

<b>SREG: Status Register</b>		<b>Bit</b>	<b>Registers and Operands</b>	
<b>C</b>	Carry flag in Status Register	0	<b>PC</b>	Program Counter
<b>Z</b>	Zero flag in Status Register	1	<b>Rd</b>	Destination (and source) Working Register in the register file
<b>N</b>	Negative flag in Status Register	2	<b>Rd(n)</b>	Bit n of the destination Working Register
<b>V</b>	Two's complement overflow indicator	3	<b>Rr</b>	Source Working Register in the register file
<b>S</b>	N \$ V, for signed tests	4	<b>P</b>	I/O-Register and Port-Register
<b>H</b>	Half carry flag in Status Register	5	<b>K</b>	Constant data
<b>T</b>	Transfer bit used by BLD and BST instructions	6	<b>k</b>	Constant address
<b>I</b>	Global interrupt enable/disable flag	7	<b>b</b>	Bit in the Working Register or I/O-Register
<b>Logic Operations</b>			<b>s</b>	Bit in the Status Register
&: AND #: OR \$: exclusive OR			<b>X,Y,Z</b>	Indirect Address Register (X=R27:R26, Y=R29:R28, Z=R31:R30)
<b>Hinweis für die Immediate-Befehle</b>			<b>(X), (Y), (Z)</b>	Contents at indirect address X, Y or Z
* Befehl nicht erlaubt für Register R0...R15			<b>q</b>	Displacement for direct addressing

## Data Transfer Instructions

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
MOV	Rd, Rr	Move Between Working Registers	Rd:=Rr	None	1	
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd := Rr+1:Rr	None	1	
LDI	Rd, K	Load Immediate *	Rd:=K	None	1	
LD	Rd, X	Load Indirect	Rd:=(X)	None	2	
LD	Rd, X+	Load Indirect and Post-increment	Rd:= (X),X:= X+1	None	2	
LD	Rd, -X	Load Indirect and Pre-decrement	X:= X-1,Rd:=(X)	None	2	
LD	Rd, Y	Load Indirect	Rd:=(Y)	None	2	
LD	Rd, Y+	Load Indirect and Post-increment	Rd:= (Y),Y:= Y+1	None	2	
LD	Rd, -Y	Load Indirect and Pre-decrement	Y:= Y-1,Rd:=(Y)	None	2	
LDD	Rd, Y+q	Load Indirect with Displacement	Rd:=(Y+q)	None	2	
LD	Rd, Z	Load Indirect	Rd:=(Z)	None	2	
LD	Rd, Z+	Load Indirect and post-increment	Rd:= (Z),Z:= Z+1	None	2	
LD	Rd, -Z	Load Indirect and pre-decrement	Z:= Z-1,Rd:=(Z)	None	2	
LDD	Rd, Z+q	Load Indirect with Displacement	Rd:=(Z+q)	None	2	
LDS	Rd, k	Load Direct from SRAM	Rd:=(k)	None	2	
ST	X, Rr	Store Indirect	(X):=Rr	None	2	
ST	X+, Rr	Store Indirect and post-increment	(X):=Rr,X:= X+1	None	2	
ST	-X, Rr	Store Indirect and pre-decrement	X:= X-1,(X):= Rr	None	2	
ST	Y, Rr	Store Indirect	(Y):=Rr	None	2	
ST	Y+, Rr	Store Indirect and post-increment	(Y):=Rr,Y:= Y+1	None	2	
ST	-Y, Rr	Store Indirect and pre-decrement	Y:= Y-1,(Y):= Rr	None	2	
STD	Y+q, Rr	Store indirect with Displacement	(Y+q):=Rr	None	2	
ST	Z, Rr	Store Indirect	(Z):=Rr	None	2	
ST	Z+, Rr	Store Indirect and post-increment	(Z):=Rr,Z:= Z+1	None	2	
ST	-Z, Rr	Store Indirect and pre-decrement	Z:= Z-1,(Z):= Rr	None	2	
STD	Z+q, Rr	Store indirect with Displacement	(Z+q):=Rr	None	2	
STS	k, Rr	Store Direct to SRAM	(k):= Rr	None	2	
LPM		Load Program Memory	R0:=(Z)	None	3	
LPM	Rd, Z	Load Program Memory	Rd := (Z)	None	3	
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd := (Z), Z := Z+1	None	3	
SPM		Store Program Memory	(Z) := R1:R0	None	-	
IN	Rd, P	In Port	Rd:=P	None	1	
OUT	P, Rr	Out Port	P:=Rr	None	1	
PUSH	Rr	Push Working Register on Stack	STACK:=Rr	None	2	
POP	Rd	Pop Working Register from Stack	Rd:=STACK	None	2	

## Bit and Bit-Test Instructions

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
SBI	P, b	Set Bit in Port-Register	I/O(P,b):= 1	None	2	
CBI	P, b	Clear Bit in Port-Register	I/O(P,b):=0	None	2	
SBR	Rd, K	Set Bit(s) in Working Register	Rd:= Rd # K	Z,N,V,S	1	
CBR	Rd, K	Clear Bit(s) in Working Register	Rd:= Rd & (0xFF-K)	Z,N,V,S	1	
LSL	Rd	Logical Shift Left	Rd(n+1):= Rd(n), Rd(0):=0, C:=Rd(7)	Z,C,N,V,H	1	

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
LSR	Rd	Logical Shift Right	$Rd(n) := Rd(n+1), Rd(7) := 0, C := Rd(0)$	Z,C,N,V	1	
ROL	Rd	Rotate Left through Carry	$Rd(0) := C, Rd(n+1) := Rd(n), C := Rd(7)$	Z,C,N,V,H	1	
ROR	Rd	Rotate Right through Carry	$Rd(7) := C, Rd(n) := Rd(n+1), C := Rd(0)$	Z,C,N,V	1	
ASR	Rd	Arithmetic Shift Right	$Rd(n) := Rd(n+1), n=0...6, C := Rd(0)$	Z,C,N,V	1	
SWAP	Rd	Swap Nibbles	$Rd(3...0) := Rd(7...4), Rd(7...4) := Rd(3...0)$	None	1	
BSET	S	Flag Set	$SREG(s) := 1$	SREG(s)	1	
BCLR	S	Flag Clear	$SREG(s) := 0$	SREG(s)	1	
BST	Rr, b	Bit Store from Work Register to T	$T := Rr(b)$	T	1	
BLD	Rd, b	Bit Load from T to Work Register	$Rd(b) := T$	None	1	
SEC		Set Carry	$C := 1$	C	1	
CLC		Clear Carry	$C := 0$	C	1	
SEN		Set Negative Flag	$N := 1$	N	1	
CLN		Clear Negative Flag	$N := 0$	N	1	
SEZ		Set Zero Flag	$Z := 1$	Z	1	
CLZ		Clear Zero Flag	$Z := 0$	Z	1	
SEI		Global Interrupt Enable	$I := 1$	I	1	
CLI		Global Interrupt Disable	$I := 0$	I	1	
SES		Set Signed Test Flag	$S := 1$	S	1	
CLS		Clear Signed Test Flag	$S := 0$	S	1	
SEV		Set Two's Complement Overflow	$V := 1$	V	1	
CLV		Clear Two's Complement Overflow	$V := 0$	V	1	
SET		Set T in SREG	$T := 1$	T	1	
CLT		Clear T in SREG	$T := 0$	T	1	
SEH		Set Half-carry Flag in SREG	$H := 1$	H	1	
CLH		Clear Half-carry Flag in SREG	$H := 0$	H	1	
NOP		No Operation		None	1	
SLEEP		Sleep		None	3	
WDR		Watchdog Reset		None	1	
BREAK		Break	For On-Chip Debug Only	None	N/A	

