ANSI-C for L-Kiss

Version Kiss 2.2, Dez 2009

Die grundlegenden Syntaxelemente von C sind hier aufgelistet und sollen dem Einsteiger helfen, sich rasch mit C zurecht zu finden. Diese Zusammenfassung ist als Arbeitsunterlage und Nachschlagwerk gedacht. Neben den Standard C-Elementen und Funktionen sind hier auch Werkzeuge zum Erstellen einer saubereren Softwarestruktur beschrieben. Für das L-Kiss sind die systemspezifischen Funktionen zusammengestellt. Angaben zu einem LCD-Interface, zum LCD-Charakterset und zu der MCU ATMega32 finden sich im hintersten Teil dieser Zusammenfassung.

Inhalt

Siehe auch: http://www.lemps.ch/kiss

Allgemeines zu C

Die Programmiersparache C ist Case-Sensitiv. d.h. es wird immer zwischen **Gross– und Kleinschreibung unterschieden!**

- C ist formatfrei. Innerhalb von Programmanweisungen werden Zeilenumbrüche und Zwischenräume ignoriert.
- C besteht nur aus Funktionen. Die Funktion **main** muss in jedem Programm enthalten sein. Sie wird nach dem Programmstart zuerst ausgeführt.
- Jeder Funktionscode (Block) beginnt mit **{** und endet mit **}**
- Jede C-Anweisung muss mit einem Semikolon **;** abgeschlossen werden
- Zuordnungen erfolgen mit einem = Zeichen:
  PORTA = 0x55;
  Auch Mehrfachzuordnungen sind möglich:
  PORTA = PORTB = 0x22;
- Vergleiche auf Gleichheit in Entscheidungen erfolgen mit zwei Gleichheitszeichen:
  if (PORTA == PORTB){ …… }
- Variablen und Konstanten müssen in Funktionen **unmittelbar nach dem Funktionskopf** deklariert werden (vor dem Programmcode)!
- Kommentare beginnen mit ***/**** und enden mit ****/*** . Solche Kommentare können sich über mehrere Zeilen erstrecken.
- Von den meisten MCU-C Copilern wird auch die C++ Syntax ***//*** unterstützt. Die beiden Zeichen ***//*** leiten einen Kommentar bis zum Ende der Zeile ein.

C-Compiler für Microcontroller unterstützen nicht alle ANSI-C Funktionen da gewisse Funktionen bei Kleinsystemen keinen Sinn ergeben (z.B. scanf).

Berechnungen von Kommastellen (float) sind mit grossem Aufwand verbunden und benötigen sehr viel Rechenzeit und Speicherplatz. Wenn immer möglich, sollte daher in Kleinsystemen mit **ganzen Zahlen ohne Kommastellen** gerechnet werden!

Darstellung von Zahlen und Zeichen

Bezeichnung	***Beispiele***
Dezimal	**`100, 55, -12, 10203`**
Oktal	**`01234,022,0745`** vorgestellte 0 bedeutet Oktal
Hexadezimal	**`0xFF, 0xAFFE, 0x22`** vorgestellt 0x bedeutet Hex
Boolean (in C kein Zahlentyp)	**`Zahl = 0 -> FALSE`** **`Zahl != 0 -> TRUE`** (!= bedeutet ungleich)
Ascii Zeichen (Charakter)	**`'A', '5', 'b', '&', '=', '\n','\x20'`**
Zeichenketten (Stings)	**`"hallo", "guten Tag", "23.Januar\n",`**

Escape Sequenzen

Escape Sequenzen können als Charkter oder in Strings wie ein ASCII-Zeichen eingesetzt werden. So bewirkt z.B. ein \n einen Zeilenvorschub. Escape Sequenzen werden hauptsächlich zusammen mit Strings angewendet.

Sequ	***Name***	***Hex***	***Dez***
`\n`	newline, linefeed	0A	10
`\t`	tab (Tabulator)	09	9
`\b`	backspace	08	8
`\f`	formfeed	0C	12
`\a`	bell (alarm)	07	7
`\r`	return	0D	13
`\v`	vertical tab	0B	12
`\0`	null	00	0
`\"`	ASCII quote	22	34
`\\`	ASCII back slash	5C	92
`\'`	ASCII single quote	27	39
`\?`	Fragezeichen	5F	95
`\x40`	ASCII Zeichen 40h (@) einfügen		

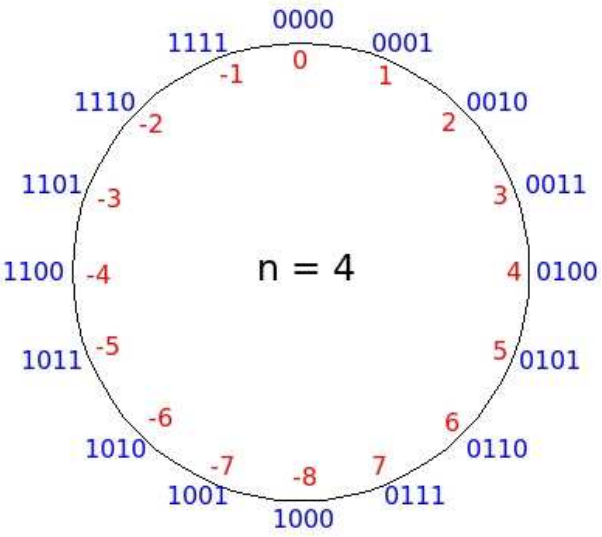
Beispiel: Neue Zeile vor und nach Hello world

```
printf("\nHello world\n");
```

Daten- (Zahlen) Typen in C

Datentypen geben an, wie eine gespeicherte Zahl behandelt wird (mit oder ohne Vorzeichen, Kommastelle oder nicht usw.). Viele MCU C-Compiler definieren zusätzlich noch eigene Typen.

Variablen mit Vorzeichen werden im 2er Komplement dargestellt. Das MSBit ist damit immer das Vorzeichen (MSB = 1 bedeutet eine negative Zahl).



Darstellung der vorzeichenbehafteten 4-Bit Zahlen im Zweierkomplement

Die Zahlenbereiche der Typen int, short und long sind abhängig von der verwendeten MCU oder dem PC. Gebräuchliche Standard C-Typen für MCUs sind:

Bezeichnung	***Bereich***	***Bytes***
char	-128..+127	1
unsigned char	0..255	1
int short	-32768..+32767	2
unsigned int unsigned short	0..65535	2
long	-2147483648.. +2147483647	4
float double	1.17549E-38 .. 3.40282E+38	4

Zum hier beschriebenen Zahlen-Typ kann noch eine weitere Definition hinzukommen:

Bez.	***Bedeutung***
volatile	Kann den Inhalt auch unabhängig vom Programm ändern (z.B. Port oder Interruptbehandlungsroutine)
const	Fester Wert, kann den Wert zur Laufzeit nicht ändern
signed	Bereich erstreckt sich über positive und negative Zahlen
auto	Automatisch, auf dem Stack untergebracht
extern	In einem anderen Programmfile definiert
static	Bleibt während der ganzen Programmlaufzeit bestehen

Deklaration der Variablen

- Variablen müssen vor dem ersten Auftreten deklariert werden.
- Variablen die für alle Funktionen Gültigkeit haben, werden zu oberst im Programmdeklariert (globale Variablen). Globale Variablen möglichst sparsam einsetzen.
- Variablen die in einer Funktion Gültigkeit haben müssen unmittelbar nach dem Funktionskopf, vor der ersten Programmanweisung deklariert werden!

Variablen **nur** für den Bereich deklarieren, in welchem sie angewendet werden z.B. in einer Funktion!

Beispiele: Deklarationen von Variablen:

```
char Buchstabe;
unsigned char MyVar;
unsigned int i,n,k;
```

Deklarationen mit Anfangswert:

```
char Ziffer = 'A';
unsigned int Faktor = 0xAFFE;
```

Deklaration von Konstanten:

```
const int ANZWD = 7;
```

In Funktionen definierte Variablen verlieren beim Austritt aus der Funktion ihren Wert. Soll der Inhalt einer Variablen erhalten bleiben, muss sie als static deklariert werden:

```
static int E_Count = 0;
```

Deklarationen von Arrays

Arrays sind indizierte Variablen oder Konstanten und werden durch eckige Klammern gekennzeichnet. Die Variablen Array Messw hat hier 20 int Elemente. Die Indizierung beginnt in C immer bei 0! Hier also 0..19.

```
int Messw[20];
```

Zugriff auf einzelne Felder:

```
Messw[0] = 12;
Messw[19] = Messw[12];
```

Bemerkung: Soll eine Array gelöscht werden, muss jedes Element einzeln gelöscht werden:

```
For(i=0;i<20;i++) Messw[i]=0;
```

Strings (Zeichenketten) werden in C als char Arrays behandelt. Zeichenketten sind immer mit dem Wert 0 abgeschlossen. Beide Zuordnungen ergeben einen String:

```
char Wort[] = {‘h‘,‘o‘,‘i‘,0};
char Wort[] = {“hoi“};
```

Array mit Konstanten: Anhand der Initialisierung wird bei leeren eckigen Klammern die Arraygrösse automatisch bestimmt.

Bemerkung: Nicht mit allen C-Compilern können Variablen-Arrays initialisiert werden. Bei globalen Konstanten ist das jedoch möglich (evtl. müssen die Arrays mit static deklariert werden):

```
const int Step[]={12,22,0x33};
const char MyStr[] = {“Hallo“};
static const int Werte[] =
                 {1,3,5,7,9,11};
```

Arrays mit Strings können wie folgt deklariert werden:

```
const char *WDAYS[] =
      {“MO“,“DI“,“MI“,“DO“,“FR“};
```

Mehrdimensionale Arrays:

```
static const int dat[2][3] =
               {{12,15,16},
                {22,33,55}};
```

Zugriff auf ein Feld dieser Array:

```
n = dat[1,2]  ;  /* n = 55 */
n = dat[0,0]  ;  /* n = 12 */
```

Typen umwandeln (Casts)

Der C-Compiler wandelt selber Daten von einem Typ in den anderen um, wenn dies erforderlich ist.

Mit Casts zwingt man den Compiler, an einer bestimmten Stelle eine solche Umwandlung durchzuführen. Dabei wird der Typ in Klammern dem Ausdruck vorgestellt. Beispiele:

```
int MyI1 = 3,MyI2 = 5;
float Res;

Res=MyI1/MyI2;          //Res=0
Res=(float)MyI1/MyI2;   //Res=0.6
Res=MyI1/(float)MyI2;   //Res=0.6
Res=(float)(MyI1/MyI2); //Res=0
Res=3/5;                //Res=0
Res=3/5.0;              //Res=0.6
```

Meine Notizen:

Aufzählungskonstanten (enum)

Eine Aufzählung ist die Folge von konstanten Werten:

```
enum {aus,links,rechts};
enum status {aus,links,rechts};
```

Deklariert die drei Konstanten
aus=0, links=1, rechts=2.

Ohne Angabe wird dem ersten Namen 0 zugewiesen, dem zweiten Namen 1 usw.

```
enum {susi=5,maya,doris};
```

Hier erhält susi = 5, maya = 6 usw.

Es kann auch jedem Namen ein Wert zugeordnet werden:

```
enum {peter=25,hans=33,alois=16};
```

Makros (#define)

Einem definierten Namen kann mit einem Macro C-Code zugeordnet werden. Beim Compilieren setzt der Compiler dann den vordefinierten Code ein. So wird Programm-Code übersichtlicher. Definitionen werden oben im Programm definiert.

