit **ldi R16,'a'** haben wir den ersten Buchstaben gesetzt und diesen in der Schleife um Eins erhöht (ja, ihr lieben C-Sklaven: das geht, man kann in Assembler auch mit Buchstaben rechnen und den Buchstaben mit **SUBI R16,-1** um Eins erhöhen, kein Problem! In C darfst Du Typsklave das nicht, da darfst Du nicht mit ASCII-Zeichen einfach rechnen!).

Nun soll dieser Text, der auch weniger regelmäßig sein könnte, von hinten nach vorne in einen anderen SRAM-Bereich kopiert werden. Dazu brauchen wir einen weiteren Zeiger, also Y oder Z, und entweder der erste oder der zweite Zeiger muss abwärts gezählt werden, während der andere aufwärts zählt. Wenn du nun mutmaßen solltest, dass Du nur Y- statt Y+ schreiben müsstest, um den zweiten Zeiger Y rückwqärts laufen zu lassen, hast Du das Prinzip zwar verstanden. Aber der Assembler gavrasm quittiert das mit dieser Fehlermeldung:

Nun, aus der Fehlermeldung geht hervor, dass das Minuszeichen vor dem X, Y oder Z stehen muss. Aber

```
Fehler 027: ST-Anweisung kann nur -XYZ+ verwenden, nicht Y-!
  Datei: adressieren.asm, Zeile: 36
  Quellzeile:    st Y-,R16
```

das ist nicht nur Schreibstil, sondern hat auch noch weitere Konsequenzen: das Auto-Dekrement passiert zuerst, danach erfolgt dann erst der Zugriff auf die SRAM-Zelle. Also muss der Y-Zeiger zu Beginn auf die Zelle nach dem Ziel für das erste zu kopierende Zeichen stehen, damit beim vorausgehenden Auto-Dekrement die korrekte Zelle angesteuert wird.

Dies hier ist der gesamte Quellcode des umgekehrten Kopierens.

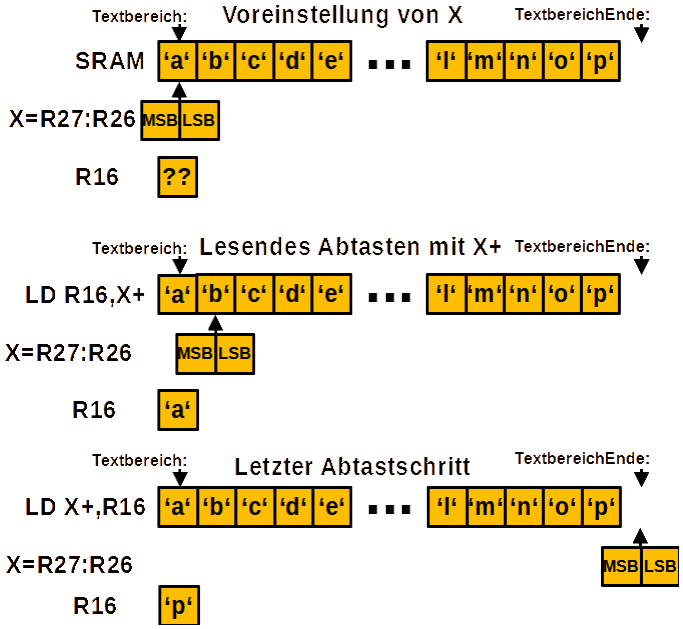
SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	abcdefghijklmnop
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0080	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	ponmlkjihgfedcba
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

```
.dseg
Textbereich:
.byte 16
TextbereichEnde:
.byte 16
Kopierbereich:
.byte 16
KopierbereichEnde:
;
.cseg
  ldi XH,High(Textbereich)
  ldi XL,Low(Textbereich)
  ldi R16,'a'
Fuellschleife:
  st X+,R16
  inc R16
  cpi XL,Low(TextbereichEnde)
  brne Fuellschleife
;
  ldi XH,High(Textbereich)
  ldi XL,Low(Textbereich)
  ldi YH,High(KopierbereichEnde)
  ldi YL,Low(KopierbereichEnde)
Kopierschleife:
  ld R16,X+
  st -Y,R16
  cpi XL,Low(TextbereichEnde)
  brne Kopierschleife
```

Der erste Teil des Programms funktioniert genauso wie das Befüllen mit einer Konstante, nur wird hier R16 bei jedem Schleifendurchlauf um Eins erhöht, so dass wir die Reihe mit 'a' bis 'p' vollkriegen.

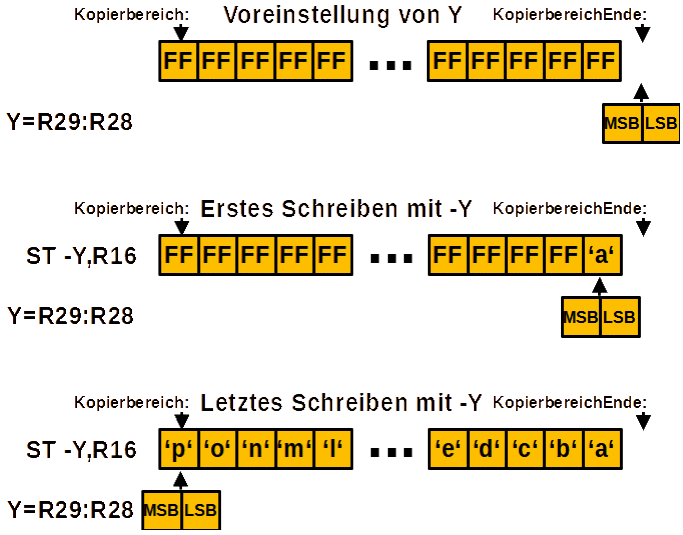
Hier ist der Zeiger X beim Lesen dargestellt. Zu sehen ist der Beginn und darunter der erste Leseschritt mit X+ sowie der letzte Leseschritt.



Die beiden Schritte, Lesen und Schreiben, werden in der Kopierschleife so lange wiederholt, bis entweder

- der X-Zeiger auf die erste Position hinter dem Textbereich zeigt, oder
- der Y-Zeiger auf den Beginn des Kopierbereichs zeigt.

Dies war ein schnelles Umdrehen des Textes, weil alle Adressmanipulationen das Auto-Inkrementieren und -Dekrementieren verwenden und keinerlei langwierige ADIW- oder SBIW-Instruktionen nötig wurden. Das sind so die Vorteile, die ein gut durchdachter Befehlssatz wie der von AVRs so bietet. Aber der kennt noch einen weiteren Befehl.



16.1.5 Zugriff auf SRAM-Zellen mit Verschiebung (displacement)

AVRs haben einen zusätzlichen Adressierungsmodus: er basiert auf der zeitweiligen Addition von konstanten Werten zum Zeigerwert. Zeitweilig deswegen, weil der eigentliche Zeigerwert, anderes als beim Auto-Inkrement, dabei gar nicht verändert wird, nur das Schreiben oder Lesen findet an der temporären Stelle statt.

Diese Art der Adressierung wird auch als "indirekt" bezeichnet. Es geht aber nur mit den Zeigerregistern Y und Z, nicht mit X. Die beiden Instruktionen dafür heißen STD und LDD. Beim Schreiben lautet das Mnemonic **STD Y/Z+d,Register** beim Lesen **LDD Register,Y/Z+d**. **d** ist dabei die Verschiebung, die zur Addresse in Y oder Z temporär addierte Konstante.

In unserer Textreihe im vorherigen Kapitel würde, wenn Z auf den Textbereichs-Beginn zeigen würde, **LDD R16,Z+1** das Zeichen 'b' in das Register R16 schreiben, **LDD R16,Z+2** das Zeichen 'c'. Ein STD

ersetzt daher folgende drei Instruktionen:

```
adiw Z,d ; Addiere Verschiebung zu Z
st Z,R16 ; Schreibe R16 an die verschobene Adresse
sbiw ZL,d ; Subtrahiere Verschiebung von Z
```

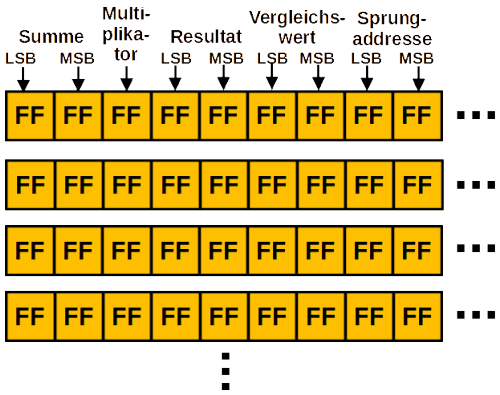
Der Unterschied zwischen STD und dieser Ersatzformulierung ist, dass

- STD keinerlei Flags im Statusregister verändert, und
- STD nur zwei Taktzyklen und nicht sechs benötigt.

STD und **LDD** sind in Fällen nützlich, wenn auf benachbarte Zellen zugegriffen werden soll oder wenn mit dem Zeiger zwischen ganzen Datensätzen gewechselt werden soll. Ein Beispiel dafür.

Nehmen wir an, dein ADC hat acht Kanäle, jeder Kanal braucht einen eigenen Speicherbereich, in dem die Summe der Einzelmessungen gebildet wird, die dann mit einem spezifischen 8-Bit-Multiplikator zu multiplizieren ist, dessen Ergebnis als 16-Bit-Wert benötigt wird und der mit einem kanalspezifischen Unterprogramm mit eigener Adresse weiterverarbeitet werden muss. Normalerweise würde man die Adresse des aktiven Kanals in einen Zeiger schreiben, aber bei jeder Operation, bei der auf einen der Werte im Kanal zugegriffen werden muss, müsste der Zeiger mit ADIW diesem speziellen Wert angepasst werden. Genau das braucht man aber mit STD und LDD nicht: es genügt, die Basisadresse nach Y zu schreiben und mit **LDD R16,Y+3** kann man den Multiplikator des Kanals kriegen. Auch alle anderen Werte der Datenstruktur eines Kanals sind so direkt zugänglich, ohne dass Zeigeranpassungen nötig würden. So kann dann beispielsweise der Messwert zur Summe addiert werden:

```
ld R16,Y ; Lese das LSB der Summe
add R16,R0 ; Addiere das LSB in R0
st Y,R16 ; Speichere das LSB
 ldd R16,Y+1 ; Lese das MSB der Summe mit Verschiebung
adc R16,R1 ; Addiere das MSB und das Carry
std Y+1,R16 ; Speichere das MSB mit Verschiebung
```



So ein Record kann übrigens maximal 64 Bytes groß sein.

Aber die zeitweilige Verschiebung von Zeigern kann auch in anderen Fällen hilfreich sein. Als Beispiel ein Textstring im SRAM wie der abgebildete. Wir wissen schon aus den vorangegangenen Kapiteln, wie man so was in das SRAM geschrieben kriegt.

SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	ABCDEFGHIJKLMNOP
$0070	51	52	53	54	55	56	57	58	59	5A	FF	FF	FF	FF	FF	FF	QRSTUVWXYZ.....
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....

Nun soll an einer beliebigen Stelle, vorzugsweise am Beginn des Strings, ein weiteres Zeichen eingefügt werden. Das erweitert unseren Textstring um ein Zeichen, der bestehende Textstring muss dabei nach rechts kopiert werden. Damit jeweils Platz ist für das vorausgehende Zeichen, muss das wieder von hinten nach vorne passieren, also mit minus X/Y/Z. Natürlich könnten wir dafür zwei Zeiger opfern: einer, der auf den Ziel-Textbereich zeigt, einer der auf den Herkunfts-Textbereich zeigt. Die Differenz zwischen beiden wäre minus 1. Zu viel Aufwand, es geht auch viel einfacher.

SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	@ABCDEFGHIJKLMNO
$0070	50	51	52	53	54	55	56	57	58	59	5A	FF	FF	FF	FF	FF	PQRSTUVWXYZ.....
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....

