Berufspädagogische Akademie des Bundes in Graz
Abteilung für technischen und gewerblichen Fachunterricht

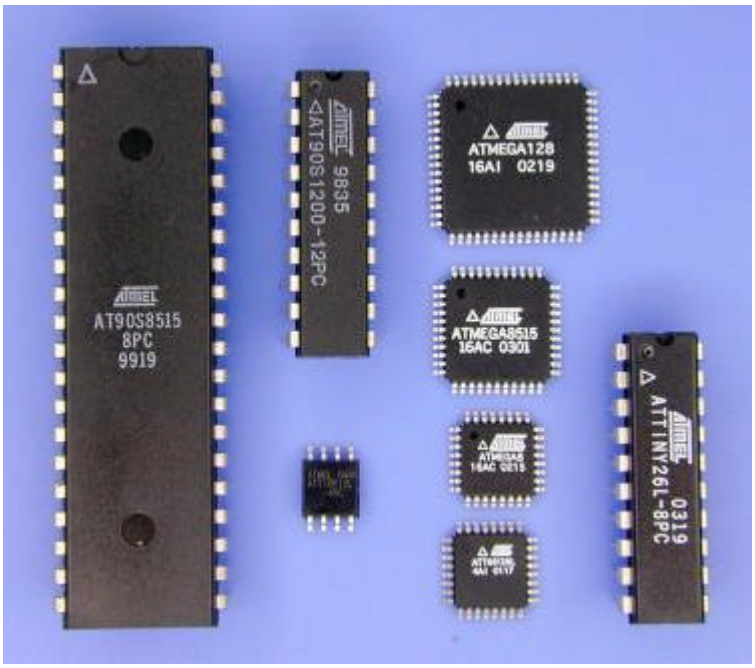


Abb. 1: Verschiedene AVR-µC von Atmel[1]

AVR-µC
Lernunterlage für das Selbststudium

Ersatzarbeit für die Lehrveranstaltung
„Fachliche Bildung“

Begutachter: **Dipl.-Ing. Franz Mitterer**
Eingereicht von **Ing. Hans Hafner**
Graz, 2004

[1] Vgl. URL: http://elm-chan.org/docs/avr_e.html, [10.2.2004].

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

1. EINLEITUNG

Der Umfang dieser Lernunterlage beschränkt sich auf die Entwicklungsumgebung **„AVRStudio 4“**, den darin enthaltenen **„AVR Simulator“** als ***Debug-Plattform***, den **„AT90S8515“** als ***Device (µC)*** und den **„AVR Assembler“** als ***Programmiersprache***. So zum besseren Verständnis Beispiele Verwendung finden, die nicht meiner eigenen Kreativität entspringen, verweise ich natürlich auf die Herkunft der verwendeten Quellcodes. Diese Lernunterlage erhebt keinen Anspruch auf vollständige Beschreibung des verwendeten **AVR**, sondern soll als Einstiegshilfe gedacht sein.

Da ich mich hier voll und ganz auf den technischen Inhalt dieser Arbeit konzentriert habe und einiges an Wissen aus meiner bisherigen Programmiererfahrung mit höheren Programmiersprachen hier verwendet habe, wurden von mir die bei wissenschaftlichen Arbeiten gültigen Zitierregeln nicht angewendet. Die von mir verwendeten Informationsquellen habe ich jedoch im Quellenverzeichnis zusammengestellt.

2. AVR-MICROCONTROLLER AT90S8515

2.1 ALLGEMEINES

Was ist ein Mikrocontroller (µC)?

Ein Mikrocontroller ist ein Mikrochip, der einen Mikroprozessor, Speicher, Digital- und Analog- Ein- und Ausgänge, etc. integriert hat, so dass eine Mikrocontroller-Anwendung oft mit wenigen anderen noch notwendigen Bauteilen auskommt. µC werden nach der Breite des internen Datenbusses unterschieden und brauchen zumeist eine extern eingespeiste Taktfrequenz zum Betrieb. Der hier behandelte **AVR AT90S8515** hat eine Datenbusbreite von 8-Bit und arbeitet mit einer Taktfrequenz von 4 MHz bzw. 8 MHz. Da die meisten Befehle nur einen Taktzyklus benötigen, kann dieser µC fast vier bzw. acht Millionen Befehle pro Sekunde verarbeiten.

Welcher µC soll verwendet werden?

Die Anwendungsgebiete für einem µC sind vielfältig und reichen von einfachen Regelungen bis hin zu komplizierten Messwerterfassungssystemen. Im Normalfall wird sich die Auswahl des zu verwendenden µC an der Anwendung orientieren. Für den Anfänger sollen jedoch folgende Kriterien erfüllt sein:

- geringer Preis
- handliche Bauform (weniger Pins ergeben zumeist weniger Probleme)
- In-System-Programmierbarkeit (ISP), um den µC zur Programmierung nicht aus der Schaltung entfernen zu müssen
- möglichst kostenlose, einfach verständliche Programmiersoftware
- Simulationsmöglichkeit am Personal Computer (PC)

Der hier verwendete **AVR AT90S8515** erfüllt diese für den Anfänger notwendigen Kriterien. Die Programmiersprache Assembler ist maschinennahe und hilft, die grundlegende Funktionsweise des AVR-µC leichter zu verstehen. Die Entwicklungs- und Simulationsumgebung ***„AVR Studio 4“*** ermöglicht die Codierung (Programmerstellung) in AVR Assembler, die Simulation des erstellten Programms am Personal Computer und auch die Übertragung des in Maschinensprache übersetzten (compilierten, assemblierten) Programms zum µC.

Als Alternative zur Programmiersprache AVR Assembler gibt es auch einen kostenlosen C-Compiler (**AVR-GCC**), einen kostenlosen Pascal-Compiler (...) und einen kostenpflichtigen BASIC-Compiler (**BASCOM AVR**).

Wozu eine Simulation?

Die Zeit, welche zur Fehlersuche in einem Programm i.A. notwendig ist, kann die Codierzeit bei weitem übersteigen. Hier spart man sich mit einem Simulator, der ein Abbild des verwendeten µC enthält, sicher einiges an Zeit. Simulation ist vor allem dann sinnvoll, wenn die Abfolge dessen, was der µC im Hinblick auf die Anwendung machen soll, noch nicht ganz ausgereift ist (generate and test).

2.2 AVR-ARCHITEKTUR

Die AVR-µC-Familie von Atmel weist eine **RISC**-Architektur (**R**educed **I**nstruction **S**et **C**omputer) auf. Im Gegensatz zu anderen µC, welche einen Bus für Daten und Befehle verwenden (von Neumann-Architektur), benutzen die AVRs für Daten und Befehle getrennte Busse und Speicher (Harvard-Architektur), wodurch verschiedene Busbreiten ermöglicht werden.[2]

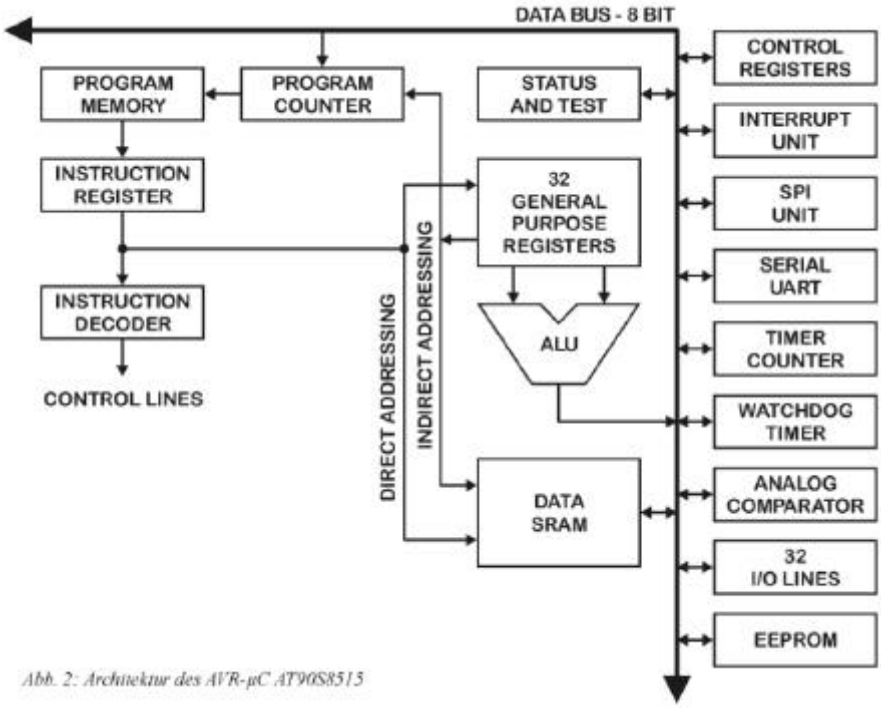


Abb. 2: Architektur des AVR-µC AT90S8515

Eigenschaften des verwendeten AVR-µC AT90S8515:

- 118 Befehle (benötigen zumeist nur einen Taktzyklus zur Abarbeitung)
- Taktfrequenz 4 MHz bzw. 8 MHz (extern eingespeist)
- 8-Bit breiter Datenbus
- 16-Bit breiter Bus für Befehle
- 32 Universalregister (General Purpose Register)
- Direkte, indirekte und relative Adressierung
- 2 Timer/Counter (8-Bit bzw. 16-Bit mit PWM) mit programmierbaren Vorteiler
- 8 kByte internes FLASH für Programme
- 512 Byte SRAM
- 512 Byte internes EEPROM
- 32 programmierbare Ein-/Ausgabe-Pins
- Programmierbarer serieller UART
- Serielle Master/Slave SPI Schnittstelle
- Integrierter Analog-Comparator
- Externe und interne Interruptabarbeitung
- Programmierbarer Watchdog-Timer mit einem am Chip integrierten Oszillator
- Programmierbare Sicherung gegen das Auslesen des Programmcodes

Da die meisten Befehle nur einen Taktzyklus benötigen (16-Bit-Befehle) erreicht der **AVR AT90S8515** mit 4 MHz nahezu 4 **MIPS** (**M**ega-**I**nstructions-**P**er-**S**econd) und jener mit 8 MHz nahezu 8 MIPS.

[2] Vgl. Volpe u.a., S. 12.

2.3 PIN-BELEGUNG

Den **AVR AT90S8515** gibt es in den Gehäusevarianten **TQFP** für die SMD-Technik, **PLCC** zum Einsetzen in einen quadratischen IC-Sockel und die Dual-Inline-Package-Variante (**PDIP**).

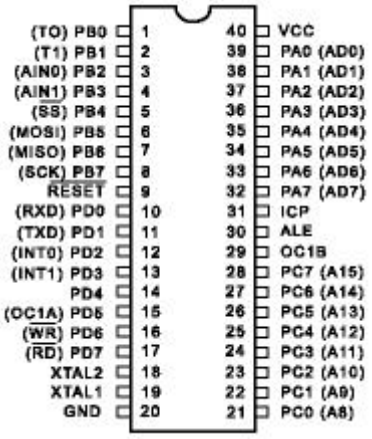


Abb. 3: Pin-Configuration des AT90S8515 (PDIP-Gehäuse)

Beschreibung:

VCC	Versorgungsspannung (2,7 .. 6,0V DC)
GND	**G**rou**nd**
RESET	**Reset**-Input. Ein LOW-Signal an diesem Pin, welches eine Dauer von mindestens von 50ns aufweist, generiert einen Reset des µC.
XTAL1	Anschluss des externen Oszillatorquarzes.
XTAL2	Anschluss des externen Oszillatorquarzes.
ICP	**I**nput-**C**apture-**P**in für Timer/Counter 1.
OC1B	**O**utput-**C**ompare**B**-Pin – PWM-Mode des Timer/Counter **1**.
ALE	**A**dress-**L**atch-**E**nable – wird benötigt, wenn externer Speicher verwendet wird.

PA7 – PA0, PB7 – PB0, PC7 – PC0, PD7 – PD0:

Der **AVR AT90S8515** hat 32 programmierbare bidirektionale I/O-Linien, die in vier Ports (PA-PD) zu je acht Pins (0-7) zusammengefasst sind. Jede dieser I/O-Linien kann entweder als Eingangs- oder als Ausgangsleitung programmiert werden. Jeder dieser Pins kann bei Verwendung als Ausgangsleitung mit 20mA belastet werden, d.h. dass LED-Displays direkt angesteuert werden können.

Einige der Pins werden alternativ dazu auch für spezielle Funktionen verwendet, so kann der µC z.B. über PortA (AD7-AD0) und PortC (A15-A8) einen externen Speicher (SRAM) ansteuern.

Die Zweitbelegungen der Pins weisen auf die Verwendung durch spezielle Funktionen des AVR hin. Hier eine Übersicht:

T0	**(PB0)**	**T**imer/Counter **0** – External Counter Input: Eingang für Zeitgeber/Zähler 0
T1	**(PB1)**	**T**imer/Counter **1** – External Counter Input: Eingang für Zeitgeber/Zähler 1
AIN0	**(PB2)**	**AC** (**A**nalog **C**omparator) – positive **A**nalog **In**put **0** (AC+): positiver analoger Eingang des Analog-Vergleichers
AIN1	**(PB3)**	**AC** (**A**nalog **C**omparator) – negative **A**nalog **In**put **1** (AC–): negativer analoger Eingang des Analog-Vergleichers
SS	**(PB4)**	**SPI** (**S**erial **P**eripheral **I**nterface) – **S**lave **S**elect
MOSI	**(PB5)**	**SPI** (**S**erial **P**eripheral **I**nterface) – Bus **M**aster **O**ut **S**lave **I**n
MISO	**(PB6)**	**SPI** (**S**erial **P**eripheral **I**nterface) – Bus **M**aster **I**n **S**lave **O**ut
SCK	**(PB7)**	**SPI** (**S**erial **P**eripheral **I**nterface) – Bus **S**erial **C**loc**k** (Taktgeber)

RXD (PD0) **U**ART (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter): **R**eceive **D**ata – Eingang der asynchronen seriellen Schnittstelle zum Empfangen von Daten

TXD (PD1) **U**ART (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter): **T**ransmit **D**ata – Ausgang der asynchronen seriellen Schnittstelle zum Senden von Daten

INT0 (PD2) External **Int**errupt **0** Input: Eingang für einen externen Interrupt

INT1 (PD3) External **Int**errupt **1** Input: Eingang für einen externen Interrupt

OC1A (PD5) **O**utput-**C**ompare**A**-Pin des Timer/Counter **1**.

$\overline{\text{WR}}$ (PD6) **Wr**ite Strobe to external Memory

$\overline{\text{RD}}$ (PD7) **R**ea**d** Strobe to external Memory

AD7 – AD0 (PA7-PA0):
Ausgang: Speicheradressenansteuerung (Lower-Byte: A7-A0) bei Verwendung eines externen Speichers (SRAM) gemultiplext mit Eingang/Ausgang: Daten (D7-D0) von und zur zuvor angesteuerten Speicherstelle im externen SRAM.

A15 – A8 (PC7-PC0):
Ausgang: Speicheradressenansteuerung (Higher-Byte: A15-A8) bei Verwendung eines externen Speichers (SRAM).

2.4 SPEICHER

2.4.1 PROGRAMMSPEICHER

Der Programmspeicher ist als Flash ausgeführt und 16-Bit breit. Flash-Speicher bieten den Vorteil, dass sie elektrisch programmierbar- und elektrisch wieder löschbar sind. Die Programmierung dieses Programmspeichers kann parallel über die Anschlüsse des PortB (PB7-PB0) oder seriell über die SPI-Schnittstelle vorgenommen werden.

Die Adressierung dieses Programmspeichers erfolgt linear von der Adresse **$000** bis zur Endadresse **$FFF**. Es besteht die Möglichkeit, Daten bzw. Konstanten in eine Tabelle abzulegen, die sich im Programmspeicher befindet.

2.4.2 DATENSPEICHER (SRAM)

Der Datenspeicher des **AVR AT90S8515** hat eine Breite von 8-Bit und teilt sich in von Registern belegten Speicher, internen SRAM und optional externen SRAM. Der Adressraum des Datenspeichers beginnt bei der Adresse **$0000** und endet bei der Adresse **$FFFF**. An den Adressen **$00** (**$0000**) bis **$1F** (**$001F**) sind die 32 Universalregister und an den Adressen **$20** (**$0020**) bis **$5F** (**$005F**) die I/O-Register bzw. Speicherstellen in den Datenspeicher gespiegelt. Das interne SRAM, welches auch zur Aufnahme des Stacks (Stapels) dient, beginnt dann im Anschluss daran ab der Adresse **$60** (**$0060**) und hat einen Umfang von 512 Bytes.

Universal-register	Daten-speicher
R0	$0000
R1	$0001
R2	$0002
...	...
R29	$001D
R30	$001E
R31	$001F

I/O-Register	Daten-speicher
$00	$0020
$01	$0021
$02	$0022
...	...
$3D	$005D
$3E	$005E
$3F	$005F

Internes SRAM
$0060
$0061
$0062
...
$025D
$025E
$025F

Externes SRAM
$0260
$0261
$0262
...
$FFFD
$FFFE
$FFFF

Tabelle 1: Übersicht Datenspeicheradressierung

An das interne SRAM kann optional ein externes SRAM anschließen, welches beim **AVR AT90S8515** ab der Adresse **$0260** angesprochen wird. Die Gesamtgröße des externen SRAM ergibt sich aus der Differenz der maximal möglichen Adresse (**$FFFF**) und der ersten adressierbaren Speicherstelle des externen SRAM.

