Informatikwerkstatt, Foliensatz 3
C-Programmierung

G. Kemnitz

Institut für Informatik, Technische Universität Clausthal

10. November 2014

Inhalt des Foliensatzes

- Variablen
- Typumwandlung
- Funktionen
- Fallunterscheidungen
- Schleifen
- Zeichenkettenfunktionen für das Finalprojekt

Variablen

Variablen

- Variablen sind Symbole für Adressen von Speicherplätzen, die beschrieben und gelesen werden können.
- Eine Variablenvereinbarung definiert Typ (z.B. uint8_t), Namen (z.B. dat) und optional einen Anfangswert (z.B. 45):

```
uint8_t dat = 45;
```

- Der Typ legt fest, wie viele Bytes zur Variablen gehören (z.B. 1 Byte) und was die Bytes darstellen (z.B. eine Zahl ohne Vorzeichen im Bereich von 0 bis 255).

C unterscheidet globale und lokale Variablen:

- Global: Außerhalb einer Funktion vereinbart. Feste Adressen. Existieren während der gesamten Programmlaufzeit.
- Lokal: Innerhalb einer Funktion vereinbart. Speicherplatz wird erst bei Funktionsaufruf auf dem sog. Stack reserviert. Existieren nur während der Funktionsabarbeitung.

Im Debugger können Variableninhalte im Schrittbetrieb oder an Unterbrechungspunkten gelesen und verändert werden. Dafür sind folgende Fenster in Atmel-Studio zu öffnen:

- IO View: Register mit Spezialfunktionen und ihre einzelnen Bits (Ports, USART, ...). Fenster öffnen:

  `Debug > Windows > IO-View`

- Locals: Innerhalb einer Funktion existierende lokale Variablen. Fenster öffnen:

  `Debug > Windows > Locals`

- Watch 1 bis 4: Globale Variablen. Fenster öffnen:

  `Debug > Windows > Watch > Watch1`

  Die Namen der zu beobachtenden Variablen eingeben.

- Memory 1 bis 4, Einstellung »data REGISTERS«: Ausschnitte aus dem Datenspeicher als Bytetabelle. Öffnen:

  `Debug > Windows > Memory`

Beispielprogramm

Zu vereinbarende globale Variablen:

- gi32: für 4-Byte mit Vorzeichen (WB: -2^{31} bis $2^{31}-1$)
- gu8: für 1-Byte ohne Vorzeichen (WB: 0 bis 2^8-1)

Zu vereinbarende lokale Variablen:

- lu8: für 1-Byte ohne Vorzeichen (WB: 0 bis 2^8-1)
- li32: für 4-Byte mit Vorzeichen (WB: -2^{31} bis $2^{31}-1$)
- lpu8: Zeiger auf Variablen vom Typ uint8_t[1]

Anfangsinitialisierung der lokalen Variablen:

```
uint8_t  lu8  = 0x2D;       // dezimal 2·16+13 = 45
int32_t  li32 = 0x023...F4; // 4 Byte, 8 Hex.-Ziffern
uint8_t *lpu8[2] = &gu8;    // Anfangswert Adresse von gu8
```

*ptr: Wert der Variablen mit der Adresse »ptr«.
&v Adresse des Speicherplatzes der Variablen v

[1] Datenadressen sind in unserem Prozessor 16 Bit groß.
[2] Der Wert, auf den lpu8 zeigt, hat den Typ uint8_t.

Aufgabe 3.1: Projekt zum Experimentieren

- Legen Sie ein neues Projekt »TestVariablen« an. Geben Sie das Beispielprogramm auf der nachfolgenden Folie ein[3].
- Stellen Sie Compiler-Optimierung auf »O0«[4].
- Wählen Sie als Tool den Simulator.
- Debugger mit Halt vor der ersten Anweisung starten.
- Ein wie auf der nachfolgenden Folie Watch-, das Local- und zwei Memory-Fenster öffnen und einstellen.

[3] Zeilennummern einschalten: Tools > Options > Text Editor > All languages > General: Display on line numbers√

[4] Der Compiler verwendet sonst Register als Speicherplätze für Variablen und optimiert Anweisungen weg, deren Ergebnisse nicht weiter verarbeitet werden. Im Beispiel wären das alle Anweisungen. Wegoptimierte Anweisungen sind daran zu erkennen, dass der Debugger im Schrittbetrieb nicht vor ihnen anhält bzw. das vor ihnen gesetzte Unterbrechungspunkte im Debug-Modus verschwinden.

- Stopp nach Anfangsinitialisierung der »Locals«

```
 9  #include <avr/io.h>
10
11  // globale Variablen
12  int32_t gi32;
13  uint8_t gu8;
14
15  int main(void)
16  {
17    // lokale Variablen
18    uint8_t lu8 = 0x2D;
19    int32_t li32=0x23451F4;
20    uint8_t *lpu8 = &gu8;     // Variable für Adressen
21
22    gi32  = li32 + 1;
23    *lpu8 = lu8 - 4;
24    lpu8  = &lu8;
25    *lpu8 = 0xA5;
26    lu8 = 23;
27  }
```

Watch 1

Name	Value	Type
gu8	0x00	uint8_t{data}@0x0100
gi32	0x00000000	int32_t{data}@0x0101

Locals

Name	Value	Type
lu8	0x2d	uint8_t{data}@0x10fb
li32	0x023451f4	int32_t{data}@0x10f5
⊞ lpu8	0x0100	uint8_t*{data}@0x10f9

Memory 1

data 0x10F4	00	f4	51	34
data 0x10F8	02	00	01	2d
data 0x10FC	10	ff	80	61

Memory 2

data 0x0100	00	00	00	00
data 0x0104	00	00	00	00
data 0x0108	00	00	00	00

- Memory 1: Adressbereich der lokalen Variablen einstellen.
- Memory 2: Adressbereich der globalen Variablen einstellen.