So soll das fertige Ergebnis aussehen: der String ist nach rechts verschoben und von links wurde ein weiteres Zeichen hereingeschoben. Wir könnten zuerst das letzte Zeichen mit **ld R16,-Y** lesen und müssten es, um eine Stelle nach rechts verschoben, wieder einfügen. Das ginge dann zum Beispiel mit den drei Instruktionen

```
adiw YL,1
st Y,R16
sbiw YL,1
```

Aber diese drei können wir mit dem Verschiebebefehl **std Y+1,R16** ganz einfach ersetzen, denn wir können damit die Verschiebung um eine Stelle höher temporär ausführen und müssen dazu den Zeiger gar nicht erst verlegen.

Kein zweiter Zeiger, das spart zwei Register, keine unnötige Zeigeranpassung, das spart unnötige Takte, einfach nur optimaler Einsatz der eingebauten Adressierungsarten beim AVR. Das geht aber nur mit Y oder Z, nicht mit X.

Wenn der AVR das kann, der PIC oder andere Prozessoren aber nicht, um wieviel schneller sind dann AVRs? Um den Faktor 2 oder 3? Mal eben aus einem zwei oder drei MHz gemacht, und das ohne jeden erhöhten Betriebsstrom.

Schlussfolgerung:

Wenn Dein Programm in Sachen Komplexität weit über eine einfache Blinkroutine hinausgeht, dann denke über solche Optimierungsmethoden nach. Es lohnt sich, denn Du kriegst damit elegantere, schnellere und effektivere Abläufe hin. Und wenn Du den Quellcode auch noch mit aussagekräftigen Kommentaren spickst, sind die dann auch noch schneller lesbar und leichter verständlich als Spaghetticode, gespickt mit so vielen ADIW und SBIW, dass Dir das Hirn schon ganz schummrig wird.

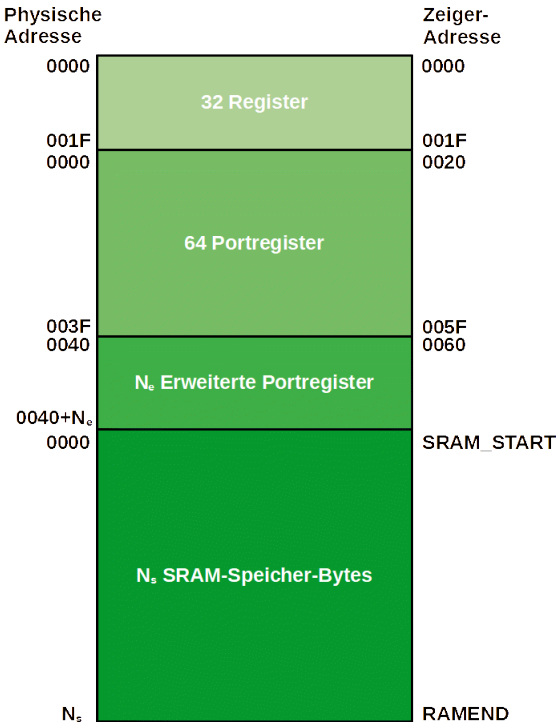
Nur: kennen musst Du die vielen Zugriffsarten dann schon.

16.2 Adressen von Portregistern in AVRs

Mit Portregistern wird der Unterschied zwischen physikalischen Adressen und Zeigeradressen besonders relevant. Beide Adressierungsarten kommen vor, und der Anfänger kann dadurch verwirrt werden.

Zwei Arten von Portregistern sind hier zu unterscheiden:

1. klassische Portregister, die mit den Instruktionen **OUT** und **IN** beschrieben und gelesen werden können, sowie
2. erweiterte Portregister, die nur mit Zeigeroperationen (STS/LDS, ST/LD, STD/LDT) beschrieben und gelesen werden können.



16.2.1 Adressieren von klassischen Portregistern

Diese Zugriffsart verwendet die Instruktionen **OUT Ausgabeport,Register** zum Schreiben und **IN Register,Einbgabeport** zum Lesen. Beide Instruktionen verwenden die physische Adresse des Aus- und Eingabeports.

Der Zugriff mit OUT und IN ist auf die ersten 64 Bytes der physischen Portadressen begrenzt. Erweiterte Portadressen, wie sie neuere und größere AVRs haben, sind damit nicht zugänglich (siehe weiter unten).

Wenn in einem 8 Bit umfassenden Portregister nur eines der acht Bits geändert werden soll, kann man dafür die Instruktionen **SBI Port,Bit** (zum Setzen) und **CBI Port,Bit** verwenden. Diese beiden Instruktionen ersetzen die folgende Befehlsfolge:

```
in R16,port ; Lese den Port
; Setze das Bit
sbr R16,1<<Bit ; Setze das Bit, oder:
ori R16,1<<Bit ;   alternatives Setzen des Bits
; Loesche das Bit
cbr R16,1<<bit ; Loesche (Clear) das Bit, oder:
andi R16,256-1<<bit ; alternatives Loeschen des Bits
out port,R16 ; Schreibe den Port
```

Während diese etwas längere Version bei allen 64 Portregistern funktioniert, ist das kürzere SBI oder CBI nur bei der unteren Hälfte der Portregister zulässig.

SBI und CBI benötigen zwei Taktzyklen, während die allgemeingültige Methode drei Taktzyklen und ein Register benötigt.

Das Torkeln eines Bits in einem Portregister kann mit der Exklusiv-ODER-Instruktion EOR ausgeführt werden, die im ersten Register alle diejenigen Bits umkehrt, die im zweiten Register Eins sind.

```
in R16,port ; Lese den Port
ldi R17,1<<bit ; Das Portbit das torkeln soll, oder
ldi R17,(1<<bit1)|(1<<bit2) ; zwei Bits auf einmal torkeln
eor R16,R17 ; Exklusiv-ODER
out port,R16 ; Schreibe den getorkelten port
```

Wenn der Port, der getorkelt werden soll, ein I/O-Port ist, dann gibt es bei den modereren AVR-Typen die Möglichkeit, durch Schreiben in den Eingabeport PINx die Portpins in PORTx torkeln zu lassen:

```
ldi R16,(1<<1)|(1<<3) ; Torkele Pin 1 und 3
out PINA,R16 ; in PORTA
```

Aber der Zugriff auf die untersten 64 Bytes der Portregister ist mit der Zeigeradresse möglich, man muss nur 0x0020 zur physischen Adresse hinzu addieren:

```
ldi R16,0xAA ; AA in Register R16
sts PORTA+0x20,R16 ; und in PORTA's Zeigeradresse schreiben
```

16.2.2 Adressieren von erweiterten Portregistern

Wenn ein IN- oder OUT-Zugriff auf ein existierendes Portregister mit der Fehlermeldung endet, die Adresse liege außerhalb des zulässigen Adressbereichs, dann handelt es sich bei diesem Portregister um eines

im erweiterten Portbereich oberhalb von 0x003F. Solche erweiterten Portregister-Bereiche gibt es bei vielen neueren AVRs, da die 64 Byte nicht mehr ausgereicht haben, um die interne umfangreiche Hardware zu steuern.

In diesem Fall muss die Zeigeradresse verwendet werden, um mit STS/LDS oder ST/LD auf diese Portregister zuzugreifen. In der def.inc solcher AVR-Typen sind solche Ports direkt mit der Zeigeradresse adressiert und man muss keine 0x0020 hinzuaddieren. Man kann also einfach statt OUT **STS Port,Register** und statt IN **LDS Register,Port** verwenden. Beide Instruktionen benötigen allerdings zwei Taktzyklen.

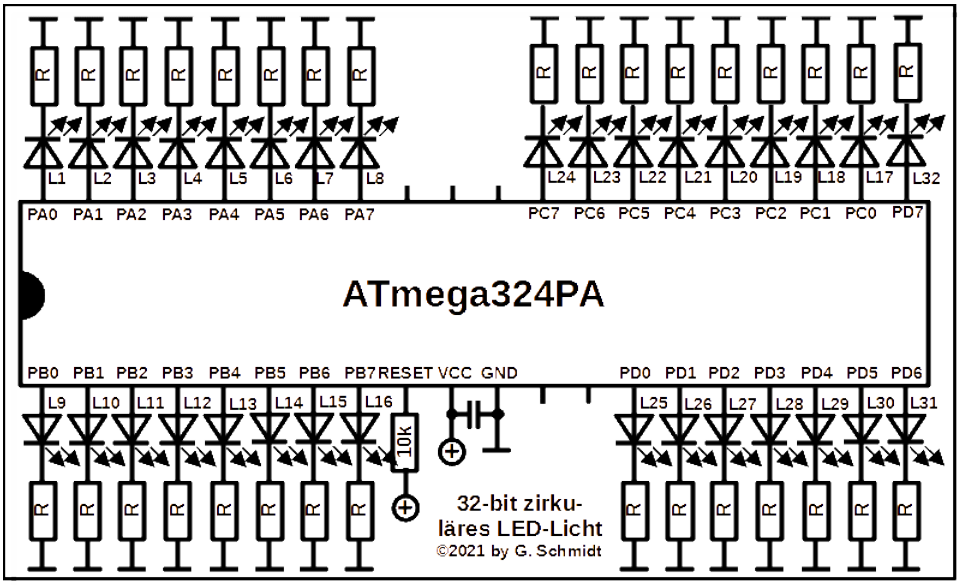
Natürlich sind auch die Instruktionen SBI/CBI und die Verzweigungs- Instruktionen SBIS/SBIC für den erweiterten Portregister-Bereich nicht zulässig und müssen durch andere Instruktionen ersetzt werden.

16.2.3 Adressieren mit Zeigern, Beispiel: zirkuläres LED-Licht

Um die Adressierung mit Zeigern aufzuzeigen, hier das Beispiel eines mit 32 LEDs bestückten Zirkularbeleuchtung, wie sie in Flugzeugen zum Einsatz kommt, um den geneigten Passagieren im Notfall den Weg zum nächsten Notausgang anzuzeigen. Wenn der gesamte Zyklus eine Sekunde dauern soll, müssen die 32 LEDs mit einer Frequenz von 32 Hz umgeschaltet werden, so dass jede LED für 21,25 ms lang an ist. Das erfordert einen AVR mit vier kompletten 8-Bit-I/O-Ports. Dieses Bild zeigt alle AVR-Typen, die dafür in Frage kommen. Wir können jeden aufgelisteten Typ verwenden, z. B. einen ATmega324PA.

```
23 devices out of 478
==> ATmega <==
ATmega16 - PDIP40
ATmega16A - PDIP40
ATmega32 - PDIP40
ATmega32A - PDIP40
ATmega83 - PDIP40
ATmega161 - PDIP40
ATmega162 - PDIP40
ATmega163 - PDIP40
ATmega164A - PDIP40
ATmega164P - PDIP40
ATmega164PA - PDIP40
ATmega323 - PDIP40
ATmega324A - PDIP40
ATmega324P - PDIP40
ATmega324PA - PDIP40
ATmega644 - PDIP40
ATmega644A - PDIP40
ATmega644P - PDIP40
ATmega644PA - PDIP40
ATmega1284 - PDIP40
ATmega1284P - PDIP40
ATmega8515 - PDIP40
ATmega8535 - PDIP40
```

Dies zeigt die nötige Hardware. Sieht ziemlich einfach aus, viele Widerstände und LEDs. Wenn immer gleichzeitig nur eine einzige LED an sein soll und alle anderen Aus, können wir alle Kathoden mit einem einzigen Widerstand verbinden und brauchen nur einen einzigen Widerstand. Wenn immer nur eine LED in einem 8-Bit-I/O-Port an sein soll, reduziert sich die Anzahl Widerstände auf vier.



