Inhalt

4.1
Beschreibung einzelner Funktionen

getopt (Linux – PC)

printf - Derivate

- printf
- sprintf
- my_printf

5.0
Hardware

Pinbelegungen von Entwicklungsboards / - Plattformen (Arduino, Bluepill, Minumumboards etc.), Modulen und integrierten Schaltungen (IC), sowie Klein- und/oder Teilschaltungen.

Pinbelegungen und Plaene

- Developmentboards
  - Arduino Uno (r3)
  - Arduino Nano
  - STM8S103F3P6 Minimum Dev. Board (China)
  - STM8S103F3P6 Bootloaderboard (r3)
  - STM32F030F4P6 Dev. Board rev 0.b (r3)
  - STM32F103C8T6 Minimum Dev.-Board (BluePill)

6.0
Danke an ...

1.0

Menu-Leiste der IDE

File

Der Menupunkt File stellt folgende Funktionen zur Dateibehandlung in Menupunkten zur Verfuegung:

- New (Shortkey F4)
  Oeffnet ein neues leeres Fenster zur Codeeingabe
- New from template
  noch nicht implementiert
-Open (Shortkey F3)
  Oeffnet ein Dialogfenster zur Auswahl der Datei, die bearbeitet werden soll. In diesem Fenster kann im Eingabefeld *File to open* entweder die Datei, die geladen werden soll direkt angegeben werden, oder eine Suchmaske, deren Suchoptionen mittels eines Strichpunkts getrennt ist.

  Bsp.:

  .c;*.h;*.cpp;*.hpp;Ma

  listet alle C und C++ sowie deren Headerfiles und zusaetzlich Dateien ohne Dateinamensextension wie beispielsweise *Makefile*
- Reload
  sollte die aktuell sichtbare Datei zwischenzeitlich von einem anderen Programm modifiziert worden sein, so wird die modifizierte Datei geladen
- Save (Shortkey F2)
  speichert die aktuell sichtbare Datei.
- Save as...
  speichert die aktuell sichtbare Datei unter einem anderen Namen. Im aufgehenden Dialogfeld kann im Eingabefeld *Name* entweder der Name unter dem die Datei gespeichert werden soll, oder eine Suchmaske die Dateien im aktuellen Verzeichnis auflistet, eingegeben werden.
- Change Dir :
  wechselt das aktuelle Verzeichnis. Im aufgehenden Dialog kann das Verzeichnis aus dem Verzeichnisbaum gewaehlt werden
- Command Shell :
  ruft die Textkonsole auf. Ein „exit“ auf der Konsole kehrt zu CIDE zurueck
- Exit (Shortkey Alt - x) :
  beendet CIDE. Bei offenen noch nicht gespeicherten Dateien erfolgt eine Rueckfrage, ob diese gespeichert werden sollen

Edit

Der Menupunkt Edit stellt Blockoperationen innerhalb des Editors zur Verfuegung

- Undo
  macht die zuletzt getaetigte Operation im Editor (Blockoperation oder Tastatureingabe rueckgaengig
- ReDo :
  stellt eine mit Undo gemachte Operation wieder her
- Cut (Shortkey Strg - X)
  schneidet einen markierten Textabschnitt (Block) aus dem Editor aus und speichert diesen im Clipboard zwischen
- Copy (Shortkey Strg - C)
  kopiet einen markierten Textabschnitt (Block) aus dem Editor und speichert im Clipboard zwischen
- Paste (Shortkey Strg - V)
  fuegt den im Clipboard zwischengespeicherten Text an der aktuellen Cursorposition ein
- Select all
  markiert den gesamten Textinhalt eines Editorfensters
- Unselect

hebt die Markierung eines Textblocks auf

- Options
  erlaubt Einstellungen zum Editor, Suchmasken in Dialogfeldern und den Shortkeys zu den Blockoperationen

Search

Der Menupunkt Search stellt Funktionen fuer die Textsuche innerhalb eines Fensters zur Verfuegung:

- Find (Shortkey Strg - F)
  Oeffnet einen Dialog in dem die zu findende Textpassage eingegeben werden muss
- Replace (Shortkey Strg-Q-A)
  Oeffnet einen Dialog, in dem der zu ersetzende Text sowie der Text der ihn ersetzt eingegeben werden kann
- Go to line number
  springt an die im aufgehenden Dialog angegebene Zeilennummer des Textes. Findet der Compiler an einer bestimmten Stelle des Quelltextes einen Fehler, kann mit der Angabe der Zeilennummer an die entsprechende Stelle gesprungen werden.

Call make

Der Menupunkt *Call make* enthaelt in den Untermenupunkten die Funktionen, die benoetigt werden um ein Programm zu compilieren, linken und bei Bedarf auszufuehren. Die verschiedenen Aufrufe von Make erfordern eine im Projektverzeichnis liegende Datei mit dem Namen *Makefile*. Makefiles werden in aller Regel dem Aufruf mit angegebenen Parametern angegeben. Damit die in diesem Menupunkt erfolgenden Aufrufe von Make funktionieren, muessen innerhalb des Makefiles entsprechende Labels:

- all
- clean
- flash

vorhanden sein. Normalerweise wird ein Makefile mit der *all* so geschrieben, dass *all* alle zum Erstellen eines lauffaehigen Programms benoetigten Schritte vornimmt (Compilieren und linken). Die hier aufrufbaren Funktionen entsprechen einer Konsolen-Kommandoeingabe:

make all
make clean
make flash

Compilermeldungen werden in einem eigenen Fenster (Compiler messages) angezeigt. Sollte dieses Fenster nicht sichtbar sein, kann dieses mittels dem Menupunkt *compiler messages* (oder F12) sichtbar gemacht werden.

Da mit CIDE auch Konsolenprogramme fuer den PC geschrieben werden koennen, kann ein Makefile keine Parameterauswertung *flash* fuer PC-Programme enthalten (da ein PC nicht „geflasht“ werden kann). Um ein PC-Programm von CIDE aus starten zu koennen, ist der Menupunkt *run (PC only)* auszuwaehlen.

Unter dem Menupunkt *Programarguments* koennen einem Konsolenprogramm Programmargumente zum Programmstart mitgegeben werden.

Zusammenfassung der Menupunkte:

make all (Shortkey F9)
make flash (Shortkey F8)
make clean
Programarguments
Run (PC only) (Shortkey Alt-R)
Compiler messages (Shortkey F12)

RS232-Monitor

Folgende Menupunkte stehen zur Verfuegung:

Open Monitor:

startet das externe Terminalprogramm *picocom* Hierzu wird CIDE temporaer verlassen und *picocom* hat die alleinige Kontrolle ueber den Bildschirm. picocom kann mit der Tastenkombination STRG-A-X beendet werden (was eine Rueckkehr zu CIDE bedeutet).

Settings:

stellt die Parameter der seriellen Schnittstelle ein, mit der picocom geoeffnet wird. Einstellbare Parameter sind:

- serieller Anschluss am PC
- Baudrate
- Databits
- Stopbits
- Parity
- Return Key (CR oder CR+LF)

Windows

Der Menupunkt bietet Moeglichkeiten zum Zeigen oder Veraendern der geoeffneten Quellcodefenster sowie der Compiler-, Schnttstellen- und der Utilityfenster an.

Tile:
ordnet die Quellcodefenster uebereinander an

Cascade:
ordnet die Quellcodefenster kaskadiert an. Diese Anordnung bietet eine gute Moeglichkeit die Fenster so anzuordnen, dass sie mit einer Mouse fokusiert werden koennen

Close All:
schliesst alle Quellcodefenster

Size/Move (Tasten Strg-F5):
waehlt ein Fenster an, damit mit der Tastatur dessen Position und Groesse veraendert werden kann.
Nach der Anwahl kann mit den Cursortasten die Position veraendert werden. Wird gleichzeitig die Shifttaste mit den Cursortasten verwendet, kann hiermit die Groesse des Fensters eingestellt werden.

Zoom (Taste F5):
vergroessert/verkleinert das aktuelle Fenster

Next (Taste F6):
bringt das naechste Fenster in den Fokus

Hide (Tasten Strg-F6):
versteckt das Fenster und ist auf dem Desktop nicht mehr sichtbar. Im Menupunkt *List* kann ein verstecktes Fenster wieder sichtbar gemacht werden.

Close (Tasten Alt-F3):
schliesst ein Fenster.

List:
listet alle fuer den Desktop verfuegbaren Fenster

Refresh Display:
bei einem Fehler (sollte nicht vorkommen) wird das Fenster neu aufgebaut !

1.1

Editor

Innerhalb des Editors sind folgende Funktionen ueber Shorkeys verfuegbar:

suchen	Ctrl-Q F
naechstes suchen	Ctrl-L
suchen / ersetzen	Ctrl- Q A
naechstes ersetzen	Ctrl - L
Block bilden	Shift-Cursortasten
Block kopieren	Ctrl-C
Block ausschneiden	Ctrl-X
Block einfuegen	Ctrl-V
Block loeschen	Ctrl-Entf
Block aufheben	Ctrl-K H
Blockanfang markieren	Ctrl-K B
Blockende markieren	Ctrl-K K
Block verschieben	Ctrl-K V
Block von Datentraeger lesen	Ctrl-K R
Block auf Datentraeger schreiben	Ctrl-K W
Block ein Zeichen einruecken	Ctrl-K I
Block ein Zeichen ausruecken	Ctrl-K U
Wort markieren	Ctrl-K T
Zeile markieren	Ctrl-K L

Aktivierung einer gewuenschten Funktion am Beispiel der Textsuche innerhalb des gerade aktiven Fensters:

- Taste Ctrl (deutsche Tastatur Strg) druecken, und gedrueckt halten
- Taste Q druecken und wieder loslassen
- Taste F druecken und wieder loslassen
- Taste Ctrl loslassen

2.0

Verzeichnisstruktur fuer Projekte

Um mittels CIDE Projekte fuer verschiedene Mikrocontroller oder PC-Konsolenprogramme erstellen zu koennen, werden Projekte in dem, dem Mikrocontroller zugeordneten Verzeichnis abgelegt. Hierbei liegen alle Projekte fuer einen Mikrocontroller, einer Mikrocontrollerfamilie oder eines PC-Programmes in einem Verzeichnis:

Beispiel fuer Atmel-AVR Familie

/home/mcu/atmel_avr

Beispiel fuer einen STM32F103

/home/mcu/stm32f103

Innerhalb dieses Ordners gibt es zwei wichtige (im Falle von STM32 drei) Unterordner:

./src
./include

Im Ordner *./src* sind Quellcodedateien gespeichert, die einem Projekt bei Bedarf hinzugelinkt werden koennen. Im Ornder *./include* sind die fuer diese Quellcodedateien dazugehoerenden Header-Dateien enthalten.

Im Falle von STM32 existiert ein weiterer Unterordner:

./lib

Da dieses System die LIBOPENCM3 Bibliothek benutzt, sind hier die speziell fuer den verwendeten Controller die Softwaresourcen der LIBOPENCM3 gespeichert.

Um ein Projekt compilieren und linken zu koennen arbeitet CIDE mit Makefiles. In den Projekten selbst bedarf es nur rudimentaerer Makefiles, die ihrerseits den funktionalen Teil inkludieren. Dieser funktionale Teil liegt bei NICHT-STM32 Controllern in dem den Projekten uebergeordneten Verzeichnis und lautet:

makefile.mk

Bei STM32 Controllern heisst die Datei libopencm3.mk und liegt im Verzeichnis

./lib

2.1

Makefile

Lauffaehige Programme (binaries) werden bei Benutzung von CIDE mittels sogenannter Makefiles erstellt. Ein Makefile enthaelt Angaben darueber, wie der Compiler und Linker ein lauffaehiges Programm zu erstellen hat.

Makefiles koennen sehr komplex werden und aus diesem Grund bedient sich das vorliegende System sehr vereinfachter Makefiles, die in jedem Projektordner abgelegt sein muessen. Das Makefile in einem Projektordner enthaelt hierbei keine funktionalen Teile, es enthaelt lediglich Angaben darueber, wie die Projektdatei heisst (das ist die Datei, die die Funktion *main* beinhaltet) und welche Dateien eventuell hinzugelinkt werden muessen. Desweiteren beinhaltet sie Angaben fuer die gewuenschte Uploadmethode fuer den Controller.

Nach diesen Angaben wird der Funktionale Teil des Makefiles nachgeladen (bei STM32 ist dies die Datei libopencm3.mk im Ordner *../lib* , bei allen anderen ist dies die Datei *makefile.mk* in dem dem Projekt uebergeordneten Verzeichnis.

Diese Makefiles koennen entweder mit dem CIDE Editor erstellt und bearbeitet, oder mit dem Makefile-Generator genmakef erstellt werden.

Beispiel fuer AVR-Atmel Makefile

```
##############################################################################
#
#                                Makefile
#
#   einfaches Makefile zum "builden" von HEX-Dateien fuer Atmel (c) AVR-
#   Mikrocontroller.
#
#
#
#   Mai 2017,  R. Seelig
#
##############################################################################


PROJECT           = serial_demo

SRCS              = ../src/uart.o
SRCS             += ../src/my_printf.o

INCLUDE_PATHS     = -I. -I../include


PRINTF_FL         = 0
SCANF_FL          = 0
MATH              = 0

# fuer Compiler / Linker
FREQ              = 16000000ul
MCU               = atmega328p

# fuer AVRDUDE
PROGRAMMER        = stk500v2
SERPORT           = /dev/ttyUSB0
BRATE             = 115200
DUDEOPTS          = B1


include ../makefile.mk
```

Hier wird aus einer Quelldatei *serial_demo.c* ein lauffaehiges Binary erstellt. Zu beachten ist hierbei, dass die Dateinamensendung .c NICHT mit angegeben werden darf. Zusaetzlich werden die Dateien *uart.c* und *my_printf.c* im Ordner *../src* kompiliert und hinzugelinkt.

Aus den Standardbibliotheken wird weder die Funktionalitaet printf, scanf und math hinzugelinkt. Das zu erstellende Programm wird fuer einen ATmega328p mit einer Taktfrequenz von 16 MHz erstellt. Erfolgt bei diesem Makefile ein Flashvorgang, so wird dieser mittels eines STK500V2 Programmers vorgenommen.

Moegliche Angaben fuer Programmer sind hier:
stk500v2, arduino, usbasp, tinyusb, avrisp, wiring und ponyser

Beispiel fuer STM32 Makefile

```
##############################################################
#
#                         Makefile
#
##############################################################

PROJECT       = serial_demo

# hier alle zusaetzlichen Softwaremodule angegeben
SRCS          = ../src/sysf103_init.o
SRCS         += ../src/my_printf_float.o
SRCS         += ../src/uart.o


INC_DIR       = -I./ -I../include

LSCRIPT       = stm32f103c8.ld

# FLASHERPROG Auswahl fuer STM32:
# 0 : STLINK-V2, 1 : 1 : stm32flash_rts  2 : stm32chflash 3 : DFU_UTIL
# FLASHERPROG Auswahl fuer LPC
# 4 : flash1114_rts

PROGPORT      = /dev/ttyUSB0
CH340RESET    = 1
ERASEFLASH    = 1
FLASHERPROG   = 1


include ../lib/libopencm3.mk
```

3.0

C, eine Einfuehrung

3.1

Der Compiler

Bevor ein Programm ausgefuehrt werden kann, muss es von einem Programm - dem Compiler - in Maschinensprache uebersetzt werden. Dieser Vorgang wird als Kompilieren, oder schlicht als uebersetzen, bezeichnet.

Maschinensprache besteht aus einer Abfolge von Befehlen, die vom Prozessor direkt verarbeitet werden koennen. Jeder einzelne Befehl hiervon ist ein sogenannter Maschinencode (Mnemonic), der letztlich jedoch nur eine Zahl ist, die die Funktion des Maschinenbefehls darstellt.

Ein C-Programm wird als Quelltext bezeichnet, aus dem nach dem Kompilieren ein ausfuehrbares Programm entsteht. Dieses Programm ist das sogenannte Binary und kann, je nach Zielsystem und Compiler, unterschiedliche Dateinamenserweiterungen besitzen:

- Linux:
  keine spezielle Extension (meistens jedoch haben Linuxprogramme keine Dateinameserweiterung
- Windows:
  .com oder *.exe*
- AVR (ATmega / Attiny):
  .hex
- MCS51:
  .ihx
- ARM-Cortex:
  .bin oder *.hex*
- STM8:
  .ihx
- PIC16F:
  .hex

Neben dem Compiler werden fuer das Uebersetzen des Quelltextes folgenden Programme benoetigt:

Praeprozessor
Linker

Umgangssprachlich wird oft die Gesamtheit aller Programme, die am Erstellungsvorgang eines Binaries beteiligt sind, vereinfacht als Compiler bezeichnet.

Tatsaechlich uebernimmt oft nur ein ein einzelnes Programm die Funktion des Praeprozessors, Compilers und Linkers. So hat der GCC Compiler (in allen seinen Portierungen fuer bspw. AVR oder ARM) immer den Praeprozessor integriert und obwohl das Compilerpaket ein externes Link-Programm enthaelt, kann GCC mehrere Softwaremodule zu einem lauffaehigen Programm zusammenfuegen (linken, binden).

