

Mikrocontroller

1. Ziel des Kapitels

- Einführung in die uC vermitteln.
- Verständnis für HW / SW Co-design erarbeiten.

2. Vorgangsweise

- Beschäftigung mit uC (hauptsächlich AVR uC).
- Lesen von Artikeln und Fachliteratur sowie user manuals im Unterricht und zu Hause, in Englisch und in Deutsch.
- Erstellen einer eigenen Mitschrift.
- Installieren einer IDE (AVR Studio 7).
- Programmierung in C und teilweise in Assembler.
- Simulation von uC Programmen.
- Jede Woche eine Mitarbeitskontrolle in schriftlicher Form bezüglich der Theorie.

Ziel:

Anhand von kleinen Beispielen sollte vermittelt werden:

- Was mit einem Microcontroller möglich ist,
- Wie es danach realisierbar ist,
- Welche Probleme lösbar sind,
- Wo die Grenzen liegen.

3. Was ist ein Mikrocontroller?

<http://www.mikrocontroller.net/articles/Mikrocontroller>

<http://de.wikipedia.org/wiki/Mikrocontroller>

http://de.wikipedia.org/wiki/Liste_von_Mikrocontrollern

(ARM, Atmel, „Motorola“ = Freescale, Infineon, Intel, NXP, ...)

4. Unterschiede zwischen uC und uP

Zusammen mit einem Prozessor werden auch Peripherie Komponenten auf einem Chip vereint.

Diese sind im Wesentlichen:

- ❖ Programm und Arbeitsspeicher
- ❖ UART (universal asynchronous receiver transmitter), I2C bus
- ❖ LIN, CAN, USB controller
- ❖ ADC
- ❖ PWM Ausgänge
- ❖ Gemultiplexte Ports (= Ein- / Ausgänge)
- ❖ Architektur
 - Neumann Architektur
http://en.wikipedia.org/wiki/Von_neumann_architecture
 - Harvard Architektur
http://en.wikipedia.org/wiki/Harvard_architecture

Man spricht in diesem Zusammenhang auch von einem Ein Chip Computersystem. Ein etwas weiter gefasster Begriff dafür ist auch system on chip (SoC).

5 Beispiel an der Schule

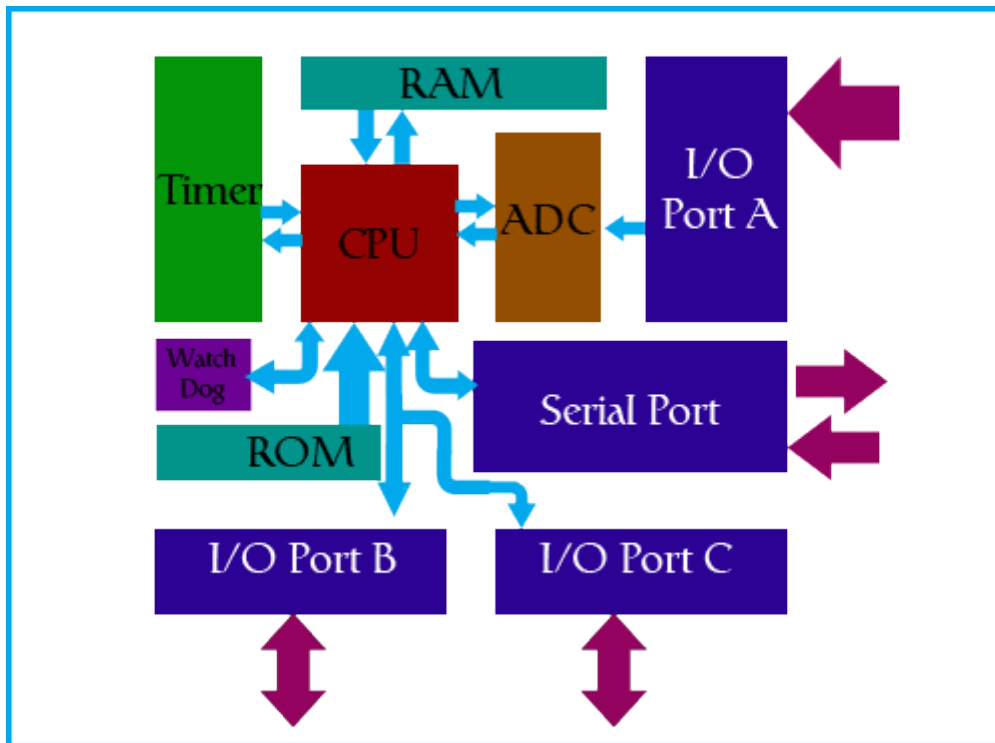
http://shop.chip45.com/epages/es10644620.sf/de_DE/?ObjectPath=/Shops/es10644620/Products/Crumb644-1.1/SubProducts/crumb644-1.1-2

Innerer Aufbau eines Mikrocontrollers

Eine einfache und intuitive Darstellung des Aufbaus eines Mikrocontrollers ist unter dem folgenden Link zu finden:

http://www.pictutorials.com/what_is_microcontroller.htm

Einfache Betrachtung



Wichtige Komponenten und Peripherals:

In Partnerarbeit sollen die wichtigsten Informationen zu den nachfolgenden Komponenten eines Mikrocontrollers aus dem oben angegebenen Kapitel zu entnehmen

- 1) CPU
- 2) RAM / ROM
- 3) I/O PORTS
- 4) TIMER
- 5) WATCHDOG:
Implementation:
Simple counter implementation
Seed / key principle
- 6) ADC:
- 7) SERIAL PORT:

Zusammenfassung

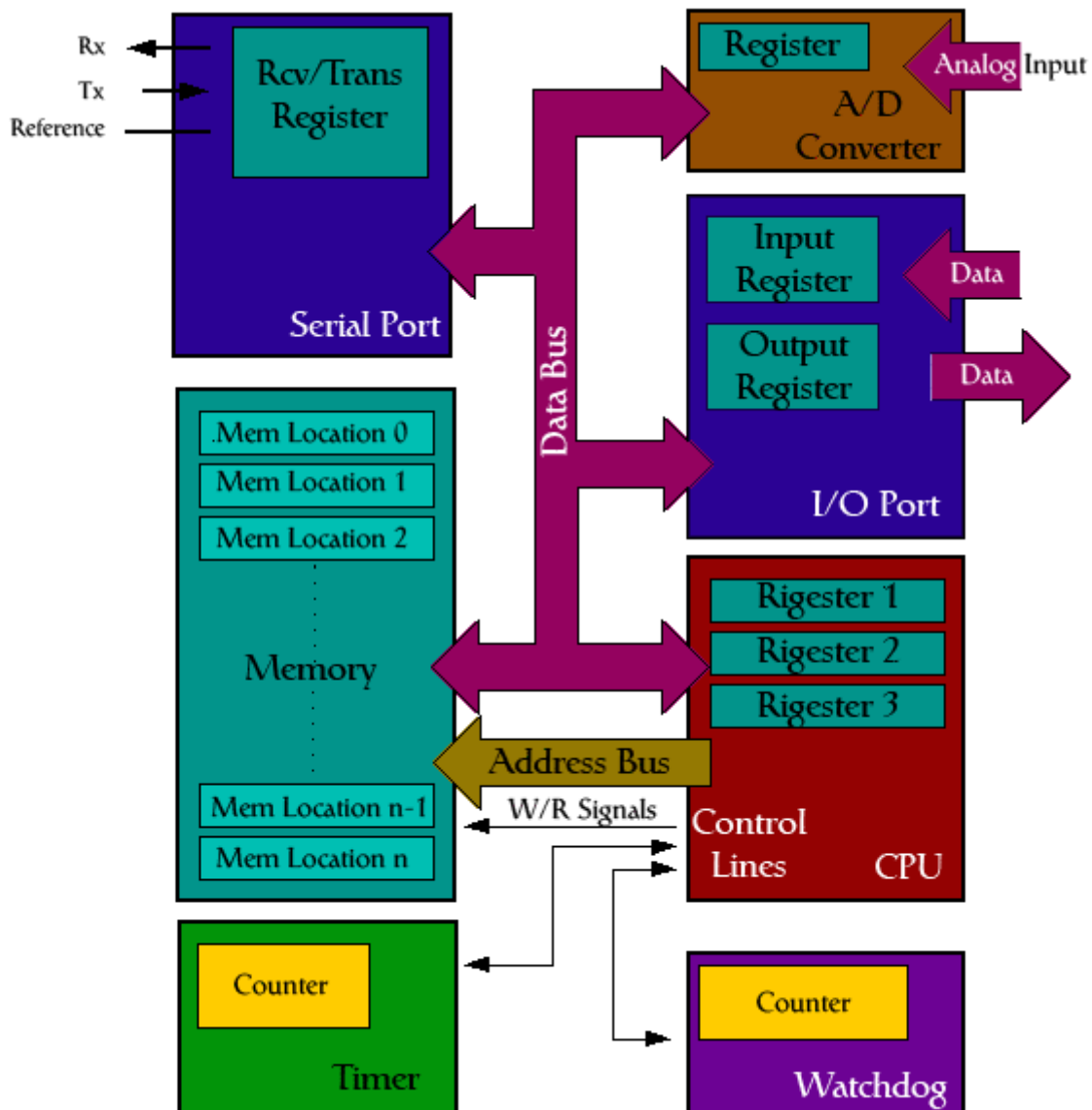


Abb 1: Innerer Aufbau eines Mikrocontrollers (siehe PIC tutorial), unter Register sind natürlich Register zu verstehen.

Aufgabenstellung:

Zur Vertiefung in die Thematik zu Mikrocontroller-Architekturen sind die folgenden Zusatzmaterialien zu empfehlen.

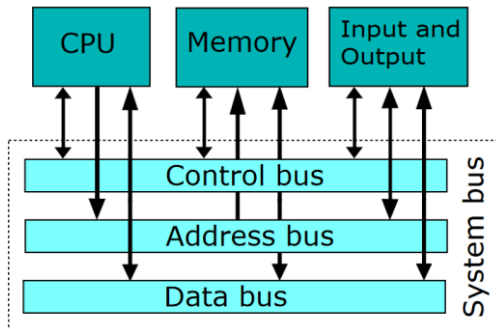
- 1) Mikrocontroller, Kapitel 7 aus Taschenbuch Mikroprozessortechnik (TB MPT)
- 2) Ziele von Mikroprozessorarchitekturen, Kapitel 3.1.1 aus TB MPT

Architektur von Microcontrollern

Von Neumann Architektur

“The term **Von Neumann architecture**, also known as the **Von Neumann model** or the **Princeton architecture**, derives from a 1945 [computer architecture](#) description by the [mathematician](#) and early

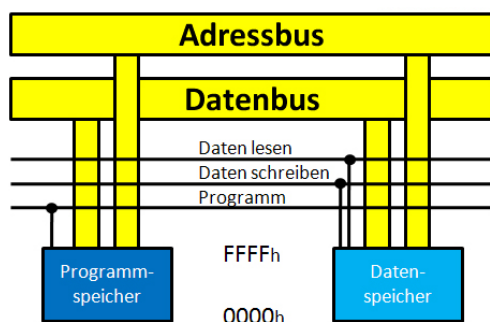
computer scientist [John von Neumann](#) and others, [First Draft of a Report on the EDVAC^{\[1\]}](#). This describes a design architecture for an electronic [digital computer](#) with subdivisions of a [processing unit](#) consisting of an [arithmetic logic unit](#) and [processor registers](#), a [control unit](#) containing an [instruction register](#) and [program counter](#), a [memory](#) to store both data and [instructions](#), external [mass storage](#), and [input and output](#) mechanisms.^{[1][2]} The meaning of the term has evolved to mean a [stored-program computer](#) in which an instruction fetch and a data operation cannot occur at the same time because they share a common [bus](#). This is referred to as the [Von Neumann bottleneck](#) and often limits the performance of the system.” (Wikipedia, engl, 2013-04-23, 20:15)



Harvard Architektur

Harvard-Architektur bezeichnet in der [Informatik](#) ein Schaltungs-konzept zur Realisierung besonders schneller [CPUs](#) und [Signalprozessoren](#). Der Befehlsspeicher ist physisch vom Datenspeicher getrennt und beide werden über getrennte Busse angesteuert. Der Vorteil dieser Architektur besteht darin, dass Befehle und Daten gleichzeitig geladen, bzw. geschrieben werden können. Bei einer klassischen [Von-Neumann-Architektur](#) sind hierzu mindestens zwei aufeinander folgende Buszyklen notwendig. Zudem sorgt die physikalische Trennung von Daten und Programm dafür, dass bei Softwarefehlern kein Programmcode überschrieben werden kann. Nachteilig ist allerdings, dass nicht benötigter Datenspeicher nicht als Programmspeicher genutzt werden kann.

Die Harvard-Architektur wurde zunächst überwiegend in [RISC](#)-Prozessoren konsequent umgesetzt. Moderne Prozessoren in Harvard-Architektur sind in der Lage, parallel mehrere Rechenwerke gleichzeitig mit Daten und Befehlen zu füllen.



Auswahlkriterien

Aufgabenstellung

Messen-Steuern-Regeln, Signalverarbeitung
Überwachen

Architekturmerkmale

8, 16, 32 bit

Speichertiefe

Taktfrequenz

Befehlssatz (Art der Befehle, Zyklen pro Befehl)

Peripherie Komponenten (ADC, externe Schnittstellen, Timer, Interrupt, Serielle I/O,)

Energiebedarf (power save mode)

Ressourcen / Projekttechnische Kriterien

Kostentreiber

- bereits vorhandene Plattformen nutzen (HW und SW)

- minimaler technischer Aufwand

- Umstiegskosten (neues Equipment, neue Entwicklungsumgebung,...)

