Inhaltsverzeichnis

Vielen Dank an Reinhold Birk, Gottlieb-Daimler-Schule 2 für die Bereitstellung der Experimentierplatine und der Bibliothek XMC1100-Lib sowie das Expertenwissen in jeder Situation.

1 C lernen mit Programmbeispielen

1.1 Lauflicht mit Schiebebefehl

1.1.1 Konzept

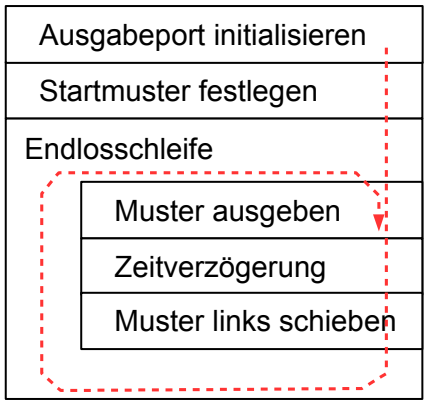
Eine leuchtende LED an P0 „soll von rechts nach links geschoben werden“, 1 = leuchtende LED.
0000 0000 0000 0001 → 0000 0000 0000 0010 → 0000 0000 0000 0100 → usw.
Dazu wird eine Variable „muster“ definiert, in der das aktuelle Ausgabemuster gespeichert wird.
unit16_t muster;
Zu Beginn muß das gewünschte Ausgabemuster in die Variable geschrieben werden:
muster = 1; // als Dualzahl 0000 0000 0000 0001
Nach der Ausgabe wird dieses Muster um eine Stelle nach links verschoben
muster << 1;
Allein dieser Befehl reicht nicht, denn das um eine Stelle verschobene Muster muss anschließend wieder gespeichert werden:
muster = muster << 1; // nach dem ersten Schieben steht nun 0000 0000 0000 0010 in der
// Variablen, nach dem nächsten Ausführung 0000 0000 0000 0100 usw.

1.1.2 Programm Lauflicht

```
#include <XMC1100-Lib.h>          // Hilfsfunktionen fuer XMC1100

uint16_t muster;                  // Speichervariable für Lauflichtmuster
int main(void)                    // Hauptprogramm
{
  port_init(P0,OUTP);             // Port0 auf Ausgabe programmieren
  muster = 1;                     // als Dualzahl 0000 0000 0000 0001(an LEDs sichtbar)
  while(1U)                       // Endlosschleife, immer bei Controllern
  {
          port_write(P0,muster);  // Muster an Port0 ausgeben
          delay_ms (100);         // Zeitverzoegerung 500ms
          muster = muster << 1;   // Muster um 1 Stelle nach links schieben
                                  //  0000 0000 0000 0010
                                  //  0000 0000 0000 0100 usw.
          //muster = ((muster&0x8000)>>15)|(muster<<1); // Ringschieben
  }//while
}//main
```

1.2 Struktogramm

Eine grafische Beschreibung bietet das Struktogramm, in dem die Struktur des Programms deutlich wird. Die Aktionen werden im Klartext angegeben. Hier werden keine einzelnen Befehle beschrieben, sondern die Wirkungsweise des Programms erklärt. Die rot gepunkte Linie gehört nicht zum Struktogramm, Sie soll zeigen, in welcher Reihenfolge das Programm abgearbeitet wird und wie das Struktogramm zu lesen ist.



Das höchstwertigste, linke Bit „geht beim Schiebevorgang verloren“. Dies hat zur Folge, dass nach 16 mal schieben die gesamte Information einer 16-Bit-Variablen „herausgeschoben“ wurde und verloren geht. Alle LEDs sind dann aus.
Von rechts wird immer eine 0 „nachgeschoben“.

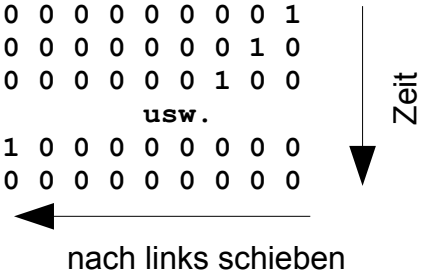
Mit folgender Befehlsfolge können wir das Bit, das links „rausfällt“, rechts „wieder reinschieben“:
muster = ((muster & 0x8000)>>15) | (muster<<1)

Diese Befehlsfolge können wir erst später verstehen. (&, | sind bitweise logische Verknüpfungen.)

1.3 Verzweigung mit if und else

1.3.1 Aufgabe und Konzept

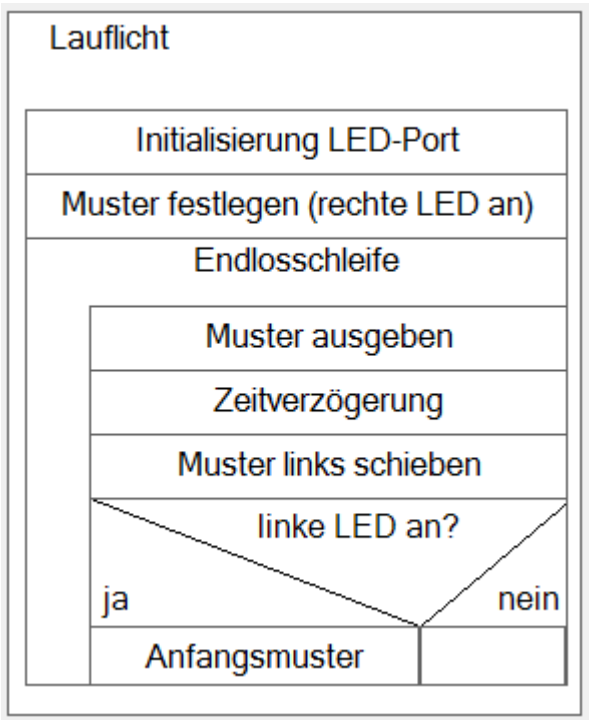
Das Lauflicht 1.1.2 wird leider beendet, wenn 15 mal nach links geschoben wurde, weil „von rechts“ immer nur Nullen „nachgefüllt“ werden.



In diesem Programm soll das Lauflicht wieder automatisch von rechts beginnen. Dazu benötigt man die Abfrage: Ist die linke LED an? Wenn ja, dann soll wieder mit dem ersten Bitmuster begonnen werden.

Eine solche Programmstruktur nennt sich Verzweigung, da der Programmablauf sich verzweigt: es wird entweder das eine oder das andere ausgeführt.

Rechts ist das zugehörige Struktogramm dargestellt.



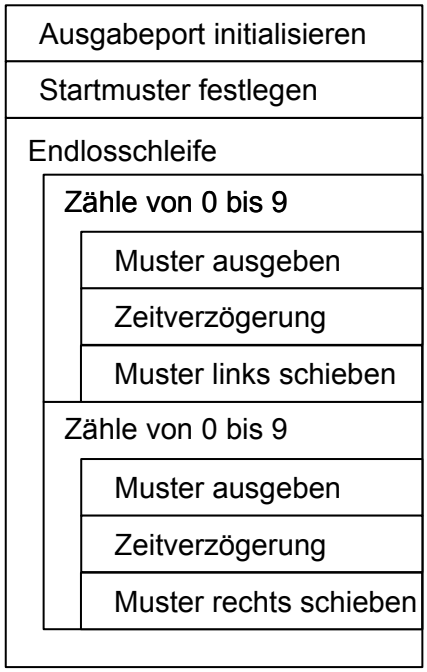
1.3.2 Programm

```
#include <XMC1100-Lib.h>          // Hilfsfunktionen fuer XMC1100

uint16_t muster;                  // Speichervariable für Lauflichtmuster
int main(void)                    // Hauptprogramm
{
  port_init(P0,OUTP);             // Port0 auf Ausgabe programmieren
  muster = 1;                     // als Dualzahl 0000 0000 0000 0001(an LEDs sichtbar)
  while(1U)                       // Endlosschleife, immer bei Controllern
  {
          port_write(P0,muster);  // Muster an Port0 ausgeben
          delay_ms (100);         // Zeitverzoegerung 500ms
          muster = muster << 1;   // Muster um 1 Stelle nach links schieben
                                  //  0000 0000 0000 0010
                                  //  0000 0000 0000 0100 usw.
          if (muster == 0x8000) muster = 1;
  }//while
}//main
```

1.4 Zählschleife mit FOR: Anzahl der Verschiebungen festlegen

1.4.1 Aufgabe und Konzept

Das Lauflichtmuster soll periodisch 10 mal nach links und anschließend 10 mal nach rechts geschoben werden. Dazu ist eine Zählschleife nowendig.



Jede Zählschleife benötigt eine Zählvariable, die zuvor definiert werden muss:

```
uint16_t  z;                    // Zählvariable
```

Die Zählvariable wird z genannt und ist vom Typ vorzeichenlose ganze Zahl mit 16 Bit: uint16_t

Die Zählschleife ist folgendermaßen aufgebaut:
for (z=0; z<10; z++)

Dabei bedeuten die Angaben folgendes: z beginnt bei 0 zu zählen (z=0) und wird solange die Bedingung z<10 erfüllt ist bei jedem Schleifendurchlauf um eins erhöht (z++).

Rückwärts zählen wird so angegeben werden: for (z=10; z>0; z--)

Die Befehle, die in der Schleife auszuführen sind, werden mit { } eingeschlossen.

```
 for (z=0;z<10;z++)                 // 10 mal schieben
         {
           port_write(P0,muster);   // Muster an Port0 ausgeben
           delay_ms (100);          // Zeitverzoegerung 100ms
           muster = muster << 1;    // Muster um 1 Stelle nach links
         }
```

1.4.2 Gesamtes Programm

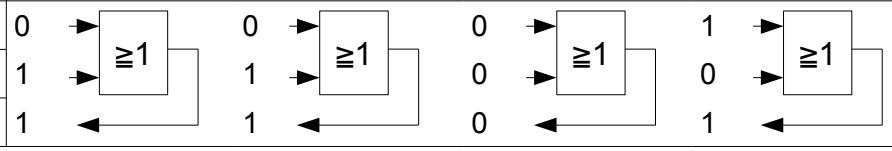
```
/* Lauflicht mit 1 LED, FOR-Befehl kennen lernen */
#include <XMC1100-Lib.h>// Hilfsfunktionen fuer XMC1100

uint16_t z,muster;                  // Laufvariable, Speichervariable für Lauflichtmuster
int main(void)                      // Hauptprogramm
{
  port_init(P0,OUTP);               // Port0 auf Ausgabe programmieren
  muster = 1;                       // als Dualzahl 0000 0000 0000 0001
  while(1U)                         // Endlosschleife, immer bei Controllern
  {
    for (z=0;z<10;z++)              // 10 mal schieben
         {
           port_write(P0,muster);   // Muster an Port0 ausgeben
           delay_ms (100);          // Zeitverzoegerung 100ms
           muster=muster<<1;        // Muster um 1 Stelle nach links
         }
    for (z=0;z<10;z++)              // 10 mal schieben
         {
           port_write(P0,muster);   // Muster an Port0 ausgeben
           delay_ms (100);          // Zeitverzoegerung 100ms
           muster=muster>>1;        // Muster um 1 Stelle nach rechts
         }
  }//while
}//main
```

1.5 Bitweise logische Verknüpfungen

1.5.1 Beispiel ODER

```
muster = 0b0001;          // Wert vor der Verküpfung
muster = muster | 0b1100; // bitweise logische ODER-Verknüpfung
port_write (P0,muster);   // an den LEDs sieht man 1101
```

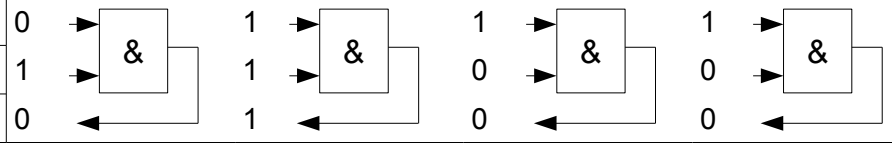
	Binäre Darstellung			
muster = 0b0001;	0	0	0	1
Konstante 0b1100	1	1	0	0
Wert muster nachher	1	1	0	1



1.5.2 Beispiel UND

```
muster = 0b0111;          // Wert vor der Verküpfung
muster = muster & 0b1100; // bitweise logische UND-Verknüpfung
port_write (P0,muster);   // an den LEDs sieht man 0100
```

	Binäre Darstellung			
muster = 0b0111;	0	1	1	1
Konstante 0b1100	1	1	0	0
Wert muster nachher	0	1	0	0



1.5.3 Lauflicht mit „Thermometercode“

Funktionsweise des Programms wie 1.4.1, jedoch soll das LED-Muster sich auf folgende Art ändern:

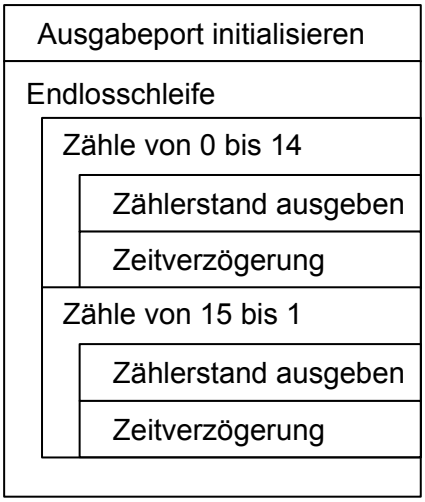
```
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 1 1
 usw.
1 1 1 1 1 1 1 1 1
0 1 1 1 1 1 1 1 1
 usw.
0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1
 usw.
```

1.5.4 Lauflicht mit Ringschieben

Nur Schieben in eine Richtung. Die links „herausfallenden“ Bits sollen rechts wieder „reingeschoben“ werden.

1.5.5 Variablenwert der Zählschleife anzeigen

Statt des Lauflichtmusters soll nun der Wert der Zählschleife angezeigt werden. Setzen Sie das Struktogramm in ein C-Programm um und testen Sie es.




```
/* Zaehlvariable ausgeben, FOR-Befehl kennen lernen */
#include <XMC1100-Lib.h>              // Hilfsfunktionen fuer XMC1100

uint16_t z;                           // Zaehlvariable
int main(void)                        // Hauptprogramm
{
  port_init(P0,OUTP);                 // Port0 auf Ausgabe programmieren
  while(1U)                           // Endlosschleife, immer bei Controllern
            {
            for (z=0;z<15;z++)        // Zaehle aufwaerts von 0 bis 14
                    {
                      port_write(P0,z);   // Zaehlvariable an Port0 ausgeben
                      delay_ms (500);     // Zeitverzoegerung 500ms
                    }
            for (z=15;z>0;z--)        // Zaehle abwaerts von 15 bis 1
                    {
                      port_write(P0,z);   // Muster an Port0 ausgeben
                      delay_ms (500);     // Zeitverzoegerung 500ms
                    }
            }//while
}//main
```

Erklären Sie, welche Wirkung erzeugt wird, wenn aufwärts von 0 bis 14 und abwärts von 15 bis 1 gezählt wird.

1.6 Tasterabfrage

Möchte man einen Taster oder Sensor abfragen, so muss der entsprechende **Pin als Eingang** deklariert werden mit

void bit_init(uint8_t port, uint8_t bitnr, uint8_t direction) Beispiel: bit_init(P2, 2, INP);

Die Abfrage (Einlesen) eines einzelnen Pins erfolgt mit

uint8_t **bit_read**(uint8_t port, uint8_t bitnr) Beispiel: wert = bit_read (P2, 2);

Die Deklaration eines Pins als Ausgang erfolgt wieder mit

void bit_init(uint8_t port, uint8_t bitnr, uint8_t direction) Beispiel: bit_init(P0, 2, OUTP);

Die Ausgabe an ein einzelnes Pin ermöglicht

void bit_write(uint8_t port, uint8_t bitnr, uint8_t value) Beispiel: bit_write(P0, 2, 1);

1.6.1 Programmbeispiel Tasterzustand kopieren und anzeigen

```
/* Tasterabfrage, Bitein- und Ausgabe kennen lernen
 * P2.9 (Taster) wird nach P0.1 kopiert
 */
#include <XMC1100-Lib.h>              // Hilfsfunktionen fuer XMC1100

uint8_t taster;                       // Zustand Taster merken
int main(void)                        // Hauptprogramm
{
  bit_init(P2, 9, INP);               // P2.9 (Taster) auf Eingabe
  bit_init(P0, 1, OUTP);              // P0.1 (Anzeige) auf Ausgabe
  while(1U)                           // Endlosschleife, immer bei Controllern
  {
       taster = bit_read(P2,9);       // (lowaktiver) Taster P2.9
       bit_write (P0,1,taster);       // nach P0.1 kopieren
  }//while
}//main
```

1.6.2 Vereinfachung

Wenn der Zustand des Tasters nicht in der Variablen taster gespeichert werden braucht, kann man den mit der Funktion bit_read gelesenen Wert direkt mit bit_write ausgeben:

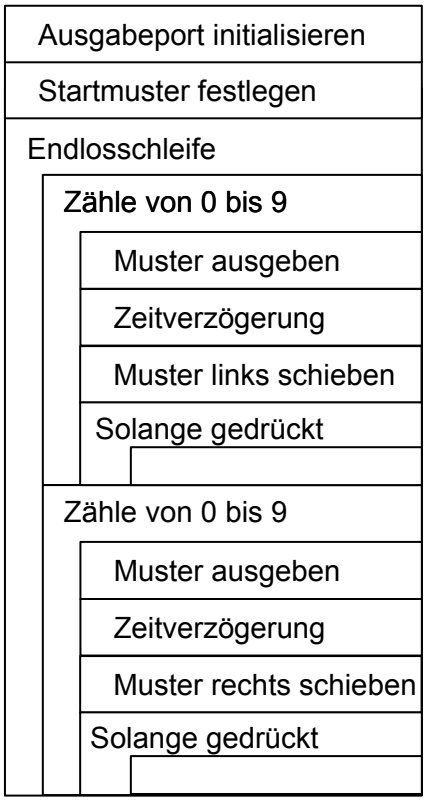
```
bit_write (P0,1,bit_read(P2,9));      // Taster P2.9 nach P0.1 kopieren
```

Der Taster ist lowaktiv angeschlossen, was bedeutet, dass bei Tastendruck ein low (0) erzeugt wird und der ungedrückte Taster ein high (1) liefert. Soll bei Tastendruck die LED an P0.1 leuchten, so muss der Zustand des Tasters invertiert werden:

```
bit_write (P0,1,~bit_read(P2,9));     // Taster P2.9 invertiert nach P0.1 kopieren
```

1.6.3 Lauflicht unterbrechen wenn Taster gedrückt

Das Lauflicht 1.4.2 soll abgeändert werden.
Solange der Taster P2.9 gedrückt ist, soll das Lauflicht anhalten.

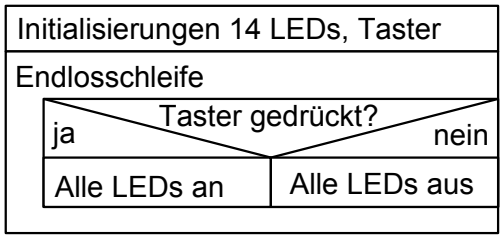



```
/* Tasterabfrage, Eingabe-Befehl kennen lernen
 */
#define gedrueckt 0
#include <XMC1100-Lib.h>       // Hilfsfunktionen fuer XMC1100

uint16_t z,muster;             // Laufvariable, Speichervariable für Lauflichtmuster
uint8_t taster;
int main(void)                 // Hauptprogramm
{
  port_init(P0,OUTP);          // Port0 auf Ausgabe programmieren
  bit_init(P2, 9, INP);        // P2.2 auf Eingabe
  bit_init(P1, 0, OUTP);       // P1.0 auf Ausgabe
  muster = 1;                  // als Dualzahl 0000 0000 0000 0001(an LEDs sichtbar)
  while(1U)                    // Endlosschleife, immer bei Controllern
  {
  for (z=0;z<10;z++)                    // 10 mal schieben
          {
            port_write(P0,muster);      // Muster an Port0 ausgeben
            delay_ms (100);             // Zeitverzoegerung 100ms
            muster=muster<<1;           // Muster um 1 Stelle nach links
            do
                    taster = bit_read (P2,9);// Taster links
            while (taster == gedrueckt);    // nichts tun solange Taster gedrückt
          }
  for (z=0;z<10;z++)                    // 10 mal schieben
          {
            port_write(P0,muster);      // Muster an Port0 ausgeben
            delay_ms (100);             // Zeitverzoegerung 100ms
            muster=muster>>1;           // Muster um 1 Stelle nach rechts
            do
                    taster = bit_read (P2,9); // Taster links
            while (taster == gedrueckt);    // nichts tun solange Taster gedrückt
          }
  }//while
```

1.7 Verzweigung mit if und else

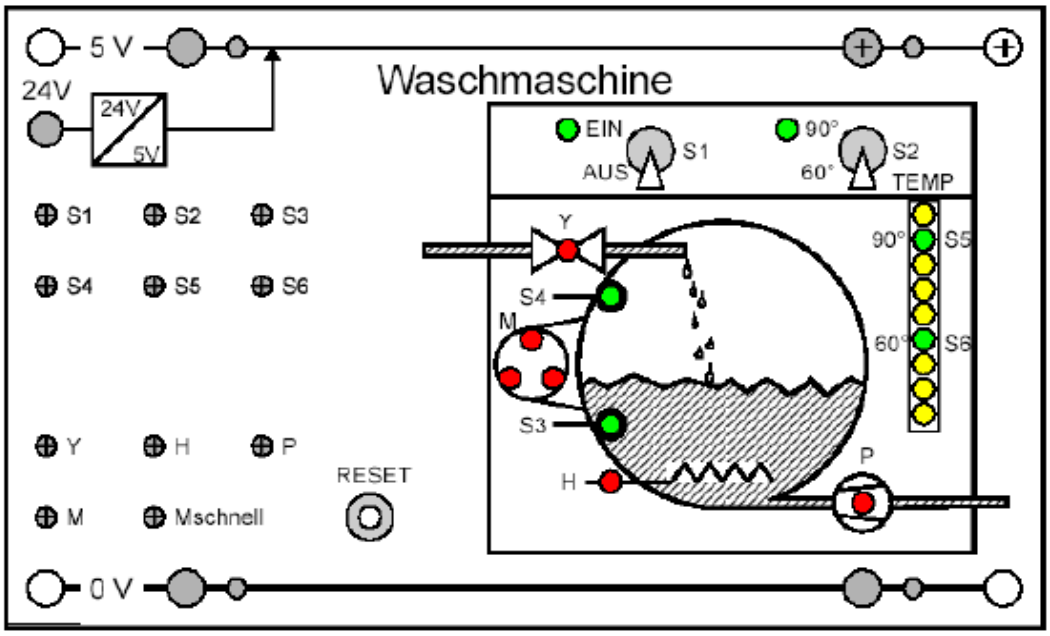
Wenn der Taster P2.2 gedrückt ist, sollen alle 14 LEDs an P0 an gehen. Wenn man den Taster wieder loslässt, gehen die LEDs wieder aus.

Das Programm muss also je nach Zustand des Tasters zwei verschiedene Dinge tun, dies nennt man eine Verzweigung, die man mit if und else programmiert.




```
/* Verzweigung mit if, alle LEDs P0 mit Taster P2.2 einschalten */
#include <XMC1100-Lib.h>              // Hilfsfunktionen fuer XMC1100
#define gedrueckt 0
int main(void)                        // Hauptprogramm
{
  port_init(P0,OUTP);                 // Port0 auf Ausgabe programmieren
  bit_init(P2,2,INP);
  while(1U)                           // Endlosschleife, immer bei Controllern
  {
    if (bit_read(P2,2) == gedrueckt) port_write(P0,0x3FFF);
                                      // Taster gedrückt -> 14 LEDs an
    else port_write(P0,0);            // sonst LEDs aus
  }//while
}//main
```

1.8 Bitabfragen und Verzweigungen bei der Waschmaschinensteuerung



Die folgenden Aufgaben simulieren Teile des Ablaufs eines Waschmaschinenprogramms. Skizzieren Sie zuerst immer den Programmablauf mit Hilfe eines Struktogramms. Zur sensorabhängigen Steuerung werden die Melder des Modells wie Schalter, Temperaturmelder .... eingelesen mit bit_read() und je nach Zustand eine Aktion mit einem bit_write ()-Befehl ausgelöst. Verwenden Sie Abkürzungen, z.B. **#define** Grad60 5, damit das Programm besser lesbar wird, z.B if (bit_read (P0,Grad60) == an) Heizung = aus;

P0.11	P0.10	P0.9	P0.8	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
P	H	Mschnell	M	Y	S6	S5	S4	S3	S2	S1	
Pumpe	Heizung	Mschnell	Motor	Zulauf	Grad60	Grad90	Fuelloben	Fuellunten	Tempwahl	Schaltein	

1.8.1 Temperaturregelung

Schreiben Sie ein Programm, das die Regelung der Temperatur nachbildet. Die Heizung wird solange mit log. 1 angesteuert, bis am Modell der Temperatursensor 60° meldet. Ist die Temperatur erreicht, soll die Heizung wieder abgeschaltet werden. Die Heizung bleibt solange abgeschaltet, bis die Temperatur unter 60° fällt (S6).

