Informatikwerkstatt, Foliensatz 2
C-Programmierung

G. Kemnitz

Institut für Informatik, TU Clausthal (IW-F2)

16. November 2015

Inhalt Foliensatz 2

C-Programmierung

Wiederholung Bitverarb.

Wiederholungsaufgabe 2.1

```
#include <avr/io.h>
int main(){
 // Port-Initialisierung Variablenvereinbarungen
  DDRB = 0b00110011;
  uint8_t a;
  while(1){
  // Eingabe ⇒ Verarbeitung ⇒ Ausgabe


  }
}
```

1. Was passiert, wenn die Include-Anweisung fehlt?
2. Welche Pins von Port B sind Ein- und welche Ausgänge?
3. Anweisungen so ergänzen, dass an allen Ausgängen eine 1 ausgegeben wird, wenn alle Eingänge 0 sind, und sonst alles 0 ausgegeben wird.

Wiederholungsaufgabe 2.2

```
#include <avr/io.h>
int main(){
// Port-Initialisierung Variablenvereinbarungen


  while(1){
  // Eingabe ⇒ Verarbeitung ⇒ Ausgabe


  }
}
```

Ergänzen Sie

1. Initialisierung Port A als Ein- und Port J als Ausgang.
2. Endlosschleife mit der Schrittfunktion:

   `PJ.4 = PA.0` $\oplus$ `PA.1` ($\oplus$ – Exor)

   An die anderen Bits von Port J soll null ausgegeben werden.

Wiederholungsaufgabe 2.3

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

1. Die Anschlüsse PA.0 und PA.1 seien Ein- und alle Anschlüsse von Port J sowie die restlichen Anschlüsse von Port A Ausgänge.
2. Schrittfunktion:
   - Wenn (PA.0==1) $\wedge$ (PA.1==0): Erhöhung der Ausgabe an Port J um eins.
   - Sonst wenn (PA.1==1): Verringerung der Ausgabe an Port J um eins.
   - Sonst Ausgabe unverändert.
3. Übergangsereignis alle 2 s. Programmierung mit Warteschleife.

```
#include <avr/io.h>
int main(){
// Port-Initialisierung Variablenvereinbarungen


  while(1){
  // Eingabe ⇒ Verarbeitung ⇒ Ausgabe




  }
}
```

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	$[-2^{15}, 2^{15} - 1]$	

- Byte-Anzahl / Vorzeichen der C-Typen »int« und »char«?

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. uint16_t):

```
uint16_t *ptr;
```

- Vereinbarung eines Zeigers für beliebige Adressen:

```
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 Adressen im Datensegment. Existieren während der gesamten Programmlaufzeit.
- Lokal: Innerhalb einer Funktion vereinbart. Existieren nur während der Funktionsabarbeitung. Speicherplatz wird erst bei Funktionsaufruf auf dem sog. Stack reserviert.
- Lokale Variablen werden relativ zum Frame-Pointer (in unserem Prozessor Registerpaar r28:r29) adressiert.

```
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)