2.4.3 EEPROM

Das interne EEPROM des **AVR AT90S8515** befindet sich in einem von Programm- und Datenspeicher getrennten Adressraum. Es kann über spezielle Register (**EEARL**, **EEDR** und **EECR**) des AVR angesprochen werden. Weitere Informationen dazu sind der technischen Beschreibung der Firma Atmel und diverser Fachliteratur zu entnehmen.

2.5 REGISTER

2.5.1 UNIVERSALREGISTER (GENERAL-PURPOSE-REGISTER)

Der **AVR AT90S8515** besitzt 32 Universalregister (General Purpose Register – R0 bis R31), im folgenden einfach als Register bezeichnet, die alle mit dem Herzstück des µC, der **ALU** (**A**rithmetic-**L**ogical-**U**nit), verbunden sind. Alle registerorientierten Befehle des AVR haben dadurch einen direkten Zugriff auf diese Register und benötigen zur Abarbeitung zumeist nur einen einzigen Taktzyklus.

7 0	Adresse	
R0	$00	
R1	$01	
R2	$02	
...	...	
R13	$0D	
R14	$0E	
R15	$0F	
R16	$10	
R17	$11	
...	...	
R26	$1A	X-Register LOW-Byte
R27	$1B	X-Register HIGH-Byte
R28	$1C	Y-Register LOW-Byte
R29	$1D	Y-Register HIGH-Byte
R30	$1E	Z-Register LOW-Byte
R31	$1F	Z-Register HIGH-Byte

Tabelle 2: Universalregister des AVR

Diese Register können auch über den Adressraum des Datenspeichers angesprochen werden.

Zu beachten ist, dass alle Befehle, die als Argumente ein Register und einen unmittelbaren Wert haben (z.B. **SBCI, SUBI, CPI, ANDI, ORI, LDI**) nur auf die oberen Register (**R16** bis **R31**) angewendet werden können.

Besondere Bedeutung haben die Register R26 bis R31. Das Registerpaar R26/R27 bildet das 16-Bit breite X-Register, R28/R29 das 16-Bit breite Y-Register und R30/R31 das 16-Bit breite Z-Register.

2.5.2 I/O-REGISTER

Alle Ports und Peripherie-Module sind im I/O-Adressraum **$00** bis **$3F** abgelegt. Diese 64 I/O-Register können über die Befehle IN (Daten von einem Universalregister zu einem I/O-Register) und OUT (Daten von einem I/O-Register zu einem Universalregister) angesprochen werden. Will man die I/O-Register über die Adressierung im Datenspeicher ansprechen, so muss man zur jeweiligen I/O-Adresse den Wert **$20** addieren.

Einzelne Bits der I/O-Register **$00** bis **$1F** können über spezielle Befehle gesetzt (**SBI**) oder gelöscht (**CBI**) werden.

2.5.3 STATUSREGISTER

Eines der wichtigsten I/O-Register ist das Statusregister (SREG), welches Auskunft über den Status der **CPU** (**C**entral **P**rocessing **U**nit) und der **ALU** (**A**rithmetical **L**ogical **U**nit) gibt. Im Statusregister hat jedes Bit eine bestimmte Bedeutung und wird zumeist Flag genannt. Diese Bits werden von Vergleichs- und Rechenoperationen gesetzt und rückgesetzt und anschließend für Entscheidungen und Verzweigungen im Programm verwendet.

Wichtig ist, dass beim Aufruf einer Interrupt-Routine das Statusregister nicht automatisch gesichert wird und bei der Rückkehr aus der Interrupt-Routine nicht automatisch wiederhergestellt wird. Dies muss im Programm vom Programmschreiber selbst organisiert werden.

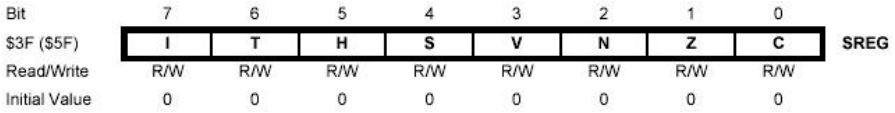


Abb. 4: Statusregister (SREG)

Bit	Flag	Bedeutung
7	**I**	Global Interrupt enable: Dieses Flag muss auf 1 gesetzt werden, wenn man mit Interrupts verwenden will. Die indiviuellen Einstellungen zu den einzelnen Interrupts geschieht in eigenen Kontrollregistern. Wird dieses Flag auf 0 gesetzt, werden Interrupts unabhängig von den Einstellungen in den diversen Kontrollregistern gesperrt.
6	**T**	Bit Copy Storage: Dieses Flag dient als Zwischenspeicher für diverse Bitoperationen wie z.B. die Bit-Copy-Befehle BLD und BST.
5	**H**	Half Carry Flag: Das Half-Carry-Flag wird auf 1 gesetzt, wenn gewisse arithmetische Befehle einen halben Übertrag zur Folge hatten.
4	**S**	Sign Flag: Das S-Flag wird bei arithmetischen oder logischen Befehlen verwendet, die das Zweierkomplement benötigen, und ist mit dem V-Flag in Zusammenhang. Diese beiden Flags sind nie gleichzeitig auf 0 oder gleichzeitig auf 1 (Exklusiv-Oder).
3	**V**	Two’s complement overflow-Flag: Das V-Flag wird bei arithmetischen oder logischen Befehlen verwendet, die das Zweierkomplement benötigen, und ist mit dem S-Flag in Zusammenhang. Diese beiden Flags sind nie gleichzeitig auf 0 oder gleichzeitig auf 1 (Exklusiv-Oder).
2	**N**	Negative Flag: Das Negative-Flag wird auf 1 gesetzt, wenn das Ergebnis eines arithmetischen oder logischen Befehls negativ ist.
1	**Z**	Zero Flag: Das Zero-Flag wird auf 1 gesetzt, wenn das Ergebnis eines arithmetischen oder logischen Befehls Null ist oder ein Vergleich keinen Unterschied zwischen den verglichenen Werten ergab.
0	**C**	Carry Flag: Das Carry-Flag wird auf 1 gesetzt, wenn bei der Ausführung eines arithmetischen oder logischen Befehls ein Übertrag stattgefunden hat. Im gegenteiligen Fall wird das Carry-Flag auf 0 gesetzt.

Tabelle 3: Statusregister (SREG)

Anmerkung zum I-Flag (Global Interrupt enable):
Wird aufgrund des Auftretens eines Interrupts eine Interrupt-Routine (Interrupt-Unterprogramm) aufgerufen, so wird dieses Flag automatisch auf **0** gesetzt und damit andere Interrupts gesperrt. Beim Verlassen der Interrupt-Routine durch den Befehl **RETI** wird dieses Flag dann wieder auf **1** gesetzt und Interrupts wieder erlaubt. Während der Abarbeitung einer Interrupt-Routine sind also alle Interrupts unabhängig von den Einstellungen der verschiedenen Kontrollregister gesperrt. Interrupts können nur nacheinander abgearbeitet werden.

Die Verwendung der einzelnen Flags hängt immer von den benutzen Befehlen ab. Genauere Informationen zu den einzelnen Flags und ihrer Verwendung sind der jeweiligen Befehlsbeschreibung zu entnehmen.

2.5.4 PROGRAMMZÄHLER (PROGRAM COUNTER)

Der Programmzähler ist ein Zeiger, der auf jene Stelle im Programmtext hinweist, die gerade abgearbeitet wird. Diese Abarbeitung erfolgt in der Regel sequentiell, d.h. Zeile für Zeile bzw. Befehl für Befehl. Bei einem Sprungbefehl wird der Programmzähler mit der Programmadresse des Sprungzieles geladen.

Gelangt der Programmablauf an eine Stelle, an der eine Subroutine (Unterprogramm) aufgerufen wird, so wird der um eins inkrementierte Inhalt des Programmzählers automatisch am Stapel abgelegt und mit der Einsprungadresse des Unterprogramms geladen. Danach folgt nacheinander die Abarbeitung der einzelnen Befehle dieses Unterprogramms, wobei der Programmzähler dabei natürlich mitwandert. Ist das Unterprogramm abgearbeitet, so wird die zuvor am Stapel gesicherte Rücksprungstelle in den Programmzähler geladen und der Programmablauf dort fortgesetzt. Die Rücksprungstelle ist dabei jener Befehl, der im Programmtext dem Befehl des Unterprogrammaufrufes folgt. Dies ist auch der Beweggrund für die vorhin durchgeführte Inkrementierung.

Der Aufruf und die Abarbeitung einer Interrupt-Routine erfolgt prinzipiell gleich wie bei einer Subroutine, jedoch mit einem kleinen Unterschied. Da der Aufruf einer solchen Routine nicht aus dem Programmablauf geschieht, sondern wie der Name schon sagt, von einem Interrupt ausgelöst wird, kann dieser Aufruf an verschiedenen Stellen im Programm passieren. Es wird der gerade bearbeitete Befehl fertig ausgeführt und danach mit der Abarbeitung der Interrupt-Routine begonnen.

2.5.5 STAPEL (STACK)

Der Stapel bzw. Stack ist ein Speicher, in dem Daten und insbesondere die Rücksprungadresse beim Aufruf eines Unterprogramms oder einer Interrupt-Routine abgelegt werden. Die organisatorische Abwicklung des Speicherns von Daten auf den Stapel und das spätere Lesen der Daten ist am **LIFO**-Prinzip (**L**ast **i**n – **F**irst **o**ut) orientiert. Dies hat zur Folge, dass jene Daten, welche zuletzt auf den Stapel gelegt werden, als erstes wieder von diesem gelesen werden. Ein sogenannter Stapelzeiger (Stack-Pointer) zeigt dabei immer auf die „oberste“ Stelle im Stapel.

Bei der Verwendung von Unterprogrammen ist es notwendig, die Rücksprungadresse ins Hauptprogramm zu sichern. Der Unterprogrammaufruf **RCALL** kopiert den um eins erhöhten Inhalt des Programmzählers auf die oberste Ebene des Stapels und der Stapelzeiger zeigt fortan auf diese Ebene. Nach dem Rücksprung aus dem Unterprogramm wird der Programmzähler mit dem Inhalt (Programmzählerinhalt zum Zeitpunkt des Aufrufes des Unterprogramms plus 1) der obersten Ebene des Stapels geladen. Der Stapelzeiger wird wieder auf die vorhergehende Ebene des Stapels zurückgesetzt. Da der Inhalt des Programmzählers eine Breite von 16-Bit aufweist, werden also für den Fall eines Unterprogrammaufrufes zwei Bytes (Lower-Byte und Higher-Byte)am Stapel abgelegt und der Stapelzeiger um zwei Speicherstellen im 8-Bit breiten Datenspeicher verändert. Gleiches gilt sinngemäß auch für Interrupt-Routinen-Aufrufe.

Der Stapel befindet sich im 8-Bit breiten Datenspeicher und muss vor der Verwendung bzw. vor dem Aufruf von Unterprogrammen, am besten in einer Initialisierungssequenz am Beginn des Programms, bereitgestellt werden. Allgemein ist es üblich, den Stack vom obersten Rand des zur Verfügung stehenden Datenspeichers nach unten gerichtet einzurichten, jedoch kann im Prinzip jede beliebige freie Stelle im Datenspeicher (SRAM) außerhalb der Universal- und I/O-Register verwendet werden.

Der Stapel wächst also von einer höheren Adresse im Datenspeicher zu einer niedrigeren. Die beim Initialisieren des Stapels bzw. Einrichten des Stapelzeigers verwendeten I/O-Register sind SPL (Stack-Pointer-Low) für das Lower-Byte der Speicheradresse des Stapels und SPH für das Higher-Byte der Speicheradresse des Stapels.

Es sei hier erwähnt, dass der Inhalt des Statusregisters sowie der Universalregister beim Aufruf einer Interrupt-Routine nicht automatisch gesichert wird und bei der Rückkehr aus der Interrupt-Routine nicht automatisch wiederhergestellt wird. Dies muss im Programm vom Programmschreiber selbst organisiert werden. Mit dem Befehl PUSH kann ein Byte auf den Stapel gelegt werden und mit dem Befehl POP kann ein Byte wieder vom Stapel geholt werden. Der Stapelzeiger wird dabei nur um eins inkrementiert bzw. dekrementiert, da es sich nur um ein Byte handelt und nicht um zwei, wie dies beim Programmzähler der Fall ist.

2.5.6 RESET

Als Reset wird das Zurücksetzen des µC bzw. des in den µC geladenen Programms bezeichnet. Der Programmzähler wird beim Reset auf Null gesetzt und verweist damit auf die erste Speicherstelle des Programmspeichers, an der die Abarbeitung eines Programms begonnen wird. Sinnvollerweise muss der Programmcode an dieser Stelle beginnen, d.h. der erste Befehl im Programmcode an dieser Stelle stehen. Bei einem Reset werden die Register- und Port-Inhalte zurückgesetzt.

Es gibt mehrere Möglichkeiten einen Reset auszulösen:

- Anlegen der Betriebsspannung an den µC – Beim Start des µC wird über die Hardware ein Reset ausgelöst.
- Reset-Pin – Ein mindestens 50ns andauerndes Low-Signal am Reset-Pin generiert einen Reset des µC.
- Watchdog-Reset – Ist die zuvor programmierte Zeit der Überwachungsuhr (Watchdog-Timer) des µC erreicht, wird ein Reset generiert.
- direkter Sprungbefehl an den Beginn des Programmspeichers – die Register- und Port-Inhalte bleiben dabei erhalten bzw. werden nicht zurückgesetzt.

2.6 TIMER UND COUNTER

Der **AVR AT90S8515** verfügt über einen 8-Bit Timer/Counter (Timer/Counter0) und einen 16-Bit Timer/Counter (Timer/Counter1), wobei jeder der beiden als Zeitgeber (Timer) oder als Zähler (Counter) verwendet werden kann. In dieser Lernunterlage wird die allgemeine Funktionsweise des Timer/Counter0 beschrieben. Weitere Informationen zum Timer/Counter1 möge man der technischen Beschreibung der Firma Atmel und diverser Fachliteratur entnehmen.

2.6.1 8-BIT TIMER/COUNTER0

Der 8-Bit Timer/Counter0 kann mit der Taktfrequenz (**CK**), der durch den Vorteiler (Prescaler) geteilten bzw. gedrosselten Taktfrequenz oder extern getaktet betrieben werden und es ist auch möglich, ihn wieder anzuhalten. Dies geschieht über das Timer/Counter0-Control-Register (**TCCR0**).

Bit	7	6	5	4	3	2	1	0	
$33 ($53)	–	–	–	–	–	CS02	CS01	CS00	TCCR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abb. 5: Timer/Counter0 – Control Register (TCCR0)

Die Bits 7 bis 3 werden beim **AVR AT90S8515** nicht verwendet und sind dem Wert 0 belegt. Die folgende Tabelle zeigt die möglichen Vorteilereinstellungen, die im **TCCR0** vorgenommen werden können.

CS02	CS01	CS00	Beschreibung
0	**0**	**0**	Stop - der Timer/Counter0 wird angehalten
0	**0**	**1**	CK
0	**1**	**0**	CK/8
0	**1**	**1**	CK/64
1	**0**	**0**	CK/256
1	**0**	**1**	CK/1024
1	**1**	**0**	fallende Flanke an externem Pin T0
1	**1**	**1**	steigende Flanke an externem Pin T0

Tabelle 4: Timer/Counter0 – Control Register (TCCR0)

Wird der Timer/Counter0 mit einem externen Takt an **T0** betrieben, so muss diese eine geringere Taktfrequenz aufweisen als die Taktfrequenz (**CK**) des **AVR AT90S8515**, da nur bei einer steigenden Taktflanke der internen Taktfrequenz (**CK**) das Signal an T0 abgetastet wird.

Bit	7	6	5	4	3	2	1	0	
$32 ($52)	MSB							LSB	TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abb. 6: Timer/Counter0 – Counter Register (TCNT0)

Der Zähler des Timer/Counter0 ist als Aufwärtszähler realisiert, der sowohl mit einem Wert beschrieben (vorbesetzt), als auch ausgelesen werden kann. Wird dieses Register beschrieben und der Zähler ist aktiv, so wird mit dem der Schreiboperation folgenden Taktzyklus mit dem Zählen fortgesetzt.