## Arithmetic and logic Instructions

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
ADD	Rd, Rr	Add Two Working Registers	$Rd := Rd + Rr$	Z,C,N,V,H,S	1	
ADC	Rd, Rr	Add with Carry Two Working Registers	$Rd := Rd + Rr + C$	Z,C,N,V,H,S	1	
ADIW#	Rdl, K	Add immediate to Word	$Rdh:Rdl := Rdh:Rdl + K$	Z,C,N,V,S	2	
SUB	Rd, Rr	Subtract Two Working Registers	$Rd := Rd - Rr$	Z,C,N,V,H,S	1	
SUBI	Rd, K	Subtract Constant from Working Register*	$Rd := Rd - K$	Z,C,N,V,H,S	1	
SBC	Rd, Rr	Subtract with Carry Two Working Registers	$Rd := Rd - Rr - C$	Z,C,N,V,H,S	1	
SBCI	Rd, K	Subtract with Carry Constant from Work Registers*	$Rd := Rd - K - C$	Z,C,N,V,H,S	1	
SBIW#	Rdl, K	Subtract Immediate from Word	$Rdh:Rdl := Rdh:Rdl - K$	Z,C,N,V,S	2	
AND	Rd, Rr	Logical AND Working Registers	$Rd := Rd \& Rr$	Z,N,V,S	1	
ANDI	Rd, K	Logical AND Working Register and Constant*	$Rd := Rd \& K$	Z,N,V,S	1	
OR	Rd, Rr	Logical OR Working Registers	$Rd := Rd \# Rr$	Z,N,V,S	1	
ORI	Rd, K	Logical OR Working Register and Constant*	$Rd := Rd \# K$	Z,N,V,S	1	
EOR	Rd, Rr	Exclusive OR Working Registers	$Rd := Rd \$ Rr$	Z,N,V,S	1	
COM	Rd	One's Complement (Inversion)	$Rd := 0xFF - Rd$	Z,C,N,V,S	1	
NEG	Rd	Two's Complement	$Rd := 0x00 - Rd$	Z,C,N,V,H,S	1	
INC	Rd	Increment	$Rd := Rd + 1$	Z,N,V,S	1	
DEC	Rd	Decrement	$Rd := Rd - 1$	Z,N,V,S	1	
TST	Rd	Test for Zero or Minus	$Rd := Rd \& Rd$	Z,N,V,S	1	
CLR	Rd	Clear Working Register	$Rd := 0x00$	Z,N,V,S	1	
SER	Rd	Set Working Register	$Rd := 0xFF$	None	1	
MUL	Rd, Rr	Multiply Unsigned	$R1:R0 := Rd \times Rr$	Z,C	2	X
MULS	Rd, Rr	Multiply Signed	$R1:R0 := Rd \times Rr$	Z,C	2	X
MULSU	Rd, Rr	Multiply Signed with Unsigned	$R1:R0 := Rd \times Rr$	Z,C	2	X
FMUL	Rd, Rr	Fractional Multiply Unsigned	$R1:R0 := (Rd \times Rr) \ll 1$	Z,C	2	X
FMULS	Rd, Rr	Fractional Multiply Signed	$R1:R0 := (Rd \times Rr) \ll 1$	Z,C	2	X
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	$R1:R0 := (Rd \times Rr) \ll 1$	Z,C	2	X

#) Wortoperationen nur mit Wortregistern R25:R24, R27:R26, R29:R28, R31:R30 und 6 Bit Konstante 0..63. adiw R24,13

## Branch Instructions

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
RJMP	k	Relative Jump	PC:=PC+k+1	None	2	
JMP	k	Jump	PC:= k	None	2/3	X
IJMP		Indirect Jump to (Z)	PC:= Z	None	2	
RCALL	k	Relative Subroutine Call	PC:=PC+k+1	None	3	
ICALL		Indirect Call to (Z)	PC:=Z	None	3	
CALL	k	Direct Subroutine Call	PC := k	None	4	X
RET		Subroutine Return	PC:=STACK	None	4	
RETI		Interrupt Return	PC:=STACK, I:=1	I	4	
CPSE	Rd, Rr	Compare, Skip if Equal	If(Rd=Rr) THEN PC:=PC+2 or 3	None	1/2/3	
CP	Rd, Rr	Compare	Rd-Rr	Z,N,V,C,H,S	1	
CPC	Rd, Rr	Compare with Carry	Rd-Rr-C	Z,N,V,C,H,S	1	
CPI	Rd, K	Compare Working Register with Immediate *	Rd-K	Z,N,V,C,H,S	1	
SBRC	Rr, b	Skip if Bit in Working Register is Cleared	If(Rr(b)=0) THEN PC:=PC+2 or 3	None	1/2/3	
SBRS	Rr, b	Skip if Bit in Working Register is Set	If(Rr(b)=1) THEN PC:=PC+2 or 3	None	1/2/3	
SBIC	P, b	Skip if Bit in I/O-Register is Cleared	If(P(b)=0) THEN PC:=PC+2 or 3	None	1/2/3	
SBIS	P,b	Skip if Bit in I/O-Register is Set	If(P(b)=1) THEN PC:=PC+2 or 3	None	1/2/3	
BRBS	s, k	Branch if Status Flag Set	If(SREG(s)=1) THEN PC:=PC+k+1	None	1/2	
BRBC	s, k	Branch if Status Flag Cleared	If(SREG(s)=0) THEN PC:=PC+k+1	None	1/2	
BREQ	k	Branch if Equal	If (Z=1) THEN PC:=PC+k+1	None	1/2	
BRNE	k	Branch if Not Equal	If (Z=0) THEN PC:=PC+k+1	None	1/2	
BRCS	k	Branch if Carry Set	If (C=1) THEN PC:=PC+k+1	None	1/2	
BRCC	k	Branch if Carry Cleared	If (C=0) THEN PC:=PC+k+1	None	1/2	
BRSH	k	Branch if Same or Higher	If (C=0) THEN PC:=PC+k+1	None	1/2	
BRLO	k	Branch if Lower	If (C=1) THEN PC:=PC+k+1	None	1/2	
BRMI	k	Branch if Minus	If (N=1) THEN PC:=PC+k+1	None	1/2	
BRPL	k	Branch if Plus	If (N=0) THEN PC:=PC+k+1	None	1/2	
BRGE	k	Branch if Greater or Equal, Signed	If (N\$V=0) THEN PC:=PC+k+1	None	1/2	
BRLT	k	Branch if Less Than Zero, Signed	If (N\$V=1) THEN PC:=PC+k+1	None	1/2	
BRHS	k	Branch if Half-Carry Flag Set	If (H=1) THEN PC:=PC+k+1	None	1/2	
BRHC	k	Branch if Half-Carry Flag Cleared	If (H=0) THEN PC:=PC+k+1	None	1/2	
BRTS	k	Branch if T-Flag Set	If (T=1) THEN PC:=PC+k+1	None	1/2	
BRTC	k	Branch if T-Flag Cleared	If (T=0) THEN PC:=PC+k+1	None	1/2	
BRVS	k	Branch if OverflowFlag is Set	If (V=1) THEN PC:=PC+k+1	None	1/2	
BRVC	k	Branch if Overflow Flag is Cleared	If (V=0) THEN PC:=PC+k+1	None	1/2	
BRIE	k	Branch if Interrupt Enabled	If (I=1) THEN PC:=PC+k+1	None	1/2	
BRID	k	Branch if Interrupt Disabled	If (I=0) THEN PC:=PC+k+1	None	1/2	

## Assembler-Direktiven

Element	Bedeutung	Beispiel	Erklärung zum Beispiel
;	Einleiten eines Kommentars	; Kommentar bla bla	
.include	Einbinden einer Datei	.include "tn2313def.inc"	Die 2313-Definitionen werden eingebunden
.equ	Deklaration von Konstanten. Wert ist nicht mehr änderbar Im weiteren Quelltext	.equ papagei = 1 .equ fisch = papagei * 2	Der Bezeichner papagei hat nun den Wert 1 Werte können auch durch Ausdrücke (Expression) berechnet werden
.set	Deklaration / Definition einer Variablen. Erneute Zuweisung eines Wertes ist möglich	.set cpuclock = 6000 .set cycle = cpuclock*20/8	
.def	Weist einem Register einen Symbolischem Namen zu	.def tmp = R16	tmp ist R16
.org	Legt Adresse fest ab der folgender Code hinterlegt werden soll	.org \$2A ldi R16,3	Der ldi-Befehl wird in Adresse \$2A Geschrieben
label:	Eine Einsprungmarke	init:	Die Marke (engl. label) init

Darstellung von Werten				
Dezimal	255		10	
Hexadezimal	0xFF	\$FF	0xA	\$0A
Binär	0b11111111		0b00001010	
Oktal	0377		012	

**Hinweise zu .equ, .set:** Der Assembler kann mühselige Rechenarbeit übernehmen: Bestimmte Werte, die öfter im Programm gebraucht werden, z.B. die Frequenz mit der der Controller arbeitet können unter **symbolischen Namen** gespeichert werden. Die Werte lassen sich direkt oder durch einfache Ausdrücke beschreiben. Der Assembler errechnet die Werte und setzt diese an den entsprechenden Stellen ein.

