

**Anfängerkurs zum Erlernen der
Assemblersprache von
ATMEL-AVR-Mikroprozessoren**

Gerhard Schmidt
<http://www.avr-asm-tutorial.net>
Dezember 2010

Historie:

Korrektur BCD-Addition Dezember 2010

Ergänzte Fassung vom September 2010

Vollständig überarbeitete Neufassung Dezember 2008 / Januar 2009

Originalversion vom September 2003

Inhaltsverzeichnis

1 Warum Assembler lernen?.....	1
2 Das Konzept hinter der Sprache Assembler.....	3
2.1 Die Hardware von Mikrocontrollern.....	3
2.2 Die Arbeitsweise der CPU.....	3
2.3 Instruktionen in Assembler.....	4
2.4 Unterschied zu Hochsprachen.....	5
2.5 Assembler und Maschinensprache.....	5
2.6 Interpreterkonzept und Assembler.....	6
2.7 Hochsprachenkonzepte und Assembler.....	6
2.8 Was ist nun genau warum einfacher?.....	7
3 Hardware für die AVR-Assembler-Programmierung.....	8
3.1 Das ISP-Interface der AVR-Prozessoren.....	8
3.2 Programmierer für den PC-Parallel-Port.....	9
3.3 Experimentierschaltung mit ATtiny13.....	9
3.4 Experimentalschaltung mit AT90S2313/ATtiny2313.....	10
3.5 Fertige Programmierboards für die AVR-Familie.....	12
3.5.1 STK500.....	12
3.5.2 AVR Dragon.....	12
3.5.3 Andere Boards.....	12
3.6 Wie plane ich ein Projekt in Assembler?.....	13
3.6.1 Überlegungen zur Hardware.....	13
3.6.2 Überlegungen zum Timing.....	14
4 Werkzeuge für die AVR-Assembler-Programmierung.....	16
4.1 Der Editor.....	16
4.2 Der Assembler.....	17
4.3 Das Programmieren des Chips.....	19
4.4 Das Simulieren im Studio.....	20
5 Struktur von AVR-Assembler-Programmen.....	26
5.1 Kommentare.....	26
5.2 Angaben im Kopf des Programmes.....	26
5.3 Angaben zum Programmbeginn.....	27
5.4 Beispielgliederung.....	28
5.5 Ein Programm zur Erzeugung strukturierter Quellcode-Dateien.....	30
6 Register.....	31
6.1 Was ist ein Register?.....	31
6.2 Unterschiede der Register.....	32
6.3 Pointer-Register.....	33
6.4 Empfehlungen zur Registerwahl.....	35
7 Ports.....	36
7.1 Was ist ein Port?.....	36

7.2 Port-Organisation.....	36
7.3 Port-Symbole, Include.....	37
7.4 Ports setzen.....	37
7.5 Ports transparenter setzen.....	37
7.6 Ports lesen.....	38
7.7 Auf Portbits reagieren.....	38
7.8 Porteinblendung im Speicherraum.....	39
7.9 Details wichtiger Ports in den AVR.....	39
7.10 Das Statusregister als wichtigster Port.....	40
8 SRAM.....	42
8.1 Verwendung von SRAM in AVR Assembler.....	42
8.2 Was ist SRAM?.....	42
8.3 Wozu kann man SRAM verwenden?.....	42
8.4 Wie verwendet man SRAM?.....	43
8.5 Verwendung von SRAM als Stack.....	45
9 Steuerung des Programmablaufes in AVR Assembler.....	48
9.1 Was passiert beim Reset?.....	48
9.2 Linearer Programmablauf und Verzweigungen.....	49
9.3 Zeitzusammenhänge beim Programmablauf.....	50
9.4 Makros im Programmablauf.....	51
9.5 Unterprogramme.....	52
9.6 Interrupts im Programmablauf.....	54
10 Schleifen, Zeitverzögerungen und Co.....	58
10.1 8-Bit-Schleifen.....	58
10.2 Das Summprogramm.....	60
10.3 16-Bit-Schleife.....	60
10.4 Das Blinkprogramm.....	62
11 Mehr über Interrupts.....	64
11.1 Überlegungen zum Interrupt-Betrieb.....	64
11.2 Die Interrupt-Vektoren-Tabelle.....	65
11.3 Das Interruptkonzept.....	68
11.4 Ablauf eines interruptgesteuerten Programms.....	68
11.5 Interrupts und Ressourcen.....	70
11.6 Quellen von Interrupts.....	73
11.6.1 Arten Hardware-Interrupts.....	74
11.6.2 AVR-Typen mit Ein-Wort-Vektoren.....	76
11.6.3 AVR-Typen mit Zwei-Wort-Vektoren.....	77
12 Rechnen in Assemblersprache.....	83
12.1 Zahlenarten in Assembler.....	83
12.1.1 Positive Ganzzahlen.....	83
12.1.2 Vorzeichenbehaftete Zahlen.....	83

12.1.3 Binary Coded Digit, BCD.....	84
12.1.4 Gepackte BCD-Ziffern.....	84
12.1.5 Zahlen im ASCII-Format.....	85
12.2 Bitmanipulationen.....	85
12.3 Schieben und Rotieren.....	86
12.4 Addition, Subtraktion und Vergleich.....	88
12.5 Umwandlung von Zahlen - generell.....	91
12.6 Multiplikation.....	92
12.7 Division.....	96
12.8 Binäre Hardware-Multiplikation.....	100
13 Zahlenumwandlung in AVR-Assembler.....	108
13.1 Allgemeine Bedingungen der Zahlenumwandlung.....	108
13.2 Von ASCII nach Binär.....	109
13.3 Von BCD zu Binär.....	109
13.4 Binärzahl mit 10 multiplizieren.....	109
13.5 Von binär nach ASCII.....	110
13.6 Von binär nach BCD.....	110
13.7 Von binär nach Hex.....	110
13.8 Von Hex nach Binär.....	110
13.9 Quellcode.....	110
14 Umgang mit Festkommazahlen in AVR Assembler.....	118
14.1 Sinn und Unsinn von Fließkommazahlen.....	118
14.2 Lineare Umrechnungen.....	118
14.3 Beispiel: 8-Bit-AD-Wandler mit Festkommaausgabe.....	119
15 Tabellenanhang.....	123

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.

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 schlankeren Maschinencode leichter entwanzeln, weil jeder einzelne Schritt Sinn macht und zu Aufmerksamkeit zwingt.

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.

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.

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 Diagnosezeilen ergänzt und mal eben in den Chip programmiert werden kann. Vorbei die Zeiten mit EPROM löschen, programmie-

ren, 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.

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.

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 Befehlsatz 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 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

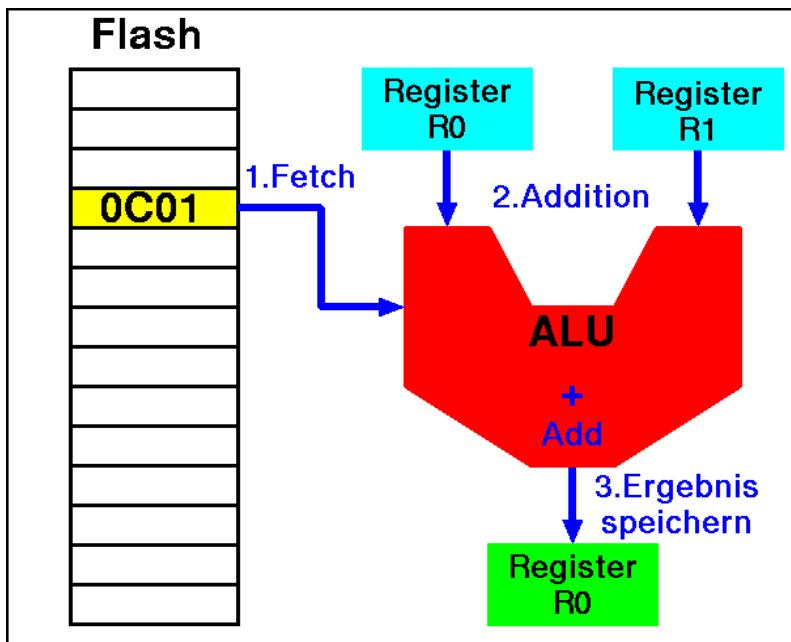
- die zentrale Steuer- und Recheneinheit (CPU) und deren Rechenknecht, die Arithmetische und Logik-Einheit (ALU),
- die diversen Speichereinheiten (interne oder externe RAM, EEPROM-Speicher),
- die Ports, die das Verhalten von Portbits ebenso beeinflussen wie auch Zeitgeber (Timer), AD-Wandler und andere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



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 $256 \cdot 16 = 4.096$ 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 verbrauchen je $32 \cdot 32 = 1.024$ 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 müssen. 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.

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

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 bemerkte. 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 benutzer 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.

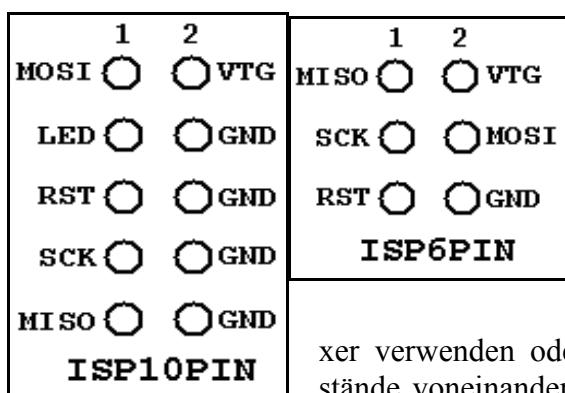
Wer sich mit Löten nicht herumschlagen will und nicht jeden Euro umdrehen muss, kann ein fertiges Programmier-Board erstehen. Die Eigenschaften solcher Boards werden hier ebenfalls beschrieben.

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 Programmierananschluss durch Widerstände voneinander entkoppeln. Was nötig ist, richtet sich nach dem, was die wilden Programmierimpulse mit dem Rest der Schaltung anstellen können.

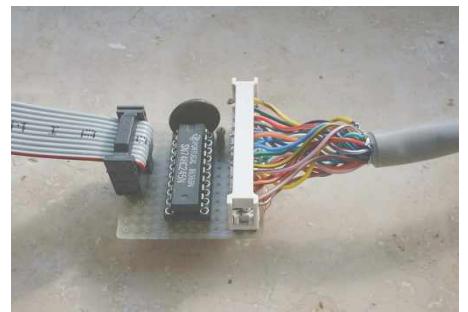
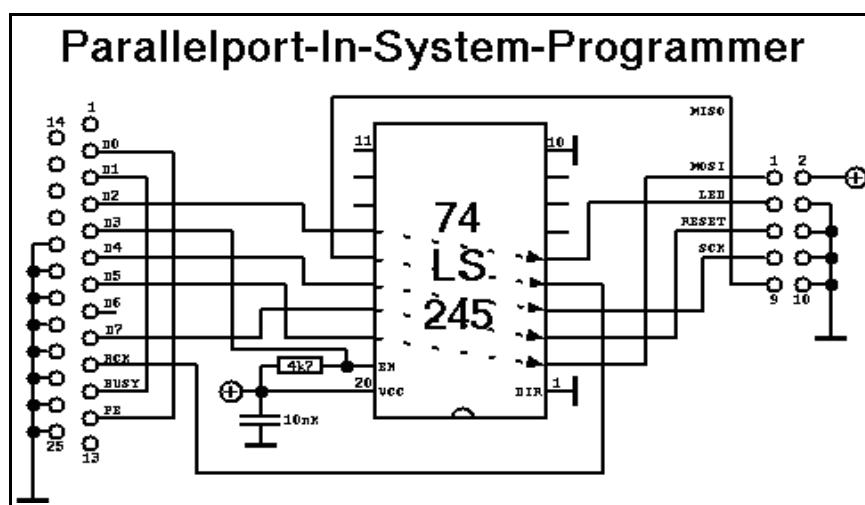
Nicht notwendig, aber bequem ist es, die Versorgungsspannung von Schaltung und Programmier-Interface gemeinsam zu beziehen und dafür zwei weitere Leitungen vorzusehen. GND versteht sich von selbst, VTG bedeutet Voltage Target und ist die Betriebsspannung des Zielsystems. Damit wären wir bei der 6-Draht-ISP-Programmierleitung. 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, unnötig Strom verbrauen und HF erzeugen.

