Informatikwerkstatt, Foliensatz 2
C-Programmierung

G. Kemnitz

Institut für Informatik, TU Clausthal (IW2)

1. November 2016

Inhalt:

- Wiederholung
- Variablen
- Typecast
- Modularisierung
- Simulation von Modultests
- Aufgaben
- Zusatzteil

Interaktive Übungen:

1. Globale und lokale Variablen (glvar)
2. Kapselung von Funktionen und Objekten (bit_io3_mod)
3. Simulation eines Modultests (mtest_quad)

Wiederholung

Wiederholungsaufgabe 2.1: Hex.-Zahlen

1. Zuordnung der Hexadezimalziffern:

bin.	hex.	bin.	hex.	bin.	hex.	bin.	hex.
0000		0100		1000		1100	
0001		0101		1001		1101	
0010		0110		1010		1110	
0011		0111		1011		1111	

2. Umrechnung nach binär:

```
0x15   = 0b . . . | . . .
0xAF2  = 0b . . . | . . . | . . .
0xABCD = 0b . . . | . . . | . . . | . . .
```

3. Umrechnung nach hexadezimal:

```
0b10010110110   = 0x
0b0100111001011 = 0x
0b00110110      = 0x
```

Lösung

1. Zuordnung der Hexadezimalziffern:

bin.	hex.	bin.	hex.	bin.	hex.	bin.	hex.
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

2. Umrechnung nach binär:

```
0x15   = 0b0.0.0.1|0.1.0.1
0xAF2  = 0b1.0.1.0|1.1.1.1|0.0.1.0
0xABCD = 0b1.0.1.0|1.0.1.1|1.1.0.0|1.1.0.1
```

3. Umrechnung nach hexadezimal:

```
0b100.1011.0110     = 0x4B6
0b0.1001.1100.1011  = 0x9CB
0b0011.0110         = 0x36
```

Wiederholungsaufgabe 2.2: Bitverarbeitung

x_1	x_0	$\bar{x}_0$	$x_1 \wedge x_0$	$x_1 \vee x_0$	$x_1 \oplus x_0$
0	0				
0	1				
1	0				
1	1				

```
uint8_t a, b, c, d, e, f, g;
...
a = 0x3E;                  //a=0b . . . | . . .
b = a & 0b11100010;        //b=0b . . . | . . . =0x
c = b | 0b10010001;        //c=0b . . . | . . . =0x
d = c ^ 0b01100111;        //d=0b . . . | . . . =0x
e = ~d;                    //e=0b . . . | . . . =0x
f = e >> 2;                //f=0b . . . | . . . =0x
g = f << 1;                //g=0b . . . | . . . =0x
```

Tragen Sie in die Kommentare die zugewiesen Werte ein.

Lösung

x_1	x_0	$\bar{x}_0$	$x_1 \wedge x_0$	$x_1 \vee x_0$	$x_1 \oplus x_0$
0	0	1	0	0	0
0	1	0	0	1	1
1	0	–	0	1	1
1	1	–	1	1	0

```
uint8_t a, b, c, d, e, f, g;
...
a = 0x3E;                 //a=0b0.0.1.1|1.1.1.0
b = a & 0b11100010;       //b=0b0.0.1.0|0.0.1.0 =0x22
c = b | 0b10010001;       //c=0b1.0.1.0|0.0.1.1 =0xA3
d = c ^ 0b01100111;       //d=0b1.1.0.0|0.1.0.0 =0xC4
e = ~d;                   //e=0b0.0.1.1|1.0.1.1 =0x3B
f = e >> 2;               //f=0b0.0.0.0|1.1.1.0 =0x0E
g = f << 1;               //g=0b0.0.0.1|1.1.0.0 =0x1C
```

Wiederholungsaufgabe 2.3: Schalter und LEDs

```
#include <avr/io.h>
int main(){
 DDRA =              //Init. als Eingänge
 DDRJ =              //Init. als Ausgänge
 ...      a;         //Variablenvereinbarungen
  while(...){       //
  ...                //Lesen der Eingabe in a
                     //EXOR des gelesenen mit dem
                     //nach rechts verschobenen
                     //gelesenen Wert
                     //löschen der Bits 1 bis 7
                     //Ausgabe von Bit 0 auf LED4
  }
}
```

Ergänzung, so dass in einer Endlosschleife an PJ4 die EXOR-Verknüpfung von PA0 PA1 ausgegeben wird.

Lösung

```
#include <avr/io.h>
int main(){
 DDRA =              //Init. als Eingänge
 DDRJ =              //Init. als Ausgänge
 uint8_t a;          //Variablenvereinbarungen
  while(1){          //
   a = PINA;         //Lesen der Eingabe in a
   a = (a>>1)^a;     //EXOR des gelesenen mit dem
                     //nach rechts verschobenen Wert
   a = a & ~1;       //löschen der Bits 1 bis 7
   PORTJ = a<<4;     //Ausgabe von Bit 0 auf LED4
  }
}
```

Wiederholungsaufgabe 2.4: Automat

In dem Programmrahmen auf der nächsten Folie ist folgende Funktionalität zu ergänzen:

1. Die Anschlüsse PA0 und PA1 seien Ein- und alle Anschlüsse von Port J sowie die restlichen Anschlüsse von Port A Ausgänge.
2. Übergangs- und Ausgabefunktion in der Endlosschleife:
   - Wenn (PA0=1) $\wedge$ (PA1=0): Erhöhung der Ausgabe an Port J um eins.
   - Sonst wenn (PA1=1): Verringerung der Ausgabe an Port J um eins.
   - Sonst: Ausgabe unverändert.
3. Die Übergangs- und Ausgabefunktion soll nur ca. einmal in der Sekunde ausgeführt werden, zu programmieren mit einer Warteschleife mit viermal so vielen Schleifendurchläufen wie bei der 0,25 ms-Warteschleife von Foliensatz IW1 (800.000 statt 200.000 Schleifendurchläufe).

```
#include <avr/io.h>
...          Ct=0;      // Zähler von 0 bis 800000

int main(){
 DDRA  = ...          ; //PA0 und PA1 Eingänge
 DDRJ  = ...          ; //Port J Ausgänge
 PORTJ = ...          ; //Anfangsausgabewert
 while(1){             //Endlosschleife
  if (...             )// Wenn PA1=0 und PA0=1
   ...                 // Port J hochzählen
  else if (...            )// sonst wenn PA1=1
   ...                 // Port J abwärts zählen
  for (                    );// Warteschl. 1s
 }
}
```

Lösung

```
#include <avr/io.h>
uint32_t Ct=0;                  //Zähler von 0 bis 800000
int main(){
 DDRA = ~0x03;                  //PA0 und PA1 Eingänge
 DDRJ = 0xFF;                   //Port J Ausgänge
 PORTJ = 0;                     //Anfangsausgabewert
 while(1){                      //Endlosschleife
  if ((PINA&0x3)==1)            //Wenn PA1=0 und PA0=1
   PORTJ++;                     //Port J hochzählen
  else if (PINA & 2)            //sonst wenn PA1=1
   PORTJ--;                     //Port J abwärtszählen
   for (ct=0;ct<800000;ct++); // Warteschl. 1s
  }
}
```

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).