Der erste Schritt in der Software ist, alle I/O-Richtungsregister als Ausgang zu schalten. Das kann ohne und mit Zeigerzugriff erfolgen. Die Version ohne wäre:

```
ldi R16,0xFF ; Alle Bits als Ausgang
out DDRA,R16
out DDRB,R16
out DDRC,R16
out DDRD,R16
clr R16 ; Alle oberen 24 Bits loeschen
out PORTD,R16
out PORTC,R16
out PORTB,R16
```

```
  ldi R16,0x01 ; Bit 0 setzen
  out PORTA,R16
```

Dies hier sind die Portregister, die die vier I/O-Ports in einem ATmega324PA steuern. PINx, DDRx und PORTx liegen alle schön regelmäßig in einer Reihe. Mit einer Zeiger-Basis-Adresse von 0x0020 in Y oder Z sind die Portregister PINx mit einem Versatz von 0, 3, 6 und 9, die DDRx mit einem Versatz von 1, 4, 7 und 10 und die PORTx mit 2, 5, 8 und 11 für die Instruktion **STD Y+Versatz,R16** zugänglich. Wer STD und die Bedeutung von "Versatz" noch nicht kennt, liest sich das vorhergehende Kapitel noch mal kurz durch.

	Symbol	Physische Adresse	Zeiger-adresse							
	PINA	0x00	0x20	7	6	5	4	3	2	1	0
I/O-Port A	DDRA	0x01	0x21	7	6	5	4	3	2	1	0
	PORTA	0x02	0x22	7	6	5	4	3	2	1	0
	PINB	0x03	0x23	7	6	5	4	3	2	1	0
I/O-Port B	DDRB	0x04	0x24	7	6	5	4	3	2	1	0
	PORTB	0x05	0x25	7	6	5	4	3	2	1	0
	PINC	0x06	0x26	7	6	5	4	3	2	1	0
I/O-Port C	DDRC	0x07	0x27	7	6	5	4	3	2	1	0
	PORTC	0x08	0x28	7	6	5	4	3	2	1	0
	PIND	0x09	0x29	7	6	5	4	3	2	1	0
I/O-Port D	DDRD	0x0A	0x2A	7	6	5	4	3	2	1	0
	PORTD	0x0B	0x2B	7	6	5	4	3	2	1	0

Die Version mit dem Zeiger würde daher folgendermaßen aussehen:

```
  ldi R16,0xFF ; Alle Bits auf Ausgang
  ldi YH,High(PINA+0x20) ; Zeige Y auf PINA's Zeigeradresse
  ldi YL,Low(PINA+0x20)
  std Y+1,R16 ; Setze die Richtungsregister der vier I/O-Ports
  std Y+4,R16
  std Y+7,R16
  std Y+10,R16
  clr R16 ; Loesche die obersten 24 Bits
  std Y+5,R16 ; Zugriff auf die PORT-Portregister
  std Y+8,R16
  std Y+11,R16
  ldi R16,0x01 ; Setze das niedrigste Bit
  std Y+2,R16
```

Gegenüber der direkten Adressierung oben ist dies jetzt weniger effizient, weil jeder STS-Zugriff zwei Taktzyklen braucht. Zum Init würden wir daher die klassische OUT-Methode oben verwenden.

Aber nun ist es an der Zeit, die Interrupt-Service-Routine für den Compare-Match des Timers zu schreiben, der für die Geschwindigkeit des Lichtumlaufs zuständig ist. Die klassische Methode ginge folgendermaßen:

```
Tc0CmpIsr: ; 7 Taktzyklen fuer Interrupt und rjmp
  in R16,PORTA ; +1 = 8
  lsl R16 ; +1 = 9
  out PORTA,R16 ; +1 = 10
  in R16,PORTB ; +1 = 11
  rol R16 ; +1 = 12
  out PORTB,R16 ; +1 = 13
  in R16,PORTC ; +1 = 14
  rol R16 ; +1 = 15
  out PORTC,R16 ; +1 = 16
  in R16,PORTD ; +1 = 17
  rol R16 ; +1 = 18
  out PORTD,R16 ; +1 = 19
  brcc Tc0CmpIsrReti ; +1/2 = 20/21
  in R16,PORTA ; +1 = 21
```

```
  rol R16 ; +1 = 22
  out PORTA,R16 ; +1 = 23
Tc0CmpIsrReti: ; 21/23 Taktzyklen
  reti ; +4 = 25/27 Zyklen
  ; Gesamt-Zyklen: 31 * 25 + 27 = 802 Zyklen
```

Die angegebenen Taktzyklen gelten für jeden Interrupt. 31 Mal wird kein Neustart nötig, ein Mal ist Neustart angesagt. Zusammen mit 32-maligem Aufwachen und 64 Zyklen zum erneuten Schlafenlegen ergeben sich etwa 866 Taktzyklen pro Sekunde. Der Sleep-Anteil läge bei 99,13%.

Aber: drei der vier OUT-Instruktionen sind überflüssig, weil sich die aktive LED gar nicht in diesem I/O-Port befindet. Daher wäre es schon ausreichend, nur den jeweils aktiven I/O-Port zu beschreiben.

In den seltenen Fällen, in denen die LED den I/O-Port wechselt (einer von acht Fällen), muss dann sowohl der neue als auch der vorherige I/O-Port beschrieben werden. Das ergibt einen etwas anderen Algrithmus, dieses Mal sind Zeiger nötig und sinnvoll. Zuerst das Initiieren des Zeigers:

```
  ldi XH,High(PORTA+0x20) ; Zeigeradresse in X, +1 = 1
  ldi XL,Low(PORTA+0x20) ; dto. LSB, +1 = 2
  ldi rShift,0x01 ; Starte Schieberegister, +1 = 3
  st X,rShift ; Schreibe das in den PORTA, +2 = 5
```

Da diese fünf Instruktionen nur ein einziges Mal beim Anfahren ausgeführt werden, zählen sie nicht beim Ermitteln der Taktzyklen. Die Interrupt-Service-Routine ginge dann so:

```
OC0AIsr: ; 7 Zykles fuer Int und rjmp
  lsl rShift ; +1 = 8
  st X,rShift ; +2 = 9
  brcc Tc0CmpIsrReti ; +1/2 = 10/11
  ; Next I/O port
  adiw XL,3 ; Zeige auf nächsten Kanal, +2 = 12
  cpi XL,Low(PORTD+3+0x20) ; +1 = 13
  brne Tc0CmpIsr1 ; +1/2 = 14/15
  ldi XH,High(PORTA+0x20) ; +1 = 16
  ldi XL,Low(PORTA+0x20) ; +1 = 17
Tc0CmpIsr1: ; 15/17 Zyklen
  ; Neustart von Beginn an
  ldi rShift,0x01 ; Set bit 0, +1 = 16/18
  st X,rShift ; +2 = 18/20
Tc0CmpIsrReti: ; 11/18/20
  reti ; +4 = 15/22/24
  ; Gesamtzyklen: = 28*15 + 3*22 + 1*24 = 510
```

Die Anzahl der nötigen Zyklen ist etwa halb so groß wie bei der klassischen Methode ohne Zeiger. Das ist ein klarer Indikator dafür, dass die Zeigermethode doppelt so effizient ist, weil sie keine Zeitverschwendung betreibt und nur Schritte macht, die auch einen Sinn ergeben. Das steigert den Schlafanteil auf 99,43% und reduziert auch etwas den Strom aus der Notstromversorgung des Flugzeugs.

Wenn wir nun die Schaltung etwas ändern und die LED-Kathoden an die I/O-Pins anschließen und dafür die Widerstände an den Anoden an Plus, dann sind im Quellcode ein paar wenige Änderungen nötig. Aus **CLR rShift** würde **SER rShift**, aus **LDI rShift,0x01** würde **LDI rShift,0xFE** und aus **brcc Tc0CmpIsrReti** würde **brcs Tc0CmpIsrReti**. Diese Änderungen betreffen sowohl die Initiierung als auch die ISR. Etwas trickreicher ist der Ersatz von **lsl rShift**: das würde in die beiden Instruktionen, **sec**

und **rol rShift** umgestellt werden. Damit wäre die Software für pull-down-LEDs anstelle von push-up umgestellt.

Die Assembler-Software dafür kann hier heruntergeladen werden. Mit der Software kann die Zyklusdauer zwischen 50 Millisekunden und zwei Sekunden verstellt werden als auch die Polarität der LEDs gewählt werden (die Konstanten im Kopf des Quellcodes machen das).

Was, wenn wir mehr als eine von den 32 LEDs anmachen wollen? Wenn wir davon vier gleichzeitig anmachen wollen und eine davon in jedem 8-Bit-I/O-Port, ist die Software dafür ziemlich einfach und nicht mit Zeigern: einfach **rShift** in jedem Port mit OUT ausgeben.

Wenn Du zwei LEDs auf einmal anmachen willst und diese jeweils in größter Distanz zueinander, also L1 mit L16, L2 mit L17, etc., dann ist es etwas trickreicher, denn der Neustart mit PORTA erfolgt in beiden I/O-Ports jeweils abwechselnd nach 16 Schiebeoperationen.

Wenn jeweils acht der 32 LEDs an sein sollen, sollte man vielleicht über eine Tabelle im Flash nachdenken, in der kann man flexibel alles verewigen und es wie hier gezeigt auslesen.

Wenn noch mehr LEDs gleichzeitig an sein sollen, bekommst Du es vielleicht mit einer Strombeschränkung zu tun: der ATmega324PA kann nur 200 mA über seinen GND- oder VCC-Pin liefern. Daher muss z. B. bei 16 LEDs gleichzeitig der LED-Strom auf 12,5 mA und bei 32 LEDs gleichzeitig auf 6,25 mA begrenzt werden, damit der Prozessor nicht ausflippt.

Schlussfolgerung:

Wenn Dein Programm die immer gleichen Operationen mit verschiedenen I/O-Ports ausführen muss, lohnt es sich über eine Programmierung mittels Zeigern nachzudenken und unnütze Schreiboperationen durch ein intelligentere Programmierung zu ersetzen.

16.3 Adressieren von EEPROM

Alle AVRs haben mindetens 64 Byte und bis zu 4.096 Bytes EEPROM an Bord. EEPROM ist ein Speichertyp, der seinen Inhalt dauerhaft behält, selbst wenn die Betriebsspannung verloren geht. Wenn dann die Betriebsspannung wieder da ist, kann sein Inhalt erneut ausgelesen werden.

Unbeschriebene EEPROM-Speicherbereiche sind alle auf 0xFF eingestellt. Die Inhalte von EEPROM-Speicherzellen können mit zwei Methoden beschrieben werden:

1. Beim Assemblieren von Quellcode werden alle Inhalte, die in das .ESEG-Segment geschrieben werden, in eine Hex-Datei mit der Endung .eep geschrieben. Diese Datei kann mit der Brenner-Software in das EEPROM geschrieben werden.
2. In dem aktiven Programm des Controllers können EEPROM-Zellen gelöscht (auf 0xFF eingestellt) und danach erneut beschrieben werden. Die dazu notwendige Prozedur benötigt etwas Zeit, das Ende des Schreibvorganges kann über einen Interrupt signalisiert werden, wenn dies gewünscht und eingestellt ist.

Das Auslesen von EEPROM-Speicherzellen funktioniert etwas anders. Lesen ist sehr schnell und braucht nur wenige Taktzyklen.

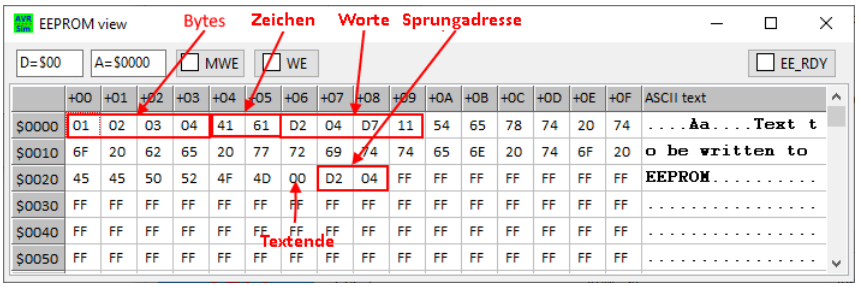
16.3.1 EEPROM-Initierung mit der .ESEG-Direktive

Um das EEPROM als Speicher zu nutzen, sollte es zu Beginn mit einem Inhalt befüllt werden. Das macht die .ESEG-Direktive möglich.