Die Schaltung ist fliegend aufgebaut. Das 10-adige 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:

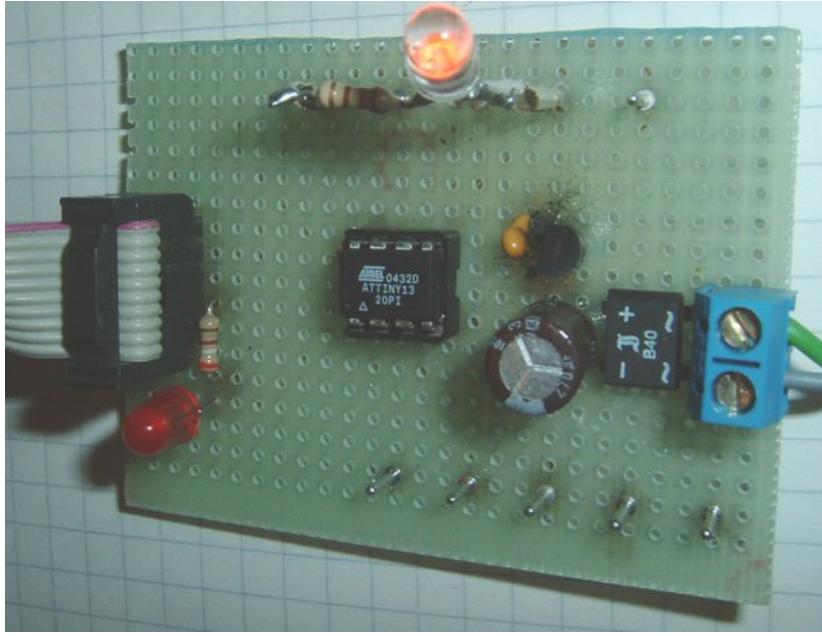
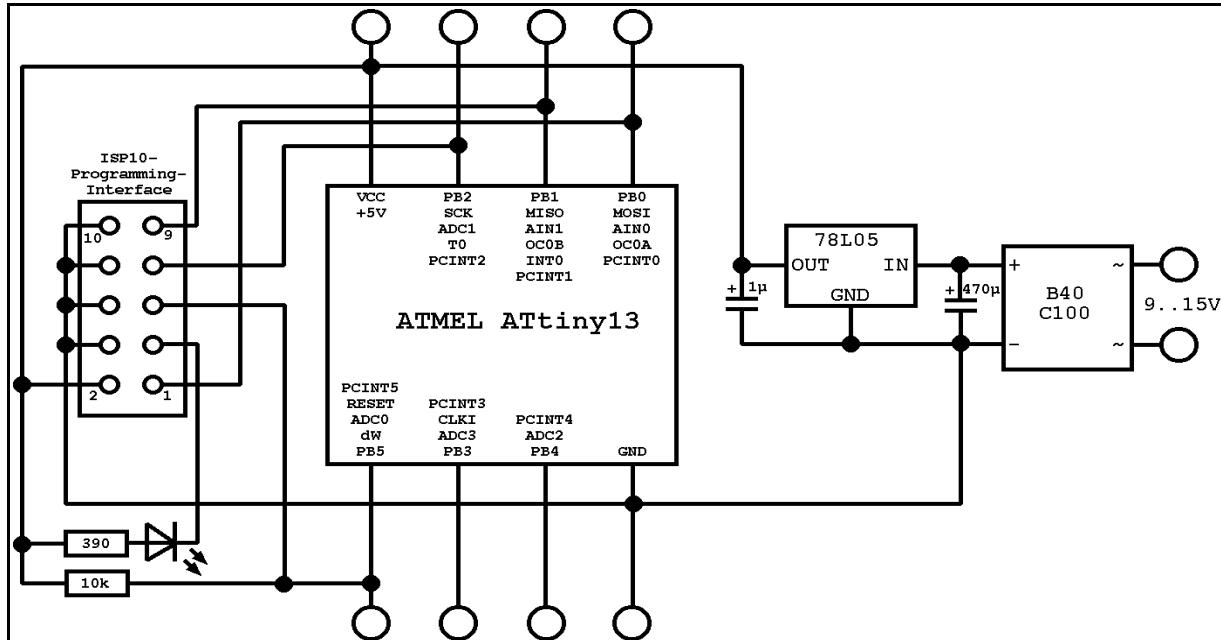
<http://www.klaus-leidinger.de/mp/Mikrocontroller/AVR-Prog/AVR-Programmer.html>

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 $390\ \Omega$ an die Betriebsspannung angeschlossen ist, den ATtiny13, dessen Reset-Eingang an Pin 1 mit einem Widerstand von $10\ k\Omega$ 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.



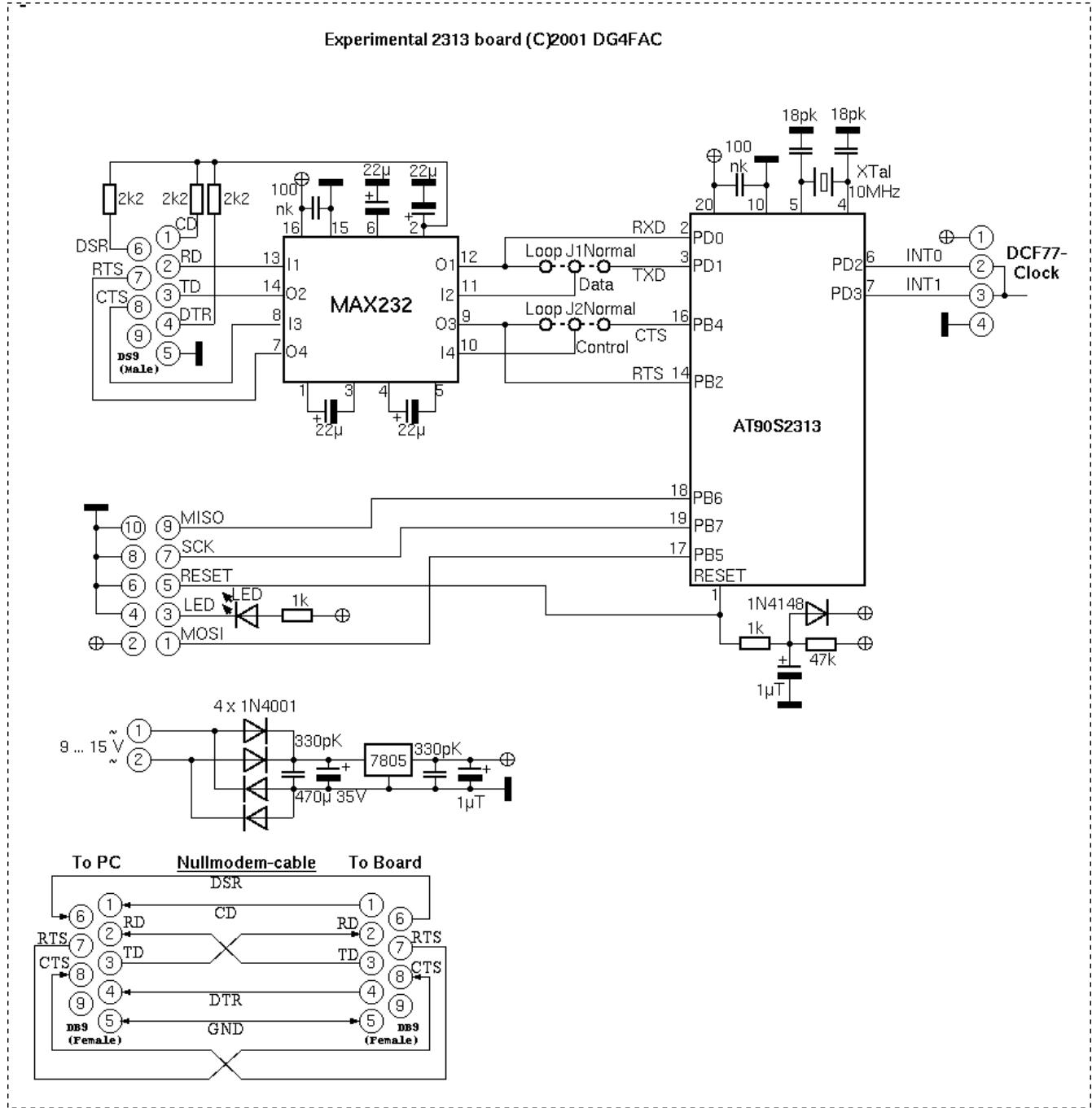
Der ATtiny13 braucht keinen externen Quarz oder Taktgenerator, weil er einen internen 9,6 MHz-RC-Oszillator hat und von der Werksausstattung her mit einem Verteiler von 8 bei 1,2 MHz arbeitet.

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.

3.5.1 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.

3.5.2 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.

3.5.3 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:
 - 2. 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).
 - 3. 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.
 - 4. Soll der Analogkomparator verwendet werden, dann sind AIN0 und AIN1 dafür zu reservieren.
 - 5. Sollen externe Signale auf Flanken überwacht werden, dann sind INT0 bzw. INT1 dafür zu reservieren.
 - 6. 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.
 - 7. 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.
 - 8. Wird externer SRAM-Speicher benötigt, müssen alle nötigen Address- und Datenports sowie ALE, RD und WR dafür reserviert werden.
 - 9. Soll der Takt des Prozessors aus einem externen Oszillatortsignal bezogen werden, ist XTAL1 dafür zu reservieren. Soll ein externer Quarz den Takt bestimmen, sind die Anschlüsse XTAL1 und XTAL2 dafür zu verwenden.
 - 10. 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.
 - 11. Erfordert die Ansteuerung eines externen Gerätes das Schreiben oder Lesen von mehr als einem Bit gleichzeitig, z.B. eine vier- oder achtbitige LCD-Anzeige, sollten die nötigen Portbits in dieser Reihenfolge platziert werden. Ist das z.B. bei achtbittigen Interfaces nicht möglich, können auch zwei vierbitige Interfaces vorgesehen werden. Die Software wird vereinfacht, wenn diese 4-Bit-Interfaces links- bzw. rechtsbündig im Port angeordnet sind.
 - 12. 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.
 - 13. Welche frei platzierbaren Portbits werden noch gebraucht? Jetzt wird alles zugeordnet, was

keinen bestimmten Platz braucht.

14. 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.

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 Prozessor 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 Teilverhältnisse (höhere Taktfrequenz) oder auf einen höheren Wert umgestellt werden (z.B. 128), um durch

niedrigere Taktfrequenz z.B. verstrt 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-Oszillatior zu ungenau ist, kann per Fuse eine externe RC-Kombination, einen externen Oszillatior, einen Quarz oder einen Keramikschwinger auswhlen. Gegen falsch gesetzte Fuses ist kein Kraut gewachsen, es muss dann schon ein Board mit eigenem Oszillatior sein, um doch noch was zu retten.

Die Hhe der Taktfrequenz sollte der Aufgabe angemessen sein. Dazu kann grob die Wiederholfrequenz der Ttigkeiten dienen. Wenn eine Taste z.B. alle 2 ms abgefragt werden soll und nach 20 Mal als entprellt ausgefhrt werden soll, dann ist bei einer Taktfrequenz von 1 MHz fr 2000 Takte zwischen zwei Abfragen Platz, 20.000 Takte fr die wiederholte Ausfhrung des Tastenbefehls. Weit ausreichend fr eine gemliche Abfrage und Ausfhrung.

Eng wird es, wenn eine Pulsweitenmodulation mit hoher Auflung und hoher PWM-Taktfrequenz erreicht werden soll. Bei einer PWM-Taktfrequenz von 10 kHz und 8 Bit Auflung sind 2,56 MHz schon zu langsam fr eine software-gesteuerte Lsung. Wenn das ein Timer mit Interrupt-Steuerung bernimmt, um so besser.

4 Werkzeuge für die AVR-Assembler-Programmierung

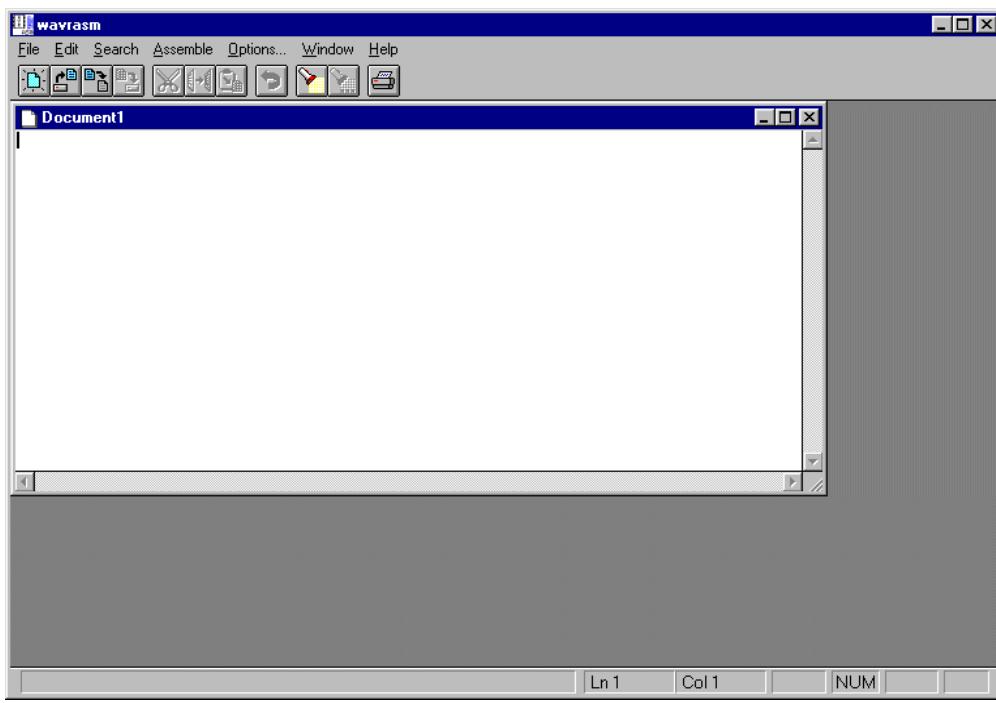
In diesem Abschnitt werden die Werkzeuge vorgestellt, die zum Assembler-Programmieren nötig sind. Dabei werden zunächst Werkzeuge besprochen, die jeden Arbeitsschritt separat erledigen, damit die zugrunde liegenden Schritte einzeln nachvollzogen werden können. Erst danach wird eine integrierte Werkzeugumgebung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



Quellcode.

