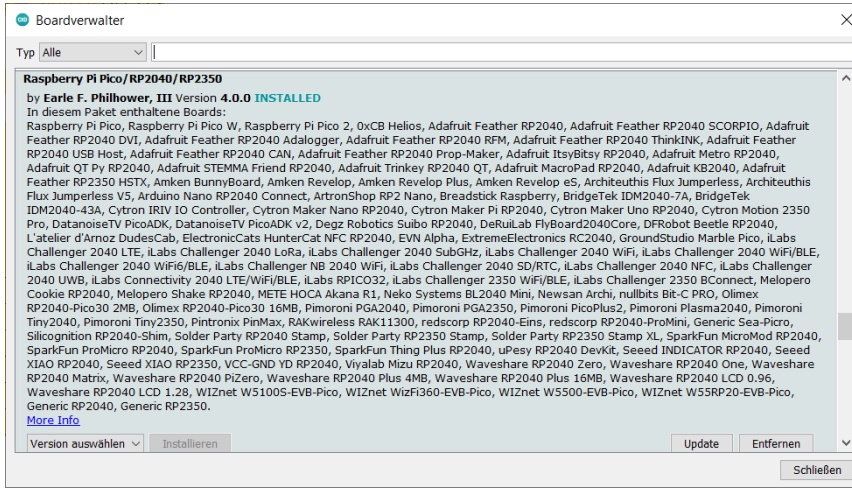
RPi Pico Test-Labor
und PicoBasic

9 Die PicoBasic-Firmware

Die gesamte Firmware des Projekts wurde mit der Arduino-Software und der Boarderweiterung von Earle F. Philhower entwickelt. Wer den frei verfügbaren Quelltext bearbeiten möchte, muss die Pico-Erweiterung über den Boardverwalter installieren.



Es beginnt mit Deklaration der globalen Variablen. Die in PicoBasic verwendeten Variablen erkennt man sofort; a, b, c und d. Das 256 Elemente lange Array code[] ist mit dem End-Befehl 0x4900 vorbelegt und wird beim Laden eines Programms anders gefüllt. Der 256 Byte lange Datenspeicher heißt ram[].

```
//PicoBasic V2.2 B.Kainka  https://www.elektronik-labor.de
#include <EEPROM.h>
#include "hardware/pwm.h"
```

```
uint16_t code[256] = {0x4900};
uint32_t mask = 0x000000FF;
uint8_t ram[256];
uint8_t a = 0, b = 0, c = 0, d = 0;
uint8_t pw1=0;
uint8_t pw2=0;
uint16_t ad;
uint16_t rx = 1000;
char ch;
uint16_t adr = 0;
uint16_t return_adr[10];
uint16_t return_nr = 0;
uint16_t kom = 0;
uint16_t dat = 0;
uint32_t warten, weiter;

void setup() {
  Serial.begin(115200);
  EEPROM.begin(512);
}

void loop(){}

void setup1() {
  gpio_init_mask (mask);
  gpio_put_masked (mask, 0);
  gpio_set_function(8,GPIO_FUNC_PWM);
  gpio_set_function(9,GPIO_FUNC_PWM);
  pwm_config cfg = pwm_get_default_config();
  pwm_config_set_clkdiv_int (&cfg, 250);
  pwm_init(4, &cfg, true);
  pwm_set_wrap(4, 511);
  pwm_set_gpio_level(8,0);
  pwm_set_gpio_level(9,0);
  pwm_set_enabled(4,true);
}
```

Die Funktionen void setup() und void loop() laufen im ersten Kern des Pico, wo auch die serielle USB-Schnittstelle und die Zeiterfassung

ablaufen. Der eigentliche Basic-Interpreter verwendet den zweiten Kern und läuft in setup1() mit der Einrichtung der Ports und der PWM-Ausgabe sowie in loop1() mit der eigentlichen Abarbeitung aller Befehle.

Für jeden Basic-Befehl gibt es eine eigene Funktion. In den meisten Fällen ist leicht erkennbar, wie die Bearbeitung umgesetzt wird. Wo im PicoBasic-Programm A = 100 steht, wird a= dat ausgeführt, wobei dat gerade 100 enthält, und es wird die Adresse um 1 erhöht. Insgesamt gibt es die Funktionen k1 bis k73.