1.8.2 Temperaturregelung für 2 Waschprogramme

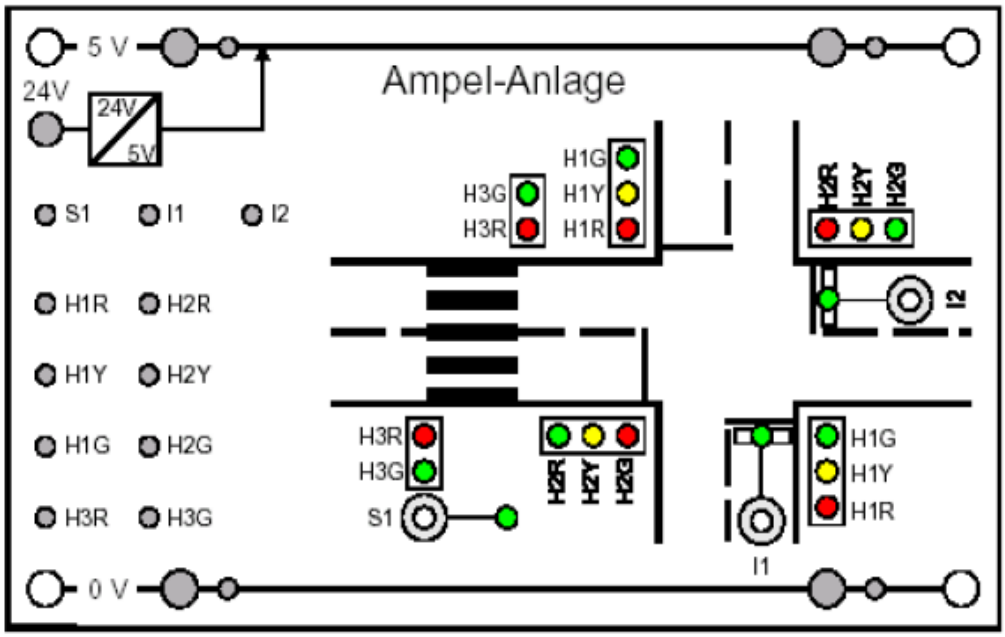
Der Auswahlschalter 60° / 90° (S2) soll zusätzlich abgefragt werden (log. 1 bei 90°C). Je nach Einstellung soll die Temperatur auf 60° oder auf 90° geregelt werden.

1.8.3 Ein-Aus-Funktion

3. Bauen Sie die Funktion EIN/AUS (S1) ein. Schalten Sie bei “AUS“ alle Ausgänge auf 0.
4. Vor Beginn der Heizungsregelung soll der Wasserzulauf bis zu dem Pegel (S3) gesteuert werden. Mit der Temperaturregelung soll auch der Motor (M) aktiviert werden.
Dieser darf aber mit der Temperaturregelung nicht ständig aus- und eingeschaltet werden.

1.9 Ampelsteuerung

1.9.1 Einfaches Programm nur mit Ausgabebefehlen und Zeitverzögerungen



Die obenstehende Ampel soll so programmiert werden, dass sich ein Zyklus für Hauptstraße (H2), Nebenstraße (H1) und Fußgängerampel ergibt. Schließen Sie die Ampel wie folgt an und beginnen sie mit der Ampelphase „H2 grün“.

P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
H3R	H3G	H2R	H2Y	H2G	H1R	H1Y	H1G

Verwenden Sie zunächst Port-Ausgabe-Befehle und die Zeitverzögerung delay_ms(). Unsere Entwicklungsumgebung lässt die Angabe von Bitkombinationen als Binärzahl zu. Dadurch können Sie sehr leicht die Ampelmuster erkennen: `port_write(P0,0b10001100);`

1.9.2 Ampelphasen abgelegt in einer Tabelle

Ampelphasen

=VERKETTEN("0b";E4;F4;G4;H4;I4;J4;K4;L4;M4)

Nr	H3R	H3G	H2R	H2Y	H2G	H1R	H1Y	H1G		
0	1	0	0	0	1	1	0	0	,	0b10001100,
1	1	0	0	1	0	1	0	0	,	0b10010100,
2	1	0	1	0	0	1	0	0	,	0b10100100,
3	1	0	1	0	0	1	0	0	,	0b10100100,
4	0	1	1	0	0	1	0	0	,	0b01100100,
5	0	1	1	0	0	1	1	0	,	0b01100110,
6	0	1	1	0	0	0	0	1	,	0b01100001,
7	0	1	1	0	0	0	0	1	,	0b01100001,
8	0	0	0	0	0	0	0	0	,	0b00000000,
9	0	0	0	0	0	0	0	0	,	0b00000000,
10	0	0	0	0	0	0	0	0	,	0b00000000,
11	0	0	0	0	0	0	0	0	,	0b00000000,
12	0	0	0	0	0	0	0	0	,	0b00000000,
13	0	0	0	0	0	0	0	0	,	0b00000000,
14	0	0	0	0	0	0	0	0	,	0b00000000,
15	0	0	0	0	0	0	0	0	,	0b00000000,

Nach Doppelklick auf die Tabelle, können Sie Ampelphasen eintragen. In der rechten Spalte wird automatisch die entsprechende binäre Darstellung erzeugt. Jede Zeile dauert 1s, damit muß nach 16s ein Zyklus beendet sein. Im C-Programm holen Sie dann jede Sekunde eine neue Zeile aus der Tabelle und geben diese an die Ampel aus.

Eine solche Tabelle heißt in C „Array“ und wird folgendermaßen definiert: `uint16_t ampel[16];`

Wenn Sie, wie bei der Ampel, feste Werte in das Array schreiben wollen, geben sie diese bei der Initialisierung an: uint16_t ampel [ ] = {2,9,4,6};

In unserem Fall ist es sinnvoll, die Werte untereinander zu schreiben, kopieren Sie dazu einfach die rechte Spalte der ausgefüllten Tabelle (vorhergehende Seite) in ihr Programm:

```
uint16_t ampel [ ] = {                    // Tabelle der Ampel-Anzeigen
                    0b10001100,
                    usw.
                    0b10001100};
```

Der Zugriff auf die einzelnen Werte erfolgt durch wert = ampel [0]); Die Zahl in der eckigen Klammer gibt an, auf welchen Wert (welche Zeile) der tabelle sie zugreifen möchten.

In Ihrem Ampelprogramm verwenden Sie nun eine For-Schleife, in der Sie nacheinander die Ampel-Bitmuster aus der Tabelle holen und ausgeben port_write (*P0*, ampel [i]);

Jede Ausgabe erfolgt gleich lang delay_ms (1000); **// 1 Sekunde**

1.9.3 Ampelphasen in einer Tabelle mit Angabe der Zeitdauer

Nun soll die Zeitdauer einer Ampelphase nicht durch mehrere Zeilen mit gleichem Bitmuster erfolgen, sondern jedes Bitmuster kommt nur einmal vor, gefolgt von der Zeitdauer dieser Phase.

```
uint16_t ampel [ ] = {  // Tabelle Ampel-Anzeigen mit Zeitdauer
                    0b10001100,1, // Phase 0
                    0b10010100,1, // Phase 1
```

Diese Tabelle erzeugt Ihnen wieder die kopierfähige rechte Spalte für das C-Programm.

Ampelphasen

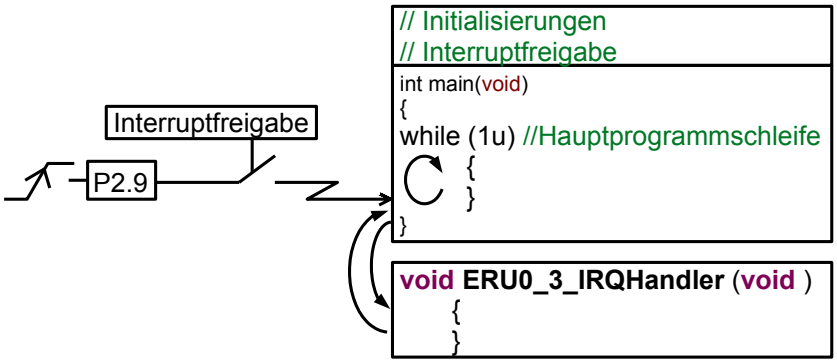
	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0							=VERKETTEN("0b";E4;F4;G4;H4;I4;J4;K4;L4;M4)
Nr	H3R	H3G	H2R	H2Y	H2G	H1R	H1Y	H1G		Zeit in s					
0	1	0	0	0	1	1	0	0	,	1	,	// Phase	0		0b10001100,1, // Phase 0
1	0	0	0	0	0	0	0	0	,	1	,	// Phase	1		0b00000000,1, // Phase 1
2	0	0	0	0	0	0	0	0	,	1	,	// Phase	2		0b00000000,1, // Phase 2
3	0	0	0	0	0	0	0	0	,	1	,	// Phase	3		0b00000000,1, // Phase 3
4	0	0	0	0	0	0	0	0	,	1	,	// Phase	4		0b00000000,1, // Phase 4
5	0	0	0	0	0	0	0	0	,	1	,	// Phase	5		0b00000000,1, // Phase 5
6	0	0	0	0	0	0	0	0	,	1	,	// Phase	6		0b00000000,1, // Phase 6
7	0	0	0	0	0	0	0	0	,	1	,	// Phase	7		0b00000000,1, // Phase 7
8	0	0	0	0	0	0	0	0	,	1	,	// Phase	8		0b00000000,1, // Phase 8
9	0	0	0	0	0	0	0	0	,	1	,	// Phase	9		0b00000000,1, // Phase 9
10	0	0	0	0	0	0	0	0	,	1	,	// Phase	10		0b00000000,1, // Phase 10

1.9.4 Grünanforderung

Die Integration einer Grünanforderung durch Tastendruck eines Fußgängers oder die Induktionsschleife in der Straße unter einem Auto gestaltet sich schwierig aus folgenden Gründen. Die Abarbeitung der bisher erstellten Programme befindet sich aus Controllersicht zu 99% innerhalb einer Zeitschleife. Wenn diese abgearbeitet wird, kann nicht gleichzeitig mit bit_read() abgefragt werden, ob eine Grünanforderung vorliegt. Die Grünanforderung wird nicht gespeichert und kann daher verloren gehen, z.B. wenn der Fußgänger die Taste wieder loslässt, bevor das Programm die Zeitschleife verlassen hat. Darum wählen wir ein anderes Konzept:

1.10 Externer Interrupt

1.10.1 Prinzip



Wenn der Interrupt freigegeben ist, löst ein Ereignis (hier steigende Flanke an P2.) einen Interrupt aus. Das Hauptprogramm main() wird dann unterbrochen und eine spezielle Funktion wird bearbeitet. Diese nennt man Interrupt-Service-Routine, bei unserem Controller sind dies vordefinierte Interrupt-Request-Handler (IRQHandler). Nach Beendigung dieser Funktion wird das Hauptprogramm an der unterbrochenen Stelle fortgesetzt.

Auf unserer Platine löst ein Signalwechsel von 0 nach 1 (steigende Flanke) an P2.9 einen Interrupt aus und **ERU0_3_IRQHandler** wird bearbeitet. Da die Taster lowaktiv sind, wird der Interrupt beim Loslassen der Taste ausgelöst. (→ externer Interrupt2)

Ein Signalwechsel von 1 nach 0 (abfallende Flanke) an P2.10 (Taster drücken) löst auch einen Interrupt aus und **ERU0_2_IRQHandler** wird bearbeitet.(→ externer Interrupt1)

1.10.2 Testprogramm

```
/* Test externe Interrupts
 * abfallende Flanke   Taster P2.9   (Taster drücken, weil lowaktiv) → binär aufwärts zählen
 * ansteigende flanke Taster P2.10 (Taster loslassen, weil lowaktiv) → binär abwärts zählen */
#include <XMC1100-Lib.h>              // Hilfsfunktionen für XMC1100
uint8_t zaehler;
int main(void)                        // Hauptprogramm
{
  ext_interrupt1_init(RE);            // externer Interrupt1 bei steigender Flanke
  ext_interrupt2_init(FE);            // externer Interrupt2 bei fallender Flanke
  ext_interrupt_enable1 ();           // Int. an P2.10 freigeben
  ext_interrupt_enable2 ();           // Int an P2.9 freigeben
  port_init(P0,OUTP);                 // LEDs an P0 -> Ausgabe
  while(1)                            // Endlos
  {
          port_write(P0,zaehler);     // dualer Zählerstand an LEDs
  }//while
}//main

// Interrupt-Service-Routinen  ( ISR)
void ERU0_3_IRQHandler (void )        // P2.9 steigende Flanke wg. RE (ext. Interrupt2)
{                                     // Taster loslassen weil lowaktiver Taster
   zaehler++;
}
void ERU0_2_IRQHandler(void)          // P2.10 fallende  Flanke wg. FE (ext. Interrupt1)
{                                     // Taster drücken weil lowaktiver Taster
   zaehler--;
}
```

1.10.3 Grün-Anforderung durch Fußgänger oder Induktionsschleife der Nebenstraße

Kopieren Sie das Programm 1.9.3 in ein neues Projekt und ergänzen Sie es durch eine interruptgesteuerte Grünanforderungen durch Fußgänger und Autos der Nebenstraße.

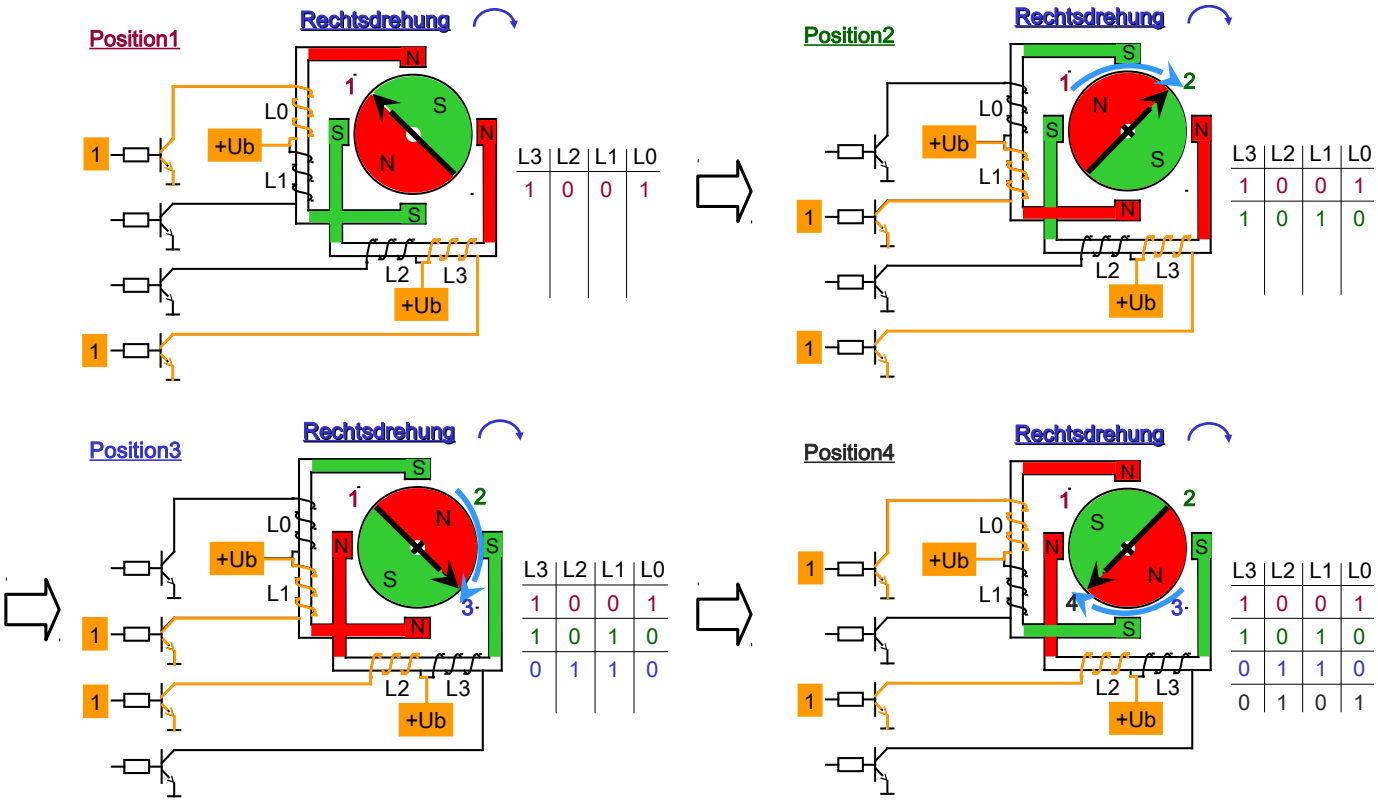
Lösungshinweis: Solange keine Anforderung vorliegt, bleibt das Programm bei Ampelphase 0 stehen. In den Interrupt-Routinen wird eine Variable uint8_t anforderung zu 1 gesetzt. Dann laufen die Ampelphasen ab. Zeigen Sie die Anforderung durch die LED P1.4 an.

1.11 Schrittmotorsteuerung

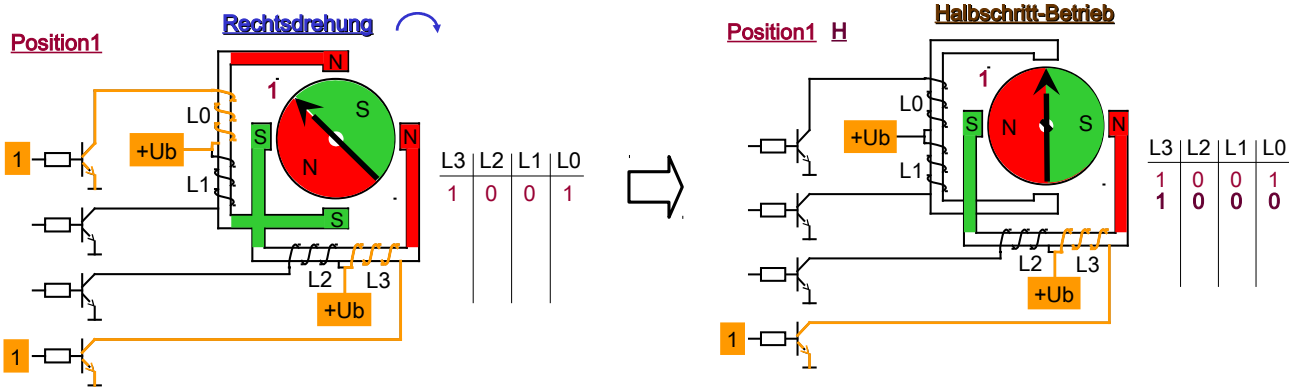
1.11.1 Funktionsweise

Aufbau: Rotor (Läufer) mit Permanent-Magneten, Spulen L0 bis L3 (4 Stränge) erzeugen Magnetfelder in den Polpaaren, externe Transistoren zur Stromverstärkung notwendig

Erzeugung der **Drehbewegung im Vollschrittbetrieb** durch 4 Bitkombinationen:



Halbschrittbetrieb: Die Zwischenschritte werden durch Abschalten einer Spule erzeugt.



Die Änderung der **Drehrichtung** erfolgt durch die Änderung der Reihenfolge der Bitkombinationen:

Vollschritt-Betrieb

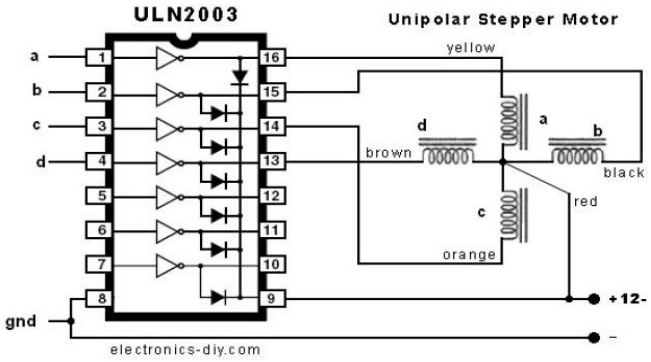
L3	L2	L1	L0
1	0	0	1
1	0	1	0
0	1	1	0
0	1	0	1

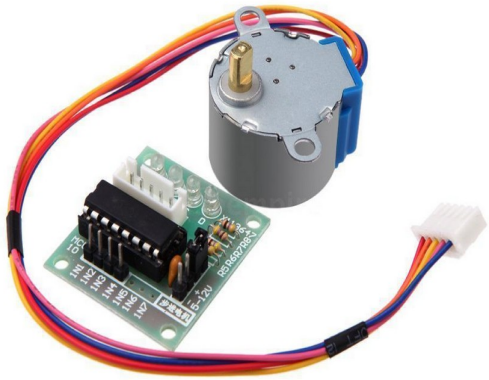
Rechtsdrehung ↓ / Linksdrehung ↑

Halbschritt-Betrieb

L3	L2	L1	L0
1	0	0	1
1	**0**	**0**	**0**
1	0	1	0
0	**0**	**1**	**0**
0	1	1	0
0	**1**	**0**	**0**
0	1	0	1
0	**1**	**0**	**1**

1.11.2 Anschluss des Schrittmotors





1.11.3 Testprogramm

```
/* Schrittmotorsteuerung Anschluss an P0.4 bis P0.7 damit P0.0 bis P0.3 (Dip-Schalter) frei bleibt*/

#include <XMC1100-Lib.h>              // Hilfsfunktionen fuer XMC1100

int main(void)                        // Hauptprogramm
{
  port_init(P0,OUTP);                 // Port0 auf Ausgabe programmieren
  while(1U)                           // Hauptprogramm-Endlosschleife
  {
          port_write (P0,0b01010000); // Rechtsdrehung
          delay_ms (50);
          port_write (P0,0b01100000); // durch 4 Bitkombinationen
          delay_ms (50);
          port_write (P0,0b10100000); //
          delay_ms (50);
          port_write (P0,0b10010000); //
          delay_ms (50);

  }//while
}//main
```

1.11.4 Schrittmotorsteuerung mit Umschaltung Rechts-/Linkslauf

```
/* Schrittmotorsteuerung mit Starttaster P2.2 und Linkslauf P2.9
 Anschluss an P0.4 bis P0.7 damit P0.0 bis P0.3 (Dip-Schalter) frei bleibt */

#include <XMC1100-Lib.h>          // Hilfsfunktionen fuer XMC1100
#define motor P0                  // Motorport
#define schritt1 0b01010000       // Bitkombinationen
#define schritt2 0b01100000
#define schritt3 0b10100000
#define schritt4 0b10010000
#define zeitkonst 50              // Zeitverzögerung für Drehgeschwindigkeit
#define taster P2                 // Tasterport
#define start 2                   // P2.2 ganz rechts für Start
#define linkslauf 9               // P2.9 ganz links für Linkslauf
#define gedrueckt 0               // lowaktive Taster

int main(void)                    // Hauptprogramm
{
  port_init(motor,OUTP);          // Motorport auf Ausgabe programmieren
  bit_init(taster,start,INP);     // Tasterpin auf Eingabe
  bit_init(taster,linkslauf,INP); // Tasterpin auf Eingabe
  while(1U)                       // Hauptprogramm-Endlosschleife
  {
        while (bit_read (taster,start) != gedrueckt);  // warten auf Start
        if (bit_read (taster,linkslauf) == gedrueckt)  //
        {
          port_write (motor,schritt4);                 // Linkssdrehung
          delay_ms (zeitkonst);
          port_write (motor,schritt3);                 // durch 4 Bitkombinationen
          delay_ms (zeitkonst);
          port_write (motor,schritt2);                 //
          delay_ms (zeitkonst);
          port_write (motor,schritt1);                 //
          delay_ms (zeitkonst);
        }//if
        else                                           // Taster Linkslauf nicht gedrückt
        {
          port_write (motor,schritt1);                 // Rechtsdrehung
          delay_ms (zeitkonst);
          port_write (motor,schritt2);                 // durch 4 Bitkombinationen
          delay_ms (zeitkonst);
          port_write (motor,schritt3);                 //
          delay_ms (zeitkonst);
          port_write (motor,schritt4);                 //
          delay_ms (zeitkonst);
        }//else
  }//while
}//main
```

1.11.5 Schrittmotorsteuerung mit Tabelle

```
//Deklaration der Variablen
uint8_t schritt [ ] = {schritt1,schritt2,schritt3,schritt4};
                      // Array, globale Variable für die 4 Schritte
uint8_t z=0;          // globale Zählvariable für die Schritte

//Schleife im Hauptprogramm für Rechtsdrehung, 4 Schritte ausgeben

for (z=0;z<4;z++)
     {
      port_write (motor,schritt [z]);                  // Rechtsdrehung
     delay_ms (50);
      } // for
```

Die 8-Bit-Variable schritt [ ] ist ein Array. Man kann sich dies vorstellen als „einen Schrank mit mehreren Schubladen“, die durchnummeriert sind. Der Zugriff auf die einzelnen Schubladen geschieht durch den Variablennamen, gefolgt von der „Schubladennummer“ in eckigen Klammern, z.B. schritt [2].

