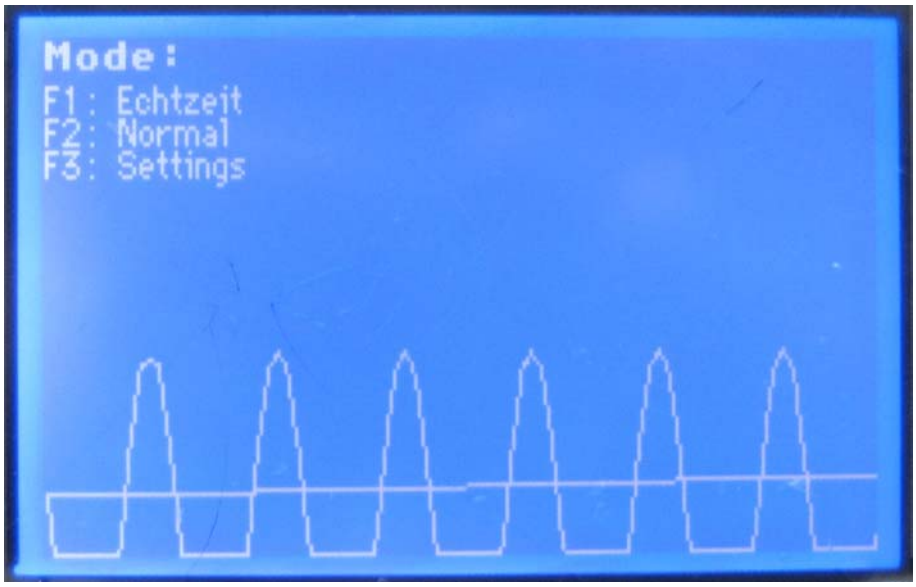
Selbstgebautes portables digitales PIC-gestütztes Oszilloskop



Wettbewerb „Jugend Forscht“ 2010

Lucas Jürgens (14 Jahre)

Arbeitsgemeinschaft "Jugend Forscht"
des Christian-Gymnasiums Hermannsburg

Leitung: StD Thomas Biedermann

Inhaltsverzeichnis

1 Zielsetzung des Projekts

Das Ziel meinen Projektes ist es, ein einfaches Oszilloskop zu bauen, welches möglichst klein und portabel ist. Um dies umzusetzen verwende ich einen PIC-Mikrokontroller, ein graphisches Display zur Ausgabe und eine Tastatur zur Eingabe von Daten.

2 PIC-Mikrocontroller

2.1 Funktionseinheiten

Der von mir verwendete PIC18F452 verfügt über mehrere Komponenten:

- 32 KByte Programmspeicher
- 1536 Byte Datenspeicher
- Interrupts
- 5 I/O-Ports
- 4 Timer
- 2 Pulsweitenmodulatoren (PWM)
- 3 Hardware-Schnittstellen USART, I²C, SPI
- 8 AD-Wandler mit 10 Bit Auflösung

2.2 Programmierung

Damit der Mikrocontroller tut, was er soll, muss man ihn mit einem vorher geschriebenen Programm versehen. Ich verwende dazu die Software MPLAB und den MC18-Compiler von Microchip (Abb. 1).