	1 Byte		2 Byte		
ohne VZ	uint8_t	$[0, 255]$	uint16_t	$[0, 2^{16} - 1]$	
mit VZ	int8_t	$[-128, 127]$	int16_t	$[-2^{15}, 2^{15} - 1]$	

Kontrollfragen

- Welche Byteanzahl und Wertebereiche haben »uint32_t« und »int32_t?

Für Studierende mit C-Vorkenntnissen:

- Welchen der behandelten Typen entsprechen die Standard-C-Typen »char« und »unsigned integer«?
- Weiß jemand, was die Vorsätze vor Typdefinitionen »register« und »volatile« bedeuten und bewirken?

Wert und Adresse einer Variablen

- Der Compiler ordnet jeder Variablen eine Adresse oder ein Register zu. Adresse/Register im Debugger visualisierbar.

```
uint8_t a, b, *ptr;
int main(void){
  a   = 0x4D;
  ptr = &a;
  b   = *ptr + 3;
}
```

Watch 1

Name	Value	Type
a	0x4d	uint8_t{data}@0x0204
b	0x50	uint8_t{data}@0x0200
ptr	0x0204	uint8_t*{data}@0x0201
	0x4d	uint8_t{data}@0x0204

- C kennt auch Variablen für Adressen. Vereinbarung mit »<Typ> *<name>«, z.B.:

```
uint8_t *ptr;
```

Zeiger

Ein Zeiger ist eine Variable für eine Adresse.

- Vereinbarung eines Zeigers auf Variablen eines bestimmten Typs (z.B. uint8_t):

```
uint8_t *ptr;
```

- Vereinbarung eines Zeigers für beliebige Datentypen (auch für die Aufrufadressen von Funktionen):

```
void *ptr;
```

- Die Adresse einer Variablen liefert der Operator »&«, z.B.:

```
ptr = &a;
```

- Den Wert zu einer Adresse liefert der Operator »*«, z.B.:

```
b = *ptr;
```

Globale und lokale Variablen

- Global: Außerhalb einer Funktion vereinbart. Feste Datenspeicheradresse. Existieren während der gesamten Programmlaufzeit.
- Lokal: Innerhalb eines Blocks (innerhalb von {...}) vereinbart. Existieren nur bis zum Verlassen des Blocks. Speicherplatz wird erst bei Eintritt in den Block auf dem sog. Stack reserviert.
- Die Adressierung lokaler Variablen erfolgt relativ zum Frame-Pointer (in unserem Prozessor Registerpaar r28:r29).

```
uint8_t a;
int main(void){
  uint8_t b = 0x21;
  a = b + 3;
}
```

Watch 1

Name	Value	Type
a	0x24	uint8_t{data}@0x0201
b	0x21	uint8_t{data}@0x21fa ([R28]+1)

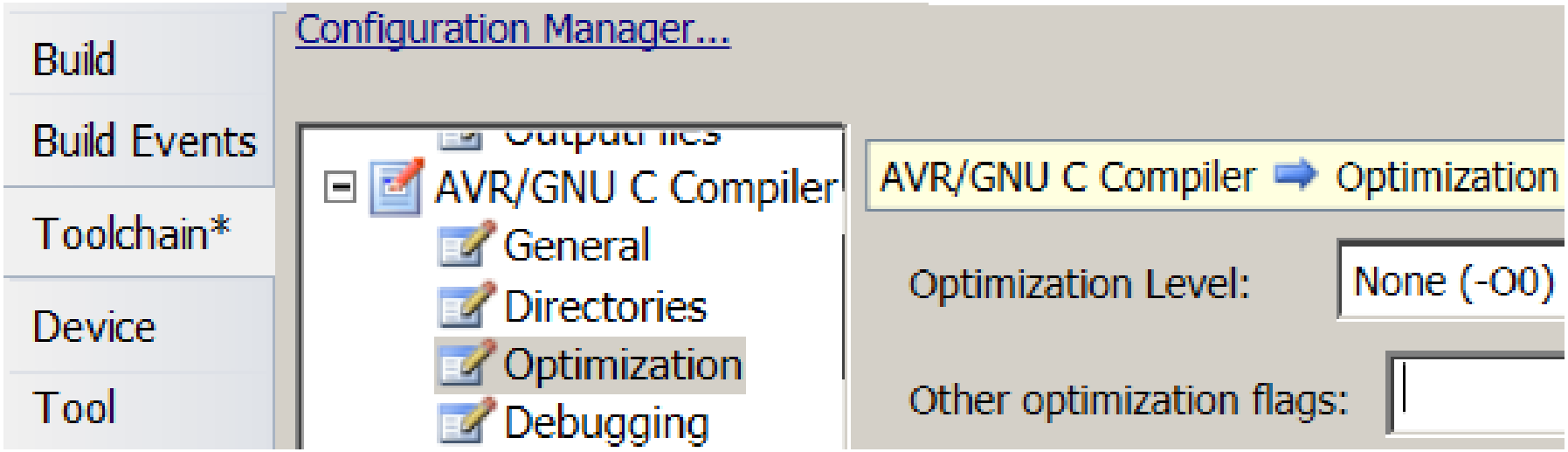
Experiment

Öffnen Sie im Verzeichnis »P02\F2-glvar« das Projekt »glvar« und die Datei »glvar.c«:

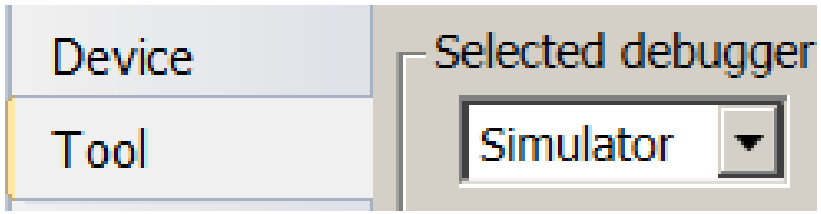
```
#include <avr/io.h>
int16_t gi16;      //global 2 Byte, VZ, AW 0
uint8_t gu8;       //global 1 Byte, NVZ, AW 0
int main(void){
 uint8_t  lu8  = 0x2D;  //1 Byte, NVZ, AW 0x2D
 int16_t  li16 = 0x51F4;//2 Byte, VZ, AW 0x51F4
 uint8_t *lpu8 = &gu8;  //Zeiger auf uint8_t,
                        //AW Adresse von gu8
 gi16  = li16 + 1;
 *lpu8 = lu8 - 4;//Wertzuw. an Adresse, hier gu8
 lpu8  = &lu8;    //Zuweisung Adresse von lu8
 *lpu8 = 0xA5;    //Wertzuw. an Adresse, hier lu8
 lu8   = 23;
}
```

- Übersetzen mit -O0

```
Project > glvar Properties... (Alt+F7)
```



- Auswahl des Simulators als »Debugger«



- Debugger starten:

- Öffnen »Locals«, »Watch 1« und zwei Speicherfenster mit

```
Debug > Windows > Locals (Alt+4)
Debug > Windows > Watch > Watch 1 (Ctrl+Alt+W+1)
Debug > Windows > Memory > Memory 1 (Alt+6)
Debug > Windows > Memory > Memory 2 (Ctrl+Alt+M,2)
```

- In den Memory-Fenstern »IRAM« für internen Speicher auswählen und wie auf der Folgefolie den Adressbereich der globalen bzw. lokalen Variablen einstellen.