2.6.2 TIMER/COUNTER INTERRUPT MASK REGISTER (TIMSK)

Bit	7	6	5	4	3	2	1	0	
$39 ($59)	TOIE1	OCIE1A	OCIE1B	–	TICIE1	–	TOIE0	–	TIMSK
Read/Write	R/W	R/W	R/W	R	R/W	R	R/W	R	
Initial Value	0	0	0	0	0	0	0	0	

Abb. 7: Timer/Counter – Interrupt Mask Register (TIMSK)

Bit		Bedeutung
7	**TOIE1**	Timer/Counter1 Overflow Interrupt Enable
6	**OCE1A**	Timer/Counter1 Output CompareA Match Interrupt Enable
5	**OCE1B**	Timer/Counter1 Output CompareB Match Interrupt Enable
4	**-**	nicht verwendet
3	**TICIE1**	Timer/Counter1 Input Capture Interrupt Enable
2	**-**	nicht verwendet
1	**TOIE0**	Timer/Counter0 Overflow Interrupt Enable: Wenn dieses TOIE0-Bit und das I-Flag des Statusregisters gesetzt sind, dann ist ein Timer/Counter0 Overflow Interrupt möglich. Tritt beim Timer/Counter0 ein Überlauf auf (Sprung von $FF auf $00 im Zählregister TCNT0), so wird im Timer/Counter Interrupt Flag Register (TIFR) das Flag TOV0 gesetzt und der dazugehörige Interrupt aus der Interrupt-Vektor-Tabelle ($007) ausgeführt.
0	**-**	nicht verwendet

Tabelle 5: Timer/Counter – Interrupt Mask Register (TIMSK)

2.6.3 TIMER/COUNTER INTERRUPT FLAG REGISTER (TIFR)

Bit	7	6	5	4	3	2	1	0	
$38 ($58)	TOV1	OCF1A	OCIFB	–	ICF1	–	TOV0	–	TIFR
Read/Write	R/W	R/W	R/W	R	R/W	R	R/W	R	
Initial Value	0	0	0	0	0	0	0	0	

Abb. 8: Timer/Counter – Interrupt Flag Register (TIFR)

Bit		Bedeutung
7	**TOV1**	Timer/Counter1 Overflow Flag
6	**OCE1A**	Timer/Counter1 Output Compare Flag 1A
5	**OCE1B**	Timer/Counter1 Output Compare Flag 1B
4	**-**	nicht verwendet
3	**TICIE1**	Timer Counter1 Input Capture Flag 1
2	**-**	nicht verwendet
1	**TOV0**	Timer/Counter0 Overflow Flag: Tritt beim Timer/Counter0 ein Überlauf auf (Sprung von $FF auf $00 im Zählregister TCNT0), so wird dieses Flag (TOV0) gesetzt. Dieses Flag wird gelöscht, wenn der zugehörige Interrupt ausgeführt wird. Der Timer/Counter0-Interrupt wird ausgeführt, wenn das I-Flag im SREG, dasTOIE0-Bit im TCCR0 und dieses TOV0-Flag gesetzt sind.
0	**-**	nicht verwendet

Tabelle 6: Timer/Counter – Interrupt Flag Register (TIFR)

2.7 PERIPHERIE

2.7.1 I/O-PORTS

Der **AVR AT90S8515** verfügt über 32 programmierbare bidirektionale I/O-Linien, die in vier Ports (PortA – PortD) zu je acht Pins (0-7) zusammengefasst sind. Jede dieser I/O-Linien kann entweder als Eingangs- oder als Ausgangsleitung programmiert werden. Jeder dieser Pins kann bei Verwendung als Ausgangsleitung mit 20mA belastet werden, d.h. dass LED-Displays direkt angesteuert werden können.

Zur Definition, ob ein Port als **Eingang** oder als **Ausgang** verwendet wird, existiert zu jedem dieser Ports ein sogenanntes **Data-Direction-Register** (**DDRA – DDRD**), welches die Richtung jedes einzelnen Port-Pins angibt.

Werden z.B. alle Bits in PortA auf **Ausgang** geschaltet (DDRA wird auf $FF bzw. 0b11111111 gesetzt), können Daten über das I/O-Register **PORTA** ausgegeben werden und wirken damit an den zugehörigen Pins des PortA. *Da ein gesetztes Bit in einem Port einen Low-Pegel am zugehörigen Pin ergibt und umgekehrt, ist hier äußerste Sorgfalt angebracht.*

Wird ein oder mehrere Bits im PortA mit dem Data-Direction-Register auf **Eingang** geschaltet (DDRA wird auf $00 bzw. 0b00000000 gesetzt), so kann der Zustand an den zugeordneten Pins über das I/O-Register **PINA** ausgelesen werden.

Die Stellenwertigkeit eines Bits im DDRA entspricht dabei immer der Stellenwertigkeit des jeweiligen Bits im I/O-Register PORTA, d.h. dass z.B. Bit 4 im DDRA dem Bit 4 im PORTA bzw. PINA zugeordnet ist. Zu beachten ist, dass die einzelnen Port-Pins alternative Funktionen haben können.

2.7.2 SYNCHRONE SERIELLE SCHNITTSTELLE (SPI)

Die **SPI**-Schnittstelle (**S**erial **P**eripheral **I**nterface) dient zur schnellen synchronen Datenübertragung. Die Datenübertragung wird dabei über die Anschlüsse **MOSI** (**M**aster **O**ut **S**lave **I**n), **MISO** (**M**aster **I**n **S**lave **O**ut), **SCK** (**S**erial **C**loc**k**) und zusätzlich noch über $\overline{\mathbf{SS}}$ (Slave Select) vorgenommen. Diese Anschlüsse sind Zweitbelegungen diverser Pins am PortB des µC.

Der Datentransfer von und zur SPI-Schnittstelle wird byteweise über die I/O-Register **SPDR** (**SP**I **D**ata **R**egister), **SPSR** (**SP**I **S**tatus **R**egister) und **SPCR** (**SP**I **C**ontrol **R**egister) bewerkstelligt. Weitere Informationen dazu sind der technischen Beschreibung der Firma Atmel und diverser Fachliteratur zu entnehmen.

2.7.3 ASYNCHRONE SERIELLE SCHNITTSTELLE (UART)

Zur asynchronen Datenübertragung besitzt der **AVR AT90S8515** einen **UART** (**U**niversal **A**synchronous **R**eceiver & **T**ransmitter), welcher das byteweise Senden und Empfangen von Daten ermöglicht.

Um gleichzeitig senden und empfangen zu können, besitzt der UART einen Empfänger (Receiver) und einen Sender (Transmitter), welche unabhängig voneinander arbeiten können. Die Generierung der Baud-Rate (Übertragungsrate) geschieht durch den UART.

Der UART wird über vier spezielle Register im I/O-Adressraum gesteuert: **UDR** (**U**ART I/O **D**ata **R**egister), **USR** (**U**ART **S**tatus **R**egister), **UCR** (**U**ART **C**ontrol **R**egister) und das **UBRR** (**U**ART **B**aud **R**ate **R**egister). Weitere Informationen dazu sind der technischen Beschreibung der Firma Atmel zu entnehmen.

2.7.4 ANALOG-KOMPARATOR

Der Analog-Komparator vergleicht die Spannungen, welche an den Eingängen AIN0 und AIN1 anliegen. Ist die Spannung am Eingang AIN0 größer als am Eingang AIN1, so wird der Analog-Komparator-Ausgang (**ACO** - **A**nalog-**C**omparator-**O**utput) auf 1 gesetzt.

Dieser Ausgang kann dazu benutzt werden, den Timer/Counter 1 zu triggern, oder um den Analog-Komparator-Interrupt auszulösen. Die Steuerung des Analog-Komparators geschieht über das I/O-Register **ACSR** (**A**nalog-Comparator-**C**ontrol-and-**S**tatus-**R**egister). Weitere Informationen dazu sind der technischen Beschreibung der Firma Atmel und diverser Fachliteratur zu entnehmen.

2.7.5 WATCHDOG-TIMER

Watchdog Timer (WDT) werden verwendet, um zu verhindern, dass sich µCs „aufhängen“. Ist der WDT eingeschaltet, dies wird durch ein Bit im Watchdog Timer Control Register (WDTCR) gesteuert, löst dieser nach einer fest vorgegebenen Zeit im µC einen Reset aus und stellt so wieder einen definierten Zustand her. Mit einem Vorteiler kann Prescaler kann diese vorgegebene Zeit beeinflusst werden. Weitere Informationen zum Watchdog Timer kann man der technischen Beschreibung der Firma Atmel und diverser Fachliteratur entnehmen.

2.7.6 INTERRUPT-HANDLING

Der **AVR AT90S8515** hat zwei 8-Bit breite Interrupt Mask Control Register, GIMSK (General Interrupt Mask Register) und TIMSK (Timer/Counter Interrupt Mask Register).

Tritt ein Interrupt auf, dann wird das I-Bit im Statusregister gelöscht und alle anderen Interrupts gesperrt. Dieses I-Bit wird wieder gesetzt, sobald der aufgetretene Interrupt abgearbeitet wurde und der Befehl RETI zur Rückkehr an die Stelle im Programm zum Zeitpunkt des Aufrufes ausgeführt wurde. Wird ein Interrupt-Flags gesetzt, dann bleibt es bis zur Abarbeitung oder Löschung durch das Programm erhalten. Treten mehrer Interrupts auf, werden sie nacheinander abgearbeitet. Da einem externen Interrupt kein Flag zugeordnet ist, bleibt dieser nur solange in Evidenz, solange jene Bedingung besteht, die zu seinem Auftreten geführt hat.

Die beiden Interrupt-Pins (INT0 und INT1) am **AVR AT90S8515** können im General Interrupt Mask Register (GIMSK) freigegeben werden. Diverse Einstellungen dazu können im Mikrocontroller-Unit Control Register vorgenommen werden. Weitere Informationen zum Interrupt-Handling möge man der technischen Beschreibung der Firma Atmel und diverser Fachliteratur entnehmen.

3. AVR-ASSEMBLER

Die Programmiersprache Assembler entstand um 1950 als Möglichkeit, Maschinenbefehle sprachlich zu formulieren. Der Vorteil dieser Programmiersprache liegt in der hohen Abarbeitungsgeschwindigkeit. Es gibt verschiedene Assemblerdialekte, da praktisch jeder Prozessor bzw. jede Prozessorfamilie einen eigenen Befehlssatz hat. Im folgenden wird die von Atmel verwendete Programmiersprache AVR-Assembler beschrieben.

3.1 PROGRAMMENTWICKLUNG

So wie in der Softwaretechnologie sollte man auch beim Lösen eines Problems mittels AVR-µC sich strukturiert an die Aufgabenstellung herantasten. Dabei steht immer das zu lösende Problem am Beginn einer jeden Überlegung und ob man das Problem mit einem µC überhaupt lösen kann bzw. ob es andere Lösungsmöglichkeiten gibt. Dabei soll auch die Kostenfrage nicht außer Acht gelassen werden. Als Ergebnis dieser Phase der Problemanalyse und Spezifikation erhält man die Aufgabenstellung bzw. die Anforderung an die zur Verfügung stehende Hard- und Software (µC und Assemblerprogramm) und eventueller Testmöglichkeiten.

In der Detailplanung sollte man sich dann Gedanken machen, wie das Problem gemäß der Aufgabenstellung programmtechnisch gelöst wird. Dies beinhaltet quasi die Erstellung der Programmablaufes mit Eingaben, Ausgaben, Verzweigungen, Strukturierung in Unterprogramme für mehrmals benötigte Programmteile, Verwendung von speziellen Fähigkeiten des AVR-µC, Interrupts, etc. „am Papier“. Das Ergebnis diese Phase ist die Programmstruktur, die grafisch oder auch umgangssprachlich darstellbar ist. Dabei soll es gelingen, das zu erstellende Programm in kleine Teileinheiten zu zerlegen, die dann in sich leichter codiert (programmiert) werden können.

Fragestellungen zur Detailplanung:	
***Kommentar*:**	Welcher Sinn steckt hinter einem Befehl?
***Zuweisungen*:**	Mit welchen (Variablen-)Werten soll ein Programm arbeiten?
***Verzweigung*:**	Wie soll auf verschiedene Argumente verschieden reagiert werden?
***Schleifen*:**	Wie oft soll etwas durchgeführt werden?
Ein-/Ausgabe:	Welche Daten sollen woher genommen werden und wohin sollen Ergebnisse geschrieben werden?
***Verarbeitung*:**	Wohin sollen Zwischenergebnisse gespeichert werden?
***Unterprogramme*:**	Welche Aktionen (Programmteile) werden sehr oft gebraucht?
Interrupt-Routinen:	Wie soll auf spezielle Ereignisse reagiert werden?

In der Phase der Programmierung werden dann die am Papier vorhandenen Teilprobleme nacheinander codiert (programmiert), wobei einerseits immer wieder Tests durchgeführt werden sollen, um sicherzugehen, dass das bisher erstellte Programm auch abforderungsgemäß arbeitet und andererseits laufend dokumentiert werden, was gerade bei der Assemblerprogrammierung immens wichtig ist. Ist das Programm fertig, so kann der damit programmierte AVR im Feld getestet werden.

Warum überhaupt Überlegungen dieser Art?
Je später man in einem Projekt auf einen Fehler gedanklicher oder programmtechnischer Natur stößt, desto zeit-, nerven- und auch kostenintensiver ist die Fehlersuche und Fehlerbehebung.

3.2 SPRACHKONSTRUKTE

Assemblerprogramme werden in normalen ASCII-Textdateien gespeichert. Diese Textdateien nennt man Programmtext, Quelltext oder auch Source-Code und haben die Dateierweiterung ***„asm“*** (z.B. ***„datei.asm“***). Um Assemblerprogrammtextdateien erstellen bzw. bearbeiten zu können ist ein Texteditor notwendig, der keinerlei spezielle Formatierungszeichen, wie dies z.B. in einem Winword-Dokument der Fall ist, speichert, sondern nur den reinen Programmtext. Die Länge einer Zeile des Programmtextes soll im AVR-Assembler 120 Zeichen nicht übersteigen. AVR-Studio von Atmel beinhaltet einen solchen Texteditor und bietet außerdem noch einen Simulator, mit dem erstellte Assemblerprogramme virtuell getestet werden können.

Die Programmiersprache AVR Assembler ist relativ einfach aufgebaut und besteht aus **Befehlen** (instructions), **Argumenten** (operands), **Anweisungen** (directives), **Sprungmarken** (labels) und **Kommentaren** (comments).

Befehle: Die Assemblerbefehle sind das Herz dieser Programmiersprache. Sie geben an, welche Aktion das Programm an der Stelle ihres Vorkommens tun soll (z.B. ***„ADD r17, 0x1B“***). Oft werden diese Befehle ***„Mnemonics“*** genannt.

Argumente: Befehle oder Anweisungen sind oft Argumente beigestellt. Argumente können Register, Konstanten, Sprungmarken oder ähnliches sein (z.B. ***„ADD r17, 0x1B“*** oder ***„.INCLUDE „8515def.inc“***) oder ***„RJMP sprungmarke“***).

Anweisungen: Sie weisen den Compiler (Übersetzer von Programmtext in ausführbaren Maschinencode) an, etwas auszuführen (z.B. ***„.INCLUDE „8515def.inc“***), beginnen mit einem Punkt und können beigestellte Argumente haben. In der Literatur werden Anweisungen auch Direktiven genannt.

Sprungmarken: Jede Zeile kann mit einer Sprungmarke, welche alphanumerisches Format (Buchstaben und Zahlen) haben muss und mit einem Doppelpunkt beendet wird, beginnen (z.B. ***„sprungmarke:“***). Sprungmarken können vom Programmtext mit einem Sprungbefehl angesteuert werden (z.B. ***„RJMP sprungmarke“***).

Kommentartext: Jeder Kommentartext beginnt mit einem Semikolon und endet mit dem Ende der Zeile (***EOL-End of Line***), in der er steht. Kommentartext (z.B. ***„;Kommentar“***) wird vom Compiler nicht in Maschinencode übersetzt, dient der Beschreibung des Programmtextes und erhöht damit dessen Lesbarkeit.

AVR Assembler ist nicht ***„case-senisitive“***, d.h. die Elemente dieser Programmiersprache können klein oder groß geschrieben werden. Es ist jedoch für die Lesbarkeit eines Assembler-Programms von Vorteil, wenn innerhalb des Programmtextes (gilt auch für eingebundene Dateien) eine gewisse Einheitlichkeit gegeben ist, d.h. Befehle und Anweisungen großgeschrieben, Sprungmarken kleingeschrieben.

Eine Assemblerzeile kann folgende Formate haben:

```
[Sprungmarke:] Anweisung [Argumente] [Kommentar]
```

oder

```
[Sprungmarke:] Befehl [Argumente] [Kommentar]
```

oder

```
Kommentar
```

oder

```
[leere Zeile]
```

Elemente, die hier in eckigen Klammern dargestellt werden, sind optional, d.h. sie können aber sie müssen nicht angegeben werden. Ob einer Anweisung oder einem Befehl Argumente beigestellt werden müssen, ist der jeweiligen Befehls- bzw. Anweisungssyntax zu entnehmen.

Es gibt zwar kein fixes Programmtextlayout im Assembler, jedoch ist es aus Gründen der leichteren Lesbarkeit unerlässlich, innerhalb eines Programmtextes ein einheitliches, spaltenorientiertes Format einzuhalten. Dieses Format kann vom Programmhersteller frei gewählt werden.

1. Spalte: Sprungmarken
2. Spalte: Anweisungen oder Befehle (mit den dazugehörigen Operanden)
3. Spalte: Kommentartext

Beispiel:

```
.INCLUDE  "8515def.inc"         ;Definitionsdatei einbinden

          LDI r16, 0xFF         ;0xFF ins Arbeitsregister r16 laden
          OUT DDRB, r16         ;Inhalt von r16 ins I/O-Register DDRB ausgeben

          LDI r16, 0b11111100   ;0b11111100 in r16 laden
          OUT PORTB, r16        ;r16 ins I/O-Register PORTB ausgeben

ende:     RJMP ende             ;Sprung zur Marke "ende" -> Endlosschleife
```

Der Kommentartext sollte aussagekräftig sein, d.h. mit möglichst wenigen Worten soll das Geschehen in der Zeile beschrieben werden. Man soll sich schon bei kleinen Programmen die Mühe machen, den Programmtext zu beschreiben, da mit zunehmender Länge eines unkommentierten Programmtextes die Lesbarkeit dessen immens sinkt. AVR Studio bietet den Vorteil, dass sowohl Befehle, als auch Kommentartext automatisch jeweils in einer anderen Farbe dargestellt werden als z.B. Sprungmarken oder Argumente (Syntax-Highlighting).