Eine Besonderheit stellen die Befehle Delay ms, Delay s und Delay min dar. Sie werden nicht direkt ausgeführt, sondern füllen die Variable warten mit der gewünschten Anzahl der Millisekunden. Die eigentliche Warteschleife ist so in die Interpreterschleife eingebaut, dass sie einen möglichen Datenempfang während der Wartezeit nicht behindert.

```
void k1(){a = dat;adr++;}
void k2(){b = dat;adr++;}
void k3(){c = dat;adr++;}
void k4(){d = dat;adr++;}
void np(){adr++;}
void k8(){gpio_put_masked (mask, dat);adr++;} //Pout dat
void k9(){gpio_set_dir_masked (mask, dat);adr++;} //Pdir dat
void k10(){for(int j=0; j<8; j++){gpio_set_pulls(j,1 &(dat>>j)
      , 0);}adr++;} //Pullup dat
void k11(){for(int j=0; j<8; j++){gpio_set_pulls(j, 0,1
      (dat>>j));}adr++;} //Ppulldown dat
 void k16(){pwm_set_gpio_level(8,dat*2); pw1=dat;adr++;} //PWM1
void k17(){pwm_set_gpio_level(9,dat*2); pw2=dat;adr++;} //PWM2
void k18(){a &= dat;adr++;} // A = A AND dat
void k19(){a |= dat;adr++;} // A = A Or dat
void k24(){delayMicroseconds(dat);adr++;} // Delay µs
void k25(){warten = dat;adr++;}          // Delay ms
void k26(){warten = dat*1000;adr++;}     // Delay s
void k27(){warten = dat*60000;adr++;}    // Delay min
void k32(){adr++;adr = dat;} // Goto L
void k33(){adr++;return_nr++;return_adr[return_nr]=adr;  adr = dat;}
      // Gosub L
```

```
void k34(){adr++;if (a == b){adr = dat;};} // If A = B Goto L
void k35(){adr++;if (a > b){adr = dat;};} // If A > B Goto L
void k36(){adr++;if (a < b){adr = dat;};} // If A < B Goto L
void k37(){adr++;if (c > 0){adr = dat;}; c--;} // C*Goto L
void k38(){adr++;if (d > 0){adr = dat;}; d--;} // D*Goto L
void k40(){a += 1;adr++;} // A = A + 1
void k41(){a -= 1;adr++;} // A = A - 1
void k42(){a += b;adr++;} // A = A + B
void k43(){a -= b;adr++;} // A = A - B
void k44(){a = a*b;adr++;} // A = A * B
void k45(){a /= b;adr++;} // A = A / B
void k46(){a &= b;adr++;} // A = A AND B
void k47(){a |= b;adr++;} // A = A OR B
void k48(){a = a^b;adr++;} // A = A XOR B
void k49(){a = a << 1;adr++;} // A = Shl 1
void k50(){a = a >> 1;adr++;} // A = Shr 1
void k51(){a = ~a;adr++;} // A = NOT A
void k52(){b = a;adr++;} // B = A
void k53(){a = b;adr++;} // A = B
void k54(){c = a;adr++;} // C = A
void k55(){a = c;adr++;} // A = C
void k56(){d = a;adr++;} // D = A
void k57(){a = d;adr++;} // A = D
void k58(){a = ram[b];b++;adr++;} // A = [B+]
void k59(){ram[b] = a;b++;adr++;} // [B+] = A
void k60(){a = analogRead(A0) >>2;adr++;} // A = AD0
void k61(){a = analogRead(A1) >>2;adr++;} // A = AD1
void k62(){a = analogRead(A2) >>2;adr++;} // A = AD2
void k63(){a = 255 & gpio_get_all();adr++;} // A = Pin
void k64(){a = 1 & (gpio_get_all());adr++;} // A = Pin0
void k65(){if(rx < 1000){a = rx; rx = 1000;}adr++;} // Input A
void k66(){Serial.println(a);adr++;} // Print A
void k67(){pwm_set_gpio_level(8,a*2); pw1=a;adr++;} // PWM1 = A
void k68(){pwm_set_gpio_level(9,a*2); pw2=a;adr++;} // PWM2 = A
void k69(){gpio_put_masked (mask, a);adr++;} // Pout = A
void k72(){adr++;adr = return_adr[return_nr]; return_nr--;} // Ret
void k73(){adr++;adr--;} // End

void d65(){Serial.println(a);}  //A
```

```
void d66(){Serial.println(b);}  //B
void d67(){Serial.println(c);}  //C
void d68(){Serial.println(d);}  //D
void d69(){Serial.println(analogRead(A0)>>2);}  //E  AD0
void d70(){Serial.println(analogRead(A1)>>2);}  //F  AD1
void d71(){Serial.println(analogRead(A2)>>2);}  //G  AD2
void d72(){Serial.println(255 & gpio_get_all());} //H  Pin
void d73(){weiter = 0xFFFFFFFF;} //I  Stop!
void d74(){weiter = 0;} //J  Go!
void d75(){uint16_t n = Serial.parseInt();gpio_set_dir_masked
      (mask, n);} //K  DIR
void d76(){uint16_t n = Serial.parseInt();gpio_put_masked
      (mask, n);} //L  OUT
void d77(){uint16_t n = Serial.parseInt();for(int j=0; j<8; j++)
      {gpio_set_pulls(j,1 &(n>>j), 0);}} //M  Pullup
void d78(){uint16_t n = Serial.parseInt();for(int j=0; j<8; j++)
      {gpio_set_pulls(j, 0,1 &(n>>j));}} //N  Pulldown
void d79(){uint16_t n = Serial.parseInt();pwm_set_gpio_level(8,n*2);
      pw1=n;} //O  PWM1
void d80(){uint16_t n = Serial.parseInt();pwm_set_gpio_level(9,n*2);
      pw2=n;} //P  PWM2
void d81(){for(int j=0; j<256; j++){uint16_t n = Serial.parseInt();
      ram[j]=n;}} //Q  RAM füllen
void d82(){uint16_t n = Serial.parseInt();a = n;} //R  A=
void d83(){uint16_t n = Serial.parseInt();b = n;} //S  B=
void d84(){uint16_t n = Serial.parseInt();c = n;} //T  C=
void d85(){uint16_t n = Serial.parseInt();d = n;} //U  D=
void d86(){                                 //V  PWM-Vorteiler
  uint16_t n = Serial.parseInt();
  pwm_set_enabled(4,true);
  pwm_config cfg = pwm_get_default_config();
  pwm_config_set_clkdiv_int (&cfg, n);
  pwm_init(4, &cfg, true);
  pwm_set_wrap(4, 511);
  pwm_set_gpio_level(8,pw1*2);
  pwm_set_gpio_level(9,pw2*2);
  pwm_set_enabled(4,true);
}
```