Variablenwerte und Adressen vor Zuweisung 1

```
int main(void){
  uint8_t  lu8  = 0x2D;
  int16_t  li16 = 0x51F4;
  uint8_t *lpu8 = &gu8;

  gi16  = li16 + 1;
  *lpu8 = lu8 - 4;
  lpu8  = &lu8;
  *lpu8 = 0xA5;
  lu8   = 23;
}
```

Watch 1

Name	Value	Type
gi16	0x0000	int16_t{data}@0x020C
gu8	0x00	uint8_t{data}@0x0202

Locals

Name	Value	Type
lu8	0x2d	uint8_t{data}@0x21fa ([R28]+5)
li16	0x51f4	int16_t{data}@0x21f6 ([R28]+1)
lpu8	0x0202	uint8_t*{data}@0x21f8 ([R28]+3

Memory 1

Memory: data IRAM

data 0x0200 00 00 00 00

Memory 2

Memory: data IRAM

data 0x21F5 00 f4 51 02 02 2d

data 0x21FB 21 ff 00 00 83 00

- eine Anweisung weiter:

```
int main(void){
  uint8_t  lu8  = 0x2D;
  int16_t  li16 = 0x51F4;
  uint8_t *lpu8 = &gu8;

  gi16  = li16 + 1;
  *lpu8 = lu8 - 4;
  lpu8  = &lu8;
  *lpu8 = 0xA5;
  lu8   = 23;
}
```

Watch 1

Name	Value	Type
gi16	0x51f5	int16_t{data}@0x0200
gu8	0x00	uint8_t{data}@0x0202

Locals

Name	Value	Type
lu8	0x2d	uint8_t{data}@0x21fa ([R28]+5)
li16	0x51f4	int16_t{data}@0x21f6 ([R28]+1)
lpu8	0x0202	uint8_t*{data}@0x21f8 ([R28]+3)

Memory 1

Memory: data IRAM

```
data 0x0200  f5 51 00 00
```

Memory 2

Memory: data IRAM

```
data 0x21F5  00 f4 51 02 02 2d
data 0x21FB  21 ff 00 00 83 00
```

- Noch eine Anweisung weiter:

```
int main(void){
    uint8_t  lu8  = 0x2D;
    int16_t  li16 = 0x51F4;
    uint8_t *lpu8 = &gu8;

    gi16  = li16 + 1;
    *lpu8 = lu8 - 4;/
    lpu8  = &lu8;   /
    *lpu8 = 0xA5;   /
    lu8   = 23;
}
```

Watch 1

Name	Value
gi16	0x51f5
gu8	0x29

Locals

Name	Value
lu8	0x2d
li16	0x51f4
lpu8	0x0202

- Noch eine Anweisung weiter:

```
int main(void){
    uint8_t  lu8  = 0x2D;
    int16_t  li16 = 0x51F4;
    uint8_t *lpu8 = &gu8;

    gi16  = li16 + 1;
    *lpu8 = lu8 - 4;/
    lpu8  = &lu8;   /
    *lpu8 = 0xA5;   /
    lu8   = 23;
}
```

Watch 1

Name	Value
gi16	0x51f5
gu8	0x29

Locals

Name	Value
lu8	0x2d
li16	0x51f4
lpu8	0x21fa

- Noch eine Anweisung weiter:

```
int main(void){
    uint8_t  lu8  = 0x2D;
    int16_t  li16 = 0x51F4;
    uint8_t *lpu8 = &gu8;

    gi16  = li16 + 1;
    *lpu8 = lu8 - 4;/
    lpu8  = &lu8;   /
    *lpu8 = 0xA5;   /
    lu8   = 23;
}
```

Watch 1

Name	Value
gi16	0x51f5
gu8	0x29

Locals

Name	Value	Type
lu8	0xa5	
li16	0x51f4	
lpu8	0x21fa	

- Variablenwerte nach der letzten Zuweisung:

```
int main(void){
   uint8_t  lu8  = 0x2D;
   int16_t  li16 = 0x51F4;
   uint8_t *lpu8 = &gu8;

   gi16  = li16 + 1;
   *lpu8 = lu8 - 4;/
   lpu8  = &lu8;   /
   *lpu8 = 0xA5;   /
   lu8   = 23;
}
```

Watch 1

Name	Value
gi16	0x51f5
gu8	0x29

Locals

Name	Value	Type
lu8	0x17	
li16	0x51f4	
lpu8	0x21fa	

Typecast

Typprüfung und 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; //Zuweisung mit Typecast
```

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

Was macht Atmel-Studio?