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-Menü in irgendein Verzeichnis, am besten in ein eigens dazu errichtetes, abgespeichert. Fertig ist der Assembler-

Manche Editoren erkennen Instruktionen und Symbole und färben diese Worte entsprechend ein, so

```

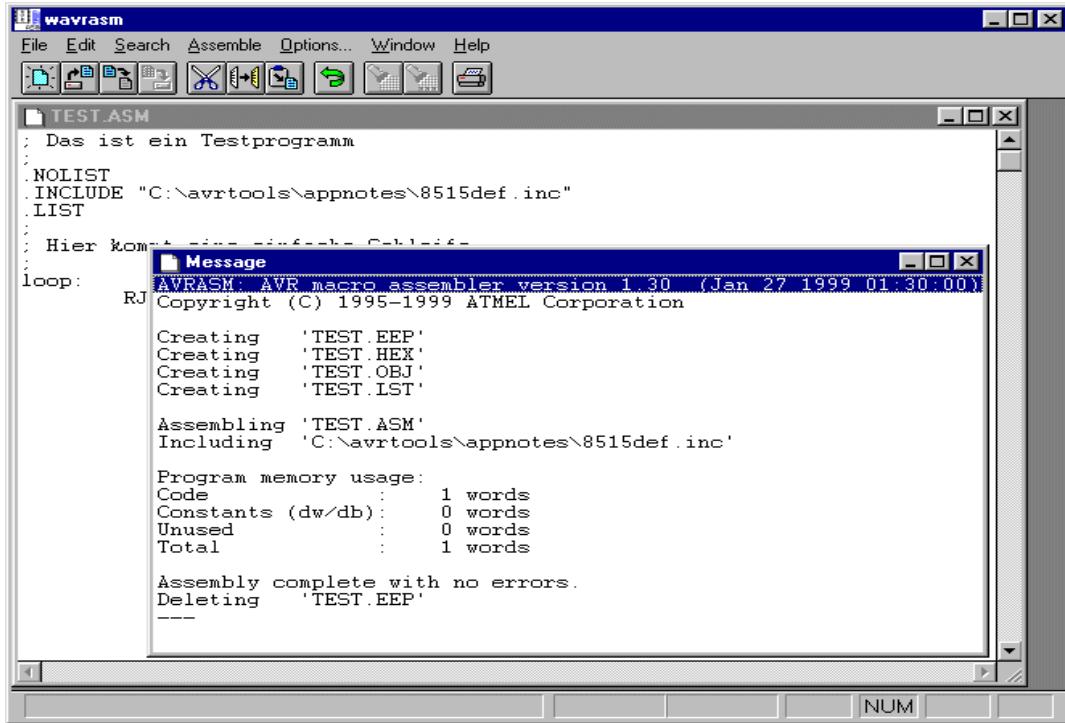
; Testprojekt für ATMEL Studio
;
; INCLUDE "C:\Programme\Atmel\AVR Studio\Appnotes\8515def.inc"
;
.DEF    mp = R16
;
.CSEG
.ORG 0000
Start:
    ldi    mp, 0xFF ; Alle Pins Port B auf Ausgang
    out   DDRB, mp
    ldi    mp, 0xAA ; Startwert 1010.1010
Loop:
    out   PORTB, mp ; an Port B
    com   mp          ; alle bits invertieren
    rjmp  Loop        ; und endlos weitermachen
;
```

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.

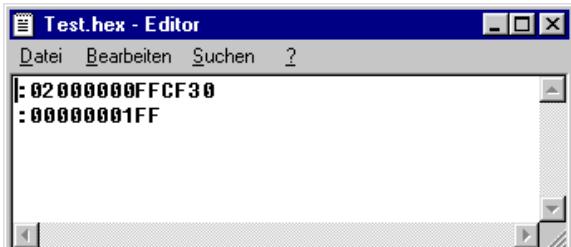
4.2 Der Assembler

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 Menüpunkt 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 würde er mit dem Schlagwort „Error“ um sich.

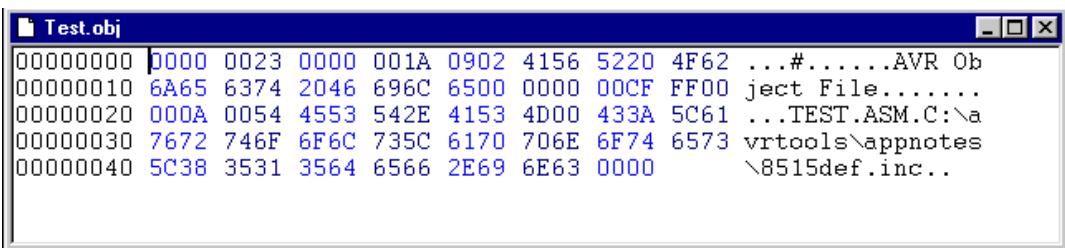
Immerhin ein Wort Code ist dabei erzeugt worden. Und er hat aus unserer einfachen Textdatei gleich vier neue Dateien erzeugt. In der ersten der vier neuen Dateien, TEST.EEP, befindet sich der Inhalt, der in das EEPROM geschrieben werden soll. Er ist hier ziemlich uninteressant, weil wir nichts ins EEPROM 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.

```

Test.ist - Editor
Datei Bearbeiten Suchen ?
AURASM ver. 1.30 TEST.ASM Sun Jun 10 01:46:13 2001

; Das ist ein Testprogramm
;
.NOLIST
;
; Hier kommt eine einfache Schleife
;
loop:
000000 cfff        RJMP loop

Assembly complete with no errors.

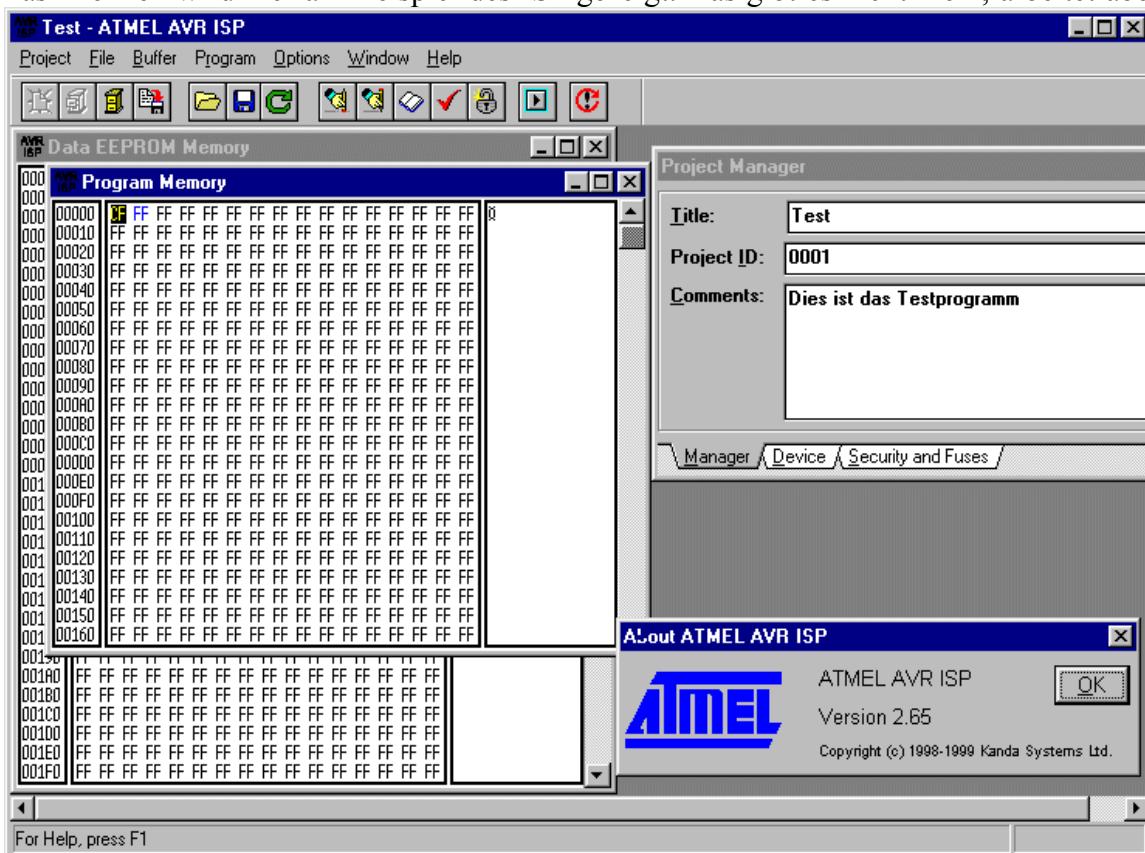
```

Sie enthält das Nebenstehende. Wir sehen das Programm mit allen Adressen (hier: "000000"), Maschineninstruktionen (hier: "cff") 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!

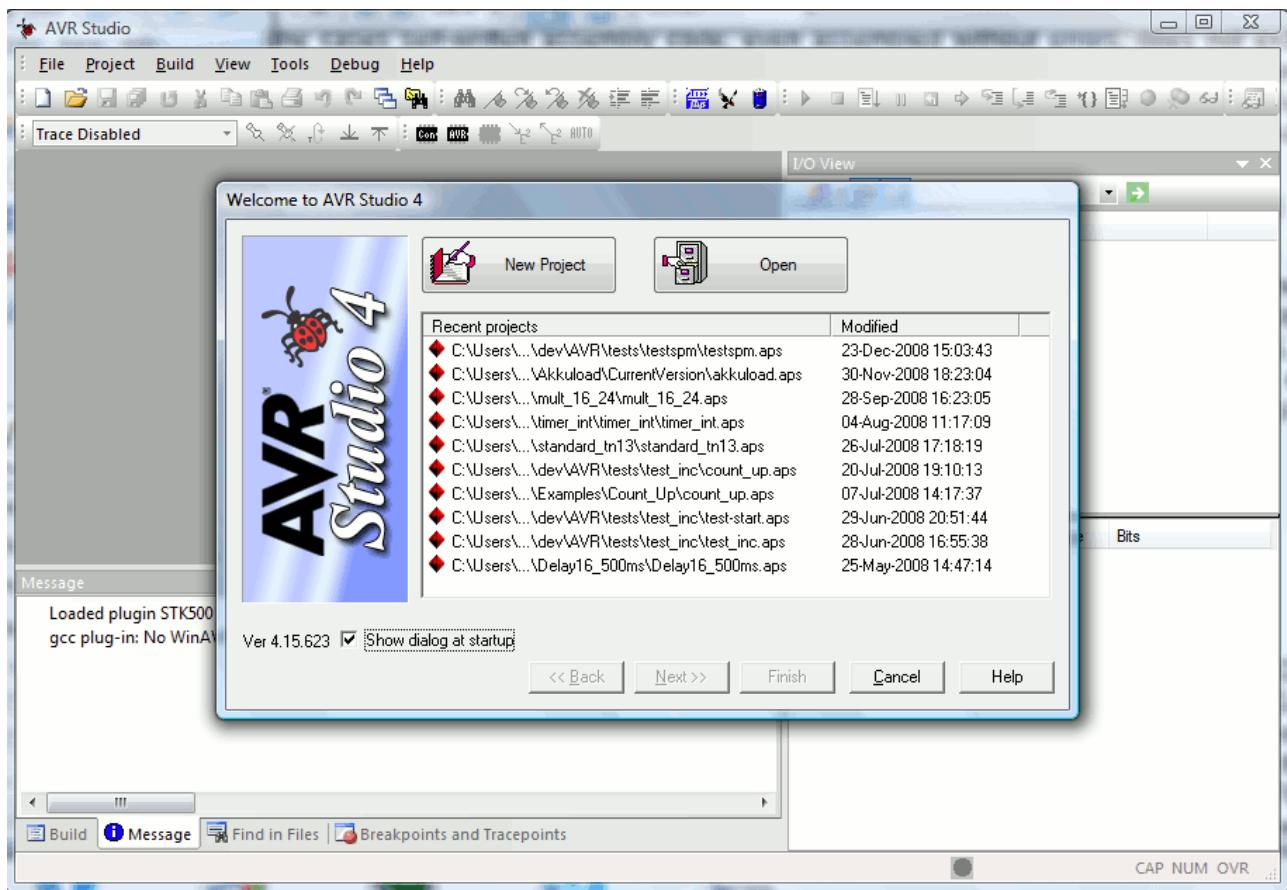
Wir starten das Programm ISP, erzeugen ein neues Projekt und laden die gerade erzeugte Hex-Datei mit LOAD PROGRAM. Das sieht dann wie im Bild aus. Wenn wir nun mit dem Menü „Program“ den Chip programmieren, dann legt dieser gleich los. Beim Brennen gibt eine Reihe von weiteren

Voraussetzungen (richtige Schnittstelle auswählen, Adapter an Schnittstelle angeschlossen, Chip auf dem Programmierboard vorhanden, Stromversorgung auf dem Board eingeschaltet, ...), ohne die das natürlich nicht geht.

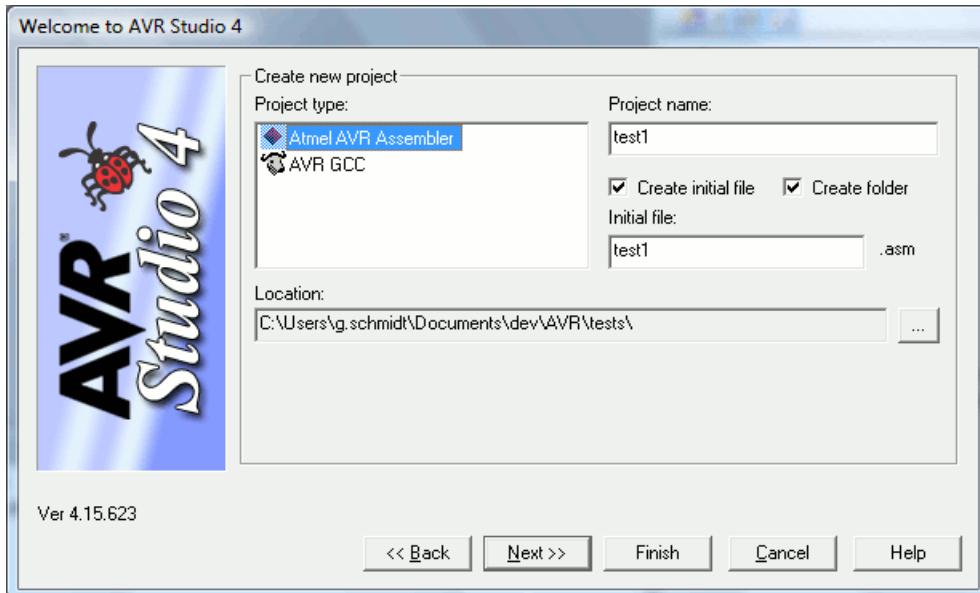
4.4 Das Simulieren im Studio

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 ATMEIL einen Simulator, der für die Entwanzung ideale Möglichkeiten bietet. Das Programm kann Schritt für Schritt abgearbeitet werden, die Zwischenergebnisse in Prozessorregistern sind überwachbar, etc.