- Schulungskosten für Mitarbeiter

Marketing

Kundenakzeptanz

Marktetablierung eines uC

Kontrollfragen:

- 1) Beschreibe anhand eines Blockschaltbildes den Aufbau eines Microcontrollers.
- 2) Welche Aufgabe hat die CPU.
- 3) Beschreibe die notwendigen Speicher eines Microcontrollersystems.
- 4) Was ist der Unterschied zwischen einem Microcontroller und einem Microprozessor?
- 5) Aus welchen Leitungen besteht ein Bussystem?
- 6) Was versteht man unter dem Begriff SoC, und beschreibe diesen.
- 7) Was ist die „von Neumann“ Architektur?
- 8) Was ist der Neumann bottleneck?
- 9) Was ist die Harvard Architektur?
- 10) Welche Kriterien gibt es zur Auswahl eines uC? Nenne mindestens 5 Kriterien.

ATmega64 (Atmel)

Der ATmega 64 ist ein 8bit Microcontroller der Firma Atmel.

Features des ATmega64

Siehe datasheet am Skriptenserver/Schrempf/DIC3 bzw unter www.atmel.com.

Link to datasheet (ggf ein update suchen):

http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42717-ATmega644PA_Datasheet.pdf

- High speed low power 8bit controller architecture
- RISC architecture
- Fully static operation
- up to 64/128kB flash memory
- up to 16kB internal RAM
- 10.000 write/erase cycles for flash memory
- Bootloaderfähig
- etc, siehe datasheet

Aufbau der CPU

“CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.” (Atmel datasheet)

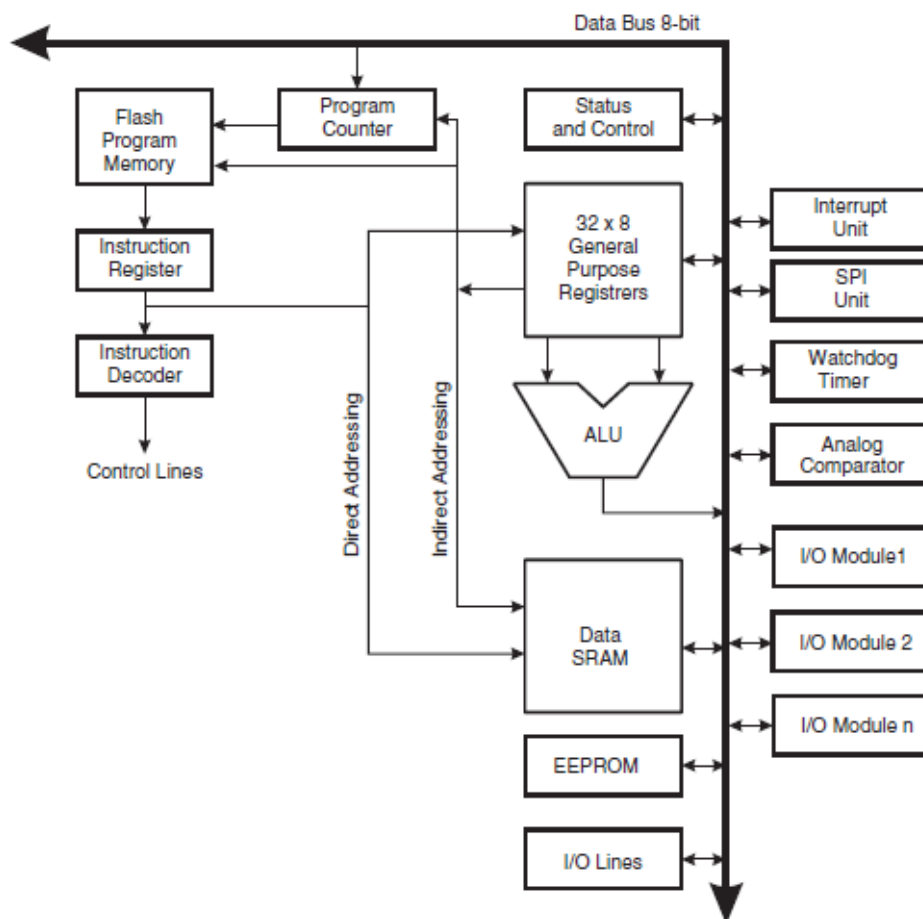


Abb 2: Innerer Aufbau der CPU des ATmega32

Die CPU ist als 8bit Harvard Architektur ausgeführt, mit separatem Programm und Datenspeicher. Für maximalen Durchsatz wurde ein single level pipelining eingeführt. Dies bedeutet das während ein Befehl ausgeführt wird im Hintergrund bereits der nächste Befehl aus dem Speicher geladen wird. Somit kann in jedem Taktzyklus ein Befehl geladen werden.

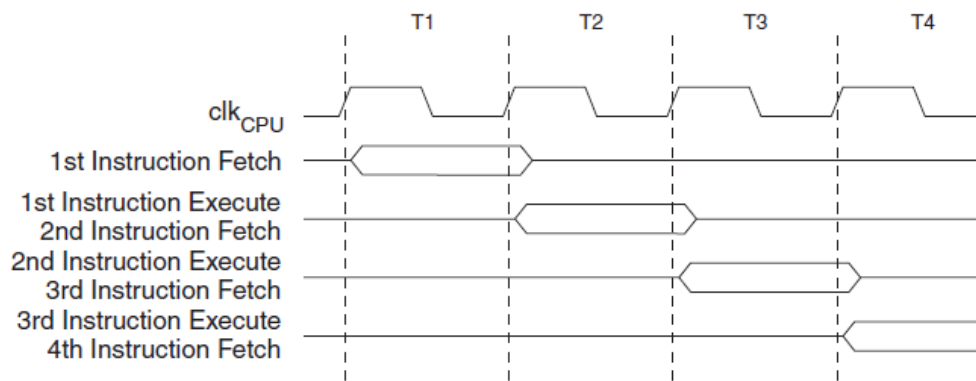


Abb 3: Pipeliningstruktur von Befehl laden (fetch) und Befehlsausführung (execute).

Weitere Informationen sind dem AVR-datasheet auf den Seiten 10 und 11 zu entnehmen.

Aufgabenstellung:

Erarbeite eigenständig die folgenden Fragenstellungen.

- 1) Wozu dienen die Register?
- 2) Wozu dient der stackpointer (SP) und was ist dabei zu beachten?
- 3) Welche Befehle kann die ALU ausführen?
- 4) Welche Funktion erfüllt der programcounter (PC)?

Ein besonderes Register: Das Statusregister (SREG)

Das Statusregister beinhaltet Information über die zuletzt in der ALU ausgeführte Rechenoperation. Das Ergebnis einer solchen Operation kann zB Null sein, einen Überlauf erzeugen, usw... In Abhängigkeit von gesetzten bits im Statusregistster können zum Beispiel bedingte Programmverzweigungen ausgeführt werden.

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
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 4: Das ATmega32 Statusregister

Bit 0 – C: Carry Flag

The Carry Flag C indicates a carry in an arithmetic or logic operation.

Bit 1 – Z: Zero Flag

The Zero Flag Z indicates a zero result in an arithmetic or logic operation.

Bit 2 – N: Negative Flag

The Negative Flag N indicates a negative result in an arithmetic or logic operation.

Bit 3 – V: Two's Complement Overflow Flag

The Two's Complement Overflow Flag V supports two's complement arithmetics.

Bit 4 – S: Sign Bit, $S = N \oplus V$

The S-bit is always an exclusive or between the Negative Flag N and the Two's Complement Overflow Flag V.

Bit 5 – H: Half Carry Flag

The Half Carry Flag H indicates a half carry in some arithmetic operations. Half Carry is useful in BCD arithmetic.

Bit 6 – T: Bit Copy Storage

Speicher für BitLoad oder BitStore Operationen.

Bit 7 – I: Global Interrupt Enable

The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers.

Assembler

Die Adressierung durch die CPU

Den technischen Ablauf, den die CPU verwendet, um an den Inhalt einer Speicherzelle zu gelangen, nennt man Adressierung. Es gibt hierzu mehrere Wege und so spricht man von Adressierungsarten.

Beim ATmega32 gibt es drei Adressierungsarten:

Immediate Addressing

Beispiel:

LDI Load Immediate

Loads an 8 bit constant directly to register 16 to 31.

LDI R17, 0xA3

Analyse des Befehls:

16 bit opcode

- 1) Instruction set manual: LDI → 1110 kkkk dddd kkkk
 kkkk constant upper / lower nibble
 dddd register block 16 -31

- 2) Befehl zusammenbauen: 1110 1010 0001 0011 = EA 13

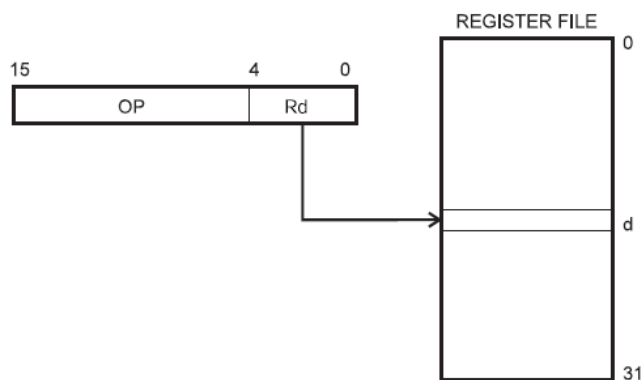


Abb 5: Struktur des immediate addressing modes am ATmega64

Direct Addressing

Beispiel 1:

LDS Load Direct from Data Space

LDS R2, \$01A3

Analyse des Befehls:

32 bit opcode

- 1) Instruction set manual: LDS → 1001 000d dddd 0000
kkkk kkkk kkkk kkkk
- 2) Befehl zusammenbauen: 90 20
01 A3

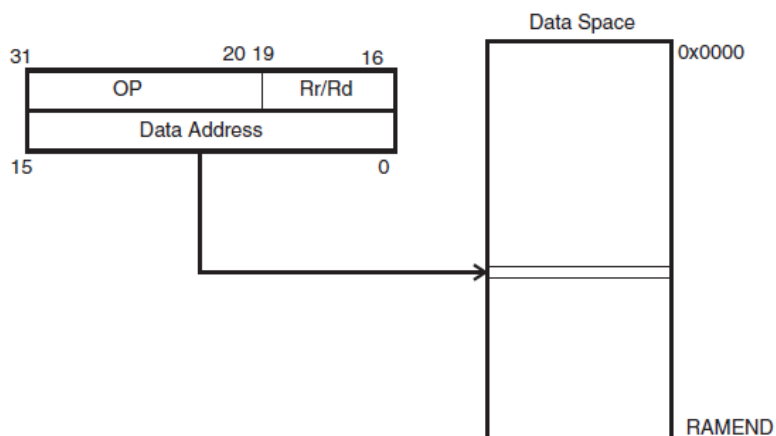


Abb 6: Struktur des direct addressing modes am ATmega64

Beispiel 2:

STS Store Direct to Data Space

STS \$03FE, R18

Analyse des Befehls0x93

Wie lautet der Dezimalwert der Adresse?

Beispiel 3:

Kombination der beiden obigen Befehle in einer Sequenz.

Im Register R1 steht der Wert 0x02.

- a) Was passiert in der nachfolgenden Sequenz?
- b) Assembliere die gesamte Befehlssequenz mit dem instruction manual.

Opcode	Erklärung
LDS R2, \$03FF	
ADD R2, R1	
STS \$03FF, R2	

Indirect Addressing

Beispiel:

LD – Load Indirect from Data Space to Register using Index X.

LD R16, X X: R26, R27

Das Register X besteht aus zwei Einzelregistern. Es hat eine Auflösung von 16bit. X wird als Zeiger im Adressraum verwendet. Bei 16 bit kann man 2^{16} Adressen ansprechen (= 64kByte)

Beispiel:

CLR R27 R27 = 0, Register löschen oder leeren

LDI R26, 0x60 R26 = 0x60
X = 0x0060

LD R0, X Inhalt der Adresse 0x0060 wird auf R0 gespeichert

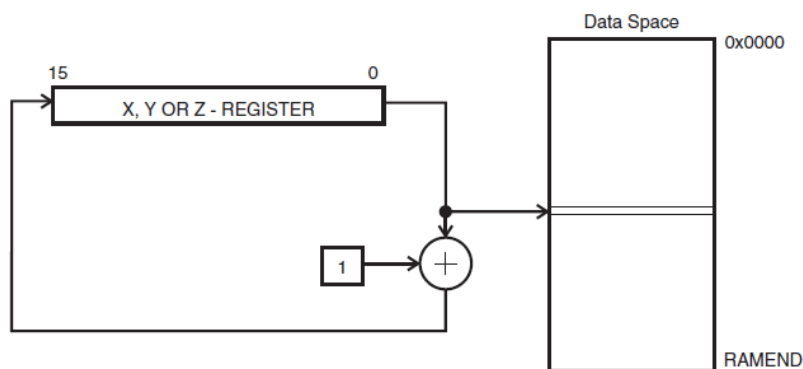


Abb 7: Struktur des indirect addressing modes am ATmega64

Neben dem Speichern auf die angegebene Adresse stehen auch die Optionen pre-decrement, sowie post-increment für die indirekte Adressierung zur Verfügung.

LD R2, X+ post-increment
Der Inhalt der Adresse wird in R2 gespeichert, danach wird X um eins incrementiert. X=0x0061 R27=0x00 R26=0x61

LD R3, -X pre-decrement
Der Inhalt der Adresse in X, welche zuvor um ein decrementiert wurde, wird in R2 gespeichert

Aufgabenstellung:

Beispiel 1:

```
LDI R18, 0x03
ADD R3, R18
LDI R29, 0x03
LDI R28, 0x00
ST Y, R3
```

Initial sind alle Register auf 0x05 vorgeladen.

- Wie lange ist die Ausführungszeit in Taktzyklen?
- Wie groß ist der Code (in Byte)
- Wie wird der Code in den Speicher geschrieben?
- Was steht in Speicherzelle 0x0300, wenn R3 mit 0xFF geladen ist?