3.3 KONTROLLSTRUKTUREN

Strukturiertes Programmieren ist auch im Assembler möglich bzw. notwendig. In der Praxis hat ein AVR-Assemblerprogramm eine Vielfalt an Befehlen, die abgearbeitet werden. Diese Abarbeitung geschieht sequentiell, d.h. Befehl für Befehl bzw. Zeile für Zeile. Es kommt jedoch häufig vor, dass Programmteile mehrmals hintereinander ausgeführt werden müssen (Schleifen) oder im Programmtext auf Ereignisse unterschiedlich reagiert werden muss, d.h. an unterschiedliche Stellen im Programmtext verzweigt wird. Zu den Kontrollstrukturen eines Assemblerprogramms gehören auch Unterprogramme i.A. und Interrupt-Service Routinen im Speziellen.

3.3.1 VERZWEIGUNGEN

Bezeichnend für Verzweigungen sind die Fragestellungen ***„WENN ereignis DANN machwas“*** oder ***„WENN ereignis DANN machwas SONST machwasanderes“***. Fast jeder Sprungbefehl ist ein Hinweis auf eine solche Fragestellung.

3.3.2 SCHLEIFEN

Schleifen erkennt man, dass sie mit einer ***Sprungmarke*** beginnen und irgendwann in den darauffolgenden Befehlen ein Sprungbefehl erkennbar ist, der zu dieser Sprungmarke verzweigt. Eine Schleifenbedingung wäre die Fragestellung ***„Durchlaufe die Schleife solange, bis ein gewisses Ereignis eintritt.“***. Schleifen ohne eine zugehörige Bedingung und ohne einen Befehl zum Verlassen der Schleife nennt man **Endlosschleifen**.

Wird in einer Schleife ein Zähler geführt, d.h. die Anzahl der Schleifendurchläufe gezählt und aufgrund dieser Anzahl die Schleifenbedingung gesetzt (z.B. bis der Zähler einen gewissen Wert erreicht hat), so spricht man von **Zählschleifen**. Wichtig ist bei Zählschleifen die Initialisierung des Zählers.

Schleifen können auch **verschachtelt** (ineinanderliegende Schleifen, wie z.B. eine äußere und eine innere Schleife) verwendet werden. Die Abfolge ist dabei:

- Schleifenbeginn der äußeren Schleife
- Schleifenbeginn der inneren Schleife
- Schleifenende der inneren Schleife
- Schleifenende der äußeren Schleife

Hält man diese Struktur bei verschachtelten Schleifen nicht ein, ist zumeist eine programmtechnische Fehlfunktion die Folge.

3.3.3 UNTERPROGRAMME

Befehlsfolgen, die an mehreren Stellen im Programm benötigt werden, können in sogenannte Unterprogramme verpackt werden. Interrupt-Service-Routinen sind eine speziell verwendete Form von Unterprogrammen und weisen eine ähnliche Struktur auf. Unterprogramme werden oft in externe Programmtextdateien ausgelagert und diese Dateien dann in einem Programm eingebunden.

3.4 AUFBAU EINES ASSEMBLERPROGRAMMS

Am Beginn eines jeden Programmtextes sollen mittels Kommentartext folgende Informationen festgehalten werden: ***Programmtitel (eventuell mit Versionsbezeichnung), Programmdateiname, Autor, Erstellungsdatum, eingebundene Dateien und eine kurze Funktionsbeschreibung.*** Eventuell können auch Angaben zur Version der Programmiersprache und zur Entwicklungsumgebung gemacht werden.

```
;=====================================================================
;Titel:       Testprogramm Version 1.0
;Datei:       test.asm
;Autor:       Hans Hafner
;Datum:       10.2.2004
;Eingebunden: 8515def.inc
;Dies ist ein Testprogramm und soll eine Konstante am PORTB ausgeben.
;=====================================================================
```

Da der AVR Assembler mit allen µC der AVR-Familie funktioniert ist es wichtig, die Entwicklungsumgebung darüber zu informieren, für welchen AVR das betreffende Programm geschrieben wurde. Die Entwicklungsumgebung kennt die Daten der einzelnen µCs und prüft beim Übersetzen des Programmtextes in Maschinencode, ob Befehle etc. vorkommen, die vom verwendeten µC nicht unterstützt werden.

Dies geschieht für den **AVR AT90S8515** mit dem Einbinden der Datei ***„8515def.inc“.*** In dieser Definitionsdatei werden u.a. die einzelnen Registerbezeichnungen den Speicherstellen zugeordnet und der Entwicklungsumgebung mittels Anweisung (Direktive) ***„.DEVICE“*** der verwendete AVR-µC mitgeteilt.

Der übersetzte (compilierte, assemblierte) Programmtext liegt im Programmspeicher an der Adresse $000 (***„.ORG $000“***) und beginnt mit einem Sprung zu jenem Programmteil, in dem Initialisierungen vorgenommen werden. Nach diesem Sprungbefehl steht i.A. die Interrupt-Vektor-Tabelle, die als Einsprungpunkt für Interrupt-Service-Routinen zu auftretenden Interrupts dient.

In der Initialisierungsroutine können u.a. mittels der Anweisung ***„.EQU“*** Konstanten oder Variablen definiert werden. Konstanten sind Bezeichner mit einem vordefinieren Wert. Variablen sind entweder Bezeichner für Speicherstellen im Datenspeicher oder alternative, benutzerdefinierte Bezeichner für verwendete Register.

Beispiel:

```
.EQU    zahl = $0D      ;Dem Bezeichner zahl wird der konstante
                        ;hexadezimale Wert 0D zugewiesen.
.EQU    zeler = r16     ;Dem Universalregister r16 wird der Bezeichner
                        ;zeler zugeordnet.

        LDI zeler, zahl ;Verwendung des alternativen Bezeichners zeler
                        ;für das Register r16 und der vordefinierten
                        ;Konstanten zahl
```

Danach erfolgt üblicherweise ein Sprung an den Beginn des Hauptprogramms. Zwischen dem Sprungbefehl zum Hauptprogramm und der Sprungmarke, die dabei angesprungen wird können auch noch Unterprogramme platziert werden.
Das Hauptprogramm soll jedenfalls als Endlosschleife ausgeführt sein.

4. AVR-STUDIO

Im dem **AVR-Studio** kann man Assembler-Programmtexte editieren (schreiben), assemblieren bzw. compilieren (übersetzen im Maschinencode), übersetzte Programme am PC simulieren bzw. debuggen oder testen. Der Vorteil des **AVR-Studio** liegt in der Funktionalität als Entwicklungs- und Testumgebung.

4.1 ANLEGEN EINES NEUEN PROJEKTES

Beim Starten des **AVR-Studios** erscheint ein Fenster, in dem man entweder ein bereits bestehendes Projekt öffnen oder ein neues Projekt anlegen kann. Beim Anlegen eines neuen Projektes ist darauf zu achten, dass man die richtige **Debug-Plattform** auswählt.

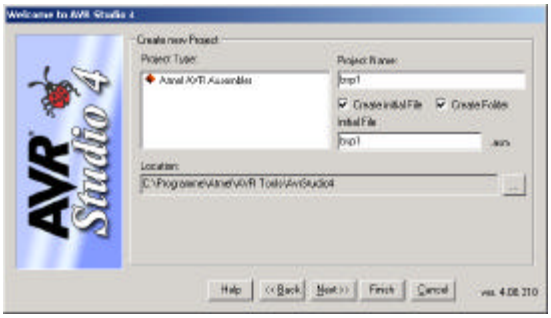


Abb. 9: Anlegen eines Projektes im AVR-Studio

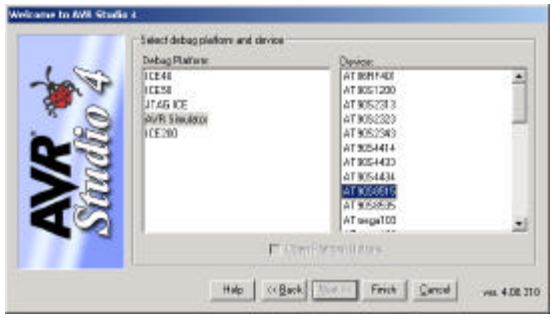


Abb. 10: Auswahl der Debug-Plattform

4.2 ENTWICKLUNGSUMGEBUNG

Die Entwicklungsumgebung des **AVR-Studio** ist prinzipiell wie andere Windowsprogramme aufgebaut und umfasst neben der Menüleiste und der Symbolleiste einen Programmtexteditor (**Source-Window**), den Arbeitsbereich (**Workspace**) und den Ausgabebereich (**Output-Space**).

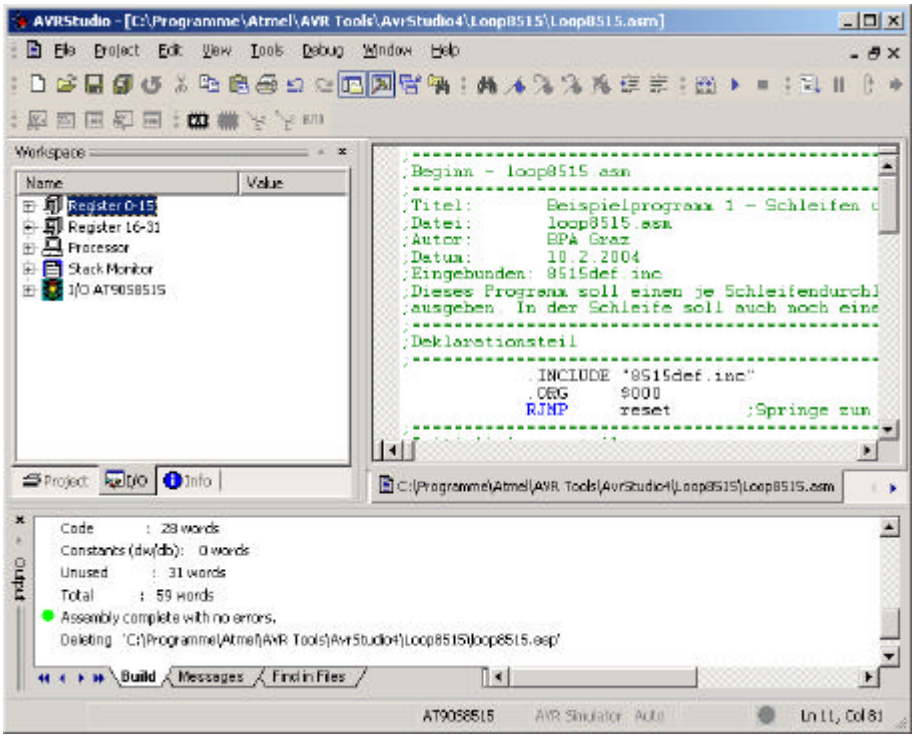


Abb. 11: Entwicklungsumgebung des AVR-Studios

- **Source-Window:** In diesem Fenster ist der Editor untergebracht, mit dem man Programmtext erstellen und editieren kann. Natürlich funktionieren hier auch die Windows-Kopierfunktionen über die Zwischenablage des Betriebssystems, so dass man Programmtexte aus anderen Textdateien mit **Strg+C** bzw. **Strg+V** in dieses Fenster bringen kann. Man kann auch mehrere Programmtexte des gleichen Projekts gleichzeitig offen haben, wobei jeder in einem eigenen Source-Window angezeigt wird.

- **Workspace:** In diesem Bereich hat man mit dem I/O-Window eine Übersicht über alle Universalregister, Programmzähler, Stapelzeiger und alle I/O-Register des AVR. Dies ist beim Testen des Programmes von wesentlichem Vorteil. Es besteht auch die Möglichkeit, in diesem Bereich ein Register-Window einzurichten, in dem eigens nur die Inhalte der Universalregister gesammelt angezeigt werden.

- **Output-Bereich:** In diesem Fenster erscheinen Meldungen, wie z.B. Informationen zum Übersetzungsvorgang des Programmtextes in Maschinencode (Build), wie z.B. Syntaxfehler.

4.3 DEBUGGEN IM AVR-STUDIO

Über die Symbolleiste der Entwicklungsumgebung **AVR-Studio** können die für die Simulation eines Programm benötigten Funktionen komfortabel aufgerufen werden.

BUILD (F7): Der Programmtext wird mitsamt den eingebundenen Dateien in Maschinencode übersetzt und eine Objekt-Datei erzeugt. Eventuelle Syntaxfehler werden im Output-Bereich angegeben.

START DEBUGGING: Der Debug-bzw. Testvorgang wird gestartet und die Kontrollfunktionen aktiviert, die Objekt-Datei geladen und automatisch ein Reset ausgeführt.

STOP DEBUGGING: Der Debug- bzw. Testvorgang wird gestoppt.

RESET (SHIFT+F5): Befindet man sich im Debug-Vorgang, so wird mit dieser Funktion der Ablauf gestoppt und der Debug-Vorgang zurückgesetzt.

RUN (F5): Die Ausführung des übersetzten Programms wird gestartet und läuft bis zur Unterbrechung durch den Benutzer bzw. bis zu einem Haltepunkt (Breakpoint) im Programmtext, den der Benutzer setzen kann, wenn er an einer bestimmten Stelle im Programmablauf z.B. die Registerinhalte einsehen will.

SINGLE STEP / TRACE INTO (F11): Im Source-Window wird ein einzelner Programmschritt ausgeführt. Wird ein Unterprogramm oder eine Interrupt-Routine aufgerufen, so werden mit dieser Funktion auch die Befehlszeilen in diesem Unterprogramm bzw. in der Interrupt-Routine ausgeführt.

 STEP OVER: Diese Funktion führt ein Unterprogramm bzw. eine Interrupt-Routine in einem Stück aus und gibt keinen Einblick in die einzelnen Schritte des aufgerufenen Unterprogramms bzw. der aufgerufenen Interrupt-Routine. Ein vom Benutzer gesetzter Haltepunkt wird jedoch berücksichtigt.

 AUTOSTEP: Diese Funktion arbeitet i.A. wie die Funktion RUN, wobei jedoch die ausgeführten Schritte einzeln visualisiert werden.

Wird der Programmtext während der Testphase verändert, so wird man zu einem neuerlichen Übersetzen des Programmtextes in Maschinencode aufgefordert.

Eine nicht von der Hand zu weisende Unterstützung bietet die Hilfe-Funktion an, die mit der Funktionstaste F1 bzw. via Symbol oder Menüleiste aufgerufen werden kann.

4.4 EXTERNE OBJEKT-DATEIEN

Das AVR-Studio bietet die Möglichkeit, extern in einer anderen Entwicklungsumgebung (C, BASIC, PASCAL) erstellte AVR-Objekt-Dateien zu testen, wobei diese über die Menüleiste aufgerufen werden können.