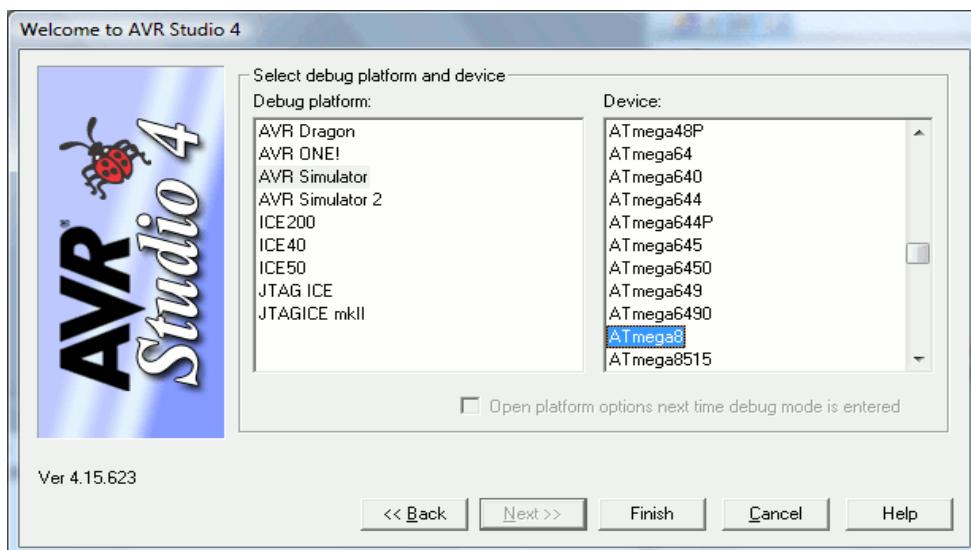
Die folgenden Bilder sind der Version 4 entnommen, die ältere Version 3 sieht anders aus, macht aber in etwa dasselbe. Die Studio Software enthält alles, was man zur Entwicklung, zur Fehlersuche, zur Simulation, zum Brennen der Programme und an Hilfen braucht. Nach der Installation und dem Start des 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 Erstinstallatiion ist "New Project" die korrekte Antwort.



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.

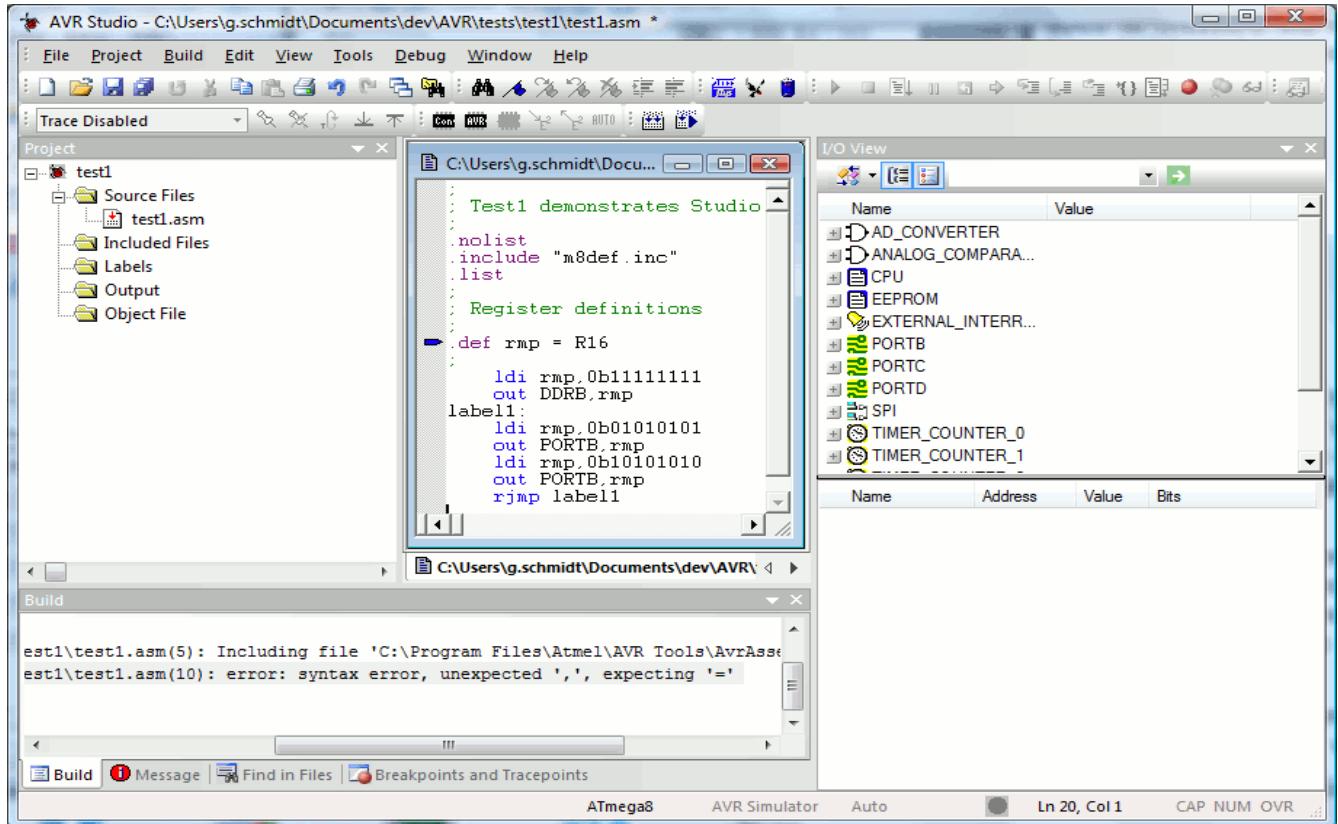


Hier wird die Plattform "Simulator" ausgewählt. Ferner wird hier der Prozessor-Zieltyp (hier: ATmega8) ausgewählt. Der Dialog wird mit "Finish" abgeschlossen.

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"-Fenster, in dem Diagnose-Ausgaben erscheinen, und
- rechts unten ein weiteres Feld, in dem diverse Werte beim Entwerten 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:

The Build window displays the assembly output and memory usage summary:

```

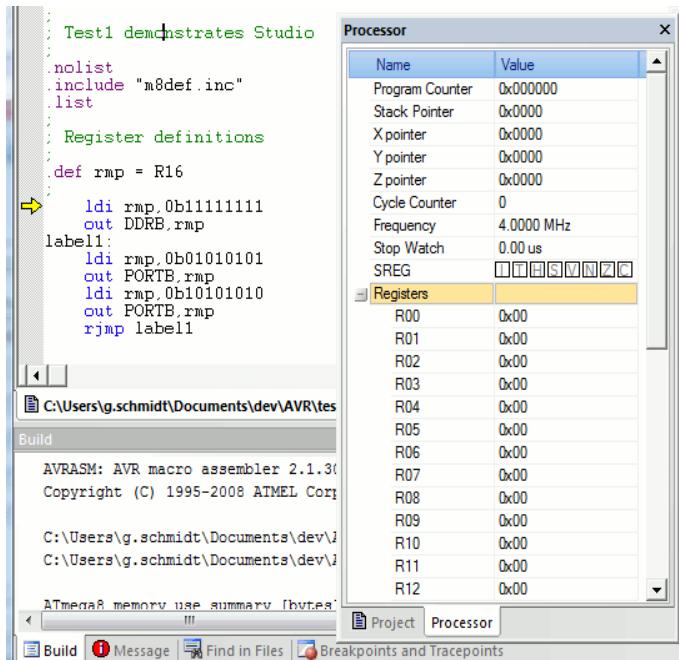
AVRASM: AVR macro assembler 2.1.30 (build 592 Nov 7 2008 12:38:17)
Copyright (C) 1995-2008 ATMEL Corporation

C:\Users\g.schmidt\Documents\dev\AVR\tests\test1\test1.asm(5): Including file 'C:\Program Files\Atmel\AVR Tools\AvrAssembler2\Appnotes\m8def.inc'
C:\Users\g.schmidt\Documents\dev\AVR\tests\test1\test1.asm(20): No EEPROM data, deleting C:\Users\g.schmidt\Documents\dev\AVR\tests\test1\test1.eep

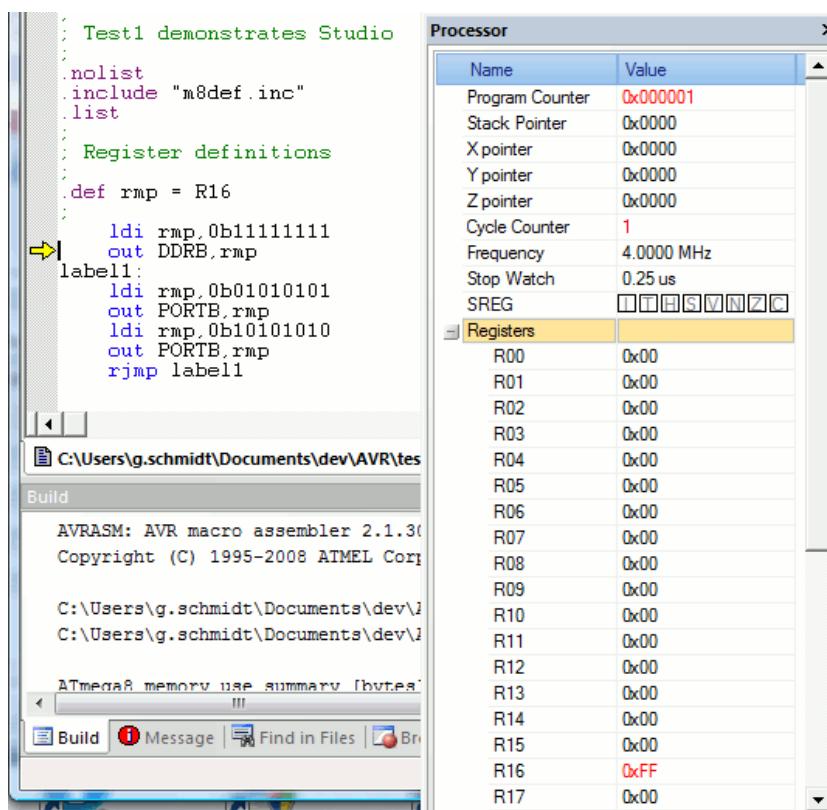
ATmega8 memory use summary [bytes]:
Segment Begin End Code Data Used Size Use%
-----
[.cseg] 0x000000 0x00000e 14 0 14 8192 0.2%
[.dseg] 0x000060 0x000060 0 0 0 1024 0.0%
[.eseg] 0x000000 0x000000 0 0 0 512 0.0%

● Assembly complete, 0 errors. 0 warnings
```

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.

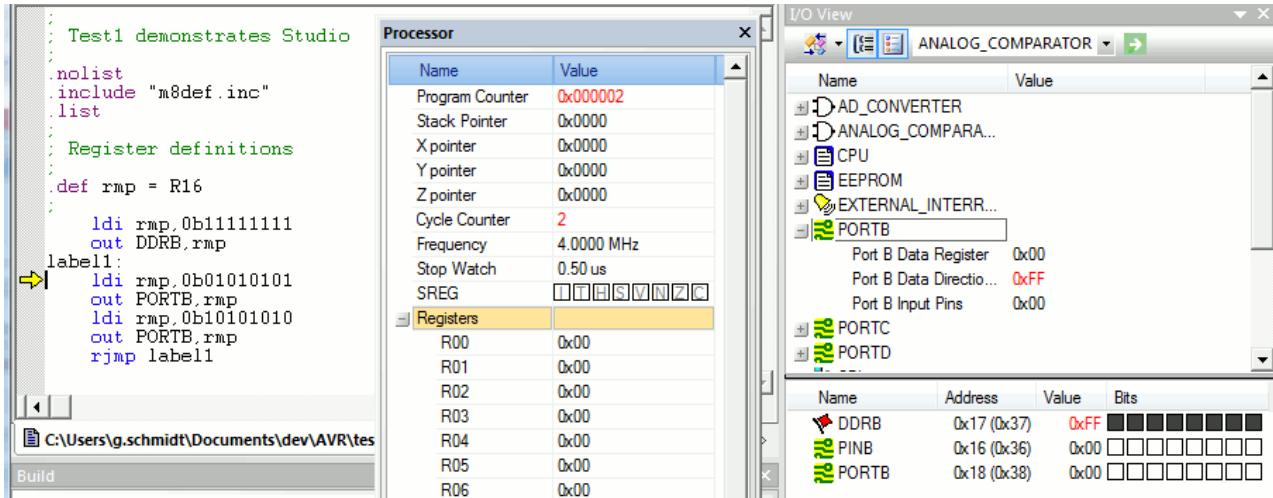


Im linken Editor-Fenster taucht nun ein gelber Pfeil auf, der auf die erste ausführbare Instruktion des Programmes zeigt. Mit "View", "Toolbars" und "Processor" bringt man das Prozessorfenster rechts zur Anzeige. Es bringt uns Informationen über die aktuelle Ausführungsadresse, die Anzahl der verarbeiteten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