Beispiele:

Taste Stop ist an PORTC Bit 2 angeschlossen. Das folgende Makro maskiert die Taste:

```
#define Taste_Stop (PORTC&0x04)
```

Im C-Code kann dann das Makro wie folgt eingesetz werden:

```
if( Taste_Stop ) {...}
```

Einem Macro können auch Parameter übergeben werden (es sind auch mehrere Parameter möglich).

```
#define LED_on(Nr) PORTB|=(1<<Nr)
```

Aufruf des Makros im Programm:

```
LED_on(7);    //LED Nr7 einschalten
```

Dieses Makro setzt den Portausgang PORTB Bit7 auf 1 (LED_on)

Datei einbinden (#include)

In einem C-Programm kann an beliebigen Stellen eine bereits bestehende Datei (z.B. mit Standardfunktionen) eingebunden werden:

Dateien aus der Compilerbibliothek einbinden:

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

Datei von einem beliebigen Verzeichnis einbinden:

```
#include "C:\MyProgs\MyFunc.c"
```

Pointer (Zeiger)

Für jede Variable wird im Speicher des Microcontroller Systems ein Platz mit einer Adresse reserviert. Diese Adresse wird in C als Pointer (Zeiger) bezeichnet. In Microcontrollersystemen wird meistens mit 2 Byte grossen Adressen gearbeitet. Die Adresse eines Ausdrucks kann mit dem **&-**Operator ermittelt werden:

```
unsigned int Res, Zahl;

Res=PORTJ //Res=Inhalt PortJ
Res=&PORTJ    //Res=Adresse PortJ
```

In diesem Fall ist jedoch Res eine Variable und kein Pointer! Ein eigentlicher Pointer muss mit dem *****-Operator als solcher definiert werden:

```
int *Ptr;
/* Deklariert einen Zeiger Ptr der
auf einen int zeigt */

Ptr = &Zahl;
/* Nun zeigt Ptr auf die int Var
Zahl */

*Ptr = 0x55;
/* Inhalt der Adr. Ptr und damit die
Var Zahl hat den Wert 0x55  */
```

Einem Pointer kann nicht direkt ein Zahlenwert zugeordnet werden. Pointer können jedoch inkrementiert und dekrementiert werden!

Syntax	*Bedeutung*
&MyVar	Liefert die Adresse von MyVar
int *MyPtr	Deklariert den Pointer MyPtr
MyVar=*MyPtr	Übergibt Inhalt von MyPtr

Einwertoperatoren

Diese Operatoren wirken auf einen Wert und ergeben ein Resultat. In den untenstehenden Beispielen werden folgende Variablen verwendet:

```
int Wert    /* -32768..+32767     */
int *Zeiger /* Zeiger auf Speicher*/
int Bool    /* 0=false, sonst=true*/
```

Op	*Funktion*	*Darst.*	*Beispiel, Resultat*
~	Komplement	~Wert	~0xAA = 0x55
!	Logisches Komplement	!Bool	!5=0; !100=0; !0=1; !1=0;
&	Adresse von	&Wert	Gibt die Adresse der Variablen Wert an
-	negativ	-Wert	-(-100)=100
+	positiv	+Wert	+100=100
++	Pre-increment	++Wert	Wert vor Operation incrementieren
++	Post-increment	Wert++	Wert nach Operation incrementieren
--	Pre-decrement	--Wert	Wert vor Operation decrementieren
--	Post-decrement	Wert--	Wert nach Operation decrementieren
*****	Referenz	*Zeiger	Inhalt auf welchen der Zeiger verweist

Arithmetische Operatoren

Bei der Durchführung von arithmetischen Operationen ist unbedingt darauf zu achten, dass die Zahlenbereiche nicht überschritten werden (dies gilt auch für Zwischenresultate!). Auch Operationen zwischen ‚signed' und ‚unsigned' Variablen können zu Berechnungsfehlern führen. Es ist von Vorteil, für arithmetische Operationen vorzeichenbehaftete Variablen zu verwenden.

Beachten Sie, dass der Divisionsoperator für den Typ **int** und **float** derselbe ist. Bei der Integerdivision werden die Stellen hinter dem Komma abgeschnitten (keine Rundung). Mit der Eingabe einer Kommastelle oder dem Cast-Operator kann eine Floatdivision erreicht werden.

Operation	*Res.*	*Bemerkung*
15/2	7	Integerdivision
15.0/2	7.5	Floatdivision
(float)15/2	7.5	Floatdivision (Cast Op.)

Op.	*Funktion*	*Beispiel*
+	Addition	20+0x10=36
-	Subtraktion	40-60=-20
*****	Multiplikation	10*25=250
/	Division	159/10=15
%	Modulo (Rest)	159/10=9
<<	Links schieben	24<<2=96
>>	Rechts schieben	24>>2=6

Logische Operatoren

Mit den **bitweise** wirkenden logischen Operatoren können mit einer Maske innerhalb eines Bytes oder Wortes einzelne Bits gesetzt (OR), gelöscht (AND) oder invertiert (EXOR) werden.
Wichtig: Logische Operationen immer in Klammern setzen:

```
if((PORTA & 0x80) == 0x80){...}
```

Op.	*Funktion*	*Zahlenbeispiel*
&	Bitweise AND	0xAA & 0x0F ergibt 0x0A
|	Bitweise OR	0xAA | 0x0F ergibt 0xAF
^	Bitweise EXOR	0xAA ^ 0x0F ergibt 0xA5
~	Bitweise NOT	~0xAA ergibt 0x55

Mit den folgenden logischen Operatoren werden boolesche Ausdrücke in Vergleichen zusammen verknüpft. Ein Wert (int, char) wird als FALSE angenommen wenn er gleich 0 ist. Ist ein Wert ungleich 0 wird er als TRUE angenommen.

Op.	*Funktion*	*Beispiel*
&&	AND	25 && 0 = 0 (FALSE) 25 && 3 = 1 (TRUE)
||	OR	25 || 0 = 1 (TRUE) 0 || 0 = 0 (FALSE)
!	NOT	!25 = 0; !0 = 1

Vergleichsoperatoren

Vergleichsoperatoren werden in Entscheiden zum Verzweigen und in Schlaufen als Abbruchkriterien eingesetzt. Oft müssen mehrere Vergleiche zusammen ein bestimmtes Resultat ergeben. In solchen Fällen können einzelne Vergleiche mit AND (&&), OR (||) und NOT (!) kombiniert werden. Das Resultat ist immer 1 (TRUE) oder 0 (FALSE).

Op.	*Funktion*	*Beispiel*
==	gleich	100 == 4 (0, FALSE) 12 == 12 (1, TRUE)
!=	ungleich	100 != 4 (1, TRUE) 12 != 12 (0, FALSE)
<	kleiner	100 < 4 (0, FALSE) 12 < 12 (0, FALSE)
<=	kleiner gleich	100 <= 4 (0, FALSE) 12 <= 12 (1, TRUE)
>	grösser	100 > 4 (1, TRUE) 12 >12 (0, FALSE)
>=	grösser gleich	100 >= 4 (1, TRUE) 12 >= 12 (1, TRUE)

Zuordnungsoperatoren

Zuordnungsoperatoren weisen einer Variablen eine Konstante oder den Inhalt einer anderen Variablen zu.

Operation	*Ergebnis*
A=5	Zahl 5 wird der Var A zugewisen
A=B=C=4	A,B,C werden auf 4 gesetzt
A=B=C	A, B sind gleich wie C
A+=B	A=A+B
A-=B	A=A-B
A*=B	A=A*B
A/=B	A=A/B
A<<=B	A=A<<B
A>>=B	A=A>>B
A&=B	A=A&B
A|=B	A=A|B
A^=B	A=A^B

```
/* Beispiele für Zuordnungen:     */

PORTA=0xFF      /* PORTA Bits ->1 */

PORTA|=0x01     /* PAJ0 auf 1     */

PORTA&=(~0x80)  /* PA7 auf 0      */

PORTA^=0x0F     /* PA0..3 invers  */

PORTA=PORTB=0   /* PA, PB auf 0   */

PORTA<<=4       /* PA 4 bit links */
```

Reservierte Ausdrücke

Die hier aufgeführten Ausdrücke haben in C eine spezielle Bedeutung und dürfen nicht für andere Namen (Variablen, Funktionen) eingesetzt werden!

Ausdruck	*Bedeutung*
asm	Assembler Code einfügen
auto	Definiert Variable auf Stack
break	Bricht Programmablauf innerhalb einer Struktur ab
case	Pfad in einer Mehrfachverzweigung
char	8 Bit Integer Deklaration
const	Konstantendeklaration (ROM)
continue	Gehte zum Loop-Anfang
default	Mehrfachverzweigung: Pfad wenn keine andere Bedingungen zutrifft
do	do..while Schlaufe
double	Deklaration Var mit Gleitkomma
else	Verzweigung alternativer Pfad
extern	In einem anderen File definiert
float	Deklaration Var mit Gleitkomma
for	Schlaufe mit def. Anzahl Durchläufe
goto	Spring zu einem bestimmten Label
if	Einfache Verzweigung
int	16Bit Integer (wie short beim HC12)
long	32Bit Integer
register	Spezifiziert eine locale Var
return	Parameterrückgabe in Funktion
short	16Bit Integer
signed	Var.-Deklaration vorzeichenbehaftet
sizeof	Gibt Grösse eines Objekts zurück
static	Var.-Deklaration immer im Speicher
struct	Definition einer Datenstruktur
switch	Mehrfachverzweigung
typedef	Definition von neuen Datentypen
unsigned	Nicht vorzeichenbehaftet >=0
void	Kein Parameter wird übergeben
volatile	Kann unabhängig vom Prog. ändern
while	While Schlaufe

Bindungsprioritäten

In der Mathematik bindet die Multiplikation Multiplikation stärker als die Addition (Punkt- vor Strichrechnung). In C gibt es verschiedene Stufen von Operationsprioritäten, die in der folgenden Tabelle in abnehmender Bindungsstärke (von oben unten) dargestellt sind.

Op. Gruppe	*Operatoren*	*Reihenfolge*
Expression	`( )[ ] . ->`	links nach rechts
Unary	`- + ~ ! * & ++ --`	rechts nach links
Multipl.	`* / %`	links nach rechts
Additive	`+ -`	links nach rechts
Shift	`<< >>`	links nach rechts
Relational (inequality)	`< <= > >=`	links nach rechts
Relational (equality)	`== !=`	links nach rechts
Bitwise And	`&`	links nach rechts
Bitwise Xor	`^`	links nach rechts
Bitwise Or	`|`	links nach rechts
Logical And	`&&`	links nach rechts
Logical Or	`||`	links nach rechts
Conditional	`? :`	rechts nach links
Assignment	`= *= /= %= += -= <<= >>= &= |= ^=`	rechts nach links

Formatierung mit printf, scanf

printf und scanf sind wichtige I/O Funktionen in C bei Systemen mit Bildschirm und Tastatur.

printf gibt formatierten Text und Zahlenwerte an das ‚standard output file' (Bildschirm) aus.