- Ein Schritt weiter nach Abarbeitung »gi32 = gi32 + 1«:

```
 9  #include <avr/io.h>
10
11  // globale Variablen
12  int32_t gi32;
13  uint8_t gu8;
14
15  int main(void)
16  {
17    // lokale Variablen
18    uint8_t lu8 = 0x2D;
19    int32_t li32=0x23451F4;
20    uint8_t *lpu8 = &gu8;
21
22    gi32  = li32 + 1;
23    *lpu8 = lu8 - 4;
24    lpu8  = &lu8;
25    *lpu8 = 0xA5;
26    lu8 = 23;
  }
```

Watch 1

Name	Value	Type
gu8	0x00	uint8_t{data}@0x0100
gi32	0x023451f5	int32_t{data}@0x0101

Locals

Name	Value	Type
lu8	0x2d	uint8_t{data}@0x10fb
li32	0x023451f4	int32_t{data}@0x10f5
lpu8	0x0100	uint8_t*{data}@0x10f9

// Variable für Adressen

Memory 1

data 0x10F4	00	f4	51	34
data 0x10F8	02	00	01	2d
data 0x10FC	10	ff	80	61

Memory 2

data 0x0100	00	f5	51	34
data 0x0104	02	00	00	00
data 0x0108	00	00	00	00

Auf Adresse 0x101 bis 0x104 erscheint der neue Werte:

$$0x023451F4 + 1 = 0x023451F5$$

- Nach Zuweisung des Inhaltes von lu8 (0x2d) minus 4 an die Adresse in lpu8 (0x100, d.h. die Variable gu8):

```
 9  #include <avr/io.h>
10
11  // globale Variablen
12  int32_t gi32;
13  uint8_t gu8;
14
15  int main(void)
16  {
17    // lokale Variablen
18    uint8_t lu8 = 0x2D;
19    int32_t li32=0x23451F4;
20    uint8_t *lpu8 = &gu8;     // Variable für Adressen
21
22    gi32  = li32 + 1;
23    *lpu8 = lu8 - 4;
24    lpu8  = &lu8;
25    *lpu8 = 0xA5;
26    lu8 = 23;
  }
```

Watch 1

Name	Value	Type
gu8	0x29	uint8_t{data}@0x0100
gi32	0x023451f5	int32_t{data}@0x0101

Locals

Name	Value	Type
lu8	0x2d	uint8_t{data}@0x10fb
li32	0x023451f4	int32_t{data}@0x10f5
lpu8	0x0100	uint8_t*{data}@0x10f9

Memory 1

data 0x10F4	00	f4	51	34
data 0x10F8	02	00	01	2d
data 0x10FC	10	ff	80	61

Memory 2

data 0x0100	29	f5	51	34
data 0x0104	02	00	00	00
data 0x0108	00	00	00	00

Auf Adresse 0x100 erscheint der neue Werte $0x2D - 4 = 0x29$.

- Ein Schritt weiter übernimmt lpu8 die Adresse von lu8 (0x10FB)

```
 9  #include <avr/io.h>
10
11  // globale Variablen
12  int32_t gi32;
13  uint8_t gu8;
14
15  int main(void)
16  {
17    // lokale Variablen
18    uint8_t lu8 = 0x2D;
19    int32_t li32=0x23451F4;
20    uint8_t *lpu8 = &gu8;     // Variable für Adressen
21
22    gi32  = li32 + 1;
23    *lpu8 = lu8 - 4;
24    lpu8  = &lu8;
25    *lpu8 = 0xA5;
26    lu8 = 23;
  }
```

Watch 1

Name	Value	Type
gu8	0x29	uint8_t{data}@0x0100
gi32	0x023451f5	int32_t{data}@0x0101

Locals

Name	Value	Type
lu8	0x2d	uint8_t{data}@0x10fb
li32	0x023451f4	int32_t{data}@0x10f5
lpu8	0x10fb	uint8_t*{data}@0x10f9

Memory 1

data 0x10F4	00	f4	51	34
data 0x10F8	02	fb	10	2d
data 0x10FC	10	ff	80	61

Memory 2

data 0x0100	29	f5	51	34
data 0x0104	02	00	00	00
data 0x0108	00	00	00	00

Auf der Adresse von lpu8 (0x10F9 bis 0x10FA) erscheint der neue Wert 0x10FB.