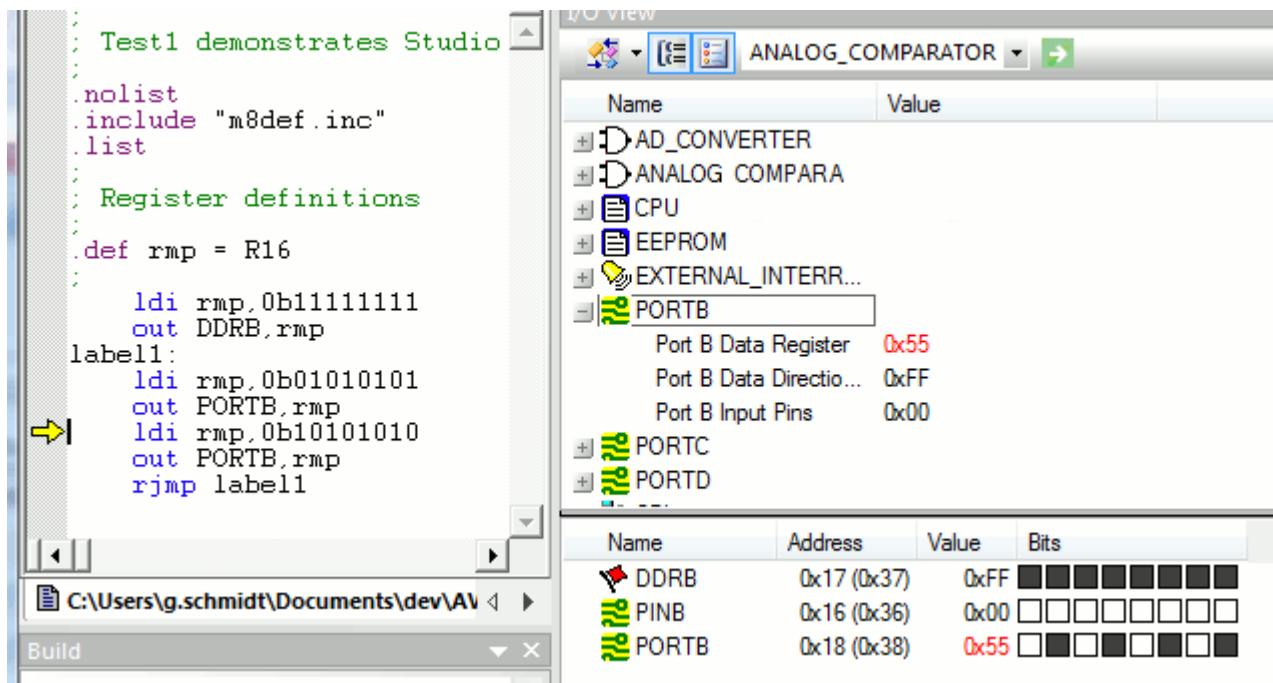


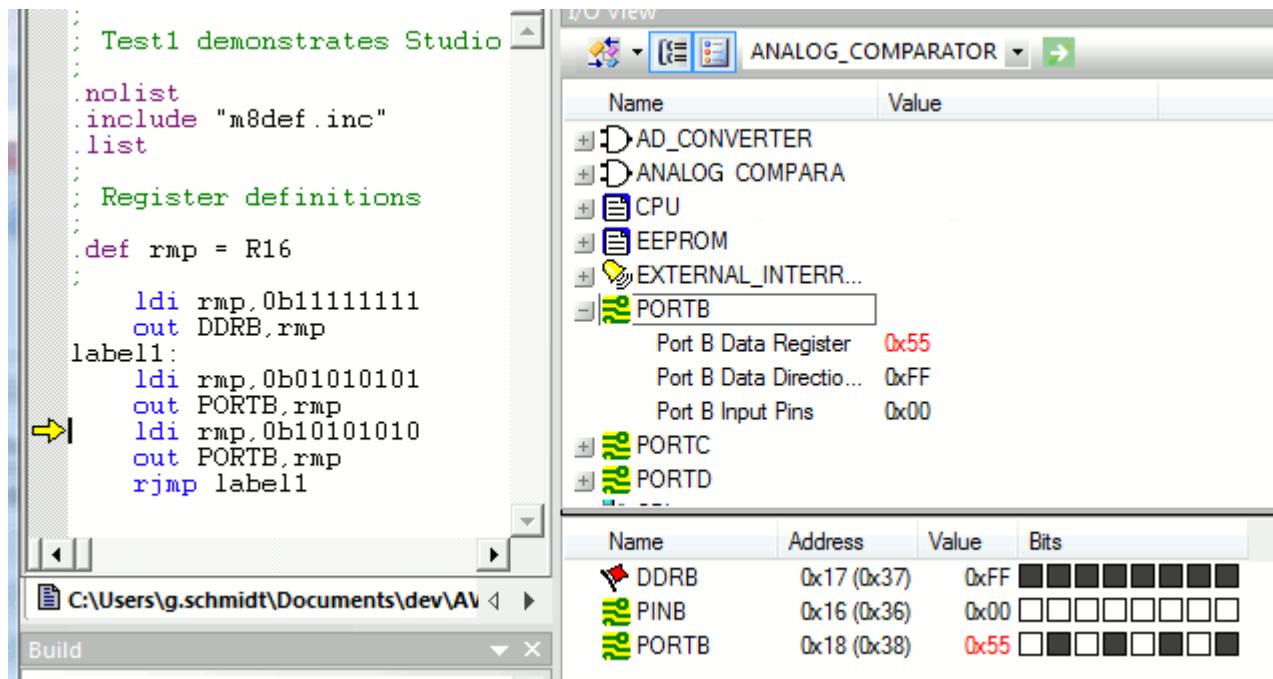
Mit "Debug" und "Step into" oder der Taste F11 wird nun ein Einzelschritt vorgenommen. Der Programmzähler hat sich nun verändert, er steht auf "0x000001". Der Schrittzähler hat einen Zyklus gezählt und im Register R16 steht nun hexadezimal "0xFF" oder dezimal 255. LDI lädt also einen Hexadezimalwert 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".



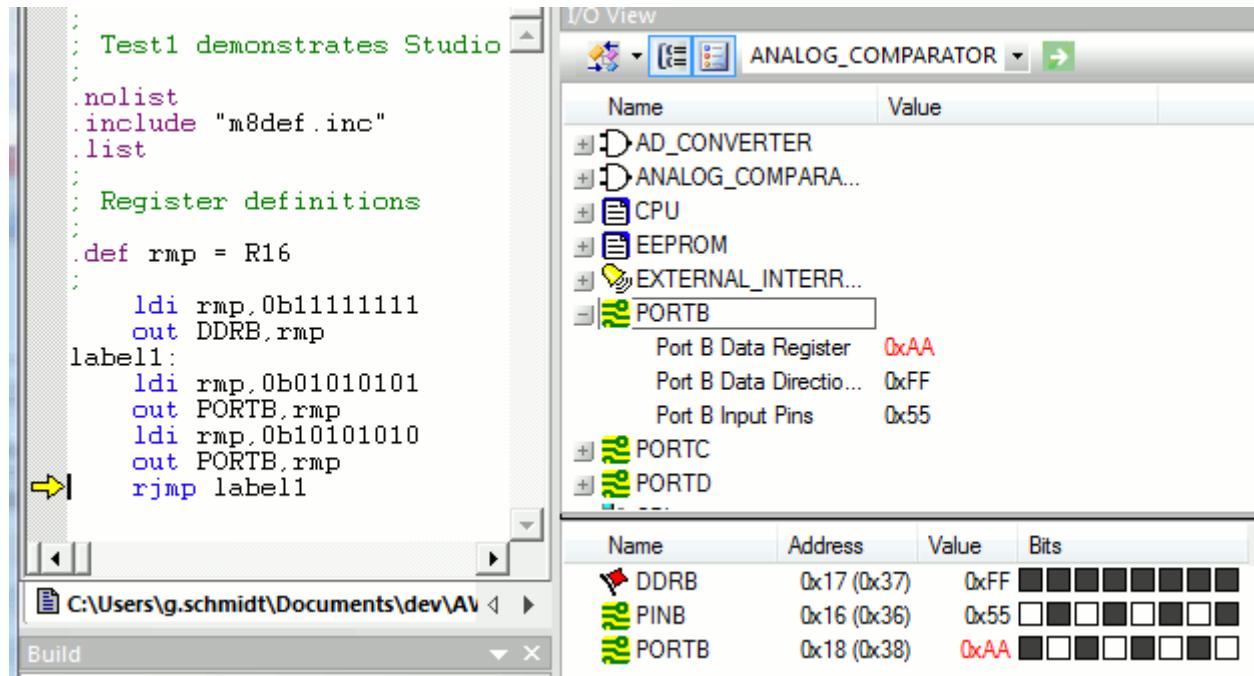
Das Richtungs-Port-Register Data Direction PortB (DDRB) hat nun 0xFF und in der Bitanzeige unten acht schwarze Kästchen.





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 Registrernamen.

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 Oszillatorkennfrequenz 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 Unterpro-

gramme. 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
; =====
```

```
; [Alle 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 Initialisierungen 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:
```

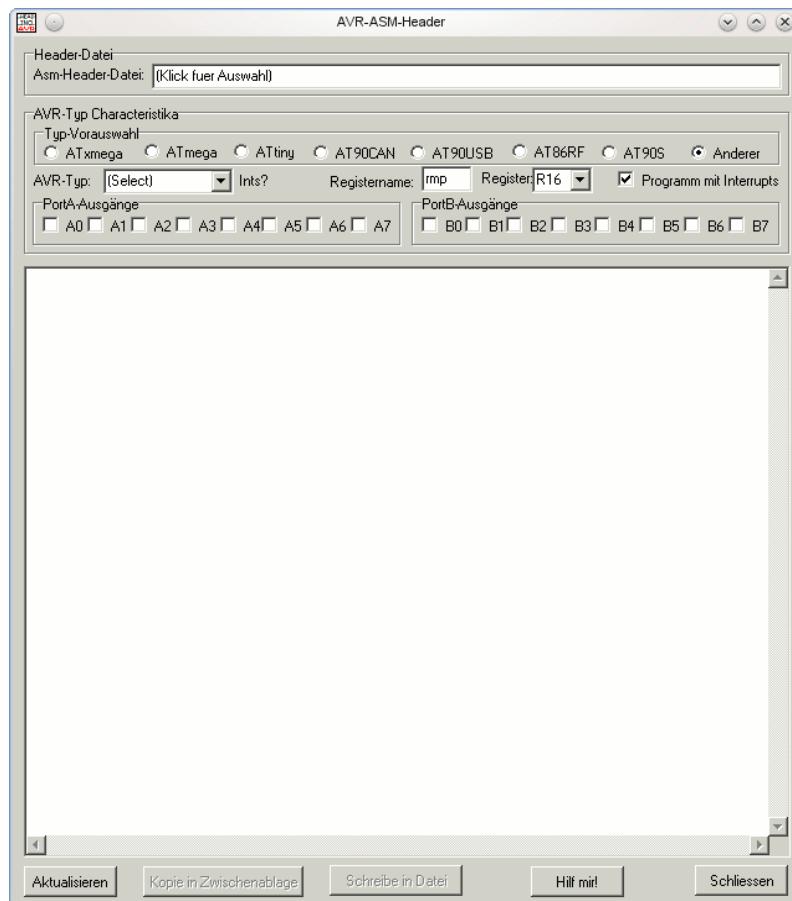
```
sleep ; schlafen legen
nop ; Dummy fuer Aufwachen
rjmp loop ; zurueck in die Schlafposition
;
; Ende Quellcode
;
.db „(C)201x by mich!“ ; menschenlesbar
.db „C(2)10 xybm ci!h“ ; wortweise lesbar
;
```

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 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:

<i>Zeiger</i>	<i>Vorgang</i>	<i>Beispiele</i>
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 wortweisen 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	RWWBSB	—	RWWBSRE	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 wohlgelingenden 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!

<i>Gerät</i>	<i>Kurz</i>	<i>Port</i>	<i>Name</i>
Akkumulator	SREG	Status Register	SREG
Stack	SPL/SPH	Stackpointer	SPL/SPH
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

<i>Gerät</i>	<i>Kurz</i>	<i>Port</i>	<i>Name</i>
Timer 0	Timer 0	Timer/Counter 0 Control Register	TCCR0
		Timer/Counter 0	TCNT0
Timer 1	<u>Timer 1</u>	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

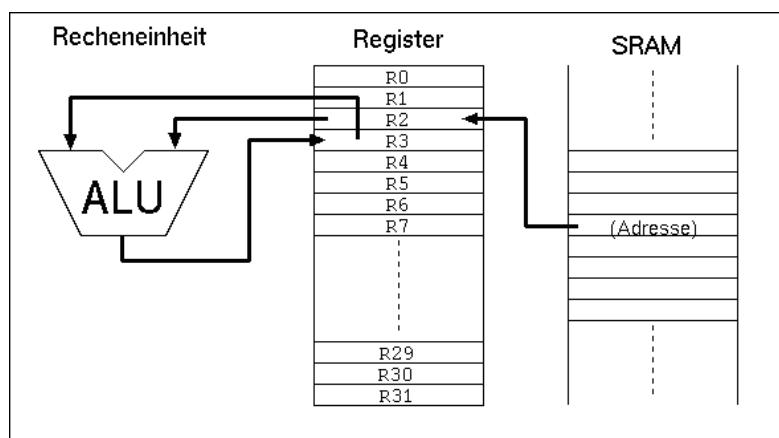
8.1 Verwendung von SRAM in AVR Assembler

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 relativ 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 Wer-

te zeitweise ablegen kann, seien es Rücksprungadressen beim Aufruf von Unterprogrammen, bei der Unterbrechung 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 fortgeschrittenen 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

```
LDI 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 Stacks

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 Stacks 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 Pro-

zessor. 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 Stacks

Die Verwendung des Stacks ist unproblematisch. So lassen sich Werte von Registern auf den Stack 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 Stack abgelegt wurde. Wichtig: Selbst wenn der Wert vielleicht gar nicht mehr benötigt wird, muss er mit Pop wieder vom Stack! Das Ablegen des Registers auf den Stack 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 Stacks ziemlich nutzlos und verschwendet bloß Zeit.

Stack zum Ablegen von Rücksprungadressen

Noch wertvoller ist der Stack 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 Stack abgelegt, nach Beendigung wieder vom Stack 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 Stapsels 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 Stapsels, 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 Ausführungsmitik 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 assemblyieren 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 Assemblyieren 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.

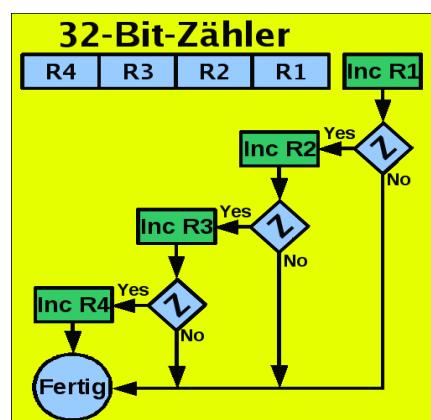
Die Instruktion RJMP bewirkt einen Sprung an die Stelle im Programm mit der Kennzeichnung Start:, auch ein Label genannt. Die Stelle Start: folgt weiter unten im Programm und ist dem Assembler hiermit entsprechend mitgeteilt: Labels beginnen immer in Spalte 1 einer Zeile und enden mit einem Doppelpunkt. 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 Erhöhorei 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 $1/4 \mu\text{s}$ 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 Sprungin-

struktionen 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 Makronamens 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 kommt 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 Rückprungadresse beim RCALL auf dem Stapel abgelegt wird.