Praeprozessor

Vor der Uebersetzung des Quelltextes wird dieser vom sogenannten Praeprozessor verarbeitet, dessen Resultat anschliessend dem Compiler uebergeben wird.

Um hier kein Missverstaendnis aufkommen zu lassen: Der Praeprozessor veraendert nur den Text des Quellprogrammes bevor er diesen dem Compiler uebergibt, er veraendert jedoch NICHT die Quelltextdatei selbst, in dem ein Aufruf des Praeprozessors erfolgt.

Anweisungen an den Praeprozessor werden immer mit dem Hashtagzeichen # eingeleitet:

#include
#define

#define wird entweder als *Flag*, Wertezuweisung, Textersetzung oder als Makro verwendet.

Hinweis: Ein Flag signalisiert, ob etwas gegeben ist oder eben nicht.

Zur Auswertung der mit *#define als Flag* gemachten Angaben werden folgende Praeprozessoranweisungen verwendet:

- #ifndef
- #undef
- #ifdef
- #else
- #elif (funktional ein *else if*)
- #endif

<
#define mit Wertezuseisungen koennen mit folgenden Praeprozessoranweisungen ausgewertet werden:

- #if
- #else
- #elif
- #endif

Ein *Macro* ist ein kleines Stueck Programmcode, das an der Stelle im Quelltext eingefuegt wird, an der es angegeben ist. Macros koennen mit und ohne Parameter codiert sein und wird haeufig dort eingesetzt, wo die Ausfuehrungs geschwindigkeit im Vergleich zu einem Funktionsaufruf erhoeht werden soll.

Da jedoch jedes mal an der Stelle, an der das Macro eingefuegt wird, Programmcode generiert wird, kann eine hohe Nutzung von Macros zu aufgeblaehten grossen Programmen fuehren. Macros werden ebenfalls mittels *#define* definiert.

#include

include bindet eine externe Quelltextdatei an der Stelle ein, an der sie angegeben ist. Im Beispiel wird angenommen, dass es 2 Dateien, *mymath.h* und *main.c* gibt:

mymath.h

```
// ------------------------
// Demo fuer #include
// ------------------------

int addiere(int z1, int z2);
int subtrahiere(int z1, int z2);
```

main.c

```
// -------------------------
// main.c
// -------------------------

#include “mymath.h“

int main(void)
{
  int ergeb;
  ergeb= addiere(2,3);
}
```

Aus diesen beiden Quelltexten generiert der Praeprozessor folgenden Text, den er dem Compiler zum Uebersetzen gibt.

```
// -------------------------
// main.c
// -------------------------

// -------------------------
// Demo fuer #include
// -------------------------

int addiere(int z1, int z2);
int subtrahiere(int z1, int z2);

int main(void)
{
  int ergeb;
  ergeb= addiere(2,3);
}
```

Hinweis:
Die Datei *mymath.h* deklariert nur zwei Funktionen - addiere und subtrahiere -. D.h. dass irgendwo im gesamten Programm Funktionen mit den entsprechenden Namen existieren muessen und stellt diese Funktionen dem Hauptprogramm zur Verfuegung. Der Linker wird an der Stelle, an der diese Funktionen aufgerufen werden, einen call (Aufruf) an die Adresse vornehmen, an der die Funktionen definiert (programmiert) sind.

Beispiel fuer #define als Flag:

```
...
#define showhinweis               // das Flag "showhinweis" ist gesetzt

// das nachfolgende Flag ist durch "auskommentierung" nicht gesetzt
// und somit nicht aktiv

// #define copyrightmsg

int main(void)
{
  #ifdef showhinweis              // wenn das Flag gesetzt ist

    printf("\n\r Hinweistext auf irgendetwas");

  #else                           // Flag "showhinweis" ist nicht gesetzt

    printf("\n\r ... ein Hinweistext existiert nicht");

  #endif

  #ifndef copyrightmsg
    printf("\n\r dieses Programm unterliegt keinem Copyright");
  #endif
```

```
}
```

An den Compiler wird der folgende Text zum Uebersetzen gegeben (die printf Funktion nach #else wird nicht uebersetzt, ist im Programm nicht vorhanden und erzeugt somit auch keinen Code, wodurch die erzeugte Programmgroesse kleiner bleibt, als waere dieses mittels Funktionen geloest worden):

```
...
int main(void)
{

  printf("\n\r Hinweistext auf irgendetwas");

  printf("\n\r dieses Programm unterliegt keinem Copyright");

}
```

Hinweis: Zu jedem #ifdef, #ifndef oder #if gehoert ein abschliessendes #endif !

Beispiel fuer #define mit Wertezuweisung:

```
...
// nur eine der nachfolgenden defines darf den Wert 1 haben, alle anderen
// den Wert 0. Hier soll eine italienische Sprachausgabe erfolgen

#define de_language    0
#define us_language    0
#define it_language    1

// einfache Zahlenwertangabe

#define zahlenwert     42

int main(void)
{
  printf("\n\r Der Zahlenwert des #defines ist: %d", zahlenwert);

  #if (de_language == 1)
    printf("\n\r Hallo Welt \n\r");
  #elif (us_language == 1)
    printf("\n\r Hello world \n\r");
  #elif (it_language == 1)
    printf("\n\r Ciao mondo \n\r");
  #else
    printf("\n\r keine Angabe zur Sprache \n\r");
  #endif
}
```

An den Compiler wird der folgende Text zum Compilieren uebergeben:

```
int main(void)
{
  printf("\n\r Der Zahlenwert des #defines ist: %d", 42);

  printf("\n\r Ciao mondo \n\r");
}
```

Beispiel fuer Macros mit #define realisert

```
...

// Berechnung des Indexes eines 10x10 grossen Wertefeldes von
// linear im Speicher abgelegten Werten (eindimensionales Array)

#define GETINDEX(x,y)       ( (y*10)+x )

int main(void)
{
  int index;
  int x,y;

  index= GETINDEX(4, 2);
  printf("\n\r Berechnung fuer Position x= 4, y= 2");
  printf("\n\r Position= %d", index);

  x= 9; y= 1;
  index= GETINDEX(x, y);
  printf("\n\r Berechnung fuer Position x= %d, y= %d", x,y);
  printf("\n\r Position= %d", index);
}
```

An den Compiler wird der folgende Text zum Compilieren uebergeben:

```
int main(void)
{
  int index;
  int x,y;

  index= (2*10)+4;
  printf("\n\r Berechnung fuer Position x= 4, y= 2");
  printf("\n\r Position= %d", index);

  x= 9; y= 1;
  index= (y*1)+x;
  printf("\n\r Berechnung fuer Position x= %d, y= %d", x,y);
  printf("\n\r Position= %d", index);
}
```

Linker

Ein uebersetztes Quellprogramm fuer sich ist noch kein lauffaehiges Programm !!!

In aller Regel verwendet ein Programm Funktionen (Unterprogramme), die bereits geschrieben sind. Bspw. koennen in einer vom Hauptprogramm (die Datei, die die Funktion main beinhaltet) losgeloesten anderen Datei Funktionen zur Darstellung von Grafik, Text oder Toenen enthalten sein. Der Compiler uebersetzt jede einzelne Datei in Maschinensprache und der Linker *bindet* nun aus allen am Programm beteiligten angegebenen Dateien ein lauffaehiges Programm.

Werden, wie zum Beispiel bei *libc* mehrere bereits uebersetzte Quellprogramme (sogenannte Objektdateien) zu einer Datei zusammengefuegt, so wird diese als Bibliothek (oder englisch: Library) bezeichnet.

Soll der Linker zusaetzlich zu den uebersetzten Programmteilen Bibliotheksfunktione mit einbinden, wird die Library nach den benoetigten Funktionen durchsucht und zum generierenden Hauptprogramm hinzugefuegt.

Der Linker bindet also eine oder mehrere uebersetzte Quellprogramme zu einem lauffaehigen Programm zusammen.

Standardmaessig gehoeren zu einem Ansi C-Compiler die Bibliotheken libc und libm die, wie oben beschrieben, wiederum aus mehreren bereits uebersetzten Programmteilen bestehen.

In den beiden Bibliotheken libc und libm sind u.a. Funktionsdefinitionen fuer folgende Headerdateien (Auszug) enthalten:

- string.h
- math.h
- stdint.h
- stdio.h
- -stdlib.h
- stdarg.h
- time.h
- inttypes.h

Es empfiehlt sich daher, diese Biblootheken mitzulinken, der Linker wird in aller Regel die nicht benoetigten Programmfunktionen in das zu generierende Gesamtprogramm NICHT integrieren.

Anmerkung: Leider ist die libc des SDCC (small-device-c-compiler) in der Portierung fuer einen PIC16F Microcontroller nur sehr unvollstaendig und einige Teile muessen, wenn mit diesem Chip gearbeitet werden soll, in Verbindung mit dem SDCC, nachprogrammiert werden.

Kommentare

Bei Programmen empfiehlt es sich, vor allem wenn sie eine gewisse Groesse erreichen, diese zu kommentieren. Selbst wenn Sie das Programm uebersichtlich gliedern, wird es fuer eine zweite Person schwierig werden, zu verstehen, welche Logik hinter Ihrem Programm steckt. Vor dem gleichen Problem stehen Sie aber auch, wenn Sie sich nach ein paar Wochen oder gar Monaten in Ihr eigenes Programm wieder einarbeiten muessen.

Fast alle Programmiersprachen besitzen deshalb eine Moeglichkeit, Kommentare in den Programmtext einzufuegen. Diese Kommentare bleiben vom Compiler unberuecksichtigt. In C werden Kommentare in /* und */ eingeschlossen. Ein Kommentar darf sich ueber mehrere Zeilen erstrecken.

Ausserdem ist es moeglich, einzeilige Kommentare, beginnend mit // (2 Slashzeichen), zu verfassen. Ein solcher Kommentar muss nicht unbedingt am Anfang einer Zeile stehen, sondern kann bspw. am Ende einer Befehlszeile als Kommentar zu derselbigen stehen. Mit dem Ende einer Zeile ist ein solchiger Kommentar abgeschlossen.

Ein Beispiel fuer Kommentare:

```
/* Dieser Kommentar erstreckt sich
   ueber mehrere Zeilen */

#include <stdio.h>    // Dieser Kommentar endet am Zeilenende

int main()
{
  printf("Beispiel fuer Kommentare\n");

  // printf("Diese Zeile wird niemals ausgegeben\n");

  return 0;
}
```

Anmerkungen zu Beispielen

In diesem Text werden zum Verdeutlichen von Funktionalitaeten kurze Programme oder Programmausschnitte gezeigt.

Damit diese nachvollzogen werden koennen, benoetigt es zumindest eines minimalistischen Ein-Ausgabesystems. Auf einem PC ist das relativ leicht zu bewerkstelligen, da hier das Ein-Ausgabesystem die Konsole ist (oder sein kann). Hierfuer gibt es Funktionen printf und scanf. Diese stehen zwar grundsaetzlich Mikrocontrolleranwendungen auch zu Verfuegung, ist jedoch mit Schwierigkeiten verbunden.

Zum einen gibt es keine Klartextausgabeeinheit, diese muss erst implementiert werden (und die Hardware hierfuer kann sehr unterschiedlich sein), zum Anderen ist die printf - Funktionalitaet recht umfangreich, so dass bei einem Mikrocontroller mit bspw. 2 KByte Flash (wie bei einem ATtiny2313 oder ATtiny24) der Speicher alleine fuer ein printf nicht ausreicht.

Um nun Beispiele dennoch zeigen und nachvollziehen zu koennen, wurde ein sehr minimalistisches Ein-Ausgabesystem geschaffen, das selbst noch auf einem 2 KByte Mikrocontroller funktionsfaehig ist.

Hierfuer muss einem Beispielsprojekt smallio.o hinzugelinkt werden (siehe auch Makefile).

smallio bedient sich zur Kommunikation fuer Ein- und Ausgabe der seriellen Schnittstelle, so dass es hier eines Schnittstellenwandlers USB zu UART (mit den entsprechenden Logikpegeln) bedarf. Im Falle einer Arduino Hardware ist dieser Schnittstellenwandler bereits gegeben. Im Falle eines PC-Programmes wird die Konsole verwendet.

Die Parameter den UART sind hierbei:

> 4800 Bd, 8 Datenbits, 1 Startbit, 1 Stopbit, keine Paritaet

smallio stellt als rudimentaere Ein- Ausgabeeinheit folgende (Prototypen) Funktionen zur Verfuegung:

```
void    smallio_init(void);
int     putchar(int ch);
uint8_t keypressed( void );
uint8_t getchar( void );
int16_t readint();

void    my_printf(const uint8_t *s,...);

#define printf(str,...)  printf
```

Aus Kompatiblitaetsgruenden zu Mikrocontrolleranwendungen wurde auch bei PC-Projekten in den Ordnern *.src* und *.include* ein smallio erstellt. Somit sind die Beispiele auch in einer PC-Konsole lauffaehig.

Hallo Welt

Das "beruehmte" Hallo-Welt-Programm:

```
/* ----------------- 000example.c ----------------
                     Hallo Welt

            Beispiel fuer die Hilfedatei
   ------------------------------------------------ */

#include "smallio.h"

int main(void)
{
  smallio_init();                 // damit I/O Anweisungen funktioneren

  printf("Hallo Welt\n\r");
  while (1)
  {
  }
}
```

Die ersten Zeilen sind Kommentar, grundsaetzlich sollte am Anfang eines Programmes ein Kommentar stehen damit leicht erkennbar ist, fuer WAS dieses Programm gut ist.

```
#include smallio.h
```

bindet zusaetzlich die Funktionalitaet einer seriellen Schnittstelle (oder die Ausgabe auf der PC-Konsole) ein.

Fuer die Funktionalitaet von smallio siehe auch "Anmerkungen zu Beispielen“.

```
int main(void)
```

Bezeichnet das "Hauptprogramm" des Programms. Jedes Programm muss eine solche Zeile besitzen, ab hier wird beim Start (bei einer MCU nach anlegen der Betriebsspannung) das Programm ausgefuehrt.

Geschweifte Klammern:

Eine oeffnende geschweifte Klammer signalisiert den Beginn eines Blockes, eine schliessende das Ende.

```
smallio_init();
```

smallio_init() initialisiert die serielle Schnittstelle der MCU zur Benutzung. Sie stellt automatisch eine Baudrate von 4800 Bd, 8 Datenbits,1 Start und 1 Stopbit sowie keine Paritaet ein.

```
printf("Hallo Welt\n\r");
```

sendet auf der seriellen Schnittstelle den Text "Hallo Welt" und anschliessen ein Carriage Return (das ist das Kommando, das den Cursor an den Anfang einer Zeile setzt) sowie ein Line Feed (das ist das Kommando, dass eine neue Zeile beginnt)

```
while(1)
{
}
```

solange die Bedingung in der geschweiften Klammer groesser als 0 ist (und sie ist hier mit konstant 1 angegeben) wird der nachfolgende Block innerhalb der geschweiften Klammern ausgefuehrt. In diesem Programm ist das also eine sogenannte Endlosschleife und das Programm wird ab dieser Stelle nichts mehr weiteres machen.

Die abschliessende schliessende Klammer zeigt das Ende des Quellprogrammes an.

Grundsaetzlicher Programmaufbau

In C besteht ein Programm aus mehreren, mindestens jedoch einer Funktion. Wie das „Hallo Welt“ Programm zeigt, kommt einer Funktion eine besondere Rolle zu:

main

Diese Funktion wird bei einem Programmstart automatisch ausgefuehrt, alle anderen werden im Programmablauf (von Interrupts sei hier fuer das erste abgesehen) aufgerufen.

Ein grundsaetzliches abstraktes Programmgeruest sieht in etwa folgendermassen aus:

- Hinzufuegende Programmfunktionen / Bibliotheken
- Funktionen
- Hauptfunktion main

Geschweifte Klammern:

Funktional zusammengehoerende Programmteile werden in geschweiften Klammern eingeschlossen.

Ein Beispiel in einer Pseudoanweisung:

```
- Anweisung vor "wenn"
Wenn (Wasserbehaelter == leer) dann
{
   - Wasserbehaelter entfernen
   - unter Wasserhahn halten
   - Wasser aufdrehen
   solange
   {
     - Wasser laufen lassen
    } Behaelter nicht voll

   - Wasser abdrehen
   - Wasserbehaelter wieder einsetzen
}
- Anweisung nach “wenn“
```

Das Beispiel zeigt, wie die Klammern anzuwenden sind. Ist der Wasserbehaelter nicht leer, wird sofort mit der *Anweisung nach "wenn"* fortgefahren. Die Anweisungen in den Klammern gehoeren also nur zur *"wenn" Anweisung*.

Innerhalb dieser "wenn" Bedingung gibt es einen weiteren, von geschweiften Klammern eingeschlossenen Bereich. Alle Anweisungen hierin (in unserem Fall nur eine einzige) werden ausgefuehrt, solange der Behaelter nicht voll ist.