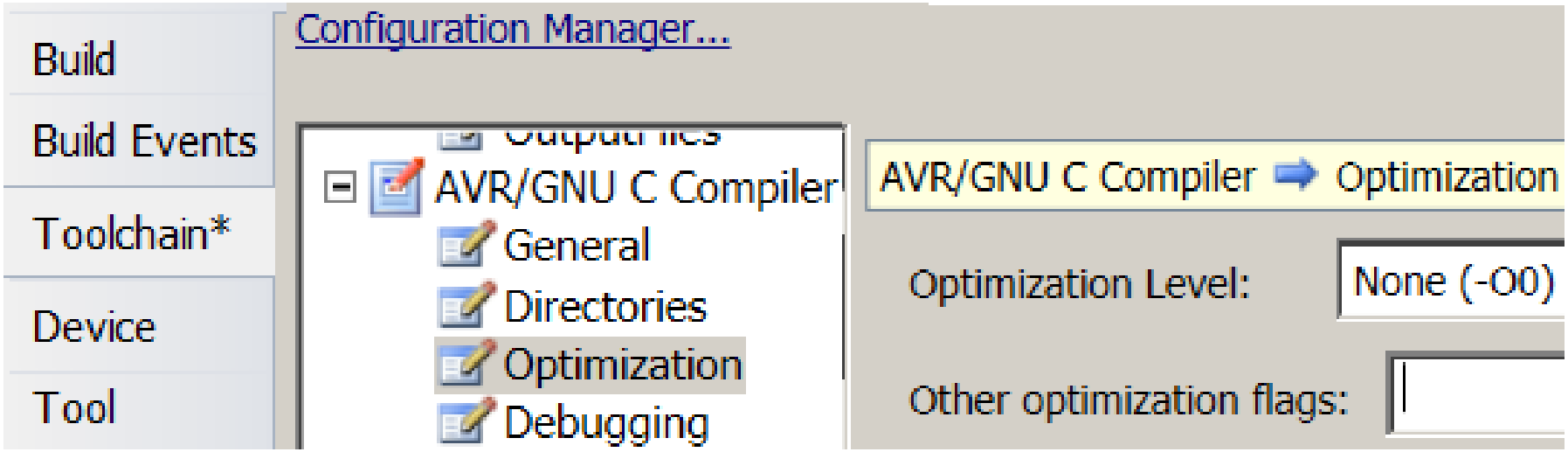
Experiment

Öffnen Sie das Projekts »glvar« und die Datei »glvar.c«:

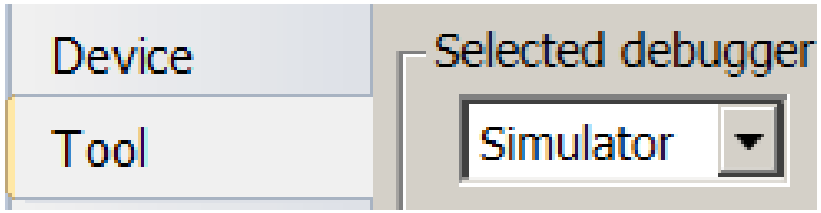
```
#include <avr/io.h>
int16_t gi16;              // global 2 Byte, VZ, AW 0
uint8_t gu8;               // global 1 Byte, NVZ, AW 0
int main(void){
 uint8_t  lu8  = 0x2D;  // 1 Byte, NVZ, AW 0x2D
 int16_t  li16 = 0x51F4;// 2 Byte, VZ, AW 0x51F4
 uint8_t *lpu8 = &gu8;  // Zeiger auf uint8_t,
                        // 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 > glvr 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}@0x020(
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	

Typ- und Wertekontrollen

Typfremde Zuweisung 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;
```

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 int8_t oder uint8_t sein
uint8_t u, *u_ptr;   // in der Tool-Chain als uint8_t
int8_t i, *i_ptr;    //  definiert

int main(void){
  c=u;               // laut Toolchain korrekt
  c=i;               // laut Toolchain falsch, keine Warnung
  u = i;             // unzulässig, keine Warnung
  c_ptr = &c;        // zulässig, keine Warnung
  c_ptr = &u;        // laut Toolchain korrekt, Warnung
  c_ptr = (char*)&u; // mit Typecast, keine Warnung
  c_ptr = &i;        // laut Toolchain falsch, Warnung
  c_ptr = (char*)&i; // mit Typecast, keine Warnung
}
```

Fazit

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

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?
```

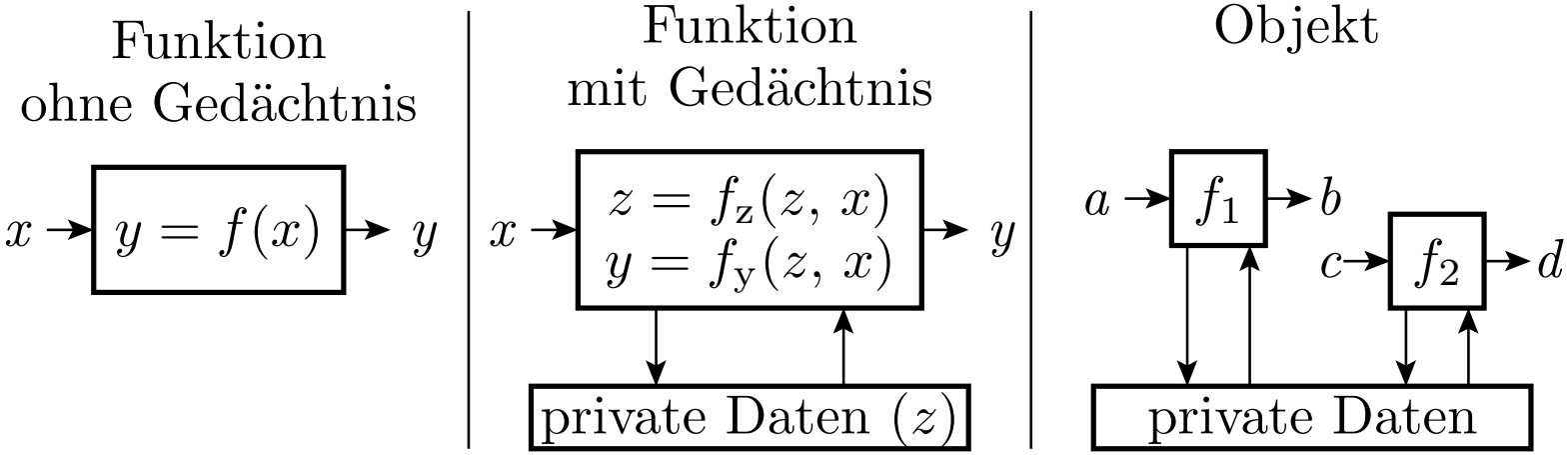
- Was erlaubt der Compiler, wofür gibt er Warnungen aus?
- Was verursacht bei der Abarbeitung Probleme?
- Unter welchen Bedingungen arbeiten die Programme trotzdem richtig?