## Assembler-Speicher-Direktiven

Element	Bedeutung	Beispiel	Erklärung zum Beispiel
<b>.cseg</b>	Es folgt ein Code-Segment	<code>.cseg ; Flash</code>	Der folgende Code soll ins Flash
<b>.dseg</b>	Variablen im SRAM	<code>.dseg ; SRAM</code> <code>x: .byte 1 ; 1 Byte Variable</code> <code>feld: .byte 6 ; Feld mit 6 Byte</code>	Speicherbereiche werden so im SRAM reserviert
<b>.eseg</b>	Es folgt ein EEPROM-Segment	<code>.eseg ; EEPROM</code>	Die folgenden Daten sollen ins EEPROM
<b>.db</b>	Liste mit Bytekonstanten	<code>hein: .db 1,2,3,4 ; Zahlen</code> <code>.db 'a','A' ; Zeichen</code> <code>text: .db "Lara" ; String</code>	8bit-Werte für Flash oder EEPROM
<b>.dw</b>	Liste mit Wortkonstanten	<code>rage: .dw 4711 ; Zahl</code> <code>.dw hein ; Adresse</code>	16bit-Werte für Flash oder EEPROM
<b>.byte</b>	Reserviert n Bytes	<code>werte: .byte 10 ; 10 Bytes</code>	Bytebereich in SRAM oder EEPROM

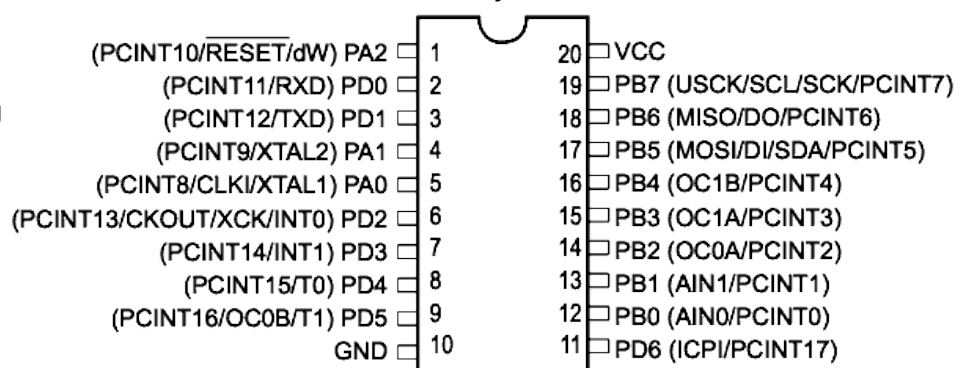
## Assembler Funktionen

Funktion	Bedeutung	Beispiel
<b>LOW(Ausdruck)</b>	liefert Low-Byte = Bit 0-7	<code>LOW(\$123456) -&gt; \$56</code>
<b>HIGH(Ausdruck)</b>	liefert High-Byte = Bit 8-15	<code>HIGH(\$123456) -&gt; \$34</code>
<b>BYTE2(Ausdruck)</b>	liefert High-Byte = Bit 8-15	<code>BYTE2(\$123456) -&gt; \$34</code>
<b>PAGE(Ausdruck)</b>	liefert Bit 16-21	<code>PAGE(\$23456) -&gt; \$2</code>
<b>BYTE3(Ausdruck)</b>	liefert Bit 16-23	<code>BYTE3(\$12345678) -&gt; \$34</code>
<b>BYTE4(Ausdruck)</b>	liefert Bit 24-31	<code>BYTE4(\$12345678) -&gt; \$12</code>
<b>LWRD(Ausdruck)</b>	liefert Low-Wort = Bit 0-15	<code>LWRD(\$12345678) -&gt; \$5678</code>
<b>HWRD(Ausdruck)</b>	liefert High-Wort = Bit 16-31	<code>HWRD(\$12345678) -&gt; \$1234</code>
<b>EXP2(Ausdruck)</b>	liefert 2 <sup>Ausdruck</sup>	<code>EXP2(3) -&gt; 8</code>
<b>LOG2(Ausdruck)</b>	liefert log <sub>2</sub> (Ausdruck) ganzzahlig	<code>LOG2(17) -&gt; 4</code>

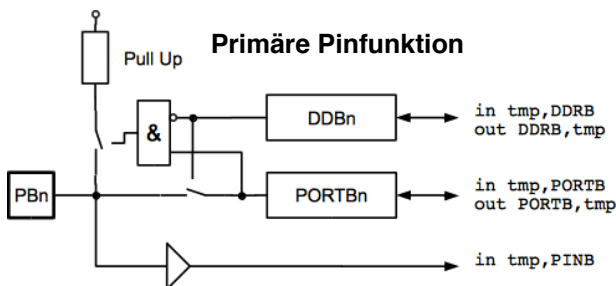
## Umgang mit Ports (Datenrichtung, Ein- Ausgabe)

Der Mikrokontroller hat mehrere I/O Pins (Anschlüsse) die als 8Bit Ports gruppiert sind. Beim ATtiny2313A gibt es PORTB (PBn) und PORTD (PDn). Jedes Pin (z.B. PB0) kann als Ein- oder Ausgang geschaltet werden, die Richtung wird in dem Data Direction FlipFlop (z.B. DDB0) festgelegt. Nach einem Reset sind alle Pins Eingänge weil die FlipFlops den Wert 0 (Initial Value) haben. Um ein Pin als Ausgang zu schalten, muss in seinem DD-FlipFlop der Wert 1 gespeichert werden. Der Ausgangswert wird dann in dem PORT-FlipFlop (z.B. PORTB0) bestimmt.

ATtiny2313



Die Pins können zusätzliche Funktionen haben, siehe die Bezeichner in den Klammern bei der Zeichnung oben, z.B. serielle Schnittstelle.



Hier begnügen wir uns mit der normalen I/O-Funktion, dazu links ein Prinzip-Schaltbild eines Pins. Die einzelnen FlipFlops (z.B. PORTBn) werden zu Registern (z.B. PORTB) zusammengefasst.