[0] →

Nr 0	schritt 1
Nr 1	schritt 2
Nr 2	schritt 3
Nr 3	schritt 4

Mit schritt [ ] = {schritt1,schritt2,schritt3,schritt4} werden die unter **#define** schritt1 0b01010000 usw. definierten Zahlen in die „Schubladen“ geschrieben. In ein Array kann man mehrere Werte in einer Schleife abspeichern oder zurückholen.

Mit dem %-Operator ist es einfach möglich, eine Art Zähler von 0 bis 4 zu erstellen: während n durch den ++ Operator von 0 bis 255 zählt, kann das Ergebnis n % 4 nacheinander nur die 4 Werte 0, 1, 2, 3 annehmen.

n	Rechnung n / 4	Ergebnis mit Ganzzahl und Rest	C-Berechnung n / 4 → Ganzahliges Ergebnis	C-Berechnung n % 4 → Rest
0	0 / 4 =	0 Rest 0	0 / 4 → 0	0 % 4 → 0
1	1 / 4 =	0 Rest 1	1 / 4 → 0	1 % 4 → 1
2	2 / 4 =	0 Rest 2	2 / 4 → 0	2 % 4 → 2
3	3 / 4 =	0 Rest 3	3 / 4 → 0	3 % 4 → 3
4	4 / 4 =	1 Rest 0	4 / 4 → 1	4 % 4 → 0
5	5 / 4 =	1 Rest 1	5 / 4 → 1	5 % 4 → 1
6	6 / 4 =	1 Rest 2	6 / 4 → 1	6 % 4 → 2
7	7 / 4 =	1 Rest 3	7 / 4 → 1	7 % 4 → 3
8	8 / 4 =	2 Rest 0	8 / 4 → 2	8 % 4 → 0
9	9 / 4 =	2 Rest 1	9 / 4 → 2	9 % 4 → 1
...	...	...	...	...
254	254 / 4 =	63 Rest 0	254 / 4 → 63	254 % 4 → 2
255	254 / 4 =	63 Rest 1	254 / 4 → 63	255 % 4 → 3

```
//Verbesserung: nun ist die Ausgabe von beliebig vielen Schritten möglich
while(1U)                                          // Hauptprogramm-Endlosschleife
  {
        while (bit_read (taster,start) != gedrueckt);   // warten auf Start
        if (bit_read (taster,linkslauf) == gedrueckt)  z--; else z++; //
        port_write (motor,schritt [z % 4]);             // Bitkombination für Schritt ausgeben
        // Anmerkung: z % 4 ist der Rest, der bei z/4 entsteht, also 0 oder 1 oder 2 oder 3
        delay_ms (zeitkonst);
  }//while
```

Nun soll eine eigene Funktion definiert werden, die dann im Hauptprogramm aufgerufen werden kann: motordrehung (links,20,zeitkonst); // 20 Schritte nach links

```
void motordrehung (uint8_t richtung, uint8_t schrittzahl, uint8_t geschwindigkeit)
```

1.12 Analog-Digital-Converter

1.12.1 Prinzip

Mit der Funktion `adc_init();` werden die Analog-Digital-Converter im Contoller initialisiert.

Die Funktion `adc_wert = adc_in(0)` holt einen convertierten 12-Bit-Wert vom Analog-Eingang 0.

Mit `port_write (P0,adc_wert)` wird dieser Wert an den LEDs an P0 angezeigt.

Am Eingang AN0 ist ein Poti angeschlossen, das analoge Werte zwischen 0V und 5V liefern sollte. Ist das Poti in der Stellung <ganz links>, sollte es 0V liefern und alle 12 LEDs sollten aus sein. Dreht man das Poti langsam, so steigt die Spannung und der angezeigte Zahlenwert ebenfalls. Es sieht so aus, als ob die LEDs „dual hochzählen“. Bei der Stellung <ganz rechts> sollte das Poti 5V liefern. Die 5V entstammen der USB-Schnittstelle und werden nicht ganz erreicht. Bei 5V würde man die Zahl 2^{12} -1= 4095 dez = 1111 1111 1111 dual, also 12 leuchtende LEDs sehen.

Beim Programmtest sieht man, dass weder die 0V noch die 5V-Grenze erreicht wird.

```
/* Analog-Digital-Converter Poti an A0 liefert Wert zwischen fast 0V und fast 5V
 */
#include <XMC1100-Lib.h>              // Hilfsfunktionen fuer XMC1100

uint16_t adc_wert;                    // Wert vom AD-Converter 16 Bit unsigned
int main(void)                        // Hauptprogramm
{
  port_init(P0,OUTP);                 // Port0 auf Ausgabe programmieren
  adc_init();                         // Analog-Digital-Converter initialisieren
  while(1U)                                   // Endlosschleife, immer bei Controllern
  {
          //adc_wert = adc_in(0);     // 12-Bit-Wert von linkem Poti an A0 holen
          adc_wert = adc_in(0)>>4;    // 8-Bit-Wert von linkem Poti an A0 holen
    port_write (P0,adc_wert);         // ausgeben an LEDs
  }//while
}//main
```

Oft benötigt man nur eine Auflösung des ADC von 8-Bit statt 12-Bit. Dann kann man einfach den adc-Wert um 4 Stellen nach rechts verschieben: Der 12-Bit-Wert 1111 1111 1111 um 4 nach rechts verschoben ergibt den 8-Bit-Wert 1111 1111: `adc_wert = adc_in(0)>>4;`

1.12.2 LCD-Anzeige mit Hilfsfunktionen

```
/*  ADC-Test mit LED und LCD-Ausgabe  */
#include <XMC1100-Lib.h>              // Hilfsfunktionen für XMC1100

uint16_t adc_wert;                    // Wert vom AD-Converter 16 Bit unsigned
int main(void)                        // Hauptprogramm
{
  port_init(P0,OUTP);                 // Port0 auf Ausgabe
  adc_init();                         // Analog-Digital-Converter initialisieren
  lcd_init();                         // LC-Display initialisieren
  while(1U)                           // Endlosschleife, immer bei Controllern
  {
          adc_wert = adc_in(0);       // Wert vom ADC
          port_write(P0,adc_wert);    // ADC-Wert an LEDs ausgeben
          lcd_setcursor (2,3);        // 2. Zeile, 3. Spalte
          lcd_print ("ADC:");
          lcd_int(adc_wert);          // AD-Wert anzeigen
          lcd_print ("dez");
          delay_ms (200);             // ruhigere Anzeige
  }//while
}//main
```

1.13 Spannungsmessung mit LCD-Ausgabe und sprintf-Funktion

1.13.1 12-Bit-Auflösung

Der Analog-Digital-Converter hat eine Auflösung von 12 Bit, daher liefert er Zahlen zwischen 0 und 2^{12}-1, also 2^{12} = 4096 Werte. Der größtmögliche Spannungswert 5V entspricht der Zahl 4096. Dies wird bei der Umrechnung des ADC-Werts in eine Spannung berücksichtigt: u = adc_wert * 5.0/4096;
Die Variable u muss eine Kommazahl sein, C-Typ float.

Achtung: Wenn Sie `u = adc_wert * 5/4096;` schreiben, rechnet C auf der rechten Seite mit ganzen Zahlen und weist diese u zu. Sie sehen also 0.000 oder 1.000 oder 2.000 oder…
Auf der rechten Seite muss also eine float-Zahl stehen, z.B. `u = adc_wert * 5.0/4096;` oder Sie geben explizit an, dass bitte in float gerechnet werden soll: `u = (float) adc_wert * 5/4096;`

1.13.2 Formatierte Textausgaben mit sprintf

Mit der Funktion sprintf können Sie Variablentypen formatieren und Werte in Texte einfügen um sie anschließend mit lcd_print an das LC-Dispaly auszugeben.

```
sprintf (lcdtext,"U = %4.3f V",u);      // Text erzeugen, Spannung mit 4 Stellen, 3 Nachkommast.
lcd_print (lcdtext);                    // anzeigen
```

Anzeigebeispiel: U = 4.123 V

Bedeutung: An der Stelle im Text, an der % erscheint, wird die Variable hinter dem Komma, hier u, eingefügt. Hinter % stehen Formatierungsanweisungen: 4 Stellen, 3 Nachkommastellen, float.

Weitere Formatierungen:

```
%d %i   Decimal signed integer.
%o      Octal integer.
%x %X   Hex integer.
%u      Unsigned integer.
%c      Character.
%s      String. siehe unten.
%f      double
%e %E   double.
%g %G   double.
%p      zeiger.
%n      Anzahl Zeichen
%%      %. No argument expected.
%#x %#X 0x Präfix wird bei
Werten ungleich Null eingefügt.
```

1.13.3 Programm

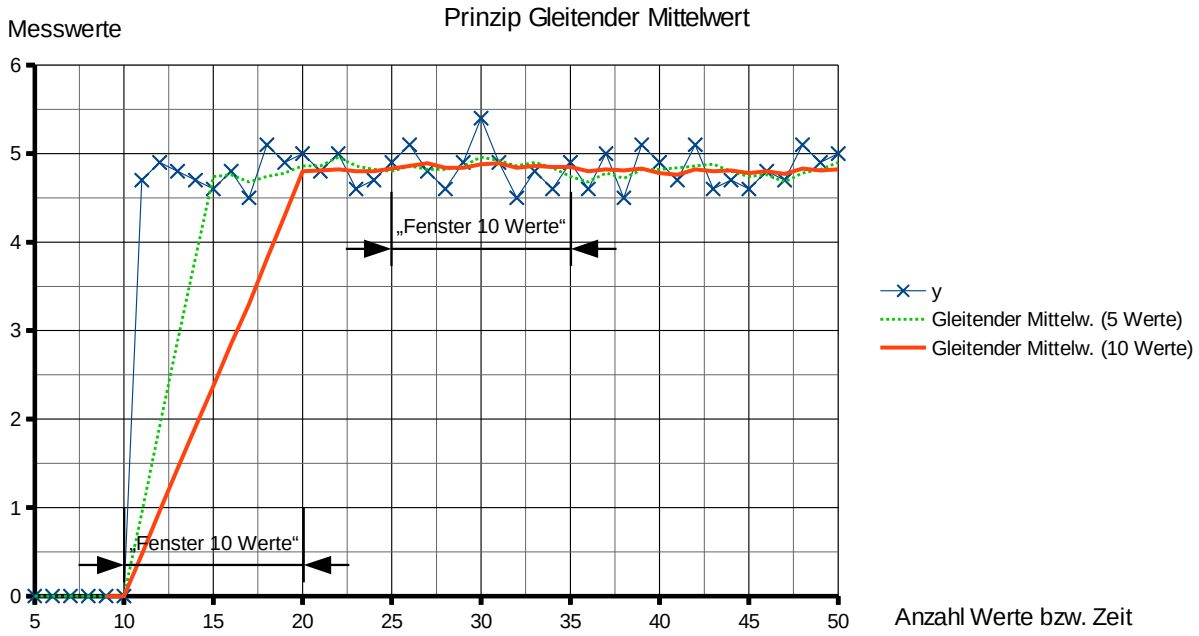
```
/* sprintf verwenden zur Anzeige von Float-Werten auf dem LC-Display
 *  Bei der Projekterstellung muss ein Haken bei sprintf-Funktionen einbinden gemacht werden!!! */
#include <XMC1100-Lib.h>                // Hilfsfunktionen für XMC1100
#include <stdio.h>                      // sprintf steht in dieser Bibliothek

uint16_t adc_wert;                      // Wert vom AD-Converter
char lcdtext [20];                      // Anzeigetext 20 Zeichen
// bei Deklaration mit uint8_t (auch int8_t ???) kommt ein Warning, da sprintf gerne in (signed) char schreiben möchte
float u;                                // Spannung in Volt
int main(void)                          // Hauptprogramm
{
  port_init(P0,OUTP);                   // Port0 auf Ausgabe
  adc_init();                           // Analog-Digital-Converter initialisieren
  delay_ms(500);                        // warten bis LC bereit
  lcd_init();                           // LC-Display initialisieren
  while(1U)                             // Endlosschleife Hauptprogramm
  {
      adc_wert = adc_in(0);             // Messwert vom ADC holen
      port_write(P0,adc_wert);          // Wert an LEDs dual anzeigen
      u = adc_wert * 5.0/4096;          // Normierung auf 5V
      lcd_setcursor (2,3);              // 2. Zeile, 3. Spalte
      sprintf (lcdtext,"U = %4.3f V",u);// Text erzeugen, Spannung mit 4 Stellen, 3 Nachkommast.
      lcd_print (lcdtext);              // anzeigen
      delay_ms (500);                   // ruhigere Anzeige
  }//while
}//main
```

1.14 Gleitende Mittelwertbildung

1.14.1 Sinn der gleitenden Mittelwertbildung

Messwerte, die man über den ADC einliest, schwanken oft. Dies liegt häufig nicht an der schwankenden Messgröße, sondern an der ungenauen Messeinrichtung, deren Spannungsversorgung z.B. schwankt. Wenn man Steuerungen in Abhängigkeit von Messwerten programmiert, stören schwankende Messwerte sehr.

Beispiel: Der Abstand eines Objekts, das 4,8cm entfernt ist, soll gemessen werden. Ein Ultraschall-Abstandsmesser liefert die im Diagramm gezeigten Werte (X). Die Anzeige würde nun stark schwanken und wäre praktisch unbrauchbar ,weil nicht ablesbar. Nun wird immer von 10 aufeinander folgenden Werten der Mittelwert gebildet. Immer wenn ein neuer Messwert



hinzukommt, wird der älteste verworfen. Also erhält man immer den Mittelwert der letzten 10 Messwerte. Dieses Verfahren nennt man gleitende Mittelwertbildung. Wie man sieht, liefert dieses Verfahren rechts stabile Werte, die man z.B. anzeigen kann.

Noch viel wichtiger sind stabile Messwerte bei einer Steuerung oder Regelung, Beispiel: „Bewege ein Fahrzeug so vor- und zurück, dass es einen Abstand von 4,8cm von einer Wand hat“.

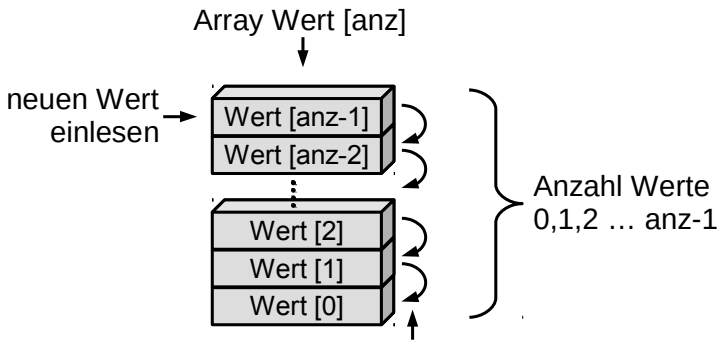
Ohne Mittelwertbildung würde das Fahrzeug dauernd vor- und zurückbewegt werden, weil die Steuerung denkt, das Fahrzeug bewegt sich, obwohl nur die Messwerte schwanken. Mit Mittelwertbildung würde die Regelung das Fahrzeug wahrscheinlich nicht oder kaum bewegen, weil der Abstand stimmt.

Einen Nachteil besitzt die Mittelwertbildung jedoch auch: Schnelle Änderungen der Messgröße (zum Beispiel am Anfang 0→ 4,8) können nicht berücksichtigt werden bzw. führen zu einer sich erst langsam dem Endwert annähernden Änderung des Mittelwerts.

1.14.2 Umsetzung in C

Zur Umsetzung der gleitenden Mittelwertbildung von „Anzahl“ Werten in ein Programm werden die Werte in einem Array gespeichert. Immer wenn ein neuer Wert vorliegt, wird dieser ins Array aufgenommen und der älteste gespeicherte Wert verworfen: Durchzuführende Schritte:

1) Array mit Anzahl Speicherstellen initialisieren.
2) Vor dem Einlesen eines neuen Werts alle Werte um 1 Stelle „nach unten“ verschieben.
3) Neuesten Wert „oben einlesen“.
4) Alle Werte addieren (Vorsicht: Summe benötigt mehr Bit als die Werte)
5) $\text{Mittelwert} = \frac{\text{Summe}}{\text{Anzahl}}$

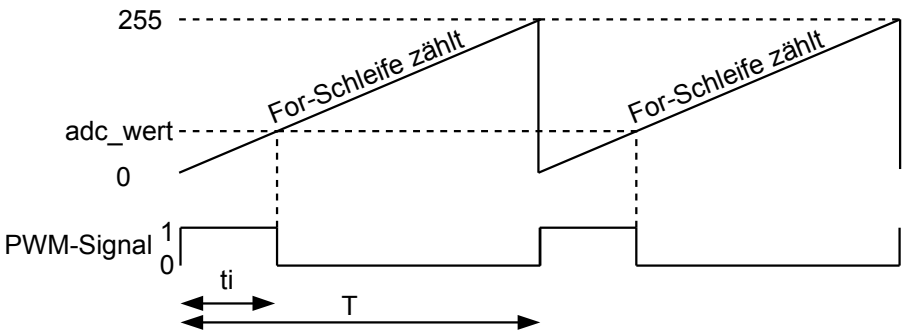



```
/* Funktion Mittelwertbildung  von anzahl ADC-Werten des Poti links */
#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100
#include <stdio.h>                          // für sprintf
#define adc_kanal 0                         // 0 für Verwendung Poti
#define anzahlwerte 20                      // Anzahl der Werte
uint16_t werte[anzahlwerte];                // Speicherung der Werte für Mittelwertbildung
uint16_t mittelw;                           // Ergebnisspeicher für Mittelwert
char lcdtext[20];                           // für sprintf und LCD-Ausgabe
uint16_t adcwert;                           // vom ADC engelesene Werte
float spannung;                             // Spannung in V als Gleitkommazahl
//-------------------- Funktion zur Mittelwertberechnung ----------------------------------------------
// Übergabe: Array der Werte, von denen der Mittelwert berechnet werden soll und Anzahl der Werte
// Achtung Ergebnis ist eine ganze Zahl!
uint16_t mittelwert (uint16_t *arr, uint8_t anzahl,uint16_t aktualwert)
{
        uint8_t i;                          // Zähler
        uint32_t sum=0;                     // Summe Achtung "größere" Variable notwendig!!!
        for (i=0;i<anzahl-1;++i) arr[i]= arr[i+1];  // alte Messwerte um 1 Stelle im Speicher verschieben
        arr[anzahl-1] = aktualwert;         // neuester Messwert in den Speicher
        for (i=0;i<anzahl;++i)     sum += arr[i];   // letzte <anzahl> Messwerte aufaddieren
        return sum / anzahl;                // geteilt durch Anzahl ist Mittelwert = Rückgabewert
//-------------------- Hauptprogramm -----------------------------------------------------------------
int main(void)
{   uint8_t i;                              // Zählvariable nur für main
  adc_init();                               // Analoge Eingänge initialisieren
  adcwert = adc_in(adc_kanal);
  for (i=0;i<anzahlwerte;i++) werte[i]=adcwert;   // Messwert-Array initialisieren,
  delay_ms(500);                            // warten bis LC bereit
  lcd_init();                               // LC-Display initialisieren
  while(1U)                                 // Endlosschleife Hauptprogramm
  {
          adcwert = adc_in(adc_kanal);
          mittelw = mittelwert (werte,anzahlwerte,adcwert);   // gleitenden Mittelwert berechnen
          spannung = 5.0/4096 * mittelw;    // Spannung berechnen
          sprintf (lcdtext,"U = %4.3f V ",spannung);   // Ausgabetext erzeugen
          lcd_setcursor (1,1);
          lcd_print (lcdtext);              // anzeigen
          delay_ms(50);
  }//while
}//main
```

1.15 PWM-Signal mit FOR-Schleife und if

Ein PWM-Signal schaltet einen angeschlossenen Verbraucher, z.B. eine LED sehr schnell ein und aus. Der Beobachter sieht das Schalten nicht, sondern nimmt eine Art Mittelwert war. Je nachdem, wie lange ein- bzw. ausgeschaltet wird, ist dieser Mittelwert und damit z.B. die Helligkeit der LED größer oder kleiner. Der Tastgrad (duty-cycle) ist gleich dem Verhältnis von Einschaltzeit ti zu Periodendauer T des Signals und kann zwischen 0% (Aus) und 100% (An) liegen.

Mit Hilfe einer einfachen FOR-Schleife soll ein PWM-Signal erzeugt werden, dessen Tastgrad mit dem Poti an A0 verändert werden kann.



Das Poti-Stellung stellt einen 8-Bit-ADC-Wert zwischen 0 bis 255 ein. Eine FOR-Schleife zählt mit i von 0 bis 255. In der Schleife wird das PWM-Signal erzeugt: Ist i kleiner als der ADC-Wert, so ist das PWM-Signal 1, anderfalls ist es 0.

```
/* PWM mit FOR-Schleife, IF... ELSE und Poti
 */
#include <XMC1100-Lib.h>                  // Hilfsfunktionen fuer XMC1100

uint16_t i;                               // Laufvariable
uint16_t adc_wert;                        // Wert vom AD-Converter 16 Bit unsigned
int main(void)                            // Hauptprogramm
{
  port_init(P0,OUTP);                     // Port0 auf Ausgabe programmieren
  adc_init();                             // Analog-Digital-Converter initialisieren
  while(1U)                               // Endlosschleife, immer bei Controllern
  {
    for (i=0;i!=255;i++)                  // Schleife für PWM-Signal
      {
      adc_wert = adc_in(0)>>4;            // 8-Bit-Wert vom ADC
      port_write (P0,adc_wert<<2);        // ausgeben um 2 Bits nach links verschoben
      if (i < adc_wert)   bit_write (P0,0,1);   // PWM-Signal mit Tastgrad ADC-Wert erzeugen
          else            bit_write (P0,0,0);
      }
  }//while
}//main
```

1.16 PWM mit internen Timern und Hilfsfunktionen

Da PWM-Signale sehr häufig verwendet werden, besitzt der Controller eine einfache Möglichkeit, drei unabhängige PWM-Signale „im Hintergrund“ zu erzeugen, ohne dass Zählschleifen in einem Programm benötigt werden. Das Zählen übernehmen Hardware-Timer, deren Periodendauer programmiert werden kann und die jederzeit einen neuen Vergleichswert zur Einstellung des Tastgrads erhalten können.

Die Ausgabe der PWM-Signale erfolgt auf festgelegte Ausgabe-Pins, diese sind:
PWM1 → P0.6, PWM2 → P0.7, PWM3 →P0.8

Sechs selbst erklärende Funktionen vereinfachen die Anwendung:

pwm1_init(); pwm2_init(); pwm3_init();

pwm1_start(); pwm2_start(); usw.

pwm1_start_interrupt(void); usw. //mit Interrupt bei jeder Periode

pwm1_duty_cycle(uint8_t compare); usw.

pwm1_duty_cycle_period (uint8_t compare, uint8_t period); usw.

pwm1_stop; usw.

```
/*
 PWM-Tastgrad mit Poti A0 einstellen
 Ausgabe PWM an P0.6
 */

#include <XMC1100-Lib.h>                // Hilfsfunktionen für XMC1100

int main(void)                          // Hauptprogramm
{
  port_init(P0,OUTP);                   // Port0 -> Ausgabe, nicht notwendig für PWM
  pwm1_init();                          // PWM1 initialisieren (an P0.6)
  pwm1_start();                         // PWM-Ausgabe starten
  adc_init();                           // ADC initialisieren (Poti links)
  while(1U)                             // Endlosschleife, immer bei Controllern
  {
        pwm1_duty_cycle(adc_in(0)>>4);  // 8 Bit des 12-Bit-ADU-Wertes für Tastgrad
  }//while
}//main
```

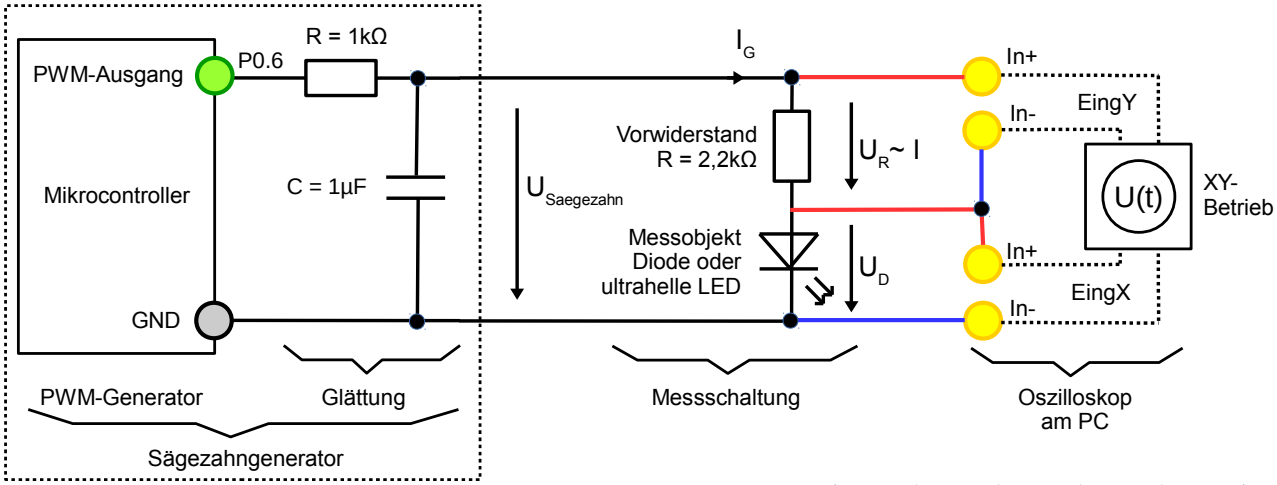
1.16.1 Aufgabe1

Schreiben Sie ein Programm, das die Helligkeit einer LED automatisch von <aus> bis <hell> und umgekehrt ändert. Verwenden Sie die PWM-Hilfsfunktionen und Zählschleifen zur automatischen Änderung des Tastgrads.