Das Folgende schreibt Inhalte in dieses Segment:

```
.CSEG ; Code-Segment
Sprungadresse:
  ; Instruktionen zum Ausfuehren
```

```
;
.ESEG
.ORG 0x0000 ; Die Startadresse des EEPROM-Inhaltes, Default=0
  .db 1,2,3,4,'A','a' ; Bytes und ASCII-Zeichen in das ESEG
  .dw 1234, 4567 ; 16-Bit-Worte in das ESEG
  .db "Text to be written to EEPROM",0x00 ; Textzeichenkette null-terminiert
  .dw Sprungadresse ; Sprungadresse als Wort in das ESEG
; Ende ESEG
;
.CSEG ; Code-Segment
; ... Weiterer ausführbarer Code
```

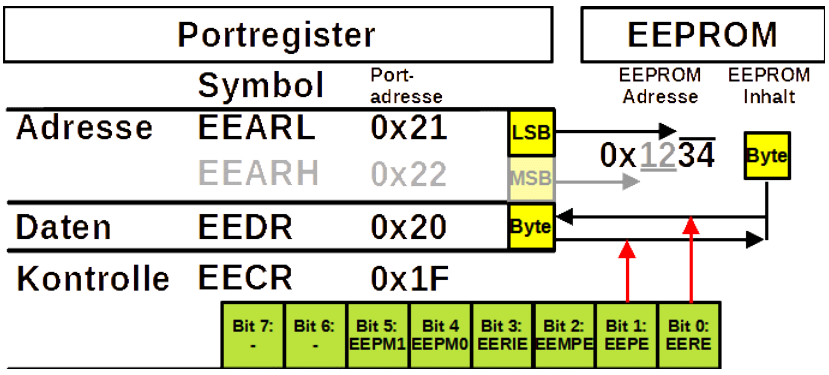


Nach dem Assemblieren und Programmieren des EEPROMs mit der .eep-Datei sieht der Inhalt des EEPROMs dann so aus wie im Bild zu sehen.

Normalerweise wird beim Brennen des Flashspeichers mit ausführbarem Code gleichzeitig auch das EEPROM gelöscht. Soll dies nicht erfolgen und der bisherige Inhalt des EEPROM erhalten bleiben, dann gibt es dafür eine Fuse. Wenn die gesetzt ist, bleibt alles im EEPROM wie es ist. Da einige Speicherbits im EEPROM aber von vorher bereits Nullen enthalten können und weil Nullen im EEPROM nur beim vorherigen generellen Löschen zu Einsen gemacht werden können, würde das erneute Brennen des EEPROMs beim Verifizieren Fehler anzeigen, da die neu programmierten Nullen sich mit den schon vorhandenen Nullen mischen würden und Nullen, die nunmehr Einsen werden sollen, schlicht Nullen bleiben würden.

16.3.2 EEPROM-Portregister

Das Schreiben in oder das Lesen aus dem EEPROM erfolgt über Portregister. Davon gibt es entweder drei oder vier:



1. Das oder die **Adressregister**: Die niedigen 8 Bits der EEPROM-Adresse werden in das Portregister **EEARL** geschrieben. Falls der AVR-Typ mehr als 256 Bytes EEPROM besitzt, wird das MSB der Adresse noch nach **EEARH** geschrieben.
2. Ein **Datenregister**: Beim Schreiben in das EEPROM werden die 8 Bits in das Daten-Portregister **EEDR** geschrieben, bevor der Schreibprozess gestartet wird. Beim Lesen aus dem EEPROM wird das gelesene Byte in diesem Datenregister abgelegt und kann von dort mit IN ausgelesen werden.
3. Ein **Kontrollregister**: Das Schreiben nach und das Lesen aus dem EEPROM wird von einigen

Bits kontrolliert, die im Portregister **EECR** gesetzt oder gelöscht werden können. Beim Schreiben ist das **EEPE**-Bit maßgebend, beim Lesen das **EERE**-Bit.

Bitte beachten, dass die hier angegebenen Portadressen und Bitbenennungen auch anders lauten können. Daher bitte immer die Symbole aus der def.inc als Portadresse und Bitbenennung verwenden.

16.3.3 Einstellung der EEPROM-Adresse

Wenn im Programm EEPROM beschrieben oder gelesen werden soll, muss stets zuerst die Adresse gesetzt werden. Der dazu notwendige Assembler-Quellcode:

```
.equ EepromAdresse = 0x0010
;
; Zuerst Warten bis irgendwelche Schreibvorgaenge zu Ende sind
WarteEep:
  sbic EECR,EEPE ; Pruefe EEPE-Bit
  rjmp WarteEep ; Warte weiterhin
;
; Setzen der Adresse
  ldi R16,Low(EepromAdresse)
  out EEARL,R16
  .ifdef EEARH ; Wenn mehr als 256 Bytes EEPROM vorhanden sind
    ldi R16,High(EepromAdresse)
    out EEARH,R16
    .endif
  ; Hier die Lese- oder Schreibprozedur
  ReadWriteEep:
    ; Die Lese- oder Schreibprozedur hier
```

Bitte beachten, dass vor jedem Schreiben der EEPROM-Adresse zuerst geprüft werden sollte, ob kein Schreibvorgang mehr aktiv ist.

Die .IFDEF-Direktive fügt auch das MSB noch hinzu, aber nur dann, wenn mehr als 256 Bytes EEPROM vorhanden sind.

Die eingestellte Adresse wird beim Lesen und Schreiben nicht geändert. Wenn mehrere Lese- und Schreibvorgänge in einer Reihe ausgeführt werden sollen, muss die Adresse nicht gänzlich neu geschrieben werden. Soll ein gespeichertes Byte nur um Eins erhöht werden, kann auf den Lesevorgang direkt der Schreibvorgang folgen, ohne dass die Adresse neu geschrieben werden muss. Unter Umständen kann auch nur das LSB neu beschrieben werden, wenn sich das MSB nicht ändert.

In dieser Prozedur wird ein EEPROM-Bereich gelesen oder beschrieben und immer die Adresse neu angepasst.

```
.equ EepStartAdr = 0x0010
.equ EepEndAdr = 0x001F
  ; Setze die Adresse in das Registerpaar R1:R0
  ldi R16,High(EepStartAdr)
  mov R1,R16
  ldi R16,Low(EepStartAdr)
  mov R0,R16
EepSchleife:
  sbic EECR,EEPE
  rjmp EepLoop
  ; Ausgabe des LSB der Adresse
  out EEARL,R0
```

```
  .ifdef EEARH
    ; Ausgabe des MSB der Adresse
    out EEARH,R1
    .endif
EepReadWrite:
  ; Fuege Read- oder Write-Prozedur hier ein
  inc R0 ; Erhoehe LSB
  brne EepChkEnd
  inc R1
EepChkEnd:
  ldi R16,Low(EepEndAdr+1)
  cp R0,R16
  brne EepSchleife
  ldi R16,High(EepEndAdr+1)
  brne EepSchleife
```

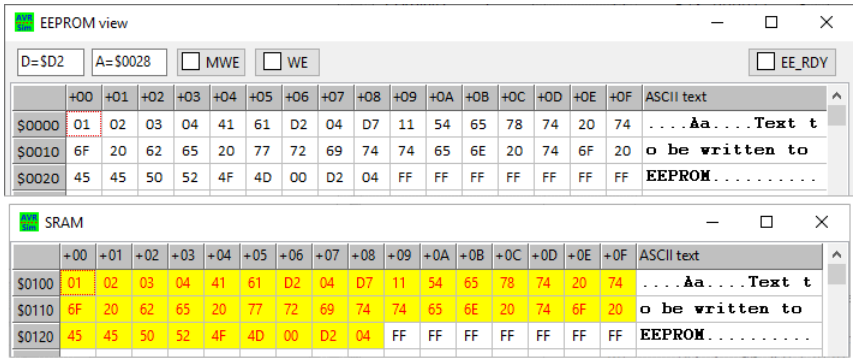
16.3.4 Lesen aus dem EEPROM

Das Auslesen aus dem EEPROM wird gestartet, indem das **EERE**-Bit auf Eins gesetzt wird. Das hält die CPU für vier Takte lang an und liefert dann im Portregister **EEDR** das Ergebnis. Das Auslesen eines Bytes sieht dann so aus:

```
.equ EepAdrs = 0x0010
EepWarte:
  sbic EECR,EEPE ; Warte bis die Schreiboperation beendet ist
  rjmp EepWarte
  ldi R16,Low(EepAddrs)
  out EEARL,R16
  .ifdef EEARH
    ldi R16,High(EepAddrs)
    out EEARH,R16
    .endif
  sbi EECR,1<<EERE
  ; Vier Taktzyklen Pause
  in R16,EEDR ; Lese Byte nach R16
```

In Fällen, in denen mehr als ein einziges Byte zu lesen ist, brauchen wir etwas mehr Speicher. Bei zwei Bytes z. B. ein zweites Register, bei mehr als nur drei oder vier einen Platz im SRAM. In diesem Fall verwenden wir dann auch Zeiger, um z. B. das gelesene Byte mit **ST X+,R16** fortlaufend abzulegen.

Dies hier zeigt, wie wir den kompletten EEPROM-Inhalt in einen SRAM-Bereich ablegen können. Bitte beachten, dass der gesamte Prozess nahezu eine Millisekunde dauert, wegen der Verzögerung beim Lesezugriff, wegen des Checks des Programmierbits, den Doppel-Byte-Check auf die Endadresse und die Zeigeroperationen.



16.3.5 Schreibzugriffe in das EEPROM

Um zu vermeiden, dass es versehentliche Schreibzugriffe auf das EEPROM gibt, ist der Schreibablauf etwas umständlicher gestaltet:

1. Zuerst ist zu prüfen, ob vorherige Schreibzyklen beendet sind. Das erkennt man daran, dass das **EEPE**-Bit im EEPROM-Kontrollregister **EECR** gelöscht ist. Andernfalls muss man halt weiter warten.
2. Dann wird die Schreibadresse in die Portregister **EEARL/EEARH** geschrieben.
3. Danach wird das zu schreibende Datenbyte in das Portregister **EEDR** geschrieben.
4. Dann wird das Master Programming Enable bit **EEMPE** in **EECR** gesetzt, zusammen mit den beiden Programming Mode Bits **EEPM0** (Nur löschen, Erase only) und **EEPM1** (Nur schreiben, Write only), wenn das so gewünscht ist. Das Interrupt Enable Bit **EEPIE** kann ebenfalls gesetzt werden, wenn Interrupts erwünscht sind.
5. Innerhalb der folgenden vier Taktzyklen (Interrupts vorher abschalten) wird das Programming Enable Bit **EEPE** in **EECR** gesetzt.

Das startet das Programmieren an der eingestellten Adresse nach zwei Taktzyklen. Die Interrupts sollten dann wieder eingeschaltet werden.

Das Programmieren dauert 3,4 ms. Wenn diese Zeit abgelaufen ist, setzt der Controller das **EEPE**-Bit auf Null und, falls eingeschaltet, löst den EEP-READY-Interrupt aus und verzweigt zum entsprechenden Vektor.

Ein Beispiel: dieser null-terminierte Text soll aus dem SRAM in das EEPROM kopiert werden. Wir können das entweder interrupt-gesteuert erledigen oder in einer Schleife das **EEPE**-Bit abfragen und dann, wenn es Null ist, das nächste Zeichen in das EEPROM schreiben. Hier mal die Variante mit Interrupts.

SRAM

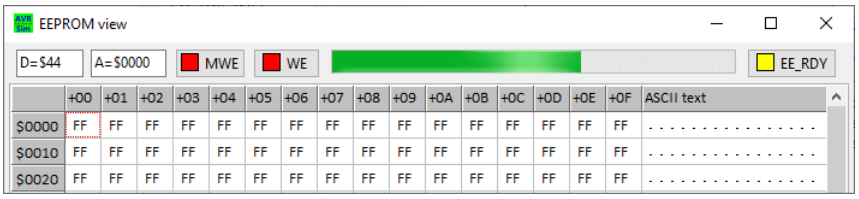
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	44	69	65	73	65	72	20	54	65	78	74	20	73	6F	6C	6C	Dieser Text soll
$0070	20	69	6E	20	64	61	73	20	45	45	50	52	4F	4D	00	FF	in das EEPROM..