Schreibender Zugriff auf PINB mit Bit = 1 invertiert den Zustand in PORTB. Bsp: PINB=1 invertiert PORTB0

DDBn	PORTBn	I/O	Pull-up ein	Kommentar
0	0	Input	nein	Tri-State (Hochohmig)
0	1	Input	ja	Der Ausgang liefert einen Strom für z.B. Taster auf GND
1	0	Output	nein	Push-pull Zero Output (Ausgang ist 0)
1	1	Output	nein	Push-pull One Output (Ausgang ist 1)

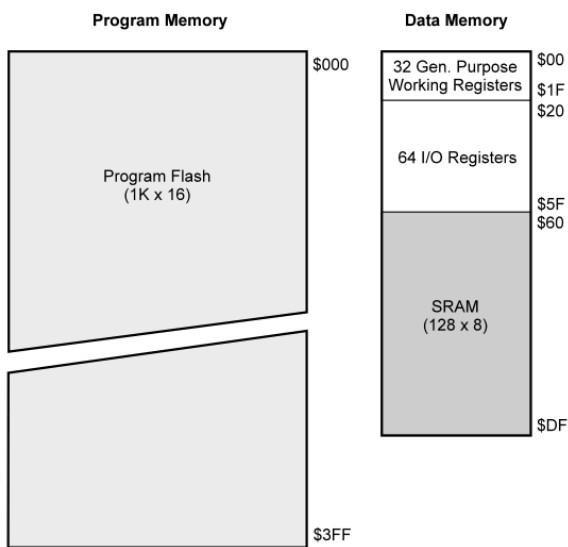
## Assembler-Befehle, die mit Ports zu tun haben

Befehl	Operand	Beschreibung	Beispiel	Erklärung zum Beispiel
<b>in</b>	Rd,P	Einlesen eines Port in Register	in R16,PIND	PortD einlesen in R16
<b>out</b>	P, Rd	Ausgeben eines Register in Port	out PORTB,R16	R16 an PortB ausgeben
<b>sbi</b>	P, b	Setze Bit b in Port P	sbi PORTB,2	Das Bit 2 an PortB setzen
<b>cbi</b>	P, b	Lösche Bit b in Port P	cbi PORTB,2	Das Bit 2 an PortB löschen
<b>sbic</b>	P, b	Überspringe, wenn Bit b in Port P gelöscht	sbic PIND,4	wenn Bit gelöscht ist, wird der folgende Befehl übersprungen. D.h. der folgende Befehl wird ausgeführt wenn Bit gesetzt..
<b>sbis</b>	P, b	Überspringe, wenn Bit b in Port P gesetzt	sbis PIND,4 rjmp testmode	Gehe in Tesmode wenn Pin 4 von PD 0 ist.

### Initialisierungsbeispiel

Assembler	C Code
ldi r16,0b00000011 out PORTB,r16 ldi r16,0b00000100 out DDRB,r16	// PB2 ist Ausgang PB1 und PB0 Eingang mit PullUp PORTB = (1<<PB1) (1<<PB0); // PullUps ein  DDRB = (1<<DDB2); // oder (1<<2)

## Atmel-RISC-8Bit Speicherkonzept am Beispiel des ATtiny2313



Der Speicher ist gemäß der Harvard-Architektur in einen Programm- und einen Datenspeicher unterteilt. Der Programmspeicher ist wortweise (16 Bit) organisiert, d.h. Den 2 KiByte Flash-Speicher stehen nur 1 Ki Adressen \$000..\$3FF gegenüber. Der Datenspeicher ist byteweise organisiert:

\$00 .. \$1F sind die Register R0 bis R31

\$20 .. \$5F ist den I/O Komponenten (Ports, Timer, UART, usw.) zugeordnet

\$60 .. \$DF sind 128 Byte SRAM

**Anmerkung:** Das EEPROM (ohne Abbildung) wird über einen besonderen Mechanismus angesprochen, es gibt spezielle Befehle..

**Wichtig:** Zur Vereinfachung und zum Einsparen von Programmspeicher werden den Registern und I/O Adressen besondere Kurz-Zugriffe zugeordnet.

**Beispiel:** PORTB hat innerhalb des I/O Blocks die Adresse \$18, bezogen auf den geammten Datenbereich die Adresse \$38.

D.h. für manche Daten-Adressen gibt es zwei Zugriffsmöglichkeiten.

## C Datentypen, Konstanten und Variablen

Datentyp	Bit	Dezimalbereich	Bemerkung
<b>unsigned char</b>	8	255	vorzeichenlose kleine Zahlen
<b>char</b>	8	-128..+127	kleine Zahlen mit Vorzeichen
<b>unsigned int</b>	16	65535	
<b>int</b>	16	-32768..+32767	
<b>unsigned long</b>	32	4294967295	*Hinweis: 9600UL
<b>long</b>	32	-2147483648..+2147483647	*Hinweis: 9600L
<b>float</b>	32	$\pm 10^{-37}$ .. $\pm 10^{+38}$	
<b>void</b>			leer oder unbestimmt

Variablen werden vom Compiler in Registern oder im SRAM angelegt. Konstanten und vorbesetzte Variablen werden üblicherweise in einem Prolog aus dem Flash-Speicher in den SRAM kopiert -der Compiler erzeugt eine Befehlsfolge zur Initialisierung. Die Art wie Werte gespeichert werden kann durch Kennwörter manuell beeinflusst werden:

\*Als 32Bit Symbolkonstante muss L bzw. UL angehängt werden damit der Präprozessor nicht mit 16Bit rechnet

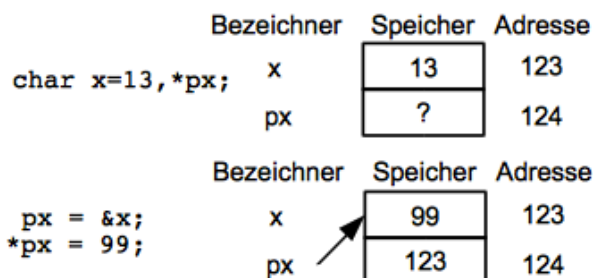
## C Speicherklassen für Variablen

Kennwort	Beschreibung
<b>static</b>	auf fester Adresse (statisch) anlegen. Lokale Variablen behalten ihren Wert auch nach Verlassen des Blocks.
<b>const</b>	Daten sind konstant und dürfen nicht geändert werden. Compiler legt keine Speicherstelle an, sondern setzt die zugewiesenen Werte ein, vgl. Immediate-Befehle (ldi usw.).
<b>volatile</b>	Daten können von außen geändert werden (flüchtig) -z.B. Ports sind so vereinbart.

C-Code	Entsprechung in Assembler
<pre>const unsigned char ZORA = 7; // nicht Variable angelegen unsigned char bernd = ZORA; // global statisch vor main  void main() {     unsigned char temp = 123; // lokal dynamisch in main     PORTB++; // Port B um 1 erhöhen }</pre>	<pre>.def bernd = R20 ldi bernd,7  .def temp = R16 ldi temp,123; Initialisierung in R24, PORTB inc R24 out PORTB,R24</pre>

## Zeiger

<b>Datentyp *Pointername</b>	char *p	Zeiger deklarieren
<b>Pointer = &amp;Variablenname</b>	p = &variable	Zeiger Adresse von Variable zuweisen
<b>*Pointername</b>	*p = 1	Speicherstelle verwenden de-referenzieren



## Call by Reference

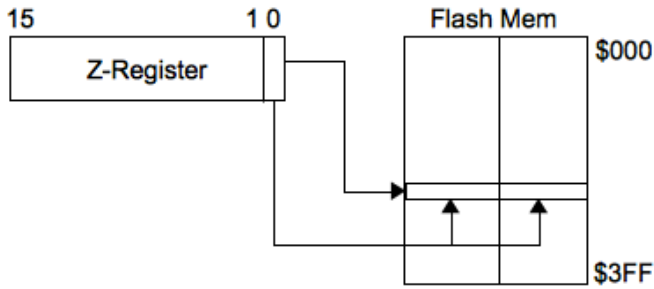
```
void test(char a, char *b){ // Call by Value bei a, Wert wird kopiert, b ist Zeiger auf char Wert
    a=3; // Lokaler Variablen a wird 3 zugewiesenen
    *b=3; // Zeiger b wird dereferenziert und der Speicherstelle 3 zugewiesen
}

void main(){
    char x=7,y=7; // Lokale Variablen x,y in main
    test(x,&y); // der Wert von x und die Referenz auf y (die Adresse von y) als Parameter von test
} // nun ist x immer noch 7 aber y hat den Wert 3!
```