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

[...] irgendwo im Programm

RJMP Delay10

Zurück:

[...] 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 Instruktionssworte. 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-Registerpaars 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 fragt, ob er sich nun geändert hat. Die Methode heisst Polling, weil es wie bei den 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 Pro-

zessor 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			<i>ausgelöst durch ...</i>
	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
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 durchnummieriert. 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 AT-mega 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 Serviceroutine).

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.

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 Zeitspanne eines Prozessortakts.

Die Anzahl Prozessortakte, die die einzelnen Instruktionen benötigen, stehen in den Datenbüchern der AVR. 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 verstehen:

- 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.

- AVR 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.
- AVR mit internen RC-Oszillatoren bieten ferner die Möglichkeit, diesen zu verstehen. 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 Verteiler 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 Verteiler 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 (f_c), dann ergibt sich die Dauer eines Prozessortakts (t_c) zu: $t_c [\mu\text{s}] = 1 / f_c [\text{MHz}]$ 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.

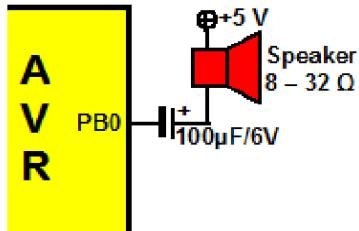
Verlängerung!

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
        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 μ F, und lassen Sie das folgende Programm auf den ATtiny13 los:

```
.include "tn13def.inc" ; für einen ATtiny13
.equ c1 = 0 ; Bestimmt die Tonhöhe
            ; 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:

```

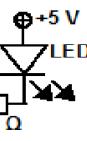
.equ c1 = 0 ; 0 Takte, macht der Assembler alleine
    ldi R25,HIGH(c1) ; 1 Takt
    ldi R24,LOW(c1) ; 1 Takt
Loop: ; Schleifenbeginn
    nop ; tue nichts, 1 Takt
    sbiw R24 ; 2 Takte
    brne Loop ; 2 Takte wenn nicht Null, 1 Takt bei Null

```

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 Sekundenrhythmus blinkenden LED an einem AVR-Port. Die Hardware, die es dazu braucht, ist bescheiden. Die Software steht da:

```
.include "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 Adresse 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-Adressen 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 können. Die Verwendung der .ORG-Direktive zum Einstellen der ISR-Adresse 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-Adressen nach Prioritäten geordnet: die ISR mit der niedrigsten Adresse 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 Prozessor mit Routinen des Hauptprogramms beschäftigt ist, die das Statusregister verwenden, - kann eine Störung dieses Registers unvorhersehbare Folgen haben.

- Beim Ansprung der ISR wird die Rücksprungadresse 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 Interrupts 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 überlaufen 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 so-nannten 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 zuzählende 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 toll 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 Initialisierung
rjmp Int0Sr ; Externer Interrupt an INT0, Sprung zur Service-Routine
reti ; Irgendein anderer Interrupt, nicht benutzt
rjmp IntTc00vflwSr ; 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 Initialisierung
.org $0001
    rjmp Int0Sr ; Externer Interrupt an INT0, Sprung zur Service-Routine
.org $0003
    rjmp IntTc00vflwSr ; 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 Unterrouitnen 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 lief 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 Scheinkompatibilität 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 Initialisierung
jmp Int0Sr ; Externer Interrupt an INT0, Sprung zur Service-Routine
reti ; Irgendein anderer Interrupt, nicht benutzt
nop
```

```

jmp IntTc00vflwSr ; 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

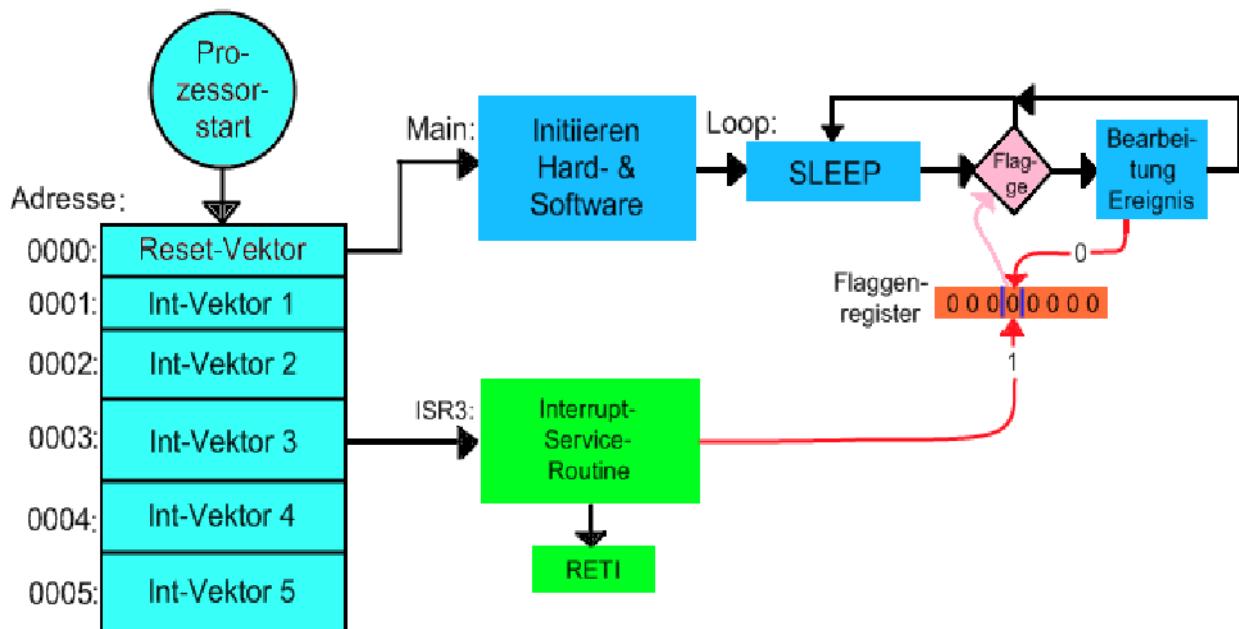
Den Standardablauf eines interrupt-gesteuerten Propgrammes in allgemeiner Form zeigt das folgende Bild.

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 interruptgesteuerten 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

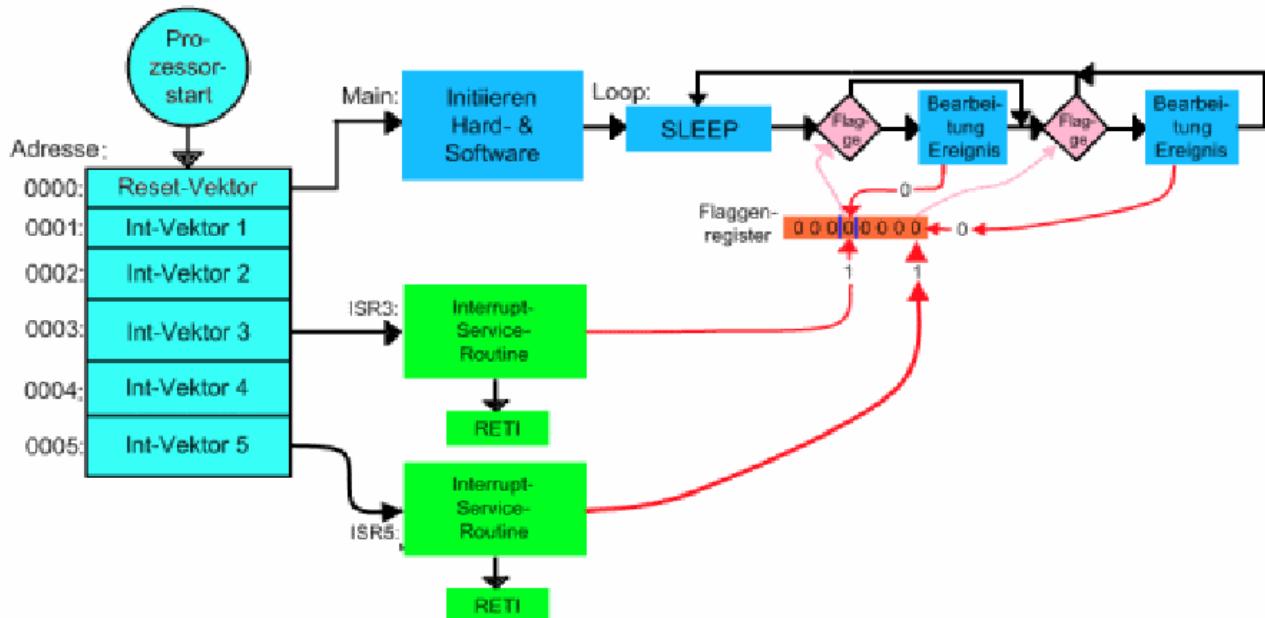
Irgendwann schlägt nun Interrupt 3 zu. Der Prozessor

- 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.

ziert: 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.).

1.1 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 Programmablaufs auf die Interrupts. Sie sind die zentrale Steuerung.

11.5 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 Registerverbrauch	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 Zählrythmus empfindlich stören. Dabei hilfe 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 nacheinander 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 Behandlungs Routinen 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 wirkende 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.

11.6 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 gleichzeitiger auftretender Interruptanforderung wird der Interrupt mit der höheren Priorität zuerst bearbeitet.

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

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

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 Polaritätswechsel oder nur bei positiven bzw. negativen Flanken	alle	ANA ANAn
Weitere	(diverse)	(weitere Interrupt-Arten sind den Handbüchern zu entnehmen)	einige	

11.6.2 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
ATtiny24	RES	INT0	PCI0	PCI1	WDT	TC1P	TC1A	TC1B	TC1O	TC0A	TC0B	TC0O	ANA	ADC	EERY	USIS	USIO				
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.

11.6.3 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 jeweils 16 Vektoren unterteilt.

Interrupts mit den Vektoren 0000 bis 001E

<i>AVR-Typ/Adr</i>	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	TC2O	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

<i>AVR-Typ/Adr</i>	<i>0000</i>	<i>0002</i>	<i>0004</i>	<i>0006</i>	<i>0008</i>	<i>000A</i>	<i>000C</i>	<i>000E</i>	<i>0010</i>	<i>0012</i>	<i>0014</i>	<i>0016</i>	<i>0018</i>	<i>001A</i>	<i>001C</i>	<i>001E</i>
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
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

<i>AVR-Typ/Adr</i>	<i>0020</i>	<i>0022</i>	<i>0024</i>	<i>0026</i>	<i>0028</i>	<i>002A</i>	<i>002C</i>	<i>002E</i>	<i>0030</i>	<i>0032</i>	<i>0034</i>	<i>0036</i>	<i>0038</i>	<i>003A</i>	<i>003C</i>	<i>003E</i>
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**

<i>AVR-Typ/Adr</i>	<i>0040</i>	<i>0042</i>	<i>0044</i>	<i>0046</i>	<i>0048</i>	<i>004A</i>	<i>004C</i>	<i>004E</i>	<i>0050</i>	<i>0052</i>	<i>0054</i>	<i>0056</i>	<i>0058</i>	<i>005A</i>	<i>005C</i>	<i>005E</i>
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

<i>AVR-Typ/Adr</i>	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 defi-

niert, 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: 1000.0000) 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:

<i>Byte</i>	<i>Ziffern</i>	<i>Wert</i>	<i>8</i>	<i>4</i>	<i>2</i>	<i>1</i>	<i>8</i>	<i>4</i>	<i>2</i>	<i>1</i>
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 hexadezimale (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 Zahlenaufwandlung 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) fünfbitigen 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 Platzverschwendug 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-Flag 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 ($9+9=18 = \text{hex } 12$). 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 Halübertrags-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.