Der in lpu8 gespeicherten Adresse (0x10FB, Variable lu8) wird der Wert 0xA5 zugewiesen.

```
 9  #include <avr/io.h>
10
11  // globale Variablen
12  int32_t gi32;
13  uint8_t gu8;
14
15  int main(void)
16  {
17    // lokale Variablen
18    uint8_t lu8 = 0x2D;
19    int32_t li32=0x23451F4;
20    uint8_t *lpu8 = &gu8;
21
22    gi32  = li32 + 1;
23    *lpu8 = lu8 - 4;
24    lpu8  = &lu8;
25    *lpu8 = 0xA5;
26  } lu8 = 23;
```

Watch 1

Name	Value	Type
gu8	0x29	uint8_t{data}@0x0100
gi32	0x023451f5	int32_t{data}@0x0101

Locals

Name	Value	Type
lu8	0xa5	uint8_t{data}@0x10fb (
li32	0x023451f4	int32_t{data}@0x10f5 (
⊞ lpu8	0x10fb	uint8_t*{data}@0x10f9

// Variable für Adressen

Memory 1

data 0x10F4	00	f4	51	34
data 0x10F8	02	fb	10	a5
data 0x10FC	10	ff	80	61

Memory 2

data 0x0100	29	f5	51	34
data 0x0104	02	00	00	00
data 0x0108	00	00	00	00

Der Variablen lu8 wird über ihren Namen der Wert dezimal 23 (hexadezimal 0x17) zugewiesen.

```
 9  #include <avr/io.h>
10
11  // globale Variablen
12  int32_t gi32;
13  uint8_t gu8;
14
15  int main(void)
16  {
17    // lokale Variablen
18    uint8_t lu8 = 0x2D;
19    int32_t li32=0x23451F4;
20    uint8_t *lpu8 = &gu8;     // Variable für Adressen
21
22    gi32  = li32 + 1;
23    *lpu8 = lu8 - 4;
24    lpu8  = &lu8;
25    *lpu8 = 0xA5;
26    lu8 = 23;
27  }
```

Watch 1

Name	Value	Type
gu8	0x29	uint8_t{data}@0x0100
gi32	0x023451f5	int32_t{data}@0x0101

Locals

Name	Value	Type
lu8	0x17	uint8_t{data}@0x10fb
li32	0x023451f4	int32_t{data}@0x10f5
lpu8	0x10fb	uint8_t*{data}@0x10f9

Memory 1

data 0x10F4	00	f4	51	34
data 0x10F8	02	fb	10	17
data 0x10FC	10	ff	80	61

Memory 2

data 0x0100	29	f5	51	34
data 0x0104	02	00	00	00
data 0x0108	00	00	00	00

Aufgabe 3.2: Andere Variablenzuweisungen

Modifizieren Sie das Beispielprogramm:

- andere Variablen mit anderen Typen
- Zuweisung anderer Werte.
- Benutzen Sie insbesondere auch Zeiger, denen Sie nacheinander die Adressen unterschiedlicher Variablen zuweisen und diese Variablen damit adressieren.

Beobachten Sie in derselben Weise die Werte und Adressen der Variablen.

Weitere Datentypen und ihre Wertebereiche:

`int8_t:` 1 Byte mit Vorzeichen, WB: -2^7 bis $2^7 - 1$
`int16_t:` 2 Byte mit Vorzeichen, WB: -2^{15} bis $2^{15} - 1$
`uint16_t:` 2 Byte ohne Vorzeichen, WB: 0 bis $2^{16} - 1$
`uint32_t:` 4 Byte ohne Vorzeichen, WB: 0 bis $2^{32} - 1$

Typumwandlung

Typumwandlung (typecast)

Wenn einer Variablen ein Wert mit einem anderen Typ zugewiesen wird, sollte der Übersetzer eine Warnung oder eine Fehlermeldung ausgeben:

```
uint16_t a;
int16_t  b;
a = b;        // sollte mindestens Warnung verursachen
```

Es gibt Situationen, in denen typfremde Zuweisungen gewollt und richtig sind. Dann ist dem zugewiesenen Ausdruck geklammert der Typ des Zuweisungsziels, im Beispiel (uint16_t) voranzustellen:

```
a = (uint16_t)b;
```

Nur so sollte die Zuweisung einer vorzeichenbehafteten an eine vorzeichenfreie Variable erlaubt sein.

Was macht Atmel-Studio?

```
 9  #include <avr/io.h>
10  char c, *c_ptr;     // char kann int8_t oder uint8_t sein
11                      // in der Tool-Chain als uint8_t definiert
12  uint8_t u, *u_ptr;
13  int8_t i, *i_ptr;
14
15  int main(void)
16  {
17      c=u;                // laut Toolchain korrekt
18      c=i;                // laut Toolchain falsch, keine Warnung
19      u = i;              // unzulässig, keine Warnung
20      c_ptr = &c;         // zulässig, keine Warnung
21      c_ptr = &u;         // laut Toolchain korrekt, Warnung
22      c_ptr = (char*)&u;  // mit Typecast, keine Warnung
23      c_ptr = &i;         // laut Toolchain falsch, Warnung
24      c_ptr = (char*)&i;  // mit Typecast, keine Warnung
25  }
```

Selbst mit »Toolchain > Compiler > Warnings: Pedantic √« sind die Warnungen weder vollständig noch schlüssig.

Aufgabe 3.3: Fehlverhalten provozieren

Anregung zum Experimentieren:

```
uint8_t a; int8_t b;
a = 56;
b = a;   // Kommt die 56 richtig an?
a = 200;
b = a;   // b ≤ 127. Was wird aus 200?
b = 200; // Akzeptiert das der Compiler?
b = -10;
a = b;   // a ≥ 0. Was wird aus −10?
```

- Was erlaubt der Compiler, wofür gibt er Warnungen aus?
- Was verursacht bei der Abarbeitung Probleme?
- Unter welchen Bedingungen arbeiten die Programme trotzdem richtig?