Beispiel:

```
printf("PJ=%x,PD=%x\n",PORTJ,PORTD);
```

Ausgabe am Bildschirm:
PJ=004F,PD=003B

%x Platzhalter an dessen Stelle der aktuelle Portzustand im angegebenen Format eingesetzt wird.
\n bewirkt eine neue Zeile nach der Ausgabe

scanf liest Charakter vom ‚stream' (meistens Tastatur) ein, und weist die Daten entsprechend den Formatierungszeichen den C Variablen zu.

Beispiel:

```
scanf ("%f",&Var_U);
```

Auch scanf braucht einen Formatstring mit dem Format der Zahl die eingelesen werden soll. Der eingelesene Wert wird dann der Variablen übergeben deren Adresse nach dem Formatstring angegeben wird.

Die Kombination %f ist ein Platzhalter für eine einzusetzende Float-Zahl. Die wichtigsten Formate für solche Platzhalter sind in der folgenden Tabelle aufgeführt.

Format	*Typ*	*Bemerkung*
%c	char	Einzelner Charakter
%d	int	Integer dezimal
%e	float	Signed Wert im ENG Format
%f	float	Signed Wert in exponential Darstellung
%i	int	Integer (signed)
%s	char string	Zeichenkette
%u	int	Unsigned integer Darstellung
%x	int	Hexadezimale Ausgabe

Fliesskomma-Formatierung:

Unmittelbar nach dem % Zeichen kann mit dem Zeichen # die Fliesskomma-Ausgabe noch formatiert werden:

%#[W].[P]f

[W] : Eine Zahl die angibt, in wie vielen Stellen die Ausgabe im Minimum erfolgen soll (optional).

[P] : Eine Zahl die angibt, wie viele Stellen im Maximum nach dem Dezimalpunkt ausgegeben werden sollen (optional).

Beispiel:

```
Var_U = 230.56;
printf ("U=%#8.1f V",Var_U);
```

Ausgabe am Bildschirm:
U=   230.6 V

Darstellung eines C-Programms

```
/*********************************************************
  Projectname:  balken
  Filename:     balken
  Autor:        Bruno Wamister
  Version:      1.0
  Date:         31.07.2007 06:40:24
  Function:     Der Wert des AD-Wandlers Kanal 1
                (Potentiometer) wird als Balken auf der
                LED-Zeile dargestellt

*********************************************************/
// Einbinden von bestehenden Modulen
#include <kiss.h>         // ATMega32 Register

//-------------------------------------------------------
// Definitionen
#define BALKEN PORTB     //PORTB heisst nun auch BALKEN
#define BREITE 255/8      //Anzeige-Breite pro LED
//-------------------------------------------------------
// Globale Konstanten und Variablen


//-------------------------------------------------------
//Berechnung der Balkengrösse anhand des eingegebenen
//Wertes In_Wert
char Balken_Groesse(unsigned int In_Wert){
     int Balken_Wert;
     Balken_Wert = ~(0xFF<<(In_Wert/BREITE));
     return Balken_Wert;   //Rückgabewert der Funktion
}
//-------------------------------------------------------
// Hauptfunktion (wird beim Start ausgeführt)
int main(void){
     init_kiss();         //Initialisierung der Ports
     AtdInit();           //Initialisierung des ADC
     while(1==1){         //endlose Schlaufe
          BALKEN=Balken_Groesse(AtdResult());
     }
     return 0;
}
//-------------------------------------------------------
```

Programmkopf: Enthält alle wichtigen Angaben zum Programm. Wer hat wann an diesem Programm gearbeitet? Funktion des Programms? Um welche Version des Programms handelt es sich hier?

Kommentare: Programme immer kommentieren!
Über mehrere Zeilen innerhalb der Zeichen /*...*/:
`/* dies ist ein ANSI-C Kommentar */`
C++ auch nach // bis zum Zeilenende:
`// Dies ist C++ Kommentar`

Include: Bestehende Standardfunktionen je nach Anwendung einbinden. Auch die MCU Registerdefinitionen werden so eingebunden.

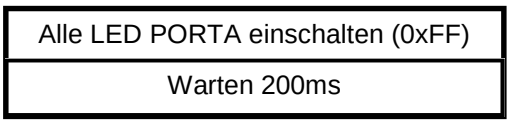
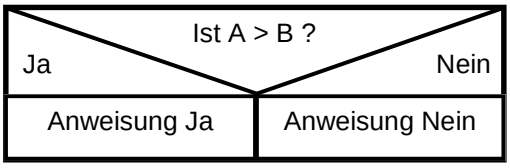
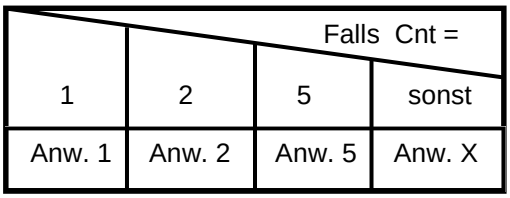
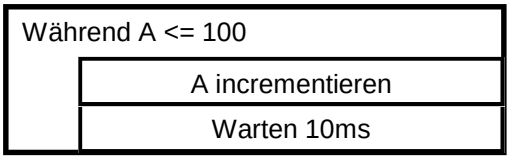
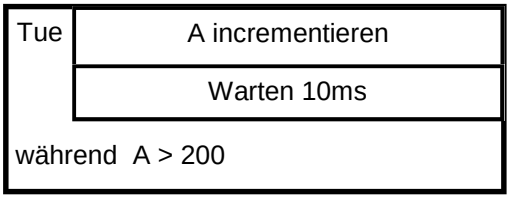
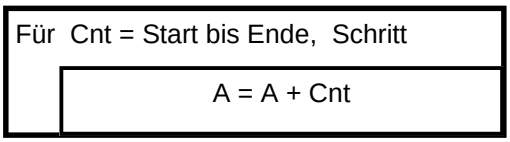
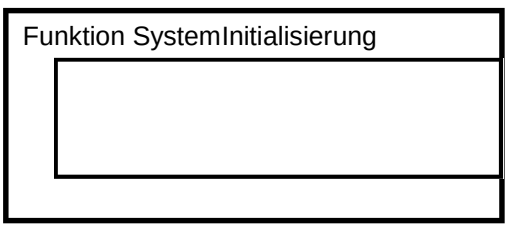
Define: Zuordnung von Namen zu Variablen, Ports oder Register. Den Ein- und Ausgängen immer die im Systen verwendeten Signalnamen zuordnen! Auch Macros können hier definiert werden

Globale Deklarationen: Hier werden Konstanten und Variablen deklariert, die in allen Funktionen gebraucht werden. Möglichst vermeiden! Variablen dort definieren wo sie angewendet werden!

Function: Unterprogramme, die aus anderen Unterprogrammen (inkl. main) aufgerufen werden können. An Funktionen können beliebig viele Werte (Parameter) übergeben werden. Eine Funktion kann einen Wert zurückgeben. Gibt die Funktion einen Wert zurück, muss das Statement ‚return' auf den Rückgabewert verweisen. In einer Funktion definierte Variablen sind nur innerhalb dieser Funktion bekannt!

Function main: Die Funktion ‚main' muss in jedem C-Programm enthalten sein. Beim Start des Programms wird diese Funktion zuerst ausgeführt.
In MCU-Anwendungen enthält die Funktion main meistens Aufrufe zur Systeminitialisierungen und eine endlose Schlaufe in welcher die verschiedenen Funktionen in der richtigen Reihenfolge aufgerufen werden. Die Verwendung von weiteren Funktionen erhöht die Übersicht! Die Funktion main sollte nicht grösser als eine Bildschirmseite sein!

Strukturen nach Nassi-Shniderman

Strukturblock	*Funktion, Eigenschaft*
	**Anweisung:** Block für Eingaben, Ausgaben, Berechnungen usw.
	**Verzweigung:** Anhand einer Bedingung (Frage), die nur die Antworten *ja* oder *nein* zulässt wird das Programm verzweigt. Entsprechen dem Resultat wird die Anweisung Ja oder Nein ausgeführt
	**Mehrfachverzweigung:** Verzweigung anhand des Inhaltes einer Variablen. Je nach Inhalt wird der entsprechende Block ausgeführt. Ist für einen Inhalt der Variablen keine Anweisung vorgesehen, wird durch *sonst* verzweigt.
	**Schlaufe mit Anfangstest:** Trifft die Anfangsbedingung zu, werden die Anweisungen in der Schlaufe ausgeführt bis die Anfangsbedingung *falsch* ist. Ist die Anfangsbedingung bereits beim Eintritt *falsch*, werden die Anweisungen nicht ausgeführt.
	**Schlaufe mit Endtest:** Die Anweisungen in der Schlaufe werden sicher einmal ausgeführt, am Ende wird geprüft, ob die Schlaufe nochmals durchlaufen wird (Endtest *richtig*) oder ob die Schlaufe verlassen wird (Endtest *falsch*).
	**Schlaufe für eine bestimmte Anzahl Durchläufe:** Die Anzahl der Schlaufendurchläufe ist vor dem Eintritt in die Schlaufe bekannt. Beim Eintritt wird der Zählvariablen *Cnt* der Startwert *Start* zugeordnet. Die Zählvariable wird bei jedem Durchgang um *Schritt* erhöht. Es wird aus der Schlaufe ausgetreten sobald *Cnt* grösser als *Ende* ist.
	**Aufruf eines Unterprogramms (Funktion, Prozedur):** Für das aufgerufene Unterprogramm existiert ein separates Struktogramm
	**Programm, Unterprogramm, Block:** Um jedes in sich abgeschlossene Programm oder Unterprogramm wird diese Struktur gelegt. Oben wird der Name des Programms angegeben, der auch bei einem Aufruf verwendet wird.

Strukturblöcke und C-Syntax

Struktur:	*Programmierung in C*
Programm Demo Struktur-Block	**Programmrahmen:** umschliesst das Demo Hauptprogramm ‘main‘ oder eine Funktion

DDRA = $FF
PORTA = PORTB +5
	Anweisungen:
	`DDRA=0xFF;`
	`PORTA=PORTB+5;`

Unterprogramm SysInit	**Aufruf einer Funktion ohne Parameter:** `SysInit();`

Wenn PortB = $A3
Ja: PortA = $AA
Nein: PortA = $55
	Einfache Verzweigung (if, else):
	`if (PORTB == 0xA3){`
	`  PORTA = 0xAA;`
	`}`
	`else{`
	`  PORTA = 0x55;`
	`}`

Falls Eing **aus:**
1: PortA increm
2: PortA decrem
3: PortA = 0
sonst: PortA = $FF
	Mehrfachverzweigung (switch):
	`switch (Eing) {`
	`  case 1: ++PORTA; break;`
	`  case 2: --PORTA; break;`
	`  case 3: PORTA=0; break;`
	`  default: PORTA=0xFF;`
	`}`

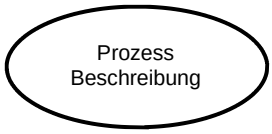
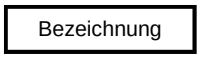
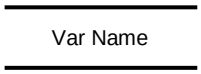
Während PortA <= $80
PortA mit 2 multiplizieren
1000ms warten
	Schlaufe mit Anfangstest (while):
	`while (PORTA<=0x80){`
	`  PORTA<<1; //1 Mal links schieben`
	`  delay_ms(1000); //Befehl systemabhängig`
	`}`

tue
PortA lesen und invertiert an PortB ausgeben
während PortA <> $FF
	Schlaufe mit Endtest (do...while):
	`do {`
	`  PORTB=(~PORTA);`
	`}`
	`while(PORTA != 0xFF);`