## C-Operatoren

Rang	Operator	Syntax	Richtung	Bedeutung
1	->	Zeiger -> Element	->	Elementselektion
1	[]	Zeiger [Ausdruck], Arrayelement	->	Indizierung
1	()	Ausdruck, Typ (Ausdrucksliste)	->	Funktionsaufruf, Werterzeugung
1	++, --	Lvalue++, Lvalue--	->	Post- In-,De-krement
2	++, --	++Lvalue, --Lvalue	<-	Prä- In-,De-krement
2	~, !	~Ausdruck, !Ausdruck	<-	Komplement, Negation
2	-, +	-Ausdruck, +Ausdruck	<-	unäres Minus, Plus
2	()	(Typ) Ausdruck	<-	Typkonvertierung
2	&	&Variablenbezeichner	<-	unär: Adressoperator
2	*	*Zeigerbezeichner	<-	unär: Indirektionsoperator
2	sizeof()	sizeof(Bezeichner)	<-	Operandengröße in Byte
3	*, /	Ausd. * Ausd., Ausd. / Ausd.	->	Multiplikation, Division
3	%	Ausdruck % Ausdruck	->	Modulo, Divisionsrest
4	+, -	Ausd. + Ausd., Ausd. - Ausd.	->	Addition, Subtraktion
5	<<, >>	Ausd. << Ausd., Ausd. >> Ausd.	->	Linksshift, Rechtsshift
6	<, >	Ausd. < Ausd., Ausd. > Ausd.	->	kleiner, größer als
6	<=, >=	Ausd. <= Ausd., Ausd. >= Ausd.	->	kleiner, größer gleich als
7	==, !=	Ausd. == Ausd., Ausd. != Ausd.	->	gleich, ungleich
8	&	Ausdruck & Ausdruck	->	bitweises Und
9	^	Ausdruck ^ Ausdruck	->	bitweises EXOR
10		Ausdruck   Ausdruck	->	bitweises Oder
11	&&	Ausdruck && Ausdruck	->	logisches Und
12		Ausdruck    Ausdruck	->	logisches Oder
13	?:	Ausdruck ? Ausdruck : Ausdruck	<-	Bedingte Zuweisung
14	=	Lvalue = Ausdruck	<-	einfache Zuweisung
14	+=, -=, *=, /=, %=	Lvalue op Ausdruck	<-	Kombinierte Zuweisung (arithmetisch)
14	<<=, >>=, &=, ^=,  =	Lvalue op Ausdruck	<-	Kombinierte Zuweisung (bitweise)
15	,	Ausdruck , Ausdruck	->	Folge von Ausdrücken

## Assembler: Adressierung von Konstanten im Flash



Im Programm-Speicher können Konstanten abgelegt werden mit der Assembler-Direktive `.db`. Mit dem LPM-Befehl kann ein Byte aus dem Programmspeicher via Z-Register (Z=R31:R30) in ein Register kopiert werden. Da der Programmspeicher 16 Bit organisiert ist wird mit dem LSB des Z-Registers zwischen dem High- und Low-Byte einer Programm-Speicher-Adresse unterschieden.

```
ldi R31, high(kzeiger*2) ; High-Byte in Z
ldi R30, low(kzeiger*2) ; Low-Byte in Z
lpm R16, Z+ ; lade erste Konstante und Z++
; usw.
kzeiger: .db "ABI",0 ; die Konstante
```

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
<b>lpm</b>		Load Program Memory	R0:=(Z)	None	3	
<b>lpm</b>	Rd, Z	Load Program Memory	Rd := (Z)	None	3	X
<b>lpm</b>	Rd, Z+	Load Program Memory and Post-Inc	Rd := (Z), Z := Z+1	None	3	X
<b>spm</b>		Store Program Memory	(Z) := R1:R0	None	-	X

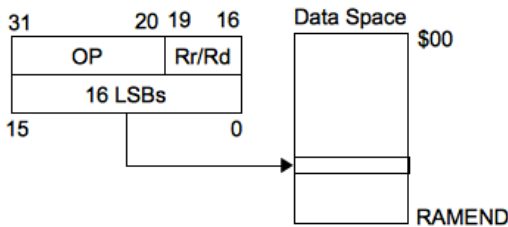
### C: Speicherung und Adressierung von Flashkonstanten

```
#include <avr/pgmspace.h>
//Konstante im Flash speichern
const char PROGMEM einByte = 0x22;
// Lesen mit besonderen Funktionen
char bla = pgm_read_byte(&einByte);
```

Ergebnis	Aufruf	Beschreibung
8 Bit	<code>pgm_read_byte(&amp;Name)</code>	Byte aus Programmspeicher
16 Bit	<code>pgm_read_word(&amp;Name)</code>	Wort aus Programmspeicher
32 Bit	<code>pgm_read_dword(&amp;Name)</code>	Dwort aus Programmspeicher
	<code>strcpy_P(Ziel,Quelle)</code>	Kopiert String aus Flash in SRAM

## Adressierung von Variablen im SRAM

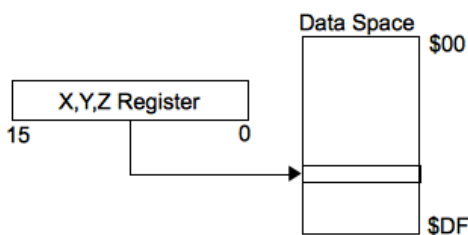
### Data Direct



Mnemonic	Operands	Description	Operation	Flags	# Clocks
<b>lds</b>	Rd, k	Load Direct from SRAM	Rd:=(k)	None	2
<b>sts</b>	k, Rr	Store Direct to SRAM	(k):= Rr	None	2

**Beispiel:** `lds R16, 0x60` ; Lade Inhalt des ersten SRAM-Bytes in R16. Dieser Befehl braucht zwei Wörter Platz wegen der 16 Bit SRAM Adresse (es gibt auch größere Controller)

### Data Indirect



Mnem	Operand	Description	Operation	Flags	Clocks
<b>ld</b>	Rd, X	Load Indirect	Rd:=(X)	None	2
<b>ld</b>	Rd, X+	Load Indirect and Post-increment	Rd:= (X),X:= X+1	None	2
<b>ld</b>	Rd, -X	Load Indirect and Pre-decrement	X:= X-1,Rd:=(X)	None	2
<b>st</b>	X, Rr	Store Indirect	(X):=Rr	None	2
<b>st</b>	X+, Rr	Store Indirect and post-increment	(X):=Rr,X:= X+1	None	2
<b>st</b>	-X, Rr	Store Indirect and pre-decrement	X:= X-1,(X):= Rr	None	2

Möglich mit (X=R27:R26, Y=R29:R28, Z=R31:R30) Register

**Beispiel:** Tabelle `tab[2] = {2,4}` im SRAM anlegen:

Assembler	C-Code
<pre>ldi R27, HIGH(tab); Prolog zur ldi R26, LOW(tab); Initialisierung ldi R24,2 ;lade 2 in R24 st x+,R24 ;speichere indirekt via x und x++ ldi R24,4 ;lade 4 in R24 st x+,R24 ;speichere 4 in [tab+1] und x++ .dseg; Datenbereich tab: .byte 2; Feld mit 2 Byte</pre>	<pre>unsigned char tab[2] = {2,4}; // mit Anfangswerten</pre>

## Data Indirect with Displacement

Mnem	Operands	Description	Operation	Flags	Clocks
<b>ldd</b>	Rd, Y+q	Load Indirect with Displacement	Rd:=(Y+q)	None	2
<b>std</b>	Y+q, Rr	Store indirect with Displacement	(Y+q):=Rr	None	2

Möglich mit (Y=R29:R28, Z=R31:R30) Register. Bei komplexeren Datenstrukturen z.B. Listen von Records bzw. Objekten wäre es praktisch, zu der Basisadresse des Objekts (Zeiger, Pointer auf Record bzw. Objekt) ein

bestimmtes Feld bzw. Attribut ansprechen zu können. Der Versatz der Feld- bzw. Attributadresse wird auch als "Displacement" bezeichnet. **Beachte:** Displacement hat nur 6Bit -> 0 .. 63

## Interrupts

Zwischen Interruptereignis und Verzweigung zur Interruptadresse in der Tabelle vergehen mindestens 4 Takte (Befehle mit mehreren Takten werden erst beendet).