Zusätzlich existieren die Funktionen d65 bis d86 für die direkten Zugriffe aus dem TestLab, die man einzeln mit A bis V aufrufen kann. Der einzige Direktbefehl, der über die Möglichkeiten von PicoBasic hinausgeht, ist d86() zur Einstellung des PWM-Vorteilers. In der Arduino-Software laufen die PWM-Ausgaben beim Pico immer mit genau 1 kHz. Damit auch höhere Frequenzen möglich werden, musste eine eigne Initialisierung der PWM-Ausgänge mit einer dazu passenden PWM-Ausgabe verwendet werde. So werden PWM-Frequenzen bis 250 kHz möglich. Um genaue Ausgangsfrequenzen zu bekommen, muss der Prozessortakt auf 128 MHz eingestellt werden. Die PWM-Kanäle laufen tatsächlich mit einer Auflösung von 9 Bit, sodass die PWM-Timer den Takt durch 512 teilen. Bei der Übergabe eines Bytes müssen daher die Bits um eine Stelle nach links geschoben werden. Dabei entsteht die höchste PWM-Frequenz von 250 kHz.

Alle diese Funktionen werden in Funktionen-Arrays zusammengefasst, um sie zeitsparend aufrufen zu können.

```
void loop1(){
  void (*befehl[76])() = {
  np, k1, k2, k3, k4, np, np, np,
  k8, k9, k10, k11, np, np, np, np,
  k16, k17, k18, k19, np, np, np, np,
  k24, k25, k26, k27, np, np, np, np,
  k32, k33, k34, k35, k36, k37, k38, np,
  k40, k41, k42, k43, k44, k45, k46, k47,
  k48, k49, k50, k51, k52, k53, k54, k55,
  k56, k57, k58, k59, k60, k61, k62, k63,
  k64, k65, k66, k67, k68, k69, np, np,
  k72, k73, np};

  void (*direkt[22])() = {
  d65, d66, d67, d68, d69, d70, d71, d72,
  d73, d74, d75, d76, d77, d78, d79, d80,
  d81, d82, d83, d84, d85, d86};

  for (adr = 0; adr < 255; adr++) {
    uint16_t eedat = (EEPROM.read(2*adr))<<8;
    eedat = eedat + EEPROM.read(2*adr+1);
```

```
  if (eedat>0x4900){eedat=0x4900;}
  code[adr] = eedat;
  weiter = 0; //keine Wartezeit
  warten = 0;
 }
```

Es folgt die Übertragung des PicoBasic-Programms aus dem EEPROM, sodass ein eventuell gespeichertes Programm sofort ausgeführt werden kann. Hier wird berücksichtigt, dass es kein Befehls-Token über 0x4900 geben darf, da sonst eine Adresse außerhalb des Arrays aufgerufen würde. Vor der ersten Verwendung des EEPROMs könnte sonst 0xFFFF gelesen werden und einen Absturz verursachen.

Die eigentliche Interpreterschleife im zweiten Kern ist sehr kurz. Der Code wird in Kommando und Daten zerlegt. Damit kann dann das entsprechende Kommando aus dem Funktions-Array aufgerufen werden. Auch die Wartezeiten werden hier ausgeführt. Grundlage ist die Funktion millis() zur Abfrage der verstrichenen Millisekunden seit dem letzten Start des Pico. Erst wenn die letzte Millisekunde der Wartezeit abgelaufen ist, wird der nächste Basic-Befehl ausgeführt.

```
 while(1){
…
  if(warten>0){weiter = millis()+warten; warten = 0;}
  if(millis()>=weiter){
    kom = code[adr] >> 8;
    dat = code[adr] & 255;
    befehl[kom]();
  }
 }
```

Egal, ob gerade ein Programm läuft oder nicht, die Firmware muss jederzeit auf die Schnittstelle lauschen, um ein neues Programm zu laden, Zahlenwerte zu empfangen oder Direktkommandos auszuführen.

Nur wenn mindestens ein Byte empfangen wurde, wird das laufende Programm kurz unterbrochen, um darauf zu reagieren. Das erste Zeichen wird empfangen und analysiert. Dabei kommen folgende Fälle vor:

Wenn es eine Ziffer 0 bis 9 enthält, soll offensichtlich eine Zahl empfangen werden. Der Empfang wird dann fortgeführt, bis die Eingabe durch ein Zeilenende beendet wird.

Wenn das erste Zeichen ein kleines p ist, soll ein neues Programm übertragen werden. Es folgt die Anzahl der zu übertragenden Bytes. Entsprechend viele Daten werden empfangen und in das code-Array geschrieben. Danach wird die Adresse auf null gesetzt, sodass das neue Programm von vorn beginnt.