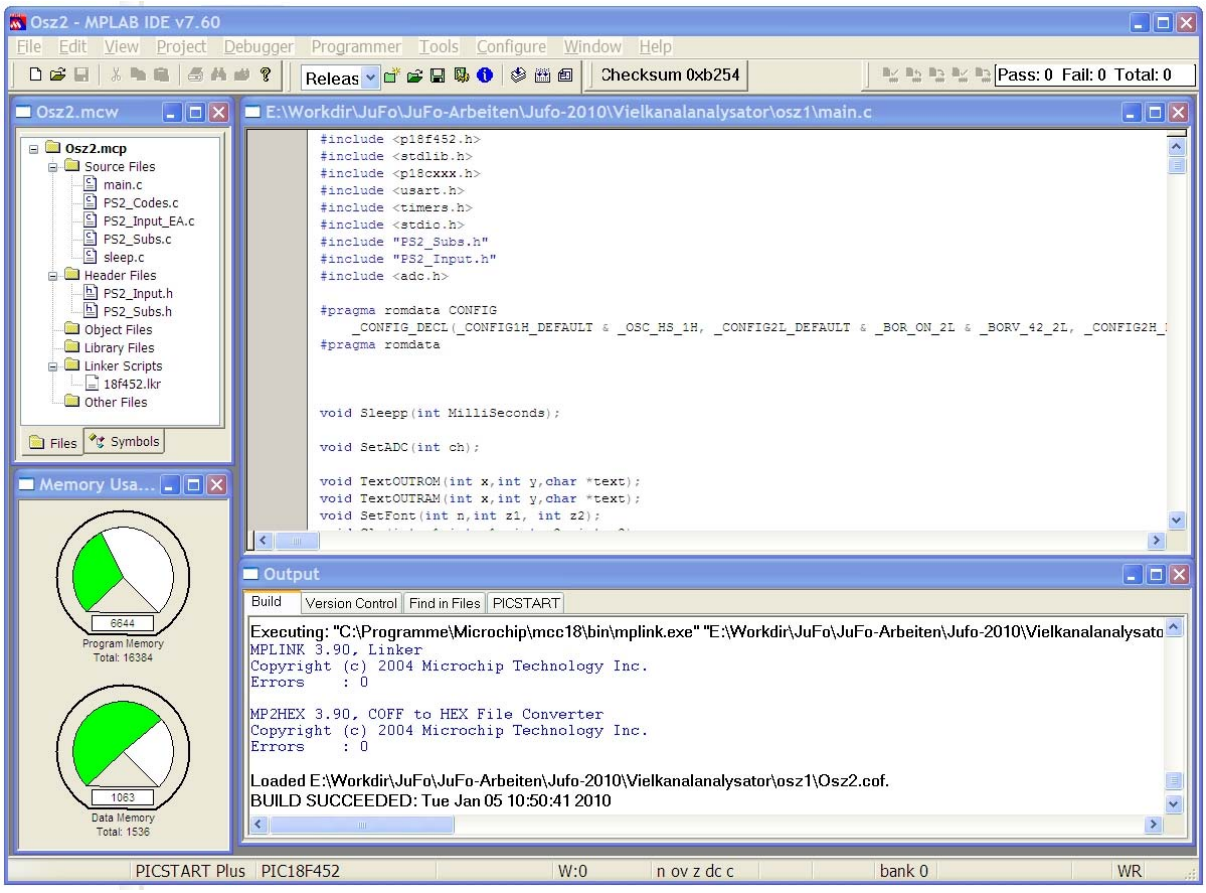


Abb. 1: Screenshot der Programmieroberfläche MPLAB IDE

Damit kann man den Mikrocontroller in der Programmiersprache „C“ programmieren.
Ein Programm um eine Leuchtdiode an dem Port RA0 blinken zu lassen sieht darin zum Beispiel so aus:

```
#include <p18f452.h>          // Einbindung von Bibliotheken
#include <delays.h>           // Einbindung von Bibliotheken

#pragma romdata CONFIG        // Systemkonfigurationen
    _CONFIG_DECL(_CONFIG1H_DEFAULT & _OSC_HS_1H,
    _CONFIG2L_DEFAULT & _BOR_ON_2L & _BORV_42_2L,
    _CONFIG2H_DEFAULT & _WDT_OFF_2H, _CONFIG3H_DEFAULT,
    _CONFIG4L_DEFAULT & _LVP_OFF_4L, _CONFIG5L_DEFAULT,
    _CONFIG5H_DEFAULT, _CONFIG6L_DEFAULT,
    _CONFIG6H_DEFAULT, _CONFIG7L_DEFAULT,
    _CONFIG7H_DEFAULT);
#pragma romdata

void main(void) {             // Beginn der Hauptroutine
    TRISA = 0b11111110;       // RA0 an PortA auf 0 = Ausgang setzten

    while(1) {                // Beginn der Endlosschleife
        LATAbits.LATA0 = 1; // LED einschalten
        Delay1KTCYx(4000);  // 4000 mal 1K Taktzyklen = 1 s warten (bei 16 MHz)
        LATAbits.LATA0 = 0; // LED ausschalten
        Delay1KTCYx(4000);  // 1 Sekunde warten
    }                         // Ende der Endlosschleife
}                             // Ende der Hauptroutine
```

Abb. 2: Einfaches Programmbeispiel zum Blinken einer LED

Wenn dann ein Programm fertig ist, muss es kompiliert werden. Wenn das zuvor geschriebene Programm fehlerfrei ist, erstellt der Compiler den zum C-Script passenden Maschinencode, der anschließend auf den PIC übertragen werden kann. Nachdem ich in den vergangenen Jahren mit einem Programmiersystem gearbeitet hatte, das den (mittlerweile veralteten) Parallel-Port des PC benutzte, verwende ich für die Übertragung nunmehr ein USB-gestütztes PC-Interface „PIC-Kit 3“ (Abb. 3) der Firma Microchip, mit dem sich der Code sehr viel schneller auf den PIC schreiben lässt.

Abb. 3: Programmiergerät PIC-Kit 3

3 Hardware

Die gesamte Hauptplatine (s. Abb. 4) enthält alle für den Betrieb des PIC (Mitte) benötigten Komponenten sowie universell nutzbare Steckerleisten zum Anschluss von Erweiterungen oder Peripheriegeräten. Ein PS/2-Anschluss (links) erlaubt den Anschluss einer PC-Tastatur, ein 16-poliger Pfostenverbinder (Mitte oben) den Anschluss eines standardisierten LC-Textdisplays. Der Schaltplan sowie das Platinenlayout wurden mit Eagle Light 4.1 erstellt. Das Leiterbahnlayout für die Platine wurde mit einem Laserdrucker ausgedruckt und anschließend mit dem Tonerverfahren auf das Platinenmaterial übertragen. Nach dem Ätzen und Bohren konnte die Platine bestückt und in Betrieb genommen werden. Den zugehörigen Schaltplan zeigt Abb. 5 auf der nachfolgenden Seite.