```
#include <avr/io.h>
char c, *c_ptr;      //char kann [u]int8_t sein,
uint8_t u, *u_ptr;   //ist lt. Toochain uint8..
int8_t i, *i_ptr;
int main(){  //Warnung, wenn nicht int, warum?
 c=u;    //laut Toochain korrekt, keine Warnung
 i=c;    //laut Toolchain falsch, keine Warnung
 c_ptr = &c;  //zulässig, keine Warnung
 c_ptr = &u;  //laut Toochain korrekt, Warnung
 c_ptr = (char*)&u;//Typcast, keine Warnung
 c_ptr = &i;  //laut Toochain falsch, Warnung
 c_ptr = (char*)&i;// Typcast, keine Warnung
}
```

Was zeigt uns das Experiment

- Selbst mit »Project > ...Properties > Toolchain > AVR/GNU C Compiler > Warnings > Pedantic $\sqrt{}$« sind die Warnungen weder vollständig noch immer schlüssig.
- Kein Programm, auch nicht der Compiler liefert immer vernüftige / zulässige / richtige Ergebnisse.

Sie werden es in dieser LV noch mit vielen Fehlern zu tun bekommen

- eigenen und in benutzten Programmbausteinen,
- fehlende Fehlermeldungen und Übersetzungsfehler,
- falsch zusammengebaute und kaputte Hardware,
- vermeindliche Fehler, die in Wirklichkeit keine sind, ..

Ein praktisch arbeitender Informatiker beschäftigt sich die meiste Zeit mit Fehlern, Fehlersuche, Workarounds für erkannte Fehlfunktionen, ...

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?

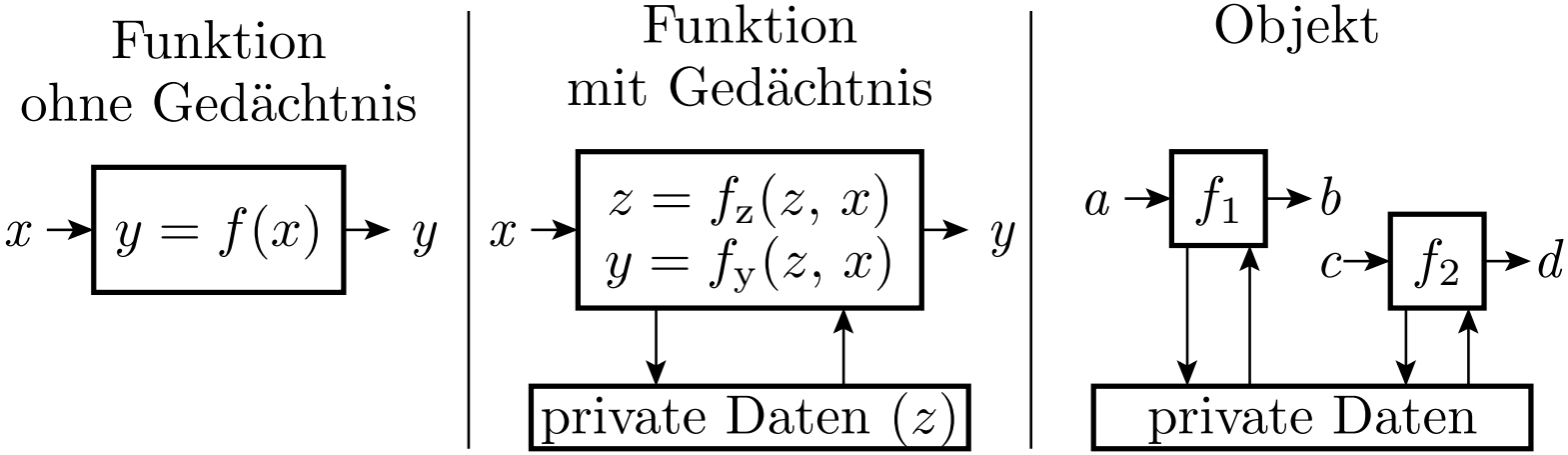
(Fortsetzung als Zusatzaufgabe 2.1.)

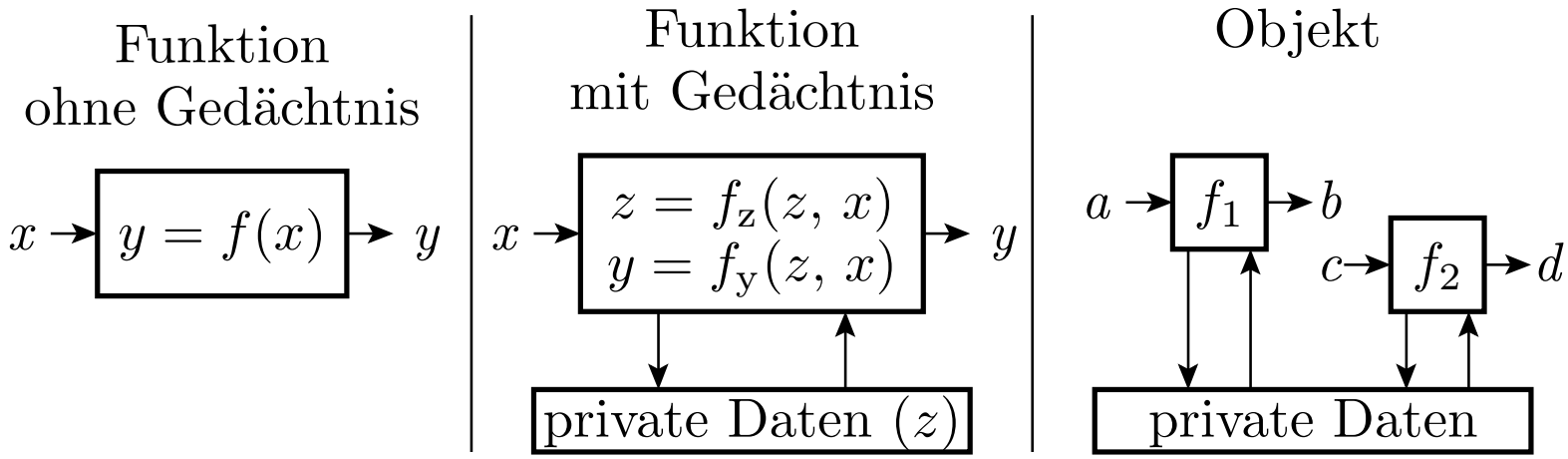
Modularisierung

Modularisierung

Größere Programme bestehen aus Modulen:

- Funktionen
  - ohne Gedächtnis ($y = f(x)$),
  - mit Gedächtnis ($z = f_z(z, x)$; $y = f_y(z, x)$),
- Objekten (Datenobjekt mit Bearbeitungsfunktionen),
- Bibliotheken (Funktions- und Objektsammlungen), ...





Grundprinzipien der Fehlervermeidung, Wartbarkeit, ...:

- Kapselung: Kein Lese- oder Schreibzugriff fremder Programmteile auf private Daten.
- Abstraktion: Übergeordnete Module kennen nur die Schnittstellen und Funktionen genutzter Module, nicht aber deren Realisierung, Adressen und Codierung privater Daten, ...
- ...

≈75% der Software-Kosten entfallen auf prüfgerechte Programmierung, Test und Fehlerbeseitigung.

Funktionsdefinition und Aufruf

- Eine (reine) Funktion berechnet aus Eingabewerten ein Ergebnis:

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

- Zustandsdaten, die nach Beenden der Funktion erhalten bleiben sollen, sind global oder vor Aufruf zu vereinbaren:

```
int16_t err_ct;     //glob. Variable Fehlerzähler
...
void test_lim(int16_t val, int16_t max,
                           int16_t min){
  if ((val>max) || (val<min))
    err_ct++;        //Fehlerzähler erhöhen
}                    //Rücksprung ohne Ergebnis
```

Mit Zeigern als Aufrufparameter kann eine Funktion auch mehrere Ergebnisse zurückgeben:

```
void bytecopy(uint8_t *ziel, uint8_t *quelle,
              uint8_t anz){
  uint8_t idx;
  for (idx=0;idx<anz;idx++)
    ziel[idx] = quelle[idx];
    //identisch mit *(ziel+idx) = *(quelle+idx);
}
```

Nutzung zum Kopieren einer Zeichenkette:

```
uint8_t a[]="Text";//Feld: 0x54,0x65,0x78,0x74,0
uint8_t b[8];      //Feld: 0,0,0,0,0,0,0,0
...
bytecopy(b, a, 4); //a, b: Zeiger auf Feldanfang
//neue Werte in b: 0x54,0x65,0x78,0x74,0,0,0,0
```

(Fortsetzung als Zusatzaufgabe 2.2.)

Kapselung von Funktionen und Objekten

- Beschreibung der Funktion(en) und Vereinbarung der privaten Daten in einer eigenen c-Datei.
- Schnittstellenvereinbarung in der zugehörigen Header-Datei.
- Separates Übersetzen (Compilieren) jeder c-Datei.
- Verbinden (Linken) über die Header-Informationen.

Als Beispiel werden aus dem Programm »bit_io3.c« für das Vorwärts-/Rückwärts-Lauflicht die Warteschleife und die Übergangsfunktion als eigenes Modul in eine neue c-Datei ausgelagert, Header ergänzt und das Hauptprogramm angepasst. Praktische Programmerstellung und Test siehe später Zusatzaufgabe 2.3.

Auszulagernde Anweisungen aus »bit_io3.c«