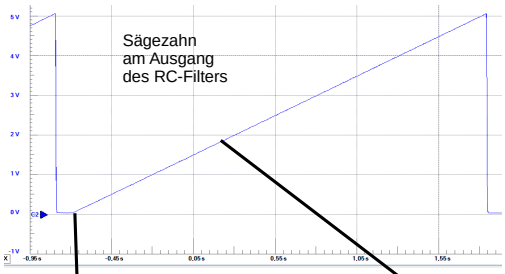
1.16.2 Aufgabe2

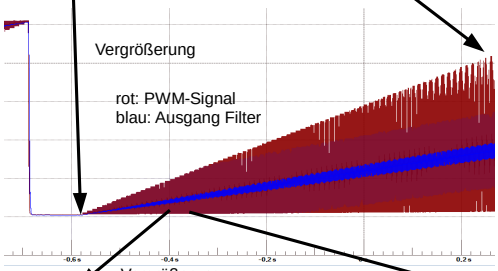
Erweitern Sie Ihr Programm zur automatischen Farbenänderung einer RGB-LED. Verwenden Sie 3 PWM-Kanäle für die 3 Farben R,G,B.

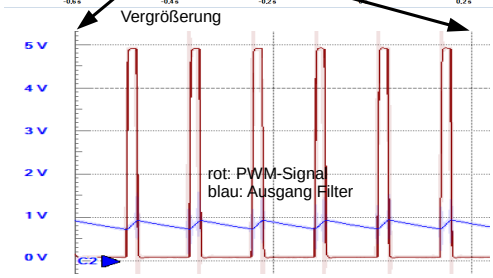
1.17 I(U)-LED-Kennlinienschreiber



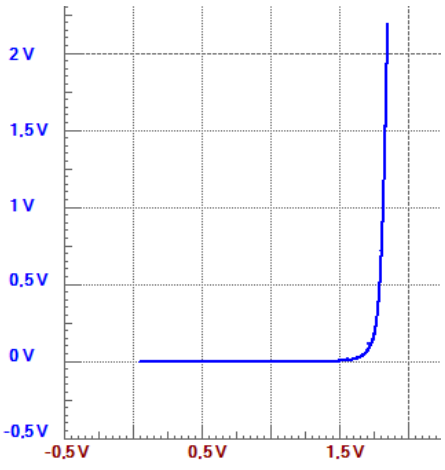
Der Mikrocontroller erzeugt ein PWM-Signal, dessen Tastgrad sich linear von 0 bis 100% ändert. Nach der Glättung mit einem Kondensator ergibt sich eine sägezahnförmige Spannung. Diese wird an eine Schaltung aus Widerstand und LED angeschlossen. Mit einem Oszilloskop im XY-Betrieb kann so die Kennlinie der Diode dargestellt werden.








```
/* Sägezahnspannung (Rampe) mit PWM erzeugen an P0.6
 * für LED-Kennlinienschreiber  R=1kOhm, C=1µF an P0.6*/

#include <XMC1100-Lib.h>      // Hilfsfunktionen für XMC1100
uint8_t z;                    // Zählvariable

int main(void)                // Hauptprogramm
{
  pwm1_init();                // PWM Kanal 0 initialisieren
  pwm1_start();               // Ausgabe starten
  while(1U)                   // Endlosschleife
  {
    pwm1_duty_cycle(0);       // 0V ausgeben
    delay_ms(100);            // warten bis Kondens. entladen
    for (z=0; z<255;z++)      // Spannung 0...5V erhöhen
          { pwm1_duty_cycle(z); // dazu PWM-Tastgrade ändern
            delay_ms(10);     // legt Steilheit Sägezahn fest
          }
  }//while
}//main
```

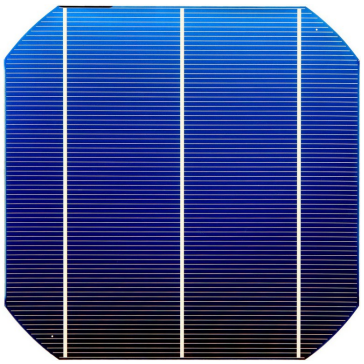


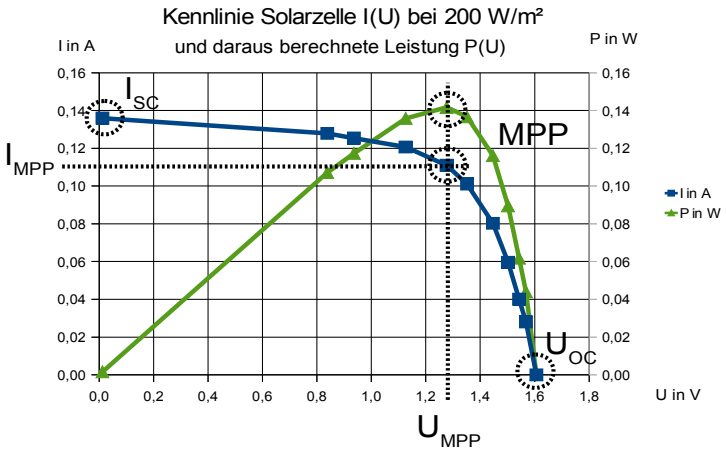
1.17.1 Oszillogramm im XY-Modus

Die Spannung am Vorwiderstand der LED ist proportional dem Strom I durch die LED und wird in Y-Richtung dargestellt. Maßstab: I = U / 2,2kΩ. Damit beträgt der maximale LED-Strom 1 mA.
Die Spannung an der LED wird x-Richtung dargestellt.
Die LED-Spannung bei I = 1 mA beträgt U = 1,8V.

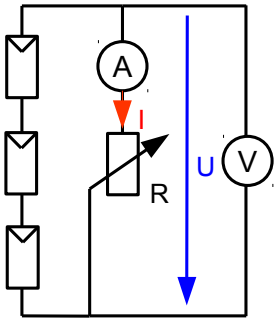
2 Projekte

2.1 Leistungsmessung und Modell eines MPP-Trackers





Der MPP (Maximum Power Point) einer Solarzelle kann nur durch Messung bestimmt werden. Die Funktionsweise des MPP-Trackers in einem Wechselrichter kann man sich so vorstellen: Der Verbraucherstrom wird solange geändert, bis das Produkt aus gemessener Spannung und gemessenem Strom maximal wird. Vereinfacht stellen wir uns vor, dass dazu der Widerstand R des Wechselrichters verändert wird.



In unserem „mechanischen“ MPP-Tracker wird ein Motorpotenziometer verwendet. Der Mikrocontroller steuert den Motor an und verändert dadurch den Widerstand $R_{Verbraucher}$ solange, bis $U_R * I$ maximal wird.
Wir verwenden einen Schrittmotor und die im Kapitel 1.11.5 Seite 17 (Schrittmotorsteuerung mit Tabelle) entwickelten C-Funktionen zur Rechts-Links-Steuerung.

Der Controller berechnet die Leistung $P_{Verbr} = U_{Verbr} \cdot I_{Verbr}$. Die analogen Eingänge haben eine Auflösung von 12 Bit und abeiten mit einem Spannungsbereich von 0 V bis 5 V.

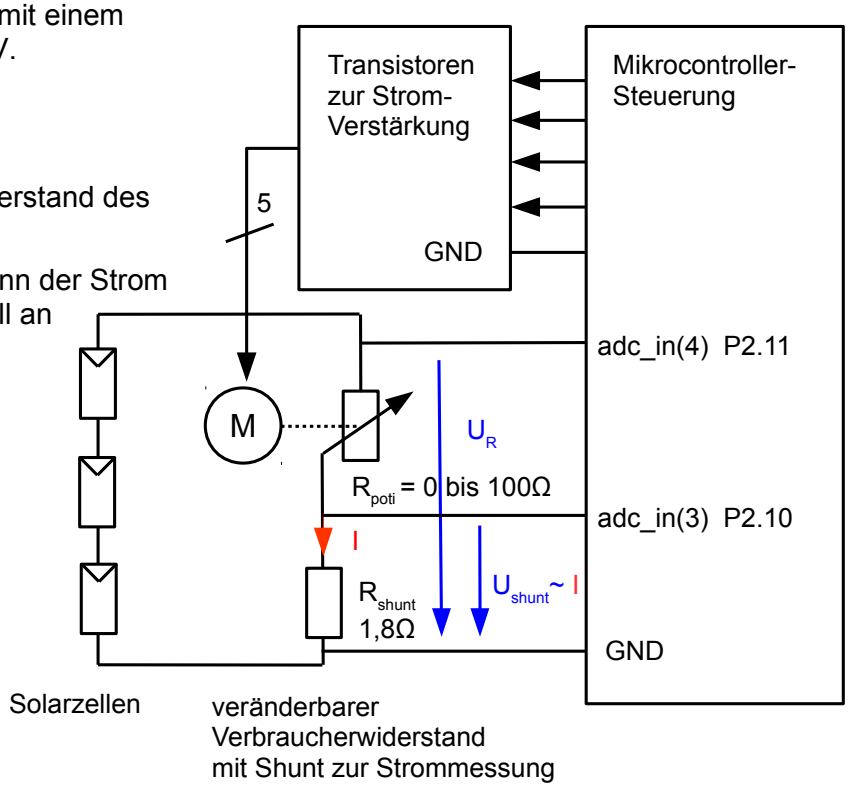
$$U_{Verbr} = \frac{5\,V}{2^{12}} \cdot (adcin\,4)$$

Als Verbraucher wird hier der Widerstand des Potis plus der Shunt bezeichnet.

Wie auch bei allen Multimetern kann der Strom nur indirekt durch Spannungsabfall an einem Messwiderstand (Shunt) ermittelt werden.

$$I_{Verbr} = \frac{U_{Shunt}}{R_{Shunt}}$$

$$U_{Shunt} = \frac{5\,V}{2^{12}} \cdot adcin\,3$$



2.1.1 Programm Leistungsmessung Solarzelle mit gleitender Mittelwertbildung

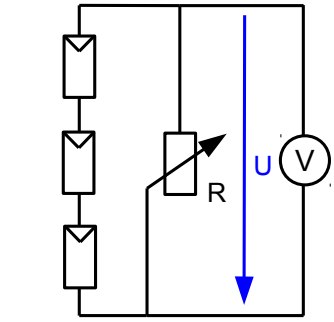
```
/* U,I,P-Messung Solarzellen
 * Rlast = 22 Ohm und 100 Ohm-Poti und 1,8 Ohm-Strommesswiderstand in Reihe
 * Solarzelle auf Holzbrett GDS2 C=1000µF parallel zur Solarzelle
 * Bei der Projekterstellung muss ein Haken bei sprintf-Funktionen einbinden gemacht werden!!!*/
#include <XMC1100-Lib.h>          // Hilfsfunktionen für XMC1100
#include <stdio.h>                // für sprintf
#define kanal_u 4                 // normal 4 -> P2.11, für Programmtest mit Poti nehme 0
#define kanal_i 3                 // normal 3 -> P2.10, für Programmtest mit Poti nehme 1
#define anzahlwerte 20            // Anzahl der Werte (max.6 wegen LC-Anzeige der einzelnen Werte)
#define r_shunt 1.8               // Strommesswiderstand
uint16_t u_werte[anzahlwerte];    // Array für Spannungswerte
uint16_t i_werte[anzahlwerte];    // Array für Stromwerte
uint16_t mw_u,mw_i;               // Ergebnisspeicher für Mittelwerte
char lcdtext[20];                 // für sprintf und LCD-Ausgabe
uint16_t adc_u,adc_i;             // von den ADC engelesene Werte
float u_float, i_float, p_float;  // Messwerte in V, A, W

//-------------------- Funktion zur Mittelwertberechnung ----------------------------------------------
// Übergabe: Array der Werte, von denen der Mittelwert berechnet werden soll und Anzahl der Werte
// Achtung Ergebnis ist eine ganze Zahl!
uint16_t mittelwert (uint16_t *arr, uint8_t anzahl,uint16_t aktualwert)
{
        uint8_t i;                                  // Zähler
        uint32_t sum=0;                             // Summe Achtung "größere" Variable notwendig!!!
        uint32_t mw;                                // Mittelwert
        for (i=0;i<anzahl-1;++i) arr[i]= arr[i+1];  // alte Messwerte um 1 Stelle im Speicher verschieben,
                                                    // ältester Messwert entfällt
        arr[anzahl-1] = aktualwert;                 // neuester Messwert in den Speicher
        for (i=0;i<anzahl;++i)  sum += arr[i];      // letzte <anzahl> Messwerte aufaddieren
        mw = sum / anzahl;                          // geteilt durch Anzahl ist Mittelwert
        return  (uint16_t) mw;                      // Rückgabewert umwandeln in 16-Bit
}
//-------------------- Hauptprogramm -----------------------------------------------------------------
int main(void)
{
  uint8_t i;
  port_init(P0,OUTP);                               // Port0 auf Ausgabe
  adc_init();                                       // Analoge Eingänge initialisieren
  adc_u = adc_in(kanal_u);
  for (i=0;i<anzahlwerte;i++) u_werte[i]=adc_u;     // Messwert-Arrays initialisieren,
  adc_i = adc_in(kanal_i);
  for (i=0;i<anzahlwerte;i++) i_werte[i]=adc_i;     //
  delay_ms(500);                                    // warten bis LC bereit
  lcd_init();                                       // LC-Display initialisieren
  while(1U)                                         // Endlosschleife Hauptprogramm
  {
        adc_u = adc_in(kanal_u);
        mw_u = mittelwert (u_werte,anzahlwerte,adc_u);  // gleitenden Mittelwert berechnen
        u_float = 5.0/4096 * mw_u;                      // Spannung berechnen
        sprintf (lcdtext,"U = %4.3f V ",u_float);       // Ausgabetext erzeugen
        lcd_setcursor (1,1);
        lcd_print (lcdtext);                            // anzeigen
        adc_i = adc_in(kanal_i);
        mw_i = mittelwert (i_werte,anzahlwerte,adc_i);  // gleitenden Mittelwert berechnen
        i_float = 5.0/4096 * mw_i/r_shunt;              // normieren auf 5V und i berechnen
        sprintf (lcdtext,"I = %4.3f A ",i_float);
        lcd_setcursor (2,1);
        lcd_print (lcdtext);                            // anzeigen
        p_float = u_float * i_float;                    // Leistung der Mittelwerte berechnen
        sprintf (lcdtext,"P = %4.3f W ",p_float);
        lcd_setcursor (3,1);
        lcd_print (lcdtext);                            // anzeigen
        delay_ms(50);                                   // Bestimmt die Geschwindigkeit der Messungen
  }//while
}//main
```

2.2 Einstrahlungsmessgerät mit Solarzelle

Das Einstrahlungsmessgerät (Solar Power Meter) zeigt die Einstrahlung E in W / m² an.

Die Funktion dieses Gerätes soll mit einer Solarzelle und dem Mikrocontroller mit LCD-Anzeige „nachgebaut“ werden.
Der Zusammenhang zwischen Spannung U der Solarzelle und Einstrahlungsstärke E wird mit durch viele Messungen bei unterschiedlichen Bestrahlungsstärken ermittelt: Messtabelle mit E und U anlegen, E(U)-Diagramm mit Excel erzeugen und Trendlinie einfügen (Typ: wahrscheinlich polynomisch). Die gefundene Funktion E(U) wird dann in den Mikrocontroller programmiert.

In Vorversuchen ist zu klären, bei welchem Widerstandswert R sich die Spannung bei unterschiedlichen Beleuchtungsstärken besonders stark ändert.



Solarzellen Verbraucher

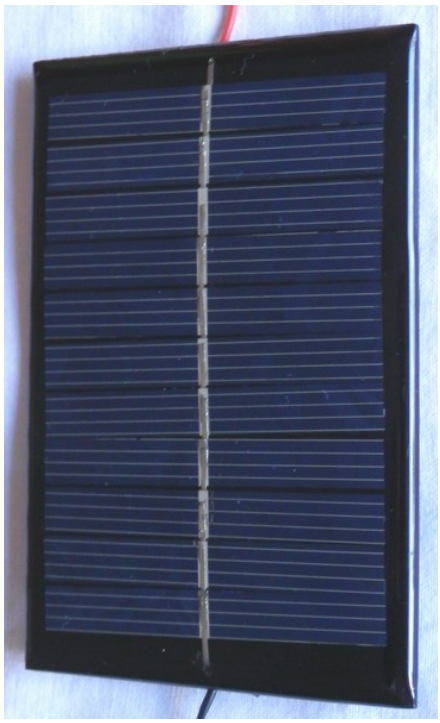


Abbildung 2.1: Solarmodul mit 11 in Reihe geschalteten Solarzellen

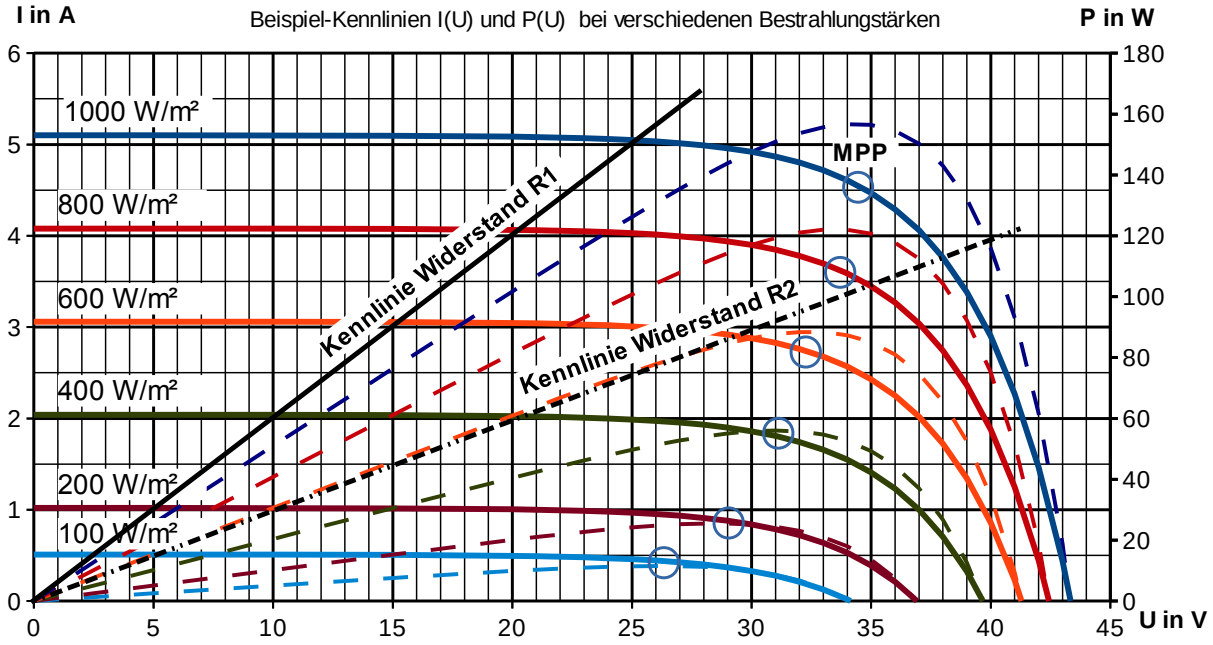
2.2.1 Theoretischer Hintergrund

An dem unten stehenden Kennlinienfeld sollen grundlegende Abhängigkeiten erklärt werden. Achtung: die abgildeten Kennlinien sind nicht identisch mit den verwendeten Modulen, sondern sollen nur beispielhaft verwendet werden.

Der Kurzschlusstrom (Schnittpunkte I(U)-Kennlinien mit der I-Achse) ist proportional zur Bestrahlungsstärke E (Angabe in W/m²).

Die Leerlaufspannung (Schnittpunkte I(U) mit der U-Achse) sind **nicht** proportional zur Bestrahlungsstärke. Daher messen wir **nicht** die Leerlaufspannung U_{OC} um daraus die Bestrahlungsstärke zu berechnen.

Wenn man die Solarzellen mit einem festen Widerstand belastet, ergibt sich als „Arbeitspunkt“ der Schnittpunkt aus I/U-Kennlinie der Solarzelle und I/U-Kennlinie des Widerstands. Die dann fließenden Ströme und anleigenden Spannungen kann man direkt dem Kennlinienfeld entnehmen.



Man sieht, dass bei Verwendung des Widerstands R2 die (gemeinsame) Spannung am Widerstand/ an der Solarzelle **nicht** proportional zur Bestrahlungsstärke ist.

Bei Verwendung des Widerstands R1, dessen Kennlinie die 1000 W/m²-Kennlinie der Solarzelle im „flachen Bereich“ weit „links“ vom MPP schneidet, **ist eine lineare Abhängigkeit zwischen U und E zu erkennen**:
E = 200 W/m² → U = 5V. 5-fache Einstrahlung (E = 1000 W/m²) → 5-fache Spannung (U = 25 V).

Der verwendete Widerstand beträgt: $R = \frac{U}{I} = \frac{25\ V}{5\ A} = 5\Omega$

Die sich ergebene Zusammenhang zwischen E und U wäre in diesem Fall:

$$E = \frac{1000\,W/m^2}{25\,V} \cdot U \quad \rightarrow \quad E = 40\,\frac{W}{m^2 \cdot V} \cdot U \quad \text{nur bei R = 5 Ω!}$$

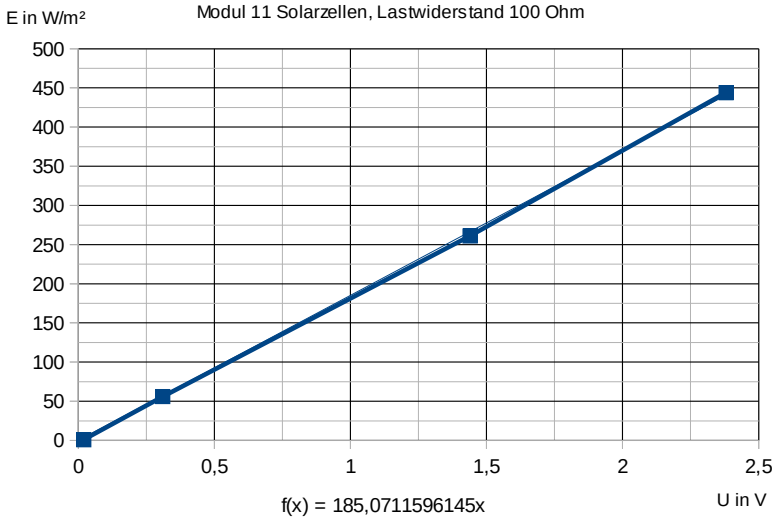
2.2.2 Messungen zur Bestimmung der Abhängigkeit U(E)

Mini-Solarmodul mit 11 Zellen
und Lastwiderstand R= 100 Ohm (gemessen)

U in V	E in W/m²		E/U
0,02	1	Lampe aus	
0,31	56	Lampe Stufe 1	180,6
1,44	261	Lampe Stufe 2	181,3
2,38	444	Lampe Stufe 3	186,6

E gemessen mit Amprobe Solar-100
Lampe ca 27cm von Solarmodul entfernt

Diagramm E(U) zur Programmierung eines Einstrahlungsmessgeräts



Wie auf der letzten Seite erklärt, entsteht eine lineare Abhängigkeit zwischen Einstrahlung E und Spannung U an dem mit einem geeigneten Widerstand belasteten Solarmodul.

Für die eingefügte Trendlinie ergibt sich die Funktion E = 185,1 * U. Diese wird im Programm zur Emittlung der Einstrahlung bei gemessener Spannung verwendet.