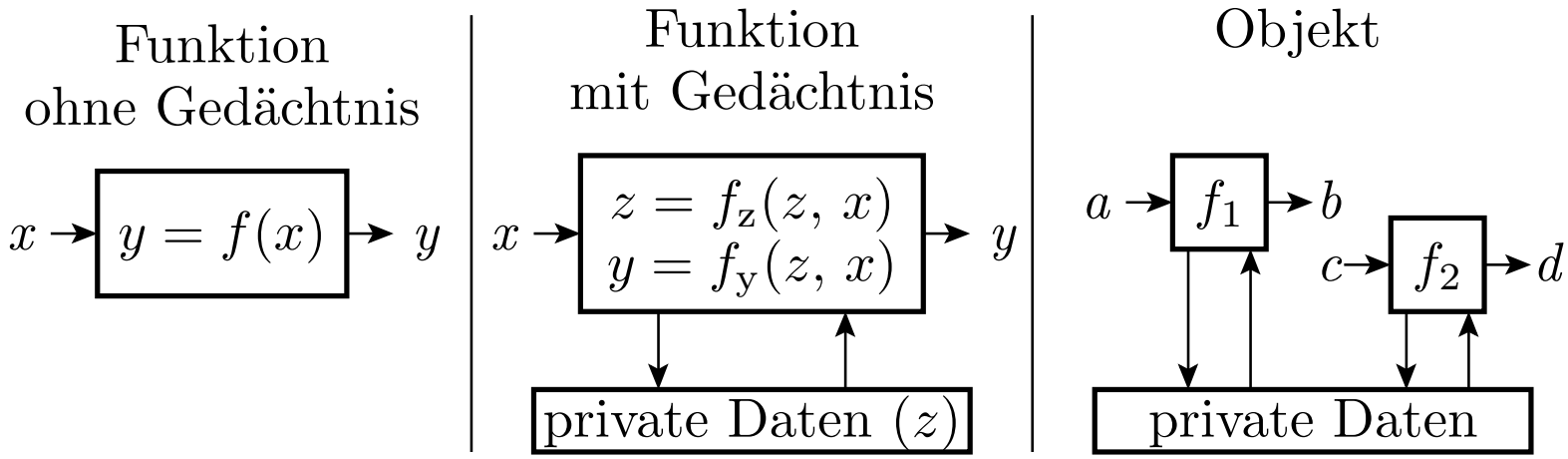
Modularisierung

Modularisierung

Größere Programme bestehen aus Modulen:

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





Grundprinzipien der Fehlervermeidung, Wartbarkeit, ...:

- Kapselung: Kein Lese- oder Schreibzugriff fremder Programme auf private Daten.
- Abstraktion: Übergeordnete Module kennen nur die Schnittstellen genutzter Module, nicht aber deren Realisierung und die Codierung der privaten 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];
}
```

Nutzung zum Kopieren einer Zeichenkette:

```
uint8_t a[] = "Text";
uint8_t b[8];
...
bytecopy(b, a, 4); // a, b: Zeiger auf Feldanfang
```

Kapselung von Funktionen und Objekten

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

Aufgabe zum Ausprobieren:

- Kopieren Sie aus dem Projektverzeichnis »bit_io3« die C- und die Dateien »bit_io3.c« und »bit_io3.cproj« in ein neu anzulegendes Projektverzeichnis »bit_io3_mod«.
- Öffnen des neuen Projekts.
- Erzeugung einer neuen C-Datei »myfkt.c« und einer neuen Header-Datei »myfkt.c«[1].

[1] Solution Explorer > bit_io3 auswählen > Rechts-Klick > Add > New Item > ...

- Aufteilung des C-Programms aus bit_io3.c:

```
#include <avr/io.h>
uint32_t Ct;          // 32-Bit-Zähler
uint8_t a=1;          // 8-Bit Ausgabewert
int main(void){       // Initialisierung
  DDRA = 0;           // Port A (Schalter) Eingänge
  DDRJ = 0xFF;        // Port J (LEDs) Ausgänge
  while(1){           // Warteschleife
    for (Ct=0; Ct<200000; Ct++);
    if (PINA & 0b1) // Rotation links
      a = (a<<1) | (a>>7);
    else              // Rotation rechts
      a = (a>>1) | (a<<7);
    PORTJ = a;        // Ausgabe
  }
}
```

Datei: myfkt.c

```
#include <avr/io.h>
uint8_t a=1;  // private globale Daten

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(){
uint32_t Ct;
for (Ct=0; Ct<200000; Ct++);
}
```

Datei: myfkt.h

```
#ifndef MYFKT_H_
#define MYFKT_H_
  uint8_t Schrittfunktion(uint8_t x);
  void Warte_1s();
#endif /* MYFKT_H_ */
```

Datei: bit_io3_mod.c

```
#include <avr/io.h>
#include "myfkt.h"
int main(void){
  DDRA = 0;          // Port A (Schalter) Eingänge
  DDRJ = 0xFF;       // Port J (LEDs) Ausgänge
  while(1){
    PORTJ = Schrittfunktion(PINA);
    Warte_1s();
  }
}
```

Simulation Modultest

Simulation von Modultests

Ein Modultest stellt Eingaben bereit, arbeitet das zu testende Modul ab und wertet die Ausgaben aus. Testbeispiel sei eine Funktion mit Aufrufliste, z.B.:

```
uint32_t quad(int16_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
};
```

Eine Testsatz als Menge von Tests ist in C ein initalisiertes Feld:

```
struct test testsatz[] ={{<Tupel 1},{...},...};
```

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
};
```

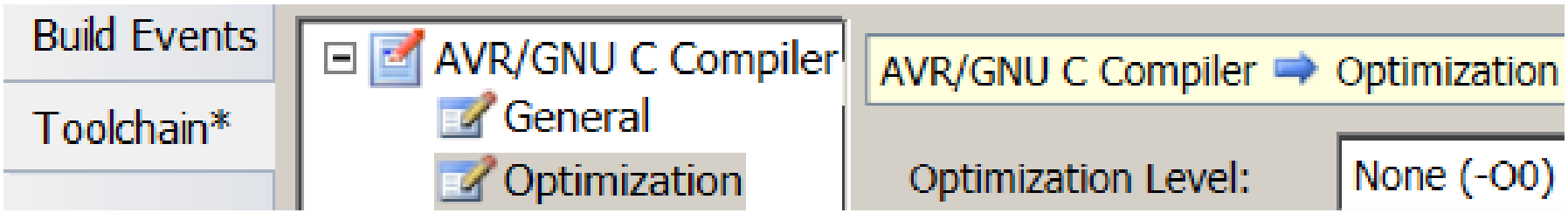
Der Testrahmen ist ein Hauptprogramm, ...:

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

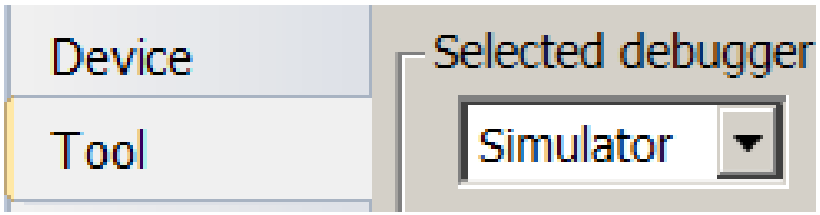
Projekt »mtest_quad« ausprobieren

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

```
Project > glvr 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, 0c07, 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);
```

Füllen Sie dazu die Tabelle auf der nächsten Folie aus.

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

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 im Schrittbetrieb mit dem Simulator.

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.

- Ergänzen Sie in der nachfolgenden Tabelle mit Testbeispielen 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.