Für Var_i **von** 1 **bis** 10, Schritt 1 (aufwärts)
Var_i an PortJ ausgeben
500ms warten
	Schlaufe für bestimmte Anzahl Durchläufe
	(for (Anfangswert; Laufbedingung; Increment)):
	`for(Var_i=1;Var_i<=10;Var_i++){`
	`  PORTJ=Var_i;`
	`  Delay_ms(500); //Befehl systemabhängig`
	`}`

DFD-Diagramme nach DeMarco

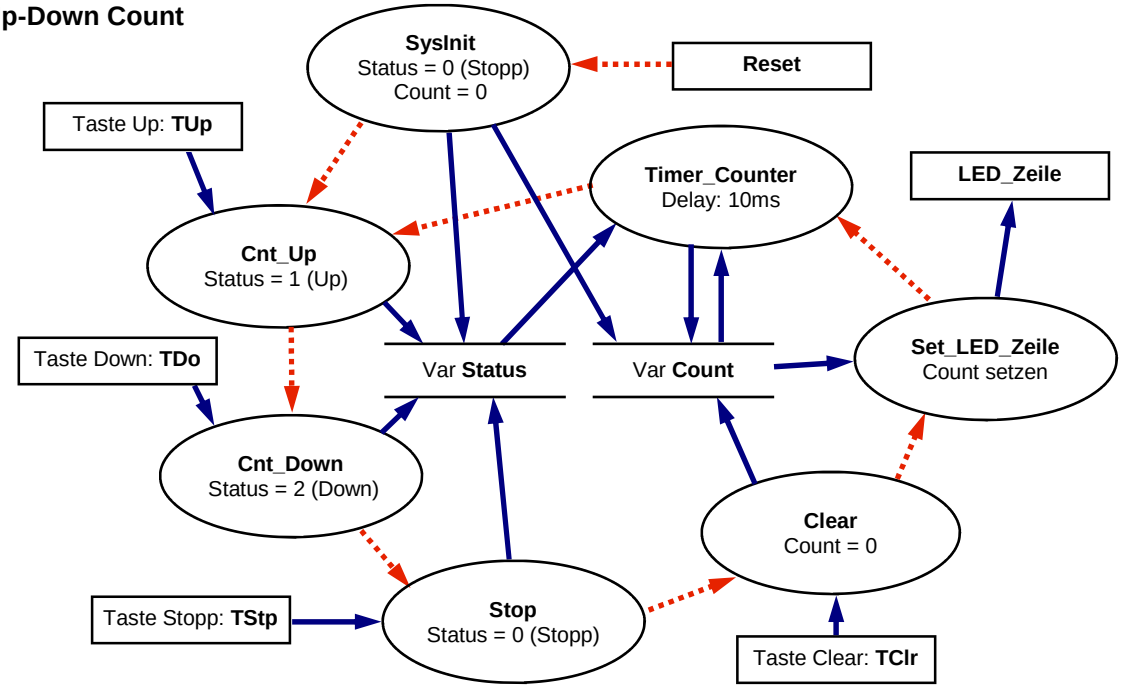
Datenfluss-Diagramme nach DeMarco lassen sich in der Steuerungstechnik, beim Erstellen der Software-Grobstruktur, sehr gut anwenden. Dazu werden die folgenden 5 Symbole eingesetzt:

Symbol:	***Beschreibung:***
	**Data-Process** (Daten-Prozess) Einlesen - Verarbeiten - Ausgeben (EVA) von Daten. Es sind mehrere Daten Ein– und Ausgänge möglich. Durch den Kontrollfluss werden die Prozesse in einer bestimmten Reihenfolge abgearbeitet.
	**Data-Source, Data-Sink** (Daten-Quelle, Daten-Senke) Quellen können nur Daten ausgeben, Senken können nur Daten aufnehmen. Quellen können jedoch auch einen Kontrollfluss auslösen (z.B. Resettaste). Quellen: Sensoren, Schalter, Taster, AD-Wandler, usw. Senken: Aktoren, Anzeige, Motor, Ventil, usw.
	**Data-Memory** (Datenspeicher) Ein Datenspeicher macht nur Sinn, wenn er einen Datenfluss-Eingang und einen Datenfluss-Ausgang aufweist Variablen, Buffer, Arrays usw.
	**Data-Flow** (Datenfluss) Zeigt die Richtung des Datenflusses an. (z.B. Aktor - Prozess - Sensor).
	**Control-Flow** (Kontrollfluss) Zeigt die Richtung des Kontrollflusses (Programmablaufes) an. Ein Prozess kann durch verschiedene Kontrollflüsse aufgerufen werden, sollte in der Regel jedoch nur einen Kontrollfluss-Ausgang aufweisen

Problemstellung: MCU-Anwendung Up-Down-Count

Eine 8Bit-LED-Zeile soll mit einer Frequenz von 1Hz binär auf- oder abwärts zählen. Die LED-Zeile zählt nach einem Impuls mit der Taste TUp aufwärts und nach einem Impuls mit der Taste TDo abwärts. Ein Impuls mit der Taste Clr setzt die LED-Zeile auf 0 und ein Impuls der Taste TStp stoppt den Zählvorgang. Nach einem Stopp starten die Tasten TUp oder TDo den Zählvorgang erneut. Die Taste TStp hat Priorität vor den übrigen Tasten. *Hinweis:* Tastenimpulse haben eine Impulsdauer von >= 40ms.

DFD: Up-Down Count



C-Programm der Aufgabe Up-Down-Count gemäss DFD

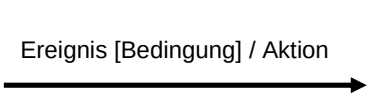
```
/*****************************************************************************
  Projectname:  Up-Down-Count
  Filename:     UpDownCnt
  Author:       Bruno Wamister
  Version:      1.0
  Date:         09.08.2007 17:28:30

  Function:     Impuls TUp    -> LED-Zeile zählt binär aufwärts
                Impuls TDo    -> LED-Zeile zählt binär abwärts
                Impuls TClr   -> LED-Zeile wird auf 0 gesetzt
                Impuls TStp   -> LED-Zeile stoppt zählen
                Impulse >=40ms haben Wirkung, Tasten blockiern sich nicht

*****************************************************************************/
// Einbinden von bestehenden Modulen
#include <kiss.h>        // ATMega32 Register
//---------------------------------------------------------------------------
// Definitionen Maskierung der Tasten. Bezeichnungen gemäss DFD
#define TUp             (TASTER&0x01)          //Taste 0
#define TDo             (TASTER&0x02)          //Taste 1
#define TClr            (TASTER&0x40)          //Taste 6
#define TStp            (TASTER&0x80)          //Taste 7
#define LED_Zeile       LED
//---------------------------------------------------------------------------
//Globale System Variablen gemäss DFD
char Status;
unsigned int Count;
//---------------------------------------------------------------------------
//Eine Funktion für jeden DFD-Data-Process:
void SysInit(void){
      kiss_init();                             //Ports initialisieren
      Status=0;
      Count=0;
}
void Cnt_Up(void){
      if(TUp) Status=1;
}
void Cnt_Down(void){
      if(TDo) Status=2;
}
void Stop(void){
      if(TStp) Status = 0;
}
void Clear(void){
      if(TClr) Count = 0;
}
void Set_LED_Zeile(void){
      LED_Zeile = Count;
}
void Timer_Counter(void){
      static unsigned int Zeit;
      if(++Zeit>=100){
            switch(Status){
                  case 1: Count++; break;       //Cnt_Up
                  case 2: Count--; break;       //Cnt_Down
            }
            Zeit =0;
      }
      delay_ms(10);          //Loop Delay
}
// Hauptfunktion (wird beim Start ausgeführt)
int main(void){              //Ablauf gemäss DFD
      SysInit();
      while(1){              //endlose Schlaufe
            Cnt_Up();
            Cnt_Down();
            Stop();
            Clear();
            Set_LED_Zeile();
            Timer_Counter();
      }
return 0;
}
//---------------------------------------------------------------------------
```

State-Event-Diagramme

Zu steuernde Systeme können verschiedene Zustände (States) annehmen. Durch eine bestimmtes Ereignis (Event) wechselt das System von einem Zustand in einen anderen Zustand und kann dabei Aktionen auslösen. Im State-Event-Diagramm werden zuerst die möglichen Zustände dargestellt und dann die Ereignisse, welche den Zustandswechsel bewirken. Wichtig ist, dass zwischen Zuständen und Ereignissen genau unterschieden wird!

Symbolik für die Zustandsdiagramme (State-Event-Diagram):

Im Zustandsdiagramm werden die folgenden vier Symbole (gemäss SWISSMEM ETMC 1K) verwendet:

Symbol:	***Beschreibung:***
	Startpunkt eines Programms. Systemstart z.B. nach einem Reset oder einem Interrupt. Zeigt den Ausgangszustand eines Systems an.
	Endpunkt eines Programms. Zeigt, wo das Programm endet. Bei Programmen die in einer endlosen Schlaufe ablaufen fehlt dieses Symbol
	Zustand (State). In diesem Zustand verweilt das System bis zu einem neuen Ereignis. Kurze, zutreffende Namengebung! Der Name des Zustandes erscheint dann auch im Programmcode!
	Ereignis (Event). Beschreibt das Ereignis und die Bedingungen, welches den Wechsel von einem alten in einen neuen Zustand bewirken sowie die beim Übergang ausgeführten Aktionen. Angegeben werden: **Ereignis** welches einen Übergang auslöst **Bedingungen** die zusätzlich zum Ereignis vorliegen müssen **Aktionen** die beim Übergang in den neuen Zustand ausgelöst werden

Aufbau der Zustandstabelle:

Zu jedem State-Event-Diagramm gehört auch eine Zustandstabelle aus welcher die Details der einzelnen Funktionen ersichtlich sind. Die Zustandstabelle ist wie folgt aufgebaut:

Zustand (State)	***Ereignis (Event)***	***#***	***Aktion***	***Neuer Zustand***

Das folgende Beispiel zeigt, wie ein Programm mit der State-Event-Technik entwickelt wird:

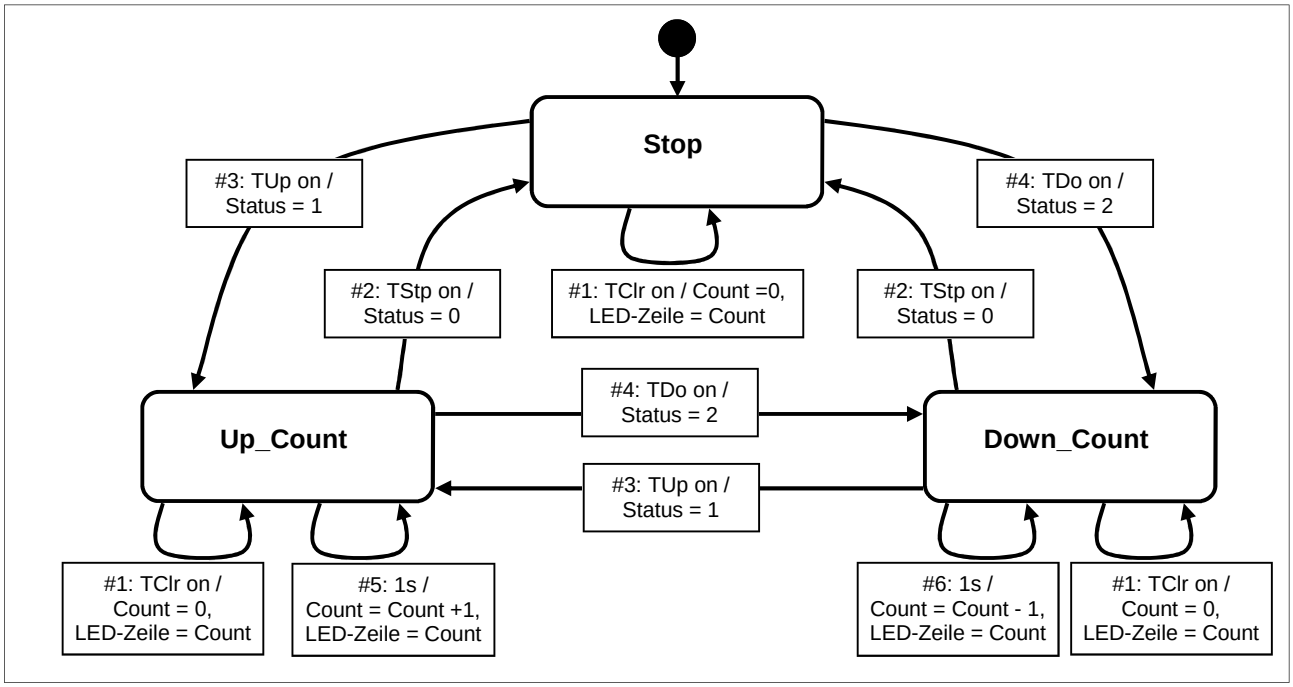
Problemstellung: MCU-Anwendung Up-Down-Count

Eine 8Bit-LED-Zeile soll mit einer Frequenz von 1Hz binär auf- oder abwärts zählen. Die LED_Zeile zählt nach einem Impuls mit der Taste TUp aufwärts und nach einem Impuls mit der Taste TDo abwärts. Ein Impuls mit der Taste TClr setzt die LED_Zeile auf 0 und ein Impuls der Taste TStp stoppt den Zählvorgang. Nach einem Stopp starten die Tasten TUp oder TDown den Zählvorgang erneut. Die Taste TStp hat Priorität vor den übrigen Tasten. *Hinweis:* Tastenimpulse haben eine Impulsdauer von >= 40ms.

Lösungsansatz Up-Down-Count

Das System wird in die drei Zustände (States) Stop, Up_Count, Down_Count aufgeteilt.

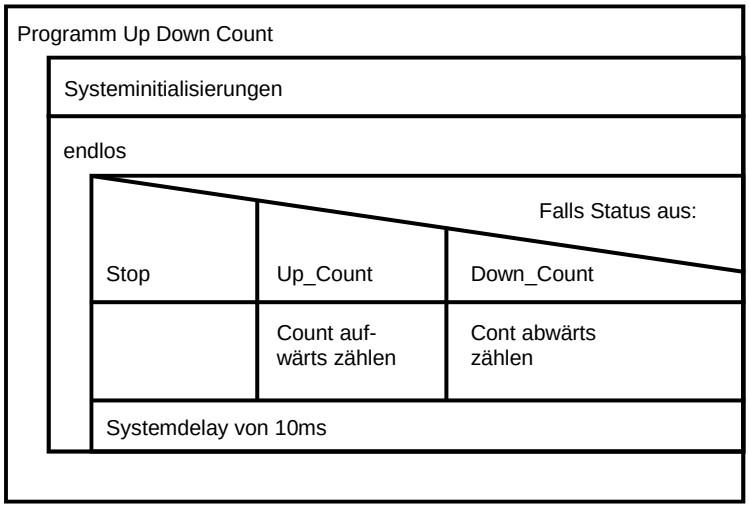
Zustandsdiagramm Up-Down-Count:



Zustandstabelle Up-Down-Count:

Zustand (State)	***Ereignis (Event)***	***#***	***Aktion***	***Neuer Zustand***
Stop (0)	TUp on TDo on TClr on	3 4 1	Status = 1 Status = 2 Count = 0, LED_Zeile = Count	Up_Count Down_Count Stop
Up_Count (1)	TStp on TDo on TClr on Alle 1s	2 4 1 5	Status = 0 Status = 2 Count = 0, LED_Zeile = Count Count = Count + 1, LED_Zeile = Count	Stop Down_Count Up_Count Up_Count
Down_Count (2)	TStp on TUp on TClr on Alle 1s	2 3 1 6	Status = 0 Status = 1 Count = 0, LED_Zeile = Count Count = Count - 1, LED_Zeile = Count	Stop Up_Count Down_Count Down_Count

Grobstruktur des Programms Up-Down-Count:



Bemerkungen:

Damit keine Tasten-Inputs verpasst werden, müssen die Eingänge alle 10ms abgefragt werden. Alle 100 Durchläufe (1s) wird dann Count incrementiert oder decrementiert (je nach aktivem Zustand).

Ein Entprellen der Tasten ist nicht nötig.

C-Programm der Aufgabe Up-Down-Count gemäss State-Event-Diagramm

```
/*****************************************************************************
  Projectname:    Up-Down-Count mit State-Event
  Filename:       UpDownCount_SE
  Author:         Bruno Wamister
  Version:        1.0
  Date:           11.08.2007 08:02:33

  Function:       Impuls TUp  -> LED-Zeile zählt binär aufwärts
                  Impuls TDo        -> LED-Zeile zählt binär abwärts
                  Impuls TClr       -> LED-Zeile wird auf 0 gesetzt
                  Impuls TStp       -> LED-Zeile stoppt zählen
                  Impulse >=40ms haben Wirkung, Tasten blockiern sich nicht

*****************************************************************************/
// Einbinden von bestehenden Modulen
#include <kiss.h>        // ATMega32 Register

//---------------------------------------------------------------------------
// Definitionen Maskierung der Tasten. Bezeichnungen gemäss State-Event
#define TUp             (TASTER&0x01)           //Taste 0
#define TDo             (TASTER&0x02)           //Taste 1
#define TClr            (TASTER&0x40)           //Taste 6
#define TStp            (TASTER&0x80)           //Taste 7
#define LED_Zeile       LED
//---------------------------------------------------------------------------
//Globale System Variablen gemäss State-Event
char Status;
unsigned int Count;
//---------------------------------------------------------------------------
//Funktionen für StateEvent:
void SysInit(void){
      kiss_init();                      //Ports initialisieren
      Status=0;
      Count=0;
}
void Set_New(void){                     //Status, Count, LED_Zeile neu setzen
      if(TDo)  Status=2;
      if(TUp)  Status=1;
      if(TStp) Status=0;
      if(TClr) Count=0;
      LED_Zeile=Count;
}
//---------------------------------------------------------------------------
// Hauptfunktion (wird beim Start ausgeführt)
int main(void){
      static unsigned int Zeit;
      SysInit();
      while(1){                         // endlose Schlaufe
            switch(Status){
                  case 0:               // State Stop
                        Set_New();
                        break;
                  case 1:               //State Up_Count
                        if(++Zeit>=100){
                              Count++;    //aufwärts zählen
                              Zeit =0;
                        }
                        Set_New();
                        break;
                  case 2:               //State Down_Count
                        if(++Zeit>=100){
                              Count--;    //abwärts zählen
                              Zeit =0;
                        }
                        Set_New();
                        break;
            }
            delay_ms(10);               //Systemdelay
      }
      return 0;
}
```

Notizen:

Standard C-Funktionen

Hier eine alphabetische Übersicht über die in C implementierten Standardfunktionen. Diese Funktionen können mit der entsprechenden Headerdatei z.B. #include <stdio.h> in ein Programm eingebunden werden. Je nach Zielsystem und dessen Möglichkeiten (DOS-PC, MCU usw.) werden durch den C-Compiler mehr oder weniger Standardfunktionen unterstützt.

abort()	(void) Programmabbruch <stdlib.h>
abs(i)	Absolutwert <stdlib.h>|<math.h>
access($filnam,mode)	File acc. (0=addr,1=x,2=w,4=r) -> 0:y, -1:n <unistd.h>
acos(Dx)	(double) arccos <math.h>
alloca(size)	(void *) dynam. Speicherreservierung <alloca.h>
asctime(&tm)	(char *) Zeit als String (aus struct tm *) <time.h>
asin(Dx)	(double) arcsin <math.h>
assert(expr)	prueft expr (wahr?) <assert.h>
atan(Dx)	(double) arctan <math.h>
atan2(Dx,Dy)	(double) arctan(x/y) <math.h>
atof($str)	string to float ->double <stdlib.h>|<math.h>
atoi($str)	string to integer ->int <stdlib.h>
atol($str)	string to long ->long <stdlib.h>
brk(void *addr)	Break-Adresse (niedrigste unbenutzte Adr.)
bsearch(,,,,)	(void *) binares Absuchen eines Array <stdlib.h>
cabs(complex z)	(double) Absolutwert komplexer Zahl <math.h>
calloc(nelem,nsize)	(void *) Speicher fuer Vektor reservieren <stdlib.h>
ceil(Dx)	(double) -> aufgerundete int-Zahl aus double <math.h>
clearerr(fp)	(void) loescht den Fehler-Indikator <stdio.h>
clreol()	(void) löscht Bildschirmzeile ab der Cursorpos bis Zeilenende <conio.h>
clrscr()	(void) löscht den Textbildschirm <conio.h>
clock()	(clock_t) Systemzeit in Sekunden <time.h>
close(fildes)	schliesst Datei (Systemaufruf)
cos(Dx)	(double) cos <math.h>
cosh(Dx)	(double) cosh <math.h>
cprintf($format[,arg ...])	Wie printf jedoch für Ausgabe auf Textbildschirm mit Textattr <conio.h>
cputs($s)	schreibt s+LF auf Textbildschirm mit entspr. Textattributen <conio.h>
creat($filnam,mode)	Datei erzeugen, UNIX-mode (Systemaufruf) <sys/stat.h>
ctime(&zeit)	(char *) Zeit als String <time.h>
difftime(time_t,time_t)	(double) Zeitdifferenz in Sek. <time.h>
delline(void)	(void) löscht die Bildschirmzeile auf welcher der Cursor steht <conio.h>
exit(i)	(void) beendet das Programm (0: fehlerfrei) <stdlib.h>
exp(Dx)	(double) exp <math.h>
fabs(Dx)	(double) abs <math.h> (nur fuer double)
fclose(fp)	schliesst Datei (mit FILE *fp) <stdio.h>
fdopen(fildes,type)	(FILE *fp) (vgl. open, creat) <stdio.h>
feof(fp)	EOF, wenn !=0 (FILE *fp) <stdio.h>
ferror(fp)	Fehler, wenn !=0 <stdio.h>
fflush(fp)	schreibt Puffer auf Datei <stdio.h>
fgetc(fp)	holt das naechste Zeichen ->int <stdio.h>
fgets($s,n,fp)	(char *) liest max. n-1 Zeichen (mit LF) <stdio.h>
fileno(fp)	-> Handle einer Datei <stdio.h>
floor(Dx)	(double) -> abgerundete int-Zahl aus double <math.h>
fmod(Dx,Dy)	(double) x modulo y <math.h>
fopen($fname,$mode)	(FILE *) oeffnet eine Datei ("r/w/a +")
fprintf(fp,$format[,arg ...])	formatierte Ausgabe <stdio.h>
fputc(Char,fp)	Ausgabe eines Zeichens <stdio.h>
fputs($s,fp)	Ausgabe einer Zeichenkette (ohne LF, NULL) <stdio.h>
free($ptr)	(void) Freigabe eines Blocks (malloc) <stdlib.h>
freopen($fname,$mode,fp)	ersetzt geoeffneten File fp <stdio.h>
frexp(Dx,&Exp)	teilt double-Zahl in ->Mantisse u. Exp. <math.h>
fscanf(file,$format[,&arg ...])	formatierte Eingabe ->Zaehler <stdio.h>