5. BEISPIELE

5.1 BEISPIEL 1 – SCHLEIFEN / UNTERPROGRAMME

```
;***************************************************************************
;Beginn – loop8515.asm
;***************************************************************************
;Titel:       Beispielprogramm 1 – Schleifen und Unterprogramme
;Datei:       loop8515.asm
;Autor:       BPA Graz
;Datum:       10.2.2004
;Eingebunden: 8515def.inc
;Dieses Programm soll einen je Schleifendurchlauf um eins erhöhten Zähler am PORTB
;ausgeben. In der Schleife soll auch noch eine Bremse (Warteschleife) eingebaut sein.
;***************************************************************************
;Deklarationsteil
;***************************************************************************
            .INCLUDE "8515def.inc"
            .ORG     $000
            RJMP     reset       ;Springe zum Initialisierungsteil
;***************************************************************************
;Initialisierungsteil
;***************************************************************************
            .CSEG
            .ORG     $20
reset:      LDI      r16, $80    ;Stapel vor Unterprogrammaufruf (RCALL)
            OUT      SPL, r16    ;initialisieren !!! Wert $80 frei gewählt

            LDI      r16, $00
            OUT      DDRD, r16   ;alle Bits in PORTD auf Eingang schalten ($00)

            LDI      r16, $FF
            OUT      DDRB, r16   ;alle Bits in PORTB auf Ausgabe schalten ($FF)

            LDI      r16, $FF
            OUT      PORTB, r16  ;PORTB mit $00 initialisieren (invertiert!!!)

            LDI      r17, $00    ;Zählregister mit $00 initialisieren
;***************************************************************************
;Hauptprogramm Beginn
;***************************************************************************
schleife:
            IN       r20, PIND   ;Eingang an PORTD abfragen
            ANDI     r20, $10    ;wurde Taste an PD4 betätigt?
            BRNE     noout       ;Wenn Ja, dann keine Ausgabe (noout) an PORTB

            INC      r17         ;Zählregisterinhalt um eins erhöhen

            MOV      r18, r17    ;Zählregisterinhalt muss zur Ausgabe invertiert
            COM      r18         ;werden, da z.B. beim Wert 0 im I/O-Register PORTB
            OUT      PORTB, r18  ;an den Pins PB0-PB7 der Wert 1 anliegen würde

noout:      RCALL    warte       ;Aufruf des Unterprogramms - Warteschleife
            RJMP     schleife    ;das Ganze immer wieder
;***************************************************************************
;Hauptprogramm Ende
;***************************************************************************
;Unterprogramm Beginn
;***************************************************************************
warte:                           ;Warteschleife als verschachtelte Schleife
            LDI      r18, $02    ;äußeren Schleifenzähler (r18) initialisieren auf $02
außen:      LDI      r19, $02    ;inneren Schleifenzähler (r19) initialisieren auf $02
innen:      DEC      r19         ;inneren Schleifenzähler um 1 verringern
            CPI      r19, 0      ;Vergleich - bis der Wert 0 erreicht ist zurück
            BRNE     innen       ;zum Anfang der inneren Schleife, sonst fertig
            DEC      r18         ;äußeren Schleifenzähler um 1 verringern
            CPI      r18, 0      ;Vergleich – bis der Wert 0 erreicht ist zurück
            BRNE     außen       ;zum Anfang der äußeren Schleife, sonst fertig
            RET                  ;Rücksprung zur aufrufenden Stelle im Hauptprogramm
;***************************************************************************
;Unterprogramm Ende
;***************************************************************************
;Ende – loop8515.asm
;***************************************************************************
```

5.2 BEISPIEL 2 – TIMER UND INTERRUPTS

```
;***************************************************************************
;Beginn – irq8515.asm
;***************************************************************************
;Titel:       Beispielprogramm 2 – Timer und Interrupts
;Datei:       irq8515.asm
;Autor:       BPA Graz
;Datum:       10.2.2004
;Eingebunden: 8515def.inc, UProgs.asm
;Dieses Programm soll einen durch einen Timer-Interrupt gesteuerten Sekunden-Zähler
;am PORTB ausgeben.
;***************************************************************************
;Deklarationsteil
;***************************************************************************
            .INCLUDE "8515def.inc"
            .ORG     $000
            RJMP     reset       ;Springe zum Initialisierungsteil
            RETI                 ;INT0      +----------------------------------------
            RETI                 ;INT1      |
            RETI                 ;T1CAP     |  Interrupt-Vektor-Tabelle am Beginn des
            RETI                 ;T1COMPA   |  Programmspeichers bei der Verwendung
            RETI                 ;T1COMPB   |  von Interrupts zur Sicherheit mit RETI
            RETI                 ;T1OVF     |
            RJMP     T0OVF       ;T0OVF - Interrupt-Vektor Einsprung ($007)
            RETI                 ;SPI       |
            RETI                 ;URX       |  versehen, damit ein unabsichtlich
            RETI                 ;UDRE      |  verwendeter Interrupt ins Leere läuft.
            RETI                 ;UTX       |
            RETI                 ;ANCOMP    +----------------------------------------
;***************************************************************************
;Initialisierungsteil
;***************************************************************************
            .CSEG
            .ORG     $20
reset:      LDI      r16, $80    ;Stapel vor Unterprogrammaufruf (RCALL)
            OUT      SPL, r16    ;initialisieren !!! Wert $80 frei gewählt
            RCALL    init        ;Initialisierungsunterprogramm
;***************************************************************************
;Hauptprogramm - Beginn
;***************************************************************************
loop:
            CPI      r20, 1      ;abfragen, ob T0-Irq-Flag (r20) gesetzt?
            BRNE     weiter      ;wenn nicht gesetzt, dann weiter
            LDI      r20, 0      ;sonst Flag quittieren
            INC      r19         ;Hundertstel-Zähler (r19) um 1 hochzählen
            CPI      r19, 100    ;Sekunde erreicht? (nach 100/100sec)
            BRNE     weiter      ;wenn nicht erreicht, dann weiter
            LDI      r19, 0      ;sonst Hundertstel-Zähler (r19) auf 0 setzen und
            INC      r17         ;Sekundenzähler (r17) hochzählen und ausgeben
            MOV      r18, r17    ;Zählregisterinhalt muss zur Ausgabe invertiert
            COM      r18         ;werden, da z.B. beim Wert 0 im I/O-Register PORTB
            OUT      PORTB, r18  ;an den Pins PB0-PB7 der Wert 1 anliegen würde

weiter:     RJMP     loop        ;das Ganze immer wieder

            .INCLUDE "UProgs.asm"
                                 ;init- und ToVF-IRG-Routine in der Include-Datei
;***************************************************************************
;Hauptprogramm - Ende
;***************************************************************************
;Ende – irq8515.asm
;***************************************************************************
```

5.2.1 UNTERPROGRAMMDATEI ZUM BEISPIEL 2

```
;===============================================================================
;Beginn – UProgs.asm
;===============================================================================
;Titel:       Unterprogrammsammlung zu Beispielprogramm 2 – Timer und Interrupts
;Datei:       UProgs.asm
;Autor:       BPA Graz
;Datum:       10.2.2004
;Eingebunden: -
;Dieses Programm soll einen durch einen Timer-Interrupt gesteuerten Sekunden-Zähler
;am PORTB ausgeben.
;===============================================================================
;Deklarationsteil
;===============================================================================
            .CSEG
;===============================================================================
;init - Unterprogramm - Beginn
;===============================================================================
init:       LDI     r16, $FF
            OUT     DDRB, r16   ;alle Bits in PORTB auf Ausgabe schalten ($FF)

            LDI     r16, $FF
            OUT     PORTB, r16  ;PORTB mit $00 initialisieren (invertiert!!!)

            LDI     r17, $00    ;Zählregister mit $00 initialisieren

            LDI     r16, $05
            OUT     TCCR0, r16  ;Vorteiler für T0 auf CK/1024 ($05) einstellen
            LDI     r16, $DC
            OUT     TCNT0, r16  ;Zeitzähler (TCNT0) für 10ms-Overflow ($DC) vorladen

            IN      r16, TIMSK
            ORI     r16, $02
            OUT     TIMSK, r16  ;Interrupt für Timer0 erlauben und freigeben ($02)

            LDI     r19, 0      ;Hundertstel-Zähler auf 0 initialisieren
            LDI     r17, 0      ;Sekunden-Zähler auf 0 initialisieren

            SEI                 ;Global-Interrupt freigeben (I-Flag/Statusregister)

            RET                 ;Initialisierung fertig - Rücksprung
;===============================================================================
;init - Unterprogramm - Beginn
;===============================================================================
;T0OVF – Interrupt-Service-Routine - Beginn
;===============================================================================
T0OVF:      LDI     r16, $DC
            OUT     TCNT0, r16  ;Zeitzähler (TCNT0) für 10ms-Overflow ($DC) vorladen
            LDI     r20, 1      ;T0-Irq-Flag setzen
            RETI                ;Rückkehr zum Ort zum Zeitpunkt des Interrupteintritts

            .EXIT               ;Ende des zu übersetzenden Programmtextes
;===============================================================================
;T0OVF – Interrupt-Service-Routine - Ende
;===============================================================================
;Ende – UProgs.asm
;===============================================================================
```

5.3 KREATIVE IDEEN

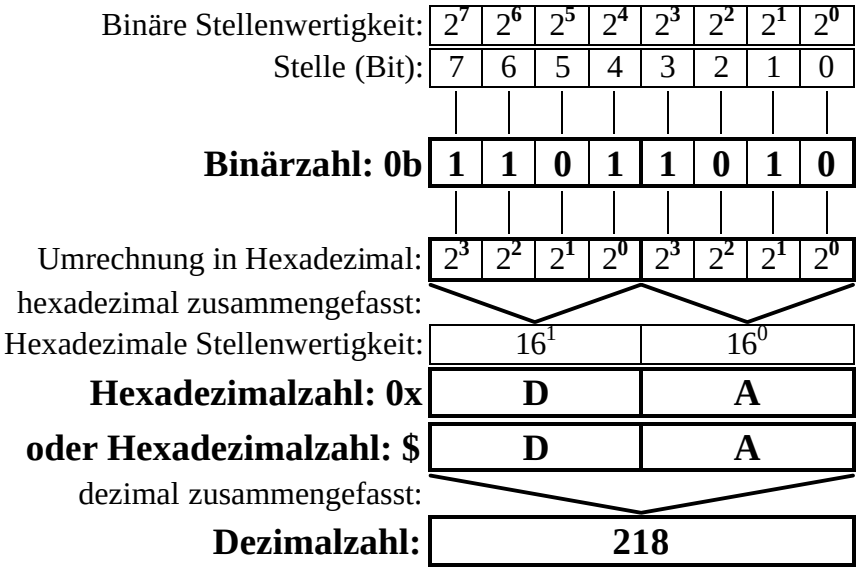
Der **AVR AT90S8515** könnte für folgende Problemstellungen eine Lösung bringen:

- Richtungswechselndes Lauflicht mit 16 LEDs wie in der Serie Knight-Rider
- Timergesteuerte Flughafenbefeuerung mit 32 LEDS
- Schaltpult für LEGO-Motorenansteuerung
- Datenlogger

6. ANHANG

6.1 ZAHLENSYSTEME

Im Assembler werden zumeist Binärzahlen (z.B. ***„0b11011010“***) oder der Einfachheit halber Hexadezimalzahlen (z.B. ***„0xDA“***) verwendet. Hier ein Beispiel:



Aus dieser Grafik ist der Zusammenhang zwischen den verwendeten Zahlensystemen leicht erkennbar. Jedes Bit der Binärzahl steht z.B. für einen Pin an einem I/O-PORT eines AVR, wobei Bit 7 das höchstwertige Bit (MSB-most-significant-bit) und Bit 0 das niederwertigste Bit (LSB-least-significant-bit) ist. Die Bitzahlen 0 bis 7 stehen also für die Wertigkeit des Bits und entsprechen den jeweiligen Hochzahlen der Zweierpotenzen. Dies ist für das Umrechnen zwischen den Zahlensystemen wichtig.

Eine Stelle im Hexadezimalsystem steht für jeweils vier Bit. Die kleinste mit acht Bit darstellbare Binärzahl ist ***„0b00000000“***, dies entspricht der Hexadezimalzahl ***„0x00“*** bzw. ***„$00“*** und der Dezimalzahl ***„0“***. Die größte mit acht Bit darstellbare Binärzahl ist ***„0b11111111“***, dies entspricht der Hexadezimalzahl ***„0xFF“*** bzw. ***„$FF“*** und der Dezimalzahl ***„255“***. Weiterführende Angaben zu den Zahlensystemen können der facheinschlägigen Literatur (z.B. Friedrich: Tabellenbücher) entnommen werden.

6.2 ÜBERSICHT I/O-REGISTER

Die folgende Übersicht zu den I/O-Registern des AVR AT90S8515 ermöglicht eine Abgleich zwischen der I/O-Adresse eines I/O-Registers und dessen gespiegelte Adresse im Datenspeicher. I/O-Adressen, welche nicht in dieser Liste aufscheinen haben entweder keine Funktion oder sind für ein I/O-Register eines anderen AVR reserviert.

I/O-Adresse	Datenspeicher-adresse	I/O-Register	Funktion
$08	$28	ACSR	Analog Comparator Control and Status Register
$09	$29	UBRR	UART Baud Rate Register
$0A	$2A	UCR	UART Control Register
$0B	$2B	USR	UART Status Register
$0C	$2C	UDR	UART I/O Data Register
$0D	$2D	SPCR	SPI Control Register
$0E	$2E	SPSR	SPI Status Register
$0F	$2F	SPDR	SPI I/O Data Register
$10	$30	PIND	Input Pins, Port D
$11	$31	DDRD	Data Direction Register, Port D
$12	$32	PORTD	Data Register, Port D
$13	$33	PINC	Input Pins, Port C
$14	$34	DDRC	Data Direction Register, Port C
$15	$35	PORTC	Data Register, Port C
$16	$36	PINB	Input Pins, Port B
$17	$37	DDRB	Data Direction Register, Port B
$18	$38	PORTB	Data Register, Port B
$19	$39	PINA	Input Pins, Port A
$1A	$3A	DDRA	Data Direction Register, Port A
$1B	$3B	PORTA	Data Register, Port A
$1C	$3C	EECR	EEPROM Control Register
$1D	$3D	EEDR	EEPROM Data Register
$1E	$3E	EEARL	EEPROM Address Register Low Byte
$1F	$3E	EEARH	EEPROM Address Register High Byte (AT90S8515)
$21	$41	WDTCR	Watchdog Timer Control Register
$24	$44	ICR1L	T/C 1 Input Capture Register Low Byte
$25	$45	ICR1H	T/C 1 Input Capture Register High Byte
$28	$48	OCR1BL	Timer/Counter1 Output Compare Register B Low Byte
$29	$49	OCR1BH	Timer/Counter1 Output Compare Register B High Byte
$2A	$4A	OCR1AL	Timer/Counter1 Output Compare Register A Low Byte
$2B	$4B	OCR1AH	Timer/Counter1 Output Compare Register A High Byte
$2C	$4C	TCNT1L	Timer/Counter1 (16-Bit) Low Byte
$2D	$4D	TCNT1H	Timer/Counter1 (16-Bit) High Byte
$2E	$4E	TCCR1B	Timer/Counter1 Control Register B
$2F	$4F	TCCR1A	Timer/Counter1 Control Register A
$32	$52	TCNT0	Timer/Counter0 (8-Bit)
$33	$53	TCCR0	Timer/Counter0 Control Register
$35	$55	MCUCR	MCU general Control Register
$38	$58	TIFR	Timer/Counter Interrupt Flag register
$39	$59	TIMSK	Timer/Counter Interrupt Mask register
$3A	$5A	GIFR	General Interrupt Flag Register
$3B	$5B	GIMSK	General Interrupt Mask register
$3D	$5D	SPL	Stack Pointer Low
$3E	$5E	SPH	Stack Pointer High
$3F	$5F	SREG	Status Register

Tabelle 7: Übersicht I/O-Register

6.3 ANWEISUNGEN BZW. DIREKTIVEN

Was der Assembler ausführen soll, oder was bei der Ausführung zu beachten ist, wird ihm in Anweisungen (Direktiven) mitgeteilt. Die folgende Kurzübersicht mit den wichtigsten Anweisungen soll einen kleinen Einblick gewähren.

Syntax:	**.CSEG**	Beispiel:	**.CSEG**

Diese Anweisung definiert den Start eines Codesegmentes. Ein Codesegment ist ein Speicherbereich, in dem Programmcode (Maschinencode) gespeichert wird. Ein Assembler-Programmtext kann mehrere Codesegmente beinhalten, die aber beim übersetzen in Maschinencode in einem einzigen Codesegment zusammengeschlossen werden, welches sich dann im Programmspeicher befindet. Diese Anweisung hat keine Argumente beigestellt.

Syntax:	**.DEF Bezeichner = Register**	Beispiel:	**.DEF zeler = r16**

Diese Anweisung weist einem Register einem Namen (Bezeichner) zu, unter dem es im Assembler-Programmtext angesprochen werden kann. Einem Register können mehrere Bezeichner zugeordnet werden. Ein mit dieser Anweisung definierter Bezeichner kann im Programmtext umdefiniert werden, d.h. z.B. auf ein anderes Register verweisen.

Syntax:	**.DSEG**	Beispiel:	**.DSEG**

Diese Anweisung definiert den Start eines Datensegmentes. Ein Datensegment ist ein Speicherbereich, in dem mittels Anweisungen z.B. Variablen definiert werden können. Ein Assembler-Programmtext kann mehrere Datensegmente beinhalten, die aber beim übersetzen in Maschinencode in einem einzigen Datensegment zusammengeschlossen werden, welches sich dann im Datenspeicher befindet. Diese Anweisung hat keine Argumente beigestellt.

Syntax:	**.EQU Bezeichner = Wert**	Beispiel:	**.EQU maxanzahl = 68**

Diese Anweisung weist einem konkreten Wert einem Namen (Bezeichner) zu, unter dem es im Assembler-Programmtext angesprochen werden kann. Ein mit dieser Anweisung definierter Bezeichner kann im Programmtext nicht verändert werden.

Syntax:	**.EXIT**	Beispiel:	**.EXIT**

Diese Anweisung weist darauf hin, die Übersetzung des Programmtextes an dieser Stelle zu stoppen und nicht bis zum Ende der Datei zu übersetzen.

Syntax:	**.INCLUDE „dateiname“**	Beispiel:	**.INCLUDE „8515DEF.INC“**

Diese Anweisung weist darauf hin, die angeführte Datei an der angegeben Stelle einzubinden. Eine mittels dieser Anweisung eingebundene Datei kann diese Anweisung ebenfalls enthalten.

Syntax:	**.ORG expression**	Beispiel:	**.ORG $20**

Diese Anweisung setzt den Beginn eines zuvor angegebenen Segmentes fest. In einem Codesegment bezieht sich diese Anweisung auf den Programmspeicher und setzt den PC (Programmzähler) an diese Stelle. In einem Datensegment kann damit dieses in den internen SRAM gelegt werden.

Syntax:	**.SET Bezeichner = Wert**	Beispiel:	**.SET maxanzahl = 68**

Diese Anweisung weist einem konkreten Wert einen Namen (Bezeichner) zu, der später im Programmtext verändert werden kann.

6.4 BEFEHLSSATZ DES AVR AT90S8515

Die Befehle des AVR AT908515 lassen sich im Groben in die Teilbereiche ***„arithmetische und logische Befehle“, „Sprungbefehle“, „Datentransferbefehle“*** und ***„Bitmanipulations- und –inspektionsbefehle“*** unterteilen.

Um die auf den folgenden Seiten Befehlsbeschreibungen leichter lesbar und verstehbar zu machen ist es notwendig, gewisse Konventionen bei den Bezeichnungen einzuhalten.