Beispiel 2:

```
LDS R2, $0815
LDI R17, 0x03
EOR R2, R17
CLR R31
LDI R30, 0x03
ST Z+,R2
```

Beispiel 3:

CLR R29 ;	Clear Y high byte
LDI R28,0x60 ;	Set Y low byte to 0x60
LD R0,Y+ ;	Load r0 with data space loc. 0x60(Y post inc)
LD R1,Y ;	Load r1 with data space loc. 0x61
LDI R28,0x63 ;	Set Y low byte to 0x63
LD R2,Y ;	Load r2 with data space loc. 0x63
LD R3,-Y ;	Load r3 with data space loc. 0x62(Y pre dec)
LDD R4,Y+2 ;	Load R4 with data space loc. 0x64

Beispiel 4:

```
LDI R18, 0x33
LDI R19, 0x14
ADD R18, R19
STS $0020, R19
LDI R27, 0x00
LDI R26, 0x20
ST X+, R19
```

Der Assembler Code (siehe Beispiel 1) wird von der IDE compiliert und in Form im HEX Format abgespeichert. Nachfolgend kann es über ein Programmierinterface in den Programmspeicher (prog FLASH) des Mikrocontrollers geladen werden.

Der von der Assembler Sequenz errechnete Wert ist im Datenspeicher (data IRAM) auf der Speicheradresse \$0300 abgelegt.

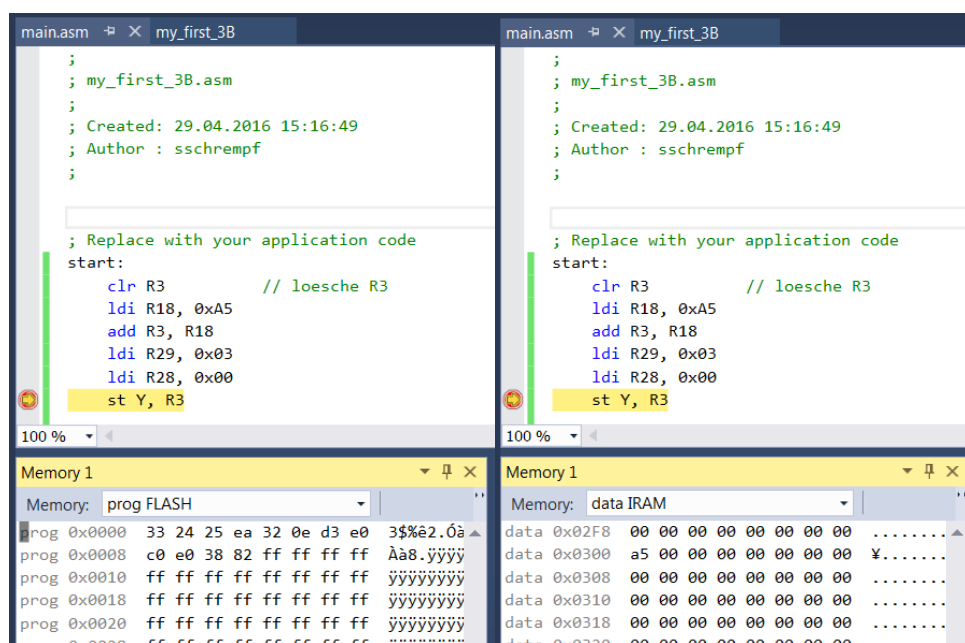


Abb 8: Darstellung des assemblierten Codes im Programm Memory und der gespeicherten Daten im Data Memory.

Organisation des Programmspeichers

Die Größe des Programmspeichers beträgt 64k-Byte. Es gibt somit 64k Adressen welche jeweils ein Byte adressieren. Die maximale Speicheradresse beträgt FFFFh.

Da alle Assembler Befehle 16 oder 32 Bit breit sind, erfolgt die Adressierung der Assemblerbefehle aus Sicht des Program-Counters bezogen auf ein word zu 16 Bit. Die maximale Adresse des Program-Counters beträgt 7FFFh. Daraus abgeleitet beträgt die Breite des Program-Counters (PC) 15bit.

Befehl	Assembler Code		#Bytes	program FLASH			
	HB	LB		Memory Addr	HB (addr+1)	LB (addr+0)	PC Addr
CLR R23	27	88	2	0x0000	27	88	0x0000
LDI R17, 0x06	E0	16	2	0x0002	E0	16	0x0001
LDS R24, \$0300	91	80	2	0x0004	91	80	0x0002
	03	00	2	0x0006	03	00	0x0003
NOP	00	00	2	0x0008	00	00	0x0004
				0x000A			
				usw...			
				0xFFFF			0x7FFF

Abb 9: Adressierung des Programmspeichers des ATmega 64

Organisation des Datenspeichers

Das Ansprechen der insgesamt 4352 Byte des Data Memories (Datenspeicher) erfolgt über die zuvor behandelten Adressierungsarten.

Der Speicher unterteilt sich in 4096 Byte exklusiven Datenspeicher, sowie eigene Speicherbereiche für die 32 Register in der CPU, sowie weitere 224 Byte als Register zur Ansteuerung sämtlicher Peripherals (siehe Abb 10).

	Load/Store
32 registers	0x0000 – 0x001F
64 I/O registers	0x0020 – 0x005F
160 Ext I/O registers	0x0060 – 0x00FF
Internal SRAM (4096x8)	0x0100
	0x10FF

Abb 10: Data Memory Map mit 4096 Byte an internem SRAM Bereich.

Gruppierung der Assemblerbefehle

Gruppierung lt instruction set manual

- 1) Arithmetische und logische Befehle
- 2) Sprünge und Verzweigungen
- 3) Data transfer Befehle
- 4) Bit und bit-test Befehle
- 5) CPU control Befehle

Sprünge und Verzweigungen

Unbedingte Sprünge

JMP **Jump absolute**

kann innerhalb des (bis zu) 4M words großen Program Memory herumspringen.

Achtung: ATmega64 ist eine Harvard Maschine, dh

das Programm Memory ist bis zu 4MB groß

das Daten Memory ist bis zu 8kB

k ist eine 22bit Konstante welche die Adresse angibt.

siehe auch RJMP

RJMP springt innerhalb von +/- 2k im Program Memory

Assembliere den Befehl: RJMP -4

- 1) -4 als Zweierkomplement ausdrücken
- 2) Assemblieren mit Hilfe des ISM

Bedingte Sprünge

Beispiel 1:

BREQ **Branch if Equal**

Wertet den CP (Compare) Befehl aus und springt wenn Rd = Rr zur Zieladresse (Program counter) plus einem Offset k mit $-64 \leq k < 64$.

Der Program counter zeigt auf die Instruktion die ausgeführt wird. Der PC ist daher eine Adresse.

Codesequenz:

CP R1, R0 Vergleiche den Inhalt der Register 1 und 0, dh es wird die Operation $Rd - Rr$ ausgeführt (CP Rd, Rr)

Ist das zero flag im Statusregister SREG gesetzt, wird der Sprung ausgeführt.

BREQ k Springe nach $PC = PC + 1 +/- k$
k im signed, dh Zweierkomplement Format

k wird in der Codesequenz durch einen label ersetzt (siehe Beispiel 2)

Beispiel 2:

Komplementärbefehl zu BREQ: BRNE **Branch if Not Equal**

Codesequenz:

CP R11, R12

BRNE loop Ist das zero flag im Statusregister SREG nicht gesetzt, wird der Sprung ausgeführt.

loop: Hier ist der PC-Offset (k) durch einen label (loop) ersetzt worden
NOP Die Sprungweite k, darf jedoch nicht überschritten werden.

Beispiel 3:

BRGE Branch if Greater or Equal (Signed)

CP Rd, Rr

BRGE equal

```
....
equal:
    NOP
```

Aufgabenstellungen:

Initialisieren eines Speicherbereiches unter Verwendung des Befehls **BRNE**:

- 1) Counterregister R16
- 2) Initialisierungsregister R17
- 3) Befehl zum Schreiben auf eine Speicherzelle ST
- 4) Branch Befehl BRNE

Was macht die folgende Codesequenz?

- 1) Interpretiere jede Befehlszeile im einzelnen
- 2) Was macht das Assemblerprogramm genau?

```
// Verwendung von BREQ

.def ctri = r17
.def memi = r18

ldi r16, 0x0A // Schleifenendwert
ldi ctri, 0x00 // Schleifenzähler
ldi memi, 0x55 // Initialisierungswert

ldi r28, 0x00 // Pointer auf Arrayanfang
ldi r29, 0x01

init:
    cp r16, ctri
    breq end // Schleifenbedingung
    st Y+, memi
    inc ctri
    rjmp init

end: // Endlosschleife
    jmp end
```

Data Transfer Befehle

Bewegen von Daten zur ALU oder von der ALU weg.

Siehe **data memory map** im ATmega64 datasheet Seite 30.

Beispiele:

LDI, LDS, STS, LD, ST, MOV

MOV

Copy Register

MOV Rd, Rr

Kopiert einen Registerinhalt in ein anderes Register Rd = Rr

Der Inhalt von Rr bleibt unverändert

Spezielle Befehle sind **IN** und **OUT** womit das **I/O memory adressiert werden kann**. Das heißt mit OUT und IN können Daten auf die Ports geschrieben werden bzw Daten von den Ports gelesen werden.

Die jeweilige Belegung der Register im I/O memory findet sich im datasheet.

Siehe Kapitel Memories, keyword: I/O Memory.

\$1C (\$3C)	EECR	—	—	—	—	EEIE	EEWE	EEWE	EEWE	IV
\$1B (\$3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	64
\$1A (\$3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	64
\$19 (\$39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	64
\$18 (\$38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	64
\$17 (\$37)	DDRB	ddb7	ddb6	ddb5	ddb4	ddb3	ddb2	ddb1	ddb0	64
\$16 (\$36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	65
\$15 (\$35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	65

Abb 11: Belegung der Adressen im I/O memory Adressbereich. Die Adressen in der Klammer beziehen sich auf Adressierung mittels LD und ST Befehlen.

IN Loads data from the I/O Space (Ports, Timers, Configuration Registers etc.) into register Rd in the Register File.

OUT Stores data from register Rr in the Register File to I/O Space (Ports, Timers, Configuration Registers etc.).

Beispiel:

IN R25, \$0016 Read PINB (read from memory location)
 CPI R25, 0x04 Compare immediate register with constant
 BREQ manip Branch if R25=4 (PC offset k = 6)

...

manip:

STS \$003B, R26 Write register content to PORTA (write to memory location)

- 1) Assembliere die Sequenz2
- 2) Berechne den Offset des Branch-Befehls, Achtung der Offset ist PC bezogen
- 3) Berechne die Laufzeit des Codes
- 4) Berechne die Größe des Codes

Arithmetische und logische Befehle

Befehle die die ALU verwenden

ADD Add without Carry

ADD Rd, Rr $Rd = Rd + Rr$

Addition von zwei Registern ohne Berücksichtigung des carry Bits (Übertrag).

ADC Add with Carry

ADC Rd, Rr $Rd = Rd + Rr + C$

Addition von zwei Registern unter Berücksichtigung des carry Bits (Übertrag).

Das carry wird gesetzt, wenn das Ergebnis einer Addition größer als 255d ist.

Aufgabenstellungen:

Sequenz 1:

Addiere einen 16bit Offset zu einem Adresszeiger in Y.

```
CLR R29
LDI R28, 0x60
LDI R17, 0x03
LDI R16, 0xFF
```


ADD R28, R16	Add low byte
ADC R29, R17	Add with carry high byte
LD R0, Y+	

- 1) Von welcher Adresse wird gelesen?
- 2) Welche Adresse steht nach dem Ausführen der Sequenz 1 im Y Register?

Sequenz 2: siehe Mitschrift

Die Startadresse der Sequenz liegt auf \$0030

label 1:

```
LDI R17, 0xF0
LDI R18, 0x1F
ADC R17, R18
BRCS label1
```

- 1) Assembliere die Sequenz2
- 2) Berechne den Offset des Branch-Befehls, Achtung der Offset ist PC bezogen
- 3) Berechne die Laufzeit des Codes
- 4) Berechne die Größe des Codes

Programmierung unter ANSI C

ANSI C referenziert auf den Standard für die Programmiersprache C des American National Standards Institute (ANSI).

http://en.wikipedia.org/wiki/ANSI_C

Für fast alle Prozessoren gibt es zumindest einen C Compiler.

CROSS Compiler: Unter einem **Cross-Compiler** versteht man einen Compiler, der auf einem bestimmten System (auch Hostplattform genannt) läuft, aber Kompilate (Objektdateien oder ausführbare Programme) für andere Systeme erzeugt.

Gründe für den Einsatz eines Cross Compilers:

- Für „embedded systems“ welches nicht die Ressourcen (eg genug Arbeitsspeicher) für einen eigenen Compiler hat.
- Die Entwicklung kann mittels einer IDE (Integrated Develeopment Environment) komfortabler durchgeführt werden. Die von uns verwendete IDE ist AVR Studio 4.
- In eigenen Sessions können unterschiedlich Kompilate für mehrere Plattformen erzeugt werden.
- Sie können auf schnellen Systemen laufen und für langsamere Systeme Kompilate erzeugen.

Als Cross Compiler für den AVR verwenden wir den GNU C Compiler gcc (open source). Atmel empfiehlt einen Compiler der Firma „Tarsking“. Dieser wird durch die installation von WinAVR in die IDE integriert.

Toolchain für die Entwicklung und dem Test einer uC Software

Die folgende Abbildung gibt einen Überblick über die notwendigen Entwicklungsschritte zur Erstellung eines uC Programms und dem nachfolgenden „flashen“ auf den AVR controller.