fseek(fp,Loffset,pos)	Positionierung (pos=0 Anf.,1 akt.,2 Ende) <stdio.h>
ftell(fp)	(long) liefert aktuelle Position <stdio.h>
getc(fp)	Makro, holt ein Zeichen (vgl. fgetc) <stdio.h>
getchar()	liest ein Zeichen von stdin <stdio.h>
getenv($name)	(char *) -> &wert fuer environment-name <stdlib.h>
getpass($s)	(char *) liest ein Passwort ein
gets($s)	(char *) liest String (ohne LF) von stdin <stdio.h>
getw(fp)	holt das naechste Wort ->int <stdio.h>
gmtime(&tm)	(tm *) GMT Zeit <time.h>
gotoxy(int x, int y)	(void) Setzt den Textcursor auf Zeile y in die Spalte x <conio.h>
highvideo()	(void) setzt den folgenden Text auf high-intensity <conio.h>
index($str,Char)	evtl. nicht definiert -> strchr
insline()	(void) fügt an der Cursorposition eine neue Zeile ein <conio.h>
isalnum(c)	Zeichen alphanumerisch? <ctype.h>
isalpha(c)	Buchstabe? <ctype.h>
isascii(c)	ASCII-Zeichen? <ctype.h>
isatty(i)	prueft den Geraetetyp
iscntrl(c)	Control-Zeichen? <ctype.h>
isdigit(c)	Ziffer? <ctype.h>
isgraph(c)	druckbares Zeichen ausser Leerzeichen? <ctype.h>
islower(c)	Kleinbuchstabe? <ctype.h>
isprint(c)	druckbares Zeichen (32..126)? <ctype.h>
ispunct(c)	Satz- oder Sonderzeichen? <ctype.h>
isspace(c)	whitespace? <ctype.h>
isupper(c)	Grossbuchstabe? <ctype.h>
isxdigit(c)	Hexadezimalziffer? <ctype.h>
kill(pid,signal)	sendet Signal an Prozess (pid) <signal.h>
labs(long)	(long) Absolutwert <stdlib.h>|<math.h>
ldexp(Dx,exp)	Zahl aus Bruchteil und Exp. <math.h>
localtime(&tm)	(tm *) Ortszeit <time.h>
log(Dx)	(double) ln <math.h>
log10(Dx)	(double) lg <math.h>
longjmp(env,val)	(jmp_buf env) restauriert Umgebung <setjmp.h>
lowhvideo()	(void) setzt den folgenden Text auf low-intensity <conio.h>
lsearch(,,,,)	(void *) lineares Absuchen eines Array <stdlib.h>
lseek(fildes,off,whence)	setzt Position des Dateizeigers
malloc(size)	(char *) reserviert Speicher ->Block <stdlib.h>
memccpy(dest,src,c,n)	(void *) kopiert Block mit n Byte <string.h>|<mem.h>
memchr($str,c,n)	(void *) sucht n Byte nach c ab <string.h>|<mem.h>
memcmp($s1,$s2,n)	vergleicht n Byte zweier Bloecke <string.h>|<mem.h>
memcpy(dest,src,n)	(void *) kopiert n Byte <string.h>|<mem.h>
memmove(dest,src,n)	(void *) kopiert n Byte <string.h>|<mem.h>
memset($str,c,n)	(void *) setzt n Byte auf c <string.h>|<mem.h>
modf(Dx,&int)	(double) Aufteilung in Int.-Teil und Frak. <math.h>
normvideo()	(void) setzt Textausgabe auf Standard-Vorgaben <conio.h>
open($filnam,mode)	Datei oeffnen (Systemaufruf) <fcntl.h>
perror($s)	(void) -> errno, Ausgabe einer Fehlermeldung <stdio.h>
pow(Dbase,Dexp)	(double) -> Dbase^Dexp (Potenz) <math.h>
printf($format[,arg ...])	formatierte Ausgabe <stdio.h>
putc(c,fp)	Makro: Ausgabe eines Zeichens <stdio.h>
putchar(c)	Ausgabe eines Zeichens an stdout <stdio.h>
puts($s)	schreibt s+LF auf stdout <stdio.h>
putw(i,fp)	Ausgabe eines Wortes an stdout <stdio.h>
qsort(Ptr,anz,size,comp())	(void) sortiert Datenfeld (Quicksort) <stdlib.h>
rand()	generiert Zufallszahl (seed: srand) <math.h>
read(fildes,buffer,nbyte)	Einlesen (Systemaufruf)
realloc(Ptr,size)	(void *) Aenderung eines Speicherbereichs <stdlib.h>
rename($oldnam,$newnam)	Umbenennung einer Datei <stdio.h>
rewind(fp)	(void) positioniert auf den Dateianfang <stdio.h>
rindex($str,Char)	evtl. nicht definiert -> strrchar

scanf($format[,&arg ...])	formatierte Eingabe ->Zaehler <stdio.h>
setbuf(fp,&buf)	(void) Zuordnung eines Puffers zu Stream <stdio.h>
setjmp(env)	(jmp_buf env) speichert Stapelumgebung <setjmp.h>
(*signal(sig,func))()	setzt Reaktion auf Signale fest (System) <signal.h>
sin(Dx)	(double) sin <math.h>
sinh(Dx)	(double) sinh <math.h>
sprintf($s,$format[,arg ...])	formatierte Ausgabe auf String <stdio.h>
sqrt(Dx)	(double) sqrt <math.h>
srand(unsigned seed)	(void) Seed fuer rand() <stdlib.h>
sscanf($s,$format[,&arg ...])	formatierte Eingabe von s->Zaehler <stdio.h>
strcat($s1,$s2)	(char *) -> catenation s1//s2 (-> s1) <string.h>
strchr($str,Char)	(char *) erstes Auftreten von Char <string.h>
strcmp($s1,$s2)	-> <0,0,>0 lexikographisch <string.h>
strcpy($s1,$s2)	(char *) kopiert s2 nach s1 (-> s1) <string.h>
strcspn($s,$sdef)	-> Laenge von $s ohne $sdef <string.h>
strdup($s)	(char *) kopiert s in neuen Bereich <string.h>
strlen($s)	-> Laenge des Strings <string.h>
strncat($s1,$s2,n)	(char *) -> cat. s1//s2, max. n Zeichen <string.h>
strncmp($s1,$s2)	-> <0,0,>0 max. n Zeichen <string.h>
strncpy($s1,$s2,n)	(char *) kopiert s2 nach s1 (n Zeichen) <string.h>
strpbrk($s,$sdef)	(char *) -> erstes Auftreten von sdef in s <string.h>
strrchr($str,Char)	(char *) letztes Auftreten von Char <string.h>
strspn($s,$sdef)	-> Laenge von $s mit Zeichen aus $sdef <string.h>
strstr($s1,$s2)	(char *) -> erstes Vorkommen von s2 in s1 <string.h>
strtod($s,char **endptr)	(double) String nach double <stdlib.h>
strtok($s1,$s2)	(char *) Spaltung von s1 mit Token s2 <string.h>
strtol($s,char **ptr,i)	(long) String nach long <stdlib.h>
system($command)	Ausgabe eines Shell-Befehls <stdlib.h>
tan(Dx)	(double) tan <math.h>
tanh(Dx)	(double) tanh <math.h>
textattr(int attr)	(void) Setzt Textattribut attr:0..3fg, 4..6bg, 7blink mit cprintf <conio.h>
textcolor(int color)	(void) Farbe (0..15) für Textausgabe mit cprintf <conio.h>
textbackground(int color)	(void) Hintergrundfarbe (0..7) für Textausgabe mit cprintf <conio.h>
time(&Lsec)	(long) Zeit in sec seit 1.1.1970, 00.00.00 <time.h>
tmpfile()	(FILE *) temporaere Datei <stdio.h>
tmpnam($filnam)	(char *) Name fuer temporaere Datei <stdio.h>
toascii(i)	convert integer to ascii <ctype.h>
tolower(Char)	convert character to lowercase <ctype.h>
toupper(Char)	convert character to uppercase <ctype.h>
ungetc(int c,fp)	schiebt Zeichen in Datei zurueck <stdio.h>
unlink($filnam)	loescht eine Datei
wherex()	(int) gibt x Koordinate des Textcursors zurück <conio.h>
wherey()	(int) gibt y Koordinate des Textcursors zurück <conio.h>
window(int l,int t, int r, int b)	(void) definiert ein Textfenster innerhalb des Bildschirmes <conio.h>
write(fildes,buffer,nbyte)	Schreiben (Systemaufruf)

Notizen:

L-Kiss: Compiler spezifische Variablen-Typen

Der für das L-Kiss verwendete GNU C-Compiler arbeitet neben den Standard C Variablen-Typen auch mit den folgenden Variablen-Typen:

Bezeichnung	*Zahlenbereich*	*Bezeichnung*	*Zahlenbereich*
uint8_t	0 bis 255	**int8_t**	-128 bis +127
uint16_t	0 bis 65‘535	**int16_t**	-32768..+32767
uint32_t	0 bis 4‘294‘967‘296	**int32_t**	-2‘147‘483‘648 bis 2‘147‘483‘647

L-Kiss: Allgemeine Funktionen <kiss.h>

Mit der Datei kiss.h werden automatisch auch alle Register– und Bitbezeichnungen des ATMEGA32 eingebunden. Diese Datei ist im Memorystick unter: **..\Kiss\WinAvr\avr\include**

L-Kiss #define Makros	
bitset(Register,Bit)	Setzt (auf 1) das Bit (0..7) in der Variablen Register
bitclr(Register,Bit)	Löscht (auf 0) das Bit (0..7) in der Variablen Register
bitpats(Register,Pattern)	Setzt (auf 1) in Register die in Pattern auf 1 gesetzten Bits
bitpatc(Register,Pattern)	Löscht (auf 0) in Register die in Pattern auf 0 gesetzten Bits
char bittst(Register,Bit)	Gibt 0 (false) zurück, wenn das Bit (0..7) in Register den Wert 0 hat sonst ist der Rückgabewert (!=0) true
TRUE	Hat den Wert 1 (wahr)
FALSE	Hat den Wert 0 (falsch)
LED	Wie PORTB (für die Ausgabe an die LED-Zeile)
TASTER	Wie PINC jedoch jedes Bit invertiert (Taster sind 0 aktiv) zum Einlesen der Tasterzeile
L-Kiss allgemeine Funktionen	
void kiss_init(void);	Initialisiert PortB als Ausgang (LED) und PortC als Eingang (Taster)
void atd_init(void);	Initialisiert den ADC Kanal 0 für dauernde Wandlung
uinit8_t atd_result(void);	Gibt den aktuellen 8Bit-Wert des ADC-Kanals 0 zurück
void sound(uint16_t Frequenz);	Gibt am Lautsprecher einen Ton mit der Frequenz aus
void sound_off(void);	Schaltet den Ton am Lautsprecher aus
void delay_ms(uint16_t ms);	Verzögert den Programmablauf um ms
void loop_delay_init(void)	Installiert einen Interrupt mit dem Timer 2 Compare
void loop_delay(uint16_t ms);	Wartet bis die angegebene Zeit erreicht ist und setzt den Zeitzähler wieder zurück. Der Zeitzähler wird durch den Interrupt weiter incrementiert.
void boot_mode(void);	Durch den Aufruf dieser Funktion wird das Anwenderprogramm abgebrochen und der Boot-Mode gestartet (System LED brennt rot)

