



Informatikwerkstatt, Foliensatz 2

C-Programmierung

G. Kemnitz

Institut für Informatik, TU Clausthal (IW-F2)

24. Oktober 2023



Inhalt:

Wiederholung
Variablen

Typecast
Aufgaben

Interaktive Übungen:

- Globale und lokale Variablen (glvar).



Wiederholung



W2.1: Hexadezimal- und Binärzahlen



1 Zuordnung der Hexadezimalziffern:

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

2 Umrechnung nach binär:

$$0x15 = 0b \dots | \dots$$

$$0xAF2 = 0b \dots | \dots | \dots$$

$$0xABCD = 0b \dots | \dots | \dots | \dots$$

3 Umrechnung nach hexadezimal:

$$0b10010110110 = 0x \dots$$

$$0b0100111001011 = 0x \dots$$

$$0b00110110 = 0x \dots$$



Lösung

1 Zuordnung der Hexadezimalziffern:

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

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:

$$0b100.1011.0110 = 0x4B6$$

$$0b0.1001.1100.1011 = 0x9CB$$

$$0b0011.0110 = 0x36$$



W2.2: 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 .

```

Vervollständigen Sie die Kommentare.



1. Wiederholung

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=0b1.0.1.1|0.0.1.1 =0xB3
```

```
d = c ^ 0b01100111; //d=0b1.1.0.1|0.1.0.0 =0xD4
```

```
e = ~d;           //e=0b0.0.1.0|1.0.1.1 =0x2B
```

```
f = e >> 2;      //f=0b0.0.0.0|1.0.1.0 =0x0A
```

```
g = f << 1;      //g=0b0.0.0.1|0.1.0.0 =0x14
```



Programm vervollständigen



```
1 #include <avr/io.h>
2 uint8_t a;           //Variablenvereinbarung
3 int main(){
4     DDRA =           ; //Port A Eingänge
5     DDRJ =           ; //Port J Ausgänge
6     uint8_t b;
7     while(...){    //Endlosschleife
8         a =         ; //Eingabewerte lesen
9                 //a.0=(a.0&a.1)|(a.2&a.3)
10                ;
11                ; //Ausgabe an Led 0 ohne
12    }              //andere Led's am Port J
13 }              //zu ändern
```

1 Was passiert, wenn die Include-Anweisung fehlt?

2 Welchen Wertebereich hat die Variable a?

4, 5 Was ist zuzuweisen?

6 bis 11 $PORTJ.0 = (a.0 \wedge a.1) \vee (a.2 \wedge a.3)$

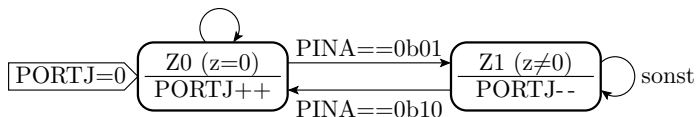


Lösung

- 1 Compiler meldet DDRB, PINB oder PORTB nicht definiert.
- 2 Vervollständigtes Programm:

```
#include <avr/io.h>
uint8_t a;           //Variablenvereinbarung
int main(){
    DDRA = 0         ; //Port A Eingänge
    DDRJ = ~0        ; //Port J Ausgänge
    uint8_t b;
    while( 1 ){     //Endlosschleife
        a = PINA     ; //Eingabewerte lesen
        a = (a & (a>>1)) | //a.0=(a.0&a.1)|(a.2&a.3)
              ((a>>2)&(a>>3));
        PORTJ = (PORTJ&(~1)) | (a&1); //Ausgabe an Led 0
    }               //ohne andere Led's am
}                  //Port J zu ändern
```

Vor/Rückwärtszähler



Initialisierung:

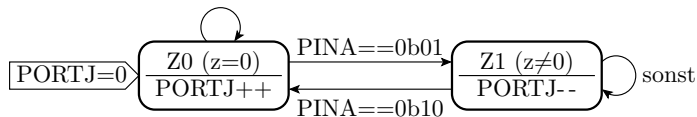
- Startzustand: z , genutzter Wertebereich $\{0, 1\}$, Anfangswert 0
- Port A Anschl. 0 und 1 Eingänge. Port J alle Anschl. Ausgänge.