Bezeichnungen:

Rd	Ziel- und Quellregister (Universalregister)
Rr	Quellregister (Universalregister)
Port	I/O-Adresse (I/O-Register)
addr	konstante Adresse für Programmzähler
maske	Maske für Bitmanipulation
←	Zuweisung
K63	konstanter Wert (**0** bis **63**, **$00** bis **$3F**, **0b000000** bis **0b111111**)
K255	konstanter Wert (**0** bis **255**, **$00** bis **$FF**, **0b00000000** bis **0b11111111**)
K65535	konstanter Wert (**0** bis **65535**, **$0000** bis **$FFFF**)
bit	bit in einem Register (**0** bis **7**)
Rd(7)	**Bit 7** des Registers **Rd**
k	konstante Adresse
PC	Programmzähler, Program-Counter
SP	Stapelzeiger, Stack-Pointer

Logische Verknüpfungen:

.AND.	logische **UND**-Verknüpfung
.OR.	logische **ODER**-Verknüpfung (*inklusives ODER*)
.XOR.	logische **XODER**-Verknüpfung (*exklusives ODER*)

Flags im Statusregister (SREG):

C	Carry-Flag
Z	Zero-Flag
N	Negativ-Flag
V	Zweierkomplement-Overflow-Indikator
S	Signed-Flag (für vorzeichenbehaftete Vergleiche)
H	Half-Carry-Flag
T	Transfer-Bit
I	Global-Interrupt-Enable/Disable-Flag

Eine bei einem ***„beeinflussten“*** Flag (Bit) des Statusregisters in Klammer gesetzte **0** zeigt an, dass das betreffende Flag durch den jeweiligen Befehl **gelöscht** wird, eine in Klammer gesetzte **1** zeigt an, dass das betreffende Flag durch den Befehl **gesetzt** wird.

6.4.1 ARITHMETISCHE UND LOGISCHE BEFEHLE

6.4.1.1 ARITHMETISCHE BEFEHLE

ADC – ADDIERE REGISTER MIT CARRY-FLAG

Syntax:			Funktion:	
ADC *Rd, Rr*			***Rd ← Rd + Rr + C***	
Beschreibung:				
Der Inhalt des Registers **Rr** und das **C**-Flag (Carry-Flag) des Statusregisters werden zum Inhalt des Registers **Rd** addiert. Das Ergebnis der Addition steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

ADD – ADDIERE REGISTER OHNE CARRY-FLAG

Syntax:			Funktion:	
ADD *Rd, Rr*			***Rd ← Rd + Rr***	
Beschreibung:				
Der Inhalt des Registers **Rr** wird zum Inhalt des Registers **Rd** addiert. Das Ergebnis der Addition steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

ADIW – ADDIERE WERT ZUM REGISTERPAAR

Syntax:			Funktion:	
ADIW *Rd+1:Rd, K63*			***Rd+1:Rd ← Rd+1:Rd + K63***	
Beschreibung:				
Der Wert **K63** wird zum Inhalt des Registerpaares **Rd+1:Rd** addiert. Dabei gibt **Rd** das untere der beiden Register an. *(zulässig für **Rd**: **r24, r26, r28** und **r30**)(zulässig für **K63**: **0** bis **63**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z C	**2**	Befehlszeit:	**500 ns**	**250 ns**

SBC – SUBTRAHIERE REGISTER MIT CARRY-FLAG

Syntax:			Funktion:	
SBC *Rd, Rr*			***Rd ← Rd – Rr – C***	
Beschreibung:				
Der Inhalt des Registers **Rr** und das **C**-Flag (Carry-Flag) aus dem Statusregister werden vom Inhalt des Registers **Rd** subtrahiert. Das Ergebnis der Subtraktion steht im Register **Rd**. Der Inhalt des Registers **Rr** wird dabei nicht verändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

SBCI – SUBTRAHIERE WERT UND CARRY-FLAG VON REGISTER

Syntax:			Funktion:	
SBCI *Rd, K255*			***Rd ← Rd – K255 – C***	
Beschreibung:				
Der unmittelbare Wert **K255** und das **C**-Flag (Carry-Flag) aus dem Statusregister werden vom Inhalt des Registers **Rd** subtrahiert. Das Ergebnis der Subtraktion steht im Register **Rd**. *(zulässig für **Rd**: **r16** bis **r31**) (zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

SUB – SUBTRAHIERE REGISTER OHNE CARRY-FLAG

Syntax:			Funktion:	
SUB *Rd, Rr*			***Rd ← Rd – Rr***	
Beschreibung:				
Der Inhalt des Registers **Rr** wird vom Inhalt des Registers **Rd** subtrahiert. Das Ergebnis der Subtraktion steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert. *(zulässig für **Rd, Rr: r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

SUBI – SUBTRAHIERE WERT VOM REGISTER

Syntax:			Funktion:	
SUBI *Rd, K255*			***Rd ← Rd – K255***	
Beschreibung:				
Der unmittelbare Wert **K255** wird vom Inhalt des Registers **Rd** subtrahiert. Das Ergebnis der Subtraktion steht im Register **Rd**. *(zulässig für **Rd: r16** bis **r31**)(zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

SBIW – SUBTRAHIERE WERT VOM REGISTERPAAR

Syntax:			Funktion:	
SBIW *Rd+1:Rd, K63*			***Rd+1:Rd ← Rd+1:Rd + K63***	
Beschreibung:				
Der unmittelbare Wert **K63** wird vom Inhalt des Registerpaares **Rd+1:Rd** subtrahiert, wobei **Rd** das untere der beiden Register angibt. *(zulässig für **Rd**: **r24, r26, r28** und **r30**)(zulässig für **K63**: **0** bis **63**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z C	**2**	Befehlszeit:	**500 ns**	**250 ns**

DEC – DEKREMENT EINES REGISTERINHALTS

Syntax:	Funktion:
DEC Rd	***Rd ← Rd – 1***

Beschreibung:
Der Inhalt des Registers **Rd** wird um **1** dekrementiert (vermindert). Das Ergebnis steht im Register **Rd**. Das **C**-Flag (Carry-Flag) des Statusregisters wird dabei nicht beeinflusst, so dass der Befehl in Schleifen für arithmetische Berechnungen benutzt werden kann. Dieser Befehl arbeitet bei vorzeichenlosen Werten nur mit den bedingten Sprungbefehlen **BREQ** und **BRNE** konsistent zusammen. Werden hingegen Werte in Zweierkomplement-Darstellung verwendet, stehen alle bedingten Sprungbefehle für vorzeichenbehaftete Werte zur Verfügung.
*(zulässig für **Rd**: **r0** bis **r31**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

INC – INKREMENT EINES REGISTERINHALTES

Syntax:	Funktion:
INC Rd	***Rd ← Rd + 1***

Beschreibung:
Der Inhalt des Registers **Rd** wird um **1** inkrementiert (erhöht). Das Ergebnis steht im Register **Rd**. Das **C**-Flag (Carry-Flag) des Statusregisters wird dabei nicht beeinflusst, so dass der Befehl in Schleifen für arithmetische Berechnungen benutzt werden kann. Dieser Befehl arbeitet bei vorzeichenlosen Werten nur mit den bedingten Sprungbefehlen **BREQ** und **BRNE** konsistent zusammen. Werden hingegen Werte in Zweierkomplement-Darstellung verwendet, stehen alle bedingten Sprungbefehle für vorzeichenbehaftete Werte zur Verfügung.
*(zulässig für **Rd**: **r0** bis **r31**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

NEG – ZWEIERKOMPLEMENT

Syntax:	Funktion:
NEG Rd	***Rd ← $00 – Rd***

Beschreibung:
Es wird das Zweierkomplement des Inhalts des Registers **Rd** gebildet. Dabei wird vom Wert **$00** der Inhalt des Registers abgezogen.
*(zulässig für **Rd**: **r0** bis **r31**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

TST – TESTE, OB REGISTER NULL ODER NEGATIV

Syntax:	Funktion:
TST *Rd*	***Rd ← Rd .AND. Rd***

Beschreibung:
Dieser Befehl überprüft, ob der Inhalt des Registers **Rd** null oder negativ ist. Zu diesem Zweck wird der Inhalt dieses Register mit sich selbst **UND**-verknüpft.
*(zulässig für **Rd: r0** bis **r31**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.1.2 LOGISCHE BEFEHLE

AND – LOGISCHES UND ZWEIER REGISTER

Syntax:			Funktion:	
AND *Rd, Rr*			***Rd ← Rd .AND. Rr***	
Beschreibung: Der Inhalt des Registers **Rr** wird mit dem Inhalt des Registers **Rd** logisch **UND**-verknüpft. Das Ergebnis der Verknüpfung steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

ANDI – LOGISCHES UND REGISTER MIT WERT

Syntax:			Funktion:	
ANDI *Rd, K255*			***Rd ← Rd .AND. K255***	
Beschreibung: Der unmittelbare Wert **K255** wird mit dem Inhalt des Registers **Rd** logisch **UND**-verknüpft. Das Ergebnis der Verknüpfung steht im Register **Rd**. *(zulässig für **Rd**: **r16** bis **r31**)(zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

OR – LOGISCHES ODER ZWEIER REGISTER

Syntax:			Funktion:	
OR *Rd, Rr*			***Rd ← Rd .OR. Rr***	
Beschreibung: Der Inhalt des Registers **Rr** wird mit dem Inhalt des Registers **Rd** logisch **ODER**-verknüpft. Das Ergebnis der Verknüpfung steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

ORI – LOGISCHES ODER REGISTER MIT WERT

Syntax:			Funktion:	
ORI *Rd, K255*			***Rd ← Rd .OR. K255***	
Beschreibung: Der unmittelbare Wert **K255** wird mit dem Inhalt des Registers **Rd** logisch **ODER**-verknüpft. Das Ergebnis der Verknüpfung steht im Register **Rd**. *(zulässig für **Rd**: **r16** bis **r31**)(zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

EOR – EXKLUSIVES ODER ZWEIER REGISTER

Syntax:			Funktion:	
EOR *Rd, Rr*			***Rd ← Rd .XOR. Rr***	
Beschreibung: Der Inhalt des Registers **Rr** wird mit dem Inhalt des Registers **Rd** logisch **XODER**-verknüpft. Das Ergebnis der Verknüpfung steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

COM – EINSERKOMPLEMENT EINES REGISTERS

Syntax:			Funktion:	
COM Rd			***Rd ← $FF - Rd***	
Beschreibung: Das Einserkomplement aus dem Inhalt des Registers **Rd** wird gebildet. Dazu wird vom konstanten Wert **$FF** der Inhalt des Registers **Rd** subtrahiert. Jene Bits im Register **Rd**, welche auf **1** sind, werden dabei auf **0** gesetzt und jene Bits im Register **Rd**, welche auf **0** sind, werden dabei auf **1** gesetzt. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z C(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.2 SPRUNGBEFEHLE

6.4.2.1 BEDINGTE SPRUNGBEFEHLE

SBIC – SPRUNG, WENN BIT IN I/O-REGISTER GELÖSCHT

Syntax:	Funktion:
SBIC addr, bit	***If I/O(bit) = 0 Then PC ← PC + 2(3) Else PC ← PC + 1***

Beschreibung:

Der nächste Befehl im Programmspeicher wird übersprungen, wenn das Bit **bit** im I/O-Register **addr** gelöscht ist. Ist dies nicht der Fall, dann wird die Programmausführung mit dem nächsten Befehl fortgesetzt. Im Falle des Überspringens eines Einwortbefehles wird der **PC** (Programmzähler) um **2** erhöht und im Falle des Überspringens eines Zweiwortbefehles um **3**.

*(zulässig für **bit**: **0** bis **7**)(zulässig für **addr**: **0** bis **31** bzw. **$00** bis **$1F**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1(2/3)**	Befehlszeit:	**250(500/750) ns**	**125(250/375) ns**

SBIS – SPRUNG, WENN BIT IN I/O-REGISTER GESETZT

Syntax:	Funktion:
SBIS addr, bit	***If I/O(bit) = 1 Then PC ← PC + 2(3) Else PC ← PC + 1***

Beschreibung:

Der nächste Befehl im Programmspeicher wird übersprungen, wenn das Bit **bit** im I/O-Register **addr** gesetzt ist. Ist dies nicht der Fall, dann wird die Programmausführung mit dem nächsten Befehl fortgesetzt. Im Falle des Überspringens eines Einwortbefehles wird der **PC** (Programmzähler) um **2** erhöht und im Falle des Überspringens eines Zweiwortbefehles um **3**.

*(zulässig für **bit**: **0** bis **7**)(zulässig für **addr**: **0** bis **31** bzw. **$00** bis **$1F**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1(2/3)**	Befehlszeit:	**250(500/750) ns**	**125(250/375) ns**

SBRC – SPRUNG, WENN BIT IN REGISTER GELÖSCHT

Syntax:	Funktion:
SBRC Rr, bit	***If Rr(bit) = 0 Then PC ← PC + 2(3) Else PC ← PC + 1***

Beschreibung:

Der nächste Befehl im Programmspeicher wird übersprungen, wenn das Bit **bit** im Register **Rr** gelöscht ist. Ist dies nicht der Fall, dann wird die Programmausführung mit dem nächsten Befehl fortgesetzt. Im Falle des Überspringens eines Einwortbefehles wird der **PC** (Programmzähler) um **2** erhöht und im Falle des Überspringens eines Zweiwortbefehles um **3**.

*(zulässig für **bit**: **0** bis **7**)(zulässig für **Rr**: **r0** bis **r31**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1(2/3)**	Befehlszeit:	**250(500/750) ns**	**125(250/375) ns**

SBRS – SPRUNG, WENN BIT IN REGISTER GESETZT

Syntax:	Funktion:			
SBIS Rr, bit	***If Rr(bit) = 1 Then PC ← PC + 2(3) Else PC ← PC + 1***			
Beschreibung:				
Der nächste Befehl im Programmspeicher wird übersprungen, wenn das Bit **bit** im Register **Rr** gesetzt ist. Ist dies nicht der Fall, dann wird die Programmausführung mit dem nächsten Befehl fortgesetzt. Im Falle des Überspringens eines Einwortbefehles wird der **PC** (Programmzähler) um **2** erhöht und im Falle des Überspringens eines Zweiwortbefehles um **3**. *(zulässig für **bit**: **0** bis **7**)(zulässig für **Rr**: **0** bis **31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1(2/3)**	Befehlszeit:	**250(500/750) ns**	**125(250/375) ns**

BRBC – SPRUNG, WENN STATUSREGISTER-BIT GELÖSCHT

Syntax:	Funktion:			
BRBC bit, k	***If SREG(bit)=0 Then PC ← PC + k + 1 Else PC ← PC + 1***			
Beschreibung:				
Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das mit **bit** angegebene Bit im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Mit dem Wert **bit** lässt sich jedes Bit im Statusregister adressieren. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRBS – SPRUNG, WENN STATUSREGISTER-BIT GESETZT

Syntax:	Funktion:			
BRBS bit, k	***If SREG(bit)=0 Then PC ← PC + k + 1 Else PC ← PC + 1***			
Beschreibung:				
Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das mit **bit** angegebene Bit im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Mit dem Wert **bit** lässt sich jedes Bit im Statusregister adressieren. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRCC – SPRUNG, WENN CARRY-FLAG GELÖSCHT

Syntax:	Funktion:
BRCC k	***If C = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **C**-Flag (Carry-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

(*äquivalent zu **„BRBC 0, k“***)

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRCS – SPRUNG, WENN CARRY-FLAG GESETZT

Syntax:	Funktion:
BRCS k	***If C = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **C**-Flag (Carry-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

(*äquivalent zu **„BRBS 0, k“***)

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BREQ – SPRUNG, WENN ZERO-FLAG GESETZT

Syntax:	Funktion:
BREQ k	***If Z = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **Z**-Flag (Zero-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Steht dieser Befehl unmittelbar nach dem Befehl ***„CP Rd, Rr“, „CPI Rd, K255,“ „SUB Rd, Rr“*** oder ***„SUBI Rd, K255“***, wird der Sprung nur dann ausgeführt, wenn die Werte (mit oder ohne Vorzeichen) in den Registern **Rd** und **Rr** bzw. **K255** gleich sind. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

(*äquivalent zu **„BRBS 1, k“***)

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRGE – SPRUNG, WENN GRÖSSER ODER GLEICH (vorzeichenbehaftet)

Syntax:	Funktion:
BRGE k	***If S = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **S**-Flag (Signed-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Steht dieser Befehl unmittelbar nach dem Befehl ***„CP Rd, Rr“, „CPI Rd, K255,“ „SUB Rd, Rr“*** oder ***„SUBI Rd, K255“***, wird der Sprung nur dann ausgeführt, wenn der vorzeichenbehaftete Wert im Register **Rd** größer oder gleich dem vorzeichenbehafteten Wert im Register **Rr** bzw. dem vorzeichenbehafteten Wert **K255** ist. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 4, k“**)(zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRHC – SPRUNG, WENN HALF-CARRY-FLAG GELÖSCHT

Syntax:	Funktion:
BRHC k	***If H = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **H**-Flag (Half-Carry-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 5, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRHS – SPRUNG, WENN HALF-CARRY-FLAG GESETZT

Syntax:	Funktion:
BRHS k	***If H = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **H**-Flag (Half-Carry-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBS 5, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRID – SPRUNG, WENN INTERRUPT-FLAG GELÖSCHT