Hier zunächst das Initiieren. Es beginnt mit dem Initiieren des Stapelzeigers, der wegen der Interrupts benötigt wird. Zuerst wird dann der Text aus dem Flash in das SRAM kopiert. Dann werden zwei Zeiger eingerichtet: Z zeigt auf die Adresse im EEPROM, X auf den Text im SRAM. Das erste Zeichen wird in das Daten-Portregister geschrieben und das Adress-Portregister wird auf die erste zu beschreibende Adresse gesetzt. Dann wird das erste Zeichen in das EEPROM geschrieben, indem zuerst das Master-Program-Enable-Bit und danach das Program-Enable-Bit Eins gesetzt werden. Der weitere Ablauf erfolgt interruptgesteuert.

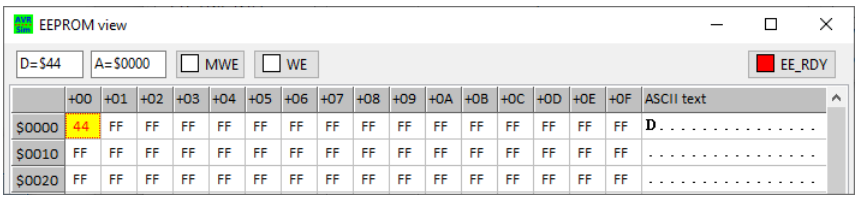
```
.dseg
sText:
;
.cseg
;
; ***********************************
;  H A U P T P R O G R A M M  I N I T
; ***********************************
;
Main:
.ifdef SPH ; Wenn RAMEND groesser als 255
  ldi R16,High(RAMEND) ; Stapelzeiger MSB
  out SPH,R16 ; setzen
  .endif
        ldi R16,Low(RAMEND) ; Stapelzeiger LSB
```

```
        out SPL,R16 ; setzen
; Zeiger Z auf Flash-Tabelle mit Text
  ldi ZH,High(2*Text) ; MSB
  ldi ZL,Low(2*Text) ; LSB
; Zeiger X auf SRAM-Bereich
  ldi XH,High(sText) ; MSB
  ldi XL,Low(sText) ; LSB
Kopie:
  lpm R16,Z+ ; Lese aus Flash
  st X+,R16 ; Schreibe in das SRAM
  tst R16 ; Ende des Textes?
  brne Kopie ; Nein, weiter
;
WarteEep:
  sbic EECR,EEPE ; EEPE-Bit in EECR gleich Null?
  rjmp WarteEep ; Nein, weiter warten
  ldi XH,High(sText) ; X auf SRAM-Bereich, MSB
  ldi XL,Low(sText) ; dto., LSB
  clr ZH ; EEPROM-Adresse, MSB
  clr ZL ; dto., LSB
.ifdef EEARH
  out EEARH,ZH ; EEPROM-Adresse in Portregister, MSB
  .endif
  out EEARL,ZL ; dto., LSB
  ld R16,X ; Lese erstes Zeichen
  out EEDR,R16 ; und schreibe in Daten-Portregister
  ldi R16,(1<<EERIE)|(1<<EEMPE) ; Interrupt Enable und Master Programming Enable
  out EECR,R16 ; in Kontroll-Portregister
  sbi EECR,EEPE ; Setze Progamming Enable Bit
; Interrupts
        sei ; Enable interrupts
;
; ***********************************
;    P R O G R A M   L O O P
; ***********************************
;
Loop:
        rjmp loop
```

Das Schreiben ist mit dem ersten Byte, dem 'D' an die Adresse 0 gestartet, die beiden Schreib-Bits sind aktiv und der Interrupt ist scharf geschaltet. Die erste Phase des Schreibens, das Löschen der EEPROM-Zelle, ist absolviert, nun hat das Schreiben begonnen.

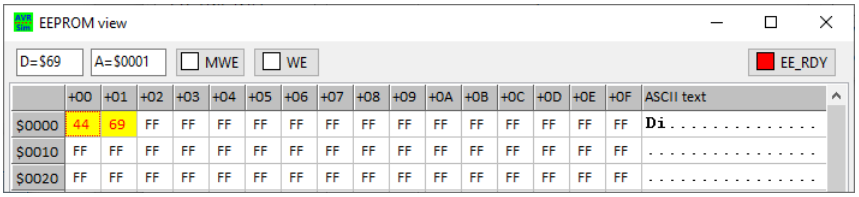


Wenn das zu Ende ist, wird der erste Interrupt ausgelöst. Die erste Speicherzelle hat ihr 'D' und die Interrupt-Service-Routine EERDY wird angesprungen. Die sieht so aus:




```
.cseg
```

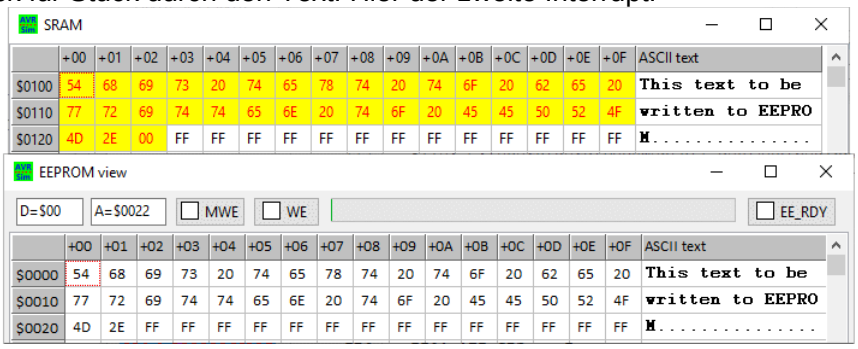
```
.org 000000
;
; ************************************
; R E S E T  &  I N T - V E K T O R E N
; ************************************
        rjmp Main ; Reset vector
        reti ; EXT_INT0
        reti ; PCI0
        reti ; PCI1
        reti ; WATCHDOG
        reti ; ICP1
        reti ; OC1A
        reti ; OC1B
        reti ; OVF1
        reti ; OC0A
        reti ; OC0B
        reti ; OVF0
        reti ; ACI
        reti ; ADCC
        rjmp EerdyIsr ; ERDY
        reti ; USI_STR
        reti ; USI_OVF
;
; *********************************
;  I N T - S E R V I C E   R O U T .
; *********************************
;
EerdyIsr:
  ld R16,X+ ; Lese das letzte geschriebene Zeichen noch einmal
  tst R16 ; War es ASCII-Null?
  brne EerdyIsr1 ; Nein, weiter schreiben
  cbi EECR,EERIE ; Loesche Interrupt-Enable-Bit
  reti ; Fertig
EerdyIsr1:
  ld R16,X ; Lese das naechste Zeichen aus dem SRAM
  out EEDR,R16 ; Schreibe es in das Datenregister
  inc ZL ; Erhoehe LSB der EEPROM Adresse
  out EEARL,ZL ; und schreibe diese in das Adress-Portregister
  brne EerdyIsr2 ; Kein Ueberlauf
  inc ZH ; Erhoehe MSB der EEPROM-Adresse
  out EEARH,ZH ; und schreibe in das Adress-Portregister
EerdyIsr2:
  sbi EECR,EEMPE ; Setze Master Programming Enable-Bit
  sbi EECR,EEPE ; Setze Programming Enable-Bit
  reti ; Fertig
```



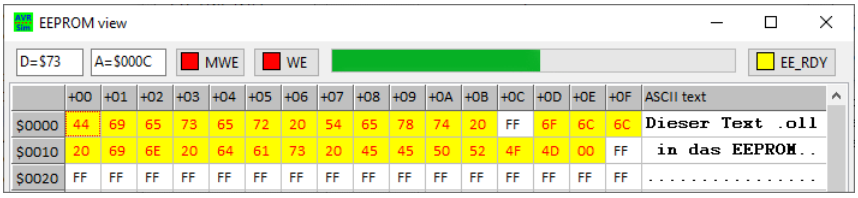
Diese Routine prüft zu Beginn, ob das letzte geschriebene Zeichen eine ASCII-Null war. Wenn das der Fall ist, wird einfach das Interrupt-Enable-Bit gelöscht. Wer da jetzt noch etwas Weiteres tun will, kann es ab hier einfügen. Wenn nicht, wird die Adresse im EEPROM erhöht, das nächste Zeichen gelesen und der Schreibvorgang neu gestartet.

So nudeln sich nun die Interrupts Stück für Stück durch den Text. Hier der zweite Interrupt.

Bis dann auch das letzte Zeichen, die ASCII-Null fertig geschrieben ist und das Programm keine Interrupts mehr produziert. Das Ganze dauerte nun etwa 110 ms.



Hier noch der Abschluss der Löschphase bei einem Zeichen, avr_sim modelliert das Löschen auch zeitlich korrekt.



Ein Hinweis noch zur Interruptprogrammierung: wenn das Interrupt-Enable-Bit **EEIE** gesetzt ist und keine höherwertigen Interrupts anstehen, stößt ein gelöschtes **EEPE**-Bit immer wieder den EEP-READY-Interrupt an. Dadurch kann der komplette Programmablauf mit diesem Interrupt blockiert werden, wenn in der Service-Routine nicht entweder das nächste Schreiben ins EEPROM erfolgt oder, wenn alle Schreibvorgänge erledigt sind, das Interrupt-Enable-Bit gelöscht wird.

Zu guter Letzt noch eine Warnung: die Anzahl Schreiboperationen in EEPROM-Zellen ist nur für einige Tausend mal garantiert. Da ein Tag im Leben eines Prozessors ganze 80.000 Sekunden hat, ein Jahr mehr als 31 Millionen Sekunden, kannst Du das EEPROM leicht totschreiben, wenn Dein Programm häufiger als alle 3.100 Sekunden oder 53 Minuten in das EEPROM schreibt. Daher unbedingt unnötiges Schreiben unterbinden und die Schreibzugriffe auf ein Mindestmaß beschränken. Nur wenn es echt sein muss, und z. B. wenn der Anwender eine Taste gedrückt hat, mal wieder einen Schreibvorgang anstoßen. Es ist daher eine gute Idee, eine Kopie des EEPROMs im SRAM anzulegen und nur bei wesentlichen Änderungen, und nur die tatsächlich geänderten Zellen, neu zu beschreiben.

Ein Beispiel hierfür: wenn Du den aktuellen Status eines Schrittmotors im EEPROM festhalten willst, um beim Neustart des Controllers genau bei diesem Zählerstand zu beginnen, dann wäre es keine gute Idee, das Schreiben bei jedem Einzelschritt anzustoßen. Wir könnten uns darauf beschränken, das Schreiben nur dann auszuführen, wenn der Schrittmotor seine Endstellung erreicht hat und wenn er mindestens zwei oder drei Einzelschritte vollführt hat. Und wenn der Schrittzähler aus vier Bytes besteht: schreibe nur diejenigen Bytes neu, die sich auch tatsächlich geändert haben. Und hoffe halt, dass die LSB-Zelle lange genug durchhält.

Wer ganz unbedingt ganz viele Schreibvorgänge machen muss, muss sich einen Adresswechsel-Wechsel überlegen und programmieren. Der kann z. B. daraus bestehen, dass ein und diesselbe Zelle immer auf korrekte Inhalte geprüft wird. Passieren dabei Fehler wird die Zelle adressmäßig verlegt. Die aktuelle Stelle, zu der die defekte Zelle verlegt wurde, kommt in eine eigene Zelle. Auf diese Weise kann man solange Ersatzzellen nehmen, wie man sie reserviert hat und die Lebensdauer des EEPROMs so verlängern.

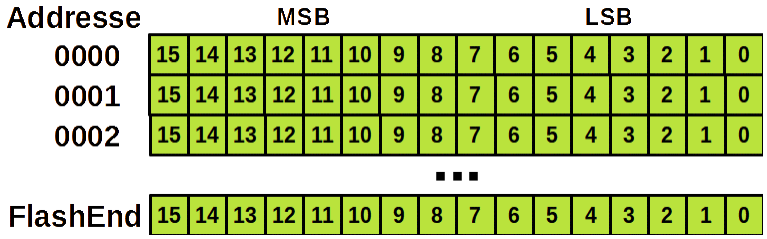
Schlussfolgerung:

Lesezugriffe auf das EEPROM sind sehr schnell, aber langsamer als SRAM-Zugriffe oder Zugriffe auf Register. Daher vorzugsweise das EEPROM in Register oder das SRAM kopieren und dort damit operieren.