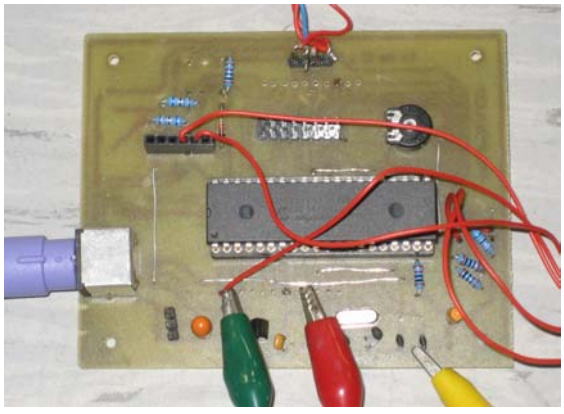


Abb. 4: Hauptplatine mit dem PIC18F452

3.1 Schaltplan

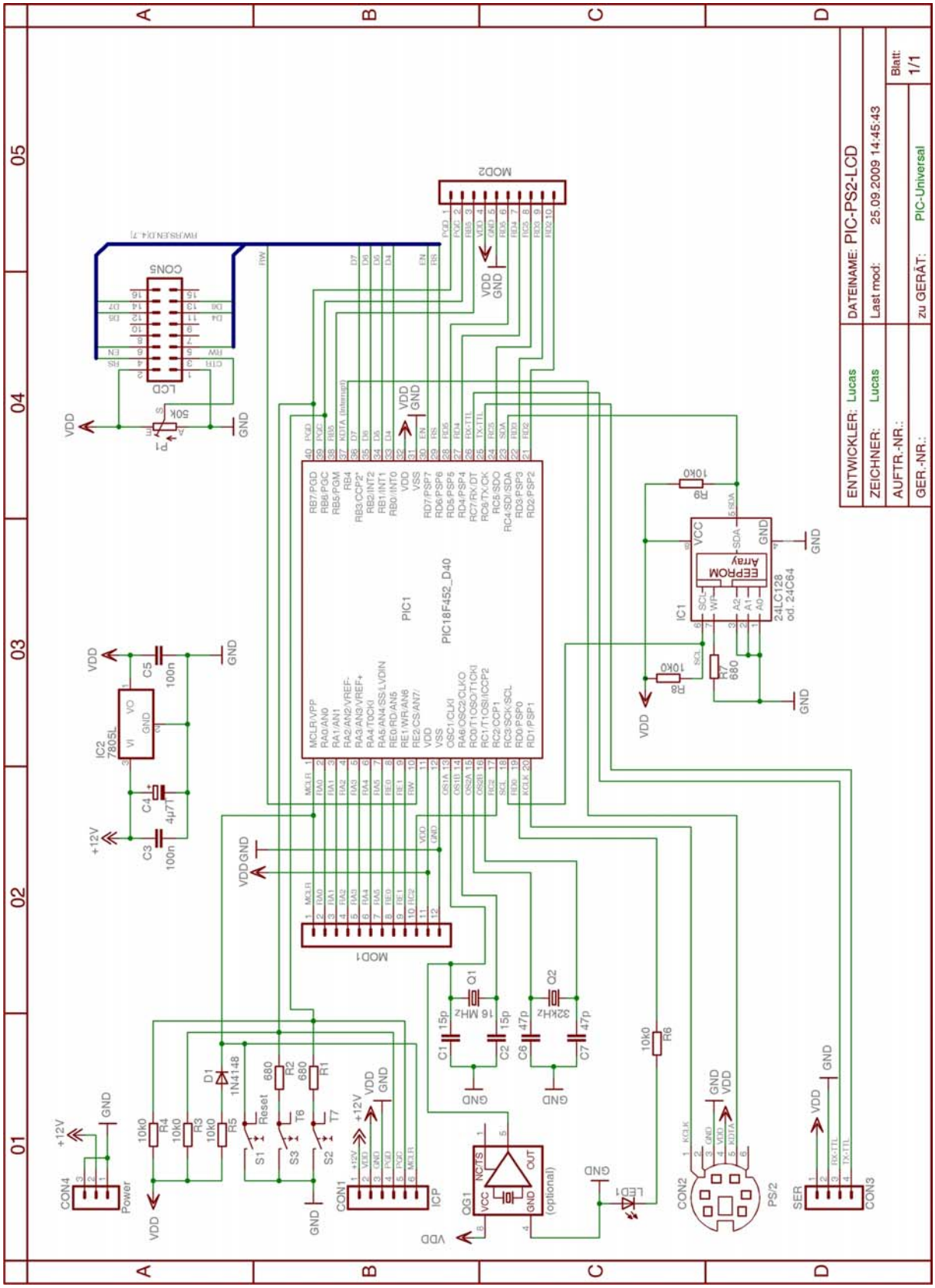


Abb. 5: Schaltplan der PIC-Platine
Con1: Steckverbinder, PC-Interface „PIC-KIT 3“
Con2: Buchse PS2, Tastaturanschluss
Con3: Steckverbinder, Display-Anschluss, RS232
Con4: Steckverbinder, Stromversorgung 12V
Con5: Steckverbinder, Anschluss für opt. LC-Display
Mod1: Steckerweiterung für weitere Platinen
Mod2: “
IC1: opt. Speicherbaustein (nicht verwendet)
IC2: Spannungsregler, 5V
Q1: Quarz, 16Mhz
Q2: opt. Uhrenquarz (nicht verwendet)

3.2 Display mit Schnittstellen

Damit ich die gemessenen Daten möglichst gut darstellen kann, verwende ich das grafische Display eDIP240B-7LWTP von Electronic Assembly (EA). Es hat eine Auflösung von 240 x 128 Bildpunkten und verfügt über einen Touchscreen. Das Display verfügt dank des eingebauten Mikrokontrollers über sehr leistungsfähige Grafikbefehle, erlaubt die Speicherung von wiederkehrenden Befehlsfolgen in Form von Makros und stellt verschiedene auch ladbare Zeichensätze und Bitmap-Grafiken auf Abruf zur Verfügung. Der Touchscreen ersetzt außerdem in weiten Grenzen eine sonst notwendige externe Tastatur, was ebenfalls durch leistungsfähige Makros unterstützt werden kann. Die gesamte Kommunikation mit dem Display erfolgt über einfache Befehlssequenzen, die sowohl in binärer Form als auch in Form von ASCII-Zeichen erfolgen kann. Seine Universalität wird lediglich durch den recht hohen Stromverbrauch von ca. 1 W (5 V, 210 mA) und die nur mäßig schnelle Grafikdarstellung eingeschränkt.

Das Display stellt für die Kommunikation mit einem externen Gerät drei verschiedene Schnittstellen zur Verfügung:

RS232 (Radio Sector 232 – Standard der Electronic Industrie Alliance)
- Serielle asynchrone Übertragung.
- Vorteile:
  - Einfache Implementierung
- Nachteile:
  - Relativ langsam
  - asynchron -> unsicher

I²C (Inter-Integrated Circuit)
- Serielle synchrone Übertragung.
- Vorteile:
  - Schnell
  - Clock-gesteuert -> sicher
- Nachteile:
  - aufwendige Datenpakete zur Adressierung
  - umständlich zu programmieren

SPI (Serial Peripheral Interface)
- Serielle synchrone Übertragung.
- Vorteile:
  - sehr schnell
  - bis auf Konfiguration einfache Implementierung
  - Clock-gesteuert -> sicher
- Nachteile:
  - hoher Konfigurationsaufwand

Da sie am einfachsten in einem Programm realisiert werden kann, habe ich mich zunächst für die RS232-Schnittstelle entschieden. Da diese für meine Zwecke auf Dauer aber zu langsam ist und außerdem durch die Verwendung am Display die Kommunikation mit einem externen PC erschwert, möchte ich baldmöglichst auf die SPI-Schnittstelle übergehen können, da diese erheblich schneller ist.