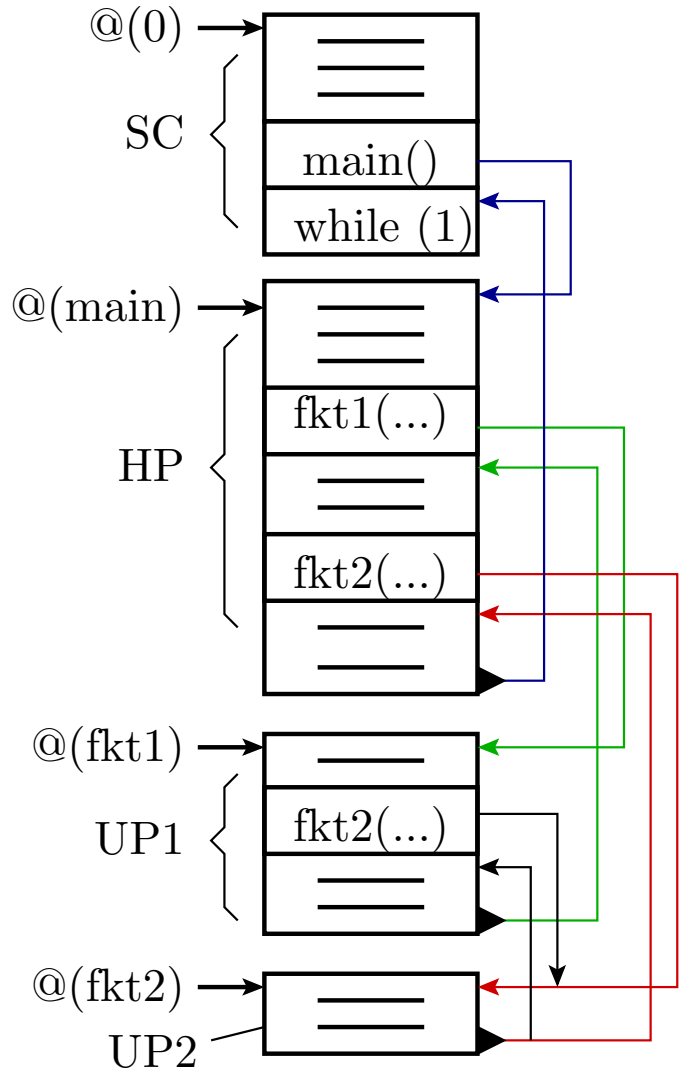
Funktionen

Funktionen

Funktionen sind Programmbausteine,

- die als Befehlsfolge vom Programmiergerät in den Befehlsspeicher (Flash) geschrieben werden.
- Start über Aufruf (Sprung zur Startadresse).
- Wenn fertig, Rücksprung.

SC	automatisch eingefügter Startcode
HP	Hauptprogramm
UPi	Unterprogramm i
@(0)	Adresse 0, Startadresse Mikrorechner nach Neuprogrammierung, Einschalten,
@(...)	Startadresse einer Funktion
▶	Rücksprung
***(...)	Aufruf von *** mit den Parametern ...
while (1)	Endlosschleife
═	andere Anweisungen



- Der Programmtext definiert nur den Funktionsnamen (z.B. BerechneSumme) und die auszuführenden Anweisungen (z.B. return a+b). Den Rest macht der Compiler:

```
uint8_t BerechneSumme(uint8_t a, b){
 return a+b;
}
```

- Der Typ (uint8_t) ist für den Rückgabewert.
- Die Parameterliste (uint8_t a, b) definiert lokale Variablen für die Übergabe von Aufrufwerten.

- Aufruf der Beispielfunktion:

```
uint8_t Summand1, Summand2, Summe;
...
Summe = BerechneSumme(Summand1, Summand2);
```

- In vorherigen Beispielen gab es nur die Funktion main():

```
int main(void){
... // Anweisungen
}
```

Funktionen für die serielle Ein- und Ausgabe

Das Echo-Programm von Foliensatz 2 enthält drei auch für andere Programme interessante Funktionsbausteine:

```
#include <avr/io.h>
int main(void)  {
  UBRR0H = 0; UBRR0L = 49;          [1]
  UCSR0B = 0b00011000;
  UCSR0C = 0b00111110;
  DDRE = 0xF0;

  uint8_t daten, Ct=0;
  while(1) {
    while (!(UCSR0A & (1<<RXC0))){} // warte auf Receive-Bit   [2]
    daten = UDR0;                   // empfangene Daten lesen
    while (!(UCSR0A & (1<<UDRE0))){}// warten bis Sendepuffer frei   [3]
    UDR0 = daten+1;                 // Daten+1 senden
    PORTE = Ct<<4; Ct++;            // Zähler ausgeben und erhöhen
  }
}
```