```
#include <avr/io.h>
uint32_t Ct;                    //
uint8_t a=1;                    //
int main(void){                 //
 DDRA = 0;                      //
 DDRJ = 0xFF;                   // --------
 while(1){for (Ct=0;            // Warte-
    Ct<200000; Ct++);           // schleife
  if (PINA & 0b1)               // --------
   a = (a<<1) | (a>>7);         // Übergangs- und
  else                          // Ausgabefunktion
   a = (a>>1) | (a<<7);         // --------
  PORTJ = a;                    //
 }
}
```

Auslagerung der Übergangs- und Ausgabefunktion und der Warteschleife in eine neue c-Datei »myfkt.c«.

Inhalt der Datei »myfkt.c«

```
#include "myfkt.h" //Header eigene Funktionen
uint8_t a=1;       //Automatenzustand
uint32_t Ct;       //Zählerzustand

uint8_t Schrittfunktion(uint8_t x){
  if (x & 0b1)     //Rotation links
    a = (a<<1) | (a>>7);
  else             //Rotation rechts
    a = (a>>1) | (a<<7);
  return a;
}

void Warte_1s(){for (Ct=0; Ct<200000; Ct++);}
```

- Automaten- und Zählerzustand werden in privaten, globalen Variablen gespeichert.
- Nur die Funktionsschnittstellen nach außen sichtbar.

Header-Datei

```
#ifndef MYFKT_H_
#define MYFKT_H_
#include <avr/io.h>
  uint8_t Schrittfunktion(uint8_t x);
  void Warte_1s();
#endif /* MYFKT_H_ */
```

- #ifndef ... dient dazu, dass, auch wenn die Header in mehrere Projektdateien eingefügt werden, die eingerahmten Schnittstellendefinitionen nur einmal im zu übersetzenden Code übrig bleiben.
- In Header gehören nur die Definitionen von Funktionsaufrufschnittstellen, Konstantenvereinbarungen, aber nichts dass
  - für andere Programmteile nicht sichtbar sein soll oder
  - in abzuarbeitende Anweisungen übersetzt wird.

Hauptprogramm (bit_io3_mod.c)

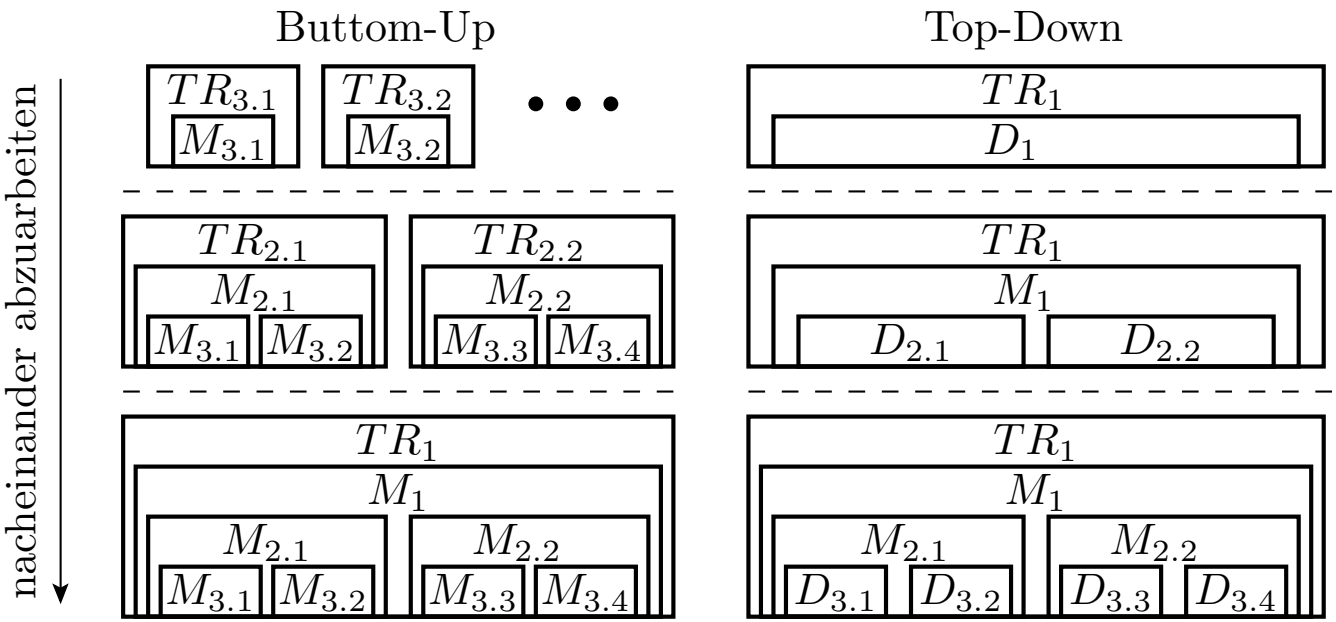
```
#include <avr/io.h>
#include "myfkt.h" //Header eigene Funktionen
int main(void){
  DDRA = 0;          //Port A Schaltereingänge
  DDRJ = 0xFF;       //Port J LED-Ausgänge
  while(1){
    PORTJ = Schrittfunktion(PINA);
    Warte_1s();
 }
}
```

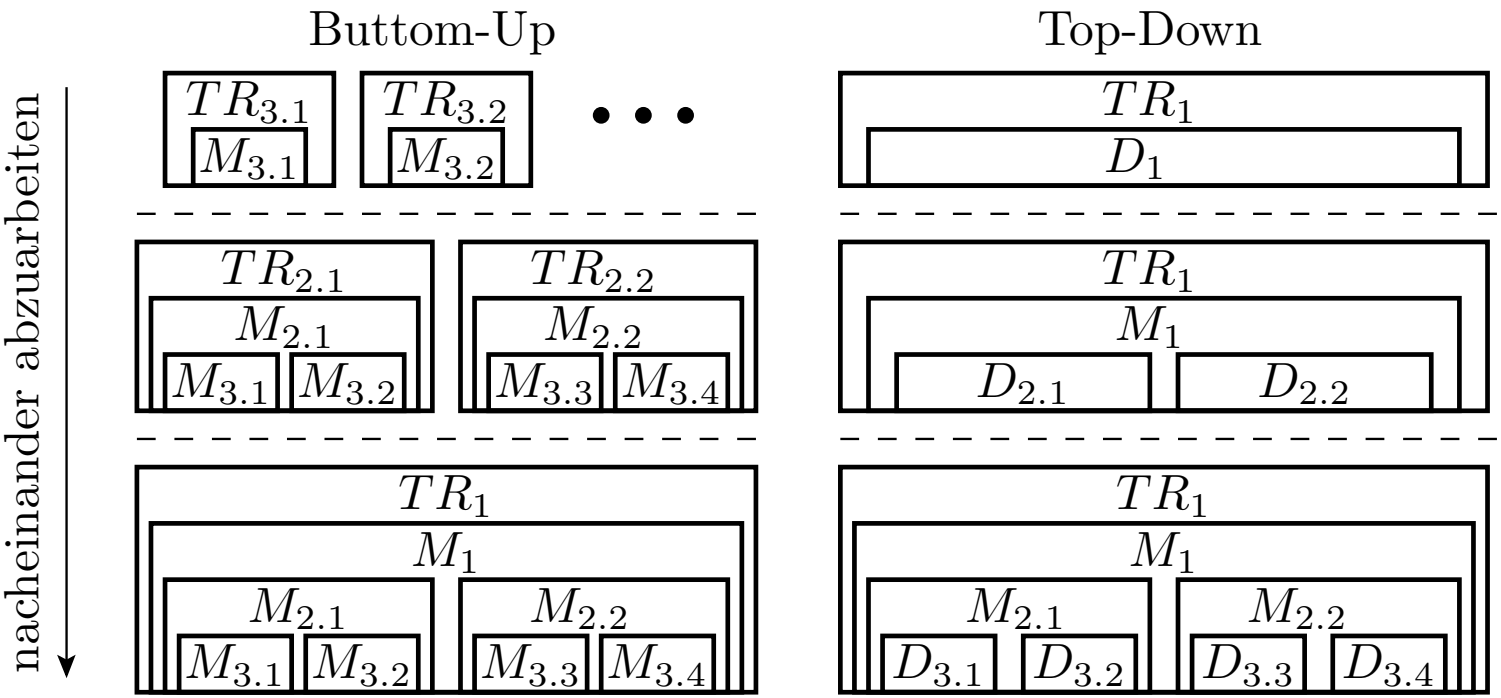
- Ersatz der Warteschleife und der Anweisungen für die Warteschleife und die Übergangs- (Schritt-) Funktion durch Funktionsaufrufe.
- Praktische Programmerstellung und Test siehe später Zusatzaufgabe 2.3.

Simulation von Modultests

Testrahmen und Dummies

- Entwurf und Test komplexer Systeme erfolgt modulweise.
- Für jeden Modultest stellt ein übergeordnetes Modul oder ein Testrahmen Eingaben bereit und wertet die Ausgaben aus.
- Noch nicht programmierte / getestete Teilmodule werden zum Teil durch Dummies ersetzt.





M_i Modul i D_i Dummie für Modul i TR_i Testrahmen für Modul i

- Ein Buttom-Up-Entwurf beginnt mit den untersten genutzten Modulen und kommt im Idealfall ohne Dummies aus.
- Ein Top-Down-Entwurf beginnt mit dem obersten Modul und kommt im Idealfall mit nur einem Testrahmen aus.
- Im Normalfall ist eine Vielzahl von Testrahmen und Dummies mit zu entwerfen und zu testen.

Ein Testobjekt und sein Testrahmen

Testobjekt sei das folgende Unterprogramm zur Quadrierung[1]:

```
uint32_t quad(uint16_t a){ //Quadratberechnung
  return (uint32_t)a * a;
};
```

Ein Test ist ein Tupel aus Eingaben und Soll-Ausgaben. Tupel werden in C als »struct« programmiert, z.B.:

```
struct test {
  int16_t x;                    //Eingabe
  uint32_t y;                   //Sollausgabe
};
```

[1] Den Grund für den Typecast »(uint32_t)a« sollen Sie selbst herausfinden, indem Sie kontrollieren, welche der Testbeispiele ohne den Typecast falsch ausgeführt werden.

Testsatz

Eine Testsatz als Menge von Tests wird in C im einfachsten Fall als initalisiertes Feld von Testbeispielen programmiert:

```
struct test testsatz[] ={{<Tupel1>},{...},...};
```

Testbeispiele für die Quadratberechnung:

```
struct test testsatz[] ={//Eingabe   Soll-Wert
  {0, 0},                  // 0x0000   0x00000000
  {1, 1},                  // 0x0001   0x00000001
  {9, 81},                 // 0x0009   0x00000051
  {-5, 25},                // 0xFFFB   0x00000019
  {463, 214369},           // 0x01CF   0x00034561
  {0x7FFF, 1073676289}     // 0x7FFF   0x3FFF0001
};
```

Welche Testbeispiel könnten ohne den Typecast im Ausdruck »(uint32_t)a * a;« versagen?

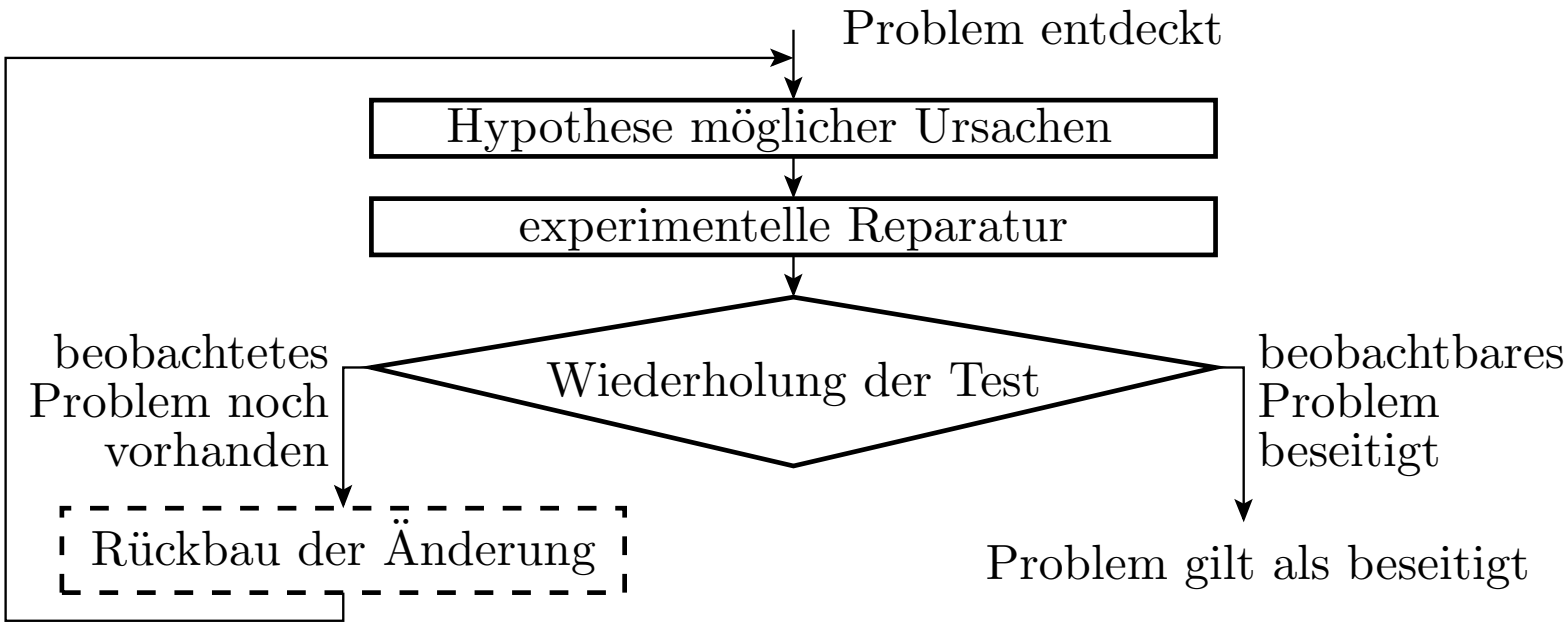
Testrahmen

Der Testrahmen ist ein Hauptprogramm, das in einer Schleife alle Testbeispiele abarbeitet und Ergebnisse kontrolliert, ...:

```
int main(){
 uint8_t idx, err_ct=0;
 uint32_t erg;
 for (idx=0; idx<6;idx++){    //für alle Tests
 erg = quad(testsatz[idx].x);//Istwertberechnung
 if (erg != testsatz[idx].y) //Soll/Ist-Vergl.
  err_ct++;                  //Fehlfkt. zählen
 }
}
```

Testdurchführung mit dem im Simulator im Debug-Modus:

- Unterbrechungspunkt vor dem Fehlerzähler.
- Bei jeder Fehlfunktion Halt am Unterbrechungspunkt.
- Für alle Fehlfunktionen (Halt am Unterbrechungspunkt) ...



Für alle Fehlfunktionen Problembeseitigungsiteration:

- Fehlerhypothese, Reparaturversuch,
- wenn erfolglos, Rückbau und nächster Reparaturversuch.

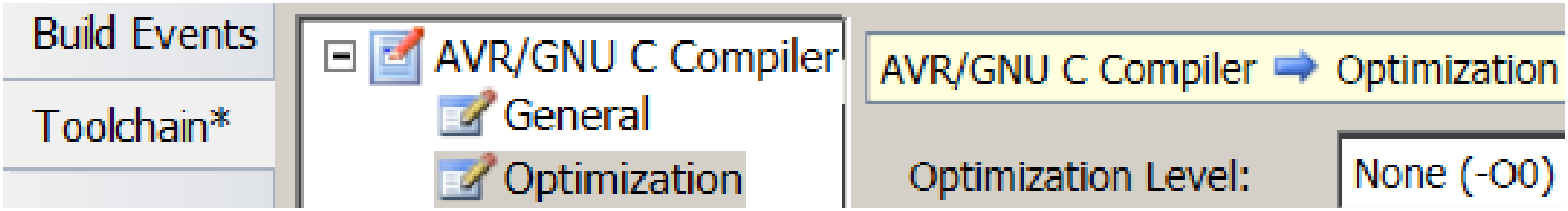
Zu unserem Testobjekt und Testrahmen:

- Der Testrahmen ist fehlerhaft. Fehler suchen und beseitigen.
- Dann den Typecast »(uint32_t)a« im Testobjekt weglassen,
- Test wiederholen, ...

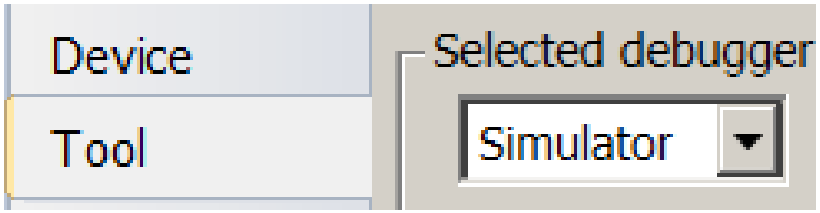
Projekt mtest_quad« ausprobieren

- Projekt »mtest_quad« öffnen.
- Compiler-Optimierung ausschalten (in -O0 ändern):

  `Project > mtest_quad Properties... (Alt+F7)`



- Auswahl des Simulators als »Debugger«:



- Übersetzen.
- Debugger starten:

Test und Fehlerlokalisierung im Schrittbetrieb:

```
struct{
  int16_t x;
  uint32_t y;
} testsatz[]={
  {0, 0}, {1, 1}, {9, 81},
  {-5, 25}, {463, 214369},
  {0x7FFF, 1073676289}
};

int main(){
  uint8_t idx, err_ct=0;
  uint32_t erg;
  for (idx=0; idx<6;idx++){
    erg = quad(testsatz[idx].y);
    if (erg != testsatz[idx].y)
      err_ct++;
  }
}
```

Watch 1

Name	Value
testsatz	{struct [6]{data}@0x0200}
[0]	{struct {data}@0x0200}
[1]	{struct {data}@0x0206}
[2]	{struct {data}@0x020c}
x	0x0009
y	0x00000051
[3]	{struct {data}@0x0212}
[4]	{struct {data}@0x0218}
[5]	{struct {data}@0x021e}
idx	0x02
erg	0x000019a1

Berechnet das Quadrat von 81, statt von 9!

Aufgaben

Aufgabe 2.1: Globale und lokale Variablen

Vereinbaren Sie folgende Variablen global

```
uint8_t a, b;
```

und folgende Variablen lokal im Hauptprogramm:

```
uint8_t c=0x7, *ptr=&a;
uint8_t strg[]={0x11, 0x32, 0x07, 0x02};
```

Bestimmen Sie die Werte nach Abarbeitung folgender Programmzeilen:

```
1: a = 0x56; b = 0x27;
2: ptr = &a; c = strg[2];
3: b = *ptr+2; c += *(strg+3);
4: ptr = strg; a += *(ptr+1);
```

Arbeitsschritte siehe nächste Folie.

Arbeitsschritte:

- Projekt anlegen, Programm vervollständigen.
- Abarbeitung im Schrittbetrieb mit dem Simulator.
- Übernahme der nachfolgenden Tabelle auf Papier und Ausfüllen.

Zeile	a	b	c	ptr	*ptr	strg[0]	strg[1]	strg[2]	strg[3]
0									
1									
2									
3									
4									

Aufgabe 2.2: Untersuchung von Zuweisungen

```
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?

- Projekt anlegen, Programm vervollständigen.
- Abarbeitung mit dem Simulator im Schrittbetrieb.

Aufgabe 2.3: Multiplikationsfehler

Das Ergebnis der nachfolgenden Multiplikation ist falsch.

```
#include <avr/io.h>
int main(void){
    uint16_t a = 0x1FA;
    uint16_t b = 0x100;
    uint32_t p = a*b;
}
```

Locals

Name	Value	Type
a	0x01fa	uint16_t{data}@0x21f3
b	0x0100	uint16_t{data}@0x21f5
p	0x0000fa00	uint32_t{data}@0x21f7

- Überprüfen Sie das im Simulator!
- Wie lautet das richtige Ergebnis von 0x1FA*0x0100?
- Ändern Sie die Multiplikation bzw. die Datentypen so, dass das Ergebnis richtig berechnet wird.