3.3 Tastatur

Da bei meinem Display leider derzeit der Touchscreen wegen eines Glasbruchs (s. Abb. 6) defekt ist, kann ich dessen Funktionen nicht nutzen und muss deshalb auf eine externe PS/2-Tastatur zurückgreifen.
Um möglichst einfach Daten eingeben zu können, verwende ich deswegen für Eingaben statt der Touchfunktionen eine normale Computer-Tastatur.
Die Tastatur besitzt eine PS2-Schnittstelle, wobei die Daten seriell über eine clock-gesteuerte Datenleitung übertragen werden (synchrone Übertragung). Die Clockrate beträgt ca. 10 kHz.

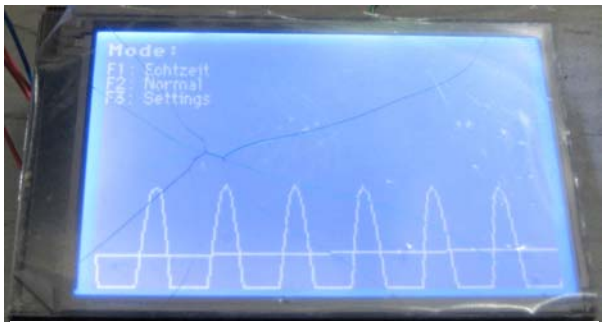


Abb. 6: Glasbruch beim aktuellen Display

Beim Betätigen einer Taste sendet die Tastatur sowohl beim Drücken als auch beim Loslassen einer Taste eine eindeutige Bytesequenz, man unterscheidet deshalb zwischen Make- und Breakcode. Der Makecode ist ein für jede Taste festgelegtes Byte, spezielle Tasten schicken diesem Byte ein zusätzliches Byte mit dem Wert 0xE0 voraus. Zur Erzeugung des Breakcodes wird zunächst das Byte mit dem Wert 0xF0 gesendet und anschließend der Makecode wiederholt. Zur Tastenerkennung müssen somit mindestens drei aufeinanderfolgende Bytes ausgewertet werden können.

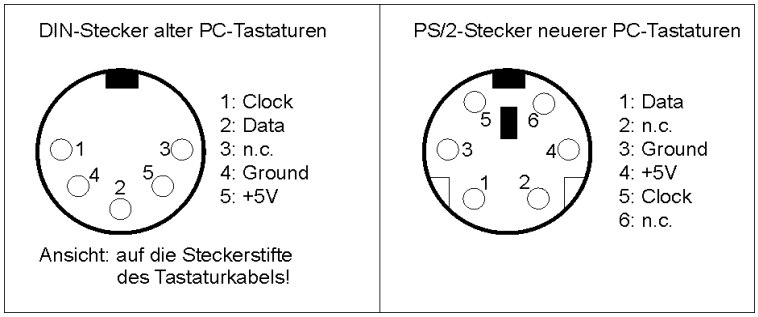


Abb. 7: Pinbelegung Stecker PC-Tastatur

Das Übertragungsprotokoll einer PS/2-Tastatur ähnelt dem einer seriellen Schnittstelle, verwendet allerdings TTL-Pegel. Abb. 7 zeigt die entsprechende Pinbelegung des Mini-DIN-Verbinders. Wie bei der RS232-Übertragung werden ein Startbit (immer Low), 8 Datenbits (LSB ... MSB), 1 Parity-Bit sowie ein Stopbit (immer High) übertragen. Die Datenbits werden jeweils mit der fallenden Flanke des Taktsignals übernommen. Insgesamt sind dies 11 Bit, die pro Byte zu verarbeiten sind. Das Übertragen der Information, dass eine Taste und welche gedrückt wurde, dauert also je nach Anzahl zu übertragender Bytes lediglich 1 bis 3 ms.
Da die USART-Schnittstelle des PIC diese Art der synchronen Datenübertragung nicht unterstützt, verwende ich eine eigene Abfrageroutine. Dazu löst ein Pegelwechsel an der Datenleitung der Tastatur (Startbit) einen Interrupt hoher Priorität aus, mit dem ein Byte von der Tastatur eingelesen wird. Werden mehrere Byte gesendet, wird dieser Interrupt mehrfach aufgerufen. Eine Ablaufsteuerung erkennt, wann ein gültiger Tastencode vorliegt und setzt ein entsprechendes Flag.

4 Software

4.1 Programmstruktur

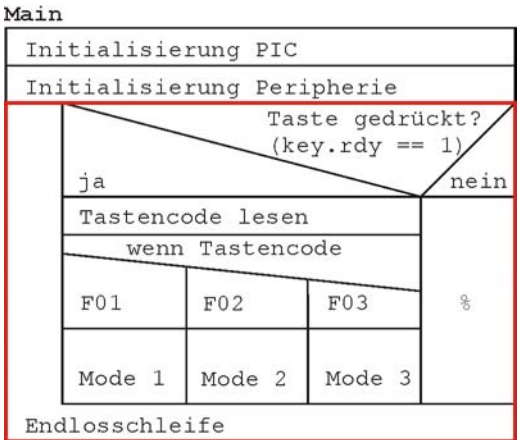


Abb. 8: Struktogramm Main (Hauptprogramm)

Abb. 8 stellt den Programmablauf direkt nach dem Einschalten dar.
Zuerst werden der PIC und die Peripherie initialisiert.
Als nächstes beginnt der rot eingerahmte Code, der eine Endlosschleife darstellt. Hier wird nur dann ein Ereignis ausgeführt, wenn eine Taste gedrückt worden ist.
Dazu wird zunächst geprüft, ob die Variable „Taste gedrückt“ (key.rdy), die vom Interrupt gesetzt wird (s.u.), auf „1“ steht, wenn ja, wurde eine Taste gedrückt, der zugehörige Tastencode wurde bereits während der Ausführung des Interrupts eingelesen. Nun wird der zugeordnete Tastencode verwendet, um den gewünschten Betriebs-Modus auszuführen.
Wenn nun also z.B. die Taste „F01“ auf der Tastatur betätigt wurde, wird der Ablauf „Mode 1“ ausgeführt.