Mit einem kleinen e wird ebenfalls ein neues Programm übertragen, diesmal allerdings ins code-Array und ins EEPROM. Nach Beendigung der Übertragung wird das neue Programm gestartet. Zusätzlich steht es im EEPROM bereit für den nächsten Neustart

```
if (Serial.available() > 0){
  ch = Serial.read();
  delay(2);
  if (ch>47 && ch<58){
     rx = ch - 48;
    while (ch!=13){
      if (Serial.available() > 0){
        ch = Serial.read();
        if (ch>47 && ch<58){
          rx = rx * 10;
          rx = rx + ch - 48;
          rx = rx & 255;
        }
      }
    }
  Serial.println (rx);
  }
  if (ch==112){   //p
    uint16_t n = Serial.parseInt();
    for (uint16_t i=0;i<n;i++){
      uint16_t pdat = Serial.parseInt();
      code[i] = pdat;
    }
    Serial.flush();
```

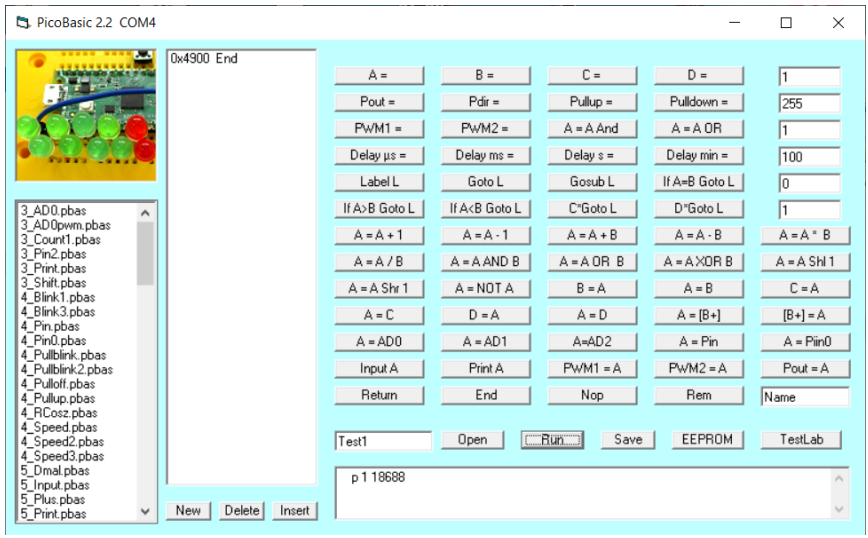
```
  adr = 0;
  a=0; b=0; c=0; d=0;
  return_nr = 0;
  rx = 1000;
  weiter = 0; //keine Wartezeit
  warten = 0;
 }
 if (ch==101){   //e
  uint16_t n = Serial.parseInt();
  for (uint16_t i=0;i<n;i++){
   uint16_t pdat = Serial.parseInt();
   code[i] = pdat;
   EEPROM.write(2*i, pdat>>8);
   EEPROM.write(2*i+1, pdat&255);
   if (i==31){EEPROM.commit();}
   if (i==63){EEPROM.commit();}
   if (i==95){EEPROM.commit();}
   if (i==127){EEPROM.commit();}
   if (i==159){EEPROM.commit();}
   if (i==191){EEPROM.commit();}
   if (i==223){EEPROM.commit();}
  }
  Serial.flush();
  EEPROM.commit();
  adr = 0;
  a=0; b=0; c=0; d=0;
  return_nr = 0;
  rx = 1000;
  weiter = 0; //keine Wartezeit
  warten = 0;
 }
 if (ch>64 && ch<88){
  direkt[ch-65]();
 }
}
```

Ganz am Ende der Möglichkeiten nach dem Empfang eines Bytes steht die Behandlung der Direktkommandos. Wenn es sich um ein ASCII-

Zeichen ab 65 (= A) oder bis 88 (= V) handelt, wird das zugehörige Direkt-Kommando über das Funktionen-Array aufgerufen.

10 Das Anwenderprogramm

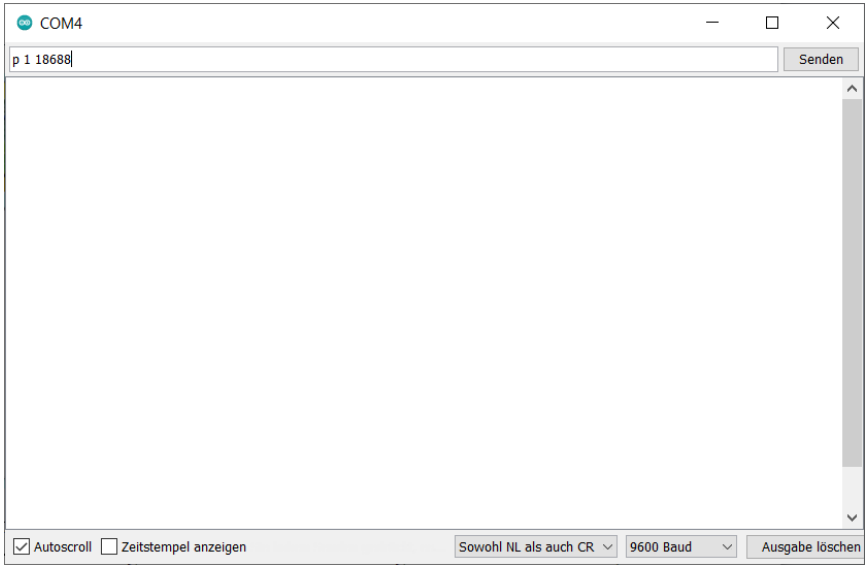
Die gesamte Kommunikation mit der PicoBasic-Firmware und dem darin integrierten TestLab läuft über die USB-serielle Schnittstelle. Deshalb kann man einzelne Kommandos mit einem beliebigen Terminal testen.