L-Kiss allgemeine Funktionen	(Ab Kiss-Version 4.2)
`void taster_entprell(void);`	TASTER (PortC) werden entprellt. Diese Funktion muss in einen Loop mit einem Loop-Delay von 4 bis 10ms eingebunden werden. Beim dritten Durchgang durch den Loop werden die den Tastern entsprechenden Bits entprellt in den Variablen TASTER_EPS und TASTER_EPF ausgegeben.
`TASTER_EPS`	Taster statisch entprellt. Bei aktiver Taste hat das entsprechende Bit in TASTER_EPS nach dem dritten Loop-Durchlauf den Wert ‚1' bist der Taster nicht mehr aktiv ist (Zustandstriggerung)
`TASTER_EPF`	Taster dynamisch entprellt. Bei aktiver Taste hat das entsprechende Bit in TASTER_EPF nur während dem dritten Loop-Durchlauf nach dem Tasterdruck den Zustand ‚1' (Flankentriggerung)

L-Kiss: LCD Funktionen <kisslcd.h>

Standardmässig ist der LCD an PortB angeschlossen. Diese Einstellung kann in der Datei **`..\Kiss\WinAvr\avr\include\kisslcd.h`** auf ein beliebiges Port geändert werden.

L-Kiss Funktionen für ein LCD	
`void lcd_init (void);`	LCD für 4 Bit Datenübertragung initialisieren
`void lcd_clr (void);`	LCD löschen, Cursor auf Spalte 1, Zeile 1,
`void lcd_ctrl (char ctrl);`	Ein Byte als Kontrollzeichen zum LCD senden
`void lcd_dat (char dat);`	Ein Byte Daten zum LCD senden
`void lcd_curs_set (char pos);`	Cursor an eine bestimmte Position setzen.
`void lcd_write (char *string);`	String an der Cursorposition am LCD ausgeben
`void lcd_back_on  (void);`	Hintergrundbeleuchtung ein
`void lcd_back_off  (void);`	Hintergrundbeleuchtung aus

Cursorpositionen der einzelnen Zeilen: Die folgenden Werte gelten für Standard LCD (z.B. LM044L, HD44780) mit 4 Zeilen und 20 Spalten			
Anfang der Zeile 1	0x80...0x93	Anfang der Zeile 2	0xC0...0xC3
Anfang der Zeile 3	0x94...0xA7	Anfang der Zeile 4	0xD4...0xE7

L-Kiss: RS232 Funktionen <rs232.h>

Initialisierung der seriellen Schnittstelle (auch für USB). Standard Funktionen für das Empfangen und Senden einzelner Char. Auch wenn die C-Standardfunktion **`printf`** verwendet werden soll, muss zuvor die serielle Schnittstelle mit der Funktion **`rs232_init`** initialisiert werden. Die untenstehenden Funktionen installieren keine Interrupts und ankommende Zeichen werden auch nicht in einem FIFO gepuffert.
Die Datei rs232.h ist im Pfad: **`..\Kiss\WinAvr\avr\include\rs232.h`**

L-Kiss Funktionen für RS232	
`void rs232_init(uint32_t BAUDRATE,uint32_t F_CPU);`	Initialisiert die serielle Schnittstelle (RS232) mit Baudrate (z.B. 19200) und CPU Frequenz [Hz] (z.B. 8000000 für 8MHz)
`char rx_ready(void);`	Gibt true zurück, wenn ein ankommendes Zeichen abholbereit ist
`void tx_ch(char c);`	Wartet bis der Sender der seriellen Schnittstelle frei ist und sendet einen Char über die serielle Schnittstelle
`char rx_ch(void);`	Wartet bis ein Char auf der RS232 empfangen wurde und gibt ihn anschliessend zurück.

L-Kiss: i2c Funktionen <i2c.h>

In der Datei i2c.h sind Funktionen für den Betrieb der i2c Schnittstelle vorbereitet. Die Funktionen benutzen die i2c Hardware im ATMega32. Damit sind auch die Anschlüsse verbindlich:

SDA (PC1) Stecker Port C Pin 4
SCL (PC0) Stecker Port C Pin 3

Hinweis: Parallel zu diesen Port-Pins sind beim L-Kiss Taster und Schalter angeschlossen. Darauf achten, dass Taster und Schalter aus sind.

Die i2c Schnittstelle braucht an beiden Leitungen noch pull up Widerstände. Diese Widerstände müssen extern angebracht werden!
Die Datei rs232.h ist im Pfad: **`..\Kiss\WinAvr\avr\include\i2c.h`**

L-Kiss Funktionen für i2c	(Ab Kiss-Version 4.2)
`void i2c_init(void);`	Initialisiert die Hardware des ATMega32 für den Betrieb der i2c Schnittstelle. (CPU-Clock 8MHz, i2c-Clock 100kHz)
`void i2c_start(void);`	Gibt die Startbedingung aus
`void i2c_write(uint8_t Data);`	Sendet ein Byte auf die i2c Schnittstelle
`uint8_t i2c_read(uint8_t ACK);`	Liest ein Byte von der i2c Schnittstelle. Ist ACK=1 wird ein Aknowledge gesendet (Bit 9 auf Zustand ‚0'), weitere Bytes können gelesen werden.
`void i2c_stop(void);`	Gibt die Stoppbedingung aus

Notizen:

ASCII - Tabelle (AscII = American Standard Code for Information Interchange)

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

Bemerkungen zum ASCII:

Die Zeichen 0 bis 31 sind Kontrollzeichen für die Datenübertragung. Die dargestellten 128 Zeichen könnten mit 7Bit dargestellt werden. In den meisten Fällen wird jedoch der ASCII Code als 8Bit (1Byte) Zahl dargestellt. Im Zahlenbereich 128 bis 255 werden dann Spezialzeichen eingefügt z.B. Umlaute (ä, ö, ü usw.). Hier gibt es verschiedene länderspezifische Varianten.

Dez-, Hex-, Dual- und Graycode

DEZ	10^0		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEX	16^0		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D	2^0	A	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
U	2^1	B	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
A	2^2	C	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
L	2^3	D	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
G	-	A	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
R	-	B	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
A	-	C	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Y	-	D	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Maskierung mit AND

Bei der AND-Verknüpfung bleiben die mit 1 maskierten Bits der Zahl erhalten. Die mit 0 maskierten Bits werden auf 0 gesetzt

			HEX		DEZ		DUAL							
ZAHL	Z		0xAA				1	0	1	0	1	0	1	0
MASKE	M		0xF0				1	1	1	1	0	0	0	0
AND Verknüpfung	Z & M		0xA0				1	0	1	0	0	0	0	0

Maskierung mit OR

Bei der OR-Verknüpfung bleiben die mit 0 maskierten Bits der Zahl erhalten. Die mit 1 maskierten Bits werden auf 1 gesetzt

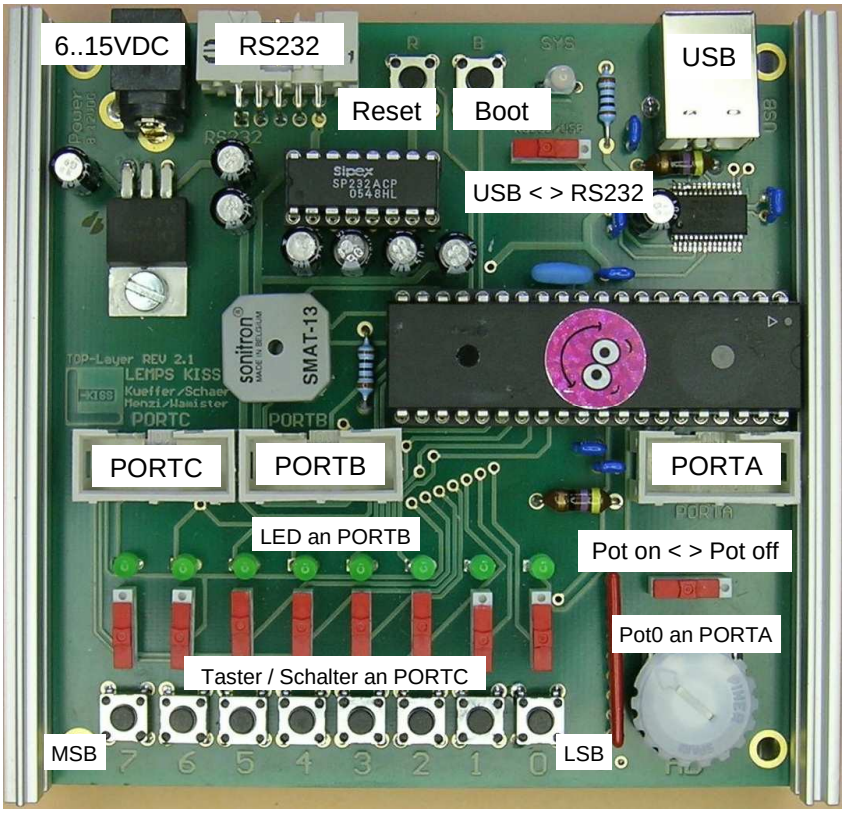
			HEX		DEZ		DUAL							
ZAHL	Z		0xAA				1	0	1	0	1	0	1	0
MASKE	M		0xF0				1	1	1	1	0	0	0	0
OR Verknüpfung	Z | M		0xA0				1	1	1	1	1	0	1	0

Maskierung mit EXOR

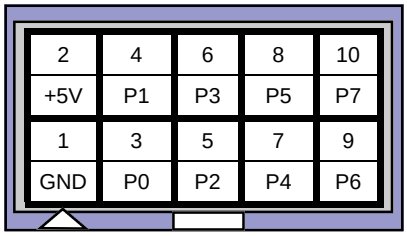
Bei der EXOR-Verknüpfung bleiben die mit 0 maskierten Bits der Zahl erhalten. Die mit 1 maskierten Bits werden invertiert

			HEX		DEZ		DUAL							
ZAHL	Z		0xAA				1	0	1	0	1	0	1	0
MASKE	M		0xF0				1	1	1	1	0	0	0	0
EXOR Verknüpfung	Z ^ M		0xA0				0	1	0	1	1	0	1	0