Schreibzugriffe auf das EEPROM benötigen längere Zeiten und sind über die Lebensdauer des Prozes-

sors hin limitiert. Es bedarf daher sorgfältiger Überlegung, wann genau was zu schreiben ist. Abhängig von der restlichen Organisation Deines Programmes solltest Du interrupt-getriebenes Schreiben bevorzugen, um zeitliche Konflikte mit anderen Vorgängen zu vermeiden.

16.4 Zugriff auf Flashspeicher



Alle AVRs haben einen Speicher, der Flashspeicher genannt wird und der das ausführbare Programm aufnimmt. Weil der ausführbare Code in AVRs 16-bittig ist, ist dieser Speicher ebenfalls 16 Bits breit (in binären Worten, nicht in Bytes).

Die Größe des Flashspeichers kann Werte zwischen 512 und bis zu 393.216 annehmen. Die Adressen reichen daher von 0x000000 bis 0x0001FF oder 0x05FFFF. In den meisten Fällen gibt die Konstante **FlashEnd** aus der def.inc-Datei die letzte bzw. höchste Adresse an.

Die Adresse 0x000000 ist dabei sehr speziell: sie ist die Startadresse nach jedem Reset, nach der Bereitstellung der Betriebsspannung und beim Watchdog-Reset, hier befindet sich die allererste Instruktion, die ausgeführt wird.

16.4.1 Die .CSEG-Direktive

Beim Assemblieren von Quellcode landet ausführbarer Code als Programmworte sowie Tabellen im Code 16-bittig in einer Hexadezimaldatei namens .hex. Dies ist die Default-Einstellung des Assemblers. Wird mit der Direktive .ESEG (EEPROM-Segment) oder .DSEG (Daten- oder SRAM-Segment) das Ziel des Assemblers umgestellt, kehrt man mit .CSEG wieder in das Code-Segment zurück.

Der Inhalt der .hex-Datei landet mittels Brenner-Software im Flashspeicher des AVR.

Im Code-Segment landen Instruktionen wie **NOP** als 16-Bit-Worte in der .hex-Datei. Zahlen können als hexadezimale Worte mit der Direktive **.DW 16-Bit-Werte** ebenfalls in den Flashspeicher geschrieben werden, als wären sie ausführbarer Code.

Aber auch Bytes, also 8-Bit-Zahlen, können mit der Direktive **.DB Ein-8-Bit-Wert** eingefügt werden, allerdings ist dabei dann das höherwertige Byte Null. Werden mit der Direktive **.DB 8-Bit-Wert1, 8-Bit-Wert2** zwei Byte-Werte gleichzeitig eingefügt, dann landet der 8-Bit-Wert1 im LSB des Programmspeichers, der 8-Bit-Wert2 im MSB. Werden drei oder mehr Byte-Werte in eine .DB-Zeile geschrieben, landen immer zwei davon in einem Wort. Ist die Anzahl Bytes ungeradzahlig, dann ist das MSB des letzten Wortes Null und der Assembler gibt eine Warnung aus.

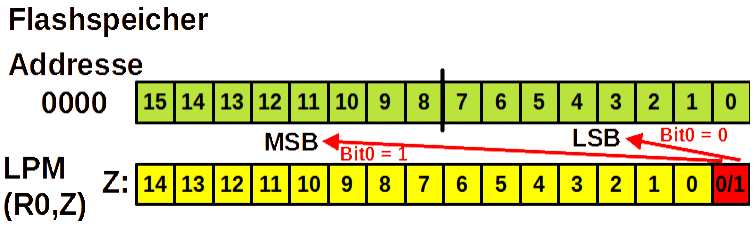
Der Assember-Quellcode links wird assembliert, das Ergebnis kann im erzeugten Listing (nächste Seite) betrachtet werden.

```
E   18   nop ; Eine Instruktion
E   19   add R16,R16 ; Noch eine Instruktion
    20 Schleife: ; Eine Marke (Label)
E   21   rjmp Schleife ; Und noch eine Instruktion
    22 ;
    23 ; Eine Tabelle mit Bytes
    24 .db 1 ; Ein Byte
    25 .db 1,2 ; Zwei Bytes
    26 .db 1,2,3 ; Drei Bytes
    27 .db "Eine Textzeile",0
    28 ;
    29 ; Tabelle mit Worten
    30 .dw 1234 ; Ein Wort
    31 .dw 1234,5678 ; Zwei Worte
    32 .dw 1234,5678,9012 ; Drei Worte
    33 .dw Schleife ; Eine 16-Bit-Adresse
```

- Aus dem **NOP** in Zeile 18 des Quellcodes, welches eine zulässige Instruktion darstellt, hat der Assembler ein 0x0000 an der Adresse 0x000000 gemacht und diese in die Hexdatei .hex geschrieben.
- Aus dem **ADD R16,R16** in Zeile 19 wurde 0x0F00 an Adresse 0x000001.
- Aus der folgenden Zeile wurde gar nichts, die hat der Assembler einfach nur aufgefressen (bzw. er hat sich gemerkt, dass **Schleife:** gleich 0x0002 ist).
- Aus **RJMP Schleife** hat er 0xCFFF an Adresse 0x000002 gemacht, ein ein Sprung zurück, wo er herkam (oder eine unendliche Schleife).
- In der Byte-Tabelle hat er aus **.db 1** 0x0001 an Adresse 0x000003 gemacht, aber sich mit einer Warnung beschwert, dass die Zeile ungerade sei.
- Aus **.db 1,2** hat er 0x0201 an Adresse 0x000004 gemacht und daher die erste Zahl als LSB, die zweite als MSB verwendet.
- Aus **.db 1,2,3** hat er zwei 16-Bit-Worte gemacht: die 1 und 2 als erstes Wort, die 3 als zweites Wort. Hier wieder die schon bekannte Warnung.
- Die Zeile **.db "Eine Textzeile",0** hat er zeichenweise aufgesplittet: In den ersten beiden Worten steckt das Wort "Eine", dann kommt mit 0x20 das Leerzeichen als LSB, zusammen mit dem 'T' als 0x5420, und dann der Rest der Zeile. Da er am Ende eine einsame Null übrig hatte, hat er daraus ein 0x0000 gemacht und eine Beschwerde. Textzeilen mit einer einsamen ASCII-Null am Ende nennt man übrigens null-terminierte Strings.
- Keine Warnungen gibt es in der nächsten Tabelle mit Worten und .dw. Alles passt prima in den Flashspeicher. Der letzte Eintrag hat die Sprungadresse von oben eingefügt, die ja 0x0002 war. Die liegt jetzt an Adresse 0x000015 herum und wartet geduldig auf ihr Gelesen-Werden.

```
  18: 000000   0000  nop ; Eine Instruktion
  19: 000001   0F00  add R16,R16 ; Noch eine Instruktion
  20: Schleife: ; Eine Marke (Label)
  21: 000002   CFFF  rjmp Schleife ; Und noch eine Instruktion
  22: ;
  23: ; Eine Tabelle mit Bytes
  24: .db 1 ; Ein Byte
-> Warnung 004: Anzahl Bytes in der Zeile ist ungerade, 00 hinzugefuegt!
  Datei: beginner_flash-source.asm, Zeile: 24
  Quellzeile: .db 1 ; Ein Byte
      000003 0001
  25: .db 1,2 ; Zwei Bytes
      000004 0201
  26: .db 1,2,3 ; Drei Bytes
-> Warnung 004: Anzahl Bytes in der Zeile ist ungerade, 00 hinzugefuegt!
  Datei: beginner_flash-source.asm, Zeile: 26
  Quellzeile: .db 1,2,3 ; Drei Bytes
      000005 0201 0003
  27: .db "Eine Textzeile",0
-> Warnung 004: Anzahl Bytes in der Zeile ist ungerade, 00 hinzugefuegt!
  Datei: beginner_flash-source.asm, Zeile: 27
  Quellzeile: .db "Eine Textzeile",0
      000007 6945 656E 5420 7865
      00000B 7A74 6965 656C 0000
  28: ;
  29: ; Tabelle mit Worten
  30: .dw 1234 ; Ein Wort
      00000F 04D2
  31: .dw 1234,5678 ; Zwei Worte
      000010 04D2 162E
  32: .dw 1234,5678,9012 ; Drei Worte
      000012 04D2 162E 2334
  33: .dw Schleife ; Eine 16-Bit-Adresse
      000015 0002
```

16.4.2 Die LPM-Instruktion

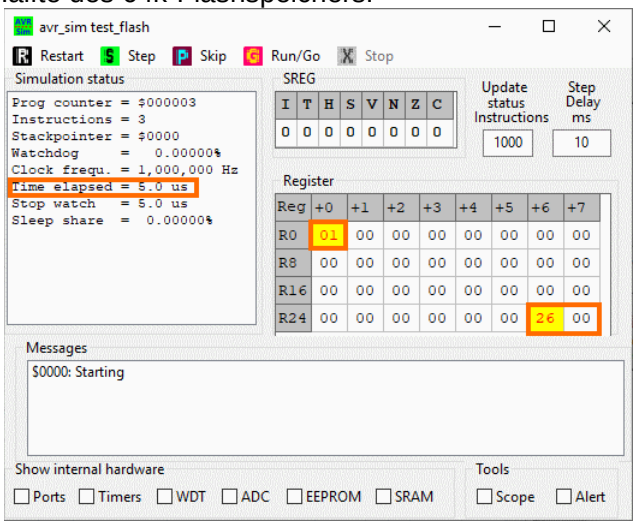


Die Instruktion **LPM** oder "Load from Program Memory" liest ein Byte aus dem Flash aus. Dazu nimmt es die Adresse aus dem Z-Registerpaar (ZH:ZL = R31:R30), liest das Flash und schreibt das Ergebnis in das Register R0.

Aber welche Adresse? Jede Adresse des Flashspeichers zeigt auf zwei Bytes gleichzeitig: ein LSB und ein MSB. Welches der beiden nun gelesen werden soll, muss irgendwie in diese Adresse in Z herein. Der Trick ist, die Adresse des Flashspeichers in Z um Eins nach links zu schieben und von rechts her ein Bit hereinzuschmuggeln. Ist das eingeschmuggelte Bit eine Null, dann wird das LSB gelesen, ist es eine Eins dann ist das MSB dran.

Einen Nachteil hat dieser Trick: Das Bit 15 der Flashadresse kann nicht benutzt werden. Daher müssen Tabellen mit tausenden von Einzelwerten in die untere Hälfte des 64k-Flashspeichers.

Um das Linksschieben der Flash-Speicheradresse zu bewerkstelligen, gibt es drei Möglichkeiten, die alle im folgenden Quellcode demonstriert sind.




```
.equ FlashAddr = ByteTabelle ; Setze die Flashadresse auf die Tabelle
  ; Formulierung 1
  ldi ZH,High(FlashAddr+FlashAddr+0) ; Zugriff aus das LSB, MSB von Z
    ldi  ZL,Low(FlashAddr+FlashAddr+0)  ; dto., LSB von Z
  lpm
  ldi ZH,High(FlashAddr+FlashAddr+1) ; Zugriff auf das MSB, MSB von Z
    ldi  ZL,Low(FlashAddr+FlashAddr+1)  ; dto., LSB von Z
  lpm
  ;
  ; Formulation 2
  ldi ZH,High(2*FlashAddr+0) ; Zugriff auf das LSB, MSB von Z
  ldi ZL,Low(2*FlashAddr+0) ; dto., LSB von Z
  lpm
  ldi ZH,High(2*FlashAddr+1) ; Zugriff auf das MSB, MSB von Z
  ldi ZL,Low(2*FlashAddr+1) ; dto., LSB von Z
  lpm
  ;
  ; Formulation 3
  ldi ZH,High((FlashAddr<<1)|0) ; Zugriff auf das LSB, MSB von Z
  ldi ZL,Low((FlashAddr<<1)|0) ; dto., LSB von Z
  lpm
  ldi ZH,High((FlashAddr<<1)|1) ; Zugriff auf das MSB, MSB von Z
  ldi ZL,Low((FlashAddr<<1)|1) ; dto., LSB von Z
  lpm
Loop:
```

```
  rjmp Loop
;
ByteTabelle:
  .db 1
  .db 1,2
  .db 1,2,3
  .db "Dies ist eine Textzeile"
```

Natürlich muss man nicht unbedingt eine Konstante mit dem Namen **FlashAddr**, man kann auch direkt die Marke **ByteTable:** als Adresse in die LDI-Instruktionen schreiben. Und alle +0 und |0 in den formulierungen sind auch überflüssig, weil sie an dem eigentlichen Wert nichts mehr ändern.