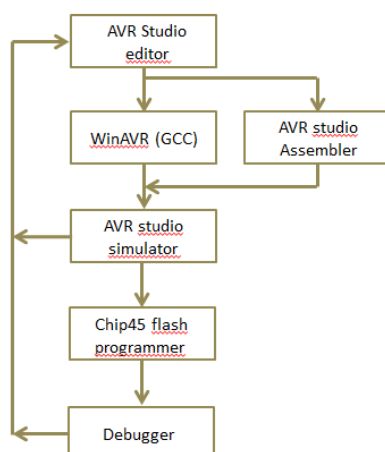


Abb 12 AVR software development toolchain.

Atmel Studio

AVR Studio ist eine IDE Plattform welche von Atmel als freeware zur Verfügung gestellt wird. Die DIE Plattform dient zum Entwickeln eines Programmes, dh Editieren, Simulieren, Erstellen des Maschinen Codes (Hex-Code) sowie zum Debuggen eines Programmcodes.

Ein Assembler ist in einer IDE immer mit dabei.

Hardware zum Debuggen des Hex Codes am Prozessor: **AVR Dragon, ICE II**

Installation des AVR Studios (Version 7.0):

- Setup, siehe: <http://www.atmel.com/microsite/atmel-studio/>

Installation des chip45 flash programmer:

- Download-Link: http://www.chip45.com/avr_bootloader_atmega_xmega_chip45boot2.php

Weiterführende Links:

- <http://www.mikrocontroller.net/>

Atmel Studio quick reference

Die nachfolgende Tabelle listet die für einen „jump-start“ wichtigsten Befehle und Funktionen für Atmel Studio auf.

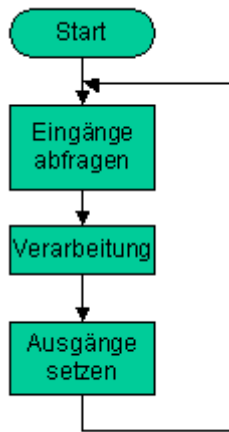
Klassifikation	Details	check
AVR Studio Entwicklungsumgebung		
Projektverwaltung	Einfügen, verwalten von files, Projektsetup	
Texteditor	Eingabefenster für den source code	
Simulator	Simulatordetails (eg Registerwerte)	
Simulatordetails	Simulator und Microcontroller interne Werte	
Ausgabebereich	Status und Fehlermeldungen des Compilers	
Optimizer	Einstellen des optimizer levels	
AVR Studio Bedienung		
Build	Aufsetzen des compile, simulate environments Im Ausgabefenster erscheint eine entsprechende Statusmeldung. Nach einem erfolgreichen Durchlauf wird unter default ein hex file	
Rebuild all	Nach source code Änderungen muss der compile run neu durchlaufen werden	
Debug	Simulation des source codes	
Disassembler	Assembler code zur C source	
Watch Window	Zustand von Variablen	
AVR Simulator Details		
Start debugging		
Stop debugging		
Breakpoints		
Reset debugging		
Add to watch window		
cycle counter		
SREG		

Tabelle 1: Wichtige Funktionen und Features des Atmel Studio 7

Aufbau eines uC Programmes

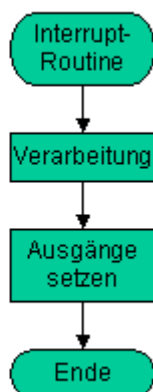
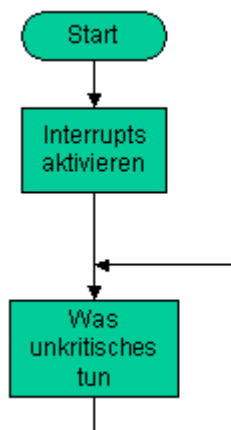
Grundsätzlich unterscheidet man einen sequentiellen und einen interrupt-gesteuerten Ablauf.

Sequentieller Ablauf



Bei dieser Programmiertechnik wird eine Endlosschleife programmiert, welche im Wesentlichen immer den gleichen Aufbau hat. Es wird hier nach dem sogenannten EVA-Prinzip gehandelt. EVA steht für "Eingabe, Verarbeitung, Ausgabe".

Interruptgesteuerter Ablauf



Bei dieser Methode werden beim Programmstart zuerst die gewünschten Interrupt-Quellen aktiviert und dann in eine Endlosschleife gegangen, in welcher Dinge erledigt werden können, welche nicht zeitkritisch sind. Wenn ein Interrupt ausgelöst wird, so wird automatisch die zugeordnete Interrupt-Funktion ausgeführt.

Variablen und Datentypen

In einem Programmablauf ist es häufig notwendig Informationen kurzfristig abzuspeichern. Dies geschieht in sogenannten Variablen. Die einzelnen Variablentypen unterscheiden sich durch Verwendung, ihrer Größe und ihrem Vorzeichen.

Da C eine relativ einfache Sprache ist, kennt sie auch nur vier Datentypen. Diese sind **int**, **char**, **float** und **double**.

<i>Datentyp</i>	<i>Länge</i>	<i>Bereich dezimal</i>	<i>Anwendung</i>
uint8_t unsigned char	8 bit	0 .. 255	vorzeichenlose kleine Zahlen, Zähler, Zeichen
uint16_t unsigned int	16 bit	0 .. 65535	vorzeichenlose große Zahlen und Zähler
uint32_t unsigned long	32 bit	0 .. 4294967295	vorzeichenlose sehr große Zahlen und Zähler
int8_t char signed char	8 bit	-128 .. +127	vorzeichenbehaftete kleine Zahlen
int16_t int short int	16 bit	-32768 .. +32767	vorzeichenbehaftete große Zahlen
int32_t long long int	32 bit	-2147483648 .. +2147483647	vorzeichenbehaftete sehr große Zahlen
float	32 bit	$\pm 10^{-37} \dots \pm 10^{38}$	reelle Zahlen mit ca. 7 Dezimalstellen
double	64 bit	$\pm 10^{-307} \dots \pm 10^{308}$	reelle Zahlen mit ca. 15 Dezimalstellen
void			leer oder unbestimmt oder nicht verwendet

Tabelle 2: Variablentypen

Beispiele:

```
// Präprozessor Anweisung als reine textuelle Ersetzung.
#define BAUD 9600L    // Constante als signed (L)ong integer definiert

const int wert = 12;    // Definition einer Konstanten
```

Template eines C-Programms

Grundlegend werden an den Beginn eines uC Programms alle notwendigen Einbindungen von externen Headerdateien gestellt, welche vom sgn „Präprozessor“ durchgeführt werden.

```
# include <stdio.h>
# include <stdlib.h>
```

Danach erfolgt die Definition von Variablen und ihren entsprechenden Typen, welche somit global sind, was bedeutet, dass sie während der gesamten Laufzeit des Programms existieren.

```
int counter, a, b, c;
```

Weiters werden an den Beginn des Programms häufig verwendete gleiche Werte als Konstanten deklariert.

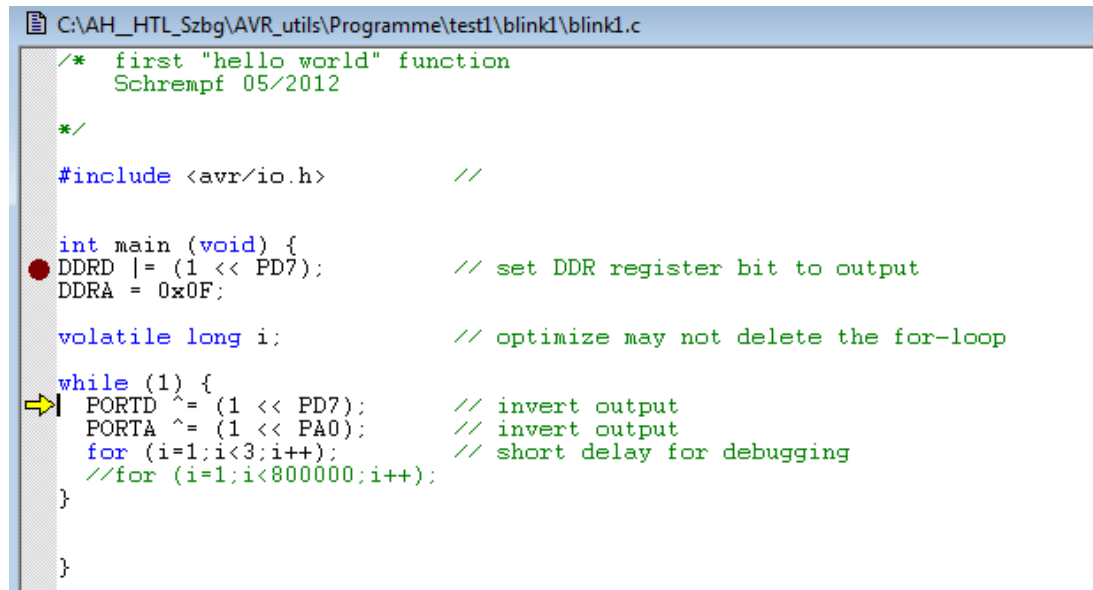
```
const float PI_val = 3.141592;
```

Eine Sonderform von Konstanten stellen „Präprozessor Direktive“ dar welche mittels dem Schlüsselwort #define eingebunden werden. Ein Beispiel dafür ist das Einstellen der Taktfrequenz mit welcher der uC (fiktiv im Simulator) betrieben wird.

```
#define F_CPU 1000000;
```

Die Definition von Prozeduren und Funktionen sowie der entsprechenden Übergabeparameter wird ebenfalls noch vor der eigentlichen Hauptroutine durchgeführt.

Die Hauptroutine wird am Ende des Programms implementiert. Von ihr aus erfolgt der Aufruf der zuvor deklarierten Prozeduren und Funktionen.



```
C:\AH_HTL_Szbg\AVR_utils\Programme\test1\blink1\blink1.c

/* first "hello world" function
   Schrempf 05/2012
*/

#include <avr/io.h>           //

int main (void) {
    DDRD |= (1 << PD7);      // set DDR register bit to output
    DDRA = 0x0F;

    volatile long i;         // optimize may not delete the for-loop

    while (1) {
        PORTD ^= (1 << PD7);  // invert output
        PORTA ^= (1 << PA0);  // invert output
        for (i=1;i<3;i++);    // short delay for debugging
        //for (i=1;i<800000;i++);
    }
}
```

Abb 13: „Hello World“ Version eines Microcontrollers, das Blinklicht.

Ausgewählte uC Themen

Zugriff auf die I/O ports

Der Zugriff auf die I/O ports erfolgt über die Register in der I/O memory map. Initial gilt es zu jedem port die Datenrichtung einzustellen, sowie bei Ausgängen den Zustand zu erfassen und bei Eingängen den Zustand festzulegen.

Im Allgemeinen werden pins zu den ports zusammengefasst. Bei den Standard Mikrocontrollern sind meist 8 pins zu einem port gebündelt (eg PortA = {A7, A6, A5, A0})

Hinter jedem pin verbirgt sich eine Menge an Funktionalität. Die kann sein:

- die Richtung (Eingang oder Ausgang) festzulegen,
- der Zustand zu schreiben bzw zu lesen,
- pull-up Widerstände zu setzen,
- den Eingang zu enablen

Einen Überblick über die komplexe Struktur eines einzelnen pins gibt die nachfolgende Abbildung eines GPIO pins.

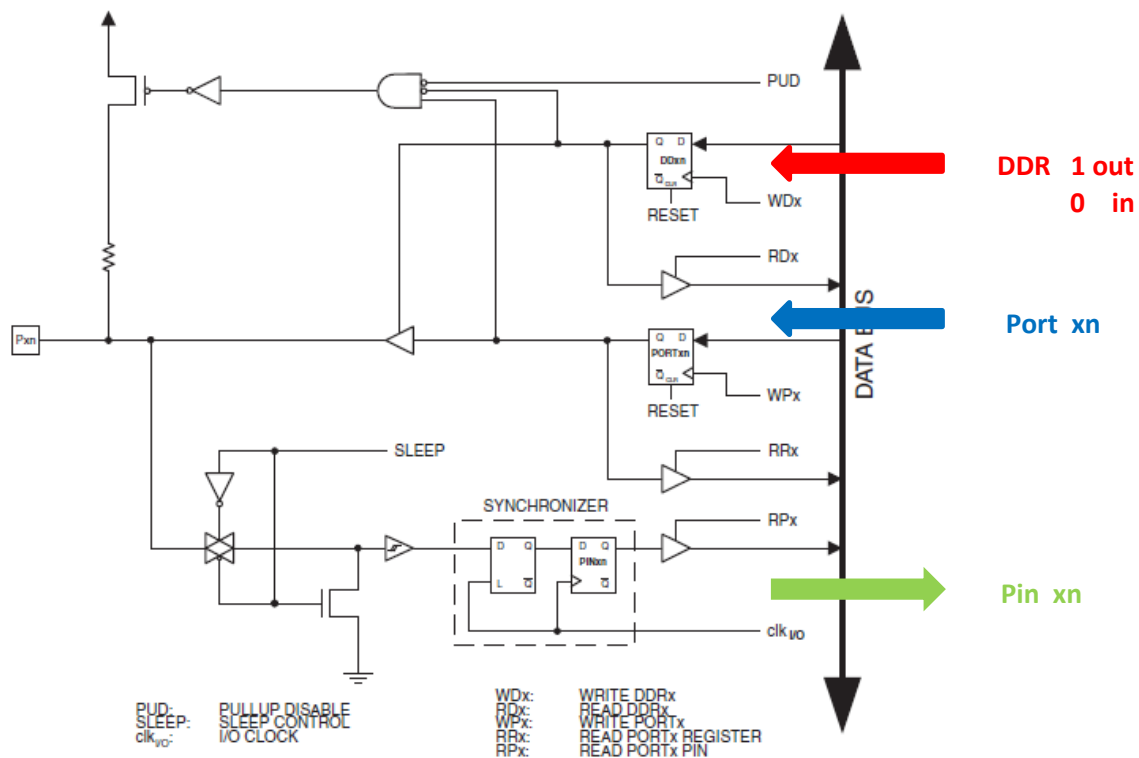


Abb 14: Schaltbild eines General Purpose Digital I/O pins (GPIO)

Die Steuerung der ports erfolgt über drei Register.

x entspricht den ports A, B, C, D.