ADD R1,R2 (Half)Carry-Bit	ADD Nibble,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 R17,0xF0
```

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 Be-rechnung ist fertig.

Kürzer geht es so.

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

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 niedrigwertige 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:

$$\begin{array}{r}
 1234 * 567 = ?
 \\ -----
 \\ 1234 * 7 = 8638
 \\ + 12340 * 6 = 74040
 \\ + 123400 * 5 = 617000
 \\ -----
 \\ 1234 * 567 = 699678
 \\ =====
 \end{array}$$

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 rml = 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 rml,rmp ; in erstes Multiplikationsregister
    ldi rmp,0x55 ; Beispielzahl 0101.0101
    mov rm2,rmp ; in zweites Multiplikationsregister
; Hier beginnt die Multiplikation der beiden Zahlen
; in rml 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:rml zum Ergebnis
; in reh:rel (mit Überlauf der unteren 8 Bits!)
    add rel,rml ; addiere LSB von rml zum Ergebnis
    adc reh,rmh ; addiere Carry und MSB von rml
MULT8b:
; Vierter Schritt: Multiplizierte Multiplikator rmh:rml mit
; Zwei (16-Bit, links schieben)
    clc ; Carry bit auf Null setzen
    rol rml ; 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:

```

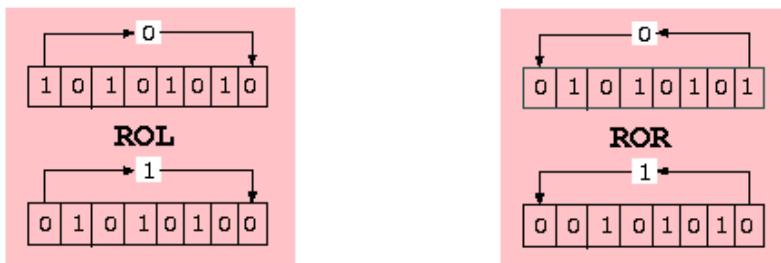
```
r jmp loop
```

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 Links-schieben mit dem Inhalt des Registers von 0xAA, für das Rechtsschieben mit 0x55 gezeigt:



Multiplikation im Studio

```

AVR Studio - Mult8.asm
File Edit Project Debug Breakpoints Trace & triggers Watch Options View Tools Window Help
File Project Debug Breakpoints Trace & triggers Watch Options View Tools Window Help
Mult8
Mult8.asm
.MDEF rel = R3 ; Ergebnis, LSB (16 Bit)
.MDEF reh = R4 ; Ergebnis, MSB
.MDEF rmh = R16 ; Hilfsregister zum Laden
.CSEG
.ORG 0000

: ldi rmh, 0xAA ; Beispielzahl 1010.1010
: mov rml, rmh ; in erstes Multiplikationsreg
: ldi rm1, 0x55 ; Beispielzahl 0101.0101
: mov rm2, rm1 ; in zweites Multiplikationsreg
;
; 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

```

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.

```

; rjmp START
START:
    ldi rmp.0xAA ; Beispielzahl 1010.1010
    mov rm1.rmp ; in erstes Multiplikationsreg
    ldi rmp.0x55 ; Beispielzahl 0101.0101
    mov rm2.rmp ; in zweites Multiplikationsreg

; Hier beginnt die Multiplikation der beiden Zahlen
; in rm1 und rm2, das Ergebnis ist in reh:rel (16 Bit)

MULT8:
    ; Anfangswerte auf Null setzen
    clc rmh ; Hilfsregister leeren
    clc rel ; Ergebnis auf Null setzen
    clc reh

; Hier beginnt die Multiplikationsschleife
MULT8a:

```

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

```

    clr reh

; Hier beginnt die Multiplikationsschleife

MULT8a:
    ; Erster Schritt: Niedrigstes Bit von Multiplikator 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.rml zum Ergebnis
; in reh:rel (mit Überlauf der unteren 8 Bits!)
    add rel.rml ; addiere LSB von rml zum Ergebnis

```

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 reteste 1 liegt im Carry-Bit des Status-Registers herum.

```

; 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

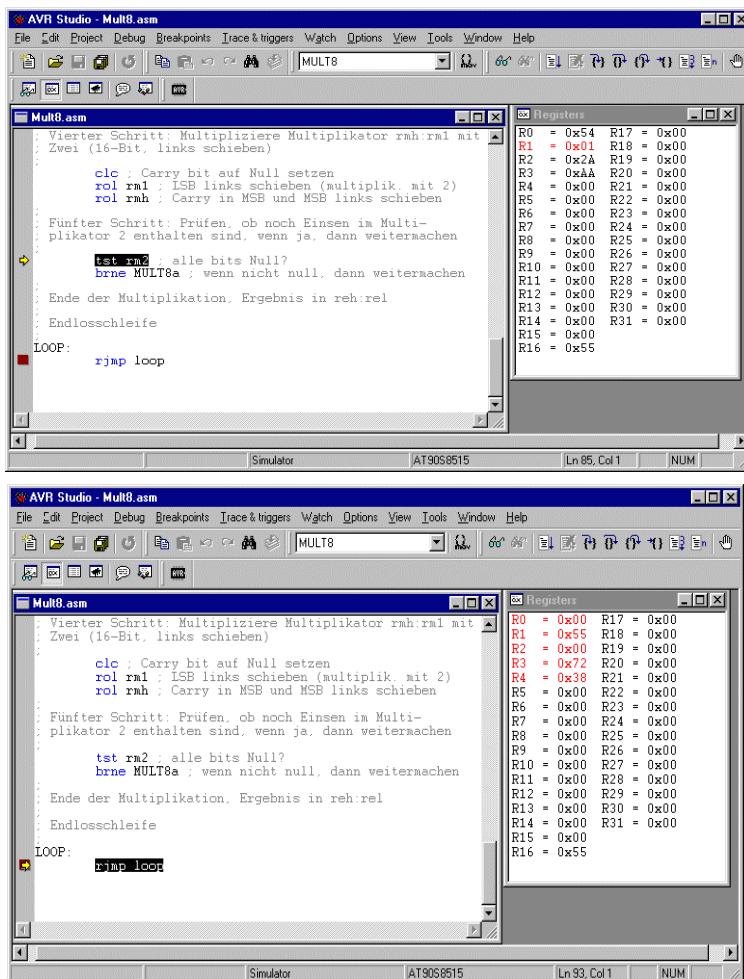
; Dritter Schritt: Addiere 16 Bits in rmh.rml zum Ergebnis
; in reh:rel (mit Überlauf der unteren 8 Bits!)
    add rel.rml ; addiere LSB von rml zum Ergebnis
    adc reh.rmh ; addiere Carry und MSB von rml

MULT8b:
    ; Vierten Schritt: Multiplizierte Multiplikator rmh:rml 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-

```

Weil im Carry eine Eins war, wird 00AA in den Registern R1:R0 zu dem (leeren) Registerpaar R4:R3 hinzugiert. 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 ; Hilfsregister
.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 Multiplikation mit 2)
        brcc div8b ; Eine 1 ist herausgerollt, ziehe ab
        cp rd1u,rd2 ; Divisionsergebnis 1 oder 0?
        brcc 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
```

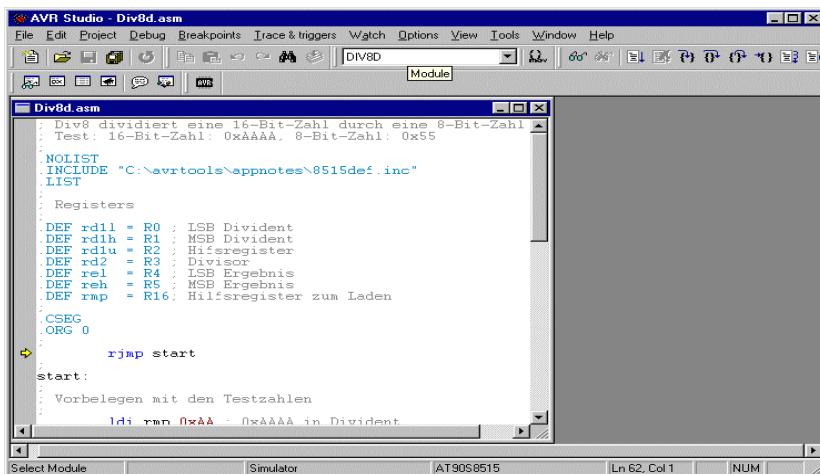
Programmschritte beim Dividieren

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.

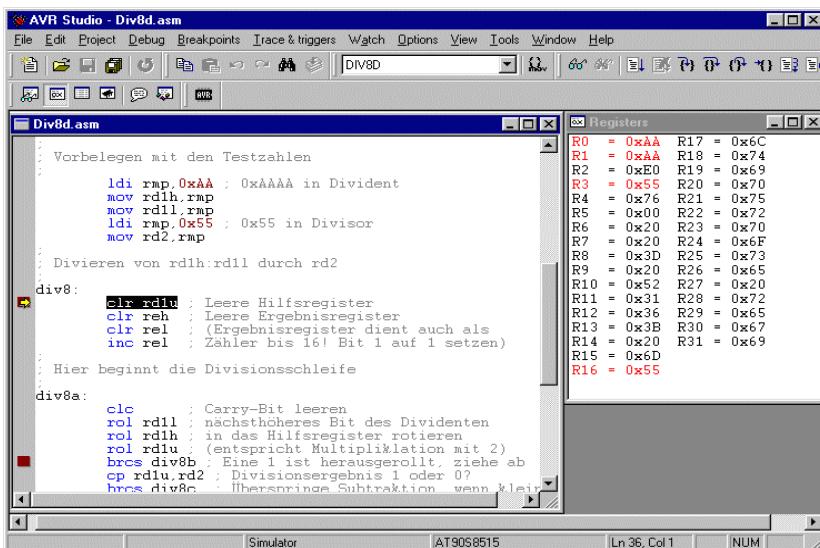
Für das Rotieren gilt das oben dargestellte Procedere (siehe oben, zum Nachlesen).

Das Dividieren im Simulator



Die folgenden Bilder zeigen die Vorgänge im Simulator, dem Studio. Dazu wird der Quellcode assembled 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.

The screenshot shows the AVR Studio interface with the assembly file 'Div8d.asm' open. The code implements a 8-bit division algorithm. The registers window shows initial values: R0=0xAA, R1=0xAA, R2=0x00, R3=0x55, R4=0x01, R5=0x00, R6=0x20, R7=0x20, R8=0x3D, R9=0x20, R10=0x52, R11=0x31, R12=0x36, R13=0x3B, R14=0x20, R15=0x6D, R16=0x55.

```

; Vorbelegen mit den Testzahlen
ldi rmp, 0xAA ; 0AAAA in Divident
mov rd1h,rmp
mov rd1l,rmp
ldi rmp, 0x55 ; 0x55 in Divisor
mov rd2,rmp

; Dividieren 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 Multiplikation mit 2)
    brcc div8b ; Eine 1 ist herausgerollt, ziehe ab
    cp rd1u,rd2 ; Divisionsergebnis 1 oder 0?
    brcc div8c ; Überspringe Subtraktion, wenn klein

```

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

The screenshot shows the AVR Studio interface with the assembly file 'Div8d.asm' open. The code is at the start of the division loop. The registers window shows updated values: R0=0x54, R1=0x55, R2=0x01, R3=0x55, R4=0x01, R5=0x00, R6=0x20, R7=0x20, R8=0x3D, R9=0x20, R10=0x52, R11=0x31, R12=0x36, R13=0x3B, R14=0x20, R15=0x6D, R16=0x55.

```

; Vorbelegen mit den Testzahlen
ldi rmp, 0xAA ; 0AAAA in Divident
mov rd1h,rmp
mov rd1l,rmp
ldi rmp, 0x55 ; 0x55 in Divisor
mov rd2,rmp

; Dividieren 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 Multiplikation mit 2)
    brcc div8b ; Eine 1 ist herausgerollt, ziehe ab
    cp rd1u,rd2 ; Divisionsergebnis 1 oder 0?
    brcc div8c ; Überspringe Subtraktion, wenn klein

```

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

The screenshot shows the AVR Studio interface with the assembly file 'Div8d.asm' open. The code is at the start of the second iteration of the loop. The registers window shows updated values: R0=0x54, R1=0x55, R2=0x01, R3=0x55, R4=0x02, R5=0x00, R6=0x20, R7=0x20, R8=0x3D, R9=0x20, R10=0x52, R11=0x31, R12=0x36, R13=0x3B, R14=0x20, R15=0x6D, R16=0x55.

```

; Vorbelegen mit den Testzahlen
ldi rmp, 0xAA ; 0AAAA in Divident
mov rd1h,rmp
mov rd1l,rmp
ldi rmp, 0x55 ; 0x55 in Divisor
mov rd2,rmp

; Dividieren 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 Multiplikation mit 2)
    brcc div8b ; Eine 1 ist herausgerollt, ziehe ab
    cp rd1u,rd2 ; Divisionsergebnis 1 oder 0?
    brcc div8c ; Überspringe Subtraktion, wenn klein
    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
    brcc div8a ; solange Nullen aus dem Ergebnis
    ; rotieren: weitermachen
; Ende der Division erreicht
stop:
    rjmp stop ; Endlosschleife

```

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.

The screenshot shows the AVR Studio interface. On the left, the assembly code for 'Div8d.asm' is displayed:

```

    brcs div8b ; Eine 1 ist herausgerollt, ziehe ab
    cp rd1u.rd2 ; Divisionsergebnis 1 oder 0?
    brcs div8c ; Uberspringe Subtraktion, wenn &leir
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
    brcs div8a ; solange Nullen aus dem Ergebnis
    ; rotieren: weitermachen
; Ende der Division erreicht
stop:
    rjmp stop      ; Endlosschleife

```

On the right, the 'Registers' window shows the state of all 16 registers (R0-R16) after the division. The relevant values are:

Register	Value
R0	0x00
R1	0x00
R2	0x00
R3	0x55
R4	0x02
R5	0x02
R6	0x20
R7	0x20
R8	0x3D
R9	0x20
R10	0x52
R11	0x31
R12	0x36
R13	0x3B
R14	0x20
R15	0x6D
R16	0x55
R17	0x6C
R18	0x74
R19	0x69
R20	0x70
R21	0x75
R22	0x72
R23	0x70
R24	0x6F
R25	0x73
R26	0x65
R27	0x20
R28	0x72
R29	0x65
R30	0x67
R31	0x69

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 0xFFFF 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 und 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.

Hardware-Multiplication 8-Bit * 8-Bit

$$a1 * b1 = e2:e1$$

$$a1 * b1$$

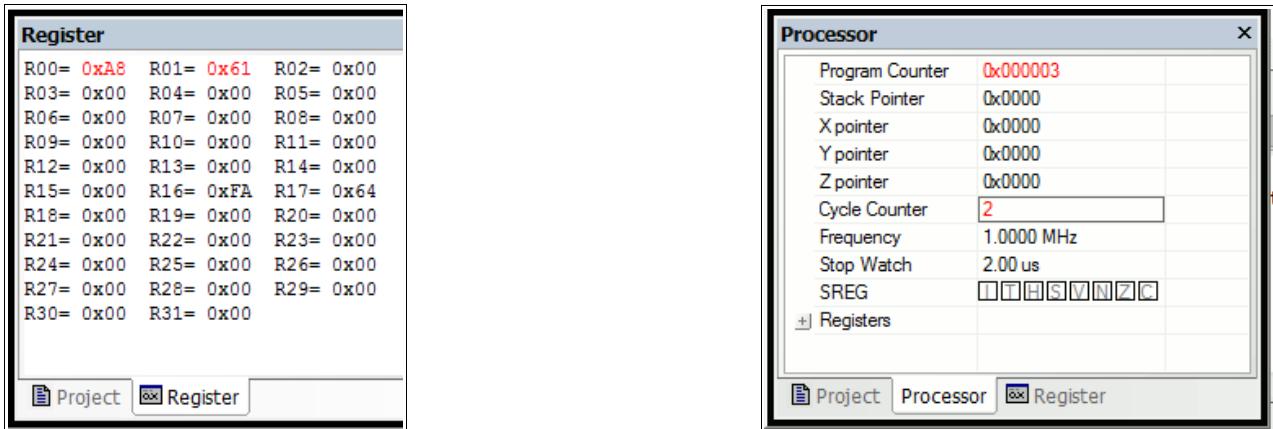


Das Programm demonstriert die Simulation im Studio. Es multipliziert dezimal 250 (hex FA) mit dezimal 100 (hex 64) in den Registern R16 und R17.