Zum Test soll ein PicoBasic-Programm mit nur einer Befehlszeile mit dem End-Befehl hochgeladen werden. Man darf eine beliebige Baudrate außer 1200 verwenden, muss aber auch DTR einschalten. Was dazu gesendet werden muss, sieht man in der Anwendersoftware: p 1 18688

Nun starte ich das Programm Count1.pbas, damit sich etwas an den LEDs bewegt. Wenn es mir gelingt, das End-Programm zu übertragen, muss die Anzeige einfrieren.

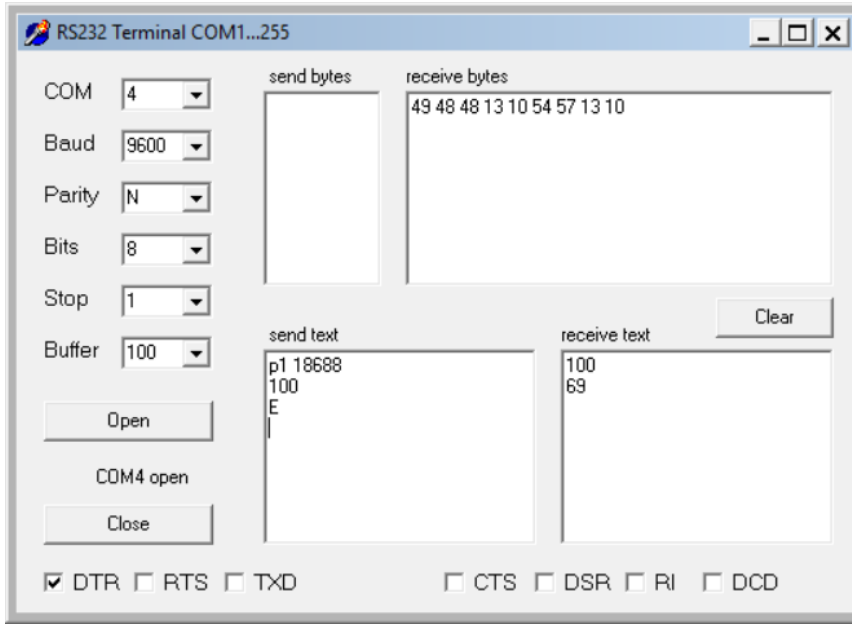
Zuerst soll hierfür das Terminal in der Arduino-IDE verwendet werden. Ich starte den seriellen Monitor und gebe den Text zum Laden des Programms ein: p 1 18688 Und tatsächlich friert das Programm ein. Die zuletzt aktiven LEDs bleiben an.



Ein Versuch mit Terminal.exe ist ebenfalls erfolgreich, allerdings nicht ganz ohne Schwierigkeiten. Während im seriellen Monitor die gesamte Nachricht erst nach einem Return abgeschickt wird, sendet Terminal.exe jedes Zeichen sofort, wenn es eingetippt wird. Bei der seriellen Übertragung gibt es allerdings im Pico ein Timeout nach einer zu langen Wartezeit. Man muss die Zeichen also sehr zügig eintippen.

Tatsächlich ist es mit ausreichend schneller Eingabe nach einigen Versuchen auch hier gelungen, das kleine Programm zu übertragen und damit ein laufendes Programm zu stoppen. Falls hier ein Fehler passiert, kann es zu einem Absturz kommen, weil die Firmware vergeblich auf weitere Eingaben wartet. Dann hilft ein Druck auf den Reset-Taster.

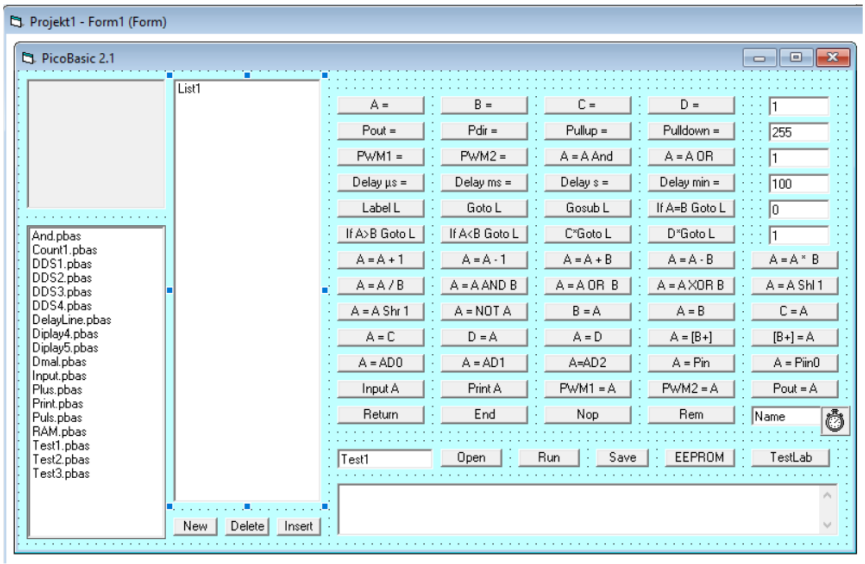
Danach wurden gleich weitere Dinge getestet. Die Eingabe einer Zahl 100 wurde wie erwartet mit dem Echo 100 beantwortet. Und das TestLab-Kommando E führte zur Messung am analogen Eingang A0 und der Rücksendung des Ergebnisses 69. In der Byte-Anzeige ist zu erkennen, dass nach jeder Zahl CR und LF gesendet wird, also eine neue Zeile kommen soll.