- 1) **Datenrichtungsregister**, data direction register (DDRx)
0 Eingang
1 Ausgang
- 2) Eingangsadresse, **PIN I/O address** (PINx)
Über dieses Register wird der Zustand der portpins eingelesen.
0 Zustand des portpins ist „L“.

1 Zustand des portpins ist „H“.

3) **Datenregister für Portx**, portx data register

Über dieses Register werden die ports angesteuert.

Bei denjenigen Pins die über DDRx auf Eingang geschaltet wurden, können über Portx die internen pull-up Widerstände zugeschaltet werden. Dies gilt sofern keine externe pull-ups an den pin geschaltet sind.

1 interner pull-up aktiv
0 interner pull-up inaktiv

```
void main(void)
{
    unsigned char wert;    // variable declaration

    DDRA = 0xF0;           // set bit 3 : bit 0 to input, bit 7 : bit 4 to output
    PORTA = 0x0F;          // activate internal pull-up for defined inputs

    while(1)
    {
        wert = PINA;       // read data from portA
        _delay_ms(1000);   // wait for ....
    }
}
```

Maskieren

Unter einer **Bitmaske** versteht man eine mehrstellige Binärzahl (meist in der Datenwortbreite des Controllers), mit der ein anderes Datenwort mittels einer logischen Verknüpfung (AND, OR, XOR) verändert werden kann.

Grundsätzlich unterscheidet man drei Maskierungsoperationen.

- 1) Ein bit auf log Null setzen (AND Operator)
- 2) Ein bit auf log Eins setzen (OR Operator)
- 3) Ein bit toggeln (XOR Operator)

Siehe dazu auch das Beiblatt [Maskierung_Resultate.pdf](#) am Moodle server.

Aufgabenstellungen:

Hinweis: Zum praktischen Test der Software werden AVR Crumb644 boards zur Verfügung gestellt, welche mit Eingängen und Anzeige LEDs versehen sind. Der default Aufbau ist so vorzusehen, dass die Eingänge auf PortB (CON1, Pin 1-8) liegen und die Ausgänge auf PORTA (CON2, Pin 20-13) geschaltet sind.

- 1) Schreibe auf PortA, (B, C, D) abwechselnd 0x55 und 0xAA.
- 2) Schreibe eine einfache Blinkroutine.
- 3) Schreibe mehrere Bits eines Ports als Blinkroutine.
- 4) Maskiere Bits und schreibe sie auf einen bestimmten Port.
- 5) Generiere ein Lauflicht auf Portx.
- 6) Programmiere einen Binärzähler auf Portx.
- 7) Lies auf PortA einen Wert ein.
- 8) Lies einen Wert auf PortB ein und gib ein Signal auf PortD aus wenn der Wert \$F5 erkannt wurde.

- 9) Lies auf PortA einen bestimmten Wert ein, und maskiere (AND) diesen mit 0x0F und gib den auf PortC wieder aus.
- 10) Lies einen Wert auf PortB ein, invertiere die Bits 1, 3, 5, 7 und gib den Wert verzögert um 500ms verzögert wieder aus.
- 11) Schreibe auf die Ports A-D in abwechselnder Reihenfolge (2s) „DEAD“ – „BEEF“. Die Ausgabe erfolgt dabei auf vier Siebensegment Elementen.
- 12) Lies einen Wert auf PortA ein, und gib das obere und untere Daten-Nibbel auf PortB und PortC beginnend mit dem LSB am Port wieder aus (Shiftoperation).

Weitere Aufgabenstellungen siehe im Skriptum [uC Programmierübungen.docx](#)

Sonderthemen

Entprellen eines Tasters/Schalters:

Prellen: kurzzeitiges mehrfaches Schließen und Öffnen eines mechanischen Kontaktes bei seiner Betätigung (siehe Abb 15).

Die herkömmliche Entprellschaltung mittels RS Flip Flops kann bei Verwendung von uC sehr einfach in Software realisiert werden. Der Zustand eines Tasters an Portx wird in einer Endlosschleife permanent abgefragt, eine entsprechende Verzögerungszeit sorgt für die notwendige Entprelldauer des Tasters.

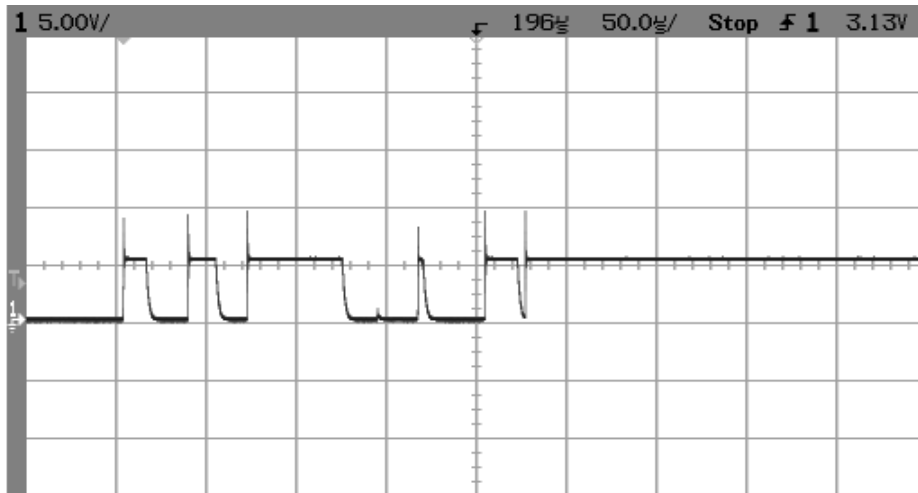


Abb 15: Oszillogramm des „Prellens“ eines Schalters.

Aufgabenstellung 1:

- Block diagram as intro, AVR with pull down switch and external LED
- Switching positions (active low, pull up resistor)
 - open
 - falling edge
 - closed
 - rising edge
- Set up of the state diagram
- C program
- Demo board crumb644

Aufgabenstellung 2:

Binary counter on every rising edge

- Reduced state diagram
- C program w/o any delay and observe the counter value on PortA
 - Does the controller software work properly? Explain the result.
- Expand C program w/ debouncing on rising edge and observe counter value on PortA
 - What has been improved, what is still missing?
- Implement debouncing on the falling edge and again observe the result.

Interruptsteuerung (ATmega644)

Bisher wurde davon ausgegangen, daß ein uC Programm Schritt für Schritt ausgeführt wird. Es findet keine Nebenläufigkeit statt.

Für manche Anwendungen ist es notwendig, daß das Eintreten eines bestimmten Ereignisses innerhalb kürzester Zeit registriert, und darauf entsprechend reagiert wird. Hierbei spricht man von Interrupt Steuerung, wenn es möglich ist den eigentlichen Programmablauf zu unterbrechen und stattdessen eine ISR (Interrupt Service Routine) aufzurufen, die das aufgetretene Ereignis behandelt. Die jeweilige ISR wird nur ausgeführt wenn zuvor das *global interrupt flag* im SREG aktiviert wurde. Die aufzurufende ISR wird über sign Interrupt Vektoren erkannt (siehe Abb 16, bzw datasheet).

Vector No.	Program Address ⁽²⁾	Source	Interrupt Definition
1	\$000 ⁽¹⁾	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$002	INT0	External Interrupt Request 0
3	\$004	INT1	External Interrupt Request 1
4	\$006	INT2	External Interrupt Request 2
5	\$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	\$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	\$00C	TIMER1 CAPT	Timer/Counter1 Capture Event

Abb 16: Reset und Interruptvektoren

Die Interrupt Service Routine ist als C-code zu implementieren. Dieser Code wird aufgerufen wenn der Interrupt auftritt. Dazu dient das Makro ISR(). Dieses Makro muss mit dem entsprechenden Interrupt Vektor als Argument aufgerufen werden.

Die Interrupt Vektoren sind in der include-library iomxx.h zu finden.

Nachfolgend ein Beispiel einer ISR auf den INT0, bei dessen Auftreten der PortC invertiert wird.

```
ISR (INT0_vect)
{
    PORTC = ~PORTC;
}
```

Eine Abarbeitung von Ereignissen kann auch durch regelmäßige Abfrage der Interruptflags erfolgen. Es wird hierbei jedoch nicht unmittelbar, wie bei Interruptverarbeitung, reagiert, sondern in einer Art Zeitscheibenverfahren nach zuvor festgelegten Intervallen.

Der Fall der sequentiellen Abfrage aller möglichen Ereignisse wird „Polling“ genannt.

Anmerkung: Es ist zu beachten, daß sowohl polling als auch interrupt handling den Controller bis an seine Grenzen auslasten können. Die jeweilige Ereigniskontrolle ist daher bei der Software-Architektur sorgfältig zu planen.

Externer Interrupt

Die externen Interrupts werden von den pins INT0, INT1 und INT2 gesteuert, und sind wahlweise level oder flankengetriggert. Diese Funktionalität wird im **External Interrupt Control Register (EICRA)** eingestellt.

INT0, INT1 und INT2 sind sowohl level als auch flankengetriggert.

Über das **External Interrupt Mask Register (EIMSK)** werden die Interrupts aktiviert. Dazu muss auch das I-bit (Interrupt) im Status Register gesetzt sein.

Tritt ein Interrupt Ereignis am jeweiligen Pin auf, dann wird das entsprechend Bit im **External Interrupt Flag Register (EIFR)** gesetzt. Nach dem Abarbeiten der Interrupt Service Routine wird das bit wieder gelöscht.

Bsp: Interruptgesteuertes Einlesen eins Datenports gesteuert über eine fallende Flanke an INT0 am Beispiel eines ATmega64.

```
#include <avr/io.h>
#include <avr/interrupt.h>

unsigned char buf = 0x00;

// Interrupt Service Routine
ISR(INT0_vect){
    buf = PINC;
}

int main(void){
    sei(); // global interrupt enable

    // INT0 configuration
    // configure to falling edge
    EICRA |= (1 << ISC01);
    EICRA &= ~(1 << ISC00);
    EIMSK |= (1 << INT0); // local enable
    EIFR = (1 << INTF0); // clear int flag, write one

    while(1)
    {
        return 0
    }
}
```

Volatile Variablen

Das Schlüsselwort volatile teilt dem Compiler mit, dass ausnahmslos alle Zugriffe auf eine Variable auch tatsächlich auszuführen sind und keine Optimierungen am Code gemacht werden dürfen. Dh der Compiler darf eine Variable nicht wegoptimieren, weil die Variable so benutzt werden kann, wie dies für den Compiler prinzipiell nicht einsichtig ist.

Timer Interrupt / (PWM)

In diesem Kapitel wird neben dem Aspekt der Timer Steuerungen im Allgemeinen, auch noch im Besonderen auf die Pulsweitenmodulation (PWM) eingegangen. Die Verknüpfung mit der PWM (engl puls width modulation) ist notwendig, weil der Timer/Counter ein Grundbaustein eines PWM Signals ist.

Der Timer ist ein Hardware-Element welches über den Datenbus mit der CPU verbunden ist. Zentrales Element dieser Hardware ist ein Zähler. Wird ein bestimmter Grenzwert, oder der Zählerüberlauf erreicht wird ein Interrupt ausgelöst. Das Hochzählen kann durch unterschiedliche Quellen erfolgen. Dies geschieht entweder durch ein periodisches Signal (iA ein Taktsignal), welches zusätzlich durch einen Prescaler in seiner Frequenz abgesenkt werden kann, oder aber durch eine externe Quelle, deren Flanken gezählt werden sollen, und ab einer bestimmten Anzahl einen Interrupt auslösen.

Im Nachfolgenden wird auf den Timer Betriebsmode CTC (clear timer in compare match) Bezug genommen. Alle anderen Modi lassen sich in ähnlicher Weise aus dem Datenblatt ableiten.

Allgemeine Konfigurationsschritte:

- 1) Timer/Counter Control Register TCCR0
In diesem Register werden die verschiedenen Betriebsmodi des Timers festgelegt.
Für den CTC Mode ist im TCCR0 das bit WGM01 zu setzen.
- 2) Aktivieren des Prescalers zur Einstellung der Taktfrequenz des Timer/Counters.
- 3) Setzen des Vergleichswertes im output compare register.
- 4) Aktivieren des Timer Interrupts.

CTC Mode:

Jeder Timer match mit dem OCR Register bewirkt ein Wechseln des Ausgangszustandes im OC0 Register (siehe Configuration der OCn Registers im Datasheet).

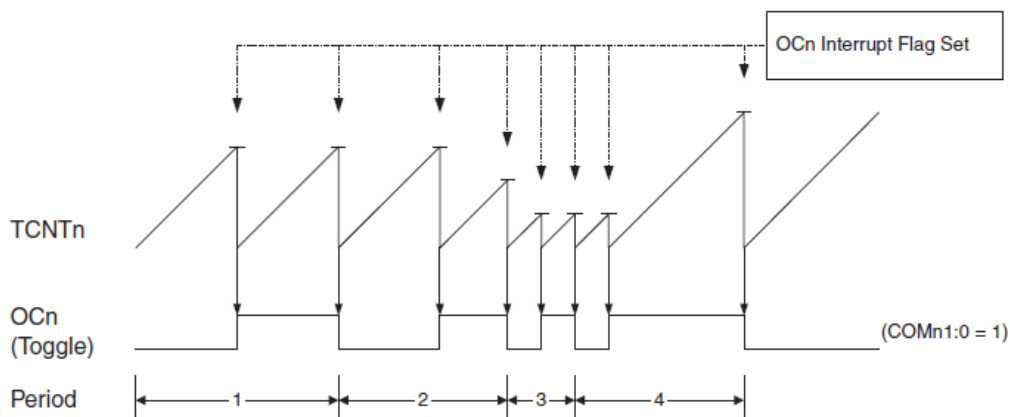


Abb 17: CTC timer mode and OCn toggle behaviour.