Welche der angebotenen Formulierungen auch immer Du bevorzugst, es kommt immer dasselbe heraus. Wie die Simulation mit avr_sim zeigt, landet das Ergebnis in allen Fällen im Register R0. Die Simulation zeigt auch, dass so ein Zugriff mit LPM drei Taktzyklen braucht, zusammen mit den beiden LDIs fünf Mikrosekunden. Flashzugriffe sind daher etwas langsamer als SRAM-Zugriffe.

16.4.3 Fortgeschrittene LPM-Instruktionen

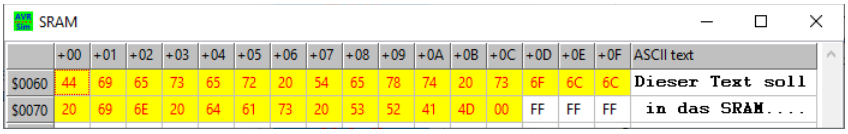
ATMEL hat der LPM-Instruktion später noch die Möglichkeit hingefügt, mehr mit LPM zu machen. Die erste Möglichkeit ist es, der LPM-Instruktion mit **lpm Register,Z** ein anderes Ziel zuzuordnen, eines von den 32 Registern.

Noch ein wenig später kam das Auto-Inkrementieren der Speicheradresse, im Anschluss an den Lesezugriff, hinzu. Das vereinfacht die beiden Instruktionen **lpm** und **adiw ZL,1** zu einer, mit **lpm register,Z+**. Das zusätzliche Inkrementieren verlängert nicht die Ausführungsdauer.

Das Gegenteil, das Auto-Dekrementieren, um Tabellen auch von hinten nach vorne lesen zu können, ist wie beim SRAM auch umgekehrt ausgeführt: zuerst wird dekrementiert, dann erst gelesen. Die Formulierung **lpm Register,-Z** bringt das zum Ausdruck und ersetzt **sbiw ZL,1** und **lpm Register,Z**.

16.4.4 Anwendungsbeispiele für LPM

Das erste Anwendungsbeispiel verwendet LPM um einen null-terminierten Text aus dem Flashspeicher in das SRAM zu kopieren.




```
; Datensegment vorbereiten
.dseg
sText:
;
.cseg
  ; Z auf Flashspeicher-Text
  ldi ZH,High(2*Text)
  ldi ZL,Low(2*Text)
  ; X auf Zieladresse im SRAM
  ldi XH,High(sText)
  ldi XL,Low(sText)
Kopierschleife:
  lpm R16,Z+ ; Load from program memory
  st X+,R16 ; Speichern im SRAM
  tst R16 ; Null-termination?
  brne Kopierschleife ; Nein, weiter machen
```

```
; Nicht in die Tabelle laufen
Loop:
  rjmp Loop
; Text im Flashspeicher
Text:
  .db "Dieser Text soll in das SRAM",0x00
```

Die gesamte Ablauf braucht 235 µs.

Das zweite Anwendungsbeispiel ist ein wenig komplexer und auch ein bisschen akademisch: es nimmt an, dass auf einen Tastendruck hin zehn verschiedene Unterprogramme ausgeführt werden sollen. Einige sind kurz, andere länger, alle sind jedenfalls verschieden lang. Nun kann man sich natürlich mit CPI und BRNE durch die zehn Werte hangeln und das jeweilige Unterprogramm damit herausfinden. Schneller und eleganter ist es aber,

- die zehn Unterprorgamm-Adressen in eine Tabelle zu schreiben,
- die Adresse in der Tabelle mit zwei LPM auszulesen, und
- die Adresse in Z zu schreiben und die Unterroutine mit ICALL auszuführen.

Das bringt Überblick und Flexibilität gleichzeitig und ist nicht mit Verzweigungsorgien gespickt.

Das ist der Quellcode.

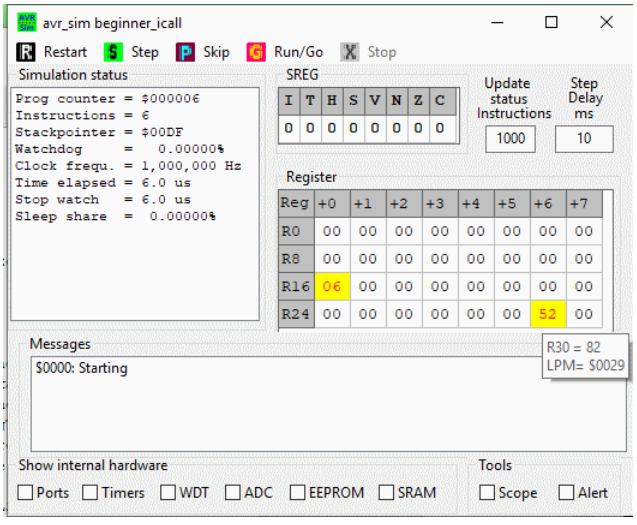
```
.cseg
.equ select = 3 ; Kann zwischen Null und Fuenf sein
  .ifdef SPH
    ldi R16,High(RAMEND) ; Stapelzeiger MSB
    out SPH,R16
    .endif
  ldi R16,Low(RAMEND) ; Stapelzeiger LSB
  out SPL,R16
  ldi R16,select ; Lade Nummer der gewaehlten Routine
  lsl R16 ; Multiplikation mit zwei (Adressen haben 16 Bits)
  ldi ZH,High(2*SprungTabelle) ; Zeige mit Z auf Tabelle, MSB
  ldi ZL,Low(2*SprungTabelle) ; LSB
  add ZL,R16 ; Addiere die Auswahl zum Tabellenanfang
  ldi R16,0 ; Addiere Ueberlauf, wenn nötig
  adc ZH,R16
  lpm R16,Z+ ; Lese LSB
  lpm ZH,Z ; Lese MSB (kein +, wuerde ZH ueberschreiben)
  mov ZL,R16 ; Kopiere LSB nach ZL
  icall ; Rufe Routine in Z auf
Loop:
  rjmp Loop
;
; Routinen
Routine0:
  nop
  ret
Routine1:
  nop
  nop
  ret
Routine2:
  nop
  nop
```

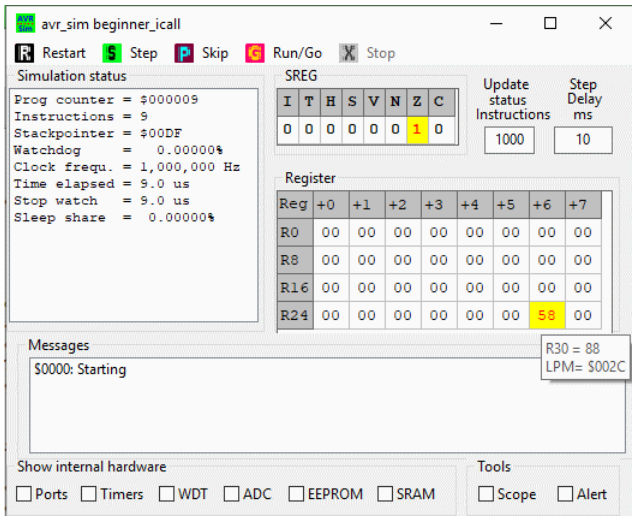
```
  nop
  ret
Routine3:
  nop
  nop
  nop
  nop
  ret
Routine4:
  nop
  nop
  nop
  nop
  nop
  nop
  ret
Routine5:
  nop
  nop
  nop
  nop
  nop
  nop
  nop
  ret
;
; Sprungtabelle
SprungTabelle:
  .dw Routine0,Routine1,Routine2
  .dw Routine3,Routine4,Routine5
  ; Weitere Routinen hier
```

So sieht die Sprungtabelle aus, die der Assembler im Flash angelegt hat. Die erste Sprungadresse ist auf der Adresse 0x000029 zu finden, alle nachfolgenden jeweils Eins höher.

```
71: ; Sprungtabelle
72: SprungTabelle:
73:   .dw Routine0,Routine1,Routine2
    000029 000E 0010 0013
74:   .dw Routine3,Routine4,Routine5
    00002C 0017 001C 0022
75:   ; Weitere Routinen hier
```

Die zwei LDI-Instruktionen haben die Startadresse der Sprungtabelle in Z eingetragen. avr_sim zeigt die Adresse an, auf die Z zeigt.



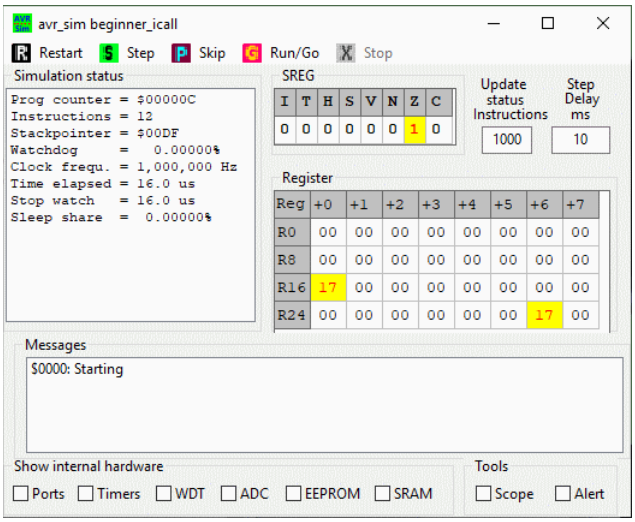


Zu dieser Basisadresse ist nun zwei Mal die 3 aus select addiert, der Z-Zeiger zeigt nun auf 0x002C. in der Tabelle enthält diese Position die Adresse von Routine3.

Mit zwei LPM und einem MOV ist nunmehr die eigentliche Adresse im Z-Register zusammengesetzt worden. Z ist nun fertig für den Sprung.

Das war nun das ICALL, die weitere Ausführung erfolgt in Routine3, wie von select vorherbestimmt.

```
49: Routine3:
50: 000017   0000   nop
51: 000018   0000   nop
52: 000019   0000   nop
53: 00001A   0000   nop
54: 00001B   9508   ret
```



Schlussfolgerung: LPM und seine modernen Varianten bieten weitgefächerte Möglichkeiten, um umfangreiche Texte und Tabellen im großen Flashspeicher anzulegen und zu verwenden. Effektive Programmierung kann bequem auf diese Tools zugreifen.

17 Tabellenanhang

Instruktionen nach Funktion geordnet

Zur Erklärung über Abkürzungen bei Parametern siehe die Liste der Abkürzungen am Ende des Anhangs.