Syntax:	Funktion:
BRID k	***If I = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **I**-Flag (Global-Interrupt-Enable-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.
*(äquivalent zu **„BRBC 7, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRIE – SPRUNG, WENN INTERRUPT-FLAG GESETZT

Syntax:	Funktion:
BRIE k	***If I = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **I**-Flag (Global-Interrupt-Enable-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.
*(äquivalent zu **„BRBS 7, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRLO – SPRUNG, WENN KLEINER (VORZEICHENLOS)

Syntax:	Funktion:
BRLO k	***If C = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **C**-Flag (Carry-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Steht dieser Befehl unmittelbar nach dem Befehl ***„CP Rd, Rr“, „CPI Rd, K255,“ „SUB Rd, Rr“*** oder ***„SUBI Rd, K255“***, wird der Sprung nur dann ausgeführt, wenn der vorzeichenlose Wert im Register **Rd** niedriger als der vorzeichenlose Wert im Register **Rr** bzw. vorzeichenlose Wert **K255** ist. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.
*(äquivalent zu **„BRBS 0, k“**) (zulässig für **K255: 0** bis **255** bzw. **$00** bis **$FF**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRLT – SPRUNG, WENN KLEINER (VORZEICHENBEHAFTET)

Syntax:	Funktion:
BRLT k	***If S = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **S**-Flag (Signed-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Steht dieser Befehl unmittelbar nach dem Befehl ***„CP Rd, Rr“, „CPI Rd, K255,“ „SUB Rd, Rr“*** oder ***„SUBI Rd, K255“***, wird der Sprung nur dann ausgeführt, wenn der vorzeichenbehaftete Wert im Register **Rd** kleiner als der vorzeichenbehaftete Wert im Register **Rr** bzw. vorzeichenbehaftete Wert **K255** ist. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBS 4, k“**) (zulässig für **K255: 0** bis **255** bzw. **$00** bis **$FF**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRMI – SPRUNG, WENN NEGATIV

Syntax:	Funktion:
BRMI k	***If N = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **N**-Flag (Negative-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBS 2, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRNE – SPRUNG, WENN UNGLEICH

Syntax:	Funktion:
BRNE k	***If Z = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **Z**-Flag (Zero-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Steht dieser Befehl unmittelbar nach dem Befehl ***„CP Rd, Rr“, „CPI Rd, K255,“ „SUB Rd, Rr“*** oder ***„SUBI Rd, K255“***, wird der Sprung nur dann ausgeführt, wenn die Werte (mit oder ohne Vorzeichen) in den Registern **Rd** und **Rr** bzw. **K255** nicht gleich sind. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 1, k“**) (zulässig für **K255: 0** bis **255** bzw. **$00** bis **$FF**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRPL – SPRUNG, WENN POSITIV

Syntax:	Funktion:
BRPL k	***If N = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **N**-Flag (Negative-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 2, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRSH – SPRUNG, WENN GLEICH ODER HÖHER (VORZEICHENLOS)

Syntax:	Funktion:
BRSH k	***If C = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **C**-Flag (Carry-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Steht dieser Befehl unmittelbar nach dem Befehl ***„CP Rd, Rr“, „CPI Rd, K255,“ „SUB Rd, Rr“*** oder ***„SUBI Rd, K255“***, wird der Sprung nur dann ausgeführt, wenn der vorzeichenlose Wert im Register **Rd** gleich oder höher als der vorzeichenlose Wert im Register **Rr** bzw. vorzeichenlose Wert **K255** ist. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 0, k“**) (zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRTC – SPRUNG, WENN T-FLAG GELÖSCHT

Syntax:	Funktion:
BRTC k	***If T = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **T**-Flag (Transfer-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 6, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRTS – SPRUNG, WENN T-FLAG GESETZT

Syntax:	Funktion:
BRTS k	***If T = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **T**-Flag (Transfer-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBS 6, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRVC – SPRUNG, WENN OVERFLOW-FLAG GELÖSCHT

Syntax:	Funktion:
BRVC k	***If V = 0 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **V**-Flag (Zweierkomplement-Overflow-Flag) im Statusregister gelöscht ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBC 3, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

BRVS – SPRUNG, WENN OVERFLOW-FLAG GESETZT

Syntax:	Funktion:
BRVS k	***If V = 1 Then PC ← PC + k + 1 Else PC ← PC + 1***

Beschreibung:

Dieser bedingte relative Sprung um den Wert **k** wird ausgeführt, wenn das **V**-Flag (Zweierkomplement-Overflow-Flag) im Statusregister gesetzt ist **(PC ← PC + k + 1)**. Ist dies nicht der Fall, wird mit jener Befehlszeile fortgefahren, die nach diesem Sprungbefehl steht **(PC ← PC + 1)**. Der Wert **k** wird als Zweierkomplement interpretiert, so dass relative Sprünge im Bereich **–64** bis **+63** zum **PC** (Programmzähler) ausgeführt werden können. Der Wert **k** kann auch eine Sprungmarke sein.

*(äquivalent zu **„BRBS 3, k“**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

6.4.2.2 VERGLEICHSBEFEHLE

CP – VERGLEICH ZWEIER REGISTER

Syntax:			Funktion:	
CP Rd, Rr			***Rd – Rr***	
Beschreibung:				
Die Inhalte der Register **Rd** und **Rr** werden miteinander verglichen, ohne dabei die Inhalte zu verändern. Nach diesem Befehl können alle bedingten Verzweigungsbefehle (Sprungbefehle) folgen. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1 (2)**	Befehlszeit:	**250 ns (500ns)**	**125 ns (250ns)**

CPC – VERGLEICH ZWEIER REGISTER INKL. CARRY-FLAG

Syntax:			Funktion:	
CPC Rd, Rr			***Rd – Rr – C***	
Beschreibung:				
Die Inhalte der Register **Rd** und **Rr** werden miteinander verglichen, wobei auch das **C**-Flag (Carry-Flag) des Statusregisters miteinbezogen wird. Keines der verwendeten Register wird verändert. Nach diesem Befehl können alle bedingten Verzweigungsbefehle (Sprungbefehle) folgen. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

CPI – VERGLEICH REGISTER MIT WERT

Syntax:			Funktion:	
CPI Rd, K255			***Rd – K255***	
Beschreibung:				
Der Inhalt des Register **Rd** wird mit dem Wert **K255** verglichen. wobei der Inhalt des Registers erhalten bleibt. Nach diesem Befehl können alle bedingten Verzweigungsbefehle (Sprungbefehle) folgen. *(zulässig für **Rd**: **r16** bis **r31**)(zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

CPSE – VERGLEICH ZWEIER REGISTER MIT SPRUNG

Syntax:	Funktion:			
CPC Rd, Rr	***If Rd = Rr Then PC ← PC + 2(or3) Else PC ← PC + 1***			
Beschreibung:				
Die Inhalte der Register **Rd** und **Rr** werden miteinander verglichen. Sind die Inhalte beider Register gleich, so wird der nachfolgende Befehl übersprungen. Handelt es sich beim nachfolgenden Befehl um einen Einwort-Befehl, so wird der **PC** (Programmzähler) um **2** erhöht. Handelt es sich beim nachfolgenden Befehl um einen Zweiwort-Befehl, so wird der **PC** (Programmzähler) um **3** erhöht. Keines der verwendeten Register wird verändert. *(zulässig für **Rd, Rr**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.2.3 UNBEDINGTE SPRUNGBEFEHLE

IJMP – INDIREKTER SPRUNG

Syntax:			Funktion:	
IJMP			***PC ← r31:r30***	
Beschreibung:				
Ein Sprung zur Adresse, auf die der Inhalt des **Z-Zeigers** (Z-Pointers, **r31:r30**) zeigt, wird ausgeführt. Der **PC** (Programmzähler) wird mit der Adresse geladen, die im Registerpaar **r31:r30** (Z-Zeiger) steht. Dieser Sprungbefehl kann an eine beliebige Adresse innerhalb des Programmspeichers verzweigen.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

RJMP – RELATIVER SPRUNG

Syntax:			Funktion:	
RJMP k2048			***PC ← PC + 1 + k2048***	
Beschreibung:				
Ein zur gegenwärtigen Adresse (Inhalt des Programmzählers) relativer Sprung wird ausgeführt. Hierbei wird der Inhalt des **PC** (Programmzähler) um den Wert **1 + k2048** bzw. **1 – k2048** verändert.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

6.4.2.4 UNTERPROGRAMMAUFRUFE

ICALL – INDIREKTER UNTERPROGRAMMAUFRUF

Syntax:			Funktion:	
ICALL			***PC ← r31:r30***	
Beschreibung:				
Ein Unterprogramm an der Adresse, auf die der Inhalt des **Z-Zeigers** (Z-Pointers, **r31:r30**) zeigt, wird aufgerufen. der um **1** erhöhte **PC** (Programmzähler) wird auf den **Stapel** (Stack) gespeichert. Danach wird der **PC** (Programmzähler) mit der Adresse geladen, die im Registerpaar **r31:r30** (Z-Zeiger) steht. Das Unterprogramm kann an einer beliebigen Adresse innerhalb des Programmspeichers stehen.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**3**	Befehlszeit:	**750 ns**	**375 ns**

RCALL – RELATIVER UNTERPROGRAMMAUFRUF

Syntax:			Funktion:	
RCALL k2048			***PC ← PC + 1 + k2048***	
Beschreibung:				
Ein Unterprogramm mit einem Abstand von **k2048** Bytes zur gegenwärtigen Adresse wird aufgerufen. Der um den Wert **1** erhöhte **PC** (Programmzähler) wird am Stapel abgelegt und der Stapelzeiger um den Wert **2** dekrementiert, damit ein späterer Rücksprung zu jenem Befehl ermöglicht wird, der in der Zeile nach dem Aufruf des Unterprogramms steht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**3**	Befehlszeit:	**750 ns**	**375 ns**

RET – RÜCKSPRUNG VOM UNTERPROGRAMM

Syntax:			Funktion:	
RET			***PC ← Stapel***	
Beschreibung:				
Der Inhalt jener Speicherstelle, auf den der Stapelzeiger verweist, wird in den **PC** (Programmzähler) geladen. Anschließend wird der Stapelzeiger um den Wert **2** inkrementiert, da jede Adresse im Programmspeicher aus zwei Bytes (Higher-Byte, Lower-Byte) besteht. Die Abarbeitung der Befehle wird an jener Stelle im Programm fortgesetzt, auf die der **PC** (Programmzähler) nun verweist.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**4**	Befehlszeit:	**1000 ns**	**500 ns**

RETI – RÜCKSPRUNG VON INTERRUPT-ROUTINE

Syntax:			Funktion:	
RETI			***PC ← Stapel***	
Beschreibung:				
Der Inhalt jener Speicherstelle, auf den der Stapelzeiger verweist, wird in den **PC** (Programmzähler) geladen. Anschließend wird der Stapelzeiger um den Wert **2** inkrementiert, da jede Adresse im Programmspeicher aus zwei Bytes (Higher-Byte, Lower-Byte) besteht. Die Abarbeitung der Befehle wird an jener Stelle im Programm fortgesetzt, auf die der **PC** (Programmzähler) nun verweist. Das **I**-Flag (Global-Interrupt-Flag) wird auf **1** gesetzt und damit das Abarbeiten anderer Interrupts wieder ermöglicht. Das Statusregister wird weder beim Aufruf einer Interrupt-Routine noch beim Rücksprung aus der Interrupt-Routine gesichert. Dies muss durch den Programmhersteller organisiert werden.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
I(1)	**4**	Befehlszeit:	**1000 ns**	**500 ns**

6.4.3 DATENTRANSFERBEFEHLE

IN – LADE VON I/O-REGISTER IN UNIVERSALREGISTER

Syntax:			Funktion:	
IN Rd, Port			***Rd ← I/O(Port)***	
Beschreibung:				
Dieser Befehl lädt den Inhalt des durch den Wert **Port** adressierten I/O-Registers in ein Universalregister. *(zulässig für **Rd**: **r0** bis **r31**) (zulässig für **Port**: **0** bis **63** bzw. **$00** bis **$3F**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

OUT – LADE VON UNIVERSALREGISTER IN I/O-REGISTER

Syntax:			Funktion:	
OUT Port, Rr			***I/O(Port) ← Rr***	
Beschreibung:				
Dieser Befehl lädt den Inhalt des Registers **Rr** in das durch den Wert **Port** adressierte I/O-Register. *(zulässig für **Rr**: **r0** bis **r31**) (zulässig für **Port**: **0** bis **63** bzw. **$00** bis **$3F**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

PUSH – LEGE REGISTERINHALT AUF DEN STAPEL

Syntax:			Funktion:	
PUSH Rd			***Stapel ← Rd***	
Beschreibung:				
Der Stapelzeiger (Stack-Pointer) wird dekrementiert, da der Stapel von der initialisierten Startstelle im Speicher nach unten (gegen **0**) wächst und durch diesen Befehl vergrößert wird. Danach wird der Inhalt des Registers **Rd** an jener Stelle, auf die der Stapelzeiger nun zeigt, am Stapel abgelegt. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

POP – LADE VOM STAPEL IN REGISTER

Syntax:			Funktion:	
POP Rd			***Rd ← Stapel***	
Beschreibung:				
Der Inhalt der Speicherstelle, auf die der Stapelzeiger (Stack-Pointer) verweist, wird in das Register **Rd** geladen. Anschließend wird der Stapelzeiger inkrementiert, da der Stapel von der initialisierten Startstelle im Speicher nach unten (gegen **0**) organisiert ist und durch diesen abgebaut wird, d.h. nach oben wächst. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

MOV – KOPIERE REGISTERINHALT

Syntax:			Funktion:	
MOV Rd, Rr			***Rd ← Rr***	
Beschreibung:				
Der Inhalt des Registers **Rr** wird in das Register **Rd** kopiert. Der Inhalt des Registers **Rr** bleibt dabei unverändert. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

LDI – LADE REGISTER MIT WERT

Syntax:			Funktion:	
LDI Rd, K255			***Rd ← K255***	
Beschreibung:				
Der Wert **K255** wird in das Register Rd geladen. *(zulässig für **Rd**: **r16** bis **r31**)(zulässig für **K255**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

LDS – LADE REGISTER MIT BYTE AUS DATENSPEICHER

Syntax:			Funktion:	
LDS Rd, k65535			***Rd ← SRAM(k65535)***	
Beschreibung:				
Das mit **k65535** adressierte Byte aus dem Datenspeicher wird direkt in das Register **Rd** geladen. Der Datenspeicher beinhaltet die gespiegelten Universalregister und I/O-Register, sowie internen und optional externen SRAM. Das EEPROM hat einen eigenen Adressbereich. *(zulässig für **Rd**: **r0** bis **r31**)(zulässig für **k65535**: **0** bis **65535** bzw. **$0000** bis **$FFFF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

STS – SPEICHERE REGISTER IN DEN DATENSPEICHER

Syntax:			Funktion:	
STS k65535, Rr			***SRAM(k65535) ← Rr***	
Beschreibung:				
Dieser Befehl speichert den Inhalt des Registers **Rr** direkt an die mit **k65535** adressierte Stelle im Datenspeicher. Der Datenspeicher beinhaltet die gespiegelten Universalregister und I/O-Register, sowie internen und optional externen SRAM. Das EEPROM hat einen eigenen Adressbereich. *(zulässig für **Rr**: **r0** bis **r31**)(zulässig für **k65535**: **0** bis **65535** bzw. **$0000** bis **$FFFF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

Die Syntax und die Beschreibung für die Befehle zum indirekten Laden und Speichern sind dem AVR-Instruction-Set zu entnehmen.