Der Timer0 output pin OC0A ist dem Datenblatt zu entnehmen.
Für den ATmega32 ist dies der Pin PB3 (dies gilt aus Kompatibilitätsgründen auch für den Baustein ATmega644P).

Die Herleitung der Berechnung des Timer Vergleichswertes, welcher im OCn Register eingetragen wird, erfolgt im Theorieunterricht.

(Skizze zur Herleitung)

$$OCRn = -$$

Beispiel1:

Im CTC Timer Mode ist eine timerloop mit 5ms zu erzeugen. Der ausgelöste Interrupt sollte in seiner Service Routine ein den PortA toggeln. Als unkritischer task sollte die LED and PD7 mit jedem Durchlauf invertiert werden.

Die entsprechende Berechnung ist der Mitschrift zu entnehmen.

Beispiel2:

Im CTC Timer Mode ist eine Frequenz von 440Hz zu erzeugen.

Beispiel3:

Es ist ein Timer mit einer Schleifenzeit von 1ms zu erzeugen. Auf Grundlage dieses Timers ist eine Gesamtschleife für ein Multitaskingsystem von 500ms zu realisieren.

Selbsttest:

Entwirf einen Timer mit der Schleifendurchlaufzeit von 1ms (2ms, 3ms, 4ms ...) und kontrolliere die Durchlaufzeit mit Hilfe eines Breakpoints in der zugehörigen ISR.

- Bestimme die genaue Timerdurchlaufzeit:
- Optimiere die gewählte Timerdurchlaufzeit.

Fast PWM Mode:

Beschreibung folgt

Kennzeichen:

Fixe Frequenz

Der Duty Cycle wird mit dem OCR Wert gesteuert.

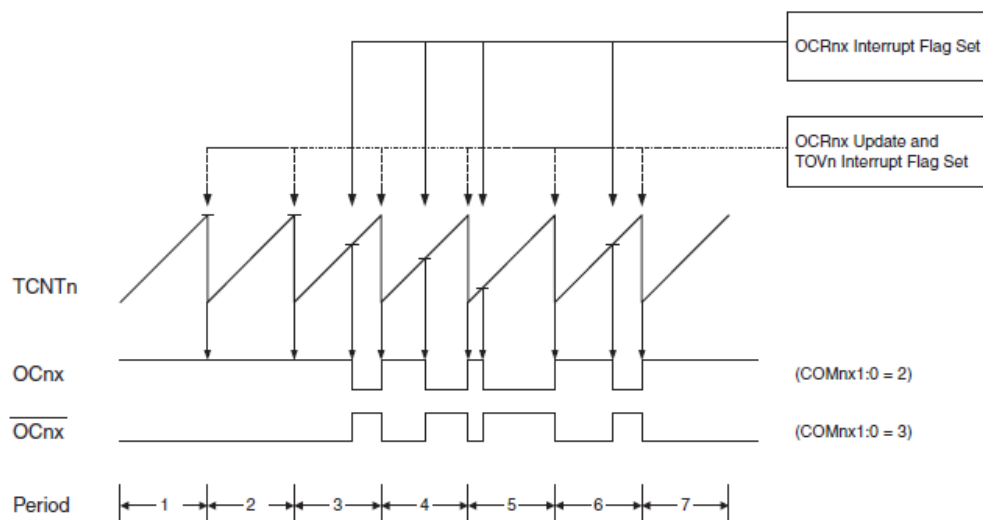


Abb 18: Fast PWM Mode, Timing Diagram

Beispiel1:

Im Fast PWM Timer Mode ist eine timerloop mit etwa 1ms, bei einem duty cycle von 25% zu erzeugen.

Skizziere das Timing und berechne die entsprechenden Registersettings.

Einlesen von Analogwerten

Voraussetzung für das Verständnis der ADC Funktionalität in Microcontrollern sind Grundkenntnisse zu Prinzipien der AD und DA Wandlung.

Der ATmega interne ADC verwendet einen 10 bit SAR zur Analog Digital Umsetzung. Es sind 8 single ended und 16 differential Eingänge über einen Multiplexer selektierbar.

Die wichtigsten Registerwerte die vor der Umsetzung konfiguriert werden müssen sind nachfolgend am Beispiel einer **single ended conversion** aufgelistet.

- 1) Einstellen des ADC input channels ADMUX bits MUX4:0
- 2) Einstellen der Referenzspannung, ADMUX bits REFS1:0
- 3) Sollte das 10 bit Umsetzungs Ergebnis linksbündig ausgegeben werden ist ADLAR zu setzen, wobei sich das Ergebnis der Umsetzung auf 8bit beschränkt
- 4) Aktivieren des ADC über ADEN (ADC enable) im Register ADCSRA.
- 5) Steuern der Umsetzung über ADSC (ADC start conversion) im Register ADCSRA.

- 6) Optional: Deaktivieren der digital input buffer über DIDR0. „Power reduction“
 7) ADCSRB: Selektieren von externen Triggerquellen zur Steuerung der ADC Umsetzung.

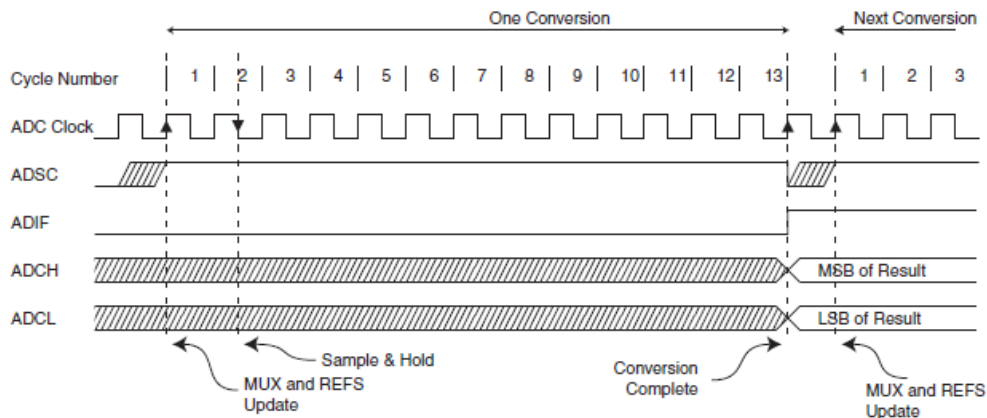


Abb 19: Timing einer 10 bit single ended conversion

In Abb 19 ist das Timing einer ADC single ended conversion dargestellt.

Nach erfolgter Umsetzung erhält man einen der Eingangsspannung proportionalen Zahlenwert als Ergebnis, der sich wie folgt aus Auflösung (2^N), Eingangsspannung V_{IN} und Referenzspannung V_{ref} ergibt.

$$Z = \frac{V_{IN} * 2^N}{V_{REF}}$$

Selbsttest:

Wieviel Prozent des ADC Eingangsspannungsbereiches sind durch die ersten 5bit des Controllers abgedeckt?

Im folgenden Beispiel wird die Initialisierung eines ADC dargestellt. Danach wird in einer Endlosschleife die ADC-Messung immer wieder neu gestartet indem da Bit ADSC (ADC start conversion) im ADC configuration register gesetzt wird. Nach jeder erfolgten Wandlung wird das bit ADSC automatisch per hardware rückgesetzt.

```
// init ADC
ADCSRA |= (1 << ADEN);
ADCSRA |= (1 << ADPS2)|(1 << ADPS1);
ADMUX |= (1 << MUX1);
ADMUX |= (1 << ADLAR);
ADMUX |= (1 << REFS1)|(1 << REFS0);

// enable ADC conversion
// set prescaler to 64
// set to ADC ch1
// left adjust result
// sel 2,56V internal reference

while(1)
{
    ADCSRA |= (1 << ADSC);           // ADC start conversion
    while (ADCSRA & (1 << ADSC))
    {
        ergebnis = ADCH;
        PORTC = ergebnis;
        _delay_ms(500);
    }
}
```

Datentransfer / Interfaces

Die Interfaces mit welchen Microcontroller ausgestattet werden, hängen stark vom Anwendungsgebiet des Controllers ab (Consumer, Automotive, usw). Die wichtigsten dabei vorkommenden Interfaces sind SPI, I2C, USB, LIN, CAN, usw.

Im nachfolgenden werden einige Interfaces exemplarisch beschrieben.

SPI Interface (Serial Peripheral Interface)

Allgemeine Betrachtungen

Das SPI bietet die Möglichkeit mit wenig Aufwand mehrere Geräte miteinander zu vernetzen. Dabei werden nur vier Leitungen benötigt:

MOSI	Master Out Slave In (auch Serial Data Out, SDO)
MISO	Master In Slave Out (auch Serial Data In, SDI)
SCK	Serial Clock
SS	Slave Select (auch Chip Select, CS)

Das SPI arbeitet nach dem Master Slave Prinzip. Der Master startet den Transfer indem er die SS Leitung des entsprechenden angeschlossenen Gerätes auf low (low activ) zieht. Der Master liefert das Clock Signal (SCK). Der Datenaustausch erfolgt synchron, durch das Clock Signal gesteuert. Um das Interface zu synchronisieren kann nach jedem übertragenen Datenwort, die SS Leitung wieder auf high gezogen werden, womit alle internen Register des SPI rückgesetzt werden. Die Datenwortbreite entspricht meist der internen Datenwortbreite des Controllers, jedoch gibt es verschiedenste SPI Implementierungen mit unterschiedlichen Datenwortbreiten, welche sich am Befehlssatz des jeweiligen Bausteins orientieren.

Abb 20 zeigt eine einfache SPI Verbindung mit nur einem angeschlossenen Gerät (Slave), welches durch die CS Leitung adressiert wird, wenn diese auf low gezogen wird.

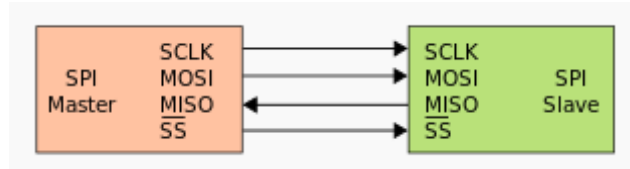


Abb 20: SPI Master / Slave Verbindung (Quelle Wikipedia)

Alternativ können auch mehrere Geräte an einen Master angeschlossen werden. Dies erfordert jedoch eine eigene SS Leitung pro angeschlossenen Gerät (siehe Abb 21).

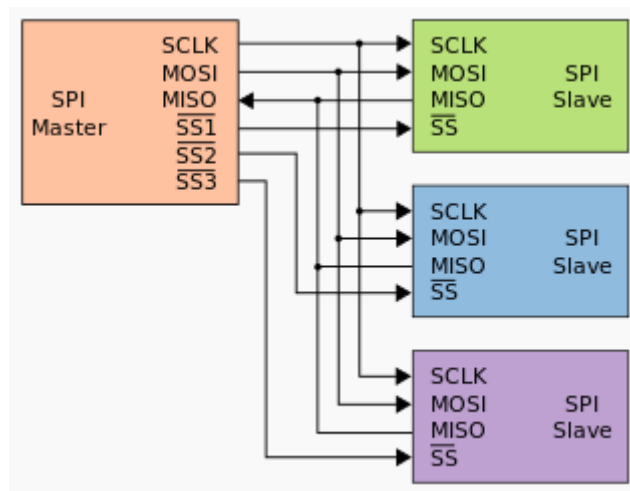


Abb 21: Mehrere Slaves kommunizieren mit einem Master, Sterntopologie (Quelle Wikipedia)

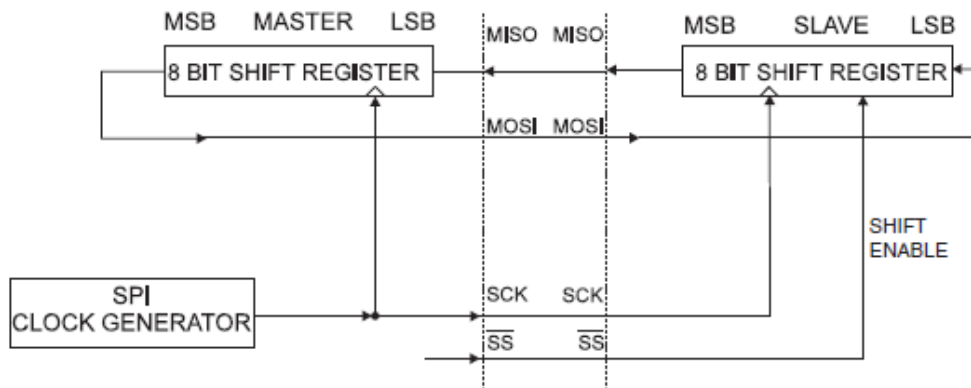


Abb 22: Master – Slave Verbindung

Das Timing des Datenaustausches ist teilweise durch die verwendete Hardware fix gebunden und kann jedoch, insbesondere in Bezug auf die Datenrate des Interfaces, durch die entsprechenden Register beeinflusst werden.

Die entsprechenden setup und hold Zeiten der Timingbedingungen sind der Spezifikation für Master und Slave zu entnehmen. Diese Zeiten beziehen sich meist auf die Flanken des Taktsignales, sowie des SS Signales.

Einstellmöglichkeiten der Timingverhältnisse können über die Signale CPOL (Clock Polarity) und CPHA (Clock Phase) vorgenommen werden, wie dies in Abb 23 illustriert ist.