- Projekt anlegen, Programm vervollständigen.
- Abarbeitung im Schrittbetrieb mit dem Simulator.

- Übernehmen Sie die nachfolgende Tabelle mit Testbeispielen auf Papier und ergänzen Sie die fehlenden Werte:

a		b		p	
18	0x0012	8	0x0008	96	0x00000090
134		270			
703		0			
8.351		407			
60.000		50.000		3.000.000.000	
	0xFFFF		0xFFFF		

- Programmieren Sie für die korrigierte Multiplikation einen Modultest mit den Testbeispielen in der Tabelle.

Aufgabe 2.4: Fehler Betragsbegrenzung

Die Funktion »limit()« soll einen 32-Bit-Festkommawert mit 8 Nachkommastellen betragsmäßig auf den ganzzahligen Wert »maxabs« begrenzen. Das Testbeispiel liefert ein falsches Ergebnis.

```
int32_t limit(int32_t val, uint16_t maxabs){
  if ((val>>8)> maxabs) return  maxabs<<8;
  if ((val>>8)<-maxabs) return -(maxabs<<8);
  return val;
}

int main(){
  int32_t v = limit(0, 0xFFFF);
}
```

Locals

Name	Value	Type
v	0x00000100	int32_t{data}@0x21f7 ([R28]+1)

- Ausprobieren im Simulator und Fehlerbeseitigung.

Aufgabe 2.5: Fehler Wurzelberechnung

Die nachfolgende Funktion berechnet die ganzzahlige Quadratwurzel einer 16-Bit-Zahl:

```
uint8_t wurzel(uint16_t x){
  uint8_t w=0; uint16_t sum=0;
  while (sum<x){
    sum += (w<<1)+1; w++;
  }
  return w;
}
```

- Ist das Ergebnis des folgenden Testbeispiels richtig?

```
int main(){
  uint16_t x = 37481;
  uint8_t y = wurzel(x);
}
```

Watch 1

Name	Value	Type
x	37481	uint16_t{data}@0x21f8
y	194	uint8_t{data}@0x21fa

- Führen Sie die Tests auch mit folgenden Testbeispielen durch:

x		y	
0	0x0000	0	0x00
1			
257			
8.351			
65025			
65026			
	0xFFFF	256	0x100

- Bei welchen Tests versagt das Programm und wie?
- Suchen und beseitigen Sie den Fehler.

Zusatzteil

Zusatzaufgabe 2.1: Typprobleme

```
uint8_t a; int8_t b;
int main(){
 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?
}
```

- Projektanlegen, Programm eingeben, mit »-O0« übersetzen.
- Compiler-Warnungen und Übersetzungsfehler beseitigen.
- Im Simulator im Schrittbetrieb abarbeiten.
- Ursachen für falsch zugewiesene Werte beseitigen.

Zusatzaufgabe 2.2: Test der Kopierfunktion

- Legen Sie für das Programm auf der nächsten Folie ein neues Projekt »test_bytecopy« mit einer c-Datei an.
- Geben Sie Unterprogramm und Hauptprogramm in der vorgegebenen Reihenfolge ein und übersetzen Sie mit »-O0«.
- Abarbeiten im Debugger im Schrittbetrieb, einmal mit »Step-Over« (Unterprogrammaufrufe als einen Schritt) und einmal mit »Step-Into« (Unterprogrammabarbeitung zeilenweise).
- Auf welche Adressen zeigen die Pointer a, b und c zum Programmbeginn und nach Abarbeitung der einzelnen Hauptprogrammzeilen? (Test mit »Step-Over.)
- Welche Zahlenfolgen stehen zum Programmbeginn in den Feldern a[] und b[] und nach Programmabschluss im Feld c[]?

```
1  void bytecopy(uint8_t *ziel, uint8_t *quelle,
2                uint8_t anz){
3    uint8_t idx;
4    for (idx=0;idx<anz;idx++)
5      ziel[idx] = quelle[idx];
6  }
7  int main(){
8   uint8_t a[] = "Text";
9   uint8_t b[] = "Welt";
10  uint8_t c[10];
11  ...
12  bytecopy(c, a, 4);
13  *c = '␣';
14  c++;
15  bytecopy(c, b, 5);
16 }
```

- Suchen Sie sich im Internet eine ASCII-Tabelle. Welche Zeichenfolgen stehen in den Feldern a[] bis c[]?

Zusatzaufgabe 2.3: Modularisierung

- Stecken Sie ein Schaltermodul PmodSWT an Stecker JA oben.
- Kopieren Sie aus dem Projektverzeichnis »bit_io3« die Dateien »bit_io3.c« und »bit_io3.cproj« in ein neu anzulegendes Projektverzeichnis »bit_io3_mod«.
- Öffnen des neuen Projekts.
- Erzeugung eine neue c-Datei »myfkt.c« und eine neue Header-Datei »myfkt.h«[2].
- Kopieren Sie die Programmzeilen für die Warteschleife und die Schrittfunktion aus »bit_io3.c« in die neue c-Datei und passen Sie beide c-Dateien und die Header-Datei entsprechend der beiden nachfolgenden Folien an.

[2] Solution Explorer > Rechtsklick auf bit_io3 > Add > New Item > ...

c-Datei »myfkt.c« (zu erstellen)

```
#include <avr/io.h>
                    // private Daten
uint8_t a=1;        // Automatenzustand
uint32_t Ct;        // Zählerzustand

uint8_t Schrittfunktion(uint8_t x){
  if (x & 0b1)        // Rotation links
    a = (a<<1) | (a>>7);
  else                // Rotation rechts
    a = (a>>1) | (a<<7);
  return a;
}

void Warte_1s(){
  for (Ct=0; Ct<200000; Ct++);
}
```

Header und Hauptprogramm

Zu erstellender Header-Datei »myfkt.h«:

```
#ifndef MYFKT_H_
#define MYFKT_H_
#include <avr/io.h>
  uint8_t Schrittfunktion(uint8_t x);
  void Warte_1s();
#endif /* MYFKT_H_ */
```

Angepasstes Hauptprogramm »bit_io3_mod.c«:

```
#include "myfkt.h" //Header eigene Funktionen
int main(void){
  DDRA = 0;          //Port A Schaltereingänge
  DDRJ = 0xFF;       //Port J LED-Ausgänge
  while(1){
    PORTJ = Schrittfunktion(PINA);
    Warte_1s();
  }
}
```

Durchzuführende Tests

- Übersetzen und im Debugger starten.
- Test freilaufend. Sollfunktion: LED-Lauflicht mit Schalter zur Richtungsumschaltung.
- Test auch im Schrittbetrieb und mit Unterbrechungspunkten.

Ein Vorteil gegenüber dem Test des bisherigen Programms ist, dass sich die Wartefunktion im Schrittbetrieb auch mit "Step-Over" und nicht nur mit »Abarbeitung bis zum nächsten Unterbrechungspunkt« überspringen lässt.