2.2.3 C-Programm Bestimmung der Einstrahlung aus der Spannung

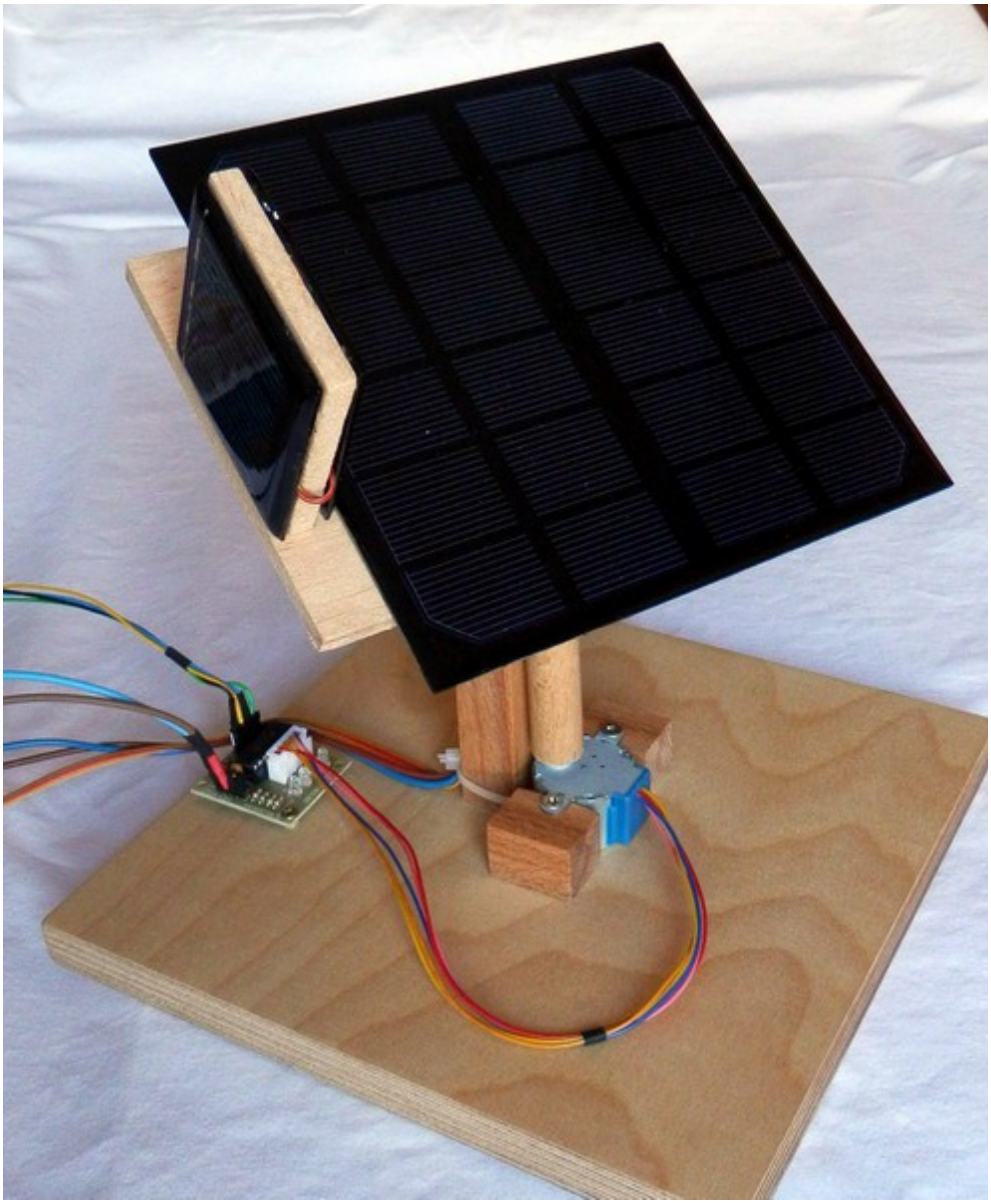
```
/* Einstrahlungsmessung über die Spannung eines mit Widerstand belastetem Solarmodul
 * Rlast = 100 Ohm an Mini-Solarmodul mit 11 Zellen
 * Bei der Projekterstellung muss ein Haken bei sprintf-Funktionen einbinden gemacht werden!!!*/

#include <XMC1100-Lib.h>          // Hilfsfunktionen für XMC1100
#include <stdio.h>

uint16_t adc_wert;                // Wert vom AD-Converter 16 Bit unsigned
float u,e;
char buf [20];
int main(void)                    // Hauptprogramm
{ delay_ms(1000);                 // wg. lcd-Display
  port_init(P0,OUTP);             // Port0 auf Ausgabe
  adc_init();                     // Analog-Digital-Converter initialisieren
  lcd_init();                     // LC-Display initialisieren
  while(1U)                       // Endlosschleife, immer bei Controllern
  {
        adc_wert = adc_in(2);     // Wert vom ADC -> Zeitverzoegerung
        port_write(P0,adc_wert);  // ADC-Wert an LEDs ausgeben
        u = adc_wert * 5.0/4096;  // Spannung in Volt
        e = 185.1 * u;                  // Einstrahlung in W/m²
        lcd_setcursor (1,3);            // 1. Zeile, 3. Spalte
        sprintf(buf,"U = %1.3fV",u);
        lcd_print (buf);
        lcd_setcursor (2,3);            // 2. Zeile, 3. Spalte
        sprintf(buf,"E = %4.1fW/m2",e);
        lcd_print (buf);
        delay_ms (200);                 // ruhigere Anzeige
  }//while
}//main
```

2.2.4 C-Programm mit gleitender Mittelwertbildung

```
/* Messung Solarzellen für Einstrahlungsmessung
 * Rlast = 100 Ohm an Mini-Solarmodul mit 11 Zellen
  * Bei der Projekterstellung muss ein Haken bei sprintf-Funktionen einbinden gemacht werden!!!*/
#include <XMC1100-Lib.h>      // Hilfsfunktionen für XMC1100
#include <stdio.h>            // für sprintf

#define kanal_u 2             // normal 2 -> P2.9, für Programmtest mit Poti nehme 0
#define anzahlwerte 20        // Anzahl der Werte (max.6 wegen LC-Anzeige der einzelnen Werte)
uint16_t u_werte[anzahlwerte]; // Array für Spannungswerte
uint16_t mw_u;                // Ergebnisspeicher für Mittelwert
char lcdtext[20];             // für sprintf und LCD-Ausgabe
uint16_t adc_u;               // von den ADC engelesene Werte
float u_float,e_float;

//-------------------- Funktion _floatzur Mittelwertberechnung ----------------------------------------------
// Übergabe: Array der Werte, von denen der Mittelwert berechnet werden soll und Anzahl der Werte
// Achtung Ergebnis ist eine ganze Zahl!
uint16_t mittelwert (uint16_t *arr, uint8_t anzahl,uint16_t aktualwert)
{
        uint8_t i;                                  // Zähler
        uint32_t sum=0;                             // Summe Achtung "größere" Variable notwendig!!!
        uint32_t mw;                                // Mittelwert
        for (i=0;i<anzahl-1;++i) arr[i]= arr[i+1];  // alte Messwerte um 1 Stelle im Speicher verschieben,
                                                    // ältester Messwert entfällt
        arr[anzahl-1] = aktualwert;                 // neuester Messwert in den Speicher
        for (i=0;i<anzahl;++i)    sum += arr[i];    // letzte <anzahl> Messwerte aufaddieren
        mw = sum / anzahl;                          // geteilt durch Anzahl ist Mittelwert
        return  (uint16_t) mw;                      // Rückgabewert umwandeln in 16-Bit
}
//-------------------- Hauptprogramm -----------------------------------------------------------------------
int main(void)
{
  uint8_t i;
  port_init(P0,OUTP);                               // Port0 auf Ausgabe
  adc_init();                                       // Analoge Eingänge initialisieren
  adc_u = adc_in(kanal_u);
  for (i=0;i<anzahlwerte;i++) u_werte[i]=adc_u;     // Messwert-Arrays initialisieren,
  delay_ms(500);                                    // warten bis LC bereit
  lcd_init();                                       // LC-Display initialisieren
  while(1U)                                         // Endlosschleife Hauptprogramm
  {
          adc_u = adc_in(kanal_u);
          mw_u = mittelwert (u_werte,anzahlwerte,adc_u);      // gleitenden Mittelwert berechnen
          u_float = 5.0/4096 * mw_u;                          // Spannung berechnen
          sprintf (lcdtext,"U = %4.3f V ",u_float);           // Ausgabetext erzeugen
          lcd_setcursor (1,1);
          lcd_print (lcdtext);                                // anzeigen
          e_float = u_float * 185.1;                          // Einstrahlung berechnen
          sprintf (lcdtext,"E = %4.0f W/m2 ",e_float);
          lcd_setcursor (2,1);
          lcd_print (lcdtext);                                // anzeigen
          delay_ms(100);                                // Bestimmt die Geschwindigkeit der Messungen
  }//while
}//main
```

2.3 1-Achs-Nachführung eines Solarpanels

2.3.1 Aufbau



2.3.2 Beschreibung

Auf ein Solarmodul sollen die Sonnenstrahlen möglichst senkrecht einfallen. Dazu wird das Modul mit einem Schrittmotor nach dem Sonnenstand ausgerichtet.

2 kleine, senkrecht zum großen Solarmodul und senkrecht zur Sonne ausgerichtete Solarzellen dienen der Richtungserkennung. Wenn beide Module die gleiche Spannung liefern, ist das große Modul optimal zur Sonne ausgerichtet.

Der Mikrocontroller muss daher nur die Spannungen der kleinen Module messen, vergleichen und folgende Steuerung ausführen: z.B. U1 > U2 → Rechtsdrehung, U2 > U1 → Linksdrehung. Beide Spannungen ungefähr gleich → keine Drehung.

2.3.3 C-Programm

```
/* Nachführung Solarmodul durch Auswertung zweier Solarzellen, die senkrecht Richtung
 * Sonne und sekrecht zum Solarmodul stehen
 * Steuerung mit Schrittmotor, ohne Endschalter
 * Mit gleitender Mittelwertbildung der ADC-Werte von den Richtungssolarzellen */
#include <XMC1100-Lib.h>                // Hilfsfunktionen für XMC1100
#include <stdio.h>                      // für sprintf
#define motor P0                        // Motorport
#define schritt1 0b01010000             // Bitkombinationen
#define schritt2 0b01100000
#define schritt3 0b10100000
#define schritt4 0b10010000
#define aus 0                           // Motor aus
#define rechts 1                        // Richtung rechts
#define links 2                         // Richtung links
uint8_t schritt [] = {schritt1,schritt2,schritt3,schritt4};
                                        // Array, globale Variable für die 4 Schritte
uint8_t z=0;                            // globale Zählvariable für die Schritte
#define zeitkonstante 50                // für Messgeschwindigkeit und Motordrehzahl verantwortlich
#define kanal_re 4                      // P2.11 orange Leitg Richtungssolarzelle
#define kanal_li 3                      // P2.10 rote Leitg Richtungssolarzelle
#define anzahlwerte 20                  // Anzahl der Werte für glieitende Mittelwertbildung
uint16_t re_werte[anzahlwerte];         // die ersten Werte werden später vom Poti links geliefert
uint16_t li_werte[anzahlwerte];         // die zweiten Werte werden später vom Poti rechts geliefert
uint16_t mw_re,mw_li;                   // Ergebnisspeicher für Mittelwerte
char lcdtext[20];                       // für sprintf und LCD-Ausgabe
uint16_t adc_re,adc_li;                 // von den ADC engelesene Werte
#define delta_u 5                       // Hysterese der ADC-Messwerte rechts/links

//-------------------- Funktion zur Mittelwertberechnung ------------------------------------------------
// Übergabe: Array der Werte, von denen der Mittelwert berechnet werden soll und Anzahl der Werte
// Achtung Ergebnis ist eine ganze Zahl!
uint16_t mittelwert (uint16_t *arr, uint8_t anzahl,uint16_t aktualwert)
{
        uint8_t i;                              // Zähler
        uint32_t sum=0;                         // Summe Achtung "größere" Variable notwendig!!!
        uint32_t mw;                            // Mittelwert
        for (i=0;i<anzahl-1;++i) arr[i]= arr[i+1];  // alte Messwerte um 1 Stelle im Speicher verschieben,
                                                // ältester Messwert entfällt
        arr[anzahl-1] = aktualwert;             // neuester Messwert in den Speicher
        for (i=0;i<anzahl;++i)     sum += arr[i];   // letzte <anzahl> Messwerte aufaddieren
        mw = sum / anzahl;                      // geteilt durch Anzahl ist Mittelwert
        return (uint16_t) mw;                   // Rückgabewert umwandeln in 8-Bit
}

// ---------- Schrittmotordrehung -------------------------------------
void motordrehung_1schritt (uint8_t richtung)
{
         if (richtung == aus)  port_write (motor,0);        // Motor aus, Energie sparen
         else {
                 if (richtung == rechts) z++; else z--;     // nächste Bitkombination
                 port_write (motor,schritt [z % 4]);        // für Schritt ausgeben
                 }
    // Anmerkung: z % 4 ist der Rest, der bei z/4 entsteht, also 0 oder 1 oder 2 oder 3
}// motordrehung Ende
```

```
//-------------------- Hauptprogramm ---------------------------------------------------------------------
int main(void)
{
  uint8_t i;
  port_init(motor,OUTP);                                  // Motorport auf Ausgabe programmieren
  adc_init();                                             // Analoge Eingänge initialisieren
  adc_re = adc_in(kanal_re);
  for (i=0;i<anzahlwerte;i++) re_werte[i]=adc_re;         // Messwert-Arrays initialisieren,
  adc_li = adc_in(kanal_li);
  for (i=0;i<anzahlwerte;i++) li_werte[i]=adc_li;         //
  delay_ms(500);                                          // warten bis LC bereit
  lcd_init();                                             // LC-Display initialisieren
  lcd_setcursor (1,1);
  lcd_print ("Solarz-rechts:");                           // Texte anzeigen
  lcd_setcursor (2,1);
  lcd_print ("Solarz-links:");
  while(1U)                                               // Endlosschleife Hauptprogramm
  {
          adc_re = adc_in(kanal_re);                              // Spannung Richtungssolarzelle rechts
          mw_re = mittelwert (re_werte,anzahlwerte,adc_re);       // gleitenden Mittelwert berechnen
          adc_li = adc_in(kanal_li);                              // Spannung Richtungssolarzelle links
          mw_li = mittelwert (li_werte,anzahlwerte,adc_li);       // gleitenden Mittelwert berechnen
          if (mw_re > (mw_li+delta_u)) motordrehung_1schritt (links);          // Entscheidung
          else if (mw_li > (mw_re+delta_u)) motordrehung_1schritt (rechts);    // Links- Rechtsdrehung
                  else motordrehung_1schritt(aus);                // oder Motor aus
          lcd_setcursor (1,15);
          lcd_int (mw_re);                                        // Wert Richtungssolarzelle anzeigen
          lcd_setcursor (2,15);
          lcd_int (mw_li);
          delay_ms(zeitkonstante);                                // bestimmt Schrittmotordrehzahl
  }//while
}//main
```

2.4 RGB-LEDs mit PWM steuern

Das nachfolgende Programm erzeugt automatische Farbübergänge bei einer RGB-LED mittels dreier PWM-Signale.

```
/*RGB-LED mit PWM steuern
 R an P0.0 (Tastgrad mit Poti0)
 G an P0.1 (Tastgrad mit Poti1)
 B an P0.2 (automatische Erhöhung und Erniedrigung des Tastgrads)*/

#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100
uint8_t z;                                  // Zählvariable

void RGBout (uint8_t tgr,uint8_t tgg,uint8_t tgb)
{
        pwm1_duty_cycle(~tgr);              // Tastgrade
        pwm2_duty_cycle(~tgg);
        pwm3_duty_cycle(~tgb);
}

int main(void)                              // Hauptprogramm
{
  port_init(P0,OUTP);                       // Port0 auf Ausgabe, nicht notwendig für PWM
  pwm1_init();                              // PWM Kanal 0 initialisieren
  pwm2_init();                              // PWM Kanal 1 initialisieren
  pwm3_init();                              // PWM Kanal 1 initialisieren
  pwm1_start();                             // Ausgabe starten
  pwm2_start();                             // Ausgabe starten
  pwm3_start();                             // Ausgabe starten
  while(1U)                                 // Endlosschleife, immer bei Controllern
  {
    for (z=0;z!=255;z++) {RGBout (z,0,0);delay_ms(10);}
    for (z=0;z!=255;z++) {RGBout (255,z,0);delay_ms(10);}
    for (z=0;z!=255;z++) {RGBout (~z,255,z);delay_ms(10);}
  }//while
}//main
```

Für den nachfolgenden Hauptprogrammausschnitt soll nun eine Funktion RGBuebergang() geschrieben werden.

```
  while(1U)
  {       //                  R       G       B       zeit
          RGBuebergang (      heller,  aus,    aus,    10) ;
          RGBuebergang (      an,      heller, aus,    10) ;
          RGBuebergang (      dunkler, an,     heller, 10) ;
          RGBuebergang (      aus,     dunkler,an,     10) ;
          RGBuebergang (      heller,  heller, dunkler,10) ;
          RGBuebergang (      dunkler, dunkler,aus,    10) ;
  }//while
```

2.5 Wandler Gleichspannung in Wechselspannung mit PWM

In jedem PV-Wechselrichter wird aus der Gleichspannung, welche die Solarmodule liefern, eine Wechselspannung erzeugt, mit der die Energie ins Netz eingespeist werden kann. Diese Wandlung erfolgt mit schaltenden Transistoren, die mit einer PWM angesteuert werden, deren Tastgrad sich sinusförmig ändert. Auch in den meisten Elektroantrieben werden die Motoren auf diese Weise angesteuert.
Mithilfe einers Energiespeichers (Spule, Kondensator) erhält man aus der ein- und ausschaltenden Spannung einen sinusförmigen Verlauf der Spannung oder des Stromes. Oft reichen die Induktivitäten der Motorspulen aus, um aus der geschalteten Spannung mit sinusförmig sich änderndem Tastgrad einen annähernd sinusförmigen Strom zu machen.

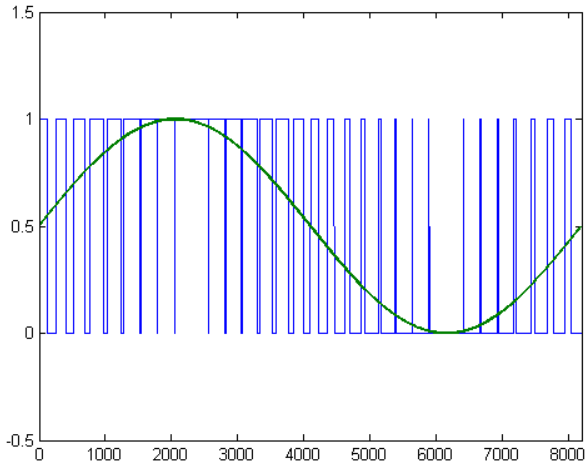


Abb 2.2 sinusförmige Änderung des Tastgrads

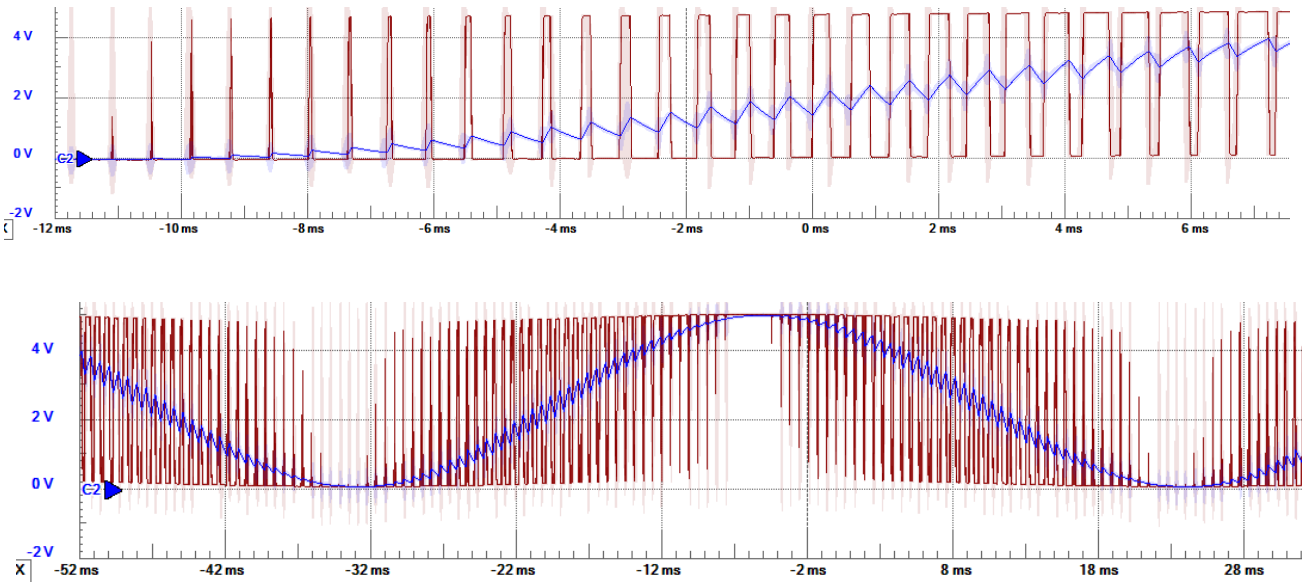
2.5.1 C-Programm Sinus mit PWM erzeugen ohne Interrupt

```
/* Sinus mit PWM erzeugen an P0.6  */

#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100
// in XMC1100-Lib.c definiert: #define periode 2550 fuer PWM-Frequenz ca. 1.5 kHz
uint8_t z;                                  // Zählvariable
uint8_t sinus [ ]= {
                    128,136,145,154,163,171,180,188,195,203,210,216,223,228,234,
                    238,243,246,249,252,254,255,255,255,255,254,252,249,246,243,
                    238,234,228,223,216,210,203,195,188,180,171,163,154,145,136,
                    128,119,110,101,92,84,75,67,60,52,45,39,32,27,21,
                    17,12,9,6,3,1,0,0,0,0,1,3,6,9,12,
                    17,21,27,32,39,45,52,60,67,75,84,92,101,110,119};
                    // Werte siehe Exceltabelle

int main(void)                              // Hauptprogramm
{
  port_init(P0,OUTP);                       // Port0 auf Ausgabe-> LEDs aus
  pwm1_init();                              // PWM Kanal 0 initialisieren
  pwm1_start();                             // Ausgabe starten
  while(1U)                                 // Endlosschleife, immer bei Controllern
  {
          for (z=0; z<90;z++)
                  { pwm1_duty_cycle(sinus[z]); // Tastgrade
                    delay_100us(1);
                  }
  }//while
}//main
```

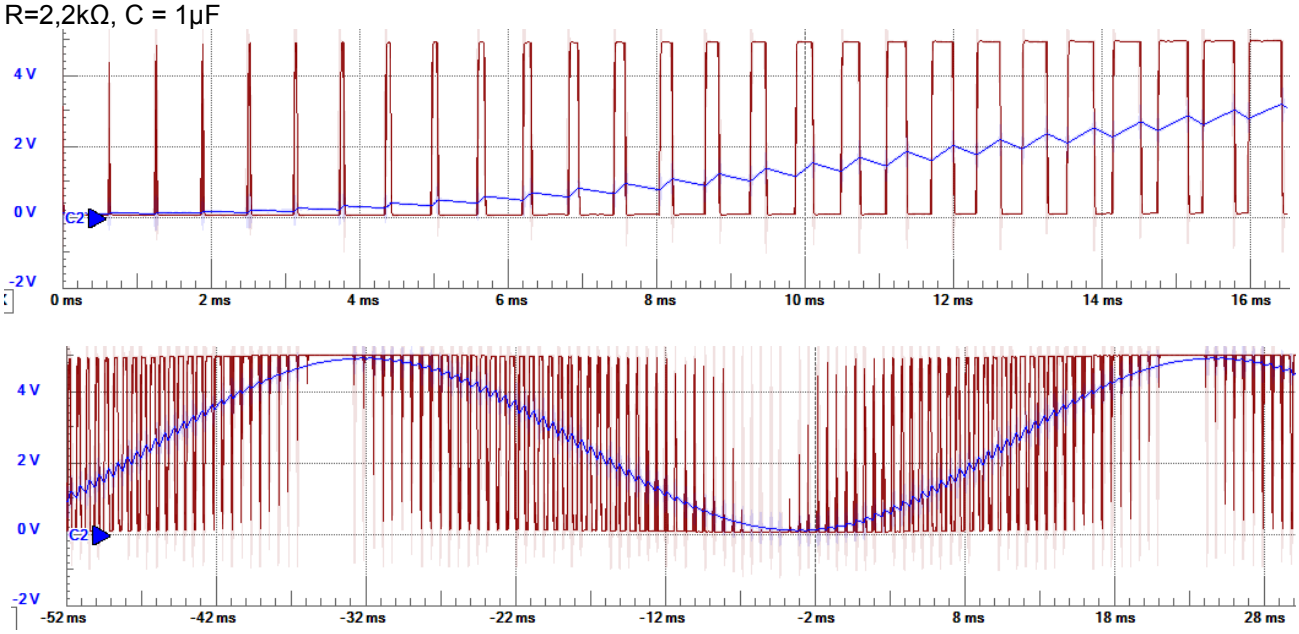
2.5.2 C-Programm Sinus erzeugen mit PWM, Interrupt nach jeder Periode

```
/* Sinus mit PWM erzeugen an P0.6, Interrupt nach jeder Periode */

#include <XMC1100-Lib.h>                // Hilfsfunktionen für XMC1100
// in XMC1100-Lib.c: #define periode 2550 fuer PWM-Frequenz ca. 1.5 kHz
uint8_t z;                              // Zählvariable
uint8_t sinus []= {
                    128,136,145,154,163,171,180,188,195,203,210,216,223,228,234,
                    238,243,246,249,252,254,255,255,255,255,254,252,249,246,243,
                    238,234,228,223,216,210,203,195,188,180,171,163,154,145,136,
                    128,119,110,101,92,84,75,67,60,52,45,39,32,27,21,
                    17,12,9,6,3,1,0,0,0,0,1,3,6,9,12,
                    17,21,27,32,39,45,52,60,67,75,84,92,101,110,119};

int main(void)                          // Hauptprogramm
{
  port_init(P0,OUTP);                   // Port0 auf Ausgabe-> LEDs aus
  pwm1_init();                          // PWM Kanal 0 initialisieren
  pwm1_start_interrupt();               // Ausgabe starten
  while(1U)                             // Endlosschleife, immer bei Controllern
  {
                                        // warten auf Interrupts
  }//while
}//main

void CCU40_0_IRQHandler (void)
{
        if (z < 90) z++;  else z = 0;   // Tabelle abarbeiten
        pwm1_duty_cycle (sinus[z]);     // neuen Tastgrad aus Tabelle holen
}
```