1 initUSART0() Initialisierung

2 getBytePC() Empfang eines Bytes

3 sendBytePC() Versenden eines Bytes

Die drei Funktionseinheiten als Unterprogramme

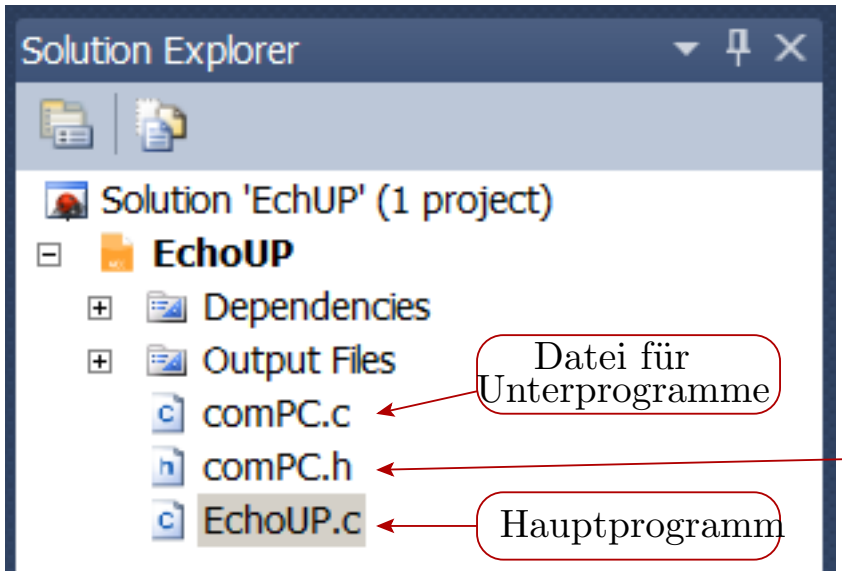
```
#include <avr/io.h>

void initUSART0(){                      // Initialisierung von
  UBRR0H = 0; UBRR0L = 49;              // USART0 für 8 Daten-
  UCSR0B = 0b00011000;                  // 1 Stoppbit, keine
  UCSR0C = 0b00000110;                  // Parität, 9600 Baud
}

uint8_t getBytePC(){
  while (!(UCSR0A & (1<<RXC0))){} // warte auf Empfang
  return UDR0;                          // Rückgabe Empfangsdaten
}

void sendBytePC(uint8_t dat){
  while (!(UCSR0A & (1<<UDRE0))){}// warten Sendepuffer frei
  UDR0 = dat;                           // Daten senden
}
```

Nachnutzbare Unterprogramme schreibt man meist in separate C-Dateien (im Bild comPC.c). Die Einbindung von Funktionen aus einer separaten Datei verlangt eine Header-Datei mit Funktionsdefinitionen (im Bild comPC.h).



Inhalt der Header-Datei

```
 9 #ifndef COMPC_H_
10 #define COMPC_H_
11
12   void initUSART0();
13   uint8_t getBytePC();
14   void sendBytePC(uint8_t dat);
15
16 #endif /* COMPC_H_ */
```

- Neue C-Datei erzeugen: Rechtsklick auf Projektnamen (EchoUP) > Add > New Item > C File, Name: comPC.c.
- Neue Header-Datei erzeugen: Rechtsklick auf Projektnamen (EchoUP) > Add > New Item > Include File, Name: comPC.h.

Im Hauptprogramm Header mit einbinden. Dann können die drei Teilfunktionen durch Funktionsaufrufe ersetzt werden.

```
#include <avr/io.h>
#include "comPC.h"              // den neuen Header mit einbinden

int main(void)
{
  initUSART0();                 // USART initialisieren
  uint8_t daten;
    while(1){
      daten = getBytePC();      // auf ein Byte warten
      sendBytePC(daten);        // empfangenes Byte zurücksenden
    }
}
```

Der Vorteil von Unterprogrammen wird erst offensichtlich, wenn die als Unterprogramme definierten Funktionen im Programm mehrfach genutzt werden.

Mehrfacher Aufruf von get- und sendBytePC

Das folgende Programm wartet in einer Endlosschleife je Durchlauf auf 2× zwei Bytes, setzt diese zu 16-Bit Zahlen zusammen, addiert diese und sendet das Ergebnis als zwei Bytes zurück:

```
int main(void){
  initUSART0();                    // USART initialisieren
  uint16_t a, b;
  while (1) {
    a = getBytePC();               // a_Byte0 empfangen
    a = a + (getBytePC()<<8);// a_Byte1 empfangen
    b = getBytePC();               // b_Byte0 empfangen
    b = b + (getBytePC()<<8);// b_Byte1 empfangen
    a += b;                        // a = a + b
    sendBytePC(a & 0xFF);          // a_Byte0 senden
    sendBytePC(a >> 8);            // a_Byte1 senden
  }
}
```

Aufgabe 3.4: getBytePC und sendBytePC testen

- Legen Sie ein neues Projekt »TestGetSendByte« mit den im »Solution Explorer« auf Folie 24 aufgelisteten Dateien an.
- Vervollständigen Sie die Programmdateien.
- Stellen Sie die serielle Kommunikation zum PC/HTerm her.
- Testen Sie das Programm einmal im Schritt- und einmal im Echtzeitbetrieb.

- Kommentieren Sie das einfache Hauptprogramm von Folie 23 aus und ersetzen Sie es durch das von Folie 26. Testen Sie auch dieses Programm.

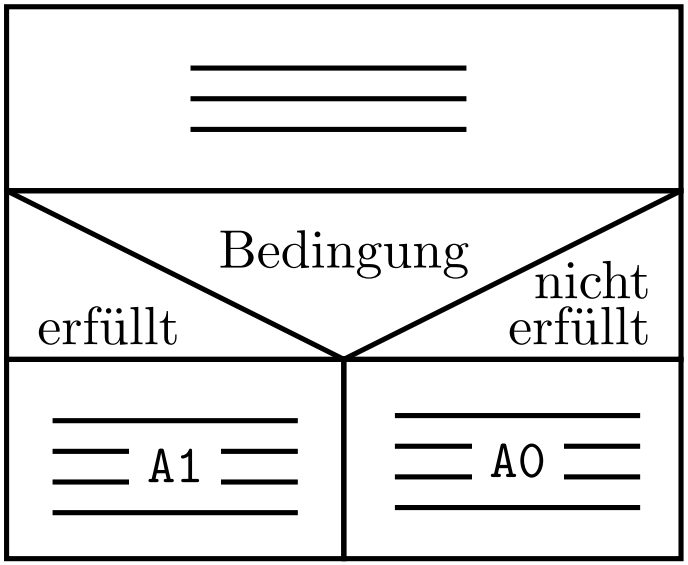
Aufgabe 3.5: Verarbeitungsprogramme mit Byteein- und Byteausgabe