Die Rot eingerahmte Endlosschleife in Abb.08 kann jederzeit durch einen Interrupt unterbrochen werden, für den es zwei mögliche Quellen gibt (Abb. 9). Die zugehörige Interrupt-Routine wird immer dann ausgeführt, wenn ein Interrupt-Ereignis eingetreten ist. Z.B. löst die Betätigung einer Taste an der Tastatur einen Interrupt am Port B aus, bei dessen Abarbeitung die Variable “Taste-gedrückt“ auf „1“ gesetzt wird, außerdem wird der zugehörige Tastencode gelesen.

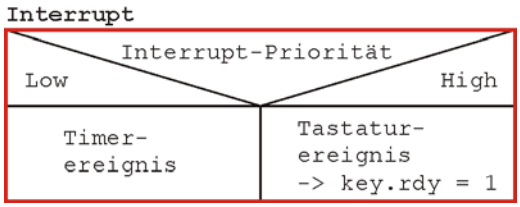


Abb. 9: Struktogramm der Interrupts

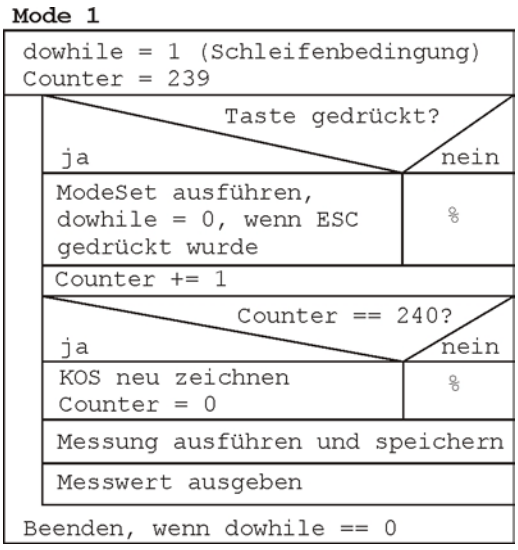


Abb. 10: Struktogramm „Mode 1“

Das Struktogramm in Abb. 10 zeigt, wie die Prozedur „Mode 1“ aufgebaut ist, die aufgerufen wird, wenn die Taste mit dem Tastencode „F01“ gedrückt wird. Nach einer Initialisierung wird eine Schleife ausgeführt. Diese enthält zwei Abfragen, die aber nicht bei jeder Messung ausgeführt werden müssen. In jedem Fall wird aber der Messwertzähler Counter erhöht, eine Messung aufgenommen, gespeichert und auf dem Display ausgegeben. Die zugehörige Umsetzung des Struktogramms in den entsprechenden C-Code wird in Abb. 11 dargestellt. Die Zahlen in Klammern verweisen dabei auf die zugehörigen Zeilennummern des Codes.

Zuerst werden die Variablen initialisiert, die für den Ablauf benötigt werden (2 - 5).
Dann wird wie auch in den anderen Modi und beim Hauptprogramm Main geprüft, ob eine Taste gedrückt wurde (7). Wenn die Taste „Escape“ (ESC) gedrückt wurde, dann wird „dowhile“ durch den Rückgabewert der „omodeset()“-Funktion auf „0“ gesetzt (7), was dazu führt, dass nach einem letztmaligen Durchlaufen der Messung die Schleife und damit diese Prozedur beendet wird, denn ESC beendet den aktuellen Modus.
Nach dieser Tastenabfrage wird die Variable „counter“ um 1 erhöht (8). Wenn „counter“ 240 ereicht hat (9), dann ist die Messung am rechten Bildschirmrand angekommen, in diesem Fall wird „counter“ wieder auf „0“ gesetzt und (10) das Display gelöscht (11). Nun wird der Messwert aufgenommen (15 - 17), formatiert (18 - 19), gespeichert (20) und als Graph ausgegeben (21 - 23). Wenn „dowhile“ zuvor auf 0 gesetzt wurde, wird dieser Modus beendet (25), sonst wird der Code in der Endlosschleife wiederholt (6 - 24).

```
1  void omode1(void){
2      char buffer[20];
3      int temp;
4      int counter=239;
5      dowhile=1;
6      while (dowhile) {
7          if (key.rdy==1) dowhile = omodeset();
8          counter++;
9          if (counter==240) {
10             counter=0;
11             putrsUSART((char*) "#DL");
12             while (BusyUSART());
13             DrawS();
14         }
15         ConvertADC();
16         while (BusyADC());
17         temp = ReadADC();
18         temp = temp >> 3;
19         temp = 128 - temp;
20         array[counter] = temp;
21         sprintf(buffer,"#GW%d%c%d",counter,",",temp);
22         putsUSART(buffer);
23         while (BusyUSART());
24     }
25 }
```

Abb. 11: C-Code Modul „Mode 1“

4.2 Displayansteuerung

Um Objekte auf dem Display auszugeben, gibt es 125 verschiedene Befehle, die in Form von ASCII-Zeichen ggf. ergänzt durch entsprechende Parameter übertragen werden. Diese Befehle sind in verschiedene Gruppen aufgeteilt, die sich in den Anfangsbuchstaben der einzelnen Befehle wiederspiegeln:

- T: Terminalbetrieb
- Z: Zeichenketten-Ausgabe
- G: Geraden und Punkte zeichnen
- R: Rechteckige Bereiche verändern/zeichnen
- U: Bitmap-Bilder Befehle
- D: Display-Befehle
- Q: Blinkbereichs-Befehle
- B: Bargraph-Befehle
- C: Clipboard-Befehle
- N: Einstellungen für Menübox
- N: Menübox-Befehle
- M: Makro-Befehle
- M: Automatische Makros
- M: Makro-Prozesse
- X,K,Y,S: Sonstige Befehle
- A: Touchpanel-Befehle

Um in der Mitte des Displays den Schriftzug „Hello World“ auszugeben, muss z.B. folgender Befehl an das Display gesendet werden:

„#ZL90,90,Hello World“

Mit „#“ beginnt ein Befehl, „Z“ ist der Befehl für eine Textausgabe. „L“ bedeutet, dass der Text linksbündig ausgerichtet sein soll. Die nächsten beiden numerischen Parameter, die durch ein Komma getrennt werden müssen, geben die Koordinaten (x, y) an, wo der Text positioniert werden soll. Dann kommt der auszugebende Text.

4.3 Displayprogrammierung

Ich habe mich zunächst entschieden, über die USART-Schnittstelle des PICs mit der RS232-Schnittstelle des Display zu kommunizieren, später werde ich versuchen, ob die Verwendung einer der anderen (schnelleren) Schnittstellen trotz der aufwendigeren Programmierung deutliche Vorteile bringt.

Damit die Datenübertragung fehlerfrei abläuft, benutzt das Display für das Datenformat ein bestimmtes Software-Protokoll, genannt „small protocoll“.
Dies funktioniert nach folgendem Schema:
Jedes Datenpaket beginnt mit dem Byte „0x11“ (<DC1>), Das nächste Byte enthält die Anzahl der nachfolgenden Datenbytes, die die Nutzdaten darstellen. Nun werden byteweise die Nutzdaten übertragen, denen als abschließendes Byte eine Prüfsumme über alle Bytes des Datenpaketes folgt. Die Prüfsumme ergibt sich also aus der Summe aller Bytes, inklusive Länge und Anfangsbyte.
Wenn das gesamte Paket fehlerfrei übertragen wurde, antwortet das Display mit dem Byte „0x06“ (<ACK>), wenn ein Fehler aufgetreten ist, mit „0x15“ (<NAK>).

Damit ich die Daten an das Display einfach übertragen kann, habe ich eine Funktion SendData() geschrieben, die eine Zeichenfolge mit den Nutzdaten selbständig in das Smartprotokoll einfügt und sowohl die erforderlichen Berechnungen ausführt als auch auf die Übertragungsbestätigung wartet:

```
char SendData(char *data)
{
      char chksum, i, n;              //Variablen-Deklarationen
      char buffer[24];                //     „
      chksum = 0;                     //     „
      buffer[0] = 0x11;               // Anfangsbyte <DC1>
      chksum += buffer[0];            // Anfangsbyte zur Prüfsumme addieren
      buffer[1] = strlen(data);       // Länge der nachfolgenden Bytes bestimmen
      chksum += buffer[1];            // zur Prüfsumme addieren
      for (i = 0; data[i] != 0; i++) { // alle Zeichen aus Data ...
            buffer[i+2] = data[i];    // in Buffer übertragen ...
            chksum += buffer[i+2];    // und zur Prüfsumme addieren
      }
      buffer[i+2] = 0;          // abschließender Null-Charakter für Zeichenketten
      buffer[i+3] = chksum;     // Prüfsumme an den Buffer anhängen

      n = i + 3;                                        // Anzahl der Zeichen im Buffer
      for (i = 0; i <= n; i++) WriteUSART(buffer[i]); // Zeichenfolge übertragen

      while (!DataRdyUSART());                          // Auf Antwort warten
      if (ReadUSART() == 0x06) return(0);               // 0x06 = <ACK>: Erfolgreich
                                                           übertragen und “0“, sonst
      return(1);                                        // Fehler = “1“ zurückgeben
}
```

Mit den beiden folgenden Anweisungen kann ich dann ganz einfach Befehle an das Display senden:

```
sprintf(buffer,“ #ZL90,90,Hello World“);      // String in buffer kopieren
SendData(buffer);                             // Buffer ans Display schicken
```

Damit man allerdings auch Werte aus numerischen Variablen als Text ausgeben kann, muss man sie zunächst mit sprintf() formatieren und in eine Puffervariable (Char-Array) schreiben:

```
char buffer[20];                  // Array buffer mit der länge 20 deklarieren
int wert;                         // Integer Wert

wert=429;                         // Wert ist 429
sprinf(buffer,“#ZL90,90,%d“,wert); // „sprintf“ speichert die aus Displaybefehl
                                     und Integervariable zusammengesetzte
                                     Zeichenkette in buffer
SendData(buffer);                 // Array Buffer an Display senden
```

4.4 Code-Optimierung

Da jede Code-Zeile Speicher im PIC und auch Zeit für die Ausführung benötigt, sollte man so optimiert wie möglich programmieren, damit man eine möglichst hohe Messrate erzielt.
So sollte man z.B. Variablen, die nie größer als 256 werden können, als „char“ und nicht als „int“ deklarieren, denn eine Integer-Variable braucht 16 Bit und eine Charakter-Variable nur 8 Bit Speicher. Außerdem erfordert der Zugriff auf eine 16-Bit-Variable doppelt so viele Taktzyklen wie der Zugriff auf eine 8-Bit-Variable.
Um Rechenschritte und somit Zeit zu sparen, kann man außerdem mehrere Schritte zu einem zusammenfassen:

```
Temp = WertLesen();
Temp = Temp - 1;          ⟶      Test = (WertLesen()-1);
Test = Temp;
```