### ATtiny2313 Interrupttabelle (Auszug)

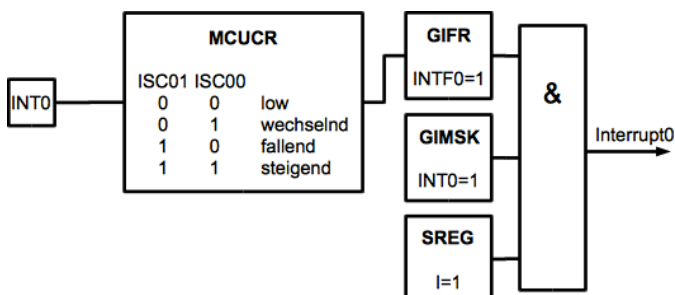
Adresse	Interrupt	Assembler-Bezeichner	C-Bezeichner
0x001	External Interrupt Request 0	INT0addr	INT0_vect
0x002	External Interrupt Request 1	INT1addr	INT1_vect
0x003	Timer/Counter1 Capture Event	ICP1addr	TIMER1_CAPT_vect
0x004	Timer/Counter1 Compare Match A	OC1Aaddr	TIMER1_COMPA_vect
0x005	Timer/Counter1 Overflow	OVF1addr	TIMER1_OVF_vect
0x006	Timer/Counter0 Overflow	OVF0addr	TIMER0_OVF_vect
0x007	UART, Rx Complete	URXCaddr	UART_RX_vect
0x008	UART Data Register Empty	UDREaddr	UART_UDRE_vect
0x009	UART, Tx Complete	UTXCaddr	UART_TX_vect
0x00A	Analog Comparator	ACladdr	ANA_COMP_vect
0x00B	Pin Change Interrupt Request 0	PCladdr	PCINT_vect
0x00C	Timer/Counter1 Compare Match B	OC1Baddr	TIMER1_COMPB_vect
0x00D	Timer/Counter0 Compare Match A	OC0Aaddr	TIMER0_COMPA_vect
0x00E	Timer/Counter0 Compare Match B	OC0Baddr	TIMER0_COMPB_vect

.equ INT\_VECTORS\_SIZE = 21 ; size in words

### Assemblerbefehle, die mit Interrupts zu tun haben

Mnemonic	Operands	Description	Operation	Flags	# Clocks
<b>sei</b>		Global Interrupt Enable	I:=1	I	1
<b>di</b>		Global Interrupt Disable	I:=0	I	1
<b>reti</b>		Interrupt Return	PC:=STACK, I:=1	I	4
<b>in</b>	Rd, SREG	In Port	Rd:=SREG	None	1
<b>out</b>	SREG, Rr	Out Port	SREG:=Rr	ALL	1
<b>brie</b>	k	Branch if Interrupt Enabled	If (I=1) THEN PC:=PC+k+1	None	1 oder 2
<b>brid</b>	k	Branch if Interrupt Disabled	If (I=0) THEN PC:=PC+k+1	None	1 oder 2
<b>sleep</b>		Sleep		None	3
<b>wdr</b>		Watchdog Reset		None	1

### Externe Interrupts



Bit	7	6	5	4	3	2	1	0
<b>MCUCR</b>	-	-	SE	SM	ISC11	ISC10	ISC01	ISC00
<b>GIFR</b>	INTF1	INTF0	-	-	-	-	-	-
<b>GIMSK</b>	INT1	INT0	-	-	-	-	-	-

Externe Interrupts können durch die INT0 (PD2) und INT1 (PD3) Pins ausgelöst werden. Sind die Pins als Ausgang geschaltet, lassen sich hierdurch auch Softwareinterrupts realisieren. Ob Low-Pegel oder steigende bzw. fallende Flanke

den Interrupt auslösen wird über die Bits ISC des MCU Control Register MCUCR definiert. Ein Interrupt wird ausgeführt, wenn er im General Interrupt MaSK Register GIMSK freigeschaltet wurde und im SREG eine generelle Interruptfreigabe existiert.



### Beispiel -externer Interrupt INT0 mit fallender Flanke:

Assembler	C-Code
<pre>.include "tn2313def.inc" rjmp reset; .org INT0addr; Adresse des Vektors rjmp isrINT0; Behandeln von INT0 Interrupt  .org INT_VECTORS_SIZE; weiter nach Interrupttabelle reset: ; Initialisierung ldi r16,low(RAMEND) out SPL,r16 ; Set Stack Pointer to top of RAM ldi R16,1&lt;&lt;ISC01 ; setze Bit ISC01 out MCUCR,R16 ldi R16,1&lt;&lt;INT0; setze Bit INT0 out GIMSK, R16 sei ; Enable interrupts ; -- weitere Befehle --  isrINT0: push r16; Register retten in r16,SREG; Statusregister laden push r16; Statusregister retten ; -- weitere Befehle -- pop r16; Statusregister vom Stapel holen out SREG,r16; Statusregister wieder herstellen pop r16 reti; Interrupt behandelt</pre>	<pre>#include &lt;avr/io.h&gt; // Definitionen laden #include &lt;avr/interrupt.h&gt;  ISR(INT0_vect){ // Befehle für Interrupt-Behandlung }  void main(){ // Befehle für Initialisierung MCUCR  = (1 &lt;&lt; ISC01); // INT0 fallende Flanke GIMSK  = (1 &lt;&lt; INT0); // INT0 freigeschaltet sei(); alle Interrupts frei while (1){ // weiteres Hauptprogramm } }</pre>

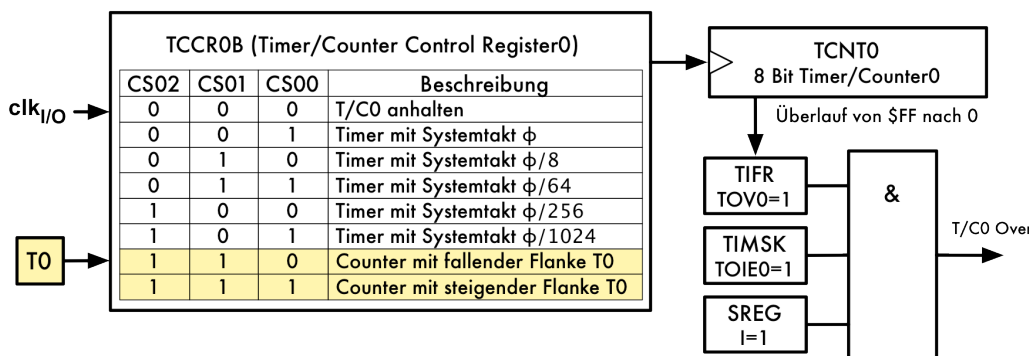
### Timer / Zähler

Der ATtiny2313 verfügt über zwei Zähler, die auch als Timer verwendet werden können. Die vielen möglichen Betriebsmodi sind in der Hersteller-Dokumentation ausführlich beschrieben. Folgend werden nur zwei einfache Verfahren zur Interrupterzeugung behandelt.

#### 8 Bit Timer / Counter0

Bit		7	6	5	4	3	2	1	0
<b>TCNT0</b>	Timer/Counter 0	MSB							LSB
<b>OCROA</b>	Output Compare Register A	MSB							LSB
<b>OCROB</b>	Output Compare Register B	MSB							LSB
<b>TCCROA</b>	Timer/Counter Control Register A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
<b>TCCROB</b>	Timer/Counter Control Register B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
<b>TIFR</b>	Timer/Counter Interrupt Flag Register	TOV1	OCF1A	OCF1B	-	ICF1	OCF0B	TOV0	OCF0A
<b>TIMSK</b>	Timer/Counter Interrupt Mask Register	TOIE1	OCIE1A	OCIE1B	-	TICIE1	OCIE0B	TOIE0	OCIE0A

#### 8 Bit Timer / Counter0 Overflow-Interrupt



Der **TCNT0** hat 8 Bit, und zählt aufwärts. Mit dem Timer Control Register 0 **TCCROB** kann die Art des Taktsignals bestimmt werden. Neben verschiedenen Teilverhältnissen des Systemtaktes =  $clk_{I/O}$ , kann auch mit externem Signal an PD4 gezählt werden. Ein Überlauf (Zählstand \$FF -> 0) kann ein Interrupt **Timer/Counter 0 Overflow** auslösen.