Sprachelemente

Datentypen

C definiert fuer Variable grundsaetzliche Datentypen. Die Bezeichnungen fuer die Ganzzahlentypen ist nicht ganz unproblematisch denn die Speichergroesse (und damit der Wertebereich) kann auf unterschiedlichen Prozessorsystemen auch unterschiedlich ausfallen.

Der Datentyp ' int ' hat auf 8 und 16 Bit Prozessoren in aller Regel eine Speichergroesse von 2 Byte (16 Bit), jedoch auf 32 Bit Systemen eine Speichergroesse von 4 Byte (und hiermit auch einen anderen Wertebereich). C-Compiler fuer 64 Bit CPU's verwenden fuer einen Integer sogar 8 Byte.

Eine klare Zuordnung, welcher Speicherbereich und welcher Wertebereich fuer eine Variable gegeben ist, kann mit der Verwendung der Datentypen aus stdint.h. erfolgen Hier gilt fuer alle C Compiler verbindlich, dass ein uint16_t (in AVR-GCC ein 'unsigned int') 2 Bytes Speicher belegt (und hierbei einen entsprechenden Wertebereich besitzt).

Die nachfolgende Tabelle ist gueltig fuer 8 Bit AVR MCU's

Grundtypen von C Variable			
Type	Groesse	Wertebereich	Type in stdint.h
char	1 Byte	-128 … 127	int8_t
unsigned char	1 Byte	0 … 255	uint8_t
int	2 Byte	-32768 … 32767	int16_t
unsigned int	2 Byte	0 … 65535	uint16_t
long	4 Byte	-231 ... 231-1	int32_t
unsigned long	4 Byte	0 … 232-1	uint32_t
long long	8 Byte	-263 … 263-1	int64_t
unsigned long long	8 Byte	0 … 264-1	uint64_t
Type fuer Gleitkommazahlen			
Float	4 Byte	1,5E-45 … 3,4E38	

Anmerkung: AVR-GCC kennt auch die Gleitkommazahlentypen fuer:

- double
- long double

Gleitkommazahlen sind jedoch fuer den AVR-GCC wie folgt definiert:

```
#define FLOAT_TYPE_SIZE 32
#define DOUBLE_TYPE_SIZE 32
#define LONG_DOUBLE_TYPE_SIZE 32
```

Variable

Programme muessen die, waehrend eines Programmlaufesn verwendeten Daten hauefig zwischenspeichern um mit ihnen spaeter weiter rechnen zu koennen.

Eine Variable dient grundsaetzlich dazu, eine Information zu speichern. Hierfuer wird eine Stelle oder ein Speicherbereich des Computers benutzt um diese Information speichern und wieder abrufen zu koennen.

Grundsaetzlich sind saemtliche Speicherstellen eines Speichers nummeriert, d.h. jede Speicherstelle wird durch eine Nummer identifiziert. Fuer den Programmierer waere es sehr muehsam, innerhalb eines Programms mit der Nummer des Speicherorts der die Information traegt, zu arbeiten.

Angenommen, es wuerde an Speicherstelle 2087 (das waere die Adresse im Speicher) die Laenge eines Rechtecks gespeichert werden, so muesste er, um an die Information zu gelangen, innerhalb eines Programms sagen:

"Programm, lese mir den Wert an der Speicherstelle 2087 aus, damit ich die Kantenlaenge des Rechtecks weiss"

Wuerde dieses Vorgehen mit wenigen zu speichernden Informationen noch handhabbar sein, waere es jedoch spaetestens bei Verwendung von vielleicht 20, 100 oder gar 1000 Informationen gaenzlich unpraktikabel.

In C (und in jeder anderen Programmiersprache auch) koennen zu diesem Zweck den Speicherstellen Namen zugewiesen werden. Hierbei wird dem C Compiler (in aller Regel) ueberlassen,an welcher Speicherstelle er diese Variable ablegt.

Damit ein Programm mit Variablen arbeiten kann, muss im Programm festgelegt werden, dass es die zu verwendende Variable geben soll.

Da es jedoch Daten unterschiedlichster Art gibt, muss zudem angegeben werden, um welche Art (Datentyp) es sich bei der Variable handelt.

Eine Variable, die die Kantenlaenge eines Rechtecks aufnehmen soll wuerde bspw. Mit

```
int kantenlaenge;
```

oder vielleicht auch mit

```
int laenge;
```

definiert werden. Fortan kann innerhalb des Programms dieser Variable ein Wert des Datentyp's Integer zugewiesen werden (ein Integer kann ganze Zahlen, keine Kommazahlen aufnehmen).

```
/* ----------------- 001example.c ----------------

           Beispiel fuer die Hilfedatei

   ------------------------------------------------ */

#include "smallio.h"

int   laenge, breite, hoehe, flaeche;
char  ch;

int main(void)
{

  smallio_init();

  while (1)
  {
    printf("\n\n\rDemo fuer Variable");
    printf("\n\rQuaderberechnung\n\r");
```

```
    printf("\n\rLaenge des Quaders: ");
    laenge= readint();
    printf("\n\rBreite des Quaders: ");
    breite= readint();
    printf("\n\rHoehe des Quaders : ");
    hoehe= readint();
    flaeche= (2 * (laenge*breite + laenge*hoehe + breite*hoehe));
    printf("\n\rQuaderoberflaeche: %d", flaeche);
    ch= getchar();
    printf("\n\r");
  }
}
```

Im Beispielsprogramm "001example.c" werden insgesamt 4 Variable deklariert: *laenge, breite* und *hoehe* vom Typ Integer sowie eine Variable des Datentyps char mit dem Namen *ch*. Da diese Variablen nicht explizit dem Hauptprogramm *main* zugeteilt sind (und auch keiner anderen Funktion) sind diese sogenannte globale Variable.

Global / Local

Variable besitzen eine sogenannte "Gueltigkeit". Im vorangegangenen Beispiel *001example.c* koennen die Variable innerhalb des gesamten Programms verwendet werden und auf sie zugegriffen werden. Sie haben somit eine *globale* Gueltigkeit.

In vielen Faellen ist dieses jedoch **NICHT** wuenschenswert. Verwendet eine Funktion bspw. denselben Namen fuer eine Variable wie das Hauptprogramm, so ist nach einem Funktionsaufruf eventuell ein weiterhin benoetigter Wert verloren. Variable, die nur innerhalb einer Funktion benoetigt werden sollten deshalb nur innerhalb ihrer eigenen Funktion Gueltigkeit haben. Hiermit ist es moeglich, in einer Funktion A denselben Variablennamen wie in Funktion B oder im Hauptprogramm (*main*) zu verwenden ohne dass der Inhalt einer anderen lokalen Variable ueberschrieben wird.

Haeufig findet dies Anwendung fuer bspw. Zaehlschleifen:

```
/* ----------------- 002example.c ----------------

     Beispiel fuer die Hilfedatei  lokale und
     globale Variable
   ---------------------------------------------- */

#include "smallio.h"

void myfunc(void)
// Funktion mit lokaler Variable i
{
  int i;                      // lokale Variabel, nur
                              // gueltig innerhalb von myfunc

  printf("Zaehler innerhalb von myfunc\n\r");
  for (i= 0; i< 10; i++)
  {
    printf("%d ",i);
  }
  printf("\n\r\n\r");
}

int main(void)
{
  int i;                       // Variable i ist nur im
                               // Hauptprogramm main gueltig

  smallio_init();              // damit I/O Anweisungen funktioneren

  printf("\n\r");
  for (i= 0; i< 3; i++)
  {
    printf("%d-ter Aufruf von myfunc\n\r",i+1);
    myfunc();
```

```
  }
  while (1);
}
```

static

Eine statische Variable (static) ist grundsaetzlich auch eine lokale Variable, die nur innerhalb einer Funktion verwendet werden kann.

Im Unterschied zu einer rein lokalen Variable wird der Speicherplatz, den eine statische Variable belegt nach Beenden der Funktion in der sie benutzt wird NICHT freigegeben. D.h. dass die Variable nach Beenden der Funktion ihren Werteinhalt beibehaelt und bei einem spaeteren erneuten Aufruf der Funktion dieser Inhalt verfuegbar ist.

```
/* ----------------- 003example.c ----------------
     Beispiel fuer die Hilfedatei
     statische Variable
   ------------------------------------------- */

#include "smallio.h"

void myfunc(void)
{
  // Funktion mit statisch lokaler Variable i

  static int i = 1;           // statisch lokale Variabel, nur
                              // gueltig innerhalb von myfunc

  printf("%d-ter Aufruf von myfunc\n\r", i);
  i++;
}

int main(void)
{
  char ch;

  smallio_init();              // damit I/O Anweisungen funktioneren

  printf("Demo -statische Variable-\n\r");
  printf("Taste zum Zaehlen druecken\n\r");
  while (1)
  {
    ch= getchar();
    myfunc();
  }
}
```

volatile

Wird 'volatile' bei der Definition einer Variablen benutzt, nimmt der Compiler an diesen keinerlei Optimierungen vor. Als *volatile* definierte Variablen werden darum nie in Prozesorregister geladen.

volatile Variablen werden eingesetzt, wenn die Variablen zu einem Zeitpunkt veraendert werden koennen, die ausserhalb der Kontrolle des Programmablauf liegen liegen. Typischerweise ist dies bei Variablen der Fall, die in einer Interruptroutine (ISR, Interrupt Service Routine) verwendet werden.

```
volatile static char myvariable;
```

definiert eine Variable namentlich *myvariable*, die:

- nicht 'wegoptimiert' werden kann (d.h. der Compiler stellt die Variable zur Verfuegung auch wenn dieser 'glaubt' dass diese Variable nicht benutzt wird ('volatile').

- die im Speicher immer denselben Platz einnimmt (*static*) und dieser Speicherplatz somit anderen NICHT zur Verfuegung steht.

struct

Fasst Variablen zusammen

```
struct [<Strukturtyp-Name>]
  { [ Typ Feldname ] ;
   ...
  } [ struct-Variable ] ;
```

Eine struct-Definition fasst mehrere Felder unterschiedlichen Typs unter einem gemeinsamen Typbezeichner zusammen und kann (bei Angabe eines Strukturtyp-Namens) als Typdefinition oder als aktuelle Variablendeklaration erfolgen. Ein einmal definierter Strukturtyp laesst sich (wie char, int usw.) fuer beliebige Variablendeklarationen verwenden.

Die Definition eines Feldes besteht aus einem Typ-Bezeichner, gefolgt von einem oder mehreren Feldnamen, die durch Komms voneinander getrennt sind. Felder verschiedener Typen werden durch Semikolons voneinander getrennt:

```
struct my_struct                    // Strukturtyp
{
  char  name[80], tel_nummer[20];
  int   alter, groesse;
} freund, kollege;                  // <- Variable
```

Hier werden sowohl ein Strukturtyp (*my_struct*) als auch zwei Variablen dieses Typs (*freund* und *kollege*) vereinbart. Jede dieser beiden Variablen enthaelt vier Felder.

Der Zugriff auf ein Feld einer struct-Variablen erfolgt ueber <struct-Name>.<Feldname>:

```
strcpy(freund.name,"Max Mustermann");
freund.alter= 39;
```

Nach der Vereinbarung eines Strukturtyp-Namens koennen mit diesem Namen weitere Variablen desselben Typs deklariert werden, hier ein Array mit 100 Elementen der Struktur:

```
struct my_struct meine_freunde[100];
```

Der Zugriff auf ein Element des Arrays geschieht folgendermassen:

```
strcpy(meine_freunde[32].name,"Michaela Musterfrau");
meine_freunde[32].alter= 34;
```

Da der Speicherbedarf einer Struktur (vor allem bei Verwendung eines Array welches als Arrayelement die Struktur beinhaltet) sehr gross werden kann, empfiehlt es sich, Bearbeitungen von Strukturen mittels "call by reference" (Zeigerprogrammierung) vorzunehmen.

Der Zugriff auf ein Element einer Struktur, die mittels einer Adresse referenziert ist funktioniert folgenderweise:

```
alter= (*freund).alter ;
```

Eine bessere Moeglichkeit auf das Mitglied (engl. member) <alter> der zeigerreferenzierten Struktur <freund> zuzugreifen ist: ->

```
alter= freund -> alter;
```

Hinweis: Das folgende Beispiel hat einen erhoehten RAM-Speicherbedarf und funktioniert erst mit Systemen, die mindestens 2 kByte RAM aufweisen!

Beispiel:

```
/* ------------------------------------------------------
     example_struct.c

     Beispiel zur Verwendung von struct
   ------------------------------------------------------ */
#include <string.h>
#include "smallio.h"

#define name_len     30

struct my_struct                      // Strukturtyp
{
  char  name[name_len], tel_nummer[20];
  int   alter, groesse;
};


/* ------------------------------------------------------
     zeigt die Mitglieder einer einzelnen Struktur an
   ------------------------------------------------------ */
void show_struct_val(struct my_struct *freunde)
{
  char my_name[name_len];
  char my_tel[20];
  int  alter, groesse;

  alter= freunde -> alter;                    // Zugriff auf Mitglied alter
  groesse= freunde -> groesse;                // dto. groesse
  strcpy(my_name, freunde -> name);           // dto. name
  strcpy(my_tel, freunde -> tel_nummer);      // dto. tel_nummer

  // Anzeige der Mitglieder
  printf ("\n\r Name   : %s", my_name);
  printf ("\n\r Tel.   : %s", my_tel);
  printf ("\n\r Alter  : %d", alter);
  printf ("\n\r Groesse: %d cm \n\r", groesse);
}

/* ------------------------------------------------------
     zeigt die Mitglieder eines Arrayelementes ueber einen
     Index an
   ------------------------------------------------------ */
void show_struct_indexval(struct my_struct *freunde, int index)
{
  char my_name[name_len];
  char my_tel[20];
  int  alter, groesse;

   freunde += index;                              // Zeiger um die Elementposition der
Struktur um
                                              // deren Position erhoehen

  // Mitglieder  der Struktur auslesen
  alter= freunde -> alter;                    // Zugriff auf Mitglied alter
  groesse= freunde -> groesse;                // dto. groesse
  strcpy(my_name, freunde -> name);           // dto. name
  strcpy(my_tel, freunde -> tel_nummer);      // dto. tel_nummer

  // Anzeige der Mitglieder
  printf ("\n\r Name   : %s", my_name);
  printf ("\n\r Tel.   : %s", my_tel);
  printf ("\n\r Alter  : %d", alter);
  printf ("\n\r Groesse: %d cm \n\r", groesse);
}
```

```
/* ------------------------------------------------------------------------
                                  M A I N
   ------------------------------------------------------------------------ */
int main(void)
{
  struct my_struct meine_freunde[10];

  int index;

  smallio_init();

  index= 5;
  strcpy(meine_freunde[index].name,      "Max Mustermann");
  strcpy(meine_freunde[index].tel_nummer,"0555/323232");
  meine_freunde[index].alter=             39;
  meine_freunde[index].groesse=           181;

  index= 7;
  strcpy(meine_freunde[index].name,      "Michaela Musterfrau");
  strcpy(meine_freunde[index].tel_nummer,"0555/454545");
  meine_freunde[index].alter=             34;
  meine_freunde[index].groesse=           163;

  show_struct_val(&meine_freunde[5]);
  show_struct_indexval(&meine_freunde[0], 7);
}
```

Ausgabe:

```
      Name   : Max Mustermann
      Tel.   : 0555/323232
      Alter  : 39
      Groesse: 181 cm

      Name   : Michaela Musterfrau
      Tel.   : 0555/454545
      Alter  : 34
      Groesse: 163 cm
```

Zeiger (Pointer)

Zeiger sind Variable, die anstelle des direkten Dateninhalts die Adresse (reference), an der diese Daten gespeichert sind, beinhalten.

Ein Zeiger ist gewissermassen der Verweis oder der Link auf eine Variable, an der diese im Speicher zu finden ist.

Ein Zeiger kann auf Variable unterschiedlicher Datentypen wie char, int, float aber auch Strukturen verweisen.

```
Datentyp   *variable;
```

Das ***** Zeichen zeigt an, dass es sich bei der Variable um einen Zeiger handelt. Zu beachten ist hierbei, dass das ***** Zeichen VOR dem Variablennamen steht. Ein Zeiger muss immer vom selben Datentyp sein, auf den er zeigt.

Zeiger kommen vor allem immer dann zum Einsatz, wenn Variable die mehr als aus einem Wert (Value) bestehen, bearbeitet werden sollen.

Beispiel:

Ein Char-Array fuer einen Text wird definiert:

```
char  mytext[4]= "Ich";
```

Hier wird der Compiler fuer diesen Text insgesamt 4 Bytes des Speichers belegen. Es wird angenommen, dass der Compiler hierfuer die Adresse 0x0210 gewaehlt hat. Die Adresse wurde fuer dieses Beispiel ist willkuerlich gewaehlt, welche Adresse tatsaechlich benutzt wird, bestimmt der Compiler eigenstaendig).