Schrittfunktion:

- Zustand Z0 ($z = 0$):
 - Port J aufwärts zählen.
 - Wenn $(a.0 = 1) \wedge (a.1 = 0)$ wechsel nach Z1 ($z = 1$).
- Sonst (Zustand Z1, $z = 0$):
 - Port J abwärts zählen.
 - Wenn $(a.0 = 0) \wedge (a.1 = 0)$ wechsel nach Z0 ($z = 0$).
- Verlängerung der Schrittdauer auf ≈ 2 s (Warteschleife).



1. Wiederholung



```
1 #include <avr/io.h>
2 ...     z=0;           // WB: 0 bis 1
3 ...     Ct;           // Zähler Warteschl. 0 .. 400000
4 int main(){
5     DDRA = ...        ; // PA0 und PA1 Eingänge
6     DDRJ = ...        ; // Port J Ausgänge
7     PORTJ = ...       ; // Anfangswert 0
8     ...               { // Endlosschleife
9     ...               // Übergangsfunktion siehe
10    ...               // nächste Folie
11 }
12 }
```

2, 3 Datentyp, bzw. wie viele Bytes erforderlich?

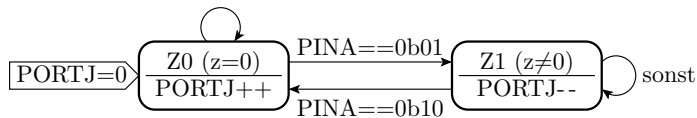
5 bis 7 Welche Werte sind den Spezialregistern zuzuweisen?

8 Programmzeile vervollständigen

11 und 12 In welcher Zeile beginnen die hier endenden Blöcke?



1. Wiederholung



```

1  if (...          ){           // Wenn Zustand Z0
2  ...                ;// Port J aufwärts zählen
3  if (...          )           ;// PINA==0b01, Folgezust. Z1
4  }
5  else {              // sonst (nicht Zustand Z0)
6  ...                ;// Port J abwärts zählen
7  if (...          )           ;// PINA==0b01, Folgezust. Z0
8  }
9  for (...          );// Warteschleife ca. 2s
  
```

- 1 Bedingung »z ist 0« ergänzen.
- 2 Port J aufwärts zählen.
- 3 Bedingung $(a.0 = 1) \wedge (a.1 = 0)$ und Folgezustand »Z1«.
- 6 Port J abwärts zählen.
- 7 Bedingung $(a.0 = 0) \wedge (a.1 = 0)$ und Folgezustand »Z0«.
- 9 Zähler Ct von 0 bis 400.000 zählen lassen.



1. Wiederholung

Lösung:

```
#include <avr/io.h>
uint8_t z=0;
uint32_t Ct;
int main(){
    DDRA = ~0x03;           // PA0 und PA1 Eingänge
    DDRJ = 0xFF;           // Port J Ausgänge
    PORTJ = 0;              // Anfangsausgabewert
    while(1){               // Endlosschleife
        if (z == 0){        // Wenn Zustand Z0
            PORTJ++;         // Port J aufwärts zählen
            if (PINA&3==1) z=1; // wenn PINA==0b01, Folgezust. Z1
        } else {            // sonst (nicht Zustand Z0)
            PORTJ--;         // Port J abwärts zählen
            if (PINA&3==2) z=0; // wenn PINA==0b10, Folgezust. Z0
        }
        for (Ct=0; Ct<400000; Ct++); // Warteschleife
    } // Welcher Block endet hier?
} // Welcher Block endet hier?
```