```

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

```



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 $256*m1M + m1L$, wobei m1M das MSB und m1L das LSB ist.

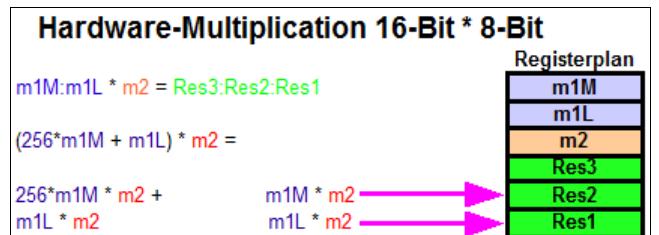
Multiplikation dieser Zahl mit der 8-Bit-Zahl m2 ist also mathematisch ausgedrückt:

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

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
```

Register			
R00= 0x00	R01= 0x00	R02= 0x00	
R03= 0x00	R04= 0x00	R05= 0x00	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xFA	R19= 0x00	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

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

Register			
R00= 0xA0	R01= 0x0F	R02= 0x00	
R03= 0x00	R04= 0x00	R05= 0x00	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xFA	R19= 0x00	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

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.

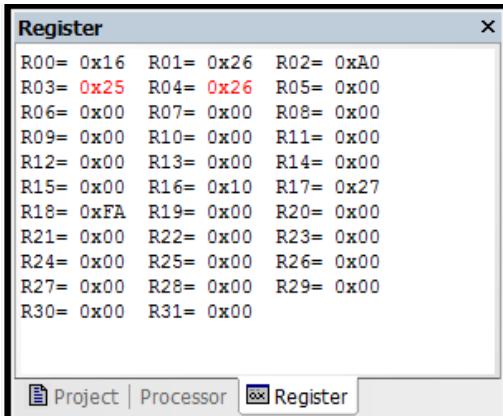
Register			
R00= 0x16	R01= 0x26	R02= 0xA0	
R03= 0x0F	R04= 0x00	R05= 0x00	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xFA	R19= 0x00	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

```
mov Res3,R1 ; Kopiere MSB Ergebnis zum Byte 3
add Res2,R0 ; Addiere LSB Ergebnis zum Byte 2
```

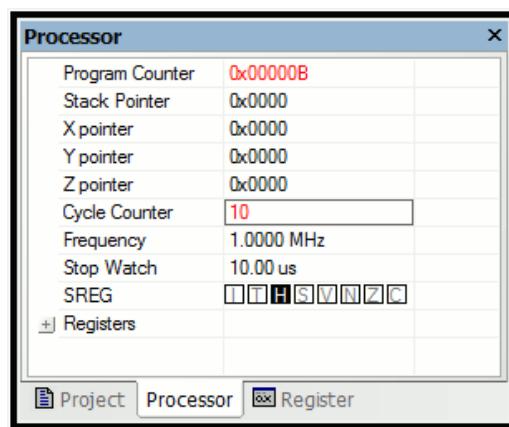
```

brcc NoInc ; Wenn Carry nicht gesetzt, springe
inc Res3 ; erhöhe 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:

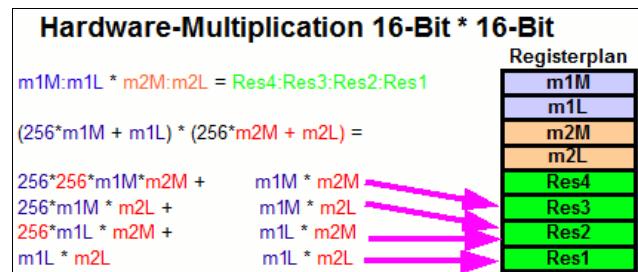
$$\begin{aligned}
m1 * m2 = \\
(256*m1M + m1L) * (256*m2M + m2L) = \\
65536*m1M*m2M + 256*m1M*m2L + 256*m1L*m2M + m1L*m2L
\end{aligned}$$

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
;

```

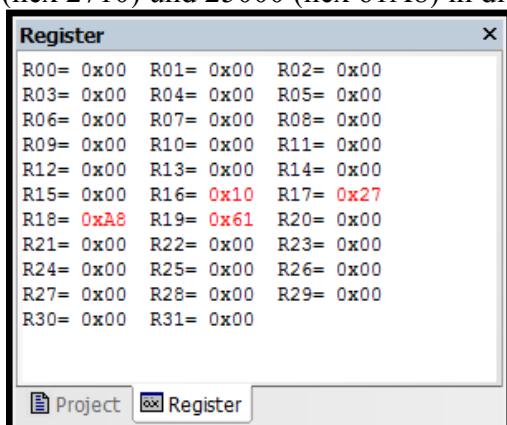


```

; 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) ...

Register	R00= 0xC7	R01= 0x0E	R02= 0x00
R03= 0x00	R04= 0xC7	R05= 0x0E	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xA8	R19= 0x61	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

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

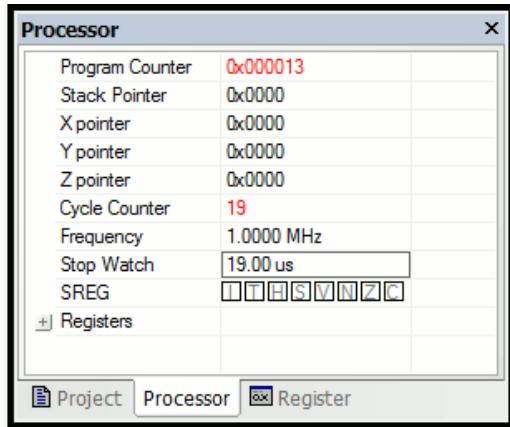
Register	R00= 0x80	R01= 0x0A	R02= 0x80
R03= 0xA0	R04= 0xC7	R05= 0x0E	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xA8	R19= 0x61	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

Register	R00= 0x98	R01= 0x19	R02= 0x80
R03= 0xA2	R04= 0xE0	R05= 0x0E	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xA8	R19= 0x61	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

Register	R00= 0x10	R01= 0x06	R02= 0x80
R03= 0xB2	R04= 0xE6	R05= 0x0E	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0x10	R17= 0x27	
R18= 0xA8	R19= 0x61	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

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 ...

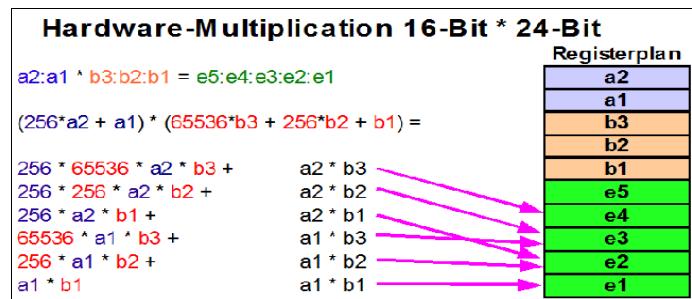


Die Multiplikation hat 19 Taktzyklen gebraucht, das ist sehr viel schneller als die Software-Multiplikation.

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 Zahenumwandlung 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 Zahenumwandlung 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 Zahenumwandlung

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 $1 \cdot (2^0) + 1 \cdot (2^1) + 0 \cdot (2^2) + 1 \cdot (2^3)$, so ähnlich wie dezimal 1234 gleich $4 \cdot (10^0) + 3 \cdot (10^1) + 2 \cdot (10^2) + 1 \cdot (10^3)$ 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 $1 \cdot (2^3) + 1 \cdot (2^1)$, 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 herumstehen könnte. Sorgfältige Planung der Register ist dann heftig angesagt!
- Paketeinteilung: Weil man nicht immer alle Umwandlungs routinen 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 Unterrountinen: 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 enthält ein ungültiges
; Zeichen
;         (Z zeigt auf das ungültige Zeichen).
; Benötigte Register: rBin1H:L (Ergebnis), R0 (wiederhergestellt), rBin2H:L (wieder
; hergestellt), rmp
; Aufgerufene Unterrountinen: 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 Unterrountinen: 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 ; Multiplizierte 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 Unterrouitinen: 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 Unterrouitinen: 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: Ergebnis in Z (bei Aufruf), Z um eine Stelle erhöht, keine Fehlerbehandlung
; Benutzte Register: rBin1H:L (enthält Rest der Binärzahl), rBin2H (bleibt erhalten),
rmp
; Aufgerufene Unterrouitinen: -
;
Bin2ToDigit:
    clr rmp ; Zähler auf Null
Bin2Digita:
    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 Unterrouitinen: 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 Unterrountinen: 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 ; Ergebnis 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
    brccs Hex1ToBin1b ; Fehler, kleiner als 0
    cpi rmp,10 ; A..F ; Ziffer größer als 9?
    brccs Hex1ToBin1c ; nein, fertig
    cpi rmp,$30 ; Kleinbuchstaben?
    brccs 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?
    brccs Hex1ToBin1b ; Ja, Fehler
    cpi rmp,16 ; Ziffer größer als $0F
    brccs Hex1ToBin1c ; Nein, Ziffer in Ordnung
Hex1ToBin1b: ; Error
    sbiw ZL,1 ; Ein Zeichen zurück
    set ; Setze Fehlerflagge
Hex1ToBin1c:
    ret ; Zurück

```

14 Umgang mit Festkommazahlen in AVR Assembler

14.1 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.

14.2 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 $500 \cdot 256 / 255$ 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.

14.3 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 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-
; gestern 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
; Ablaufsteuerung 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-Umwandlungs routinen
; Ende des Umwandlungstestprogramms
```

15 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
Addition	Register => Port	OUT p1, r1		1
	Register => Stack	PUSH r1		2
	8 Bit, +1	INC r1	Z N V	1
	8 Bit	ADD r1, r2	Z C N V H	1
Subtraktion	8 Bit+Carry	ADC r1, r2	Z C N V H	1
	16 Bit, Konstante	ADIW rd, k63	Z C N V S	2
	8 Bit, -1	DEC r1	Z N V	1
	8 Bit	SUB r1, r2	Z C N V H	1
Multiplikation	8 Bit, Konstante	SUBI rh, k255	Z C N V H	1
	8 Bit - Carry	SBC r1, r2	Z C N V H	1

Funktion	Unterfunktion	Instruktion	Flags	Clk
	8 Bit - Carry, Konstante	SBCI rh,k255	Z 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
	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

Funktion	Unterfunktion	Instruktion	Flags	Clk
Vergleiche	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
Unbedingte Verzweigung	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
Bedingte 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
	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 gelösctem T-Bit	BRTC k127		1/2
	Springe bei Zweierkomplementüberlauf	BRVS k127		1/2

Funktion	Unterfunktion	Instruktion	Flags	Clk
Bedingte Sprünge	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

Instruktionen, alphabetisch

ADC r1,r2	CLR r1	RJMP k4096
ADD r1,r2	CLS	ROL r1
ADIW rd,k63	CLT	ROR r1
AND r1,r2	CLV	SBC r1,r2
ANDI rh,k255	CLZ	SBCI rh,k255
ASR r1	COM r1	SBI pl,b7
BLD r1,b7	CP r1,r2	SBIC pl,b7
BRCC k127	CPC r1,r2	SBIS pl,b7
BRCS k127	CPI rh,k255	SBIW rd,k63
BREQ k127	CPSE r1,r2	SBR rh,255
BRGE k127	DEC r1	SBRC r1,b7
BRHC k127	EOR r1,r2	SBRS r1,b7
BRHS k127	ICALL	SEC
BRID k127	IJMP IN r1,p1	SEH
BRIE k127	INC r1	SEI
BRLO k127	LD rp,(rp,rp+,-rp)	SEN
BRLT k127	LDL r1,ry+k63	SER rh
BRMI k127	LDI rh,k255	SES
BRNE k127	LDS r1,k65535	SET
BRPL k127	LPM	SEV
BRSH k127	LSL r1	SEZ
BRTC k127	LSR r1	SLEEP
BRTS k127	MOV r1,r2	ST (rp/rp+/-rp),r1
BRVC k127	NEG r1	STD ry+k63,r1
BRVS k127	NOP	STS k65535,r1
BST r1,b7	OR r1,r2 ORI rh,k255 OUT p1,r1	SUB r1,r2 SUBI rh,k255
CBI pl,b7	POP r1	SWAP r1
CBR rh,255	PUSH r1	TST r1
CLC	RCALL k4096	WDR
CLH	RET	
CLI		
CLN	RETI	

Ports, alphabetisch

ACSR, Analog Comparator Control & Status Reg.	SPCR, Serial Peripheral Control Register
DDR _x , Port x Data Direction Register	SPDR, Serial Peripheral Data Register
EEAR, EEPROM Address Register	SPSR, Serial Peripheral Status Register
ECCR, EEPROM Control Register	SREG, Status Register
EEDR, EEPROM Data Register	TCCR0, Timer/Counter Control Register 0
GIFR, General Interrupt Flag Register	TCCR1A, Timer/Counter Control Register 1 A
GIMSK, General Interrupt Mask Register	TCCR1B, Timer/Counter Control Register 1 B
ICR1L/H, Input Capture Register 1	TCNT0, Timer/Counter Register, Counter 0
MCUCR, MCU General Control Register	TCNT1, Timer/Counter Register, Counter 1
OCR1A, Output Compare Register 1 A	TIFR, Timer Interrupt Flag Register
OCR1B, Output Compare Register 1 B	TIMSK, Timer Interrupt Mask Register
PIN _x , Port Input Access	UBRR, UART Baud Rate Register
PORT _x , Port x Output Register	UCR, UART Control Register
SPL/SPH, Stackpointer	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	: Alternativecode, 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