Im Arbeitsspeicher sieht dieses folgendermassen aus:

Adresse	**Inhalt**
0x210	0x49 (‘I‘)
0x211	0x63 (‘c‘)
0x212	0x68 (‘h‘)
0x213	0x00 (Endekennung eines Strings)

Moechte man nun auf die einzelnen Elemente des Arrays zugreifen um diese zu bearbeiten oder auszulesen hat man nun 2 Moeglichkeiten:

Man kann eine "Indexvariable" erstellen die dann zum einen im Stile von

```
char  mytext[4]= "Ich";
int   strindex;
char  ch;

strindex= 0;
ch= mytext[strindex];

// ch beinhaltet nun das 'I' Zeichen
```

auf das Array zugreift. Wird nun aber eine Funktion benoetigt, die die Bearbeitung des Arrays vornimmt funktioniert dieses Vorgehen nur noch bedingt.
Hier kommen die Zeiger ins Spiel und dies ist somit die zweite Moeglichkeit:

```
char  mytext[4]= "Ich";
char  *p;              // ein Zeiger auf Datentyp char
char  ch;

p= &mytext[0];

ch= *p;

// ch beinhaltet nun das 'I' Zeichen
```

Erklaerung:

Das ***** Zeichen vor der Variable *p* ist ein Operator und bedeutet, dass dies keine "normale" Variable ist, sondern dass sie als Inhalt eine Speicheradresse aufnehmen kann, die auf eine Variable vom Typ char zeigt.

Somit ist p eine Zeigervariable (korrekt bezeichnet als Indirektionsoperator).

```
ch = *p;
```

Das Sternchen VOR dem p gibt an, dass ch den Wert uebergeben werden soll, auf den p zeigt.

Das **&** Zeichen in der Anweisung

```
p = &mytext[0];
```

gibt an, dass dem Zeiger p die Adresse des Arrays "mytext[0]" zugewiesen wird. Somit hat die Zeigervariable p nach obigem Beispiel die Adresse 0x210 und zeigt somit auf den Anfang des Arrays. Da dem Zeiger eine Adresse uebergeben wird (und nicht der Inhalt), erhaelt die Variable p KEIN Sternchen davor.

Mit `ch = *p;` wird angewiesen, dass der Variable *ch* (vom Typ char) der Wert zugewiesen werden soll, auf den p zeigt (in unserem Fall auf den Buchstaben ‘I‘ der im Speicher abgelegt ist).

Beispielprogramm mit Zeiger

Das folgende Beispielprogramm zeigt, wie ein Array, dessen Inhalt einen Text ist, auf der seriellen Schnittstelle ausgegeben werden kann:

```
#include "smallio.h"

int main(void)
{

  char   mytext[] = "Mein Name ist Hase\n\r";

  char   *p;                  // unser Zeiger
  char   *p2;                 // ein zweiter Zeiger
  char   ch;

  smallio_init();             // fuer Ausgaben via printf

  p= &mytext[0];              // p beinhaltet die Adresse des Textes
  p2= p;                      // ... und p2 auch !

  while(*p)                   // wiederhole so lange, wie das Element,
  {                           // auf das p zeigt groesser, als 0 ist
                              // (also keine Endekennung ist).

    ch= *p;                   // ch beinhaltet das Zeichen auf
                              // das p zeigt

    p++;                      // Zeiger auf naechstes Element
                              // setzen

    putchar(ch);              // Zeichen ausgeben
  }

  p= p2;                      // da der originale Zeiger in der
                              // While-Schleife veraendert wurde
                              // zeigt er hiermit wieder auf den
                              // originalen Speicherbereich
  while (1);
}
```

Programmablaufkontrolle

Damit Programme in einer Programmiersprache erstellt werden koennen, ist es notwendig, dass diese Sprache sogenannte Kontrollstrukturen besitzt, die in der Lage sind, bestimmte Anweisungsbloecke nur unter bestimmten Bedingungen auszufuehren oder diese bei Bedarf mehrfach zu wiederholen.

Die haeufigsten in C verwendeten Kontrollstrukturen sind das if-Statement zur bedingten Ausfuehrung und die for-Schleife zur Wiederholung von gebuendelten Anweisungen.

Daneben existieren noch die *while*- und *do{}while* Schleife sowie *das switch() case:*- Statement, das aufgrund des Inhalts einer Variablen einen auszufuehrenden Block anspringt.

If - else

Bedingte Verzweigung

```
   if ( ' ausdruck ' )
        ' anweisung1 '

         ODER

   if ( ' ausdruck ' )
        ' anweisung1 '
   else
        ' anweisung2 '
```

Wenn der Wert von *ausdruck* `!= 0` (ungleich 0 entspricht 1 oder groesser) ergibt, wird `anweisung1` ausgefuehrt, ansonsten wird diese Anweisung (Befehl) uebersprungen.

if..else:
Wenn *ausdruck != 0* ergibt, wird *anweisung1* ausgefuehrt und *anweisung2* uebersprungen.
Wenn *ausdruck == 0* ergibt, wird *anweisung1* uebersprungen und *anweisung2* ausgefuehrt.

Das else ist optional, aber zwischen einer if-Anweisung und einem else duerfen keine weiteren Anweisungen stehen.

Beispiel:

```
/* ----------------- 004example.c ----------------
     Beispiel fuer die Hilfedatei
     if - else Demo
   ---------------------------------------------- */

#include "smallio.h"

int main(void)
{
  char ch;

  smallio_init();

  printf("\n\rif - else Demo\n\r--------------\n\n\r");
  while(1)
  {
    printf("\n\rgeben sie eine Zahl ein: ");
    ch= readint();

    if (ch > 50)
    {
      printf("\n\rEingabe war groesser 50...\n\r");
    }
    else
    {
      printf("\n\rEingabe war kleiner-gleich 50...\n\r");
```

```
    }

    printf("\n\rgeben sie eine Zahl ein: ");
    ch= readint();
    if (ch)
      { printf("\n\rEingabe war NICHT 0!\n\r"); }
    else
       { printf("\n\rEingabe war 0 !\n\r"); }
  }
}
```

while

Wiederholt Ausfuehrung

```
      while ( 'Ausdruck' ) 'Befehl';
```

Befehl wird so oft wiederholt, bis die Auswertung von *Ausdruck* den Wert FALSE (0) ergibt. Jeder andere Wert als FALSE (also jeder Wert groesser 0) wiederholt den / die Befehle.

Die Pruefung von *Ausdruck*' findet jeweils **VOR** der Ausfuehrung von Befehl statt, d.h. zu Anfang jedes Schleifendurchlaufs.

Beispiele zu while:

```
/* ----------------- 005example.c ----------------
     Beispiel fuer die Hilfedatei
     while - Demo
   ---------------------------------------------- */

#include "smallio.h"

int main(void)
{
  char  mytext[] = "Mein Name ist Hase\n\r";
  char  index;
  char  ch;
  char  *ptr;

  smallio_init();

  ptr= &mytext[0];              // Zeiger erhaelt Adresse von mytext

  // solange der Wert, auf den ptr zeigt ungleich 0
  // ist, wird die Schleife wiederholt.

  while (*ptr)
  {
    putchar(*ptr);
    ptr++;
  }

  index= 0;;

  // wiederhole so lange, wie der gelesene
  // Buchstabe in mytext ungleich 'H' ist

  while (mytext[index] != 'H')
  {
    index++;
  }

  printf("Buchstabe 'H' wurde gefunden an Position: %d \n\r", index);

  while(1);                     // Endlosschleife
}
```

do - while

Do - While Schleife

```
      do 'Befehl' while ( 'Ausdruck' );
```

Befehl wird solange ausgefuehrt, bis die Auswertung von *Ausdruck* den Wert 0 (FALSE) ergibt. Die Pruefung von *Ausdruck* findet jeweils **NACH** der Ausfuehrung von *Befehl* (d.h. nach durchlaufen des Schleifenrumpfes) statt.

Somit findet **MINDESTENS** ein Schleifendurchlauf statt (im Gegensatz zu einer while - Schleife, bei der die Pruefung **VOR** dem Schleifendurchlauf stattfindet).

Beispiel 1:

```
/* ----------------- 006aexample.c ----------------

     Beispiel fuer die Hilfedatei
     do - while Demo1
   ---------------------------------------------- */

#include "smallio.h"

int main(void)
{
  unsigned int ch;

  smallio_init();

  printf("\n\rdo - while Demo1\n\r----------------\n\r");

  do
  {
    printf("\n\rEingabe Zahl 1..99 (inkl.):  ");
    ch= readint();
  } while ( !((ch> 0) && (ch  < 100)));

  printf("\n\rEingabe korrekt...\n\r");
  while(1);
}
```

Beispiel 2 – Zahlensuchspiel:

```
/* ---------------- 006bexample.c ----------------
     Beispiel fuer die Hilfedatei
     do - while Demo
   ---------------------------------------------- */

#include "smallio.h"

int main(void)
{
  char suchzahl, versuche, ch;

  smallio_init();

  printf("\n\rdo - while Demo: Zahlensuchspiel");
  printf("\n\r--------------------------------\n\r");

  while(1)
  {
    // Zufallszahl generieren, Zaehler zaehlt so
    // lange immer wieder von 1..99 , bis ein Zeichen
    // auf der seriellen Schnittstelle eingeht und
    // stopt somit den Zaehler

    printf("\n\r generiere Zufallszahl, Taste fuer Stop\n\n\r");
    do
```

```
    {
      suchzahl++;
      suchzahl= (suchzahl % 99)+1;
      printf(" **\r ");
    } while (!keypressed());

    printf("\r    ");                // Zahlenanzeige loeschen
    ch= getchar();

    versuche= 1;

    do
    {
      do
      {
        printf("\n\r %d", versuche);
        printf(". Eingabe ( 1..99 ):  ");
        ch= readint();

      } while ( !((ch > 0) && (ch < 100)));

      if (ch == suchzahl)
      {
        printf("\n\n\r   Yeaaah, Zahl wurde gefunden");
        printf("\n\r   Benoetigte Versuche: %d", versuche);
        printf("\n\n\rTaste fuer weiter...\n\r");
      }
      else
      {
        if (ch > suchzahl) { printf("\n\rGesuchte Zahl ist kleiner !"); }
                       else { printf("\n\rGesuchte Zahl ist groesser !"); }
      }
      versuche++;
    } while (ch != suchzahl);
    ch= getchar();
  }
}
```

for

for-Schleife

```
for ( ['expr1'] ; ['expr2'] ; ['expr3'] ) 'Befehl' ;
```

Befehl wird solange wiederholt ausgefuehrt, bis die Auswertung von *expr2* FALSE ergibt (d.h. den Wert 0).

expr1 wird vor dem ersten Durchlauf ausgefuehrt und initialisiert eine sogenannte Laufvariable, die im Test mit *expr2* verwendet wird.

expr3 wird nach jedem Durchlauf ausgefuehrtt und in aller Regel zur Veraenderung der Laufvariablen verwendet. Alle drei Ausdruecke sind optional. Fuer ein nicht definiertes *expr2* wird eine 1 angenommen (was zu einer Endlosschleifefuehrt).

Beispiel fuer *for*:

```
/* ----------------- 007example.c ----------------

     Beispiel fuer die Hilfedatei
     for Demo
   ---------------------------------------------- */

#include "smallio.h"

int main(void)
{
  char ch;
  char i;

  smallio_init();

  printf("\n\rfor Demo\n\r--------\n\r");

  for (i= 11; i< 21; i++)
  {
    printf("\n\rdas Quadrat von  %d = %d", i, i*i);
  }
}
```

switch - case

Mehrweg-Verzweigung

```
      switch ( 'Ausdruck' ) 'Anweisung'
```

Ausdruck wird berechnet und muss einen Integerwert ergeben, danach folgt ein Vergleich mit der *Konstante* einen jeden case-Zweiges. Bei Uebereinstimmung wird der *Befehl* dieses Zweiges ausgefuehrt. Jeder Konstanten-Wert darf in der case-Liste nur einmal erscheinen.

Die optionale break-Anweisung hinter `Befehl'` fuehrt einen Sprung zum Ende von switch aus, d.h. zum naechsten auf das switch-Konstrukt folgenden Befehl. Der (optionale) default-Zweig wird nur dann ausgefuehrt, wenn alle vorherigen Vergleiche fehlgeschlagen sind.

```
  switch(Ausdruck)
  {
    case Konstante1 : Anweisung;
                      break;
    case Konstante2 : Anweisung;
                      break;
    .
    .
    .
    default:          Anweisung;
  }
```

Switch - case ist somit in Verbindung mit vielen Verzweigungsoptionen meistens uebersichtlicher als es stark verschachtelte *else - if* Ketten sind.

Beispiel zu switch – case:

```
/* ----------------- 008example.c ----------------

     Beispiel fuer die Hilfedatei
     switch - case Demo
      27.08.2015

   ---------------------------------------------- */

#include "smallio.h"

uint16_t calculate(char func, uint16_t x, uint16_t y)
{
  uint16_t z;

  switch (func)
  {
    case '/' : { z= x / y; break; }      // div
    case '+' : { z= x+y; break; }        // add
    case '-' : { z= x-y; break; }        // subb
    case 'i' : { z= x+1; break; }        // inc
    case 'x' :
    case 'X' :
    case '*' : { z= x*y; break; };       // mul

    case 's' :                           // sqrt
    case 'e' :                           // exp
    case 'm' : { printf(" Funktion nicht verfuegbar !");    // mod
                 break;
               }
    default : {
                printf(" Unerwarteter Fehler !");
               }
  }
  return z;
}

int main(void)
{
  char func;
  uint16_t op1, op2;

  smallio_init();                // damit I/O Anweisungen funktioneren

  printf("\n\rswitch - case demo : Integer calc\n\r");
  printf(    "---------------------------------");
  while(1)
  {
    printf("\n\n\rOperand1: ");
    op1= readint();

    printf("\n\rRechenfunktion (* + - / i): ");


    func= getchar();            // bei PC-Programmen getch anstelle von getchar verwenden
    printf("%c", func);

    if (func != 'i')
    {
      printf("\n\rOperand2: ");
      op2= readint();
    }

    printf("\n\n\r Ergebnis: %d", calculate(func,op1,op2));
  }
}
```

Funktionen

Eine der wichtigsten Herangehensweise bei der Programmierung von Computern / Mikrocontrollern ist das Aufteilen des Programms in Haupt- und Unterprogramme.

Unterprogramme dienen dafür, eine bestimmte und oefters wiederkehrende Aufgabenstellung (und nur diese) zu bearbeiten. Unterprogramme werden in C als *function* bezeichnet.

Jedem C-Compiler liegt ein Softwarepaket bei – den sogenannten Standardbibliotheken – die eine Vielzahl von Funktionen zur Verfügung stellen. Eine gute Unterteilung eines Programms in sinnvolle Funktionen sollte in jedem Programm stattfinden, damit dieses gut wartbar und an evtl. geaenderte Anforderungen anpassbar ist.

Grundsaetzlich gibt es 2 Arten von Funktionen:

- ohne Rueckgabe eines Ergebniswertes
- mit Rueckgabe eines Ergebniswertes.

Bei der Deklaration einer Funktion ist IMMER der Datentyp des Rueckgabewertes anzugeben. Funktionen, die keinen Wert zurueck geben haben den Datentyp void (engl. leere, nichts).

Funktion mit Rueckgabewert:

Beispiel:

```
/* ------------------------------------------------------------
     example_f01.c

     Beispiel zur Verwendung von Funktionen
   ------------------------------------------------------------ */

#include "smallio.h"

/* ----------------------------------------
                  tu_etwas
               eine eigene Funktion
   ---------------------------------------- */
void tu_etwas(void)                    // die Funktion heisst “tu_etwas“
{
  printf("\n\r Ich tu ja schon was...\n\r");
}

/* ----------------------------------------
                     main
     auch “main“ ist eine Funktion, sie ist
     die Hauptfunktion des Programmes
   ---------------------------------------- */
int main(void)
{
  smallio_init();

  tu_etwas();                          // Aufruf der eigenen Funktion
  while(1)
  {
  }
}
```

Das Beispiel oben macht nicht wirklich viel, sie verdeutlich jedoch den Aufruf einer Funktion. Hat die Funktion wie im Fall oben keinerlei weitere Funktionsargumente, so ist der Argumentenliste das Schluesselwort *void* (fuer “nichts“) einzutragen.

Funktionen koennen optional ein oder mehrere Funktionsargumente enthalten, die innerhalb der Funktion ausgewertet werden koennen. Fuer jedes Funktionsargument ist der Datentyp anzugeben. Ein solches Funktionsargument ist, wenn nicht durch *const* eingeschraenkt, auch als Variable innerhalb der Funktion verwendbar.

Hierfuer wird beim Funktionsaufruf Speicher fuer diese Variable reserviert und in der Funktion benutzt, der nach Beendigung der Funktion wieder frei gegeben wird. Gleiches gilt fuer Variable, die innerhalb einer Funktion angelegt werden. Fuer diese werden ebenfalls beim Aufruf Speicher reserviert, der bei Beendigung der Variable wieder freigegeben wird (so diese nicht durch das Schluesselwort *static* gekennzeichnet ist).