Variablen

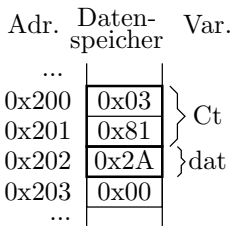


Variablen

- Variablen sind Symbole für Adressen von Speicherplätzen, die beschrieben und gelesen werden können.
- Eine Variablenvereinbarung definiert Typ (z.B. `uint8_t`), Name (z.B. `dat`) und optional einen Anfangswert (z.B. `0x2A`):

```
uint8_t dat = 0x2A;
```

- Der Typ legt fest, wie viele Bytes zur Variablen gehören (z.B. 1, 2 oder 4) und was die Bytes darstellen (z.B. eine ganze Zahl ohne oder mit Vorzeichen).



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



Kontrollfragen



- Welche Byteanzahl und Wertebereiche haben »uint32_t« und »int32_t (4 Byte ohne/mit Vorzeichen)?
- Was vermuten Sie, welchen der eingeführten Typen

$$(u)int < n > _t$$

entsprechen die C-Standardtypen in der nachfolgenden Tabelle?

C Basistyp	$(u)int < n > _t$	C Basistyp	$(u)int < n > _t$
char		unsigned char	
short int		unsigned short int	
long int		unsigned long int	
int		unsigned int	

Hinweis: »chr« steht für char (Textzeichen). Anzahl unterschiedliche Textzeichen $\leq 2^8$, $\leq 2^{16}$, ...? »int« steht für integer, ganze Zahl.



Wert und Adresse einer Variablen

Werte und Adressen von Variablen sind 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

- »ptr« ist ein Zeiger (Variable für eine Adresse). Der vereinbarte Typ eines Zeigers ist der Typ der Variablen, deren Adressen der Zeiger speichern darf. Typ »void« für »beliebiger Typ«.
- Byteanzahl Adresse: $\geq \log_2(AS)$ (AS – Anzahl Speicherplätze. Unser Prozessor 2-Byte Adressen, genutzter Adressraum 0 bis 0x1FF Spezialregister, 0x200 bis 0x21FF Daten.
- In der aufgeklappten Zeile unter der Zeiger-Variablen stehen Wert und Adresse der adressierten Variablen.



Vereinbarung und Verwendung von Zeigern

- Vereinbarung mit dem Referenzierungsoperator »*« (Inhalt, auf den der Zeiger zeigt (auf den der Zeiger zeigt (...)) hat den Typ:

```
uint8_t a;           // Variable
uint8_t *ptr;       // Zeiger auf uint8_t-Variable
uint8_t **pptr;    // Zeiger auf Zeiger auf uint8_t
```

- Zeiger für beliebige Datentypen (auch Programmadressen¹):

```
void *vptr;
```

- Adressoperator: & (liefert Adresse des Datenobjekts rechts)

```
ptr = &a; pptr = &ptr;
```

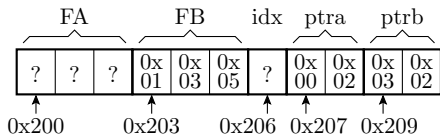
- Referenzierung: * (liefert den Wert der Adresse rechts)

```
a = *ptr; ptr = *pptr;
```

¹Üblicher Weise nur Funktionszeiger zum Start von Unterprogrammen, die am Ende zum aufrufenden Programm zurück springen. Besser Funktionszeigertyp vereinbaren.



Felder und Schleifen



- Feld: Zusammenfassung gleicher Datenobjekte:

```

1  uint8_t FA[3];           // Felder mit 3 Elementen
2  uint8_t FB[3] = {1,3,5}; // initialisiertes Feld
3  uint8_t idx;            // Feldindex
4  uint8_t *ptra = FA;     // Zeiger auf Feldanfang
5  uint8_t *ptrb = FB;     // Zeiger auf Feldanfang