Legen Sie ein neues Projekt »TestGetSendByte1« an und übernehmen Sie die Dateien ComPC.h und ComPC.c. Schreiben Sie als neues Hauptprogramm ein Programm mit einer Endlosschleife, das in jedem Durchlauf

1. auf drei Ascii-Zeichen wartet und diese in umgekehrter Reihenfolge zurückgeschickt,
2. immer auf drei Bytes wartet und die Summe zurückgeschickt.

Fallunterscheidungen

Viele Aufgaben erfordern, dass für unterschiedliche Eingaben unterschiedliche Befehlsfolgen abgearbeitet werden:




```
                                  } gemeinsame
                                    Anweisungsfolge
if  (<Bedingung>){
      A1                          } Anweisungsfolge, wenn
                                    Bedingung erfüllt ist
}
 else {
      A0                          } Anweisungsfolge, wenn
                                    Bedingung nicht
}                                   erfüllt ist
```

Beispiel:

- Wenn ein Buchstabe empfangen wird, Zeichenzähler +1.
- Wenn eine Ziffer empfangen wird: Ziffernzähler + 1.
- Wenn ein Leerzeichen empfangen wird, beide Zählerwerte an PC senden und Zähler löschen.

```
#include <avr/io.h>
#include "comPC.h"

int main(void){
  initUSART0();
  uint8_t zkb=0, zgb=0; // Zähler Klein- und Großbuchstaben
  uint8_t zz=0, dat;    // Zähler Ziffen, Variable für Daten
  while(1) {
    dat = getBytePC();
    if      (dat>='a' && dat<='z') { // 0x61 <= dat <= 0x7A
      zkb++;}
    else if (dat>='A' && dat<='Z') { // 0x41 <= dat <= 0x5A
      zgb++;}
    else if (dat>='0' && dat<='9') { // 0x30 <= dat <= 0x39
      zz++;}
    else if (dat==' ') {
      sendBytePC(zkb+zgb);
      sendBytePC(zz);
      zkb=0; zgb=0; zz=0;
    }                                  // Ende else
  }                                   // Ende while
}                                    // Ende main
```

Erläuterungen:

dat>='a'	Eins (wahr) wenn der Zeichenwert von »dat« mindesten den Wert des Zeichens 'a' (0x61) hat, sonst null.
dat<='z'	Eins (wahr) wenn der Zeichenwert von »dat« höchstens den Wert des Zeichens 'z' (0x7A) hat, sonst null.
... && ...	Logische UND-Verknüpfung zweier Bedingungen, nur eins (wahr), wenn beide Bedingungen eins (wahr) sind.
... || ...	Logische ODER-Verknüpfung, eins (wahr), wenn mindestens eine der beide Bedingungen eins (wahr) ist.

Die Bedingung zur Kontrolle, ob der Wert von »dat« einen Groß- oder ein Kleinbuchstabe darstellt, ist die ODER-Verknüpfung beider Einzelbedingungen:

```
(dat>='a' && dat<='z') || (dat>='A' && dat<='Z')
```

Aufgabe 3.6: Ziffern- und Buchstabenzähl-programm testen und modifizieren

1. Legen Sie ein neues Projekt »BZ_Zaehler« an. Binden Sie die Dateien comPC.c und comPC.h als »Existing Item« ein. Testen Sie es.
2. Vereinfachen Sie das Programm so, dass es nur noch einen Zähler für Buchstaben, statt getrennte Zähler für Klein- und Großbuchstaben benötigt. (Vorheriges Hauptprogramm als Backup oder Kommentar aufheben.)
3. Schreiben Sie ein Programm, das fortlaufend auf Zeichen wartet, sich immer den größten Zeichenwert merkt und diesen bei Empfang eines Leerzeichens zurücksendet. (Vorheriges Hauptprogramm als Backup oder Kommentar aufheben.)

Schleifen

Abweisschleifen

Schleifen dienen zur Wiederholung einer Anweisungsfolge:

```
while (<Bedingung>) {<Anweisungsfolge>)
```

- Wiederhole, solange <Bedingung> nicht null.
- Die Bedingung kann der Wert einer Variablen, ein Vergleichsergebnis, ... oder bei einer Endlosschleife »1« sein.

Beispiel Ausgabefunktion für Zeichenketten:

- Aufruf mit einem Zeiger auf den Anfang der Zeichenkette
- Wiederhole, bis der Zeiger auf den Platz mit Wert null zeigt:
  - Zeichen versenden,
  - Zeiger erhöhen.