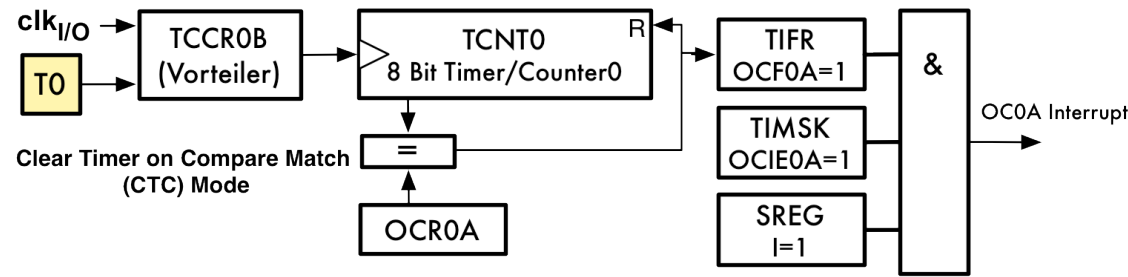
Dazu muss im Timer/Counter-Interruptmaskenregister **TIMSK** das Bit **TOIE0** (Timer Overflow Interrupt Enable) und die generelle Interruptfreigabe gesetzt sein. Durch Vorstellen des Timers in der ISR kann die Zeit verkürzt werden.

$$\text{Interruptzeit} = (256\text{-Vorsteller}) * \text{Vorteiler} / \text{Systemtaktfrequenz} = (256\text{-Vorsteller}) * \text{Vorteiler} * \text{Systemtaktdauer}$$

#### C-Initialisierung

```
TCCR0B |= ?; // Vorteiler 1: /1; 2: /8; 3: /64; 4: /256; 5: /1024
TIMSK |= 1<<TOIE0; // TimerOverflowInterruptEnable0
```

### 8 Bit Timer / Counter0 Clear Timer on Compare Match (CTC) Mode (Auto Reload)



Im **CTC** Modus wird der Wert des **TCNT0** mit dem Register **OCR0A** verglichen. Bei Gleichheit wird der Zähler wieder auf 0 gesetzt und es kann dabei ein **OC0A** Interrupt ausgelöst werden.

$$\text{Interruptzeit} = (1 + \text{OCR0A}) * \text{Vorteiler} / \text{Systemtaktfrequenz} = (1 + \text{OCR0A}) * \text{Vorteiler} * \text{Systemtaktperiode}$$

#### C-Initialisierung

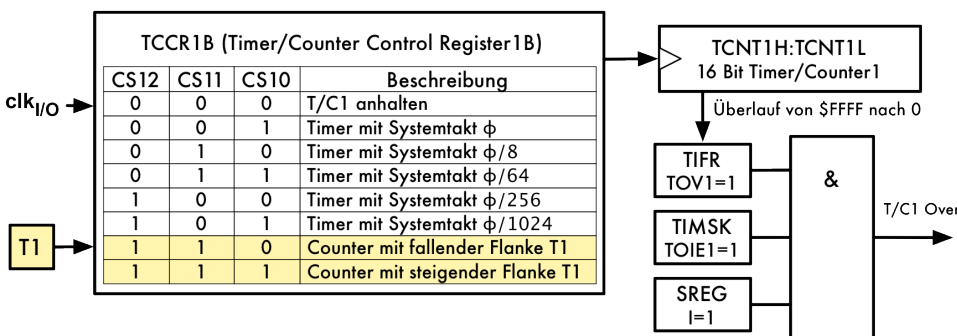
```
TCCR0A |= 1 << WGM01; // Timer im CTC-Mode
TCCR0B |= ?; // Vorteiler 1: /1; 2: /8; 3: /64; 4: /256; 5: /1024
TIMSK |= 1 << OCIE0A; // Timer/Counter0 Output Compare Match A Interrupt Enable
OCR0A = ?; // Wert bei dem der Timer wieder auf 0 gesetzt wird
```

### 16 Bit Timer / Counter1

Die 16 Bit Register des Timers sind aus Assembersicht Doppelregister, d.h. es muss jeweils ein 8 Bit Low- und ein High-Register benutzt werden. Die Reihenfolge des Lesens und Schreibens ist wichtig, siehe Herstellerdoku. Lesen: Zuerst Low, dann High. Schreiben: Zuerst High dann Low. In C ist eine Unterscheidung zwischen Low und High nicht notwendig.

	Bit	7	6	5	4	3	2	1	0
<b>ICR1L</b>	Timer/Counter1 Input Capture Register Low								LSB
<b>ICR1H</b>	Timer/Counter1 Input Capture Register High	MSB							
<b>OCR1AL</b>	Timer/Counter1 Output Compare Register A Low								LSB
<b>OCR1AH</b>	Timer/Counter1 Output Compare Register A High	MSB							
<b>TCNT1L</b>	Timer/Counter1 Low								LSB
<b>TCNT1H</b>	Timer/Counter1 High	MSB							
<b>TCCR1A</b>	Timer/Counter1 Control Register A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
<b>TCCR1B</b>	Timer/Counter1 Control Register B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
<b>TIFR</b>	Timer/Counter Interrupt Flag Register	TOV1	OCF1A	OCF1B	-	ICF1	OCF0B	TOV0	OCF0A
<b>TIMSK</b>	Timer/Counter Interrupt Mask Register	TOIE1	OCIE1A	OCIE1B	-	ICIE1	OCIE0B	TOIE0	OCIE0A

### 16 Bit Timer / Counter1 Overflow-Interrupt



Gleiches Prinzip wie bei 8-Bit Timer, nur der Zähler hat mehr Bit und bei Assembler muss die Reihenfolge des Lesens und Schreibens auf die Low und High Teile der Doppelregister beachtet werden siehe oben.

$$\text{Interruptzeit} = (65536\text{-Vorsteller}) * \text{Vorteiler} / \text{Systemtaktfrequenz} = (65536\text{-Vorsteller}) * \text{Vorteiler} * \text{Systemtaktperiode}$$

#### C-Initialisierung

```
TCCR1B |= ?; // Vorteiler 1: /1; 2: /8; 3: /64; 4: /256; 5: /1024
TIMSK |= 1 << TOIE1; // TimerOverflowInterruptEnable1
```

### 16 Bit Timer / Counter1 Clear Timer on Compare Match (CTC) Mode (Auto Reload)

$$\text{Interruptzeit} = (1 + \text{OCR1A}) * \text{Vorteiler} / \text{Systemtaktfrequenz} = (1 + \text{OCR1A}) * \text{Vorteiler} * \text{Systemtaktperiode}$$

#### C-Initialisierung

```
TCCR1B |= 1 << WGM12; // Timer im CTC-Mode mit OCR1A als Top
TCCR1B |= ?; // Vorteiler 1: /1; 2: /8; 3: /64; 4: /256; 5: /1024
TIMSK |= 1 << OCIE1A; // Timer/Counter1 Output Compare Match A Interrupt Enable
OCR1A = ?; // Wert bei dem der Timer wieder auf 0 gesetzt wird
```

## Assembler Makros

Makros sind Befehlsfolgen, die vom Assembler in den Code bei Aufruf eingebaut werden Beispiel:

```
; Definition eines Makros
.MACRO addi ;Aufruf: addi R16..R31, Konstante
subi @0,-@1 ;subtrahiere das Komplement
.ENDM
; Aufruf wie ein Befehl dabei Makroexpansion
addi R16,12 ; liefert subi R16,-12
```