Beispiel:

```
/* ------------------------------------------------------
     example_f02.c

     Beispiel zur Verwendung von Funktionen
   ------------------------------------------------------ */

#include "smallio.h"

void print_add(uint16_t z1, uint16_t z2)
{
  uint16_t ergeb;

  ergeb= z1 + z2;
  printf("\n\r %d + %d = %d \n\r", z1, z2, ergeb);
}

/* ---------------------------------------
                  main
   --------------------------------------- */
int main(void)
{
  uint16_t zahl;                  // Variable, nur gueltig fuer main

  smallio_init();

  zahl= 42;
  print_add(zahl, 119);           // Aufruf der eigenen Funktion
  while(1)
  {
  }
}
```

Funktion mit Rueckgabewert

Funktionen, die ein Ergebnis zurueck geben sollen, muss anstelle von *void* der Datentyp angegeben werden, den er zurueckliefern soll.

```
int16_t addiere(int16_t zahl1, int16_t zahl2);
```

Die Funktion *addiere* liefert hier einen Wert vom Datentyp *uint16_t* zurueck, dessen Ergebnis bspw. einer Variablen vom selben Typ zugewiesen werden kann.

Das Ergebnis, das eine solche Funktion zurueckliefert, wird durch eine Angabe von

```
return wert;
```

erreicht. return beendet auch die Funktion (unabhaengig davon, ob im Quelltext weitere Anweisungen folgen oder nicht).

Beispiel:

```
/* ----------------------------------------------------------
     example_f03.c

     Beispiel zur Verwendung von Funktionen
   ---------------------------------------------------------- */

#include "smallio.h"

int16_t addiere(int16_t zahl1, int16_t zahl2)
{
  int16_t ergeb;

  ergeb= zahl1 + zahl2;
  return ergeb;
}

/* ----------------------------------------
                     main
   ---------------------------------------- */
int main(void)
{
  int16_t z1, z2, summe;               // Variable nur gueltig fuer main

  smallio_init();

  z1= 42;
  z2= 121;
  summe= addiere(z1, z2);
  printf("\n\r %d + %d = %d \n\r", z1, z2, summe);

  summe= addiere(128, 384);
  printf("\n\r %d + %d = %d \n\r", z1, z2, summe);
  while(1)
  {
  }
}
```

Erklärungen:

Mit *addiere* wird eine Funktion definiert (programmiert), die zwei Zahlen miteinander addiert und die Summe hiervon als Rueckgabewert der Funktion liefert.

Im Hauptprogramm *main* sind Variable *z1, z2* und *summe* definiert. *z1* und *z2* werden Werte zugewiesen und diese Variable der Funktion *addiere* uebergeben. Das Ergebnis wird der Variable *summe* zugewiesen und ein anschliessendes *printf* gibt das Ergebnis der Addition aus.

Ein zweiter Aufruf von *addiere* erfolgt dieses mal nicht mit Variablen, sondern mit konstanten Werten. Auch diese werden von der Funktion *addiere* verarbeitet (und anschliessend ausgegeben).

Call by reference

Wird an eine Funktion anstelle eines Wertes (value) die Speicheradresse (reference) einer Variable uebergeben die den Wert beinhaltet, so wird der Aufruf einer solchigen Funktion als "call by reference" bezeichnet. Die Uebergabe der Speicheradresse wird mittels Zeigervariablen vorgenommen (siehe auch Zeiger - Pointer).

Nachfolgendes Beispielprogramm beinhaltet 2 Funktionen, die zu "*call by reference*" Funktionen gehoeren.

- *myprint* gibt einen Text aus
- *searchreplace* sucht nach einem Zeichen in diesem Text und ersetzt alle vorkommenden Zeichen durch ein anderes.

Beispiel:

```
/* ------------------------------------------------------
     example_f04.c

     Beispiel zur Verwendung von Funktionen
   ------------------------------------------------------ */

#include "smallio.h"

void myprint(char *mytext)     // beinhaltet Adresse des Textes
{
  char   ch;

  while(*mytext)               // wiederhole solange, bis das Zeichen
  {                            // auf das *mytext zeigt > 0 ist und
                               // somit keine Endekennung ist.
    ch= *mytext;
    putchar(ch);
    mytext++;                  // Zeiger auf naechstes Element zeigen
                               // lassen
  }
}

void searchreplace(char search, char replace, char *mytext)
{
  while(*mytext)
  {
    if (*mytext == search)      // bei Uebereinstimmung des Zeichens
    {
      *mytext = replace;        // an dieselbe Stelle das Ersatz-
    }                           // zeichen setzen
    mytext++;                   // naechstes Element
  }
}

int main(void)
{
  char   mytext_dt[] = "Mein Name ist Hase\n\r";
  char   meldung[]   = "\n\rErsetze Buchstaben a durch einen Punkt .\n\n\r";
  smallio_init();              // fuer Ausgaben via printf

  myprint(&mytext_dt[0]);      // Funktion wird die Adresse von
                               // mytext_dt mit dem ersten Element ueber-
                               // geben.
  myprint(&meldung[0]);        // dto. mit Meldungstext

  searchreplace('a','.', &mytext_dt[0]);
  myprint(&mytext_dt[0]);

  while (1);
}
```

4.0

LIBC Standard - Dateien

Die Vollstaendigkeit der Beschreibungen zu den zu C gehoerenden LIBC Dateien ist nicht gegeben. Hier werden lediglich haeufig benutzte Funktionen und Makros aufgefuehrt. Eine komplette Dokumentation der AVR-GCC Libraries ist (in Englisch) zu finden unter:

http://www.nongnu.org/avr-libc/user-manual/modules.html

Eine komplette Beschreibung fuer glibc (gueltig fuer PC- und in sehr grossen Teilen fuer ARM Cortex-Programmierung findet sich unter:

http://www.gnu.org/software/libc/manual/html_node/index.html

string.h

Die Bibliothek string.h beinhaltet grundsaetzlich Funktionen zum Umgang mit Zeichenketten (Strings). Alle Funktionen arbeiten mit sogenannten '*Ascii-Zero*' Strings, das bedeutet, dass das Ende eines Strings mit einem 0-Byte markiert ist.

Werden Strings als Konstante im Quelltext eingegeben, so ist das 0-Byte nicht mit hinzuzufuegen, dieses erledigt der Compiler eigenstaendig. Bei Array-Dimensionierungen ist jedoch darauf zu achten, dass der Speicherplatz fuer dieses 0-Byte mit reserviert werden muss.

Die Prototypen aus string.h

- memccpy
- memchr
- memcmp
- memcpy
- memmove
- memset
- strcat
- strchr
- strcmp
- strcpy
- strcspn
- strdup
- strlen
- strlwr
- strncat
- strncmp
- strncpy
- strpbrk
- strrchr
- strrev
- strspn
- strstr
- strtok
- strupr

memccpy

kopiert einen Block von n Bytes im Speicher von *src* nach *dest*.

```
       void *memccpy(void *dest, const void *src, int c, size_t n);
```

Kopiert einzelne Bytes von *src* nach *dest* und bricht entweder nach n Bytes oder nach der Kopie eines Bytes mit dem Wert c ab. Im ersten Fall ist das Ergebnis NULL, im zweiten Fall liefert *memccpy* einen Zeiger auf das Byte zurueck, das unmittelbar in *dest* auf c folgt.

```
/* ------------------------------------------------------
     example_str01.c

     Beispiel zur Verwendung von memccpy
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
   char *src = "Dies ist die Source - Datei";
   char dest[50];
   char *ptr, *ptrs;

   smallio_init();

   ptrs = &dest[0];
   ptr = memccpy(dest, src, 'c', strlen(src));
   if (ptr)
   {
      printf("Das Zeichen wurde gefunden !\n\r");
      printf("Kopierter Text ist: %s\n\r", dest);
      printf("Kopierte Bytes: %d\n\r", ptr-ptrs);
      *ptr = '\0';
   }
   else
      printf("Das Zeichen wurde nicht gefunden\n\r");
   while(1);
}
```

Ausgabe:

```
       Das Zeichen wurde gefunden !
       Kopierter Text ist: Dies ist die Sourc
       Kopierte Bytes: 18
```

memchr

sucht die ersten n Bytes eines Arrays nach dem Wert c ab.

```
       *memchr(const void *s, int c, size_t n);
```

Liefert einen Zeiger auf das erste Vorkommen von c in s bzw. den Wert NULL, wenn die Suche erfolglos war.
dest

```
/* ------------------------------------------------------
     example_str02.c

     Beispiel zur Verwendung von memchr
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
   char str[17];
   char *ptr;

   smallio_init();

   strcpy(str, "Das ist reiner Wahnsinn");
   ptr = memchr(str, 'r', strlen(str));
   if (ptr)
      printf("Das Zeichen 'r' ist an Position: %d\n\r", ptr-str);
   else
      printf("Das Zeichen wurde nicht gefunden\n\r");
   while(1);
}
```

Ausgabe:

```
       Das Zeichen 'r' ist an Position: 8
```

memcmp

vergleicht die ersten n Bytes zweier Strings miteinander.

```
  int memcmp(const void *s1, const void *s2, size_t n);
```

Liefert folgendes Ergebnis

kleiner 0, wenn s1 kleiner s2 ist
gleich 0, wenn s1 identisch mit s2 ist
groesser 0, wenn s1 groesser als s2 ist

```
/* ------------------------------------------------------
     example_str03.c

     Beispiel zur Verwendung von memcmp
   ------------------------------------------------------ */

#include <string.h>

#include "smallio.h"

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   smallio_init();

   memcpy(str1, "abcdef", 6);
   memcpy(str2, "ABCDEF", 6);

   ret = memcmp(str1, str2, 5);

   if(ret > 0)
   {
      printf("str2 ist kleiner als str1\n\r");
   }
   else if(ret < 0)
   {
```

```
      printf("str1 ist kleiner als str2\n\r");
   }
   else
   {
      printf("str1 und str2 sind identisch\n\r");
   }
   return(0);
}
```

memcpy

kopiert n Bytes von src nach dest.

```
       *memcpy(void *dest, const void *src, size_t n);
```

Liefert den als *dest* uebergebenen Zeiger. Die Funktionsweise von *memcpy* ist undefiniert, falls sich *src* und *dest* ueberlappen.

```
/* ------------------------------------------------------------
     example_str04.c

     Beispiel zur Verwendung von memcpy
   ------------------------------------------------------------ */
/*

#include <string.h>
#include "smallio.h"

int main(void)
{

   smallio_init();

   char src[] = "***************";
   char dest[] = "abcdefghijlkmnopqrstuvwxy";
   char *ptr;

   printf("Zielstring vor memcpy : %s\n\r", dest);
   ptr = memcpy(dest, src, strlen(src));
   if (ptr)
      printf("Zielstring nach memcpy: %s\n\r", dest);
   else
      printf("memcpy failed\n\r");
   while(1);
}
```

Ausgabe:

```
       Zielstring vor memcpy : abcdefghijlkmnopqrstuvwxy
       Zielstring nach memcpy: ***************pqrstuvwxy
```

memmove

kopiert n Bytes von src nach dest.

```
       void *memmove(void *dest, const void *src, size_t n);
```

Liefert den als *dest* uebergebenen Zeiger. *memmove* funktioniert auch dann korrekt, wenn sich *src* und *dest* ueberlappen.

```
/* ------------------------------------------------------
     example_str05.c

     Beispiel zur Verwendung von memmove
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
  char dest[] = ("abcdefghijklmnopqrstuvwxyz");
  char src[]  = ("*****************");

  smallio_init();

  printf("Zielstring vor memmove : %s\n\r", dest);
  memmove(dest, src, strlen(src));
  printf("Zielstring nach memmove: %s\n\r", dest);
}
```

Ausgabe:

```
       Zielstring vor memmove : abcdefghijklmnopqrstuvwxyz
       Zielstring nach memmove: ****************qrstuvwxyz
```

memset

setzt die ersten n Bytes von s auf den Wert c.

```
       void *memset(void *s, int c, size_t n)
```

Liefert den als s ubergebenen Zeiger zurueck.

```
/* ------------------------------------------------------
     example_str06.c

     Beispiel zur Verwendung von memset
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
  char buffer[] = "Hallo Welt\n\r";
  smallio_init();

  printf("Bufferspeicher vor memset : %s", buffer);
  memset(buffer, '*', strlen(buffer) - 2);
  printf("Bufferspeicher nach memset: %s", buffer);
  while(1);
}
```

Ausgabe:

```
       Bufferspeicher vor memset : Hallo Welt
       Bufferspeicher nach memset: **********
```

strcat

haengt den Inhalt von *src* an den String *dest* an.

```
   char *strcat(char *dest, const char *src);
```

```
/* ------------------------------------------------------------
     example_str07.c

     Beispiel zur Verwendung von strcat
   ------------------------------------------------------------ */

/*

#include <string.h>
#include "smallio.h"

int main(void)
{

  char destination[25];
  char *atmel = "Atmel";
  char *blank = " ";
  char *avr   = "AVR";

  smallio_init();

  strcpy(destination, atmel);
  strcat(destination, blank);
  strcat(destination, avr);

  printf("%s\n\r", destination);
  while(1);
}
```

Ausgabe:

```
       Atmel AVR
```

strchr

sucht in einem String ein Zeichen.

```
       char *strchr(const char *str, int c);
```

Liefert einen Zeiger auf das erste Vorkommen von c in str bzw. den Wert NULL, wenn c in str nicht enthalten ist.

```
/* ------------------------------------------------------------
     example_str08.c

     Beispiel zur Verwendung von strchr
   ------------------------------------------------------------ */

/*

#include <string.h>
#include "smallio.h"

int main(void)
{
  char string[15];
  char *ptr, c = 'r';

  smallio_init();
  smallio_init();

  strcpy(string, "Das ist reiner Wahnsinn");
  ptr = strchr(string, c);
  if (ptr)
    printf("Das Zeichen %c ist an Position: %d\n\r", c, ptr-string);
```

```
  else
    printf("Das Zeichen wurde nicht gefunden\n\r");

  while(1);
}
```

Ausgabe:

```
      Das Zeichen r ist an Position: 8
```

strcmp

vergleicht zwei Strings miteinander.

```
   int strcmp(const char *s1, const char *s2);
```

Liefert folgendes Ergebnis

kleiner 0, wenn s1 kleiner s2 ist
gleich 0, wenn s1 identisch mit s2 ist
groesser 0, wenn s1 groesser als s2 ist

```
/* ------------------------------------------------------------
     example_str09.c

     Beispiel zur Verwendung von strcmp
   ------------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
   char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
   int ptr;

   smallio_init();

   ptr = strcmp(buf2, buf1);
   if (ptr > 0)
      printf("Buffer 2 ist groesser als Buffer 1\n\r");
   else
      printf("Buffer 2 ist kleiner als Buffer 1\n\r");

   ptr = strcmp(buf2, buf3);
   if (ptr > 0)
      printf("Buffer 2 ist groesser als  Buffer 3\n\r");
   else
      printf("Buffer 2 ist kleiner als Buffer 3\n\r");

   while(1);
}
```

strcpy

kopiert den Inhalt von *src* in den String *dest* (und ueberschreibt den vorherigen Inhalt von *dest* dabei).

```
char *strcpy(char *dest, const char *src);
```

Liefert die als *dest* uebergebene Adresse.

```
/* ------------------------------------------------------
     example_str010.c

     Beispiel zur Verwendung von strcmp
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{

  smallio_init();

  char string[10];
  char *str1 = "abcdefghi";

  strcpy(string, str1);
  printf("%s\n\r", string);
  while(1);
}
```

Ausgabe:

```
abcdefghi
```

strcspn

durchsucht einen String nach dem Vorkommen bestimmter Zeichen.

```
size_t strcspn(const char *s1, const char *s2);
```

Liefert die Position des ersten Zeichens in s1, das sowohl in s1 als auch in s2 enthalten ist. wird kein Zeichen gefunden, so wird die Stringlaenge (0-Byte gefunden) von s1 zurueckgegeben.

```
/* ------------------------------------------------------
     example_str11.c

     Beispiel zur Verwendung von strcspn
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"
int main(void)
{
  smallio_init();

  char string1[] = "Atmel AVR";
  char string2[] = "efghi";
  int length;

  length = strcspn(string1, string2);
  printf("\n\r Das erste Zeichen in string1 welches ebenso in string2 ist,");
  printf("\n\r ist an Position: %d (%c)", length, string1[length]);
  while(1);
}
```

Ausgabe:

```
Das erste Zeichen in string1 welches ebenso in string2 ist,
ist an Position: 3 (e)
```

strdup

belegt dynamisch Speicher und erzeugt ein String-Duplikat.

```
char *strdup(const char *s);
```

Wenn kein Platz auf dem Heap vorhanden ist, liefert strdup den Wert NULL; ansonsten einen Zeiger auf den neu erzeugten String. Der entsprechende Bereich muss vom Programmierer explizit wieder freigegeben werden.

```
/* ------------------------------------------------------
     example_str12.c

     Beispiel zur Verwendung von strdup
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{

  smallio_init();

  char *dup_str, *string = "abcde";

  dup_str = strdup(string);
  printf("%s\n", dup_str);
  free(dup_str);

  while(1);
}
```

Ausgabe:

```
abcde
```

strlen

ermittelt die Laenge eines Strings.

```
size_t strlen(const char *s);
```

Das abschliessende NULL-Zeichen des Strings wird nicht mitgezaehlt.

```
/* ------------------------------------------------------
     example_str13.c

     Beispiel zur Verwendung von strlen
   ------------------------------------------------------ */
/*

#include <string.h>
#include "smallio.h"

int main(void)
{

  smallio_init();

  char *string = "Dies ist ein Testtext";

  printf("\n\r%s", string);
```

```
  printf("\n\rDie Laenge des obigen Textes betraegt %d Zeichen", strlen(string));
  fflush(stdout);
  while(1);
}
```

Ausgabe:

```
       Dies ist ein Testtext
       Die Laenge des obigen Textes betraegt 21 Zeichen
```

strlwr

verwandelt alle Kleinbuchstaben eines Strings in Grossbuchstaben; funktioniert nicht fuer die deutschen Umlaute.

```
       char *strlwr(char *s);
```

Liefert einen Zeiger auf den String s zurueck.