- Wiederhole, bis der Zeiger auf den Platz mit Wert null zeigt:
  - Zeichen versenden,
  - Zeiger erhöhen.

```
// Schleifen und Programmende
void sendStringPC(uint8_t *z_ptr){
 while (*z_ptr) {
  sendBytePC(*z_ptr);
  z_ptr++;
 }
}
```

Mit dieser Funktion zusätzlich in comPC() lässt sich die Textausgabe im Hauptprogramm stark vereinfachen:

```
int main(){
  initUSART1();
  uint8_t Testvariable[] = "Das ist eine Textzeile";
  sendStringPC(Textvariable);
  sendStringPC(" So geht es auch.");
}
```

Zählschleifen

Wiederhole eine bestimmte Anzahl mal:

```
for (i=0; i<4; i++) {
  <Anweisungsfolge>)
}
```

Diese Schleife ist identisch mit:

```
i=0;                    // Zähler null setzen
while (i<4){            // solange kleiner Wiederholzahl
 <Anweisungsfolge>
 i++;                   // Zähler erhöhen
}
```

Anwendungen:

- Warteschleife: Zählschleife mit Anweisungen, die nur Zeit verbrauchen.
- Warten auf eine feste Anzahl von Eingabezeichen.

Konvertierung einer Ascii-Folge in eine Zahl

Warten auf vier Hex-Ziffern von USART0. Umwandlung in eine vorzeichenfreie 16-Bit-Zahl:

```
uint16_t Hex2uint16(){
uint8_t i, dat;   // Schleifenzähler, empfangenes Byte
uint16_t erg;     // Variable für das Ergebnis
for (i=0;i<4;i++){// Wiederholschleife
  erg = erg <<4;  // bisherige Werte vier Bit verschieben
  dat = getBytePC();
  if (dat>='0' && dat<='9')
   erg |= (dat-'0')&0xf;
  else if (dat>='a' && dat<='f')
   erg = erg <<4 | (dat-'a'+10)&0xf;
  else if (dat>='A' && dat<='F')
   erg = erg <<4 |(dat-'A'+10)&0xf;
 }
 return erg;
}
```

Ergänzungen zu Feldern und Zeigern

Algorithmen mit Schleifen benötigen oft Felder. Das sind gleichartige Datenobjekte, die hintereinander im Speicher stehen:

```
uint8_t feld[10];     // Feld für 10 Ascii-Zeichen
uint8_t *ptr = feld; // Zuweisung der Adresse des
                     // Feldanfangs[5]
```

- Möglichkeiten, ein Feldelement zu beschreiben:

```
feld[0] = 0x31;          // '1' => Platz 0
*(ptr+1)= 0x61;          // 'a' => Platz 1
ptr += 2; *ptr = 0x62; // 'b' => Platz 2
feld[3] = 0;             //  Abschlussnull
```

[5] »feld« ist identisch mit »&feld[0]« und liefert die Adresse des ersten Feldelements.

Aufgabe 3.7: SendString() testen und verbessern

- Legen Sie ein neues Projekt »TestSendString« an und übernehmen Sie aus Aufgabe 3.4 die Dateien ComPC.c und ComPC.h und fügen Sie das Unterprogramm sendStringPC() von Folie 36 hinzu.
- Verwenden Sie zum Test das Hauptprogramm von Folie 36.
- Zur Vermeidung, dass der halbe Datenspeicher versendet wird, ist es sinnvoll, dem Unterprogramm auch einen Maximalwert für die Anzahl der zu versendenden Zeichen mit zu übergeben

  ```
  void sendStringPC(uint8_t *z_ptr, uint8_t max);
  ```

  und einen Abbruch spätestens nach dieser Anzahl versendeter Zeichen einzuprogrammieren. Verbessern Sie ihr Programm dahingehend.

Aufgabe 3.8: Hex2uint16() testen

- Legen Sie ein neues Projekt »TestHex2uint16()« an und übernehmen Sie aus Aufgabe 3.4 die Dateien ComPC.c und ComPC.h oder nehmen Sie das aus der vorherigen Aufgabe und fügen Sie das Unterprogramm HexStringPC() von Folie 38 hinzu.
- Entwickeln Sie für den Test ein geeignetes Hauptprogramm.

Zeichenkettenfunktionen für das Finalprojekt

Zeichenkettenfunktionen für das Finalprojekt

Ab hier beginnt die gezielte Entwicklung von Funktionsbausteinen für das Fahrzeug.

Die nachfolgend beschriebenen Funktionen:

- Den Inhalt einer Zeichenkette in eine andere kopieren.
- Den Inhalt einer Zeichenkette an den eine anderen anfügen.
- Die Umwandlung einer Zahl in eine Zeichenkette.

finden Sie in strg.h / strg.c auf der Webseite.

Legen Sie ein neues Projekt »TestStrg« an. Binden Sie die Dateien strg.h, strg.c und TestStrgLib.c von der Web-Seite in das Projekt ein. Übersetzten Sie das Programm und starten Sie es mit dem Simulator im Debug-Modus.

Funktion zum Kopieren einer Zeichenkette

```
// kopieren einer Zeichenkette von x nach y
// x_ptr : Zeiger auf den Anfang von x
// y_ptr : Zeiger auf den Anfang von y
// y_len : Länge von y (Schutz vor Überlauf)
void strgcpy(uint8_t *y_ptr, uint8_t *x_ptr, uint8_t y_len){
    while (y_len>0){
        *y_ptr = *x_ptr;
        if (*x_ptr==0) return;
        y_ptr++; x_ptr++;
        y_len--;
    }
}
```

Funktion zum Anhängen einer Zeichenkette

```
// Hängt Zeichenkette x an Zeichenkette y an
// x_ptr : Zeiger auf den Anfang von x
// y_ptr : Zeiger auf den Anfang von y
// y_len : Länge von y
void strgcat(uint8_t *y_ptr, uint8_t *x_ptr,
                             uint8_t y_len){
  while (1){
    if (y_len==0) return;
    if (*y_ptr==0) break;
    y_ptr++;
    y_len--;
  }
  strgcpy(y_ptr, x_ptr, y_len);
}
```