R=1kΩ, C = 1µF



R=2,2kΩ, C = 1µF

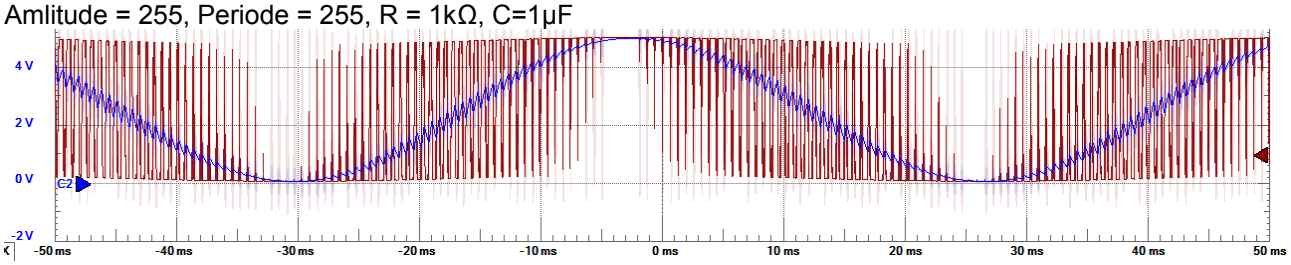


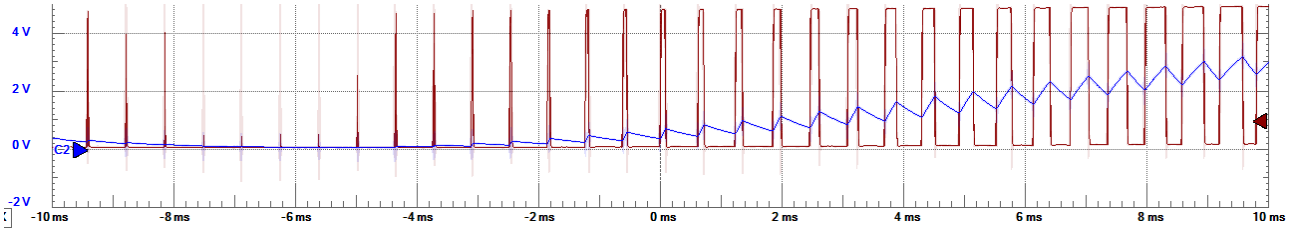
2.5.3 C-Programm Sinus mit PWM, Amplitude und Frequenz veränderbar

```
/* Sinus mit PWM erzeugen an P0.6  mit Amplitudeneinstellung am Poti links
 * und Periodendauer am Poti rechts */
#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100
uint8_t z;                                  // Zählvariable
uint16_t sinus [ ]= {                       // wegen Multiplikation unten 16Bit
                    128,136,145,154,163,171,180,188,195,203,210,216,223,228,234,
                    238,243,246,249,252,254,255,255,255,255,254,252,249,246,243,
                    238,234,228,223,216,210,203,195,188,180,171,163,154,145,136,
                    128,119,110,101,92,84,75,67,60,52,45,39,32,27,21,
                    17,12,9,6,3,1,0,0,0,0,1,3,6,9,12,
                    17,21,27,32,39,45,52,60,67,75,84,92,101,110,119};
uint16_t ampl,periode;                      // wegen Multiplikation unten 16Bit
int main(void)                              // Hauptprogramm
{ delay_ms(500);                            // Auf LC-Display-Bereitschaft warten
  lcd_init();
  lcd_setcursor ( 1,1 );                    // 1.Zeile, 1. Spalte
  lcd_print  ( "Amplitu:    links");
  lcd_setcursor ( 2,1 );                    // 2.Zeile, 1. Spalte
  lcd_print   ("Periode:    rechts");
  lcd_setcursor ( 3,1 );                    // 3.Zeile, 1. Spalte
  lcd_print   ("PWM an P0.6");
  lcd_setcursor ( 4,1 );                    // 4.Zeile, 1. Spalte
  lcd_print   ("R=1kOhm,C=1uF->Sinus");
  port_init(P0,OUTP);                       // Port0 auf Ausgabe-> LEDs aus
  adc_init();                               // Analog-Digital-Converter
  pwm1_init();                              // PWM Kanal 0 initialisieren
  pwm1_start_interrupt();                   // Ausgabe starten mit Interrupt nach jeder Periode
  while(1U)                                 // Endlosschleife, immer bei Controllern
  {
          ampl = adc_in(0)>>4;              // 8-Bit-wert von Poti links -> Amplitude
          periode = adc_in(1)>>4;           // 8-Bit-Wert von Poti rechts -> Periodendauer
          lcd_setcursor ( 1,10 );           // 1.Zeile, 10. Spalte
          lcd_byte      ( ampl  );          // Wert Poti -> Amplitude 0 ... 255
          lcd_setcursor ( 2,10 );           // 1.Zeile, 10. Spalte
          lcd_byte      ( periode  );       // Wert Poti-> Periodendauer 0 ... 255
  }//while
```

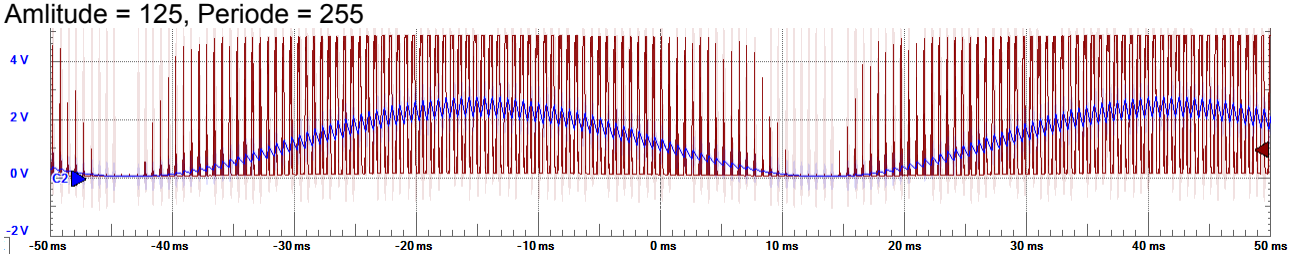
```
}//main

void CCU40_0_IRQHandler (void)                  // Interrupt-Service-Routine
{
        if (z < 90) z++;  else z = 0;          // Tabelle abarbeiten
        // Sinus hat negativsten Wert immer bei 0,
        //Offset verschieb sich nach oben mit steigender Amplitude:
        pwm1_duty_cycle_period (ampl*sinus[z]/255*periode/255,periode);
        // Amplitudewert auf Größe max 255 normieren
        // mal periode/255 damit Amplitude sich mit mit Periode ändert
}
```

Amlitude = 255, Periode = 255, R = 1kΩ, C=1µF

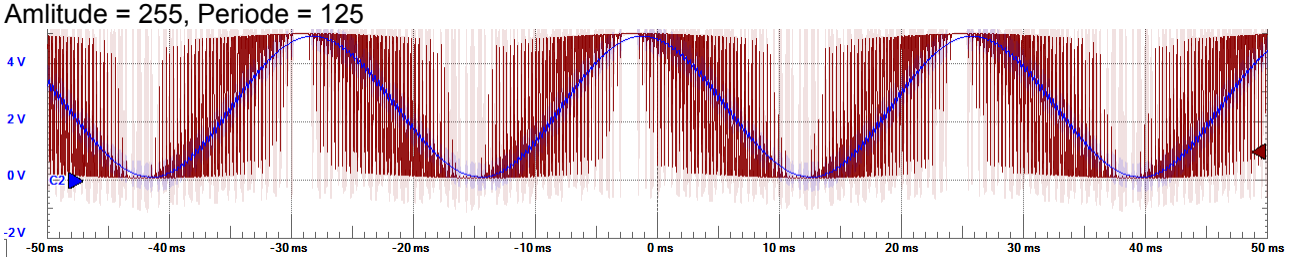




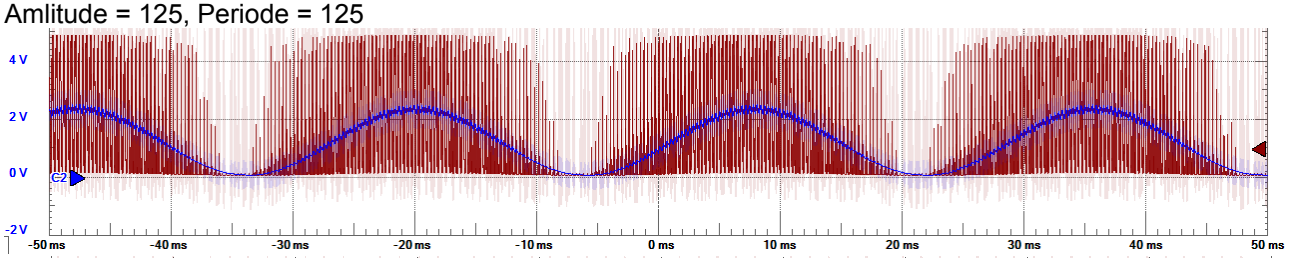
Amlitude = 125, Periode = 255

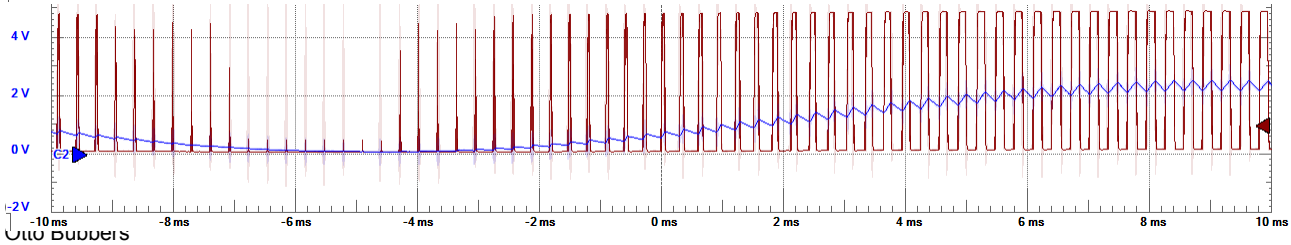


Amlitude = 255, Periode = 125



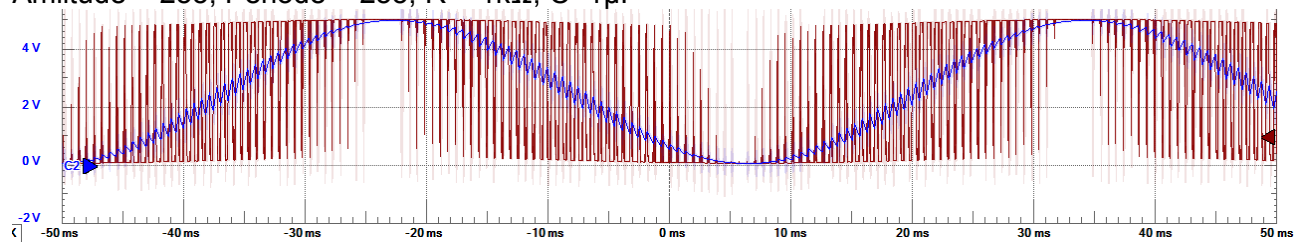
Amlitude = 125, Periode = 125

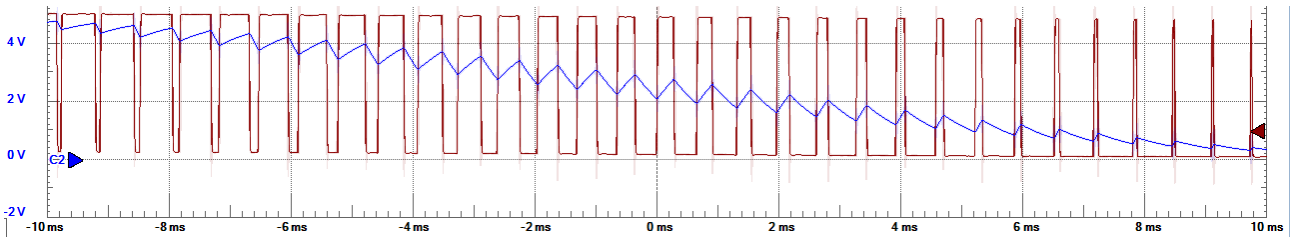




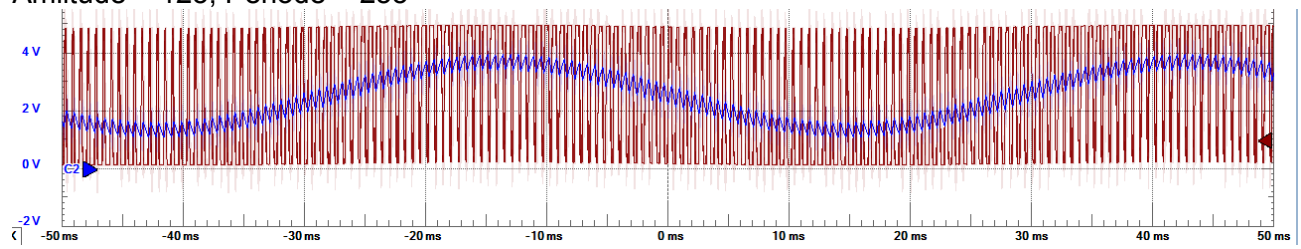
```
void CCU40_0_IRQHandler (void)              // Interrupt-Service-Routine
{
        if (z < 90) z++;  else z = 0;       // Tabelle abarbeiten
        // Sinus hat Mittelwert immer bei 2,5V:
        pwm1_duty_cycle_period (periode/2+ampl*(sinus[z]-127)*periode/255/255,periode);
        //-127 -> "nach unten" schieben, Werte werden auch negativ
        // mal Amplit, auf max 255 normieren, mal periode/255, damit Amplit unabhängig änderbar
        // und wieder um periode/2 nach oben oben verschieben (Offset des Sinus)
}
```

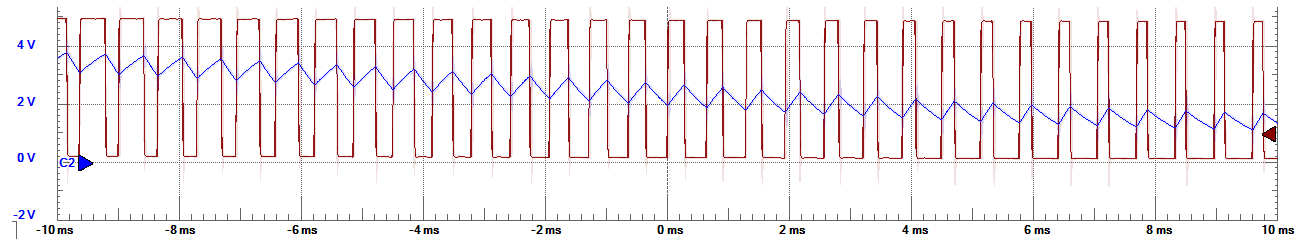
Amlitude = 255, Periode = 255, R = 1kΩ, C=1µF



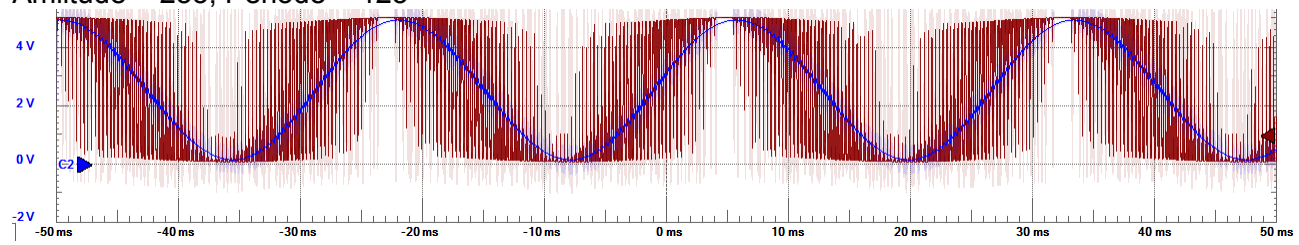


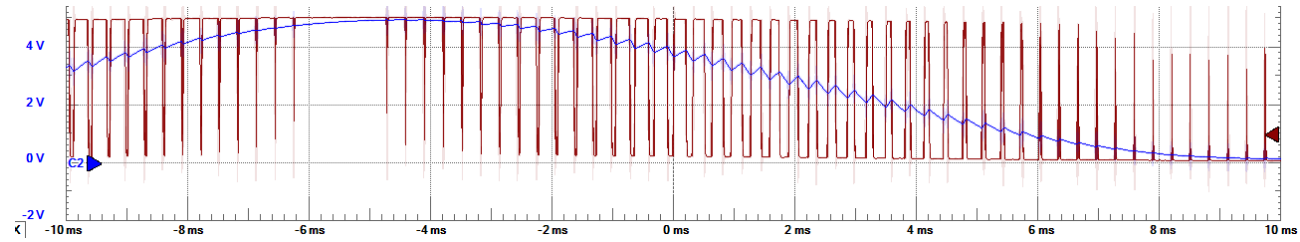
Amlitude = 125, Periode = 255



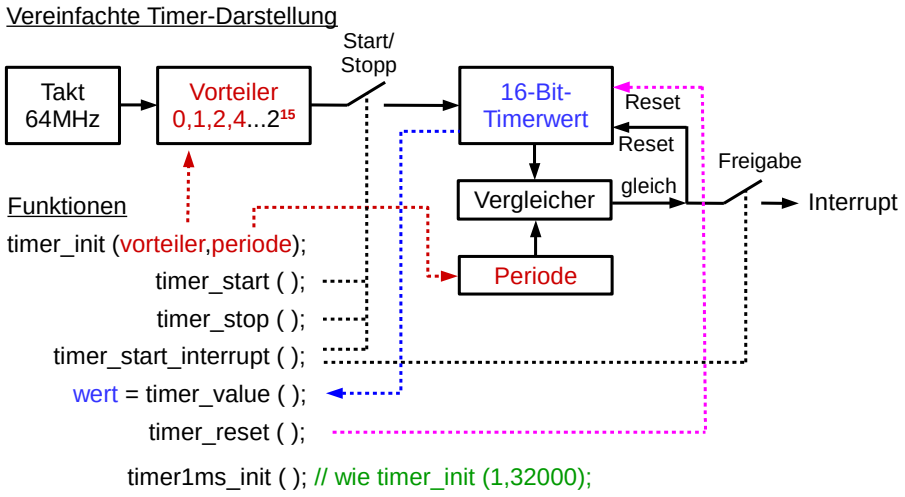


Amlitude = 255, Periode = 125





2.6 Reaktionstester mit Timer-Zeitmessung




```
/* Reaktionstester und Testen der Ganzzahl-/Floatberechnungen
 * Start mit P2.9 Stopp mit P2.2
 * Wichtig: bei der Erstellung des Simple Main Projekts Haken bei include sprintf!!!!! */
#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100
#include <stdio.h>                          // für sprintf
#define gedrueckt 0                         // lowaktive Taster
uint16_t timer_wert,zeit;
char lcdtext [20];                          // Anzeigetext 20 Zeichen

int main(void)                              // Hauptprogramm
{
  bit_init(P2,9,INP);                       // Start-Taster
  bit_init(P2,2,INP);                       // Stopp-Taster
  delay_ms(500);                            // warten bis LC bereit
  lcd_init();                               // LC-Display initialisieren
  timer_init(0xC,0xFFFF);                   // 16 uSek Timertakt -> 4,2 Sek Periode max!!!
  lcd_print ("Zeit =      ms int");
  lcd_setcursor (3,1);                      //
  lcd_print ("P2.9 Start P2.2 Stop");
  while(1U)                                 // Endlosschleife Hauptprogramm
  {
          timer_reset();                    // Timer vor dem Start rücksetzen
          while (bit_read(P2,9)!= gedrueckt);  // warten start gedrückt
          timer_start();                    // Timer starten
          while (bit_read(P2,2)!= gedrueckt);  // warten stopp gedrückt
          timer_stop();                     // Timer anhalten
          timer_wert = timer_value();       // Timer Wert auslesen
          zeit = timer_wert*64/1000;        // Zeit in ms
          lcd_setcursor (1,7);              //
          lcd_int(zeit);                    // Zeit anzeigen in ms als ganze Zahl
          lcd_setcursor (2,1);              //
          sprintf (lcdtext,"Zeit = %4.0f ms float",timer_wert*64.0/1000);
                                            // Anzeigetext mit float-Zahl erzeugen
          lcd_print (lcdtext);              // anzeigen
  }//while
}//main
```

2.7 Ultraschall-Abstandsmessung mit Modul SFR04

2.7.1 Funktionsweise des Moduls

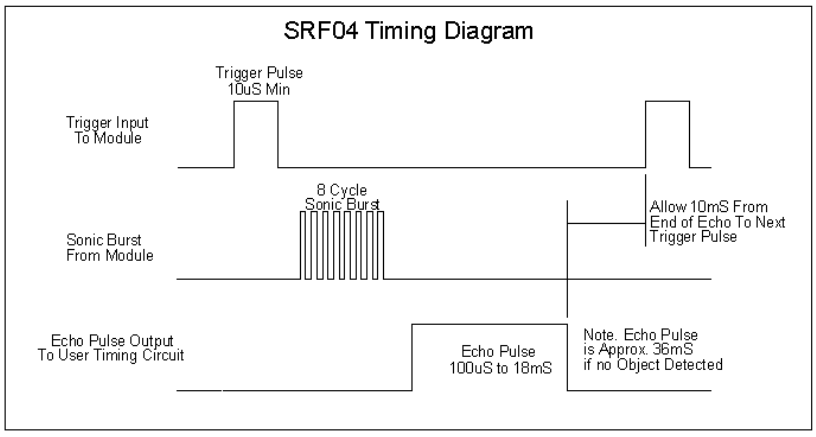
Um eine Messung zu starten, wird ein Impuls (TTL-Pegel, mind. 10us) an den Triggereingang gelegt. Der Wandler wird von der Ablaufsteuerung auf dem Modul für 200µs (8 Zyklen, 40kHz) getaktet und der Echo-Ausgang des Moduls auf High gelegt. Das erste hereinkommende Echo schaltet den Echo-Ausgang wieder auf Low, so dass ein direkt zur Entfernung des Objektes proportionaler Impuls entsteht. Der Triggerimpuls sollte nach Möglichkeit nicht länger als 200µs, muss aber auf jeden Fall vor Ende des Echo-(Mess-) Impulses wieder Low sein. Die Entfernung ergibt sich rechnerisch als Produkt aus Schallgeschwindigkeit (344m/s in Luft bei 21°C) und der Länge des Echo-Impulses. Da die Strecke vom Schall doppelt zurückgelegt wird, ist das Ergebnis durch 2 zu dividieren:
s = 344m/s * ti/ 2 , s in Meter, ti in Sekunden

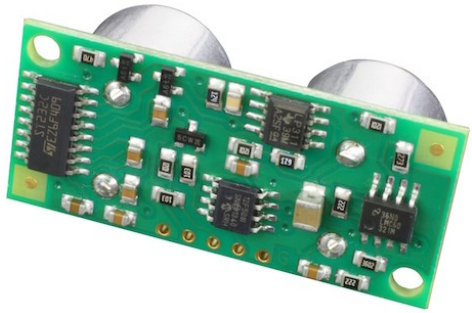
Je nach Genauigkeit und Zielsystem sind Vereinfachungen und Optimierungen möglich und sinnvoll, um z.B. mit Ganzzahlarithmetik auszukommen. z.B.: len[cm] = 172 * t[µs] / 10000

Steht Floatingpoint Arithmetik auf dem Zielsystem zur Verfügung, liefert die Gleichung len[cm] = t [µs]/ 58,066 [µs/cm] sehr gute Ergebnisse. Bei Auswertung der Entfernung über die Formel nimmt der Messfehler bei kürzeren Entfernungen zu. Das hängt damit zusammen, dass der Echoimpuls und damit die Zeitmessung erst nach dem 200us langen 40kHz Burst startet. Bei kurzen Entfernungen erreicht das Echo schon den Empfänger, bevor dieser überhaupt aktiv wird Hier bietet sich bei entsprechenden Genauigkeitsanforderungen die Umrechnung über eine Tabelle an.