## Präprozessor Befehle

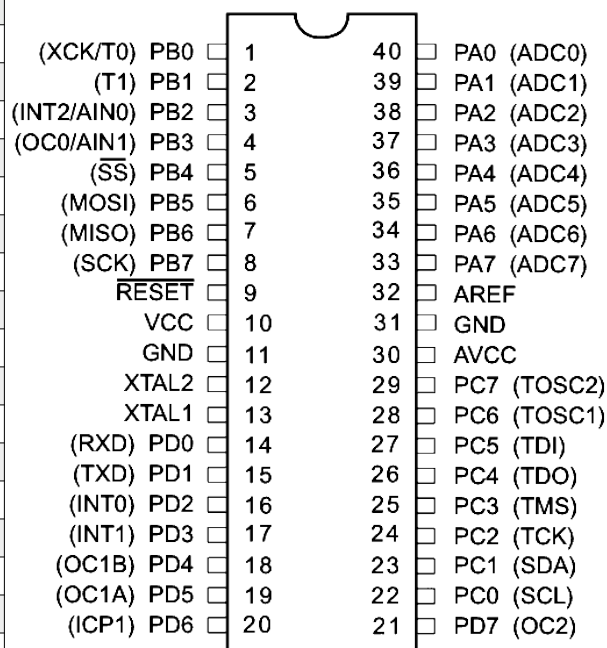
Befehl	Bedeutung
#include	Fügt Text aus einer anderen Quelltextdatei ein
#define	Definiert ein Makro
#undef	Entfernt ein Makro
#if	Bedingte Compilierung in Abhängigkeit der nachstehenden Bedingung
#ifdef	Bedingte Compilierung in Abhängigkeit, ob ein Makroname definiert ist
#ifndef	Bedingte Compilierung in Abhängigkeit, ob ein Makroname nicht definiert ist
#else	Alternative Compilierung, wenn die Bedingung des vorstehenden #if, #elif, #ifdef oder #ifndef nicht erfüllt ist
#elif	Alternative Compilierung in Abhängigkeit der nachstehenden Bedingung (eine Art Kombination aus #else und #if)
#endif	Beendet den Block mit der bedingten Compilierung
#line	Liefert eine Zeilennummer für Compiler-Meldungen
defined	Liefert eine 1, wenn der nachstehende Makroname definiert ist, sonst eine 0. Kann nur zusammen mit #if und #elif verwendet werden.
# operator	Ersetzt innerhalb eines Makros einen Makroparameter durch eine konstante Zeichenkette, die den Wert des Parameters enthält
## operator	Erzeugt ein einzelnes Token aus zwei hintereinander stehenden Tokens
#pragma	Gibt Compiler- und System-abhängige Informationen an den Compiler
#error	Erzeugt einen Fehler während der Compilierung mit der angegebenen Fehlermeldung

## Anhang ATmega 16

Adresse	Interrupt	Ass-Bezeichner	C-Bezeichner
0x002	External Interrupt Request 0	INT0addr	INT0_vect
0x004	External Interrupt Request 1	INT1addr	INT1_vect
0x006	Timer/Counter2 Compare Match	OC2addr	TIMER2_COMP_vect
0x008	Timer/Counter2 Overflow	OVF2addr	TIMER2_OVF_vect
0x00A	Timer/Counter1 Capture Event	ICP1addr	TIMER1_CAPT_vect
0x00C	Timer/Counter1 Compare Match A	OC1Aaddr	TIMER1_COMPA_vect
0x00E	Timer/Counter1 Compare Match B	OC1Baddr	TIMER1_COMPB_vect
0x010	Timer/Counter1 Overflow	OVF1addr	TIMER1_OVF_vect
0x012	Timer/Counter0 Overflow	OVF0addr	TIMER0_OVF_vect
0x014	Serial Transfer Complete	SPIaddr	SPI_STC_vect
0x016	USART, Rx Complete	URXCaddr	USART_RXC_vect
0x018	USART Data Register Empty	UDREaddr	USART_UDRE_vect
0x01A	USART, Tx Complete	UTXCaddr	USART_TXC_vect
0x01C	ADC Conversion Complete	ADCCaddr	ADC_vect
0x01E	EEPROM Ready	ERDYaddr	EE_RDY_vect
0x020	Analog Comparator	ACIaddr	ANA_COMP_vect
0x022	2-wire Serial Interface	TWIaddr	TWI_vect
0x024	External Interrupt Request 2	INT2addr	INT2_vect
0x026	Timer/Counter0 Compare Match	OC0addr	TIMER0_COMP_vect
0x028	Store Program Memory Ready	SPMRaddr	SPM_RDY_vect

```
.equ INT_VECTORS_SIZE = 42 ; size in words
```

## PDIP



**Bsp. ATtiny2313 Timer / Counter0 als Zähler mit ext. Signal fallende Flanke, Ausg. auf PortB**

Assembler	C-Code
<pre> rjmp reset; .org INT_VECTORS_SIZE; nach Interrupttabelle reset: ldi r16,low(RAMEND) out SPL,r16 ser R16 ; R16 &lt;- \$FF out DDRB,R16; PortB als Ausgang ldi R16,0b110; ext Takt fallende Flanke out TCCR0B,R16; neues Steuerregister loop: in R16,TCNT0; laufenden Zählerstand out PORTB,R16; auf PortB ausgeben rjmp loop                     </pre>	<pre> #include &lt;avr/io.h&gt; // Definitionen laden #include &lt;avr/interrupt.h&gt;  void main(){   DDRB = 0xff; // PortB als Ausgang   TCCR0B  = 6; // 110 ext. Takt fallende Flanke   while (1){     PORTB = TCNT0; // Zähler ausgeben   } }                     </pre>

**ASCII-Tabelle**

Dez	Hex	Zeichen	Dez	Hex	Zeichen	Dez	Hex	Zeichen	Dez	Hex	Zeichen	Dez	Hex	Zeichen
0	0	Nul	26	1A	SUB	52	34	4	78	4E	N	104	68	h
1	1	SOH	27	1B	ESC (Escape)	53	35	5	79	4F	O	105	69	i
2	2	STX	28	1C	FS	54	36	6	80	50	P	106	6A	j
3	3	ETX	29	1D	GS	55	37	7	81	51	Q	107	6B	k
4	4	EOT (End Of Transmission)	30	1E	RS	56	38	8	82	52	R	108	6C	l
5	5	ENQ	31	1F	US	57	39	9	83	53	S	109	6D	m
6	6	ACK	32	20	SP (Space)	58	3A	:	84	54	T	110	6E	n
7	7	BELL (Klingel)	33	21	!	59	3B	;	85	55	U	111	6F	o
8	8	BS (Back Space)	34	22	"	60	3C	<	86	56	V	112	70	p
9	9	Hor. Tabulator	35	23	#	61	3D	=	87	57	W	113	71	q
10	0A	LF (Line Feet)	36	24	\$	62	3E	>	88	58	X	114	72	r
11	0B	Ver. Tabulator	37	25	%	63	3F	?	89	59	Y	115	73	s
12	0C	FF (Form Feet)	38	26	&	64	40	@	90	5A	Z	116	74	t
13	0D	CR (Carriage Return)	39	27		65	41	A	91	5B	[	117	75	u
14	0E	SO	40	28	(	66	42	B	92	5C	\	118	76	v
15	0F	SI	41	29	)	67	43	C	93	5D	]	119	77	w
16	10	DLE	42	2A	*	68	44	D	94	5E	^	120	78	x
17	11	DC1	43	2B	+	69	45	E	95	5F	_	121	79	y
18	12	DC2	44	2C	,	70	46	F	96	60		122	7A	z
19	13	DC3	45	2D	-	71	47	G	97	61	a	123	7B	{
20	14	DC4	46	2E	.	72	48	H	98	62	b	124	7C	
21	15	NAK	47	2F	/	73	49	I	99	63	c	125	7D	}
22	16	SYN	48	30	0	74	4A	J	100	64	d	126	7E	~
23	17	ETB	49	31	1	75	4B	K	101	65	e	127	7F	DEL (Delete)
24	18	CAN	50	32	2	76	4C	L	102	66	f			
25	19	EM	51	33	3	77	4D	M	103	67	g			

**Zylonenaug Schema**