Diese und ähnliche Tests waren während der Entwicklung der Anwendersoftware immer wieder nötig, um die einzelnen Schritte zu überprüfen und mögliche Fehler aufzudecken.

Die Software wurde komplett mit dem betagten Visual Basic 6 entwickelt. Es gäbe viele Gründe, eine neuere Programmiersprache zu verwenden, aber für mich war es einfacher, VB6 zu verwenden, das ich besser kenne. Der gesamte Quelltext ist zu lang, um ihn komplett abzudrucken. Stattdessen werden hier nur Ausschnitte exemplarisch erläutert. Der komplette Quelltext liegt aber offen und kann frei verwendet werden. Dahinter steht auch meine stille Hoffnung, dass jemand ein Programm damit baut, das nicht nur unter Windows läuft. Sehr wahrscheinlich ist das allerdings nicht, denn dieser müsste ein erfahrener Programmierer sein und trotzdem einen Nutzen in dem sehr einfachen PicoBasic sehen.

Also los! Die entscheidende Aufgabe der Software ist es, aus den Klicks des Anwenders einen Quelltext und den Zwischencode zu bilden, der die Basic-Tokens und die mitgelieferten Datenbytes enthält. Die neu gebildeten Zeilen werden dann in die ListBox eingetragen.



Ein Klick auf die Schaltfläche A = ruft die Sub Command1_Click auf. Weil dieser Befehl einen Parameter braucht, wird zuerst die Zahl in Text1 gelesen, das ist das Eingabefenster in derselben Zeile im Form1. Das in der Firmware festgelegte Kommando für diesen Befehl lautet 1. Entsprechend wird das Kommando k = 1 * 256 + n mit der 1 im Highbyte und dem gelesenen Parameterbyte im Lowbyte gebildet. In der Textzeile für die ListBox erscheint diese 16-Bit-Zahl als Hexzahl und zusätzlich der Befehl mit Parameter im Klartext: 0x0101 A = 1

```
Private Sub Command1_Click()
   n = Zahl(Text1.Text)
   k = 1 * 256 + n
   Textzeile = "0x0" + hex(k) + "  A = " + Text1.Text
        + Chr$(10) + Chr$(13)
   ZeileEinsetzen
End Sub
```

Ganz ähnlich geht das bei allen PicoBasic-Befehlen. Bei allen Befehlen ohne Parameter steht im Lowbyte einfach nur 00. Als Beispiel soll hier der Befehl A = A + 1 gezeigt werden. Hier wird Command27_Click aufgerufen und das Basic-Token heißt 40. Die Nummerierung der Subs

folgt keinem bestimmten Muster, weil während der Entwicklung immer neue Befehle hinzukamen.

```
Private Sub Command27_Click()
   k = 40 * 256
   Textzeile = "0x" + hex(k) + "  A = A + 1 "
   ZeileEinsetzen
End Sub
```

Die Sub ZeileEinsetzen tut genau dieses und erhöht außerdem die Adresse um eins. Eine Besonderheit bilden die Zeilen zur Festlegung eines Labels bzw. Sprungziels und die zugehörigen Sprungbefehle. Ein Klick auf Label L ruft die Sub Command18_Click auf. Hier wird die bisherige Label-Nummer um eins erhöht und diesem neuen Label die aktuelle Adresse zugeordnet. Weil ZeileEinsetzen die Adresse immer um eins erhöht, wird sie danach wieder um eins verkleinert, denn das Setzen eines Labels ist kein eigentlicher Basic-Befehl. Entsprechend enthält die Zeile auch kein Basic-Token.

```
Private Sub Command18_Click()
   n = Val(Text5.Text)
   LabelNr = n + 1
   LabelNrTxt = Str(LabelNr)
   LabelNrTxt = Right$(LabelNrTxt, Len(LabelNrTxt) - 1)
   Text5.Text = LabelNrTxt
   Text6.Text = LabelNrTxt
   Textzeile = "           " + "  L" + Text5.Text + ":"
   Label(LabelNr) = Adresse
   ZeileEinsetzen
   Adresse = Adresse - 1
End Sub
```