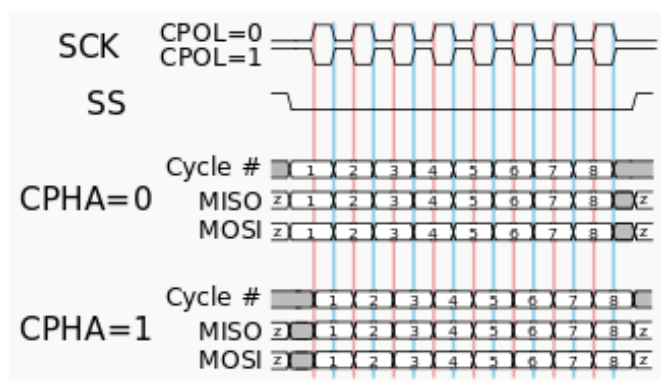


Abb 23: Timingdiagramm eines SPI (Quelle Wikipedia)

SPI im ATmega Controllern (ATmega64)

Das SPI nimmt bei der Kommunikation von Microcontrollern eine sehr verbreitete Stellung ein. Die meisten Microcontroller sind daher mit einem „embedded SPI“ als Peripherie Modul, neben anderen Interfaces, ausgestattet.

Zur Steuerung des SPI dienen dem Controller mehrere Register. Für den Controller ATmega64 sind dies folgenden:

SPI Control Register (SPCR)

Aktivieren des Interface, Einstellen des Timings, Interrupt enablen, usw

SPI Status Register (SPSR)

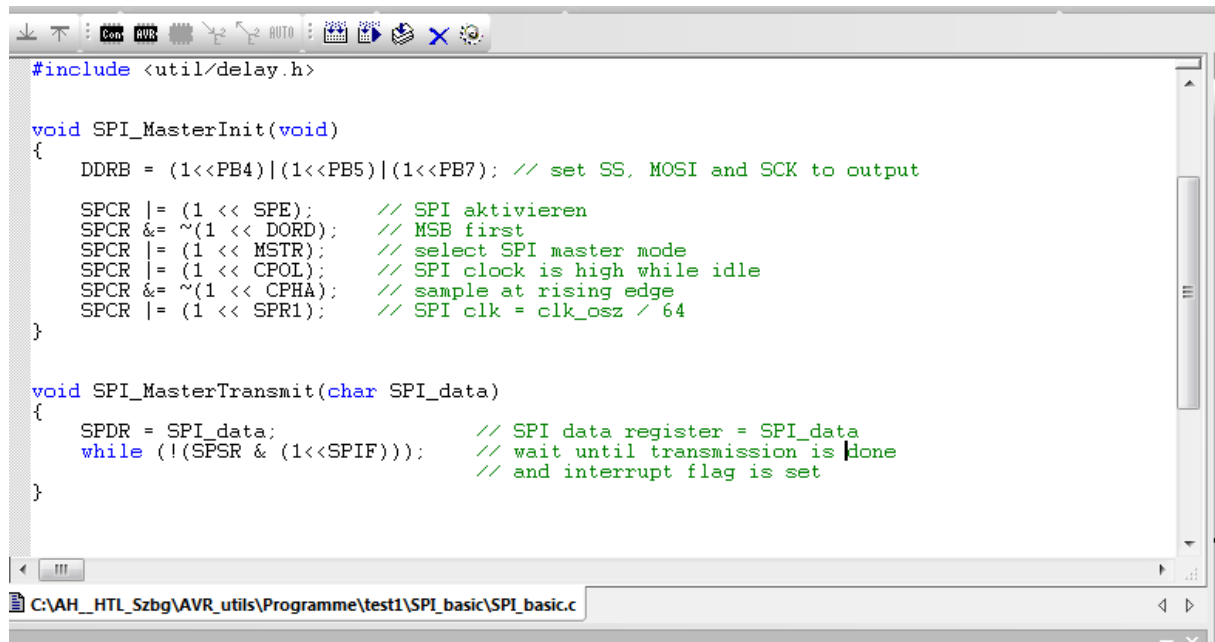
Double speed, Interrupt flag, Write collision bit

SPI Data Register (SPDR)

Datenregister des Interfaces

SPI-Write:

Das nachfolgende Beispiel sollte den Aufbau einer einfachen SPI Datenübertragung veranschaulichen. Nach der Prozedur **SPI_MasterInit** zum des Interfaces erfolgt in der **main** Routine der Aufruf der Prozedur **SPI_MasterTransmit**, welche als Übergabeparameter das zu sendende Datenwort enthält. Die Datenübertragung erfolgt von der Hardware des Interfaces selbständig, und ist mit dem Setzen des Interrupt Flags (SPIF) im SPSR beendet.



```
#include <util/delay.h>

void SPI_MasterInit(void)
{
    DDRB = (1<<PB4)|(1<<PB5)|(1<<PB7); // set SS, MOSI and SCK to output

    SPCR |= (1 << SPE);           // SPI aktivieren
    SPCR &= ~(1 << DORD);         // MSB first
    SPCR |= (1 << MSTR);          // select SPI master mode
    SPCR |= (1 << CPOL);          // SPI clock is high while idle
    SPCR &= ~(1 << CPHA);         // sample at rising edge
    SPCR |= (1 << SPR1);          // SPI clk = clk_osz / 64
}

void SPI_MasterTransmit(char SPI_data)
{
    SPDR = SPI_data;              // SPI data register = SPI_data
    while (!(SPSR & (1<<SPIF))); // wait until transmission is done
                                // and interrupt flag is set
}
```

Abb 24: Initialisierungsroutinen einer SPI Kommunikation (Write)

SPI-Read:

4.6 Die serielle SPI-Schnittstelle

377

Für den *Empfang* eines Bytes vom Sender-Slave muss nach dessen Freigabe ein beliebiges Start-Byte in das Datenregister SPDR geschrieben werden, um die Übertragung am Eingang **MISO (Master In)** zu starten. Das Ende der Übertragung wird im Anzeigebit SPIF des SPI-Statusregisters angezeigt, und das empfangene Byte kann aus dem Datenregister SPDR ausgelesen werden.

Für das *Senden* an den Empfänger-Slave am Ausgang **MOSI (Master Out)** wird nach dessen Freigabe das auszugebende Byte in das Datenregister SPDR geschrieben, um die Ausgabe zu starten. Das Ende der Übertragung wird im Anzeigebit SPIF des SPI-Statusregisters angezeigt. Bei einer *gleichzeitigen* Freigabe von Sender- und Empfänger-Slave durch den Master wird das an **MOSI** ausgegebene Byte durch das an **MISO** ankommende Byte ersetzt.

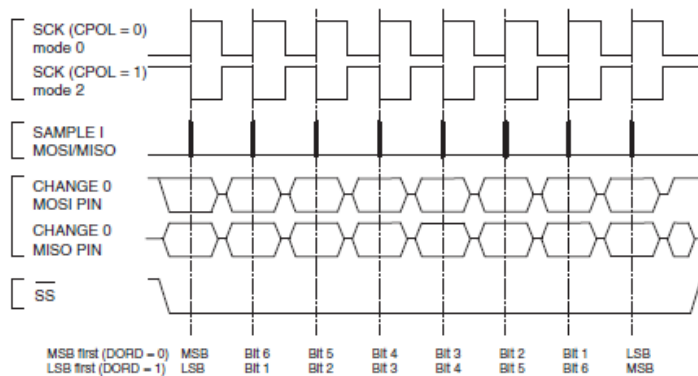
Abb 25: Beschreibung einer SPI read Kommunikation(vgl. Schmitt 2008, S. 377 ff¹)Aufgabenstellung:

Ermittle den Timing-Angaben im ATmega64 Datenblatt die maximale Datenrate des SPI interfaces. Ermittle die Timing-Bedingungen zwischen den Signalen SS, SCK, MOSI, MISO.

Durchsprache der Timing-Bedingungen inclusive CPOL und CPHA.

¹ Günther Schmitt: Microcomputertechnik mit Controllern der Atmel AVR-RISK-Familie. (München 2008)

SPI Mode	Conditions	Leading Edge	Trailing Edge
0	CPOL=0, CPHA=0	Sample (Rising)	Setup (Falling)
1	CPOL=0, CPHA=1	Setup (Rising)	Sample (Falling)
2	CPOL=1, CPHA=0	Sample (Falling)	Setup (Rising)
3	CPOL=1, CPHA=1	Setup (Falling)	Sample (Rising)



Anwendungsbeispiele zu den bisherigen Themen Port, Timer, ADC und SPI

Aufgabe1: „Programmierbare Spannungsquelle“

Es ist ein externer DAC Baustein (LTC1661) mittels SPI anzusteuern und ein vorgegebener Wert in eine Analogspannung umzuwandeln.

Die Programmierung des DAC ist dem Datasheet zu entnehmen, ebenso die theoretische Umrechnung eines Digitalwerts in eine Analogspannung.

Aufgabe2: „Pegelwandler“

Ein Eingangsspannungsbereich 0-1,8V sollte mit Hilfe des DAC (LTC1661) in einen Ausgangsspannungsbereich 0-3,3V umgesetzt werden.

Aufgabe3: „Funktionsgenerator“

Ausgehend von A1 sollte ein digitaler Sinusgenerator entwickelt werden. Die Sinuswerte sind dabei in einem Array abzubilden und mit unterschiedlicher Auflösung auszulesen und an den DAC über das SPI übergeben werden.

Alternativ kann über Taster up/down an den externen Interrupts die Frequenz verändert werden.

Aufgabe4: „Abstandssensor“

Verwendung des SPI gesteuerten Abstandssensors GPD12D.

I2C Interface / (TWI Interface)

Beim I2C (Inter-Integrated-Circuit) handelt es sich um einen von der Firma Philips (heute NXP) entwickelten und spezifizierten seriellen Datenbus.

http://www.nxp.com/documents/user_manual/UM10204.pdf (2014-03-04, 13:18)

Der Bus wird hauptsächlich zur Kommunikation zwischen integrierten Bausteinen (ASIC's) benutzt, zB von einem Controller zu verschiedenen Peripheriegeräten.

Aus lizenzrechtlichen Gründen hat Atmel für diesen Bus die Bezeichnung TWI (Two Wire Interface) eingeführt.

Allgemeines

Durch seinen einfachen Aufbau ist das I²C Interface ideal für Microcontroller Applikationen zugeschnitten. Es besteht aus zwei Leitungen SDA (Serial Data) und SCL (Serial Clock) zur seriellen Datenübertragung. Durch seinen sieben bit breiten Adressraum können bis zu 127 Peripheriebausteine an ein Leitungspaar angeschlossen werden. Der Ruhezustand der Leitungen ist der high Pegel. Dies wird entweder durch externe oder interne Pull-up Widerstände realisiert. Die angeschlossenen Geräte haben hierzu einen open-collector Ausgang.

Keywords: open-collector, wired-and

Das I²C Interface ist ein synchrones Interface und stellt entsprechend seiner Spezifikation die folgenden Betriebsmodi zur Verfügung.

Standard-mode bis 100kHz

Fast-mode bis 400kHz

Fast-mode Plus bis 1000kHz

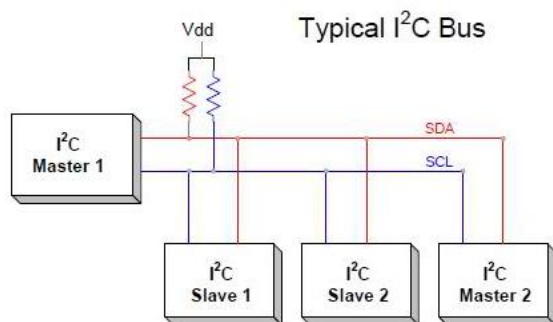


Abb 26: I²C Busverbindung mit zwei master und zwei slave Bausteinen.

Wie das SPI arbeitet auch das I²C Interface nach dem Master-Slave Prinzip. Abb 27 zeigt dabei den prinzipiellen Datentransfer am I²C entsprechend der Spezifikation von NXP.

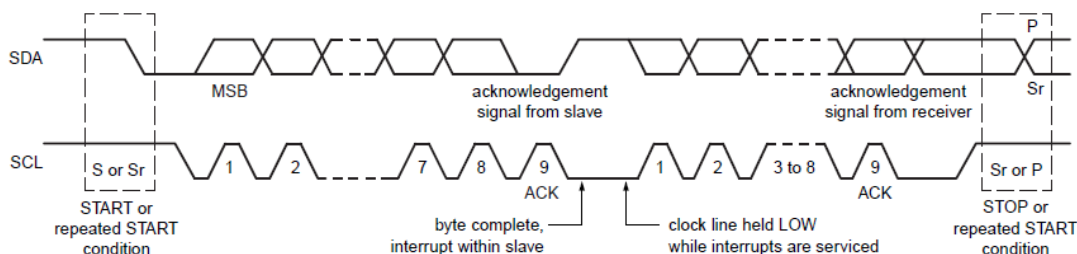


Abb 27: Datentransfer am TWI. Im Bild zu sehen ist eine Unterbrechung des I²C Transfers durch den Slave um eine andere Aktion, hier eine Interrupt Routine, abzuarbeiten. Die SCL Leitung wird dazu vom Slave für die Dauer der Unterbrechung auf Low gehalten.

Der Datentransfer am I²C Interface erfolgt byteweise. Es können pro Transfer beliebig viele Bytes übertragen werden. Jede Übertragung eines Bytes wird von einem Acknowledge bit abgeschlossen. Der Master liefert dabei die Startbedingung (SCL ist high, SDA falling edge), das Taktsignal SCL, sowie die Stopbedingung (SCL ist high, SDA rising edge). Der Datenwechsel auf SDA erfolgt während der low-Phase des Taktes, und die Datenleitung ist stabil während der high-Phase des Taktes. Die Senderichtung des Datenwortes ist generell mit „MSB first“ festgelegt.

Ein Slave wird durch einen Master über eine Adresse angesprochen, welche sieben bit breit ist. Das bit acht liefert die read/write (read = 1/write=0) Information an den Slave. Wenn die Adresse korrekt gesendet wurde erfolgt die Bestätigung durch ein Acknowledge des Slave. Danach werden je nach R/W bit entweder vom Master oder vom Slave Daten auf den Bus gelegt. Der korrekte Empfang der Daten wird wiederum durch ein Acknowledge (ACK) des Empfängers bestätigt.