6.4.4 BITMANIPULATIONSBEFEHLE

6.4.4.1 SETZBEFEHLE

SER – SETZE ALLE BITS IM REGISTER

Syntax:			Funktion:	
SER Rd			***Rd ← $FF***	
Beschreibung:				
Dieser Befehl setzt alle Bits im Register **Rd**. *(zulässig für **Rd: r16** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

SBR – SETZE BITS IM REGISTER

Syntax:			Funktion:	
SBR Rd, maske			***Rd ← Rd .OR. maske***	
Beschreibung:				
Setzt die mit dem Wert **maske** spezifizierten (auf **1** gesetzten) Bits im Register **Rd**. Der Inhalt des Registers **Rd** wird dabei mit dem Wert **maske** logisch **ODER**-verknüpft. An allen binären Stellen, an denen im Wert **maske** der Wert **1** steht, wird das zugehörige Bit im Register **Rd** gesetzt. An allen binären Stellen, an denen im Wert **maske** der Wert **0** steht, bleibt der Wert des zugehörigen Bits im Register **Rd** erhalten. Das Ergebnis steht im Register **Rd**. *(zulässig für **Rd: r16** bis **r31**)(zulässig für **maske: 0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

SBI – SETZE BIT IM I/O-REGISTER

Syntax:	Funktion:			
SBI addr, bit	***I/O(bit) ← 1***			
Beschreibung:				
Dieser Befehl setzt das mit **bit** angegebene Bit im I/O-Register **addr**. *(zulässig für **bit: 0** bis **7**)(zulässig für **addr: 0** bis **31** bzw. **$00** bis **$1F**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

BSET – SETZE BIT IM STATUSREGISTER

Syntax:			Funktion:	
BSET bit			***SREG(bit) ← 1***	
Beschreibung:				
Das Flag an der Position **bit** im Statusregister wird mit diesem Befehl gesetzt. Der Zustand der anderen Flags im Statusregister wird dabei nicht verändert. *(zulässige Werte für **bit: 0** bis **7**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
I T H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

SEC – SETZE CARRY-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SEC			***C ← 1***	
Beschreibung:				
Das **C**-Flag (Carry-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
C(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SEZ – SETZE ZERO-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SEZ			***Z ← 1***	
Beschreibung:				
Das **Z**-Flag (Zero-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
Z(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SEN – SETZE NEGATIVE-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SEN			***N ← 1***	
Beschreibung:				
Das **N**-Flag (Negative-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
N(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SEV – SETZE OVERFLOW-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SEV			***V ← 1***	
Beschreibung:				
Das **V**-Flag (Zweierkomplement-Overflow-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
V(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SES – SETZE SIGNED-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SES			***S ← 1***	
Beschreibung:				
Das **S**-Flag (Signed-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SEH – SETZE HALF-CARRY-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SEH			***H ← 1***	
Beschreibung:				
Das **H**-Flag (Half-Carry-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SET – SETZE T-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SET			***T ← 1***	
Beschreibung:				
Das **T**-Flag (Transfer-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
T(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

SEI – SETZE INTERRUPT-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
SEI			***I ← 1***	
Beschreibung:				
Das **I**-Flag (Global-Interrupt-Flag) im Statusregister wird gesetzt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
I(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

BLD – LADE T-FLAG IN REGISTER

Syntax:			Funktion:	
BLD Rd, bit			***Rd(bit) ← T***	
Beschreibung:				
Das **T**-Flag (Transfer-Flag) aus dem Statusregister wird mit diesem Befehl an die Position **bit** des Registers **Rd** geladen. Der Zustand des **T**-Flags (Transfer-Flag) wird dabei nicht verändert. *(zulässige Werte für **bit**: **0** bis **7**)(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

BST – SPEICHERE BIT AUS REGISTER IM T-FLAG

Syntax:			Funktion:	
BST Rd, bit			***T ← Rd(bit)***	
Beschreibung:				
Das Bit an der Position **bit** des Registers **Rd** wird in das **T**-Flag (Transfer-Flag) im Statusregister geladen. Der Zustand der anderen Flags im Statusregister wird dabei nicht verändert. *(zulässige Werte für **bit**: **0** bis **7**)(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
T	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.4.2 LÖSCHBEFEHLE

CLR – LÖSCHE ALLE BITS IM REGISTER

Syntax:		Funktion:		
CLR Rd		***Rd ← Rd .XOR. Rd***		
Beschreibung:				
Das Register **Rd** wird gelöscht (auf **0** gesetzt), indem es mit sich selber logisch **XODER**-verknüpft wird. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S(0) V(0) N(0) Z(1)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CBR – LÖSCHE BITS IM REGISTER

Syntax:		Funktion:		
CBR Rd, maske		***Rd ← Rd .AND. ($FF – maske)***		
Beschreibung:				
Löscht die mit dem Wert **maske** spezifizierten (auf 1 gesetzten) Bits im Register **Rd**. Der Inhalt des Registers **Rd** wird dabei mit dem Einserkomplement (z.B. ***0b11100110 → 0b00011001***) des Wertes **maske** logisch **UND**-verknüpft. An allen binären Stellen, an denen im Wert **maske** der Wert **1** steht, wird das zugehörige Bit im Register **Rd** gelöscht. An allen binären Stellen, an denen im Wert **maske** der Wert **0** steht, bleibt der Wert des zugehörigen Bits im Register **Rd** erhalten. Das Ergebnis steht im Register **Rd**. *(zulässig für **Rd**: **r16** bis **r31**)(zulässig für **maske**: **0** bis **255** bzw. **$00** bis **$FF**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V(0) N Z	**1**	Befehlszeit:	**250 ns**	**125 ns**

CBI – LÖSCHE BIT IM I/O-REGISTER

Syntax:		Funktion:		
CBI Port, bit		***I/O(Port, bit) ← 0***		
Beschreibung:				
Das Bit an der Position **bit** im I/O-Register **Port** wird gelöscht. Der Zustand der anderen Bits in diesem I/O-Register werden dabei nicht verändert. *(zulässige Werte für **bit**: **0** bis **7**)(zulässig für **Port**: **0** bis **31** bzw. **$00** bis **$1F**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**2**	Befehlszeit:	**500 ns**	**250 ns**

BCLR – LÖSCHE EIN BIT IM STATUSREGISTER

Syntax:		Funktion:		
BCLR bit		***SREG(bit) ← 0***		
Beschreibung:				
Dieser Befehl löscht das Flag **bit** im Statusregister. Die anderen Bits werden von diesem Befehl nicht verändert. *(zulässige Werte für **bit**: **0** bis **7**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
I T H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLC – LÖSCHE CARRY-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLC			***C ← 0***	
Beschreibung:				
Das **C**-Flag (Carry-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
C(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLZ – LÖSCHE ZERO-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLZ			***Z ← 0***	
Beschreibung:				
Das **Z**-Flag (Zero-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
Z(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLN – LÖSCHE NEGATIVE-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLN			***N ← 0***	
Beschreibung:				
Das **N**-Flag (Negative-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
N(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLV – LÖSCHE V-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLV			***V ← 0***	
Beschreibung:				
Das **V**-Flag (Zweierkomplement-Overflow-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
V(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLS – LÖSCHE S-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLS			***S ← 0***	
Beschreibung:				
Das **S**-Flag (Signed-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLH – LÖSCHE HALF-CARRY-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLH			***H ← 0***	
Beschreibung:				
Das **H**-Flag (Half-Carry-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLT – LÖSCHE T-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLT			***T ← 0***	
Beschreibung:				
Das **T**-Flag (Transfer-Flag) im Statusregister wird gelöscht.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
T(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

CLI – LÖSCHE INTERRUPT-FLAG IM STATUSREGISTER

Syntax:			Funktion:	
CLI			***I ← 0***	
Beschreibung:				
Das **I**-Flag (Global-Interrupt-Flag) im Statusregister wird gelöscht. Die Interrupts werden sofort gesperrt. Nach dem Aufrufen dieses Befehls werden keine Interrupts abgearbeitet. Nur ein gerade während der Ausführung dieses Befehls eingetretener Interrupt wird (noch) ausgeführt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
I(0)	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.4.3 SCHIEBEBEFEHLE

LSL – LOGISCHES LINKSSCHIEBEN

Syntax:	Funktion:			
LSL *Rd*	***C;Rd(7)..Rd(1);Rd(0) ← Rd(7);Rd(6)..Rd(0);0***			
Beschreibung:				
Der Inhalt des Registers **Rd** wird um eine Stelle logisch nach links geschoben. Dabei wird der Inhalt in **Rd(7)** in das **C**-Flag (Carry-Flag) des Statusregisters übertragen, **Rd(6)..Rd(0)** wird nach links auf **Rd(7)**..**Rd(1)** geschoben und **Rd(0)** mit **0** überschrieben. Dieser Befehl multipliziert eine vorzeichenlose Zahl mit zwei. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

LSR – LOGISCHES RECHTSSCHIEBEN

Syntax:	Funktion:			
LSR *Rd*	***Rd(7), Rd(6)..Rd(0);C ← 0;Rd(7)..Rd(1);Rd(0)***			
Beschreibung:				
Der Inhalt des Registers **Rd** wird um eine Stelle logisch nach rechts geschoben. Dabei wird der Inhalt in **Rd(0)** in das **C**-Flag (Carry-Flag) des Statusregisters übertragen, **Rd(7)..Rd(1)** wird nach links auf **Rd(6)**..**Rd(0)** geschoben und **Rd(7)** mit **0** überschrieben. Dieser Befehl dividiert eine vorzeichenlose Zahl mit zwei. Das **C**-Flag (Carry-Flag) kann zum Runden herangezogen werden. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

ROL – LINKSROTATION ÜBER CARRY-FLAG

Syntax:	Funktion:			
ROL *Rd*	***C;Rd(7)..Rd(1);Rd(0) ← Rd(7);Rd(6)..Rd(0);C***			
Beschreibung:				
Der Inhalt des Registers **Rd** rotiert um eine Stelle nach links. **Rd(7)** wird in das **C**-Flag (Carry-Flag) des Statusregisters geschoben, **Rd(6)..Rd(0)** gleichzeitig auf **Rd(7)..Rd(1)** nach links verschoben und das **C**-Flag (Carry-Flag) aus dem Statusregister zeitgleich in **Rd(0)** gespeichert. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

ROR – RECHTSROTATION ÜBER CARRY-FLAG

Syntax:	Funktion:			
ROR *Rd*	***Rd(7);Rd(6)..Rd(0);C ← C;Rd(7)..Rd(1);Rd(0)***			
Beschreibung:				
Der Inhalt des Registers **Rd** rotiert um eine Stelle nach rechts. **Rd(0)** wird in das **C**-Flag (Carry-Flag) des Statusregisters geschoben, **Rd(7)..Rd(1)** gleichzeitig auf **Rd(6)..Rd(0)** nach rechts verschoben und das **C**-Flag (Carry-Flag) aus dem Statusregister zeitgleich in **Rd(7)** gespeichert. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

ASR – ARITHMETISCHES RECHTSSCHIEBEN

Syntax:	Funktion:
ASR *Rd*	***Rd(7);0;Rd(5)..Rd(0);C ← Rd(7),Rd(6)..Rd(1);Rd(0)***

Beschreibung:

Der Inhalt des Registers **Rd** wird arithmetisch um eine Stelle nach rechts verschoben. Dabei bleibt der Inhalt in **Rd(7)** erhalten. **Rd(0)** wird in das **C**-Flag (Carry-Flag) des Statusregisters übertragen, **Rd(6)..Rd(1)** auf **Rd(5)..Rd(0)** nach rechts verschoben und **Rd(6)** mit **0** überschrieben. Dieser Befehl teilt eine Zahl, die als Zweierkomplement gespeichert ist, durch **2**, wobei das Vorzeichen der Zahl nicht geändert wird.

*(zulässig für **Rd**: **r0** bis **r31**)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
S V N Z C	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.5 SONSTIGE BEFEHLE

NOP – KEINE OPERATION

Syntax:			Funktion:	
NOP			***Verzögerung***	
Beschreibung: Es wird ein funktionsloser Befehl ausgeführt, der im Programmablauf eine Verzögerung um einen Taktzyklus erzeugt.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

SLEEP – SETZE DEN µC IN DEN SLEEP-MODUS

Syntax:			Funktion:	
SLEEP			***keine***	
Beschreibung: Dieser Befehl setzt den AVR-µC in den Sleep-Modus, der im MCU-Control-Register definiert ist. Wenn ein Interrupt den µC aus dem Sleep-Modus holt, so wird zunächst der dem Sleep-Befehl folgende Befehl ausgeführt, bevor der Interrupt-Handler gestartet wird.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

SWAP – VERTAUSCHE DIE HALB-BYTES EINES REGISTERS

Syntax:	Funktion:			
SWAP Rd	***Rd(7)..Rd(4);Rd(3)..Rd(0) ← Rd(3)..Rd(0);Rd(7)..Rd(4)***			
Beschreibung: Dieser Befehl tauscht die Halb-Bytes (Nibbles) eines Bytes. *(zulässig für **Rd**: **r0** bis **r31**)*				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

WDR – SETZE DEN WATCHDOG-TIMER ZURÜCK

Syntax:			Funktion:	
WDR			***WDT ← 0***	
Beschreibung: Dieser Befehl setzt den Watchdog-Timer zurück.				
beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
keine	**1**	Befehlszeit:	**250 ns**	**125 ns**

6.4.6 ALPHABETISCHES BEFEHLSVERZEICHNIS

ADC	Addiere Register mit Carry-Flag
ADD	Addiere Register ohne Carry-Flag
ADIW	Addiere Wert zum Registerpaar
AND	Logisches UND zweier Register
ANDI	Logisches UND Register mit Wert
ASR	Arithmetisches Rechtsschieben
BCLR	Lösche ein Bit im Statusregister
BLD	Lade T-Flag in Register
BRBC	Sprung, wenn Statusregister-BIT gelöscht
BRBS	Sprung, wenn Statusregister-BIT gesetzt
BRCC	Sprung, wenn Carry-Flag gelöscht
BRCS	Sprung, wenn Carry-Flag gesetzt
BREQ	Sprung, wenn Zero-Flag gesetzt
BRGE	Sprung, wenn grösser oder gleich (vorzeichenbehaftet)
BRHC	Sprung, wenn Half-Carry-Flag gelöscht
BRHS	Sprung, wenn Half-Carry-Flag gesetzt
BRID	Sprung, wenn Interrupt-Flag gelöscht
BRIE	Sprung, wenn Interrupt-Flag gesetzt
BRLO	Sprung, wenn kleiner (vorzeichenlos)
BRLT	Sprung, wenn kleiner (vorzeichenbehaftet)
BRMI	Sprung, wenn negativ
BRNE	Sprung, wenn ungleich
BRPL	Sprung, wenn positiv
BRSH	Sprung, wenn gleich oder höher (vorzeichenlos)
BRTC	Sprung, wenn T-Flag gelöscht
BRTS	Sprung, wenn T-Flag gesetzt
BRVC	Sprung, wenn Overflow-Flag gelöscht
BRVS	Sprung, wenn Overflow-Flag gesetzt
BSET	Setze Bit im Statusregister
BST	Speichere Bit aus Register im T-Flag
CBI	Lösche Bit im I/O-Register
CBR	Lösche Bits im Register
CLC	Lösche Carry-Flag im Statusregister
CLH	Lösche Half-Carry-Flag im Statusregister
CLI	Lösche Interrupt-Flag im Statusregister
CLN	Lösche Negative-Flag im Statusregister
CLR	Lösche alle Bits im Register
CLS	Lösche S-Flag im Statusregister
CLT	Lösche T-Flag im Statusregister
CLV	Lösche V-Flag im Statusregister
CLZ	Lösche Zero-Flag im Statusregister
COM	Einserkomplement eines Registers
CP	Vergleich zweier Register
CPC	Vergleich zweier Register inkl. Carry-Flag
CPI	Vergleich Register mit Wert
CPSE	Vergleich zweier Register mit Sprung
DEC	Dekrement eines Registerinhalts
EOR	Exklusives ODER zweier Register

ICALL	Indirekter Unterprogrammaufruf
IJMP	Indirekter Sprung
IN	Lade von I/O-Register in Universalregister
INC	Inkrement eines Registerinhaltes
LDI	Lade Register mit Wert
LDS	Lade Register mit Byte aus Datenspeicher
LSL	Logisches linksschieben
LSR	Logisches rechtsschieben
MOV	Kopiere Registerinhalt
NEG	Zweierkomplement
NOP	Keine Operation
OR	Logisches ODER zweier Register
ORI	Logisches ODER Register mit Wert
OUT	Lade von Universalregister in I/O-Register
POP	LADE vom Stapel in Register
PUSH	Lege Registerinhalt auf den Stapel
RCALL	Relativer Unterprogrammaufruf
RET	Rücksprung vom Unterprogramm
RETI	Rücksprung von Interrupt-Routine
RJMP	Relativer Sprung
ROL	Linksrotation über Carry-Flag
ROR	Rechtsrotation über Carry-Flag
SBC	Subtrahiere Register mit Carry-Flag
SBCI	Subtrahiere Wert und Carry-Flag von Register
SBI	Setze Bit im I/O-Register
SBIC	Sprung, wenn Bit in I/O-Register gelöscht
SBIS	Sprung, wenn Bit in I/O-Register gesetzt
SBIW	Subtrahiere Wert vom Registerpaar
SBR	Setze Bits im Register
SBRC	Sprung, wenn Bit in Register gelöscht
SBRS	Sprung, wenn Bit in Register gesetzt
SEC	Setze Carry-Flag im Statusregister
SEH	Setze Half-Carry-Flag im Statusregister
SEI	Setze Interrupt-Flag im Statusregister
SEN	Setze Negative-Flag im Statusregister
SES	Setze Signed-Flag im Statusregister
SET	Setze T-Flag im Statusregister
SEV	Setze Overflow-Flag im Statusregister
SEZ	Setze Zero-Flag im Statusregister
SLEEP	Setze den µC in den Sleep-Modus
STS	Speichere Register in den Datenspeicher
SUB	Subtrahiere Register ohne Carry-Flag
SUBI	Subtrahiere Wert vom Register
SWAP	Vertausche die Halb-Bytes eines Registers
TST	Teste, ob Register null oder negativ
WDR	Setze den Watchdog-Timer zurück

7. QUELLENANGABEN

Volpe, Safinaz / Volpe, Francesco: AVR-Mikrocontroller-Praxis, Aachen, 2002.
Paulus, Gerhard: Wie sag ich’s meinem AVR?, Dresden, 2002.
Paulus, Gerhard: Assembler ab 0 und 1, Dresden, 2002.

Atmel: 8-bit AVR Microcontroller AT90S8515 Guide, Online im WWW unter URL: http://www.atmel.com, [10.2.2004].
Atmel: 8-bit AVR Instruction Set, Online im WWW unter URL: http://www.atmel.com, [10.2.2004].
Atmel: AVR Assembler User Guide, Online im WWW unter URL: http://www.atmel.com, [10.2.2004].
Schmidt, Gerhard: Anfängerkurs zum Erlernen der Assemblersprache von ATMEL AVR Mikroprozessoren, Online im WWW unter URL: http://www.avr-asm-tutorial.net, [21.1.2004].
N.N.: N.N., Online im WWW unter URL: http://elm-chan.org/docs/avr_e.html, [10.2.2004].