Hinweis: *strlwr* ist (warum auch immer) fuer den AVR-GCC, jedoch nicht fuer den GCC (PC-Programmierung) verfuegbar. strlwr kann fuer den GCC folgenderweise nachgebildet werden:

```
#include <ctype.h>               // fuer tolower

char *strlwr(char *str)
{
  unsigned char *p = (unsigned char *)str;

  while (*p)
  {
    *p = tolower((unsigned char)*p);
    p++;
  }
  return str;
}
```

Beispiel zu *strlwr*:

```
/* ------------------------------------------------------------
     example_str14.c

     Beispiel zur Verwendung von strlwr
   ------------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{

  smallio_init();

  char string[] = "Dies ist ein Testtext";

  printf("String vor strlwr : %s\n\r", string);
  strlwr(string);
  printf("String nach strlwr: %s\n\r", string);
  while(1);
}
```

Ausgabe:

```
       String vor strlwr : Dies ist ein Testtext
       String nach strlwr: dies ist ein testtext
```

strncat

haengt maximal maxlen Zeichen von src an dest an.

```
       char *strncat(char *dest, const char *src, size_t maxlen);
```

Liefert den als dest uebergebenen Zeiger.

```
/* ------------------------------------------------------
     example_str15.c

     Beispiel zur Verwendung von strncat
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{

  smallio_init();

  char destination[40];
  char *source = "Deutschland";

  strcpy(destination, "Bundesrepublik ");
  strncat(destination, source, 11);
  printf("%s\n\r", destination);
  while(1);
}
```

Ausgabe:

```
       Bundesrepublik Deutschland
```

strncmp

vergleicht die ersten maxlen Zeichen zweier Strings.

```
       int strncmp(const char *s1, const char *s2, size_t maxlen);
```

Fuehrt einen signed Vergleich durch und liefert folgendes Ergebnis

kleiner 0, wenn s1 kleiner s2 ist
gleich 0, wenn s1 identisch mit s2 ist
groesser 0, wenn s1 groesser als s2 ist

```
/* ------------------------------------------------------
     example_str16.c

     Beispiel zur Verwendung von strncmp
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
   char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
   int ptr;

   smallio_init();

   ptr = strncmp(buf2,buf1,3);
   if (ptr > 0)
      printf("Buffer 2 ist groesser als Buffer 1\n\r");
   else
      printf("Buffer 2 ist kleiner als Buffer 1\n\r");

   ptr = strncmp(buf2,buf3,3);
   if (ptr > 0)
      printf("Buffer 2 ist groesser als Buffer 3\n\r");
   else
      printf("Buffer 2 ist kleiner als Buffer 3\n\r");
```

```
   while(1);
}
```

Ausgabe:
```
       Buffer 2 ist groesser als Buffer 1
       Buffer 2 ist kleiner als Buffer 3
```

strncpy

kopiert maximal maxlen Zeichen von *src* nach *dest* (und ueberschreibt dest dabei).

```
       char *strncpy(char *dest, const char *src, size_t maxlen);
```

Liefert den als dest uebergebenen Zeiger.

```
/* ------------------------------------------------------
     example_str17.c

     Beispiel zur Verwendung von strncpy
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{

  char string[10];
  char *str1 = "abcdefghi";

  smallio_init();

  strncpy(string, str1, 3);
  string[3] = '\0';
  printf("%s\n\r", string);
  while(1);
}
```

Ausgabe:
```
       abc
```

strpbrk

sucht das erste Vorkommen eines Zeichens aus s2 in s1.

```
       char *strpbrk(const char *s1, const char *s2);
```

Liefert einen Zeiger auf das erste Vorkommen eines beliebigen Zeichens von s2 in s1 bzw. den Wert NULL, wenn s1 ueberhaupt keines der Zeichen von s2 enthaelt.

```
/* ------------------------------------------------------
     example_str18.c

     Beispiel zur Verwendung von strpbrk
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{

  char *string1 = "abcdefghijklmnopqrstuvwxyz";
  char *string2 = "onm";
  char *ptr;
  smallio_init();
  ptr = strpbrk(string1, string2);
  if (ptr)
     printf("Erstes von strpbrk gefundenes Zeichen: %c\n\r", *ptr);
  else
     printf("strpbrk hat kein Zeichen gefunden\n\r");
  while(1);
}
```

Ausgabe:

```
      Erstes von strpbrk gefundenes Zeichen: m
```

strrchr

sucht das letzte Vorkommen des Zeichens c im String s.

```
      char *strrchr(const char *s, int c);
```

Liefert einen Zeiger auf das letzte Vorkommen von c in s bzw. den Wert NULL, wenn s das Zeichen c ueberhaupt nicht enthaelt.

```
/* ------------------------------------------------------
     example_str19.c

     Beispiel zur Verwendung von strrchr
   ------------------------------------------------------ */
/*
#include <string.h>
#include "smallio.h"

int main(void)
{
  char mystring[20];
  char *ptr;
  char c = 'i';

  smallio_init();

  strcpy(mystring, "Dies ist ein String");
  printf("\n\r.........1.........2\n\r");
  printf("01234567890123456789\n\r");

  printf("%s\n\n\r",mystring);

  ptr = strrchr(mystring, c);

  if (ptr)
    printf("Das letzte Vorkommen von '%c' ist an Pos.: %d\n\r", c, ptr - mystring);
  else
    printf("Das Zeichen wurde nicht gefunden\n\r");
  while(1);
}
```

Ausgabe:

```
      .........1.........2
```

```
       01234567890123456789
       Dies ist ein String

       Das letzte Vorkommen von 'i' ist an Pos.: 16 df
```

strrev

Dreht die Reihenfolge der Zeichen eines Strings s um (das abschliessende Nullzeichen ausgeschlossen).

```
       char *strrev(char *s);
```

Liefert einen Zeiger auf den gespiegelten String zurueck.

Hinweis: *strrev* ist (warum auch immer) fuer den AVR-GCC, jedoch nicht fuer den GCC (PC-Programmierung) verfuegbar. *strrev* kann fuer den GCC folgenderweise nachgebildet werden:

```
char *strrev(char *str)
{
  char *p1, *p2;

  if (! str || ! *str) return str;

  for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2)
  {
    *p1 ^= *p2;
    *p2 ^= *p1;
    *p1 ^= *p2;
  }
  return str;
}
```

Beispiel:

```
/* ------------------------------------------------------------
     example_str20.c

     Beispiel zur Verwendung von strrev
   ------------------------------------------------------------ */

#include <string.h>
#include "smallio.h"



int main(void)
{
  char forward[] = "AVR ist super";

  smallio_init();

  printf("\n\rText vor strrev() : %s", forward);
  strrev(forward);
  printf("\n\rText nach strrev(): %s\n", forward);

  while(1);
}
```

Ausgabe:

```
       Text vor strrev() : AVR ist super
       Text nach strrev(): repus tsi RVA
```

strspn

Sucht einen String nach dem Vorkommen eines bestimmten Teilstrings ab.

```
size_t strspn(const char *s1, const char *s2);
```

Liefert die Laenge des Teilstrings von s1 zurueck, der ausschliesslich aus den in s2 angegebenen Zeichen besteht.

```
/* ------------------------------------------------------
     example_str21.c

     Beispiel zur Verwendung von strspn
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
  char *string1 = "1234567890";
  char *string2 = "123DC8";
  int length;

  smallio_init();

  length = strspn(string1, string2);
  printf("Zeichen, von dem die Strings voneinander abweichen ist an "    \
         "Position %d\n\r", length);
  while(1);
}
```

Ausgabe:

```
Zeichen, von dem die Strings voneinander abweichen ist an Position 3
```

strstr

sucht s1 nach dem ersten Vorkommen des Strings s2 ab.

```
char *strstr(const char *s1, const char *s2);
```

Liefert einen Zeiger auf den Anfang von s2 in s1 bzw. den Wert NULL, wenn s2 in s1 ueberhaupt nicht vorkommt.

```
/* ------------------------------------------------------
     example_str22.c

     Beispiel zur Verwendung von strstr
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
  char *str1 = "Mein Name ist Hase";
  char *str2 = "Hase";
  char *ptr;

  smallio_init();

  ptr = strstr(str1, str2);
  printf("123456789012345678\n\r");
  printf("%s\n\r",str1);
  printf("Substring '%s' wurde gefunden an Pposition: %d\n\r", str2, ptr-str1+1);
  while(1);
}
```

Ausgabe:

```
       123456789012345678
       Mein Name ist Hase
       Substring 'Hase' wurde gefunden an Pposition: 15
```

strtok

Sucht das erste (oder das naechste) 'token' aus s1heraus, d.h. eine Zeichenfolge, die kein Zeichen aus s2 enthaelt (sucht somit einen String bis zu einem Trennzeichen ab).

```
       char *strtok(char *s1, const char *s2);
```

s2 definiert das Trennzeichen. *strtok* interpretiert s1 als eine Reihe von 'tokens', die durch ein oder mehrere Zeichen aus s2 voneinander getrennt sind.

Falls kein 'token' gefunden wurde, ist das Ergebnis NULL, ansonsten schreibt *strtok* an die Stelle nach dem 'token' in s1 ein NULL-Zeichen und liefert einen Zeiger auf den Anfang des 'tokens.

Nachfolgende Aufrufe von strtok mit NULL als erstem Argument bearbeiten den String s1 weiter. Es koennen dabei auch andere Trennzeichen (in s2) verwendet werden.

```
/* ------------------------------------------------------
     example_str23.c

     Beispiel zur Verwendung von strtok
   ------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
   char myzahlen[30] = "13.89 #14.92 #15.97";
   char *ptr;

   smallio_init();

   printf("Extraktion einer Zahlenreihe\n\r\n\r");

   /* strtok platziert ein 0-Byte an der Stelle
     an der das Token gefunden wurde. ptr ist nun ein
     Zeiger auf einen String der an der Stelle
     terminiert ist, an der das Token WAR               */
```

```
   ptr = strtok(myzahlen, "#");

   while (ptr)               // solange ptr nicht auf ein
   {                         // Textende zeigt

     if (ptr)  printf("%s\n\r", ptr);

     /* Ein weiterer Aufruf von strtok mit einem NULL
        Zeiger sucht das naechste Token ersetzt dieses,
        uebergibt wieder den Zeiger darauf */
     ptr = strtok(NULL, "#");
   }
   while(1);
}
```

Ausgabe:

```
      Extraktion einer Zahlenreihe

      13.89
      14.92
      15.97
```

strupr

Wandelt alle Kleinbuchstaben eines Strings in Grossbuchstaben um.

```
      char *strupr(char *s);
```

Liefert einen Zeiger auf s zurueck.

Hinweis: *strupr* ist (warum auch immer) fuer den AVR-GCC, jedoch nicht fuer den GCC (PC-Programmierung) verfuegbar. strupr kann fuer den GCC folgenderweise nachgebildet werden:

```
#include <ctype.h>              // fuer toupper

char *strupr(char *str)
{
  unsigned char *p = (unsigned char *)str;

  while (*p)
  {
    *p = toupper((unsigned char)*p);
    p++;
  }
  return str;
}
```

Beispiel:

```
/* ------------------------------------------------------------
     example_str24.c

     Beispiel zur Verwendung von strupr
   ------------------------------------------------------------ */

#include <string.h>
#include "smallio.h"

int main(void)
{
  smallio_init();

  char string[] = "abcdefghijklmnopqrstuvwxyz";
  char *ptr;

  // Konvertierung des Strings nach Grossbuchstaben
  ptr = strupr(string);
  printf("%s\n\r", ptr);
```

```
  while(1);
}
```

Ausgabe:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

stdlib.h

Die Bibliothek stdlib.h beinhaltet verschiedene Funktionen zum Umgang mit Zahlen sowie Funktionen zur Speicherverwaltung.

Die Prototypen aus stdlib.h

- abs
- atof
- atoi
- atol
- bsearch
- calloc
- div
- free
- itoa
- labs
- ldiv
- ltoa
- malloc
- qsort
- rand
- realloc
- srand
- utoa
- ultoa

abs

Das Makro abs berechnet den absoluten Betrag einer Integer-Zahl.

```
int abs(int x);
```

Prototyp in math.h (fuer real) / stdlib.h

```
/* ------------------------------------------------------
     example_slib01.c

     Beispiel zur Verwendung von abs
   ------------------------------------------------------ */

#include <math.h>
#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  int zahl = -1234;

  smallio_init();

  printf("Zahl: %d  Absolutwert: %d\n",    \
         zahl, abs(zahl));
  while(1);
}
```

Ausgabe:

```
Zahl: -1234  Absolutwert: 1234
```

atof

Konvertiert einen String in eine Gleitkommazahl.

```
double atof(const char *s);
```

atof liefert das Ergebnis der Interpretation bzw. den Wert 0, wenn die Konvertierung nicht fehlerfrei ausgefuehrt werden konnte.

```
/* ------------------------------------------------------
     example_slib02.c

     Beispiel zur Verwendung von atof
   ------------------------------------------------------ */
#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  float f;
  char *str = "12345.67";

  smallio_init();

  f = atof(str);
  printf("Text = %s Kommazahl = %f\n", str, f);
  while(1);
}
```

Ausgabe:

```
Text = 12345.67 Kommazahl = 12345.669922
```

atoi

Das Makro atoi konvertiert einen String in einen Integerwert.

```
int atoi(const char *s);
```

atoi liefert das Ergebnis der Interpretation bzw. den Wert 0, wenn die Konvertierung nicht fehlerfrei ausgefuehrt werden konnte.

```
/* ------------------------------------------------------
     example_slib03.c

     Beispiel zur Verwendung von atoi
   ------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  int n;
  char *str = "12345.67";

  smallio_init();

  n = atoi(str);
  printf("String = %s;  Integer = %d\n", str, n);
  while(1);
}
```

Ausgabe:

```
String = 12345.67;  Integer = 12345
```

atol

Konvertiert den Inhalt eines Strings in einen long-Wert.

```
long atol(const char *s);
```

atol liefert das Ergebnis der Interpretation bzw. den Wert 0, wenn die Konvertierung nicht fehlerfrei ausgefuehrt werden konnte.

```
/* ------------------------------------------------------
     example_slib04.c

     Beispiel zur Verwendung von atol
   ------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  long l;
  char *str = "98765432";

  smallio_init();

  l = atol(str);
  printf("Text = %s Integer = %ld\n", str, l);
  while(1);
}
```

Ausgabe:

```
Text = 98765432 Integer = 98765432
```

bsearch

Binaeres Absuchen einer Liste.

```
void *bsearch(const void *key,
      const void *base, size_t *nelem,
      size_t width, int (*fcmp)(const void*, const void*));
```

Liefert die Adresse des ersten Elements von *base*, dessen Inhalt mit dem von *key* uebereinstimmt. Wenn die Suche erfolglos bleibt, ist das Ergebnis NULL.

bsearch interpretiert den Rueckgabewert der (benutzerdefinierten) Vergleichsfunktion **fcmp* wie folgt:

kleiner 0 bedeutet:	*elem1 < *elem2
gleich 0 bedeutet:	*elem1 == *elem2
groesser 0 bedeutet:	*elem1 > *elem2

Die Tabelle muss in aufsteigender Reihenfolge sortiert sein.

```
/* ------------------------------------------------------
     example_slib05.c

     Beispiel zur Verwendung von bsearch
   ------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))

int numarray[] = {123, 145, 512, 627, 800, 933};

int numeric (const int *p1, const int *p2)
{
   return(*p1 - *p2);
}

int lookup(int key)
{
  int *itemptr;

  /* Cast von (int(*)(const void *,const void*))
     wird benoetigt um einen 'type mismatch' Fehler
     beim Kompilieren zu verhindern */

  itemptr = bsearch (&key, numarray, NELEMS(numarray),
            sizeof(int),
            (int(*)(const void *,const void *))numeric);

  return (itemptr != NULL);
}

int main(void)
{
  smallio_init();

  if (lookup(512))
     printf("512 ist in der Tabelle.\n\r");
  else
     printf("512 ist nicht in der Tabelle.\n\r");
  while(1);
}
```

Ausgabe:

```
      512 ist in der Tabelle.
```

calloc

Reserviert Platz im Hauptspeicher.

```
       void *calloc(size_t nitems, size_t size);
```

Reserviert einen Block von *nitems* * size Bytes und initialisiert sie mit 0.