```

- Kopierschleife Feld »FB« nach Feld »FA«:

```

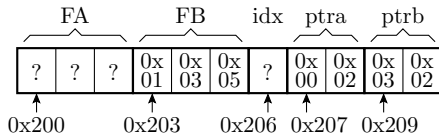
1  #define size_FA sizeof(FA)/sizeof(uint8_t)
2                               // Precompiler-Definition
3  for (idx=0; idx < size_FA; idx++){
4    FA[idx] = FB[idx]; // kopiere Element i
5  }

```

Hinweis zu Zeile 3: Die Anzahl der Feldelemente ist immer »Bytanzahl Feld« durch »Bytanzahl Element«



2. Variablen



- Feldelement »FA[i]« ist dasselbe wie »Inhalt von Feldanfang plus Elementennummer *(FA+i)«:

```
1 for (idx=0; idx < size_FA; idx++){
2   *(FA+idx) = *(FB+idx);
3 }
```

- Programmoptimierung durch »Zeiger weitererschalten«:

```
1 ptra = FA;           // Zeiger auf Anfang Feld A
2 ptrb = FB;           // Zeiger auf Anfang Feld B
3 while(ptra < FA + size_FA){
4   *ptra = *ptrb;     // Inhalt ptra = Inhalt ptrb;
5   ptra++; ptrb++;    // Zeiger auf nächstes Element
6 }
```

Zeile 3: »FA + size_FA« erste Speicheradresse hinter dem Feld.



Globale (statische) 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:R29²).

```
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)

²Im Watchfenster steht nur [R28]. ...@0x21fa ([R28]+1) bedeutet Datenspeicheradresse 0x21fa und, dass im Frame-Pointer R28:R29 die Adresse 0x21fa -1, d.h. 0x21f9 steht.



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;
}
```



2. Variablen

- Übersetzen mit -O0

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



Configuration Manager...

Build

Build Events

Toolchain*

Device

Tool

AVR/GNU C Compiler

Optimization

AVR/GNU C Compiler → Optimization

Optimization Level: None (-O0)

Other optimization flags:

- Auswahl des Simulators als »Debugger«

Device

Tool

Selected debugger

Simulator

- 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.



2. Variablen

Werte 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}@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 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



2. Variablen

- 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



2. Variablen

- 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



2. Variablen

- 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;
}
```

Name	Value
gi16	0x51f5
gu8	0x29

Name	Value
lu8	0x2d
li16	0x51f4
lpu8	0x21fa



2. Variablen

- 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	



2. Variablen

- 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

Zweisungen an Variablen mit einem anderen Typ, z.B.:

```
uint16_t a; int16_t b;  
a = b; // Fehler für b<0
```

sind oft Programmierfehler. Wenn dennoch gewollt, Typecast:

```
a = (uint16_t)b; //Zuweisung mit Typecast
```

Beispiel Betragsbildung:

```
if (b<0) a = (uint16_t)(-b);  
else     a = (uint16_t)b;
```

Beispiel »nach WB-Verkleinerung«:

```
uint8_t a; uint16_t b;  
a = (uint8_t)(b>>8);
```




Fehler ohne Typecast

```
uint8_t  a;  
uint16_t b;  
...  
b = a<<8;           // ergibt immer null  
b = (uint16_t)(a<<8); // ergibt immer null  
b = ((uint16_t)a)<<8; // ergibt a * 256
```

Compiler erzeugen bei einigen, aber nicht allen typfremden Zuweisungen Warnungen oder Fehlermeldungen:

- Typ-Fehlerwarnungen nicht ignorieren,
- Nicht darauf verlassen, dass Compiler alle Typ-Fehler erkennt.



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. Toolchain uint8_t
int8_t i, *i_ptr;
int main(){ //Warnung, wenn nicht int, warum?
  c=u;    //laut Toolchain korrekt, keine Warnung
  i=c;    //laut Toolchain falsch, keine Warnung
  c_ptr = &c; //zulässig, keine Warnung
  c_ptr = &u; //laut Toolchain korrekt, Warnung
  c_ptr = (char*)&u; //Typcast, keine Warnung
  c_ptr = &i; //laut Toolchain falsch, Warnung
  c_ptr = (char*)&i; // Typcast, keine Warnung
}
```

Empfehlung: Verwendung von [u]int... und explizite Typcasts.