Da der Spannungswandler für den Ultraschall-Sender während des Echo-Empfanges abgeschaltet wird, muss nach Ende des Echo-Pulses noch mindestens 10ms bis zum nächsten Triggerimpuls gewartet werden, damit sich die Arbeitspunkte wieder stabilisieren.

2.7.2 Timing-Diagramm





2.7.3 Kennwerte des Moduls

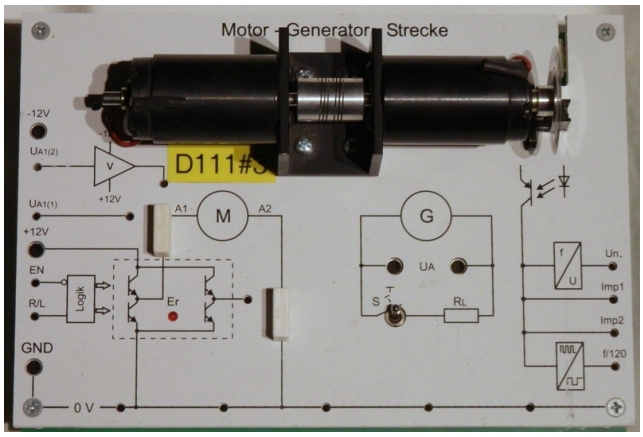
Betriebsspannung	5V (stabilisiert)
Stromaufnahme	30mA typ. 50mA max.
Frequenz	40KHz
Max. Reichweite	3 m
Min. Reichweite	3 cm
Empfindlichkeit	Erkennt 3cm Besenstiel in 2,4 m Entfernung
Triggerimpuls	10us min. TTL-Pegel Impuls
Echo Impuls	TTL-Pegel Signal, Impulsweite proportional zur Entfernung
Abmessungen	43mm x 20mm x 17mm

2.7.4 Programm

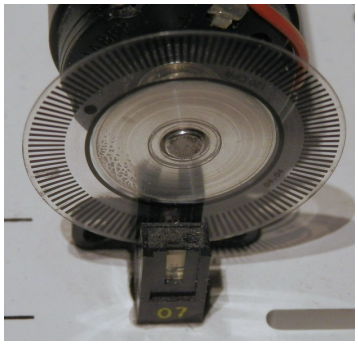
Prinzip:
1) Triggerimpuls von 100µs ausgeben: 1 ausgeben, 100µs warten, 0 ausgeben
Dann Länge des Echoimpulses messen:
2) warten bis Echo auf 1 geht, Zeitmessung starten,
3) warten bis Echo auf 0 geht, Zeitmessung stoppen
4) Wert der Zeitmessung aus Timer auslesen
5) und umrechnen in zurückgelegten Weg.

```
/* sprintf verwenden zur Anzeige von Float-Werten auf dem LC-Display
 * Wichtig: bei der Erstellung des Simple Main Projekts Haken bei include sprintf!!!!! */
#include <XMC1100-Lib.h>                // Hilfsfunktionen für XMC1100
#include <stdio.h>                      // für sprintf

uint16_t adc_wert;                      // Wert vom AD-Converter
uint16_t timer_wert;
char lcdtext [20];                      // Anzeigetext 20 Zeichen
// bei Deklaration mit uint8_t (auch int8_t ???) kommt ein Warning,
// da sprintf gerne in (signed) char schreiben möchte
float entfernung;                               // Entfernung in m

int main(void)                          // Hauptprogramm
{
  port_init(P0,OUTP);                   // Port0 auf Ausgabe Pulse
  bit_init(P1,0,OUTP);                  // Trigger
  bit_init(P1,1,INP);                   // Echo
  delay_ms(500);                        // warten bis LC bereit
  lcd_init();                           // LC-Display initialisieren
  timer_init(6,0xFFFF);                 // 1,024 uSek Timertakt-> 67.1 mSek Periode max
  while(1U)                             // Endlosschleife Hauptprogramm
  {
          timer_reset();                        // Timer vor dem Start rücksetzen
          bit_write(P1,0,1);                    // Triggerimpuls
          delay_100us(1);                       // 100µs High
          bit_write(P1,0,0);

          while (bit_read(P1,1)==0);            // warten bis Modul antwortet und Echopuls beginnt
          timer_start();                        // Timer starten

          while (bit_read(P1,1)==1);            // warten bis Modul Ende des Echoimpulses meldet
          timer_stop();                         // Timer wieder stoppen
          timer_wert = timer_value( );          // Timer Wert auslesen
          port_write(P0,timer_wert);            // Wert an LEDs dual anzeigen
          entfernung = timer_wert/58.066;       // Entfernung in cm berechnen
          lcd_setcursor (2,3);                  // 2. Zeile, 3. Spalte
        // Anzeigetext erzeugen, Entfernung mit 4 Stellen, 3 Nachkommastellen:
          sprintf (lcdtext,"Entf = %3.0f cm",entfernung);
          lcd_print (lcdtext);                  // anzeigen
          delay_ms (100);                       // ruhigere Anzeige
  }//while
}//main
```

2.8 Drehzahlmessung mit Timer und ext. Interrupt



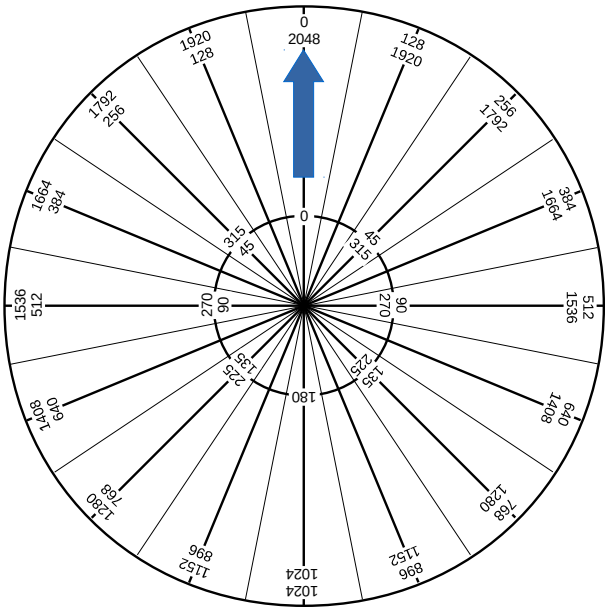

```
/* Drehzahlmessung mit Impulscheibe
 * Scheibe liefert 120 Imp pro Umdrehung
 * 100ms sind vorbei, wenn Timer_1ms 100 mal aufgerufen wurde
 * Impulsmessung mit ext. Interrupt  */

#include <XMC1100-Lib.h>                //
Hilfsfunktionen für XMC1100
uint16_t timer1ms_zaehler=0;    // zählt ms bei jedem Timerinterrupt
uint16_t impulszaehler=0;       // zählt die Impulse der Impulsscheibe (120 pro Umdrehung)
uint16_t impulse_in100ms=0;     // Zwischenspeicher für Anzahl Impulse in 100ms
uint16_t drehzahl=0;            // Drehzahl pro Minute

int main(void)                  // Hauptprogramm
{
  port_init(P0,OUTP);           // Port0 auf Ausgabe-> LEDs aus
  ext_interrupt_init();
  ext_interrupt_enable1();      // Int an P2.9
  timer1ms_init();              // Timer -Interrupt alle 1ms initialisieren
  timer_start();
  lcd_init();
  lcd_setcursor(1,1);
  lcd_print("Impulse/100ms: "); //
  lcd_setcursor(2,1);
  lcd_print("Umdr/min: ");
  while(1U)                     // Endlosschleife ----------------------------------------
  {
          port_write (P0,impulse_in100ms);      // Impulse/100ms an LEDs anzeigen
          lcd_setcursor(1,15);
          lcd_int(impulse_in100ms);             //Impulse/100ms an LCD anzeigen
          drehzahl = impulse_in100ms*5;         // imp/min = 10*60/120 * imp/100ms
          lcd_setcursor(2,15);
          lcd_int(drehzahl);
  }//while
}//main

void CCU40_3_IRQHandler (void)      // Timer-Interrupt jede 1 ms ----------------------------
        {
                timer1ms_zaehler++;                     // Zähler 100ms
                if (timer1ms_zaehler == 100)
                {  impulse_in100ms = impulszaehler;     // Wert merken
                        impulszaehler = 0;              // Impulszähler zurücksetzen
                        timer1ms_zaehler = 0;           // 100ms-Zähler zurücksetzen
                }
   }

void ERU0_3_IRQHandler(void)        // ext. Int bei steigender Flanke an P2.9 ------------
        {
                        impulszaehler++;        // Interrupt-Servie-Routine Impulse zählen
        }
```



2.9 Schrittmotor als Sekundenanzeige mit Timerinterrupt

2.9.1 Prinzip Timerprogrammierung

Der verwendete Schrittmotor besitzt ein Getriebe und benötigt daher 2048 Schritte für eine Umdrehung.

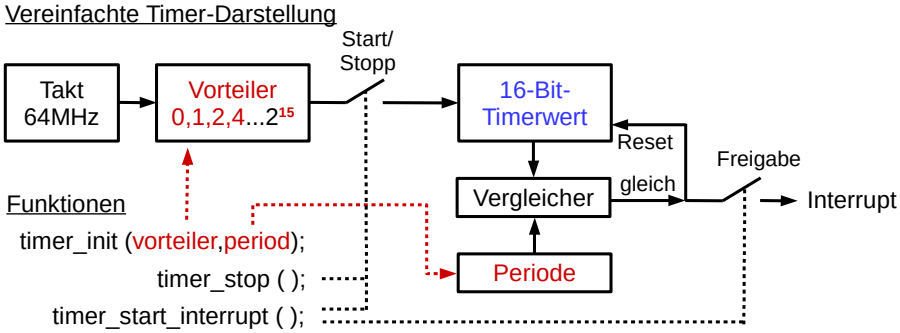


Der Motor soll für eine genaue analoge Sekundenanzeige verwendet werden. Die Ausgabe eines einzelnen Schritts erfolgt in der Timer-Interrupt-Routine. Dazu wird der Timer so programmiert, dass die Scheibe auf dem Motor (siehe Abb. Rechts) in 60 Sekunden genau eine Umdrehung vollzieht. Daher muss die Interrupt-Service-Routine genau alle

$$\frac{60\ \text{s}}{2048\ \text{Schritte}} = 0{,}029296875\ \text{s}$$

aufgerufen werden.

Berechnen Sie die Werte für den Vorteiler und die Periode des 16-Bit-Timers entsprechend.



$$T = \frac{2^{\text{vorteiler}} \cdot \text{Periode}}{64\ \text{MHz}}$$ Wertebereich Vorteiler 0,1,2,3,...15, Wertebereich Periode 1...65536

z.B. $$\text{Periode} = \frac{64\ \text{MHz} \cdot 0{,}029296875}{2^5} = 58593{,}75$$ → keine genaue Uhr möglich

→ In der ISR ist ein weiterer Frequenzteiler möglich:

Nur bei jedem 100. Aufruf der ISR darf der Schrittmotor sich drehen:

$$\text{Periode} = \frac{64\ \text{MHz} \cdot 0{,}029296875}{100 \cdot 2^1} = 9375$$

Nur bei jedem 10. Aufruf der ISR darf der Schrittmotor sich drehen:

$$\text{Periode} = \frac{64\ \text{MHz} \cdot 0{,}029296875}{10 \cdot 2^2} = 46875$$

→ timer_init (2,46875);

und in der Interrupt-Service-Routine:

```
intz++;                                  // Interrupts zählen
if (intz==10)                            // Motor ansteuern nur jeden 10. Interrupt
        {       intz=0;                          // Interruptzähler null
                motordrehung1Schritt();          // Motor um 1 Schritt drehen
        }
```

2.9.2 C-Programm Schrittmotor als Sekundenanzeige

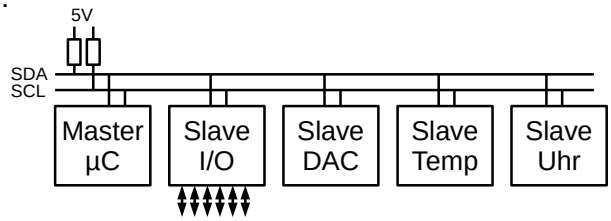
```
/* Schrittmotor als Sekundenanzeige mit Starttaster P2.2
 Anschluss an P0.4 bis P0.7 damit P0.0 bis P0.3 (Dip-Schalter) frei bleibt
 Schrittmotor mit 2048 Schritten pro Umdrehung anschließen */
#include <XMC1100-Lib.h>                    // Hilfsfunktionen fuer XMC1100
#define motor P0                            // Motorport
#define schritt1 0b01010000                 // Bitkombinationen
#define schritt2 0b01100000
#define schritt3 0b10100000
#define schritt4 0b10010000
#define taster P2                           // Tasterport
#define Tstart 2                            // P2.2 ganz rechts für Start
#define Sstart 0                            // alternativ P0.0 Schalter Start
#define San 1                               // Schalter Stellung an
#define Tgedrueckt 0                        // lowaktiver Taster
uint8_t schritt [] = {schritt1,schritt2,schritt3,schritt4};     // Array, globale Variable für die 4 Schritte
uint16_t z=0,intz;                          // globale Zählvariable für die Schritte
uint8_t motorlauf;                          // Merker Zustand Motor
#define motor_an 1
#define motor_aus 0

//---------- Hauptprogramm ----------------------------------------------------
int main(void)                              // Hauptprogramm
{
  port_init(motor,OUTP);                    // Motorport auf Ausgabe programmieren
  port_write (motor,motor_aus);             // Motor aus
  motorlauf = motor_aus;                    // Merker Zustand Motor
  timer_init(2,46875);                      // Interrupt alle 60s/2048 Schritte
                                            // damit bei einem Schrittmotorumlauf 60s vergehen
  bit_init(taster,Tstart,INP);              // Tasterpin auf Eingabe
  //bit_init(motor,Sstart,INP);             // Alternativ Schalter
  lcd_init();                               // Anzeige Anzahl Schritt seit letztem Stopp
  lcd_print("Schrittzahl: ");
  lcd_setcursor(2,1);
  lcd_print("Start-Stopp: P2.2");
  while(1U)                                 // Hauptprogramm-Endlosschleife
  {
//if (bit_read(motor,Sstart)==San) timer_start_interrupt(); else timer_stop(); //Start-Stopp mit Schalter
        if (bit_read (taster,Tstart) == Tgedrueckt)         // Start-Stopp mit Taster
        {
                while(bit_read (taster,Tstart) == Tgedrueckt);  // warten bis Taster losgelassen
                if (motorlauf == motor_aus)                 // An-Aus toggeln
                {
                        motorlauf = motor_an;               // Zustand merken
                        timer_start_interrupt();            // Timer starten mit Interrupt
                }
                else
                {
                        motorlauf = motor_aus;              // Zustand merken
                        timer_stop();                       // Timer stopp
                        port_write (motor,motor_aus);       // Motor aus wg. Strom
                        z=0;                                // Schrittzähler null setzen
                }
                // delay_ms(100);                           // ggfs Entprellung

        }//if (bit_read)
        lcd_setcursor(1,14);                                //
        lcd_int(z);                                         // Schrittzahl ausgeben
  }//while
}//main
```

```
void CCU40_3_IRQHandler (void)  // ---------- Timer-Interrupt jede Periode ----------
        {                                       // Interrupt-Servie-Routine
         intz++;                                // Interrupts zählen
         if (intz==10)                          // Motor ansteuern nur jeden 10. Interrupt
                 {
                 intz=0;                        // Interruptzähler null
                 port_write (motor,schritt [z % 4]);  // Bitkombination für Schritt ausgeben
                 z++;                           // Schrittzähler zur Abarbeitung Tabelle, Anzeige
                 }
           // Anmerkung: z % 4 ist der Rest, der bei z/4 entsteht, also 0 oder 1 oder 2 oder 3
  }
```

2.10 Kommunikation über den I²C-Bus: Port-Expander

2.10.1 Prinzip und Protokoll am Beispiel Port-Expander

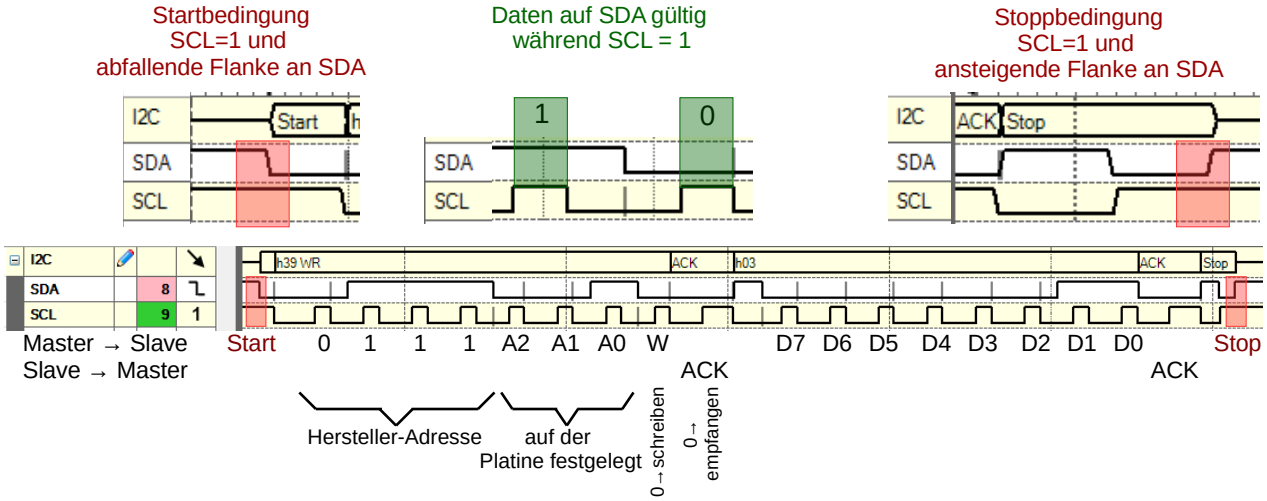
Der I²C-Bus arbeitet nach dem Master-Slave-Prinzip. Ein Datentransfer wird immer durch einen Master begonnen. Der über eine Adresse angesprochene Slave reagiert darauf.
I²C benötigt ein Taktleitung (SCL) und eine Datenleitung (SDA), beide werden durch Widerstände auf +Versorgungsspannung gezogen. OV wird durch Schalten von Transistoren auf GND im jeweils senden Baustein erreicht. Logisch 1 (High) entspricht hier +5V, logisch 0 (Low) 0V.



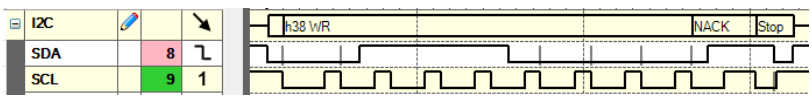
Jeder Baustein besitzt eine Adresse, dies wird als erstes Byte vom Master gesendet, wobei 4 Bit vom Hersteller festgelegt werden und 3 Bit durch Anschlussleitungen auf der Platine. Das achte Bit (R/W-Bit) teilt dem Slave mit, ob er Daten vom Master empfangen soll (LOW) oder Daten zum Master übertragen soll (HIGH).

Die Übertragung beginnt mit einer speziellen Signalkombination an den Leitungen, sie wird Startbedingung genannt. Dann überträgt der Master die Adresse und das R/W-Bit. Der angesproche Slave zieht anschließend die Leitung auf 0V und bestätigt damit den Empfang. Dies nennt man ACK-Bit. Je nachdem, ob der Slave senden oder empfangen soll, schickt nun der Master oder der slave weitere Daten. Die Taktleitung wird immer vom Master geschalten. Falls sich kein Slave angesprochen gefühlt hat, bleibt das ACK-Bit high, dies nennt man NACK. Immer nach einem Datenbyte sendet der Empänger ein ACK. Wenn der Master das letzte Byte von einem Slave gelesen hat, kündigt er das Übertragungsende mit einem NACK an. Die Übertragung endet mit der Stoppbedingung. Es wird immer das niederwertigste Bit zuerst gesendet.

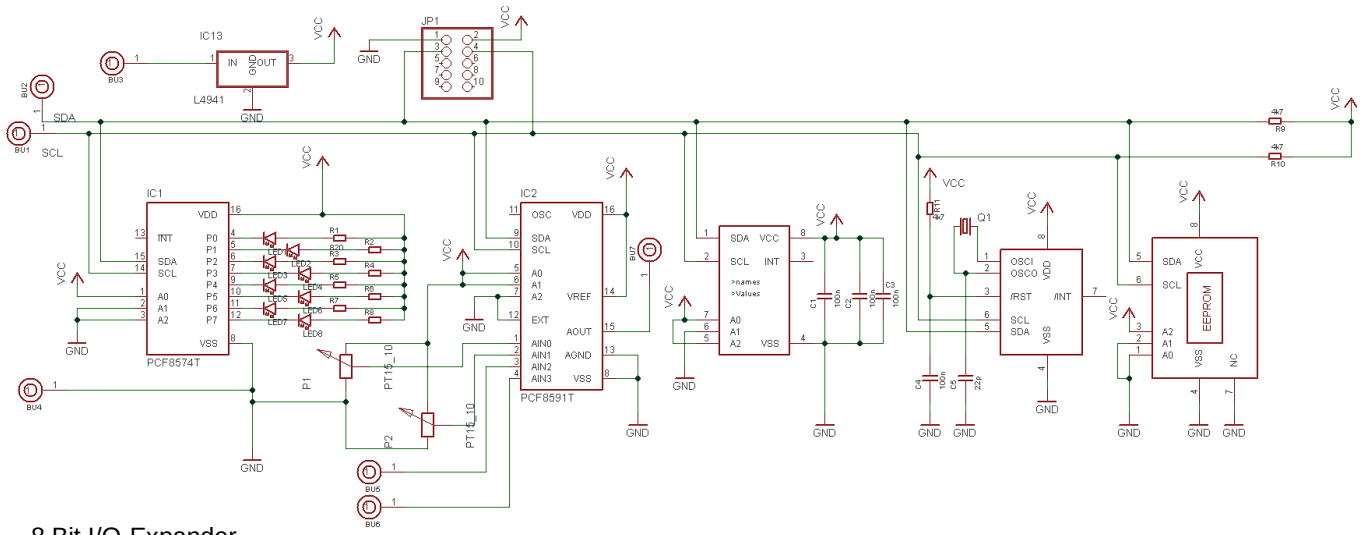
Beispiel: Übertragung der Daten 00000011 an den IO-Expander PCF8574A (Herstelleradresse 0111, IC-Adresse 001)



Beispiel: Falsche Adresse (0111000) → ACK = 1 → „NACK“



2.10.2 Platine I²C-Bus-Trainer mit I²C-Adressen



8 Bit I/O-Expander	8 Bit DA/AD-Converter	9 Bit Temperaturfühler	Realtime Clock	EEPROM 8kx8
PCF 8574A Adr.001	PCF 8591 Adr.011	DS1621 Adr.101	PCF 8593	M24C64 Adr.100
Lowaktive LEDs				

Hersteller-Adressen:

IO-Expander	PCF 8574A	**0111**001x	
IO-Expander	PCF 8574	**0100**001x	
DA/AD-Converter	PCF 8591	**1001**011x	x=0: Master schreibt
Temperaturfühler	DS1621	**1001**101x	x=1: Master liest
Realtime Clock	PCF 8593	**1010001**x	
EEPROM	M24C64	**1010**100x	

2.10.3 C-Programm I/O-Portexpander

```
/* I2C-Bus Portexpander mit lowaktiven LEDs SDA -> P1.0, SCL -> P1.1 */

#include <XMC1100-Lib.h>              // Hilfsfunktionen für XMC1100

#define schalter4 p0                  // 4 Schalter P0.0 bis P0.3
#define sendetaster 2                 // P2.2
#define ack_anzeige 4                 // P1.4
uint8_t ack_bit;                      // Ack-Bit Empfangen-> 0

int main(void)                        // Hauptprogramm
{
        port_init(P0,INP);            // später 4 Schaltern einlesen
        i2c_init();                   // I2C initialisieren
        bit_init(P2,sendetaster,INP); // für spätere Ausgabe an I2C-Bus nach Tastendruck P2.2
        bit_init(P1,ack_anzeige,OUTP); // ACK anzeigen an LED: ACK LED aus, NACK LED an
        bit_write(P1,ack_anzeige,1);  // zuerst NACK
 while(1U)                                          // Endlosschleife
 {
        while(bit_read(P2,sendetaster)==1);         //warten auf Tastendruck P2.2
        i2c_start();                                // Startbedingung
        ack_bit = i2c_write(0b01110010);            // PortExpander TypA Adr 001
        if (ack_bit==0) ack_bit=i2c_write(port_read(P0));  // 8 Bit P0 kopieren an Portexpander
                                                    // LEDs dort sind lowaktiv!!
        i2c_stop();                                 // Stoppbedingung
        bit_write(P1,ack_anzeige,ack_bit);          // ACK / NACK anzeigen an LED
 }//while
}//main
```

2.11 Temperaturmessung mit dem I²C-Sensor DS1621

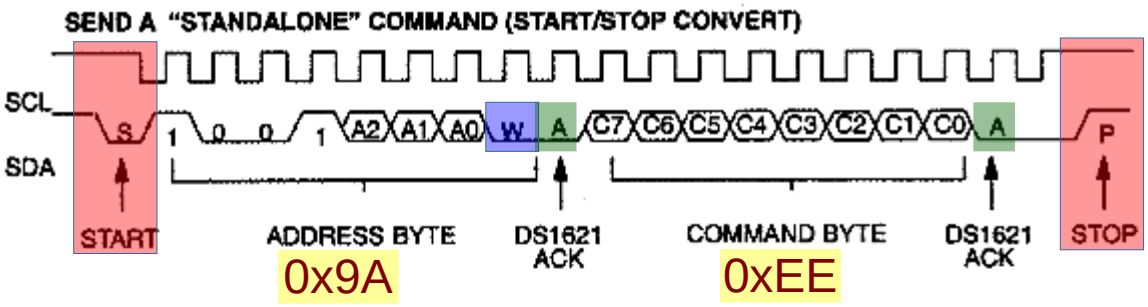
2.11.1 Protokoll Sensor DS1621

Der Sensor DS1621 kann bei Überschreiten einer frei programmierbaren Maximaltemperatur oder Unterschreiten einer Minimaltemparatur „Alarm“ geben. Wir verwenden nur die Temperaturmessfunktion, die in Schritten von 0,5°C im Bereich von -55°C bis +125°C gemessen werden kann. Die Wandlungszeit des internen Analog-Digital-Converter beträgt max 750ms.