L-Kiss Stecker

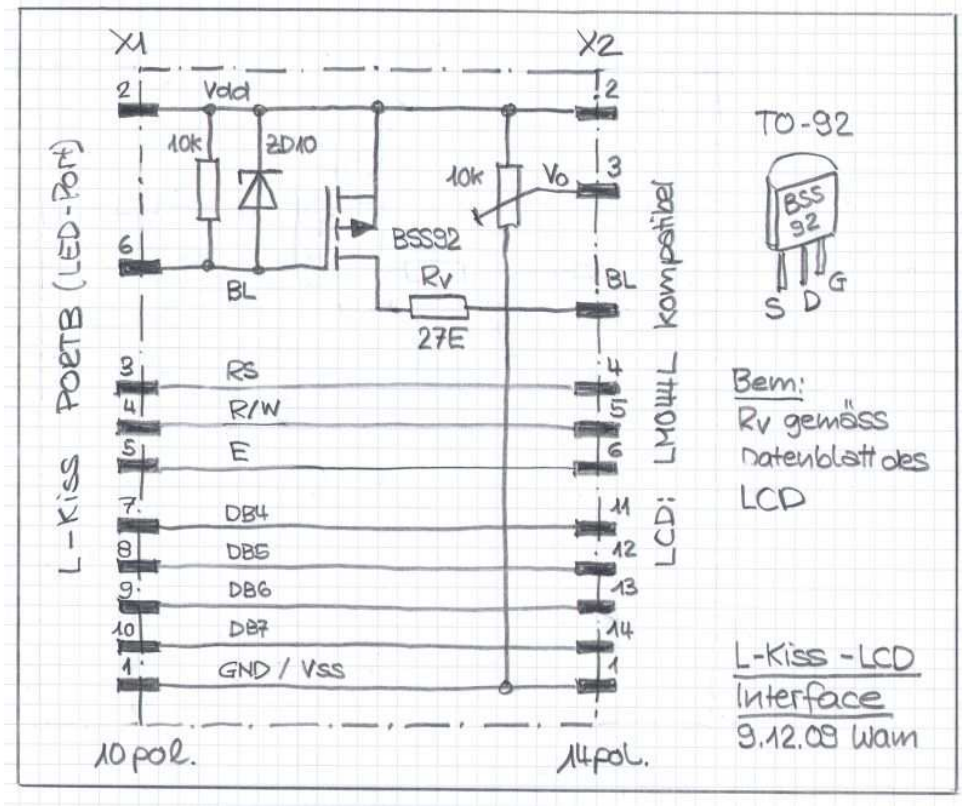


Steckerbelegung 10pol



Pin Nr	Signal
1	GND (0V)
2	Vdd (+5V)
3	Port P0
4	Port P1
5	Port P2
6	Port P3
7	Port P4
8	Port P5
9	Port P6
10	Port P7

L-Kiss Interface für LCD mit Hintergrundbeleuchtung



Hinweise zum LCD:

Das dargestellte Interface gilt für alle zum Hitachi-Standard kompatiblen LCD.

LCD's mit Hintergrundbeleuchtung haben unterschiedliche Anschlüsse für das BL Signal.

Der Strom für die LED der Hintergrundbeleuchtung variiert je nach LCD Typ.

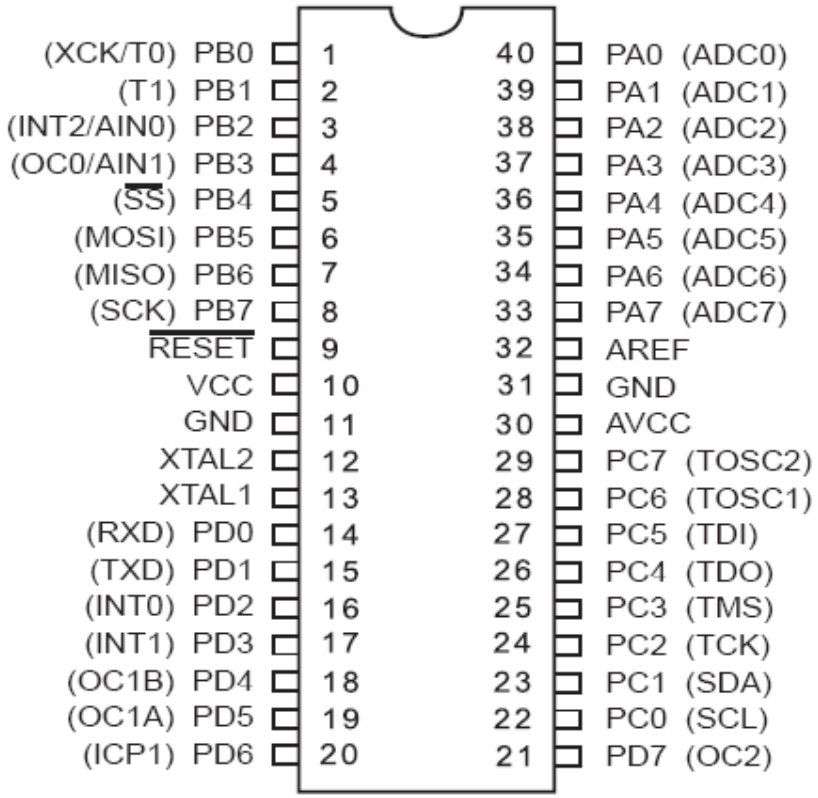
Falls ein LCD ohne Hintergrundbeleuchtung eingesetzt wird, kann der Teil mit dem MOSFET weggelassen werden.

LCD Charakter Set nach Hitachi Standard

ATMega32 Pinbelegung DIP

8-bit AVR® Microcontroller with 32K Bytes In-System Programmable Flash

ATmega32



ATMega32 I/O Ports:

Jedes der vier Ports (A, B, C, D) des ATMega32 hat neben den speziellen Funktionen (Timer, ADC, serielle Schnittstellen) an jedem Anschluss auch normale I/O Funktionen.

Die I/O-Funktonen werden für jedes dieser Ports über 3 Register gesteuert: DDRx, PORTx, PINx

DDRx Register (Port In/Out Richtung setzen):

Dieses Register gibt für jeden Anschlusspin im entsprechenden Bit an, ob der Anschlusspin ein Ein– oder ein Ausgang ist.
1 = Ausgang, 0 = Eingang

PINx Register (Zustand vom Port lesen):

Im PINx Register kann der Port-Eingangswert gelesen werden. Dazu muss jedoch im DDRx der zu lesende Pin als Eingang gesetzt sein.

PORTx Register (Zustand an Port ausgeben):

Über das PORTx Register können Zustände an einem Port gesetzt werden. Dazu muss im entsprechenden DDRx der zu schreibende Pin als Ausgang gesetzt sein.

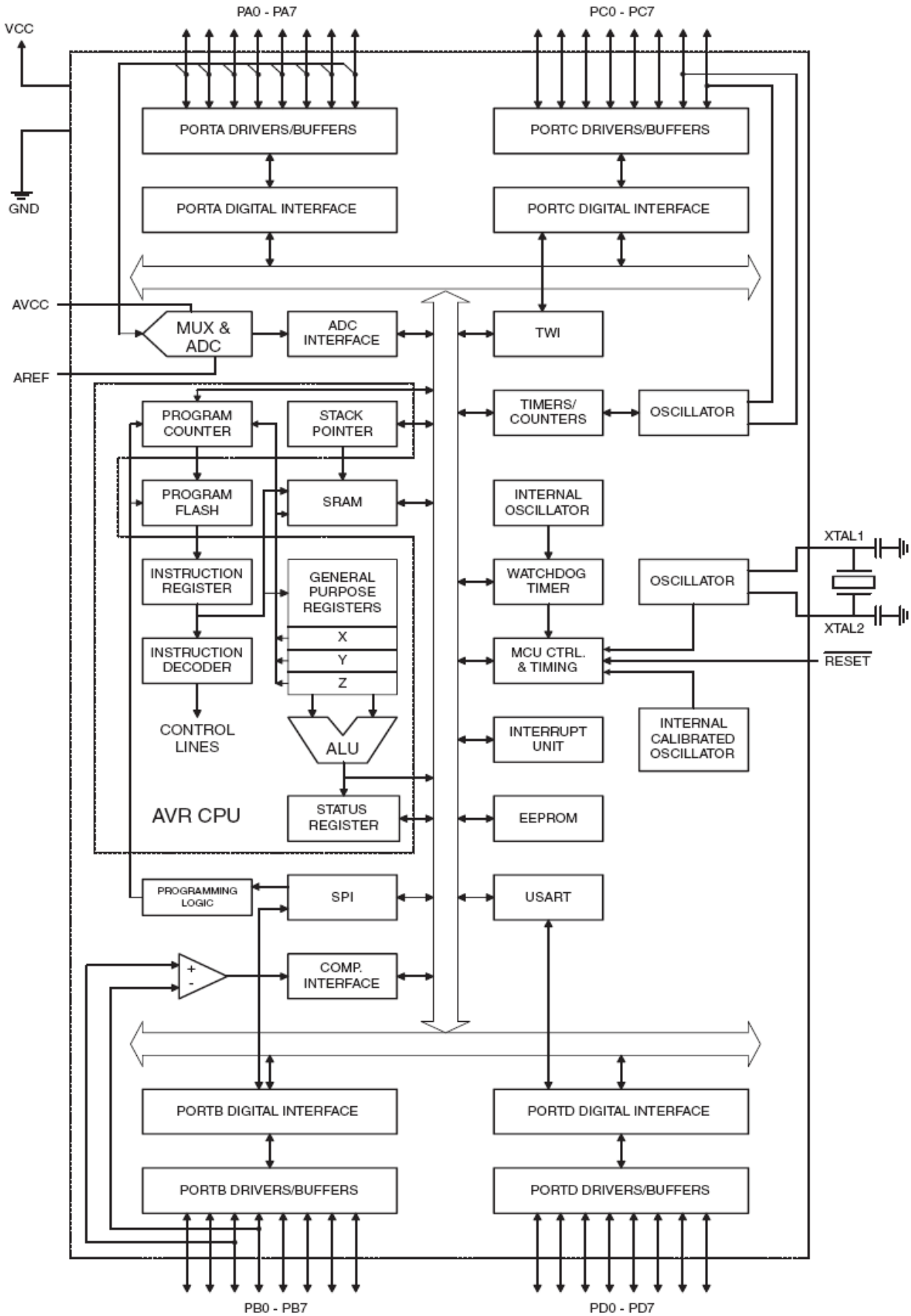
Pull-up Widerstände an den Ports

An jeden Port-Pin kann einzeln ein interner pull-up Widerstand angeschlossen werden. Dazu muss der entsprechende Pin im DDR als Eingang gesetzt sein (‚0'). Das entsprechende PORT Bit muss zudem auf den Zustand ‚1' gesetzt werden. Gelesen wird der Eingang nun über das PIN Register.

Beispiel: Alle 8 Bit des PORTC sollen mit einem pull-up Widerstand versehen werden:

```
DDRC=0x00;  //Alle Pin Eingänge
PORTC=0xFF; //Alle Pin pull-up
InVal=PINC; //Portzustand lesen
```

ATMega32 Blockschaltbild



Brian Kernighan und **Dennis Ritchie** (Erfinder von C) schreiben zu Beginn ihres Buches:

'The only way to learn a programming language is by writing programs in it.'

Literaturhinweise

Programmieren in C
Kerninghan/Ritchie
ISBN 3-446-15497-3

C Programmieren von Anfang an
Helmut Erlenkötter
ISBN 3-499-60074-9

C++ für Dummies
Namir, Clement, Shammas
ISBN 3-8266-2775-X

Mikrocomputertechnik Modell-Lehrgang
Swissmem Berufsbildung
ETMC 1K

C, C++ Lehrgänge im Internet

http://info.baeumle.com/ansic.html

http://static.sws.bfh.ch/skripte/C-Kurs.pdf

http://www.c-programmieren.com/C-Lernen.html

http://www2.its.strath.ac.uk/courses/c/

http://pronix.linuxdelta.de/C/standard_C/index.shtml

Alle Links funktionierten am 6.12.2009, Wam