So sieht als Beispiel eine unoptimierte Mess-Routine aus:

```
void omode1(void){
      int buffer[20];
      int temp;
      int counter;
      counter=239;
      dowhile=1;
      while(dowhile){
            if (key.rdy==1)dowhile=omodeset();
            counter++;
            if (counter==240){
                  counter=0;
                  sprintf(buffer,"#DL");
                  SendData(buffer);
                  DrawS();
            }
            ConvertADC();
            while(BusyADC());
            temp=ReadADC();
            temp=temp>>3;
            temp=128-temp;
            array[counter]=temp;
            sprintf(buffer,"#GW%d%c%d",counter,",",temp);
            SendData(buffer);
      }
}
```

Im nachfolgenden Codebeispiel sind die Optimierungen **hervorgehoben**:

```
void omode1(void){
       char buffer[20];
       char temp;
       char counter=239;
       dowhile=1;
       while(dowhile){
              if (key.rdy==1)dowhile=omodeset();
              if (++counter==240){
                     counter=0;
                     sprintf(buffer,"#DL");
                     SendData(buffer);
                     DrawS();
              }
              ConvertADC();
              while(BusyADC());
              array[counter]=128-(ReadADC()>>3)
              GWZ(counter,temp);
       }
}
```

Damit ist der Programmablauf auf dem Mikrocontroller erkennbar schneller als vorher.

5 Praktische Erprobung

5.1 Zeitliche Auflösung

Um festzustellen, welche Anteile die Zeit für die Datenübertragung an das Display und die eigentliche Zeit für die Messung benötigen, habe ich verschiedene Übertragungsraten für das Display eingestellt und die Messzeit für einen vollständigen Sweep (240 Messwerte) im Echtzeitmodus (jeder Messwert wird sofort angezeigt) bestimmt. Da bei den höheren Übertragungsraten diese Zeit sehr kurz wurde, habe ich dort die Zeit für mehrere Durchläufe gemessen. Die entsprechenden Werte sind in der folgenden Tabelle zusammengestellt:

Abb. 12: Messaufbau zur praktischen Erprobung des Gerätes.
Mitte von links: Oszilloskop, Netzteil, Platine, Display
unten von links: Tastatur für den PIC und den PC, PC-Monitor

Baudrate	Anzahl Sweeps	Gemessene Zeit	Zeit für einen Sweep
1200	1	28,9 s	28,9 s
4800	3	22,7 s	7,8 s
19200	10	22,0 s	2,2 s
38400	20	26,0 s	1,3 s
115200	20	14,5 s	0,73 s

Die Zeit für die Darstellung eines Messwertes setzt sich zusammen aus der Zeit t_W für den Wandlervorgang sowie der Zeit $t_Ü$ für die Übertragung an das Display.

Da ein Sweep aus 240 Einzelmessungen besteht, ergibt sich für die Gesamtzeit T eines Sweeps die Gleichung

$$T = 240 \cdot (t_W + t_Ü)$$

Die Zeit für den Wandlervorgang t_W ist unabhängig von der Übertragungsrate, da sie vor allem durch die Taktfrequenz des PIC festgelegt ist. Verwendet man für die Berechnung die höchste und die niedrigste Baudrate, ergeben sich zwei verschiedene Werte für $t_Ü$, wobei ich mit $t_{Ü,1200}$ die Zeit bei der niedrigsten und mit $t_{Ü,115k}$ die Zeit bei der höchsten Baudrate bezeichne.
Damit ergeben sich für die beiden ausgewählten Messwerte folgende Gleichungen:

$$28{,}9s = 240 \cdot \left(t_W + t_{Ü,1200}\right) \qquad (1)$$

$$0{,}73s = 240 \cdot \left(t_W + t_{Ü,115k}\right) \qquad (2)$$

Unter der Annahme, dass die Übertragungszeit umgekehrt proportional zur Baudrate ist, gilt:

$$t_{Ü,115k} = \frac{1}{96} t_{Ü,1200}$$

Setzt man diesen Ausdruck in Glg. (2) ein, erhält man

$$0{,}73s = 240 \cdot \left(t_W + \frac{1}{96} t_{Ü,1200} \right) \qquad (2')$$

Damit hat man nun nur noch zwei Gleichungen mit zwei Unbekannten. Nach Auflösen der Klammern ergeben sich die beiden Gleichungen

$$28{,}9s = 240 \cdot t_W + 240 \cdot t_{Ü,1200} \qquad (1')$$

$$0{,}73s = 240 \cdot t_W + \frac{240}{96} \cdot t_{Ü,1200} \qquad (2'')$$

Dieses Gleichungssystem kann man mit einem GTR (Grafik-fähiger Taschenrechner) einfach lösen. Mit dem Ansatz

$$\begin{vmatrix} 240 & 240 & 28{,}9 \\ 240 & 2{,}5 & 0{,}73 \end{vmatrix}$$

(der Taschenrechner benutzt die Form ax + by = c, deswegen die Umstellung der Spalten) erhält man

$t_W = 1{,}7$ ms und $t_{Ü,1200} = 119$ ms bzw. $t_{Ü,115k} = 1{,}2$ ms

Bei der höchsten Übertragungsrate dauert damit die Messwerterfassung bereits etwas länger als die Datenübertragung. Insgesamt wird für die Erfassung und Darstellung eines Messwertes eine Zeit von ca. 2,9 ms benötigt, das entspricht einer Samplerate von etwa 345 Messungen pro Sekunde.

Im Hochgeschwindigkeits-Modus werden zuerst alle 240 Messwerte aufgenommen und gespeichert und erst anschließend komplett ausgegeben. Die benötigte Zeit wird nun vor allem vom Messvorgang des Analog-Digital-Wandlers, der Speicherung im RAM und der Schleifensteuerung bestimmt. So sieht man zwar nur das Signal erst nach erfolgter Messung, allerdings erreicht man so eine Samplerate von 35 000 Messungen pro Sekunde und ist folglich etwa 100 Mal schneller als bei einer sofortigen Anzeige!