Umwandlung einer Zahl in eine Zeichenkette

```
// Erzeugung einer Zeichenkette für eine vorzeichenfreie Zahl
//  N       : Wert der Zahl
//  Rückgabe: Zeiger auf den Anfang der Zeichenkette

// Rückgabezeichenkette: 32 Bit => max 10 Ziffer plus Abschlussnull
uint8_t uint2str_dat[11];

uint8_t * uint2str(uint32_t N){
    uint8_t *ptr = uint2str_dat +10; // Zeiger auf das letze Element
    *ptr = 0;                        // Schreibe Abschlussnull
    while (1){
        ptr--;                       // Zeiger auf Element davor
        *ptr = (uint8_t)(N%10) + '0';// Schreibe Divisionsrest
                                     // plus Wert des Zeichens '0'
        N /= 10;                     // Zahlenwert durch zehn
        if (N==0) return ptr;        // wenn null, Rückgabe Zeiger
    }                                // auf führende Ziffer
}
```

Testrahmen für die Textverarbeitungsfunktionen

```
 9  #include <avr/io.h>
10  #include "strg.h"
11  #define SIZE 20
12  uint8_t text[SIZE];
13
14  int main(void)
15  {
16    strgcpy(text, (uint8_t*)"Nummer ", SIZE);
17    strgcat(text, uint2str(134), SIZE);
18    strgcat(text, (uint8_t*)": zugehoeriger Text\n", SIZE););
    }
```

Test mit

- »Step over« (Halt nach Unterprogrammabarbeitung) und
- »Step into« (Schrittbetrieb durch die Unterprogramme).
- Zeile 17 arbeitet zuerst uint2str() ab, dann Rücksprung, dann Aufruf »strgcat()«.

Debug-Ausgaben

Halt nach Zeile 16

Watch 1

Name	Value
text	{uint8_t[20]{
[0]	0x4e N
[1]	0x75 u
[2]	0x6d m
[3]	0x6d m
[4]	0x65 e
[5]	0x72 r
[6]	0x20 ␣
[7]	0x00
[8]	0x00

Halt nach Zeile 17

[5]	0x72
[6]	0x20
[7]	0x31 1
[8]	0x33 3
[9]	0x34 4
[10]	0x00
[11]	0x00

uint2str_da	{uint8_t[11]
[0]	0x00
[5]...	0x00
[6]	0x00
[7]	0x31
[8]	0x33
[9]	0x34
[10]	0x00

Halt nach Zeile 18

[8]	0x33
[9]	0x34
[10]	0x3a :
[11]	0x20 ␣
[12]	0x7a z
[13]	0x75 u
[14]	0x67 g
[15]	0x65 e
[16]	0x68 h
[17]	0x6f o
[18]	0x65 e
[19]	0x72 r

```
16 strgcpy(text, (uint8_t*)"Nummer ", SIZE);
17 strgcat(text, uint2str(134), SIZE);
18 strgcat(text, (uint8_t*)": zugehoeriger Te...
```

Aufgabe 3.9: Test der Zeichenkettenfunktionen

Ändern und erweitern Sie den Testrahmen:

- Vergrößern Sie den Puffer, damit die erzeugte Zeichenkette nicht wie im Beispiel abgeschnitten wird.
- Ergänzen Sie weitere Aufrufe der zu testenden Funktionen mit anderen Daten.
- Lassen Sie das Programm den Puffer mehrfach überschreiben.
- Verwenden Sie mehrere Puffer für die Bearbeitung und Zwischenspeicherung der zu erzeugenden Texte.

*Aufgabe 3.10: Funktion zur Umwandlung vorzeichenbehafteter ganzer Zahlen in eine Textdarstellung [6]

Entwickeln Sie für strg.c/strg.h eine Funktion zur Erzeugung einer Textdarstellung für eine vorzeichenbehaftete Zahl:

```
uint8_t int2str(int32_t N);
```

Lösungsidee: Mit uint2str() Betrag in Text umwandeln und für negative Werte minus davor schreiben. Dazu muss u.a. der Puffer uint2str um ein Element verlängert werden, damit auch hier ein Pufferüberlauf ausgeschlossen bleibt.

[6] In den mit * gekennzeichneten Aufgaben werden Funktionsbausteine für das finale Projekt entwickelt. Die hier entwickelten Lösungen gut dokumentieren und gründlich testen.

*Aufgabe 3.11: Festkommazahl als Text

Ausgaben für Wege, Geschwindigkeiten etc. haben oft Nachkommastellen.

Eine Festkommazahl ist eine ganze Zahl mit einem gedachten Komma nach Bit k. Übergabeparameter sind zusätzlich zum Wert die Anzahl der binären Nachkommabits k und die gewünschte Anzahl der dezimalen Nachkommastellen n:

```
uint8_t int2str(int32_t W, uint8_t k, uint8_t n);
```

Entwickeln Sie eine solche Funktion. Testbeispiele:

Wert	0x378F	0xF1D3	0x7FFF
k/n	8/3	8/3	4/1
Text	55,559	-14,176	2047,9

Test mit dem LC-Display

Entwickeln Sie eine Zeichenkettenschreibfunktion für das LC-Display und schreiben Sie unter Nutzung der Funktionen aus strg.c ein Programm, das Text und Zahlen auf das LC-Display schreibt.