Ein Sprung wie z.B. Goto L1: wird mit Command17_Click behandelt. Das eigentliche Kommando heißt 32, und der Byte-Parameter besteht aus der Adresse, die vorher dem Label zugeordnet wurde. Das alles sieht sehr einfach und klar aus. Aber was passiert, wenn jemand eine Zeile löscht oder eine neue Zeile einfügt? Dem Label wurde ja bereits eine Adresse zugeordnet, aber jetzt könnte sich alle Adressen verschoben haben.

```
Private Sub Command17_Click()
   n = Zahl(Text5.Text)
   k = 32 * 256 + Label(n)
   Textzeile = "0x" + hex(k) + "  Goto L" + Text5.Text + ":"
   ZeileEinsetzen
End Sub
```

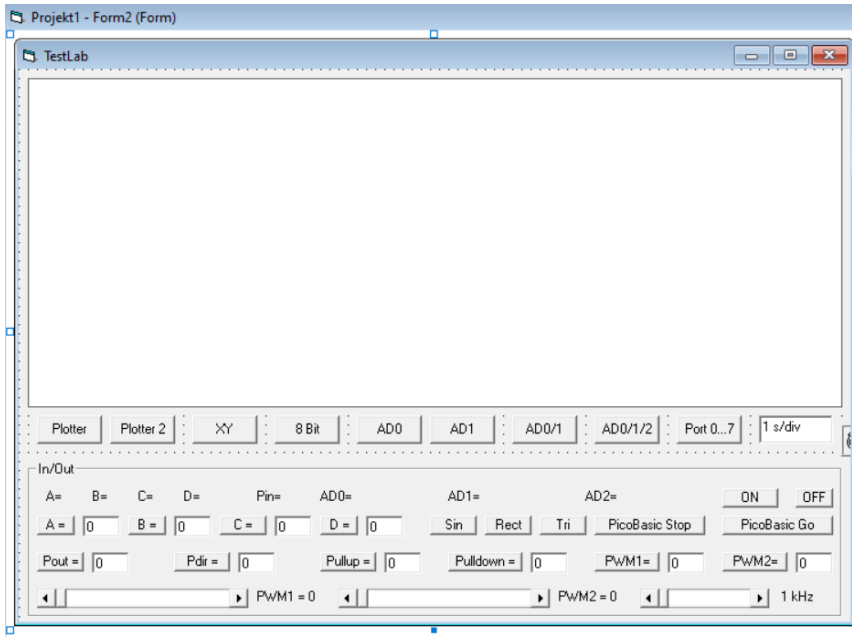
In der bei jedem neuen Befehl aufgerufenen Sub ZeileEinsetzen wird auch die Adresse erhöht. Danach wird eine Sub LabelKorrektur aufgerufen, in der der aktuelle Quelltext von vorn bis zur letzten Zeile durchgegangen wird, um die Label-Adressen und Sprungziele zu aktualisieren. Damit wird das Problem eingefügter oder gelöschter Zeilen gelöst.

```
Private Sub ZeileEinsetzen()
   X = List1.ListIndex
   L = List1.ListCount
   'Text8 = Text8 + Chr(x) + Chr(l)
   If X > -1 Then
     Zeile = X
     For i = L To X + 1 Step -1
       List1.List(i) = List1.List(i - 1)
     Next i
     List1.RemoveItem X
   End If
   List1.List(Zeile) = Textzeile
   Zeile = List1.ListCount
   Adresse = Adresse + 1
   LabelKorrektur
End Sub
```

Die Sub LabelKorrektur soll hier nicht abgedruckt werden, weil sie so chaotisch zusammengehackt wurde, dass sie nicht vorzeigbar ist und nur Verwirrung stiften würde. Aber wer sich den gesammte Quelltext genau ansieht wird erkennen, dass sie auch beim Laden eines Programmquell-textes verwendet wird. Es ist nämlich nicht auszuschließen, dass ein Quelltext einmal mit einem externen Editor bearbeitet wurde.

Das TestLab

Die Oberfläche zum TestLab befindet sich im Form2 des Projekts, das geöffnet wird, wenn man in der PicoBasic-Oberfläche auf TestLab klickt.



Bei einem Klick auf A 0 wird die Sub Command23 aufgerufen. Sie sendet das Direktkommando R und den zugehörigen Byte-Parameter. Ganz ähnlich funktionieren alle Kommandos, die mit einer Zahleneingabe verbunden sind.

```
Private Sub Command23_Click()
   n = Zahl(Text3.Text)
    SENDSTRING "R" + Str(Zahl(Text3))
    SENDBYTE 13
End Sub
```

Vergleichbar sind auch die Kommandos zur Einstellung von PWM1 und PWM2 über einen Schieberegler. Bei jeder Änderung der Position zu

PWM1 wird die Sub HScoll1_Change aufgerufen. Der Parameter wird gelesen und mit einem vorangestellten O gesendet.

```
Private Sub HScroll1_Change()
   d = HScroll1.Value
   Label9.Caption = "PWM1 = " + Str(d)
   SENDSTRING "O" + Str(d)
   SENDBYTE 13
End Sub
```