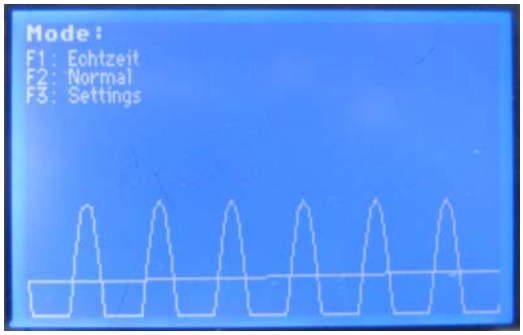


Abb. 13: Display-Anzeige mit Menü

Drücken der Taste <ESC> beendet die Messung, der letzte Graph bleibt erhalten und das Menü wird wieder eingeblendet (s. Abb. 13).

5.2 Genauigkeit

Wichtig bei einem Oszilloskop ist neben der Geschwindigkeit auch die Genauigkeit der Messung. Der Analog-Digital-Wandler des PICs, der zum Messen verwendet wird, stellt Werte mit einer Auflösung von 10 Bit zu Verfügung, welche auf Grund der Auflösung des Displays (7 Bit) allerdings nicht voll ausgenutzt werden kann. Deswegen wird der gemessene Wert durch 8 geteilt, damit ergibt sich bei maximaler Spannung der Wert 128, welcher der Höhe des Displays in Pixeln entspricht. Mit dieser Auflösung von 7 Bit werden auch die Messungen gespeichert, da sie so noch in ein „char"- Array passen (die Software des PIC erlaubt keine Variablenstrukturen, die größer sind als 256 Byte).

5.3 Bedienerfreundlichkeit

Auf Grund der großen PC-Tastatur mit 105 Tasten können sehr viele verschiedene Eingaben sowohl in den Menüs als auch in dem eigentlichen Messvorgang gemacht werden. Weil so keine Touchflächen als Button benötigt werden, kann der volle Platz des Displays ausgenutzt werden um die gemessenen Daten auszugeben. Die bei Bedarf eingeblendete Menüführung ist einfach und übersichtlich gestaltet, wichtige Werte können im Einstellungs-Menü geändert werden.
Wenn später ein Display mit intaktem Touchscreen eingebaut wird, ist außerdem eine Touchbedienung möglich, auf die externe Tastatur kann dann verzichtet werden..

6 Weiterentwicklung

6.1 Sample-Rate

Die maximale Samplerate meines Oszilloskops beträgt momentan noch 35.000 Messungen pro Sekunde. Um diesen Wert zu verbessern gibt es mehrere Ansatzpunkte:

- Taktfrequenz des PICs erhöhen:
  Ich betreibe den PIC momentan mit einem 16 Mhz-Quarz, welcher aber durch einen 40 Mhz-Quarzgenerator ersetzt werden könnte. Allerdings müsste dann ein Teil des Codes angepasst werden, der zeitabhängige Komponenten enthält.
- Schnellere Verbindung zum Display:
  Durch eine schnellere Verbindung können die Messungen im Echtzeitmodus schneller ausgegeben werden und die Samplerate würde dadurch steigen.
- Optimierung des Codes:
  Wie schon in 3.4 beschrieben kann der Code weiter optimiert werden, wodurch Zeit gewonnen werden kann und somit ebenfalls die maximale Samplerate steigt.

6.2 Eingangspegelanpassung

Zur Zeit kann das Oszilloskop durch den AD-Wandler nur in einem Bereich von 0 bis +5 V messen, wodurch sich die Einsatzmöglichkeiten einschränken. Um auch in anderen Spannungsbereichen messen zu können, könnte man den Eingangspegel durch eine einfache Operationsverstärker-Schaltung anpassen. Weitere Ergänzungen dieser Schaltung - wie z.B. die Verstärkung - könnten per Softwaresteuerung über derzeit noch ungenutzte I/O-Pins des PICs eingestellt werden.

6.3 Erweiterung des Funktionsumfangs

Da dem PIC noch sehr viel andere Funktionen zur Verfügung stehen und die Software bis zur Ausnutzung des gesamten Speicherplatzes beliebig erweiterbar ist, können immer weitere Funktionen implementiert werden:

- Verbindung zum PC

  Um gespeicherte Messungen zu übertragen oder eine Echtzeit-Datenausgabe auf den PC zu ermöglichen, könnte eine Verbindung über die USART-Schnittstelle des PICs über einen Pegelwandler für die RS232-Schnittstelle des Computer hergestellt werden.
- Erweiterte Trigger-Funktion

  Durch eine erweiterte Triggerfunktion könnte ein Messvorgang durch externe Ereignisse ausgelöst werden (z.B. durch Sensoren). Wegen der Speicherfunktionen könnten dann die Messungen sowohl vor als auch nach dem Trigger-Ereignis ausgegeben werden.
- Frequenz- oder Ereignis-Zähler

  Durch diese Funktion könnte die Frequenz eines anliegenden Signal bestimmt und ausgegeben werden. Der Ereigniszähler kann ebenfalls als Trigger-Ereignis dienen, um nach einer bestimmten Anzahl von Vorgängen eine Messung auszulösen.

7 Quellen, Literatur, Werkzeuge

- Datenblätter der Firma Microchip:
  - MPLAP_C18_Libraries.pdf (Handbuch zum MCC18 C-Compiler)
  - PIC18Fxx2 datasheet.pdf (Datenblatt zum PIC18F452)
- Software der Firma Microchip:
  - MPLAP IDE v8.36 (Entwicklungsumgebung für Microchip PICs)
  - MCC18 v3.34 (C-Compiler für Microchip PICs)
- Taschenrechner:
  - Casio fx-9860G (an der Schule eingeführter Taschenrechner)
- Schaltpläne erstellt mit
  - Eagle 4.16 r2 Light (Frei verfügbare Version)

8 Danksagung

Ich bedanke mich bei meinem Betreuungslehrer Thomas Biedermann, für seinen großen Zeiteinsatz, seine Geduld und seine „Fehler-Lösungs-Strategien“, die mir sehr geholfen haben. Außerdem bedanke ich mich natürlich bei Susanne Biedermann für die hervorragende Verpflegung.
DANKE!