Funktion	Unterfunktion	Instruktion	Flags	Clk
Register setzen	0	CLR r1	Z N V	1
	255	SER rh		1
	Konstante	LDI rh,k255		1
Kopieren	Register => Register	MOV r1,r2		1
	SRAM => Register, direkt	LDS r1,k65535		2
	SRAM => Register	LD r1,rp		2
	SRAM => Register mit INC	LD r1,rp+		2
	DEC, SRAM => Register	LD r1,-rp		2
	SRAM, indiziert => Register	LDD r1,ry+k63		2
	Port => Register	IN r1,p1		1
	Stack => Register	POP r1		2
	Programmspeicher Z => R0	LPM		3
	Register => SRAM, direkt	STS k65535,r1		2
	Register => SRAM	ST rp,r1		2
	Register => SRAM mit INC	ST rp+,r1		2
	DEC, Register => SRAM	ST -rp,r1		2
	Register => SRAM, indiziert	STD ry+k63,r1		2
	Register => Port	OUT p1,r1		1
	Register => Stack	PUSH r1		2
Addition	8 Bit, +1	INC r1	Z N V	1
	8 Bit	ADD r1,r2	Z C N V H	1
	8 Bit+Carry	ADC r1,r2	Z^1 C N V H	1
	16 Bit, Konstante	ADIW rd,k63	Z C N V S	2
Subtraktion	8 Bit, -1	DEC r1	Z N V	1
	8 Bit	SUB r1,r2	Z C N V H	1
	8 Bit, Konstante	SUBI rh,k255	Z C N V H	1
	8 Bit - Carry	SBC r1,r2	Z^1 C N V H	1
	8 Bit - Carry, Konstante	SBCI rh,k255	Z^1 C N V H	1
	16 Bit	SBIW rd,k63	Z C N V S	2
Schieben	Logisch, links	LSL r1	Z C N V	1
	Logisch, rechts	LSR r1	Z C N V	1
	Rotieren, links über Carry	ROL r1	Z C N V	1
	Rotieren, rechts über Carry	ROR r1	Z C N V	1
	Arithmetisch, rechts	ASR r1	Z C N V	1
	Nibbletausch	SWAP r1		1
Binär	Und	AND r1,r2	Z N V	1
	Und, Konstante	ANDI rh,k255	Z N V	1
	Oder	OR r1,r2	Z N V	1

Funktion	Unterfunktion	Instruktion	Flags	Clk
	Oder, Konstante	ORI rh,k255	Z N V	1
	Exklusiv-Oder	EOR r1,r2	Z N V	1
	Einer-Komplement	COM r1	Z C N V	1
	Zweier-Komplement	NEG r1	Z C N V H	1
Bits ändern	Register, Setzen	SBR rh,k255	Z N V	1
	Register, Rücksetzen	CBR rh,255	Z N V	1
	Register, Kopieren nach T-Flag	BST r1,b7	T	1
	Register, Kopie von T-Flag	BLD r1,b7		1
	Port, Setzen	SBI pl,b7		2
	Port, Rücksetzen	CBI pl,b7		2
Statusbit setzen	Zero-Flag	SEZ	Z	1
	Carry Flag	SEC	C	1
	Negativ Flag	SEN	N	1
	Zweierkompliment Überlauf Flag	SEV	V	1
	Halbübertrag Flag	SEH	H	1
	Signed Flag	SES	S	1
	Transfer Flag	SET	T	1
	Interrupt Enable Flag	SEI	I	1
Statusbit rücksetzen	Zero-Flag	CLZ	Z	1
	Carry Flag	CLC	C	1
	Negativ Flag	CLN	N	1
	Zweierkompliment Überlauf Flag	CLV	V	1
	Halbübertrag Flag	CLH	H	1
	Signed Flag	CLS	S	1
	Transfer Flag	CLT	T	1
	Interrupt Enable Flag	CLI	I	1
Vergleiche	Register, Register	CP r1,r2	Z C N V H	1
	Register, Register + Carry	CPC r1,r2	Z C N V H	1
	Register, Konstante	CPI rh,k255	Z C N V H	1
	Register, ≤0	TST r1	Z N V	1
Unbedingte Verzweigung	Relativ	RJMP k4096		2
	Indirekt, Adresse in Z	IJMP		2
	Unterprogramm, relativ	RCALL k4096		3
	Unterprogramm, Adresse in Z	ICALL		3
	Return vom Unterprogramm	RET		4
	Return vom Interrupt	RETI	I	4
Bedingte Verzweigung	Statusbit gesetzt	BRBS b7,k127		1/2
	Statusbit rückgesetzt	BRBC b7,k127		1/2
	Springe bei gleich	BREQ k127		1/2
	Springe bei ungleich	BRNE k127		1/2
	Springe bei Überlauf	BRCS k127		1/2
	Springe bei Carry=0	BRCC k127		1/2

Funktion	Unterfunktion	Instruktion	Flags	Clk
	Springe bei gleich oder größer	BRSH k127		1/2
	Springe bei kleiner	BRLO k127		1/2
	Springe bei negativ	BRMI k127		1/2
	Springe bei positiv	BRPL k127		1/2
	Springe bei größer oder gleich (Vorzeichen)	BRGE k127		1/2
	Springe bei kleiner Null (Vorzeichen)	BRLT k127		1/2
	Springe bei Halbübertrag	BRHS k127		1/2
	Springe bei HalfCarry=0	BRHC k127		1/2
	Springe bei gesetztem T-Bit	BRTS k127		1/2
	Springe bei gelöschtem T-Bit	BRTC k127		1/2
	Springe bei Zweierkomplementüberlauf	BRVS k127		1/2
	Springe bei Zweierkomplement-Flag=0	BRVC k127		1/2
	Springe bei Interrupts eingeschaltet	BRIE k127		1/2
	Springe bei Interrupts ausgeschaltet	BRID k127		1/2
Bedingte Sprünge	Registerbit=0	SBRC r1,b7		1/2/3
	Registerbit=1	SBRS r1,b7		1/2/3
	Portbit=0	SBIC pl,b7		1/2/3
	Portbit=1	SBIS pl,b7		1/2/3
	Vergleiche, Sprung bei gleich	CPSE r1,r2		1/2/3
Andere	No Operation	NOP		1
	Sleep	SLEEP		1
	Watchdog Reset	WDR		1

[1]: Das Z-Flag wird bei diesen Operationen nur dann gesetzt, wenn es schon vorher gesetzt war UND wenn bei der Operation selbst Null herauskam. Das ermöglicht z. B. 16- oder mehr-Bit-Vergleiche.

Instruktionen, alphabetisch

ADC r1,r2
ADD r1,r2
ADIW rd,k63
AND r1,r2
ANDI rh,k255
ASR r1
BLD r1,b7
BRCC k127
BRCS k127
BREQ k127
BRGE k127
BRHC k127
BRHS k127
BRID k127
BRIE k127
BRLO k127
BRLT k127
BRMI k127
BRNE k127
BRPL k127
BRSH k127
BRTC k127
BRTS k127
BRVC k127
BRVS k127
BST r1,b7
CBI pl,b7
CBR rh,255
CLC
CLH
CLI
CLN
CLR r1
CLS
CLT
CLV
CLZ
COM r1
CP r1,r2
CPC r1,r2
CPI rh,k255
CPSE r1,r2
DEC r1
EOR r1,r2
ICALL
IJMP IN r1,p1
INC r1
LD rp,(rp,rp+,-rp)
LDD r1,ry+k63
LDI rh,k255
LDS r1,k65535
LPM
LSL r1
LSR r1
MOV r1,r2
NEG r1
NOP
OR r1,r2 ORI rh,k255 OUT p1,r1
POP r1
PUSH r1
RCALL k4096
RET
RETI
RJMP k4096
ROL r1
ROR r1
SBC r1,r2
SBCI rh,k255
SBI pl,b7
SBIC pl,b7
SBIS pl,b7
SBIW rd,k63
SBR rh,255
SBRC r1,b7
SBRS r1,b7
SEC
SEH
SEI
SEN
SER rh
SES
SET
SEV
SEZ
SLEEP
ST (rp/rp+/-rp),r1
STD ry+k63,r1
STS k65535,r1
SUB r1,r2
SUBI rh,k255
SWAP r1
TST r1
WDR

Ports, alphabetisch

ACSR, Analog Comparator Control & Status Reg.
DDRx, Port x Data Direction Register
EEAR, EEPROM Adress Register
EECR, EEPROM Control Register
EEDR, EEPROM Data Register
GIFR, General Interrupt Flag Register
GIMSK, General Interrupt Mask Register
ICR1L/H, Input Capture Register 1
MCUCR, MCU General Control Register
OCR1A, Output Compare Register 1 A
OCR1B, Output Compare Register 1 B
PINx, Port Input Access
PORTx, Port x Output Register
SPL/SPH, Stackpointer
SPCR, Sreial Peripheral Control Register
SPDR, Serial Peripheral Data Register
SPSR, Serial Peripheral Status Register
SREG, Status Register
TCCR0, Timer/Counter Control Register 0
TCCR1A, Timer/Counter Control Register 1 A
TCCR1B, Timer/Counter Control Register 1 B
TCNT0, Timer/Counter Register, Counter 0
TCNT1, Timer/Counter Register, Counter 1
TIFR, Timer Interrupt Flag Register
TIMSK, Timer Interrupt Mask Register
UBRR, UART Baud Rate Register
UCR, UART Control Register
UDR, UART Data Register
WDTCR, Watchdog Timer Control Register

Assemblerdirektiven

.BYTE x	: reserviert x Bytes im Datensegment (siehe auch .DSEG)
.CSEG	: compiliert in das Code-Segment
.DB x,y,z	: Byte(s), Zeichen oder Zeichenketten einfügen (in .CSEG, .ESEG)
.DEF x=y	: dem Symbol x ein Register y zuweisen
.DEVICE x	: die Syntax-Prüfung für den AVR-Typ x durchführen (in Headerdatei enthalten)
.DSEG	: Datensegment, nur Marken und .BYTE zulässig
.DW x,y,z	: Datenworte einfügen (.CSEG, .ESEG)
.ELIF x	: .ELSE mit zusätzlicher Bedingung x
.ELSE	: Alternativcode, wenn .IF nicht zutreffend war
.ENDIF	: schließt .IF bzw. .ELSE ab
.EQU x=y	: dem Symbol x einen festen Wert y zuweisen
.ERROR x	: erzwungener Fehler mit Fehlertext x
.ESEG	: compiliert in das EEPROM-Segment
.EXIT	: Beendet die Compilation
.IF x	: compiliert den folgenden Code, wenn Bedingung x erfüllt ist
.IFDEF x	: compiliert den Code, wenn Variable x definiert ist
.IFNDEF x	: compiliert den Code, wenn Variable x undefiniert ist
.INCLUDE x	: fügt Datei "Name/Pfad" x in den Quellcode ein
.MESSAGE x	: gibt die Meldung x aus
.LIST	: Schaltet die Ausgabe der List-Datei ein
.LISTMAC	: Schaltet die vollständige Ausgabe von Makrocode ein
.MACRO x	: Definition des Makros mit dem Namen x
.ENDMACRO	: Beendet die Makrodefinition (siehe auch .ENDM)
.ENDM	: Beendet die Makrodefinition (siehe auch .ENMACRO)
.NOLIST	: Schaltet die Ausgabe der List-Datei aus
.ORG x	: Setzt den CSEG-/ESEG-/DSEG-Zähler auf den Wert x
.SET x=y	: Dem Symbol x wird ein variabler Wert y zugewiesen

Verwendete Abkürzungen

Die in diesen Listen verwendeten Abkürzungen geben den zulässigen Wertebereich mit an. Bei Doppelregistern ist das niederwertige Byte-Register angegeben. Konstanten bei der Angabe von Sprungzielen werden dabei automatisch vom Assembler aus den Labels errechnet.

Kategorie	Abk.	Bedeutung	Wertebereich
Register	r1	Allgemeines Quell- und Zielregister	R0..R31
	r2	Allgemeines Quellregister	
	rh	Oberes Register	R16..R31
	rd	Doppelregister	R24(R25), R26(R27), R28(R29), R30(R31)
	rp	Pointerregister	X=R26(R27), Y=R28(R29), Z=R30(R31)
	ry	Pointerregister mit Ablage	Y=R28(R29), Z=R30(R31)
Konstante	k63	Pointer-Konstante	0..63
	k127	Bedingte Sprungdistanz	-64..+63
	k255	8-Bit-Konstante	0..255
	k4096	Relative Sprungdistanz	-2048..+2047
	k65535	16-Bit-Adresse	0..65535
Bit	b7	Bitposition	0..7
Port	p1	Beliebiger Port	0..63
	pl	Unterer Port	0..31