Ein Klick auf Sin erzeugt in der Sub Comman12_Click eine Sinustabelle, die mit einem Q in das Byte-Array der Firmware übertragen wird. Ganz ähnlich funktionieren die Signalfunktionen Tri und Rec für Dreieck- und Rechtecksignale.

```
Private Sub Command12_Click()
Timer1.Enabled = False
SENDSTRING "Q"  'RAM füllen
For n = 0 To 256
   d = Int(127 + 127 * Sin(n * 2 * 3.1415 / 256))
   SENDSTRING (Str(d))
   SENDBYTE 13
   DELAY 1
 Next n
End Sub
```

Der Plotter im TestLab wird in der Sub Command9_Click gebildet, indem 250 empfangene Bytes in das Diagramm in der PictureBox geplottet werden. Plotter2 arbeitet genauso, mit dem Unterschied, dass jedes zweite Byte dem zweiten Kanal zugeordnet wird. Und in XY wird jedes zweite Byte als y-Koordinate interpretiert.

```
Private Sub Command9_Click()
   Timer1.Enabled = False
   Raster
   CLEARBUFFER
   Do
      DoEvents
   Loop Until INBUFFER() > 0
   txt$ = GetNr
```

```
   If txt$ <> "-1" Then
      y_alt = Val(txt$)
      For n = 0 To 249
         Do
            DoEvents
         Loop Until INBUFFER() > 0
         Y = Val(GetNr())
         Picture1.Line (2 * n, 256 - y_alt)-(2 * n + 2, 256 - Y)
         y_alt = Y
      Next n
      CLEARBUFFER
   End If
End Sub
```

Die Oszilloskop-Funktionen funktionieren ähnlich, mit dem Unterschied, dass hier jedes Byte einzeln abgefragt wird. Außerdem liegt diesmal in X-Richtung die einstellbare Zeitachse. Ein Klick auf AD0 ruft die Sub Command10_Click auf. Hier wird jedes Messergebnis von AD0 als Byte mit dem Kommando E abgerufen und geplottet.

Der zeitliche Ablauf wird durch die Abfrage von TIMEREAD, einer Funktion aus der RSCOM.DLL bestimmt, die die Zeit seit dem Programmstart in Millisekunden liefert. Das Zeitinterwall hängt von der Eingabe in Text2 ab. Wenn hier z.B. 1 s/div steht, entspricht das 1000 ms für 50 Messungen, die ein Skalenteil füllen. Das entspricht 20 ms pro Messung. Mit 0.2 s/div kommt man auf 4 ms und bleibt noch im genauen Millisekundenraster. Nur 2 ms bei der Einstellung 0.1 s/div sind weniger als die Zeit zur Übertragung zweier Bytes und die Messzeit das AD-Wandlers. Die Messung läuft dann einfach mit der maximal möglichen Geschwindigkeit. Ganz ähnlich funktionieren auch die anderen Oszilloskope für AD1, die Zweikanalmessung und die Dreikanalmessung.

```
Private Sub Command10_Click()
   Timer1.Enabled = False
   Raster
   CLEARBUFFER
```

```
    t = Val(Text2.Text) * 20
    tt = TIMEREAD() + t
    SENDSTRING "E"
    y_alt = Val(GetNr())
    For n = 0 To 499
       Do
          DoEvents
       Loop Until TIMEREAD() >= tt
       tt = tt + t
       SENDSTRING "E"
       Y = Val(GetNr())
       Picture1.Line (n, 256 - y_alt)-(n + 1, 256 - Y)
       y_alt = Y
    Next n
End Sub
```

Die oberste Zeile im In/Out-Rahmen liefert Ergebnisse der direkten Abfragen der Variabel A bis D, der Portzustände und der drei AD-Kanäle. All das wird über den Timer1 gesteuert. Den Timer kann man mit On oder Off ein- und ausschalten. Das ist nötig, wenn die Abfragen unerwünscht sind, weil sie den zeitlichen Ablauf eines Programms oder einer Oszilloskop-Messung stören. Beim Start eines Plotters oder eines Oszilloskops wird der Timer automatisch gestoppt.

```
Private Sub Timer1_Timer()
    CLEARBUFFER
    SENDSTRING "A": Label1.Caption = "A=" + GetNr()
    SENDSTRING "B": Label2.Caption = "B=" + GetNr()
    SENDSTRING "C": Label3.Caption = "C=" + GetNr()
    SENDSTRING "D": Label4.Caption = "D=" + GetNr()
    CLEARBUFFER
    SENDSTRING "E": d$ = GetNr(): n = Val(d$)
    Label5.Caption = "AD0=" + d$ + ", " + Str(Int(n * 3300 / 255)) + "
mV"
    SENDSTRING "F": d$ = GetNr(): n = Val(d$)
    Label6.Caption = "AD1=" + d$ + ", " + Str(Int(n * 3300 / 255)) + "
mV"
    SENDSTRING "G": d$ = GetNr(): n = Val(d$)
```

```
   Label7.Caption = "AD2=" + d$ + ", " + Str(Int(n * 3300 / 255))
         + " mV"
   SENDSTRING "H": Label8.Caption = "Din=" + GetNr()
End Sub
```

Und schließlich gibt es noch die Schaltflächen PicoBasic Stop und PicoBasic Go, die nichts anderes bewirken als die Kommandos I und J abzusenden.

```
Private Sub Command1_Click()
   SENDSTRING "I"
End Sub

Private Sub Command2_Click()
   SENDSTRING "J"
End Sub
```