Liefert einen Zeiger auf den neu belegten Block. Wenn nicht genug Platz vorhanden ist oder *nitems* oder *size* den Wert 0 hat, ist das Ergebnis NULL.

```
/* ------------------------------------------------------
     example_slib06.c

     Beispiel zur Verwendung von calloc
   ------------------------------------------------------ */

#include <string.h>
#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  char *str = NULL;

  smallio_init();

  // Speicher fuer einen String reservieren (allozieren)
  str = calloc(10, sizeof(char));

  strcpy(str, "Hallo");
  printf("Der Text ist: %s\n\r", str);
  free(str);                              // Speicher freigeben
  while(1);
}
```

Ausgabe:

```
       Der Text ist: Hallo
```

div

dividiert zwei Integerwerte.

```
       div_t div(int numer, int denom);
```

div dividiert die als *numer* (Dividend) und als *denom* (Divisor) uebergebenen Integerwerte und liefert eine Struktur des Typs *div_t* zurueck. Dieser Datentyp ist in stdlib.h wie folgt definiert:

```
 typedef struct
 {
   int quot;  // Quotient (Divisionsergebnis)
   int rem;   // Divisionsrest
 } div_t;
```

Beispiel:

```
/* ------------------------------------------------------
     example_slib07.c

     Beispiel zur Verwendung von div
   ------------------------------------------------------ */
#include <stdlib.h>
#include "smallio.h"

int main(void)
{
```

```
   div_t x;

   smallio_init();

   x = div(10,3);
   printf("10 div 3 = %d Rest %d\n", x.quot, x.rem);
   while(1);
}
```

Ausgabe:

```
       10 div 3 = 3 Rest 1
```

free

gibt einen mit *malloc* oder *calloc* dynamisch reservierten Speicherbereich wieder frei.

```
       void free(void *block);
```

Beispiel:

```
/* ------------------------------------------------------------
     example_slib08.c

     Beispiel zur Verwendung von free
   ------------------------------------------------------------ */

#include <string.h>
#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  char *str;

  smallio_init();

  // Speicher fuer einen String reservieren
  str = malloc(10);
  strcpy(str, "Hallo");
  printf("Text ist: %s\n", str);
  free(str);           // Speicher freigeben
  while(1);
}
```

Ausgabe:

```
       Text ist: Hallo
```

itoa

Konvertiert einen Integerwert in einen String.

```
char *itoa(int value, char *string, int radix);
```

Zurueckgeliefert wird ein Zeiger auf den als string uebergebenen String. radix beinhaltet die Angabe, zu welcher Zahlenbasis die Konvertierung erfolgen soll.

Hinweis: *itoa* ist kein ANSI-C aber fuer den AVR-GCC und ARM-NONE-EABI-GCC, jedoch nicht fuer den GCC (PC-Programmierung) verfuegbar. Dieses hat u.a. seinen Ursprung darin, dass der Datentyp Integer je nach verwendetem Betriebssystem 32 oder 64 Bit lang sein kann. *itoa* kann fuer den GCC folgenderweise nachgebildet werden:

```
char *strrev(char *str)
// strrev ist ebenfalls in GCC fuer PC-Programmierung nicht verfuegbar
{
  char *p1, *p2;

  if (! str || ! *str) return str;

  for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2)
  {
    *p1 ^= *p2;
    *p2 ^= *p1;
    *p1 ^= *p2;
  }
  return str;
}

int itoa(int num, unsigned char *str, int base)
// itoa - Ersatz bei Verwendung des GCC
{
  int sum = num;
  int i = 0;
  int digit;

  do
  {
    digit = sum % base;
    if (digit < 0xA) str[i++] = '0' + digit;
                else str[i++] = 'A' + digit - 0xA;
    sum /= base;
  } while (sum && (i < (63)));

  if (i == (63) && sum) return -1;
  str[i] = '\0';
  strrev(str);
  return 0;
}
```

Beispiel:

```
/* ------------------------------------------------------------
     example_slib09.c

     Beispiel zur Verwendung von itoa
   ------------------------------------------------------------ */

#include <stdlib.h>
#include <string.h>
#include "smallio.h"

int main(void)
{
  int zahl = 123412;
  char string[25];
```

```
  smallio_init();

  itoa(zahl, string, 10);
  printf("Zahl = %d; resultierender String       : %s\n", zahl, string);
  itoa(zahl, string, 16);
  printf("Zahl = %d; resultierender String (hex) : %s\n", zahl, string);
  while(1);
}
```

Ausgabe:

```
       Zahl = 123412; resultierender String       : 123412
       Zahl = 123412; resultierender String (hex) : 1E214
```

labs

Liefert den absoluten Betrag eines long-Wertes.

```
       long int labs(long int x);
```

```
/* ------------------------------------------------------
     example_slib10.c

     Beispiel zur Verwendung von labs
   ------------------------------------------------------ */

#include <stdlib.h>
#include <math.h>
#include "smallio.h"

int main(void)
{
   long result;
   long x = -12345678L;

   smallio_init();

   result= labs(x);
   printf("Zahl: %ld Absolutwert: %ld\n",
      x, result);
   while(1);
}
```

Ausgabe:

```
       Zahl: -12345678 Absolutwert: 12345678
```

ldiv

Dividiert zwei long-Werte und liefert sowohl den Quotienten als auch den Divisionsrest in einer Struktur zurueck.

```
       ldiv_t ldiv(long int numer, long int denom);
```

ldiv dividiert die als *numer* und *demon* uebergebenen long-Werte und liefert eine Struktur des Typs *ldiv_t* zurueck. Dieser Datentyp ist in *stdlib.h* wie folgt definiert:

```
       typedef struct
       {
         long int quot;    // Quotient
         long int rem;     // Divisionsrest
       } ldiv_t;
```

```
/* ------------------------------------------------------
     example_slib11.c

     Beispiel zur Verwendung von ldiv
   ------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main(void)
{
   ldiv_t lx;

   smallio_init();

   lx = ldiv(100000L, 30000L);
   printf("100000 div 30000 = %ld Rest %ld\n", lx.quot, lx.rem);
   while(1);
}
```

Ausgabe:

```
       100000 div 30000 = 3 Rest 10000
```

ltoa

konvertiert einen long-Wert in einen String.

```
       char *ltoa(long value, char *string, int radix);
```

radix beinhaltet die Angabe, zu welcher Zahlenbasis die Konvertierung erfolgen soll. *radix* muss groesser als 1 und kleiner als 37 sein. radix= 10 erzeugt dezimale, radix= 16 hexadezimale Darstellung

Hinweis: *ltoa* ist kein ANSI-C aber fuer den AVR-GCC, jedoch nicht fuer den GCC und ARM-NONE-EABI-GCC implementiert. Fuer ein Programmbeispiel zur Realisierung eines *itoa*, siehe auch Erklaerung zu *itoa*.

```
/* ------------------------------------------------------
     example_slib12.c

     Beispiel zur Verwendung von ltoa
   ------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  char string[25];
  long zahl = 123456789L;

  smallio_init();

  ltoa(zahl,string,10);
  printf("\n\rText: %s\n\r", zahl, string);
  while(1);
}
```

Ausgabe:

```
       Text: 123456789
```

malloc

dynamische Belegung von Speicherplatz.

```
       void *malloc(size_t size);
```

size gibt die Groesse des zu reservierenden Speichers in Bytes an. *malloc* liefert einen Zeiger auf den neu belegten Block bzw. den Wert NULL, wenn nicht genuegend Platz auf dem Heap vorhanden ist oder *size* den Wert 0 hat.

```
/* ------------------------------------------------------------
     example_slib13.c

     Beispiel zur Verwendung von malloc
   ------------------------------------------------------------ */

#include <string.h>
#include <stdlib.h>
#include "smallio.h"

int main(void)
{
   char *str;

   smallio_init();

   // Speicher fuer einen String reservieren (allozieren)
   if ((str = malloc(10)) == NULL)
   {
      printf("Nicht genug Speicher vorhanden...\n\r");
      while(1);              // Programm anhalten
   }
   strcpy(str, "Hallo");
   printf("Der Text ist: %s\n",str);
   free(str);                // Speicher freigeben
   while(1);
}
```

Ausgabe:

```
       Der Text ist: Hallo
```

qsort

sortiert die Elemente eines Arrays mit dem QuickSort Algorithmus.

```
       void qsort(void *base, size_t nelem,
                  size_t width, int(*fcmp)(const void *, const void *));
```

nelem = Elementenzahl des Arrays. base, width = Groesse eines Elements. Die Routine fcmp muss ein Vergleichsergebnis (< 0, 0, > 0) zurueckliefern.

Der Rueckgabewert von fcmp wird so interpretiert:

kleiner 0 bedeutet:	*elem1 < *elem2
gleich 0 bedeutet:	*elem1 == *elem2
groesser 0 bedeutet:	*elem1 > *elem2

```
/* ------------------------------------------------------
     example_slib14.c

     Beispiel zur Verwendung von qsort
   ------------------------------------------------------ */

#include <stdlib.h>
#include <string.h>
#include "smallio.h"

int sort_function( const void *a, const void *b)
{
   return( strcmp(a,b) );
}

int main(void)
{
   char list[5][8] = { "Maier", "Mueller", "Schulz ",
                               "Schmidt", "Albers" };
   int  x;

   smallio_init();

   qsort((void *)list, 5, sizeof(list[0]), sort_function);
   for (x = 0; x < 5; x++)
      printf("%s\n\r", list[x]);
   while(1);
}
```

Ausgabe:

```
       Albers
       Maier
       Mueller
       Schmidt
       Schulz
```

rand

liefert eine "Zufallszahl" zurueck.

```
       int rand(void);
```

Das Ergebnis liegt im Bereich von 0...*RAND_MAX*. *RAND_MAX* ist in stdlib.h definiert

- fuer 32-Bit Systeme mit dem Wert: 0x7FFFFFFF
- fuer 8-Bit Systeme mit dem Wert: 0x7FFF

```
/* ------------------------------------------------------
     example_slib15.c

     Beispiel zur Verwendung von rand
   ------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main(void)
{
  int i;

  smallio_init();

  srand(231);               // Initialisierung des Zufallszahlengenerators


  printf("10 Zufallszahlen von 0 bis 99\n\n\r");
  for(i=0; i<10; i++)
     printf("%d, ", rand() % 100);
```

```
  while(1);
}
```

Ausgabe:
```
       49, 64, 11, 3, 84, 36, 51, 3, 20, 89,
```

realloc

aendert die Groesse eines dynamisch belegten Speicherblocks.

```
       void *realloc(void *block, size_t size);
```

Liefert die neue (und evtl. veraenderte) Adresse des Blocks (mit *size* Bytes) zurueck bzw. den Wert NULL, wenn der Block aus irgendeinem Grund nicht modifiziert werden konnte (oder *size* den Wert 0 hat).

```
/* ------------------------------------------------------------
     example_slib16.c

     Beispiel zur Verwendung von realloc
   ------------------------------------------------------------ */

#include <stdlib.h>
#include <string.h>
#include "smallio.h"

int main(void)
{
  char *str;

  smallio_init();

  str = malloc(10);      // Speicher reservieren (allozieren)
  strcpy(str, "Hallo");
  printf("Text : %s\n\r  Addresse     : 0x%x\n\r", str, str);
  str = realloc(str, 20);
  printf("Text : %s\n\r  Neue Adresse : 0x%x\n\r", str, str);
  free(str);            // Speicher freigeben
  while(1);
}
```

Ausgabe:
```
       Text : Hallo
         Addresse     : 0x96f7008
       Text : Hallo
         Neue Adresse : 0x96f7420
```

srand

setzt einen Startwert fuer die Erzeugung von "Zufallszahlen".

```
       void srand(unsigned int seed);
```

Identische Startwerte (*seed*) erzeugen identische Zufallszahlenfolgen. Beispiel siehe *rand*.

utoa

konvertiert einen vorzeichenlosen Integer Wert in einen String.

```
char *utoa(unsigned int value, char *string, int radix);
```

Liefert den als *string* uebergebenen Zeiger zurueck. *radix* beinhaltet die Angabe, zu welcher Zahlenbasis die Konvertierung erfolgen soll.

Hinweis: *utoa* ist nur fuer den AVR-GCC verfuegbar. Fuer 32-Bit Systeme siehe auch *itoa*.

```
Beispiel:
/* ------------------------------------------------------------
     example_slib18.c

    Beispiel zur Verwendung von utoa
   ------------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main( void )
{
  unsigned int zahl = 52768;
  char string[6];

  smallio_init();

  utoa(zahl,string,10);
  printf("Ergebnis utoa: Zahl als Text = %s\n\r",string);
  while(1);
}
```

Ausgabe:

```
Ergebnis utoa: Zahl als Text = 52768
```

ultoa

konvertiert einen vorzeichenlosen long-Wert in einen String.

```
char *ultoa(unsigned long value, char *string, int radix);
```

Liefert den als string uebergebenen Zeiger zurueck. *radix* beinhaltet die Angabe, zu welcher Zahlenbasis die Konvertierung erfolgen soll.

Hinweis: *utoa* ist nur fuer den AVR-GCC verfuegbar. Fuer 32-Bit Systeme siehe auch *itoa*.

```
/* ------------------------------------------------------------
     example_slib19.c

    Beispiel zur Verwendung von ultoa
   ------------------------------------------------------------ */

#include <stdlib.h>
#include "smallio.h"

int main( void )
{
  unsigned long lzahl = 3123456789UL;
  char string[25];

  smallio_init();

  ultoa(lzahl,string,10);
  printf("Ergebnis ultoa: Zahl als Text = %s\n\r",string);
  while(1);
}
```

4.0

Beschreibung einzelner Funktionen

An dieser Stelle hier erfolgen Beschreibungen einzelner Funktionen, die verschiedenen Zwecken dienen koennen und auch aus nicht standardisierten, aber gaengigen Bibliotheken stammen koennen.

getopt (nur fuer Linux - Programme)

Kommandozeilenargumente auswerten

getop ist eine Funktion, die das Auslesen von Kommandozeilenparametern erleichtert und ist im weiteren Sinne ein Kommandozeilenparser.

Prototyp in:

```
#include <unistd.h>
```

Syntax:

```
int getopt(int argc, char * const argv[ ], const char *optstring);
```

getopt definiert globale Variable, die zur Auswertung der Kommandozeile benoetigt werden:

```
extern char *optarg;
extern int optind, opterr, optopt;
```

Programmaufruf eines Kommandozeilenprogramms

getopt liefert das aktuell durch *argc* angesprochene Optionszeichen zurueck. Ist das Optionszeichen "--" so liefert *getopt* den Wert -1 zurueck.

argc und *argv* sind die Variable, wie sie der Hauptfunktion *main* uebergeben worden sind (*argc* zaehlt die Anzahl der uebergebenen Parameter, *argv* nimmt die die einzelnen Argumentenstrings in einem Array auf).

optstring ist ein String, der alle erlaubten Optionszeichen enthaelt (case sensitive).

Einem Optionszeichen muss beim Aufruf ein Minuszeichen "-" vorangestellt werden. Folgt einem Optionszeichen ein Doppelpunkt, so erwartet die Optionsangabe ein Optionsargument.

Beispiel `"abc:d:"`

Wertet *getopt* einen Programmstart mit obigem Zeichen fuer *optstring* aus, so koennen dem aufgerufenem Programm die Parameter a,b,c und d mitgegeben werden, wobei fuer die Parameter c und d ein zusaetzliches Argument verlangt wird.

```
programmname -a -d test1
```

Wird eine Option, die ein Optionsargument erwartet (nachgestellter Doppelpunkt), ohne dieses Argument angegegeben, so liefert getopt ein Fragezeichen '?' zurueck. In diesem Falle ist in optopt die Option gespeichert, die das Argument erwaret hat.

Beispiel:

```
/* ------------------------------------------------------------
     example_getopt.c

     Beispiel zur Auswertung von Kommandozeilenparametern
   ------------------------------------------------------------ */

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char **argv)
{
  int aflag = 0;
  int bflag = 0;
  char *cvalue = NULL;
  char *fvalue = NULL;
  int index;
  int c;

  opterr = 0;

  while ((c = getopt (argc, argv, ":af:bc:")) != -1)
  {
    switch (c)
      {
      case 'a':
        aflag = 1;
        break;
      case 'f':
        fvalue = optarg;
        break;
      case 'b':
        bflag = 1;
        break;
      case 'c':
        cvalue = optarg;
        break;
      case '?':
        if ((optopt == 'c') || (optopt == 'f'))
        {
          printf (" Option -%c benoetigt Argument.\n", optopt);
        }
        else
        {
          printf (" Unbekannter Parameter -%c'.\n", optopt);
        }
        return 1;
      default:
        abort ();
      }
  }

  printf("\n");

  if (aflag)
    printf("\n Option -a ausgewaehlt");
  else
    printf("\n Option -a nicht ausgewaehlt");

  if (bflag)
    printf("\n Option -b ausgewaehlt");
  else
    printf("\n Option -b nicht ausgewaehlt");

  if (cvalue == NULL)
    printf("\n Keine Angabe fuer Option -c");
  else
    printf("\n Optionsargument fuer Option -c: %s", cvalue);
```

```
  if (fvalue == NULL)
    printf("\n Keine Angabe fuer Option -f");
  else
    printf("\n Optionsargument fuer Option -f: %s", fvalue);

  for (index = optind; index < argc; index++)
  {
    printf ("Argument ohne Optionszeichen: %s\n", argv[index]);
  }
  printf("\n");
  return 0;
}
```

printf - Derivate

printf gibt formatierte Ausgaben auf dem Standardausgabekanal aus. Bei Verwendung eines PC-Betriebssystems ist der Standardausgabekanal die Konsole. Bei Verwendung mittels eines Mikrocontrollers muss je nach benutzter Hardware ein Ausgabekanal geschaffen werden.