I²C-Adresse: 1 0 0 1 A2 A1 A0 R/W wobei A2=1, A1=0, A0=1 auf der Platine festgelegt ist.

Zugriff: Nach der Übertragung der Adresse wird mit dem Kommando 0xEE die kontinuierliche Wandlung gestartet.

INSTRUCTION	DESCRIPTION	PROTOCOL	2-WIRE BUS DATA AFTER ISSUING PROTOCOL	NOTES
TEMPERATURE CONVERSION COMMANDS				
Read Temperature	Read last converted temperature value from temperature register.	AAh	<read 2 bytes data>	
Read Counter	Reads value of Count_Remain	A8h	<read data>	
Read Slope	Reads value of the Count_Per_C	A9h	<read data>	
Start Convert T	Initiates temperature conversion.	EEh	idle	1
Stop Convert T	Halts temperature conversion.	22h	idle	1



Danach werden 2 Bytes aus dem 16-Bit-Temperaturregister gelesen. Nur 9 Bit sind von Bedeutung.

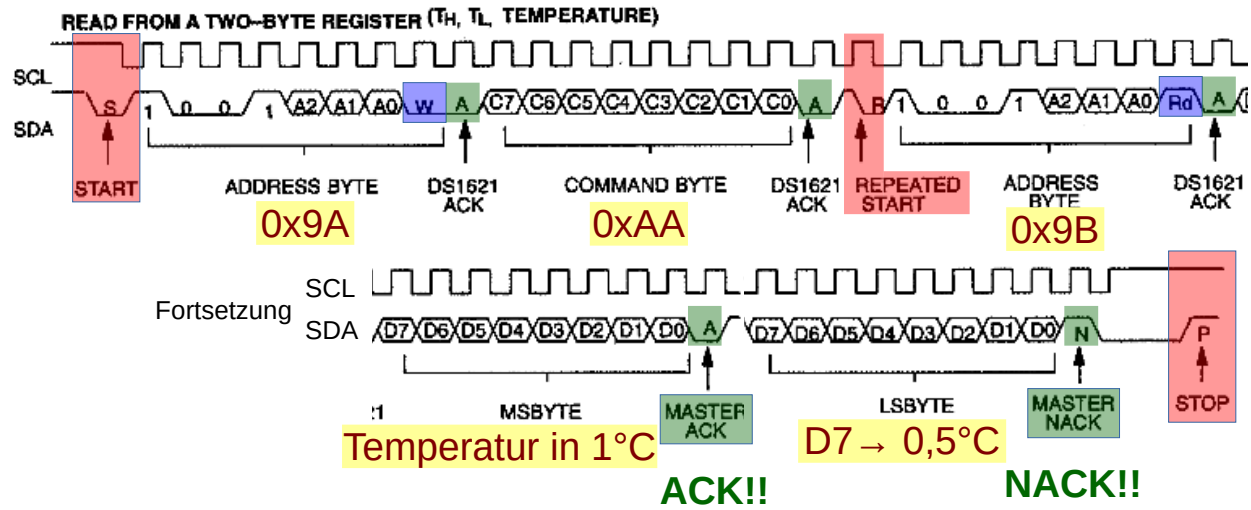
MSB								LSB							
1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0

T = -25°C

Das LSB stellt die 0,5°C-Nachkommastelle dar. Die vorderen 8 Bit stellen die Temperatur in Grad als int8_t dar, also von -128 bis +127, wobei der messbereich des Sensors nur von -55 bis +125 reicht.

TEMPERATURE	DIGITAL OUTPUT (Binary)	DIGITAL OUTPUT (Hex)
+125°C	01111101 00000000	7D00h
+25°C	00011001 00000000	1900h
+½°C	00000000 10000000	0080h
+0°C	00000000 00000000	0000h
-½°C	11111111 10000000	FF80h
-25°C	11100111 00000000	E700h
-55°C	11001001 00000000	C900h

Temperatur lesen

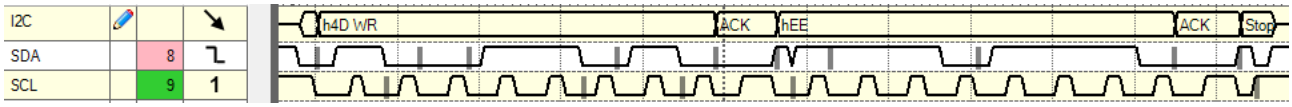


2.11.2 C-Programm Temperaturmessung mit DS1621

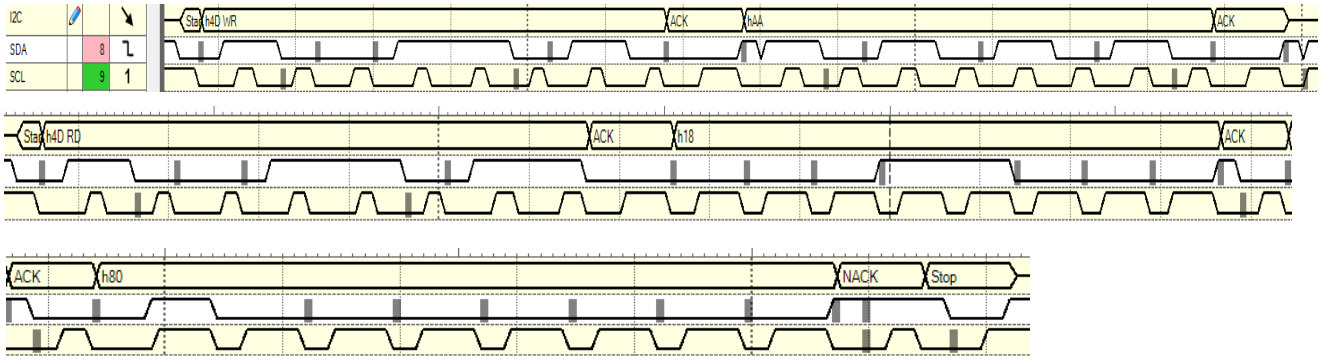
```
/* I2C-Bus Temperaturmessung SDA -> P1.0, SCL -> P1.1 */
#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100
#define ack_anzeige 4
uint8_t ack_bit;                            // Ack-Bit Empfangen-> 0
uint8_t tempL,temp;                         // Temperaturspeicher

int main(void)                              // Hauptprogramm
{
      delay_ms(1000);                       // warten bis LCD-Controller gestartet
      lcd_init();                           // LCD initialisieren
      i2c_init();                           // I2C initialisieren
      bit_init(P1,ack_anzeige,OUTP);        // ACK LED aus, NACK LED an
      bit_write(P1,ack_anzeige,1);          // zuerst NACK
      i2c_start();                          // Startbedingung
      i2c_write(0x9A);                      // Adresse, R/W=0
      i2c_write(0xEE);                      // kontinuierliche Wandlung
      i2c_stop();                           // Stoppbedingung
      lcd_print ( "Temperatur" );           // Textausgabe LCD
  while(1U)                                 // Endlosschleife
  {
        i2c_start();                        // Startbedingung
        i2c_write(0x9A);                    // Adresse R/W=0
        i2c_write(0xAA);                    // Command_ Temperatur lesen , R/W=0
        i2c_start();                        // repeatet Start
        ack_bit=i2c_write(0x9B);            // Adresse und R/W=1
        temp=i2c_read(ACK);                 // Temp in Grad lesen
        tempL=i2c_read(NACK);               // Temp 0,5 Grad, letztes Lesebyte
        i2c_stop();                         // Stoppbedingung
        lcd_setcursor ( 2,1 );              // Cursorposition LCD 2. Zeile
        lcd_byte ( temp );                  // Temperatur anzeigen
        lcd_char ('.');                     // Komma
        if (tempL==0) lcd_char ('0'); else lcd_char ('5'); // Nachkommastelle
        lcd_char('°');lcd_char('C');        // Grad C
        bit_write(P1,ack_anzeige,ack_bit); // Ack-Bit IC angesprochen?
  }//while
}//main
```

2.11.3 Beobachteter Datenaustausch

Initialisierung



Temperatur lesen



übertragener Temperaturwert: hex18 → 24°, hex80 → +0,5° => Temperatur = 24,5°C

2.11.4 Aufgabe: Abfrage AD-Wandlung fertig?

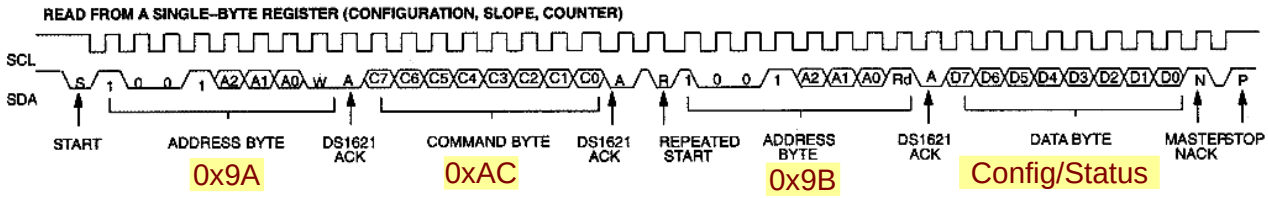
Nach dem Einschalten wird zuerst eine falsche Temperatur (196°) angezeigt. Dies liegt daran, dass der AD-Converter noch nicht fertig ist. Daher soll das Programm so abgeändert werden, dass erst dann eine Temperatur gelesen wird, wenn der ADC die Wandlung beendet hat. Dazu muss das configuration/status-Register abgefragt werden:

configuration/status register:

DONE = Conversion Done bit. “1” = Conversion complete, “0” = Conversion in progress

Access Config [ACh]

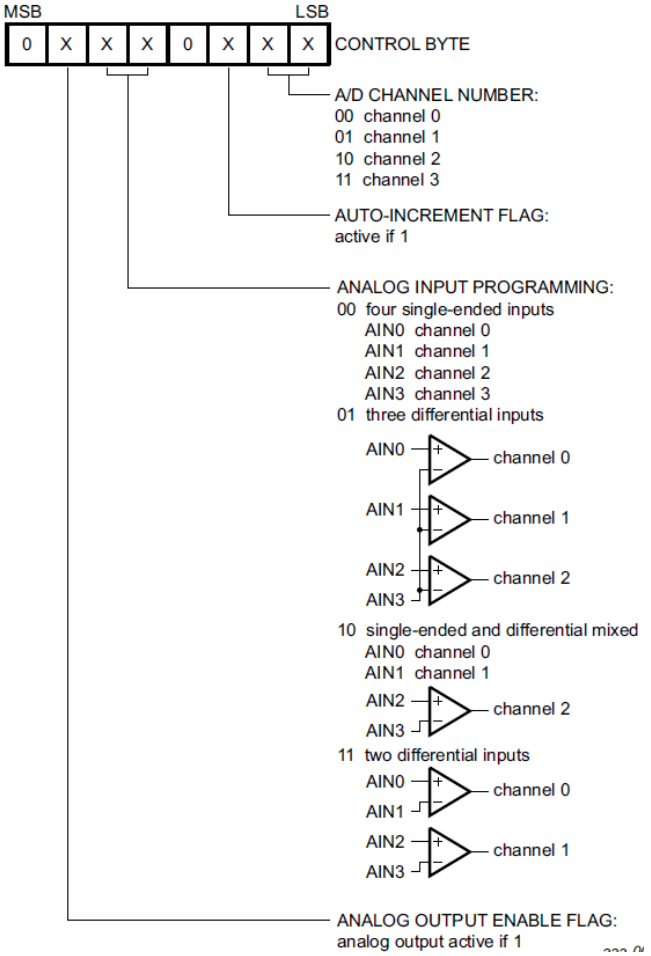
If R/ W is “0” this command writes to the configuration register. After issuing this command, the next data byte is the value to be written into the configuration register. If R/W is “1” the next data byte read is the value stored in the configuration register.



2.12 Sinusausgabe über I²C-DAC PCF 8591

Adresse:
1001011R/W
R/W=0 schreiben R/W=1 lesen

Control-Byte siehe rechts.

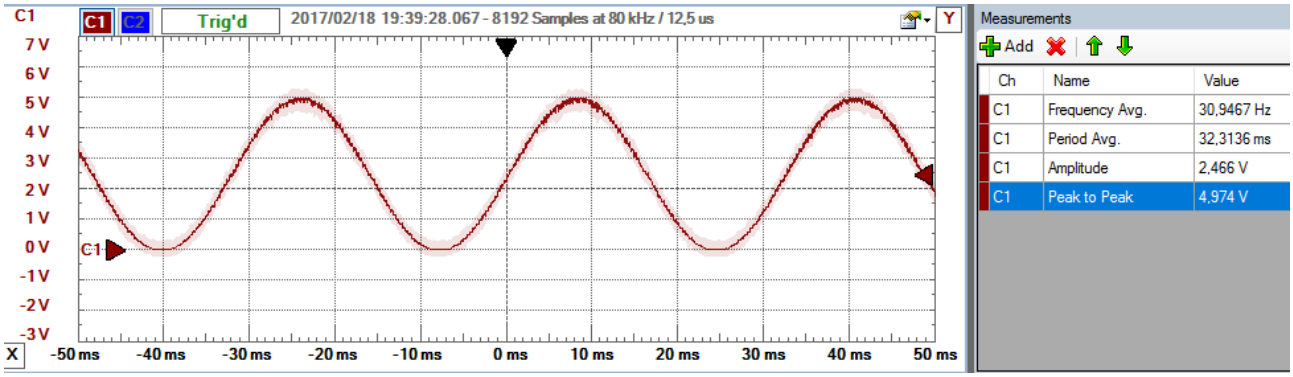


Protokoll DAC
Startbedingung
Adresse schreiben
Control-Byte schreiben
beliebig viele Datenbytes an den DAC
Stoppbedingung

Immer mit dem Schreiben des Datenbytes wird das vorhergehende ausgegeben.

2.12.1 Aufgabe

Aus einer Tabelle werden 90 Sinuswerte nacheinander an den DAC ausgeben.



Am Ausgang des DAC entsteht eine sinusförmige Spannung.
Die Ausgabe soll mit dem Druck auf den taster P2.2 unterbrochen werden.

2.12.2 C-Programm Sinus mit I²C-DAC

```
/* I2C-Bus DA-Converter SDA -> P1.0, SCL -> P1.1
 * Sinus aus 90 Werten ausgeben*/

#include <XMC1100-Lib.h>                    // Hilfsfunktionen für XMC1100

uint8_t z=0;                                // Zählvariable
uint8_t sinus [ ]= {
                    128,136,145,154,163,171,180,188,195,203,210,216,223,228,234,
                    238,243,246,249,252,254,255,255,255,255,254,252,249,246,243,
                    238,234,228,223,216,210,203,195,188,180,171,163,154,145,136,
                    128,119,110,101,92,84,75,67,60,52,45,39,32,27,21,
                    17,12,9,6,3,1,0,0,0,0,1,3,6,9,12,
                    17,21,27,32,39,45,52,60,67,75,84,92,101,110,119};

int main(void)                              // Hauptprogramm
{
  i2c_init();                               // I2C initialisieren
  bit_init(P2,2,INP);
  while(1U)                                 // Endlosschleife
  {
            i2c_start();                    // Startbedingung
            i2c_write(0x96);                // Adresse, Master schreibt R/W=0
            i2c_write(0x40);                // Controlbyte DAC-Ausgang freigeben
            while (bit_read(P2,2)==1)       // ausgeben bis Taste gedrückt
            {
                      i2c_write(sinus[z]);  // Wert ausgeben
                      if (z < 89) z++;  else z = 0;  // Tabelle abarbeiten
            }// while (bit-read)
            i2c_stop();                     // Stoppbedingung
  }//while
}//main
```

2.12.3 Erweiterungen

- Amplitude über Poti änderbar

2.13 Datenübertragung COM-Schnittstelle

2.13.1 Ascii „A“ dauernd senden, 1Byte empfangen und binär und als Ascii darstellen

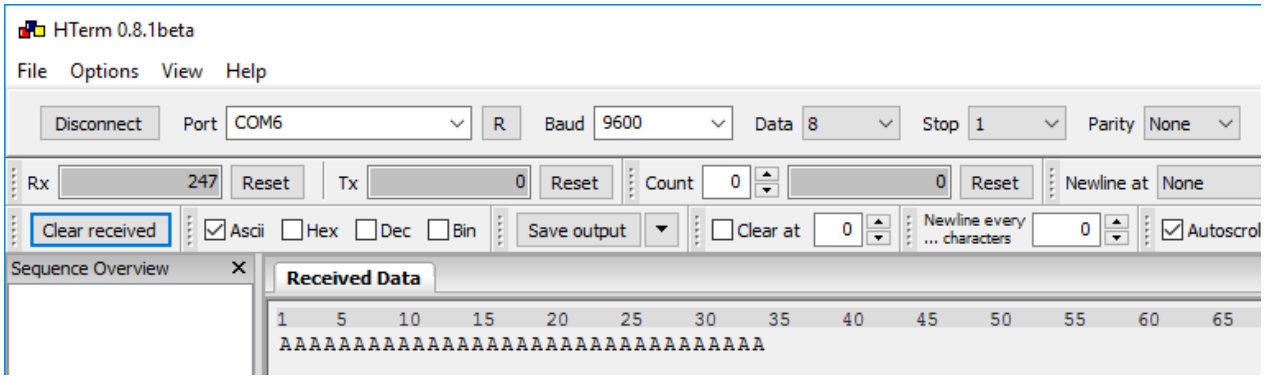
```
/* Test serielle Kommunikation, Am PC HTerm starten mit 9600Baud und Connect
drücken */
#include <XMC1100-Lib.h>                    // Hilfsfunktionen fuer XMC1100

uint8_t sendebyte,empfbyte;
int main(void)                              // Hauptprogramm
{ delay_ms(500);                            // Start LCD-controller
  rs232_init();                             // Com-Schnittstelle initialisieren
  lcd_init();                               // LCD initialisieren
  port_init(P0,OUTP);                       // Port0 auf Ausgabe
  while(1U)                                 // Endlosschleife
  {
        sendebyte = 'A';                    // Code vom Ascii-Zeichen A bestimmen
        //sendebyte = 0x41;                 // alternativ als Hexzahl
        rs232_put ( sendebyte );            // senden
        //rs232_print ( char *text )
        empfbyte = rs232_get();             // 1 Byte abholen
        if (empfbyte != 0)
              {
              port_write(P0,empfbyte);      // und als Dualzahl an LEDs ausgeben
              lcd_setcursor(1,1);           // links oben
              lcd_char(empfbyte);           // anzeigen
              }
      delay_ms (100);                       // Zeitverzoegerung
  }//while
}//main
```

2.13.2 Oszillogramm

Grundzustand 1, dann Startbit 0, dann 10000010, dann Stoppbit=Grundzustand der Leitung
Links steht das zuerst empfangene, LSB → Binär 01000001 = hex 41 = Ascii „A“

Beim Controller wird dies über die USB-Schnittstelle und über P1.2 übertragen, dort kann man oszilloskopieren.

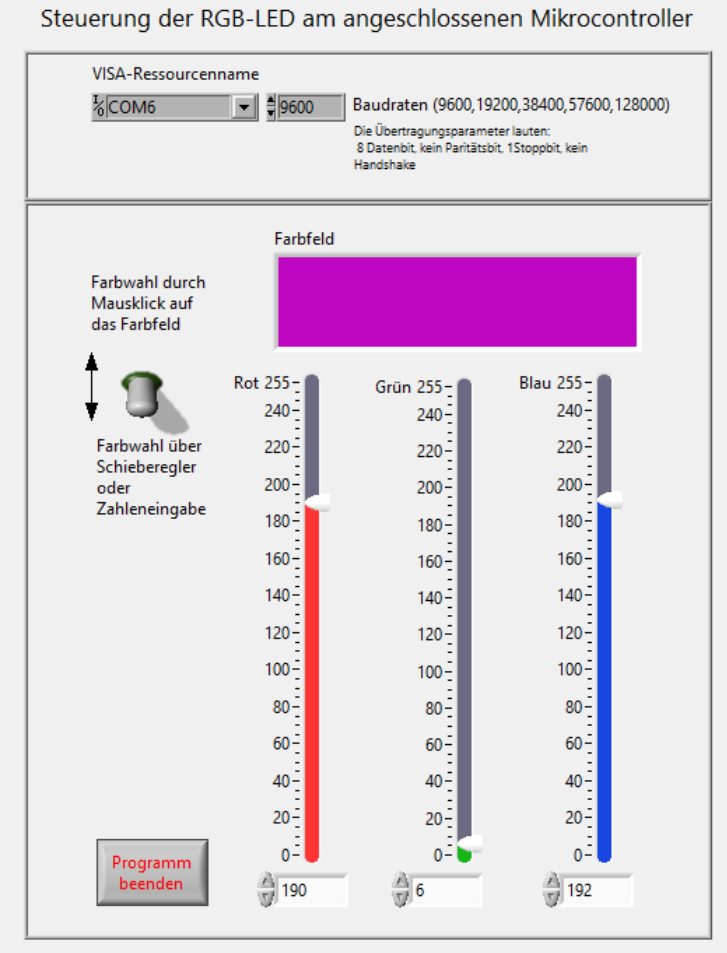
2.13.3 Hterm-Programm



Empfangene Zeichen werden als Ascii-Zeichen dargestellt: AAAA…
Protokoll: 9600 Bit/s, 8 Datenbit, 1 Sopbit, kein Paritätsbit
USB-Verbindung der Entwicklungsumgebung Dave wird für die Datenübertragung verwendet um am PC als COM6 für die Com-Verbindung verwendet.

2.13.4 Fernsteuerung einer RGB-LED-Lichterkette am Controller über Labview am PC

- C-Programm (unten) am Controller starten.
- Labview am PC starten
- Schnittstelle wählen.
- Farbe der RGB-LED mit Schiebereglern oder ober das Farbfeld einstellen.
- Die Kommunikation erfolgt über die COM-Schnittstelle. 3 Bytes übertragen die Tastgrade der PWM-Signale, die der Controller erzeugt.




```
/* Serielle Kommunikation: Labview-Programm steuert RGB-LED am Controller  */
#include <XMC1100-Lib.h>              // Hilfsfunktionen fuer XMC1100
uint8_t r_byte, g_byte, b_byte;
int main(void)                        // Hauptprogramm
{
  pwm1_init();                        // PWM Kanal 0 initialisieren
  pwm2_init();                        // PWM Kanal 1 initialisieren
  pwm3_init();                        // PWM Kanal 1 initialisieren
  pwm1_start();                       // Ausgabe starten
  pwm2_start();                       // Ausgabe starten
  pwm3_start();                       // Ausgabe starten
  pwm1_duty_cycle(0);                 // Tastgrade
  pwm2_duty_cycle(0);
  pwm3_duty_cycle(0);
  rs232_init();
  while(1U)                                   // Endlosschleife
  {     if (rs232_char_received()!=0){
              r_byte = rs232_wait_get();      // Tastgrad für Rot vom PC holen
              g_byte = rs232_wait_get();      // Tastgrad für Gruen vom PC holen
              b_byte = rs232_wait_get();      // Tastgrad für Blau vom PC holen
              }// if
        pwm1_duty_cycle(~r_byte);             // Tastgrade
        pwm2_duty_cycle(~g_byte);             // Invertierung bei lowaktiver RGB-LED
        pwm3_duty_cycle(~b_byte);
  }//while
}//main
```