Aufgaben



Hausaufgabe

Vorbereitung auf den schriftlichen Montagstest. Themen³:

- Umwandlung zwischen dezimaler, hexadezimaler und binärer Zahlendarstellung.
- Werte logischer Ausdrücke mit $\gg \sim \ll$, $\gg \& \ll$, $\gg | \ll$, $\gg ^ \ll$, $\gg > \ll$, $\gg < \ll$.
- Setzen und Löschen von Bits einer Variablen.
- Werte und Wertebereichsverletzung bei Zuweisung von Ausdrücken mit $\gg + \ll$ und $\gg - \ll$ an `[u]int<n>_t` Variablen.
- Vereinbarung von Zeigern und Feldern.
- Zeigerfehler bei Zuweisung von Ausdrücken mit $\gg \& \ll$ und $\gg * \ll$, z.B. Zuweisungsziel Zeiger, Typ des Ausdrucks ist aber Wert.

Ziel der Montagstests ist die Feststellung, welche Teilnehmer welche der Themen verstanden haben⁴.

³Erlaubte Hilfsmittel: Folien, eigene Notizen, Atmel Studio (Simulator) und Taschenrechner.

⁴Erfolgreiche Teilnehmer werden von weiteren Tests zum selben Thema freigestellt.



4. Aufgaben

Aufgabe 2.1: Ausführung Experiment ab Folie 22

```
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)@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	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



Aufgabe 2.2: 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.



4. Aufgaben

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 2.3: Untersuchung Zuweisungen



```
uint8_t a; int8_t b;
a = 56;
b = a; // Kommt die 56 richtig an?
a = 200;
b = a; //WB(b): [-128, 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?
-
- Projektanlegen, Programm vervollständigen und eingeben.
 - Mit »-O0« übersetzen und im Simulator im Schrittbetrieb starten.
 - Compiler-Warnungen und Übersetzungsfehler beseitigen.
 - Ursachen für falsch zugewiesene Werte beseitigen.



Aufgabe 2.4: Test einer Kopierfunktion

```
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 uint8_t a[] = "Text";
8 uint8_t b[] = "Welt";
9 uint8_t c[10], *ptr=c;

10 int main(){
11     bytecopy(ptr, a, 4);
12     ptr += 4;
13     *ptr = '\u0000';
14     ptr++;
15     bytecopy(ptr, b, 5);
16 }
```



4. Aufgaben

Arbeitsschritte:

- 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 »-OO«.
- Abarbeiten im Debugger im Schrittbetrieb, einmal mit »Step-Over« (Unterprogrammaufrufe als einen Schritt) und einmal mit »Step-Into« (Unterprogrammabarbeitung zeilenweise).

Aufgaben zur Kontrolle durch die HiWis⁵:

- 1 Auf welche Adressen zeigen die Pointer a, b und c zum Programmbeginn und nach Abarbeitung der einzelnen Hauptprogrammzeilen? (Test mit »Step-Over«)
- 2 Welche Zahlenfolgen stehen zum Programmbeginn in den Feldern a[] und b[] und nach Programmabschluss im Feld c[]?⁶

⁵ Antworten als Kommentare in das Programm schreiben.

⁶ Suchen Sie sich hierzu im Internet eine ASCII-Tabelle zur Kontrolle.



Experimentieren nach eigenen Vorgaben

- Jeder der Teilnehmer sollte versuchen die Aufgaben 2.1 bis 2.3 zu lösen.
- Mindestens eine der Aufgaben 2.1 bis 2.3 sind beim Betreuer abzurechnen.
- Für die verbleibende Übungszeit:
 - Wenn letzte Woche nichts abgerechnet, bitte nachholen.
 - Sonst ungelösten Aufgaben von Foliensatz 1 oder selbst gewählte Aufgaben bearbeiten.