In Verbindung mit einem Mikrocontroller ist der Einsatz eines "full featured" printf abzuwaegen, da ein printf sehr speicherumfaenglich ist.

Ein im Leistungsumfang reduzierter, aber in sehr vielen Faellen ausreichender Ersatz fuer printf ist *my_printf*, der auch gut in einem Mikrocontroller integrierbar ist.

printf

Syntax von printf :	`int printf(const char *format [, argument, ...]);`
Prototypdeklaration in :	`stdio.h`

printf unterscheidet sich im Gegensatz zu den meisten Funkktionen in C darin, dass fuer diese Funktion eine unterschiedliche (variable) Anzahl Argumente uebergeben werden koennen. Bei fehlerfreier Ausfuehrung liefert printf die Anzahl insgesamt ausgegebener Zeichen als Funktionsargument zurueck.

`const char *format` enthaelt den zwingend notwendigen String, der ausgegeben wird. Alle eventuell nachfolgende Argumente sind optional.

Beispiel:

```
printf("Hallo Welt");
```

Innerhalb des Argumentstrings *format werden Umwandlungszeichen (engl. conversion modifier) fuer weitere Parameter eingesetzt, die eine Formatierung der Ausgabe ermoeglichen. Ausserdem kann dieser String Escape-Sequenzen enthalten, die ebenfalls der formatierten Ausgabe dienen.

Umwandlungszeichen %

Prinzipiell funktionieren die Umwandlungszeichen nach folgendem Schema:

Ein Umwandlungszeichen wird mit dem %-Zeichen (Ascii-Code 37) als Einleitungszeichen eingeleitet.

Direkt anschliessend an dieses Einleitungszeichens koennen folgen:

- ein Flagzeichen (erlaubte Zeichen: SPACE,-,+,#,0)
- die Feldbreite
- durch einen Punkt getrennt die Anzahl der Nachkommastellen
- an letzter Stelle das Umwandlungszeichen (und somit den Ausgabetyp)

Einfaches Beispiel nur mit Umwandlungszeichen (ohne Flags, ohne Feldbreite und ohne Punkt):

```
printf("1. Zahl= %d, 2. Zahl= %d, 3. Zahl= %d", 34, 65, 99);
```

Erlaeuterung:

Da im Ausgabestring insgesamt 3 Umwandlungszeichen enthalten sind (3 mal %d), werden diese %d durch die jeweils dem String folgenden Parametern in der Reihenfolge ihres auftretens ersetzt. D.h. anstelle des ersten im String enthaltenen %d wird der erste Parameter ausgegeben. Im vorliegenden Falle wird also in den Text die Zahl 34 mit dezimaler Ausgabe eingefuegt. Anstelle des zweiten Vorkommens von %d wird 65, anstelle des dritten Vorkommens 99 ausgegeben.

Die Ausgabe lautet:

```
1. Zahl= 34, 2. Zahl= 65, 3. Zahl= 99
```

Verfuegbare Umwandlungszeichen sind:

Zeichen	Datentyp und Darstellung
%d	Integer als dezimale Zahl ausgeben
%i	Integer als dezimale Zahl ausgeben
%c	Ausgabe eines char als (Ascii)-Zeichen
%e	Ausgabe von double Gleitkommazahl im wissenschaftlichen Format (bspw.: 3.342409e+3)
%E	Ausgabe von double Gleitkommazahl im wissenschaftlichen Format (bspw.: 3.342409E+3)
%f	Ausgabe von double als Gleitkommadarstellung (bsw.: 33424.092342)
%o	oktale Ausgabe von int
%p	Ausgabe der Adresse eines Zeigers
%s	Zeichenkette (String) ausgeben
%u	Ausgabe eines vorzeichenlosen Integers
%x	Integer als hexadezimale Zahl ausgeben (bspw.: 4f3a)
%X	Integer als hexadezimale Zahl ausgeben (bspw.: 4F3A)
%%	Ausgabe des Prozentzeichens

Beispiele

```
printf("int             : %d \n", 65);
printf("negative int    : %d \n",-65);
printf("Gleitkomma      : %.6f \n", M_PI);
printf("Asciizeichen    : %c \n", 'z');
printf("Zeichenkette    : %s \n", "AbCdEf");
printf("65 in Oktal     : %o\n", 65);
printf("65 in Hex       : %x\n", 65);
printf("Prozentzeichen  : %%\n");
```

Ausgabe:

```
int             : 65
negative int    : -65
Gleitkomma      : 3.141593
Asciizeichen    : z
Zeichenkette    : AbCdEf
65 in Oktal     : 101
65 in Hex       : 41
Prozentzeichen  : %
```

Flag / Flagzeichen (optional)

Direkt nach dem %-Einleitungszeichen kann ein sogenanntes Flag-Zeichen (Kennzeichnung) angegeben werden. Dieses Zeichen beeinflusst das Erscheinungsbild auf dem Ausgabekanal.

Zeichen	Bedeutung
-	Text wird linksbuendig ausgerichtet
+	auch bei einem positiven Zahlenwert wird ein Vorzeichen mit ausgegeben
SPACE	Leerzeichen, ein Leerzeichen wird ausgegeben, wenn der Wert positiv ist
#	Abhaengig vom Format und Betriebssystem. In Verbindung mit hexadezimaler Ausgabe wird der Ausgabe "0x" vorangestellt.
0	die Auffuellung erfolgt mit Nullen anstelle von Leerzeichen

Beispiel

```
printf("Hex-Zahl: %#x\n", 76);
```

Ausgabe

```
Hex-Zahl: 0x4c
```

Feldbreite und Nachkommastellen

Nach den Flags koennen Angaben zur Gesamtzeichenanzahl der Ausgabe (Feldbreite) und / oder Angabe zur Anzahl der Nachkommastellen gemacht werden.

Wird fuer die Gesamtbreite eine hoehere Anzahl Stellen angegeben, als die auszugebende Zahl selbst besitzt, so wird die Ausgabe linksseitig um Leerstellen aufgefuellt, so dass die geforderte Gesamtbreite erfuellt ist.

Ist als Flag die Ziffer "0" angegeben, so wird die Gesamtausgabe mit Nullen anstelle von Leerzeichen aufgefuellt.

Soll eine Gleitkommazahl ausgegeben werden (float oder double), so kann die Feldbreite durch ein "." Zeichen auch den Nachkommaanteil beinhalten. Hierbei gibt die Angabe fuer die Feldbreite die Gesamtzeichenzahl der Ausgabe an, die Ziffernangabe nach dem Punkt die Anzahl Stellen des Nachkommateils. Ueberschreitet die Anzahl der Zeichen fuer den Nachkommaanteil die Angabe fuer die Gesamtzeichenanzahl, so werden jedoch die Zeichen fuer den Nachkommateil garantiert ausgegeben, sodass die Gesamtzeichenanzahl die angegebene Feldbreite ueberschreitet.

Der Punkt, der den Vorkommaanteil vom Nachkommateil trennt zaehlt hierbei als Zeichen fuer die Gesamtbreite. Es kann auch nur die Anzahl der Nachkommastellen ohne fuehrende Gesamtanzahl angegeben werden.

Beispiel:

```
printf("\n1234567890");
printf("\n%10d",23);
printf("\n%010d",65);
printf("\n%10.4f",M_PI);
printf("\n%.3f",M_PI);
printf("\n");
```

Ausgabe:

```
1234567890
        23
0000000065
    3.1416
3.142
```

Escape-Sequenzen / Steuerzeichen

Der Ausgabestring kann Steuerzeichen (Escape-Sequenzen) enthalten. Das Ausgabeverhalten haengt vom verwendeten Terminal bzw. der verwendeten Terminalemulation ab (die Konsole von Windows und Linux verhalten sich leider abweichend voneinander).

Ein Steuerzeichen wird durch einen Schraegstrich nach rechts unten (engl. backslash) "angemeldet".

Zeichen	Bedeutung
\n	(new line) fuegt eine neue Zeile unterhalb der aktuellen Zeile ein. Hierbei ist das Ausgabeverhalten von der Terminalemulation abhaengig. Die Linuxkonsole generiert bei einem "new line" Kommando auch ein Bewegen des Cursors an den Zeilenanfang (was einem "carriage return" entspricht). Bei der Windowskonsole bleibt der Cursor in der X-Position verharren, an der er sich befindet. Ein "/n" unter Linux bewirkt somit dasselbe wie ein "/n/r" unter Windows.
\r	(carriage return) bewirkt ein setzen des Cursors an den Zeilenanfang.
\t	setzt den Cursor auf die naechste horizontale Tabulatorposition.
\b	setzt den Cursor ein Zeichen nach links OHNE das aktuelle Zeichen hierbei zu loeschen.
\f	setzt den Cursor auf die Startposition der naechsten Seite
\v	setzt den Cursor auf die naechste vertikale Tabulatorposition
\0	markiert das Ende einer Textzeile (nachfolgende Zeichen werden nicht mehr interpretiert und auch nicht ausgegeben)

Beispiel (unter Verwendung von \n \r \t und Flags):

```
// Hinweis: \r Escape-Sequenz waere fuer Programmlauf unter Linux nicht notwendig

printf("\n\rQuadratzahlen\t Wurzel\t\t  Reziprokenwerte");
printf("\n\r-------------------------------------------------\n\r");
for (i= 1; i< 17; i++)
{
  printf("\n\r %2d * %2d= %4d\t sqrt(%2d)= %4.2f\t  1/%2d= %.4f",         \
                i,    i,  i*i,    i,   sqrt((float)i), i,  1.0/(float)i);
}
```

Ausgabe:

```
   Quadratzahlen    Wurzel           Reziprokenwerte
   -------------------------------------------------
      1 *  1=    1   sqrt( 1)= 1.00   1/ 1= 1.0000
      2 *  2=    4   sqrt( 2)= 1.41   1/ 2= 0.5000
      3 *  3=    9   sqrt( 3)= 1.73   1/ 3= 0.3333
      4 *  4=   16   sqrt( 4)= 2.00   1/ 4= 0.2500
      5 *  5=   25   sqrt( 5)= 2.24   1/ 5= 0.2000
      6 *  6=   36   sqrt( 6)= 2.45   1/ 6= 0.1667
      7 *  7=   49   sqrt( 7)= 2.65   1/ 7= 0.1429
      8 *  8=   64   sqrt( 8)= 2.83   1/ 8= 0.1250
      9 *  9=   81   sqrt( 9)= 3.00   1/ 9= 0.1111
     10 * 10=  100   sqrt(10)= 3.16   1/10= 0.1000
     11 * 11=  121   sqrt(11)= 3.32   1/11= 0.0909
     12 * 12=  144   sqrt(12)= 3.46   1/12= 0.0833
     13 * 13=  169   sqrt(13)= 3.61   1/13= 0.0769
     14 * 14=  196   sqrt(14)= 3.74   1/14= 0.0714
     15 * 15=  225   sqrt(15)= 3.87   1/15= 0.0667
     16 * 16=  256   sqrt(16)= 4.00   1/16= 0.0625
```

sprintf

sprintf verhaelt sich genau wie *printf* mit dem Unterschied, dass die Ausgabe nicht auf dem Standardausgabekanal, sondern in einen String (d.h. also ein Char-Array) erfolgt.

```
Syntax      : int sprintf(char *buffer, const char *format [, argument, ...]);
Prototyp in : stdio.h
```

Saemtliche Angaben fuer Umwandlungszeichen, Flags sowie Feldbreitenbezeichner sind auch fuer sprintf verfuegbar.

Beispiel:

```
char  txtbuffer[100];

sprintf(txtbuffer, "\n%.3f * sqrt(%.1f) = %.3f\n", M_PI, 2.0, M_PI * sqrt(2.0));
printf("%s", txtbuffer);
```

Ausgabe:

```
3.142 * sqrt(2.0) = 4.443
```

my_printf

my_printf ist ein in der Funktionalitaet reduzierter Ersatz fuer *printf,* speziell zur Benutzung in Verbindung mit Mikrocontrollern. Er macht dann Sinn, wenn es darum geht (Flash)speicher zu sparen und ermoeglicht den Einsatz auf Mikrocontrollersystemen ab 2 kByte Flashspeicher.

Ein vom Standard abweichendes Umwandlungszeichen %k ermoeglicht es, Pseudo-Kommazahlen auszugeben.

my_printf existiert im Verzeichnis *../src* in den Quelltextvarianten:

- my_printf.c fuer 8-Bit, 32-Bit Systeme und PC-Systeme
- my_printf32.c nur fuer 8-Bit Systeme
- my_printf_float.c fuer 8-Bit, 32-Bit Systeme und PC-Systeme

```
Syntax von printf  :  void my_printf(const char *s,...);
Prototyp in :  ../include/my_printf.h
```

Um my_printf innerhalb eines Programms zur Verfuegung zu haben, muss entsprechend der gewuenschten Funktionalitaet die benoetigte Datei compiliert und hinzugelinkt werden. Bei Verwendung der hier benutzten Makefile Vorlagen wird dies erreicht durch Angaben (innerhalb des Makefiles) von:

```
SRCS += ../src/my_printf.c
```

Die zu den verschiedenen *my_printf* gehoerende Header-Datei ist fuer alle Varianten unabhaengig der Funktionalitaet gleich:

```
#include "my_printf.h"
```

my_printf benoetigt zur Ausgabe irgendwo in den zu einem Programmprojekt gehoerenden Dateien eine Funktion namens:

```
void my_putchar(char ch);
```

my_printf bedient sich dieser Funktion zur Zeichenausgabe. Soll bspw. ein Programm die Ausgaben auf der seriellen Schnittstelle vornehmen, existiert wahrscheinliche eine Funktion aehnlich

```
void uart_putchar(char ch);
```

Um nun diese Ausgabefunktion nutzen zu koennen koennte ein einfaches Programm folgendermassen aussehen:

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

#include "uart_all.h"
#include "my_printf.h"

#define printf    my_printf

/* ------------------------------------------------------
   my_putchar

   wird von my-printf aufgerufen und hier muss
   eine Zeichenausgabefunktion angegeben sein, auf das
   printf dann schreibt !
   ------------------------------------------------------ */
void my_putchar(char ch)
{
  uart_putchar(ch);
}

int main(void)
{
  uart_init();
  printf("\n\r Hallo Welt\n\r");
  while(1);
}
```

Im gezeigten Beispiel wird die Zeile

```
#define printf my_printf
```

dazu genutzt, um im gesamten Programm *printf* anstelle von *my_printf* schreiben und somit bereits bestehende Programme mit *my_printf* (anstelle von *printf* aus *stdio.h*) nutzen zu koennen. Mit dieser Angabe jedoch ist dann eine eventuell vorhandene originale *printf* Funktion nicht mehr verfuegbar.

Umwandlungszeichen von my_printf

Zeichen	Datentyp und Darstellung
%d	Integer als dezimale Zahl ausgeben int ist 16-Bit Wert auf 8-Bit Systemen int ist 32-Bit Wert auf 32-Bit Systemen
%l	nur bei printf32.c: 32-Bit Integerwert ausgeben (verfuegbar fuer AVR, STM8 und MCS-51)
%c	Ausgabe eines char als (Ascii)-Zeichen
%f	nur bei my_printf_float: Ausgabe eines (float) Gleitkommawertes. Ein Punkt zur Angabe der Nachkommastellen wird interpretiert
%k	(Pseudo-Kommazahl) Integerwert mit eingesetztem Kommapunkt ausgeben, auf 32-Bit Systemen oder bei my_printf32 als 32-Bit Wert, ansonsten als 16-Bit Wert. Mit dem Wert in globaler Variable printfkomma wird angegeben, an welcher Position ein Dezimalpunkt ausgegeben wird.
%s	Zeichenkette (String) ausgeben
%x	Integer als hexadezimale Zahl ausgeben
%%	Ausgabe des Prozentzeichens

Beispiel:

```
#define printf    my_printf

int a,a2,b,c;

printf("\n\r ASCII-Zeichen '%c' = dezimal %d = hexadezimal 0x%x\n\r", 'M', 'M', 'M');

a= 42; b= 13;

// Variable a um eine Zehnerstelle nach links,
// um spaeter eine Nachkommastelle anzeigen zu koennen
a2= a*100;
c= a2 / b;

printfkomma= 2;
printf(" %d / %d = %k", a, b, c);
```

Ausgabe:

```
ASCII-Zeichen 'M' = dezimal 77 = hexadezimal 0x4D
42 / 13 = 3.23
```