

U3 Grundlagen der AVR-Programmierung

- Enums und Typedefs
- Deklaration und Definition
- Compileroptimierungen
- Bitoperationen

U3-1 Enums

- Der Enum-Typ (engl. Enumeration = Aufzählung)
 - ◆ Zuweisung eines Namens zu einem Integertyp
 - ◆ C erhöht Enums automatisch um 1
 - ◆ Beispiel:

```
enum LED_e{
    RED0    = 0,
    YELLOW0 = 1,
    GREEN0  = 2,
    BLUE0   = 3,
    RED1    = 4,
    YELLOW1 = 5,
    GREEN1  = 6,
    BLUE1   = 7
};
```

```
enum LED_e{
    RED0 = 0,
    YELLOW0,
    GREEN0,
    BLUE0,
    RED1,
    YELLOW1,
    GREEN1,
    BLUE1
};
```

U3-1 Enums (2)

- Verwendung von Enums

```
enum LED_e{
    RED0 = 0,
    YELLOW0,
    GREEN0,
    BLUE0,
    RED1,
    YELLOW1,
    GREEN1,
    BLUE1
};
```

```
enum LED_e meineLed;
meineLed = RED1;
meineLed++;
if(meineLed == YELLOW1){
    /* Dies wird ausgeführt */
}
```

- Achtung! C überprüft den Wertebereich von Enums nicht! `meineLed++`; kann also einen ungültigen Wert ergeben! Die Einhaltung des Wertebereichs ist Aufgabe des Programmierers.

U3-2 Typedefs

- Typedefs erlauben es, neue Variablentypen zu deklarieren
- Beispiel `stdint.h` der `libavr`:

```
typedef signed char int8_t;
typedef unsigned char uint8_t;
typedef signed int int16_t;
typedef unsigned int uint16_t;
typedef signed long int int32_t;
typedef unsigned long int uint32_t;
```

- Dies ist auch mit Enums oder Strukturen (werden später eingeführt) möglich
- Verkürzte Schreibweise aus der `led.h`:

```
typedef enum {
    RED0=0, YELLOW0=1, GREEN0=2, BLUE0=3,
    RED1=4, YELLOW1=5, GREEN1=6, BLUE1=7
} LED;
```

- Compiler arbeiten den Quelltext von oben nach unten ab
- Deklaration
 - ◆ Das "Versprechen", dass es eine Funktion/Variable geben wird, die einen bestimmten Rückgabewert hat und bestimmte Parameter übergeben bekommt.

```
uint8_t meineFunktion(uint8_t w1, uint16_t w2);
```

- Definition
 - ◆ Die eigentliche Funktion

```
uint8_t meineFunktion(uint8_t w1, uint16_t w2) {
    /* Hier passiert was */
}
```

- Die Funktionen der libspicboard werden in Headerdateien deklariert.

- Eine CPU arbeitet nicht direkt im Speicher:
 - (1) Laden aus dem Speicher in Register
 - (2) Abarbeiten der Operationen in den Registern
 - (3) Zurückschreiben in den Speicher
- Der Compiler macht Annahmen, um den Code zu optimieren. Beispiele:
 - ◆ Variableninhalte sind beständig. Sie ändern sich nicht "von alleine".
 - ◆ Operationen, die den Zustand nicht ändern, können entfernt werden.

- Typische Optimierungen:
 - ◆ Code wird weggelassen.
 - ◆ Die Reihenfolge des Codes wird umgestellt.
 - ◆ Für lokale Variablen wird kein Speicher reserviert; es werden stattdessen Register verwendet.
 - ◆ Wenn möglich, übernimmt der Compiler die Berechnung:
 - a = 3 + 5; wird zu a = 8;
 - ◆ Der Wertebereich wird geändert:
 - Statt von 0 bis 10 wird von 246 bis 256 (= 0 für uint8_t) gezählt und dann getestet, ob ein Überlauf stattgefunden hat.

- Codebeispiel

```
void wait(void) {
    uint8_t u8;
    while(u8 < 200) {
        u8++;
    }
}
```

■ Codebeispiel ohne Optimierung:

```

;void wait(void){
; uint8_t u8;
; [Prolog (Register sichern, etc)]
;   rjmp while;   Springe zu while
; u8++;
addone:
;   ldd r24, Y+1; Lade Daten aus Y+1 in Register 24
;   subi r24, 0xFF; Ziehe 255 ab (addiere 1)
;   std Y+1, r24; Schreibe Daten aus Register 24 in Y+1
; while(u8 < 200)
while:
;   ldd r24, Y+1; Lade Daten aus Y+1 in Register 24
;   cpi r24, 0xC8; Vergleiche Register 24 mit 200
;   brcs addone; Wenn kleiner dann springe zu addone
; [Epilog (Register wiederherstellen)]
;   ret;   Kehre aus der Funktion zurück
}
    
```

SPIC-Ü

■ Codebeispiel mit Optimierung:

```

; void wait(void){
;   ret;   Kehre aus der Funktion zurück
; }
    
```

- Die Schleife hat keine Auswirkung auf den Zustand.
- Lösung: Variable als volatile (engl. unbeständig) deklarieren
 - ◆ Für Variablen bedeutet dies: Sie müssen immer in den Speicher gelegt und vor und nach jeder Operation mit diesem synchronisiert werden; ihr Wertebereich darf nicht geändert werden.

SPIC-Ü

■ Einsatzmöglichkeiten von volatile:

- ◆ Warteschleifen

```

void wait(void){
;   volatile uint8_t u8;
;   while(u8 < 200){
;       u8++;
;   }
}
    
```

- ◆ Zugriff auf Hardware (z. B. Pins). Wird in der nächsten Tü besprochen.
- ◆ Debuggen; der Wert wird nicht wegoptimiert.

SPIC-Ü

U3-5 Logische Operatoren

■ Logische Operatoren:

"nicht"		"und"			"oder"		
!		&&	f	w		f	w
f	w	f	f	f	f	f	w
w	f	w	f	w	w	w	w

SPIC-Ü

U3-5 Bit-Operatoren

■ Bit-Operatoren

"nicht"	"und"	"oder"	"x-oder"
\sim	$\&$	$ $	\wedge
f w	f f f	f f w	f f w
w f	w f w	w w w	w w f

■ Beispiel:

	1100	1100	1100
\sim	$\&$	$ $	\wedge
1001	1001	1001	1001
0110	1000	1101	0101

U3-5 Shift-Operatoren

➔ Bits werden im Wort verschoben

\ll Links-Shift
 \gg Rechts-Shift

■ Beispiel:

x	1	0	0	1	1	1	0	1
x << 2	0	1	1	1	0	1	0	0
x >> 2	0	0	0	1	1	1	0	1