Das I2C kann sowohl als Master als auch als Slave Daten senden und empfangen. Demnach gibt es auch vier verschiedene Übertragungsmodi, wobei nachfolgend, im ATmega spezifischen Teil, nur näher auf MT (Master Transmit) und MR (Master Receive) eingegangen wird.

TWI im ATmega Controllern (ATmega64)

Zur Steuerung eines TWI Datentransfers stehen beim Controller ATmega64 mehrere Register zur Verfügung, welche nachfolgend aufgezählt und in ihrer allgemeinen Funktion beschrieben werden. Nähere Informationen zu den Inhalten der Control Register sind dem Datenblatt zu entnehmen.

TW Data Register (TWDR)

Im Fall eines Schreibzugriffs auf den Slave enthält das TWDR das nächste zu sendende Datenwort. Im Fall eines Lesezugriffs enthält das TWDR die empfangenen Daten.

TW Control Register (TWCR)

Aktivieren des TWI mit TWEN, Startbedingung TWSTA, Stopbedingung TWSTO, TW Interrupt TWINT

TW Status Register (TWSR)

Zeigt den Status der TWI Hardware nach jedem initiierten Datentransfer an (eg Adresse übertragen und durch ACK bestätigt).

TW Bit Rate Register (TWBR)

Bitratenfaktor zur Bestimmung des SCL Taktes. Zusammen mit dem Statusregister kann die Taktrate nach einer Funktion eingestellt werden. Die Funktion ist dem Datenblatt zu entnehmen.

TW Address Register (TWAR)

Dieses Register enthält die 7 bit Slave Adresse im Betriebsmodus Slave Transmitter oder Slave Receiver beginnend mit dem MSD.

Programmierung des TWI

Die Programmierung erfolgt entsprechend der Vorgaben im Datasheet des ATmega.

Es ist im Allgemeinen zu beachten das Interface aktiviert ist (TWEN=1). Eine TWI Operation wird durch Schreiben einer „1“ in das TWCR gestartet. Damit wird das Interrupt bit TWINT gelöscht, und die Operation für die TWI hardware freigegeben. Nach dem Ausführen der Operation wird das bit TWINT wiederum gesetzt.

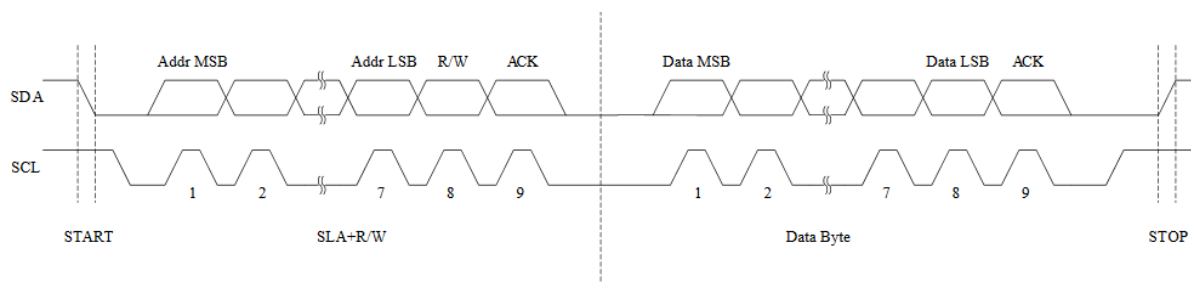


Abb 28: Typischer TWI Transfer am ATmega644 Baustein (Datasheet-release 10/2016)

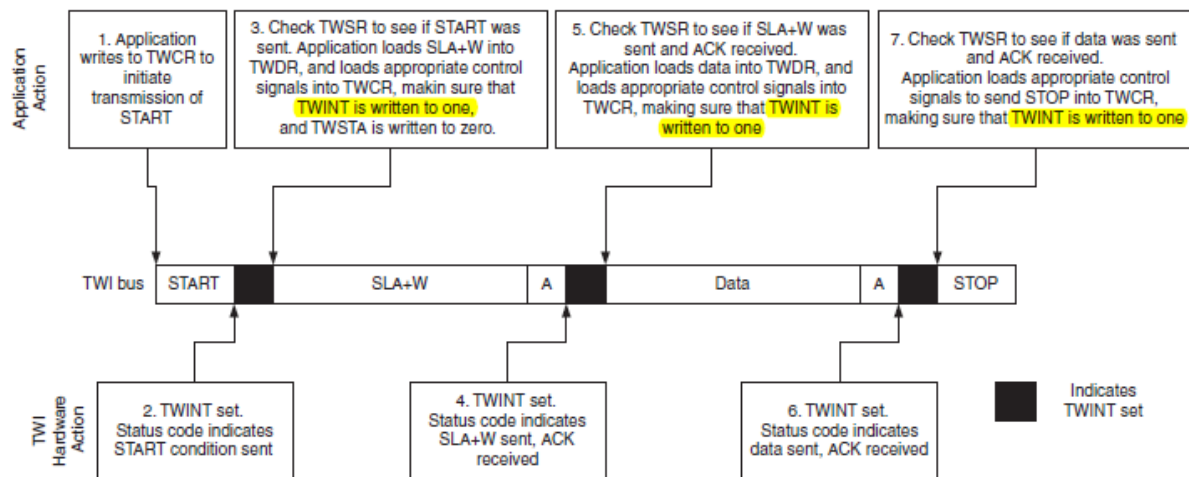


Abb 29: Ablauf einer typischen TWI write transmission

Anwendungsbeispiel I2C Temperatursensor

Basierend auf dem Sensor LM75 von Maxim sollte die Temperatur über den I2C Bus ausgelesen werden.

Dazu ist es notwendig zuvor das Datasheet des entsprechenden Bausteins zu laden.

In den nachfolgenden Schritten wird die Programmierung des Sensor laut Datasheet, die configuration, die write transmission, sowie die read transmission im Atmel Studio gezeigt.

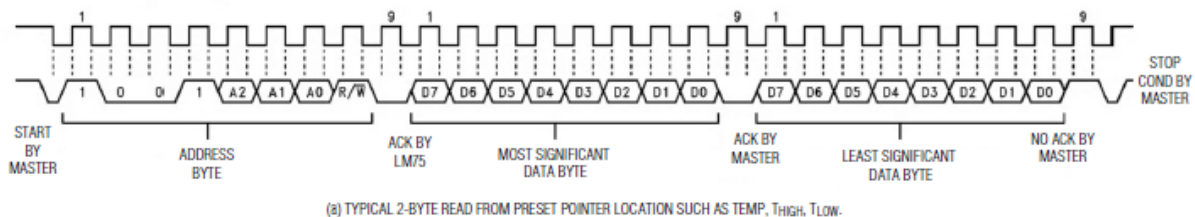


Abb 30: Temperatur Lesezyklus am I2C Interface des LM75

```
#define F_CPU 16000000 // 16MHz
#define WR_DEV 0x90 // user defined write device address
#define RD_DEV 0x91 // user defined read device address
#define BITRATE 42 // overall division factor 100

#include <avr/io.h>
#include <util/delay.h>

// Initialize TWI interface
// Set SCL clock frequency
void TWinit(unsigned char bitrate)
{
    // set SCL to 1 MHz
    TWBR |= bitrate;
    TWSR = 0x00; // set division factor 4^0=1
}
```

Abb 31: Configuration des I2C Interfaces für den ATmega644

```
// Transmit data
void TWtransmit(unsigned char address, unsigned char data_tx)
{
    TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);    // set start condition
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set

    TWDR = address;                                  // prepare address in advance
    TWCR = (1<<TWINT) | (1<<TWEN);                  // send address
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set

    TWDR = data_tx;                                  // send data
    TWCR = (1<<TWINT) | (1<<TWEN);                  // wait until TWINT is set
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set

    TWCR = (1<<TWINT) | (1<<TWSTO) | (1<<TWEN);      // set stop condition
}
```

Abb 32: TWI write access

```
// Receive data
unsigned char TWreceive(unsigned char address)
{
    unsigned char data_rx;                          // received data

    TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);    // set start condition
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set

    TWDR = address;                                  // prepare address in advance
    TWCR = (1<<TWINT) | (1<<TWEN);                  // send address
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set

    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);      // start receiving 1st data + ACK
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set
    data_rx = TWDR;

    TWCR = (1<<TWINT) | (1<<TWEN);                  // start receiving 2nd data + NACK
    while (!(TWCR & (1<<TWINT)));                    // wait until TWINT is set
    data_rx = TWDR;

    TWCR = (1<<TWINT) | (1<<TWSTO) | (1<<TWEN);    // set stop condition
    return data_rx;
}
```

Abb 33: TWI read access

Aufgabenstellung:

Die nachfolgende TWI Sequenz stellt die Kommunikation zwischen dem uC und dem Temperatursensor als client am TWI dar. Analysiere die Sequenz.

- Welche Adresse hat der Client?
- Handelt es sich um eine read oder write Sequenz?
- Gib das übertragene Byte hexadezimal an.

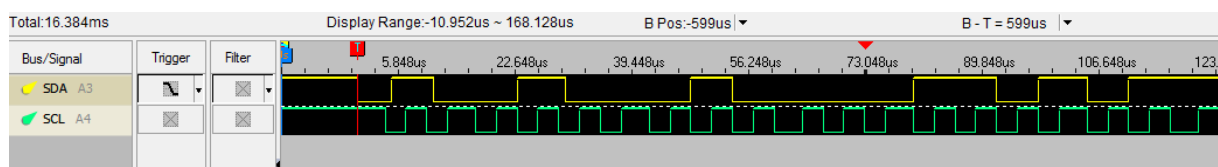


Abb 34: TWI Kommunikation mit dem Temperatursensor LM75 von MAXIM.

Aufgabenstellung:

Zeichne die Zustände der SDA und SCL Leitung des I2C Busses auf, wenn auf den Slave mit der Adresse 0x34 das Byte 0xC4 geschrieben wird. Gib dazu die Zustände der einzelnen Leitungen (SDA und CLK) bitweise an.

UART Interface

Ansteuerung eines LCD Displays

Programmiertechniken

Coding style, Reuse, Coding guidelines

Warum schreiben wir leserlichen Code?

“Nearly all the code is written in C, with only a few assembler functions where completely unavoidable. This does not result in tightly optimized code, but does mean the code is readable, maintainable and easy to port. If performance were an issue it could easily be improved at the cost of portability.
(Quelle: FreeRTOS user manual)

Multitasking bei Microcontrollern

Unter Multitasking versteht man das quasi parallele Ausführen von mehreren Prozessen auf einem Microcontroller.



Abb 35: Person (gegendert!) bei der Ausführung mehrerer paralleler Prozesse unterschiedlicher Priorität.

Präemptives Multitasking

Kooperatives Multitasking

State Machines

Anhang

Pinbelegung (pinout) des crumb644 Entwicklungsboards:

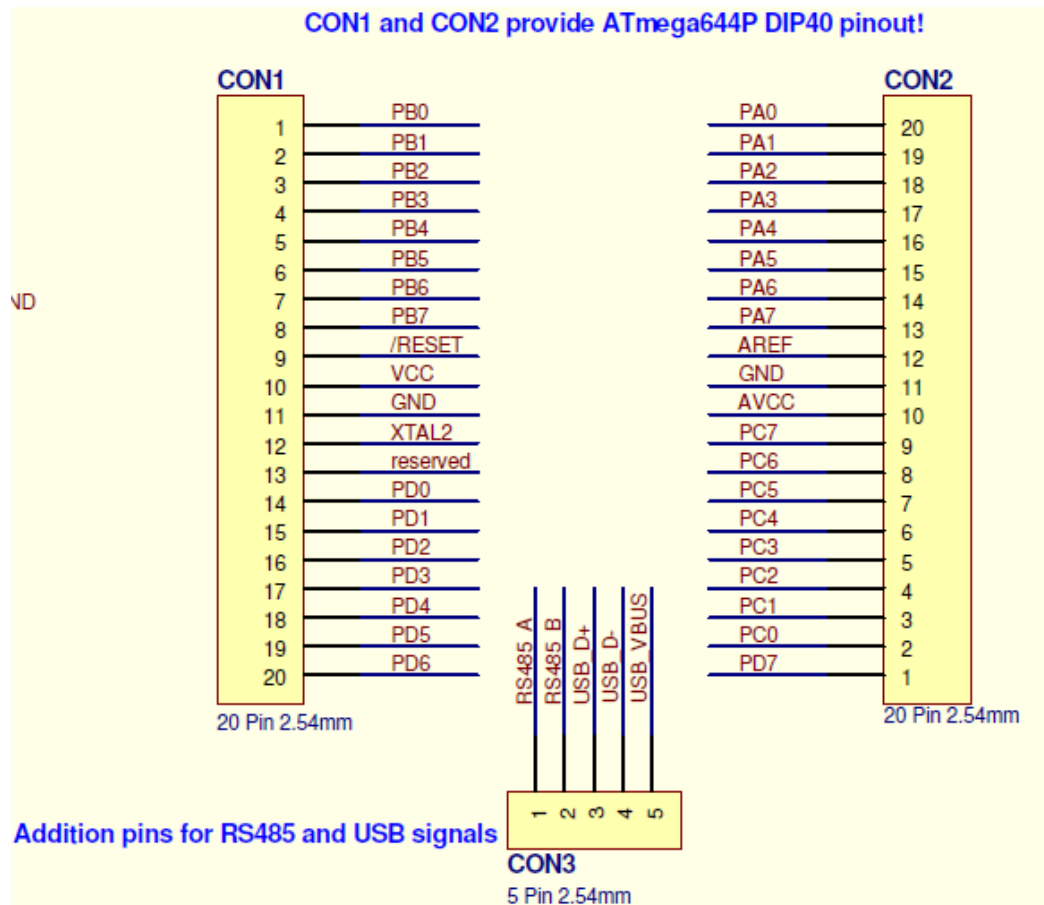


Abb 36 Pinout Crumb644B